Java for Android

Budi Kurniawan

Java for Android

First Edition: August 2014

ISBN: 978-0-9921330-3-0

Printed in the United States of America
Book and Cover Designer: Brainy Software Team

Technical Reviewer: Paul Deck
Indexer: Chris Mayle

Trademarks

Warning and Disclaimer

Table of Contents

Introduction

Welcome to *Java for Android*.

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

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

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

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

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

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

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

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

Java, the Language and the Technology

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

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

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

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

That was Java version 1.0.

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

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

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

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

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

In 2010 Oracle acquired Sun.

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

What Makes Java Platform Independent?

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

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

Figure I.1: Traditional programming paradigm

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

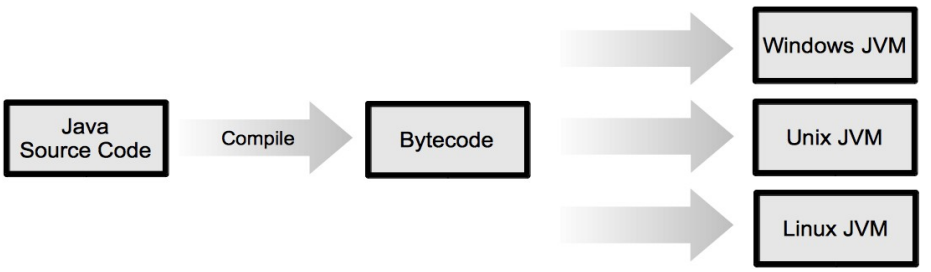

Figure I.2: Java programming model

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

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

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

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

The Java software is available in two distributions:

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

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

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

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

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

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

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

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

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

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

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

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

- JBoss
- Jonas
- Apache Geronimo

The complete list can be found here.

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

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

The Java Community Process (JCP) Program

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

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

An Overview of Object-Oriented Programming

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

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

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

The Benefits of OOP

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

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

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

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

Is OOP Hard?

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

First the bad news.

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

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

Now the good news.

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

About This Book

The following presents the overview of each chapter.

Part I: Java

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

Part II: Android

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

Chapter 43, “Handling the Handler” talks about the **Handler** class, which can be used, among others, to schedule a **Runnable** at a future time.

Chapter 44, “Asynchronous Tasks” explains how to handle asynchronous tasks in Android.

Appendixes

Appendix A, “javac”, Appendix B, “java”, and Appendix C, “jar” explain the **javac**, **java**, and **jar** tools, respectively.

Appendix D, “NetBeans” and Appendix E, “Eclipse” provide brief tutorials on NetBeans and Eclipse, respectively.

Downloading and Installing Java

Before you can start compiling and running Java programs, you need to download and install the JDK as well as configure some system environment variables. To develop Android applications, you need JDK 6 or later. You should download and install the latest version of Java available.

Downloading the JDK

You can download the JRE and the JDK for Windows, Linux, Mac, and Solaris from Oracle’s website:

`http://www.oracle.com/technetwork/java/javase/downloads/index.html`

If you click the Download link on the page, you’ll be redirected to a page that lets you select an installation for your platform: Windows, Linux, Solaris, or Mac OS. The 64 bit versions for those platforms are available. Also, note that the same link also provides the JRE. However, for development you need the JDK not only the JRE, which is only good for running compiled Java classes. The JDK includes the JRE.

JDKs for Mac

Pre-7 JDKs for Mac are available from Apple’s website at http://support.apple.com/downloads. Apple used to port and maintain Mac JDKs but no longer does so after its last update of JDK 6. You can download JDK 7 and JDK 8 for Mac from the same Oracle website that provides JDKs for other operating systems.

After downloading the JDK, you need to install it. Installation varies from one operating system to another. These subsections detail the installation process.

Installation on Windows

Installation on Windows is easy. Simply double-click the icon of the executable file you downloaded and follow the instructions. Figure I.3 shows the first dialog of the installation wizard.

Figure I.3: Installing JDK 8 on Windows

Installation on Linux

On Linux platforms, the JDK is available in two installation formats.

- RPM, for Linux platforms that supports the RPM package management system, such as Red Hat and SuSE.
- Self-extracting package. A compressed file containing packages to be installed.

If you are using the RPM, follow these steps:

1. Become root by using the **su** command
2. Extract the downloaded file.
3. Change directory to where the downloaded file is located and chmod:

   ```
   chmod a+x rpmFile
   ```

   where *rpmFile* is the RPM file.
4. Run the RPM file:

   ```
   ./rpmFile
   ```

If you are using the self-extracting binary installation, follow these steps.

1. Extract the downloaded file.
2. Use **chmod** to give the file the execute permissions:

   ```
   chmod a+x binFile
   ```

   Here, *binFile* is the downloaded bin file for your platform.
3. Change directory to the location where you would like the files to be installed.
4. Run the self-extracting binary. Execute the downloaded file with the path prepended to it. For example, if the file is in the current directory, prepend it with "./" (necessary if "." is not in the PATH environment variable):

   ```
   ./binFile
   ```

Setting System Environment Variables

After you install the JDK, you can start compiling and running Java programs. However, you can only invoke the compiler and the JRE from the location of the **javac** and **java** programs or by including the installation path in your command. To make compiling and running programs easier, it is important that you set the **PATH** environment variable on your computer so that you can invoke **javac** and **java** from any directory.

Setting the Path Environment Variable on Windows 8

To set the **PATH** environment variable on Windows 8, follow these steps:

1. Hover your mouse to the right bottom corner of the screen.
2. Click on the Search icon and type **Control Panel**.
3. Click on Control Panel and then click on **System**.
4. Click on **Advanced system settings**. In the System Properties window that appears, click on the **Advanced** tab.
5. Click on the **Environment Variables** button.
6. Locate the **Path** environment variable in the **User Variables** or **System Variables** panes. The value of **Path** is a series of directories separated by semicolons. Now, add the full path to the **bin** directory of your Java installation directory to the end of the existing value of **Path**. The directory looks something like:

   ```
   C:\Program Files\Java\jdk1.7.0_<version>\bin
   ```

7. Click **Set**, **OK**, or **Apply**.

Setting the Path Environment Variable on Windows 7

To set the **PATH** environment variable on Windows 7, follow these steps:

1. Select **Computer** from the Start menu.
2. Click on the **Control Panel** link. If no such a link exists, type **Control Panel** in the Search box and press Enter.
3. Click on the **Systems** icon.
4. Click on **Advanced system settings**. In the System Properties window that appears, click on the **Advanced** tab.
5. Click on the **Environment Variables** button.
6. Locate the **Path** environment variable in the **User Variables** or **System Variables** panes. The value of **Path** is a series of directories separated by semicolons. Now, add the full path to the **bin** directory of your Java installation directory to the end of the existing value of **Path**. The directory looks something like:

   ```
   C:\Program Files\Java\jdk1.7.0_<version>\bin
   ```

7. Click **Set**, **OK**, or **Apply**.

Setting the Path Environment Variable on UNIX and Linux

Set the path environment variable on these operating systems depends on the shell you use.

For the C shell, add the following to the end of your **~/.cshrc** file:

```
set path=(path/to/jdk/bin $path)
```

where *path/to/jdk/bin* is the bin directory under your JDK installation directory.

For the Bourne Again shell, add this line to the end of your **~/.bashrc** or **~/.bash_profile** file:

```
export PATH=/path/to/jdk/bin:$PATH
```

Here, *path/to/jdk/bin* is the **bin** directory under your JDK installation directory.

Testing the Installation

To confirm that you have installed the JDK correctly, type **javac** on the command line from any directory on your machine. If you see instructions on how to correctly run **javac**, then you have successfully installed it. On the other hand, if you can only run **javac** from the **bin** directory of the JDK installation directory, your **PATH** environment variable was not configured properly.

Downloading Java API Documentation

When programming Java, you will invariably use classes from the core libraries. Even seasoned programmers look up the documentation for those libraries when they are coding. Therefore, you should download the documentation from here.

```
http://www.oracle.com/technetwork/java/javase/downloads/index.html
```

(You need to scroll down until you see "Java SE 8 Documentation.")

The API is also available online here:

```
http://docs.oracle.com/javase/8/docs/api
```

Downloading Program Examples

The program examples accompanying this book can be downloaded from this URL:

```
http://books.brainysoftware.com/download
```

Extract the zip file to a working directory and you are good to go.

Chapter 1
Your First Taste of Java

Developing a Java program involves writing code, compiling it into bytecode, and running the bytecode. This is a process you will repeat again and again during your career as a Java programmer, and it is crucial that you feel comfortable with it. The main objective of this chapter therefore is to give you the opportunity to experience the process of software development in Java.

As it is important to write code that not only works but that is also easy to read and maintain, this chapter introduces you to Java code conventions. And, since smart developers use an integrated development environments (IDEs) to make their lives easier, the last section of this chapter provides some advice on Java IDEs.

Your First Java Program

This section highlights steps in Java development: writing the program, compiling it into bytecode, and running the bytecode.

Writing a Java Program

You can use any text editor to write a Java program. Open a text editor and write the code in Listing 1.1. Alternatively, if you have downloaded the program examples accompanying this book, you can simply copy it to your text editor.

Listing 1.1: A simple Java program

```
class MyFirstJava {
    public static void main(String[] args) {
        System.out.println("Java rocks.");
    }
}
```

For now, suffice it to say that Java code must reside in a class. Also, make sure you save the code in Listing 1.1 as a **MyFirstJava.java** file. All Java source files must have **java** extension.

Compiling Your Java Program

You use the **javac** program in the **bin** directory of your JDK installation directory to compile Java programs. Assuming you have edited the **PATH** environment variable in your computer (if not, see the section "Downloading and Installing Java" in Introduction), you should be able to invoke **javac** from any directory. To compile the **MyFirstJava** class in Listing 1.1, do the following:

1. Open a command prompt and change directory to the directory where the **MyFirstProgram.java** file was saved in.

2. Type the following command.

```
javac MyFirstJava.java
```

If everything goes well, **javac** will create a file named **MyFirstJava.class** in your working directory.

Note

The **javac** tool has more features that you can use by passing options. For example, you can tell it where you want the generated class file to be created. Appendix A, "javac" discusses **javac** in clear detail.

Running Your Java Program

To run your Java program, use the **java** program that is part of the JDK. Again, having added the **PATH** environment variable, you should be able to invoke **java** from any directory. From your working directory, type the following and press Enter.

```
java MyFirstJava
```

Note that you do not include the **class** extension when running a Java program.

You will see the following on your console.

```
Java rocks.
```

Congratulations. You have successfully written your first Java program. Since the sole aim of this chapter is to familiarize yourself with the writing and compiling process, I will not be attempting to explain how the program works.

You can also pass arguments to a Java program. For example, if you have a class named **Calculator** and you want to pass two arguments to it, you can do it like this:

```
java Calculator arg-1 arg-2
```

Here, *arg-1* is the first argument and *arg-2* the second. You can pass as many arguments as you want. The **java** program will then make these arguments available to your Java program as an array of strings. You'll learn to handle arguments in Chapter 5, "Core Classes."

Note

The **java** tool is an advanced program that you can configure by passing options. For instance, you can set the amount of memory allocated to it. Appendix B, "java" explains these options.

Note

The **java** tool can also be used to run a Java class that is packaged in a jar file. Check the section "Setting an Application's Entry Point" in Appendix C, "jar."

Java Code Conventions

It is important to write correct Java programs that run. However, it is also crucial to write programs that are easy to read and maintain. It is believed that eighty percent of the lifetime cost of a piece of software is spent on maintenance. Also, the turnover of programmers is high, thus it is very likely that someone other than you will maintain your code during its lifetime. Whoever inherits your code will appreciate clear and easy-to-read program sources.

Using consistent code conventions is one way to make your code easier to read. (Other ways include proper code organization and sufficient commenting.) Code conventions include filenames, file organization, indentation, comments, declaration, statements, white space, and naming conventions.

A class declaration starts with the keyword **class** followed by a class name and the opening brace {. You can place the opening brace on the same line as the class name, as shown in Listing 1.1, or you can write it on the next line, as demonstrated in Listing 1.2.

Listing 1.2: MyFirstJava written using a different code convention

```
class MyFirstJava
{
    public static void main(String[] args)
    {
        System.out.println("Java rocks.");
    }
}
```

The code in Listing 1.2 is as good as the one in Listing 1.1. It is just that the class has been written using a different convention. You should adopt a consistent style for all your program elements. It is up to you to define your own code conventions, however Sun Microsystems has published a document that outlines standards that its employees should follow. The document can be viewed here. (Of course, the document is now part of Oracle.com)

`http://www.oracle.com/technetwork/java/codeconventions-150003.pdf`

Program samples in this book will follow the recommended conventions outlined in this document. I'd also like to encourage you to develop the habit of following these conventions since the first day of your programming career, so that writing clear code comes naturally at a later stage.

Your first lesson on styles is about indentation. The unit of indentation must be four spaces. If tabs are used in place of spaces, they must be set every eight spaces (not four).

Integrated Development Environments (IDEs)

It is true that you can write Java programs using a text editor. However, an IDE will help. Not only will it check the syntax of your code, an IDE can also auto complete code, debug, and trace your programs. In addition, compilation can happen automatically as you type, and running a Java program is simply a matter of clicking a button. As a result, you will develop in much shorter time.

There are dozens of Java IDEs out there, and, fortunately, the best of them are free. Here is a brief list.

- NetBeans (free and open source)
- Eclipse (free and open source)
- Sun's Java Studio Enterprise (free)
- Sun's Java Studio Creator (free)
- Oracle JDeveloper (free)
- Borland JBuilder
- IBM's WebSphere Studio Application Developer
- BEA WebLogic Workshop

- IntelliJ IDEA

The two most popular are NetBeans and Eclipse and the past few years have seen a war between the two to become the number one. NetBeans and Eclipse are both open source with strong backers. Sun Microsystems launched NetBeans in 2000 after buying the Czech company Netbeans Ceska Republika and is still heavily investing in it. Eclipse was originated by IBM to compete with NetBeans.

Which one is better depends on who you ask, but their popularity has become the impetus that propelled other software makers to give away their IDEs too. Even Microsoft, whose .NET technology is Java's most fierce competitor, followed suit by no longer charging for the Express Edition of its Visual Studio.NET.

This book provides a brief tutorial of NetBeans and Eclipse in Appendix D and Appendix E, respectively. Do consider using an IDE because it helps a lot.

Summary

This chapter helped you write your first Java program. You used a text editor to write the program, used **javac** to compile it to a class file, and ran the class file with the **java** tool.

As programs grow in complexity and projects get larger, an IDE will help expedite application development. Two tutorials on open source IDEs are given in Appendix D and Appendix E.

Questions

1. If you had saved the code in Listing 1.1 using a different name, such as **whatever.java**, would it have compiled?
2. If you had used a file extension other than java when saving the code in Listing 1.1, for example as **MyFirstJava.txt**, would it have compiled?

Chapter 2
Language Fundamentals

Java is an object-oriented programming (OOP) language, therefore an understanding of OOP is of utmost importance. Chapter 4, “Objects and Classes” is the first lesson of OOP in this book. However, before you explore many features and techniques in OOP, make sure you study the prerequisite: basic programming concepts discussed in this chapter. The topics covered are as follows.

- Encoding Sets. Java supports the Unicode character encoding set and program element names are not restricted to ASCII (American Standard Code for Information Interchange) characters. Text can be written using characters in practically any human language in use today.
- Primitives. Java is an OOP language and Java programs deal with objects most of the time. However, there are also non-object elements that represent numbers and simple values such as true and false. These simple non-object programming elements are called primitives.
- Variables. Variables are place holders whose contents can change. There are many types of variables.
- Constants. Place holders whose values cannot be changed.
- Literals. Literals are representations of data values that are understood by the Java compiler.
- Primitive conversion. Changing the type of a primitive to another type.
- Operators. Operators are notations indicating that certain operations are to be performed.

Note
If you have programmed with C or C++, two popular languages at the time Java was invented, you should feel at home learning Java because Java syntax is very similar to that of C and C++. However, the creator of Java added a number of features not available in C and C++ and excluded a few aspects of them.

ASCII and Unicode

Traditionally, computers in English speaking countries only used the ASCII (American Standard Code for Information Interchange) character set to represent alphanumeric characters. Each character in the ASCII is represented by 7 bits. There are therefore 128 characters in this character set. These include the lower case and upper case Latin letters, numbers, and punctuation marks.

The ASCII character set was later extended to include another 128 characters, such as the German characters ä, ö, ü, and the British currency symbol £. This character set is called extended ASCII and each character is represented by 8 bits.

ASCII and the extended ASCII are only two of the many character sets available. Another popular one is the character set standardized by the ISO

(International Standards Organization), ISO-8859-1, also known as Latin-1. Each character in ISO-8859-1 is represented by 8 bits as well. This character set contains all the characters required for writing text in many of the western European languages, such as German, Danish, Dutch, French, Italian, Spanish, Portuguese, and, of course, English. An eight-bit-per-character character set is convenient because a byte is also 8 bits long. As such, storing and transmitting text written in an 8-bit character set is most efficient.

However, not every language uses Latin letters. Chinese, Korean, and Thai are examples of languages that use different character sets. For example, each character in the Chinese language represents a word, not a letter. There are thousands of these characters and 8 bit is not enough to represent all the characters in the character set. The Japanese use a different character set for their language too. In total, there are hundreds of different character sets for all languages in the world. This is confusing, because a code that represents a particular character in a character set represents a different character in another character set.

Unicode is a character set developed by a non-profit organization called the Unicode Consortium (www.unicode.org). This body attempts to include all characters in all languages in the world into one single character set. A unique number in Unicode represents exactly one character. Currently at version 6, Unicode is used in Java, XML, ECMAScript, LDAP, etc. It has also been adopted by industry leaders such as IBM, Microsoft, Oracle, Google, HP, Apple, and others.

Initially, a Unicode character was represented by 16 bits, which were enough to represent more than 65,000 different characters. 65,000 characters are sufficient for encoding most of the characters in major languages in the world. However, the Unicode consortium planned to allow for encoding for as many as a million more characters. With this amount, you then need more than 16 bits to represent each character. In fact, a 32 bit system is considered a convenient way of storing Unicode characters.

Now, you see a problem already. While Unicode provides enough space for all the characters used in all languages, storing and transmitting Unicode text is not as efficient as storing and transmitting ASCII or Latin-1 characters. In the Internet world, this is a huge problem. Imagine having to transfer 4 times as much data as ASCII text!

Fortunately, character encoding can make it more efficient to store and transmit Unicode text. You can think of character encoding as analogous to data compression. And, there are many types of character encodings available today. The Unicode Consortium endorses three of them:

- UTF-8. This is popular for HTML and for protocols whereby Unicode characters are transformed into a variable length encoding of bytes. It has the advantages that the Unicode characters corresponding to the familiar ASCII set have the same byte values as ASCII, and that Unicode characters transformed into UTF-8 can be used with much existing software. Most browsers support the UTF-8 character encoding.
- UTF-16. In this character encoding, all the more commonly used characters fit into a single 16-bit code unit, and other less often used characters are accessible via pairs of 16-bit code units.
- UTF-32. This character encoding uses 32 bits for every single character. This is clearly not a choice for Internet applications. At least, not at present.

ASCII characters still play a dominant role in software programming. Java too uses ASCII for almost all input elements, except comments, identifiers, and the contents

of characters and strings. For the latter, Java supports Unicode characters. This means, you can write comments, identifiers, and strings in languages other than English. For example, if you are a Chinese speaker living in Beijing, you can use Chinese characters for variable names. As a comparison, here is a piece of Java code that declares an identifier named **password**, which consists of ASCII characters:

```
String password = "secret";
```

By contrast, the following identifier is in simplified Chinese characters.

```
String 密码 = "secret";
```

Separators

There are several characters Java uses as separators. These special characters are presented in Table 2.1.

Symbol	Name	Description
( )	Parentheses	Used in: 1. method signatures to contain lists of arguments. 2. expressions to raise operator precedence. 3. narrowing conversions. 4. loops to contain expressions to be evaluated
{ }	Braces	Used in: 1. declaration of types. 2. blocks of statements 3. array initialization.
[ ]	Brackets	Used in: 1. array declaration. 2. array value dereferencing
< >	Angle brackets	Used to pass parameter to parameterized types.
;	Semicolon	Used to terminate statements and in the **for** statement to separate the initialization code, the expression, and the update code.
:	Colon	Used in the **for** statement that iterates over an array or a collection.
,	Comma	Used to separate arguments in method declarations.
.	Period	Used to separate package names from subpackages and type names, and to separate a field or method from a reference variable.

Table 2.1: Java separators

It is very important that you familiarize yourself with the symbols and the names. Don't worry if you don't understand the terms in the Description column after reading it.

Primitives

As mentioned in Introduction, when writing an object-oriented (OO) application, you create an object model that resembles the real world. For example, a payroll application would have **Employee** objects, **Tax** objects, **Company** objects, etc. In Java, however, objects are not the only data type. There is another data type called

primitive. There are eight primitive types in Java, each with a specific format and size. Table 2.2 lists Java primitives.

Primitive	Description	Range
byte	Byte-length integer (8 bits)	-128 (-2^7) to 127 (2^7-1)
short	Short integer (16 bits)	-32,768 (-2^{15}) to 32,767 (-2^{15}-1)
int	Integer (32 bits)	-2,147,483,648 (-2^{31}) to 2,147,483,647 (-2^{31}-1)
long	Long integer (64 bits)	-9,223,372,036,854,775,808 (-2^{63}) to 9,223,372,036,854,775,807 (2^{63}-1)
float	Single-precision floating point (32-bit IEEE Standard 754)	Smallest positive nonzero: $14e^{-45}$ Largest positive nonzero: 3.4028234^{e38}
double	Double-precision floating point (64-bit IEEE Standard 754)	Smallest positive nonzero: $4.9e^{-324}$ Largest positive nonzero: $1.7976931348623157e^{308}$
char	A Unicode character	[See Unicode 6 specification]
boolean	A boolean value	true or false

Table 2.2: Java primitives

The first six primitives (**byte**, **short**, **int**, **long**, **float**, **double**) represent numbers. Each of these has a different size. For example, a **byte** can contain any whole number between -128 and 127. To understand how the smallest and largest numbers for an integer were obtained, look at its size in bits. A byte is 8 bits long so there are 2^8 or 256 possible values. The first 128 values are reserved for -128 to -1, then 0 takes one place, leaving 127 positive values. Therefore, the range for a byte is -128 to 127.

If you need a placeholder to contain number 1000000, you need an **int**. A **long** is even larger, and you might ask, if a **long** can contain a larger set of numbers than a **byte** and an **int**, why not always use a **long**? It is because a **long** takes 64 bits and therefore consume more memory space than **byte**s and **int**s. Thus, to save space, you want to use the primitive with the smallest possible data size.

The primitives **byte**, **short**, **int**, and **long** can only hold integers or whole numbers, for numbers with decimal points you need either a **float** or a **double**.

A **char** type can contain a single Unicode character, such as ‘a’, ‘9’, and ‘&’. The use of Unicode allows **char**s to also contain characters that do not exist in the English alphabet, such as this Japanese character ‘の’. A **boolean** can contain one of two possible states (**false** or **true**).

Note

The reason why in Java not everything is an object is speed. Objects are more expensive to create and operate on than primitives. In programming an operation is said to be ‘expensive’ if it is resource intensive or consumes a lot of CPU cycles to complete.

Now that you know that there are two types of data in Java (primitives and objects), let’s continue by studying how to use primitives. (Objects are discussed in Chapter 4, “Objects and Classes.”) Let’s start with variables.

Variables

Variables are data placeholders. Java is a strongly typed language, therefore every

variable must have a declared type. There are two data types in Java:

- reference types. A variable of reference type provides a reference to an object.
- primitive types. A variable of primitive type holds a primitive.

How Java Stores Integer Values

You must have heard that computers work with binary numbers, which are numbers that consists of only zeros and ones. This section provides an overview that may come in useful when you learn mathematical operators.

A byte takes 8 bits, meaning there are eight bits allocated to store a byte. The leftmost bit is the sign bit. 0 indicates a positive number, and 1 denotes a negative number. 0000 0000 is the binary representation of 0, 0000 0001 of 1, 0000 0010 of 2, 0000 0011 of 3, and 0111 1111 of 127, which is the largest positive number that a byte can contain.

Now, how do you get the binary representation of a negative number? It's easy. Get the binary representation of its positive equivalent first, and reverse all the bits and add 1. For example, to get the binary representation of -3 you start with 3, which is 0000 0011. Reversing the bits results in

```
1111 1100
```

Adding 1 gives you

```
1111 1101
```

which is -3 in binary.

For **int**s, the rule is the same, i.e. the leftmost bit is the sign bit. The only difference is that an **int** takes 32 bits. To calculate the binary form of -1 in an **int**, we start from 1, which is

```
0000 0000 0000 0000 0000 0000 0000 0001
```

Reversing all the bits results in:

```
1111 1111 1111 1111 1111 1111 1111 1110
```

Adding 1 gives us the number we want (-1).

```
1111 1111 1111 1111 1111 1111 1111 1111
```

In addition to the data type, a Java variable also has a name or an identifier. There are a few ground rules in choosing identifiers.

1. An identifier is an unlimited-length sequence of Java letters and Java digits. An identifier must begin with a Java letter.
2. An identifier must not be a Java keyword (given in Table 2.3), a **boolean** literal, or the **null** literal.
3. It must be unique within its scope. Scopes are discussed in Chapter 4, "Objects and Classes."

Java Letters and Java Digits

Java letters include uppercase and lowercase ASCII Latin letters A to Z (\u0041-\u005a—note that \u denotes a Unicode character) and a to z (\u0061-\u007a), and, for historical reasons, the ASCII underscore (_ or \u005f) and the dollar sign ($, or \u0024). The $ character should be used only in mechanically generated source code or, rarely, to access preexisting names on legacy systems.

Java digits include the ASCII digits 0-9 (\u0030-\u0039).

abstract	continue	for	new	switch
assert	default	if	package	synchronized
boolean	do	goto	private	this
break	double	implements	protected	throw
byte	else	import	public	throws
case	enum	instanceof	return	transient
catch	extends	int	short	try
char	final	interface	static	void
class	finally	long	strictfp	volatile
const	float	native	super	while

Table 2.3: Java keywords

Here are some legal identifiers:

```
salary
x2
_x3
row_count
密码
```

Here are some invalid variables:

```
2x
java+variable
```

2x is invalid because it starts with a number. **java+variable** is invalid because it contains a plus sign.

Also note that names are case-sensitive. **x2** and **X2** are two different identifiers.

You declare a variable by writing the type first, followed by the name plus a semicolon. Here are some examples of variable declarations.

```
byte x;
int rowCount;
char c;
```

In the examples above you declare three variables:

- The variable **x** of type **byte**
- The variable **rowCount** of type **int**
- The variable **c** of type **char**

x, **rowCount**, and **c** are variable names or identifiers.

It is also possible to declare multiple variables having the same type on the same line, separating two variables with a comma. For instance:

```
int a, b;
```

which is the same as

```
int a;
int b;
```

However, writing multiple declarations on the same line is not recommended as it reduces readability.

Finally, it is possible to assign a value to a variable at the same time the variable is declared:

```
byte x = 12;
int rowCount = 1000;
```

```
char c = 'x';
```

Sun's Naming Convention for Variables

Variable names should be short yet meaningful. They should be in mixed case with a lowercase first letter. Subsequent words start with capital letters. Variable names should not start with underscore _ or dollar sign $ characters. For example, here are some examples of variable names that are in compliance with Sun's code conventions: **userName**, **count**, **firstTimeLogin**.

Constants

In Java constants are variables whose values, once assigned, cannot be changed. You declare a constant by using the keyword **final**. By convention, constant names are all in upper case with words separated by underscores.

Here are examples of constants or final variables.

```
final int ROW_COUNT = 50;
final boolean ALLOW_USER_ACCESS = true;
```

Literals

From time to time you will need to assign values to variables in your program, such as number 2 to an **int** or the character 'c' to a **char**. For this, you need to write the value representation in a format that the Java compiler understands. This source code representation of a value is called *literal*. There are three types of literals: literals of primitive types, string literals, and the **null** literal. Only literals of primitive types are discussed in this chapter. The **null** literal is discussed in Chapter 4, "Objects and Classes" and string literals in Chapter 5, "Core Classes."

Literals of primitive types have four subtypes: integer literals, floating-point literals, character literals, and boolean literals. Each of these subtypes is explained below.

Integer Literals

Integer literals may be written in decimal (base 10, something we are used to), hexadecimal (base 16), or octal (base 8). For example, one hundred can be expressed as **100**. The following are integer literals in decimal:

```
2
123456
```

As another example, the following code assigns 10 to variable **x** of type **int**.

```
int x = 10;
```

Hexadecimal integers are written by using the prefixes **0x** or **0X**. For example, the hexadecimal number 9E is written as 0X9E or 0x9E. Octal integers are written by prefixing the numbers with 0. For instance, the following is an octal number 567:

```
0567
```

Integer literals are used to assign values to variables of types **byte**, **short**, **int**, and **long**. Note, however, you must not assign a value that exceeds the capacity of a

variable. For instance, the highest number for a **byte** is 127. Therefore, the following code generates a compile error because 200 is too big for a **byte**.

```
byte b = 200;
```

To assign a value to a **long**, suffix the number with the letter **L** or **l**. L is preferable because it is easily distinguishable from digit 1. A **long** can contain values between -9223372036854775808L and 9223372036854775807L (2^{63}).

Beginners of Java often ask why we need to use the suffix l or L, because even without it, such as in the following, the program still compiles.

```
long a = 123;
```

This is only partly true. An integer literal without a suffix L or l is regarded as an **int**. Therefore, the following will generate a compile error because 9876543210 is larger than the capacity for an **int**:

```
long a = 9876543210;
```

To rectify the problem, add an L or l at the end of the number like this:

```
long a = 9876543210L;
```

Longs, ints, shorts, and bytes can also be expressed in binaries by prefixing the binaries with **0B** or **0b**. For instance:

```
byte twelve = 0B1100; // = 12
```

If an integer literal is too long, readability suffers. For this reason, starting from Java 7 you can use underscores to separate digits in integer literals. For example, these two have the same meaning but the second one is obviously easier to read.

```
int million = 1000000;

int million = 1_000_000;
```

It does not matter where you put the underscores. You can use one every three digits, like the example above, or any number of digits. Here are some more examples:

```
short next = 12_345;

int twelve = 0B_1100;

long multiplier = 12_34_56_78_90_00L;
```

Floating-Point Literals

Numbers such as 0.4, 1.23, $0.5e^{10}$ are floating point numbers. A floating point number has the following parts:

- a whole number part
- a decimal point
- a fractional part
- an optional exponent

Take 1.23 as an example. For this floating point, the whole number part is 1, the fractional part is 23, and there is no optional exponent. In $0.5e^{10}$, 0 is the whole number part, 5 the fractional part, and 10 is the exponent.

In Java, there are two types of floating points:

- **float**. 32 bits in size. The largest positive float is 3.40282347e+38 and the smallest positive finite nonzero float is 1.40239846e-45.
- **double**. 64 bits in size. The largest positive double is 1.79769313486231570e+308 and the smallest positive finite nonzero double

is 4.94065645841246544e-324.

In both **float**s and **double**s, a whole number part of 0 is optional. In other words, 0.5 can be written as .5. Also, the exponent can be represented by either e or E.

To express float literals, you use one of the following formats.

```
Digits . [Digits] [ExponentPart] f_or_F
. Digits [ExponentPart] f_or_F
Digits ExponentPart f_or_F
Digits [ExponentPart] f_or_F
```

Note that the part in brackets is optional.

The *f_or_F* part makes a floating point literal a **float**. The absence of this part makes a float literal a **double**. To explicitly express a double literal, you can suffix it with D or d.

To write double literals, use one of these formats.

```
Digits . [Digits] [ExponentPart] [d_or_D]
. Digits [ExponentPart] [d_or_D]
Digits ExponentPart [d_or_D]
Digits [ExponentPart] [d_or_D]
```

In both floats and doubles, *ExponentPart* is defined as follows.

```
ExponentIndicator SignedInteger
```

where *ExponentIndicator* is either **e** or **E** and *SignedInteger* is .

$Sign_{opt}$ *Digits*

and *Sign* is either **+** or **-** and a plus sign is optional.

Examples of **float** literals include the following:

```
2e1f
8.f
.5f
0f
3.14f
9.0001e+12f
```

Here are examples of **double** literals:

```
2e1
8.
.5
0.0D
3.14
9e-9d
7e123D
```

Boolean Literals

The **boolean** type has two values, represented by literals **true** and **false**. For example, the following code declares a **boolean** variable **includeSign** and assigns it the value of **true**.

```
boolean includeSign = true;
```

Character Literals

A character literal is a Unicode character or an escape sequence enclosed in single

quotes. An escape sequence is the representation of a Unicode character that cannot be entered using the keyboard or that has a special function in Java. For example, the carriage return and linefeed characters are used to terminate a line and do not have visual representation. To express a linefeed character, you need to escape it, i.e. write its character representation. Also, single quote characters need to be escaped because single quotes are used to enclosed characters.

Here are some examples of character literals:

```
'a'
'z'
'A'
'Z'
'0'
'ü'
'%'
'我'
```

Here are character literals that are escape sequences:

```
'\b'   the backspace character
'\t'   the tab character
'\\'   the backslash
'\''   single quote
'\"'   double quote
'\n'   linefeed
'\r'   carriage return
```

In addition, Java allows you to escape a Unicode character so that you can express a Unicode character using a sequence of ASCII characters. For example, the Unicode code for the character ⊙ is 2299. You can write the following character literal to express this character:

```
'⊙'
```

However, if you do not have the tool to produce that character using your keyboard, you can escape it this way:

```
'\u2299'
```

Primitive Conversions

When dealing with different data types, you often need to perform conversions. For example, assigning the value of a variable to another variable involves a conversion. If both variables have the same type, the assignment will always succeed. Conversion from a type to the same type is called identity conversion. For example, the following operation is guaranteed to be successful:

```
int a = 90;
int b = a;
```

However, conversion to a different type is not guaranteed to be successful or even possible. There are two other kinds of primitive conversions, the widening conversion and the narrowing conversion.

The Widening Conversion

The widening primitive conversion occurs from a type to another type whose size

is the same or larger than that of the first type, such as from **int** (32 bits) to **long** (64 bits). The widening conversion is permitted in the following cases:

- **byte** to **short**, **int**, **long**, **float**, or **double**
- **short** to **int**, **long**, **float**, or **double**
- **char** to **int**, **long**, **float**, or **double**
- **int** to **long**, **float**, or **double**
- **long** to **float** or **double**
- **float** to **double**

A widening conversion from an integer type to another integer type will not risk information loss. At the same token, a conversion from **float** to **double** preserves all the information. However, a conversion from an **int** or a **long** to a **float** may result in loss of precision.

The widening primitive conversion occurs implicitly. You do not need to do anything in your code. For example:

```
int a = 10;
long b = a; // widening conversion
```

The Narrowing Conversion

The narrowing conversion occurs from a type to a different type that has a smaller size, such as from a **long** (64 bits) to an **int** (32 bits). In general, the narrowing primitive conversion can occur in these cases:

- **short** to **byte** or **char**
- **char** to **byte** or **short**
- **int** to **byte**, **short**, or **char**
- **long** to **byte**, **short**, or **char**
- **float** to **byte**, **short**, **char**, **int**, or **long**
- **double** to **byte**, **short**, **char**, **int**, **long**, or **float**

Unlike the widening primitive conversion, the narrowing primitive conversion must be explicit. You need to specify the target type in parentheses. For example, here is a narrowing conversion from **long** to **int**.

```
long a = 10;
int b = (int) a; // narrowing conversion
```

The **(int)** on the second line tells the compiler that a narrowing conversion should occur.

The narrowing conversion may incur information loss, if the converted value is larger than the capacity of the target type. The preceding example did not cause information loss because 10 is small enough for an **int**. However, in the following conversion, there is some information loss because 9876543210L is too big for an **int**.

```
long a = 9876543210L;
int b = (int) a; // the value of b is now 1286608618
```

A narrowing conversion that results in information loss introduces a defect in your program.

Operators

A computer program is a collection of operations that together achieve a certain

function. There are many types of operations, including addition, subtraction, multiplication, division, and bit shifting. In this section you will learn various Java operations.

An operator performs an operation on one, two, or three operands. Operands are the objects of an operation and the operator is a symbol representing the action. For example, here is an additive operation:

```
x + 4
```

In this case, **x** and 4 are the operands and + is the operator.

An operator may or may not return a result.

Note

Any legal combination of operators and operands are called an expression. For example, **x + 4** is an expression. A boolean expression results in either **true** or **false**. An integer expression produces an integer. And, the result of a floating-point expression is a floating point number.

Operators that require only one operand are called unary operators. There are a few unary operators in Java. Binary operators, the most common type of Java operator, take two operands. There is also one ternary operator, the **? :** operator, that requires three operands.

Table 2.4 list Java operators.

=	>	<	!	~	?	:				
==	<=	>=	!=	&&	||	++	--			
+	-	*	/	&	|	^	%	<<	>>	>>>
+=	-=	*=	/=	&=	|=	^=	%=	<<=	>>=	>>>=

Table 2.4: Java operators

In Java, there are six categories of operators.

- Unary operators
- Arithmetic operators
- Relational and conditional operators
- Shift and logical operators
- Assignment operators
- Other operators

Each of these operators is discussed in the following sections.

Unary Operators

Unary operators operate on one operand. There are six unary operators, all discussed in this section.

Unary Minus Operator –

The unary minus operator returns the negative of its operand. The operand must be a numeric primitive or a variable of a numeric primitive type. For example, in the following code, the value of **y** is -4.5;

```
float x = 4.5f;
float y = -x;
```

Unary Plus Operator +

This operator returns the value of its operand. The operand must be a numeric

primitive or a variable of a numeric primitive type. For example, in the following code, the value of **y** is 4.5.

```
float x = 4.5f;
float y = +x;
```

This operator does not have much significance since its absence makes no difference.

Increment Operator ++

This operator increments the value of its operand by one. The operand must be a variable of a numeric primitive type. The operator can appear before or after the operand. If the operator appears before the operand, it is called the prefix increment operator. If it is written after the operand, it becomes the postfix increment operator.

As an example, here is a prefix increment operator in action:

```
int x = 4;
++x;
```

After **++x**, the value of **x** is 5. The preceding code is the same as

```
int x = 4;
x++;
```

After **x++**, the value of **x** is 5.

However, if the result of an increment operator is assigned to another variable in the same expression, there is a difference between the prefix operator and its postfix twin. Consider this example.

```
int x = 4;
int y = ++x;
// y = 5, x = 5
```

The prefix increment operator is applied *before* the assignment. **x** is incremented to 5, and then its value is copied to **y**.

However, check the use of the postfix increment operator here.

```
int x = 4;
int y = x++;
// y = 4, x = 5
```

With the postfix increment operator, the value of the operand (**x**) is incremented *after* the value of the operand is assigned to another variable (**y**).

Note that the increment operator is most often applied to **int**s. However, it also works with other types of numeric primitives, such as **float** and **long**.

Decrement Operator --

This operator decrements the value of its operand by one. The operand must be a variable of a numeric primitive type. Like the increment operator, there are also the prefix decrement operator and the postfix decrement operator. For instance, the following code decrements **x** and assigns the value to **y**.

```
int x = 4;
int y = --x;
// x = 3; y = 3
```

In the following example, the postfix decrement operator is used:

```
int x = 4;
int y = x--;
// x = 3; y = 4
```

Logical Complement Operator !

This operator can only be applied to a **boolean** primitive or an instance of **java.lang.Boolean**. The value of this operator is **true** if the operand is **false**, and **false** if the operand is **true**. For example:

```
boolean x = false;
boolean y = !x;
// at this point, y is true and x is false
```

Bitwise Complement Operator ~

The operand of this operator must be an integer primitive or a variable of an integer primitive type. The result is the bitwise complement of the operand. For example:

```
int j = 2;
int k = ~j; // k = -3; j = 2
```

To understand how this operator works, you need to convert the operand to a binary number and reverse all the bits. The binary form of 2 in an integer is:

```
0000 0000 0000 0000 0000 0000 0000 0010
```

Its bitwise complement is

```
1111 1111 1111 1111 1111 1111 1111 1101
```

which is the representation of -3 in an integer.

Arithmetic Operators

There are four types of arithmetic operations: addition, subtraction, multiplication, division, and modulus. Each arithmetic operator is discussed here.

Addition Operator +

The addition operator adds two operands. The types of the operands must be convertible to a numeric primitive. For example:

```
byte x = 3;
int y = x + 5; // y = 8
```

Make sure the variable that accepts the addition result has a big enough capacity. For example, in the following code the value of **k** is -294967296 and not 4 billion.

```
int j = 2000000000; // 2 billion
int k = j + j; // not enough capacity. A bug!!!
```

On the other hand, the following works as expected:

```
long j = 2000000000; // 2 billion
long k = j + j; // the value of k is 4 billion
```

Subtraction Operator –

This operator performs subtraction between two operands. The types of the operands must be convertible to a numeric primitive type. As an example:

```
int x = 2;
int y = x - 1;      // y = 1
```

Multiplication Operator *

This operator perform multiplication between two operands. The type of the operands must be convertible to a numeric primitive type. As an example:

```
int x = 4;
int y = x * 4;      // y = 16
```

Division Operator /

This operator perform division between two operands. The left hand operand is the dividend and the right hand operand the divisor. Both the dividend and the divisor must be of a type convertible to a numeric primitive type. As an example:

```
int x = 4;
int y = x / 2;      // y = 2
```

Note that at runtime a division operation raises an error if the divisor is zero.

The result of a division using the / operator is always an integer. If the divisor does not divide the dividends equally, the remainder will be ignored. For example

```
int x = 4;
int y = x / 3;      // y = 1
```

The **java.lang.Math** class, explained in Chapter 5, "Core Classes," can perform more sophisticated division operations.

Modulus Operator %

This operator perform division between two operands and returns the remainder. The left hand operand is the dividend and the right hand operand the divisor. Both the dividend and the divisor must be of a type that is convertible to a numeric primitive type. For example the result of the following operation is 2.

```
8 % 3
```

Equality Operators

There are two equality operators, **==** (equal to) and **!=** (not equal to), both operating on two operands that can be integers, floating points, characters, or **boolean**. The outcome of equality operators is a **boolean**.

For example, the value of **c** is **true** after the comparison.

```
int a = 5;
int b = 5;
boolean c = a == b;
```

As another example,

```
boolean x = true;
```

```
boolean y = true;
boolean z = x != y;
```

The value of **z** is **false** after comparison because **x** is equal to **y**.

Relational Operators

There are five relational operators: **<**, **>**, **<=**, and **>=**, and **instanceof**. The last one is discussed in Chapter 4, "Objects and Classes." Each of the first four operators is explained in this section.

The **<**, **>**, **<=**, and **>=** operators operate on two operands whose types must be convertible to a numeric primitive type. Relational operations return a **boolean**.

The **<** operator evaluates if the value of the left-hand operand is less than the value of the right-hand operand. For example, the following operation returns **false**:

```
9 < 6
```

The **>** operator evaluates if the value of the left-hand operand is greater than the value of the right-hand operand. For example, this operation returns **true**:

```
9 > 6
```

The **<=** operator tests if the value of the left-hand operand is less than or equal to the value of the right-hand operand. For example, the following operation evaluates to **false**:

```
9 <= 6
```

The **>=** operator tests if the value of the left-hand operand is greater than or equal to the value of the right-hand operand. For example, this operation returns **true**:

```
9 >= 9
```

Conditional Operators

There are three conditional operators: the AND operator **&&**, the OR operator ||, and the **? :** operator. Each of these is detailed below.

The && operator

This operator takes two expressions as operands and both expressions must return a value that must be convertible to **boolean**. It returns **true** if both operands evaluate to **true**. Otherwise, it returns **false**. If the left-hand operand evaluates to **false**, the right-hand operand will not be evaluated. For example, the following returns **false**.

```
(5 < 3) && (6 < 9)
```

The || Operator

This operator takes two expressions as operands and both expressions must return a value that must be convertible to **boolean**. || returns **true** if one of the operands evaluates to **true**. If the left-hand operand evaluates to **true**, the right-hand operand will not be evaluated. For instance, the following returns **true**.

```
(5 < 3) || (6 < 9)
```

The ? : Operator

This operator operates on three operands. The syntax is

```
expression1 ? expression2 : expression3
```

Here, *expression1* must return a value convertible to **boolean**. If *expression1* evaluates to **true**, *expression2* is returned. Otherwise, *expression3* is returned.

For example, the following expression returns 4.

```
(8 < 4) ? 2 : 4
```

Shift Operators

A shift operator takes two operands whose type must be convertible to an integer primitive. The left-hand operand is the value to be shifted, the right-hand operand indicates the shift distance. There are three types of shift operators:

- the left shift operator **<<**
- the right shift operator **>>**
- the unsigned right shift operator **>>>**

The Left Shift Operator <<

The left shift operator bit-shifts a number to the left, padding the right bits with 0. The value of **n << s** is **n** left-shifted **s** bit positions. This is the same as multiplication by two to the power of s.

For example, left-shifting an **int** whose value is 1 with a shift distance of 3 (1 << 3) results in 8. Again, to figure this out, you convert the operand to a binary number.

```
0000 0000 0000 0000 0000 0000 0000 0001
```

Shifting to the left 3 shift units results in:

```
0000 0000 0000 0000 0000 0000 0000 1000
```

which is equivalent to 8 (the same as $1 * 2^3$).

Another rule is this. If the left-hand operand is an **int**, only the first five bits of the shift distance will be used. In other words, the shift distance must be within the range 0 and 31. If you pass an number greater than 31, only the first five bits will be used. This is to say, if **x** is an **int**, **x << 32** is the same as **x << 0**; **x << 33** is the same as **x << 1**.

If the left-hand operand is a **long**, only the first six bits of the shift distance will be used. In other words, the shift distance actually used is within the range 0 and 63.

The Right Shift Operator >>

The right shift operator >> bit-shifts the left-hand operand to the right. The value of **n >> s** is **n** right-shifted **s** bit positions. The resulting value is $\mathbf{n/2^s}$.

As an example, **16 >> 1** is equal to 8. To prove this, write the binary representation of 16.

```
0000 0000 0000 0000 0000 0000 0001 0000
```

Then, shifting it to the right by 1 bit results in.

```
0000 0000 0000 0000 0000 0000 0000 1000
```
which is equal to 8.

The Unsigned Right Shift Operator >>>

The value of **n >>> s** depends on whether **n** is positive or negative. For a positive **n**, the value is the same as **n >> s**.

If **n** is negative, the value depends on the type of **n**. If **n** is an **int**, the value is **(n>>s)+(2<<~s)**. If **n** is a **long**, the value is **(n>>s)+(2L<<~s)**.

Assignment Operators

There are twelve assignment operators:

```
=  +=  -=  *=  /=  %=  <<=  >>=  >>>=  &=  ^=  |=
```
Assignment operators take two operands whose type must be of an integral primitive. The left-hand operand must be a variable. For instance:

```
int x = 5;
```
Except for the assignment operator =, the rest work the same way and you should see each of them as consisting of two operators. For example, += is actually + and =. The assignment operator <<= has two operators, << and =.

The two-part assignment operators work by applying the first operator to both operands and then assign the result to the left-hand operand. For example **x += 5** is the same as **x = x + 5**.

x -= 5 is the same as **x = x - 5**.

x <<= 5 is equivalent to **x = x << 5**.

x &= 5 produces the same result as **x = x &= 5**.

Integer Bitwise Operators & | ^

The bitwise operators **&** | ^ perform a bit to bit operation on two operands whose types must be convertible to **int**. **&** indicates an AND operation, | an OR operation, and ^ an exclusive OR operation. For example,

```
0xFFFF & 0x0000 = 0x0000
0xF0F0 & 0xFFFF = 0xF0F0
0xFFFF | 0x000F = 0xFFFF
0xFFF0 ^ 0x00FF = 0xFF0F
```

Logical Operators & | ^

The logical operators **&** | ^ perform a logical operation on two operands that are convertible to **boolean**. **&** indicates an AND operation, | an OR operation, and ^ an exclusive OR operation. For example,

```
true & true  = true
true & false = false
true | false = true
false | false = false
true ^ true = false
false ^ false = false
false ^ true = true
```

Operator Precedence

In most programs, multiple operators often appear in an expression, such as.

```
int a = 1;
int b = 2;
int c = 3;
int d = a + b * c;
```

What is the value of **d** after the code is executed? If you say 9, you're wrong. It's actually 7.

Multiplication operator * takes precedence over addition operator +. As a result, multiplication will be performed before addition. However, if you want the addition to be executed first, you can use parentheses.

```
int d = (a + b) * c;
```

The latter will assign 9 to **d**.

Table 2.5 lists all the operators in the order of precedence. Operators in the same column have equal precedence.

Operator	
postfix operators	[ ] . (params) expr++ expr--
unary operators	++expr --expr +expr -expr ~ !
creation or cast	new (type)expr
multiplicative	* / %
additive	+ -
shift	<< >> >>>
relational	< > <= >= instanceof
equality	== !=
bitwise AND	&
bitwise exclusive OR	^
bitwise inclusive OR	|
logical AND	&&
logical OR	||
conditional	? :
assignment	= += -= *= /= %= &= ^= |= <<= >>= >>>=

Table 2.5: The precedence of operators

Note that parentheses have the highest precedence. Parentheses can also make expressions clearer. For example, consider the following code:

```
int x = 5;
int y = 5;
boolean z = x * 5 == y + 20;
```

The value of **z** after comparison is **true**. However, the expression is far from clear.

You can rewrite the last line using parentheses.

```
boolean z = (x * 5) == (y + 20);
```

which does not change the result because * and + have higher precedence than ==, but this makes the expression much clearer.

Promotion

Some unary operators (such as +, -, and ~) and binary operators (such as +, -, *, /) cause automatic promotion, i.e. elevation to a wider type such as from **byte** to **int**.

Consider the following code:

```
byte x = 5;
byte y = -x; // error
```

The second line surprisingly causes an error even though a byte can accommodate -5. The reason for this is the unary operator - causes the result of **-x** to be promoted to **int**. To rectify the problem, either change **y** to **int** or perform an explicit narrowing conversion like this.

```
byte x = 5;
byte y = (byte) -x;
```

For unary operators, if the type of the operand is **byte**, **short**, or **char**, the outcome is promoted to **int**.

For binary operators, the promotion rules are as follows.

- If any of the operands is of type **byte** or **short**, then both operands will be converted to **int** and the outcome will be an **int**.
- If any of the operands is of type **double**, then the other operand is converted to **double** and the outcome will be a **double**.
- If any of the operands is of type **float**, then the other operand is converted to **float** and the outcome will be a **float**.
- If any of the operands is of type **long**, then the other operand is converted to **long** and the outcome will be a **long**.

For example, the following code causes a compile error:

```
short x = 200;
short y = 400;
short z = x + y;
```

You can fix this by changing **z** to **int** or perform an explicit narrowing conversion of **x + y**, such as

```
short z = (short) (x + y);
```

Note that the parentheses around **x + y** is required, otherwise only **x** would be converted to **int** and the result of addition of a **short** and an **int** will be an **int**.

Comments

It is good practice to write comments throughout your code, sufficiently explaining what functionality a class provides, what a method does, what a field contains, and so forth.

There are two types of comments in Java, both with syntax similar to comments in C and C++.

- Traditional comments. Enclose a traditional comment in **/*** and ***/**.
- End-of-line comments. Use double slashes (//) which causes the rest of the line after // to be ignored by the compiler.

For example, here is a comment that describes a method

```
/*
  toUpperCase capitalizes the characters of in a String object
*/
public void toUpperCase(String s) {
```

Here is an end-of-line comment:

```
public int rowCount; //the number of rows from the database
```

Traditional comments do not nest, which means

```
/*
  /* comment 1 */
  comment 2 */
```

is invalid because the first ***/** after the first **/*** will terminate the comment. As such, the comment above will have the extra **comment 2 */**, which will generate a compiler error.

On the other hand, end-of-line comments can contain anything, including the sequences of characters **/*** and ***/**, such as this:

```
// /* this comment is okay */
```

Summary

This chapter presents Java language fundamentals, the basic concepts and topics that you should master before proceeding to more advanced subjects. Topics of discussion include character sets, variables, primitives, literals, operators, operator precedence, and comments.

Chapter 3 continues with statements, another important topic of the Java language.

Questions

1. What does ASCII stand for?
2. Does Java use ASCII characters or Unicode characters?
3. What are reference type variables, and what are primitive type variables?
4. How are constants implemented in Java?
5. What is an expression?
6. Name at least ten operators in Java.
7. What is the ternary operator in Java?
8. What is operator precedence?
9. Name two types of Java comments.

Chapter 3
Statements

A computer program is a compilation of instructions called statements. There are many types of statements in Java and some—such as **if**, **while**, **for**, and **switch**—are conditional statements that determine the flow of the program. Even though statements are not features specific to object-oriented programming, they are vital parts of the language fundamentals. This chapter discusses Java statements, starting with an overview and then providing details of each of them. The **return** statement, which is the statement to exit a method, is discussed in Chapter 4, "Objects and Classes."

An Overview of Java Statements

In programming, a statement is an instruction to do something. Statements control the sequence of execution of a program. Assigning a value to a variable is an example of a statement.

```
x = z + 5;
```

Even a variable declaration is a statement.

```
long secondsElapsed;
```

By contrast, an *expression* is a combination of operators and operands that gets evaluated. For example, **z + 5** is an expression.

In Java a statement is terminated with a semicolon and multiple statements can be written on a single line.

```
x = y + 1; z = y + 2;
```

However, writing multiple statements on a single line is not recommended as it obscures code readability.

Note

In Java, an empty statement is legal and does nothing:

```
;
```

Some expressions can be made statements by terminating them with a semicolon. For example, **x++** is an expression. However, this is a statement:

```
x++;
```

Statements can be grouped in a block. By definition, a block is a sequence of the following programming elements within braces:

- statements
- local class declarations
- local variable declaration statements

A statement and a statement block can be labeled. Label names follow the same rule as Java identifiers and are terminated with a colon. For example, the following

statement is labeled **sectionA**.

```
sectionA: x = y + 1;
```

And, here is an example of labeling a block:

```
start: {
    // statements
}
```

The purpose of labeling a statement or a block is so that it can be referenced by the **break** and **continue** statements.

The if Statement

The **if** statement is a conditional branch statement. The syntax of the **if** statement is either one of these two:

```
if (booleanExpression) {
    statement(s)
}


if (booleanExpression) {
    statement(s)
} else {
    statement(s)
}
```

If *booleanExpression* evaluates to **true**, the statements in the block following the **if** statement are executed. If it evaluates to **false**, the statements in the **if** block are not executed. If *booleanExpression* evaluates to **false** and there is an **else** block, the statements in the **else** block are executed.

For example, in the following **if** statement, the **if** block will be executed if **x** is greater than 4.

```
if (x > 4) {
    // statements
}
```

In the following example, the **if** block will be executed if **a** is greater than 3. Otherwise, the **else** block will be executed.

```
if (a > 3) {
    // statements
} else {
    // statements
}
```

Note that the good coding style suggests that statements in a block be indented.

If you are evaluating a boolean in your if statement, it's not necessary to use the == operator like this:

```
boolean fileExist = ...
if (fileExist == true) {
```

Instead, you can simply write

```
if (fileExists) {
```

By the same token, instead of writing

```
if (fileExists == false) {
```

write

```
if (!fileExists) {
```

If the expression to be evaluated is too long to be written in a single line, it is recommended that you use two units of indentation for subsequent lines. For example.

```
if (numberOfLoginAttempts < numberOfMaximumLoginAttempts
        || numberOfMinimumLoginAttempts > y) {
    y++;
}
```

If there is only one statement in an **if** or **else** block, the braces are optional.

```
if (a > 3)
    a++;
else
    a = 3;
```

However, this may pose what is called the dangling else problem. Consider the following example:

```
if (a > 0 || b < 5)
    if (a > 2)
        System.out.println("a > 2");
    else
        System.out.println("a < 2");
```

The **else** statement is dangling because it is not clear which **if** statement the **else** statement is associated with. An **else** statement is always associated with the immediately preceding **if**. Using braces makes your code clearer.

```
if (a > 0 || b < 5) {
    if (a > 2) {
        System.out.println("a > 2");
    } else {
        System.out.println("a < 2");
    }
}
```

If there are multiple selections, you can also use **if** with a series of **else** statements.

```
if (booleanExpression1) {
    // statements
} else if (booleanExpression2) {
    // statements
}
...
else {
    // statements
}
```

For example

```
if (a == 1) {
    System.out.println("one");
} else if (a == 2) {
    System.out.println("two");
```

```
} else if (a == 3) {
    System.out.println("three");
} else {
    System.out.println("invalid");
}
```

In this case, the **else** statements that are immediately followed by an **if** do not use braces. See also the discussion of the **switch** statement in the section, "The switch Statement" later in this chapter.

The while Statement

In many occasions, you may want to perform an action several times in a row. In other words, you have a block of code that you want executed repeatedly. Intuitively, this can be done by repeating the lines of code. For instance, a beep can be achieved using this line of code.

```
java.awt.Toolkit.getDefaultToolkit().beep();
```

And, to wait for half a second you use these lines of code. What this code does will become clear after you read Chapter 7.

```
try {
    Thread.currentThread().sleep(500);
} catch (Exception e) {
}
```

Therefore, to produce three beeps with a 500 milliseconds interval between two beeps, you can simply repeat the same code:

```
java.awt.Toolkit.getDefaultToolkit().beep();
try {
    Thread.currentThread().sleep(500);
} catch (Exception e) {
}
java.awt.Toolkit.getDefaultToolkit().beep();
try {
    Thread.currentThread().sleep(500);
} catch (Exception e) {
}
java.awt.Toolkit.getDefaultToolkit().beep();
```

However, there are circumstances where repeating code does not work. Here are some of those:

- The number of repetition is higher than 5, which means the number of lines of code increases five fold. If there is a line that you need to fix in the block, copies of the same line must also be modified.
- If the number of repetitions is not known in advance.

A much cleverer way is to put the repeated code in a loop. This way, you only write the code once but you can instruct Java to execute the code any number of times. One way to create a loop is by using the **while** statement, which is the topic of discussion of this section. Another way is to use the **for** statement, which is explained in the next section.

The **while** statement has the following syntax.

```
while (booleanExpression) {
```

```
    statement(s)
}
```

Here, *statement(s)* will be executed as long as *booleanExpression* evaluates to **true**. If there is only a single statement inside the braces, you may omit the braces. For clarity, however, you should always use braces even when there is only one statement.

As an example of the **while** statement, the following code prints integer numbers that are less than three.

```
int i = 0;
while (i < 3) {
    System.out.println(i);
    i++;
}
```

Note that the execution of the code in the loop is dependent on the value of **i**, which is incremented with each iteration until it reaches 3.

To produce three beeps with an interval of 500 milliseconds, use this code:

```
int j = 0;
while (j < 3) {
    java.awt.Toolkit.getDefaultToolkit().beep();
    try {
        Thread.currentThread().sleep(500);
    } catch (Exception e) {
    }
    j++;
}
```

Sometimes, you use an expression that always evaluates to **true** (such as the **boolean** literal **true**) but relies on the **break** statement to escape from the loop.

```
int k = 0;
while (true) {
    System.out.println(k);
    k++;
    if (k > 2) {
        break;
    }
}
```

You will learn about the **break** statement in the section, "The break Statement" later in this chapter.

The do-while Statement

The **do-while** statement is like the **while** statement, except that the associated block always gets executed at least once. Its syntax is as follows:

```
do {
    statement(s)
} while (booleanExpression);
```

With **do-while**, you put the statement(s) to be executed after the **do** keyword. Just like the **while** statement, you can omit the braces if there is only one statement within them. However, always use braces for the sake of clarity.

For example, here is an example of the **do-while** statement:

```
int i = 0;
do {
    System.out.println(i);
    i++;
} while (i < 3);
```

This prints the following to the console:

```
0
1
2
```

The following **do-while** demonstrates that at least the code in the **do** block will be executed once even though the initial value of **j** used to test the expression **j < 3** evaluates to **false**.

```
int j = 4;
do {
    System.out.println(j);
    j++;
} while (j < 3);
```

This prints the following on the console.

```
4
```

The for Statement

The **for** statement is like the **while** statement, i.e. you use it to enclose code that needs to be executed multiple times. However, **for** is more complex than **while**.

The **for** statement starts with an initialization, followed by an expression evaluation for each iteration and the execution of a statement block if the expression evaluates to **true**. An update statement will also be executed after the execution of the statement block for each iteration.

The **for** statement has following syntax:

```
for ( init ; booleanExpression ; update ) {
    statement(s)
}
```

Here, *init* is an initialization that will be performed before the first iteration, *booleanExpression* is a boolean expression which will cause the execution of *statement(s)* if it evaluates to **true**, and *update* is a statement that will be executed *after* the execution of the statement block. *init*, *expression*, and *update* are optional.

The **for** statement will stop only if one of the following conditions is met:

- *booleanEpression* evaluates to **false**
- A **break** or **continue** statement is executed
- A runtime error occurs.

It is common to declare a variable and assign a value to it in the initialization part. The variable declared will be visible to the *expression* and *update* parts as well as to the statement block.

For example, the following **for** statement loops three times and each time

prints the value of **i**.

```
for (int i = 0; i < 3; i++) {
    System.out.println(i);
}
```

The **for** statement starts by declaring an **int** named **i** and assigning 0 to it:

```
int i = 0;
```

It then evaluates the expression **i < 3**, which evaluates to **true** since **i** equals 0. As a result, the statement block is executed, and the value of **i** is printed. It then performs the update statement **i++**, which increments **i** to 1. That concludes the first loop.

The **for** statement then evaluates the value of **i < 3** again. The result is again **true** because **i** equals 1. This causes the statement block to be executed and **1** is printed on the console. Afterwards, the update statement **i++** is executed, incrementing **i** to 2. That concludes the third loop.

Next, the expression **i < 3** is evaluated and the result is **true** because **i** equals 2. This causes the statement block to be run and 2 is printed on the console. Afterwards, the update statement **i++** is executed, causing **i** to be equal to 3. This concludes the second loop.

Next, the expression **i < 3** is evaluated again, and the result is **false**. This stops the **for** loop.

This is what you see on the console:

```
0
1
2
```

Note that the variable **i** is not visible anywhere else since it is declared within the **for** loop.

Note also that if the statement block within **for** only consists of one statement, you can remove the braces, so in this case the above **for** statement can be rewritten as:

```
for (int i = 0; i < 3; i++)
    System.out.println(i);
```

However, using braces even if there is only one statement makes your code clearer.

Here is another example of the **for** statement.

```
for (int i = 0; i < 3; i++) {
    if (i % 2 == 0) {
        System.out.println(i);
    }
}
```

This one loops three times. For each iteration the value of **i** is tested. If **i** is even, its value is printed. The result of the **for** loop is as follows:

```
0
2
```

The following **for** loop is similar to the previous case, but uses **i += 2** as the update statement. As a result, it only loops twice, when **i** equals 0 and when it is 2.

```
for (int i = 0; i < 3; i += 2) {
```

```
    System.out.println(i);
}
```

The result is

```
0
2
```

A statement that decrements a variable is often used too. Consider the following **for** loop:

```
for (int i = 3; i > 0; i--) {
    System.out.println(i);
}
```

which prints:

```
3
2
1
```

The initialization part of the **for** statement is optional. In the following **for** loop, the variable **j** is declared outside the loop, so potentially **j** can be used from other points in the code outside the **for** statement block.

```
int j = 0;
for ( ; j < 3; j++) {
    System.out.println(j);
}
// j is visible here
```

As mentioned previously, the update statement is optional. The following **for** statement moves the update statement to the end of the statement block. The result is the same.

```
int k = 0;
for ( ; k < 3; ) {
    System.out.println(k);
    k++;
}
```

In theory, you can even omit the *booleanExpression* part. For example, the following **for** statement does not have one, and the loop is only terminated with the **break** statement. See the section, “The break Statement” for more information.

```
int m = 0;
for ( ; ; ) {
    System.out.println(m);
    m++;
    if (m > 4) {
        break;
    }
}
```

If you compare **for** and **while**, you’ll see that you can always replace the **while** statement with **for**. This is to say that

```
while (expression) {
    ...
}
```

can always be written as

```
for ( ; expression; ) {
    ...
}
```

Note
In addition, **for** can iterate over an array or a collection. See Chapters 5, "Core Classes" and Chapter 11, "The Collections Framework" for the discussions of the enhanced **for**.

The break Statement

The **break** statement is used to break from an enclosing **do**, **while**, **for**, or **switch** statement. It is a compile error to use **break** anywhere else.

For example, consider the following code

```
int i = 0;
while (true) {
    System.out.println(i);
    i++;
    if (i > 3) {
        break;
    }
}
```

The result is

```
0
1
2
3
```

Note that **break** breaks the loop without executing the rest of the statements in the block.

Here is another example of break, this time in a **for** loop.

```
int m = 0;
for ( ; ; ) {
    System.out.println(m);
    m++;
    if (m > 4) {
        break;
    }
}
```

The **break** statement can be followed by a label. The presence of a label will transfer control to the start of the code identified by the label. For example, consider this code.

```
start:
for (int i = 0; i < 3; i++) {
    for (int j = 0; j < 4; j++) {
        if (j == 2) {
            break start;
        }
        System.out.println(i + ":" + j);
    }
```

```
}
```

The use of label start identifies the first **for** loop. The statement **break start;** therefore breaks from the first loop. The result of running the preceding code is as follows.

```
0:0
0:1
```

Java does not have a goto statement like in C or C++, and labels are meant as a form of goto. However, just as using goto in C/C++ may obscure your code, the use of labels in Java may make your code unstructured. The general advice is to avoid labels if possible and to always use them with caution.

The continue Statement

The **continue** statement is like **break** but it only stops the execution of the current iteration and causes control to begin with the next iteration.

For example, the following code prints the number 0 to 9, except 5.

```
for (int i = 0; i < 10; i++) {
    if (i == 5) {
        continue;
    }
    System.out.println(i);
}
```

When **i** is equals to 5, the expression of the **if** statement evaluates to **true** and causes the **continue** statement to be called. As a result, the statement below it that prints the value of **i** is not executed and control continues with the next loop, i.e. for **i** equal to 6.

As with **break**, **continue** may be followed by a label to identify which enclosing loop to continue to. As with labels with **break**, employ **continue label** with caution and avoid it if you can.

Here is an example of **continue** with a label.

```
start:
for (int i = 0; i < 3; i++) {
    for (int j = 0; j < 4; j++) {
        if (j == 2) {
            continue start;
        }
        System.out.println(i + ":" + j);
    }
}
```

The result of running this code is as follows:

```
0:0
0:1
1:0
1:1
2:0
2:1
```

The switch Statement

An alternative to a series of **else if**, as discussed in the last part of the section, "The if Statement," is the **switch** statement. **switch** allows you to choose a block of statements to run from a selection of code, based on the return value of an expression. The expression used in the **switch** statement must return an **int**, a **String**, or an enumerated value.

> **Note**
> The **String** class is discussed in Chapter 5, "Core Classes" and enumerated values in Chapter 10, "Enums."

The syntax of the **switch** statement is as follows.

```
switch(expression) {
case value_1 :
    statement(s);
    break;
case value_2 :
    statement(s);
    break;
  .
  .
  .
case value_n :
    statement(s);
    break;
default:
    statement(s);
}
```

Failure to add a **break** statement after a case will not generate a compile error but may have more serious consequences because the statements on the next case will be executed.

Here is an example of the **switch** statement. If the value of **i** is 1, "One player is playing this game." is printed. If the value is 2, "Two players are playing this game is printed." If the value is 3, "Three players are playing this game is printed. For any other value, "You did not enter a valid value." will be printed.

```
int i = ...;
switch (i) {
case 1 :
    System.out.println("One player is playing this game.");
    break;
case 2 :
    System.out.println("Two players are playing this game.");
    break;
case 3 :
    System.out.println("Three players are playing this game.");
    break;
default:
    System.out.println("You did not enter a valid value.");
}
```

For examples of switching on a String or an enumerated value, see Chapter 5, "Core Classes" and Chapter 10, "Enums," respectively.

Summary

The sequence of execution of a Java program is controlled by statements. In this chapter, you have learned the following Java control statements: **if**, **while**, **do-while**, **for**, **break**, **continue**, and **switch**. Understanding how to use these statements is crucial to writing correct programs.

Questions

1. What is the difference between an expression and a statement?
2. How do you escape from the following while loop?

```
while (true) {
    // statements
}
```

3. Is there any difference between using the postfix increment operator and the prefix increment operator as the update statement of a **for** loop?

```
for (int x = 0; x < length; x++)
for (int x = 0; x < length; ++x)
```

4. What will be printed on the console if the code below is executed:

```
int i = 1;
switch (i) {
case 1 :
    System.out.println("One player is playing this game.");
case 2 :
    System.out.println("Two players are playing this game.");
    break;
default:
    System.out.println("You did not enter a valid value.");
}
```

Hint: no **break** after case 1.

Chapter 4
Objects and Classes

Object-oriented programming (OOP) works by modeling applications on real-world objects. The benefits of OOP, as discussed in Introduction, are real, which explains why OOP is the paradigm of choice today and why OOP languages like Java are popular. This chapter introduces you to objects and classes. If you are new to OOP, you may want to read this chapter carefully. A good understanding of OOP is key to writing quality programs.

This chapter starts by explaining what an object is and what constitutes a class. It then teaches you how to create objects in Java using the **new** keyword, how objects are stored in memory, how classes can be organized into packages, how to use access control to achieve encapsulation, how the Java Virtual Machine (JVM) loads and links your objects, and how Java manages unused objects. In addition, method overloading and static class members are explained.

What Is a Java Object?

When developing an application in an OOP language, you create a model that resembles a real-life situation to solve your problem. Take for example a company payroll application, which can calculate the take home pay of an employee and the amount of income tax to be paid. An application like this would have a **Company** object to represent the company using the application, **Employee** objects that represent the employees in the company, **Tax** objects to represent the tax details of each employee, and so on. Before you can start programming such applications, however, you need to understand what Java objects are and how to create them.

Let's begin with a look at objects in life. Objects are everywhere, living (persons, pets, etc) and otherwise (cars, houses, streets, etc); concrete (books, televisions, etc) and abstract (love, knowledge, tax rate, regulations, and so forth). Every object has two features: attributes and actions the object is able to perform. For example, the following are some of a car's attributes:

- color
- number of tires
- plate number
- number of valves

Additionally, a car can perform these actions:

- run
- brake

As another example, a dog has the following attributes: color, age, type, weight, etc. And it also can bark, run, urinate, sniff, etc.

A Java object also has attribute(s) and can perform action(s). In Java, attributes are called fields and actions are called methods. In other programming languages

these may be known differently. For example, methods are often called functions.

Both fields and methods are optional, meaning some Java objects may not have fields but have methods and some others may have fields but not methods. Some, of course, have both attributes and methods and some have neither.

How do you create Java objects? This is the same as asking, "How do you make cars?" Cars are expensive objects that need careful design that takes into account many things, such as safety and cost-effectiveness. You need a good blueprint to make good cars. To create Java objects, you need similar blueprints: classes.

Java Classes

A class is a blueprint or a template to create objects of identical type. If you have an **Employee** class, you can create any number of **Employee** objects. To create **Street** objects, you need a **Street** class. A class determines what kind of objects you get. For example, if you create an **Employee** class that has **age** and **position** fields, all **Employee** objects created out of this **Employee** class will have **age** and **position** fields as well. No more no less. The class determines the object.

In summary, classes are an OOP tool that enable programmers to create the abstraction of a problem. In OOP, abstraction is the act of using programming objects to represent real-world objects. As such, programming objects do not need to have the details of real-world objects. For instance, if an **Employee** object in a payroll application needs only be able to work and receive a salary, then the **Employee** class needs only two methods, **work** and **receiveSalary**. OOP abstraction ignores the fact that a real-world employee can do many other things including eat, run, kiss, and kick.

Classes are the fundamental building blocks of a Java program. All program elements in Java must reside in a class, even if you are writing a simple program that does not require Java's object-oriented features. A Java beginner needs to consider three things when writing a class:

- the class name
- the fields
- the methods

There are other things that can be present in a class, but they will be discussed later.

A class declaration must use the keyword **class** followed by a class name. Also, a class has a body within braces. Here is a general syntax for a class:

```
class className {
    [class body]
}
```

For example, Listing 4.1 shows a Java class named **Employee**, where the lines in bold are the class body.

Listing 4.1: The Employee class

```
class Employee {
    int age;
    double salary;
}
```

Note

By convention, class names capitalize the initial of each word. For example, here are some names that follow the convention: **Employee**, **Boss**, **DateUtility**, **PostOffice**, **RegularRateCalculator**. This type of naming convention is known as Pascal naming convention. The other convention, the camel naming convention, capitalize the initial of each word, except the first word. Method and field names use the camel naming convention.

A class definition must be saved in a file that has the same name as the class name. The file name must also have the **java** extension. For instance, the **Employee** class in Listing 4.1 must be saved as **Employee.java**.

Note

In UML class diagrams, a class is represented by a rectangle that consists of three parts: the topmost part is the class name, the middle part is the list of fields, and the bottom part is the list of methods. (See Figure 4.1) The fields and methods can be hidden if showing them is not important.

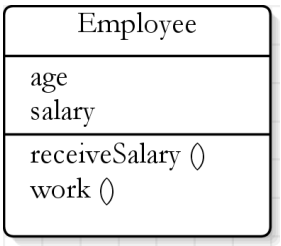

Figure 4.1: The Employee class in the UML class diagram

Fields

Fields are variables. They can be primitives or references to objects. For example, the **Employee** class in Listing 4.1 has two fields, **age** and **salary**. In Chapter 2, "Language Fundamentals" you learned how to declare and initialize variables of primitive types.

However, a field can also refer to another object. For instance, an **Empoyee** class may have an **address** field of type **Address**, which is a class that represents a street address:

```
Address address;
```

In other words, an object can contain other objects, that is if the class of the former contains variables that reference to the latter.

Field names should follow the camel naming convention. The initial of each word in the field, except for the first word, is written with a capital letter. For example, here are some "good" field names: **age**, **maxAge**, **address**, **validAddress**, **numberOfRows**.

Methods

Methods define actions that a class's objects (or instances) can do. A method has a declaration part and a body. The declaration part consists of a return value, the method name, and a list of arguments. The body contains code that perform the action.

To declare a method, use the following syntax:

```
returnType methodName (listOfArguments)
```

The return type of a method can be a primitive, an object, or void. The return type **void** means that the method returns nothing. The declaration part of a method is also called the signature of the method.

For example, here is the **getSalary** method that returns a **double**.

```
double getSalary()
```

The **getSalary** method does not accept arguments.

As another example, here is a method that returns an **Address** object.

```
Address getAddress()
```

And, here is a method that accepts an argument:

```
int negate(int number)
```

If a method takes more than one argument, two arguments are separated by a comma. For example, the following **add** method takes two **int**s and return an **int**.

```
int add(int a, int b)
```

Also note that a method may have a variable number of arguments. For details, see the section, "Varargs" in Chapter 5, "Core Classes."

The Method main

A special method called **main** provides the entry point to an application. An application normally has many classes and only one of the classes needs to have a **main** method. This method allows the class containing it to be invoked.

The signature of the **main** method is as follows.

```
public static void main(String[] args)
```

If you wonder why there is "public static void" before **main**, you will get the answer towards the end of this chapter.

In addition, you can pass arguments to **main** when using **java** to run a class. To pass arguments, type them after the class name. Two arguments are separated by a space.

```
java className arg1 arg2 arg3 ...
```

All arguments must be passed as strings. For instance, to pass two arguments, "1" and "safeMode" when running the **Test** class, you type this:

```
java Test 1 safeMode
```

Strings are discussed in Chapter 5, "Core Classes."

Constructors

Every class must have at least one constructor. Otherwise, no objects could be created out of the class and the class would be useless. As such, if your class does not explicitly define a constructor, the compiler adds one for you.

A constructor is used to construct an object. A constructor looks like a method and is sometimes called a constructor method. However, unlike a method, a constructor does not have a return value, not even **void**. Additionally, a constructor must have the same name as the class.

The syntax for a constructor is as follows.

```
constructorName (listOfArguments) {
```

```
    [constructor body]
}
```

A constructor may have zero argument, in which case it is called a no-argument (or no-arg, for short) constructor. Constructor arguments can be used to initialize the fields in the object.

If the Java compiler adds a no-arg constructor to a class because the class has none, the addition will be implicit, i.e. it will not be displayed in the source file. However, if there is a constructor, regardless of the number of arguments it accepts, no constructor will be added to the class by the compiler.

As an example, Listing 4.2 adds two constructors to the **Employee** class in Listing 4.1.

Listing 4.2: The Employee class with constructors

```
public class Employee {
    public int age;
    public double salary;
    public Employee() {
    }
    public Employee(int ageValue, double salaryValue) {
        age = ageValue;
        salary = salaryValue;
    }
}
```

The second constructor is particularly useful. Without it, to assign values to age and position, you would need to write extra lines of code to initialize the fields:

```
employee.age = 20;
employee.salary = 90000.00;
```

With the second constructor, you can pass the values at the same time you create an object.

```
new Employee(20, 90000.00);
```

The **new** keyword is new to you, but you will learn how to use it in the next section.

Class Members in UML Class Diagrams

Figure 4.1 depicts a class in a UML class diagram. The diagram provides a quick summary of all fields and methods. You could do more in UML. UML allows you to include field types and method signatures. For example, Figure 4.2 presents the **Book** class with five fields and one method.

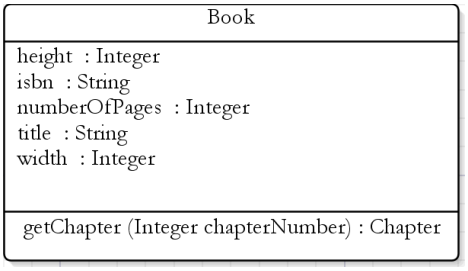

Figure 4.2: Including class member information in a class diagram

Note that in a UML class diagram a field and its type is separated by a colon. A

method's argument list is presented in parentheses and its return type is written after a colon.

Creating Objects

Now that you know how to write a class, it is time to learn how to create an object from a class. An object is also called an instance. The word construct is often used in lieu of create, thus constructing an **Employee** object. Another term commonly used is *instantiate*. Instantiating the **Employee** class is the same as creating an instance of **Employee**.

There are a number of ways to create an object, but the most common one is by using the **new** keyword. **new** is always followed by the constructor of the class to be instantiated. For example, to create an **Employee** object, you write:

```
new Employee();
```

Most of the time, you will want to assign the created object to an object variable (or a reference variable), so that you can manipulate the object later. To achieve this, you just need to declare an object reference with the same type as the object. For instance:

```
Employee employee = new Employee();
```

Here, **employee** is an object reference of type **Employee**.

Once you have an object, you can call its methods and access its fields, by using the object reference that was assigned the object. You use a period (.) to call a method or a field. For example:

```
objectReference.methodName
objectReference.fieldName
```

The following code, for instance, creates an **Employee** object and assigns values to its **age** and **salary** fields:

```
Employee employee = new Employee();
employee.age = 24;
employee.salary = 50000;
```

When an object is created, the JVM also performs initialization that assign default values to fields. This will be discussed further in the section, "Object Creation Initialization" later in this chapter.

The null Keyword

A reference variable refers to an object. There are times, however, when a reference variable does not have a value (it is not referencing an object). Such a reference variable is said to have a null value. For example, the following class level reference variable is of type **Book** but has not been assigned a value;

```
Book book; // book is null
```

If you declare a local reference variable within a method but do not assign an object to it, you will need to assign null to it to satisfy the compiler:

```
Book book = null;
```

Class-level reference variables will be initialized when an instance is created,

therefore you do not need to assign **null** to them.

Trying to access the field or method of a null variable reference raises an error, such as in the following code:

```
Book book = null;
System.out.println(book.title); // error because book is null
```

You can test if a reference variable is **null** by using the == operator. For instance.

```
if (book == null) {
    book = new Book();
}
System.out.println(book.title);
```

Objects in Memory

When you declare a variable in your class, either in the class level or in the method level, you allocate memory space for data that will be assigned to the variable. For primitives, it is easy to calculate the amount of memory taken. For example, declaring an **int** costs you four bytes and declaring a **long** sets you back eight bytes. However, calculation for reference variables is different.

When a program runs, some memory space is allocated for data. This data space is logically divided into two, the stack and the heap. Primitives are allocated in the stack and Java objects reside in the heap.

When you declare a primitive, a few bytes are allocated in the stack. When you declare a reference variable, some bytes are also set aside in the stack, but the memory does not contain an object's data, it contains the address of the object in the heap. In other words, when you declare

```
Book book;
```

Some bytes are set aside for the reference variable **book**. The initial value of **book** is **null** because there is not yet object assigned to it. When you write

```
Book book = new Book();
```

you create an instance of **Book**, which is stored in the heap, and assign the address of the instance to the reference variable **book**. A Java reference variable is like a C++ pointer except that you cannot manipulate a reference variable. In Java, a reference variable is used to access the member of the object it is referring to. Therefore, if the **Book** class has the public **review** method, you can call the method by using this syntax:

```
book.review();
```

An object can be referenced by more than one reference variable. For example,

```
Book myBook = new Book();
Book yourBook = myBook;
```

The second line copies the value of **myBook** to **yourBook**. As a result, **yourBook** is now referencing the same **Book** object as **myBook**.

Figure 4.3 illustrates memory allocation for a **Book** object referenced by **myBook** and **yourBook**.

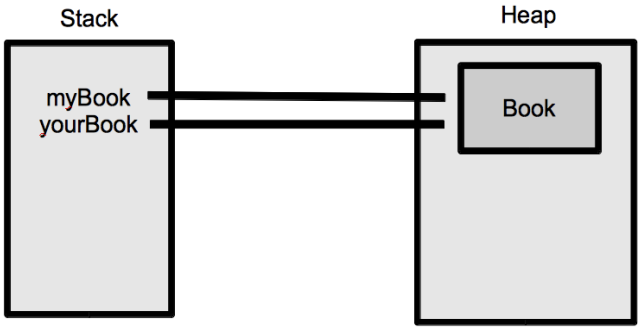

Figure 4.3: An object referenced by two variables

On the other hand, the following code creates two different **Book** objects:

```
Book myBook = new Book();
Book yourBook = new Book();
```

The memory allocation for this code is illustrated in Figure 4.4.

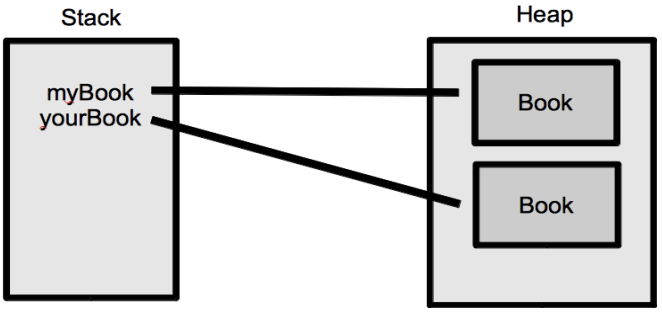

Figure 4.4: Two objects referenced by two variables

Now, how about an object that contains another object? For example, consider the code in Listing 4.3 that shows the **Employee** class that contains an **Address** class.

Listing 4.3: The Employee class that contains another class

```
package app04;
public class Employee {
    Address address = new Address();
}
```

When you create an **Employee** object using the following code, an **Address** object is also created.

```
Employee employee = new Employee();
```

Figure 4.5 depicts the position of each object in the heap.

It turns out that the **Address** object is not really inside the **Employee** object. However, the **address** field within the **Employee** object has a reference to the **Address** object, thus allowing the **Employee** object to manipulate the **Address** object. Because in Java there is no way of accessing an object except through a reference variable assigned the object's address, no one else can access the **Address** object 'within' the **Employee** object.

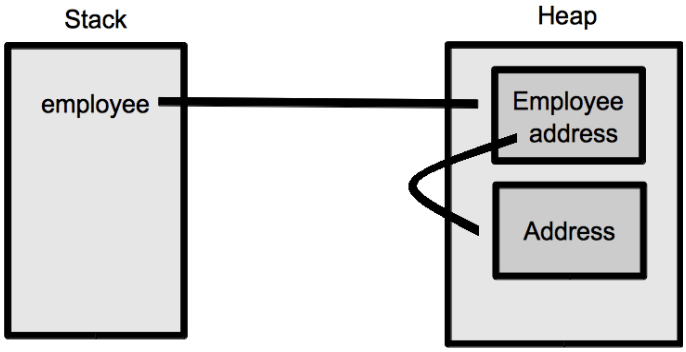

Figure 4.5: An object "within" another object

Java Packages

If you are developing an application that consists of different parts, you may want to organize your classes to retain maintainability. With Java, you can group related classes or classes with similar functionality in packages. For example, standard Java classes come in packages. Java core classes are in the **java.lang** package. All classes for performing input and output operations are members of the **java.io** package, and so on. If a package needs to be organized in more detail, you can create packages that share part of the name of the former. For example, the Java class library comes with the **java.lang.annotation** and **java.lang.reflect** packages. However, mind you that sharing part of the name does not make two packages related. The **java.lang** package and the **java.lang.reflect** package are different packages.

Package names that start with **java** are reserved for the core libraries. Consequently, you cannot create a package that starts with the word **java**. You can compile classes that belong to such a package, but you cannot run them.

In addition, packages starting with **javax** are meant for extension libraries that accompany the core libraries. You should not create packages that start with **javax** either.

In addition to class organization, packaging can avoid naming conflict. For example, an application may use the **MathUtil** class from company A and an identically named class from another company if both classes belong to different packages. For this purpose, by convention your package names should be based on your domain name in reverse. Therefore, Sun's package names start with **com.sun**. My domain name is **brainysoftware.com**, so it's appropriate for me to start my package name with **com.brainysoftware**. For example, I would place all my applets in the **com.brainysoftware.applet** package and my servlets in **com.brainysoftware.servlet**.

A package is not a physical object, and therefore you do not need to create one. To group a class in a package, use the keyword **package** followed by the package name. For example, the following **MathUtil** class is part of the **com.brainysoftware.common** package:

```
package com.brainysoftware.common;
public class MathUtil {
    ...
}
```

Java also introduces the term *fully qualified name*, which refers to a class name that carries with it its package name. The fully qualified name of a class is its package name followed by a period and the class name. Therefore, the fully qualified name of the **MathUtil** class that belongs to package **com.sun.common** is **com.sun.common.MathUtil**.

A class that has no package declaration is said to belong to the default package. For example, the **Employee** class in Listing 4.1 belongs to the default package. You should always use a package because types in the default package cannot be used by other types outside the default package (except by using a technique called reflection). It is a bad idea for a class to not have a package.

Even though a package is not a physical object, package names have a bearing on the physical location of their class source files. A package name represents a directory structure in which a period in a package name indicates a subfolder. For example, all Java source files in the **com.brainysoftware.common** package must reside in the **common** directory that is a subdirectory of the **brainysoftware** directory. In turn, the latter must be a subdirectory of the **com** directory. Figure 4.6 depicts a folder structure for the **com.brainysoftware.common.MathUtil** class.

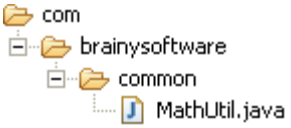

Figure 4.6: The physical location of a class in a package

Compiling a class in a non-default package presents a challenge for beginners. To compile such a class, you need to include the package name, replacing the dot (.) with /. For example, to compile the **com.brainysoftware.common.MathUtil** class, change directory to the working directory (the directory which is the parent directory of **com**) and type

```
javac com/brainysoftware/common/MathUtil.java
```

By default, **javac** will place the result in the same directory structure as the source. In this case, the **MathUtil.class** file will be created in the **com/brainysoftware/common** directory.

Running a class that belongs to a package follows a similar rule: you must include the package name, replacing . with /. For example, to run the **com.brainysoftware.common.MathUtil** class, type the following from your working directory.

```
java com/brainysoftware/common/MathUtil
```

The packaging of your classes also affects the visibility of your classes, as you will witness in the next section.

Note

Code samples accompanying this book are grouped into packages too. The packages are named **app*XX***, where ***XX*** is the chapter number.

Encapsulation and Access Control

An OOP principle, encapsulation is a mechanism that protects parts of an object that need to be secure and exposes only parts that are safe to be exposed. A television is a good example of encapsulation. Inside it are thousands of electronic

components that together form the parts that can receive signals and decode them into images and sound. These components are not to be exposed to users, however, so Sony and other manufacturers wrap them in a strong metallic cover that does not break easily. For a television to be easy to use, it exposes buttons that the user can touch to turn on and off the set, adjust brightness, turn up and down the volume, and so on.

Back to encapsulation in OOP, let's take as an example a class that can encode and decode messages. The class exposes two methods called **encode** and **decode**, that users of the class can access. Internally, there are dozens of variables used to store temporary values and other methods that perform supporting tasks. The author of the class hides these variables and other methods because allowing access to them may compromise the security of the encoding/decoding algorithms. Besides, exposing too many things makes the class harder to use. As you can see later, encapsulation is a powerful feature.

Java supports encapsulation through access control. Access control is governed by access control modifiers. There are four access control modifiers in Java: **public**, **protected**, **private**, and the default access level. Access control modifiers can be applied to classes or class members. We'll look at them in the following subsections.

Class Access Control Modifiers

In an application with many classes, a class may be instantiated and used from other classes that are members of the same package or different packages. You can control from which packages your class can be "seen" by employing an access control modifier at the beginning of the class declaration.

A class can have either the public or the default access control level. You make a class public by using the **public** access control modifier. A class whose declaration bears no access control modifier has default access. A public class is visible from anywhere. Listing 4.4 shows a public class named **Book**.

Listing 4.4: The public Book class

```
package app04;
public class Book {
    String isbn;
    String title;
    int width;
    int height;
    int numberOfPages;
}
```

The **Book** class is a member of the **app04** package and has five fields. Since **Book** is public, it can be instantiated from any other classes. In fact, the majority of the classes in the Java core libraries are public classes. For example, here is the declaration of the **java.io.File** class:

```
public class File
```

A public class must be saved in a file that has the same name as the class, and the extension must be **java**. The **Book** class in Listing 4.4 must be saved in the **Book.java** file. Also, because **Book** belongs to package **app04**, the **Book.java** file must reside inside the **app04** directory.

Note

A Java source file can only contain one public class. However, it can contain multiple classes that are not public.

When there is no access control modifier preceding a class declaration, the class has the default access level. For example, Listing 4.5 presents the **Chapter** class that has the default access level.

Listing 4.5: The Chapter class, with the default access level

```
package app04;
class Chapter {
    String title;
    int numberOfPages;

    public void review() {
        Page page = new Page();
        int sentenceCount = page.numberOfSentences;
        int pageNumber = page.getPageNumber();
    }
}
```

Classes with the default access level can only be used by other classes that belong to the same package. For instance, the **Chapter** class can be instantiated from inside the **Book** class because **Book** belongs to the same package as **Chapter**. However, **Chapter** is not visible from other packages.

For example, you can add the following **getChapter** method inside the **Book** class:

```
Chapter getChapter() {
    return new Chapter();
}
```

On the other hand, if you try to add the same **getChapter** method to a class that does not belong to the **app04** package, a compile error will be raised.

Class Member Access Control Modifiers

Class members (methods, fields, constructors, etc) can have one of four access control levels: public, protected, private, and default access. The **public** access control modifier is used to make a class member public, the **protected** modifier to make a class member protected, and the **private** modifier to make a class member private. Without an access control modifier, a class member will have the default access level.

Table 4.1 shows the visibility of each access level.

Access Level	From classes in other packages	From classes in the same package	From child classes	From the same class
public	yes	yes	yes	yes
protected	no	yes	yes	yes
default	no	yes	no	yes
private	no	no	no	yes

Table 4.1: Class member access levels

Note

The default access is sometimes called package private. To avoid confusion, this book will only use the term default access.

A public class member can be accessed by any other classes that can access the class containing the class member. For example, the **toString** method of the **java.lang.Object** class is public. Here is the method signature:

```
public String toString()
```

Once you construct an **Object** object, you can call its **toString** method because **toString** is public.

```
Object obj = new Object();
obj.toString();
```

Recall that you access a class member by using this syntax:

referenceVariable.memberName

In the preceding code, **obj** is a reference variable to an instance of **java.lang.Object** and **toString** is the method defined in the **java.lang.Object** class.

A protected class member has a more restricted access level. It can be accessed only from

- any class in the same package as the class containing the member
- a child class of the class containing the member

Note

A child class is a class that extends another class. Chapter 6, "Inheritance" explains this concept.

For instance, consider the public **Page** class in Listing 4.6.

Listing 4.6: The Page class

```
package app04;
public class Page {
    int numberOfSentences = 10;
    private int pageNumber = 5;
    protected int getPageNumber() {
        return pageNumber;
    }
}
```

Page has two fields (**numberOfSentences** and **pageNumber**) and one method (**getPageNumber**). First of all, because the **Page** class is public, it can be instantiated from any other class. However, even if you can instantiate it, there is no guarantee you can access its members by using the *referenceVariable.memberName* syntax. It depends on from which class you are accessing the **Page** class's members.

Its **getPageNumber** method is protected, so it can be accessed from any classes that belong to **app04**, the package that houses the **Page** class. For example, consider the **review** method in the **Chapter** class (given in Listing 4.5).

```
public void review() {
    Page page = new Page();
    int sentenceCount = page.numberOfSentences;
    int pageNumber = page.getPageNumber();
}
```

The **Chapter** class can access the **getPageNumber** method because **Chapter** belongs to the same package as the **Page** class. Therefore, **Chapter** can access all protected members of the **Page** class.

The default access allows classes in the same package access a class member. For instance, the **Chapter** class can access the **Page** class's **numberOfSentences** field because the **Page** and **Chapter** classes belong to the same package. However, **numberOfSentences** is not accessible from a subclass of **Page** if the subclass

belongs to a different package. This differentiates the protected and default access levels and will be explained in detail in Chapter 6, "Inheritance."

A class's private members can only be accessed from inside the same class. For example, there is no way you can access the **Page** class's private **pageNumber** field from anywhere other than the **Page** class itself. However, look at the following code from the **Page** class definition.

```
private int pageNumber = 5;
protected int getPageNumber() {
    return pageNumber;
}
```

The **pageNumber** field is private, so it can be accessed from the **getPageNumber** method, which is defined in the same class. The return value of **getPageNumber** is **pageNumber**, which is private. Beginners are often confused by this kind of code. If **pageNumber** is private, why do we use it as a return value of a protected method (**getPageNumber**)? Note that access to **pageNumber** is still private, so other classes cannot modify this field. However, using it as a return value of a non-private method is allowed.

How about constructors? Access levels to constructors are the same as those for fields and methods. Therefore, constructors can have public, protected, default, and private access levels. You may think that all constructors must be public because the intention of having a constructor is to make the class instantiatable. However, to your surprise, this is not so. Some constructors are made private so that their classes cannot be instantiated by using the **new** keyword. Private constructors are normally used in singleton classes. If you are interested in this topic, there are articles on this topic that you can find easily on the Internet.

Note

In a UML class diagram, you can include information on class member access level. Prefix a public member with +, a protected member with # and a private member with -. Members with no prefix are regarded as having the default access level. Figure 4.7 shows the **Manager** class with members having various access levels.

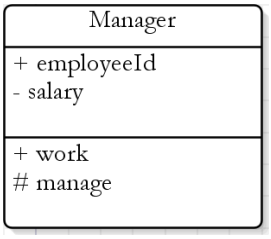

Figure 4.7: Including class member access level in a UML class diagram

---

The this Keyword

You use the **this** keyword from any method or constructor to refer to the current object. For example, if you have a class-level field having the same name as a local variable, you can use this syntax to refer to the former:

```
this.field
```

A common use is in the constructor that accepts values used to initialize fields. Consider the **Box** class in Listing 4.7.

Listing 4.7: The Box class

```
package com.brainysoftware.jdk5.app04;
public class Box {
    int length;
    int width;
    int height;
    public Box(int length, int width, int height) {
        this.length = length;
        this.width = width;
        this.height = height;
    }
}
```

The **Box** class has three fields, **length**, **width**, and **height**. Its constructor accepts three arguments used to initialize the fields. It is very convenient to use **length**, **width**, and **height** as the parameter names because they reflect what they are. Inside the constructor, **length** refers to the **length** argument, not the **length** field. **this.length** refers to the class-level **length** field.

It is of course possible to change the argument names, such as this.

```
public Box (int lengthArg, int widthArg, int heightArg) {
    length = lengthArg;
    width = widthArg;
    height = heightArg;
}
```

This way, the class-level fields are not shadowed by local variables and you do not need to use the **this** keyword to refer to the class-level fields However, using the **this** keyword spares you from having to think of different names for your method or constructor arguments.

Using Other Classes

It is common to use other classes from the class you are writing. Using classes in the same package as your current class is allowed by default. However, to use classes in other packages, you must first import the packages or the classes you want to use.

Java provides the keyword **import** to indicate that you want to use a package or a class from a package. For example, to use the **java.io.File** class from your code, you must have the following **import** statement:

```
package app04;
import java.io.File;

public class Demo {
    ...
}
```

Note that **import** statements must come after the **package** statement but before the class declaration. The **import** keyword can appear multiple times in a class.

```
package app04;
```

```
import java.io.File;
import java.util.List;

public class Demo {
    ...
}
```

Sometimes you need many classes in the same package. You can import all classes in the same package by using the wild character *. For example, the following code imports all members of the **java.io** package.

```
package app04;
import java.io.*;
public class Demo {
    ...
}
```

Now, not only can you use the **java.io.File** class, but you can use other members in the **java.io** package too. However, to make your code more readable, it is recommended that you import a package member one at a time. In other words, if you need to use both the **java.io.File** class and the **java.io.FileReader** class, it is better to have two **import** statements like the following than to use the * character.

```
import java.io.File;
import java.io.FileReader;
```

> **Note**
> Members of the **java.lang** package are imported automatically. Thus, to use the **java.lang.String**, for example, you do not need to explicitly import the class.

The only way to use classes that belong to other packages without importing them is to use the fully qualified names of the classes in your code. For example, the following code declares the **java.io.File** class using its fully qualified name.

```
java.io.File file = new java.io.File(filename);
```

If you import identically-named classes from different packages, you must use the fully qualified names when declaring the classes. For example, the Java core libraries contain the classes **java.sql.Date** and **java.util.Date**. Importing both upsets the compiler. In this case, you must write the fully qualified names of **java.sql.Date** and **java.util.Date** in your class to use them.

> **Note**
> Java classes can be deployed in a jar file. Appendix A details how to compile a class that uses other classes in a jar file. Appendix B shows how to run a Java class in a jar file. Appendix C provides instructions on the **jar** tool, a program that comes with the JDK to package your Java classes and related resources.
> A class that uses another class is said to "depend on" the latter. A UML diagram that depicts this dependency is shown in Figure 4.8. A dependency relationship is represented by a dashed line with an arrow. In Figure 4.8 the **Book** class is dependent on **Chapter** because the **getChapter** method returns a **Chapter** object.

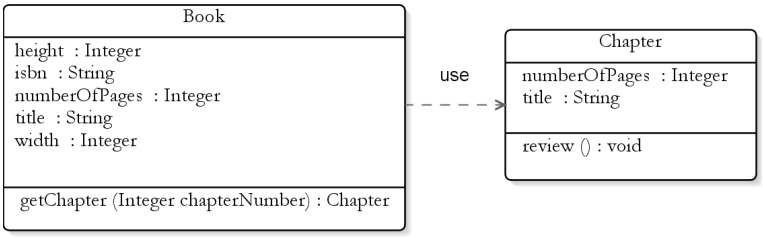

Figure 4.8: Dependency in the UML class diagram

Final Variables

Java does not reserve the keyword constant to create constants. However, in Java you can prefix a variable declaration with the keyword **final** to make its value unchangeable. You can make both local variables and class fields final.

For example, the number of months in a year never changes, so you can write:

```
final int numberOfMonths = 12;
```

As another example, in a class that performs mathematical calculation, you can declare the variable **pi** whose value is equal to 22/7 (the circumference of a circle divided by its diameter, in math represented by the Greek letter π).

```
final float pi = (float) 22 / 7;
```

Once assigned a value, the value cannot change. Attempting to change it will result in a compile error.

Note that the casting **(float)** after **22 / 7** is needed to convert the value of division to **float**. Otherwise, an **int** will be returned and the **pi** variable will have a value of 3.0, instead of 3.1428.

Also note that since Java uses Unicode characters, you can simply define the variable **pi** as π if you don't think typing it is harder than typing **pi**.

```
final float π = (float) 22 / 7;
```

Note

You can also make a method final, thus prohibiting it from being overridden in a subclass. This will be discussed in Chapter 6, "Inheritance."

Static Members

You have learned that to access a public field or method of an object, you use a period after the object reference, such as:

```
// Create an instance of Book
Book book = new Book();
// access the review method
book.review();
```

This implies that you must create an object first before you can access its members. However, in previous chapters, there were examples that used **System.out.print** to print values to the console. You may have noticed that you

could call the **out** field without first having to construct a **System** object. How come you did not have to do something like this?

```
System ref = new System();
ref.out;
```

Rather, you use a period after the class name:

```
System.out
```

Java (and many OOP languages) supports the notion of static members, which are class members that can be called without first instantiating the class. The **out** field in **java.lang.System** is static, which explains why you can write **System.out**.

Static members are not tied to class instances. Rather, they can be called without having an instance. In fact, the method **main**, which acts as the entry point to a class, is static because it must be called before any object is created.

To create a static member, you use the keyword **static** in front of a field or method declaration. If there is an access modifier, the **static** keyword may come before or after the access modifier. These two are correct:

```
public static int a;
static public int b;
```

However, the first form is more often used.

For example, Listing 4.8 shows the **MathUtil** class with a static method:

Listing 4.8: The MathUtil class

```
package app04;
public class MathUtil {
    public static int add(int a, int b) {
        return a + b;
    }
}
```

To use the **add** method, you can simply call it this way:

```
MathUtil.add(a, b)
```

The term instance methods/fields are used to refer to non-static methods and fields.

From inside a static method, you cannot call instance methods or instance fields because they only exist after you create an object. You can access other static methods or fields from a static method, however.

A common confusion that a beginner often encounter is when they cannot compile their class because they are calling instance members from the **main** method. Listing 4.9 shows such a class.

Listing 4.9: Calling non-static members from a static method

```
package app04;
public class StaticDemo {
    public int b = 8;
    public static void main(String[] args) {
        System.out.println(b);
    }
}
```

The line in bold causes a compile error because it attempts to access non-static field **b** from the **main** static method. There are two solutions to this.

1. Make **b** static

2. Create an instance of the class, then access **b** by using the object reference.

Which solution is appropriate depends on the situation. It often takes years of OOP experience to come up with a good decision that you're comfortable with.

Note

You can only declare a static variable in a class level. You cannot declare local static variables even if the method is static.

How about static reference variables? You can declare static reference variables. The variable will contain an address, but the object referenced is stored in the heap. For instance

```
static Book book = new Book();
```

Static reference variables provide a good way of exposing the same object that needs to be shared among other different objects.

Note

In UML class diagrams, static members are underlined. For example, Figure 4.9 shows the **MathUtil** class with the static method **add**.

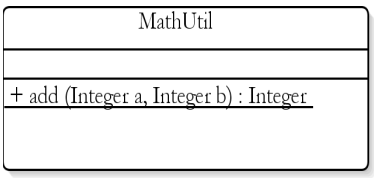

Figure 4.9: Static members in a UML class diagram

Static Final Variables

In the section, "Final Variables" earlier in the chapter, you learned that you could create a final variable by using the keyword **final**. However, final variables at a class level or local variables will always have the same value when the program is run. If you have multiple objects of the same class with final variables, the value of the final variables in those objects will have the same values. It is more common (and also more prudent) to make a final variable static too. This way, all objects share the same value.

The naming convention for static final variables is to have them in upper case and separate two words with an underscore. For example

```
static final int NUMBER_OF_MONTHS = 12;
static final float PI = (float) 22 / 7;
```

The positions of **static** and **final** are interchangeable, but it is more common to use "static final" than "final static."

If you want to make a static final variable accessible from outside the class, you can make it public too:

```
public static final int NUMBER_OF_MONTHS = 12;
public static final float PI = (float) 22 / 7;
```

To better organize your constants, sometimes you want to put all your static final variables in a class. This class most often does not have a method or other fields and is never instantiated.

For example, sometimes you want to represent a month as an **int**, therefore January is 1, February is 2, and so on. Then, you use the word January instead of number 1 because it's more descriptive. Listing 4.10 shows the **Months** class that contains the names of months and its representation.

Listing 4.10: The Months class

```
package app04;
public class Months {
    public static final int JANUARY = 1;
    public static final int FEBRUARY = 2;
    public static final int MARCH = 3;
    public static final int APRIL = 4;
    public static final int MAY = 5;
    public static final int JUNE = 6;
    public static final int JULY = 7;
    public static final int AUGUST = 8;
    public static final int SEPTEMBER = 9;
    public static final int OCTOBER = 10;
    public static final int NOVEMBER = 11;
    public static final int DECEMBER = 12;
}
```

In your code, you can get the representation of January by writing.

```
int thisMonth = Months.JANUARY;
```

Classes similar to **Months** are very common prior to Java 5. However, Java now offers the new type **enum** that can eliminate the need for public static final variables. **enum** is explain in Chapter 10, "Enums."

Static final reference variables are also possible. However, note that only the variable is final, which means once it is assigned an address to an instance, it cannot be assigned another object of the same type. The fields in the referenced object itself can be changed.

In the following line of code

```
public static final Book book = new Book();
```

book always refer to this particular instance of **Book**. Reassigning it to another **Book** object raises a compile error:

```
book = new Book(); // compile error
```

However, you can change the **Book** object's field value.

```
book.title = "No Excuses";  // assuming the title field is public
```

Static import

There are a number of classes in the Java core libraries that contain static final fields. One of them is the **java.util.Calendar** class, that has the static final fields representing days of the week (**MONDAY**, **TUESDAY**, etc). To use a static final field in the **Calendar** class, you must first import the **Calendar** class.

```
import java.util.Calendar;
```

Then, you can use it by using the notation *className.staticField*.

```
if (today == Calendar.SATURDAY)
```

However, you can also import static fields using the **import static** keywords. For

example, you can do

```
import static java.util.Calendar.SATURDAY;
```

Then, to use the imported static field, you do not need the class name:

```
if (today == SATURDAY)
```

> **Note**
> The **java.util.Calendar** class is discussed in more detail in Chapter 5, "Core Classes."

Variable Scope

You have seen that you can declare variables in several different places:

- In a class body as class fields. Variables declared here are referred to as class-level variables.
- As parameters of a method or constructor.
- In a method's body or a constructor's body.
- Within a statement block, such as inside a **while** or **for** block.

Now it's time to learn the scope of variables.

Variable scope refers to the accessibility of a variable. The rule is that variables defined in a block are only accessible from within the block. The scope of the variable is the block in which it is defined. For example, consider the following **for** statement.

```
for (int x = 0; x < 5; x++) {
    System.out.println(x);
}
```

The variable **x** is declared within the **for** statement. As a result, **x** is only available from within this **for** block. It is not accessible or visible from anywhere else. When the JVM executes the **for** statement, it creates **x**. When it is finished executing the **for** block, it destroys **x**. After **x** is destroyed, **x** is said to be out of scope.

Rule number 2 is a nested block can access variables declared in the outer block. Consider this code.

```
for (int x = 0; x < 5; x++) {
    for (int y = 0; y < 3; y++) {
        System.out.println(x);
        System.out.println(y);
    }
}
```

The preceding code is valid because the inner **for** block can access **x**, which is declared in the outer **for** block.

Following the rules, variables declared as method parameters can be accessed from within the method body. Also, class-level variables are accessible from anywhere in the class.

If a method declares a local variable that has the same name as a class-level variable, the former will 'shadow' the latter. To access the class-level variable from inside the method body, use the **this** keyword.

Method Overloading

Method names are very important and should reflect what the methods do. In many circumstances, you may want to use the same name for multiple methods because they have similar functionality. For instance, the method **printString** may take a **String** argument and prints the string. However, the same class may also provide a method that prints part of a **String** and accepts two arguments, the **String** to be printed and the character position to start printing from. You want to call the latter method **printString** too because it does print a **String**, but that would be the same as the first **printString** method.

Thankfully, it is okay in Java to have multiple methods having the same name, as long as each method accept different sets of argument types. In other words, in our example, it is legal to have these two methods in the same class.

```
public String printString(String string)
public String printString(String string, int offset)
```

This feature is called method overloading.

The return value of the method is not taken into consideration. As such, these two methods must not exist in the same class:

```
public int countRows(int number);
public String countRows(int number);
```

This is because a method can be called without assigning its return value to a variable. In such situations, having the above **countRows** methods would confuse the compiler as it would not know which method is being called when you write

```
System.out.println(countRows(3));.
```

A trickier situation is depicted in the following methods whose signatures are very similar.

```
public int printNumber(int i) {
    return i*2;
}

public long printNumber(long l) {
    return l*3;
}
```

It is legal to have these two methods in the same class. However, you might wonder, which method is being called if you write **printNumber(3)**?

The key is to recall from Chapter 2, “Language Fundamentals” that a numeric literal will be translated into an **int** unless it is suffixed **L** or **l**.. Therefore, **printNumber(3)** will invoke this method:

```
public int printNumber(int i)
```

To call the second, pass a **long**:

```
printNumber(3L);
```

Note

Static methods can also be overloaded.

By Value or By Reference?

You can pass primitive variables or reference variables to a method. Primitive variables are passed by value and reference variables are passed by reference. What this means is when you pass a primitive variable, the JVM will copy the value of the passed-in variable to a new local variable. If you change the value of the local variable, the change will not affect the passed in primitive variable.

If you pass a reference variable, the local variable will refer to the same object as the passed in reference variable. If you change the object referenced within your method, the change will also be reflected in the calling code. Listing 4.11 shows the **ReferencePassingTest** class that demonstrates this.

Listing 4.11: The ReferencePassingTest class

```
package app04;
class Point {
    public int x;
    public int y;
}
public class ReferencePassingTest {
    public static void increment(int x) {
        x++;
    }
    public static void reset(Point point) {
        point.x = 0;
        point.y = 0;
    }
    public static void main(String[] args) {
        int a = 9;
        increment(a);
        System.out.println(a); // prints 9
        Point p = new Point();
        p.x = 400;
        p.y = 600;
        reset(p);
        System.out.println(p.x); // prints 0
    }
}
```

There are two methods in **ReferencePassingTest**, **increment** and **reset**. The **increment** method takes an **int** and increments it. The **reset** method accepts a **Point** object and resets its **x** and **y** fields.

Now pay attention to the **main** method. We passed **a** (whose value is 9) to the **increment** method. After the method invocation, we printed the value of **a** and you get 9, which means that the value of **a** did not change.

Afterwards, you create a **Point** object and assign the reference to **p**. You then initialize its fields and pass it to the **reset** method. The changes in the **reset** method affects the **Point** object because objects are passed by reference. As a result, when you print the value of **p.x**, you get 0.

Loading, Linking, and Initialization

Now that you've learned how to create classes and objects, let's take a look at what happens when the JVM executes a class.

You run a Java program by using the **java** tool. For example, you use the following command to run the **DemoTest** class.

```
java DemoTest
```

After the JVM is loaded into memory, it starts its job by invoking the **DemoTest** class's **main** method. There are three things the JVM will do next in the specified order: loading, linking, and initialization.

Loading

The JVM loads the binary representation of the Java class (in this case, the **DemoTest** class) to memory and may cache it in memory, just in case the class is used again in the future. If the specified class is not found, an error will be thrown and the process stops here.

Linking

There are three things that need to be done in this phase: verification, preparation, and resolution (optional). Verification means the JVM checks that the binary representation complies with the semantic requirements of the Java programming language and the JVM. If, for example, you tamper with a class file created as a result of compilation, the class file may no longer work.

Preparation prepares the specified class for execution. This involves allocating memory space for static variables and other data structured for that class.

Resolution checks if the specified class references other classes/interfaces and if the other classes/interfaces can also be found and loaded. Checks will be done recursively to the referenced classes/interfaces.

For example, if the specified class contains the following code:

```
MathUtil.add(4, 3)
```

the JVM will load, link, and initialize the **MathUtil** class before calling the static **add** method.

Or, if the following code is found in the **DemoTest** class:

```
Book book = new Book();
```

the JVM will load, link, and initialize the **Book** class before an instance of **Book** is created.

Note that a JVM implementation may choose to perform resolution at a later stage, i.e. when the executing code actually requires the use of the referenced class/interface.

Initialization

In this last step, the JVM initializes static variables with assigned or default values and executes static initializers (code in **static** blocks). Initialization occurs just before the **main** method is executed. However, before the specified class can be initialized, its parent class will have to be initialized. If the parent class has not been loaded and linked, the JVM will first load and link the parent class. Again,

when the parent class is about to be initialized, the parent's parent will be treated the same. This process occurs recursively until the initialized class is the topmost class in the hierarchy.

For example, if a class contains the following declaration

```
public static int z = 5;
```

the variable **z** will be assigned the value 5. If no initialization code is found, a static variable is given a default value. Table 4.2 lists default values for Java primitives and reference variables.

Type	Default Value
boolean	false
byte	0
short	0
int	0
long	0L
char	\u0000
float	0.0f
double	0.0d
object reference	null

Table 4.2: Default values for primitives and references

In addition, code in **static** blocks will be executed. For example, Listing 4.12 shows the **StaticCodeTest** class with static code that gets executed when the class is loaded. Like static members, you can only access static members from static code.

Listing 4.12: StaticCodeTest

```
package app04;
public class StaticInitializationTest {
    public static int a = 5;
    public static int b = a * 2;
    static {
        System.out.println("static");
        System.out.println(b);
    }
    public static void main(String[] args) {
        System.out.println("main method");
    }
}
```

If you run this class, you will see the following on your console:

```
static
10
main method
```

Object Creation Initialization

Initialization happens when a class is loaded, as described in the section "Linking, Loading, and Initialization" earlier in this chapter. However, you can also write code that performs initialization every time an instance of a class is created.

When the JVM encounters code that instantiates a class, the JVM does the following.

1. Allocates memory space for a new object, with room for the instance variables declared in the class plus room for instance variables declared in its parent classes.
2. Processes the invoked constructor. If the constructor has parameters, the JVM creates variables for these parameter and assigns them values passed to the constructor.
3. If the invoked constructor begins with a call to another constructor (using the **this** keyword), the JVM processes the called constructor.
4. Performs instance initialization and instance variable initialization for this class. Instance variables that are not assigned a value will be assigned default values (See Table 4.2). Instance initialization applies to code in braces:

```
{
    // code
}
```

5. Executes the rest of the body of the invoked constructor.
6. Returns a reference variable that refers to the new object.

Note that instance initialization is different from static initialization. The latter occurs when a class is loaded and has nothing to do with instantiation. Instance initialization, by contrast, is performed when an object is created. In addition, unlike static initializers, instance initialization may access instance variables.

For example, Listing 4.13 presents a class named **InitTest1** that has the initialization section. There is also some static initialization code to give you the idea of what is being run.

Listing 4.13: The InitTest1 class

```
package app04;

public class InitTest1 {
    int x = 3;
    int y;
    // instance initialization code
    {
        y = x * 2;
        System.out.println(y);
    }

    // static initialization code
    static {
        System.out.println("Static initialization");
    }
    public static void main(String[] args) {
        InitTest1 test = new InitTest1();
        InitTest1 moreTest = new InitTest1();
    }
}
```

When run, the **InitTest** class prints the following on the console:

```
Static initialization
6
6
```

The static initialization is performed first, before any instantiation takes place. This

is where the JVM prints the "Static initialization" message. Afterwards, the **InitTest1** class is instantiated twice, explaining why you see "6" twice.

The problem with having instance initialization code is this. As your class grows bigger it becomes harder to notice that there exists initialization code.

Another way to write initialization code is in the constructor. In fact, initialization code in a constructor is more noticeable and hence preferable. Listing 4.14 shows the **InitTest2** class that puts initialization code in the constructor.

Listing 4.14: The InitTest2 class

```
package app04;
public class InitTest2 {
    int x = 3;
    int y;
    // instance initialization code
    public InitTest2() {
        y = x * 2;
        System.out.println(y);
    }
    // static initialization code
    static {
        System.out.println("Static initialization");
    }
    public static void main(String[] args) {
        InitTest2 test = new InitTest2();
        InitTest2 moreTest = new InitTest2();
    }
}
```

The problem with this is when you have more than one constructor and each of them must call the same code. The solution is to wrap the initialization code in a method and let the constructors call them. Listing 4.15 shows this

Listing 4.15: The InitTest3 class

```
package app04;

public class InitTest3 {
    int x = 3;
    int y;
    // instance initialization code
    public InitTest3() {
        init();
    }
    public InitTest3(int x) {
        this.x = x;
        init();
    }
    private void init() {
        y = x * 2;
        System.out.println(y);
    }
    // static initialization code
    static {
        System.out.println("Static initialization");
    }
    public static void main(String[] args) {
```

```
        InitTest3 test = new InitTest3();
        InitTest3 moreTest = new InitTest3();
    }
}
```

Note that the **InitTest3** class is preferable because the calls to the **init** method from the constructors make the initialization code more obvious than if it is in an initialization block.

Comparing Objects

In real life, when I say "My car is the same as your car" I mean my car is of the same type as yours, as new as your car, has the same color, etc.

In Java, you manipulate objects by using the variables that reference them. Bear in mind that reference variables do not contain objects but rather contain addresses to the objects in the memory. Therefore, when you compare two reference variables **a** and **b**, such as in this code

```
if (a == b)
```

you are actually asking if **a** and **b** are referencing the same object, and not whether or not the objects referenced by **a** and **b** are identical.

Consider this example.

```
Object a = new Object();
Object b = new Object();
```

The type of object **a** references is identical to the type of object **b** references. However, **a** and **b** reference two different instances and **a** and **b** contains different memory addresses. Therefore, **(a == b)** returns **false**.

Comparing object references this way is hardly useful because most of the time you are more concerned with the objects, not the addresses of the objects. If what you want is compare objects, you need to look for methods specifically provided by the class to compare objects. For example, to compare two **String** objects, you can call its **equals** method. (See Chapter 5, "Core Classes") Whether or not comparing the contents of two objects is possible depends on whether or not the corresponding class supports it. A class can support object comparison by implementing the **equals** and **hashCode** methods it inherits from **java.lang.Object**. (See the section "java.lang.Object" in Chapter 5)

In addition, there are utility classes you can use to compare objects. See the discussion of **java.lang.Comparable** and **java.util.Comparator** in the section "Making Your Objects Comparable and Sortable" in Chapter 11, "The Collections Framework."

The Garbage Collector

In several examples so far, I have shown you how to create objects using the **new** keyword, but you have never seen code that explicitly destroys unused objects to release memory space. If you are a C++ programmer you may have wondered if I had shown flawed code, because in C++ you must destroy objects after use.

Java comes equipped with a feature called the garbage collector, which

destroys unused objects and frees memory. Unused objects are defined as objects that are no longer referenced or objects whose references are already out of scope.

With this feature, Java becomes much easier than C++ because Java programmers do not need to worry about reclaiming memory space. This, however, does not entail that you may create objects as many as you want because memory is (still) limited and it takes some time for the garbage collector to start. That's right, you can still run out of memory.

Summary

OOP models applications on real-world objects. Since Java is an OOP language, objects play a central role in Java programming. Objects are created based on a template called a class. In this chapter you've learned how to write a class and class members. There are many types of class members, including three that were discussed in this chapter: fields, methods, and constructors. There are other types of Java members such as enum and inner classes, which will be covered in other chapters.

In this chapter you have also learned two powerful OOP features, abstraction and encapsulation. Abstraction in OOP is the act of using programming objects to represent real-world objects. Encapsulation is a mechanism that protects parts of an object that need to be secure and exposes only parts that are safe to be exposed. Another feature discussed in this chapter is method overloading. Method overloading allows a class to have methods with the same name as long as their signatures are sufficiently different.

Java also comes with a garbage collector that eliminates to manually destroy unused objects. Objects are garbage collected when they are out of scope or no longer referenced.

Questions

1. Name at least three element types that a class can contain.
2. What are the differences between a method and a constructor?
3. Does a class in a class diagram display its constructors?
4. What does **null** mean?
5. What do you use the **this** keyword for?
6. When you use the == operator to compare two object references, do you actually compare the contents of the objects? Why?
7. What is the scope of variables?
8. What does "out of scope" mean?
9. How does the garbage collector decide which objects to destroy?
10. What is method overloading?

Chapter 5
Core Classes

Before discussing other features of object-oriented programming (OOP), let's examine several important classes that are commonly used in Java. These classes are included in the Java core libraries that come with the JDK. Mastering them will help you understand the examples that accompany the next OOP lessons.

The most prominent class of all is definitely **java.lang.Object**. However, it is hard to talk about this class without first covering inheritance, which we will do in Chapter 6, "Inheritance." Therefore, **java.lang.Object** is only discussed briefly in this chapter. Right now we will concentrate on classes that you can use in your programs. We'll start with **java.lang.String** and other types of strings: **java.lang.StringBuffer** and **java.lang.StringBuilder**. Then, we'll discuss arrays and the **java.lang.System** class. The **java.util.Scanner** class is also included here because it provides a convenient way to take user input.

Boxing/unboxing and varargs are explained towards the end of this chapter.

Note

When describing a method in a Java class, presenting the method signature always helps. A method often takes as parameters objects whose classes belong to different packages than the method's class. Or, it may return a type from a different package than its class. For clarity, fully qualified names will be used for classes from different packages. For example, here is the signature of the **toString** method of **java.lang.Object**:

```
public String toString()
```

A fully qualified name for the return type is not necessary because the return type **String** is part of the same package as **java.lang.Object**. On the other hand, the signature of the **toString** method in **java.util.Scanner** uses a fully qualified name because the **Scanner** class is part of a different package (**java.util**).

```
public java.lang.String toString()
```

---

java.lang.Object

The **java.lang.Object** class represents a Java object. In fact, all classes are direct or indirect descendants of this class. Since we have not learned inheritance (which is only given in Chapter 6, "Inheritance"), the word descendant probably makes no sense to you. Therefore, we will briefly discuss the methods in this class and revisit this class in Chapter 6.

Here are the methods in the **Object** class.

`protected Object clone()`

Creates and returns a copy of this object. A class implements this method to

support object cloning.

`public boolean equals(Object obj)`
: Compares this object with the passed-in object. A class must implement this method to provide a means to compare the contents of its instances.

`protected void finalize()`
: Called by the garbage collector on an object that is about to be garbage-collected. In theory a subclass can override this method to dispose of system resources or to perform other cleanup. However, performing the aforesaid operations should be done somewhere else and you should not touch this method.

`public final Class getClass()`
: Returns a **java.lang.Class** object of this object. See the section "java.lang.Class" for more information on the **Class** class.

`public int hashCode()`
: Returns a hash code value for this object.

`public String toString()`
: Returns the description of this object.

In addition, there are also the **wait**, **notify**, and **notifyAll** methods that you use when writing multithreaded applications. These methods are discussed in Chapter 23, "Java Threads."

---

java.lang.String

I have not seen a serious Java program that does not use the **java.lang.String** class. It is one of the most often used classes and definitely one of the most important.

A **String** object represents a string, i.e. a piece of text. You can also think of a **String** as a sequence of Unicode characters. A **String** object can consists of any number of characters. A **String** that has zero character is called an empty **String**. **String** objects are constant. Once they are created, their values cannot be changed. Because of this, **String** instances are said to be immutable. And, because they are immutable, they can be safely shared.

You could construct a **String** object using the **new** keyword, but this is not a common way to create a **String**. Most often, you assign a string literal to a **String** reference variable. Here is an example:

```
String s = "Java is cool";
```

This produces a **String** object containing "Java is cool" and assigns a reference to it to **s**. It is the same as the following.

```
String message = new String("Java is cool");
```

However, assigning a string literal to a reference variable works differently from using the **new** keyword. If you use the **new** keyword, the JVM will always create a new instance of **String**. With a string literal, you get an identical **String** object, but the object is not always new. It may come from a pool if the string "Java is cool" has been created before.

Thus, using a string literal is better because the JVM can save some CPU cycles spent on constructing a new instance. Because of this, we seldom use the **new** keyword when creating a **String** object. The **String** class's constructors can

be used if we have specific needs, such as converting a character array into a **String**.

Comparing Two Strings

String comparison is one of the most useful operations in Java programming. Consider the following code.

```
String s1 = "Java";
String s2 = "Java";
if (s1 == s2) {
    ...
}
```

Here, **(s1 == s2)** evaluates to **true** because **s1** and **s2** reference the same instance. On the other hand, in the following code **(s1 == s2)** evaluates to **false** because **s1** and **s2** reference different instances:

```
String s1 = new String("Java");
String s2 = new String("Java");
if (s1 == s2) {
    ...
}
```

This shows the difference between creating **String** objects by writing a string literal and by using the **new** keyword.

Comparing two **String** objects using the == operator is of little use because you are comparing the addresses referenced by two variables. Most of the time, when comparing two **String** objects, you want to know whether the values of the two objects are the same. In this case, you need to use the **String** class's **equals** method.

```
String s1 = "Java";
if (s1.equals("Java")) // returns true.
```

And, sometimes you see this style.

```
if ("Java".equals(s1))
```

In **(s1.equals("Java"))**, the **equals** method on **s1** is called. If **s1** is null, the expression will generate a runtime error. To be safe, you have to make sure that **s1** is not null, by first checking if the reference variable is null.

```
if (s1 != null && s1.equals("Java"))
```

If **s1** is null, the **if** statement will return **false** without evaluating the second expression because the AND operator **&&** will not try to evaluate the right hand operand if the left hand operand evaluates to **false**.

In **("Java".equals(s1))**, the JVM creates or takes from the pool a **String** object containing "Java" and calls its **equals** method. No nullity checking is required here because "Java" is obviously not null. If **s1** is null, the expression simply returns **false**. Therefore, these two lines of code have the same effect.

```
if (s1 != null && s1.equals("Java"))

if ("Java".equals(s1))
```

String Literals

Because you always work with **String** objects, it is important to understand the rules for working with string literals.

First of all, a string literal starts and ends with a double quote ("). Second, it is a compile error to change line before the closing ". For example, this will raise a compile error.

```
String s2 = "This is an important
        point to note";
```

You can compose long string literals by using the plus sign to concatenate two string literals.

```
String s1 = "Java strings " + "are important";
String s2 = "This is an important " +
        "point to note";
```

You can concatenate a String with a primitive or another object. For instance, this line of code concatenates a **String** and an integer.

```
String s3 = "String number " + 3;
```

If an object is concatenated with a String, the **toString** method of the former will be called and the result used in the concatenation.

Escaping Certain Characters

You sometimes need to use special characters in your strings such as carriage return (CR) and linefeed (LF). In other occasions, you may want to have a double quote character in your string. In the case of CR and LF, it is not possible to input these characters because pressing Enter changes lines. A way to include special characters is to escape them, i.e. use the character replacement for them.

Here are some escape sequences:

```
\u          /* a Unicode character
\b          /* \u0008: backspace BS */
\t          /* \u0009: horizontal tab HT */
\n          /* \u000a: linefeed LF */
\f          /* \u000c: form feed FF */
\r          /* \u000d: carriage return CR */
\"          /* \u0022: double quote " */
\'          /* \u0027: single quote ' */
\\          /* \u005c: backslash \ */
```

For example, the following code includes the Unicode character 0122 at the end of the string.

```
String s = "Please type this character \u0122";
```

To obtain a **String** object whose value is John "The Great" Monroe, you escape the double quotes:

```
String s = "John \"The Great\" Monroe";
```

Switching on A String

Starting from Java 7 you can use the **switch** statement with a String. Recall the syntax of the **switch** statement given in Chapter 3, “Statements:”

```
switch(expression) {
case value_1 :
    statement(s);
```

```
    break;
case value_2 :
    statement(s);
    break;
  .
  .
  .
case value_n :
    statement(s);
    break;
default:
    statement(s);
}
```

Here is an example of using the **switch** statement on a String.

```
String input = ...;
switch (input) {
case "one" :
    System.out.println("You entered 1.");
    break;
case "two" :
    System.out.println("You entered 2.");
    break;
default:
    System.out.println("Invalid value.");
}
```

The String Class's Constructors

The **String** class provides a number of constructors. These constructors allow you to create an empty string, a copy of another string, and a **String** from an array of chars or bytes. Use the constructors with caution as they always create a new instance of **String**.

Note

Arrays are discussed in the section "Arrays."

`public String()`
: Creates an empty string.

`public String(String original)`
: Creates a copy of the original string.

`public String(char[] value)`
: Creates a **String** object from an array of chars.

`public String(byte[] bytes)`
: Creates a **String** object by decoding the bytes in the array using the computer's default encoding.

`public String(byte[] bytes, String encoding)`
: Creates a **String** object by decoding the bytes in the array using the specified encoding.

The String Class's Methods

The **String** class provides methods for manipulating the value of a **String**. However, since **String** objects are immutable, the result of the manipulation is

always a new **String** object.

Here are some of the more useful methods.

`public char charAt(int index)`
Returns the char at the specified index. For example, the following code returns 'J'.

```
"Java is cool".charAt(0)
```

`public String concat(String s)`
Concatenates the specified string to the end of this **String** and return the result. For example, **"Java ".concat("is cool")** returns “Java is cool”.

`public boolean equals(String anotherString)`
Compares the value of this **String** and *anotherString* and returns **true** if the values match.

`public boolean endsWith(String suffix)`
Tests if this **String** ends with the specified suffix.

`public int indexOf(String substring)`
Returns the index of the first occurrence of the specified substring. If no match is found, returns -1. For instance, the following code returns 8.

```
"Java is cool".indexOf("cool")
```

`public int indexOf(String substring, int fromIndex)`
Returns the index of the first occurrence of the specified substring starting from the specified index. If no match is found, returns -1.

`public int lastIndexOf(String substring)`
Returns the index of the last occurrence of the specified substring. If no match is found, returns -1.

`public int lastIndexOf(String substring, int fromIndex)`
Returns the index of the last occurrence of the specified substring starting from the specified index. If no match is found, returns -1. For example, the following expression returns 3.

```
"Java is cool".lastIndexOf("a")
```

`public String substring(int beginIndex)`
Returns a substring of the current string starting from the specified index. For instance, **"Java is cool".substring(8)** returns “cool”.

`public String substring(int beginIndex, int endIndex)`
Returns a substring of the current string starting from *beginIndex* to *endIndex*. For example, the following code returns “is”:

```
"Java is cool".substring(5, 7)
```

`public String replace(char oldChar, char newChar)`
Replaces every occurrence of *oldChar* with *newChar* in the current string and returns the new **String**. **"dingdong".replace('d', 'k')** returns “kingkong”.

`public int length()`
Returns the number of characters in this **String**. For example, **"Java is cool".length()** returns 12. Prior to Java 6, this method was often used to test if a **String** was empty. However, the **isEmpty** method is preferred because it's more descriptive.

`public boolean isEmpty()`

Returns true is the string is empty (contains no characters).

`public String[] split(String regEx)`
Splits this **String** around matches of the specified regular expression. For example, **"Java is cool".split(" ")** returns an array of three **String**s. The first array element is "Java", the second "is", and the third "cool".

`public boolean startsWith(String prefix)`
Tests if the current string starts with the specified prefix.

`public char[] toCharArray()`
Converts this string to an array of chars.

`public String toLowerCase()`
Converts all the characters in the current string to lower case. For instance, **"Java is cool".toLowerCase()** returns "java is cool".

`public String toUpperCase()`
Converts all the characters in the current string to upper case. For instance, **"Java is cool".toUpperCase()** returns "JAVA IS COOL".

`public String trim()`
Trims the trailing and leading white spaces and returns a new string. For example, **" Java ".trim()** returns "Java".

In addition, there are two static methods, **valueOf** and **format**. The **valueOf** method converts a primitive, a char array, or an instance of **Object** into a string representation and there are nine overloads of this method.

```
public static String valueOf(boolean value)
public static String valueOf(char value)
public static String valueOf(char[] value)
public static String valueOf(char[] value, int offset, int length)
public static String valueOf(double value)
public static String valueOf(float value)
public static String valueOf(int value)
public static String valueOf(long value)
public static String valueOf(Object value)
```

For example, the following code returns the string "23"

```
String.valueOf(23);
```

The **format** method allows you to pass an arbitrary number of parameters. Here is its signature.

```
public static String format(String format, Object... args)
```

However, I will hold off discussing the **format** method here because to understand it, you need to understand arrays and variable arguments. It will be explained in the section "Varargs" later in this chapter.

---

java.lang.StringBuffer and java.lang.StringBuilder

String objects are immutable and are not suitable to use if you need to append or insert characters into them because string operations on **String** always create a new **String** object. For append and insert, you'd be better off using the **java.lang.StringBuffer** or **java.lang.StringBuilder** class. Once you're finished manipulating the string, you can convert a **StringBuffer** or **StringBuilder** object

to a **String**.

Until JDK 1.4, the **StringBuffer** class was solely used for mutable strings. Methods in **StringBuffer** are synchronized, making **StringBuffer** suitable for use in multithreaded environments. However, the price for synchronization is performance. JDK 5 added the **StringBuilder** class, which is the unsynchronized version of **StringBuffer**. **StringBuilder** should be chosen over **StringBuffer** if you do not need synchronization.

> **Note**
> Synchronization and thread safety are discussed in Chapter 23, "Java Threads."

The rest of this section will use **StringBuilder**. However, the discussion is also applicable to **StringBuffer** as both **StringBuilder** and **StringBuffer** shares similar constructors and methods.

StringBuilder Class's Constructors

The **StringBuilder** class has four constructors. You can pass a **java.lang.CharSequence**, a **String**, or an **int**.

```
public StringBuilder()
public StringBuilder(CharSequence seq)
public StringBuilder(int capacity)
public StringBuilder(String string)
```

If you create a **StringBuilder** object without specifying the capacity, the object will have a capacity of 16 characters. If its content exceeds 16 characters, it will grow automatically. If you know that your string will be longer than 16 characters, it is a good idea to allocate enough capacity as it takes time to increase a **StringBuilder**'s capacity.

StringBuilder Class's Methods

The **StringBuilder** class has several methods. The main ones are **capacity**, **length**, **append**, and **insert**.

`public int capacity()`
: Returns the capacity of the **StringBuilder** object.

`public int length()`
: Returns the length of the string the **StringBuilder** object stores. The value is less than or equal to the capacity of the **StringBuilder**.

`public StringBuilder append(String string)`
: Appends the specified **String** to the end of the contained string. In addition, **append** has various overloads that allow you to pass a primitive, a char array, and an **java.lang.Object** instance.
For example, examine the following code.

```
StringBuilder sb = new StringBuilder(100);
sb.append("Matrix ");
sb.append(2);
```

After the last line, the content of **sb** is "Matrix 2".
An important point to note is that the **append** methods return the **StringBuilder** object itself, the same object on which **append** is invoked. As a result, you can chain calls to **append**.

```
sb.append("Matrix ").append(2);
```

`public StringBuilder insert(int offset, String string)`

Inserts the specified string at the position indicated by *offset*. In addition, **insert** has various overloads that allow you to pass primitives and a **java.lang.Object** instance. For example,

```
StringBuilder sb2 = new StringBuilder(100);
sb2.append("night");
sb2.insert(0, 'k'); // value = "knight"
```

Like **append**, **insert** also returns the current **StringBuilder** object, so chaining **insert** is also permitted.

`public String toString()`

Returns a **String** object representing the value of the **StringBuilder**.

Primitive Wrappers

For the sake of performance, not everything in Java is an object. There are also primitives, such as **int**, **long**, **float**, **double**, etc. When working with both primitives and objects, there are often circumstances that necessitate primitive to object conversions and vice versa. For example, a **java.util.Collection** object (discussed in Chapter 11, "The Collections Framework") can be used to store objects, not primitives. If you want to store primitive values in a **Collection**, they must be converted to objects first.

The **java.lang** package has several classes that function as primitive wrappers. They are **Boolean**, **Character**, **Byte**, **Double**, **Float**, **Integer**, **Long**, and **Short**. **Byte**, **Double**, **Float**, **Integer**, **Long**, and **Short** share similar methods, therefore only **Integer** will be discussed here. You should consult the Javadoc for information on the others.

Note

Conversions from primitives to objects and vice versa are easy thanks to the boxing and unboxing mechanisms. Boxing and unboxing are discussed in the section "Boxing and Unboxing" later in this chapter.

The following sections discuss the wrapper classes in detail.

java.lang.Integer

The **java.lang.Integer** class wraps an **int**. The **Integer** class has two static final fields of type **int**: **MIN_VALUE** and **MAX_VALUE**. **MIN_VALUE** contains the minimum possible value for an **int** (-2^{31}) and **MAX_VALUE** the maximum possible value for an **int** ($2^{31} - 1$).

The **Integer** class has two constructors:

```
public Integer(int value)
public Integer(String value)
```

For example, this code constructs two **Integer** objects.

```
Integer i1 = new Integer(12);
Integer i2 = new Integer("123");
```

Integer has the no-arg **byteValue**, **doubleValue**, **floatValue**, **intValue**, **longValue**, and **shortValue** methods that convert the wrapped value to a **byte**, **double**, **float**, **int**, **long**, and **short**, respectively. In addition, the **toString** method converts the value to a **String**.

There are also static methods that you can use to parse a **String** to an **int** (**parseInt**) and convert an **int** to a **String** (**toString**). The signatures of the methods are as follows.

```
public static int parseInt(String string)
public static String toString(int i)
```

java.lang.Boolean

The **java.lang.Boolean** class wraps a **boolean**. Its static final fields **FALSE** and **TRUE** represents a **Boolean** object that wraps the primitive value **false** and a **Boolean** object wrapping the primitive value **true**, respectively.

You can construct a **Boolean** object from a **boolean** or a **String**, using one of these constructors.

```
public Boolean(boolean value)
public Boolean(String value)
```

For example:

```
Boolean b1 = new Boolean(false);
Boolean b2 = new Boolean("true");
```

To convert a **Boolean** to a **boolean**, use its **booleanValue** method:

```
public boolean booleanValue()
```

In addition, the static method **valueOf** parses a **String** to a **Boolean** object.

```
public static Boolean valueOf(String string)
```

And, the static method **toString** returns the string representation of a **boolean**.

```
public static String toString(boolean boolean)
```

java.lang.Character

The **Character** class wraps a **char**. There is only one constructor in this class:

```
public Character(char value)
```

To convert a **Character** object to a **char**, use its **charValue** method.

```
public char charValue()
```

There are also a number of static methods that can be used to manipulate characters.

```
public static boolean isDigit(char ch)
```

Determines if the specified argument is one of these: ‘1’, ‘2’, ‘3’, ‘4’, ‘5’, ‘6’, ‘7’, ‘8’, ‘9’, ‘0’.

```
public static char toLowerCase(char ch)
```

Converts the specified char argument to its lower case.

```
public static char toUpperCase(char ch)
```

Converts the specified char argument to its upper case.

Arrays

In Java you can use arrays to group primitives or objects of the same type. The entities belonging to an array is called the components of the array. In the

background, every time you create an array, the compiler creates an object which allows you to:

- get the number of components in the array through the **length** field. The length of an array is the number of components in it.
- access each component by specifying an index. This indexing is zero-based. Index 0 refers to the first component, 1 to the second component, etc.

All the components of an array have the same type, called the *component type* of the array. An array is not resizable and an array with zero component is called an empty array.

An array is a Java object. Therefore, you treat a variable that refers to an array like other reference variables. For one, you can compare it with **null**.

```
String[] names;
if (names == null)  // evaluates to true
```

If an array is a Java object, shouldn't there be a class that gets instantiated when you create an array? May be something like **java.lang.Array**? The truth is, no. Arrays are indeed special Java objects whose class is never documented and is not meant to be extended.

Note

An array can also contain other arrays, creating an array of arrays.

To use an array, first you need to declare one. You can use this syntax to declare an array:

```
type[] arrayName;
```

or

```
type arrayName[]
```

For example, the following declares an array of **long**s named **numbers**:

```
long[] numbers;
```

Declaring an array does not create an array or allocate space for its components, the compiler simply creates an object reference. One way to create an array is by using the **new** keyword. You must specify the size of the array you are creating.

```
new type[size]
```

As an example, the following code creates an array of four **int**s:

```
new int[4]
```

Alternatively, you can declare and create an array in the same line.

```
int[] ints = new int[4];
```

To reference the components of an array, use an index after the variable name. For example, the following snippet creates an array of four **String** objects and initializes its first member.

```
String[] names = new String[4];
names[0] = "Hello World";
```

Note

Referencing an out of bound member will raise a runtime error. To be precise, a **java.lang.ArrayIndexOutOfBoundsException** will be thrown. See Chapter 7, "Error Handling" for information about exceptions.

Note

When an array is created, its components are either **null** (if the component type is an object type) or the default value of the component type (if the array contains primitives). For example, an array of **int**s contain zeros by default.

You can also create and initialize an array without using the **new** keyword. Java allows you to create an array by grouping values within a pair of braces. For example, the following code creates an array of three **String** objects.

```
String[] names = { "John", "Mary", "Paul" };
```

The following code creates an array of four **int**s and assign the array to the variable **matrix**.

```
int[] matrix = { 1, 2, 3, 10 };
```

Be careful when passing an array to a method because the following is illegal even though the average method below take an array of **int**s.

```
int avg = average( { 1, 2, 3, 10 } ); // illegal
```

Instead, you have to instantiate the array separately.

```
int[] numbers = { 1, 2, 3, 10 };
int avg = average(numbers);
```

or this

```
int avg = average(new int[] { 1, 2, 3, 10 });
```

Iterating over an Array

Prior to Java 5, the only way to iterate the members of an array was to use a **for** loop and the array's indices. For example, the following code iterates over a **String** array referenced by the variable **names**:

```
for (int i = 0; i < 3; i++) {
    System.out.println("\t- " + names[i]);
}
```

Java 5 has enhanced the **for** statement. You can now use it to iterate over an array or a collection without the index. Use this syntax to iterate over an array:

```
for (componentType variable : arrayName)
```

Where *arrayName* is the reference to the array, *componentType* is the component type of the array, and *variable* is a variable that references each component of the array.

For example, the following code iterates over an array of **String**s.

```
String[] names = { "John", "Mary", "Paul" };
for (String name : names) {
    System.out.println(name);
}
```

The code prints this on the console.

```
John
Mary
Paul
```

Changing an Array Size

Once an array is created, its size cannot be changed. If you want to change the size, you must create a new array and populates it using the values of the old array. For instance, the following code increases the size of **numbers**, an array of three **int**s, to 4.

```
int[] numbers = { 1, 2, 3 };
int[] temp = new int[4];
int length = numbers.length;
for (int j = 0; j < length; j++) {
    temp[j] = numbers[j];
}
numbers = temp;
```

A shorter way of doing this is by using the **copyOf** method of the **java.util.Arrays** class. For instance, this code creates a four-element array and copies the content of **numbers** to its first three elements.

```
int[] numbers = { 1, 2, 3 };
int[] newArray = Arrays.copyOf(numbers, 4);
```

And, of course you can reassign the new array to the original variable:

```
numbers = Arrays.copyOf(numbers, 4);
```

The **copyOf** method comes with ten overloads, eight for each type of Java primitives and two for objects. Here are their signatures:

```
public static boolean[] copyOf(boolean[] original, int newLength)
public static byte[] copyOf(byte[] original, int newLength)
public static char[] copyOf(char[] original, int newLength)
public static double[] copyOf(double[] original, int newLength)
public static float[] copyOf(float[] original, int newLength)
public static int[] copyOf(int[] original, int newLength)
public static long[] copyOf(long[] original, int newLength)
public static short[] copyOf(short[] original, int newLength)
public static <T> T[] copyOf(T[] original, int newLength)
public static <T,U> T[] copyOf(U[] original, int newLength,
        java.lang.Class<? extends T[]> newType)
```

Each of these overloads can throw a **java.lang.NullPointerException** if *original* is null and a **java.lang.NegativeArraySizeException** if *newLength* is negative.

The *newLength* argument can be smaller, equal to, or larger than the length of the original array. If it is smaller, then only the first *newLength* elements will be included in the copy. If it is larger, the last few elements will have default values, i.e. 0 if it is an array of integers or **null** if it is an array of objects.

Another method similar to **copyOf** is **copyOfRange**. **copyOfRange** copies a range of elements to a new array. Like **copyOf**, **copyOfRange** also provides overrides for each Java data type. Here are their signatures:

```
public static boolean[] copyOfRange(boolean[] original,
        int from, int to)
```

```
public static byte[] copyOfRange(byte[] original,
        int from, int to)
public static char[] copyOfRange(char[] original,
        int from, int to)
public static double[] copyOfRange(double[] original,
        int from, int to)
public static float[] copyOfRange(float[] original,
        int from, int to)
public static int[] copyOfRange(int[] original, int from, int to)
public static long[] copyOfRange(long[] original, int from, int to)
public static short[] copyOfRange(short[] original, int from,
        int to)
public static <T> T[] copyOfRange(T[] original, int from, int to)
public static <T,U> T[] copyOfRange(U[] original, int from,
        int to, java.lang.Class<? extends T[]> newType)
```

Passing a String Array to main

You invoke the static void method **main** to run a Java class. Here is the signature of the **main** method:

```
public static void main(String[] args)
```

You can pass arguments to the **main** method by feeding them to the **java** program. The arguments should appear after the class name and two arguments are separated by a space. To be precise, you should use the following syntax:

```
java className arg1 arg2 arg3 ... arg-n
```

Listing 5.1 shows a class that iterates over the **main** method's **String** array argument.

Listing 5.1: Accessing the main method's arguments

```
package app05;
public class MainMethodTest {
    public static void main(String[] args) {
        for (String arg : args) {
            System.out.println(arg);
        }
    }
}
```

The following command invokes the class and passes two arguments to the **main** method.

```
java app05/MainMethodTest john mary
```

The **main** method will then print the arguments to the console.

```
john
mary
```

java.lang.Class

One of the members of the **java.lang** package is a class named **Class**. Every time the JVM creates an object, it also creates a **java.lang.Class** object that describes the type of the object. All instances of the same class share the same **Class** object. You can obtain the **Class** object by calling the **getClass** method of the object. This method is inherited from **java.lang.Object**.

For example, the following code creates a **String** object, invokes the **getClass** method on the **String** instance, and then invokes the **getName** method on the **Class** object.

```
String country = "Fiji";
Class myClass = country.getClass();
System.out.println(myClass.getName()); // prints java.lang.String
```

As it turns out, the **getName** method returns the fully qualified name of the class represented by a **Class** object.

The **Class** class also brings the possibility of creating an object without using the **new** keyword. You achieve this by using the two methods of the **Class** class, **forName** and **newInstance**.

```
public static Class forName(String className)
public Object newInstance()
```

The static **forName** method creates a **Class** object of the given class name. The **newInstance** method creates a new instance of a class.

The following example uses **forName** to create a **Class** object of the **app05.Test** class and create an instance of the **Test** class. Since **newInstance** returns a **java.lang.Object** object, you need to downcast it to its original type.

```
Class klass = null;
try {
    klass = Class.forName("app05.Test");
} catch (ClassNotFoundException e) {
}

if (klass != null) {
try {
    // create an instance of the Test class
    Test test = (Test) klass.newInstance();

} catch (IllegalAccessException e) {
} catch (InstantiationException e) {
}
```

Do not worry about the **try … catch** blocks as they will be explained in Chapter 7, "Error Handling."

You might want to ask this question, though. Why would you want to create an instance of a class using **forName** and **newInstance**, when using the **new** keyword is shorter and easier? The answer is because there are circumstances whereby the name of the class is not known when you are writing the program.

java.lang.System

The **System** class is a final class that exposes useful static fields and static methods that can help you with common tasks.

The three fields of the **System** class are **out**, **in**, and **err**:

```
public static final java.io.PrintStream out;
public static final java.io.InputStream in;
public static final java.io.PrintStream err;
```

The **out** field represents the standard output stream which by default is the same console used to run the running Java application. You will learn more about **PrintStream** in Chapter 13, "Input Output," but for now know that you can use the **out** field to write messages to the console. You will often write the following line of code:

```
System.out.print(message);
```

where *message* is a **String** object. However, **PrintStream** has many **print** method overloads that accept different types, so you can pass any primitive type to the **print** method:

```
System.out.print(12);
System.out.print('g');
```

In addition, there are **println** methods that are equivalent to **print**, except that **println** adds a line terminator at the end of the argument.

Note also that because **out** is static, you can access it by using this notation: **System.out**, which returns a **java.io.PrintStream** object. You can then access the many methods on the **PrintStream** object as you would methods of other objects: **System.out.print**, **System.out.format**, etc.

The **err** field also represents a **PrintStream** object, and by default the output is channeled to the console from where the current Java program was invoked. However, its purpose is to display error messages that should get immediate attention of the user.

For example, here is how you can use **err**:

```
System.err.println("You have a runtime error.");
```

The **in** field represents the standard input stream. You can use it to accept keyboard input. For example, the **getUserInput** method in Listing 5.2 accepts the user input and returns it as a String:

Listing 5.2: The getUserInput method

```
public String getUserInput() {
    StringBuilder sb = new StringBuilder();
    try {
        char c = (char) System.in.read();
        while (c != '\r') {
            sb.append(c);
            c = (char) System.in.read();
        }
    } catch (IOException e) {
    }
    return sb.toString();
}
```

However, an easier way to receive keyboard input is to use the **java.util.Scanner** class, discussed in the section "java.util.Scanner" later in this chapter.

The **System** class has many useful methods, all of which are static. Some of the more important ones are listed here.

`public static void arraycopy(Object source, int sourcePos, Object destination, int destPos, int length)`

This method copies the content of an array (*source*) to another array (*destination*), beginning at the specified position, to the specified position of the destination array. For example, the following code uses **arraycopy** to copy the contents of **array1** to **array2**.

```
int[] array1 = {1, 2, 3, 4};
int[] array2 = new int[array1.length];
System.arraycopy(array1, 0, array2, 0, array1.length);
```

`public static void exit(int status)`

Terminates the running program and the current JVM. You normally pass 0 to indicate that a normal exit and a nonzero to indicate there has been an error in the program prior to calling this method.

`public static long currentTimeMillis()`

Returns the computer time in milliseconds. The value represents the number of milliseconds that has elapsed since January 1, 1970 UTC. To get the string representation of the current computer time, use this:

```
System.out.println(new java.util.Date());
```

The **Date** class is discussed in Chapter 8, Numbers and Date."
However, **currentTimeMillis** is useful if you want to time an operation. For instance, the following code measure the time it takes to perform a block of code:

```
long start = System.currentTimeMillis();
// block of code to time
long end = System.currentTimeMillis();
System.out.println("It took " + (end - start) +
        " milliseconds");
```

`public static long nanoTime()`

This method is similar to **currentTimeMillis**, but provide more precision, i.e. in nanoseconds.

```
long start = System.nanoTime();
// block of code to time
long end = System.nanoTime();
System.out.println("It took " + (end - start) +
        " nanoseconds");
```

`public static String getProperty(String key)`

This method returns the value of the specified property. It returns **null** if the specified property does not exist. There are system properties and there are user-defined properties. When a Java program runs, the JVM provides values that may be used by the program as properties.
Each property comes as a key/value pair. For example, the **os.name** system property provides the name of the operating system running the JVM. Also, the directory name from which the application was invoked is provided by the JVM as a property named **user.dir**. To get the value of the **user.dir** property, you use:

```
System.getProperty("user.dir");
```

Table 5.1 lists the system properties.

System property	Description
java.version	Java Runtime Environment version
java.vendor	Java Runtime Environment vendor
java.vendor.url	Java vendor URL
java.home	Java installation directory
java.vm.specification.version	Java Virtual Machine specification version
java.vm.specification.vendor	Java Virtual Machine specification vendor
java.vm.specification.name	Java Virtual Machine specification name
java.vm.version	Java Virtual Machine implementation version
java.vm.vendor	Java Virtual Machine implementation vendor
java.vm.name	Java Virtual Machine implementation name
java.specification.version	Java Runtime Environment specification version
java.specification.vendor	Java Runtime Environment specification vendor
java.specification.name	Java Runtime Environment specification name
java.class.version	Java class format version number
java.class.path	Java class path
java.library.path	List of paths to search when loading libraries
java.io.tmpdir	Default temp file path
java.compiler	Name of JIT compiler to use
java.ext.dirs	Path of extension directory or directories
os.name	Operating system name
os.arch	Operating system architecture
os.version	Operating system version
file.separator	File separator ("/" on UNIX)
path.separator	Path separator (":" on UNIX)
line.separator	Line separator ("\n" on UNIX)
user.name	User's account name
user.home	User's home directory
user.dir	User's current working directory

Table 5.1: Java system properties

`public static void setProperty(String property, String newValue)`

You use **setProperty** to create a user-defined property or change the value of the current property. For instance, you can use this code to create a property named **password**:

```
System.setProperty("password", "tarzan");
```

And, you can retrieve it by using **getProperty**:

```
System.getProperty("password")
```

For instance, here is how you change the **user.name** property.

```
System.setProperty("user.name", "tarzan");
```

`public static String getProperty(String key, String default)`

This method is similar to the single argument **getProperty** method, but returns a default value if the specified property does not exist.

`public static java.util.Properties getProperties()`

This method returns all system properties. The return value is a **java.util.Properties** object. The **Properties** class is a subclass of **java.util.Hashtable** (discussed in Chapter 11, "The Collections

Framework").
For example, the following code uses the **list** method of the **Properties** class to iterate and display all system properties on the console.

```
java.util.Properties properties = System.getProperties();
properties.list(System.out);
```

---

java.util.Scanner

You use a **Scanner** object to scan a piece of text. In this chapter, we will only concentrate on its use to receive keyboard input.

Receiving keyboard input with **Scanner** is easy. All you need to do is instantiate the **Scanner** class by passing **System.in**. Then, to receive user input, call the **next** method on the instance. The **next** method buffers the characters the user input from the keyboard or other devices until the user presses Enter. It then returns a **String** containing the characters the user entered excluding the carriage-return character sequence. Listing 5.3 demonstrates the use of **Scanner** to receive user input.

Listing 5.3: Using Scanner to receive user input

```
Scanner scanner = new Scanner(System.in);
String s = scanner.next();
```

Compared to the code in Listing 5.2, using **Scanner** is much simpler.

Boxing and Unboxing

Conversion from primitive types to corresponding wrapper objects and vice versa can happen automatically. Boxing refers to the conversion of a primitive to a corresponding wrapper instance, such as an **int** to a **java.lang.Integer**. Unboxing is the conversion of a wrapper instance to a primitive type, such as **Byte** to **byte**.

Here is an example of boxing.

```
Integer number = 3; // assign an int to Integer
```

Here is an example of unboxing.

```
Integer number = new Integer(100);
int[] ints = new int[2];
ints[0] = number;
```

ints is an **int** array and in the last line you assign an **Integer** object to its first member. The compiler unboxes **number** before assigning it to the first member of **ints**.

Varargs

Varargs is a Java feature that allows methods to have a variable length of argument list.

In Chapter 4, you learned that every method has a signature. For example, here is the signature of the **getInteger** method of the **java.lang.Integer** class.

```
public static Integer getInteger(String nm, Integer val)
```

There are two parameters in the argument list, indicating that you must pass two arguments when invoking the method.

Without varargs, if you want the flexibility of passing a variable number of arguments to a method, you had to wrap the arguments in an array. The **main** method is a good example of this technique, and this was discussed in the section "Arrays" earlier in this chapter. The signature of the **main** method is reprinted here.

```
public static void main(String[] args)
```

If you understand arrays well, varargs is also easy to comprehend. Basically, instead of an array, you use a type with ellipses in the signature. Therefore, using the varargs version, the **main** method can be written this way.

```
public static void main(String... args)
```

The ellipsis says that there is zero or more arguments of this type.

If an argument list contains both fixed arguments (arguments that must exist) and variable arguments, the variable arguments must come last.

In the background, the variable arguments are still passed in as an array, therefore the type of each variable argument must be the same. Also, since varargs are passed as an array, inside your method you will get an array. The code in Listing 5.4 is a rewrite of the code in Listing 5.3.

Listing 5.4: Using varargs

```
package app05;
public class MainMethodTest {
    public static void main(String... args) {
        for (String arg : args) {
            System.out.println(arg);
        }
    }
}
```

Note that if the user passes no argument, you get an array with a size of zero. There is no need to check if the array is null. Therefore, the following two methods are different.

```
public void doIt1(String... args)

public void doIt2(String[] args)
```

doIt1 allows you to pass no argument or any number of String arguments whereas **doIt2** demands an array of Strings or null.

For users of your class, using varargs is easier because they can simply list the arguments, there is no need to create an array and populate it with the arguments.

Read also the section "The format and printf Methods" that illustrates some practical uses of varargs.

The format and printf Methods

A popular example of varargs can be found in the **format** method of the **java.lang.String** and **java.io.PrintStream** classes. Here is its signature.

```
public static String format(String formatString, Object... args)
```

This method returns a **String** formatted using the specified format string and arguments. The format pattern must follow the rules specified in the **java.util.Formatter** class and you can read them in the JavaDoc for the **Formatter** class. A brief description of these rules are as follows.

To specify an argument, use the notation **%s**, which denotes the next argument in the array. For example, the following is a method call to the **printf** method.

```
String firstName = "John";
String lastName = "Adams";
System.out.format("First name: %s. Last name: %s",
        firstName, lastName);
```

This prints the following string to the console:

```
First name: John. Last name: Adams
```

Without varargs, you have to do it in a more cumbersome way.

```
String firstName = "John";
String lastName = "Adams";
System.out.println("First name: " + firstName +
        ". Last name: " + lastName);
```

Note

The **printf** method in **java.io.PrintStream** is an alias for **format**.

The formatting example described here is only the tip of the iceberg. The formatting feature is much more powerful than that and you are encouraged to explore it by reading the Javadoc for the **Formatter** class.

Summary

In this chapter you have examined several important classes such as **java.lang.String**, arrays, **java.lang.System**, and **java.util.Scanner**. You have also learned two concepts: boxing/unboxing and variable arguments. The last section covered the implementation of varargs in **java.lang.String** and **java.io.PrintStream**.

Questions

1. What does it mean when we say that **Strings** are immutable objects?
2. How did you receive user input without **Scanner**? And, how would you do it with **Scanner**?
3. Are wrapper classes still useful boxing and unboxing happens automatically in Java?
4. How do you resize an array?
5. What is varargs?

Chapter 6
Inheritance

Inheritance is a very important feature of object-oriented programming (OOP). It is what makes code extensible in any OOP language. Extending a class is also called inheriting or subclassing. In Java, by default all classes are extendible, but you can use the **final** keyword to prevent classes from being subclassed. This chapter explains inheritance in Java.

An Overview of Inheritance

You extend a class by creating a new class. The former and the latter will then have a parent-child relationship. The original class is the parent class or the base class or the superclass. The new class is the child class or the subclass or the derived class of the parent. The process of extending a class in OOP is called inheritance. In a subclass you can add new methods and new fields as well as override existing methods to change their behaviors.

Figure 6.1 presents a UML class diagram that depicts a parent-child relationship between a class and a child class.

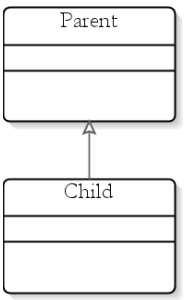

Figure 6.1: A UML class diagram for a parent class and a child class

Note that a line with an arrow is used to depict generalization, e.g. the parent-child relationship.

A child class in turn can be extended, unless you specifically make it inextensible by declaring it final. Final classes are discussed in the section “Final Classes” later in this chapter.

The benefits of inheritance are obvious. Inheritance gives you the opportunity to add some functionality that does not exist in the original class. It also gives you the chance to change the behaviors of the existing class to better suit your needs.

The extends Keyword

You extend a class by using the **extends** keyword in a class declaration, after the

class name and before the parent class. Listing 6.1 presents a class named **Parent** and Listing 6.2 a class named **Child** that extends **Parent**.

Listing 6.1: The Parent class

```
public class Parent {
}
```

Listing 6.2: The Child class

```
public class Child extends Parent {
}
```

Extending a class is as simple as that.

> **Note**
> All Java classes automatically extend the **java.lang.Object** class. **Object** is the ultimate superclass in Java. **Parent** in Listing 6.1 by default is a subclass of **Object**.

> **Note**
> In Java a class can only extend one class. This is unlike C++ where multiple inheritance is allowed. However, the notion of multiple inheritance can be achieved by using interfaces in Java, as discussed in Chapter 9, "Interfaces and Abstract Classes."

The is-a Relationship

There is a special relationship that is formed when you create a new class by inheritance. The subclass and the superclass has an "is-a" relationship.

For example, **Animal** is a class that represents animals. There are many types of animals, including birds, fish, and dogs, so you can create subclasses of **Animal** that represent specific types of animals. Figure 6.2 features the **Animal** class with three subclasses, **Bird**, **Fish**, and **Dog**.

The is-a relationship between the subclasses and the superclass **Animal** is very apparent. A bird "is an" animal, a dog is an animal, and a fish is an animal. A subclass is a special type of its superclass. For example, a bird is a special type of animal. The is-a relationship does not go the other way, however. An animal is not necessarily a bird or a dog.

Listing 6.3 presents the **Animal** class and its subclasses.

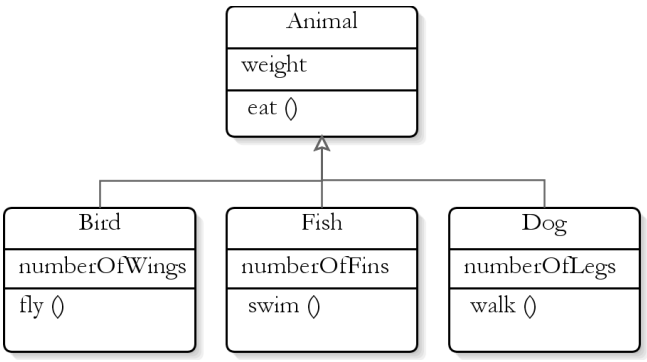

Figure 6.2: An example of inheritance

Listing 6.3: Animal and its subclasses

```
package app06;
class Animal {
    public float weight;
    public void eat() {
    }
}

class Bird extends Animal {
    public int numberOfWings = 2;
    public void fly() {
    }
}

class Fish extends Animal {
    public int numberOfFins = 2;
    public void swim() {
    }
}

class Dog extends Animal {
    public int numberOfLegs = 4;
    public void walk() {
    }
}
```

In this example, the **Animal** class defines a **weight** field that applies to all animals. It also declares an **eat** method because animals eat.

The **Bird** class is a special type of **Animal**, it inherits the **eat** method and the **weight** field. **Bird** also adds a **numberOfWings** field and a **fly** method. This shows that the more specific **Bird** class extends the functionality and behavior of the more generic **Animal** class.

A subclass inherits all public methods and fields of its superclass. For example, you can create a **Dog** object and call its **eat** method:

```
Dog dog = new Dog();
dog.eat();
```

The **eat** method is declared in the **Animal** class; the **Dog** class simply inherits it.

A consequence of the is-a relationship is that it is legal to assign an instance of a subclass to a reference variable of the parent type. For example, the following code is valid because **Bird** is a subclass of **Animal**, and a **Bird** is always an **Animal**.

```
Animal animal = new Bird();
```

However, the following is illegal because there is no guarantee that an **Animal** is a **Dog**.:

```
Dog dog = new Animal();
```

Accessibility

From within a subclass you can access its superclass's public and protected methods and fields, but not the superclass's private methods. If the subclass and

the superclass are in the same package, you can also access the superclass's default methods and fields.

Consider the **P** and **C** classes in Listing 6.4.

Listing 6.4: Showing accessibility

```
package app06;
public class P {
    public void publicMethod() {
    }
    protected void protectedMethod() {
    }
    void defaultMethod() {
    }
}
class C extends P {
    public void testMethods() {
        publicMethod();
        protectedMethod();
        defaultMethod();
    }
}
```

P has three methods, one public, one protected, and one with the default access level. **C** is a subclass of **P**. As you can see in the **C** class's **testMethods** method, **C** can access its parent's public and protected method. In addition, because **C** and **P** are in the same package, **C** can also access **P**'s default method.

However, it does not mean you can expose **P**'s non-public methods through its subclass. For example, the following code will not compile:

```
package test;
import app06.C;
public class AccessibilityTest {
    public static void main(String[] args) {
        C c = new C();
        c.protectedMethod();
    }
}
```

protectedMethod is **P**'s protected method. It is not accessible from outside **P**, except from a subclass. Since **AccessibilityTest** is not a subclass of **P**, you cannot access **P**'s protected method through its subclass **C**.

Method Overriding

When you extends a class, you can change the behavior of a method in the parent class. This is called method overriding, and this happens when you write in a subclass a method that has the same signature as a method in the parent class. If only the name is the same but the list of arguments is not, then it is method overloading. (See Chapter 4, "Objects and Classes")

You override a method to change its behavior. To override a method, you simply have to write the new method in the subclass, without having to change anything in the parent class. You can override the superclass's public and protected methods. If the subclass and superclass are in the same package, you can also

override a method with the default access level.

An example of method overriding is demonstrated by the **Box** class in Listing 6.5.

Listing 6.5: The Box class

```
package app06;
public class Box {
    public int length;
    public int width;
    public int height;

    public Box(int length, int width, int height) {
        this.length = length;
        this.width = width;
        this.height = height;
    }

    public String toString() {
        return "I am a Box.";
    }

    public Object clone() {
        return new Box(1, 1, 1);
    }
}
```

The **Box** class extends the **java.lang.Object** class. It is an implicit extension since the **extends** keyword is not used. **Box** overrides the public **toString** method and the protected **clone** method. Note that the **clone** method in **Box** is public whereas in **Object** it is protected. Increasing the visibility of a method defined in a superclass from protected to public is allowed. However, reducing visibility is illegal.

What if you create a method that has the same signature as a private method in the superclass? It is not method overriding, since private methods are not visible from outside the class.

Note

You cannot override a final method. To make a method final, use the **final** keyword in the method declaration. For example:

```
public final java.lang.String toUpperCase(java.lang.String s)
```

Calling the Superclass's Constructors

A subclass is just like an ordinary class, you use the **new** keyword to create an instance of it. If you do not explicitly write a constructor in your subclass, the compiler will implicitly add a no-argument (no-arg) constructor.

When you instantiate a child class by invoking one of its constructors, the first thing the constructor does is call the no-argument constructor of the direct parent class. In the parent class, the constructor also calls the constructor of its direct parent class. This process repeats itself until the constructor of the **java.lang.Object** class is reached. In other words, when you create a child object, all its parent classes are also instantiated.

This process is illustrated in the **Base** and **Sub** classes in Listing 6.6.

Listing 6.6: Calling a superclass's no-arg constructor

```
package app06;

class Base {
    public Base() {
        System.out.println("Base");
    }
    public Base(String s) {
        System.out.println("Base." + s);
    }
}
public class Sub extends Base {
    public Sub(String s) {
        System.out.println(s);
    }
    public static void main(String[] args) {
        Sub sub = new Sub("Start");
    }
}
```

If you run the **Sub** class, you'll see this on the console:

```
Base
Start
```

This proves that the first thing that the **Sub** class's constructor does is invoke the **Base** class's no-arg constructor. The Java compiler has quietly changed **Sub**'s constructor to the following without saving the modification to the source file.

```
public Sub(String s) {
    super();
    System.out.println(s);
}
```

The keyword **super** represents an instance of the direct superclass of the current object. Since **super** is called from an instance of **Sub**, **super** represents an instance of **Base**, its direct superclass.

You can explicitly call the parent's constructor from a subclass's constructor by using the **super** keyword, but **super** must be the first statement in the constructor. Using the **super** keyword is handy if you want another constructor in the superclass to be invoked. For example, you can modify the constructor in **Sub** to the following.

```
    public Sub(String s) {
        super(s);
        System.out.println(s);
    }
```

This constructor calls the single argument constructor of the parent class, by using **super(s)**. As a result, if you run the class you will see the following on the console.

```
Base.Start
Start
```

Now, what if the superclass does not have a no-arg constructor and you do not make an explicit call to another constructor from a subclass? This is illustrated in the **Parent** and **Child** classes in listing 6.7.

Listing 6.7: Implicit calling to the parent's constructor that does not exist

```
package app06;
class Parent {
    public Parent(String s) {
        System.out.println("Parent(String)");
    }
}

public class Child extends Parent {
    public Child() {
    }
}
```

This will generate a compile error because the compiler adds an implicit call to the no-argument constructor in **Parent**, while the **Parent** class has only one constructor, the one that accepts a **String**. You can remedy this situation by explicitly calling the parent's constructor from the **Child** class's constructor:

```
public Child() {
    super(null);
}
```

> **Note**
> It actually makes sense for a child class to call its parent's constructor from its own constructor because an instance of a subclass must always be accompanied by an instance of each of its parents. This way, calls to a method that is not overridden in a child class will be passed to its parent until the first in the hierarchy is found.

Calling the Superclass's Hidden Members

The **super** keyword has another purpose in life. It can be used to call a hidden member or an overridden method in a superclass. Since **super** represents an instance of the direct parent, super.*memberName* returns the specified member in the parent class. You can access any member in the superclass that is visible from the subclass. For example, Listing 6.8 shows two classes that have a parent-child relationship: **Tool** and **Pencil**.

Listing 6.8: Using super to access a hidden member

```
package app06;
class Tool {
    public String toString() {
        return "Generic tool";
    }
}

public class Pencil extends Tool {
    public String toString() {
        return "I am a Pencil";
    }

    public void write() {
        System.out.println(super.toString());
```

```
        System.out.println(toString());
    }

    public static void main(String[] args) {
        Pencil pencil = new Pencil();
        pencil.write();
    }
}
```

The **Pencil** class overrides the **toString** method in **Tool**. If you run the **Pencil** class, you will see the following on the console.

```
Generic tool
I am a Pencil
```

Unlike calling a parent's constructor, invoking a parent's member does not have to be the first statement in the caller method.

Type Casting

You can cast an object to another type. The rule is, you can only cast an instance of a subclass to its parent class. Casting an object to a parent class is called upcasting. Here is an example, assuming that **Child** is a subclass of **Parent**.

```
Child child = new Child();
Parent parent = child;
```

To upcast a **Child** object, all you need to do is assign the object to a reference variable of type **Parent**. Note that the **parent** reference variable cannot access the members that are only available in **Child**.

Because **parent** in the snippet above references an object of type **Child**, you can cast it back to **Child**. This time, it is called downcasting because you are casting an object to a class down the inheritance hierarchy. Downcasting requires that you write the child type in brackets. For example:

```
Child child = new Child();
Parent parent = child;// parent pointing to an instance of Child
Child child2 = (Child) parent; // downcasting
```

Downcasting to a subclass is only allowed if the parent class reference is already pointing to an instance of the subclass. The following will generate a compile error.

```
Object parent = new Object();
Child child = (Child) parent; // illegal downcasting, compile error
```

Final Classes

You can prevent others from extending your class by making it final using the keyword **final** in the class declaration. **final** may appear after or before the access modifier. For example:

```
public final class Pencil
final public class Pen
```

The first form is more common.

Even though making a class final makes your code slightly faster, the difference is too insignificant to notice. Design consideration, and not speed, should be the reason you make a class final. For example, the **java.lang.String** class is final because the designer of the class did not want us to change the behavior of the **String** class.

The instanceof Keyword

The **instanceof** keyword can be used to test if an object is of a specified type. It is normally used in an **if** statement and its syntax is.

```
if (objectReference instanceof type)
```

where *objectReference* references an object being investigated. For example, the following **if** statement returns **true**.

```
String s = "Hello";
if (s instanceof java.lang.String)
```

However, applying **instanceof** on a **null** reference variable returns **false**. For example, the following **if** statement returns **false**.

```
String s = null;
if (s instanceof java.lang.String)
```

Also, since a subclass "is a" type of its superclass, the following **if** statement, where **Child** is a subclass of **Parent**, returns **true**.

```
Child child = new Child();
if (child instanceof Parent)    // evaluates to true
```

Summary

Inheritance is one of the fundamental principles in object-oriented programming. Inheritance makes code extensible. In Java all classes by default extend the **java.lang.Object** class. To extend a class, use the **extends** keyword. Method overriding is another OOP feature directly related to inheritance. It enables you to change the behavior of a method in the parent class. You can prevent your class from being subclassed by making it final.

Questions

1. Does a subclass inherit its superclass's constructors?
2. Why is it legal to assign an instance of a subclass to a superclass variable?
3. What is the difference between method overriding and method overloading?
4. Why is it necessary for an instance of a subclass to be accompanied by an instance of each parent?

Chapter 7
Error Handling

Error handling is an important feature in any programming language. A good error handling mechanism makes it easier for programmers to write robust applications and to prevent bugs from creeping in. In some languages, programmers are forced to use multiple **if** statements to detect all possible conditions that might lead to an error. This could make code excessively complex. In a larger program, this practice could easily lead to spaghetti like code.

Java has a very nice approach to error handling by using the **try** statement. With this strategy, part of the code that could potentially lead to an error is isolated in a block. Should an error occur, this error is caught and resolved locally. This chapter teaches you how.

Catching Exceptions

You can isolate code that may cause a runtime error using the **try** statement, which normally is accompanied by the **catch** and **finally** statements. Such isolation typically occurs in a method body. If an error is encountered, Java stops the processing of the **try** block and jump to the **catch** block. Here you can gracefully handle the error or notify the user by 'throwing' a **java.lang.Exception** object. Another scenario is to re-throw the exception or a new **Exception** object back to the code that called the method. It is then up to the client how he or she would handle the error. If a thrown exception is not caught, the application will stop abruptly.

This is the syntax of the **try** statement.

```
try {
    [code that may throw an exception]
} [catch (ExceptionType-1 e) {
    [code that is executed when ExceptionType-1 is thrown]
}] [catch (ExceptionType-2 e) {
    [code that is executed when ExceptionType-2 is thrown]
}]
  ...
} [catch (ExceptionType-n e) {
    [code that is executed when ExceptionType-n is thrown]
}]
[finally {
    [code that runs regardless of whether an exception was thrown]]
}]
```

The steps for error handling can be summarized as follows:

1. Isolate code that could lead to an error in the **try** block.
2. For each individual **catch** block, write code that is to be executed if an

exception of that particular type occurs in the **try** block.
3. In the **finally** block, write code that will be run whether or not an error has occurred.

Note that the **catch** and **finally** blocks are optional, but one of them must exist. Therefore, you can have **try** with one or more **catch** blocks or **try** with **finally**.

The previous syntax shows that you can have more than one **catch** block. This is because the code can throw different types of exceptions. When an exception is thrown from a **try** block, control is passed to the first **catch** block. If the type of exception thrown matches or is a subclass of the exception in the first **catch** block, the code in the **catch** block is executed and then control goes to the **finally** block, if one exists.

If the type of the exception thrown does not match the exception type in the first **catch** block, the JVM goes to the next **catch** block and does the same thing until it finds a match. If no match is found, the exception object will be thrown to the method caller. If the caller does not put the offending code that calls the method in a **try** block, the program will crash.

To illustrate the use of this error handling, consider the **NumberDoubler** class in Listing 7.1. When the class is run, it will prompt you for input. You can type anything, including non-digits. If your input is successfully converted to a number, it will double it and print the result. If your input is invalid, the program will print an "Invalid input" message.

Listing 7.1: The NumberDoubler class

```
package app07;
import java.util.Scanner;
public class NumberDoubler {
    public static void main(String[] args) {
        Scanner scanner = new Scanner(System.in);
        String input = scanner.next();
        try {
            double number = Double.parseDouble(input);
            System.out.printf("Result: %s", number);
        } catch (NumberFormatException e) {
            System.out.println("Invalid input.");
        }
    }
}
```

The **NumberDoubler** class uses the **java.util.Scanner** class to take user input (**Scanner** was discussed in Chapter 5, "Core Classes").

```
Scanner scanner = new Scanner(System.in);
String input = scanner.next();
```

It then uses the static **parseDouble** method of the **java.lang.Double** class to convert the string input to a **double**. Note that the code that calls **parseDouble** resides in a **try** block. This is necessary because the **parseDouble** method may throw a **java.lang.NumberFormatException**, as indicated by the signature of the **parseDouble** method.

```
public static double parseDouble(String s)
        throws NumberFormatExcpetion
```

The **throws** statement in the method signature tells you that it could throw a **NumberFormatException** and it is the responsibility of the method caller to catch it.

Without the **try** block, invalid input will give you this embarrassing error message before the system crashes:

```
Exception in thread "main" java.lang.NumberFormatException:
```

---

try without catch

A try statement can be used with **finally** without a catch block. You normally use this syntax to ensure that some code always gets executed whether or not an unexpected exception has been thrown in the **try** block. For example, after opening a database connection, you want to make sure its **close** method is called after you're done with the connection. To illustrate this scenario, consider the following pseudocode that opens a database connection.

```
Connection connection = null;
try {

    // open connection
    // do something with the connection and perform other tasks

} finally {
    if (connection != null) {
        // close connection
    }
}
```

If something unexpected occurs in the **try** block, the **close** method will always be called to release the resource.

Catching Multiple Exceptions

Java 7 and later allows you to catch multiple exceptions in a single **catch** block if the caught exceptions are to be handled by the same code. The syntax of the **catch** block is as follows, two exceptions being separated by the pipe character |.

```
catch(exception-1 | exception-2 ... e) {

    // handle exceptions

}
```

For example, the **java.net.ServerSocket** class's **accept** method can throw four exceptions: **java.nio.channels.IllegalBlockingModeException**, **java.net.SocketTimeoutException**, **java.lang.SecurityException**, and **java.io.Exception**. If, say, the first three exceptions are to be handled by the same code, you can write your **try** block like this:

```
try {
    serverSocket.accept();
} catch (SocketTimeoutException | SecurityException |
        IllegalBlockingModeException e) {

    // handle exceptions
```

```
} catch (IOException e) {

    // handle IOException

}
```

The try-with-resources Statement

Many Java operations involve some kind of resource that has to be closed after use. Before JDK 7, you used finally to make sure a **close** method is guaranteed to be called:

```
try {

    // open resource

} catch (Exception e) {

} finally {
    // close resource
}
```

This syntax can be tedious especially if the close method can throw an exception and can be null. For example, here's a typical code fragment to open a database connection.

```
Connection connection = null;
try {

    // create connection and do something with it

} catch (SQLException e) {

} finally {
    if (connection != null) {
        try {
            connection.close();
        } catch (SQLException e) {
        }
    }
}
```

You see, you need quite a bit of code in the **finally** block just for one resource, and it's not uncommon to have to open multiple resources in a single **try** block. JDK 7 adds a new feature, the try-with-resource statement, to make resource closing automatic. Its syntax is as follows.

```
try ( resources ) {

    // do something with the resources

} catch (Exception e) {
    // do something with e
}
```

For example, here is opening a database connection would look like in Java 7.

```
Connection connection = null;
try (Connection connection = openConnection();
        // open other resources, if any) {

    // do something with connection

} catch (SQLException e) {

}
```

Not all resources can be automatically closed. Only resource classes that implement **java.lang.AutoCloseable** can be automatically closed. Fortunately, in JDK 7 many input/output and database resources have been modified to support this feature. You'll see more examples of try-with-resources in Chapter 13, "Input/Output" and Chapter 22, "Java Database Connectivity."

The java.lang.Exception Class

Erroneous code can throw any type of exception. For example, an invalid argument may throw a **java.lang.NumberFormatException**, and calling a method on a null reference variable throws a **java.lang.NullPointerException**. All Java exception classes derive from the **java.lang.Exception** class. It is therefore worthwhile to spend some time examining this class.

Among others, the **Exception** class has the following methods:

`public String toString()`
: Returns the description of the exception.

`public void printStackTrace()`
: Prints the description of the exception followed by a stack trace for the **Exception** object. By analyzing the stack trace, you can find out which line is causing the problem. Here is an example of what **printStackTrace** may print on the console.

```
java.lang.NullPointerException
        at MathUtil.doubleNumber(MathUtil.java:45)
        at MyClass.performMath(MyClass.java: 18)
        at MyClass.main(MyClass.java: 90)
```

This tells you that a **NullPointerException** has been thrown. The line that throws the exception is Line 45 of the **MathUtil.java** class, inside the **doubleNumber** method. The **doubleNumber** method was called by **MyClass.performMath**, which in turns was called by **MyClass.main**.

Most of the time a **try** block is accompanied by a **catch** block that catches the **java.lang.Exception** in addition to other **catch** blocks. The **catch** block that catches **Exception** must appear last. If other **catch** blocks fail to catch the exception, the last **catch** will do that. Here is an example.

```
try {
    // code
} catch (NumberFormatException e) {
    // handle NumberFormatException
} catch (Exception e) {
    // handle other exceptions
```

```
}
```

You may want to use multiple **catch** blocks in the code above because the statements in the **try** block may throw a **java.lang.NumberFormatException** or other type of exception. If the latter is thrown, it will be caught by the last **catch** block.

Be warned, though: The order of the **catch** blocks is important. You cannot, for example, put a **catch** block for handling **java.lang.Exception** before any other **catch** block. This is because the JVM tries to match the thrown exception with the argument of the **catch** blocks in the order of appearance. **java.lang.Exception** catches everything; therefore, the **catch** blocks after it would never be executed.

If you have several **catch** blocks and the exception type of one of the **catch** blocks is derived from the type of another **catch** block, make sure the more specific exception type appears first. For example, when trying to open a file, you need to catch the **java.io.FileNotFoundException** just in case the file cannot be found. However, you may want to make sure that you also catch **java.io.IOException** so that other I/O-related exceptions are caught. Since **FileNotFoundException** is a child class of **IOException**, the **catch** block that handles **FileNotFoundException** must appear before the **catch** block that handles **IOException**.

Throwing an Exception from a Method

When catching an exception in a method, you have two options to handle the error that occurs inside the method. You can either handle the error in the method, thus quietly catching the exception without notifying the caller (this has been demonstrated in the previous examples), or you can throw the exception back to the caller and let the caller handle it. If you choose the second option, the calling code must catch the exception that is thrown back by the method.

Listing 7.2 presents a **capitalize** method that changes the first letter of a **String** to upper case.

Listing 7.2: The capitalize method

```
public String capitalize(String s) throws NullPointerException {
    if (s == null) {
        throw new NullPointerException(
                "You passed a null argument");
    }
    Character firstChar = s.charAt(0);
    String theRest = s.substring(1);
    return firstChar.toString().toUpperCase() + theRest;
}
```

If you pass a null to **capitalize**, it will throw a new **NullPointerException**. Pay attention to the code that instantiates the **NullPointerException** class and throws the instance:

```
        throw new NullPointerException(
                "Your passed a null argument");
```

The **throw** keyword is used to throw an exception. Don't confuse it with the **throws** statement which is used at the end of a method signature to indicate that an exception of a given type may be thrown from the method.

The following example shows code that calls **capitalize**.

```
String input = null;
try {
    String capitalized = util.capitalize(input);
    System.out.println(capitalized);
} catch (NullPointerException e) {
    System.out.println(e.toString());
}
```

Note
A constructor can also throw an exception.

User-Defined Exceptions

You can create a user-defined exception by subclassing **java.lang.Exception**. There are several reasons for having a user-defined exception. One of them is to create a customized error message.

For example, Listing 7.3 shows the **AlreadyCapitalizedException** class that derives from **java.lang.Exception**.

Listing 7.3: The AlreadyCapitalizedException class

```
package app07;
public class AlreadyCapitalizedException extends Exception {
    public String toString() {
        return "Input has already been capitalized";
    }
}
```

You can throw an **AlreadyCapitalizedException** from the **capitalize** method in Listing 7.2. The modified **capitalize** method is given in Listing 7.4.

Listing 7.4: The modified capitalize method

```
public String capitalize(String s)
        throws NullPointerException, AlreadyCapitalizedException {
    if (s == null) {
        throw new NullPointerException(
                "Your passed a null argument");
    }
    Character firstChar = s.charAt(0);
    if (Character.isUpperCase(firstChar)) {
        throw new AlreadyCapitalizedException();
    }
    String theRest = s.substring(1);
    return firstChar.toString().toUpperCase() + theRest;
}
```

Now, the **capitalize** method may throw one of two exceptions. You comma-delimit multiple exceptions in a method signature.

Clients that call **capitalize** must now catch both exceptions. This code shows a call to **capitalize**.

```
StringUtil util = new StringUtil();
String input = "Capitalize";
try {
```

```
    String capitalized = util.capitalize(input);
    System.out.println(capitalized);
} catch (NullPointerException e) {
    System.out.println(e.toString());
} catch (AlreadyCapitalizedException e) {
    e.printStackTrace();
}
```

Since **NullPointerException** and **AlreadyCapitalizedException** do not have a parent-child relationship, the order of the **catch** blocks above is not important.

When a method throws multiple exceptions, rather than catch all the exceptions, you can simply write a **catch** block that handles **java.lang.Exception**. Rewriting the code above:

```
StringUtil util = new StringUtil();
String input = "Capitalize";
try {
    String capitalized = util.capitalize(input);
    System.out.println(capitalized);
} catch (Exception e) {
    System.out.println(e.toString());
}
```

While it's more concise, the latter lacks specifics and does not allow you to handle each exception separately.

Final Words on Exception Handling

The **try** statement imposes some performance penalty. Therefore, do not use it over-generously. If it is not hard to test for a condition, then you should do the testing rather than depending on the **try** statement. For example, calling a method on a null object throws a **NullPointerException**. Therefore, you could always surround a method call with a **try** block:

```
try {
    ref.method();
...
```

However, it is not hard at all to check if **ref** is null prior to calling **methodA**. Therefore, the following code is better because it eliminates the **try** block.

```
if (ref != null) {
    ref.methodA();
}
```

Summary

This chapter discussed the use of structured error handling and presented examples for each case. You have also been introduced to the **java.lang.Exception** class and its properties and methods. The chapter concluded with a discussion of user-defined exceptions.

Question

1. What is the advantage of the **try** statement?
2. Can a **try** statement be used with **finally** and without **catch**?
3. What is try-with-resources?

Chapter 8
Numbers and Dates

In Java numbers are represented by the primitives **byte**, **short**, **int**, **long**, **float**, **double**, and their wrapper classes, which were explained in Chapter 5, "Core Classes." Dates can be represented by different classes, most commonly by the **java.util.Date** class. There are three issues when working with numbers and dates: parsing, formatting, and manipulation.

Parsing deals with the conversion of a string into a number or a date. Parsing is commonplace because Java programs often require user input and user input is received as a **String**. If a program expects a number or a date but receives a **String**, then the **String** has to be converted into a number or a date. Conversion is not always straightforward. Before conversion can take place, you first need to read the **String** and check if it contains characters that make up a number or a date. For example, "123data" is not a number even though it starts with a number. "123.45" is a float, but not an integer. "12/25/2011" looks like a date, but this is only valid if the program is expecting a date in mm/dd/yyyy format. Converting a string to a number is called number parsing, and converting a string to a date is referred to as date parsing.

Once you have a number or a date, you may want to display it in a specific format. For instance, 1000000 may be displayed as 1,000,000 and 12/25/2011 as Dec 25, 2011. This is number formatting and date formatting, respectively.

This chapter discusses number and date parsing, as well as number and date formatting. These tasks are easily achieved in Java as it provides classes for this purpose. In addition, the **java.lang.Math** class, which provides methods to perform mathematical operations, is also discussed. On top of that, there is a section on the **java.util.Calendar** class, a utility for manipulating dates.

Number Parsing

A Java program may require that the user input a number that will be processed or become an argument of a method. For example, a currency converter program would need the user to type in a value to be converted. You can use the **java.util.Scanner** class to receive user input. However, the input will be a **String**, even though it represents a number. Before you can work with the number, you need to parse the string. The outcome of a successful parsing is a number.

Therefore, the purpose of number parsing is to convert a string into a numeric primitive type. If parsing fails, for example because the string is not a number or a number outside the specified range, your program can throw an exception.

The wrappers of primitives—the **Byte**, **Short**, **Integer**, **Long**, **Float**, and **Double** classes—provide static methods for parsing strings. For example, **Integer** has a **parseInteger** method with the following signature.

```
public static int parseInt(String s) throws NumberFormatException
```

This method parses a **String** and returns an **int**. If the **String** does not contain a valid integer representation, a **NumberFormatException** is thrown.

For example, the following snippet uses **parseInt** to parse the string "123" to 123.

```
int x = Integer.parseInt("123");
```

Similarly, **Byte** provides a **parseByte** method, **Long** a **parseLong** method, **Short** a **parseShort** method, **Float** a **parseFloat** method, and **Double** a **parseDouble** method.

For example, the **NumberTest** class in Listing 8.1 takes user input and parses it. If the user types in an invalid number, an error message will be displayed.

Listing 8.1: Parsing numbers (NumberTest.java)

```
package app08;
import java.util.Scanner;
public class NumberTest {
    public static void main(String[] args) {
        Scanner scanner = new Scanner(System.in);
        String userInput = scanner.next();
        try {
            int i = Integer.parseInt(userInput);
            System.out.println("The number entered: " + i);
        } catch (NumberFormatException e) {
            System.out.println("Invalid user input");
        }
    }
}
```

Number Formatting

Number formatting helps make numbers more readable. For example, 1000000 is more readable if printed as 1,000,000 (or 1.000.000 if your locale uses . to separate the thousands). For number formatting Java offers the **java.text.NumberFormat** class, which is an abstract class. Since it is abstract, you cannot create an instance using the **new** keyword. Instead, you instantiate its subclass **java.text.DecimalFormat**, which is a concrete implementation of **NumberFormat**.

```
NumberFormat nf = new DecimalFormat();
```

However, you should not call the **DecimalFormat** class's constructor directly. Instead, use the the **NumberFormat** class's **getInstance** static method. This method may return an instance of **DecimalFormat** but might also return an instance of a subclass other than **DecimalFormat**.

Now, how do you use **NumberFormat** to format numbers, such as 1234.56? Easy, simply pass the numbers to its **format** method and you'll get a **String**. However, should number 1234.56 be formatted as 1,234.56 or 1234,56? Well, it really depends in which side of the Atlantic you live. If you are in the US, you may want 1,234.56. If you live in Germany, however, 1234,56 makes more sense. Therefore, before you start using the **format** method, you want to make sure you get the correct instance of **NumberFormat** by telling it where you live, or, actually, in what locale you want it formatted. In Java, a locale is represented by

the **java.util.Locale** class, which I'll explain in Chapter 19, "Internationalization." For now, remember that the **getInstance** method of the **NumberFormat** class also has an overload that accepts a **java.util.Locale**.

```
public NumberFormat getInstance(java.util.Locale locale)
```

If you pass **Locale.Germany** to the method, you'll get a **NumberFormat** object that formats numbers according to the German locale. If you pass **Locale.US**, you'll obtain one for the US number format. The no-argument **getInstance** method returns a **NumberFormat** object with the user's computer locale.

Listing 8.2 shows the **NumberFormatTest** class that demonstrates how to use the **NumberFormat** class to format a number.

Listing 8.2: The NumberFormatTest class

```
package app08;
import java.text.NumberFormat;
import java.util.Locale;
public class NumberFormatTest {
    public static void main(String[] args) {
        NumberFormat nf = NumberFormat.getInstance(Locale.US);
        System.out.println(nf.getClass().getName());
        System.out.println(nf.format(123445));
    }
}
```

When run, the output of the execution is

```
java.text.DecimalFormat
123,445
```

The first output line shows that a **java.text.DecimalFormat** object was produced upon calling **NumberFormat.getInstance**. The second shows how the **NumberFormat** formats the number 123445 into a more readable form.

Number Parsing with java.text.NumberFormat

You can use the **parse** method of **NumberFormat** to parse numbers. One of this method's overloads has the following signature:

```
public java.lang.Number parse(java.lang.String source)
        throws ParseException
```

Note that **parse** returns an instance of **java.lang.Number**, the parent of such classes as **Integer**, **Long**, etc.

The java.lang.Math Class

The **Math** class is a utility class that provides static methods for mathematical operations. There are also two static final double fields: **E** and **PI**. **E** represents the base of natural logarithms (e). Its value is close to 2.718. **PI** is the ratio of the circumference of a circle to its diameter (pi). Its value is 22/7 or approximately 3.1428.

Some of the methods in the **Math** class are given below.

```
public static double abs(double a)
```

Returns the absolute value of the specified double..

`public static double acos(double a)`
Returns the arc cosine of an angle, in the range of 0.0 through pi.

`public static double asin(double a)`
Returns the arc sine of an angle, in the range of –pi/2 through pi/2.

`public static double atan(double a)`
Returns the arc tangent of an angle, in the range of –pi/2 through pi/2.

`public static double cos(double a)`
Returns the cosine of an angle.

`public static double exp(double a)`
Returns Euler's number e raised to the power of the specified double.

`public static double log(double a)`
Returns the natural logarithm (base e) of the specified double.

`public static double log10(double a)`
Returns the base 10 logarithm of the specified double.

`public static double max(double a, double b)`
Returns the greater of the two specified double values.

`public static double min(double a, double b)`
Returns the smaller of the two specified double values.

The java.util.Date Class

There are at least two classes to use when working with dates and times. The first is **java.util.Date**, a class normally used to represent dates and times. The second is **java.util.Calendar**, often used for manipulating dates. In addition, times can also be represented as a **long**. See the discussion of the **currentTimeMillis** method of the **java.lang.System** class in Chapter 5, "Core Classes."

Date has two constructors that you can safely use (the other constructors are deprecated):

```
public Date()
public Date(long time)
```

The no-arg constructor creates a **Date** representing the current date and time. The second constructor creates a **Date** that represents the specified number of milliseconds since January 1, 1970, 00:00:00 GMT.

The **Date** class features several useful methods, two of them are **after** and **before**.

```
public boolean after(Date when)

public boolean before(Date when)
```

The **after** method returns **true** if this date is a later time than the *when* argument. Otherwise, it returns **false**. The **before** method returns **true** if this date is before the specified date and returns **false** otherwise.

For example, the following code prints "date1 before date2" because the first date represents a time that is one millisecond earlier than the second.

```
Date date1 = new Date(1000);
```

```
Date date2 = new Date(1001);
if (date1.before(date2)) {
    System.out.println("date1 before date2");
} else {
    System.out.println("date1 not before date2");
}
```

Many of the methods in **Date**, such as **getDate**, **getMonth**, **getYear**, are deprecated. You should not use these methods. Instead, use similar methods in the **java.util.Calendar** class.

The java.util.Calendar Class

The **java.util.Date** class has methods that allow you to construct a **Date** object from date components, such as the day, month, and year. However, these methods are deprecated. You should use **java.util.Calendar** instead.

To obtain a **Calendar** object, use one of the two static **getInstance** methods. Here are their signatures:

```
public static Calendar getInstance()
```

```
public static Calendar getInstance(Locale locale)
```

The first overload returns an instance that employs the computer's locale.

There's a lot you can do with a **Calendar**. For example, you can call its **getTime** method to obtain a **Date** object. Here is its signature:

```
public final Date getTime();
```

The resulting **Date** object, needless to say, contains components you initially passed to construct the **Calendar** object. In other words, if you construct a **Calendar** object that represents May 7, 2000 00:00:00, the **Date** object obtained from its **getTime** method will also represent May 7, 2000 00:00:00.

To obtain a date part, such as the hour, the month, or the year, use the **get** method. A first glance at its signature does not reveal much on how to use this method.

```
public int get(int field)
```

To use it, pass a valid field to the **get** method. A valid field is one of the following values: **Calendar.YEAR**, **Calendar.MONTH**, **Calendar.DATE**, **Calendar.HOUR**, **Calendar.MINUTE**, **Calendar.SECOND**, and **Calendar.MILLISECOND**.

get(Calendar.YEAR) returns an **int** representing the year. If it is year 2010, you get 2010. **get(Calendar.MONTH)** returns a zero-based index of the month, with 0 representing January and 11 representing December. The others (**get(Calendar.DATE)**, **get(Calendar.HOUR)**, and so on) return a number representing the date/time unit.

The last thing worth mentioning: if you already have a **Date** object and want to make use of the methods in **Calendar**, you can construct a **Calendar** object by using the **setTime** method:

```
public void setTime(Date date)
```

Here is an example:

```
// myDate is a Date
```

```
Calendar calendar = Calendar.getInstance();
calendar.setTime(myDate);
```

To change a date/time component, call its **set** method:

```
public void set(int field, int value)
```

For example, to change the month component of a **Calendar** object to **December**, write this.

```
calendar.set(Calendar.MONTH, Calendar.DECEMBER)
```

There are also **set** method overloads for changing multiple components at the same time:

```
public void set(int year, int month, int date)
```

```
public void set(int year, int month, int date,
        int hour, int minute, int second)
```

Date Parsing and Formatting with DateFormat

Java's answer to date parsing and formatting is the **java.text.DateFormat** and **java.text.SimpleDateFormat** classes. **DateFormat** is an abstract class with static **getInstance** methods that allows you to obtain an instance of a subclass. **SimpleDateFormat** is a concrete implementation of **DateFormat** that is easier to use than its parent. This section covers both classes.

DateFormat

DateFormat supports styles and patterns. There are four styles for formatting a **Date**. Each style is represented by an **int** value. The four **int** fields that represent the styles are:

- **DateFormat.SHORT**. For example, 12/2/11
- **DateFormat.MEDIUM**. For example, Dec 2, 2011
- **DateFormat.LONG**. For example, December 2, 2011
- **DateFormat.FULL**. For example, Friday, December 2, 2011

When you create a **DateFormat**, you need to decide which style you will be using for parsing or formatting. You cannot change a **DateFormat**'s style once you create it, but you can definitely have multiple instances of **DateFormat** that support different styles.

To obtain a **DateFormat** instance, call this static method.

```
public static DateFormat getDateInstance(int style)
```

where *style* is one of **DateFormat.SHORT**, **DateFormat.MEDIUM**, **DateFormat.Long**, or **DateFormat.FULL**. For example, the following code creates a **DateFormat** instance having the **MEDIUM** style.

```
DateFormat df = DateFormat.getDateInstance(DateFormat.MEDIUM)
```

To format a **Date** object, call its **format** method:

```
public final java.lang.String format(java.util.Date date)
```

To parse a string representation of a date, use the **parse** method. Here is the signature of **parse**.

```
public java.util.Date parse(java.lang.String date)
        throws ParseException
```

Note that you must compose your string according to the style of the **DateFormat**.

Listing 8.3 shows a class that parses and formats a date.

Listing 8.3: The DateFormatTest class

```
package app08;
import java.text.DateFormat;
import java.text.ParseException;
import java.util.Date;
public class DateFormatTest {
    public static void main(String[] args) {
        DateFormat shortDf =
                DateFormat.getDateInstance(DateFormat.SHORT);
        DateFormat mediumDf =
                DateFormat.getDateInstance(DateFormat.MEDIUM);
        DateFormat longDf =
                DateFormat.getDateInstance(DateFormat.LONG);
        DateFormat fullDf =
                DateFormat.getDateInstance(DateFormat.FULL);
        System.out.println(shortDf.format(new Date()));
        System.out.println(mediumDf.format(new Date()));
        System.out.println(longDf.format(new Date()));
        System.out.println(fullDf.format(new Date()));

        // parsing
        try {
            Date date = shortDf.parse("12/12/2006");
        } catch (ParseException e) {
        }
    }
}
```

Another point to note when working with **DateFormat** (and **SimpleDateFormat**) is leniency. Leniency refers to whether or not a strict rule will be applied at parsing. For example, if a **DateFormat** is lenient, it will accept this **String**: Jan 32, 2011, despite the fact that such a date does not exist. In fact, it will take the liberty of converting it to Feb 1, 2011. If a **DateFormat** is not lenient, it will not accept dates that do not exist. By default, a **DateFormat** object is lenient. The **isLenient** method and **setLenient** method allow you to check a **DateFormat**'s leniency and change it.

```
public boolean isLenient()
```

```
public void setLenient(boolean value)
```

SimpleDateFormat

SimpleDateFormat is more powerful than **DateFormat** because you can use your own date patterns. For example, you can format and parse dates in dd/mm/yyyy, mm/dd/yyyy, yyyy-mm-dd, and so on. All you need to do is pass a pattern to a **SimpleDateFormat** constructor.

SimpleDateFormat is a better choice than **DateFormat** especially for parsing. Here is one of the constructors in **SimpleDateFormat**.

```
public SimpleDateFormat(java.lang.String pattern)
        throws java.lang.NullPointerException,
        java.lang.IllegalArgumentException
```

The complete rules for a valid pattern can be read in the Javadoc for the **SimpleDateFormat** class. The more commonly used patterns can be used by a combination of y (representing a year digit), M (representing a month digit) and d (representing a date digit). Examples of patterns are dd/MM/yyyy, dd-MM-yyyy, MM/dd/yyyy, yyyy-MM-dd.

Listing 8.4 shows a class that uses **SimpleDateFormat** for parsing and formatting.

Listing 8.4: The SimpleDateFormatTest class

```
package app08;
import java.text.ParseException;
import java.text.SimpleDateFormat;
import java.util.Date;
public class SimpleDateFormatTest {
    public static void main(String[] args) {
        String pattern = "MM/dd/yyyy";
        SimpleDateFormat format = new SimpleDateFormat(pattern);
        try {
            Date date = format.parse("12/31/2011");
        } catch (ParseException e) {
            e.printStackTrace();
        }
        // formatting
        System.out.println(format.format(new Date()));
    }
}
```

Summary

In Java you use primitives and wrapper classes to represent number and the **java.util.Date** class to represents dates. There are three types of operations that you frequently perform when dealing with number and dates: parsing, formatting and manipulation. This chapter showed how to perform them.

Number parsing is achieved through the use of the **parse*XXX*** methods in wrapper classes, such as **parseInteger** in **java.lang.Integer** and **parseLong** in **java.lang.Long**. The **java.text.NumberFormat** class can be used for number parsing and number formatting. For complex mathematical operations, use the static methods in **java.lang.Math**.

Date manipulation, on the other hand, is best done with the help of the **java.util.Calendar** class. Date parsing and formatting can be performed by using the **java.text.DateFormat** and the **java.text.SimpleDateFormat** classes.

Questions

1. What can you do with the **java.lang.Math** class's static methods?
2. What do you use to represents dates?
3. What class should you use if you want to define you own date pattern?

Chapter 9
Interfaces and Abstract Classes

Java beginners often get the impression that an interface is simply a class without implementation code. While this is not technically incorrect, it obscures the real purpose of having the interface in the first place. The interface is more than that. The interface should be regarded as a contract between a service provider and its clients. This chapter therefore focuses on the concepts before explaining how to write an interface.

The second topic in this chapter is the abstract class. Technically speaking, an abstract class is a class that cannot be instantiated and must be implemented by a subclass. However, the abstract class is important because in some situations it can take the role of the interface. We'll see how to use the abstract class too in this chapter.

The Concept of Interface

When learning about the interface for the first time, novices often focus on how to write one, rather than understanding the concept behind it. They would think an interface is something like a Java class declared with the **interface** keyword and whose methods have no body.

While the description is not inaccurate, treating an interface as an implementation-less class misses the big picture. A better definition of an interface is a contract. It is a contract between a service provider (server) and the user of such a service (client). Sometimes the server defines the contract, sometimes the client does.

Consider this real-world example. Microsoft Windows is the most popular operating system today, but Microsoft does not make printers. For printing, we still rely on those people at HP, Canon, Samsung, and the like. Each of these printer makers uses a proprietary technology. However, their products can all be used to print any document from any Windows application. How come?

This is because Microsoft said something to this effect to printer manufacturers, "If you want your products useable on Windows (and we know you all do), you must implement this **Printable** interface."

The interface is as simple as this:

```
interface Printable {
    void print(Document document);
}
```

where *document* is the document to be printed.

Implementing this interface, printer makers then write printer drivers. Every printer has a different driver, but they all implement **Printable**. A printer driver is

an implementation of **Printable**. In this case, these printer drivers are the service provider.

The client of the printing service is all Windows applications. It is easy to print on Windows because an application just needs to call the **print** method and pass a **Document** object. Because the interface is freely available, client applications can be compiled without waiting for an implementation to be available.

The point is, printing to different printers from different applications is possible thanks to the **Printable** interface. A contract between printing service providers and printing clients.

An interface can define both fields and methods. However, methods in an interface have no implementation. To be useful, an interface has to have an implementation class that actually performs the action.

Figure 9.1 illustrates the **Printable** interface and its implementation in an UML class diagram.

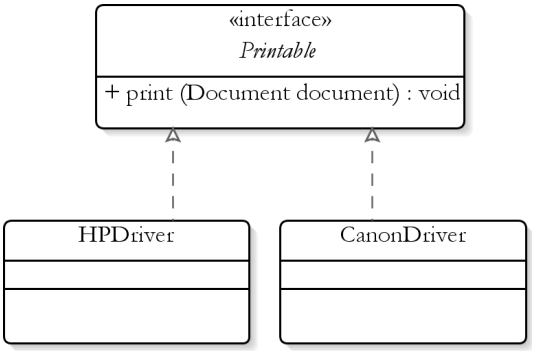

Figure 9.1: An interface and two implementation classes in a class diagram

In the class diagram, an interface has the same shape as a class, however the name is printed in italic and prefixed with <<interface>>. The **HPDriver** and **CanonDriver** classes are classes that implement the **Printable** interface. The implementations are of course different. In the **HPDriver** class, the **print** method contains code that enables printing to a HP printer. In **CanonDriver**, the code enables printing to a Canon driver. In a UML class diagram, a class and an interface are joined by a dash-line with an arrow. This type of relationship is often called realization because the class provides real implementation (code that actually works) of the abstraction provided by the interface.

Note

This case study is contrived but the problem and the solution are real. I hope this provides you with more understanding of what the interface really is. It is a contract.

The Interface, Technically Speaking

Now that you understand what the interface is, let's examine how you can create one. In Java, like the class, the interface is a type. Follow this format to write an interface:

```
accessModifier interface interfaceName {

}
```

Like a class, an interface has either a public or the default access level. An interface can have fields and methods. However, all interface members are implicitly public. Listing 9.1 shows an interface named **Printable**.

Listing 9.1: The Printable interface

```
package app09;
public interface Printable {
    void print(Object o);
}
```

The **Printable** interface has a method, **print**. Note that **print** is public even though there is no **public** keyword in front of the method declaration. You can of course use the keyword **public** before the method signature.

Just like a class, an interface is a template for creating objects. Unlike an ordinary class, however, an interface cannot be instantiated. It simply defines a set of methods that Java classes can implement.

To implement an interface, you use the **implements** keyword after the class declaration. A class can implement multiple interfaces. For example, Listing 9.2 shows the **CanonDriver** class that implements **Printable**.

Listing 9.2: An implementation of the Printable interface

```
package app09;
public class CanonDriver implements Printable {
    public void print(Object obj) {
        // code that does the printing
    }
}
```

An implementation class has to override all methods in the interface. The relationship between an interface and its implementing class can be likened to a parent class and a subclass. An instance of the class is also an instance of the interface. For example, the following **if** statement evaluates to **true**.

```
CanonDriver driver = new CanonDriver();
if (driver instanceof Printable)    // evaluates to true
```

The interface supports inheritance. An interface can extend another interface. If interface **A** extends interface **B**, **A** is said to be a subinterface of **B**. **B** is the superinterface of **A**. Because **A** directly extends **B**, **B** is the direct superinterface of **A**. Any interfaces that extend **B** are indirect subinterfaces of **A**. Figure 9.2 shows an interface that extends another interface. Note that the type of the line connecting both interfaces is the same as the one used for extending a class.

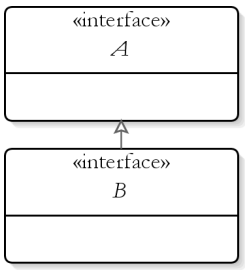

Figure 9.2: Extending an interface

Fields in an Interface

Fields in an interface must be initialized and are implicitly public, static, and final. However, you may redundantly use the modifiers **public**, **static**, and **final**. These lines of code have the same effect.

```
public int STATUS = 1;
int STATUS = 1;
public static final STATUS = 1;
```

Note that by convention field names in an interface are written in upper case.

It is a compile error to have two fields with the same name in an interface. However, an interface might inherit more than one field with the same name from its superinterfaces.

Methods

You declare methods in an interface just as you would in a class. However, methods in an interface do not have a body, they are immediately terminated by a semicolon. All methods are implicitly public and abstract, even though it is legal to have the **public** and **abstract** modifiers in front of a method declaration.

The syntax of a method in an interface is

```
[methodModifiers] ReturnType MethodName(listOfArgument)
        [ThrowClause];
```

where *methodModifiers* is **abstract** and **public**.

In addition, methods in an interface must not be declared static because static methods cannot be abstract.

Base Classes

Some interfaces have many methods, and implementing classes must override all the methods. This can be a tedious task if only some of the methods are actually used in the code. For this reason, you can create a generic implementation class that overrides the methods in an interface with default code. An implementing class can then extend the generic class and overrides only methods it wants to change. This kind of generic class, often called a base class, is handy because it helps you code faster.

For example, the **javax.servlet.Servlet** interface is the interface that must be implemented by all servlet classes. This interface has five methods: **init**, **service**, **destroy**, **getServletConfig**, **getServletInfo**. Of the five, only the **service** method is always implemented by servlet classes. The **init** method is implemented occasionally, but the rest are rarely used. Despite the fact, all implementing classes must provide implementation for all five methods. What a chore this would be for servlet programmers.

To make servlet programming easier and more fun, the Servlet API defines the **javax.servlet.GenericServlet** class, which provides default implementation for all methods in the **Servlet** interface. When you write a servlet, instead of writing a class that implements the **javax.servlet.Servlet** interface (and ending up implementing five methods), you extend the **javax.servlet.GenericServlet** and provide only implementation for methods you need to use (most probably, only the **service** method).

Compare the **TediousServlet** class in Listing 9.7, which implements **javax.servlet.Servlet**, and the one in Listing 9.8, which extends **javax.servlet.GenericServlet**. Which one is simpler?

Listing 9.7: The TediousServlet class

```
package test;
import java.io.IOException;
import javax.servlet.Servlet;
import javax.servlet.ServletConfig;
import javax.servlet.ServletException;
import javax.servlet.ServletRequest;
import javax.servlet.ServletResponse;

public class TediousServlet implements Servlet {
    public void init(ServletConfig config)
            throws ServletException {
    }
    public void service(ServletRequest request,
            ServletResponse response)
            throws ServletException, IOException {
        response.getWriter().print("Welcome");
    }
    public void destroy() {
    }
    public String getServletInfo() {
        return null;
    }
    public ServletConfig getServletConfig() {
        return null;
    }
}
```

Listing 9.8: The FunServlet class

```
package test;
import java.io.IOException;
import javax.servlet.GenericServlet;
import javax.servlet.ServletException;
import javax.servlet.ServletRequest;
import javax.servlet.ServletResponse;

public class FunServlet extends GenericServlet {
    public void service(ServletRequest request,
            ServletResponse response)
            throws ServletException, IOException {
        response.getWriter().print("Welcome");
    }
}
```

Note

Servlet programming is discussed in Chapter 26, “Java Web Applications.”

Abstract Classes

With the interface, you have to write an implementation class that perform the

actual action. If there are many methods in the interface, you risk wasting time overriding methods that you don't use. An abstract class has a similar role to an interface, i.e. provide a contract between a service provider and its clients, but at the same time an abstract class can provide partial implementation. Methods that must be explicitly overridden can be declared abstract. You still need to create an implementation class because you cannot instantiate an abstract class, but you don't need to override methods you don't want to use or change.

You create an abstract class by using the **abstract** modifier in the class declaration. To make an abstract method, use the **abstract** modifier in front of the method declaration. Listing 9.9 shows an abstract **DefaultPrinter** class as an example.

Listing 9.9: The DefaultPrinter class

```
package app09;
public abstract class DefaultPrinter {
    public String toString() {
        return "Use this to print documents.";
    }
    public abstract void print(Object document);
}
```

There are two methods in **DefaultPrinter**, **toString** and **print**. The **toString** method has an implementation, so you do not need to override this method in an implementation class, unless you want to change its return value. The **print** method is declared abstract and does not have a body. Listing 9.10 presents a **MyPrinterClass** class that is the implementation class of **DefaultPrinter**.

Listing 9.10: An implementation of DefaultPrinter

```
package app09;
public class MyPrinter extends DefaultPrinter {
    public void print(Object document) {
        System.out.println("Printing document");
        // some code here
    }
}
```

A concrete implementation class such as **MyPrinter** must override all abstract methods. Otherwise, it itself must be declared abstract.

Declaring a class abstract is a way to tell the class user that you want them to extend the class. You can still declare a class abstract even if it does not have an abstract method.

In UML class diagrams, an abstract class looks similar to a concrete class, except that the name is italicized. Figure 9.3 shows an abstract class.

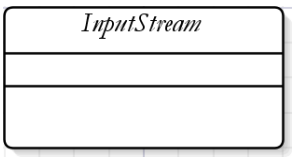

Figure 9.3: An abstract class

Summary

The interface plays an important role in Java because it defines a contract between a service provider and its clients. This chapter showed you how to use the interface. A base class provides a generic implementation of an interface and expedites program development by providing default implementation of code.

An abstract class is like an interface, but it may provide implementation of some of its methods.

Questions

1. Why is it more appropriate to regard an interface as a contract than as a implementation-less class?
2. What is a base class?
3. What is an abstract class?
4. Is a base class the same as an abstract class?

Chapter 10
Enums

In Chapter 2, “Language Fundamentals,” you learned that you sometimes use static final fields as enumerated values. Java 5 added a new type, enum, for enumerating values. This chapter presents a complete discussion of enum.

An Overview of Enum

You use enum to create a set of valid values for a field or a method. For example, in a typical application, the only possible values for the **customerType** are **Individual** or **Organization**. For the **State** field, valid values may be all the states in the US plus Canadian provinces, and some others. With **enum**, you can easily restrict your program to take only one of the valid values.

An enum type can stand alone or can be part of a class. You make it stand alone if it needs to be referenced from multiple places in your application. If it is only used from inside a class, enum is better made part of the class.

As an example, consider the **CustomerType** enum definition in Listing 10.1.

Listing 10.1: The CustomerType enum

```
package app10;
public enum CustomerType {
    INDIVIDUAL,
    ORGANIZATION
}
```

The **CustomerType** enum has two enumerated values: **INDIVIDUAL** and **ORGANIZATION**. Enum values are case sensitive and by convention are capitalized. Two enum values are separated by a comma and values can be written on a single line or multiple lines. The enum in Listing 10.1 is written in multiple lines to improve readability.

Using an enum is like using a class or an interface. For example, the **Customer** class in Listing 10.2 uses the **CustomerType** enum in Listing 10.1 as a field type.

Listing 10.2: The Customer class that uses CustomerType

```
package app10;
public class Customer {
    public String customerName;
    public CustomerType customerType;
    public String address;
}
```

You can use a value in an enum just like you would a class’s static member. For example, this code illustrates the use of **CustomerType**.

```
Customer customer = new Customer();
customer.customerType = CustomerType.INDIVIDUAL;
```

Notice how the **customerType** field of the **Customer** object is assigned the enumerated value **INDIVIDUAL** of the **CustomerType** enum? Because the **customerType** field is of type **CustomerType**, it can only be assigned a value of the **CustomerType** enum.

The use of enum at first glance is no different than the use of static finals. However, there are some basic differences between an enum and a class incorporating static finals.

Static finals are not a perfect solution for something that should accept only predefined values. For example, consider the **CustomerTypeStaticFinals** class in Listing 10.3.

Listing 10.3: Using static finals

```
package app10;
public class CustomerTypeStaticFinals {
    public static final int INDIVIDUAL = 1;
    public static final int ORGANIZATION = 2;
}
```

Suppose you have a class named **OldFashionedCustomer** that resembles the **Customer** class in Listing 10.2, but uses **int** for its **customerType** field.

The following code creates an instance of **OldFashionedCustomer** and assigns a value to its **customerType** field:

```
OldFashionedCustomer ofCustomer = new OldFashionedCustomer();
ofCustomer.customerType = 5;
```

Notice that there is nothing preventing you from assigning an invalid integer to **customerType**? In guaranteeing that a variable is assigned only a valid value, enums are better than static finals.

Another difference is that an enumerated value is an object. Therefore, it behaves like an object. For example, you can use it as a **Map** key. The section, “The Enum Class” discusses enums as objects in detail.

Enums in a Class

You can use enums as members of a class. You use this approach if the enum is only used internally inside the class. For example, the **Shape** class in Listing 10.4 defines a **ShapeType** enum.

Listing 10.4: Using an enum as a class member

```
package app10;
public class Shape {
    private enum ShapeType {
        RECTANGLE, TRIANGLE, OVAL
    };
    private ShapeType type = ShapeType.RECTANGLE;
    public String toString() {
        if (this.type == ShapeType.RECTANGLE) {
            return "Shape is rectangle";
        }
        if (this.type == ShapeType.TRIANGLE) {
```

```
            return "Shape is triangle";
        }
        return "Shape is oval";
    }
}
```

The java.lang.Enum Class

When you define an enum, the compiler creates a class definition that extends the **java.lang.Enum** class. This class is a direct descendant of **java.lang.Object**. Unlike ordinary classes, however, an enum has the following properties:

- There is no public constructor, making it impossible to instantiate.
- It is implicitly static
- There is only one instance for each enum constant.
- You can call the method values on an enum in order to iterate its enumerated values. See the next section "Iterating Enumerated Values" for more details on this.

Iterating Enumerated Values

You can iterate the values in an enum using the **for** loop (discussed in Chapter 3, "Statements"). You first need to call the **values** method that returns an array-like object that contains all values in the specified enum. Using the **CustomerType** enum in Listing 10.1, you can use the following code to iterate over it.

```
for (CustomerType customerType : CustomerType.values() ) {
    System.out.println(customerType);
}
```

This prints all values in **CustomerType**, starting from the first value. Here is the result:

```
INDIVIDUAL
ORGANIZATION
```

Switching on Enum

The **switch** statement can also work on enumerated values of an enum. Here is an example using the **CustomerType** enum in Listing 10.1 and the **Customer** class in Listing 10.2:

```
Customer customer = new Customer();
customer.customerType = CustomerType.INDIVIDUAL;

switch (customer.customerType) {
case INDIVIDUAL:
    System.out.println("Customer Type: Individual");
    break;
case ORGANIZATION:
    System.out.println("Customer Type: Organization");
```

```
    break;
}
```

Note that you must *not* prefix each case with the enum type. The following would raise a compile error:

```
case CustomerType.INDIVIDUAL:
    //
case CustomerType.ORGANIZATION:
    //
```

Summary

Java supports enum, a special class that is a subclass of **java.lang.Enum**. Enum is preferred over static finals because it is more secure. You can switch on an enum and iterate its values by using the **values** method in an enhanced **for** loop.

Questions

1. How do you write an enum?
2. Why are enums safer than static final fields?

Chapter 11
The Collections Framework

When writing an object-oriented program, you often work with groups of objects, either objects of the same type or of different types. In Chapter 5, "Core Classes" you learned that arrays can be used to group objects of the same type. Unfortunately, arrays lack the flexibility you need to rapidly develop applications. They cannot be resized and they cannot hold objects of different types. Fortunately, Java comes with a set of interfaces and classes that make working with groups of objects easier: the Collections Framework. This chapter deals with the most important types in the Collections Framework that a Java beginner should know. Most of them are very easy to use and there's no need to provide extensive examples. More attention is paid to the last section of the chapter, "Making Your Objects Comparable and Sortable" where carefully designed examples are given because it is important for every Java programmer to know how to make objects comparable and sortable.

Note on Generics

Discussing the Collections Framework would be incomplete without generics. On the other hand, explaining generics without previous knowledge of the Collections Framework would be difficult. Therefore, there needs to be a compromise: The Collections Framework will be explained first in this chapter and will be revisited in Chapter 12, "Generics." Since up to this point no knowledge of generics is assumed, the discussion of the Collections Framework in this chapter will have to use class and method signatures as they appear in pre-5 JDK's, instead of signatures used in Java 5 or later that imply the presence of generics. As long as you read both this chapter and Chapter 12, you will have up-to date knowledge of both the Collections Framework and generics.

An Overview of the Collections Framework

A collection is an object that groups other objects. Also referred to as a container, a collection provides methods to store, retrieve, and manipulate its elements. Collections help Java programmers manage objects easily.

A Java programmer should be familiar with the most important types in the Collections Framework, all of which are part of the **java.util** package. The relationships between these types are shown in Figure 11.1.

The main type in the Collections Framework is, unsurprisingly, the **Collection** interface. **List**, **Set**, and **Queue** are three main subinterfaces of **Collection**. In addition, there is a **Map** interface that can be used for storing key/value pairs. A subinterface of **Map**, **SortedMap**, guarantees that the keys are in ascending order. Other implementations of **Map** are **AbstractMap** and its concrete implementation **HashMap**. Other interfaces include **Iterator** and **Comparator**. The latter is used to make objects sortable and comparable.

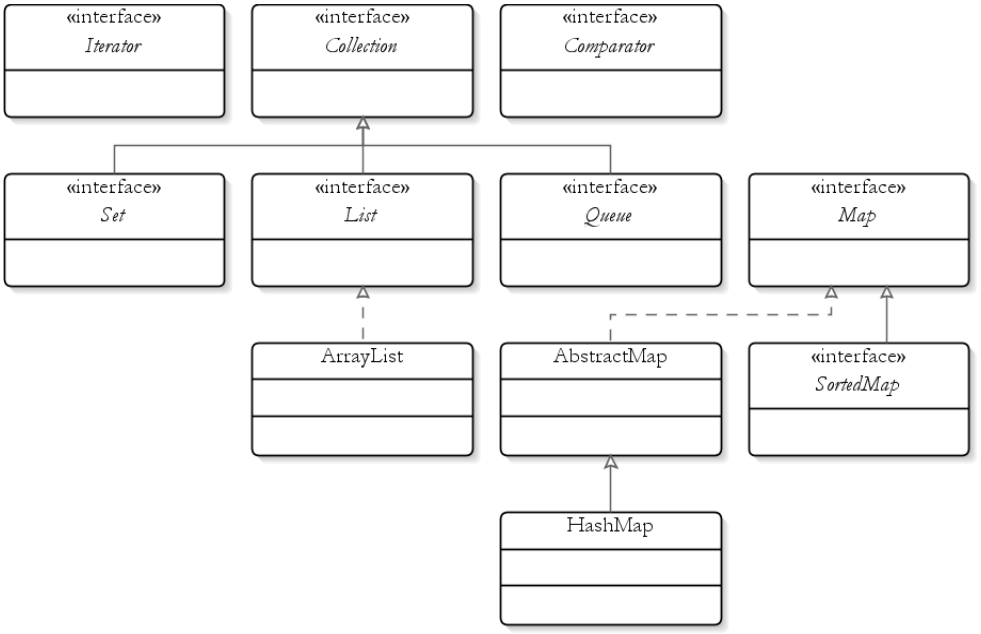

Figure 11.1: The Collections Framework

Most of the interfaces in the Collections Frameworks come with implementation classes. Sometimes there are two versions of an implementation, the synchronized version and the unsynchronized version. For instance, the **java.util.Vector** class and the **ArrayList** class are implementations of the **List** interface. Both **Vector** and **ArrayList** provide similar functionality, however **Vector** is synchronized and **ArrayList** unsynchronized. Synchronized versions of an implementation were included in the first version of the JDK. Only later did Sun add the unsynchronized versions so that programmers could write better performing applications. The unsynchronized versions should thus be in preference to the synchronized ones. If you need to use an unsynchronized implementation in a multi-threaded environment, you can still synchronize it yourself.

Note

Working in a multi-threaded environment is discussed in Chapter 19, "Java Threads."

The Collection Interface

The **Collection** interface groups objects together. Unlike arrays that cannot be resized and can only group objects of the same type, collections allow you to add any type of object and do not force you to specify an initial size.

Collection comes with methods that are easy to use. To add an element, you use the **add** method. To add members of another **Collection**, use **addAll**. To remove all elements, use **clear**. To inquire about the number of elements in a **Collection**, call its **size** method. To test if a **Collection** contains an element, use **isEmpty**. And, to move its elements to an array, use **toArray**.

An important point to note is that **Collection** extends the **Iterable** interface, from which **Collection** inherits the **iterator** method. This method returns an

Iterator object that you can use to iterate over the collection's elements. Check the section, "Iterable and Iterator" later in this chapter.

In addition, you'll learn how to use the **for** statement to iterate over a **Collection**'s elements.

List and ArrayList

List is the most popular subinterface of **Collection**, and **ArrayList** is the most commonly used implementation of **List**. Also known as a sequence, a **List** is an ordered collection. You can access its elements by using indices and you can insert an element into an exact location. Index 0 of a **List** references the first element, index 1 the second element, and so on.

The **add** method inherited from **Collection** appends the specified element to the end of the list. Here is its signature.

```
public boolean add(java.lang.Object element)
```

This method returns **true** if the addition is successful. Otherwise, it returns **false**. Some implementations of **List**, such as **ArrayList**, allow you to add null, some don't.

List adds another **add** method with the following signature:

```
public void add(int index, java.lang.Object element)
```

With this **add** method you can insert an element at any position.

In addition, you can replace and remove an element by using the **set** and **remove** methods, respectively.

```
public java.lang.Object set(int index, java.lang.Object element)
public java.lang.Object remove(int index)
```

The **set** method replaces the element at the position specified by *index* with *element* and returns the reference to the element inserted. The **remove** method removes the element at the specified position and returns a reference to the removed element.

To create a **List**, you normally assign an **ArrayList** object to a **List** reference variable.

```
List myList = new ArrayList();
```

The no-argument constructor of **ArrayList** creates an **ArrayList** object with an initial capacity of ten elements. The size will grow automatically as you add more elements than its capacity. If you know that the number of elements in your **ArrayList** will be more than its capacity, you can use the second constructor:

```
public ArrayList(int initialCapacity)
```

This will result in a slightly faster **ArrayList** because the instance does not have to grow in capacity.

List allows you to store duplicate elements in the sense that two or more references referencing the same object can be stored. Listing 11.1 demonstrates the use of **List** and some of its methods.

Listing 11.1: Using List

```
package app11;
import java.util.ArrayList;
```

```
import java.util.List;

public class ListTest {
    public static void main(String[] args) {
        List myList = new ArrayList();
        String s1 = "Hello";
        String s2 = "Hello";
        myList.add(100);
        myList.add(s1);
        myList.add(s2);
        myList.add(s1);
        myList.add(1);
        myList.add(2, "World");
        myList.set(3, "Yes");
        myList.add(null);
        System.out.println("Size: " + myList.size());
        for (Object object : myList) {
            System.out.println(object);
        }
    }
}
```

When run, here is the result on the console.

```
Size: 6
100
Hello
World
Yes
Hello
1
null
```

The **java.util.Arrays** class provides an **asList** method that lets you add multiple elements to a **List** in one go. For example, the following snippet adds multiple **String**s in a single call.

```
List members = Arrays.asList("Chuck", "Harry", "Larry", "Wang");
```

List also adds methods to search the collection, **indexOf** and **lastIndexOf**:

```
public int indexOf(java.lang.Object obj)
public int lastIndexOf(java.lang.Object obj)
```

indexOf compares the *obj* argument with its elements by using the **equals** method starting from the first element, and returns the index of the first match. **lastIndexOf** does the same thing but comparison is done from the last element to the first. Both **indexOf** and **lastIndexOf** return -1 if no match was found.

Note

List allows duplicate elements. By contrast, **Set** does not.

Iterating Over a Collection with Iterator and for

Iterating over a **Collection** is one of the most common tasks around when working with collections. There are two ways to do this: by using **Iterator** and by using

for.

Recall that **Collection** extends **Iterable**, which has one method: **iterator**. This method returns a **java.util.Iterator** that you can use to iterate over the **Collection**. The **Iterator** interface has the following methods:

- **hasNext**. **Iterator** employs an internal pointer that initially points to a place before the first element. **hasNext** returns **true** if there are more element(s) after the pointer. Calling **next** moves this pointer to the next element. Calling **next** for the first time on an **Iterator** causes its pointer to point to the first element.
- **next**. Moves the internal pointer to the next element and returns the element. Invoking **next** after the last element is returned throws a **java.util.NoSuchElementException**. Therefore, it is safest to call **hasNext** before invoking **next** to test if there is a next element.
- **remove**. Removes the element pointed to by the internal pointer.

A common way to iterate over a **Collection** using an **Iterator** is either by employing **while** or **for**. Suppose **myList** is an **ArrayList** that you want to iterate over. The following snippet uses a **while** statement to iterate over a collection and print each element in the collection.

```
Iterator iterator = myList.iterator();
while (iterator.hasNext()) {
    String element = (String) iterator.next();
    System.out.println(element);
}
```

This is identical to:

```
for (Iterator iterator = myList.iterator(); iterator.hasNext(); ) {
    String element = (String) iterator.next();
    System.out.println(element);
}
```

The **for** statement can iterate over a **Collection** without the need to call the **iterator** method. The syntax is

```
for (Type identifier : expression) {
    statement(s)
}
```

Here *expression* must be an **Iterable**. Since **Collection** extends **Iterable**, you can use enhanced **for** to iterate over any **Collection**. For example, this code shows how to use **for**.

```
for (Object object : myList) {
    System.out.println(object);
}
```

Using **for** to iterate over a collection is a shortcut for using **Iterator**. In fact, the code that uses **for** above is translated into the following by the compiler.

```
for (Iterator iterator = myList.iterator(); iterator.hasNext(); ) {
    String element = (String) iterator.next();
    System.out.println(element);
}
```

Set and HashSet

A **Set** represents a mathematical set. Unlike **List**, **Set** does not allow duplicates. There must not be two elements of a **Set**, say **e1** and **e2**, such that **e1.equals(e2)**. The **add** method of **Set** returns **false** if you try to add a duplicate element. For example, this code prints "addition failed."

```
Set set = new HashSet();
set.add("Hello");
if (set.add("Hello")) {
    System.out.println("addition successful");
} else {
    System.out.println("addition failed");
}
```

The first time you called **add**, the string "Hello" was added. The second time around it failed because adding another "Hello" would result in duplicates in the **Set**.

Some implementations of **Set** allow at most one null element. Some do not allow nulls. For instance, **HashSet**, the most popular implementation of **Set**, allows at most one null element. When using **HashSet**, be warned that there is no guarantee the order of elements will remain unchanged. **HashSet** should be your first choice of **Set** because it is faster than other implementations of **Set**, **TreeSet** and **LinkedHashSet**.

Queue and LinkedList

Queue extends **Collection** by adding methods that support the ordering of elements in a first-in-first-out (FIFO) basis. FIFO means that the element first added will be the first you get when retrieving elements. This is in contrast to a **List** in which you can choose which element to retrieve by passing an index to its **get** method.

Queue adds the following methods.

- **offer**. This method inserts an element just like the **add** method. However, **offer** should be used if adding an element may fail. This method returns **false** upon failing to add an element and does not throw an exception. On the other hand, a failed insertion with **add** throws an exception.
- **remove**. Removes and returns the element at the head of the **Queue**. If the **Queue** is empty, this method throws a **java.util.NoSuchElementException**.
- **poll**. This method is like the **remove** method. However, if the **Queue** is empty it returns null and does not throw an exception.
- **element**. Returns but does not remove the head of the **Queue**. If the **Queue** is empty, it throws a **java.util.NoSuchElementException**.
- **peek**. Also returns but does not remove the head of the **Queue**. However, **peek** returns null if the **Queue** is empty, instead of throwing an exception.

When you call the **add** or **offer** method on a **Queue**, the element is always added at the tail of the **Queue**. To retrieve an element, use the **remove** or **poll** method. **remove** and **poll** always remove and return the element at the head of the **Queue**.

For example, the following code creates a **LinkedList** (an implementation of **Queue**) to show the FIFO nature of **Queue**.

```
Queue queue = new LinkedList();
queue.add("one");
queue.add("two");
queue.add("three");
System.out.println(queue.remove());
System.out.println(queue.remove());
System.out.println(queue.remove());
```

The code produces this result:

```
one
two
three
```

This demonstrates that **remove** always removes the element at the head of the **Queue**. In other words, you cannot remove "three" (the third element added to the **Queue**) before removing "one" and "two."

Note

The **java.util.Stack** class is a **Collection** that behaves in a last-in-first-out (LIFO) manner.

Collection Conversion

Collection implementations normally have a constructor that accepts a **Collection** object. This enables you to convert a **Collection** to a different type of **Collection**. Here are the constructors of some implementations:

```
public ArrayList(Collection c)

public HashSet(Collection c)

public LinkedList(Collection c)
```

As an example, the following code converts a **Queue** to a **List**.

```
Queue queue = new LinkedList();
queue.add("Hello");
queue.add("World");
List list = new ArrayList(queue);
```

And this converts a **List** to a **Set**.

```
List myList = new ArrayList();
myList.add("Hello");
myList.add("World");
myList.add("World");
Set set = new HashSet(myList);
```

myList has three elements, two of which are duplicates. Since **Set** does not allow duplicate elements, only one of the duplicates will be accepted. The resulting **Set** in the above code only has two elements.

Map and HashMap

A **Map** holds key to value mappings. There cannot be duplicate keys in a **Map** and each key maps to at most one value.

To add a key/value pair to a **Map**, you use the **put** method. Its signature is as follows:

```
public void put(java.lang.Object key, java.lang.Object value)
```

Note that both the key and the value cannot be a primitive. However, the following code that passes primitives to both the key and the value is legal because boxing is performed before the **put** method is invoked.

```
map.put(1, 3000);
```

Alternatively, you can use **putAll** and pass a **Map**.

```
public void putAll(Map map)
```

You can remove a mapping by passing the key to the **remove** method.

```
public void remove(java.lang.Object key)
```

To remove all mappings, use **clear**. To find out the number of mappings, use the **size** method. In addition, **isEmpty** returns **true** if the size is zero.

To obtain a value, you can pass a key to the **get** method:

```
public java.lang.Object get(java.lang.Object key)
```

In addition to the methods discussed so far, there are three no-argument methods that provide a view to a **Map**.

- **keySet**. Returns a **Set** containing all keys in the **Map**.
- **values**. Returns a **Collection** containing all values in the **Map**.
- **entrySet**. Returns a **Set** containing **Map.Entry** objects, each of which represents a key/value pair. The **Map.Entry** interface provides the **getKey** method that returns the key part and the **getValue** method that returns the value.

There are several implementations of **Map** in the **java.util** package. The most commonly used are **HashMap** and **Hashtable**. **HashMap** is unsynchronized and **Hashtable** is synchronized. Therefore, **HashMap** is the faster one between the two.

The following code demonstrates the use of **Map** and **HashMap**.

```
Map map = new HashMap();
map.put("1", "one");
map.put("2", "two");

System.out.println(map.size()); //print 2
System.out.println(map.get("1")); //print "one"

Set keys = map.keySet();
// print the keys
for (Object object : keys) {
    System.out.println(object);
}
```

Making Objects Comparable and SortableIn real life, objects are comparable. Dad's car is more expensive than Mom's, this dictionary is thicker than those books, Granny is older than Auntie Mollie (well, yeah, living objects too are comparable), and so forth. In object-oriented programming there are often needs to compare instances of the same class. And, if instances are comparable, they can be sorted. As an example, given two **Employee** objects, you may want to know which one has been staying in the organization longer. Or, in a search method for **Person** instances whose first name is Larry, you may want to display search results sorted

by age in descending or ascending order. You can make objects comparable by implementing the **java.lang.Comparable** and **java.util.Comparator** interfaces. You'll learn to use these interfaces in the following sections.

Using java.lang.Comparable

The **java.util.Arrays** class provides the static method **sort** that can sort any array of objects. Here is its signature.

```
public static void sort(java.lang.Object[] a)
```

Because all Java classes derive from **java.lang.Object**, all Java objects are a type of **java.lang.Object**. This means you can pass an array of any objects to the **sort** method.

However, how does the **sort** method know how to sort arbitrary objects? It's easy to sort numbers or strings, but how do you sort an array of **Elephant** objects, for example?

First, examine the **Elephant** class in Listing 11.2.

Listing 11.2: The Elephant class

```
public class Elephant {
    public float weight;
    public int age;
    public float tuskLength; // in centimeters
}
```

Since you are the author of the **Elephant** class, you get to decide how you want **Elephant** objects to be sorted. Let's say you want to sort them by their weights and ages. Now, how do you tell **Arrays.sort** of your decision?

Arrays.sort has defined a contract between itself and objects that needs its sorting service. The contract takes the form of the **java.lang.Comparable** interface. (See Listing 11.3)

Listing 11.3: The java.lang.Comparable method

```
package java.lang;
public interface Comparable {
    public int compareTo(Object obj);
}
```

Any class that needs to support sorting by **Arrays.sort** must implement the **Comparable** interface. In Listing 11.3, the argument *obj* in the **compareTo** method refers to the object being compared with this object. The code implementation for this method in the implementing class must return a positive number if this object is greater than the argument object, zero if both are equal, and a negative number if this object is less than the argument object.

Listing 11.4 presents a modified **Elephant** class that implements **Comparable**.

Listing 11.4: The Elephant class implementing Comparable

```
package app11;
public class Elephant implements Comparable {
    public float weight;
    public int age;
    public float tuskLength;
    public int compareTo(Object obj) {
        Elephant anotherElephant = (Elephant) obj;
```

```
        if (this.weight > anotherElephant.weight) {
            return 1;
        } else if (this.weight < anotherElephant.weight) {
            return -1;
        } else {
            // both elephants have the same weight, now
            // compare their age
            return (this.age - anotherElephant.age);
        }
    }
}
```

Now that **Elephant** implements **Comparable**, you can use **Arrays.sort** to sort an array of **Elephant** objects. The **sort** method will treat each **Elephant** object as a **Comparable** object (because **Elephant** implements **Comparable**, an **Elephant** object can be considered a type of **Comparable**) and call the **compareTo** method on the object. The **sort** method does this repeatedly until the **Elephant** objects in the array have been organized correctly by their weights and ages. Listing 11.5 provides a class that tests the **sort** method on **Elephant** objects.

Listing 11.5: Sorting elephants

```
package app11;
import java.util.Arrays;

public class ElephantTest {
    public static void main(String[] args) {
        Elephant elephant1 = new Elephant();
        elephant1.weight = 100.12F;
        elephant1.age = 20;
        Elephant elephant2 = new Elephant();
        elephant2.weight = 120.12F;
        elephant2.age = 20;
        Elephant elephant3 = new Elephant();
        elephant3.weight = 100.12F;
        elephant3.age = 25;

        Elephant[] elephants = new Elephant[3];
        elephants[0] = elephant1;
        elephants[1] = elephant2;
        elephants[2] = elephant3;

        System.out.println("Before sorting");
        for (Elephant elephant : elephants) {
            System.out.println(elephant.weight + ":" +
                    elephant.age);
        }
        Arrays.sort(elephants);
        System.out.println("After sorting");
        for (Elephant elephant : elephants) {
            System.out.println(elephant.weight + ":" +
                    elephant.age);
        }
    }
}
```

If you run the **ElephantTest** class, you'll see this on your console.

```
Before sorting
100.12:20
120.12:20
100.12:25
After sorting
100.12:20
100.12:25
120.12:20
```

Classes such as **java.lang.String**, **java.util.Date**, and primitive wrapper classes all implement **java.lang.Comparable**.

Using Comparable and Comparator

Implementing **java.lang.Comparable** enables you to define one way to compare instances of your class. However, objects sometimes are comparable in more ways. For example, two **Person** objects may need to be compared by age or by last/first name. In cases like this, you need to create a **Comparator** that defines how two objects should be compared. To make objects comparable in two ways, you need two comparators.

To create a comparator, you write a class that implements the **Comparator** interface. You then provide the implementation for its **compare** method. This method has the following signature.

```
public int compare(java.lang.Object o1, java.lang.Object o2)
```

compare returns zero if *o1* and *o2* are equal, a negative integer if *o1* is less than *o2*, and a positive integer if *o1* is greater than *o2*.

As an example, the **Person** class in Listing 11.6 implements **Comparable**. Listings 11.7 and 11.8 present two comparators of **Person** objects (by last name and by first name), and Listing 11.9 offers the class that instantiates the **Person** class and the two comparators.

Listing 11.6: The Person class implementing Comparable.

```
package app11;

public class Person implements Comparable {
    private String firstName;
    private String lastName;
    private int age;
    public String getFirstName() {
        return firstName;
    }
    public void setFirstName(String firstName) {
        this.firstName = firstName;
    }
    public String getLastName() {
        return lastName;
    }
    public void setLastName(String lastName) {
        this.lastName = lastName;
    }
    public int getAge() {
        return age;
    }
    public void setAge(int age) {
```

```
        this.age = age;
    }
    public int compareTo(Object anotherPerson)
            throws ClassCastException {
        if (!(anotherPerson instanceof Person)) {
            throw new ClassCastException(
                    "A Person object expected.");
        }
        int anotherPersonAge = ((Person) anotherPerson).getAge();
        return this.age - anotherPersonAge;
    }
}
```

Listing 11.7: The LastNameComparator class

```
package app11;
import java.util.Comparator;

public class LastNameComparator implements Comparator {
    public int compare(Object person, Object anotherPerson) {
        String lastName1 = ((Person)
       person).getLastName().toUpperCase();
        String firstName1 =
                ((Person) person).getFirstName().toUpperCase();
        String lastName2 = ((Person)
                anotherPerson).getLastName().toUpperCase();
        String firstName2 = ((Person) anotherPerson).getFirstName()
                .toUpperCase();
        if (lastName1.equals(lastName2)) {
            return firstName1.compareTo(firstName2);
        } else {
            return lastName1.compareTo(lastName2);
        }
    }
}
```

Listing 11.8: The FirstNameComparator class

```
package app11;
import java.util.Comparator;

public class FirstNameComparator implements Comparator {
    public int compare(Object person, Object anotherPerson) {
        String lastName1 = ((Person)
       person).getLastName().toUpperCase();
        String firstName1 = ((Person)
       person).getFirstName().toUpperCase();
        String lastName2 = ((Person)
       anotherPerson).getLastName().toUpperCase();
        String firstName2 = ((Person) anotherPerson).getFirstName()
                .toUpperCase();
        if (firstName1.equals(firstName2)) {
            return lastName1.compareTo(lastName2);
        } else {
            return firstName1.compareTo(firstName2);
        }
    }
```

```
}
```

Listing 11.9: The PersonTest class

```
package app11;
import java.util.Arrays;

public class PersonTest {
    public static void main(String[] args) {
        Person[] persons = new Person[4];
        persons[0] = new Person();
        persons[0].setFirstName("Elvis");
        persons[0].setLastName("Goodyear");
        persons[0].setAge(56);

        persons[1] = new Person();
        persons[1].setFirstName("Stanley");
        persons[1].setLastName("Clark");
        persons[1].setAge(8);

        persons[2] = new Person();
        persons[2].setFirstName("Jane");
        persons[2].setLastName("Graff");
        persons[2].setAge(16);

        persons[3] = new Person();
        persons[3].setFirstName("Nancy");
        persons[3].setLastName("Goodyear");
        persons[3].setAge(69);

        System.out.println("Natural Order");
        for (int i = 0; i < 4; i++) {
            Person person = persons[i];
            String lastName = person.getLastName();
            String firstName = person.getFirstName();
            int age = person.getAge();
            System.out.println(lastName + ", " + firstName +
                    ". Age:" + age);
        }

        Arrays.sort(persons, new LastNameComparator());
        System.out.println();
        System.out.println("Sorted by last name");
        for (int i = 0; i < 4; i++) {
            Person person = persons[i];
            String lastName = person.getLastName();
            String firstName = person.getFirstName();
            int age = person.getAge();
            System.out.println(lastName + ", " + firstName +
                    ". Age:" + age);
        }

        Arrays.sort(persons, new FirstNameComparator());
        System.out.println();
        System.out.println("Sorted by first name");
        for (int i = 0; i < 4; i++) {
```

```
            Person person = persons[i];
            String lastName = person.getLastName();
            String firstName = person.getFirstName();
            int age = person.getAge();
            System.out.println(lastName + ", " + firstName +
                    ". Age:" + age);
        }

        Arrays.sort(persons);
        System.out.println();
        System.out.println("Sorted by age");
        for (int i = 0; i < 4; i++) {
            Person person = persons[i];
            String lastName = person.getLastName();
            String firstName = person.getFirstName();
            int age = person.getAge();
            System.out.println(lastName + ", " + firstName +
                    ". Age:" + age);
        }
    }
}
```

If you run the **PersonTest** class, you will get the following result.

```
Natural Order
Goodyear, Elvis. Age:56
Clark, Stanley. Age:8
Graff, Jane. Age:16
Goodyear, Nancy. Age:69

Sorted by last name
Clark, Stanley. Age:8
Goodyear, Elvis. Age:56
Goodyear, Nancy. Age:69
Graff, Jane. Age:16

Sorted by first name
Goodyear, Elvis. Age:56
Graff, Jane. Age:16
Goodyear, Nancy. Age:69
Clark, Stanley. Age:8

Sorted by age
Clark, Stanley. Age:8
Graff, Jane. Age:16
Goodyear, Elvis. Age:56
Goodyear, Nancy. Age:69
```

Summary

In this chapter you have learned to use the core types in the Collections Framework. The main type is the **java.util.Collection** interface, which has three direct subinterfaces: **List**, **Set**, and **Queue**. Each subtype comes with several implementations. There are synchronized implementations and there are

unsynchronized ones. The latter are usually preferable because they are faster.

There is also a **Map** interface for storing key/value pairs. Two main implementations of **Map** are **HashMap** and **Hashtable**. **HashMap** is faster than **Hashtable** because the former is unsynchronized and the latter is synchronized.

Lastly, you have also learned the **java.lang.Comparable** and **java.util.Comparator** interfaces. Both are important because they can make objects comparable and sortable.

Questions

1. Name at least seven types of the Collections Framework.
2. What is the different between **ArrayList** and **Vector**?
3. Why is **Comparator** more powerful than **Comparable**?

Chapter 12
Generics

With generics you can write a parameterized type and create instances of it by passing a reference type or reference types. The objects will then be restricted to the type(s). For example, the **java.util.List** interface is generic. If you create a **List** by passing **java.lang.String**, you'll get a **List** that will only store **String**s; In addition to parameterized types, generics support parameterized methods too.

The first benefit of generics is stricter type checking at compile time. This is most apparent in the Collections Framework. In addition, generics eliminate most type castings you had to perform when working with the Collections Framework.

This chapter teaches you how to use and write generic types. It starts with the section "Life without Generics" to remind us what we missed in earlier versions of JDK's. Then, it presents some examples of generic types. After a discussion of the syntax, this chapter concludes with a section that explains how to write generic types.

Life without Generics

All Java classes derive from **java.lang.Object**, which means all Java objects can be cast to **Object**. Because of this, in pre-5 JDK's, many methods in the Collections Framework accept an **Object** argument. This way, the collections become general-purpose utility types that can hold objects of any type. This imposes unpleasant consequences.

For example, the **add** method of **List** in pre-5 JDK's takes an **Object** argument:

```
public boolean add(java.lang.Object element)
```

As a result, you can pass an object of any type to **add**. The use of **Object** is by design. Otherwise, it could only work with a specific type of objects and there would then have to be different **List** types, e.g. **StringList**, **EmployeeList**, **AddressList**, etc.

The use of **Object** in **add** is fine, but consider the **get** method, that returns an element in a **List** instance. Here is its signature prior to Java 5.

```
public java.lang.Object get(int index)
        throws IndexOutOfBoundsException
```

get returns an **Object**. Here is where the unpleasant consequences start to kick in. Suppose you have stored two **String** objects in a **List**:

```
List stringList1 = new ArrayList();
stringList1.add("Java 5 and later");
stringList1.add("with generics");
```

When retrieving a member from **stringList1**, you get an instance of

java.lang.Object. In order to work with the original type of the member element, you must first downcast it to **String**.

```
String s1 = (String) stringList1.get(0);
```

With generic types, you can forget about type casting when retrieving objects from a **List**. And, there is more. Using the generic **List** interface, you can create specialized **List**s, like one that only accepts **String**s.

Introducing Generic Types

Like a method, a generic type can accept parameters. This is why a generic type is often called a parameterized type. Instead of passing primitives or object references in parentheses as with methods, you pass reference types in angle brackets to generic types.

Declaring a generic type is like declaring a non-generic one, except that you use angle brackets to enclose the list of type variables for the generic type.

```
MyType<typeVar1, typeVar2, ...>
```

For example, to declare a **java.util.List**, you would write

```
List<E> myList;
```

E is called a type variable, namely a variable that will be replaced by a type. The value substituting for a type variable will then be used as the argument type or the return type of a method or methods in the generic type. For the **List** interface, when an instance is created, *E* will be used as the argument type of **add** and other methods. *E* will also be used as the return type of **get** and other methods. Here are the signatures of **add** and **get**.

```
public boolean add<E o>

public E get(int index)
```

Note

A generic type that uses a type variable *E* allows you to pass *E* when declaring or instantiating the generic type. Additionally, if *E* is a class, you may also pass a subclass of *E*; if *E* is an interface, you may also pass a class that implements *E*.

If you pass **String** to a declaration of **List**, as in

```
List<String> myList;
```

the **add** method of the **List** instance referenced by **myList** will expect a **String** object as its argument and its **get** method will return a **String**. Because **get** returns a specific type of object, no downcasting is required.

Note

By convention, you use a single uppercase letter for type variable names.

To instantiate a generic type, you pass the same list of parameters as when declaring it. For instance, to create an **ArrayList** that works with **String**, you pass **String** in angle brackets.

```
List<String> myList = new ArrayList<String>();
```

The diamond language change in Java 7 allows explicit type arguments to constructors of parameterized classes, most notably collections, to be omitted in many situations. Therefore, the statement above can be written more concisely in Java 7 or later.

```
List<String> myList = new ArrayList<>();
```

In this case, the compiler will infer the arguments to the **ArrayList**.

As another example, **java.util.Map** is defined as

```
public interface Map<K, V>
```

K is used to denote the type of map keys and *V* the type of map values. The **put** and **values** methods have the following signatures:

```
V put(K key, V value)
Collection<V> values()
```

> **Note**
> A generic type must not be a direct or indirect child class of **java.lang.Throwable** because exceptions are thrown at runtime, and therefore it is not possible to check what type of exception that might be thrown at compile time.

As an example, Listing 12.1 compares **List** with and without generics.

Listing 12.1: Working with generic List

```
package app12;
import java.util.List;
import java.util.ArrayList;

public class GenericListTest {
    public static void main(String[] args) {
        // without generics
        List stringList1 = new ArrayList();
        stringList1.add("Java");
        stringList1.add("without generics");
        // cast to java.lang.String
        String s1 = (String) stringList1.get(0);
        System.out.println(s1.toUpperCase());

        // with generics and diamond
        List<String> stringList2 = new ArrayList<>();
        stringList2.add("Java");
        stringList2.add("with generics");
        // no type casting is necessary
        String s2 = stringList2.get(0);
        System.out.println(s2.toUpperCase());
    }
}
```

In Listing 12.1, **stringList2** is a generic **List**. The declaration **List<String>** tells the compiler that this instance of **List** can only store **String**s. When retrieving member elements of the **List**, no downcasting is necessary because its **get** method returns the intended type, namely **String**.

> **Note**
> With generic types, type checking is done at compile time.

What's interesting here is the fact that a generic type is itself a type and can be used as a type variable. For example, if you want your **List** to store lists of strings, you can declare the **List** by passing **List<String>** as its type variable, as in

```
List<List<String>> myListOfListsOfStrings;
```

To retrieve the first string from the first list in **myList**, you would write:

```
String s = myListOfListsOfStrings.get(0).get(0);
```

Listing 12.2 presents a class that uses a **List** that accepts a **List** of **String**s.

Listing 12.2: Working with List of Lists

```
package app12;
import java.util.ArrayList;
import java.util.List;
public class ListOfListsTest {
    public static void main(String[] args) {
        List<String> listOfStrings = new ArrayList<>();
        listOfStrings.add("Hello again");
        List<List<String>> listOfLists =
                new ArrayList<>();
        listOfLists.add(listOfStrings);
        String s = listOfLists.get(0).get(0);
        System.out.println(s); // prints "Hello again"
    }
}
```

Additionally, a generic type can accept more than one type variables. For example, the **java.util.Map** interface has two type variables. The first defines the type of its keys and the second the type of its values. Listing 12.3 presents an example that uses a generic **Map**.

Listing 12.3: Using the generic Map

```
package app12;
import java.util.HashMap;
import java.util.Map;
public class MapTest {
    public static void main(String[] args) {
        Map<String, String> map = new HashMap<>();
        map.put("key1", "value1");
        map.put("key2", "value2");
        String value1 = map.get("key1");
    }
}
```

In Listing 12.3, to retrieve a value indicated by **key1**, you do not need to perform type casting.

Using Generic Types without Type Parameters

Now that the collection types in Java have been made generic, what about legacy codes? Fortunately, they will still work in Java 5 or later because you can use generic types without type parameters. For example, you can still use **List** the old way, as demonstrated in Listing 12.1.

```
List stringList1 = new ArrayList();
stringList1.add("Java");
stringList1.add("without generics");
String s1 = (String) stringList1.get(0);
```

A generic type used without parameters is called a raw type. This means that code written for JDK 1.4 and earlier versions will continue working in Java 5 or later.

One thing to note, though, starting from Java 5 the Java compiler expects you

to use generic types with parameters. Otherwise, the compiler will issue warnings, thinking that you may have forgotten to define type variables with the generic type. For example, compiling the code in Listing 12.1 gave you the following warning because the first **List** was used as a raw type.

```
Note: app12/GenericListTest.java uses unchecked or unsafe
      operations.
Note: Recompile with -Xlint:unchecked for details.
```

You have these options at your disposal to get rid of the warnings when working with raw types:

- compile with the **–source 1.4** flag.
- use the **@SuppressWarnings("unchecked")** annotation (See Chapter 18, "Annotations")
- upgrade your code to use **List<Object>**. Instances of **List<Object>** can accept any type of object and behave like a raw type **List**. However, the compiler will not complain.

Warning

Raw types are available for backward compatibility. New development should shun them. It is possible that future versions of Java will not allow raw types.

Using the ? Wildcard

I mentioned that if you declare a **List<*aType*>**, the **List** works with instances of *aType* and you can store objects of one of these types:

- an instance of *aType*.
- an instance of a subclass of *aType*, if *aType* is a class
- an instance of a class implementing *aType* if *aType* is an interface.

However, note that a generic type is a Java type by itself, just like **java.lang.String** or **java.io.File**. Passing different lists of type variables to a generic type result in different types. For example, **list1** and **list2** below reference to different types of objects.

```
List<Object> list1 = new ArrayList<>();
List<String> list2 = new ArrayList<>();
```

list1 references a **List** of **java.lang.Object** instances and **list2** references a **List** of **String** objects. Even though **String** is a subclass of **Object**, **List<String>** has nothing to do with **List<Object>**. Therefore, passing a **List<String>** to a method that expects a **List<Object>** will raise a compile time error. Listing 12.4 shows this.

Listing 12.4: AllowedTypeTest.java

```
package app12;
import java.util.ArrayList;
import java.util.List;

public class AllowedTypeTest {
    public static void doIt(List<Object> l) {
    }
    public static void main(String[] args) {
        List<String> myList = new ArrayList<>();
```

```
        // this will generate a compile error
        doIt(myList);
    }
}
```

Listing 12.4 won't compile because you are passing the wrong type to the **doIt** method. **doIt** expects an instance of **List<Object>** and you are passing an instance of **List<String>**.

The solution to this problem is the **?** wildcard. **List<?>** means a list of objects of any type. Therefore, the **doIt** method should be changed to:

```
public static void doIt(List<?> l) {
}
```

There are circumstances where you want to use the wildcard. For example, if you have a **printList** method that prints the members of a **List**, you may want to make it accept a **List** of any type. Otherwise, you would end up writing many overloads of **printList**. Listing 12.5 shows the **printList** method that uses the **?** wildcard.

Listing 12.5: Using the ? wildcard

```
package app12;
import java.util.ArrayList;
import java.util.List;

public class WildCardTest {
    public static void printList(List<?> list) {
        for (Object element : list) {
            System.out.println(element);
        }
    }
    public static void main(String[] args) {
        List<String> list1 = new ArrayList<>();
        list1.add("Hello");
        list1.add("World");
        printList(list1);

        List<Integer> list2 = new ArrayList<>();
        list2.add(100);
        list2.add(200);
        printList(list2);
    }
}
```

The code in Listing 12.4 demonstrates that **List<?>** in the **printList** method means a **List** of any type.

Note, however, it is illegal to use the wildcard when declaring or creating a generic type, such as this.

```
List<?> myList = new ArrayList<?>(); // this is illegal
```

If you want to create a **List** that can accept any type of object, use **Object** as the type variable, as in the following line of code:

```
List<Object> myList = new ArrayList<>();
```

Using Bounded Wildcards in Methods

In the section "Using the ? Wildcard" above, you learned that passing different type variables to a generic type creates different Java types. In many cases, you might want a method that accepts a **List** of different types. For example, if you have a **getAverage** method that returns the average of numbers in a list, you may want the method to be able to work with a list of integers or a list of floats or a list of another number type. However, if you write **List<Number>** as the argument type to **getAverage**, you won't be able to pass a **List<Integer>** instance or a **List<Double>** instance because **List<Number>** is a different type from **List<Integer>** or **List<Double>**. You can use **List** as a raw type or use a wildcard, but this is depriving you of type safety checking at compile time because you could also pass a list of anything, such as an instance of **List<String>**. You could use **List<Number>**, but you must always pass a **List<Number>** to the method. This would make your method less useful because you work with **List<Integer>** or **List<Long>** probably more often than with **List<Number>**.

There is another rule to circumvent this restriction, i.e. by allowing you to define an upper bound of a type variable. This way, you can pass a type or its subtype. In the case of the **getAverage** method, you may be able to pass a **List<Number>** or a **List** of instances of a **Number** subclass, such as **List<Integer>** or **List<Float>**.

The syntax for using an upper bound is as follows:

```
GenericType<? extends upperBoundType>
```

For example, for the **getAverage** method, you would write:

```
List<? extends Number>
```

Listing 12.6 illustrates the use of such a bound.

Listing 12.6: Using a bounded wildcard

```
package app12;
import java.util.ArrayList;
import java.util.List;
public class BoundedWildcardTest {
    public static double getAverage(
            List<? extends Number> numberList) {
        double total = 0.0;
        for (Number number : numberList) {
            total += number.doubleValue();
        }
        return total/numberList.size();
    }

    public static void main(String[] args) {
        List<Integer> integerList = new ArrayList<>();
        integerList.add(3);
        integerList.add(30);
        integerList.add(300);
        System.out.println(getAverage(integerList)); // 111.0
        List<Double> doubleList = new ArrayList<>();
        doubleList.add(3.0);
        doubleList.add(33.0);
```

```
        System.out.println(getAverage(doubleList)); // 18.0
    }
}
```

Thanks to the upper bound, the **getAverage** method in Listing 12.6 allows you to pass a **List<Number>** or a **List** of instances of any subclass of **java.lang.Number**.

Lower Bounds

The **extends** keyword is used to define an upper bound of a type variable. Though useable only in very few applications, it is also possible to define a lower bound of a type variable, by using the **super** keyword. For example, using **List<? super Integer>** as the type to a method argument indicates that you can pass a **List<Integer>** or a **List** of objects whose class is a superclass of **java.lang.Integer**.

Writing Generic Types

Writing a generic type is not much different from writing other types, except for the fact that you declare a list of type variables that you intend to use somewhere in your class. These type variables come in angle brackets after the type name. For example, the **Point** class in Listing 12.7 is a generic class. A **Point** object represents a point in a coordinate system and has an X component (abscissa) and a Y component (ordinate). By making **Point** generic, you can specify the degree of accuracy of a **Point** instance. For example, if a **Point** object needs to be very accurate, you can pass **Double** as the type variable. Otherwise, **Integer** would suffice.

Listing 12.7: The generic Point class

```
package app12;
public class Point<T> {
    T x;
    T y;
    public Point(T x, T y) {
        this.x = x;
        this.y = y;
    }
    public T getX() {
        return x;
    }
    public T getY() {
        return y;
    }
    public void setX(T x) {
        this.x = x;
    }
    public void setY(T y) {
        this.y = y;
    }
}
```

In Listing 12.7, **T** is the type variable for the **Point** class. **T** is used as the return value of both **getX** and **getY** and as the argument type for **setX** and **setY**. In addition, the constructor also accepts two **T** type variables.

Using **Point** is just like using other generic types. For example, the following

code creates two **Point** objects, **point1** and **point2**. The former passes **Integer** as the type variable, the latter **Double**.

```
Point<Integer> point1 = new Point<>(4, 2);
point1.setX(7);
Point<Double> point2 = new Point<>(1.3, 2.6);
point2.setX(109.91);
```

Summary

Generics enable stricter type checking at compile time. Used especially in the Collections Framework, generics make two contributions. First, they add type checking to collection types at compile time, so that the type of objects that a collection can hold is restricted to the type passed to it. For example, you can now create an instance of **java.util.List** that hold strings and will not accept **Integer** or other types. Second, generics eliminate the need for type casting when retrieving an element from a collection.

Generic types can be used without type variables, i.e. as raw types. This provision makes it possible to run pre-Java 5 codes with JRE 5 or later. For new applications, you should not use raw types as future releases of Java may not support them.

In this chapter you have also learned that passing different type variables to a generic type results in different Java types. This is to say that **List<String>** is a different type from **List<Object>**. Even though **String** is a subclass of **java.lang.Object**, passing a **List<String>** to a method that expects a **List<Object>** generates a compile error. Methods that expect a **List** of anything can use the **?** wildcard. **List<?>** means a **List** of objects of any type.

Finally, you have seen that writing generic types is not that different from writing ordinary Java types. You just need to declare a list of type variables in angle brackets after the type name. You then use these type variables as the types of method return values or as the types of method arguments. By convention, a type variable name consists of a single uppercase letter.

Questions

1. What are the main benefits of generics?
2. What does “parameterized type” mean?

Chapter 13
Input Output

Input output (IO) is one of the most common operations performed by a computer program. Examples of IO operations are.

- Create and delete files
- Read from and write to a file or network socket.
- Serialize (or save) objects to persistent storage and retrieve saved objects.

Java provides the **java.io** package that contains types you can use to perform IO operations. Many failed IO operations may throw a **java.io.IOException**. They may also throw a **java.lang.SecurityException** if the failure is related to the lack of permission to perform a certain function.

Learning Java IO programming by iterating the members of **java.io** may not be the best approach, considering there are 12 interfaces, 50 classes, plus 16 exception classes. This chapter therefore presents topics based on functionality and select the most important members of the **java.io** package.

The **java.io.File** class is the first topic in this chapter. It provides methods for creating and deleting files and directories, checking the existence of a file, and so on.

However, the **File** class does not provide functionality to read and write a file's content. For this, you need a stream. Streams, which are discussed in the section "The Concept of Input/Output Streams," act like water pipes that facilitate the transmission of data. There are four types of streams: **InputStream**, **OutputStream**, **Reader**, and **Writer**. For better performance, there are also classes that wrap these streams and buffer the data being read or written. The names of these classes start with **Buffered** and the classes are **BufferedInputStream**, **BufferedOutputStream**, **BufferedReader**, and **BufferedWriter**.

Reading from and writing to a stream dictate that you do so sequentially, which means to read the second unit of data, you must read the first one first. The **java.io** package provides the **RandomAccessFile** for non-sequential operations. This class is the subsequent topic of discussion.

This chapter concludes with object serialization and deserialization, using the **ObjectInputStream** and **ObjectOutputStream** classes.

The File Class

A **File** object represents a file or directory pathname, and not a physical file or a directory. Therefore, the physical file/directory that a **File** object references does not need to exist. The main advantage of **File** is that it provides a system-independent way of representing a pathname. For example, in Unix/Linux you use a forward slash (/) to separate a directory from a subdirectory or a file. The

myNotes.txt file in the **tmp** directory can be written as **/tmp/myNotes.txt**. Windows, on the other hand, uses a backslash (****). Therefore, **C:\temp\myNotes.txt** specifies the **myNotes.txt** file in **C:\temp**. When writing Java code to manipulate files and directories, it would be tedious if you had to deal with different separators for different operating systems. Fortunately, the **File** class addresses this issue. For one, you can use its **separator** static field that returns a **String** to separate a directory and a subdirectory or a directory from a file. The value of **separator** depends on the operating system. In Unix/Linux, **separator** returns the string "/". In Windows, it returns "\". The **charSeparator** static field is similar to separator, but returns a **char**.

For instance, if you have a path to a directory named **parent** and a file called **filename**, you can join them in a system-independent way by using this code:

```
parent + File.separator + filename
```

The result will be the correct path to the physical file, regardless the operating system your application is running on.

File Constructors

The **File** class provides several constructors. The simplest one has the following signature:

```
public File(java.lang.String pathname)
```

where *pathname* is either an absolute or relative path name. If *pathname* is null, a **java.lang.NullPointerException** is thrown. For example, you can pass an absolute path to a file or directory like this:

```
File file1 = new File("C:\\temp\\myNote.txt"); // in Windows
File file2 = new File("/tmp/myNote.txt");      // in Linux/Unix
```

Note that in Windows you use the backslash character as the file separator and since it is also the escape character in Java, you need two backslash characters when constructing a **File** in Windows. Using a forward slash as a file separator will also work in Windows, is commonly used and, in my opinion, is less awkward:

```
File file1 = new File("C:/temp/myNote.txt"); // in Windows
```

If you pass a relative path name to the constructor, the path is taken to be relative to the directory from which you run your application. For example, if you invoke the **java** program from **C:\workDir**, the variable **file3** below references a file or directory named **music** under **C:\workDir**.

```
File file3 = new File("music");
```

If you have a file located under a certain directory, you could use this constructor to refer to the file, concatenating the directory and the file using **File.separator**:

```
// userSelectedDir is a directory selected by the user at runtime
// and filename is a String containing the name of the file.
File myFile = new File(userSelectedDir + File.separator + filename);
```

However, it is shorter to use the second constructor whose signature is as follows.

```
public File(java.lang.String parent, java.lang.String child)
```

where *parent* is an absolute or relative path to a directory and *child* is the path to a file or a subdirectory. If *child* is an absolute path, then it will be converted into a relative pathname in a system-dependent way. If *parent* is an empty string, then *child* will be converted into an abstract pathname and resolved against a system-dependent default directory.

For example, the following code references a **data** directory under **userSelectedDir**.

```
File myFile = new File(userSelectedDir, data);
```

If *parent* is **null**, it is the same as passing child to the single-argument constructor: **File(*child*)**.

The third constructor is similar to the second one, except that the parent is a **File** object instead of a **String**:

```
public File(File parent, String.java.lang child)
```

If *parent* is **null**, it's the same as invoking the single-argument constructor.

The last constructor of **File** accepts a URI.

```
public File(java.net.URI uri)
```

You use this to create a **File** object by converting the given **file:** URI into an abstract pathname.

File Methods

The following are the more important methods of **File**.

`public boolean canRead()`
: Tests if the application can read the file referenced by this **File** object.

`public boolean canWrite()`
: Tests if the application can write to the file referenced by this **File** object.

`public boolean createNewFile() throws IOException`
: Creates a new empty file in the location and using the name denoted by this File object.

`public boolean delete()`
: Deletes the file or directory referenced by this **File** object.

`public boolean makeDir()`
: Creates the directory named by this **File** object.

`public boolean isFile()`
: Tests if this **File** object references a file.

`public boolean isDirectory()`
: Tests if this **File** object references a directory.

`public boolean exists()`
: Tests if the file or directory denoted by this **File** object exists.

`public File[] listFiles()`
: If this **File** object denotes a directory, this method returns an array of **File** objects referencing the subdirectories and files in the directory. Otherwise, returns **null**.

`public long getTotalSpace()`
: Returns the size, in bytes, of the partition referenced by this **File** object.

`public long getFreeSpace()`
: Returns the amount of free space, in bytes, in the partition referenced by this **File** object.

`public long getUsableSpace()`
: Returns the number of bytes available to this virtual machine on the partition referenced by this **File** object. The difference between

getUsableSpace and **getFreeSpace** is that the former takes into account restrictions imposed by the operating system, such as write permissions. The latter does not.

The Concept of Input/Output Streams

Java IO streams can be likened to water pipes. Just like water pipes connect city houses to a water reservoir, a Java stream connects Java code to a "data reservoir." In Java terminology, this "data reservoir" is called a sink and could be a file, a network socket, or memory. The good thing about streams is you employ a uniform way to transport data from and to different sinks, hence simplifying your code. You just need to construct the correct stream. For example, if the sink is a file you need a file stream.

Depending on the data direction, there are two types of streams, input stream and output stream. You use an input stream to read from a sink and an output stream to write to a sink. Because data can be classified into binary data and characters (human readable data), there are also two types of input streams and two types of output streams. These streams are represented by the following four abstract classes in the **java.io** package.

- **Reader**. A stream to read characters from a sink.
- **Writer**. A stream to write characters to a sink.
- **InputStream**. A stream to read binary data from a sink.
- **OutputStream**. A stream to write binary data to a sink.

As mentioned before, the benefit of streams is they define methods for data reading and writing that can be used regardless the data source or destination. To connect to a particular sink, you simply need to construct the correct implementation class. For example, the **Reader** class defines method for reading characters from a sink. If the sink is a file, you instantiate the **FileReader** class. In a similar token, the **FileWriter** class is the **Writer** class implementation that writes to a file, **FileInputStream** is used to read binary data from a file, and **FileOutputStream** represents a stream to write binary data to a file. A typical sequence of operations when working with a stream is as follows:

1. Create a stream. The resulting object is already open, there is no **open** method to call.
2. Perform reading/writing operations.
3. Close the stream by calling its **close** method.

These classes will be discussed in clear detail in the following sections.

Reading Binary Data

You use an **InputStream** to read binary data from a sink. **InputStream** is an abstract class with a number of implementation classes, as shown in Figure 13.1.

This section will discuss two child classes of **InputStream**, **FileInputStream** and **BufferedInputStream**. **FileInputStream** enables easy reading from a file and **BufferedInputStream** provides data buffering that improves performance. The **ObjectInputStream** class is used in object serialization and is discussed in the section, "Object Serialization" later in this chapter.

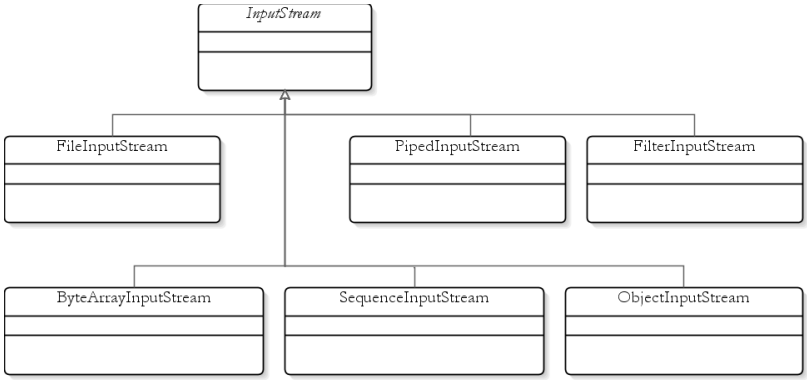

Figure 13.1: The hierarchy of InputStream

InputStream

At the core of the **InputStream** class are three **read** method overloads.

```
public int read()
public int read(byte[] data)
public int read(byte[] data, int offset, int length)
```

An **InputStream** employs an internal pointer that points to the starting position of the data to be read. Each of the **read** method overloads returns the number of bytes read or -1 if no data was read into the **InputStream**. This happens when the internal pointer has reached the end of file.

The no-argument **read** method is the easiest to use. It reads the next single byte from this **InputStream** and returns an **int**, which you can then cast to **byte**. Using this method to read a file, you use a **while** block that keeps looping until the **read** method returns -1:

```
int i = inputStream.read();
while (i != -1) {
    byte b = (byte) I;
    // do something with b
}
```

For speedier reading, you should use the second and third **read** method overloads, which require you to pass a byte array. The data will then be stored in this array. The size of array is a matter of compromise. If you assign a big number, the read operation will be faster because more bytes are read each time. However, this means allocating more memory space for the array. In practice, the array size should start from 1000 and up.

What if there are fewer bytes available than the size of the array? The **read** method overloads return the number of bytes read, so you always know which elements of your array contain valid data. For example, if you use an array of 1,000 bytes to read an **InputStream** and there are 1,500 bytes to read, you will need to invoke the **read** method twice. The first invocation gives you 1,000 bytes, the second 500 bytes.

You can choose to read fewer bytes than the array size using the three-argument **read** method overload:

```
public int read(byte[] data, int offset, int length)
```

This method overload reads *length* bytes into the byte array. The value of *offset*

determines the position of the first byte read in the array.

In addition to the **read** methods, there are also these methods:

`public int available() throws IOException`
: This method returns the number of bytes that can be read (or skipped over) without blocking.

`public long skip(long n) throws IOException`
: Skips over the specified number of bytes from this **InputStream**. The actual number of bytes skipped is returned and this may be smaller than the prescribed number.

`public void mark(int readLimit)`
: Remember the current position of the internal pointer in this **InputStream**. Calling reset afterwards will return the pointer to the marked position. The *readLimit* argument specifies the number of bytes to be read before the mark position get invalidated.

`public void reset()`
: Repositions the internal pointer in this **InputStream** to the marked position. Its signature is as follows.

`public void close()`
: Closes this **InputStream**. You should always call this method when you are done with this **InputStream** to release resources.

You will see an example of **InputStream** in the next section, "FileInputStream."

FileInputStream

The **FileInputStream** class is a subclass of **InputStream** and allows you to read binary data sequentially from a file. Its constructors allow you to pass either a **File** object or a file path. Here are the constructors.

```
public FileInputStream(String path)
public FileInputStream(File file)
```

As an example, the code in Listing 13.1 shows the **FileInputStreamTest** class that contains the **compareFiles** method. This method uses **FileInputStream** and methods from the **InputStream** class to compare files.

Listing 13.1: The compareFiles method that uses FileInputStream

```
package app13;
import java.io.File;
import java.io.FileInputStream;
import java.io.IOException;

public class FileInputStreamTest {
    public boolean compareFiles (String filePath1,
            String filePath2) {
        boolean areFilesIdentical = true;
        File file1 = new File(filePath1);
        File file2 = new File(filePath2);
        if (!file1.exists() || !file2.exists()) {
            System.out.println("One or both files do not exist");
            return false;
        }
        System.out.println("length:" + file1.length());
        if (file1.length()!= file2.length()) {
```

```
            System.out.println("lengths not equal");
            return false;
        }
        try {
            FileInputStream fis1 = new FileInputStream(file1);
            FileInputStream fis2 = new FileInputStream(file2);
            int i1 = fis1.read();
            int i2 = fis2.read();
            while (i1 != -1) {
                if (i1 != i2) {
                    areFilesIdentical = false;
                    break;
                }
                i1 = fis1.read();
                i2 = fis2.read();
            }
            fis1.close();
            fis2.close();
        } catch (IOException e) {
            System.out.println("IO exception");
            areFilesIdentical = false;
        }
        return areFilesIdentical;
    }

    public static void main(String[] args) {
        FileInputStreamTest test = new FileInputStreamTest();
        test.compareFiles("c: \\line2.bmp","c: \\line3.bmp");
    }
}
```

The **compareFiles** method returns the **boolean areFiledIdentical**, which is **true** only if the compared files are identical. The brain of the method is this block.

```
int i1 = fis1.read();
int i2 = fis2.read();
while (i1 != -1) {
    if (i1 != i2) {
        areFilesIdentical = false;
        break;
    }
    i1 = fis1.read();
    i2 = fis2.read();
}
```

It reads the next byte from the first **FileInputStream** to **i1** and the second **FileInputStream** to **i2** and compares **i1** with **i2**. It continues reading until **i1** and **i2** are different, in which case it will break from the **while** loop.

BufferedInputStream

For better performance you should wrap your **InputStream** with a **BufferedInputStream**. **BufferedInputStream** has two constructors that accept an **InputStream**:

```
public BufferedInputStream(InputStream in)
public BufferedInputStream(InputStream in, int bufferSize)
```

For example, the following code wraps a **FileInputStream** with a **BufferedInputStream**.

```
FileInputStream fis = new FileInputStream(aFile);
BufferedInputStream bis = new BufferedInputStream(fis);
```

Then, instead of calling the methods on *fis*, work with the **BufferedInputStream** *bis*.

Writing Binary Data

The **OutputStream** abstract class represents a stream for writing binary data to a sink. Its child classes are shown in Figure 13.2.

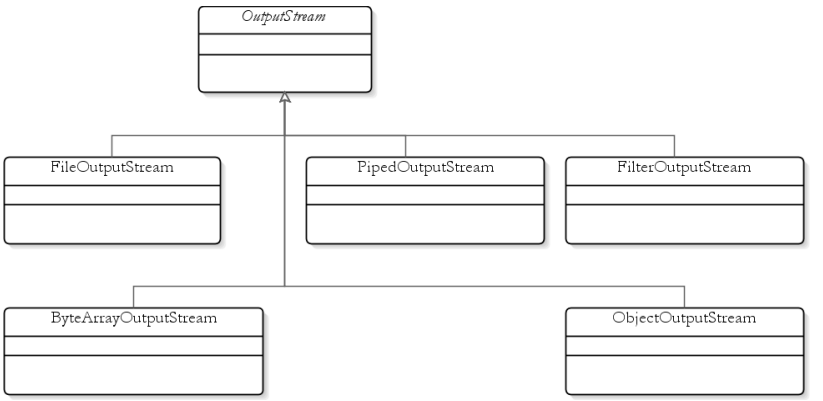

Figure 13.2: The implementation classes of OutputStream

This section discusses two implementation classes: **FileOutputStream** and **BufferedOutputStream**. **FileOutputStream** provides a convenient way to write to a file and **BufferedOutputStream** provides better performance. The **ObjectOutputStream** class plays an important role in object serialization and is discussed in the section, "Object Serialization" later in this chapter.

OutputStream

The OutputStream class defines three **write** method overloads, which are mirrors of the **read** method overloads in **InputStream**:

```
public void write(int b)
public void write(byte[] data)
public void write(byte[] data, int offset, int length)
```

The first overload writes the lowest 8 bits of the integer *b* to this **OutputStream**. The second writes the content of a byte array to this **OutputStream**. The third overload writes *length* bytes of the data starting at offset *offset*.

In addition, there are also the no-argument **close** and **flush** methods. **close** closes the **OutputStream** and **flush** forces any buffered content to be written out to the sink.

We'll look at an example in the next section, "FileOutputStream."

FileOutputStream

The **FileOutputStream** class is a subclass of **OutputStream**. You use **FileOutputStream** to write binary data to a file. The most important thing to note is its constructors. They allow you to construct a **FileOutputStream** object by passing a string containing a path name or a **File** object. You can also specify whether you want to append the output to an existing file.

Here are the signatures of some of its constructors.

```
public FileOutputStream(String path)
public FileOutputStream(String path, boolean append)
public FileOutputStream(File file)
public FileOutputStream(File file, boolean append)
```

With the first and third constructors, if a file by the specified name already exists, the file will be overwritten. To append to an existing file, pass **true** to the second or fourth constructor.

As an example, Listing 13.2 shows the **FileOutputStreamTest** class that contains the **copyFile** method.

Listing 13.2: The FileOutputStreamTest class

```
package app13;
import java.io.File;
import java.io.FileInputStream;
import java.io.FileOutputStream;
import java.io.IOException;

public class FileOutputStreamTest {
    public void copyFiles(String originPath, String destinationPath)
            throws IOException {
        File originFile = new File(originPath);
        File destinationFile = new File(destinationPath);
        if (!originFile.exists() || destinationFile.exists()) {
            throw new IOException(
                    "Origin file must exist and " +
                    "Destination file must not exist");
        }
        try {
            byte[] readData = new byte[1024];
            FileInputStream fis = new FileInputStream(originFile);
            FileOutputStream fos =
                    new FileOutputStream(destinationFile);
            int i = fis.read(readData);
            while (i != -1) {
                fos.write(readData, 0, i);
                i = fis.read(readData);
            }
            fis.close();
            fos.close();
        } catch (IOException e) {
            throw e;
        }
    }
```

```
    public static void main(String[] args) {
        FileOutputStreamTest test = new FileOutputStreamTest();
        try {
            test.copyFiles("c:\\temp\\line1.bmp",
       "c:\\temp\\line3.bmp");
            System.out.println("Copied Successfully");
        } catch (IOException e) {
        }
    }
}
```

This part of the **copyFile** method does the work.

```
byte[] readData = new byte[1024];
FileInputStream fis = new FileInputStream(originFile);
FileOutputStream fos = new FileOutputStream(destinationFile);
int i = fis.read(readData);
while (i != -1) {
    fos.write(readData, 0, i);
    i = fis.read(readData);
}
fis.close();
fos.close();
```

The **readData byte** array is used to store the data read from the **FileInputStream**. The number of bytes read is assigned to **i**. The code then calls the write method on the **FileOutputStream** object, passing the byte array and **i** as the third argument.

```
fos.write(readData, 0, i);
```

BufferedOutputStream

You should always wrap your **OutputStream** with a **BufferedOutputStream** for better performance. **BufferedOutputStream** has two constructors that accept an **OutputStream**.

```
public BufferedOutputStream(OutputStream out)
public BufferedOutputStream(OutputStream out, int bufferSize)
```

The first constructor uses the default buffer size, the second lets you decide. For example, you'll get better performance if you wrap a **FileOutputStream** like this:

```
FileOutputStream fos = new FileOutputStream(aFile);
BufferedOutputStream bos = new BufferedOutputStream(fos);
```

Writing Text (Characters)

The abstract class **Writer** defines a stream used for writing characters. Figure 13.3 shows the implementation classes of **Writer**.

OutputStreamWriter facilitates the translation of characters into byte streams using a given character set. The character set guarantees that any Unicode characters you write to this **OutputStreamWriter** will be translated into the correct byte representation. **FileWriter** is a child class of **OutputStreamWriter** that provides a convenient way to write characters to a file. However, **FileWriter** is not without flaws. When using **FileWriter** you are forced to output characters using the computer's encoding, which means characters outside the current

character set will not be translated correctly into bytes. A better alternative to **FileWriter** is **PrintWriter**.

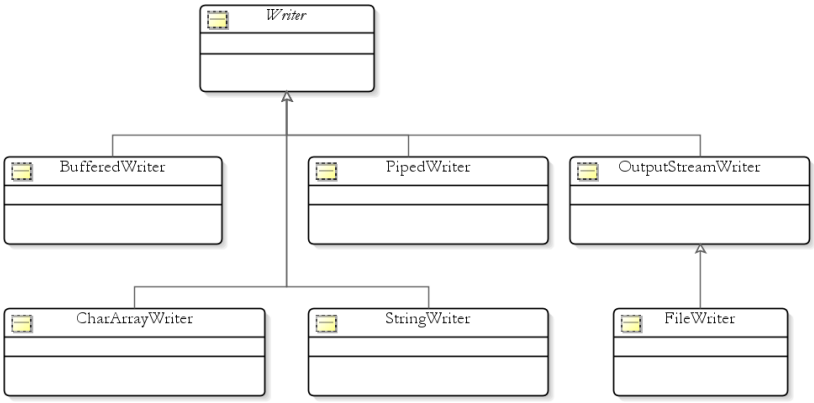

Figure 13.3: The subclasses of Writer

This section discusses the **BufferedWriter** class that buffers characters written to this **Writer** for better performance.

Writer

This class is similar to the **OutputStream** class, except that **Writer** deals with characters instead of bytes. Like **OutputStream**, the **Writer** class has three **write** method overloads:

```
public void write(int b)
public void write(char[] text)
public void write(char[] text, int offset, int length)
```

However, when working with text or characters, you ordinarily use strings. Therefore, there are two other overloads of the **write** method that accept a **String** object.

```
public void write(String text)
public void write(String text, int offset, int length)
```

The last **write** method overload allows you to pass a String and write part of the **String** to this **Writer**.

You will learn to use several implementations of **Writer** in the following subsections.

OutputStreamWriter

An **OutputStreamWriter** is a bridge from character streams to byte streams: Characters written to an **OutputStreamWriter** are encoded into bytes using a specified character set. The latter is an important element of **OutputStreamWriter** because it enables the correct translations of Unicode characters into byte representation.

Note

The **System.getProperty("file.encoding")** method returns the default encoding of your computer.

The **OutputStreamWriter** class has four constructors:

```
public OutputStreamWriter(OutputStream out)
```

```
public OutputStreamWriter(OutputStream out,
  java.nio.charset.Charset cs)
public OutputStreamWriter(OutputStream out,
  java.nio.charset.CharsetEncoder enc)
public OutputStreamWriter(OutputStream out, String encoding)
```

All the constructors accept an **OutputStream**, to which bytes resulting from the translation of characters written to this **OutputStreamWriter** will be written. Therefore, if you want to write to a file, you can pass a **FileOutputStream** to the constructor.

The first constructor creates an instance that uses the default encoding, but the others allow you to pass the encoding that will be used when translating character streams into byte streams. The **OutputStreamWriter** class adds the **getEncoding** method that will return the name of the encoding used in this **OutputStreamWriter** as a **String**.

The **java.nio.charset.Charset** class, an argument to the second constructor, is not discussed in this book, so I will show an example that uses the last constructor. This example is given in Listing 13.3..

Listing 13.3: Using OutputStreamWriter

```
package app13;
import java.io.File;
import java.io.FileOutputStream;
import java.io.IOException;
import java.io.OutputStreamWriter;

public class OutputStreamWriterTest {
    public static void main(String[] args) {
        try {
            char[] chars = new char[2];
            chars[0] = '\u4F60'; // representing ?
            chars[1] = '\u597D'; // representing ?;
            String encoding = "GB18030";
            File textFile = new File("C:\\temp\\myFile.txt");
            OutputStreamWriter writer = new OutputStreamWriter(
                    new FileOutputStream(textFile), encoding);
            writer.write(chars);
            writer.close();
        } catch (IOException e) {
            System.out.println(e.toString());
        }
    }
}
```

The code in Listing 13.3 creates an **OutputStreamWriter** based on a **FileOutputStream** that writes to **C:\temp\myFile.txt** on Windows. Therefore, if you are using Linux/Unix you need to change the value of **textFile**. The use of an absolute path is intentional since most readers find it easier to find if they want to open the file. The **OutputStreamWriter** uses the GB18030 encoding, the encoding the Chinese government requires all its software suppliers to use in their applications. This encoding defines the character set for Simplified Chinese characters. The use of an encoding other than English is good for showing how encoding works.

The code in Listing 13.3 passes two Chinese characters: 你 (represented by the Unicode 4F60) and 好 (Unicode 597D). 你好 means ‘How are you?’ in Chinese.

When executed, the **OutputStreamWriterTest** class will create the **myFile.txt** file. It is 4 bytes long. You can open it and see the Chinese characters. For the characters to be displayed correctly, you need to have the Chinese font installed in your computer.

FileWriter

FileWriter provides a convenient way of writing characters to a file. Using **FileWriter** is fine as long as you are not trying to write characters belonging to other character sets. In other words, if your computer's default language is English, writing Korean characters using **FileWriter** will not work. Unfortunately, there is no way you can change the encoding of a **FileWriter**.

FileWriter has constructors that allow you to construct an instance from a **File**, a file path, or a **FileDescriptor**. You also have the option to append to an existing file or to create a new file. Here are the constructors.

```
public FileWriter(File file)
public FileWriter(File file, boolean append)
public FileWriter(String path)
public FileWriter(String path, boolean append)
public FileWriter(FileDescriptor fileDescriptor)
```

For example, you can construct a **Writer** that writes to a file easily using **FileWriter**:

```
FileWriter writer = new FileWriter("myFile.txt");
writer.write(chars);
```

PrintWriter

PrintWriter is a better alternative to **OutputStreamWriter** and **FileWriter**. Like **OutputStreamWriter**, **PrintWriter** lets you choose an encoding by passing the encoding information to one of its constructors. Here are some of its constructors:

```
public PrintWriter(File file)
public PrintWriter(File file, String characterSet)
public PrintWriter(String filepath)
public PrintWriter(String filepath, String ccharacterSet)
public PrintWriter(OutputStream out)
public PrintWriter(Writer out)
```

For example, using the first, second, third, and fourth constructors, you can create a **PrintWriter** that writes to a file. The two-argument constructors let you pass the name of the character set to use, so you can write any Unicode characters. In addition, you can construct a **PrintWriter** object by passing an **OutputStream** or another **Writer** object.

Also, it is easier to construct a **PrintWriter** than an **OutputStreamWriter**. For example, the following line of code

```
OutputStreamWriter writer = new OutputStreamWriter(
    new FileOutputStream(filePath), encoding);
```

can be replaced by this shorter one.

```
PrintWriter writer = new PrintWriter(filePath, encoding);
```

PrintWriter is more convenient to work with than **OutputStreamWriter** because the former adds nine **print** method overloads that allow you to output any type of Java primitives and objects. Here are the method overloads:

```
public void print(boolean b)
public void print(char c)
public void print(char[] s)
public void print(double d)
public void print(float f)
public void print(int i)
public void print(long l)
public void print(Object object)
public void print(String string)
```

There are also nine **println** method overloads, which are the same as the **print** method overloads, except that they print a new line character after printing the argument.

In addition, there are two **format** method overloads that enable you to print according to a print format. This method was covered in Chapter 5, "Core Classes."

Listing 13.4 presents an example of **PrintWriter**.

Listing 13.4: Using PrintWriter

```
package app13;
import java.io.IOException;
import java.io.PrintWriter;

public class PrintWriterTest {
    public static void main(String[] args) {
        try {
            PrintWriter pw = new
                    PrintWriter("c:\\temp\\printWriterOutput.txt");
            pw.println("PrintWriter is easy to use.");
            pw.println(1234);
            pw.close();
        } catch (IOException e) {
        }
    }
}
```

The nice thing about writing using a **PrinterWriter** is, when you open the resulting file, everything is human-readable. The file created by the preceding example says:

```
PrinterWriter is easy to use.
1234
```

BufferedWriter

Always wrap your **Writer** with a **BufferedWriter** for better performance. **BufferedWriter** has the following constructors that allow you to pass a **Writer** object.

```
public BufferedWriter(Writer writer)
public BufferedWriter(Writer writer, in bufferSize)
```

The first constructor creates a **BufferedWriter** with the default buffer size (the documentation does not say how big). The second one lets you choose the buffer size.

Therefore, if you are working with a **FileWriter**, you'll get better performance

if you wrap it with a **BufferedWriter**:

```
FileWriter fw = new FileWriter(aFile);
BufferedWriter bw = new BufferedWriter(fw);
```

When working with **PrintWriter** (you'll be working mostly with this when you need to output characters to a stream), you cannot wrap it in a similar fashion, such as:

```
PrintWriter pw = new PrintWriter(aFile);
BufferedWriter bw = new BufferedWriter(pw);
```

Because then you would not be able to use the methods of the **PrintWriter**. Instead, wrap the **Writer** that is passed to a **PrintWriter**.

```
FileWriter fw = new FileWriter(aFile);
PrintWriter pw = new PrintWriter(new BufferedWriter(fw));
```

Reading Text (Characters)

You use the **Reader** class to read text (characters, i.e. human readable data). The hierarchy of this class is shown in Figure 13.4.

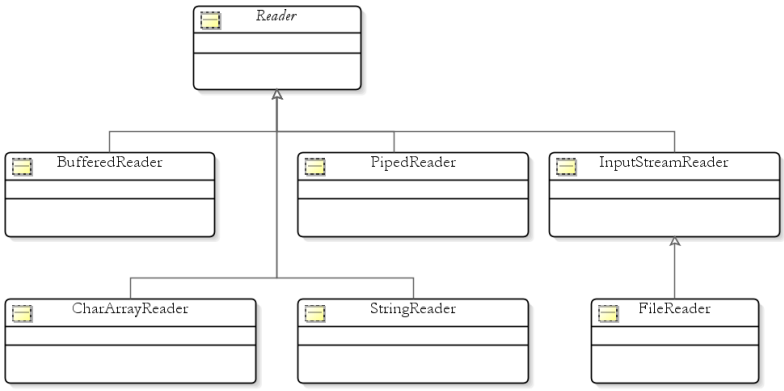

Figure 13.4: Reader and its descendants

This section discusses several child classes of **Reader**.

> ## Note
> The two more popular implementation classes of **Reader** are **InputStreamReader** and **BufferedReader**.

Reader

Reader is an abstract class that represents an input stream for reading characters. It is similar to **InputStream** except that **Reader** objects deal with characters and not bytes. The **Reader** class has three **read** method overloads that are similar to the **read** methods in **InputStream**:

```
public int read()
public int read(char[] data)
public int read(char[] data, int offset, int length)
```

These method overloads allow you to read a single character or multiple characters that will be stored in a char array. Additionally, **Reader** has the fourth read method

that enables you to read characters into a **java.nio.CharBuffer**.

```
public int read(java.nio.CharBuffer target)
```

In addition, **Reader** provides the following methods that are similar to those in **InputStream**: **close**, **mark**, **reset**, and **skip**.

InputStreamReader

An **InputStreamReader** reads bytes and translates them into characters using the specified character set. Therefore, **InputStreamReader** is ideal for reading from the output of an **OutputStreamWriter** or a **PrintWriter**. The key is you must know the encoding used when writing the characters to correctly read them back.

The **InputStreamReader** class has four constructors, all of which require you to pass an **InputStream**.

```
public InputStreamReader(InputStream in)
public InputStreamReader(InputStream in,
        java.nio.charset.Charset cs)
public InputStreamReader(InputStream in,
        java.nio.charset.CharsetDecoder, dec)
public InputStreamReader(InputStream in, String charsetName)
```

For instance, to create an **InputStreamReader** that reads from a file, you can pass a **FileInputStream** to its constructor.

```
InputStreamReader reader = new InputStreamReader(
        new FileInputStream(filePath), charSet);
```

Listing 13.5 presents the **InputStreamReaderTest** class that uses a **PrintWriter** to write two Chinese characters and read them back.

Listing 13.5: Using InputStreamReader

```
package app13;

import java.io.File;
import java.io.FileInputStream;
import java.io.IOException;
import java.io.InputStreamReader;
import java.io.PrintWriter;

public class InputStreamReaderTest {
    public static void main(String[] args) {
        try {
            char[] chars = new char[2];
            chars[0] = '\u4F60'; // representing ?
            chars[1] = '\u597D'; // representing ?;
            String encoding = "GB18030";
            File textFile = new File("C:\\temp\\myFile.txt");
            PrintWriter writer = new PrintWriter(textFile,
                    encoding);
            writer.write(chars);
            writer.close();

            // read back
            InputStreamReader reader = new InputStreamReader(
                    new FileInputStream(textFile), encoding);
            char[] chars2 = new char[2];
```

```
            reader.read(chars2);
            System.out.print(chars2[0]);
            System.out.print(chars2[1]);
            reader.close();
        } catch (IOException e) {
            System.out.println(e.toString());
        }
    }
}
```

FileReader

A subclass of **InputStreamReader**, **FileReader** is a convenient class to read characters from a file. However, like **FileWriter**, it lacks the ability to use an encoding other than the default one. **FileReader** cannot read the characters correctly if the encoding used for writing the file is different than the computer's current encoding.

To construct a **FileReader** object, use one of the following constructors.

```
public FileReader(File file)
public FileReader(FileDescriptor fileDescriptor)
public FileReader(String filePath)
```

BufferedReader

BufferedReader is good for two things:

1. Wraps another **Reader** and provides a buffer that will generally improve performance.
2. Provides a **readLine** method to read a line of text.

The **readLine** method has the following signature:

```
public java.lang.String readLine() throws IOException
```

It returns a line of text from this **Reader** or null if the end of the stream has been reached.

Using **BufferedReader** is very easy. For example, the following lines of code wraps an **InputStreamReader** (which is also a reader) that uses a certain encoding.

```
InputStreamReader inputStreamReader = new InputStreamReader(new
        FileInputStream(textFile), encoding  );
BufferedReader bufferedReader = new
        BufferedReader(inputStreamReader);
```

The resulting **BufferedReader** still supports the encoding of the underlying **InputStreamReader**, but at the same time it supports buffering and provides the **readLine** method.

As another example, this snippet reads a text file and displays it line by line.

```
FileReader fr = new FileReader(aFile);
BufferedReader br = new BufferedReader(fr);
String line = br.readLine();
while (line != null) {
    System.out.println(line);
    line = br.readLine();
```

```
}
```

Also, prior to Java 5, you used a **BufferedReader** to read user input to the console. Listing 13.6 shows the **getUserInput** method that can take user input on the console.

Listing 13.6: The getUserInput method

```
public static String getUserInput() {
    BufferedReader br = new BufferedReader(
            new InputStreamReader(System.in));
    try {
        return br.readLine();
    } catch (IOException ioe) {
    }
    return null;
}
```

You can do this because **System.in** is of type **java.io.InputStream**.

> **Note**
> The **java.util.Scanner** class can be used to easily read user input. See Chapter 5, "Core Classes" for more detail.

Logging with PrintStream

By now you must be familiar with the print method of **System.out**. You use it especially for displaying messages and this helps you debug your code. However, by default **System.out** sends the message to the console, and this is not always preferable. For instance, if the amount of data displayed exceeds a certain lines, previous messages are no longer visible. Also, you might want to process the messages further, such as sending the messages via email.

The **PrintStream** class is an indirect subclass of **OutpuStream**. It has seven constructors:

```
public PrintStream(File file)
public PrintStream(File file, String characterSet)
public PrintStream(OutputStream out)
public PrintStream(OutputStream out, boolean autoFlush)
public PrintStream(OutputStream out, boolean autoFlush,
        String encoding)
public PrintStream(String filePath)
public PrintStream(String filePath, String characterSet)
```

You can construct a **PrintStream** by passing a **File**, an **OutputStream**, or a path to a file. Some of its constructors let you choose a character set to use.

PrintStream is very similar to **PrintWriter**. For example, both have nine **print** method overloads. Also, **PrintStream** has the **format** method similar to the **format** method of the String class. See Chapter 5, "Core Classes" for more information.

System.out is of type **java.io.PrintStream**. The **System** object lets you replace the default **PrintStream** by using the **setOut** method. Listing 13.7 presents an example that redirect the output of **System.out** to a file.

Listing 13.7: Redirecting System.out to a file

```
package app13;
```

```
import java.io.File;
import java.io.IOException;
import java.io.PrintStream;

public class PrintStreamTest {
    public static void main(String[] args) {
        File file = new File("C:\\temp\\debug.txt");
        try {
            PrintStream ps = new PrintStream(file);
            System.setOut(ps);
        } catch (IOException e) {
        }
        System.out.println("To File");
    }
}
```

Note
You can also replace the default **in** and **out** in the **System** object by using **setIn** and **setErr** methods.

RandomAccessFile

Using a stream to access a file dictates that the file is accessed sequentially, e.g. the first character must be read before the second, etc. Streams are ideal when the data comes in a sequential fashion, for example if the medium is a tape (long time ago when the disk had not been invented) or a network socket. Streams are good for most of your applications, however sometimes you need to access a file randomly and using a stream would not be fast enough. For example, you may want to change the 1000th byte of a file without having to read the first 999 bytes. For random access like this, **RandomAccessFile** is ideal to work with.

RandomAccessFile derives directly from **java.lang.Object** and can perform both read and write operations. When opening a file using a **RandomAccessFile**, you can choose whether to open it read-only or read-write. **RandomAccessFile** has two constructors:

```
public RandomAccessFile(File file, String mode)
        throws FileNotFoundException
public RandomAccessFile(String filePath, String mode)
        throws FileNotFoundException
```

The value of **mode** can be one of these:

- "r". Open for reading only.
- "rw". Open for reading and writing. If the file does not already exist, **RandomAccessFile** creates the file.
- "rws". Open for reading and writing and require that every update to the file's content and metadata be written synchronously.
- "rwd". Open for reading and writing and require that every update to the file's content (but not metadata) be written synchronously.

A **RandomAccessFile** employs an internal pointer that points to the next byte to read. When first created, a **RandomAccessFile** points to the first byte. You can change the pointer's position by invoking the **seek** method. Its signature is as follows.

```
public void seek(long position) throws IOException
```

This pointer is zero-based which means the first byte is indicated by index 0. You can pass a number greater than the file size without throwing an exception, but this will not change the size of the file.

In addition to **seek**, the **skipBytes** method moves the pointer by the specified number of bytes. Here is its signature.

```
public int skipBytes(int offset)
```

If skipping *offset* number of bytes will pass the end of file, the internal pointer will only move to as much as the end of file. The **skipBytes** method returns the actual number of bytes skipped.

For reading from a file, **RandomAccessFile** provides a number of methods, each for reading a different data type:

```
public boolean readBoolean() throws IOException
public byte readByte() throws IOException
public char readChar() throws IOException
public double readDouble() throws IOException
public int readInt() throws IOException
public long readLong() throws IOException
public short readShort() throws IOException
```

In addition, **RandomAccessFile** can also behave like a **Reader** because it provides the **readLine** method that reads a line of text:

```
public String readLine()
```

For faster reading, **RandomAccessFile** provides the **readFully** method overloads, that read data to a byte array:

```
public void readFully(byte[] data)
public void readFully(byte[] data, int offset, int length)
```

For writing, **RandomAccessFile** provides methods that mirror the methods for reading, such as **writeBoolean**, **writeByte**, etc. Plus, there are two **write** method overloads to write the content of a byte array:

```
public void write(byte[] data)
public void write(byte[] data, int offset, int length)
```

RandomAccessFile is suitable for accessing a file that has a fixed structure randomly. For instance, you might use **RandomAccessFile** as a simple database to store fixed-length elements of data.

As an example, the code in Listing 13.8 employs **RandomAccessFile** to store **int**s and changes the value of the third **int**. An **int** takes 4 bytes, therefore the fourth **int** is pointed by the index 3-1 * 4 (3-1 because it's zero-based)

Listing 13.8: Using RandomAccessFile

```
package app13;
import java.io.IOException;
import java.io.RandomAccessFile;

public class RandomAccessFileTest {
    public static void main(String[] args) {
        try {
            RandomAccessFile raf = new RandomAccessFile(
                    "c:/temp/RAFsample.txt", "rw");
            raf.writeInt(10);
            raf.writeInt(43);
```

```
            raf.writeInt(88);
            raf.writeInt(455);

            // change the 3rd integer from 88 to 99
            raf.seek((3 - 1) * 4);
            raf.writeInt(99);
            raf.seek(0); // go to the first integer
            int i = raf.readInt();
            while (i != -1) {
                System.out.println(i);
                i = raf.readInt();
            }
            raf.close();
        } catch (IOException e) {
            e.printStackTrace();
        }
    }
}
```

Object Serialization

You sometimes need to persist objects in memory into permanent storage so that the states of the objects can be retained and later retrieved. Java supports this through object serialization. To serialize objects, i.e. to save objects to permanent storage, you use an **ObjectOutputStream**. To deserialize objects, namely to retrieved saved objects, use **ObjectInputStream**. **ObjectOutputStream** is a subclass of **OutputStream** and **ObjectInputStream** is derived from **InputStream**.

The **ObjectOutputStream** class has one public constructor:

```
public ObjectOutputStream(OutputStream out)
```

Therefore, to serialize objects to a file, you can pass a **FileOutputStream** to the constructor. Once you have an **ObjectOutputStream**, you can serialize objects or primitives or the combination of both. The **ObjectOutputStream** class provides the **write*XXX*** methods for each individual type, where ***XXX*** denotes a type. Here is the list of the **write*XXX*** methods.

```
public void writeBoolean(boolean value)
public void writeByte(int value)
public void writeBytes(String value)
public void writeChar(int value)
public void writeChars(String value)
public void writeDouble(double value)
public void writeFloat(float value)
public void writeInt(int value)
public void writeLong(long value)
public void writeShort(short value)
public void writeObject(java.lang.Object value)
```

For objects to be serializable their classes must implement the **java.io.Serializable** interface. This interface has no method and is a marker interface. A marker interface is one that tells the JVM that an instance of an implementing class belongs to a certain type.

If a serialized object contains other objects, the contained objects' classes must also implement **Serializable** for the contained objects to be serializable.

The **ObjectInputStream** class has a public constructor:

```
public ObjectInputStream(InputStream in)
```

Therefore, to deserialize from a file, you can pass a **FileInputStream** to the constructor. The **ObjectInputStream** class has methods that are the opposites of the **write*XXX*** methods in **ObjectOutputStream**. They are as follows:

```
public boolean readBoolean()
public byte readByte()
public char readChar()
public double readDouble()
public float readFloat()
public int readInt()
public long readLong()
public short readShort()
public java.lang.Object readObject()
```

One important thing to note: object serialization is based on a last in first out method. When deserializing multiple primitives/objects, the objects that were serialized first must be deserialized last.

Listing 13.10 shows a class that serializes an **int** and a **Customer** object. Note that the **Customer** class, given in Listing 13.9, implements **Serializable**.

Listing 13.9: The Customer class

```
package app13;
import java.io.Serializable;

public class Customer implements Serializable {
    public int id;
    public String name;
    public String address;
    public Customer (int id, String name, String address) {
        this.id = id;
        this.name = name;
        this.address = address;
    }
}
```

Listing 13.10: Object serialization example

```
package app13;
import java.io.FileInputStream;
import java.io.FileNotFoundException;
import java.io.FileOutputStream;
import java.io.IOException;
import java.io.ObjectInputStream;
import java.io.ObjectOutputStream;

public class ObjectSerializationTest {

    public static void main(String[] args) {
        // Serialize
        try {
            Customer customer = new Customer(1, "Joe Blog",
```

```
                "12 West Cost");
            FileOutputStream fos = new FileOutputStream(
                "c:\\temp\\objectOutput");
            ObjectOutputStream oos = new ObjectOutputStream(fos);
            // write first object
            oos.writeObject(customer);
            // write second object
            oos.writeObject("Customer Info");
            oos.close();
            fos.close();
        } catch (FileNotFoundException e) {
            System.out.print("FileNotFound");
        } catch (IOException e) {
            System.out.print("IOException");
        }

        // Deserialize
        try {
            FileInputStream fis =
                new FileInputStream("c:\\temp\\objectOutput");
            ObjectInputStream ois = new ObjectInputStream(fis);
            // read first object
            Customer customer2 = (Customer) ois.readObject();
            System.out.println("First Object: ");
            System.out.println(customer2.id);
            System.out.println(customer2.name);
            System.out.println(customer2.address);

            // read second object
            System.out.println();
            System.out.println("Second object: ");
            String info = (String) ois.readObject();
            System.out.println(info);
            ois.close();
            fis.close();
        } catch (ClassNotFoundException ex) {
            System.out.print("ClassNotFound " + ex.getMessage());
        } catch (IOException ex2) {
            System.out.print("IOException " + ex2.getMessage());
        }
    }
}
```

Summary

Input output operations are supported through the members of the **java.io** package. You can read and write data through streams and data is classified into binary data and text. In addition, Java support object serialization through the **Serializable** interface and the **ObjectInputStream** and **ObjectOutputStream** classes.

Questions

1. What is a stream?
2. Name four abstract classes that represent streams in the **java.io** package.
3. What is object serialization?
4. What is the requirement for a class to be serializable?

Chapter 14
Nested and Inner Classes

Nested and inner classes are often considered too confusing for beginners. However, they have some merits that make them a proper discussion topic in this book. To name a few, you can hide an implementation completely using a nested class, and it provides a shorter way of writing an event-listener.

One of the reasons nested and inner classes sometimes seem overly complex is because the terms "nested classes" and "inner classes" are often used to mean different things in different texts. This book will stick to the definitions in *The Java Language Specification, Third Edition*, which is the formal specification from Sun Microsystems (http://java.sun.com/docs/books/jls/).

This chapter starts by defining what nested classes and inner classes are and continues by explaining types of nested classes.

An Overview of Nested Classes

Let's start this chapter by learning the correct definitions of nested and inner classes. A nested class is a class declared within the body of another class or interface. There are two types of nested classes: static and non-static. Non-static nested classes are called inner classes.

There are several types of inner classes:

- member inner classes
- local inner classes
- anonymous inner classes

The term "top level class" is used to refer to a class that is not defined within another class or interface. In other words, there is no class enclosing a top level class.

A nested class behaves pretty much like an ordinary (top level) class. A nested class can extend another class, implements interfaces, become the parent class of subclasses, etc. Here is an example of a simple nested class called **Nested** that is defined within a top level class named **Outer**.

```
public class Outer {
    class Nested {
    }
}
```

And, though uncommon, it is not impossible to have a nested class inside another nested class, such as this:

```
public class Outer {
    class Nested {
        class Nested2 {
```

```
        }
    }
}
```

To a top-level class, a nested class is just like other class members, such as methods and fields. For example, a nested class can have one of the four access modifiers: private, protected, default (package), and public. This is unlike a top level class that can only have either public or default.

Because nested classes are members of an enclosing class, the behavior of static nested classes and the behavior of inner classes are not exactly the same. Here are some differences.

- Static nested classes can have static members, inner classes cannot.
- Just like instance methods, inner classes can access static and non-static members of the outer class, including its private members. Static nested classes can only access the static members of the outer class.
- You can create an instance of a static nested class without first creating an instance of its outer class. By contrast, you must first create an instance of the outer class enclosing an inner class before instantiating the inner class itself.

These are the benefits of inner classes:

1. Inner classes can have access to all (including private) members of the enclosing classes.
2. Inner classes help you hide the implementation of a class completely.
3. Inner classes provide a shorter way of writing listeners in Swing and other event-based applications.

Now, let's review each type of static class.

Static Nested Classes

A static nested class can be created without creating an instance of the outer class. Listing 14.1 shows this.

Listing 14.1: A Static Nested Class

```
package app14;
class Outer1 {
    private static int value = 9;
    static class Nested1 {
        int calculate() {
            return value;
        }
    }
}

public class StaticNestedTest1 {
    public static void main(String[] args) {
        Outer1.Nested1 nested = new Outer1.Nested1();
        System.out.println(nested.calculate());
    }
}
```

There are a few things to note about static nested classes:

- You refer to a nested class by using this format:

```
OuterClassName.InnerClassName
```

- You do not need to create an instance of the enclosing class to instantiate a static nested class.
- You have access to the outer class static members from inside your static nested class

In addition, if you declare a member in a nested class that has the same name as a member in the enclosing class, the former will shadow the latter. However, you can always reference the member in the enclosing class by using this format.

```
OuterClassName.memberName
```

Note that this will still work although *memberName* is private. Examine the example in Listing 14.2.

Listing 14.2: Shadowing an outer class's member.

```
package app14;
class Outer2 {
    private static int value = 9;
    static class Nested2 {
        int value = 10;
        int calculate() {
            return value;
        }
        int getOuterValue() {
            return Outer2.value;
        }
    }
}

public class StaticNestedTest2 {
    public static void main(String[] args) {
        Outer2.Nested2 nested = new Outer2.Nested2();
        System.out.println(nested.calculate());     // returns 10
        System.out.println(nested.getOuterValue()); // returns 9
    }
}
```

Member Inner Classes

A member inner class is a class whose definition is *directly* enclosed by another class or interface declaration. An instance of a member inner class can be created only if you have a reference to an instance of its outer class. To create an instance of an inner class from within the enclosing class, you call the inner class's constructor, just as you would other ordinary classes. However, to create an instance of an inner class from outside the enclosing class, you use the following syntax:

```
EnclosingClassName.InnerClassName inner =
        enclosingClassObjectReference.new InnerClassName();
```

As usual, from within an inner class, you can use the keyword **this** to reference the current instance (the inner class's instance). To reference the enclosing class's instance you use this syntax.

EnclosingClassName.this

Listing 14.3 shows how you can create an instance of an inner class.

Listing 14.3: A member inner class

```
package app14;
class TopLevel {
    private int value = 9;
    class Inner {
        int calculate() {
            return value;
        }
    }
}

public class MemberInnerTest1 {
    public static void main(String[] args) {
        TopLevel topLevel = new TopLevel ();
        TopLevel.Inner inner = topLevel.new Inner();
        System.out.println(inner.calculate());
    }
}
```

Notice how you created an instance of the inner class in Listing 14.3?

A member inner class can be used to hide an implementation completely, something you cannot do without employing an inner class. The following example shows how you can use a member class to hide an implementation completely.

Listing 14.4: Hiding implementations completely

```
package app14;
interface Printer {
    void print(String message);
}
class PrinterImpl implements Printer {
    public void print(String message) {
        System.out.println(message);
    }
}
class SecretPrinterImpl {
    private class Inner implements Printer {
        public void print(String message) {
            System.out.println("Inner:" + message);
        }
    }
    public Printer getPrinter() {
        return new Inner();
    }
}
public class MemberInnerTest2 {
    public static void main(String[] args) {
        Printer printer = new PrinterImpl();
        printer.print("oh");
        // downcast to PrinterImpl
        PrinterImpl impl = (PrinterImpl) printer;
```

```
        Printer hiddenPrinter =
                (new SecretPrinterImpl()).getPrinter();
        hiddenPrinter.print("oh");
        // cannot downcast hiddenPrinter to Outer.Inner
        // because Inner is private
    }
}
```

The **Printer** interface in Listing 14.4 has two implementations. The first is the **PrinterImpl** class, which is a normal class. It implements the **print** method as a public method. The second implementation can be found in **SecretPrinterImpl**. However, rather than implementing the **Printer** interface, the **SecretPrinterImpl** defines a private class called **Inner**, which implements **Printer**. The **getPrinter** method of **SecretPrinterImpl** returns an instance of **Inner**.

What's the difference between **PrinterImpl** and **SecretPrinterImpl**? You can see this from the main method in the test class:

```
Printer printer = new PrinterImpl();
printer.print("Hiding implementation");
// downcast to PrinterImpl
PrinterImpl impl = (PrinterImpl) printer;

Printer hiddenPrinter = (new SecretPrinterImpl()).getPrinter();
hiddenPrinter.print("Hiding implementation");
// cannot downcast hiddenPrinter to Outer.Inner
// because Inner is private
```

You assign **printer** an instance of **PrinterImpl**, and you can downcast **printer** back to **PrinterImpl**. In the second instance, you assign **Printer** with an instance of **Inner** by calling the **getPrinter** method on **SecretPrinterImpl**. However, there is no way you can downcast **hiddenPrinter** back to **SecretPrinterImpl.Inner** because **Inner** is private and therefore not visible.

Local Inner Classes

A local inner class, or local class for short, is an inner class that by definition is not a member class of any other class (because its declaration is not directly within the declaration of the outer class). Local classes have a name, as opposed to anonymous classes that do not.

A local class can be declared inside any block of code, and its scope is within the block. For example, you can declare a local class within a method, an **if** block, a **while** block, and so on. You want to write a local class if an instance of the class is only used within the scope. For example, Listing 14.5 shows an example of a local class.

Listing 14.5: Local inner class

```
package app14;
import java.util.Date;

interface Logger {
    public void log(String message);
}

public class LocalClassTest1 {
```

```
    String appStartTime = (new Date()).toString();
    public Logger getLogger() {
        class LoggerImpl implements Logger {
            public void log(String message) {
                System.out.println(appStartTime + " : " + message);
            }
        }
        return new LoggerImpl();
    }

    public static void main(String[] args) {
        LocalClassTest1 test = new LocalClassTest1();
        Logger logger = test.getLogger();
        logger.log("Local class example");
    }
}
```

The class in Listing 14.5 has a local class named **LoggerImpl** that resides inside a **getLogger** method. The **getLogger** method must return an implementation of the **Logger** interface and this implementation will not be used anywhere else. Therefore, it is a good idea to make an implementation that is local to **getLogger**. Note also that the **log** method within the local class has access to the instance field **appStartTime** of the outer class.

However, there is more. Not only does a local class have access to the members of its outer class, it also has access to the local variables. However, you can only access final local variables. The compiler will generate a compile error if you try to access a local variable that is not final.

Listing 14.6 modifies the code in Listing 14.5. The **getLogger** method in Listing 14.6 allows you to pass a **String** that will become the prefix of each line logged.

Listing 14.6: PrefixLogger test

```
package app14;
import java.util.Date;

interface PrefixLogger {
    public void log(String message);
}

public class LocalClassTest2 {
    public PrefixLogger getLogger(final String prefix) {
        class LoggerImpl implements PrefixLogger {
            public void log(String message) {
                System.out.println(prefix + " : " + message);
            }
        }
        return new LoggerImpl();
    }

    public static void main(String[] args) {
        LocalClassTest2 test = new LocalClassTest2();
        PrefixLogger logger = test.getLogger("DEBUG");
        logger.log("Local class example");
    }
}
```

Anonymous Inner Classes

An anonymous inner class does not have a name. A use of this type of nested class is for writing an interface implementation. For example, the **AnonymousInnerClassTest** class in Listing 14.7 creates an anonymous inner class which is an implementation of **Printable**.

Listing 14.7: Using an anonymous inner class

```
interface Printable {
    void print(String message);
}

public class AnonymousInnerClassTest 1{
    public static void main(String[] args) {

        Printable printer = new Printable() {
            public void print(String message) {
                System.out.println(message);
            }
        }; // this is a semicolon

        printer.print("Beach Music");
    }
}
```

The interesting thing here is that you create an anonymous inner class by using the **new** keyword followed by what looks like a class's constructor (in this case **Printable()**). However, note that **Printable** is an interface and does not have a constructor. **Printable()** is followed by the implementation of the **print** method. Also, note that after the closing brace, you use a semicolon to terminate the statement that instantiates the anonymous inner class.

In addition, you can also create an anonymous inner class by extending an abstract or concrete class, as demonstrated in the code in Listing 14.8.

Listing 14.8: Using an anonymous inner class with an abstract class

```
abstract class Printable {
    void print(String message) {
    }
}

public class AnonymousInnerClassTest 1{
    public static void main(String[] args) {
        Printable printer = new Printable() {
            public void print(String message) {
                System.out.println(message);
            }
        }; // this is a semicolon

        printer.print("Beach Music");
    }
}
```

Note
Anonymous classes are often used in Swing applications. See Chapter 16, "Swinging Higher" for details.

Behind Nested and Inner Classes

The JVM does not know the notion of nested classes. It is the compiler that works hard to compile an inner class into a top level class incorporating the outer class name and the inner class name as the name, both separated by a dollar sign. That is, the code that employs an inner class called **Inner** that resides inside **Outer** like this

```
public class Outer {
    class Inner {
    }
}
```

will be compiled into two classes: **Outer.class** and **Outer$Inner.class**.

What about anonymous inner classes? For anonymous classes, the compiler takes the liberty of generating a name for them, using numbers. Therefore, you'll see something like **Outer$1.class**, **Outer$2.class**, etc.

When a nested class is instantiated, the instance lives as a separate object in the heap. It does not actually live inside the outer class object.

However, with inner class objects, they have an automatic reference to the outer class object as shown. This reference does not exist in an instance of a static nested class, because a static nested class does not have access to its outer class's instance members.

How does an inner class object obtain a reference to its outer class object? Again, this happens because the compiler changes the constructor of the inner class a bit when the inner class is compiled, namely it adds an argument to every constructor. This argument is of type the outer class.

For example, a constructor like this:

```
public Inner()
```

is changed to this.

```
public Inner(Outer outer)
```

And, this

```
public Inner(int value)
```

to

```
public Inner(Outer outer, int value)
```

Note
Remember that the compiler has the discretion to change the code it compiles. For example, if a class (top level or nested) does not have a constructor, it adds a no-arg constructor to it.

The code that instantiates an inner class is also modified, with the compiler passing a reference to the outer class object to the inner class constructor. If you write:

```
Outer outer = new Outer();
Outer.Inner inner = outer.new Inner();
```

the compiler will change it to

```
Outer outer = new Outer();
Outer.Inner inner = outer.new Inner(outer);
```

When an inner class is instantiated inside the outer class, of course, the compiler passes the current instance of the outer class object using the keyword **this**.

```
// inside the Outer class
Inner inner = new Inner();
```

becomes

```
// inside the Outer class
Inner inner = new Inner(this);
```

Now, here is another piece of the puzzle. How does a nested class access its outer class's private members? No object is allowed to access another object's private members. Again, the compiler changes your code, creating a method that accesses the private member in the outer class definition. Therefore,

```
class TopLevel {
    private int value = 9;
    class Inner {
        int calculate() {
            return value;
        }
    }
}
```

is changed to two classes like this:

```
class TopLevel {
    private int value = 9;
    TopLevel() {
    }
    // added by the compiler
    static int access$0(TopLevel toplevel) {
        return toplevel.value;
    }
}
class TopLevel$Inner {
    final TopLevel this$0;
    TopLevel$Inner(TopLevel toplevel) {
        super();
        this$0 = toplevel;
    }
    int calculate() {
        // modified by the compiler
        return TopLevel.access$0(this$0);
    }
}
```

The addition happens in the background so you will not see it in your source. The compiler adds the **access$0** method that returns the private member value so that the inner class can access the private member.

Summary

A nested class is a class whose declaration is within another class. There are four types of nested classes:

- Static nested classes
- Member inner classes
- Local inner classes
- Anonymous inner classes

The benefits of using nested classes include hiding the implementation of a class completely and as a shorter way of writing a class whose instance will only live within a certain context.

Questions

1. What is a nested class and what is an inner class?
2. What can you use nested classes for?
3. What is an anonymous class?

Chapter 15
Polymorphism

Polymorphism is the hardest concept to explain to those new to object-oriented programming (OOP). In fact, most of the time its definition would not make sense without an example or two. Well, try this. Here is the definition in many programming books: Polymorphism is an OOP feature that enables an object to determine which method implementation to invoke upon receiving a method call. If you find this hard to digest, you're not alone. Polymorphism is hard to explain in simple language, but it does not mean the concept is hard to understand.

This chapter starts with a simple example that should make polymorphism crystal clear. It then proceeds with another example that demonstrates the use of polymorphism with reflection.

> **Note**
> In other programming languages, polymorphism is also called late-binding or runtime-binding or dynamic binding.

Defining Polymorphism

In Java and other OOP languages, it is legal to assign to a reference variable an object whose type is different from the variable type, if certain conditions are met. In essence, if you have a reference variable **a** whose type is **A**, it is legal to assign an object of type **B**, like this

```
A a = new B();
```

provided one of the following conditions is met.

- **A** is a class and **B** is a subclass of **A**.
- **A** is an interface and **B** or one of its parents implements **A**.

As you have learned in Chapter 6, "Inheritance," this is called upcasting.

When you assign **a** an instance of **B** like in the code above, **a** is of type **A**. This means, you cannot call a method in **B** that is not defined in A. However, if you print the value of **a.getClass().getName()**, you'll get "B" and not "A." So, what does this mean? At compile time, the type of **a** is **A**, so the compiler will not allow you to call a method in **B** that is not defined in **A**. On the other hand, at runtime the type of **a** is **B**, as proven by the return value of **a.getClass().getName()**.

Now, here comes the essence of polymorphism. If **B** overrides a method (say, a method named **play**) in **A**, calling **a.play()** will cause the implementation of **play** in **B** (and not in **A**) to be invoked. Polymorphism enables an object (in this example, the one referenced by **a**) to determine which method implementation to choose (either the one in **A** or the one in **B**) when a method is called. Polymorphism dictates that the implementation in the runtime object be invoked. But, polymorphism does not stop here.

What if you call another method in **a** (say, a method called **stop**) and the method is not implemented in **B**? The JVM will be smart enough to know this and look into the inheritance hierarchy of **B**. **B**, as it happens, must be a subclass of **A** or, if **A** is an interface, a subclass of another class that implements **A**. Otherwise, the code would not have compiled. Having figured this out, the JVM will climb the ladder of the hierarchy and find the implementation of **stop** and run it.

Now, there is more sense in the definition of polymorphism: Polymorphism is an OOP feature that enables an object to determine which method implementation to invoke upon receiving a method call.

Technically, though, how does Java achieve this? The Java compiler, as it turns out, upon encountering a method call such as **a.play()**, checks if the class/interface represented by **a** defines such a method (a **play** method) and if the correct set of parameters are passed to the method. But, that is the farthest the compiler goes. With the exception of static and final methods, it does not connect (or bind) a method call with a method body. The JVM determines how to bind a method call with the method body at runtime. In other words, except for static and final methods, method binding in Java happens at runtime and not at compile time. Runtime binding is also called late binding or dynamic binding. The opposite is early binding, in which binding occurs at compile time or link time. Early binding happens in other languages, such as C.

Therefore, polymorphism is made possible by the late binding mechanism in Java. Because of this, polymorphism is rather inaccurately also called late-binding or dynamic binding or runtime binding.

Let's look at the Java code in Listing 15.1.

Listing 15.1: An example of polymorphism

```
package app15;

class Employee {
    public void work() {
        System.out.println("I am an employee.");
    }
}

class Manager extends Employee {
    public void work() {
        System.out.println("I am a manager.");
    }

    public void manage() {
        System.out.println("Managing ...");
    }
}

public class PolymorphismTest1 {
    public static void main(String[] args) {
        Employee employee;
        employee = new Manager();
        System.out.println(employee.getClass().getName());
        employee.work();
        Manager manager = (Manager) employee;
        manager.manage();
    }
```

```
}
```

Listing 15.1 defines two non-public classes: **Employee** and **Manager**. **Employee** has a method called **work**, and **Manager** extends **Employee** and adds a new method called **manage**.

The **main** method in the **PolymorphismTest1** class defines an object variable called **employee** of type **Employee**:

```
Employee employee;
```

However, **employee** is assigned an instance of **Manager**, as in:

```
employee = new Manager();
```

This is legal because **Manager** is a subclass of **Employee**, so a **Manager** "is an" **Employee**. Because **employee** is assigned an instance of **Manager**, what is the outcome of **employee.getClass().getName()**? You're right. It's "Manager," not "Employee."

Then, the **work** method is called.

```
employee.work();
```

Guess what is written on the console?

```
I am a manager.
```

This means that it is the **work** method in the **Manager** class that got called, which was polymorphism in action.

Note

Polymorphism does not work with static methods because they are early-bound. For example, if the **work** method in both the **Employee** and **Manager** classes were static, a call to **employee.work()** would print "I am an employee."

Also, since you cannot extend final methods, polymorphism will not work with final methods either.

Now, because the runtime type of **a** is **Manager**, you can downcast **a** to **Manager**, as the code shows:

```
Manager manager = (Manager) employee;
manager.manage();
```

After seeing the code, you might ask, why would you declare **employee** as **Employee** in the first place? Why didn't you declare **employee** as type **Manager**, such as this?

```
Manager employee;
employee = new Manager();
```

You do this to ensure flexibility in cases where you don't know whether the object reference (**employee**) will be assigned an instance of **Manager** or something else. For the same reason, in your programming career you would see something like this a lot of time:

```
List list = new ArrayList();
```

However, the power of polymorphism is not really apparent here, because you can always write

```
Manager employee = new Manager();
```

or

```
ArrayList list = new ArrayList();
```

You are right. But I guarantee you will acknowledge that polymorphism is very

useful after you see the next examples.

Polymorphism and Reflection

Polymorphism is often used along with reflection. Consider this scenario.

The Order Processing application is a business application for handling purchase orders. It can store orders in various databases (Oracle, MySQL, etc) and retrieve orders for display. The **Order** class represents purchase orders. Orders are stored in a database and an **OrderAccessObject** object handles the storing and retrieval of **Order** objects.

The **OrderAccessObject** class acts as an interface between the application and the database. All purchase order manipulations are done through an instance of this class. The **OrderAccessObject** interface is given in Listing 15.2.

Listing 15.2: The OrderAccessObject Interface

```
public interface OrderAccessObject {
    public void addOrder(Order order);
    public void getOrder(int orderId);
}
```

The **OrderAccessObject** interface needs an implementation class that provides the code for the two methods in it. The application may have many implementation classes for **OrderAccessObject**, each of which caters to a specific type of database. For example, the implementation class that connects to an Oracle database is called **OracleOrderAccessObject** class and the one for MySQL is **MySQLOrderAccessObject**. Figure 15.1 shows the UML diagram for **OrderAccessObject** and its implementing classes.

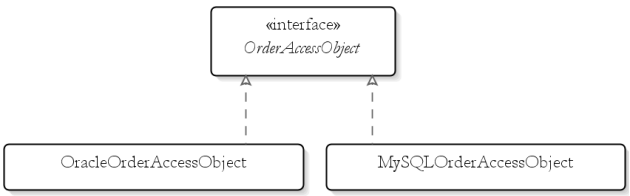

Figure 15.1: The OrderAccessObject interface and implementing classes

The need for multiple implementing classes arises from the fact that each database could have a specific command for performing certain function. For example, autonumbers are common in MySQL, but do not exist in Oracle.

The Order Processing application needs to be flexible enough that it can work with a different database without recompilation. It should also be possible to add support for a new database in the future without recompilation. In fact, you just need to specify the implementing class of **OrderAccessObject** when invoking the application. For example, to use an Oracle database you specify this

```
java OrderProcessing com.mycompany.OracleOrderAccessObject
```

And to work with MySQL, you call it using this command:

```
java OrderProcessing com.mycompany.MySqlOrderAccessObject
```

Now, here is the part of the code that instantiates an **OrderAccessObject** in the database:

```
public static void main (String[] args) {
    OrderAccessObject accessObject = null;
    Class klass = null;
    try {
        klass = Class.forName(args[0]);
        accessObject = (OrderAccessObject) klass.newInstance();
    } catch (ClassNotFoundException e) {
    } catch (Exception e) {
    }

    // continue here
}
```

This is polymorphism because the **accessObject** reference variable can be assigned a different object type each time.

Note

The **forName** and **newInstance** methods are explained in the section "java.lang.Class" in Chapter 5, "Core Classes."

Summary

Polymorphism is one of the main pillars in object-oriented programming. It is useful in circumstances where the type of an object is not known at compile time. This chapter has demonstrated polymorphism through several examples.

Questions

1. In your own words, describe polymorphism.
2. In what situations is polymorphism most useful?

Chapter 16
Annotations

Annotations are notes in Java programs to instruct the Java compiler to do something. You can annotate any program elements, including Java packages, classes, constructors, fields, methods, parameters, and local variables. Java annotations were first defined in JSR 175, "A Metadata Facility for the Java Programming Language." Later JSR 250, "Common Annotations for the Java Platform" added annotations for common concepts. Both specifications can be downloaded from http://www.jcp.org.

This chapter starts with an overview of annotations, and then teaches you how to use the standard and common annotations. It concludes with a discussion of how to write your own custom annotation types.

An Overview of Annotations

Annotations are notes for the Java compiler. When you annotate a program element in a source file, you add notes to the Java program elements in that source file. You can annotate Java packages, types (classes, interfaces, enumerated types), constructors, methods, fields, parameters, and local variables. For example, you can annotate a Java class so that any warnings that the **javac** program would otherwise issue will be suppressed. Or, you can annotate a method that you want to override to ask the compiler to verify that you are really overriding the method, not overloading it.

The Java compiler can be instructed to interpret annotations and discard them (so those annotations only live in source files) or include them in resulting Java classes. Those that are included in Java classes may be ignored by the Java virtual machine, or they may be loaded into the virtual machine. The latter type is called runtime-visible and you can use reflection to inquire about them.

Annotations and Annotation Types

When studying annotations, you will come across these two terms very frequently: annotations and annotation types. To understand their meanings, it is useful to first bear in mind that an annotation type is a special interface type. An annotation is an instance of an annotation type. Just like an interface, an annotation type has a name and members. The information contained in an annotation takes the form of key/value pairs. There can be zero or multiple pairs and each key has a specific type. It can be a **String**, **int**, or other Java types. Annotation types with no key/value pairs are called marker annotation types. Those with one key/value pair are often referred to single-value annotation types.

Annotations were first added to Java 5, which brought with it three annotation types: **Deprecated**, **Override**, and **SuppressWarnings**. They are part of the **java.lang** package and you will learn to use them in the section "Built-in

Annotations." (Java 7 adds another one, **SafeVarargs**, to **java.lang**) On top of that, there are four other annotation types that are part of the **java.lang.annotation** package: **Documented**, **Inherited**, **Retention**, and **Target**. These four annotation types are used to annotate annotations. Java 6 added common annotations, which are explained in the section "Common Annotations."

Annotation Syntax

In your code, you declare an annotation type using this syntax.

```
@AnnotationType
```

or

```
@AnnotationType(elementValuePairs)
```

The first syntax is for marker annotation types and the second for single-value and multi-value types. It is legal to put white spaces between the at sign (@) and annotation type, but this is not recommended.

For example, here is how you use the marker annotation type **Deprecated**:

```
@Deprecated
```

And, this is how you use the second syntax for multi-value annotation type **Author**:

```
@Author(firstName="Ted",lastName="Diong")
```

There is an exception to this rule. If an annotation type has a single key/value pair and the name of the key is **value**, then you can omit the key from the bracket. Therefore, if fictitious annotation type **Stage** has a single key named **value**, you can write

```
@Stage(value=1)
```

or

```
@Stage(1)
```

The Annotation Interface

An annotation type is a Java interface. All annotation types are subinterfaces of **java.lang.annotation.Annotation**. It has one method, **annotationType**, that returns a **java.lang.Class** object.

```
java.lang.Class<? extends Annotation> annotationType()
```

In addition, any implementation of **Annotation** will override the **equals**, **hashCode**, and **toString** methods from the **java.lang.Object** class. Here are their default implementations.

`public boolean equals(Object object)`
: Returns **true** if *object* is an instance of the same annotation type as this one and all members of *object* are equal to the corresponding members of this annotation.

`public int hashCode()`
: Returns the hash code of this annotation, which is the sum of the hash codes of its members

`public String toString()`
: Returns a string representation of this annotation, which typically lists all the key/value pairs of this annotation.

You will use this class when learning custom annotation types later in this chapter.

Standard Annotations

Annotations were a new feature in Java 5 and originally there were three standard annotations, all of which are in the **java.lang** package: **Override**, **Deprecated**, and **SuppressWarnings**. They are discussed in this section.

Override

Override is a marker annotation type that can be applied to a method to indicate to the compiler that the method overrides a method in a superclass. This annotation type guards the programmer against making a mistake when overriding a method.

For example, consider this class **Parent**:

```
class Parent {
    public float calculate(float a, float b) {
        return a * b;
    }
}
```

Suppose, you want to extend **Parent** and override its **calculate** method. Here is a subclass of **Parent**:

```
public class Child extends Parent {
    public int calculate(int a, int b) {
        return (a + 1) * b;
    }
}
```

The **Child** class compiles. However, the **calculate** method in **Child** does not override the method in **Parent** because it has a different signature, namely it returns and accepts **int**s instead of **float**s. In this example, such a programming mistake is easy to spot because you can see both the **Parent** and **Child** classes. However, you won't be always this lucky. Sometimes the parent class is buried somewhere in another package. This seemingly trivial error could be fatal because when a client class calls the **calculate** method on a **Child** object and passes two floats, the method in the **Parent** class will be invoked and the wrong result returned.

Using the **Override** annotation type will prevent this kind of mistake. Whenever you want to override a method, declare the **Override** annotation type before the method:

```
public class Child extends Parent {
    @Override
    public int calculate(int a, int b) {
        return (a + 1) * b;
    }
}
```

This time, the compiler will generate a compile error and you'll be notified that the **calculate** method in **Child** is not overriding the method in the parent class.

It is clear that **@Override** is useful to make sure programmers override a method when they intend to override it, and not overload it.

Deprecated

Deprecated is a marker annotation type that can be applied to a method or a type

to indicate that the method or type is deprecated. A deprecated method or type is marked so by the programmer to warn the users of his code that they should not use or override the method or use or extend the type. The reason why a method or a type is marked deprecated is usually because there is a better method or type and the deprecated method or type is retained in the current software version for backward compatibility.

For example, the **DeprecatedTest** class in Listing 16.1 uses the **Deprecated** annotation type.

Listing 16.1: Deprecating a method

```
package app16;
public class DeprecatedTest {
    @Deprecated
    public void serve() {
    }
}
```

If you use or override a deprecated method, you will get a warning at compile time. For example, Listing 16.2 shows a **DeprecatedTest2** class that uses the **serve** method in **DeprecatedTest**.

Listing 16.2: Using a deprecated method

```
package app16;
public class DeprecatedTest2 {
    public static void main(String[] args) {
        DeprecatedTest test = new DeprecatedTest();
        test.serve();
    }
}
```

Compiling **DeprecatedTest2** generates this warning:

```
Note: app16/DeprecatedTest2.java uses or overrides a deprecated
API.
Note: Recompile with -Xlint:deprecation for details.
```

> **Note**
> In Chapter 28, "Javadoc" you will learn that you can mark a method deprecated by using the **@deprecated** tag in Java Doc. By using the **–deprecation** flag in **javac**, you will be notified if any deprecated method is used.

On top of that, you can use **@Deprecated** to mark a class or an interface, as shown in Listing 16.3.

Listing 16.3: Marking a class deprecated

```
package app16;
@Deprecated
public class DeprecatedTest3 {
    public void serve() {
    }
}
```

SuppressWarnings

SuppressWarnings is used, as you must have guessed, to suppress compiler warnings. You can apply **@SuppressWarnings** to types, constructors, methods, fields, parameters, and local variables.

You use it by passing a **String** array that contains warnings that need to be suppressed. Its syntax is as follows.

```
@SuppressWarnings(value={string-1, ..., string-n})
```

where *string-1* to *string-n* indicate the set of warnings to be suppressed. Duplicate and unrecognized warnings will be ignored.

The following are valid parameters to **@SuppressWarnings**:

- **unchecked**. Gives more detail for unchecked conversion warnings that are mandated by the Java Language Specification.
- **path**. Warns about nonexistent path (classpath, sourcepath, etc) directories.
- **serial**. Warns about missing serialVersionUID definitions on serializable classes.
- **finally**. Warns about finally clauses that cannot complete normally.
- **fallthrough**. Checks switch blocks for fall-through cases, namely cases, other than the last case in the block, whose code does not include a **break** statement, allowing code execution to "fall through" from that case to the next case. As an example, the code following the **case 2** label in this **switch** block does not contain a **break** statement:

```
switch (i) {
case 1:
    System.out.println("1");
    break;
case 2:
    System.out.println("2");
    //  falling through
case 3:
    System.out.println("3");
}
```

As an example, the **SuppressWarningsTest** class in Listing16.4 uses the **SuppressWarnings** annotation type to prevent the compiler from issuing unchecked and fallthrough warnings.

Listing 16.4 Using @SuppressWarnings

```
package app16;
import java.io.File;
import java.io.Serializable;
import java.util.ArrayList;

@SuppressWarnings(value={"unchecked","serial"})
public class SuppressWarningsTest implements Serializable {
    public void openFile() {
        ArrayList a = new ArrayList();
        File file = new File("X:/java/doc.txt");
    }
}
```

Common Annotations

Java includes an implementation of JSR 250, “Common Annotations for the Java Platform,” which specifies annotations for common concepts. The goal of this JSR is to avoid different Java technologies define similar annotations which would

result in duplication.

The full list of common annotations can be found in the document that can be downloaded from http://jcp.org/en/jsr/detail?id=250.

Except for **Generated**, all of the annotations specified are, unfortunately, advanced materials or suitable for Java EE, and therefore beyond the scope of this book. As such, the only common annotation discussed is **@Generated**.

@Generated is used to mark computer generated source code, as opposed to hand-written code. It can be applied to classes, methods, and fields. The following are parameters to **@Generated**:

- **value**. The name of the code generator. The convention is to use the fully qualified name of the generator.
- **date**. The date the code was generated. It must be in a format compliant with ISO 8601.
- **comments**. Comments accompanying the generated code.

For example, in Listing 16.5 **@Generated** is used to annotate a generated class.

Listing 16.5: Using @Generated

```
package app16;
import javax.annotation.Generated;

@Generated(value="com.brainysoftware.robot.CodeGenerator",
        date="2011-12-31", comments="Generated code")
public class GeneratedTest {

}
```

Standard Meta-Annotations

Meta annotations are annotations that annotate annotations. There are four meta-annotation types that can be used to annotate annotations: **Documented**, **Inherited**, **Retention**, and **Target**. All the four are part of the **java.lang.annotation** package. This section discusses these annotation types.

Documented

Documented is a marker annotation type used to annotate the declaration of an annotation type so that instances of the annotation type will be included in the documentation generated using Javadoc or similar tools.

For example, the **Override** annotation type is not annotated using **Documented**. As a result, if you use Javadoc to generate a class whose method is annotated **@Override**, you will not see any trace of **@Override** in the resulting document.

For instance, Listing 16.6 shows a **OverrideTest2** class that uses **@Override** to annotate the **toString** method.

Listing 16.6: The OverrideTest2 class

```
package app16;
public class OverrideTest2 {
    @Override
    public String toString() {
```

```
        return "OverrideTest2";
    }
}
```

On the other hand, the **Deprecated** annotation type is annotated **@Documented**. Recall that the **serve** method in the **DeprecatedTest** class in Listing 16.2 is annotated **@Deprecated**. Now, if you use **Javadoc** to generate the documentation for **OverrideTest2** (Javadoc is discussed in Chapter 28, "Javadoc"), the details of the **serve** method in the documentation will also include **@Deprecated**, like this:

```
serve
@Deprecated
public void serve()
```

Inherited

You use **Inherited** to annotate an annotation type so that any instance of the annotation type will be inherited. If you annotate a class using an inherited annotation type, the annotation will be inherited by any subclass of the annotated class. If the user queries the annotation type on a class declaration, and the class declaration has no annotation of this type, then the class's parent class will automatically be queried for the annotation type. This process will be repeated until an annotation of this type is found or the root class is reached.

Check the section "Custom Annotation Types" on how to query an annotation type.

Retention

@Retention indicates how long annotations whose annotated types are annotated **@Retention** are to be retained. The value of **@Retention** can be one of the members of the **java.lang.annotation.RetentionPolicy** enum:

- **SOURCE**. Annotations are to be discarded by the Java compiler.
- **CLASS**. Annotations are to be recorded in the class file but not retained by the JVM. This is the default value.
- **RUNTIME**. Annotations are to be retained by the JVM so they can be queried using reflection.

For example, the declaration of the **SuppressWarnings** annotation type is annotated **@Retention** with the value of **SOURCE**.

```
@Retention(value=SOURCE)
public @interface SuppressWarnings
```

Target

Target indicates which program element(s) can be annotated using instances of the annotated annotation type. The value of **Target** is one of the members of the **java.lang.annotation.ElementType** enum:

- **ANNOTATION_TYPE**. The annotated annotation type can be used to annotate annotation type declaration.
- **CONSTRUCTOR**. The annotated annotation type can be used to annotate constructor declaration.
- **FIELD**. The annotated annotation type can be used to annotate field declaration.
- **LOCAL_VARIABLE**. The annotated annotation type can be used to

annotate local variable declaration.
- **METHOD**. The annotated annotation type can be used to annotate method declaration.
- **PACKAGE**. The annotated annotation type can be used to annotate package declarations.
- **PARAMETER**. The annotated annotation type can be used to annotate parameter declarations.
- **TYPE**. The annotated annotation type can be used to annotate type declarations.

As an example, the **Override** annotation type declaration is annotated the following **Target** annotation, making **Override** only applicable to method declarations.

```
@Target(value=METHOD)
```

You can have multiple values in the **Target** annotation. For example, this is from the declaration of **SuppressWarnings**:

```
@Target(value={TYPE,FIELD, METHOD, PARAMETER,CONSTRUCTOR,
LOCAL_VARIABLE})
```

Custom Annotation Types

An annotation type is a Java interface, except that you must add an at sign before the **interface** keyword when declaring it.

```
public @interface CustomAnnotation {
}
```

By default, all annotation types implicitly or explicitly extend the **java.lang.annotation.Annotation** interface. In addition, even though you can extend an annotation type, its subtype is not treated as an annotation type.

Writing Your Own Custom Annotation Type

Listing 16.7 shows a custom annotation type called **Author**.

Listing 16.7: The Author annotation type

```
package app16.custom;
import java.lang.annotation.Documented;
import java.lang.annotation.Retention;
import java.lang.annotation.RetentionPolicy;

@Documented
@Retention(RetentionPolicy.RUNTIME)
public @interface Author {
    String firstName();
    String lastName();
    boolean internalEmployee();
}
```

Using Custom Annotation Types

The **Author** annotation type is like any other Java type. Once you import it into a class or an interface, you can use it simply by writing

```
@Author(firstName="firstName",lastName="lastName",
internalEmployee=true|false)
```

For example, the **Test1** class in Listing 16.8 is annotated **Author**.

Listing 16.8: A class annotated Author

```
package app16.custom;
@Author(firstName="John",lastName="Guddell",internalEmployee=true)
public class Test1 {
}
```

Is that it? Yes, that's it. Very simple, isn't it?

The next subsection "Using Reflection to Query Annotations" shows how the **Author** annotations can be of good use.

Using Reflection to Query Annotations

The **java.lang.Class** class has several methods related to annotations.

```
public <A extends java.lang.annotation.Annotation> A getAnnotation
        (Class<A> annotationClass)
```

Returns this element's annotation for the specified annotation type, if present. Otherwise, returns **null**.

```
public java.lang.annotation.Annotation[] getAnnotations()
```

Returns all annotations present on this class.

```
public boolean isAnnotation()
```

Returns **true** if this class is an annotation type.

```
public boolean isAnnotationPresent(Class<? extends
        java.lang.annotation.Annotation> annotationClass)
```

Indicates whether an annotation for the specified type is present on this class

The **app16.custom** package includes three test classes, **Test1**, **Test2**, and **Test3**, that are annotated **Author**. Listing 16.9 shows a test class that employs reflection to query the test classes.

Listing 16.9: Using reflection to query annotations

```
package app16.custom;

public class CustomAnnotationTest {
    public static void printClassInfo(Class c) {
        System.out.print(c.getName() + ". ");
        Author author = (Author) c.getAnnotation(Author.class);
        if (author != null) {
            System.out.println("Author:" + author.firstName()
                    + " " + author.lastName());
        } else {
            System.out.println("Author unknown");
        }
    }

    public static void main(String[] args) {
        CustomAnnotationTest.printClassInfo(Test1.class);
        CustomAnnotationTest.printClassInfo(Test2.class);
        CustomAnnotationTest.printClassInfo(Test3.class);
        CustomAnnotationTest.printClassInfo(
                CustomAnnotationTest.class);
```

```
    }
}
```

When run, you will see the following message in your console:

```
app16.custom.Test1. Author:John Guddell
app16.custom.Test2. Author:John Guddell
app16.custom.Test3. Author:Lesley Nielsen
app16.custom.CustomAnnotationTest. Author unknown
```

Summary

You use annotations to instruct the Java compiler to do something to an annotated program element. Any program element can be annotated, including Java packages, classes, constructors, fields, methods, parameters, and local variables. This chapter explained standard annotation types and taught how to create custom annotation types.

Questions

1. What is an annotation type?
2. What is a meta-annotation?
3. What were the standard annotation types fist included in Java 5?

Chapter 17
Internationalization

In this era of globalization, it is now more compelling than ever to be able to write applications that can be deployed in different countries and regions that speak different languages. There are two terms you need to be familiar with in this regard. The first is internationalization, often abbreviated to i18n because the word starts with an i and ends with an n, and there are 18 characters between the first i and the last n. Internationalization is the technique for developing applications that support multiple languages and data formats without rewriting the programming logic. The second term in localization, which is the technique of adapting an internationalized application to support a specific locale. A locale is a specific geographical, political, or cultural region. An operation that takes a locale into consideration is said to be locale-sensitive. For example, displaying a date is locale-sensitive because the date must be in the format used by the country or region of the user. The fifteenth day of November 2010 is written 11/15/2011 in the US, but printed as 15/11/2011 in Australia. For the same reason internationalization is abbreviated i18n, localization is abbreviated l10n.

Java was designed with internationalization in mind, employing Unicode for characters and strings. Making internationalized applications in Java is therefore easy. How you internationalize your applications depends on how much static data needs to be presented in different languages. There are two approaches.

1. If a large amount of data is static, create a separate version of the resource for each locale. This approach normally applies to Web application with lots of static HTML pages. It is straightforward and will not be discussed in this chapter.
2. If the amount of static data that needs to be internationalized is limited, isolate textual elements such as component labels and error messages into text files. Each text file stores the translations of all textual elements for a locale. The application then retrieves each element dynamically. The advantage is clear. Each textual element can be edited easily without recompiling the application. This is the technique that will discussed in this chapter.

This chapter starts by explaining what a locale is, followed by a technique for internationalizing your application.

Locales

The **java.util.Locale** class represents a locale. There are three main components of a **Locale** object: language, country, and variant. The language is obviously the most important part; however, sometimes the language itself is not sufficient to differentiate a locale. For example, the English language is spoken in countries such as the US and England. However, the English language spoken in the US is not exactly the same as the one used in the UK. Therefore, it is necessary to

specify the country of the language. As another example, the Chinese language used in China is not exactly the same as the one used in Taiwan.

The variant argument is a vendor- or browser-specific code. For example, you use WIN for Windows, MAC for Macintosh, and POSIX for POSIX. Where there are two variants, separate them with an underscore, and put the most important one first. For example, a Traditional Spanish collation might construct a locale with parameters for language, country, and variant as es, ES, Traditional_WIN, respectively.

To construct a **Locale** object, use one of the **Locale** class's constructors.

```
public Locale(java.lang.String language)
public Locale(java.lang.String language, java.lang.String country)
public Locale(java.lang.String language, java.lang.String country,
        java.lang.String variant)
```

The language code is a valid ISO language code. Table 17.1 displays examples of language codes.

The country argument is a valid ISO country code, which is a two-letter, uppercase code specified in ISO 3166 (http://userpage.chemie.fu-berlin.de/diverse/doc/ISO_3166.html). Table 17.1 lists some of the country codes in ISO 3166.

Code	Language
de	German
el	Greek
en	English
es	Spanish
fr	French
hi	Hindi
it	Italian
ja	Japanese
nl	Dutch
pt	Portuguese
ru	Russian
zh	Chinese

Table 17.1: Examples of ISO 639 Language Codes

Country	Code
Australia	AU
Brazil	BR
Canada	CA
China	CN
Egypt	EG
France	FR
Germany	DE
India	IN
Mexico	MX
Switzerland	CH
Taiwan	TW
United Kingdom	GB
United States	US

Table 17.2: Examples of ISO 3166 Country Codes

For example, to construct a **Locale** object representing the English language used in Canada, write this.

```
Locale locale = new Locale("en", "CA");
```

In addition, the **Locale** class provides static final fields that return locales for specific countries or languages, such as **CANADA**, **CANADA_FRENCH**, **CHINA**, **CHINESE**, **ENGLISH**, **FRANCE**, **FRENCH**, **UK**, **US**, etc. Therefore, you can also construct a **Locale** object by calling its static field:

```
Locale locale = Locale.CANADA_FRENCH;
```

In addition, the static **getDefault** method returns the user computer's locale.

```
Locale locale = Locale.getDefault();
```

Internationalizing Applications

Internationalizing and localizing your applications require you to

1. isolate textual components into properties files
2. be able to select and read the correct properties file

This section elaborates the two steps and provides a simple example. The section "An Internationalized Swing Application" later in this chapter presents another example.

Isolating Textual Components into Properties Files

An internationalized application stores its textual elements in a separate properties file for each locale. Each file contains key/value pairs, and each key uniquely identifies a locale-specific object. Keys are always strings, and values can be strings or any other type of object. For example, to support American English, German, and Chinese you will have three properties files, all of which have the same keys.

The following is the English version of the properties file. Note that it has two keys: **greetings** and **farewell**.

```
greetings = Hello
farewell = Goodbye
```

The German version would be as follows:

```
greetings = Hallo
farewell = Tschüß
```

And the properties file for the Chinese language is as follows:

```
greetings=\u4f60\u597d
farewell=\u518d\u89c1
```

Read the sidebar "Converting Chinese Characters to Unicode" on how we arrived at the previous properties file.

Now, you need to master the **java.util.ResourceBundle** class. It enables you to easily choose and read the properties file specific to the user's locale and look up the values. **ResourceBundle** is an abstract class, but it provides static **getBundle** methods that return an instance of a concrete subclass.

Converting Chinese Characters to Unicode

In the Chinese language, 你好 (meaning hello, represented by the Unicode codes 4f60 and 597d, respectively) and 再见 (meaning good bye and is represented by Unicode codes 518d and 89c1, respectively) are the most common expressions. Of course, no one remembers the Unicode code of each Chinese character. Therefore, you create the .properties file in two steps:

1. Using your favorite Chinese text editor, create a text file like this:

```
greetings=你好
farewell=再见
```

2. Convert the content of the text file into the Unicode representation. Normally, a Chinese text editor has the feature for converting Chinese characters into Unicode codes. You will get the end result:

```
greetings=\u4f60\u597d
farewell=\u518d\u89c1
```

A **ResourceBundle** has a base name, which can be any name. In order for a **ResourceBundle** to pick up a properties file, the filename must be composed of the **ResourceBundle** base name, followed by an underscore, followed by the language code, and optionally followed by another underscore and the country code. The format for the properties file name is as follows:

```
basename_languageCode_countryCode
```

For example, suppose the base name is **MyResources** and you define the following three locales:

- US-en
- DE-de
- CN-zh

Then you would have these three properties files:

- **MyResources_en_US.properties**
- **MyResources_de_DE.properties**
- **MyResources_zh_CN.properties**

Reading Properties Files using ResourceBundle

As mentioned previously, **ResourceBundle** is an abstract class. Nonetheless, you can obtain an instance of **ResourceBundle** by calling its static **getBundle** method. The signatures of its overloads are

```
public static ResourceBundle getBundle(java.lang.String baseName)

public static ResourceBundle getBundle(java.lang.String baseName,
        Locale locale)
```

For example:

```
ResourceBundle rb =
        ResourceBundle.getBundle("MyResources", Locale.US);
```

This will load the **ResourceBundle** with the values in the corresponding properties file.

If a suitable properties file is not found, the **ResourceBundle** object will fall back to the default properties file. The name of the default properties file will be the base name with a **properties** extension. In this case, the default file would be

MyResources.properties. If this file is not found, a **java.util.MissingResourceException** will be thrown.

Then, to read a value, you use the **ResourceBundle** class's **getString** method, passing the key.

```
public java.lang.String getString(java.lang.String key)
```

If the entry with the specified key is not found, a **java.util.MissingResourceException** will be thrown.

Summary

This chapter explains how to develop an internationalized application. First it explained the **java.util.Locale** class and the **java.utilResourceBundle** class. It then continued with an example of an internationalized application.

Questions

1. What is the approach to internationalizing applications with plenty of static contents?
2. How do you isolate textual elements of a Java application?
3. What are the two classes used in internationalization and localization?

Chapter 18
Java Networking

Computer networking is concerned with communication between computers. Nowadays, this form of interaction is ubiquitous. Whenever you surf the Internet, it is your machine exchanging messages with remote servers. When you transfer a file over an FTP channel, you are also using some kind of networking service. Java comes equipped with the **java.net** package that contains types that make network programming easy. We'll examine some of the types after an overview of networking. Towards the end of the chapter, some examples are presented for you to play with.

An Overview of Networking

A network is a collection of computers that can communicate with each other. Depending on how wide the coverage is, a network can be referred to as a local area network (LAN) or a wide area network (WAN). A LAN is normally confined to a limited geographic area, such as a building, and comprises from as few as three to as many as hundreds of computers. A WAN, by contrast, is a combination of multiple LANs that are geographically separate. The largest network of all is, of course, the Internet.

The communication medium within a network can be cables, telephone lines, high-speed fiber, satellites, and so on. As the wireless technology gets more and more mature and inexpensive, a wireless local area network (WLAN) is becoming more commonplace nowadays.

Just like two people use a common language to converse, two computers communicate by using a common 'language' both agreed on. In computer jargon, this 'language' is referred to as protocol. What's confusing is that there are several layers of protocols. This is because at the physical layer two computer communicate by exchanging bitstreams, which are collections of ones and zeroes. This is too hard to be understood by applications and humans. Therefore, there is another layer that translates bitstreams into something more tangible and vice versa.

The easiest protocols are those at the application layer. Writing applications require you to understand protocols in the application layer. There are several protocols in this layer: HTTP, FTP, telnet, etc.

Application layer protocols use the protocols in the transport layer. Two popular ones at the transport layer are TCP and UDP. In turn transport layer protocols utilize the protocols at the layer below it. The diagram in Figure 18.1 shows some of these layers.

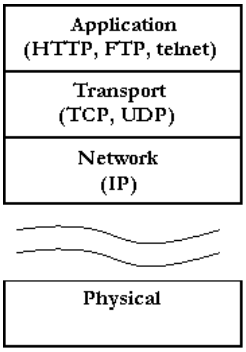

Figure 18.1: Layers of protocol in the computer network

Thanks to this strategy, you don't have to worry about protocols in other layers than the application layer. Java even goes the extra mile to provide classes that encapsulate application layer protocols. For example, with Java, you do not need to understand the HTTP to be able to send a message to an HTTP server. The HTTP, one of the most popular protocols, is covered in detail in this chapter for those who want to know more than the surface.

Another thing that you should know is that a network employs an addressing system to distinguish a computer from another, just like your house has an address so that the mailman can deliver your mail. The equivalent of the street address on the Internet is the IP address. Each computer is assigned a unique IP address.

The IP address is not the smallest unit in the network addressing system. The port is. The analogy is an apartment building that share the same street address but has many units, each with its own suite number.

The Hypertext Transfer Protocol (HTTP)

The HTTP is the protocol that allows web servers and browsers to send and receive data over the Internet. It is a request and response protocol. The client requests a file and the server responds to the request. HTTP uses reliable TCP connections—by default on TCP port 80. The first version of HTTP was HTTP/0.9, which was then overridden by HTTP/1.0. Replacing HTTP/1.0 is the current version of HTTP/1.1, which is defined in Request for Comments (RFC) 2616 and downloadable from http://www.w3.org/Protocols/HTTP/1.1/rfc2616.pdf.

In the HTTP, it is always the client who initiates a transaction by establishing a connection and sending an HTTP request. The web server is in no position to contact a client or make a callback connection to the client. Either the client or the server can prematurely terminate a connection. For example, when using a web browser you can click the Stop button on your browser to stop downloading a file, effectively closing the HTTP connection with the Web server.

HTTP Requests

An HTTP request consists of three components:

- Method—Uniform Resource Identifier (URI)—Protocol/Version
- Request headers
- Entity body

The following is an example of an HTTP request:

```
POST /examples/default.jsp HTTP/1.1
Accept: text/plain; text/html
Accept-Language: en-gb
Connection: Keep-Alive
Host: localhost
User-Agent: Mozilla/5.0 (Macintosh; U; Intel Mac OS X 10.5; en-US;
    rv:1.9.2.6) Gecko/20100625 Firefox/3.6.6
Content-Length: 33
Content-Type: application/x-www-form-urlencoded
Accept-Encoding: gzip, deflate

lastName=Franks&firstName=Michael
```

The method—URI—protocol version appears as the first line of the request.

```
POST /examples/default.jsp HTTP/1.1
```

where **POST** is the request method, **/examples/default.jsp** the URI and **HTTP/1.1** the Protocol/Version section.

Each HTTP request can use one of the many request methods as specified in the HTTP standards. The HTTP 1.1 supports seven types of request: GET, POST, HEAD, OPTIONS, PUT, DELETE, and TRACE. GET and POST are the most commonly used in Internet applications.

The URI specifies an Internet resource. It is usually interpreted as being relative to the server's root directory. Thus, it should always begin with a forward slash /. A Uniform Resource Locator (URL) is actually a type of URI (See http://www.ietf.org/rfc/rfc2396.txt). The protocol version represents the version of the HTTP protocol being used.

The request header contains useful information about the client environment and the entity body of the request. For example, it could contain the language the browser is set for, the length of the entity body, and so on. Each header is separated by a carriage return/linefeed (CRLF) sequence.

Between the headers and the entity body, there is a blank line (CRLF) that is important to the HTTP request format. The CRLF tells the HTTP server where the entity body begins. In some Internet programming books, this CRLF is considered the fourth component of an HTTP request.

In the previous HTTP request, the entity body is simply the following line:

```
lastName=Franks&firstName=Michael
```

The entity body can easily become much longer in a typical HTTP request.

HTTP Responses

Similar to an HTTP request, an HTTP response also consists of three parts:

- Protocol—Status code—Description
- Response headers
- Entity body

The following is an example of an HTTP response:

```
HTTP/1.1 200 OK
Server: Apache-Coyote/1.1
Date: Thu, 12 Aug 2010 13:13:33 GMT
```

```
Content-Type: text/html
Last-Modified: Thu, 5 Aug 2010 13:13:12 GMT
Content-Length: 112

<html>
<head>
<title>HTTP Response Example</title>
</head>
<body>
Welcome to Brainy Software
</body>
</html>
```

The first line of the response header is similar to the first line of a request header. The first line tells you that the protocol used is HTTP version 1.1, the request succeeded (200 is the success code), and that everything went okay.

The response headers contain useful information similar to the headers in the request. The entity body of the response is the HTML content of the response itself. The headers and the entity body are separated by a sequence of CRLFs.

---

java.net.URL

A URL is a unique address to an Internet resource. For example, every page on the Internet has a different URL. Here is a URL:

```
http://www.yahoo.com:80/en/index.html
```

A URL has several components. The first component denotes the protocol to use to retrieve the resource. In the preceding example, the protocol is HTTP. The second part, www.yahoo.com, is the host. It tells you where the resource resides. Number 80 after the host is the port number. The last part, /en/index.html, specifies the path of the URL. By default, the HTTP uses port 80.

The HTTP is the most common protocol used in a URL. However, it is not the only one. For example, this URL refers to a jpeg file in the local computer.

```
file://localhost/C:/data/MyPhoto.jpg
```

Detailed information about URLs can be found at this location:

```
http://www.ietf.org/rfc/rfc2396.txt
```

In Java, a URL is represented by a **java.net.URL** object. You construct a **URL** by invoking one of the **URL** class's constructors. Here are some easier constructors:

```
public URL(java.lang.String spec)

public URL(java.lang.String protocol, java.lang.String host,
        java.lang.String file)

public URL(java.lang.String protocol, java.lang.String host,
        int port, java.lang.String file)

public URL(URL context, String spec)
```

Here is an example.

```
URL myUrl = new URL("http://www.brainysoftware.com/");
```

Because no page is specified, the default page is assumed.

As another example, the following lines of code create identical URL objects.

```
URL yahoo1 = new URL("http://www.yahoo.com/index.html");
URL yahoo2 = new URL("http", "www.yahoo.com", "/index.html");
URL yahoo3 = new URL("http", "www.yahoo.com", 80, "/index.html");
```

Parsing a URL

You can retrieve the various components of a URL object by using these methods:

```
public java.lang.String getFile()
public java.lang.String getHost()
public java.lang.String getPath()
public int getPort()
public java.lang.String getProtocol()
public java.lang.String getQuery()
```

For example, the code in Listing 18.1 creates a URL and prints its various parts.

Listing 18.1: Parsing a URL

```
package app21;
import java.net.URL;

public class URLTest1 {
    public static void main(String[] args) throws Exception {
        URL url = new URL(
          "http://www.yahoo.com:80/en/index.html?name=john#first");
        System.out.println("protocol:" + url.getProtocol());
        System.out.println("port:" + url.getPort());
        System.out.println("host:" + url.getHost());
        System.out.println("path:" + url.getPath());
        System.out.println("file:" + url.getFile());
        System.out.println("query:" + url.getQuery());
        System.out.println("ref:" + url.getRef());
    }
}
```

The result of running the **URLTest1** class is as follows.

```
protocol:http
port:80
host:www.yahoo.com
path:/en/index.html
file:/en/index.html?name=john
query:name=john
ref:first
```

Reading A Web Resource

You can use the URL class's **openStream** method to read a web resource. Here is the signature of this method.

```
public final java.io.InputStream openStream()
        throws java.io.IOException
```

For example, the **URLTest2** class in Listing 18.2 prints the content of http://www.google.com.

Listing 18.2: Opening a URL's stream

```
package app21;
import java.io.BufferedReader;
import java.io.IOException;
import java.io.InputStream;
import java.io.InputStreamReader;
import java.net.MalformedURLException;
import java.net.URL;

public class URLTest2 {
    public static void main(String[] args) {
        try {
            URL url = new URL("http://www.google.com/");
            InputStream inputStream = url.openStream();
            BufferedReader bufferedReader = new BufferedReader(
                        new InputStreamReader(inputStream));
            String line = bufferedReader.readLine();
            while (line!= null) {
                System.out.println(line);
                line = bufferedReader.readLine();
            }
            bufferedReader.close();
        }
        catch (MalformedURLException e) {
            e.printStackTrace();
        }
        catch (IOException e) {
            e.printStackTrace();
        }
    }
}
```

Note

You can use a **URL** only to read a web resource. To write to a server, use a **java.net.URLConnection** object.

---

java.net.URLConnection

A **URLConnection** represents a connection to a remote machine. You use it to read a resource from and write to a remote machine. The **URLConnection** class does not have a public constructor, so you cannot construct a **URLConnection** using the **new** keyword. To obtain an instance of **URLConnection**, call the **openConnection** method on a **URL** object.

The **URLConnection** class has two **boolean** fields, **doInput** and **doOutput**, that indicate whether the **URLConnection** can be used for reading and writing, respectively. The default value of **doInput** is **true**, indicating you can always use a **URLConnection** to read a Web resource. The default value of **doOutput** is **false**, meaning a **URLConnection** is not for writing. To use a **URLConnection** object to write, you need to set the value of **doOutput** to **true**. Setting the values of **doInput** and **doOutput** can be done using the **setDoInput** and **setDoOutput** methods:

```
public void setDoInput(boolean value)
```

```
public void setDoOutput(boolean value)
```

You can use the following methods to get the values of **doInput** and **doOutput**:

```
public boolean getDoInput()
```

```
public boolean getDoOutput()
```

To read using a **URLConnection** object, call its **getInputStream** method. This method returns a **java.io.InputStream** object. This method is similar to the **openStream** method in the **URL** class. This is to say that

```
URL url = new URL("http://www.google.com/");
InputStream inputStream = url.openStream();
```

has the same effect as

```
URL url = new URL("http://www.google.com/");
URLConnection urlConnection = url.openConnection();
InputStream inputStream = urlConnection.getInputStream();
```

However, **URLConnection** is more powerful than **URL.openStream** because you can also read the response headers and write to the server. Here are some methods you can use to read the response headers:

```
public java.lang.String getHeaderField(int n)
```
Returns the value of the nth header.

```
public java.lang.String getHeaderField(java.lang.String headerName)
```
Returns the value of the named header.

```
public long getHeaderFieldDate(java.lang.String headerName,
    long default)
```
Returns the value of the named field as a date. The result is the number of milliseconds that has lapsed since January 1, 1970 GMT. If the field is missing, *default* is returned.

```
public java.util.Map getHeaderFields()
```
Returns a **java.util.Map** containing the response headers.

And, here are some other useful methods:

```
public java.lang.String getContentEncoding()
```
Returns the value of the **content-encoding** header

```
public int getContentLength()
```
Returns the value of the **content-length** header.

```
public java.lang.String getContentType()
```
Returns the value of the **content-type** header.

```
public long getDate()
```
Returns the value of the **date** header.

```
public long getExpiration()
```
Returns the value of the expires header.

Reading Web Resources

Listing 18.3 shows a class that reads from a server and displays the response headers.

Listing 18.3: Reading a web resource's headers and content

```
package app21;
import java.io.BufferedReader;
```

```
import java.io.IOException;
import java.io.InputStream;
import java.io.InputStreamReader;
import java.net.MalformedURLException;
import java.net.URL;
import java.net.URLConnection;
import java.util.List;
import java.util.Map;
import java.util.Set;

public class URLConnectionTest1 {
    public static void main(String[] args) {
        try {
            URL url = new URL("http://www.java.com/");
            URLConnection urlConnection = url.openConnection();
            Map<String, List<String>> headers =
                    urlConnection.getHeaderFields();
            Set<Map.Entry<String, List<String>>> entrySet =
                    headers.entrySet();
            for (Map.Entry<String, List<String>> entry : entrySet){
                String headerName = entry.getKey();
                System.out.println("Header Name:" + headerName);
                List<String> headerValues = entry.getValue();
                for (String value : headerValues) {
                    System.out.print("Header value:" + value);
                }
                System.out.println();
                System.out.println();
            }
            InputStream inputStream =
                    urlConnection.getInputStream();
            BufferedReader bufferedReader = new BufferedReader(
                        new InputStreamReader(inputStream));
            String line = bufferedReader.readLine();
            while (line != null) {
                System.out.println(line);
                line = bufferedReader.readLine();
            }
            bufferedReader.close();
        } catch (MalformedURLException e) {
            e.printStackTrace();
        } catch (IOException e) {
            e.printStackTrace();
        }
    }
}
```

The first few lines of the response are the headers: (you might get different ones)

```
Header Name:Set-cookie
Header value:JSESSIONID=92B6E51AD28EE79182924C33E7A3AE85;Path=/

Header Name:Transfer-encoding
Header value:chunked

Header Name:Date
```

```
Header value:Sat, 24 Jul 2010 01:01:26 GMT

Header Name:Server
Header value:Sun-Java-System-Web-Server/6.1

Header Name:Content-type
Header value:text/html; charset=UTF-8

Header Name:null
Header value:HTTP/1.1 200 OK
```

The headers are followed by the resource content (not displayed here to save space).

Writing to a web server

You can use a **URLConnection** to send an HTTP request. For example, the snippet here sends a form to http://www.mydomain.com/form.jsp page.

```
URL url = new URL("http://www.mydomain.com/form.jsp");
URLConnection connection = url.openConnection();
connection.setDoOutput(true);
PrintWriter out = new PrintWriter(connection.getOutputStream());
out.println("firstName=Joe");
out.println("lastName=Average");
out.close();
```

While you can use a **URLConnection** to post messages, you don't normally use it for this purpose. Instead, you use the more powerful **java.net.Socket** and **java.net.ServerSocket** classes discussed in the next sections.

---

java.net.Socket

A socket is an endpoint of a network connection. A socket enables an application to read from and write to the network. Two software applications residing on two different computers can communicate with each other by sending and receiving byte streams over a connection. To send a message from your application to another application, you need to know the IP address as well as the port number of the socket of the other application. In Java, a socket is represented by a **java.net.Socket** object.

To create a socket, you can use one of the many constructors of the **Socket** class. One of these constructors accepts the host name and the port number:

```
public Socket(java.lang.String host, int port)
```

where *host* is the remote machine name or IP address and *port* is the port number of the remote application. For example, to connect to yahoo.com at port 80, you would construct the following **Socket** object:

```
new Socket("yahoo.com", 80)
```

Once you create an instance of the **Socket** class successfully, you can use it to send and receive streams of bytes. To send byte streams, you must first call the **Socket** class's **getOutputStream** method to obtain a **java.io.OutputStream** object. To send text to a remote application, you often want to construct a **java.io.PrintWriter** object from the **OutputStream** object returned. To receive byte streams from the other end of the connection, you call the **Socket** class's

getInputStream method that returns a **java.io.InputStream**.

The code in Listing 18.4 simulates an HTTP client using a socket. It sends an HTTP request to the host and displays the response from the server.

Listing 18.4: A simple HTTP client

```
package app21;
import java.io.BufferedReader;
import java.io.IOException;
import java.io.InputStreamReader;
import java.io.OutputStream;
import java.io.PrintWriter;
import java.net.Socket;

public class SocketTest1 {
    public static void main(String[] args) {
        String host = "books.brainysoftware.com";
        String protocol = "http";
        try {
            Socket socket = new Socket(protocol + "://" + host,80);
            OutputStream os = socket.getOutputStream();
            boolean autoflush = true;
            PrintWriter out = new
                    PrintWriter(socket.getOutputStream(),
                    autoflush);
            BufferedReader in = new BufferedReader(
                   new InputStreamReader(socket.getInputStream()));

            // send an HTTP request to the web server
            out.println("GET / HTTP/1.1");
            out.println("Host: " + host + ":80");
            out.println("Connection: Close");
            out.println();

            // read the response
            boolean loop = true;
            StringBuilder sb = new StringBuilder(8096);
            while (loop) {
                if (in.ready()) {
                    int i = 0;
                    while (i != -1) {
                        i = in.read();
                        sb.append((char) i);
                    }
                    loop = false;
                }
            }

            // display the response to the out console
            System.out.println(sb.toString());
            socket.close();
        } catch (IOException e) {
            e.printStackTrace();
        }
    }
```

```
}
```

To get a proper response from the Web server, you need to send an HTTP request that complies with the HTTP protocol. If you have read the previous section, "The Hypertext Transfer Protocol (HTTP)," you should be able to understand the HTTP request in the code above.

> **Note**
> The HttpClient library from the Apache HTTP Components project (http://hc.apache.org) provides classes that can be used as a more sophisticated HTTP client.

---

java.net.ServerSocket

The **Socket** class represents a "client" socket, i.e. a socket that you construct whenever you want to connect to a remote server application. Now, if you want to implement a server application, such as an HTTP server or an FTP server, you need a different approach. Your server must stand by all the time as it does not know when a client application will try to connect to it. In order for your application to be able to do this, you need to use the **java.net.ServerSocket** class. **ServerSocket** is an implementation of a server socket.

ServerSocket is different from **Socket**. The role of a server socket is to wait for connection requests from clients. Once the server socket gets a connection request, it creates a **Socket** instance to handle the communication with the client.

To create a server socket, you need to use one of the four constructors the **ServerSocket** class provides. You need to specify the IP address and port number the server socket will be listening on. Typically, the IP address will be 127.0.0.1, meaning that the server socket will be listening on the local machine. The IP address the server socket is listening on is referred to as the binding address. Another important property of a server socket is its backlog, which is the maximum queue length of incoming connection requests before the server socket starts to refuse the incoming requests.

One of the constructors of the **ServerSocket** class has the following signature:

```
public ServerSocket(int port, int backLog,
        InetAddress bindingAddress);
```

Notice that for this constructor, the binding address must be an instance of **java.net.InetAddress**. An easy way to construct an **InetAddress** object is by calling its **getByName** static method, passing a **String** containing the host name, such as in the following code.

```
InetAddress.getByName("127.0.0.1");
```

The following line of code constructs a **ServerSocket** that listens on port 8080 of the local machine. The **ServerSocket** has a backlog of 1.

```
new ServerSocket(8080, 1, InetAddress.getByName("127.0.0.1"));
```

Once you have a **ServerSocket**, you can tell it to wait for an incoming connection request to the binding address at the port the server socket is listening on. You do this by calling the **ServerSocket** class's **accept** method. This method will only return when there is a connection request and its return value is an instance of the **Socket** class. This **Socket** object can then be used to send and receive byte streams from the client application, as explained in the previous section, "java.net.Socket." Practically, the **accept** method is the only method used in the application

accompanying this chapter.

The web server application in the next section, "A Web Server Application" illustrates the use of **ServerSocket**.

A Web Server Application

This application illustrates the use of the **ServerSocket** and **Socket** classes to communicate with remote computers. The Web server application contains the following three classes that belong to the **app18.webserver** package:

- **HttpServer**
- **Request**
- **Response**

The entry point of this application is the **main** method in the **HttpServer** class. The method creates an instance of **HttpServer** and calls its **await** method. The **await** method, as the name implies, waits for HTTP requests on a designated port, processes them, and sends responses back to the clients. It keeps waiting until a shutdown command is received.

The application cannot do more than sending static resources, such as HTML files and image files, residing in a certain directory. It also displays the incoming HTTP request byte streams on the console. However, it does not send any header, such as dates or cookies, to the browser.

We will now look at the three classes in the following subsections.

The HttpServer Class

The **HttpServer** class represents a web server and is presented in Listing 18.5. Note that the **await** method is given in Listing 18.6 and is not included in Listing 18.5 to save space.

Listing 18.5: The HttpServer class

```
package app18.webserver;
import java.net.Socket;
import java.net.ServerSocket;
import java.net.InetAddress;
import java.io.InputStream;
import java.io.OutputStream;
import java.io.IOException;

public class HttpServer {

    // shutdown command
    private static final String SHUTDOWN_COMMAND = "/SHUTDOWN";

    // the shutdown command received
    private boolean shutdown = false;

    public static void main(String[] args) {
        HttpServer server = new HttpServer();
        server.await();
```

```
    }

    public void await() {
        ServerSocket serverSocket = null;
        int port = 8080;
        try {
            serverSocket = new ServerSocket(port, 1, InetAddress
                    .getByName("127.0.0.1"));
        } catch (IOException e) {
            e.printStackTrace();
            System.exit(1);
        }
        // Loop waiting for a request
        while (!shutdown) {
            Socket socket = null;
            InputStream input = null;
            OutputStream output = null;
            try {
                socket = serverSocket.accept();
                input = socket.getInputStream();
                output = socket.getOutputStream();
                // create Request object and parse
                Request request = new Request(input);
                request.parse();

                // create Response object
                Response response = new Response(output);
                response.setRequest(request);
                response.sendStaticResource();

                // Close the socket
                socket.close();

                // check if the previous URI is a shutdown command
                shutdown =
                        request.getUri().equals(SHUTDOWN_COMMAND);
            } catch (Exception e) {
                e.printStackTrace();
                continue;
            }
        }
    }
}
```

The code listings include a directory called **webroot** that contains some static resources that you can use for testing this application. To request for a static resource, you type the following URL in your browser's Address or URL box:

```
http://machineName:port/staticResource
```

If you are sending a request from a different machine from the one running your application, *machineName* is the name or IP address of the computer running this application. If your browser is on the same machine, you can use **localhost** as the machine name. *port* is 8080 and *staticResource* is the name of the file requested and must reside in `WEB_ROOT`.

For instance, if you are using the same computer to test the application and

you want to ask the **HttpServer** object to send the index.html file, you use the following URL:

```
http://localhost:8080/index.html
```

To stop the server, you send a shutdown command from a web browser by typing the pre-defined string in the browser's Address or URL box, after the **host:port** section of the URL. The shutdown command is defined by the `SHUTDOWN` static final variable in the **HttpServer** class:

```
private static final String SHUTDOWN_COMMAND = "/SHUTDOWN";
```

Therefore, to stop the server, you use the following URL:

```
http://localhost:8080/SHUTDOWN
```

Now, let's look at the **await** method printed in Listing 18.6.

Listing 18.6: The HttpServer class's await method

```
public void await() {
    ServerSocket serverSocket = null;
    int port = 8080;
    try {
        serverSocket = new ServerSocket(port, 1, InetAddress
                .getByName("127.0.0.1"));
    } catch (IOException e) {
        e.printStackTrace();
        System.exit(1);
    }
    // Loop waiting for a request
    while (!shutdown) {
        Socket socket = null;
        InputStream input = null;
        OutputStream output = null;
        try {
            socket = serverSocket.accept();
            input = socket.getInputStream();
            output = socket.getOutputStream();
            // create Request object and parse
            Request request = new Request(input);
            request.parse();

            // create Response object
            Response response = new Response(output);
            response.setRequest(request);
            response.sendStaticResource();

            // Close the socket
            socket.close();

            // check if the previous URI is a shutdown command
            shutdown = request.getUri().equals(SHUTDOWN_COMMAND);
        } catch (Exception e) {
            e.printStackTrace();
            continue;
        }
    }
}
```

The method name **await** is used instead of **wait** because the latter is the name of an

important method in **java.lang.Object** that is frequently used in multithreaded programming.

The **await** method starts by creating an instance of **ServerSocket** and then entering a **while** loop.

```
serverSocket =  new ServerSocket(port, 1,
        InetAddress.getByName("127.0.0.1"));
    ...
// Loop waiting for a request
while (!shutdown) {
    ...
}
```

The code inside the **while** loop stops at the **accept** method of **ServerSocket**, which blocks until an HTTP request is received on port 8080:

```
socket = serverSocket.accept();
```

Upon receiving a request, the **await** method obtains a **java.io.InputStream** and a **java.io.OutputStream** from the **Socket** returned by the **accept** method.

```
input = socket.getInputStream();
output = socket.getOutputStream();
```

The **await** method then creates a **Request** and calls its **parse** method to parse the HTTP request raw data.

```
// create Request object and parse
Request request = new Request(input);
request.parse();
```

Afterwards, the **await** method creates a **Response**, assigns the **Request** to it, and calls its **sendStaticResource** method.

```
// create Response object
Response response = new Response(output);
response.setRequest(request);
response.sendStaticResource();
```

Finally, the **await** method closes the **Socket** and calls the **getUri** method of **Request** to check if the URI of the HTTP request is a shutdown command. If it is, the **shutdown** variable is set to **true** and the program exits the **while** loop.

```
// Close the socket
socket.close();

//check if the previous URI is a shutdown command
shutdown = request.getUri().equals(SHUTDOWN_COMMAND);
```

The Request Class

The **Request** class represents an HTTP request. An instance of this class is constructed by passing the **java.io.InputStream** object obtained from the **Socket** object that handles communication with the client. You call one of the **read** methods on the **InputStream** object to obtain the HTTP request raw data.

The **Request** class is offered in Listing 18.7. It has two public methods, **parse** and **getUri**, which are given in Listings 18.8 and 18.9, respectively.

Listing 18.7: The Request class

```
package app18.webserver;
```

```
import java.io.InputStream;
import java.io.IOException;

public class Request {
    private InputStream input;
    private String uri;

    public Request(InputStream input) {
        this.input = input;
    }

    public void parse() {
        ...
    }

    private String parseUri(String requestString) {
        ...
    }

    public String getUri() {
        return uri;
    }
}
```

The **parse** method parses the raw data in the HTTP request. Not much is done by this method. The only information it makes available is the URI of the HTTP request that it obtains by calling the private method **parseUri**. **parseUri** stores the URI in the **uri** variable. The public **getUri** method is invoked to return the URI of the HTTP request.

To understand how **parse** and **parseUri** work, you need to know the structure of an HTTP request, discussed in the previous section, "The Hypertext Transfer Protocol (HTTP)." In this section, we are only interested in the first part of the HTTP request, the request line. A request line begins with a method token, followed by the request URI and the protocol version, and ends with carriage-return linefeed (CRLF) characters. Elements in a request line are separated by a space character. For instance, the request line for a request for the index.html file using the GET method is as follows.

```
GET /index.html HTTP/1.1
```

The **parse** method reads the whole byte stream from the socket's **InputStream** that is passed to the **Request** and stores the byte array in a buffer. It then populates a **StringBuilder** called **request** using the bytes in the buffer byte array, and passes the string representation of the **StringBuilder** to the **parseUri** method.

The **parse** method is given in Listing 18.8.

Listing 18.8: The Request class's parse method

```
public void parse() {
    // Read a set of characters from the socket
    StringBuilder request = new StringBuilder(2048);
    int i;
    byte[] buffer = new byte[2048];
    try {
        i = input.read(buffer);
    } catch (IOException e) {
        e.printStackTrace();
```

```
            i = -1;
        }
        for (int j = 0; j < i; j++) {
            request.append((char) buffer[j]);
        }
        System.out.print(request.toString());
        uri = parseUri(request.toString());
    }
```

The **parseUri** method then obtains the URI from the request line. Listing 18.9 presents the **parseUri** method. This method searches for the first and the second spaces in the request and obtains the URI from it.

Listing 18.9: the Request class's parseUri method

```
private String parseUri(String requestString) {
    index1 = requestString.indexOf(' ');
    int index2;
    if (index1 != -1) {
        index2 = requestString.indexOf(' ', index1 + 1);
        if (index2 > index1) {
            return requestString.substring(index1 + 1, index2);
        }
    }
    return null;
}
```

The Response Class

The **Response** class represents an HTTP response and is given in Listing 18.10.

Listing 18.10: The Response class

```
package app18.webserver;
import java.io.OutputStream;
import java.io.IOException;
import java.io.InputStream;
import java.nio.file.Files;
import java.nio.file.Path;
import java.nio.file.Paths;

/*
 HTTP Response =
 Status-Line (( general-header | response-header | entity-header )
       CRLF)
 CRLF
 [ message-body ]
 Status-Line = HTTP-Version SP Status-Code SP Reason-Phrase CRLF
 */

public class Response {

    private static final int BUFFER_SIZE = 1024;
    Request request;
    OutputStream output;

    public Response(OutputStream output) {
        this.output = output;
```

```
    }

    public void setRequest(Request request) {
        this.request = request;
    }

    public void sendStaticResource() throws IOException {
        byte[] bytes = new byte[BUFFER_SIZE];
        Path path = Paths.get(System.getProperty("user.dir"),
                "webroot", request.getUri());
        if (Files.exists(path)) {
            try (InputStream inputStream =
                        Files.newInputStream(path)) {
                int ch = inputStream.read(bytes, 0, BUFFER_SIZE);
                while (ch != -1) {
                    output.write(bytes, 0, ch);
                    ch = inputStream.read(bytes, 0, BUFFER_SIZE);
                }
            } catch (IOException e) {
                e.printStackTrace();
            }
        } else {
            // file not found
            String errorMessage = "HTTP/1.1 404 File Not Found\r\n"
                    + "Content-Type: text/html\r\n"
                    + "Content-Length: 23\r\n" + "\r\n"
                    + "<h1>File Not Found</h1>";
            output.write(errorMessage.getBytes());
        }
    }
}
```

First note that the **Response** class's constructor accepts a **java.io.OutputStream** object:

```
public Response(OutputStream output) {
    this.output = output;
}
```

A **Response** object is constructed by the **HttpServer** class's **await** method by passing the **OutputStream** object obtained from the socket.

The **Response** class has two public methods: **setRequest** and **sendStaticResource** method. The **setRequest** method is used to pass a **Request** object to the **Response** object.

sendStaticResource is used to send a static resource, such as an HTML file. It starts by creating a **Path** that points to a resource under the **webroot** directory under the user directory:

```
        Path path = Paths.get(System.getProperty("user.dir"),
                "webroot", request.getUri());
```

It then tests if the resource exists. If it exists, **sendStaticResource** calls **Files.newInputStream** and gets an **InputStream** that connects to the resource file. Then, it invokes the **read** method of the **InputStream** and writes the byte array to the **OutputStream** output. Note that in this case the content of the static

resource is sent to the browser as raw data.

```
        if (Files.exists(path)) {
            try (InputStream inputStream =
                    Files.newInputStream(path)) {
                int ch = inputStream.read(bytes, 0, BUFFER_SIZE);
                while (ch != -1) {
                    output.write(bytes, 0, ch);
                    ch = inputStream.read(bytes, 0, BUFFER_SIZE);
                }
            } catch (IOException e) {
                e.printStackTrace();
            }
```

If the resource does not exist, **sendStaticResource** sends an error message to the browser.

```
String errorMessage = "HTTP/1.1 404 File Not Found\r\n" +
    "Content-Type: text/html\r\n" +
    "Content-Length: 23\r\n" +
    "\r\n" +
    "<h1>File Not Found</h1>";
output.write(errorMessage.getBytes());
```

Running the Application

To run the application, from the working directory, type the following:

```
java app18.webserver.HttpServer
```

To test the application, open your browser and type the following in the URL or Address box:

```
http://localhost:8080/index.html
```

You will see the index.html page displayed in your browser, as in Figure 18.2.

On the console, you can see the HTTP request similar to the following:

```
GET /index.html HTTP/1.1
Accept: image/gif, image/x-xbitmap, image/jpeg, image/pjpeg,
       application/vnd.ms-excel, application/msword,
       application/vnd.ms-powerpoint, application/x-shockwave-flash,
       application/pdf, */*
Accept-Language: en-us
Accept-Encoding: gzip, deflate
User-Agent: Mozilla/5.0 (Macintosh; U; Intel Mac OS X 10.5; en-US;
       rv:1.9.2.6) Gecko/20100625 Firefox/3.6.6
Host: localhost:8080
Connection: Keep-Alive

GET /images/logo.gif HTTP/1.1
Accept: */*
Referer: http://localhost:8080/index.html
Accept-Language: en-us
Accept-Encoding: gzip, deflate
User-Agent: Mozilla/5.0 (Macintosh; U; Intel Mac OS X 10.5; en-US;
       rv:1.9.2.6) Gecko/20100625 Firefox/3.6.6
Host: localhost:8080
Connection: Keep-Alive
```

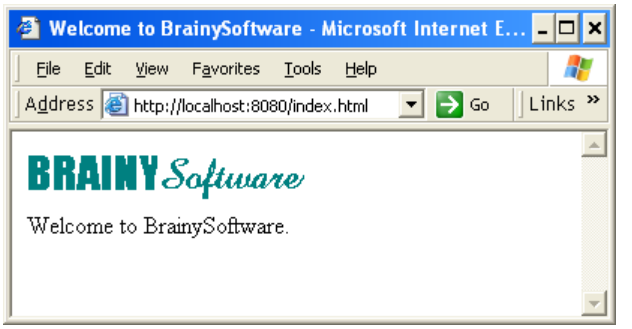

Figure 18.2: The output from the web server

Note

This simple web server application was taken from my other book, "How Tomcat Works: A Guide to Developing Your Own Java Servlet Container." Consult this book for more detailed discussion of how web servers and servlet containers work.

Summary

With the emergence of the Internet, computer networking has become an integral part of life today. Java, with its **java.net** package, makes network programming easy. This chapter discussed the more important types of the **java.net** package, including **URL**, **URLConnection**, **Socket**, and **ServerSocket**. The last section of the chapter presented a simple Web application that illustrates the use of **Socket** and **ServerSocket**.

Questions

1. Why are there several layers of protocols in computer networking?
2. What are the components of a URL?
3. What is the class that represents URLs?
4. What is a socket?
5. What is the difference between a socket and a server socket?

Chapter 19
Java Threads

One of the most appealing features in Java is the support for easy thread programming. Prior to 1995, the year Java was released, threads were the domain of programming experts only. With Java, even beginners can write multi-threaded applications.

This chapter explains what threads are and why they are important. It also talks about related topics such as synchronization and the visibility problem.

Introduction to Java Threads

The next time you play a computer game, ask yourself this question: I am not using a multi-processor computer, how come there seems to be two processors running at the same time, one moving the asteroids and one moving the spaceships? Well, the simultaneous movements are possible thanks to multi-threaded programming.

A program can allocate processor time to units in its body. Each unit is then given a portion of the processor time. Even if your computer only has one processor, it can have multiple units that work at the same time. The trick for single-processor computers is to slice processor time and give each slice to each processing unit. The smallest unit that can take processor time is called a thread. A program that has multiple threads is referred to as a multi-threaded application. Therefore, a computer game is often multi-threaded.

The formal definition of thread is this. A thread is a basic processing unit to which an operating system allocates processor time, and more than one thread can be executing code inside a process. A thread is sometimes called an lightweight process or an execution context.

Threads do consume resources, so you should not create more threads than necessary. In addition, keeping track of many threads is a complex programming task.

Every Java program has at least one thread, the thread that executes the Java program. It is created when you invoke the static **main** method of your Java class. Many Java programs have more than one thread without you realizing it. For example, a Swing application has a thread for processing events in addition to the main thread.

Multi-threaded programming is not only for games. Non-game applications can use multithreads to improve user responsiveness. For example, with only one single thread executing, an application may seem to be ‘hanging’ when writing a large file to the hard disk, with the mouse cursor unable to move and buttons refusing to be clicked. By dedicating a thread to save a file and another to receive user input, your application can be more responsive.

Creating a Thread

There are two ways to create a thread.

1. Extend the **java.lang.Thread** class.
2. Implement the **java.lang.Runnable** interface.

If you choose the first, you need to override its **run** method and write in it code that you want executed by the thread. Once you have a **Thread** object, you call its **start** method to start the thread. When a thread is started, its **run** method is executed. Once the **run** method returns or throws an exception, the thread dies and will be garbage-collected.

Note

The Concurrency Utilities, discussed in Chapter 24, provides a better way of creating and executing a thread. In most cases, you should not work with the **Thread** class directly.

In Java you can give a **Thread** object a name, which is a common practice when working with multiple threads. In addition, every **Thread** has a state and can be in one of these six states.

- new. A state in which a thread has not been started.
- runnable. A state in which a thread is executing.
- blocked. A state in which a thread is waiting for a lock to access an object.
- waiting. A state in which a thread is waiting indefinitely for another thread to perform an action.
- timed_waiting. A state in which a thread is waiting for up to a specified period of time for another thread to perform an action.
- terminated. A state in which a thread has exited.

The values that represent these states are encapsulated in the **java.lang.Thread.State** enum. The members of this enum are **NEW**, **RUNNABLE**, **BLOCKED**, **WAITING**, **TIMED_WAITING**, and **TERMINATED**.

The **Thread** class provides public constructors you can use to create **Thread** objects. Here are some of them.

```
public Thread()

public Thread(String name)

public Thread(Runnable target)

public Thread(Runnable target, String name)
```

Note

I will explain the third and fourth constructors later after the discussion of the **Runnable** interface.

Here are some useful methods in the **Thread** class.

`public String getName()`
: Returns the thread's name.

`public Thread.State getState()`
: Returns the state the thread is currently in.

`public void interrupt()`
: Interrupts this thread.

`public void start()`

Starts this thread.

`public static void sleep(long millis)`

Stops the current thread for the specified number of milliseconds.

In addition, the **Thread** class provides the static **currentThread** method that returns the current working thread.

```
public static Thread currentThread()
```

Extending Thread

The code in Listing 19.1 shows how you can create a thread by extending **java.lang.Thread**.

Listing 19.1: A simple multi-threaded program

```
package app19;
public class ThreadTest1 extends Thread {
    public void run() {
        for (int i = 1; i <= 10; i++) {
            System.out.println(i);
            try {
                sleep(1000);
            } catch (InterruptedException e) {
            }
        }
    }
    public static void main(String[] args) {
        (new ThreadTest1()).start();
    }
}
```

The **ThreadTest1** class extends the **Thread** class and overrides its **run** method. The **ThreadTest1** class begins by instantiating itself. A newly created **Thread** will be in the NEW state. Calling the **start** method will make the thread move from NEW to RUNNABLE, which causes the **run** method to be called. This method prints number 1 to 10 and between two numbers the thread sleeps for a second. When the **run** method returns the thread dies and will be garbage collected. There is nothing fancy about this class, but it gives you a general idea of how to work with **Thread**.

Of course, you do not always have the luxury of extending **Thread** from the main class. For example, if your class extends **javax.swing.JFrame**, then you cannot extend **Thread** because Java does not support multiple inherintance. However, you can always create a second class that extends **Thread**, as shown in the code in Listing 19.2. Or, if you need to access members of the main class, you can write a nested class that extends **Thread**.

Listing 19.2: Using a separate class that extends Thread

```
package app19;
class MyThread extends Thread {
    public void run() {
        for (int i = 1; i <= 10; i++) {
            System.out.println(i);
            try {
                sleep(1000);
            } catch (InterruptedException e) {
            }
```

```
        }
    }
}

public class ThreadTest2 {
    public static void main(String[] args) {
        MyThread thread = new MyThread();
        thread.start();
    }
}
```

The **ThreadTest2** class in Listing 19.2 does exactly the same thing as **ThreadTest1** in Listing 19.1. The difference is the **ThreadTest2** class is free to extend another class.

Implementing Runnable

Another way to create a thread is by implementing **java.lang.Runnable**. This interface has a **run** method that you need to implement. The **run** method in **Runnable** is the same as the **run** method in the **Thread** class. In fact, **Thread** itself implements **Runnable**.

If you use **Runnable**, you have to instantiate the **Thread** class and pass the **Runnable**. Listing 19.3 shows how to work with **Runnable**. It does the same thing as the classes in Listings 19.1 and 19.2.

Listing 19.3: Using Runnable

```
package app19;
public class RunnableTest1 implements Runnable {
    public void run() {
        for (int i = 1; i <= 10; i++) {
            System.out.println(i);
            try {
                Thread.sleep(1000);
            } catch (InterruptedException e) {
            }
        }
    }

    public static void main(String[] args) {
        RunnableTest1 test = new RunnableTest1();
        Thread thread = new Thread(test);
        thread.start();
    }
}
```

Working with Multiple Threads

You can work with multiple threads. The following example is a Swing application that creates two **Thread** objects. The first is responsible for incrementing a counter and the second for decrementing another counter. Listing 19.4 shows it.

Listing 19.4: Using two threads

```
package app19;
import java.awt.FlowLayout;
```

```
import javax.swing.JFrame;
import javax.swing.JLabel;

public class ThreadTest3 extends JFrame {
    JLabel countUpLabel = new JLabel("Count Up");
    JLabel countDownLabel = new JLabel("Count Down");

    class CountUpThread extends Thread {
        public void run() {
            int count = 1000;
            while (true) {
                try {
                    sleep(100);
                } catch (InterruptedException e) {
                }
                if (count == 0)
                    count = 1000;
                countUpLabel.setText(Integer.toString(count--));
            }
        }
    }

    class CountDownThread extends Thread {
        public void run() {
            int count = 0;
            while (true) {
                try {
                    sleep(50);
                } catch (InterruptedException e) {
                }
                if (count == 1000)
                    count = 0;
                countDownLabel.setText(Integer.toString(count++));
            }
        }
    }
    public ThreadTest3(String title) {
        super(title);
        init();
    }

    private void init() {
        this.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
        this.getContentPane().setLayout(new FlowLayout());
        this.add(countUpLabel);
        this.add(countDownLabel);
        this.pack();
        this.setVisible(true);
        new CountUpThread().start();
        new CountDownThread().start();
    }

    private static void constructGUI() {
        JFrame.setDefaultLookAndFeelDecorated(true);
        ThreadTest3 frame = new ThreadTest3("Thread Test 3");
```

```
    }

    public static void main(String[] args) {
        javax.swing.SwingUtilities.invokeLater(new Runnable() {
            public void run() {
                constructGUI();
            }
        });
    }
}
```

The **ThreadTest3** class defines two nested classes, **CountUpThread** and **CountDownThread,** that extend **Thread**. Both are nested in the main class so that they can access the **JLabel** controls and change their labels. Running the code, you will see something similar to Figure 19.1.

Figure 19.1: Using two threads

Thread Priority

When dealing with multiple threads, you sometimes have to think about thread scheduling. In other words, you need to make sure each thread gets a fair chance to run. This is achieved by calling **sleep** from a thread's **run** method. A long processing thread should always calls the **sleep** method to give other threads a slice of the CPU processing time. A thread that calls **sleep** is said to yield.

Now, if there are more than one thread waiting, which one gets to run when the running thread yields? The thread with the highest priority. To set a thread priority, call its **setPriority** method. Its signature is as follows.

```
public final void setPriority(int priority)
```

The following example is a Swing application that has two counters. The counter on the left is powered by a thread that has a priority of 10 and another by a thread whose priority is 1. Run the code and see how the thread with the higher priority runs faster.

Listing 19.5: Testing thread priority

```
package app19;
import java.awt.FlowLayout;
import javax.swing.JFrame;
import javax.swing.JLabel;

public class ThreadPriorityTest extends JFrame {
    JLabel counter1Label = new JLabel("Priority 10");
    JLabel counter2Label = new JLabel("Priority 1");

    class CounterThread extends Thread {
        JLabel counterLabel;
        public CounterThread(JLabel counterLabel) {
            super();
            this.counterLabel = counterLabel;
```

```
        }

        public void run() {
            int count = 0;
            while (true) {
                try {
                    sleep(1);
                } catch (InterruptedException e) {
                }
                if (count == 50000)
                    count = 0;
                counterLabel.setText(Integer.toString(count++));
            }
        }
    }

    public ThreadPriorityTest(String title) {
        super(title);
        init();
    }

    private void init() {
        this.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
        this.setLayout(new FlowLayout());
        this.add(counter1Label);
        this.add(counter2Label);
        this.pack();
        this.setVisible(true);
        CounterThread thread1 = new CounterThread(counter1Label);
        thread1.setPriority(10);
        CounterThread thread2 = new CounterThread(counter2Label);
        thread2.setPriority(1);
        thread2.start();
        thread1.start();
    }

    private static void constructGUI() {
        JFrame.setDefaultLookAndFeelDecorated(true);
        ThreadPriorityTest frame = new ThreadPriorityTest(
                "Thread Priority Test");
    }

    public static void main(String[] args) {
        javax.swing.SwingUtilities.invokeLater(new Runnable() {
            public void run() {
                constructGUI();
            }
        });
    }
}
```

The two threads running are instances of the same class (**CounterThread**). The first thread has a priority of 10, and the second 1. Figure 19.2 shows that even though the second thread starts first, the first thread runs faster.

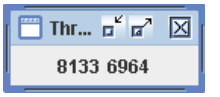

Figure 19.2: Threads with different priorities

Stopping a Thread

The **Thread** class has a **stop** method to stop a thread. However, you should not use this method because it is unsafe. Instead, you should arrange so that the **run** method exits naturally when you want to stop a thread. A common technique used is to employ a **while** loop with a condition. When you want to stop the thread, simply make the condition evaluates to false. For example:

```
boolean condition = true;
public void run {
    while (condition) {
        // do something here
    }
}
```

In your class, you also need to provide a method to change the value of **condition**.

```
public synchronized void stopThread() {
    condition = false;
}
```

Note

The keyword **synchronized** is explained in the section, "Synchronizing Threads."

Stopping a thread is illustrated in the example in Listing 19.6.

Listing 19.6: Stopping a thread

```
package app19;
import java.awt.FlowLayout;
import java.awt.event.ActionEvent;
import java.awt.event.ActionListener;
import javax.swing.JButton;
import javax.swing.JFrame;
import javax.swing.JLabel;

public class StopThreadTest extends JFrame {
    JLabel counterLabel = new JLabel("Counter");
    JButton startButton = new JButton("Start");
    JButton stopButton = new JButton("Stop");
    CounterThread thread = null;
    boolean stopped = false;
    int count = 1;

    class CounterThread extends Thread {
        public void run() {
            while (!stopped) {
                try {
                    sleep(10);
                } catch (InterruptedException e) {
```

```
            }
            if (count == 1000) {
                count = 1;
            }
            counterLabel.setText(Integer.toString(count++));
        }
    }
}

public StopThreadTest(String title) {
    super(title);
    init();
}

private void init() {
    this.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
    this.getContentPane().setLayout(new FlowLayout());
    this.stopButton.setEnabled(false);
    startButton.addActionListener(new ActionListener() {
        public void actionPerformed(ActionEvent e) {
            StopThreadTest.this.startButton.setEnabled(false);
            StopThreadTest.this.stopButton.setEnabled(true);
            startThread();
        }
    });
    stopButton.addActionListener(new ActionListener() {
        public void actionPerformed(ActionEvent e) {
            StopThreadTest.this.startButton.setEnabled(true);
            StopThreadTest.this.stopButton.setEnabled(false);
            stopThread();
        }
    });
    this.getContentPane().add(counterLabel);
    this.getContentPane().add(startButton);
    this.getContentPane().add(stopButton);
    this.pack();
    this.setVisible(true);
}

public synchronized void startThread() {
    stopped = false;
    thread = new CounterThread();
    thread.start();
}

public synchronized void stopThread() {
    stopped = true;
}

private static void constructGUI() {
    JFrame.setDefaultLookAndFeelDecorated(true);
    StopThreadTest frame = new StopThreadTest("Stop Thread
   Test");
}
```

```
    public static void main(String[] args) {
        javax.swing.SwingUtilities.invokeLater(new Runnable() {
            public void run() {
                constructGUI();
            }
        });
    }
}
```

The **StopThreadTest** class uses a **JLabel** to display a counter and two **JButton**s to start and stop the counter, respectively. An action listener is added to each **JButton**. The action listener in the Start button calls the **startThread** method and the one in the Stop button invokes the **stopThread** method.

```
    public synchronized void startThread() {
        stopped = false;
        thread = new CounterThread();
        thread.start();
    }
    public synchronized void stopThread() {
        stopped = true;
    }
```

To stop the counter, simply change the **stopped** variable to **true**. This will cause the **while** loop in the **run** method to exit. To start or restart the counter, you must create a new **Thread**. Once the **run** method of a thread exits, the thread is dead and you cannot re-call the thread's **start** method.

Figure 19.3 shows the counter from the **StopThreadTest** class. It can be stopped and restarted.

Figure 19.3: Stopping and restarting a thread

Synchronization

You've seen threads that run independently from each other. In real life, however, there are often situations whereby multiple threads need access to the same resource or data. Thread interference problems might arise if you cannot guarantee that no two threads will simultaneously have access to the same object.

This section explains the topic of thread interference and the language built-in locking mechanism for securing exclusive access to an object through the **synchronized** modifier.

Note

Java offers the Concurrency Utilities, which include better locks. When possible you should use these locks instead of **synchronized**. The Concurrency Utilities are explained in Chapter 24.

Thread Interference

To better appreciate the issues associated with multiple threads attempting to

access the same resource, consider the code in Listing 19.7.

Listing 19.7: The UserStat class

```
package app19;
public class UserStat {
    int userCount;

    public int getUserCount() {
        return userCount;
    }

    public void increment() {
        userCount++;
    }

    public void decrement() {
        userCount--;
    }
}
```

What happens if a thread attempts to read the **userCount** variable by calling **getUserCount** while another thread is incrementing it? Bear in mind that the statement **userCount++** is actually composed of three consecutive steps:

- Read the value of **userCount** and store it in some temporary storage;
- Increment the value
- Write the incremented value back to **userCount**

Suppose a thread reads and increments the value of **userCount**. Before it has the opportunity to store the incremented value back, another thread reads it and gets the old value. When the second thread finally gets a chance to write to **userCount**, it replaces the incremented value of the first thread. As a result, **userCount** does not reflect the number of users correctly. A thread interference is an event whereby two *non*-atomic operations running in different threads, but acting on the same data, interleave.

Atomic Operations

An atomic operation is a set of operations that can be combined to appear to the rest of the system as a single operation. It cannot cause thread interference. As you have witnessed, incrementing an integer is not an atomic operation.

In Java, all primitives except **long** and **double** are atomically readable and writable.

Thread Safety

Thread safe code functions correctly when accessed by multiple threads. The **UserStat** class in Listing 19.7 is not thread-safe.

Thread interference can lead to a race condition. It is one in which multiple threads are reading or writing some shared data simultaneously and the result is unpredictable. Race conditions can lead to subtle or severe bugs that are hard to find.

The next sections, “Method Synchronization” and “Block Synchronization” explain how to use **synchronized** to write thread-safe code.

Method Synchronization

Every Java object has an intrinsic lock, which is sometimes called a monitor lock.

Acquiring an object's intrinsic lock is a way of having exclusive access to the object. To acquire an object's intrinsic lock is the same as locking the object. Threads attempting to access a locked object will block until the thread holding the lock releases the lock.

Mutual Exclusion and Visibility

Because a locked object can be accessed only by one thread, locks are said to offer a mutual exclusion feature. Another feature offered by locks is visibility, which is discussed in the next section.

The **synchronized** modifier can be used to lock an object. When a thread calls a non-static synchronized method, it will automatically attempt to acquire the intrinsic lock of the method's object before the method can execute. The thread holds the lock until the method returns. Once a thread locks an object, other threads cannot call the same method or other synchronized methods on the same object. The other threads will have to wait until the lock becomes available again. The lock is reentrant, which means the thread holding the lock can invoke other synchronized methods in the same object. The intrinsic lock is released when the method returns.

Note

You can also synchronize a static method, in which case the lock of the **Class** object associated with the method's class will be used.

The **SafeUserStat** class in Listing 19.8 is a rewrite of the **UserStat** class. Unlike **UserStat**, **SafeUserStat** is thread-safe.

Listing 19.8: The SafeUserStat class

```
package app19;
public class SafeUserStat {
    int userCount;

    public synchronized int getUserCount() {
        return userCount;
    }

    public synchronized void increment() {
        userCount++;
    }

    public synchronized void decrement() {
        userCount--;
    }
}
```

Within a program, the code segments that guarantee only one thread at a time has access to a shared resource are called critical sections. In Java critical sections are achieved using the **synchronized** keyword. In the **SafeUserStat** class, the **increment**, **decrement**, and **getUserCount** methods are critical sections. Access to **userCount** is only permitted through a synchronized method. This ensures race conditions will not happen.

Block Synchronization

Synchronizing a method is not always possible. Imagine writing a multi-threaded application with multiple threads accessing a shared object, but the object class was not written with thread safety in mind. Worse still, you do not have access to the source code of the shared object. Just say, you have to work with a thread-

unsafe **UserStat** class and its source code is not available.

Fortunately, Java allows you to lock any object through block synchronization. Its syntax is this.

```
synchronized(object) {
    // do something while locking object
}
```

A synchronized block gives you the intrinsic lock of an object. The lock is released after the code in the block is executed.

For instance, the following code uses the thread-unsafe **UserStat** class in Listing 19.7 as a counter. To lock the counter when incrementing it, the **incrementCounter** method locks the **UserStat** instance.

```
UserStat userStat = new UserStat();
...
public void incrementCounter() {
    synchronized(userStat) {
        // statements to be synchronized, such as calls to
        // the increment, decrement, and getUserCount methods
        // on userStat
        userStat.increment();
    }
}
```

As an aside note, method synchronization is the same as block synchronization that locks the current object:

```
synchronized(this) {
    ...
}
```

Visibility

In the section “Synchronization” you learned to synchronize non-atomic operations that could be accessed by multiple threads. At this point you probably got the impression that if you don't have non-atomic operations, then you don't have to bother synchronizing resources that are accessed by multiple threads.

This is not true.

In a single-threaded program, reading the value of a variable always gives you the value last written to the variable. However, due to the memory model in Java, it's not always so in a multithreaded application. A thread may not see changes made by another thread unless the operations that act on the data are synchronized.

For example, the **Inconsistent** class in Listing 19.9 creates a background thread that is supposed to wait three seconds before changing the value of **started**, a boolean. The **while** loop in the **main** method should continuously check the value of **started** and continue once **started** is set to **true**.

Listing 19.9: The Inconsistent class

```
package app19;
public class Inconsistent {
    static boolean started = false;
    public static void main(String[] args) {
```

```
        Thread thread1 = new Thread(new Runnable() {
            public void run() {
                try {
                    Thread.sleep(3000);
                } catch (InterruptedException e) {
                }
                started = true;
                System.out.println("started set to true");
            }
        });
        thread1.start();

        while (!started) {
            // wait until started
        }

        System.out.println("Wait 3 seconds and exit");
    }

}
```

However, when I ran it in my computer, it never printed the string and exited. What happened? It looks like the **while** loop (running in the **main** method) never saw the value of **started** change.

You can remedy this by synchronizing access to **started**, as illustrated in the **Consistent** class in Listing 19.10.

Listing 19.10: The Consistent class

```
package app19;
public class Consistent {
    static boolean started = false;

    public synchronized static void setStarted() {
        started = true;
    }

    public synchronized static boolean getStarted() {
        return started;
    }

    public static void main(String[] args) {
        Thread thread1 = new Thread(new Runnable() {
            public void run() {
                try {
                    Thread.sleep(3000);
                } catch (InterruptedException e) {
                }
                setStarted();
                System.out.println("started set to true");
            }
        });
        thread1.start();

        while (!getStarted()) {
            // wait until started
```

```
        }

        System.out.println("Wait 3 seconds and exit");
    }

}
```

Note that both **setStarted** and **getStarted** are synchronized to have the desire effect. It won't work if only **setStarted** is synchronized.

However, synchronization comes at a price. Locking an object incurs runtime overhead. If what you're after is visibility and you don't need mutual exclusion, you can use the **volatile** keyword instead of **synchronized**.

Declaring a variable volatile guarantees visibility by all threads accessing the variable. Here is an example.

```
static volatile boolean started = false;
```

You can therefore rewrite the **Consistent** class to use **volatile** to reduce overhead.

Listing 19.11: Solving visibility problem with volatile

```
package app19;
public class LightAndConsistent {
    static volatile boolean started = false;

    public static void main(String[] args) {
        Thread thread1 = new Thread(new Runnable() {
            public void run() {
                try {
                    Thread.sleep(3000);
                } catch (InterruptedException e) {
                }
                started = true;
                System.out.println("started set to true");
            }
        });
        thread1.start();

        while (!started) {
            // wait until started
        }
        System.out.println("Wait 3 seconds and exit");
    }
}
```

Note that while **volatile** solves the visibility problem, it cannot be used to address a mutual exclusion issue.

Thread Coordination

There are even more delicate situations where the timing of a thread accessing an object affects other threads that need to access the same object. Such situations compel you to coordinate the threads. The following example illustrates this situation and presents a solution.

You own a courier service company that picks up and delivers goods. You

employ a dispatcher and several truck drivers. The dispatcher's job is to prepare delivery notes and place them in a delivery note holder. Any free driver checks the note holder. If a delivery note is found, the driver should perform a pick up and delivery service. If no delivery note is found, he/she should wait until there is one. In addition, to guarantee fairness you want the delivery notes to be executed in a first-in-first-out fashion. To facilitate this, you only allow one delivery note to be in the holder at a time. The dispatcher will notify any waiting driver if a new note is available in the holder.

The **java.lang.Object** class provides several methods that are useful in thread coordination:

`public final void wait() throws InterruptedException`
: Causes the current thread to wait until another thread invokes the **notify** or **notifyAll** method. **wait** normally occurs in a synchronized method and causes the calling thread that is accessing the synchronized method to place itself in the wait state and relinquish the object lock.

`public final void wait(long timeout) throws InterruptedException`
: Causes the current thread to wait until another thread invokes the **notify** or **notifyAll** method for this object, or the specified amount of time has elapsed. **wait** normally occurs in a synchronized method and causes the calling thread that is accessing the synchronized method to place itself in the wait state and relinquish the object lock.

`public final void notify()`
: Notifies a single thread that is waiting on this object's lock. If there are multiple threads waiting, one of them is chosen to be notified and the choice is arbitrary.

`pubic final void notifyAll()`
: Notifies all the threads waiting on this object's lock.

Let's see how we can implement the delivery service business model in Java using **wait**, **notify**, and **notifyAll**. There are three types of objects involved:

- **DeliveryNoteHolder**. Represents the note holder and is given in Listing 19.12. It is accessed by the **DispatcherThread** and **DriverThread**.
- **DispatcherThread**. Represents the dispatcher and is presented in Listing 19.13.
- **DriverThread**. Represents a driver, shown in Listing 19.14.

Listing 19.12: The DeliveryNoteHolder class

```
package app19;
public class DeliveryNoteHolder {
    private String deliveryNote;
    private boolean available = false;

    public synchronized String get() {
        while (available == false) {
            try {
                wait();
            } catch (InterruptedException e) { }
        }
        available = false;
        System.out.println(System.currentTimeMillis()
                + ": got " + deliveryNote);
        notifyAll();
        return deliveryNote;
```

```
    }

    public synchronized void put(String deliveryNote) {
        while (available == true) {
            try {
                wait();
            } catch (InterruptedException e) { }
        }
        this.deliveryNote = deliveryNote;
        available = true;
        System.out.println(System.currentTimeMillis() +
                ": Put " + deliveryNote);
        notifyAll();
    }
}
```

There are two synchronized methods in the **DeliveryNoteHolder** class, **get** and **put**. The **DispatcherThread** object calls the **put** method and the **DriverThread** object calls the **get** method. A delivery note is simply a **String** (**deliveryNote**) that contains delivery information. The **available** variable indicates if a delivery note is available in this holder. The initial value of **available** is **false**, denoting that the **DeliveryNoteHolder** object is empty. Note that only one thread at a time can call any of the synchronized methods.

If the **DriverThread** is the first thread that accesses **DeliveryNoteHolder**, it will encounter the following **while** loop in the **get** method:

```
        while (available == false) {
            try {
                wait();
            } catch (InterruptedException e) {
            }
        }
```

Since **available** is **false**, the thread will invoke **wait** that causes the thread to lie dormant and relinquish the lock. Now, other threads can access the **DeliveryNoteHolder** object.

If the **DispatcherThread** is the first thread that accesses **DeliveryNoteHolder**, it will see the following code:

```
        while (available == true) {
            try {
                wait();
            } catch (InterruptedException e) {
            }
        }
        this.deliveryNote = deliveryNote;
        available = true;
        notifyAll();
```

Because the value of **available** is **false**, it will skip the **while** loop and causes the **DeliveryNoteHolder** object's **deliveryNote** to be assigned a value. The thread will also set **available** to **true** and notify all waiting threads.

On the invocation of **notifyAll**, if the **DriverThread** is waiting on the **DeliveryNoteHolder** object, it will awaken, reacquire the **DeliveryNoteHolder** object's lock, escape from the **while** loop, and execute the rest of the **get** method:

```
        available = false;
        notifyAll();
        return deliveryNote;
```

The **available boolean** will be switched back to **false**, the **notifyAll** method called, and the **deliveryNote** returned.

Now, let's examine the **DispatcherThread** class in Listing 19.13.

Listing 19.13: The DispatcherThread class

```
package app19;

public class DispatcherThread extends Thread {
    private DeliveryNoteHolder deliveryNoteHolder;

    String[] deliveryNotes = { "XY23. 1234 Arnie Rd.",
            "XY24. 3330 Quebec St.",
            "XY25. 909 Swenson Ave.",
            "XY26. 4830 Davidson Blvd.",
            "XY27. 9900 Old York Dr." };

    public DispatcherThread(DeliveryNoteHolder holder) {
        deliveryNoteHolder = holder;
    }

    public void run() {
        for (int i = 0; i < deliveryNotes.length; i++) {
            String deliveryNote = deliveryNotes[i];
            deliveryNoteHolder.put(deliveryNote);
            try {
                sleep(100);
            } catch (InterruptedException e) {
            }
        }
    }
}
```

The **DispatcherThread** class extends **java.lang.Thread** and declares a **String** array that contain delivery notes to be put in the **DeliveryNoteHolder** object. It gets access to the **DeliveryNoteHolder** object from its constructor. Its **run** method contains a **for** loop that attempts to call the **put** method on the **DeliveryNoteHolder** object.

The **DriverThread** class also extends **java.lang.Thread** and is given in Listing 19.14.

Listing 19.14: The DriverThread class

```
package app19;
public class DriverThread extends Thread {
    DeliveryNoteHolder deliveryNoteHolder;
    boolean stopped = false;
    String driverName;

    public DriverThread(DeliveryNoteHolder holder, String
                driverName) {
        deliveryNoteHolder = holder;
        this.driverName = driverName;
    }
```

```
    public void run() {
        while (!stopped) {
            String deliveryNote = deliveryNoteHolder.get();
            try {
                sleep(300);
            } catch (InterruptedException e) {
            }
        }
    }
}
```

The **DriverThread** method attempts to obtain delivery notes by calling the **get** method on the **DeliveryNoteHolder** object. The **run** method employs a **while** loop controlled by the **stopped** variable. A method to change **stopped** is not given here to keep this example simple.

Finally, the **ThreadCoordinationTest** class in Listing 19.15 puts everything together.

Listing 19.15: ThreadCoordinationTest class

```
package app19;
public class ThreadCoordinationTest {
    public static void main(String[] args) {
        DeliveryNoteHolder c = new DeliveryNoteHolder();
        DispatcherThread dispatcherThread =
                new DispatcherThread(c);
        DriverThread driverThread1 = new DriverThread(c, "Eddie");
        dispatcherThread.start();
        driverThread1.start();
    }
}
```

Here is the output from running **ThreadCoordinationTest** class.:

```
1135212236001: Put XY23. 1234 Arnie Rd.
1135212236001: got XY23. 1234 Arnie Rd.
1135212236102: Put XY24. 3330 Quebec St.
1135212236302: got XY24. 3330 Quebec St.
1135212236302: Put XY25. 909 Swenson Ave.
1135212236602: got XY25. 909 Swenson Ave.
1135212236602: Put XY26. 4830 Davidson Blvd.
1135212236903: got XY26. 4830 Davidson Blvd.
1135212236903: Put XY27. 9900 Old York Dr.
1135212237203: got XY27. 9900 Old York Dr.
```

Summary

Multi-threaded application development in Java is easy, thanks to Java support for threads. To create a thread, you can extend the **java.lang.Thread** class or implement the **java.lang.Runnable** interface. In this chapter you have learned how to write programs that manipulate threads and synchronize threads. You have also learned how to write thread-safe code.

Questions

1. What is a thread?
2. What does the **synchronized** modifier do?
3. What are critical sections?

Chapter 20
Concurrency Utilities

Java's built-in support for writing multi-threaded applications, such as the **Thread** class and the **synchronized** keyword, are hard to use correctly because they are too low level. Java 5 added the Concurrency Utilities in the **java.util.concurrent** package and subpackages. The types in these packages have been designed to provide better alternatives to Java's built-in thread and synchronization features. This chapter discusses the more important types in the Concurrency Utilities, starting from atomic variables and followed by executors, **Callable**, and **Future**. Also included is a discussion of **SwingWorker**, which is a utility for writing asynchronous tasks in Swing.

Atomic Variables

The **java.util.concurrent.atomic** package provides classes such as **AtomicBoolean**, **AtomicInteger**, **AtomicLong**, and **AtomicReference**. These classes can perform various operations atomically. For example, an **AtomicInteger** stores an integer internally and offers method to atomically manipulate the integer, such as **addAndGet**, **decrementAndGet**, **getAndIncrement**, **incrementAndGet**, and so on.

The **getAndIncrement** and **incrementAndGet** methods return different results. **getAndIncrement** returns the current value of the atomic variable and then increments the value. Therefore, after executing these lines of code, the value of **a** is 0 and the value of **b** is 1.

```
AtomicInteger counter = new AtomicInteger(0);
int a = counter.getAndIncrement(); // a = 0
int b = counter.get();             // b = 1
```

The **incrementAndGet** method, on the other hand, increments the atomic variable first and returns the result. For instance, after running this snippet both **a** and **b** will have a value of 1.

```
AtomicInteger counter = new AtomicInteger(0);
int a = counter.incrementAndGet(); // a = 1
int b = counter.get();             // b = 1
```

Listing 20.1 presents a thread safe counter that utilizes **AtomicInteger**. Compare this with the thread-unsafe **UserStat** class in Chapter 23.

Listing 20.1: A counter with an AtomicInteger

```
package ch24;
import java.util.concurrent.atomic.AtomicInteger;

public class AtomicCounter {
    AtomicInteger userCount = new AtomicInteger(0);
```

```
    public int getUserCount() {
        return userCount.get();
    }

    public void increment() {
        userCount.getAndIncrement();
    }

    public void decrement() {
        userCount.getAndDecrement();
    }
}
```

Executor and ExecutorService

Whenever possible, do not use **java.lang.Thread** to execute a **Runnable** task. Instead, use an implementation of **java.util.concurrent.Executor** or its subinterface **ExecutorService**.

Executor has only one method, **execute**.

```
void execute(java.lang.Runnable task)
```

ExecutorService, an extension to **Executor**, adds termination methods and methods for executing **Callable**. **Callable** is akin to **Runnable** except that it can return a value and facilitate cancellation through the **Future** interface. **Callable** and **Future** are explained in the next section "Callable and Future."

You rarely have to write your own implementation of **Executor** (or **ExecutorService**). Instead, use one of the static methods defined in the **Executors** class, a utility class.

```
public static ExecutorService newSingleThreadExecutor()

public static ExecutorService newCacheThreadPool()

public static ExecutorService newFixedThreadPool(int numOfThreads)
```

newSingleThreadExecutor returns an **Executor** that contains a single thread. You can submit multiple tasks to the **Executor**, but only one task will be executing at any given time.

newCacheThreadPool returns an **Executor** that will create more threads to cater for multiple tasks as more tasks are submitted. This is suitable for running short lived asynchronous jobs. However, use this with caution as you could run out of memory if the **Executor** attempts to create new threads while memory is already low.

newFixedThreadPool allows you to determine how many threads will be maintained in the returned **Executor**. If there are more tasks than the number of threads, the tasks that were not allocated threads will wait until the running threads finish their jobs.

Here is how you submit a **Runnable** task to an **Executor**.

```
Executor executor = Executors.newSingleThreadExecutor();
executor.execute(new Runnable() {
    @Override
```

```
    public void run() {
        // do something
    }
});
```

Constructing a **Runnable** task as an anonymous class like this is suitable for short tasks and if you don't need to pass arguments to the task. For longer tasks or if you need to pass an argument to the task, you need to implement **Runnable** in a class.

The example in Listing 20.2 illustrates the use of **Executor**. It is a Swing application with a button and a list that will search for JPG files when the button is clicked. The results will be shown in the list. We limit the results to 200 files or we run the risk of running out of memory.

Listing 20.2: The ImageSearcher class

```
package app20.imagesearcher;
import java.awt.BorderLayout;
import java.awt.event.ActionEvent;
import java.awt.event.ActionListener;
import java.nio.file.FileSystems;
import java.nio.file.Path;
import java.util.concurrent.Executor;
import java.util.concurrent.Executors;
import java.util.concurrent.atomic.AtomicInteger;
import javax.swing.DefaultListModel;
import javax.swing.JButton;
import javax.swing.JFrame;
import javax.swing.JList;
import javax.swing.JScrollPane;

public class ImageSearcher extends JFrame
        implements ActionListener {
    public static final int MAX_RESULT = 300;
    JButton searchButton = new JButton("Search");
    DefaultListModel listModel;
    JList imageList;
    Executor executor = Executors.newFixedThreadPool(10);
    AtomicInteger fileCounter = new AtomicInteger(1);

    public ImageSearcher(String title) {
        super(title);
        init();
    }

    private void init() {
        this.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
        this.setLayout(new BorderLayout());
        this.add(searchButton, BorderLayout.NORTH);
        listModel = new DefaultListModel();
        imageList = new JList(listModel);
        this.add(new JScrollPane(imageList), BorderLayout.CENTER);
        this.pack();
        this.setSize(800, 650);
        searchButton.addActionListener(this);
        this.setVisible(true);
        // center frame
```

```
        this.setLocationRelativeTo(null);
    }

    private static void constructGUI() {
        JFrame.setDefaultLookAndFeelDecorated(true);
        ImageSearcher frame = new ImageSearcher("Image Searcher");
    }

    public void actionPerformed(ActionEvent e) {
        Iterable<Path> roots =
                FileSystems.getDefault().getRootDirectories();
        for (Path root : roots) {
            executor.execute(new ImageSearchTask(root, executor,
                    listModel, fileCounter));
        }
    }

    public static void main(String[] args) {
        javax.swing.SwingUtilities.invokeLater(new Runnable() {
            public void run() {
                constructGUI();
            }
        });
    }
}
```

Look carefully at the **actionPerformed** method:

```
        Iterable<Path> roots =
                FileSystems.getDefault().getRootDirectories();
        for (Path root : roots) {
            executor.execute(new ImageSearchTask(root, executor,
                    listModel, fileCounter));
        }
```

The **FileSystem.getRootDirectories** method returns the roots of the file system. If you're on Windows, then it will return Drive C, Drive D, and so on. If you're using Linux or Mac, then it will return /. Notice how it creates an **ImageSearchTask** instance and pass it to the **Executor**? It passes a root directory, the **Executor**, a **DefaultListModel** object that the task can access, and an **AtomicInteger** that records how many files have been found.

The **ImageSearchTask** class in Listing 20.3 is an implementation of **Runnable** for searching JPG files in the given directory and its subdirectories. Note that for each subdirectory it spawns a new **ImageSearchTask** and submits it to the passed in **Executor**.

Listing 20.3: The ImageSearchTask class

```
package app20.imagesearcher;

import java.io.IOException;
import java.nio.file.DirectoryStream;
import java.nio.file.Files;
import java.nio.file.Path;
import java.util.concurrent.Executor;
import java.util.concurrent.atomic.AtomicInteger;
```

```
import javax.swing.DefaultListModel;
import javax.swing.SwingUtilities;

public class ImageSearchTask implements Runnable {
    private Path searchDir;
    private Executor executor;
    private DefaultListModel listModel;
    private AtomicInteger fileCounter;

    public ImageSearchTask(Path searchDir, Executor executor,
       DefaultListModel listModel,
            AtomicInteger fileCounter) {
        this.searchDir = searchDir;
        this.executor = executor;
        this.listModel = listModel;
        this.fileCounter = fileCounter;
    }

    @Override
    public void run() {
        if (fileCounter.get() > ImageSearcher.MAX_RESULT) {
            return;
        }
        try (DirectoryStream<Path> children =
                Files.newDirectoryStream(searchDir)) {
            for (final Path child : children) {
                if (Files.isDirectory(child)) {
                    executor.execute(new ImageSearchTask(child,
                            executor, listModel, fileCounter));
                } else if (Files.isRegularFile(child)) {
                    String name = child.getFileName()
                            .toString().toLowerCase();
                    if (name.endsWith(".jpg")) {
                        final int fileNumber =
                                fileCounter.getAndIncrement();
                        if (fileNumber > ImageSearcher.MAX_RESULT){
                            break;
                        }

                        SwingUtilities.invokeLater(new Runnable() {
                            public void run() {
                                listModel.addElement(fileNumber +
                                        ": " + child);
                            }
                        });
                    }
                }
            }
        } catch (IOException e) {
            System.out.println(e.getMessage());
        }
    }
}
```

The **run** method inspects the directory passed to the task and checks its content. For each JPG file it increments the **fileCount** variable and for each subdirectory it

spawns a new **ImageSearchTask** so that the search can be done more quickly.

Callable and Future

Callable is one of the most valuable members of the Concurrency Utilities. A **Callable** is a task that returns a value and may throw an exception. **Callable** is similar to **Runnable**, except that the latter cannot return a value or throw an exception.

Callable defines a method, **call**:

```
V call() throws java.lang.Exception
```

You can pass a **Callable** to an **ExecutorService**'s **submit** method:

```
Future<V> result = executorService.submit(callable);
```

The **submit** method returns a **Future** which can be used to cancel the task or retrieve the return value of the **Callable**. To cancel a task, call the **cancel** method on the **Future** object:

```
boolean cancel(boolean myInterruptIfRunning)
```

You pass **true** to **cancel** if you want to cancel the task even though it's being executed. Passing **false** allows an in-progress task to complete undisturbed. Note that **cancel** will fail if the task has been completed or previously cancelled or for some reason cannot be cancelled.

To get the result of a **Callable**, call the **get** method of the corresponding **Future**. The **get** method comes in two overloads:

```
V get()
```

```
V get(long timeout, TimeUnit unit)
```

The first overload blocks until the task is complete. The second one waits until a specified time lapses. The *timeout* argument specifies the maximum time to wait and the *unit* argument specifies the time unit for *timeout*.

To find out if a task has been cancelled or complete, call **Future**'s **isCancelled** or **isDone** method.

```
boolean isCancelled()
```

```
boolean isDone()
```

For example, the **FileCountTask** class in Listing 20.4 presents a **Callable** task for counting the number of files in a directory and its subdirectories.

Listing 20.4: The FileCountTask class

```
package app20.filecounter;
import java.io.IOException;
import java.nio.file.DirectoryStream;
import java.nio.file.Files;
import java.nio.file.Path;
import java.nio.file.Paths;
import java.util.ArrayList;
import java.util.List;
import java.util.concurrent.Callable;

public class FileCountTask implements Callable {
    Path dir;
    long fileCount = 0L;
    public FileCountTask(Path dir) {
        this.dir = dir;
    }
```

```
    private void doCount(Path parent) {
        if (Files.notExists(parent)) {
            return;
        }
        try (DirectoryStream<Path> children =
                    Files.newDirectoryStream(parent)) {
            for (Path child : children) {
                if (Files.isDirectory(child)) {
                    doCount(child);
                } else if (Files.isRegularFile(child)) {
                    fileCount++;
                }
            }
        } catch (IOException e) {
            e.printStackTrace();
        }
    }

    @Override
    public Long call() throws Exception {
        System.out.println("Start counting " + dir);
        doCount(dir);
        System.out.println("Finished counting " + dir);
        return fileCount;
    }
}
```

The **FileCounter** class in Listing 20.5 uses **FileCountTask** to count the number of files in two directories and prints the results. It specifies a **Path** array (**dirs**) that contains the paths to the directories which you want to count the number of files of. Replace the values of **dirs** with directory names in your file system.

Listing 20.5: The FileCounter class

```
package app20.filecounter;
import java.nio.file.Path;
import java.nio.file.Paths;
import java.util.concurrent.ExecutionException;
import java.util.concurrent.ExecutorService;
import java.util.concurrent.Executors;
import java.util.concurrent.Future;

public class FileCounter {
    public static void main(String[] args) {
        Path[] dirs = {
            Paths.get("C:/temp"),
            Paths.get("C:/temp/data")
        };

        ExecutorService executorService =
                Executors.newFixedThreadPool(dirs.length);

        Future<Long>[] results = new Future[dirs.length];
        for (int i = 0; i < dirs.length; i++) {
            Path dir = dirs[i];
```

```
            FileCountTask task = new FileCountTask(dir);
            results[i] = executorService.submit(task);
        }

        // print result
        for (int i = 0; i < dirs.length; i++) {
            long fileCount = 0L;
            try {
                fileCount = results[i].get();
            } catch (InterruptedException | ExecutionException ex){
                ex.printStackTrace();
            }
            System.out.println(dirs[i] + " contains "
                    + fileCount + " files.");
        }

        // it won't exit unless we shut down the ExecutorService
        executorService.shutdownNow();
    }
}
```

When run, the **FileCounter** class creates the same number of threads as the number of directories in **dirs** using the **newFixedThreadPool** method of **ExecutorService**. One thread for each directory.

```
        ExecutorService executorService =
                Executors.newFixedThreadPool(dirs.length);
```

It also defines an array of **Future**s for containing the results of executing the **FileCountTask** tasks.

```
        Future<Long>[] results = new Future[dirs.length];
```

It then creates a **FileCountTask** for each directory and submits it to the **ExecutorService**.

```
        for (int i = 0; i < dirs.length; i++) {
            Path dir = dirs[i];
            FileCountTask task = new FileCountTask(dir);
            results[i] = executorService.submit(task);
        }
```

Finally, it prints the results and shuts down the **ExecutorService**.

```
        // print result
        for (int i = 0; i < dirs.length; i++) {
            long fileCount = 0L;
            try {
                fileCount = results[i].get();
            } catch (InterruptedException | ExecutionException ex){
                ex.printStackTrace();
            }
            System.out.println(dirs[i] + " contains "
                    + fileCount + " files.");
        }
        // it won't exit unless we shut down the ExecutorService
        executorService.shutdownNow();
```

Locks

In Chapter 19, “Java Threads” you have learned that you can lock a shared resource using the **synchronized** modifier. While **synchronized** is easy enough to use, such a locking mechanism is not without limitations. For instance, a thread attempting to acquire such a lock cannot back off and will block indefinitely if the lock cannot be acquired. Also, locking and unlocking are limited to methods and blocks; there's no way to lock a resource in a method and release it in another method.

Luckily, the Concurrency Utilities comes with much more advanced locks. The **Lock** interface, the only one discussed in this book, offers methods that overcome the limitations of Java's built-in locks. Lock comes with the **lock** method as well as the **unlock** method. This means, you can release a lock anywhere in the program as long as you retain a reference to the lock. In most circumstances, however, it is a good idea to call **unlock** in a finally clause following the invocation of **lock** to make sure unlock is always called.

```
aLock.lock();
try {
    // do something with the locked resource
} finally {
    aLock.unlock();
}
```

If a lock is not available, the **lock** method will block until it is. This behavior is similar to the implicit lock resulting from using **synchronized**.

In addition to **lock** and **unlock**, however, the **Lock** interface offers the **tryLock** methods:

```
boolean tryLock()
```

```
boolean tryLock(long time, TimeUnit timeUnit)
```

The first overload returns **true** only if the lock is available. Otherwise, it returns false. In the latter case, it does not block.

The second overload returns **true** immediately if the lock is available. Otherwise, it will wait until the specified time lapses and will return false if it fails to acquire the lock. The *time* argument specifies the maximum time it will wait and the *timeUnit* argument specifies the time unit for the first argument.

The code in Listing 20.8 shows the use of **ReentrantLock**, an implementation of **Lock**. This code is taken from a document management suite that allows users to upload and share files. Uploading a file with the same name as an existing file in the same server folder will make the existing file a history file and the new file the current file.

To improve performance, multiple users are allowed to upload files at the same time. Uploading files with different names or to different server folders poses no problem as they will be written to different physical files. Uploading files with the same name to the same server folder can be a problem if the users do it at the same time. To circumvent this issue, the system uses a **Lock** to ensure multiple threads attempting to write to the same physical file do not do so concurrently. In other words, only one thread can do the writing and other threads will have to wait until the first one is done.

In Listing 20.6, which in fact is real code taken from the document

management package by Brainy Software, the system uses a **Lock** to protect access to a file and obtains locks from a thread-safe map that maps paths with locks. As such, it only prevents writing files with the same name. Writing files with different names can occur at the same time because different paths maps to different locks.

Listing 20.6: Using locks to prevent threads writing to the same file

```
ReentrantLock lock = fileLockMap.putIfAbsent(fullPath,
        new ReentrantLock());
lock.lock();
try {

    // index and copy the file, create history etc

} finally {
    lock.unlock();
    fileLockMap.remove(fullPath, lock);
}
```

The code block starts by attempting to obtain a lock from a thread-safe map. If a lock is found, it means another thread is accessing the file. If no lock is found, the current thread creates a new **ReentrantLock** and stores it in the map so that other threads will notice that it's currently accessing the file.

```
ReentrantLock lock = fileLockMap.putIfAbsent(fullPath,
        new ReentrantLock());
```

It then calls **lock**. If the current thread is the only thread trying to acquire the lock, the **lock** method will return. Otherwise, the current thread will wait until the lock holder releases the lock.

Once a thread successfully obtains a lock, it has exclusive access to the file and can do anything with it. Once it's finished, it calls **unlock** and the map's **remove** method. The **remove** method will only removes the lock if no thread is holding it.

Summary

The Concurrency Utilities are designed to make writing multi-threaded applications easier. The classes and interfaces in the API are meant to replace Java's lower-level threading mechanism such as the **Thread** class and the **synchronized** modifier. This chapter discussed the basics of the Concurrency Utilities, including atomic variables, executors, **Callable**, and **Future**.

Questions

1. What are atomic variables?
2. How do you obtain an **ExecutorService** instance?
3. What is a **Callable** and what is a **Future**?
4. Name one of the standard implementations of the **Lock** interface.

Chapter 21
Introduction to Android

Android is the most popular mobile platform today, and you will learn how to create applications and use the Android APIs in the examples that accompany this book.

The set of APIs that ship with the Android software development kit (SDK) are comprehensive. With these APIs you can easily use user interface (UI) components, play and record audio and video, create games and animation, store and retrieve data, search the Internet, and so on.

This chapter provides an overview of the Android platform and contains instructions for installing the tool you need for Android development, the ADT Bundle.

Overview

Today Android rules the smartphone and tablet world. One of the reasons for its rapid ascent to the top is the fact that it uses Java as its programming language. But, is Android really Java? The answer is yes and no. Yes, Java is the default programming language for Android application development. No, Android applications do not run on a Java Virtual Machine as all Java applications do. Instead, Android applications run on a virtual machine called Dalvik.

After an Android program is compiled to Java bytecode, the bytecode is then cross-compiled to a dex (Dalvik executable) file that contains one or multiple Java classes. The dex file, the manifest file (an XML file that describes the application), and all resource files are then packaged using the apkbuilder tool into an apk file, which is basically a zip file that can be extracted using unzip or Winzip. APK, by the way, stands for application package.

The generated apk file can run on the target Android device or on an emulator. Deploying an Android application is easy. You can make the apk file available for download and download it with an Android device to install it. You can also email the apk file to yourself and open the email on an Android device and install it. To publish your application on Google Play, however, you need to sign the apk file using the jarsigner tool.

If you're interested in learning more, this web page explains the Android build process in detail.

`https://developer.android.com/tools/building/index.html`

Version	Code Name	API Level	Release Date
1.0		1	September 23, 2008
1.1		2	February 9, 2009
1.5	Cupcake	3	April 30, 2009
1.6	Donut	4	September 15, 2009
2.0	Eclair	5	October 26, 2009
2.0.1	Eclair	6	December 3, 2009
2.1	Eclair	7	January 12, 2010
2.2	Froyo	8	May 20, 2010
2.2.1	Froyo	8	January 18, 2011
2.2.2	Froyo	8	January 22, 2011
2.2.3	Froyo	8	November 21, 2011
2.3	Gingerbread	9	December 6, 2010
2.3.1	Gingerbread	9	December 2010
2.3.2	Gingerbread	9	January 2011
2.3.3	Gingerbread	10	February 9, 2011
2.3.4	Gingerbread	10	April 28, 2011
2.3.5	Gingerbread	10	July 25, 2011
2.3.6	Gingerbread	10	September 2, 2011
2.3.7	Gingerbread	10	September 21, 2011
3.0	Honeycomb	11	February 22, 2011
3.1	Honeycomb	12	May 10, 2011
3.2	Honeycomb	13	July 15, 2011
3.2.1	Honeycomb	13	September 20, 2011
3.2.2	Honeycomb	13	August 30, 2011
3.2.3	Honeycomb	13	---
3.2.4	Honeycomb	13	December 2011
3.2.5	Honeycomb	13	January 2012
3.2.6	Honeycomb	13	February 2012
4.0	Ice Cream Sandwich	14	October 19, 2011
4.0.1	Ice Cream Sandwich	14	October 21, 2011
4.0.2	Ice Cream Sandwich	14	November 28, 2011
4.0.3	Ice Cream Sandwich	15	December 16, 2011
4.0.4	Ice Cream Sandwich	15	March 29, 2012
4.1	Jelly Bean	16	July 9, 2012
4.1.1	Jelly Bean	16	July 23, 2012
4.1.2	Jelly Bean	16	October 9, 2012
4.2	Jelly Bean	17	November 13, 2012
4.2.1	Jelly Bean	17	November 27, 2012
4.2.2	Jelly Bean	17	February 11, 2013
4.3	Jelly Bean	18	July 24, 2013
4.3.1	Jelly Bean	18	October 3, 2013
4.4	Kitkat	19	October 31, 2013
4.4.1	Kitkat	19	December 5, 2013
4.4.2	Kitkat	19	December 9, 2013

Table 21.1: Android versions

Versions of Android

First released in September 2008, Android is now a stable and mature platform.

The current version, version 4.4, is the 19th Android API level ever released. Table 21.1 shows the code name, API level, and release date for each Android version.With each new version, new features are added. As such, you can use the most features if you develop applications that target the latest release. However, not every Android phone and tablet has the latest release because Android devices made for older APIs may not be able to support later releases and software upgrade is not always automatic. Table 21.2 shows Android versions still in use today.

Version	Codename	API	Distribution
2.2	Froyo	8	1.0%
2.3.3-2.3.7	Gingerbread	10	16.2%
3.2	Honeycomb	13	0.1%
4.0.3-4.0.4	Ice Cream Sandwich	15	13.4%
4.1.x	Jelly Bean	16	33.5%
4.2.x		17	18.8%
4.3		18	8.5%
4.4	KitKat	19	8.5%

Table 21.2: Android versions still in use (May 2014)

The data in Table 21.2 comes from this web page.

`https://developer.android.com/about/dashboards/index.html`

If you distribute your application through Google Play, the most popular marketplace for Android applications, the lowest version of Android that can download your application is 2.2, because versions older than 2.2 cannot access Google Play. In general you would want to reach as wide customer base as possible, which means supporting version 2.2 and up. If you only support version 3.0 and up, for example, you leave out 17% of Android users.

However, the lower the version, the fewer features it supports, which can be limiting. Some people risk alienating some customers in order to be able to use the more recent features.

Android is backward compatible, which means an application targeting a lower API level can always be run on a device with a higher API level.

Which Java Versions Can I Use?

To develop Android applications using tools such as Android Studio or ADT Eclipse, you need JDK 6 or 7. Support for JDK 7 was only added in late 2013. At the time of writing, there is no set date yet for supporting JDK 8. If you have JDK 8 you can still use it to write Android apps. You just cannot use features that are only available in Java 8.

Online Reference

The first challenge facing new Android programmers is understanding the components available in Android. There are four application components that any Android programmer needs to know: activity, service, content provider, and broadcast receiver. They are all unique and learning each of them takes time.

Luckily, documentation is in abundance and it is easy to find help over the Internet. The documentation of all Android classes and interfaces can be found on

Android's official website:

`http://developer.android.com/reference/packages.html`

Undoubtedly, you will frequent this website as long as you work with Android. If you had a chance to browse the website, you'd have learned that the first batch of types belong to the **android** package and its subpackages. After them come the **java** and **javax** packages that you can use in Android applications. Java packages that cannot be used in Android, such as **javax.swing**, are not listed there.

Downloading and Installing Android Development Tools

To develop Android applications, you need the Android software development kit (SDK) to compile, test, and debug your applications. In addition, an integrated development environment (IDE) will help expedite development. Fortunately, the good people at Google provide two bundles that include everything you need to develop your next applications:

- The Android Developer Tools (ADT) Bundle, a bundle that includes the SDK and a version of Eclipse
- Android Studio, an Android IDE based on IntelliJ IDEA

At the time of writing Android Studio is still in alpha version and not guaranteed to be stable. Therefore, in this book the ADT Bundle is used.

Alternatively, if you already have Eclipse on your local machine, you can install the ADT plug-in only and work with your existing Eclipse. However, note that it is easier to install the ADT bundle. If you choose to install the ADT plug-in, information on how to proceed with it can be found here.

`http://developer.android.com/sdk/installing/installing-adt.html`

This book focuses on using the ADT Bundle. Therefore, if you have not done so, please download the ADT bundle from this site.

`http://developer.android.com/sdk/index.html`

Unpack the downloaded package to your workspace. The main directory will contain two folders, **eclipse** and **sdk**. Navigate to the **eclipse** folder and double-click the Eclipse program to start it. You will be asked to select a workspace. After that, the Eclipse IDE will open. The main window is shown in Figure 21.1. Note that the application icon of ADT Eclipse is different from that of "regular" Eclipse.

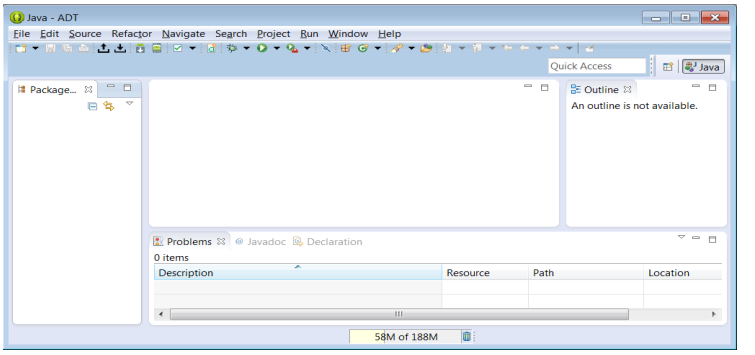

Figure 21.1: The ADT window

Chapter 22
Your First Android Application

This chapter shows how you can create an Android application using the ADT Bundle. It also explains how to setup an emulator so you can develop, test, debug, and run Android applications even if you do not have a real Android device.

Android development requires that you have a Java Development Kit (JDK) installed on your computer. If you do not have a JDK installed, make sure you download and install one by following the instructions in Introduction. It is also assumed that you have downloaded and installed the ADT Bundle, the process of which was also explained in Chapter 21, "Introduction to Android."

Creating An Application

Creating an Android application with the ADT Bundle is as easy as a few mouse clicks. This section shows how to create a Hello World application, package it, and run it on an emulator. Make sure you have installed the ADT Bundle by following the instructions in Introduction.

Next, follow these steps.

1. Click the New menu in Eclipse and select **Android Application Project**. Note that in this book Eclipse refers to the version of Eclipse included in the ADT Bundle or Eclipse with the ADT plug-in installed. The **New Android Application** window will open as shown in Figure 22.1.

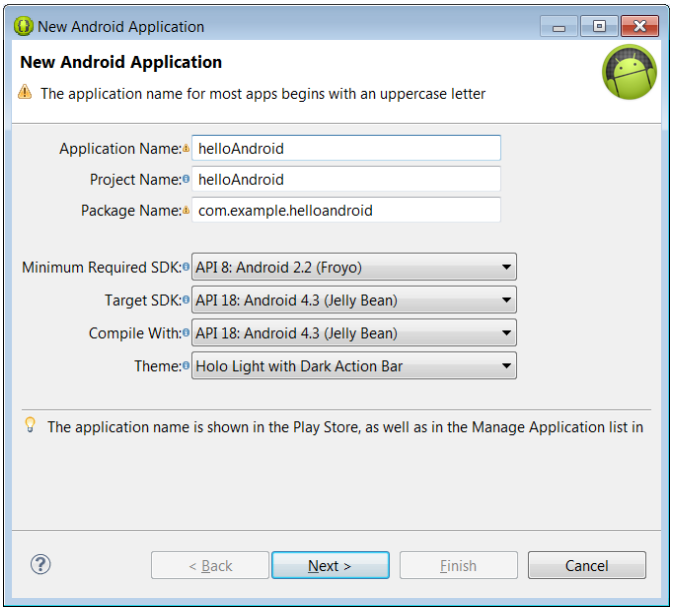

Figure 22.1: The New Android Application window

2. Type in the details of the new application. In the **Application Name** field, enter the name you want your application to appear on the Android device. In the **Project Name** field, type a name for your project. This can be the same as the application name or a different name. Then, enter a Java package name in the **Package Name** field. The package name will uniquely identify your application. Even though you can use any string that qualifies as a Java package, the package name should be your domain name in reverse order. For example, if your domain name is example.com, your package name should be **com.example**, followed by the project name.
   Now, right under the text boxes are four dropdown boxes. The **Minimum Required SDK** dropdown contains a list of Android SDK levels. The lower the level, the more devices your application can run on, but the fewer APIs and features you can use. The **Target SDK** box should be given the highest API level your application will be developed and tested against. The **Compile With** dropdown should contain the target API to compile your code against. Finally, the **Theme** dropdown should contain a theme for your application.
   For your first application, use the same values as those shown in Figure 22.1.
3. Click **Next**. You will see a window similar to the one in Figure 22.2. Accept the default settings.

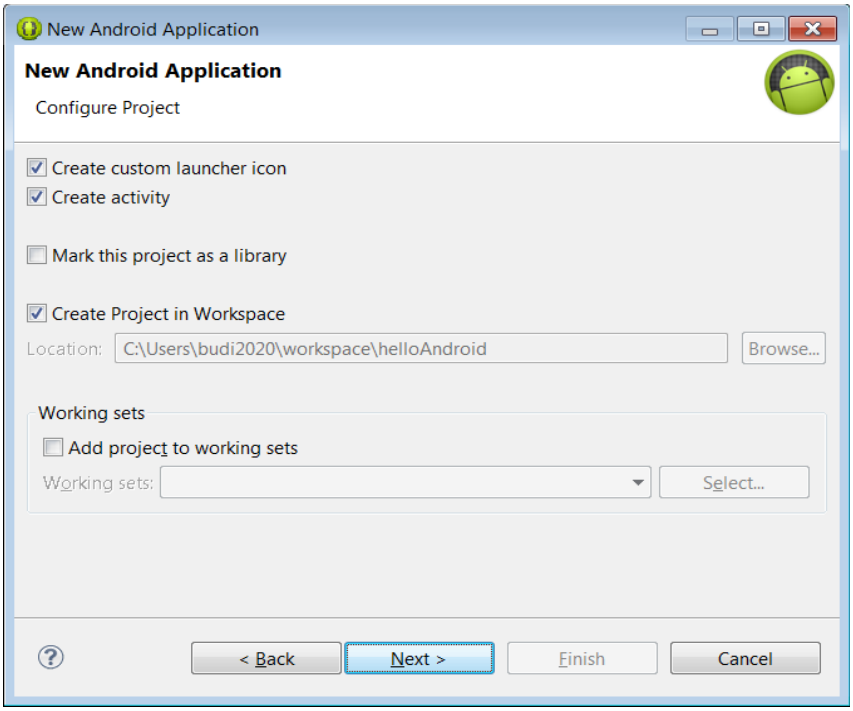

Figure 22.2: Configuring your application

4. Click **Next** again. The next window that appears will look like the window in Figure 22.3. Here you can choose an icon for your application. If you don't like the default image icon, click **Clipart** and select one from the list. In addition, you can use text as your icon if you so wish.

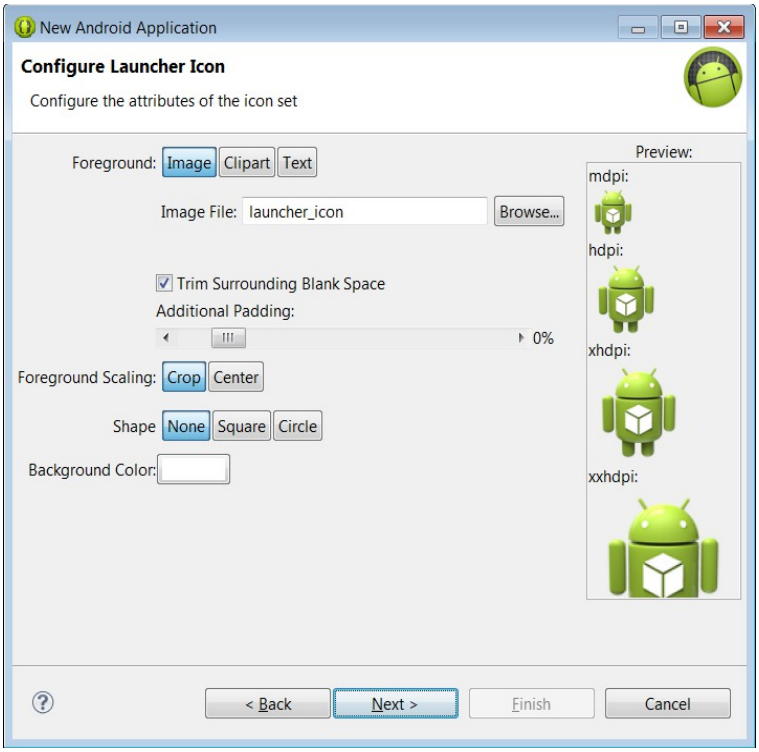

Figure 22.3: Selecting a launcher icon

5. Click **Next** again and you will be prompted to select an activity (See Figure 22.4). The activity will be explained in Chapter 23, “Activities.” For now, leave **Blank Activity** selected.

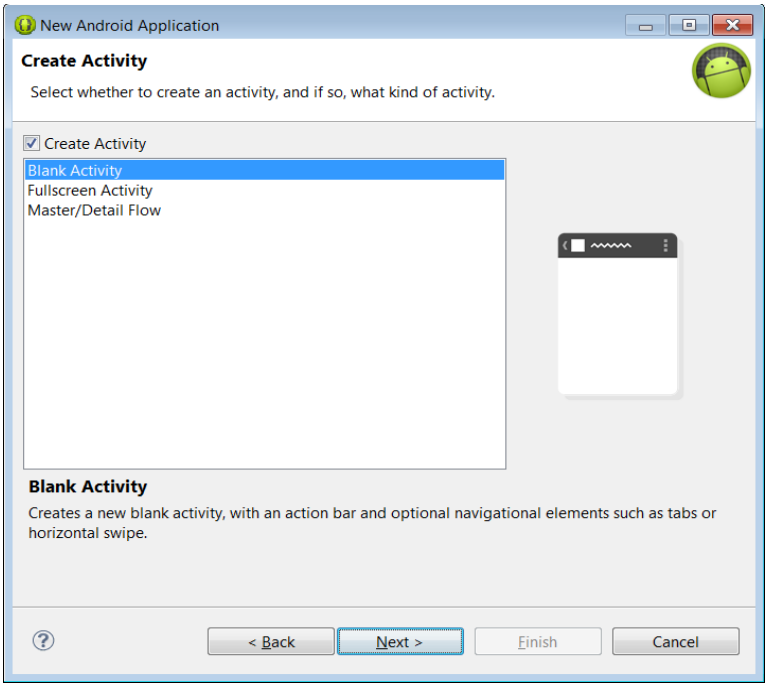

Figure 22.4: Selecting an activity type

6. Click **Next** one more time. The next window will appear as shown in Figure 22.5.

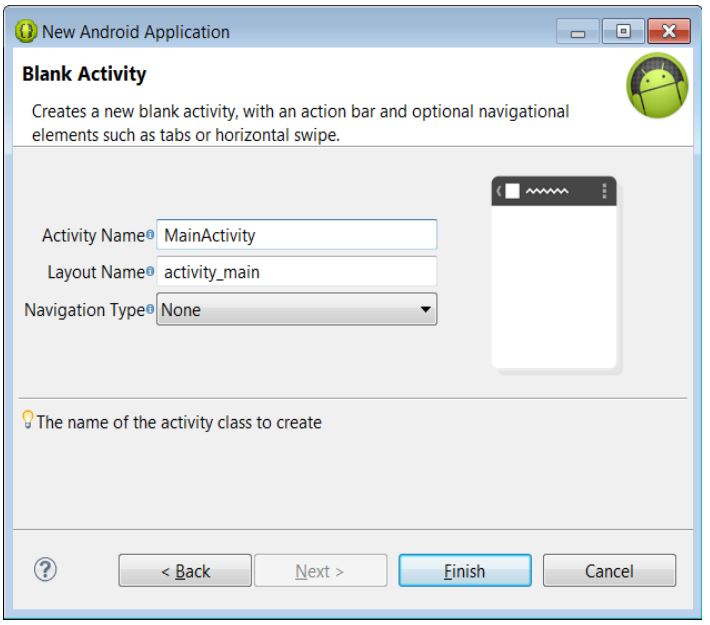

Figure 22.5: Entering the activity and layout names

7. Accept the suggested activity and layout names and click **Finish**. The ADT Bundle will create your application and you'll see your project like the

screenshot in Figure 22.6.

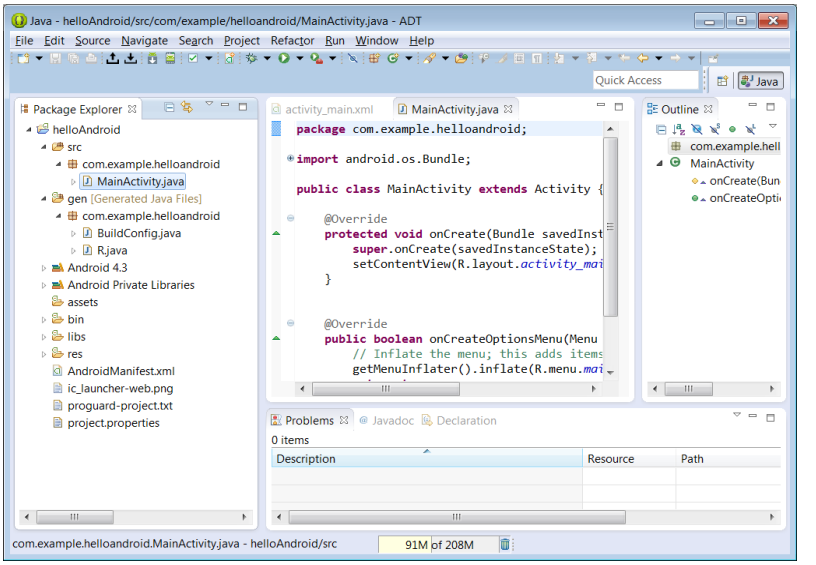

Figure 22.6: The new Android project

In the root directory of Eclipse's Package Explorer (on the left), you'll find the following files:

- **AndroidManifest.xml** file. This is an XML document that describes your application. It will be explained in more detail in the next section "The Android Manifest."
- An icon file in PNG format.
- A **project.properties** file that specifies the Android target API level.

On top of that, there are the following folders.

- **src**. This is your source code folder.
- **gen**. This is where generated Java classes are kept. The generated Java classes allow your Java source to use values defined in the layout file and other resource files. You should not edit generated files yourself.
- **bin**. This is where the project build will be saved in. The application APK will also be found here after you have run your application successfully.
- **libs**. Contains Android library files.
- **res**. Contains resource files. Underneath this directory are these directories: **drawable-*xxx*** (containing images for various screen resolutions), **layout** (containing layout files), **menu** (containing menu files), and **values** (containing string and other values).

One of the advantages of developing Android applications with an IDE, such as ADT Eclipse, it knows when you add a resource under the res directory and responds by updating the **R** generated class so that you can easily load the resource from your program. You will learn this powerful feature in the chapters to come.

The Android Manifest

Every Android application must have a manifest file called **AndroidManifest.xml** file that describes the application. Listing 22.1 shows a sample manifest file.

Listing 22.1: A sample manifest

```
<?xml version="1.0" encoding="utf-8"?>
<manifest
       xmlns:android="http://schemas.android.com/apk/res/android"
    package="com.example.helloworld"
    android:versionCode="1"
    android:versionName="1.0" >
    <uses-sdk
        android:minSdkVersion="8"
        android:targetSdkVersion="17" />
    <application
        android:allowBackup="true"
        android:icon="@drawable/ic_launcher"
        android:label="@string/app_name"
        android:theme="@style/AppTheme" >
        <activity
            android:name="com.example.helloworld.MainActivity"
            android:label="@string/app_name" >
            <intent-filter>
                <action android:name="android.intent.action.MAIN"/>
                <category
android:name="android.intent.category.LAUNCHER" />
            </intent-filter>
        </activity>
    </application>
</manifest>
```

A manifest file is an XML document with **manifest** as the root element. The **package** attribute of the **manifest** element specifies a unique identifier for the application. Android tools will also use this information to generate appropriate Java classes that are used from the Java source you write.

Under **<manifest>** are **uses-sdk** and **application** elements. **uses-sdk** defines the minimum and maximum SDK levels supported. In this example, the maximum is SDK level 17, which corresponds to Android 4.2.

The **application** element describes the application. Among others, it contains one or more **activity** elements that each describes an activity. An application also must have one main activity that serves as the entry point to the application. The main activity contains an **intent-filter** element with **MAIN** action and **LAUNCHER** category.

There are other elements that may appear in the Android manifest and you can find the complete list here.

`http://developer.android.com/guide/topics/manifest/manifest-element.html`

Running An Application on An Emulator

The ADT Bundle comes with an emulator to run your applications on if you don't have a real device. The following are the steps for running your application on an emulator.

1. Click the Android project on the Eclipse Project Explorer, then click **Run → Run As → Android Application**.

2. The **Android Device Chooser** window will pop up (see Figure 22.7). (Once you configure it, it will not appear the next time you try to run your application).

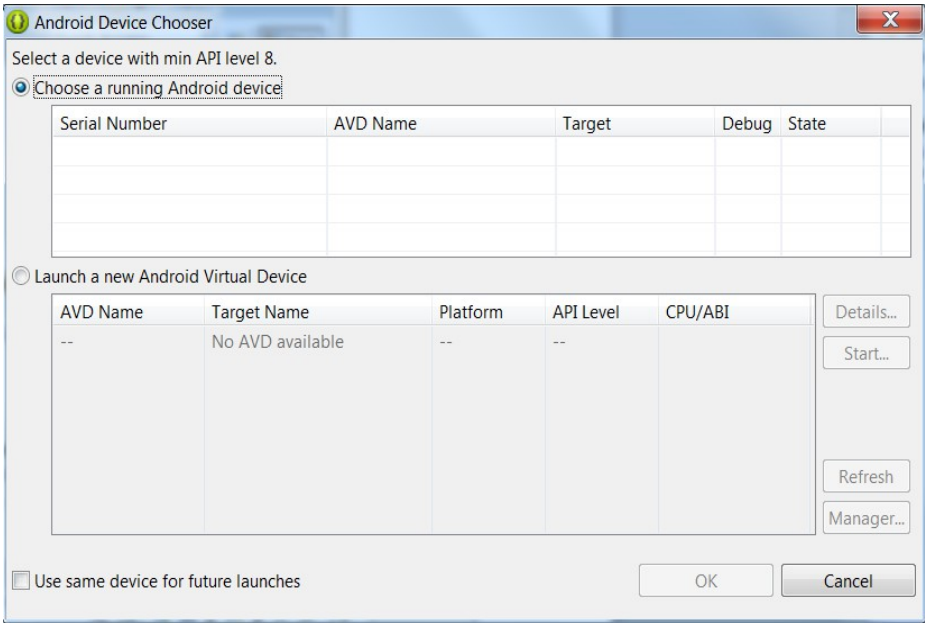

Figure 22.7: The Android Device Chooser window

3. Here you can choose to run your application on a real Android device (an Android phone or tablet) or an Android Virtual Device (emulator). In Figure 22.7 you do not see a running Android device because no real device is connected, so click the **Launch a new Android Virtual Device** radio button, and click the **Manager** button on the right. The **Android Virtual Device Manager** window will appear (See Figure 22.8).

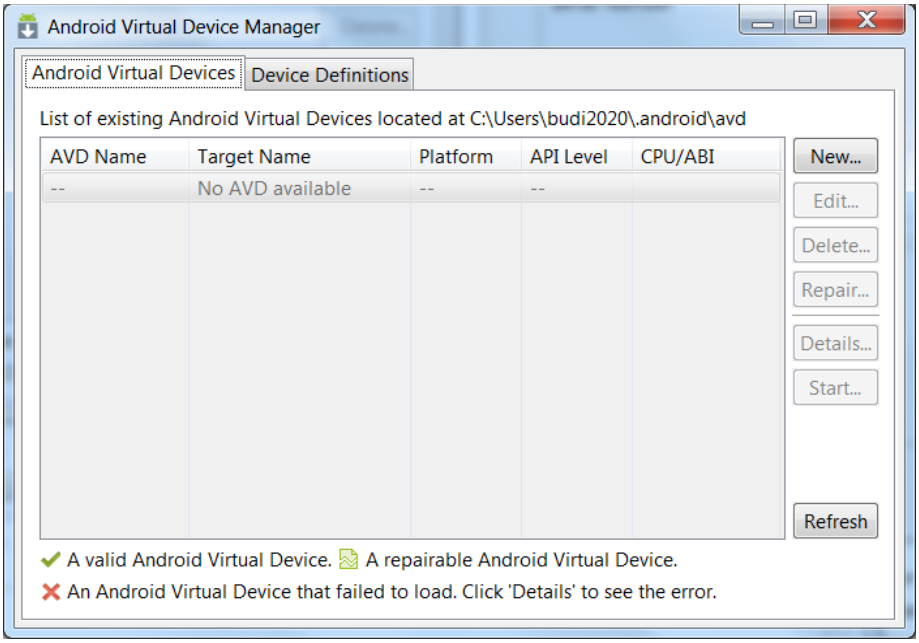

Figure 22.8: Android Virtual Device Manager

4. Click **New** on the **Android Virtual Devices** pane to display the **Create new AVD** windows (See Figure 22.9)

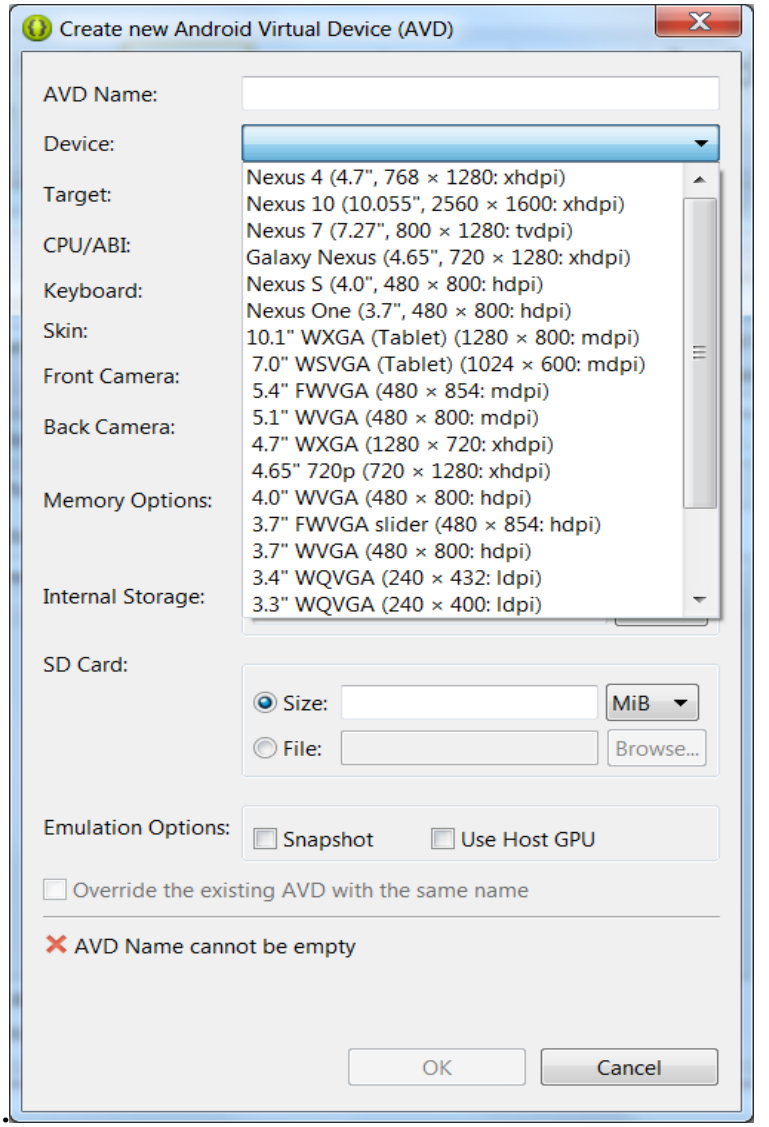

Figure 22.9: Creating a new virtual device

5. Click the **Device** drop-down to view the list of virtual devices available. Here I choose Nexus 7. Then, give your device a name. The name must not contain spaces or any special characters.
6. Choose a target and if you're using Windows, reduce the RAM to 768. For some reason, it may crash if you're using more than 768MB RAM on Windows.
7. My options are shown in the screenshot in Figure 22.10.

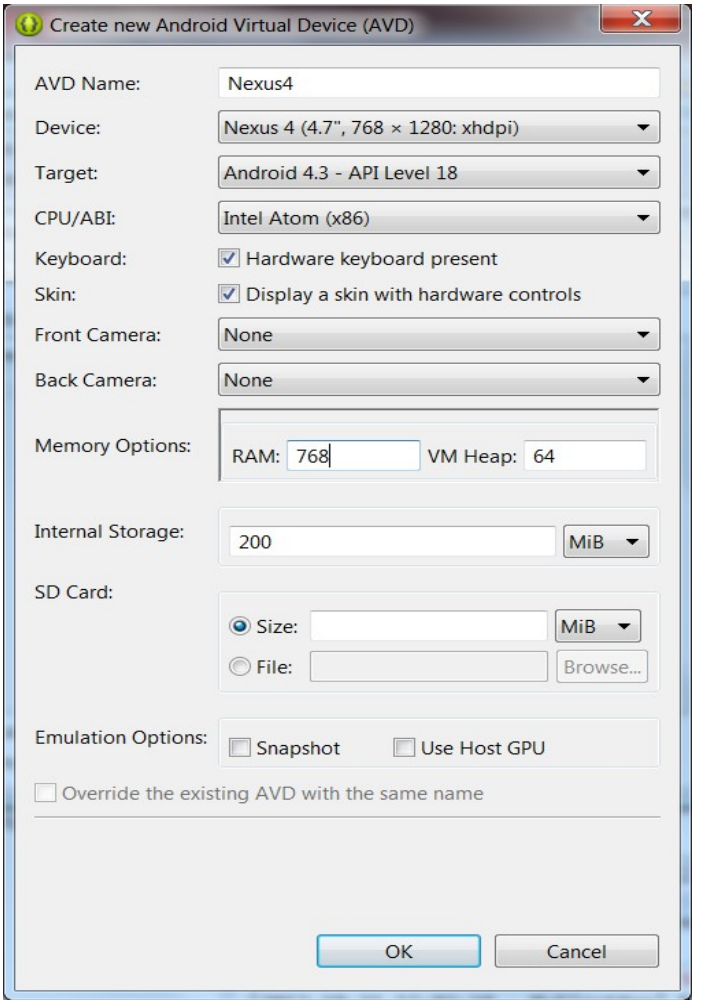

Figure 22.10: Entering values for a new virtual device

8. Click **OK**. The **Create new Android Virtual Device (AVD)** window will close and you'll be back at the **Android Virtual Device Manager** window. Your AVD will be listed there, as shown in Figure 22.11.

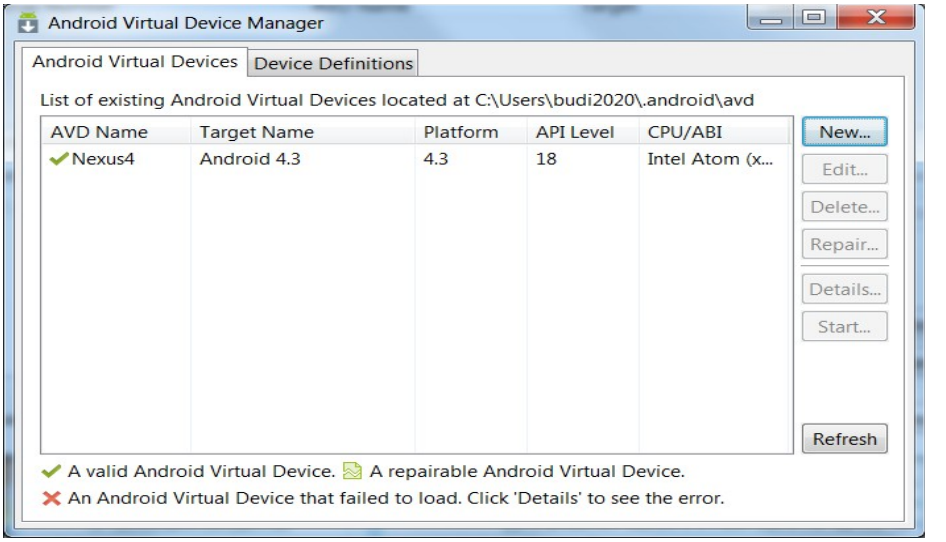

Figure 22.11: The list of virtual devices available

9. Now, click the AVD name (Nexus7) to select it and the **Start** and other buttons will be enabled. Click the **Start** button to start the AVD. You will see the Launch Options popup like that in Figure 22.12.

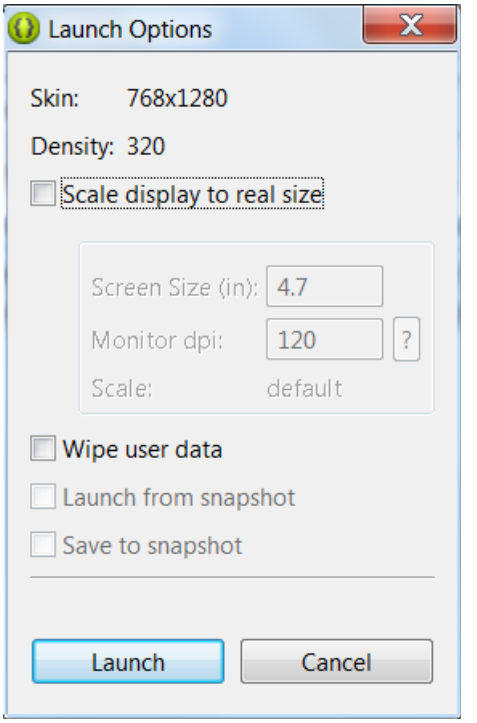

Figure 22.12: The Launch Options popup

10. Click **Launch** to launch your AVD. You'll see a window like that in Figure 22.13 when it's launching.

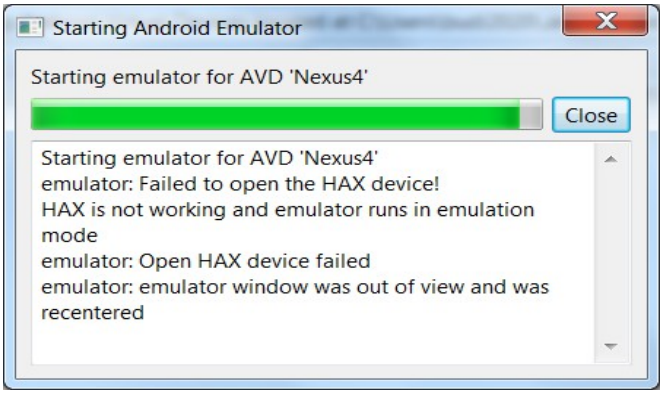

Figure 22.13: Starting the emulator

It will take a few minutes or more depending on your computer speed. To illustrate, on my Intel i5-based Windows machine, it takes eight minutes, but on my i7 Ubuntu machine with an SSD drive, it takes less than a minute. Figure 22.14 shows the emulator when it is ready.

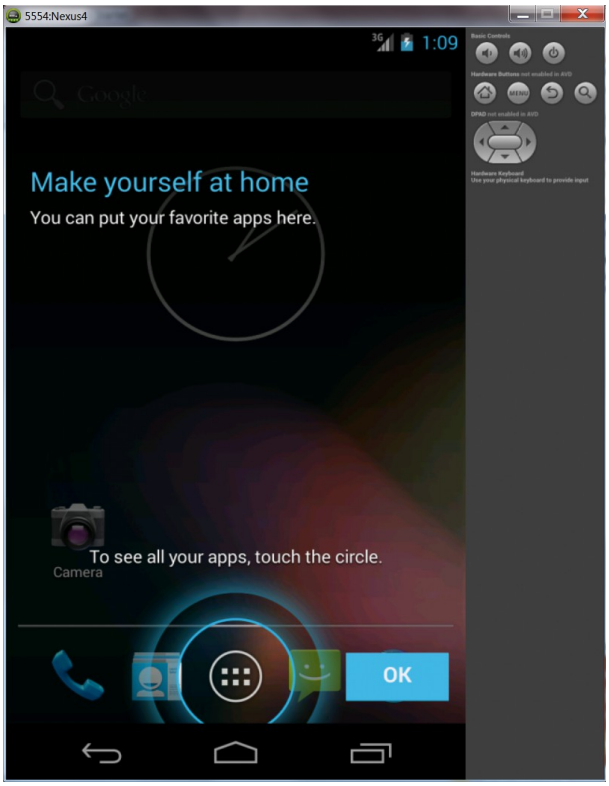

Figure 22.14: The Android emulator

As you know, the emulator emulates an Android device. You need to unlock the screen by touching (or clicking) the blue circle at the bottom.
If your application does not open automatically, locate the application icon and double-click on it. Figure 22.15 shows the HelloWorld application.

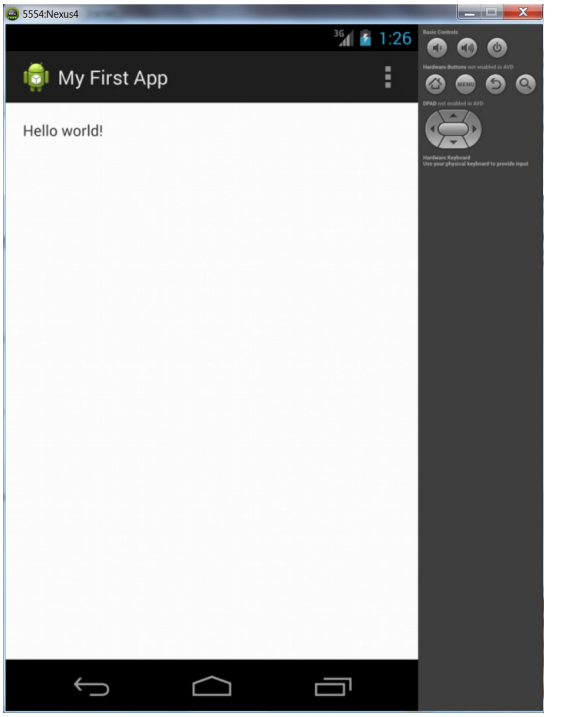

Figure 22.15: Your first application on the emulator

During development, leave the emulator running while you edit your code. This way, the emulator does not need to be loaded again every time you run your application.

Application Structure

When you run an Android application from inside the ADT Bundle, an apk file will be built for you and saved in the **bin** directory of your application directory. An apk file is basically a zip file and you can use WinZip or unzip to extract its content.

Figure 22.16 shows the structure of the helloworld.apk file that was created when you ran the application.

The manifest file is there and so are the resource files. There is also a **classes.dex** file that contains the binary translation of your Java classes into Dalvik executable. Note that even if you have multiple .java files in your application, there will only be one **classes.dex** file.

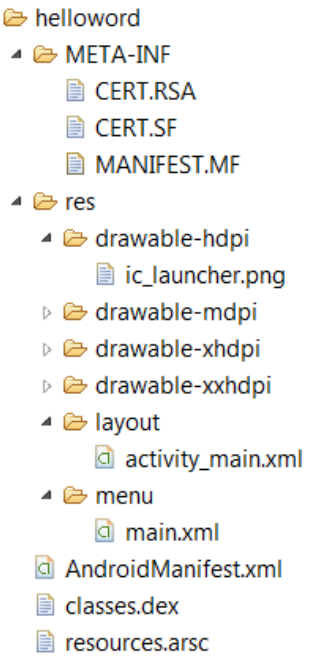

Figure 22.16: Android application structure

Logging

Java programmers like to use logging utilities, such as Commons Logging and Log4J, to log messages. The Android framework provides the **android.util.Log** class for the same purpose. The **Log** class comes with methods to log messages at different log levels. The method names are short: **d** (debug), **i** (info), **v** (verbose), **w** (warning), **e** (error), and **wtf** (what a terrible failure).

This methods allow you to write a tag and the text. For example,

```
Log.e("activity", "Something went wrong");
```

During development, messages logged using the **Log** class will appear in the LogCat view in Eclipse. If you don't see it, click **Window → Show View → LogCat** or **Window → Show View → Other → LogCat**.

The good thing about LogCat is that messages at different log levels are displayed in different colors. In addition, each message has a tag and this makes it easy to find a message. In addition, LogCat allows you to save messages to a file and filter the messages so only messages of interest to you are visible.

The LogCat view is shown in Figure 22.17.

Any runtime exception thrown, including the stack trace, will also be shown in LogCat, so you can easily identify which line of code is causing the problem.

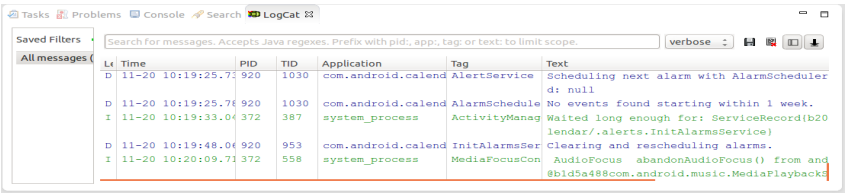

Figure 22.17: The LogCat view

Debugging An Application

Even though Android applications do not run on the JVM, debugging an Android application in Eclipse does not feel that different from debugging Java applications.

The easiest way to debug an application is by printing messages using the **Log** class. However, if this does not help and you need to trace your application, you can use the debugging tools in Android.

Try adding a line break point in your code by double-clicking the bar to the left of the code editor. Figure 22.18 shows a line breakpoint in the code editor.

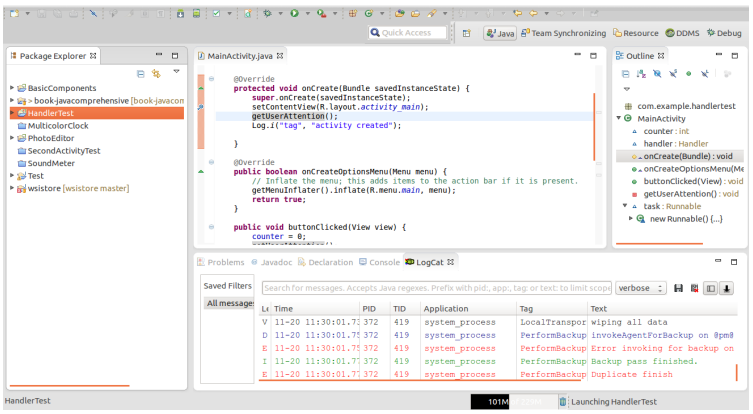

Figure 22.18: A line breakpoint

Now, debug your application by clicking the project icon in the Project Explorer and selecting **Run → Debug As → Android Application**.

Eclipse will display a dialog asking you whether you want to open the Debug perspective. Click **Yes**, and you will see the Debug perspective like the one in Figure 22.19.

Here, you can step into your code, view your variables, and so on.

In addition to a debugger, Android also ships with Dalvik Debug Monitor Server (DDMS), which consists of a set of debugging tools. You can display the DDMS in Eclipse by showing the DDMS perspective. (See Figure 22.20).

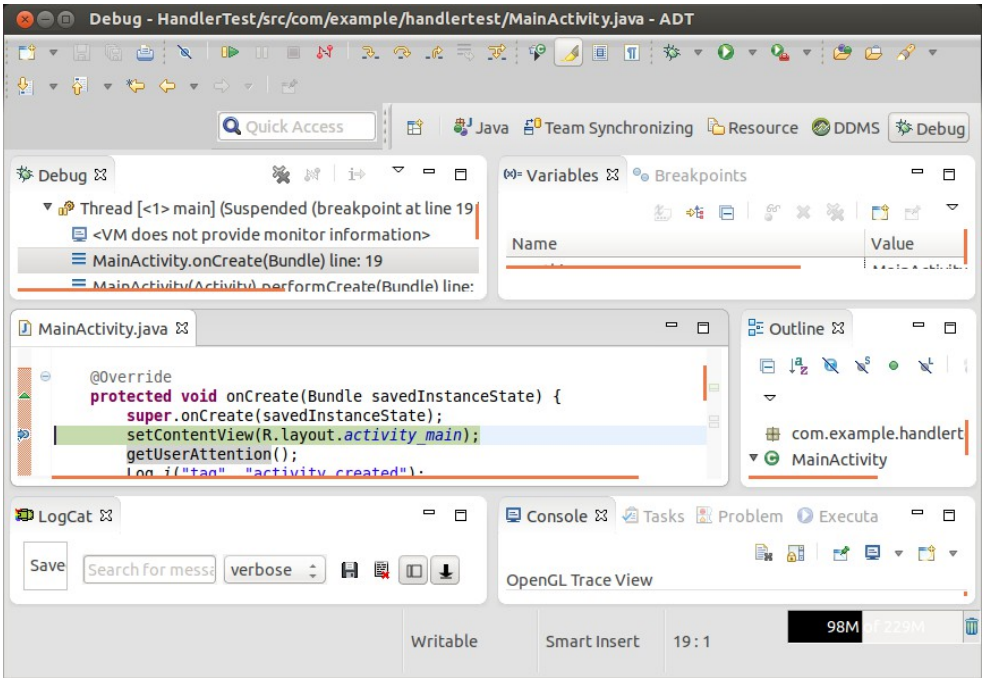

Figure 22.19: The Debug perspective

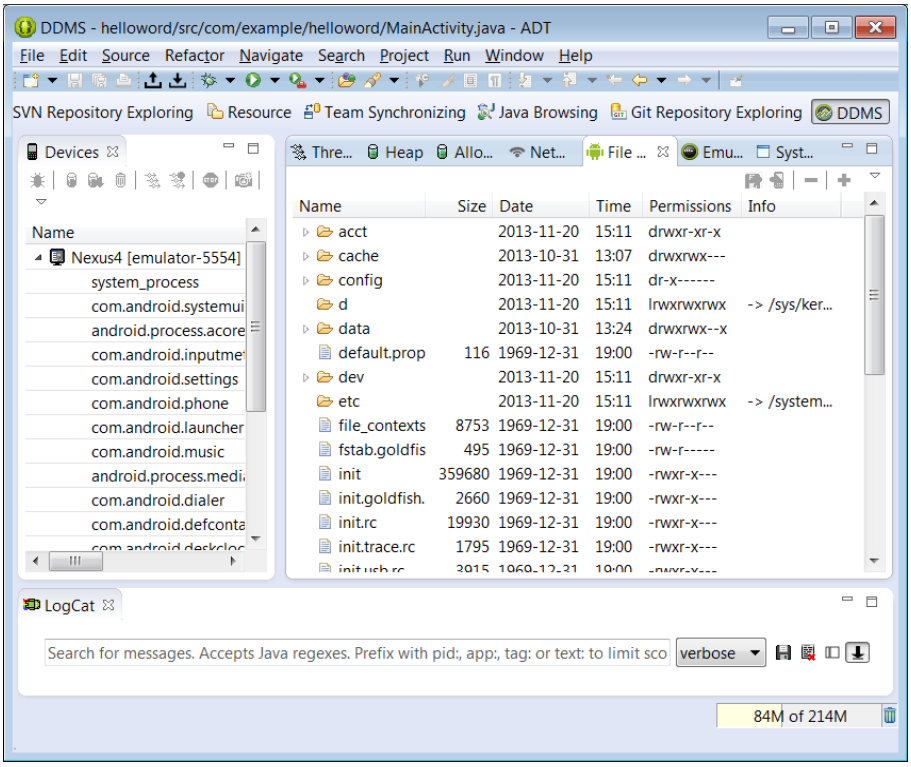

Figure 22.20: The DDMS perspective in Eclipse

You will see LogCat as one of the views in the DDMS perspective. However, you can also use DDMS to do any of these:

- Verify that a device is connected.
- View heap usage for a process
- Check object memory allocation
- Browse the file system on a device
- Examine thread information
- Monitor network traffic

Running on A Real Device

There are a couple of reasons for wanting to test your application on a real device. The most compelling one is that you should test your applications on a real device before publishing them. Other reasons include speed. An emulator may not be as fast as a new Android device. Also, it is not always easy to simulate certain user inputs in an emulator. For example, you can change the screen orientation easily with a real device. On an emulator, you have to press **Ctrl+F12**.

To run your application on a real device, follow these steps.

1. Declare your application as debuggable by adding **android:debuggable="true"** in the **application** element in the manifest file.
2. Enable USB debugging on the device. On Android 3.2 or older, the option is under **Settings > Applications > Development**. On Android 4.0 and later, the option is under **Settings > Developer Options**. On Android 4.2 and later, Developer options is hidden by default. To make it visible, go to **Settings** > **About phone** and tap **Build number** seven times.

Next, set up your system to detect the device. The step depends on what operating system you're using. For Mac users, you can skip this step. It will just work.

For Windows users, you need to install the USB driver for Android Debug Bridge (adb), a tool that lets you communicate with an emulator or connected Android device. You can find the location of the driver from this site.

`http://developer.android.com/tools/extras/oem-usb.html`

For Linux users, please see the instructions here.

`http://developer.android.com/tools/device.html`

Upgrading the SDK

The Android platform developers add a new version of the SDK several times a year. To use the new version, you do not have to re-install the SDK. You can update it using the SDK Manager.

In Eclipse, click **Window → Android SDK Manager**. Figure 22.21 shows the Android SDK Manager window.

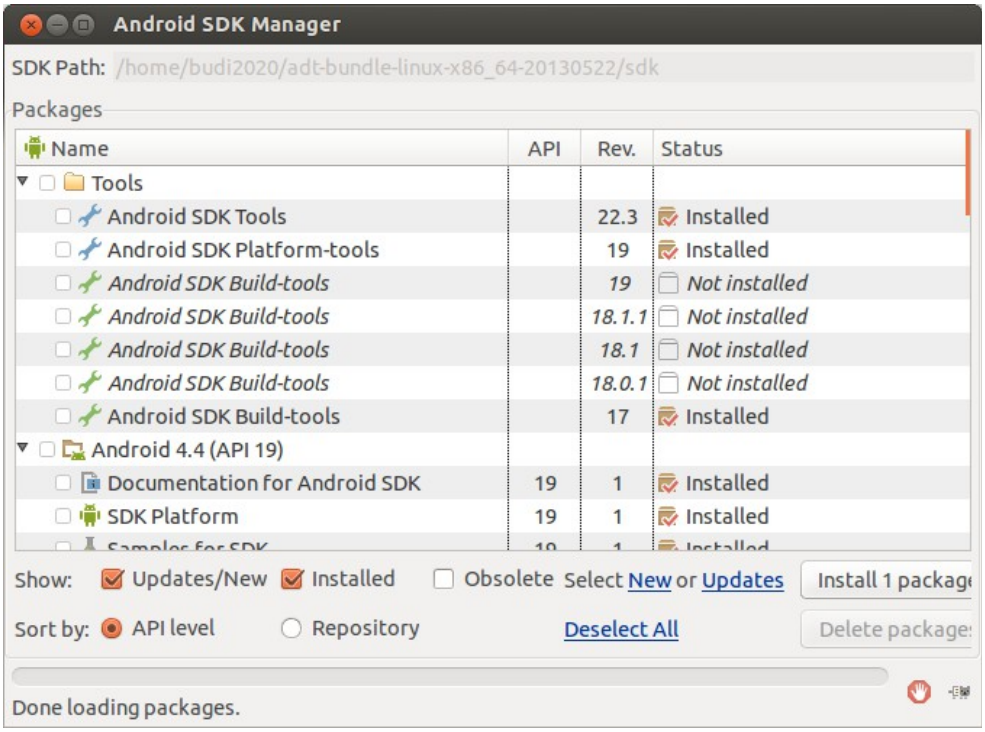

Figure 22.21: The Android SDK Manager window

If there is a new version that has not been installed, the new version will be shown here and you can click **Install** to install it.

Summary

This chapter discusses how to create your first application as well as test and debug it. It also teaches you how to configure a virtual device so you can still run your application even without a real device.

Chapter 23
Activities

In Chapter 22, “Your First Android Application” you learned to write a simple Android application. It is now time to delve deeper into the art and science of Android development. This chapter discusses one of the most important component types in Android programming, the activity.

The Activity Lifecycle

The first application component that you need to get familiar with is the activity. An activity is a thing that the user can do. This definition sounds vague, especially for beginners. However, considering that most activities involve displaying a window containing user interface (UI) components that the user can interact with, you can liken an activity to a window. Therefore, starting an activity often means displaying a window.

All activities are represented by the **android.app.Activity** class. You create an activity by subclassing this class.

An typical Android application starts by starting an activity, which, as I said, loosely means showing a window. The first window that the application creates is called the main activity and serves as the entry point to the application. Needless to say, an Android application may contain multiple activities and you specify the main activity by declaring it in the application manifest file.

For example, the following **application** element in an Android manifest defines two activities, one of which is declared as the main activity using the **intent-filter** element. To make an activity the main activity of an application, its **intent-filter** element must contain the **MAIN** action and **LAUNCHER** category like so.

```
<application ... >
    <activity
            android:name="com.example.MainActivity"
            android:label="@string/app_name" >
        <intent-filter>
            <action android:name="android.intent.action.MAIN"/>
           <category
                android:name="android.intent.category.LAUNCHER"/>
        </intent-filter>
    </activity>
    <activity
            android:name="com.example.SecondActivity"
            android:label="@string/title_activity_second" >
    </activity>
</application>
```

In the snippet above, it is not hard to see that the first activity is the main activity.

When the user selects your application icon from the Home screen, the system will look for the main activity of the application and start it. Starting an activity entails instantiating the activity class (which is specified in the **android:name** attribute of the **activity** element in the manifest) and calling its lifecycle methods. It is important that you understand these methods so you can write code correctly.

The following are the lifecycle methods of **Activity**. Some are called once during the application lifetime, some can be called more than once.

- **onCreate**
- **onStart**
- **onResume**
- **onPause**
- **onStop**
- **onRestart**
- **onDestroy**

To truly understand how these lifecycle methods come into play, consider the diagram in Figure 23.1.

The system begins by calling the **onCreate** method to create the activity. You should place the code that constructs the UI here. Once **onCreate** is completed, your application is said to be in the **Created** state. This method will only be called once during the application life time.

Next, the system calls the activity's **onStart** method. When this method is called, the application becomes visible. Once this method is completed, the application is in the **Started** state. This method may be called more than once during the application life time.

onStart is followed by **onResume** and once **onResume** is completed, the application is in the **Resumed** state. How I wish they had called it **Running** instead of **Resumed**, because the fact is this is the state where your application is fully running. **onResume** may be called multiple times during the application life time.

Therefore, **onCreated**, **onStart**, and **onResume** will be called successively unless something goes awry during the process. Once in the **Resumed** state, the application is basically running and will stay in this state until something occurs to change that, such as if the alarm clock sets off or the screen turns off because the device is going to sleep, or perhaps because another application is started.

The application that is leaving the **Resumed** state will have its running activity's **onPause** method called. Once **onPause** is completed, the application enters the **Paused** state. **onPause** can be called multiple times during the application life time.

What happens after **onPause** depends on whether or not your application becomes completely invisible. If it does, the **onStop** method is called and the application enters the **Stopped** state. On the other hand, if the application becomes active again after **onPause**, the system calls the **onResume** method and the application re-enters the **Resumed** state.

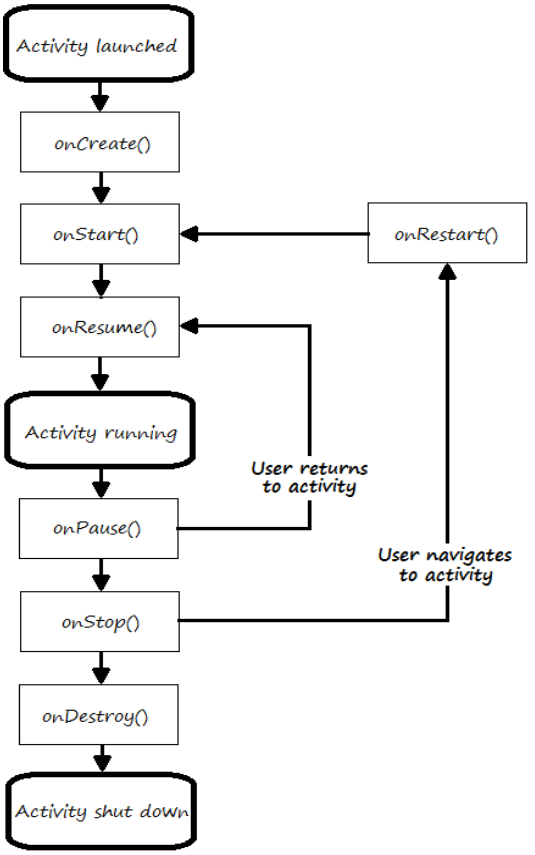

Figure 23.1: The activity lifecycle

An application in the **Stopped** state may be re-activated if the user chooses to go back to the application or for some other reason it goes back to the foreground. In this case, the **onRestart** method will be called, followed by **onStart**.

Finally, when the application is decommissioned, its **onDestroy** method is called. This method, like **onCreate**, can only be called once during the application life time.

ActivityDemo Example

The ActivityDemo application accompanying this book demonstrates when the activity lifecycle methods are called. Listing 23.1 shows the manifest for this application.

Listing 23.1: The manifest for ActivityDemo

```
<?xml version="1.0" encoding="utf-8"?>
<manifest
       xmlns:android="http://schemas.android.com/apk/res/android"
    package="com.example.activitydemo"
    android:versionCode="1"
    android:versionName="1.0" >
```

```
    <uses-sdk
        android:minSdkVersion="8"
        android:targetSdkVersion="18" />

    <application
        android:allowBackup="true"
        android:icon="@drawable/ic_launcher"
        android:label="@string/app_name"
        android:theme="@style/AppTheme" >
        <activity
            android:name="com.example.activitydemo.MainActivity"
            android:screenOrientation="landscape"
            android:label="@string/app_name" >
            <intent-filter>
                <action
       android:name="android.intent.action.MAIN" />
                <category
       android:name="android.intent.category.LAUNCHER" />
            </intent-filter>
        </activity>
    </application>

</manifest>
```

This manifest is like the one in Chapter 22, "Your First Android Application." It has one activity, the main activity. However, notice that I specify the orientation of the activity using the **android:screenOrientation** attribute of the activity element.

The main class for this application is printed in Listing 23.2. The class overrides all the lifecycle methods of **Activity** and prints a debug message in each lifecycle method.

Listing 23.2: The MainActivity class for ActivityDemo

```
package com.example.activitydemo;
import android.os.Bundle;
import android.app.Activity;
import android.util.Log;
import android.view.Menu;

public class MainActivity extends Activity {

    @Override
    protected void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        Log.d("lifecycle", "onCreate");
        setContentView(R.layout.activity_main);
    }

    @Override
    public boolean onCreateOptionsMenu(Menu menu) {
        // Inflate the menu; this adds items to the action bar if
       it is present.
        getMenuInflater().inflate(R.menu.main, menu);
        return true;
    }
```

```
    @Override
    public void onStart() {
        super.onStart();
        Log.d("lifecycle", "onStart");
    }

    @Override
    public void onRestart() {
        super.onRestart();
        Log.d("lifecycle", "onRestart");
    }

    @Override
    public void onResume() {
        super.onResume();
        Log.d("lifecycle", "onResume");
    }

    @Override
    public void onPause() {
        super.onPause();
        Log.d("lifecycle", "onPause");
    }

    @Override
    public void onStop() {
        super.onStop();
        Log.d("lifecycle", "onStop");
    }

    @Override
    public void onDestroy() {
        super.onDestroy();
        Log.d("lifecycle", "onDestroy");
    }
}
```

Before you run this application, create a Logcat message filter to show only messages from the application, filtering out system messages, by following these steps.

1. Click the green Plus sign on the **Saved Filter** pane in LogCat.
2. Type in a name in the **Filter Name** field and **lifecycle** in the **by Log Message** field. Next, select **debug** from the **by Log Level** dropdown. Figure 23.2 shows the **Logcat Message Filter Settings** window.
3. Click **OK** to create the filter.

Run the application and notice the orientation of the application. It should be landscape. Now, try running another application and then switch back to the ActivityDemo application. Check the messages printed in Logcat.

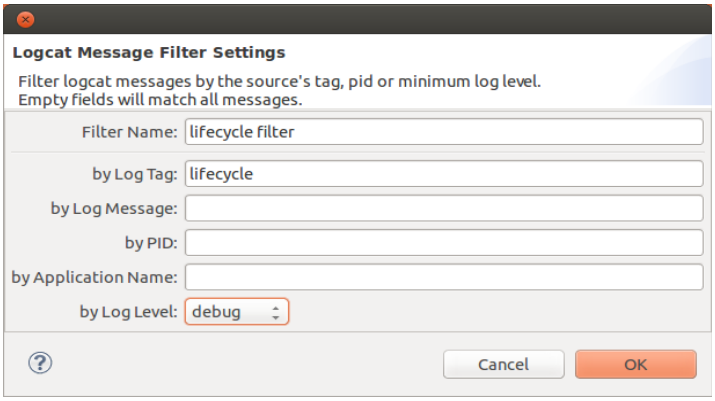

Figure 23.2: Creating a Logcat message filter

Changing the Application Icon

If you do not like the application icon you have chosen, you can easily change it by following these steps.

- Save a jpeg or png file in res/drawable (any one of them). Png is preferred because the format supports transparency.
- Edit the **android:icon** attribute of the manifest to point to the new image.

Using Android Resources

Android is rich, it comes with a bunch of assets (resources) you can use in your apps. To browse the available resources, open the application manifest in Eclipse and fill a property value by typing "**@android:**" followed by Ctrl+space. Eclipse will show the list of assets. (See Figure 23.3).

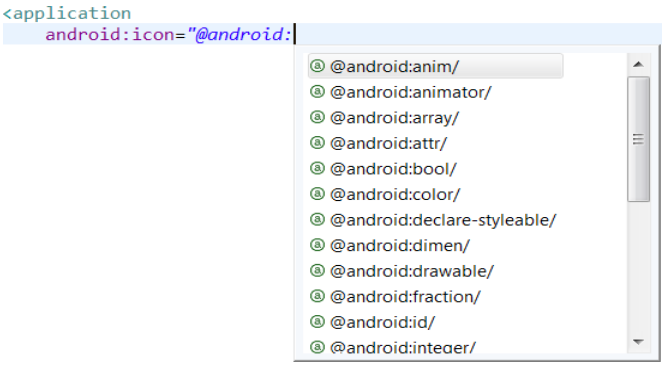

Figure 23.3: Using Android assets

For example, to see what images/icons are available, select **@android:drawable/**. To use a different icon, set this value:

```
android:icon="@android:drawable/ic_menu_day"
```

Starting Another Activity

The main activity of an Android application is started by the system itself, when the user selects the app icon from the Home screen. In an application with multiple activities, it is possible (and easy) to start another activity. In fact, starting an activity from another activity can be done simply by calling the **startActivity** method like this.

```
startActivity(intent);
```

where *intent* is an instance of **Intent**.

As an example, consider the SecondActivityTest project that accompanies this book. It has two activities, **MainActivity** and **SecondActivity**. **MainActivity** contains a button that when clicked starts **SecondActivity**. This project also shows how you can write an event listener programmatically.

The manifest for SecondActivityTest is given in Listing 23.3.

Listing 23.3: The manifest for SecondActivityTest

```
<?xml version="1.0" encoding="utf-8"?>
<manifest
       xmlns:android="http://schemas.android.com/apk/res/android"
    package="com.example.secondactivitytest"
    android:versionCode="1"
    android:versionName="1.0" >

    <uses-sdk
        android:minSdkVersion="8"
        android:targetSdkVersion="19" />

    <application
        android:allowBackup="true"
        android:icon="@drawable/ic_launcher"
        android:label="@string/app_name"
        android:theme="@style/AppTheme" >
        <activity

       android:name="com.example.secondactivitytest.MainActivity"
            android:label="@string/app_name" >
            <intent-filter>
                <action android:name="android.intent.action.MAIN"/>
                <category
       android:name="android.intent.category.LAUNCHER"/>
            </intent-filter>
        </activity>
        <activity

       android:name="com.example.secondactivitytest.SecondActivity"
            android:label="@string/title_activity_second" >
        </activity>
    </application>
</manifest>
```

Unlike the previous application, this project has two activities, one of which is declared as the main activity.

The layout files for the main and second activities are listed in Listings 23.4 and 23.5, respectively.

Listing 23.4: The activity_main.xml file

```
<RelativeLayout
       xmlns:android="http://schemas.android.com/apk/res/android"
    xmlns:tools="http://schemas.android.com/tools"
    android:layout_width="match_parent"
    android:layout_height="match_parent"
    android:paddingBottom="@dimen/activity_vertical_margin"
    android:paddingLeft="@dimen/activity_horizontal_margin"
    android:paddingRight="@dimen/activity_horizontal_margin"
    android:paddingTop="@dimen/activity_vertical_margin"
    tools:context=".MainActivity" >

    <TextView
        android:id="@+id/textView1"
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:text="@string/first_screen" />

</RelativeLayout>
```

Listing 23.5: The activity_second.xml file

```
<RelativeLayout
       xmlns:android="http://schemas.android.com/apk/res/android"
    xmlns:tools="http://schemas.android.com/tools"
    android:layout_width="match_parent"
    android:layout_height="match_parent"
    android:paddingBottom="@dimen/activity_vertical_margin"
    android:paddingLeft="@dimen/activity_horizontal_margin"
    android:paddingRight="@dimen/activity_horizontal_margin"
    android:paddingTop="@dimen/activity_vertical_margin"
    tools:context=".SecondActivity" >

    <TextView
        android:id="@+id/textView1"
        android:layout_width="wrap_content"
        android:layout_height="wrap_content" />

</RelativeLayout>
```

Both activities contain a **TextView**. When the **TextView** in the main activity is touched, it will start the second activity and pass a message for the latter. The second activity will display the message in its **TextView**.

The activity class for the main activity is given in Listing 23.6.

Listing 23.6: The MainActivity class

```
package com.example.secondactivitytest;
import android.app.Activity;
import android.content.Intent;
import android.os.Bundle;
import android.view.Menu;
import android.view.MotionEvent;
import android.view.View;
import android.view.View.OnTouchListener;
```

```
import android.widget.TextView;

public class MainActivity extends Activity implements
        OnTouchListener {

    @Override
    protected void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.activity_main);
        TextView tv = (TextView) findViewById(R.id.textView1);
        tv.setOnTouchListener(this);
    }

    @Override
    public boolean onCreateOptionsMenu(Menu menu) {
        // Inflate the menu; this adds items to the action bar if
       it
        // is present.
        getMenuInflater().inflate(R.menu.main, menu);
        return true;
    }

    @Override
    public boolean onTouch(View arg0, MotionEvent event) {
        Intent intent = new Intent(this, SecondActivity.class);
        intent.putExtra("message", "Message from First Screen");
        startActivity(intent);
        return true;
    }
}
```

To handle the touch event, the **MainActivity** class has implemented the **OnTouchListener** interface and overridden its **onTouch** method. In this method, you create an **Intent** and put a message in it. You then call the **startActivity** method to start the second activity.

The **SecondActivity** class is given in Listing 23.7.

Listing 23.7: The SecondActivity class

```
package com.example.secondactivitytest;
import android.app.Activity;
import android.content.Intent;
import android.os.Bundle;
import android.view.Menu;
import android.widget.TextView;

public class SecondActivity extends Activity {

    @Override
    protected void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.activity_second);
        Intent intent = getIntent();
        String message = intent.getStringExtra("message");
        ((TextView) findViewById(R.id.textView1)).setText(message);
    }
```

```
    @Override
    public boolean onCreateOptionsMenu(Menu menu) {
        getMenuInflater().inflate(R.menu.second, menu);
        return true;
    }
}
```

In the **onCreate** method of **SecondActivity**, you set the view content as usual. You then call the **getIntent** method and retrieve a message from its **getStringExtra** method, which you then pass to the **setText** method of the **TextView**. You retrieve the **TextView** by calling the **findViewById** method.

The main activity and the second activity are shown in Figures 23.4 and 23.5, respectively.

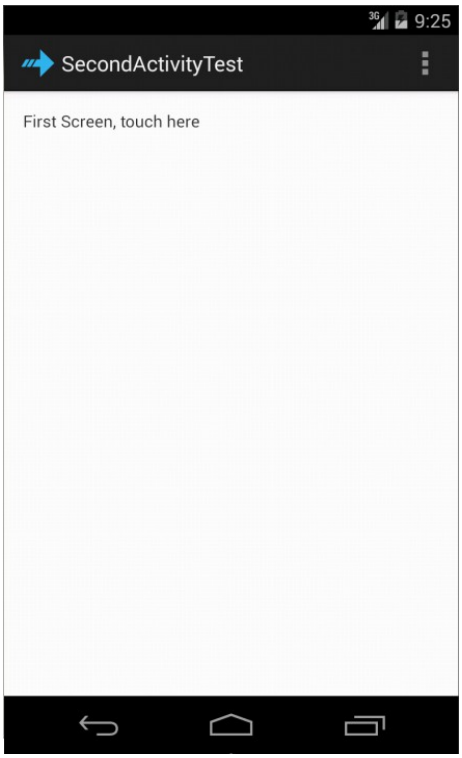

Figure 23.4: The main activity in SecondActivityTest

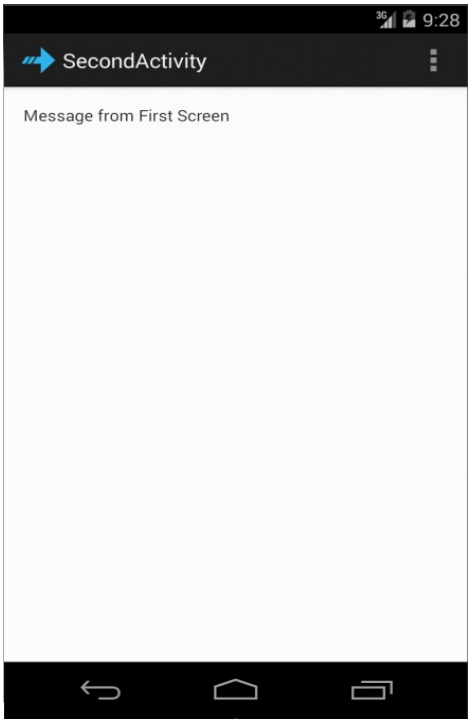

Figure 23.5: The second activity in SecondActivityTest

Summary

In this chapter you learned about the activity lifecycle and created two applications. The first application allowed you to observe when each of the lifecycle methods was called. The second application showed how to start an activity from another activity.

Chapter 24
UI Components

One of the first things you do when creating an Android application is build the user interface (UI) for the main activity. This is a relatively easy task thanks to the ready-to-use UI components that Android provides. This chapter discusses some of the more important UI components.

Overview

The Android SDK provides various UI components called widgets that include many simple and complex components. Examples of widgets include buttons, text fields, progress bars, etc. In addition, you also need to choose a layout for laying out your UI components. Both widgets and layouts are implementations of the **android.view.View** class. A view is a rectangular area that occupies the screen. **View** is one of the most important Android types. However, unless you are creating a custom view, you don't often work with this class directly. Instead, you often spend time choosing and using layouts and UI components for your activities.

Figure 24.1 shows some Android UI components.

Using the ADT Eclipse UI Tool

Creating a UI is easy with ADT Eclipse. All you need is open the layout file for an activity and drag and drop UI components to the layout. The activity files are located in the **res/layout** directory of your application.

Figure 24.2 shows the UI tool for creating Android UI. This is what you see when you open an activity file. The tool window is divided into three main areas. On the left are the widgets, which are grouped into different categories such as Form Widgets, Text Fields, Layouts, etc. Click on the tab header of a category to see what widgets are available for that category.

To choose a widget, click on the widget and drag it to the activity screen at the center. The screen in Figure 24.2 shows two text fields and a button. You can also view how your screen will look like in different devices by choosing a device from the **Devices** drop-down.

Each widget and layout has a set of properties derived from the **View** class or added to the implementation class. To change any of these properties, click on the widget on the drawing area or select it from the Outline pane in the Structure window on the right. The properties are listed in the small pane under the Layout pane.

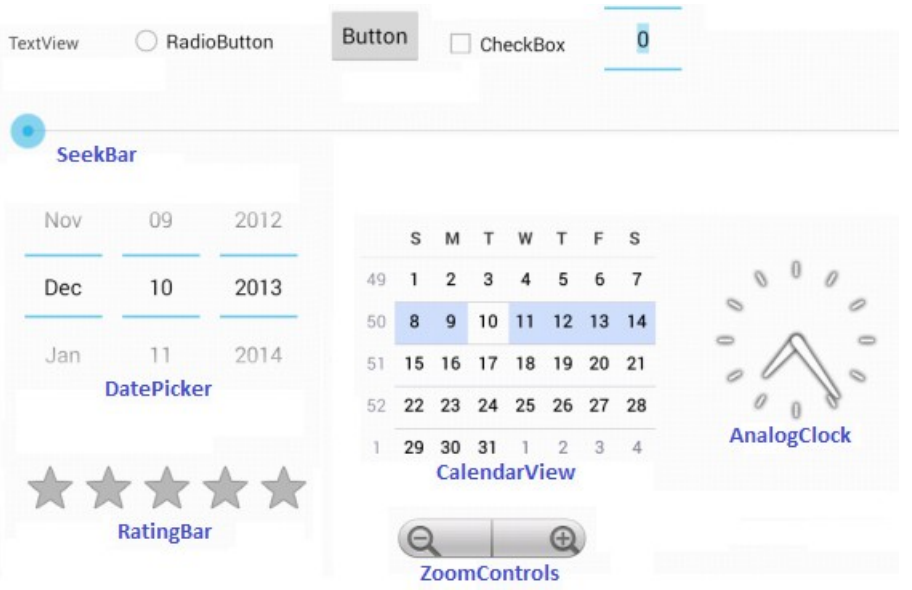

Figure 24.1: Android UI components

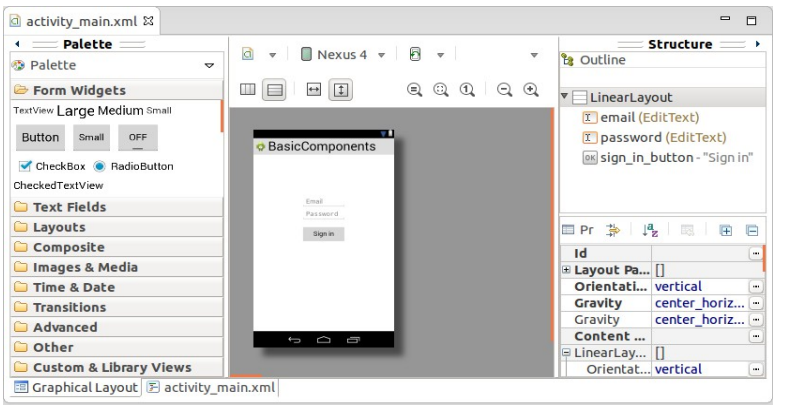

Figure 24.2: Using the UI tool

What you do with the UI tool is reflected in the layout file, in the form of XML elements. To see what has been generated for you, click the XML view at the bottom of the UI tool.

Using Basic Components

The BasicComponents project is a simple Android application with one activity. The activity screen contains two text fields and a button.

You can either open the accompanying application or create one yourself by following the instructions in Chapter 22, "Your First Android Application." I will start explaining this project by presenting the manifest for the application, which is an XML file named **AndroidManifest.xml** located directly under the root directory.

Listing 24.1 shows the **AndroidManifest.xml** for the BasicComponents project.

Listing 24.1: The manifest for BasicComponents

```
<?xml version="1.0" encoding="utf-8"?>
<manifest
       xmlns:android="http://schemas.android.com/apk/res/android"
    package="com.example.basiccomponents"
    android:versionCode="1"
    android:versionName="1.0">

    <uses-sdk
        android:minSdkVersion="8"
        android:targetSdkVersion="17" />

    <application
        android:allowBackup="true"
        android:icon="@drawable/ic_launcher"
        android:label="@string/app_name"
        android:theme="@style/AppTheme" >
        <activity
            android:name="com.example.basiccomponents.MainActivity"
            android:label="@string/app_name" >
            <intent-filter>
                <action android:name="android.intent.action.MAIN"/>
                <category
                  android:name="android.intent.category.LAUNCHER"/>
            </intent-filter>
        </activity>
    </application>
</manifest>
```

The first thing to note is the **package** attribute of the **manifest** tag, which specifies **com.example.basiccomponents** as the Java package for the generated classes. Also note that the **application** element defines one activity, the main activity. The **application** element also specifies the icon, label, and theme for this application.

```
        android:icon="@drawable/ic_launcher"
        android:label="@string/app_name"
        android:theme="@style/AppTheme">
```

It is good practice to reference a resource (such as an icon or a label) indirectly, like what I am doing here. **@drawable/ic_launcher**, the value for **android:icon**, refers to a drawable (normally an image file) that resides under the **res/drawable** directory. **ic_launcher** can mean an **ic_launcher.png** or **ic_launcher.jpg** file.

All string references start with **@string**. In the example above, **@string/app_name** refers to the **app_name** key in the **res/values/strings.xml** file. For this application, the **strings.xml** file is given in Listing 24.2.

Listing 24.2: The strings.xml file under res/values

```
<?xml version="1.0" encoding="utf-8"?>
<resources>
    <string name="app_name">BasicComponents</string>
    <string name="action_settings">Settings</string>
    <string name="prompt_email">Email</string>
    <string name="prompt_password">Password</string>
```

```
    <string name="action_sign_in"><b>Sign in</b></string>
</resources>
```

Let's now look at the main activity. There are two resources concerned with an activity, the layout file for the activity and the Java class that derives from **android.app.Activity**. For this project, the layout file is given in Listing 24.3 and the activity class (**MainActivity**) in Listing 24.4.

Listing 24.3: The layout file

```
<LinearLayout
       xmlns:android="http://schemas.android.com/apk/res/android"
    xmlns:tools="http://schemas.android.com/tools"
    android:layout_width="match_parent"
    android:layout_height="match_parent"
    android:layout_gravity="center"
    android:gravity="center_horizontal"
    android:orientation="vertical"
    android:padding="120dp"
    tools:context=".MainActivity" >

    <EditText
        android:id="@+id/email"
        android:layout_width="match_parent"
        android:layout_height="wrap_content"
        android:hint="@string/prompt_email"
        android:inputType="textEmailAddress"
        android:maxLines="1"
        android:singleLine="true" />

    <EditText
        android:id="@+id/password"
        android:layout_width="match_parent"
        android:layout_height="wrap_content"
        android:hint="@string/prompt_password"
        android:imeActionId="@+id/login"
        android:imeOptions="actionUnspecified"
        android:inputType="textPassword"
        android:maxLines="1"
        android:singleLine="true" />

    <Button
        android:id="@+id/sign_in_button"
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:layout_gravity="right"
        android:layout_marginTop="16dp"
        android:paddingLeft="32dp"
        android:paddingRight="32dp"
        android:text="@string/action_sign_in" />

</LinearLayout>
```

The layout file contains a **LinearLayout** with three children, namely two **EditText** components and a button.

Listing 24.4: The MainActivity class of Basic Components

```
package com.example.basiccomponents;
```

```
import android.os.Bundle;
import android.app.Activity;
import android.view.Menu;

public class MainActivity extends Activity {

    @Override
    protected void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.activity_main);
    }

    @Override
    public boolean onCreateOptionsMenu(Menu menu) {
        // Inflate the menu; this adds items to the action bar if
       it
        // is present.
        getMenuInflater().inflate(R.menu.main, menu);
        return true;
    }
}
```

The **MainActivity** class in Listing 24.4 is a boilerplate class created by Eclipse. It overrides the **onCreate** and **onCreateOptionsMenu** methods. **onCreate** is a lifecycle method that gets called when the application is created. In Listing 24.4, it simply sets the content view for the activity using the layout file. **onCreateOptionsMenu** initializes the content of the activity's options menu. It must return true for the menu to be displayed.

Run the application and you'll see the activity like that in Figure 24.3.

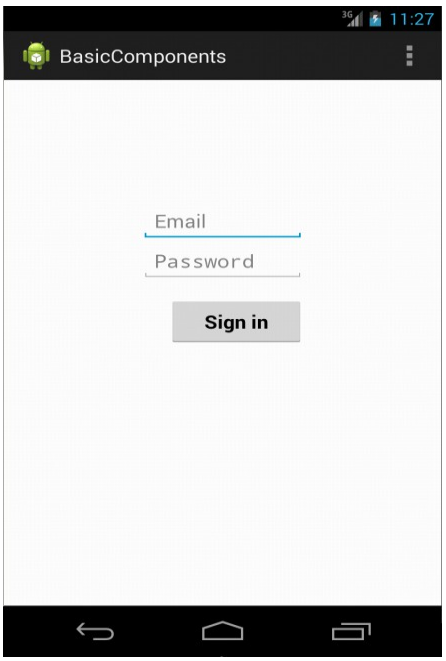

Figure 24.3: The BasicComponents project

Toast

A **Toast** is a small popup for displaying a message as feedback for the user. A **Toast** does not replace the current activity and only occupies the space taken by the message.

Figure 24.4 shows a Toast that says "Downloading file..." The **Toast** disappears after a prescribed period of time.

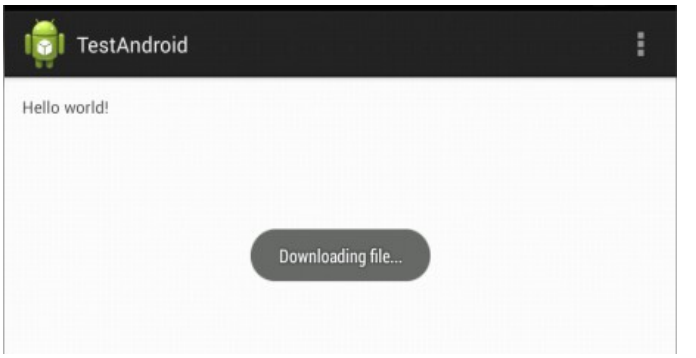

Figure 24.4: A Toast

The **android.widget.Toast** class is the template for creating a **Toast**. its To create a **Toast**, call its only constructor that takes a **Context** as argument:

```
public Toast(android.content.Context context)
```

Toast also provides two static **makeText** methods for creating an instance of **Toast**. The signatures of both method overloads are

```
public static Toast makeText (android.content.Context context,
        int resourceId, int duration)
```

```
public static Toast makeText (android.content.Context context,
        java.lang.CharSequence text, int duration)
```

Both overloads require that you pass a **Context**, possibly the active activity, as the first argument. In addition, both overloads take a string, which may come from a **strings.xml** file or a **String** object, and the duration of the display for the **Toast**. Two valid values for the duration are the **LENGTH_LONG** and **LENGTH_SHORT** static finals in **Toast**.

To display the **Toast**, call its **show** method, which takes no argument.

The following code snippet shows how to create and display a Toast in an activity class.

```
Toast.makeText(this, "Downloading...", Toast.LENGTH_LONG).show();
```

By default, a **Toast** is displayed near the bottom of the active activity. However, you can change its position by calling its **setGravity** method before calling its **show** method. Here is the signature of **setGravity**.

```
public void setGravity (int gravity, int xOffset, int yOffset)
```

The valid value for *gravity* is one of the static finals in the **android.view.Gravity** class, including **CENTER_HORIZONTAL** and **CENTER_VERTICAL**.

You can also enhance the look of a **Toast** by creating your own layout file and

passing it to the **Toast**'s **setView** method. Here is an example of how to create a custom **Toast**.

```
LayoutInflater inflater = getLayoutInflater();
View layout = inflater.inflate(R.layout.toast_layout,
        (ViewGroup) findViewById(R.id.toast_layout_root));
Toast toast = new Toast(getApplicationContext());
toast.setView(layout);
toast.show();
```

In this case, **R.layout.toast_layout** is the layout identifier for the Toast and **R.id.toast_layout_root** is the id of the **root** element in the layout file.

AlertDialog

Like a **Toast**, an **AlertDialog** is a window that provide feedback to the user. Unlike a **Toast** that fades by itself, however, an **AlertDialog** will show indefinitely until it loses focus. In addition, an **AlertDialog** can contain up to three buttons and a list of selectable items. A button added to an **AlertDialog** can be connected to a listener that gets triggered when the button is clicked.

Figure 24.5 shows a sample **AlertDialog**.

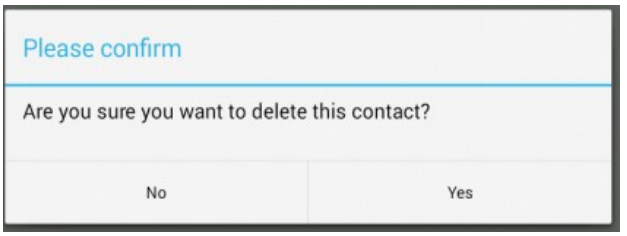

Figure 24.5: An AlertDialog

The **android.app.AlertDialog** class is the template for creating an **AlertDialog**. All constructors in the **AlertDialog** class are protected, so you cannot use them unless you are subclassing it. Instead, you should use the **AlertDialog.Builder** class to create an **AlertDialog**. You can use one of the two constructors of **AlertDialog.Builder**.

```
public AlertDialog.Builder(android.content.Context context)

public AlertDialog.Builder(android.content.Context context,
        int theme)
```

Once you have an instance of **AlertDialog.Builder**, you can call its **create** method to return an **AlertDialog**. However, before calling **create** you can call various methods of **AlertDialog.Builder** to decorate the resulting **AlertDialog**. Interestingly, the methods in **AlertDialog.Builder** return the same instance of **AlertDialog.Builder**, so you can cascade them. Here are some of the methods in **AlertDialog.Builder**.

```
public AlertDialog.Builder setIcon(int resourceId)
```

Sets the icon of the resulting **AlertDialog** with the Drawable pointed by *resourceId*.

```
public AlertDialog.Builder setMessage(
        java.lang.CharSequence message)
```

Sets the message of the resulting **AlertDialog** .

```
public AlertDialog.Builder setTitle(java.lang.CharSequence title)
```
Sets the title of the resulting **AlertDialog**.

```
public AlertDialog.Builder setNegativeButton(
        java.lang.CharSequence text,
        android.content.DialogInterface.OnClickListener listener)
```
Assigns a button that the user should click to provide a negative response.

```
public AlertDialog.Builder setPositiveButton(
        java.lang.CharSequence text,
        android.content.DialogInterface.OnClickListener listener)
```
Assigns a button that the user should click to provide a positive response.

```
public AlertDialog.Builder setNeutralButton(
        java.lang.CharSequence text,
        android.content.DialogInterface.OnClickListener listener)
```
Assigns a button that the user should click to provide a neutral response.

For instance, the following code produces an **AlertDialog** that looks like the one in Figure 24.5.

```
new AlertDialog.Builder(this)
    .setTitle("Please confirm")
    .setMessage(
        "Are you sure you want to delete " +
        "this contact?")
    .setPositiveButton("Yes",
        new DialogInterface.OnClickListener() {
            public void onClick(
                    DialogInterface dialog,
                    int whichButton) {

                // delete picture here

                dialog.dismiss();
            }
        })
    .setNegativeButton("No",
        new DialogInterface.OnClickListener() {
            public void onClick(
                    DialogInterface dialog,
                    int which) {
                dialog.dismiss();
            }
        })
    .create()
    .show();
```

Pressing the Yes button will execute the listener passed to the **setPositiveButton** method and pressing the No button will run the listener passed to the **setNegativeButton** method.

Summary

In this chapter you learned about the UI components available in Android. You also built an application that utilized these components.

Chapter 25
Layouts

Layouts are important as they directly affect the look and feel of your application. Technically, a layout is a view that arranges child views added to it. Android comes with a number of built-in layouts, ranging from **LinearLayout**, which is the easiest to use, to **RelativeLayout**, which is the most powerful.

This chapter discusses the various layouts in Android.

Overview

An important Android component, a layout defines the visual structure of your UI components. A layout is a subclass of **android.view.ViewGroup**, which in turn derives from **android.view.View**. A **ViewGroup** is a special view that can contain other views. A layout can be declared in a layout file or added programmatically at runtime.

The following are some of the layouts in Android.

- **LinearLayout**. A layout that aligns its children in the same direction, either horizontally or vertically.
- **RelativeLayout**. A layout that arranges each of its children based on the positions of one or more of its siblings.
- **FrameLayout**. A layout that arranges each of its children based on top of one another.
- **TableLayout**. A layout that organizes its children into rows and columns.
- **GridLayout**. A layout that arranges its children in a grid.

In a majority of cases, a view in a layout must have the **layout_width** and **layout_height** attributes so that the layout knows how to size the view. Both **layout_width** and **layout_height** attributes may be assigned the value **match_parent** (to match the parent's width/height), **wrap_content** (to match the width/height of its content), or a measurement unit.

The **AbsoluteLayout**, which offers exact locations for child views, is deprecated and should not be used. Use **RelativeLayout** instead.

LinearLayout

A **LinearLayout** is a layout that arranges its children either horizontally or vertically, depending the value of its **orientation** property. The **LinearLayout** is the easiest layout to use.

The layout in Listing 25.1 is an example of **LinearLayout** with horizontal orientation. It contains three children, an **ImageButton**, a **TextView**, and a

Button.

Listing 25.1: A horizontal LinearLayout

```
<LinearLayout
       xmlns:android="http://schemas.android.com/apk/res/android"
    xmlns:tools="http://schemas.android.com/tools"
    android:orientation="horizontal"
    android:layout_width="match_parent"
    android:layout_height="match_parent">

    <ImageButton
        android:src="@android:drawable/btn_star_big_on"
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"/>

    <TextView
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:text="@string/hello_world" />
    <Button android:text="Button1"
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"/>

</LinearLayout>
```

Figure 25.1 shows the views in the **LinearLayout** in Listing 25.1.

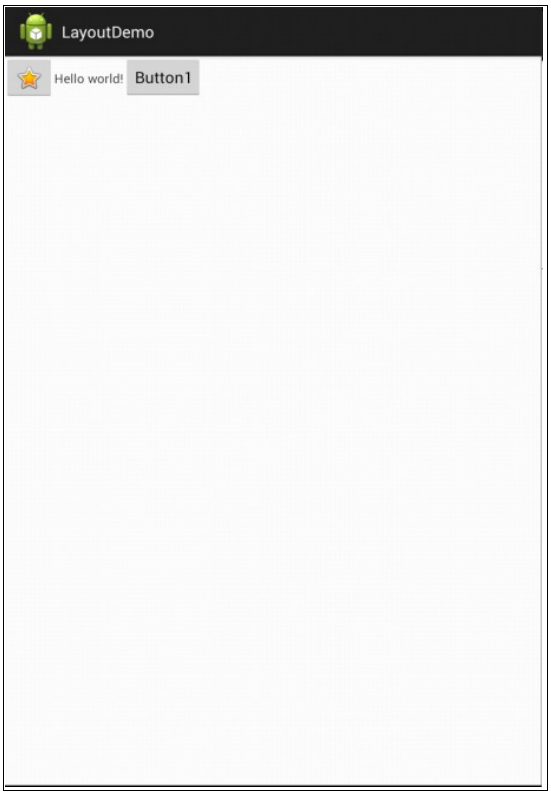

Figure 25.1: Horizontal LinearLayout example

The layout in Listing 25.2 is a vertical **LinearLayout** with three child views, an **ImageButton**, a **TextView**, and a **Button**.

Listing 25.2: Vertical linear layout

```
<LinearLayout
    xmlns:android="http://schemas.android.com/apk/res/android"
    xmlns:tools="http://schemas.android.com/tools"
    android:orientation="vertical"
    android:layout_width="match_parent"
    android:layout_height="match_parent">

    <ImageButton
        android:src="@android:drawable/btn_star_big_on"
        android:layout_gravity="center"
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"/>
    <TextView
        android:layout_gravity="center"
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:layout_marginLeft="15dp"
        android:text="@string/hello_world"/>
    <Button android:text="Button1"
        android:layout_gravity="center"
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"/>
</LinearLayout>
```

Figure 25.2 shows the vertical **LinearLayout**.

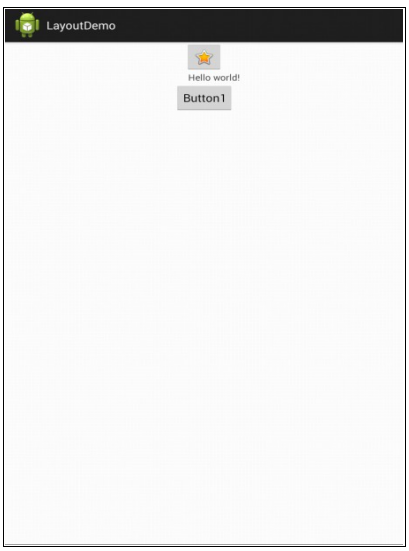

Figure 25.2: Vertical linear layout example

Note that each view in a layout can have a **layout_gravity** attribute to determine its position within its axis. For example, setting the **layout_gravity** attribute to **center** will center it.

A **LinearLayout** can also have a **gravity** attribute that affects its gravity. For example, the layout in Listing 25.3 is a vertical **LinearLayout** whose **gravity**

attribute is set to bottom.

Listing 25.3: Vertical linear layout with bottom gravity

```
<LinearLayout
       xmlns:android="http://schemas.android.com/apk/res/android"
    xmlns:tools="http://schemas.android.com/tools"
    android:orientation="vertical"
    android:layout_width="match_parent"
    android:layout_height="match_parent"
    android:gravity="bottom">

    <ImageButton
        android:src="@android:drawable/btn_star_big_on"
        android:layout_gravity="center"
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"/>
    <TextView
        android:layout_gravity="center"
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:layout_marginLeft="15dp"
        android:text="@string/hello_world"/>
    <Button android:text="Button1"
        android:layout_gravity="center"
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"/>
</LinearLayout>
```

Figure 25.3 shows the vertical **LinearLayout** in Listing 25.3.

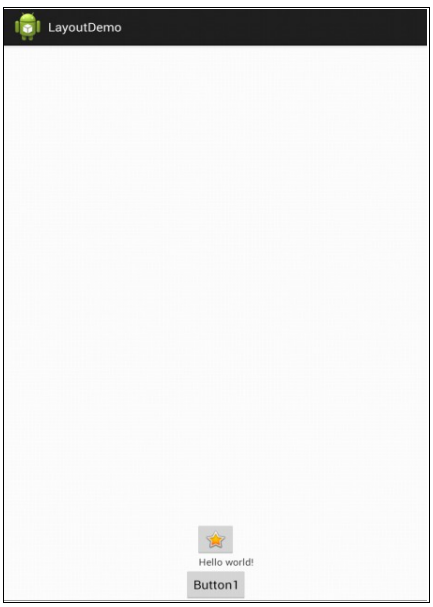

Figure 25.3: Vertical linear layout with gravity

RelativeLayout

The **RelativeLayout** is the most powerful layout available. All children in a **RelativeLayout** can be positioned relative to each other or to their parent.

Attribute	Description
layout_above	Places the bottom edge of this view above the specified view ID.
layout_alignBaseline	Places the baseline of this view on the baseline of the specified view ID.
layout_alignBottom	Aligns the bottom of this view with the specified view.
layout_alignEnd	Aligns the end edge of this view with the end edge of the specified view.
layout_alignLeft	Aligns the left edge of this view with the left edge of the specified view.
layout_alignParentBottom	A value of true aligns the bottom of this view with the bottom edge of its parent.
layout_alignParentEnd	A value of true aligns the end edge of this view with the end edge of its parent.
layout_alignParentLeft	A value of true aligns the left edge of this view with the left edge of its parent.
layout_alignParentRight	A value of true aligns the right edge of this view with the right edge of its parent.
layout_alignParentStart	A value of true aligns the start edge of this view with the start edge of its parent.
layout_alignParentTop	A value of true aligns the top edge of this view with the top edge of its parent.
layout_alignRight	Aligns the right edge of this view with the right edge of the given view.
layout_alignStart	Aligns the start edge of this view with the start edge of the given view.
layout_alignTop	Aligns the top edge of this view with the top edge of the given view.
layout_alignWithParentIfMissing	A value of true sets the parent as the anchor when the anchor cannot be found for layout_toLeftOf, layout_toRightOf, etc.
layout_below	Places the top edge of this view below the given view.
layout_centerHorizontal	A value of true centers this view horizontally within its parent.
layout_centerInParent	A value of true centers this view horizontally and vertically within its parent.
layout_centerVertical	A value of true center this view vertically within its parent.
layout_toEndOf	Places the start edge of this view to the end of the given view.
layout_toLeftOf	Places the right edge of this view to the left of the given view.
layout_toRightOf	Places the left edge of this view to the right of the given view.
layout_toStartOf	Places the end edge of this view to the start of the given view.

Table 25.1: Attributes for children of a RelativeLayout

Positioning a child in a **RelativeLayout** is achieved using the attributes summarized in Table 25.1. For example, you can tell a view to be positioned to the left or right of another view. Or, you can specify that a view is aligned to the bottom or top edge of its parent.

As an example, the layout in Listing 25.4 specifies the positions of three views and a **RelativeLayout**.

Listing 25.4: Relative layout

```
<RelativeLayout
       xmlns:android="http://schemas.android.com/apk/res/android"
    xmlns:tools="http://schemas.android.com/tools"
    android:layout_width="match_parent"
    android:layout_height="match_parent"
    android:paddingLeft="2dp"
    android:paddingRight="2dp">

    <Button
        android:id="@+id/cancelButton"
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:text="Cancel" />

    <Button
        android:id="@+id/saveButton"
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:layout_toRightOf="@id/cancelButton"
        android:text="Save" />

    <ImageView
        android:layout_width="150dp"
        android:layout_height="150dp"
        android:layout_marginTop="230dp"
        android:padding="4dp"
        android:layout_below="@id/cancelButton"
        android:layout_centerHorizontal="true"
        android:src="@android:drawable/ic_btn_speak_now" />

    <LinearLayout
        android:id="@+id/filter_button_container"
        android:layout_width="match_parent"
        android:layout_height="wrap_content"
        android:layout_alignParentBottom="true"
        android:gravity="center|bottom"
           android:background="@android:color/white"
        android:orientation="horizontal" >

        <Button
            android:id="@+id/filterButton"
            android:layout_width="wrap_content"
            android:layout_height="fill_parent"
            android:text="Filter" />

        <Button
```

```
            android:id="@+id/shareButton"
            android:layout_width="wrap_content"
            android:layout_height="fill_parent"
            android:text="Share" />

        <Button
            android:id="@+id/deleteButton"
            android:layout_width="wrap_content"
            android:layout_height="fill_parent"
            android:text="Delete" />
    </LinearLayout>
</RelativeLayout>
```

Figure 25.4 shows the **RelativeLayout** in Listing 25.4.

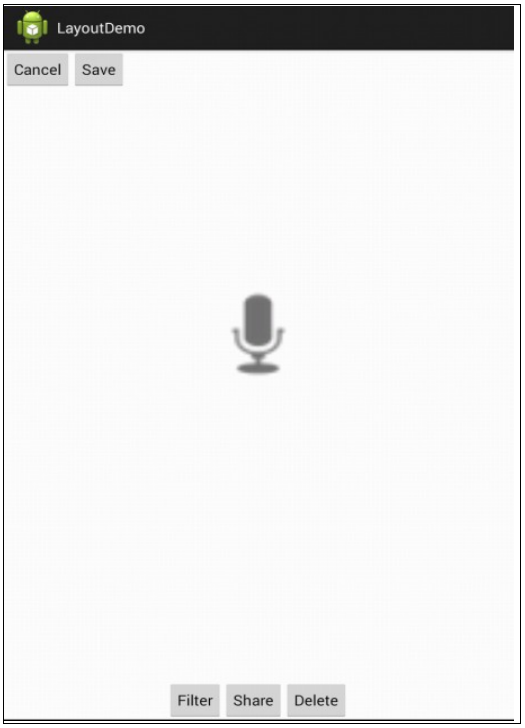

Figure 25.4: RelativeLayout

FrameLayout

A **FrameLayout** positions its children on top of each other. By adjusting the margin and padding of a view, it is possible to lay out the view below another view, as shown by the layout in Listing 25.5.

Listing 25.5: Using a FrameLayout

```
<FrameLayout
      xmlns:android="http://schemas.android.com/apk/res/android"
    xmlns:tools="http://schemas.android.com/tools"
    android:orientation="horizontal"
    android:layout_width="match_parent"
```

```
    android:layout_height="match_parent">

    <Button android:text="Button1"
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:paddingLeft="48dp"/>
    <ImageButton
        android:src="@android:drawable/btn_star_big_on"
        android:alpha="0.35"
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"/>
</FrameLayout>
```

The layout in Listing 25.5 uses a **FrameLayout** with a **Button** and an **ImageButton**. The **ImageButton** is placed on top of the **Button**, as shown in Figure 25.5.

TableLayout

A **TableLayout** is used to arrange child views in rows and columns. The **TableLayout** class is a subclass of **LinearLayout**. To add a row in a **TableLayout**, use a **TableRow** element. A view directly added to a **TableLayout** (without a **TableRow**) will also occupy a row that spans all columns.

The layout in Listing 25.6 shows a **TableLayout** with four rows, two of which are created using **TableRow** elements.

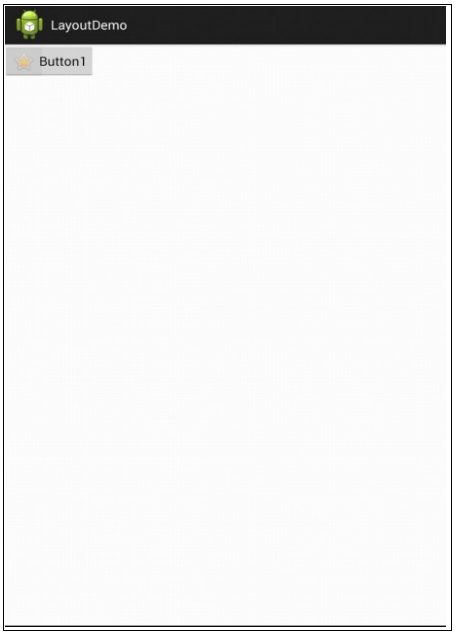

Figure 25.5: Using FrameLayout

Listing 25.6: Using the TableLayout

```
<TableLayout
        xmlns:android="http://schemas.android.com/apk/res/android"
```

```
    android:layout_width="wrap_content"
    android:layout_height="wrap_content"
    android:layout_gravity="center" >

    <TableRow
        android:id="@+id/tableRow1"
        android:layout_width="500dp"
        android:layout_height="wrap_content"
        android:padding="5dip" >

        <ImageView android:src="@drawable/ic_launcher" />
        <ImageView
       android:src="@android:drawable/btn_star_big_on" />
        <ImageView android:src="@drawable/ic_launcher" />
        </TableRow>

    <TableRow
        android:id="@+id/tableRow2"
        android:layout_width="wrap_content"
        android:layout_height="wrap_content" >

        <ImageView android:src="@android:drawable/btn_star_big_off"
       />
        <TextClock />
        <ImageView
       android:src="@android:drawable/btn_star_big_on" />
        </TableRow>

    <EditText android:hint="Your name" />

    <Button
        android:layout_height="wrap_content"
        android:text="Go" />

</TableLayout>
```

Figure 25.6 shows how the **TableLayout** in Listing 25.6 is rendered.

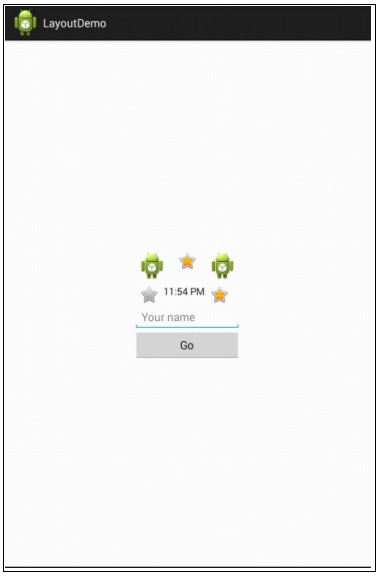

Figure 25.6: Using TableLayout

Grid Layout

A **GridLayout** is similar to a **TableLayout**, but the number of columns must be specified using a **columnCount** attribute. Listing 25.7 shows an example of **GridLayout**.

Listing 25.7: GridLayout example

```
<GridLayout
        xmlns:android="http://schemas.android.com/apk/res/android"
    android:layout_width="wrap_content"
    android:layout_height="wrap_content"
    android:layout_gravity="center"
    android:columnCount="3">

    <!-- 1st row, spanning 3 columns -->
    <TextView
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:text="Enter your name"
        android:layout_columnSpan="3"
        android:textSize="26sp"
        />
    <!-- 2nd row -->
    <TextView android:text="First Name"/>
    <EditText
        android:id="@+id/firstName"
        android:layout_width="200dp"
        android:layout_columnSpan="2"/>

    <!-- 3rd row -->
    <TextView android:text="Last Name"/>
```

```
    <EditText
        android:id="@+id/lastName"
        android:layout_width="200dp"
        android:layout_columnSpan="2"/>

    <!-- 4th row, spanning 3 columns -->
    <Button
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:layout_column="2"
        android:layout_gravity="right"
        android:text="Submit"/>
</GridLayout>
```

Figure 25.7 visualizes the **GridLayout** in Listing 25.7.

Creating A Layout Programmatically

The most common way to create a layout is by using an XML file, as you have seen in the examples in this chapter. However, it is also possible to create a layout programmatically, by instantiating the layout class and passing it to the **addContentView** method in an activity class. For instance, the following code is part of the **onCreate** method of an activity that programmatically creates a **LinearLayout**, sets a couple properties, and passes it to **addContentView**.

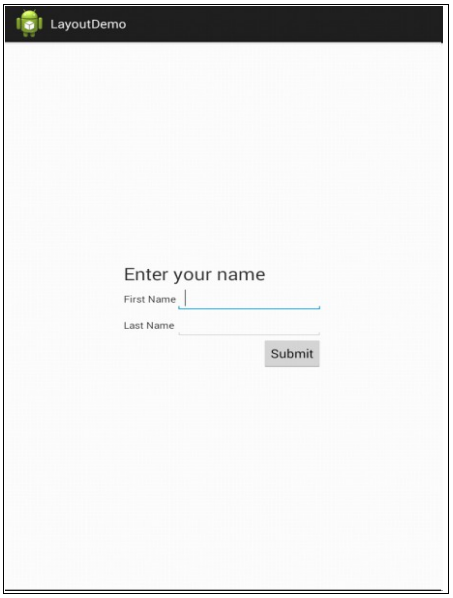

Figure 25.7: Using GridLayout

```
LinearLayout root = new LinearLayout(this);
LinearLayout.LayoutParams matchParent = new
        LinearLayout.LayoutParams(
                LinearLayout.LayoutParams.MATCH_PARENT,
                LinearLayout.LayoutParams.MATCH_PARENT);
root.setOrientation(LinearLayout.VERTICAL);
```

```
root.setGravity(Gravity.CENTER_VERTICAL);
addContentView(root, matchParent);
```

Summary

A layout is responsible for arranging its child views. It directly affect the look and feel of an application. In this chapter you learned some of the layouts available in Android, **LinearLayout**, **RelativeLayout**, **FrameLayout**, **TableLayout**, and **GridLayout**.

Chapter 26
Listeners

Like many GUI systems, Android is event based. User interaction with a view in an activity may trigger an event and you can write code that gets executed when the event occurs. The class that contains code to respond to a certain event is called an event listener. In this chapter you will learn how to handle events and write event listeners.

Overview

Most Android programs are interactive. The user can interact with the application easily thanks to the event-driven programming paradigm the Android framework offers. Examples of events that may occur when the user does something to a view are click, long-click, touch, key, and so on.

To make a program do something in response to a certain event, you need to write a listener for that event. The way to do it is by implementing an interface that is nested in the **android.view.View** class. Table 26.1 shows some of the listener interfaces in **View** and the corresponding method in each interface that will get called when the corresponding event occurs.

Interface	Method
OnClickListener	onClick()
OnLongClickListner	OnLongClick()
OnFocusChangeListener	OnFocusChange()
OnKeyListener	OnKey()
OnTouchListener	OnTouch()

Table 26.1: Listener interfaces in View

Once you create an implementation of a listener interface, you can pass it to the appropriate **setOn*XXX*Listener** method of the view you want to listen to. *XXX* in the method name is the event name. For example, to create a click listener for a button, you would write this in your activity class.

```
private OnClickListener clickListener = new OnClickListener() {
    public void onClick(View view) {
        // code to execute in response to the click event
    }
};

protected void onCreate(Bundle savedValues) {
    ...
    Button button = (Button)findViewById(...);
    button.setOnClickListener(clickListener);
    ...
}
```

Alternatively, you can make your activity class implement the listener interface and provide an implementation of the needed method as part of the activity class.

```
public class MyActivity extends Activity
        implements View.OnClickListener {
    protected void onCreate(Bundle savedValues) {
        ...
        Button button = (Button)findViewById(...);
        button.setOnClickListener(this);
        ...
    }

    public void onClick(View view) {
        // code to execute in response to the click event
    }

    ...
}
```

In addition, there is a shortcut for handling the click event. You can use the **onClick** attribute in the declaration of the target view in the layout file and write a public method in the activity class. The public method must have no return value and take a View argument. For example, if you have this method in your activity class

```
public void showNote(View view) {
    // do something
}
```

you can use this **onClick** attribute in a view to attach the method to the click event of that view.

```
<Button android:onClick="showNote" .../>
```

In the background, Android will create an implementation of the **OnClickListener** interface and attach it to the view.

In the sample applications that follow you will learn how to write event listeners.

Note

A listener runs on the main thread. This means you should use a different thread if your listener takes a long time (say, more than 200ms) to run. Or else, your application will look unresponsive during the execution of the listener code. You have two choices for solving this. You can either use a handler or an **AsyncTask**. The handler is covered in Chapter 43, "Handling the Handler" and **AsyncTask** in Chapter 44, "Asynchronous Tasks." For long-running tasks, you should also consider using the Java Concurrency Utilities.

Using the onClick Attribute

As an example of using the **onClick** attribute to handle the click event of a view, consider the MulticolorClock project that accompanies this book. It is a simple application with a single activity that shows an analog clock that can be clicked to change its color. **AnalogClock** is one of the widgets available on Android, so writing the view for the application is a breeze. The main objective of this project

is to demonstrate how to write a listener by using a callback method in the layout file.

The manifest for MulticolorClock is given in Listing 26.1. There is nothing out of ordinary here and you should not find it difficult to understand.

Listing 26.1: The manifest for MulticolorClock

```
<?xml version="1.0" encoding="utf-8"?>
<manifest
       xmlns:android="http://schemas.android.com/apk/res/android"
    package="com.example.multicolorclock"
    android:versionCode="1"
    android:versionName="1.0" >

    <uses-sdk
        android:minSdkVersion="8"
        android:targetSdkVersion="17" />

    <application
        android:allowBackup="true"
        android:icon="@drawable/ic_launcher"
        android:label="@string/app_name"
        android:theme="@style/AppTheme" >
        <activity
            android:name="com.example.multicolorclock.MainActivity"
            android:label="@string/app_name" >
            <intent-filter>
                <action
       android:name="android.intent.action.MAIN" />
                <category
       android:name="android.intent.category.LAUNCHER" />
            </intent-filter>
        </activity>
    </application>
</manifest>
```

Now comes the crucial part, the layout file. It is called **activity_main.xml** and located under the **res/layout** directory. The layout file is presented in Listing 26.2.

Listing 26.2: The layout file in MulticolorClock

```
<RelativeLayout
       xmlns:android="http://schemas.android.com/apk/res/android"
    xmlns:tools="http://schemas.android.com/tools"
    android:layout_width="match_parent"
    android:layout_height="match_parent"
    android:paddingBottom="@dimen/activity_vertical_margin"
    android:paddingLeft="@dimen/activity_horizontal_margin"
    android:paddingRight="@dimen/activity_horizontal_margin"
    android:paddingTop="@dimen/activity_vertical_margin"
    tools:context=".MainActivity">

    <AnalogClock
        android:id="@+id/analogClock1"
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:layout_alignParentTop="true"
```

```
        android:layout_centerHorizontal="true"
        android:layout_marginTop="90dp"
        android:onClick="changeColor"
    />

</RelativeLayout>
```

The layout file defines a **RelativeLayout** containing an **AnalogClock**. The important part is the **onClick** attribute in the **AnalogClock** declaration.

```
android:onClick="changeColor"
```

This means that when the user presses the **AnalogClock** widget, the **changeColor** method in the activity class will be called. For a callback method like **changeColor** to work, it must have no return value and accept a **View** argument. The system will call this method and pass the widget that was pressed.

The **changeColor** method is part of the **MainActivity** class shown in Listing 26.3.

Listing 26.3: The MainActivity class in MulticolorClock

```
package com.example.multicolorclock;
import android.app.Activity;
import android.graphics.Color;
import android.os.Bundle;
import android.view.Menu;
import android.view.View;
import android.widget.AnalogClock;

public class MainActivity extends Activity {

    int counter = 0;
    int[] colors = { Color.BLACK, Color.BLUE, Color.CYAN,
            Color.DKGRAY, Color.GRAY, Color.GREEN, Color.LTGRAY,
            Color.MAGENTA, Color.RED, Color.WHITE, Color.YELLOW };

    @Override
    protected void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.activity_main);
    }

    @Override
    public boolean onCreateOptionsMenu(Menu menu) {
        // Inflate the menu; this adds items to the action bar if
       it
        // is present.
        getMenuInflater().inflate(R.menu.main, menu);
        return true;
    }

    public void changeColor(View view) {
        if (counter == colors.length) {
            counter = 0;
        }
        view.setBackgroundColor(colors[counter++]);
    }
```

```
}
```

Pay special attention to the **changeColor** method in the **MainActivity** class. When the user presses (or touches) the analog clock, this method will be called and receive the clock object. To change the clock's color, call its **setBackgroundColor** method, passing a color object. In Android, colors are represented by the **android.graphics.Color** class. The class has pre-defined colors that make creating color objects easy. These pre-defined colors include **Color.BLACK**, **Color.Magenta**, **Color.GREEN**, and others. The **MainActivity** class defines an array of **int**s that contains some of the pre-defined colors in **android.graphics.Color**.

```
int[] colors = { Color.BLACK, Color.BLUE, Color.CYAN,
        Color.DKGRAY, Color.GRAY, Color.GREEN, Color.LTGRAY,
        Color.MAGENTA, Color.RED, Color.WHITE, Color.YELLOW };
```

There is also a counter that points to the current index position of **colors**. The **changeColor** method inquiries the value of **counter** and changes it to zero if the value is equal to the array length. It then passes the pointed color to the **setBackgroundColor** method of the **AnalogClock**.

```
view.setBackgroundColor(colors[counter++]);
```

The application is shown in Figure 26.1.

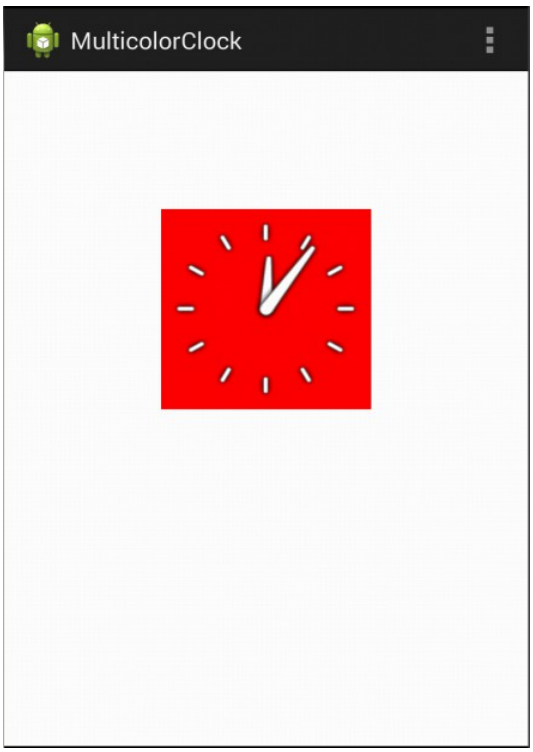

Figure 26.1: The MulticolorClock application

Touch the clock to change its color!

Implementing A Listener

As a second example, the GestureDemo application shows you how to implement the **View.OnTouchListener** interface to handle the touch event. To make it simple, the application only has one activity that contains a grid of cells that can be swapped. The application is shown in Figure 26.2.

Each of the images is an instance of the **CellView** class given in Listing 26.4. It simply extends **ImageView** and adds **x** and **y** fields to store the position in the grid.

Figure 26.2: The GestureDemo application

Listing 26.4: The CellView class

```
package com.example.gesturedemo;
import android.content.Context;
import android.widget.ImageView;

public class CellView extends ImageView {
    int x;
    int y;

    public CellView(Context context, int x, int y) {
        super(context);
```

```
        this.x = x;
        this.y = y;
    }
}
```

There is no layout class for the activity as the layout is built programmatically. This is shown in the **onCreate** method of the **MainActivity** class in Listing 26.5.

Listing 26.5: The MainActivity class

```
package com.example.gesturedemo;
import android.app.Activity;
import android.graphics.drawable.Drawable;
import android.os.Bundle;
import android.view.Gravity;
import android.view.Menu;
import android.view.MotionEvent;
import android.view.View;
import android.view.View.OnTouchListener;
import android.view.ViewGroup;
import android.widget.ImageView;
import android.widget.LinearLayout;

public class MainActivity extends Activity {

    int rowCount = 7;
    int cellCount = 7;
    ImageView imageView1;
    ImageView imageView2;
    CellView[][] cellViews;
    int downX;
    int downY;
    boolean swapping = false;

    @Override
    protected void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);

        LinearLayout root = new LinearLayout(this);
        LinearLayout.LayoutParams matchParent =
                new LinearLayout.LayoutParams(
                LinearLayout.LayoutParams.MATCH_PARENT,
                LinearLayout.LayoutParams.MATCH_PARENT);
        root.setOrientation(LinearLayout.VERTICAL);
        root.setGravity(Gravity.CENTER_VERTICAL);

        addContentView(root, matchParent);


        // create row
        cellViews = new CellView[rowCount][cellCount];
        LinearLayout.LayoutParams rowLayoutParams =
                new LinearLayout.LayoutParams(
                        LinearLayout.LayoutParams.MATCH_PARENT,
                        LinearLayout.LayoutParams.WRAP_CONTENT);

        ViewGroup.LayoutParams cellLayoutParams =
```

```
                new ViewGroup.LayoutParams(
                        ViewGroup.LayoutParams.WRAP_CONTENT,
                        ViewGroup.LayoutParams.WRAP_CONTENT);

        int count = 0;
        for (int i = 0; i < rowCount; i++) {
            CellView[] cellRow = new CellView[cellCount];
            cellViews[i] = cellRow;

            LinearLayout row = new LinearLayout(this);
            row.setLayoutParams(rowLayoutParams);
            row.setOrientation(LinearLayout.HORIZONTAL);
            row.setGravity(Gravity.CENTER_HORIZONTAL);
            root.addView(row);
            // create cells
            for (int j = 0; j < cellCount; j++) {
                CellView cellView = new CellView(this, j, i);
                cellRow[j] = cellView;
                if (count == 0) {
                    cellView.setImageDrawable(
                            getResources().getDrawable(
                                    R.drawable.image1));
                } else if (count == 1) {
                    cellView.setImageDrawable(
                            getResources().getDrawable(
                                    R.drawable.image2));
                } else {
                    cellView.setImageDrawable(
                            getResources().getDrawable(
                                    R.drawable.image3));
                }
                count++;
                if (count == 3) {
                    count = 0;
                }
                cellView.setLayoutParams(cellLayoutParams);
                cellView.setOnTouchListener(touchListener);
                row.addView(cellView);
            }
        }
    }

    @Override
    public boolean onCreateOptionsMenu(Menu menu) {
        getMenuInflater().inflate(R.menu.main, menu);
        return true;
    }

    private void swapImages(CellView v1, CellView v2) {
        Drawable drawable1 = v1.getDrawable();
        Drawable drawable2 = v2.getDrawable();
        v1.setImageDrawable(drawable2);
        v2.setImageDrawable(drawable1);
    }
```

```
    OnTouchListener touchListener = new OnTouchListener() {
        @Override
        public boolean onTouch(View v, MotionEvent event) {
            CellView cellView = (CellView) v;

            int action = getAction();
            switch (action) {
            case (MotionEvent.ACTION_DOWN):
                downX = cellView.x;
                downY = cellView.y;
                return true;
            case (MotionEvent.ACTION_MOVE):
                if (swapping) {
                    return true;
                }
                float x = event.getX();
                float y = event.getY();
                int w = cellView.getWidth();
                int h = cellView.getHeight();
                if (downX < cellCount - 1
                        && x > w && y >= 0 && y <= h) {
                    // swap with right cell
                    swapping = true;
                    swapImages(cellView,
                            cellViews[downY][downX + 1]);
                } else if (downX > 0 && x < 0
                        && y >=0 && y <= h) {
                    // swap with left cell
                    swapping = true;
                    swapImages(cellView,
                            cellViews[downY][downX - 1]);
                } else if (downY < rowCount - 1
                        && y > h && x >= 0 && x <= w) {
                    // swap with cell below
                    swapping = true;
                    swapImages(cellView,
                            cellViews[downY + 1][downX]);
                } else if (downY > 0 && y < 0
                        && x >= 0 && x <= w) {
                    // swap with cell above
                    swapping = true;
                    swapImages(cellView,
                            cellViews[downY - 1][downX]);
                }
                return true;
            case (MotionEvent.ACTION_UP):
                swapping = false;
                return true;
            default:
                return true;
            }
        }
    };
}
```

The **MainActivity** class contains a **View.OnTouchListener** called **touchListener**

that will be attached to every single **CellView** in the grid. The **OnTouchListener** interface has an **onTouch** method that must be implemented. Here is the signature of **onTouch**.

```
public boolean onTouch(View view, MotionEvent event)
```

The method should return **true** if it has consumed the event, which means that the event should not propagate to other views. Otherwise, it should return **false**.

A single touch action by the user causes the **onTouch** method to be called several times. When the user touches the view, the method is called. When the user moves his/her finger, **onTouch** is called. Likewise, **onTouch** is called when the user lifts his/her finger. The second argument to **onTouch**, a **MotionEvent**, contains the information about the event. You can inquire what type of action is triggering the event by calling the **getAction** method on the **MotionEvent**.

```
int action = event.getAction();
```

The return value will be one of the static final **int**s defined in the **MotionEvent** class. For this application we are interested in **MotionEvent.ACTION_DOWN**, **MotionEvent.ACTION_MOVE**, and **MotionEvent.ACTION_UP**. When the user touches the view, the **getAction** method returns a **MotionEvent.ACTION_DOWN**. The code simply stores the location of the event to the **x** and **y** fields and returns **true**.

```
case (MotionEvent.ACTION_DOWN):
    downX = cellView.x;
    downY = cellView.y;
    return true;
```

If the user moves his/her finger to a neighboring cell, the touch action will return a **MotionEvent.ACTION_MOVE** and you need to swap the images of the original cell and the destination cell and set the **swapping** field to true. This would prevent another swapping before the finger is lifted.

Finally, when the user lifts his/her finger, the swapping field is set to false to enable another swapping.

The layout for the activity is built dynamically in the **onCreate** method of the activity class. Each **CellView** is passed the **OnTouchListener** so that the listener will handle the **CellView**'s touch event.

```
cellView.setOnTouchListener(touchListener);
```

Summary

In this chapter you learned the basics of Android event handling and how to write listeners by implementing a nested interface in the **View** class. You have also learned to use the shortcut for handling the click event.

Chapter 27
The Action Bar

The action bar is a rectangular window area that contains the application icon, application name, menus, and other navigation buttons. The action bar normally appears at the top of the window. This chapter explains how to decorate the action bar on Android with API level 11 (Android 3.0) or higher.

Overview

The action bar is represented by the **android.app.ActionBar** class. It should look familiar to any Android user. Figure 27.1 shows the action bar of the Messaging application and Figure 27.2 shows that of Calendar.

Figure 27.1: The action bar of Messaging

Figure 27.2: The action bar of Calendar

The application icon and name on the left of the action bar are there by default. They are both optional and no programming is needed to display them. The system will use the values of the application element's **android:icon** and **android:label** attributes in the manifest. Other item types, such as navigation tabs or an options menu, have to be added using code.

The rightmost icon on the action bar (the one with three little dots) is called the (action) overflow button. When pressed, the overflow button displays action items that may do an action if selected. Important action items can be configured to display directly on the action bar instead of hidden in the overflow. An action item shown on the action bar is called an action button. An action button can have an icon, a label, or both. For example, the action bar in Figure 27.1 contains two action buttons, New Message and Search. The New Message action button has both an icon and a label. The Search action button only has an icon. The action bar in Figure 27.2 also contains two action buttons.

In Android 3.0 or higher, the action bar is shown automatically. You can hide the action bar if you wish by adding this code in the **onCreate** method of your activity.

```
getActionBar().hide();
```

To show a hidden action bar, call the **show** method:

```
getActionBar().hide();
```

The next sections show how to add action items and drop-down navigation.

Note
You can download Android's icon pack that contains icons for your action bar from this site.

```
http://developer.android.com/downloads/design/
Android_Design_Icons_20131106.zip
```

Adding Action Items

To add action items to the action overflow, follow these two steps.

1. Create a menu in an xml file and save it under the **res/menu** directory. ADT Eclipse will add a field to your **R.menu** class so that you can load the menu in your application. The field name is the same as the XML file minus the extension. If the XML file is called **main_activity_menu.xml**, for example, the field will be called **main_activity_menu**.
2. In your activity class, override the **onCreateOptionsMenu** method and call **getMenuInflater().inflate()**, passing the menu to be loaded and the menu passed to the method, like this.

```
@Override
public boolean onCreateOptionsMenu(Menu menu) {
    getMenuInflater().inflate(R.menu.main, menu);
    return true;
}
```

An action item that does nothing is useless. To respond to a action item being selected, you must override the **onOptionsItemSelected** method in your activity class. This method is called every time an item menu is selected and the system will pass the **MenuItem** selected. The signature of the method is as follows.

```
public boolean onOptionsItemSelected(MenuItem item);
```

You can find out which menu item was selected by calling the **getItemId** on the **MenuItem** argument. Normally you would use a **switch** statement like this:

```
switch (item.getItemId()) {
    case R.id.action_1:
        // do something
        return true;
    case R.id.action_2:
        // do something else
        return true;
...
```

Now that you know the theory, let's add some item actions. The ActionBarDemo application shows how to do it. It adds three action items to the action bar.

As usual, let's start with the manifest, which for this example is shown in Listing 27.1.

Listing 27.1: The manifest for ActionBarDemo

```
<?xml version="1.0" encoding="utf-8"?>
<manifest
       xmlns:android="http://schemas.android.com/apk/res/android"
    package="com.example.actionbardemo"
    android:versionCode="1"
    android:versionName="1.0" >
```

```
    <uses-sdk
        android:minSdkVersion="11"
        android:targetSdkVersion="18" />

    <application
        android:allowBackup="true"
        android:icon="@drawable/ic_launcher"
        android:label="@string/app_name"
        android:theme="@style/AppTheme" >
        <activity
            android:name="com.example.actionbardemo.MainActivity"
            android:label="@string/app_name" >
            <intent-filter>
                <action android:name="android.intent.action.MAIN"/>
                <category
android:name="android.intent.category.LAUNCHER" />
            </intent-filter>
        </activity>
    </application>
</manifest>
```

It is good practice to list action names in a resource file. Listing 27.2 shows the **strings.xml** file that contains three strings for the action items, **action_capture**, **action_profile**, and **action_about**.

Listing 27.2: The res/values/strings.xml

```
<?xml version="1.0" encoding="utf-8"?>
<resources>
    <string name="app_name">ActionBarDemo</string>
    <string name="action_capture">Capture</string>
    <string name="action_profile">Profile</string>
    <string name="action_about">About</string>
    <string name="hello_world">Hello world!</string>
</resources>
```

Next, create an XML file under **res/menu**. If you used ADT Eclipse to create the Android application, one has been created for you. You just need to add **item** elements to it. Listing 27.3 shows the menu for the action items.

Listing 27.3: The res/menu/main.xml

```
<menu xmlns:android="http://schemas.android.com/apk/res/android">
    <item
        android:id="@+id/action_capture"
        android:orderInCategory="100"
        android:showAsAction="ifRoom|withText"
        android:icon="@drawable/icon1"
        android:title="@string/action_capture"/>

    <item
        android:id="@+id/action_profile"
        android:orderInCategory="200"
        android:showAsAction="ifRoom|withText"
        android:icon="@drawable/icon2"
        android:title="@string/action_profile"/>

    <item
```

```
        android:id="@+id/action_about"
        android:orderInCategory="50"
        android:showAsAction="never"
        android:title="@string/action_about"/>
</menu>
```

The **item** element may have any of these attributes.

- **android:id**. A unique identifier to refer to the action item in the program.
- **android:orderInCategory**. The order number for this item. An item with a smaller number will be shown before items with larger numbers.
- **android:icon**. The icon for this action item if it is shown as an action button (directly on the action bar).
- **android:title**. The action label.
- **android:showAsAction**. The value can be one or a combination of these values: **ifRoom**, **never**, **withText**, **always**, and **collapseActionView**. Populating this attribute with **never** indicates that this item will never be shown on the action bar directly. On the other hand, **always** forces the system to always display this item as an action button. However, be cautious when using this value as if there is not enough room on the action bar, what will be displayed will be unpredictable. Instead, use **ifRoom** to display an item as an action button if there is room. The **withText** value will display this item with a label if this item is being displayed as an action button.

The complete list of attributes for the **item** element can be found here.

```
http://developer.android.com/guide/topics/resources/menu-
resource.html
```

Finally, Listing 27.4 presents the **MainActivity** class for the application.

Listing 27.4: The MainActivity class

```
package com.example.actionbardemo;
import android.app.Activity;
import android.app.AlertDialog;
import android.os.Bundle;
import android.view.Menu;
import android.view.MenuItem;

public class MainActivity extends Activity {

    @Override
    protected void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.activity_main);
    }

    @Override
    public boolean onCreateOptionsMenu(Menu menu) {
        getMenuInflater().inflate(R.menu.main, menu);
        return true;
    }

    @Override
    public boolean onOptionsItemSelected(MenuItem item) {
        // Handle presses on the action bar items
        switch (item.getItemId()) {
```

```
            case R.id.action_profile:
                showAlertDialog("Profile", "You selected Profile");
                return true;
            case R.id.action_capture:
                showAlertDialog("Settings",
                        "You selected Settings");
                return true;
            case R.id.action_about:
                showAlertDialog("About", "You selected About");
                return true;
            default:
                return super.onOptionsItemSelected(item);
        }
    }

    private void showAlertDialog(String title, String message) {
        AlertDialog alertDialog = new
                AlertDialog.Builder(this).create();
        alertDialog.setTitle(title);
        alertDialog.setMessage(message);
        alertDialog.show();
    }
}
```

Noticed that the activity class overrides the **onOptionsItemSelected** method? Selecting an item will invoke the **showAlertDialog** method that shows an **AlertDialog**.

Figure 27.3 shows three action items in ActionBarDemo. Two of the items are displayed as action buttons.

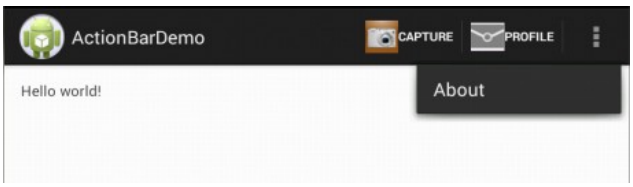

Figure 27.3: The ActionBarDemo application

Adding Dropdown Navigation

A dropdown list can be used as a navigation mode. The visual difference between a dropdown list and an options menu is that a dropdown list always displays the selected item on the action bar and hide the other options. On the other hand, an options menu may hide all of the items or show all or some of them as action buttons. Figure 27.4 shows dropdown navigation in Calendar.

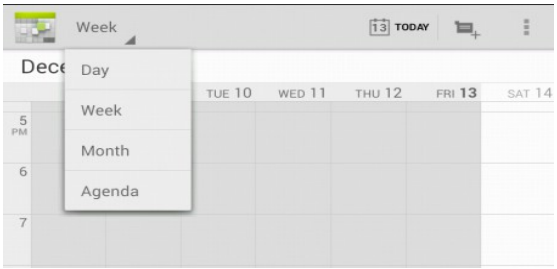

Figure 27.4: Drop down navigation in Calendar

To add drop-down navigation to the action bar, follow these three steps.

1. Declare a string array in your **strings.xml** file under **res/values**.
2. In your activity class, add an implementation of **ActionBar.OnNavigationListener** to respond to item selection.
3. Create a **SpinnerAdapter** in the **onCreate** method of your activity, pass **ActionBar.NAVIGATION_MODE_LIST** to the **ActionBar**'s **setNavigationMode** method, and pass the **SpinnerAdapter** and **OnNavigationListener** to the **ActionBar**'s **setListNavigationCallbacks** method.

```
SpinnerAdapter spinnerAdapter =
        ArrayAdapter.createFromResource(this,
        R.array.colors,
        android.R.layout.simple_spinner_dropdown_item);
ActionBar actionBar = getActionBar();
actionBar.setNavigationMode(
        ActionBar.NAVIGATION_MODE_LIST);
actionBar.setListNavigationCallbacks(spinnerAdapter,
        onNavigationListener);
```

As an example, the DropDownNavigationDemo application shows how to add dropdown navigation to the action bar. The application adds a list of five colors to the action bar. Selecting a color changes the window background color with the selected color.

Listing 27.5 shows the manifest for the application.

Listing 27.5: The DropDownNavigationDemo manifest

```
<?xml version="1.0" encoding="utf-8"?>
<manifest
       xmlns:android="http://schemas.android.com/apk/res/android"
    package="com.example.dropdownnavigationdemo"
    android:versionCode="1"
    android:versionName="1.0" >

    <uses-sdk
        android:minSdkVersion="14"
        android:targetSdkVersion="18" />

    <application
        android:allowBackup="true"
        android:icon="@drawable/ic_launcher"
        android:label="@string/app_name"
        android:theme="@style/AppTheme" >
        <activity
```

```
android:name="com.example.dropdownnavigationdemo.MainActivity"
            android:label="@string/app_name"
            android:theme="@style/MyTheme">
            <intent-filter>
                <action android:name="android.intent.action.MAIN"/>
                <category
android:name="android.intent.category.LAUNCHER" />
            </intent-filter>
        </activity>
    </application>
</manifest>
```

Listing 27.6 shows a **string-array** element that will be used to populate the drop-down. There are five items in the array.

Listing 27.6: The res/values/strings.xml file

```
<?xml version="1.0" encoding="utf-8"?>
<resources>
    <string name="app_name">DropDownNavigationDemo</string>
    <string name="action_settings">Settings</string>
    <string name="hello_world">Hello world!</string>

    <string-array name="colors">
        <item>White</item>
        <item>Red</item>
        <item>Green</item>
        <item>Blue</item>
        <item>Yellow</item>
    </string-array>
</resources>
```

Listing 27.7 shows the **MainActivity** class for the application.

Listing 27.7: The MainActivity class

```
package com.example.dropdownnavigationdemo;
import android.app.ActionBar;
import android.app.ActionBar.OnNavigationListener;
import android.app.Activity;
import android.graphics.Color;
import android.os.Bundle;
import android.view.Menu;
import android.widget.ArrayAdapter;
import android.widget.SpinnerAdapter;

public class MainActivity extends Activity {

    @Override
    protected void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.activity_main);
        SpinnerAdapter spinnerAdapter =
                ArrayAdapter.createFromResource(this,
                R.array.colors,
                android.R.layout.simple_spinner_dropdown_item);
        ActionBar actionBar = getActionBar();
        actionBar.setNavigationMode(
                ActionBar.NAVIGATION_MODE_LIST);
```

```
        actionBar.setListNavigationCallbacks(spinnerAdapter,
                onNavigationListener);
    }

    @Override
    public boolean onCreateOptionsMenu(Menu menu) {
        getMenuInflater().inflate(R.menu.main, menu);
        return true;
    }

    OnNavigationListener onNavigationListener = new
            OnNavigationListener() {
        @Override
        public boolean onNavigationItemSelected(
                int position, long itemId) {
            String[] colors = getResources().
                    getStringArray(R.array.colors);
            String selectedColor = colors[position];

            getWindow().getDecorView().setBackgroundColor(
                    Color.parseColor(selectedColor));
            return true;
        }
    };
}
```

Figure 27.5 shows the dropdown navigation.

Figure 27.5: Dropdown navigation on the action bar

Note that the action bar has been styled using the **styles.xml** file in Listing 27.8.

Listing 27.8: The res/values/styles.xml file

```
<resources>
    <style name="AppBaseTheme" parent="android:Theme.Light">
    </style>

    <style name="AppTheme" parent="AppBaseTheme">
    </style>

    <style name="MyTheme"
           parent="@android:style/Widget.Holo.Light">
        <item
       name="android:actionBarStyle">@style/MyActionBar</item>
    </style>

    <style name="MyActionBar"
```

```
parent="@android:style/Widget.Holo.Light.ActionBar.Solid.Inverse">
        <item
name="android:background">@android:color/holo_blue_bright</item>
    </style>
</resources>
```

For more information on styling UI components, see Chapter 31, “Styles and Themes.”

Going Back Up

You can set the application icon and activity label in the action bar of an activity so that the application will go one level back up when the icon is pressed. Figure 27.6 shows an action bar whose **displayHomeAsUp** property is set to **true** (indicated by the left arrow to the left of the icon). Compare this with the action bar in Figure 27.7 where its **displayHomeAsUp** property is set to **false**.

Figure 27.6: displayHomeAsUp set to true

Figure 27.7: displayHomeAsUp set to false

To enable **displayHomeAsUp**, you need to set the **parentActivityName** element in the activity declaration in the Android manifest:

```
<activity android:name="com.example.d1.ShowContactActivity"
    android:parentActivityName=".MainActivity">
</activity>
```

You must also leave the **displayHomeAsUpEnabled** property of the action bar to its default value (**true**). Setting it to **false**, as shown in the code below, will disable it.

```
getActionBar().setDisplayHomeAsUpEnabled(false);
```

Summary

The action bar provides a space for the application icon, application name, and navigation modes. This chapter shows how to add action items and dropdown navigation to the action bar.

Chapter 28
Menus

Menus are a common feature in many graphical user interface (GUI) systems. Their primary role is to provide shortcuts to certain actions.

This chapter looks at Android menus closely and provides three sample applications.

Overview

Pre-3.0 Android devices shipped with a (hardware) button for showing menus in the active application. Starting from Android 3.0, the action bar is the recommended way of achieving the same thing, in effect making a hardware Menu button redundant. With the hardware menu button gone, "soft" menus have become even more important than ever.

There are three types of menus in Android:

- Options menu
- Context menu
- Popup menu

The options menu is the type of menu you normally incorporate in the action bar, as you have seen in Chapter 27, "The Action Bar." In this chapter you will look at the options menu more closely and learn about the other two. Thankfully, no matter what kind of menu you're using in your app, you use the same API. And, yes, you can use different types of menus in the same application.

Like many other things in Android, you can define menus declaratively or programmatically. The first method offers more flexibility than the second because it allows you to change menu items using a text editor. Doing so programmatically, on the other hand, would require you to change your program and recompile every time you need to edit your menu.

Here are the three things you need to do when working with options and context menus.

1. Create a menu in an xml file and save it under the **res/menu** directory.
2. In your activity class, override either onCreateOptionsMenu or onCreateContextMenu, depending on your type of menu. Then, inside the menu, call **getMenuInflater().inflate()**, passing the menu to be used.
3. In your activity class, override either **onOptionsItemSelected** or **onContextItemSelected**, depending on the type of your menu.

Popup menus are a bit different. For working with them, you do the following:

1. Create a menu in an xml file and save it under the **res/menu** directory.
2. In your activity class, create a **PopupMenu** object and a **PopupMenu.OnMenuItemClickListener** object. In the listener class you

define a method that handles the click event that occurs when one of the popup menu items is selected.

The Menu File

Employing the first method, you start by creating an XML file and place it under the **res/menu** directory. The XML file must have the following structure.

```
<menu xmlns:android="http://schemas.android.com/apk/res/android">
    <group>...</group>
    <group>...</group>
    ...
    <item>...</item>
    <item>...</item>
    ...
</menu>
```

The root element is **menu** and it can contain any number of group and item elements. The group element represents a menu group and the item element represents a menu item.

For every menu file you create, ADT Eclipse will add a field to your **R.menu** class so that you can load the menu in your application. The field name is the same as the XML file minus the extension. If the XML file is called **main_activity_menu.xml**, for example, the field in **R.menu** will be called **main_activity_menu**.

The Options Menu

The OptionsMenuDemo application is a simple application that uses an options menu in its action bar. It is similar to the application that demonstrates the action bar in Chapter 27, “The Action Bar.”

The **AndroidManifest.xml** file for this application is shown in Listing 28.1.

Listing 28.1: AndroidManifest.xml for OptionsMenuDemo

```
<?xml version="1.0" encoding="utf-8"?>
<manifest
       xmlns:android="http://schemas.android.com/apk/res/android"
    package="com.example.optionsmenudemo"
    android:versionCode="1"
    android:versionName="1.0" >

    <uses-sdk
        android:minSdkVersion="18"
        android:targetSdkVersion="18" />

    <application
        android:allowBackup="true"
        android:icon="@drawable/ic_launcher"
        android:label="@string/app_name"
        android:theme="@style/AppTheme" >
        <activity
```

```
            android:name="com.example.optionsmenudemo.MainActivity"
            android:label="@string/app_name" >
            <intent-filter>
                <action android:name="android.intent.action.MAIN"/>
                <category
android:name="android.intent.category.LAUNCHER"/>
            </intent-filter>
        </activity>
    </application>
</manifest>
```

The manifest declares an activity, which is represented by the **MainActivity** class.

The menu for this application is defined in the **res/menu/options_menu.xml** file in Listing 28.2. It has three menu items.

Listing 28.2: The options_menu.xml File

```
<menu xmlns:android="http://schemas.android.com/apk/res/android">
    <item
        android:id="@+id/action_capture"
        android:orderInCategory="100"
        android:showAsAction="ifRoom|withText"
        android:icon="@drawable/icon1"
        android:title="@string/action_capture"/>

    <item
        android:id="@+id/action_profile"
        android:orderInCategory="200"
        android:showAsAction="ifRoom|withText"
        android:icon="@drawable/icon2"
        android:title="@string/action_profile"/>

    <item
        android:id="@+id/action_about"
        android:orderInCategory="50"
        android:showAsAction="never"
        android:title="@string/action_about"/>
</menu>
```

The titles for the menu items reference the strings defined in the **res/values/strings.xml** file in Listing 28.3.

Listing 28.3: strings.xml for OptionsMenuDemo

```
<?xml version="1.0" encoding="utf-8"?>
<resources>
    <string name="app_name">OptionsMenuDemo</string>
    <string name="action_capture">Capture</string>
    <string name="action_profile">Profile</string>
    <string name="action_about">About</string>
    <string name="hello_world">Hello world!</string>
</resources>
```

The activity class for the application, the **MainActivity** class, is shown in Listing 28.4.

Listing 28.4: MainActivity for OptionsMenuDemo

```
package com.example.optionsmenudemo;
import android.app.Activity;
```

```
import android.app.AlertDialog;
import android.os.Bundle;
import android.view.Menu;
import android.view.MenuItem;

public class MainActivity extends Activity {
    @Override
    protected void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.activity_main);
    }

    @Override
    public boolean onCreateOptionsMenu(Menu menu) {
        getMenuInflater().inflate(R.menu.options_menu, menu);
        return true;
    }

    @Override
    public boolean onOptionsItemSelected(MenuItem item) {
        // Handle click on menu items
        switch (item.getItemId()) {
            case R.id.action_profile:
                showAlertDialog("Profile", "You selected Profile");
                return true;
            case R.id.action_capture:
                showAlertDialog("Settings",
                        "You selected Settings");
                return true;
            case R.id.action_about:
                showAlertDialog("About", "You selected About");
                return true;
            default:
                return super.onOptionsItemSelected(item);
        }
    }

    private void showAlertDialog(String title, String message) {
        AlertDialog alertDialog = new
                AlertDialog.Builder(this).create();
        alertDialog.setTitle(title);
        alertDialog.setMessage(message);
        alertDialog.show();
    }
}
```

To use the options menu you need to override the **onCreateOptionsMenu** and **onOptionsItemSelected** methods. The **onCreateOptionsMenu** method is called when the activity is built. You should call the menu inflater and inflate your menu here. In addition, the **onOptionsItemSelected** method handles menu item selection.

Note that the options menu is integrated with the activity so that you do not need to create your own listener to handle item selection.

If you run the application, you will see an activity like the one in Figure 28.1.

Take a look at the action bar and try selecting one of the menu items. Every time you select a menu item, an **AlertDialog** will be shown to notify what you have selected.

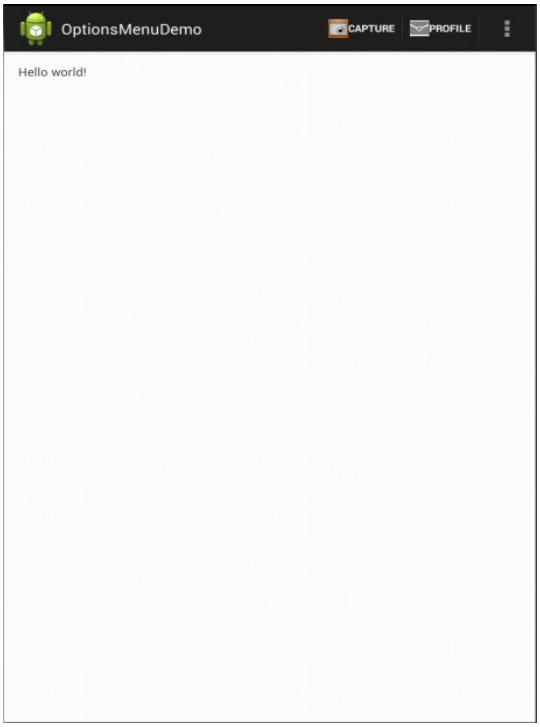

Figure 28.1: OptionsMenuDemo

The Context Menu

The ContextMenuDemo application shows how you can use a context menu in your application. The main activity of the application features an image button that you can long-press to display a context menu.

The **AndroidManifest.xml** file for this application is printed in Listing 28.5.

Listing 28.5: AndroidMenifest.xml for ContextMenuDemo

```
<?xml version="1.0" encoding="utf-8"?>
<manifest
       xmlns:android="http://schemas.android.com/apk/res/android"
    package="com.example.contextmenudemo"
    android:versionCode="1"
    android:versionName="1.0" >

    <uses-sdk
        android:minSdkVersion="18"
        android:targetSdkVersion="18" />

    <application
        android:allowBackup="true"
```

```
        android:icon="@drawable/ic_launcher"
        android:label="@string/app_name"
        android:theme="@style/AppTheme" >
        <activity
            android:name="com.example.contextmenudemo.MainActivity"
            android:label="@string/app_name" >
            <intent-filter>
                <action android:name="android.intent.action.MAIN"/>
                <category
android:name="android.intent.category.LAUNCHER"/>
            </intent-filter>
        </activity>
    </application>
</manifest>
```

The **context_menu.xml** file in Listing 28.6 is a menu file that defines menu items for the context menu used in the application.

Listing 28.6: context_menu.xml for ContextMenuDemo

```
<menu xmlns:android="http://schemas.android.com/apk/res/android">
    <item
        android:id="@+id/action_rotate"
        android:title="@string/action_rotate"/>
    <item
        android:id="@+id/action_resize"
        android:title="@string/action_resize"/>
</menu>
```

The menu file defines two menu items, whose titles get their values from the res/values/strings.xml file in Listing 28.7.

Listing 28.7: strings.xml for ContextMenuDemo

```
<?xml version="1.0" encoding="utf-8"?>
<resources>
    <string name="app_name">ContextMenuDemo</string>
    <string name="action_settings">Settings</string>
    <string name="action_rotate">Rotate</string>
    <string name="action_resize">Resize</string>
    <string name="hello_world">Hello world!</string>
</resources>
```

Finally, Listing 28.8 shows the **MainActivity** class for the application. There are two methods that you need to override to use a context menu, **onCreateContextMenu** and **onContextItemSelected**. The **onCreateContextMenu** method is called when the activity is built. You should inflate your menu here.

The **onContextItemSelected** method is called every time a menu item in the context menu is selected.

Listing 28.8: MainActivity for ContextMenuDemo

```
package com.example.contextmenudemo;
import android.app.Activity;
import android.app.AlertDialog;
import android.os.Bundle;
import android.view.ContextMenu;
import android.view.ContextMenu.ContextMenuInfo;
import android.view.MenuInflater;
```

```
import android.view.MenuItem;
import android.view.View;
import android.widget.ImageButton;

public class MainActivity extends Activity {
    @Override
    protected void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.activity_main);
        ImageButton imageButton = (ImageButton)
            findViewById(R.id.button1);
        registerForContextMenu(imageButton);
    }

    @Override
    public void onCreateContextMenu(ContextMenu menu, View v,
            ContextMenuInfo menuInfo) {
        super.onCreateContextMenu(menu, v, menuInfo);
        MenuInflater inflater = getMenuInflater();
        inflater.inflate(R.menu.context_menu, menu);
    }
    @Override
    public boolean onContextItemSelected(MenuItem item) {
        switch (item.getItemId()) {
            case R.id.action_rotate:
                showAlertDialog("Rotate", "You selected Rotate ");
                return true;
            case R.id.action_resize:
                showAlertDialog("Resize", "You selected Resize");
                return true;
            default:
                return super.onContextItemSelected(item);
        }
    }

    private void showAlertDialog(String title, String message) {
        AlertDialog alertDialog = new
                AlertDialog.Builder(this).create();
        alertDialog.setTitle(title);
        alertDialog.setMessage(message);
        alertDialog.show();
    }
}
```

Figure 28.2 shows the application. If you press (or click) the image button long enough, it will show the context menu. Note that the image comes from the Android system as is defined in the layout file.

The Popup Menu

A popup menu is associated with a view and is shown every time an event occurs to the view. The **PopupMenuDemo** application shows how to use a popup menu. It uses a button that displays a popup menu when it is clicked. Listing 28.9 shows the **AndroidManifest.xml** file for **PopupMenuDemo**.

Figure 28.2: The context menu

Listing 28.9: AndroidMenifest.xml for PopupMenuDemo

```
<?xml version="1.0" encoding="utf-8"?>
<manifest
       xmlns:android="http://schemas.android.com/apk/res/android"
    package="com.example.popupmenudemo"
    android:versionCode="1"
    android:versionName="1.0" >

    <uses-sdk
        android:minSdkVersion="18"
        android:targetSdkVersion="18" />

    <application
        android:allowBackup="true"
        android:icon="@drawable/ic_launcher"
        android:label="@string/app_name"
        android:theme="@style/AppTheme" >
        <activity
            android:name="com.example.popupmenudemo.MainActivity"
            android:label="@string/app_name" >
            <intent-filter>
                <action android:name="android.intent.action.MAIN"/>
                <category
android:name="android.intent.category.LAUNCHER"/>
            </intent-filter>
        </activity>
    </application>
```

```
</manifest>
```

The manifest in Listing 28.9 is a standard XML file that you've seen many times. It has one activity with a button that will activate the menu shown in Listing 28.10.

Listing 28.10: popup_menu.xml for PopupMenuDemo

```
<menu xmlns:android="http://schemas.android.com/apk/res/android">
    <item
        android:id="@+id/action_delete"
        android:title="@string/action_delete"/>
    <item
        android:id="@+id/action_copy"
        android:title="@string/action_copy"/>
</menu>
```

The menu in Listing 28.10 has two menu items. The titles for the items refer to the strings defined in the **res/values/strings.xml** file in Listing 28.11.

Listing 28.11: strings.xml for PopupMenuDemo

```
<?xml version="1.0" encoding="utf-8"?>
<resources>
    <string name="app_name">PopupMenuDemo</string>
    <string name="action_settings">Settings</string>
    <string name="action_delete">Delete</string>
    <string name="action_copy">Copy</string>
    <string name="show_menu">Show Popup</string>
</resources>
```

Finally, Listing 28.12 shows the **MainActivity** class for the application.

Listing 28.12: MainActivity for PopupMenuDemo

```
package com.example.popupmenudemo;
import android.app.Activity;
import android.os.Bundle;
import android.util.Log;
import android.view.MenuItem;
import android.view.View;
import android.widget.Button;
import android.widget.PopupMenu;

public class MainActivity extends Activity {

    PopupMenu popupMenu;
    PopupMenu.OnMenuItemClickListener menuItemClickListener;

    @Override
    protected void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.activity_main);
        menuItemClickListener =
                new PopupMenu.OnMenuItemClickListener() {
            @Override
            public boolean onMenuItemClick(MenuItem item) {
                switch (item.getItemId()) {
                case R.id.action_delete:
                    Log.d("menu", "Delete clicked");
                    return true;
                case R.id.action_copy:
```

```
                    Log.d("menu", "Copy clicked");
                    return true;
                default:
                    return false;
                }
            }
        };
        Button button = (Button) findViewById(R.id.button1);
        popupMenu = new PopupMenu(this, button);

       popupMenu.setOnMenuItemClickListener(menuItemClickListener);
        popupMenu.inflate(R.menu.popup_menu);
    }

    public void showPopupMenu(View view) {
        popupMenu.show();
    }
}
```

Unlike the options menu and context menu, the popup menu requires that you create a menu object and a listener object for handling item selection.

In the **onCreate** method of **MainActivity**, you create a **PopupMenu** object and a **PopupMenu.OnMenuItemClickListener** object. You then pass the listener to the **PopupMenu**. The listener class handles menu item clicks.

The **showPopupMenu** method in **MainActivity** is associated with the button using the **onClick** attribute of the button in the main activity layout file. The method shows the popup menu.

Figure 28.3 shows the popup menu that displays when the button is clicked.

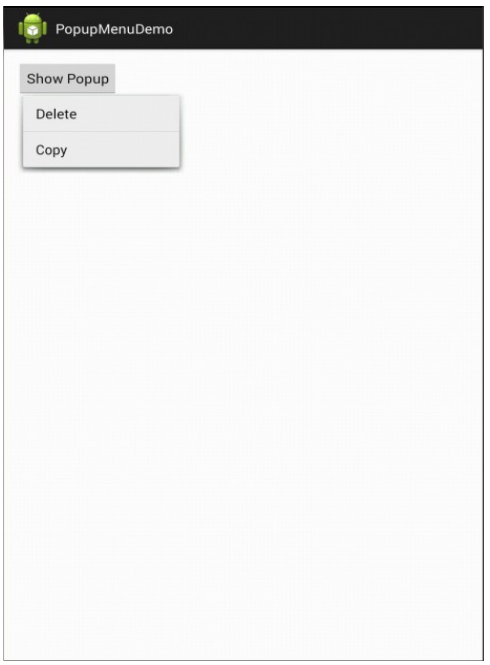

Figure 28.3: PopupMenuDemo

Summary

In this chapter you learned how to use menus to provide shortcuts to certain actions. There are three types of menus in Android, options menus, context menus, and popup menus.

Chapter 29
ListView

A **ListView** is a view for showing a scrollable list of items. which may come from a list adapter or an array adapter. Selecting an item in a **ListView** triggers an event for which you can write a listener.

If an activity contains only one view that is a **ListView**, you can extend **ListActivity** instead of **Activity** in your activity class. Using **ListActivity** is convenient as many tasks will be done for you.

This chapter shows how you can use the **ListView** and **ListActivity** as well as create a custom **ListAdapter** and style a **ListView** in three sample applications.

Overview

Technically, **android.widget.ListView**, the template for creating a **ListView**, is a descendant of the **View** class. You can use it the same way you would other views. What makes **ListView** a bit tricky to use is the fact that you have to obtain a data source for it in the form of a **ListAdapter**. The **ListAdapter** also supply the layout for each item on the **ListView**, so the **ListAdapter** really plays a very important role in the life of a **ListView**.

The **android.widget.ListAdapter** interface is a subinterface of **android.widget.Adapter**. The close relatives of this interface are shown in Figure 29.1.

Creating a **ListAdapter** is explained in the next section, "Creating a ListAdapter." Once you have a **ListAdapter**, you can pass it to a **ListView**'s **setAdapter** method:

```
listView.setAdapter(listAdapter);
```

You can also write a listener that implements **AdapterView.OnItemClickListener** and pass it to the **ListView**'s **setOnItemClickListener** method. The listener will be notified every time a list item is selected and you can write code to handle it, like so.

```
listView.setOnItemClickListener(new
        AdapterView.OnItemClickListener() {
    @Override
    public void onItemClick(AdapterView<?> parent, final View view,
            int position, long id) {
        // handle item
    });
```

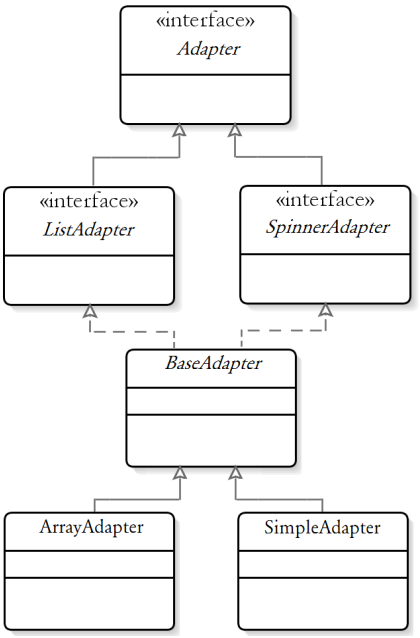

Figure 29.1: The parents and implementations of ListAdapter

Creating A ListAdapter

As mentioned in the previous section, the trickiest part of using a **ListView** is creating a data source for it. You need a **ListAdapter** and as you can see in Figure 29.1 you have at least two implementations of **ListAdapter** that you can use.

One of the concrete implementations of **ListAdapter** is the **ArrayAdapter** class. An **ArrayAdapter** is backed by an array of objects. The string returned by the **toString** method of each object is used to populate each item in the **ListView**.

The **ArrayAdapter** class offers a couple of constructors. All of them require that you pass a **Context** and a resource identifier that points to a layout that contains a **TextView**. This is because each item in a **ListView** is a **TextView**. These are some of the constructors in the **ArrayAdapter** class.

```
public ArrayAdapter(android.content.Context context, int
       resourceId)

public ArrayAdapter(android.content.Context context, int
       resourceId,
        T[] objects)
```

If you do not pass an object array in your constructor, you will have to pass one later. As for the resource identifier, Android provides some pre-defined layouts for a **ListAdapter**. The identifiers to these layouts can be found in the **android.R.layout** class. For example, you can create an **ArrayAdapter** using this code snippet.

```
ArrayAdapter<String> adapter = new ArrayAdapter<String>(this,
        android.R.layout.simple_list_item_1, objects);
```

Using **android.R.layout.simple_list_item_1** will create a **ListView** with the

simplest layout where the text of each item is printed in black. Alternatively, you can use **android.R.layout.simple_expandable_list_item_1**. However, you probably want to create your own layout and pass it instead. This way you would have more control over the look and feel of your **ListView**.

Most of the time you can use a string array as the data source for your **ListView**. You can create a string array programmatically or declaratively. Doing it programmatically is simple and you do not have to deal with an external resource:

```
String[] objects = { "item1", "item2","item-n" };
```

The disadvantage is updating the array would require you to recompile your class. Creating a string array declaratively, on the other hand, gives you more flexibility as you can easily edit the elements.

To create a string array declaratively, start by creating a **string-array** element in your **strings.xml** file under **res/values**. For example, the following is a **string-array** named **players**.

```
<string-array name="players">
    <item>Player 1</item>
    <item>Player 2</item>
    <item>Player 3</item>
    <item>Player 4</item>
</string-array>
```

When you save the **strings.xml** file, ADT Eclipse will update your **R** generated class and add a static final class named **array**, if none exists, as well as add a resource identifier for the **string-array** element to the **array** class. As a result, you now have this resource identifier to access your string array from your code:

```
R.array.players
```

To convert the string-array to a Java string array, use this code.

```
String[] values = getResources().getStringArray(R.array.players);
```

You can then use this string array to create an **ArrayAdapter**.

Using A ListView

The **ListViewDemo1** application shows how to use a **ListView** that is backed by an **ArrayAdapter**. The array that supplies values to the **ArrayAdapter** is a string array defined in the **strings.xml** file. Listing 29.1 shows the **strings.xml** file.

Listing 29.1: The res/values/strings.xml file for ListViewDemo1

```
<?xml version="1.0" encoding="utf-8"?>
<resources>
    <string name="app_name">ListViewDemo1</string>
    <string name="action_settings">Settings</string>

    <string-array name="players">
        <item>Player 1</item>
        <item>Player 2</item>
        <item>Player 3</item>
        <item>Player 4</item>
        <item>Player 5</item>
    </string-array>
```

```
</resources>
```

The layout for the **ArrayAdapter** is defined in the **list_item.xml** file presented in Listing 29.2. It is located under **res/layout** and contains a **TextView** element. This layout will be used as the layout for each item in the **ListView**.

Listing 29.2: The list_item.xml file

```
<?xml version="1.0" encoding="utf-8"?>
<TextView
       xmlns:android="http://schemas.android.com/apk/res/android"
    android:id="@+id/list_item"
        android:layout_width="fill_parent"
        android:layout_height="fill_parent"
        android:padding="7dip"
        android:textSize="16sp"
        android:textColor="@android:color/holo_green_dark"
        android:textStyle="bold" >
</TextView>
```

The application consists of only one activity, **MainActivity**. The layout file (**activity_main.xml**) for the activity is given in Listing 29.3 and the **MainActivity** class in Listing 29.4.

Listing 29.3: The activity_main.xml file for ListViewDemo1

```
<LinearLayout
    xmlns:android="http://schemas.android.com/apk/res/android"
    android:orientation="vertical"
    android:layout_width="fill_parent"
    android:layout_height="fill_parent">
    <ListView
        android:id="@+id/listView1"
        android:layout_width="wrap_content"
        android:layout_height="wrap_content" />
</LinearLayout>
```

Listing 29.4: The MainActivity class for ListViewDemo1

```
package com.example.listviewdemo1;
import android.app.Activity;
import android.app.AlertDialog;
import android.os.Bundle;
import android.util.Log;
import android.view.Menu;
import android.view.View;
import android.widget.AdapterView;
import android.widget.ArrayAdapter;
import android.widget.ListView;

public class MainActivity extends Activity {
    @Override
    protected void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.activity_main);
        String[] values = getResources().getStringArray(
                R.array.players);

        ArrayAdapter<String> adapter = new ArrayAdapter<String>(
```

```
                this, R.layout.list_item, values);

        ListView listView = (ListView)
       findViewById(R.id.listView1);
        listView.setAdapter(adapter);
        listView.setOnItemClickListener(new
                AdapterView.OnItemClickListener() {
            @Override
            public void onItemClick(AdapterView<?> parent,
                    final View view, int position, long id) {
                String item = (String)
                        parent.getItemAtPosition(position);
                AlertDialog.Builder builder = new
                        AlertDialog.Builder(MainActivity.this);
                builder.setMessage("Selected item: "
                        + item).setTitle("ListView");
                builder.create().show();
                Log.d("ListView", "Selected item : " + item);
            }
        });
    }

    @Override
    public boolean onCreateOptionsMenu(Menu menu) {
        getMenuInflater().inflate(R.menu.main, menu);
        return true;
    }
}
```

Figure 29.2 shows the application.

Figure 29.2: A simple ListView

Extending ListActivity and Writing A Custom Adapter

If your activity will only have one component that is a **ListView**, you should consider extending the **ListActivity** class instead of **Activity**. With **ListActivity**, you do not need a layout file for your activity. **ListActivity** already contains a **ListView** and you do not need to attach a listener to it. On top of that, the **ListActivity** class already defines a **setListAdapter** method, so you just need to call it in your **onCreate** method, passing a **ListAdapter**. In addition, instead of creating an **AdapterView.OnItemClickListener**, you just need to override the **ListActivity**'s **onListItemClick** method, which will be called when an item on the **ListView** gets selected.

The ListViewDemo2 application shows you how to use **ListActivity**. The application also demonstrates how to create a custom **ListAdapter** by extending the **ArrayAdapter** class and creating a layout file for the custom **ListAdapter**.

The layout file for the custom **ListAdapter** in ListViewDemo2 is presented in Listing 29.5. It is named **pretty_adapter.xml** file and is located under **res/layout**.

Listing 29.5: The pretty_adapter.xml file

```
<?xml version="1.0" encoding="utf-8"?>
<LinearLayout
       xmlns:android="http://schemas.android.com/apk/res/android">
    <ImageView
              android:id="@+id/icon"
            android:layout_width="36dp"
            android:layout_height="fill_parent"/>
    <TextView
              android:id="@+id/label"
        android:layout_width="fill_parent"
        android:layout_height="fill_parent"
        android:gravity="center_vertical"
        android:padding="12dp"
        android:textSize="18sp"
        android:textColor="@android:color/holo_blue_bright"/>
</LinearLayout>
```

Listing 29.6 shows the custom adapter class, called **PrettyAdapter**.

Listing 29.6: The PrettyAdapter class

```
package com.example.listviewdemo2;
import android.content.Context;
import android.graphics.drawable.Drawable;
import android.view.LayoutInflater;
import android.view.View;
import android.view.ViewGroup;
import android.widget.ArrayAdapter;
import android.widget.ImageView;
import android.widget.TextView;

public class PrettyAdapter extends ArrayAdapter<String> {
    private LayoutInflater inflater;
    private String[] items;
    private Drawable icon;
```

```
    private int viewResourceId;

    public PrettyAdapter(Context context,
            int viewResourceId, String[] items, Drawable icon) {
        super(context, viewResourceId, items);
        inflater = (LayoutInflater) context
                .getSystemService(Context.LAYOUT_INFLATER_SERVICE);
        this.items = items;
        this.icon = icon;
        this.viewResourceId = viewResourceId;
    }

    @Override
    public int getCount() {
        return items.length;
    }

    @Override
    public String getItem(int position) {
        return items[position];
    }

    @Override
    public long getItemId(int position) {
        return 0;
    }

    @Override
    public View getView(int position, View convertView,
            ViewGroup parent) {
        convertView = inflater.inflate(viewResourceId, null);

        ImageView imageView = (ImageView)
                convertView.findViewById(R.id.icon);
        imageView.setImageDrawable(icon);

        TextView textView = (TextView)
                convertView.findViewById(R.id.label);
        textView.setText(items[position]);
        return convertView;
    }
}
```

The custom adapter must override several methods, notably the **getView** method, which must return a **View** that will be used for each item on the **ListView**. In this example, the view contains an **ImageView** and a **TextView**. The text for the **TextView** is taken from the array passed to the **PrettyAdapter** instance.

The last piece of the application is the **MainActivity** class in Listing 29.7. It extends **ListActivity** and is the only activity in the application.

Listing 29.7: The MainActivity class for ListViewDemo2

```
package com.example.listviewdemo2;
import android.app.ListActivity;
import android.content.Context;
import android.content.res.Resources;
```

```
import android.graphics.drawable.Drawable;
import android.os.Bundle;
import android.util.Log;
import android.view.View;
import android.widget.ListView;

public class MainActivity extends ListActivity {

    @Override
    public void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        // Since we're extending ListActivity, we do
        // not need to call setContentView();

        Context context = getApplicationContext();
        Resources resources = context.getResources();

        String[] items = resources.getStringArray(
                R.array.players);
        Drawable drawable = resources.getDrawable(
                R.drawable.pretty);

        setListAdapter(new PrettyAdapter(context,
                R.layout.pretty_adapter, items, drawable));
    }

    @Override
    public void onListItemClick(ListView listView,
            View view, int position, long id) {
        Log.d("listView2", "listView:" + listView +
                ", view:" + view.getClass() +
                ", position:" + position );
    }
}
```

If you run the application, you will see an activity like that in Figure 29.3.

Styling the Selected Item

It is often desirable that the user be able to see clearly the currently selected item in a **ListView**. To make the selected item look differently than the rest of the items, set the **ListView**'s choice mode to **CHOICE_MODE_SINGLE**, like so.

```
listView.setChoiceMode(ListView.CHOICE_MODE_SINGLE);
```

Then, when constructing the underlying **ListAdapter**, use a layout with an appropriate style. The easiest is to pass the **simple_list_item_activated_1** field. For example, the following **ArrayAdapter** if used in a **ListView** will cause the selected item to have a blue background.

```
ArrayAdapter<String> adapter = new ArrayAdapter<String>(
        context, android.R.layout.simple_list_item_activated_1,
        array);
```

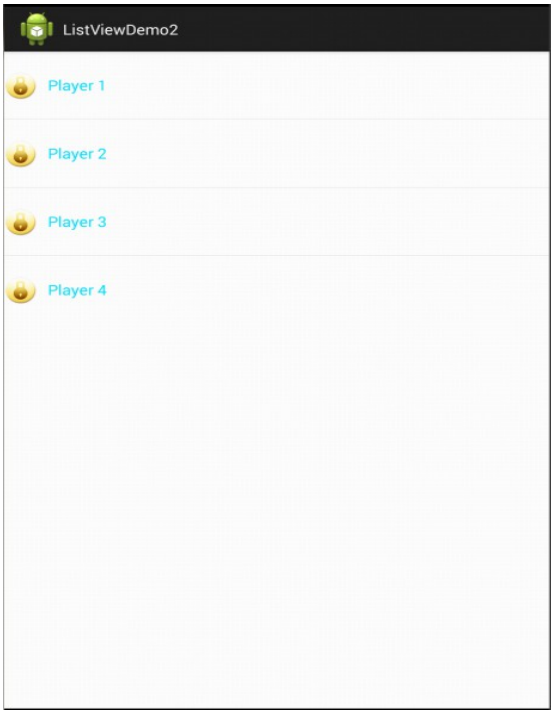

Figure 29.3: Using custom adapter in ListActivity

If the default style does not appeal to you, you can create your own style by creating a selector. A selector is a drawable that can be used as a background drawable in a **TextView**. Here is an example of a Selector file that must be saved in the **res/drawable** directory.

```
<selector
    xmlns:android="http://schemas.android.com/apk/res/android">
    <item android:state_activated="true"
        android:drawable="@drawable/activated"/>
</selector>
```

The selector must have an item whose **state_activated** attribute is set to **true** and whose **drawable** attribute refers to another drawable.

The ListViewDemo3 application contains an activity that employs two **ListView**s that are places side by side. The first **ListView** on the left is given the default style whereas the second **ListView** is decorated using a custom style. Figure 29.4 shows the application.

Now, let's look at the code.

Let's start with the activity layout file in Listing 29.8.

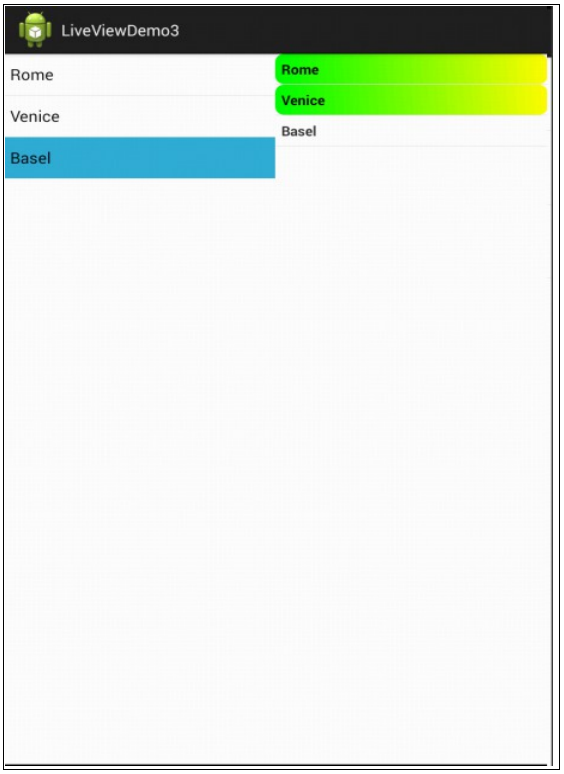

Figure 29.4: Styling the selected item of a ListView

Listing 29.8: The layout file for the main activity (activity_main.xml)

```
<LinearLayout
        xmlns:android="http://schemas.android.com/apk/res/android"
    android:layout_width="match_parent"
    android:layout_height="match_parent"
    android:orientation="horizontal">

    <ListView
        android:id="@+id/listView1"
        android:layout_weight="1"
        android:layout_width="0dp"
        android:layout_height="match_parent"/>
    <ListView
        android:id="@+id/listView2"
        android:layout_weight="1"
        android:layout_width="0dp"
        android:layout_height="match_parent"/>
</LinearLayout>
```

The layout uses a horizontal **LinearLayout** that contains two **ListView**s, named **listView1** and **listView2**, respectively. Both **ListView**s receive the same value for their **layout_weight** attribute, so they will have the same width when rendered.

The **MainActivity** class in Listing 29.9 represents the activity for the application. It's **onCreate** method loads both **ListView**s and pass them a **ListAdapter**. In additon, the first **ListView**'s choice mode is set to **CHOICE_MODE_SINGLE**, making a single item selectable at a time. The

second **ListView**'s choice mode is set to **CHOICE_MODE_MULTIPLE**, which makes multiple items selectable at a time.

Listing 29.9: The MainActivity class

```
package com.example.liveviewdemo3;
import android.app.Activity;
import android.os.Bundle;
import android.widget.ArrayAdapter;
import android.widget.ListView;
public class MainActivity extends Activity {

    @Override
    protected void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.activity_main);
        String[] cities = {"Rome", "Venice", "Basel"};
        ArrayAdapter<String> adapter1 = new
                ArrayAdapter<String>(this,
                android.R.layout.simple_list_item_activated_1,
                cities);
        ListView listView1 = (ListView)
                findViewById(R.id.listView1);
        listView1.setAdapter(adapter1);
        listView1.setChoiceMode(ListView.CHOICE_MODE_SINGLE);

        ArrayAdapter<String> adapter2 = new
                ArrayAdapter<String>(this,
                R.layout.list_item, cities);
        ListView listView2 = (ListView)
                findViewById(R.id.listView2);
        listView2.setAdapter(adapter2);
        listView2.setChoiceMode(ListView.CHOICE_MODE_MULTIPLE);
    }
}
```

The first **ListView**'s **ListAdapter** is given the default layout (**simple_list_item_activated_1**). The second **ListView**'s **ListAdapter**, on the other hand, is set to use a layout that is pointed by **R.layout.list_item**. This refers to the **res/layout/list_item.xml** file shown in Listing 29.10.

Listing 29.10: The list_item.xml file

```
<?xml version="1.0" encoding="utf-8"?>
<TextView
       xmlns:android="http://schemas.android.com/apk/res/android"
    android:id="@+id/list_item"
    android:layout_width="fill_parent"
    android:layout_height="fill_parent"
    android:padding="7dip"
    android:textSize="16sp"
    android:textStyle="bold"
    android:background="@drawable/list_selector"
/>
```

A layout file for a **ListView** must contain a **TextView**, as the **list_item.xml** file does. Note that its **background** attribute is given the value **drawable/list_selector**, which references the **list_selector.xml** file in Listing 29.11. This is a selector file that will be used to style the selected item on **listView2**. The **selector** element

contains an item whose **state_activated** attribute is set to **true**, which means it will be used to style the selected item. Its **drawable** attribute is set to **drawable/activated**, referring to the **drawable/activated.xml** file in Listing 29.12.

Listing 29.11: The drawable/list_selector.xml file

```
<?xml version="1.0" encoding="utf-8"?>
<selector
    xmlns:android="http://schemas.android.com/apk/res/android">
    <item android:state_activated="true"
        android:drawable="@drawable/activated"/>
</selector>
```

Listing 29.12: The drawable/activated.xml file

```
<shape xmlns:android="http://schemas.android.com/apk/res/android"
    android:shape="rectangle">
    <corners android:radius="8dp"/>
    <gradient
        android:startColor="#FFFF0000"
        android:endColor="#FFFF00"
        android:angle="45"/>
</shape>
```

The drawable in Listing 29.12 is based on an XML file that contains a shape with a given gradient color.

Running the application will give you an activity like that in Figure 29.4.

Summary

A **ListView** is a view that contains a list of scrollable items and gets its data source and layout from a **ListAdapter**, which in turn can be created from an **ArrayAdapter**. In this chapter you learned how to use the **ListView**. You also learned how to use the **ListActivity** and style the selected item on a **ListView**.

Chapter 30
GridView

A **GridView** is a view that can display a list of scrollable items in a grid. It is like a **ListVIew** except that it display items in multiple columns, unlike a **ListView** where items are displayed in one single column. Like a **ListView**, a **GridView** too takes its data source and layout from a **ListAdapter**.

This chapter shows how you can use the **GridView** widget and presents a sample application. You should have read Chapter 29, "ListView" before attempting to read this chapter.

Overview

The **android.widget.GridView** class is the template for creating a **GridView**. Both the **GridView** and **ListView** classes are direct descendants of **android.view.AbsListView**. Like a **ListView**, a **GridView** gets its data source from a **ListAdapter**. Please refer to Chapter 29, "ListView" for more information on the **ListAdapter**.

You can use a **GridView** just like you would other views: by declaring a node in a layout file. In the case of a **GridView**, you would use this **GridView** element:

```
<GridView
    android:id="@+id/gridView1"
    android:layout_width="fill_parent"
    android:layout_height="fill_parent"
    android:columnWidth="120dp"
    android:numColumns="auto_fit"
    android:verticalSpacing="10dp"
    android:horizontalSpacing="10dp"
    android:stretchMode="columnWidth"
/>
```

You can then find the **GridView** in your activity class using **findViewById** and pass a **ListAdapter** to it.

```
GridView gridView = (GridView) findViewById(R.id.gridView1);
gridView.setAdapter(listAdapter);
```

Optionally, you can pass an **AdapterView.OnItemClickListener** to a **GridView**'s **setOnItemClickListener** method to respond to item selection:

```
gridview.setOnItemClickListener(new
       AdapterView.OnItemClickListener() {
    public void onItemClick(AdapterView<?> parent, View v, int
            position, long id) {

        // do something here
```

```
    }
});
```

Example

The GridViewDemo1 application shows you how to use the **GridView**. The application only has an activity, which uses a **GridView** to fill its entire display area. The **GridView** in turn uses a custom **ListAdapter** for its items and layout.

Listing 30.1 shows the application manifest.

Listing 30.1: The AndroidManifest.xml file

```
<?xml version="1.0" encoding="utf-8"?>
<manifest
       xmlns:android="http://schemas.android.com/apk/res/android"
    package="com.example.gridviewdemo1"
    android:versionCode="1"
    android:versionName="1.0" >

    <uses-sdk
        android:minSdkVersion="18"
        android:targetSdkVersion="18" />

    <application
        android:allowBackup="true"
        android:icon="@drawable/ic_launcher"
        android:label="@string/app_name"
        android:theme="@style/AppTheme" >
        <activity
            android:name="com.example.gridviewdemo1.MainActivity"
            android:label="@string/app_name">
            <intent-filter>
                <action android:name="android.intent.action.MAIN"/>
                <category
android:name="android.intent.category.LAUNCHER"/>
            </intent-filter>
        </activity>
    </application>
</manifest>
```

The custom **ListAdapter** that feeds the **GridView** is an instance of **GridViewAdapter**, which is presented in Listing 30.2. **GridViewAdapter** extends **android.widget.BaseAdapter**, which in turn implements the **android.widget.ListAdapter** interface. Therefore, a **GridViewAdapter** is a **ListAdapter** and can be passed to a **GridView**'s **setAdapter** method.

Listing 30.2: The GridViewAdapter class

```
package com.example.gridviewdemo1;
import android.content.Context;
import android.view.View;
import android.view.ViewGroup;
import android.widget.BaseAdapter;
import android.widget.GridView;
import android.widget.ImageView;
```

```
public class GridViewAdapter extends BaseAdapter {
    private Context context;

    public GridViewAdapter(Context context) {
        this.context = context;
    }
    private int[] icons = {
            android.R.drawable.btn_star_big_off,
            android.R.drawable.btn_star_big_on,
            android.R.drawable.alert_light_frame,
            android.R.drawable.alert_dark_frame,
            android.R.drawable.arrow_down_float,
            android.R.drawable.gallery_thumb,
            android.R.drawable.ic_dialog_map,
            android.R.drawable.ic_popup_disk_full,
            android.R.drawable.star_big_on,
            android.R.drawable.star_big_off,
            android.R.drawable.star_big_on
    };

    @Override
    public int getCount() {
        return icons.length;
    }

    @Override
    public Object getItem(int position) {
        return null;
    }

    @Override
    public long getItemId(int position) {
        return 0;
    }

    @Override
    public View getView(int position, View convertView, ViewGroup
       parent) {
        ImageView imageView;
        if (convertView == null) {
            imageView = new ImageView(context);
            imageView.setLayoutParams(new
       GridView.LayoutParams(100, 100));

       imageView.setScaleType(ImageView.ScaleType.CENTER_CROP);
            imageView.setPadding(10, 10, 10, 10);
        } else {
            imageView = (ImageView) convertView;
        }
        imageView.setImageResource(icons[position]);
        return imageView;
    }
}
```

GridViewAdapter provides an implementation of the **getView** method that

returns an **ImageView** displaying one of Android's default drawables:

```
    private int[] icons = {
            android.R.drawable.btn_star_big_off,
            android.R.drawable.btn_star_big_on,
            android.R.drawable.alert_light_frame,
            android.R.drawable.alert_dark_frame,
            android.R.drawable.arrow_down_float,
            android.R.drawable.gallery_thumb,
            android.R.drawable.ic_dialog_map,
            android.R.drawable.ic_popup_disk_full,
            android.R.drawable.star_big_on,
            android.R.drawable.star_big_off,
            android.R.drawable.star_big_on
    };
```

Now that you know what **GridViewAdapter** does, you can now focus on the activity. The layout file for the activity is printed in Listing 30.3. It only consists of one component, a **GridView**.

Listing 30.3: The activity_main.xml file

```
<?xml version="1.0" encoding="utf-8"?>
<GridView
       xmlns:android="http://schemas.android.com/apk/res/android"
    android:id="@+id/gridview"
    android:layout_width="fill_parent"
    android:layout_height="fill_parent"
    android:columnWidth="90dp"
    android:numColumns="auto_fit"
    android:verticalSpacing="10dp"
    android:horizontalSpacing="10dp"
    android:stretchMode="columnWidth"
    android:gravity="center"
/>
```

Listing 30.4 shows the **MainActivity** class.

Listing 30.4: The MainActivity class

```
package com.example.gridviewdemo1;

import android.app.Activity;
import android.os.Bundle;
import android.view.Menu;
import android.view.View;
import android.widget.AdapterView;
import android.widget.AdapterView.OnItemClickListener;
import android.widget.GridView;
import android.widget.Toast;

public class MainActivity extends Activity {

    @Override
    protected void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.activity_main);

        GridView gridview = (GridView) findViewById(R.id.gridview);
```

```
        gridview.setAdapter(new GridViewAdapter(this));

        gridview.setOnItemClickListener(new OnItemClickListener() {
            public void onItemClick(AdapterView<?> parent,
                    View view, int position, long id) {
                Toast.makeText(MainActivity.this, "" + position,
                        Toast.LENGTH_SHORT).show();
            }
        });
    }

    @Override
    public boolean onCreateOptionsMenu(Menu menu) {
        getMenuInflater().inflate(R.menu.main, menu);
        return true;
    }
}
```

MainActivity is a simple class, with the bulk of its brain resides in its **onCreate** method. Here it loads the **GridView** from the layout and passes an instance of **GridViewAdapter** to the **GridView**'s **setAdapter** method. It also creates an **OnItemClickListener** for the **GridView** so that every time an item on the **GridView** is selected, the **onItemClick** method in the listener gets called. In this case, **onItemClick** simply creates a **Toast** that shows the position of the selected item.

Running GridViewDemo1 gives you an activity that looks like the one in Figure 30.1.

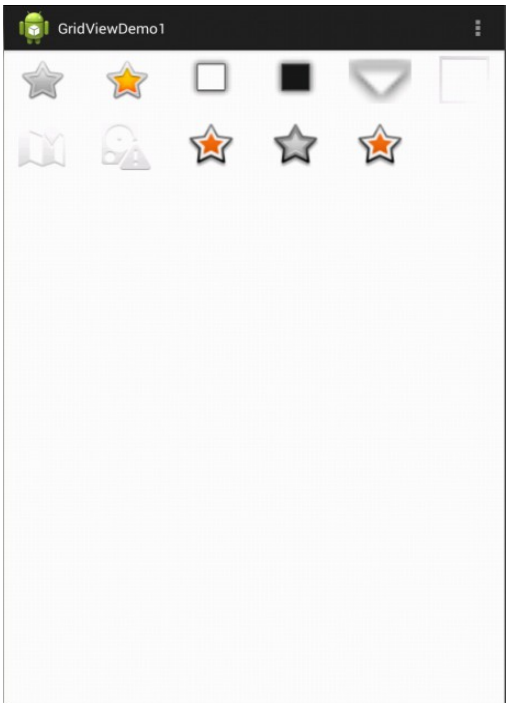

Figure 30.1: Using a GridView

Summary

A **GridView** is a view that contains a list of scrollable items displayed in a grid. Like a **ListView**, a **GridView** gets its data and layout from a **ListAdapter**. In addition, a **GridView** can also be given an **AdapterView.OnItemClickListener** to handle item selection.

Chapter 31
Styles and Themes

Your application's look and feel are governed by the styles and themes it is using. This chapter discusses these two important topics and shows you how to use them.

Overview

A view declaration in a layout file can have attributes, many of which are style-related, including **textColor**, **textSize**, **background**, and **textAppearance**.

Style-related attributes for an application can be lumped in a group and the group can be given a name and moved to a **styles.xml** file. A **styles.xml** file that is saved in the **res/values** directory will be recognized by the application as a styles file and the styles in the file can be used to style the views in the application. To apply a style to a view, use the **style** attribute. The advantage of creating a style is to make the style reusable and shareable. Styles support inheritance so you can extend a style to create a new style. Here is an example of style in a **styles.xml** file.

```
<style name="Style1">
    <item name="android:layout_width">wrap_content</item>
    <item name="android:layout_height">wrap_content</item>
    <item name="android:textColor">#FFFFFF</item>
    <item name="android:textStyle">bold</item>
    <item name="android:textSize">25sp</item>
</style>
```

To apply the style to a view, assign the style name the **style** attribute.

```
<TextView
    android:id="@+id/textView1"
    style="@style/Style1"
    android:text="Style 1"/>
```

Note that the **style** attribute, unlike other attributes, does not use the **android** prefix.

The **TextView** element declaration above is equivalent to the following.

```
<TextView
    android:id="@+id/textView1"
    android:layout_width="wrap_content"
    android:layout_height="wrap_content"
    android:textColor="#FFFFFF"
    android:textStyle="bold"
    android:textSize="25sp"
    android:text="Style 1"/>
```

The following is a style that extends another style.

```
<style name="Style2" parent="Style1">
    <item name="android:background">
        @android:color/holo_green_light
    </item>
</style>
```

The system provides a vast collection of styles you can use in your applications. You can find a reference of all available styles in the **android.R.style** class. To use the styles listed in this class in your layout file, replace all underscores in the style name with a period. For example, you can apply the **Holo_ButtonBar** style with **@android:style/Holo.ButtonBar**.

```
<Button
    style="@android:style/Holo.ButtonBar"
    android:text="@string/hello_world"/>
```

Prefixing the value of the **style** attribute with **android** indicates that you are using a system style.

A copy of the system **styles.xml** file can be viewed here:

```
https://android.googlesource.com/platform/frameworks/base/+/refs/
heads/master/core/res/res/values/styles.xml
```

Using Styles

The StyleDemo1 application shows how you can create your own styles.

Listing 31.1 shows the application's **styles.xml** file in the **res/values** directory.

Listing 31.1: The styles.xml file

```
<resources
    xmlns:android="http://schemas.android.com/apk/res/android">
    <!-- Base application theme, dependent on API level. This theme
         is replaced by AppBaseTheme from res/values-vXX/styles.xml
         on newer devices.
    -->
    <style name="AppBaseTheme" parent="android:Theme.Light">
        <!-- Theme customizations available in newer API levels can
             go in res/values-vXX/styles.xml, while customizations
             related to backward-compatibility can go here.
        -->
    </style>

    <!-- Application theme. -->
    <style name="AppTheme" parent="AppBaseTheme">
        <!-- All customizations that are NOT specific to a
             particular API-level can go here. -->
    </style>

    <style name="WhiteOnRed">
        <item name="android:layout_width">wrap_content</item>
        <item name="android:layout_height">wrap_content</item>
        <item name="android:textColor">#FFFFFF</item>
        <item name="android:background">
            @android:color/holo_red_light
```

```
        </item>
        <item name="android:typeface">serif</item>
        <item name="android:textStyle">bold</item>
        <item name="android:textSize">25sp</item>
        <item name="android:padding">30dp</item>
    </style>
    <style name="WhiteOnRed.Italic">
        <item name="android:textStyle">bold|italic</item>
    </style>
    <style name="WhiteOnGreen" parent="WhiteOnRed">
        <item name="android:background">
            @android:color/holo_green_light
        </item>
    </style>
</resources>
```

There are five styles defined in the **styles.xml** file in Listing 31.1. The first two are added by ADT Eclipse when the application was created. They will be explained in the "Themes" section later in this chapter.

The other three styles are used by the main activity of the application in the layout file for that activity. The layout file is shown in Listing 31.2.

Listing 31.2: the activity_main.xml layout file

```
<RelativeLayout
    xmlns:android="http://schemas.android.com/apk/res/android"
    xmlns:tools="http://schemas.android.com/tools"
    android:layout_width="match_parent"
    android:layout_height="match_parent"
    android:paddingBottom="@dimen/activity_vertical_margin"
    android:paddingLeft="@dimen/activity_horizontal_margin"
    android:paddingRight="@dimen/activity_horizontal_margin"
    android:paddingTop="@dimen/activity_vertical_margin"
    tools:context=".MainActivity" >

    <TextView
        android:id="@+id/textView1"
        style="@style/WhiteOnRed"
        android:text="Style WhiteOnRed" />
    <TextView
        android:id="@+id/textView2"
        android:layout_below="@id/textView1"
        android:layout_marginLeft="20sp"
        android:layout_marginTop="10sp"
        style="@style/WhiteOnRed.Italic"
        android:text="Style WhiteOnRed.Italic" />
    <TextView
        android:id="@+id/textView3"
        android:layout_below="@id/textView2"
        android:layout_toEndOf="@id/textView2"
        style="@style/WhiteOnGreen"
        android:text="Style WhiteOnGreen" />

    <TextView
        android:id="@+id/textView4"
        android:text="Style TextAppearance.Holo.Medium.Inverse"
```

```
        android:layout_below="@id/textView2"
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        style="@android:style/TextAppearance.Holo.Medium"/>
</RelativeLayout>
```

Listing 31.3 shows the activity that uses the layout file in Listing 31.2.

Listing 31.3: The MainActivity class

```
package com.example.styledemo1;
import android.os.Bundle;
import android.app.Activity;
import android.view.Menu;

public class MainActivity extends Activity {

    @Override
    protected void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.activity_main);
    }

    @Override
    public boolean onCreateOptionsMenu(Menu menu) {
        // Inflate the menu; this adds items to the action bar if
       it is present.
        getMenuInflater().inflate(R.menu.main, menu);
        return true;
    }

}
```

Figure 31.1 shows the StyleDemo1 application.

Using Themes

A theme is a style that is applied to an activity or all the activities in an application. To apply a theme to an activity, use the **android:theme** attribute in the **activity** element in the manifest file. For example, the following **activity** element uses the **Theme.Holo.Light** theme.

```
<activity
    android:name="..."
    android:theme="@android:style/Theme.Holo.Light">
</activity>
```

To apply a theme to the whole application, add the **android:theme** attribute in the **application** element in the Android manifest file. For instance,

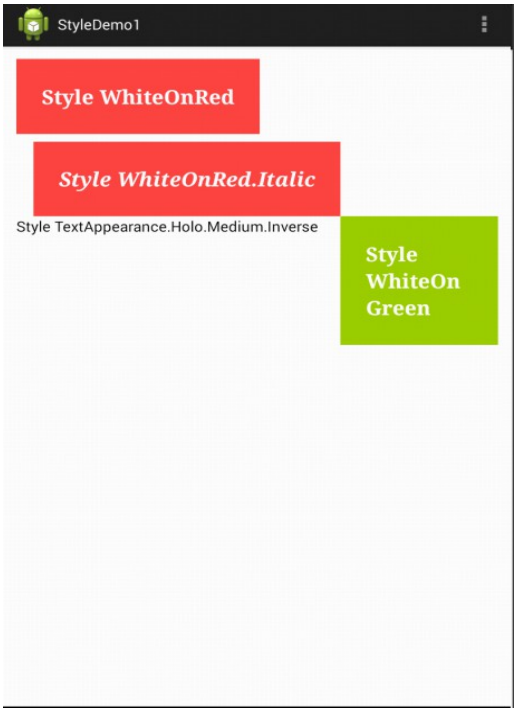

Figure 31.1: Using styles

```
<application
    android:icon="@drawable/ic_launcher"
    android:label="@string/app_name"
    android:theme="@android:style/Theme.Black.NoTitleBar">

...

</application>
```

Android provides a collection of themes you can use in your application. A copy of the theme file can be found here:

```
https://android.googlesource.com/platform/frameworks/base/+/refs/
heads/master/core/res/res/values/themes.xml
```

Figure 31.2 to 31.5 show some of the themes that comes with Android.

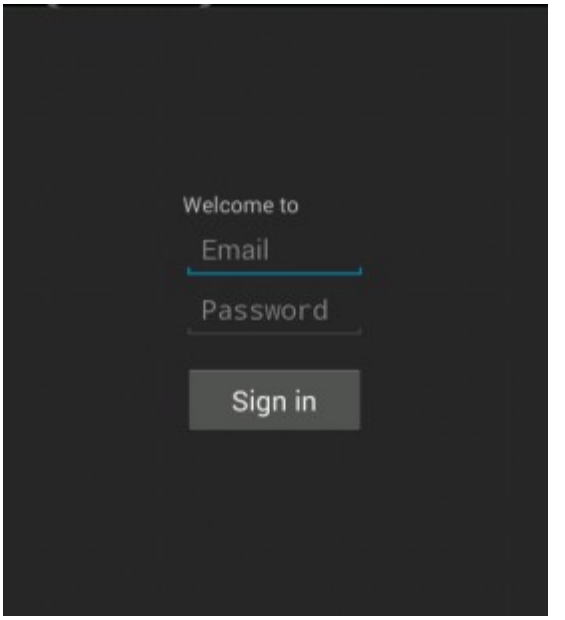

Figure 31.2: Theme.Holo.Dialog.NoActionBar

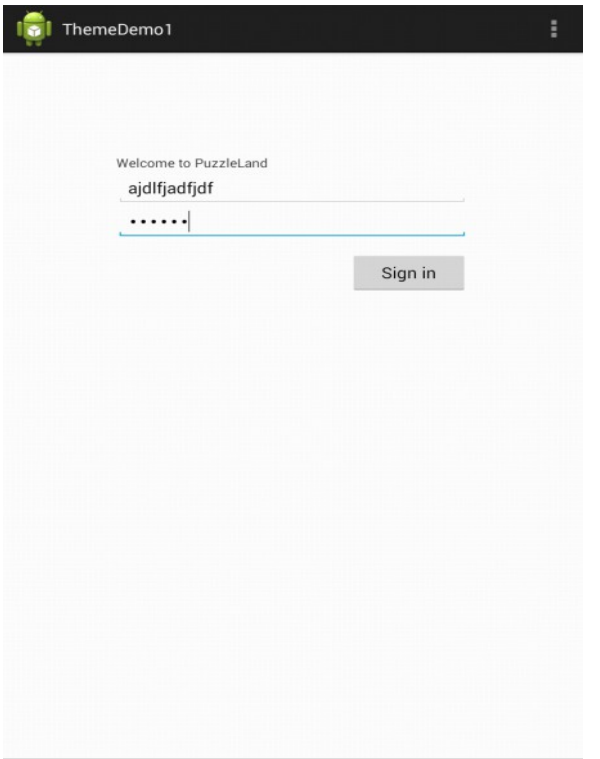

Figure 31.3: Theme.Light

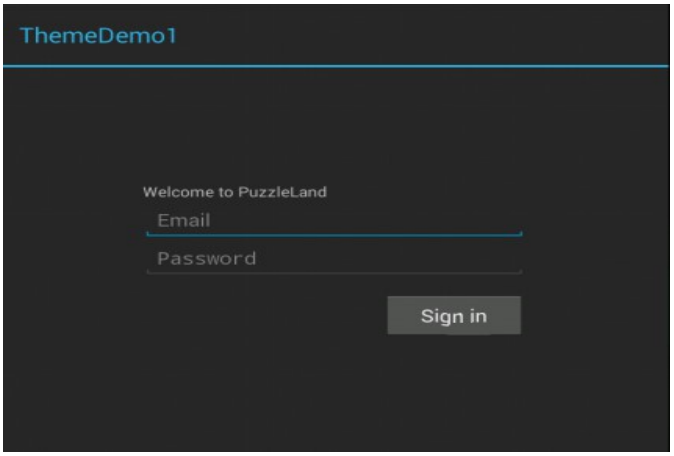

Figure 31.4: Theme.DeviceDefault.Dialog

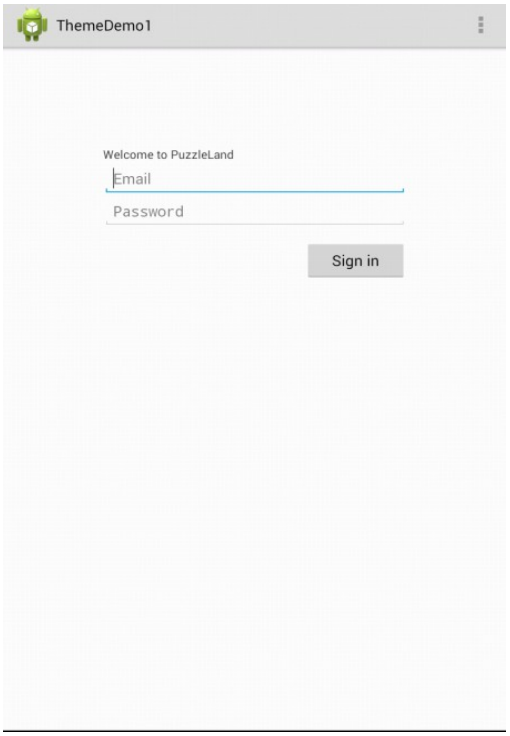

Figure 31.5: Theme.Holo.Light

Summary

A style is a collection of attributes that directly affect the look of a view. You can apply a style to a view by using the **style** attribute in the view's declaration in a layout file. A theme is a style that is applied to an activity or the entire application.

Chapter 32
Bitmap Processing

With the Android Bitmap API you can manipulate images in JPG, PNG, or GIF format, such as by changing the color or the opacity of each pixel in the image. In addition, you can use the API to down-sample a large image to save memory. As such, knowing how to use this API is useful, even when you are not writing a photo editor or an image processing application.

This chapter explains how to work with bitmaps and provides an example.

Overview

A bitmap is an image file format that can store digital images independently of the display device. Bitmap simply means a map of bits. Today the term also includes other formats that support lossy and lossless compression, such as JPEG, GIF, and PNG. GIF and PNG support transparency and lossless compression, whereas JPEG support lossy compression and does not support transparency. Another way of representing digital images is through mathematical expressions. Such images are known as vector graphics.

The Android framework provides an API for processing bitmap images. This API takes the form of classes, interfaces, and enums in the **android.graphics** package and its subpackages. A bitmap image is represented by the **Bitmap** class. A **Bitmap** can be displayed on an activity using the **ImageView** widget.

The easiest way to load a bitmap is by using the **BitmapFactory** class. This class provides static methods for constructing a **Bitmap** object from a file, a byte array, an Android resource, or an **InputStream**. Here are some of the methods.

```
public static Bitmap decodeByteArray(byte[] data, int offset,
        int length)

public static Bitmap decodeFile(java.lang.String pathName)

public static Bitmap decodeResource(
        android.content.res.Resources res, int id)

public static Bitmap decodeStream (java.io.InputStream is)
```

For example, to construct a **Bitmap** from an Android resource in an activity class, you would use this code.

```
Bitmap bmp = BitmapFactory.decodeResource(getResources(),
        R.drawable.image1);
```

Here, **getResources** is a method in the **android.app.Activity** class that returns the application's resources. The identifier (**R.drawable.image1**) allows Android to pick the correct image from the resources.

The **BitmapFactory** class also offers static methods that take options as a

BitmapFactory.Options object:

```
public static Bitmap decodeByteArray (byte[] data, int offset,
        int length, BitmapFactory.Options opts)

public static Bitmap decodeFile (java.lang.String pathName,
        BitmapFactory.Options opts)

public static Bitmap decodeResource (android.content.res.Resources
        res, int id, BitmapFactory.Options opts)

public static Bitmap decodeStream (java.io.InputStream is,
        Rect outPadding, BitmapFactory.Options opts)
```

There are two purposes of using **BitmapFactory.Options**. The first purpose is to configure the resulting bitmap as the class allows you to down-sample the bitmap, set the bitmap to be mutable and configure its density. The second purpose is to use **BitmapFactory.Options** to read the properties of a bitmap without actually loading the image. For example, you may pass a **BitmapFactory.Options** to one of the decode methods in **BitmapFactory** and read the size of the image. If the size is considered too large, then you can down-sample it, saving precious memory. Down-sampling makes sense for large bitmaps when it does not reduce render quality. For instance, a 20,000 x 10,000 bitmap can be down-sampled to 2,000 x 1,000 without degradation assuming the device screen resolution does not exceed 2,000 x 1,000. In the process, it saves a lot of memory.

To decode a **Bitmap** without actually loading the bitmap, set the **inJustDecodeBounds** field of the **BitmapFactory.Options** object to **true**.

```
BitmapFactory.Options opts = new BitmapFactory.Options()
opts.inJustDecodeBounds = true;
```

If you pass the options to one of the **decode** methods in **BitmapFactory**, the method will return null and simply populate the **BitmapFactory.Options** object that you passed. From this object, you can retrieve the bitmap size and other properties:

```
int imageHeight = options.outHeight;
int imageWidth = options.outWidth;
String imageType = options.outMimeType;
```

The **inSampleSize** field of **BitmapFactor.Options** tells the system how to sample a bitmap. A value greater than 1 indicates that the image should be down-sampled. For example, setting the **inSampleSize** field to 4 returns an image whose size is a quarter that of the original image.

Regarding this field, the Android documentation says that the decoder uses a final value based on powers of 2, which means you should only assign a power of 2, such as 2, 4, 8, and so on. However, my own test shows that this only applies to images in JPG format and does not apply to PNGs. For instance, if the width of a PNG image is 1200, assigning 3 to this field returns an image with a width of 400 pixels, which means the **inSampleSize** value does not have to be a power of two.

Finally, once you get a **Bitmap** from a **BitmapFactory**, you can pass the **Bitmap** to an **ImageView** to be displayed:

```
ImageView imageView1 = (ImageView) findViewById(...);
imageView1.setImageBitmap(bitmap);
```

Example

The BitmapDemo application showcases an activity that shows an **ImageView** that displays a **Bitmap** that can be down-sampled. There are four bitmaps (two JPEGs, one GIF, and one PNG) included and the application provides a button to change bitmaps. The main (and only) activity of the application is shown in Figure 32.1.

Listing 32.1 shows the **AndroidManifest.xml** file for the application.

Listing 32.1: The AndroidManifest.xml file

```
<?xml version="1.0" encoding="utf-8"?>
<manifest
       xmlns:android="http://schemas.android.com/apk/res/android"
    package="com.example.bitmapdemo"
    android:versionCode="1"
    android:versionName="1.0" >
    <uses-sdk
        android:minSdkVersion="18"
        android:targetSdkVersion="18" />
    <application
        android:allowBackup="true"
        android:icon="@drawable/ic_launcher"
        android:label="@string/app_name"
        android:theme="@style/AppTheme" >
        <activity
            android:name="com.example.bitmapdemo.MainActivity"
            android:label="@string/app_name" >
            <intent-filter>
                <action android:name="android.intent.action.MAIN"/>
                <category
android:name="android.intent.category.LAUNCHER"/>
            </intent-filter>
        </activity>
    </application>
</manifest>
```

There is only one activity in this application. The layout file for the activity is given in Listing 32.2.

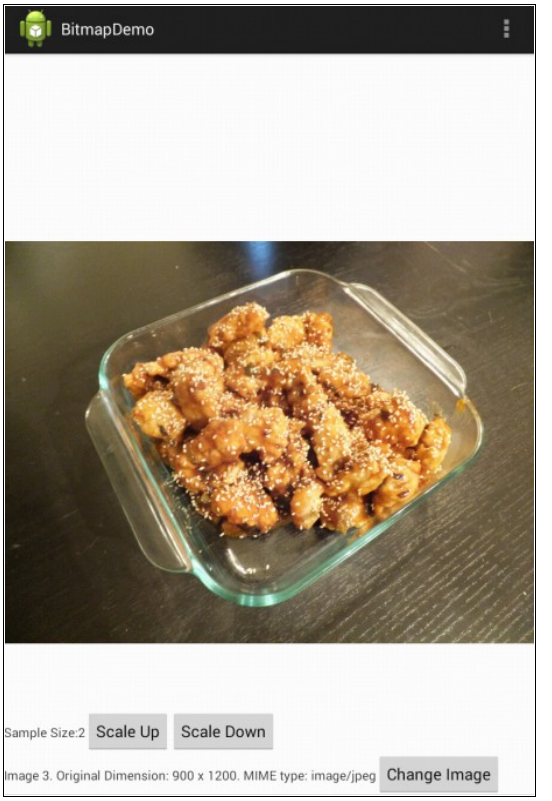

Figure 32.1: The BitmapDemo application

Listing 32.2: The activity_main.xml file

```
<LinearLayout
       xmlns:android="http://schemas.android.com/apk/res/android"
    xmlns:tools="http://schemas.android.com/tools"
    android:layout_width="match_parent"
    android:layout_height="match_parent"
    android:orientation="vertical"
    android:gravity="bottom"
    tools:context=".MainActivity" >

    <ImageView
        android:id="@+id/image_view1"
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:contentDescription="@string/text_content_desc"/>

    <LinearLayout
        android:layout_width="match_parent"
        android:layout_height="wrap_content"
        android:orientation="horizontal" >

        <TextView
            android:layout_width="wrap_content"
            android:layout_height="wrap_content"
            android:text="@string/text_sample_size"/>
```

```
        <TextView
            android:id="@+id/sample_size"
            android:layout_width="wrap_content"
            android:layout_height="wrap_content"
          />
        <Button
            android:onClick="scaleUp"
            android:layout_width="wrap_content"
            android:layout_height="wrap_content"
            android:text="@string/action_scale_up" />

        <Button
            android:onClick="scaleDown"
            android:layout_width="wrap_content"
            android:layout_height="wrap_content"
            android:text="@string/action_scale_down" />

    </LinearLayout>
    <LinearLayout
        android:layout_width="match_parent"
        android:layout_height="wrap_content"
        android:orientation="horizontal" >

        <TextView
            android:id="@+id/image_info"
            android:layout_width="wrap_content"
            android:layout_height="wrap_content"/>
        <Button
            android:onClick="changeImage"
            android:layout_width="wrap_content"
            android:layout_height="wrap_content"
            android:text="@string/action_change_image" />
    </LinearLayout>
</LinearLayout>
```

The layout contains a **LinearLayout** that in turn contains an **ImageView** and two **LinearLayout**s. The first inner layout contains two **TextView**s and buttons for scaling up and down the bitmap. The second inner layout contains a **TextView** to display the bitmap metadata and a button to change the bitmap.

The **MainActivity** class is presented in Listing 32.3.

Listing 32.3: The MainActivity class

```
package com.example.bitmapdemo;
import android.app.Activity;
import android.graphics.Bitmap;
import android.graphics.BitmapFactory;
import android.os.Bundle;
import android.view.Menu;
import android.view.View;
import android.widget.ImageView;
import android.widget.TextView;

public class MainActivity extends Activity {
    int sampleSize = 2;
    int imageId = 1;
```

```
@Override
protected void onCreate(Bundle savedInstanceState) {
    super.onCreate(savedInstanceState);
    setContentView(R.layout.activity_main);
    refreshImage();
}

@Override
public boolean onCreateOptionsMenu(Menu menu) {
    getMenuInflater().inflate(R.menu.main, menu);
    return true;
}

public void scaleDown(View view) {
    if (sampleSize < 8) {
        sampleSize++;
        refreshImage();
    }
}

public void scaleUp(View view) {
    if (sampleSize > 2) {
        sampleSize--;
        refreshImage();
    }
}
private void refreshImage() {
    BitmapFactory.Options options = new
   BitmapFactory.Options();
    options.inJustDecodeBounds = true;
    BitmapFactory.decodeResource(getResources(),
            R.drawable.image1, options);
    int imageHeight = options.outHeight;
    int imageWidth = options.outWidth;
    String imageType = options.outMimeType;

    StringBuilder imageInfo = new StringBuilder();

    int id = R.drawable.image1;
    if (imageId == 2) {
        id = R.drawable.image2;
        imageInfo.append("Image 2.");
    } else if (imageId == 3) {
        id = R.drawable.image3;
        imageInfo.append("Image 3.");
    } else if (imageId == 4) {
        id = R.drawable.image4;
        imageInfo.append("Image 4.");
    } else {
        imageInfo.append("Image 1.");
    }
    imageInfo.append(" Original Dimension: " + imageWidth
            + " x " + imageHeight);
    imageInfo.append(". MIME type: " + imageType);
```

```
        options.inSampleSize = sampleSize;
        options.inJustDecodeBounds = false;
        Bitmap bitmap1 = BitmapFactory.decodeResource(
                getResources(), id, options);
        ImageView imageView1 = (ImageView)
                findViewById(R.id.image_view1);
        imageView1.setImageBitmap(bitmap1);

        TextView sampleSizeText = (TextView)
                findViewById(R.id.sample_size);
        sampleSizeText.setText("" + sampleSize);
        TextView infoText = (TextView)
                findViewById(R.id.image_info);
        infoText.setText(imageInfo.toString());

    }

    public void changeImage(View view) {
        if (imageId < 4) {
            imageId++;
        } else {
            imageId = 1;
        }
        refreshImage();
    }
}
```

The **scaleDown**, **scaleUp** and **changeImage** methods are connected to the three buttons. All methods eventually call the **refreshImage** method.

The **refreshImage** method uses the **BitmapFactory.decodeResource** method to first read the properties of the bitmap resource, by passing a **BitmapFactory.Options** whose **inJustDecodeBounds** field is set to **true**. Recall that this is a strategy for avoiding loading a large image that will take much if not all of the available memory.

```
        BitmapFactory.Options options = new
       BitmapFactory.Options();
        options.inJustDecodeBounds = true;
        BitmapFactory.decodeResource(getResources(),
                R.drawable.image1, options);
```

It then reads the dimension and image type of the bitmap.

```
        int imageHeight = options.outHeight;
        int imageWidth = options.outWidth;
        String imageType = options.outMimeType;
```

Next, it sets the **inJustDecodeBounds** field to **false** and uses the **sampleSize** value (that the user can change by clicking the Scale Up or Scale Down button) to set the **inSampleSize** field of the **BitmapFactory.Options**, and decode the bitmap for the second time.

```
        options.inSampleSize = sampleSize;
        options.inJustDecodeBounds = false;
        Bitmap bitmap1 = BitmapFactory.decodeResource(
                getResources(), id, options);
```

The dimension of the resulting Bitmap will be determined by the value of the

inSampleSize field.

Summary

The Android Bitmap API centers around the **BitmapFactory** and **Bitmap** classes. The former provides static methods for constructing a **Bitmap** object from an Android resource, a file, an InputStream, or a byte array. Some of the methods can take a **BitmapFactory.Options** to determine what kind of bitmap they will produce. The resulting bitmap can then be assigned to an **ImageView** for display.

Chapter 33
Graphics and Custom Views

You have dozens of views and widgets at your disposal. If none of these meets your need, you can create a custom view and draw directly on it using the Android Graphics API.

This chapter discusses the use of some members of the Graphics API to draw on a canvas and create a custom view. A sample application called **CanvasDemo** is presented at the end of this chapter.

Overview

The Android Graphics API comprises the members of the **android.graphics** package. The **Canvas** class in this package plays a central role in 2D graphics. You can get an instance of **Canvas** from the system and you do not need to create one yourself. Once you have an instance of **Canvas**, you can call its various methods, such as **drawColor**, **drawArc**, **drawRect**, **drawCircle**, and **drawText**.

In addition to **Canvas**, **Color** and **Paint** are frequently used. A **Color** object represents a color code as an **int**. The **Color** class defines a number of color code fields and methods for creating and converting color **int**s. Color code fields defined in **Color** includes **BLACK**, **CYAN**, **MAGENTA**, **YELLOW**, **WHITE**, **RED**, **GREEN**, and **BLUE**.

Take the **drawColor** method in **Canvas** as an example. This method accepts a color code as an argument.

```
public void drawColor(int color);
```

drawColor changes the color of the canvas with the specified color. To change the canvas color to magenta, you would write

```
canvas.drawColor(Color.MAGENTA);
```

A **Paint** object is required when drawing a shape or text. A **Paint** determines the color and transparency of the shape or text drawn as well as the font family and style of the text.

To create a **Paint**, use one of the **Paint** class's constructors:

```
public Paint()

public Paint(int flags)

public Paint(Paint anotherPaint)
```

If you use the second constructor, you can pass one or more fields defined in the **Paint** class. For example, the following code creates a **Paint** by passing the **LINEAR_TEXT_FLAG** and **ANTI_ALIAS_FLAG** fields.

```
Paint paint = new Paint(
        Paint.LINEAR_TEXT_FLAG | Paint.ANTI_ALIAS_FLAG);
```

Hardware Acceleration

Modern smartphones and tablets come with a graphic processing unit (GPU), an electronic circuit that specializes in image creation and rendering. Starting with Android 3.0, the Android framework will utilize any GPU it can find on a device, resulting in improved performance through hardware acceleration. Hardware acceleration is enabled by default for any application targeting Android API level 14 or above.

Unfortunately, currently not all drawing operations work when hardware acceleration is turned on. You can disable hardware acceleration by setting the **android:hardwareAccelerated** attribute to **false** in either the **application** or **activity** element in your android manifest file. For example, to turn off hardware acceleration for the whole application, use this:

```
<application android:hardwareAccelerated="false">
```

To disable hardware acceleration in an activity, use this:

```
<activity android:hardwareAccelerated="false" />
```

It is possible to use the **android:hardwareAccelerated** attribute in both application or activity levels. For example, the following indicates that all except one activity in the application should use hardware acceleration.

```
<application android:hardwareAccelerated="true">
    <activity ... />
    <activity android:hardwareAccelerated="false" />
</application>
```

Note

To try out the examples in this chapter, you must disable hardware acceleration.

Creating A Custom View

To create a custom view, extend the **android.view.View** class or one of its subclasses and override its **onDraw** method. Here is the signature of **onDraw**.

```
protected void onDraw (android.graphics.Canvas canvas)
```

The system calls the **onDraw** method and pass a **Canvas**. You can use the methods in **Canvas** to draw shapes and text. You can also create path and regions to draw more complex shapes.

The **onDraw** method may be called many times during the application lifecycle. As such, you should not perform expensive operations here, such as allocating objects. Objects that you need to use in **onDraw** should be created somewhere else.

For example, most drawing methods in **Canvas** require a **Paint**. Rather than creating a **Paint** in **onDraw**, you should create it at the class level and have it available for use in **onDraw**. This is illustrated in the following class.

```
public class MyCustomView extends View {
    Paint paint;
    {
        paint = ... // create a Paint object here
```

```
    }
    @Override
    protected void onDraw(Canvas canvas) {
        // use paint here.
    }
}
```

Drawing Basic Shapes

The **Canvas** class defines methods such as **drawLine**, **drawCircle**, and **drawRect** to draw shapes. For example, the following code shows how you can draw shapes in your **onDraw** method.

```
Paint paint = new Paint(Paint.FAKE_BOLD_TEXT_FLAG);

protected void onDraw(Canvas canvas) {
    // change canvas background color.
    canvas.drawColor(Color.parseColor("#bababa"));

    // draw basic shapes
    canvas.drawLine(5,  5, 200, 5, paint);
    canvas.drawLine(5,  15, 200, 15, paint);
    canvas.drawLine(5,  25, 200, 25, paint);

    paint.setColor(Color.YELLOW);
    canvas.drawCircle(50, 70, 35, paint);

    paint.setColor(Color.GREEN);
    canvas.drawRect(new Rect(100, 60, 150, 80), paint);

    paint.setColor(Color.DKGRAY);
    canvas.drawOval(new RectF(160, 60, 250, 80), paint);

    ...
}
```

Figure 33.1 shows the result.

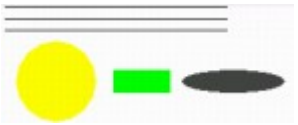

Figure 33.1: Basic shapes

Drawing Text

To draw text on a canvas, use the **drawText** method and a **Paint**. For example, the following code draws text using different colors.

```
// draw text
textPaint.setTextSize(22);
canvas.drawText("Welcome", 20, 100, textPaint);
```

```
textPaint.setColor(Color.MAGENTA);
textPaint.setTextSize(40);
canvas.drawText("Welcome", 20, 140, textPaint);
```

Figure 33.2 shows the drawn text.

Figure 33.2: Drawing text

Transparency

Android's Graphics API supports transparency. You can set the transparency by assigning an alpha value to the **Paint** used in drawing. Consider the following code.

```
// transparency
textPaint.setColor(0xFF465574);
textPaint.setTextSize(60);
canvas.drawText("Android Rocks", 20, 340, textPaint);
// opaque circle
canvas.drawCircle(80, 300, 20, paint);
// semi-transparent circles
paint.setAlpha(110);
canvas.drawCircle(160, 300, 39, paint);
paint.setColor(Color.YELLOW);
paint.setAlpha(140);
canvas.drawCircle(240, 330, 30, paint);
paint.setColor(Color.MAGENTA);
paint.setAlpha(30);
canvas.drawCircle(288, 350, 30, paint);
paint.setColor(Color.CYAN);
paint.setAlpha(100);
canvas.drawCircle(380, 330, 50, paint);
```

Figure 33.3 shows some semi transparent circles..

Figure 33.3: Transparency

Shaders

A **Shader** is a span of colors. You create a **Shader** by defining two colors as in the following code.

```
// shader
Paint shaderPaint = new Paint();
Shader shader = new LinearGradient(0, 400, 300, 500, Color.RED,
```

```
        Color.GREEN, Shader.TileMode.CLAMP);
shaderPaint.setShader(shader);
canvas.drawRect(0, 400, 200, 500, shaderPaint);
```

Figure 33.4 shows a linear gradient shader.

Figure 33.4: Using a linear gradient shader

Clipping

Clipping is the process of allocating an area on a canvas for drawing. The clipped area can be a rectangle, a circle, or any arbitrary shape you can imagine. Once you clip the canvas, any other drawing that would otherwise be rendered outside the area will be ignored.

Figure 33.5 shows a clip area in the shape of a star. After the canvas is clipped, drawn text will only be visible within the clipped area.

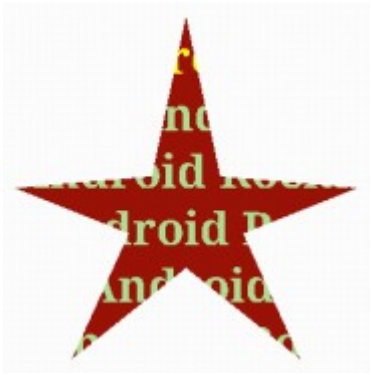

Figure 33.5: An example of clipping

The **Canvas** class provides the following methods for clipping: **clipRect**, **clipPath**, and **clipRegion**. The **clipRect** method uses a **Rect** as a clip area and **clipPath** uses a **Path**. For example, the clip area in Figure 33.5 was created using this code.

```
canvas.clipPath(starPath);
// starPath is a Path in the shape of a star, see next section
// on how to create it.
textPaint.setColor(Color.parseColor("yellow"));
canvas.drawText("Android", 350, 550, textPaint);
textPaint.setColor(Color.parseColor("#abde97"));
canvas.drawText("Android", 400, 600, textPaint);
canvas.drawText("Android Rocks", 300, 650, textPaint);
canvas.drawText("Android Rocks", 320, 700, textPaint);
canvas.drawText("Android Rocks", 360, 750, textPaint);
canvas.drawText("Android Rocks", 320, 800, textPaint);
```

You'll learn more about clipping in the next sections.

Using Paths

A **Path** is a collection of any number of straight line segments, quadratic curves, and cubic curves. A **Path** can be used for clipping or to draw text on.

As an example, this method creates a star path. It takes a coordinate that is the location of its center.

```
private Path createStarPath(int x, int y) {
    Path path = new Path();
    path.moveTo(0 + x, 150 + y);
    path.lineTo(120 + x, 140 + y);
    path.lineTo(150 + x, 0 + y);
    path.lineTo(180 + x, 140 + y);
    path.lineTo(300 + x, 150 + y);
    path.lineTo(200 + x, 190 + y);
    path.lineTo(250 + x, 300 + y);
    path.lineTo(150 + x, 220 + y);
    path.lineTo(50 + x, 300 + y);
    path.lineTo(100 + x, 190 + y);
    path.lineTo(0 + x, 150 + y);
    return path;
}
```

The following code shows how to draw text that curves along a **Path**.

```
public class CustomView extends View {
    Path curvePath;
    Paint textPaint = new Paint(Paint.LINEAR_TEXT_FLAG);
    {
        Typeface typeface = Typeface.create(Typeface.SERIF,
                Typeface.BOLD);
        textPaint.setTypeface(typeface);
        curvePath = createCurvePath();
    }

    private Path createCurvePath() {
        Path path = new Path();
        path.addArc(new RectF(400, 40, 780, 300), -210, 230);
        return path;
    }

    protected void onDraw(Canvas canvas) {
        ...
        // draw text on path
        textPaint.setColor(Color.rgb(155, 20, 10));
        canvas.drawTextOnPath("Nice artistic touches",
                curvePath, 10, 10, textPaint);
        ...
    }
}
```

Figure 33.6 shows the drawn text.

Figure 33.6: Drawing text on a path

The CanvasDemo Application

The CanvasDemo application features a custom view and contains all the code snippets presented in this chapter. Figure 33.7 shows the main activity of the application.

Figure 33.7: The CanvasDemo application

Listing 33.1 shows the **AndroidManifest.xml** file for this application. It only has one activity.

Listing 33.1: The AndroidManifest.xml file

```
<?xml version="1.0" encoding="utf-8"?>
<manifest
       xmlns:android="http://schemas.android.com/apk/res/android"
    package="com.example.canvasdemo"
    android:versionCode="1"
```

```
    android:versionName="1.0" >

    <uses-sdk
        android:minSdkVersion="18"
        android:targetSdkVersion="18" />

    <application
        android:hardwareAccelerated="false"
        android:allowBackup="true"
        android:icon="@drawable/ic_launcher"
        android:label="@string/app_name"
        android:theme="@style/AppTheme" >
        <activity
            android:name="com.example.canvasdemo.MainActivity"
            android:label="@string/app_name" >
            <intent-filter>
                <action android:name="android.intent.action.MAIN"/>
                <category
android:name="android.intent.category.LAUNCHER" />
            </intent-filter>
        </activity>
    </application>
</manifest>
```

The main actor of the application is the **CustomView** class in Listing 33.2. It extends **View** and overrides its **onDraw** method.

Listing 33.2: The CustomView class

```
package com.example.canvasdemo;
import android.content.Context;
import android.graphics.Canvas;
import android.graphics.Color;
import android.graphics.LinearGradient;
import android.graphics.Paint;
import android.graphics.Path;
import android.graphics.Rect;
import android.graphics.RectF;
import android.graphics.Shader;
import android.graphics.Typeface;
import android.view.View;

public class CustomView extends View {

    public CustomView(Context context) {
        super(context);
    }
    Paint paint = new Paint(Paint.FAKE_BOLD_TEXT_FLAG);
    Path starPath;
    Path curvePath;


    Paint textPaint = new Paint(Paint.LINEAR_TEXT_FLAG);
    Paint shaderPaint = new Paint();
    {
        Typeface typeface = Typeface.create(
                Typeface.SERIF, Typeface.BOLD);
```

```
        textPaint.setTypeface(typeface);
        Shader shader = new LinearGradient(0, 400, 300, 500,
                Color.RED, Color.GREEN, Shader.TileMode.CLAMP);
        shaderPaint.setShader(shader);
        // create star path
        starPath = createStarPath(300, 500);
        curvePath = createCurvePath();
    }

    protected void onDraw(Canvas canvas) {
        // draw basic shapes
        canvas.drawLine(5,  5, 200, 5, paint);
        canvas.drawLine(5,  15, 200, 15, paint);
        canvas.drawLine(5,  25, 200, 25, paint);

        paint.setColor(Color.YELLOW);
        canvas.drawCircle(50, 70, 35, paint);

        paint.setColor(Color.GREEN);
        canvas.drawRect(new Rect(100, 60, 150, 80), paint);

        paint.setColor(Color.DKGRAY);
        canvas.drawOval(new RectF(160, 60, 250, 80), paint);


        // draw text
        textPaint.setTextSize(22);
        canvas.drawText("Welcome", 20, 150, textPaint);
        textPaint.setColor(Color.MAGENTA);
        textPaint.setTextSize(40);
        canvas.drawText("Welcome", 20, 190, textPaint);

        // transparency
        textPaint.setColor(0xFF465574);
        textPaint.setTextSize(60);
        canvas.drawText("Android Rocks", 20, 340, textPaint);
        // opaque circle
        canvas.drawCircle(80, 300, 20, paint);
        // semi-transparent circle
        paint.setAlpha(110);
        canvas.drawCircle(160, 300, 39, paint);
        paint.setColor(Color.YELLOW);
        paint.setAlpha(140);
        canvas.drawCircle(240, 330, 30, paint);
        paint.setColor(Color.MAGENTA);
        paint.setAlpha(30);
        canvas.drawCircle(288, 350, 30, paint);
        paint.setColor(Color.CYAN);
        paint.setAlpha(100);
        canvas.drawCircle(380, 330, 50, paint);

        // draw text on path
        textPaint.setColor(Color.rgb(155, 20, 10));
        canvas.drawTextOnPath("Nice artistic touches",
                curvePath, 10, 10, textPaint);
```

```
        // shader
        canvas.drawRect(0, 400, 200, 500, shaderPaint);


        // create a star-shaped clip
        canvas.drawPath(starPath, textPaint);
        textPaint.setColor(Color.CYAN);
        canvas.clipPath(starPath);
        textPaint.setColor(Color.parseColor("yellow"));
        canvas.drawText("Android", 350, 550, textPaint);
        textPaint.setColor(Color.parseColor("#abde97"));
        canvas.drawText("Android", 400, 600, textPaint);
        canvas.drawText("Android Rocks", 300, 650, textPaint);
        canvas.drawText("Android Rocks", 320, 700, textPaint);
        canvas.drawText("Android Rocks", 360, 750, textPaint);
        canvas.drawText("Android Rocks", 320, 800, textPaint);
    }

    private Path createStarPath(int x, int y) {
        Path path = new Path();
        path.moveTo(0 + x, 150 + y);
        path.lineTo(120 + x, 140 + y);
        path.lineTo(150 + x, 0 + y);
        path.lineTo(180 + x, 140 + y);
        path.lineTo(300 + x, 150 + y);
        path.lineTo(200 + x, 190 + y);
        path.lineTo(250 + x, 300 + y);
        path.lineTo(150 + x, 220 + y);
        path.lineTo(50 + x, 300 + y);
        path.lineTo(100 + x, 190 + y);
        path.lineTo(0 + x, 150 + y);
        return path;
    }

    private Path createCurvePath() {
        Path path = new Path();
        path.addArc(new RectF(400, 40, 780, 300),
                -210, 230);
        return path;
    }
}
```

The **MainActivity** class, given in Listing 33.3, instantiates the **CustomView** class and pass the instance to its **setContentView** method. This is unlike most applications in this book where you pass a layout resource identifier to another overload of **setContentView**.

Listing 33.3: The MainActivity class

```
package com.example.canvasdemo;
import android.app.Activity;
import android.os.Bundle;

public class MainActivity extends Activity {
    @Override
```

```
    protected void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        CustomView customView = new CustomView(this);
        setContentView(customView);
    }
}
```

Summary

The Android SDK comes with a wide range of views that you can use in your applications. If none of these suits your need, you can create a custom view and draw on it. This chapter showed you how to create a custom view and draw multiple shapes on a canvas.

Chapter 34
Fragments

A powerful feature added to Android 3.0 (API level 11), fragments are components that can be embedded into an activity. Unlike custom views, fragments have their own lifecycle and may or may not have a user interface.

This chapter explains what fragments are and shows how to use them.

The Fragment Lifecycle

You create a fragment by extending the **android.app.Fragment** class or one of its subclasses. A fragment may or may not have a user interface. A fragment with no user interface (UI) normally acts as a worker for the activity the fragment is embedded into. If a fragment has a UI, it may contain views arranged in a layout file that will be loaded after the fragment is created. In many aspects, writing a fragment is similar to writing an activity.

In order to create fragments effectively, you need to know the lifecycle of a fragment. Figure 34.1 shows the lifecycle of a fragment.

A fragment's lifecycle is similar to that of an activity. For example, it has callback methods such as **onCreate**, **onResume**, and **onPause**. On top of that, there are additional methods like **onAttach**, **onActivityCreated**, and **onDetach**. **onAttach** is called after the fragment is associated with an activity and **onActivityCreated** gets called after the **onCreate** method of the activity that contains the fragment is completed. **onDetach** is called before a fragment is detached from an activity.

Here are the lifecycle methods.

- **onAttach**. Called right after the fragment is associated with its activity.
- **onCreate**. Called to create the fragment the first time.
- **onCreateView**. Called when it is time to create the layout for the fragment. It must return the fragment's root view.
- **onActivityCreated**. Called to tell the fragment that its activity's **onCreate** method has completed.
- **onStart**. Called when the fragment's view is made visible to the user.
- **onResume**. Called when the containing activity enters the **Resumed** state, which means the activity is running.
- **onPause**. Called when the containing activity is being paused.
- **onStop**. Called when the containing activity is stopped.
- **onDestroyView**. Called to allow the fragment to release resources used for its view.
- **onDestroy**. Called to allow the fragment to do final clean-up before it is destroyed.
- **onDetach**. Called right after the fragment is detached from its activity.

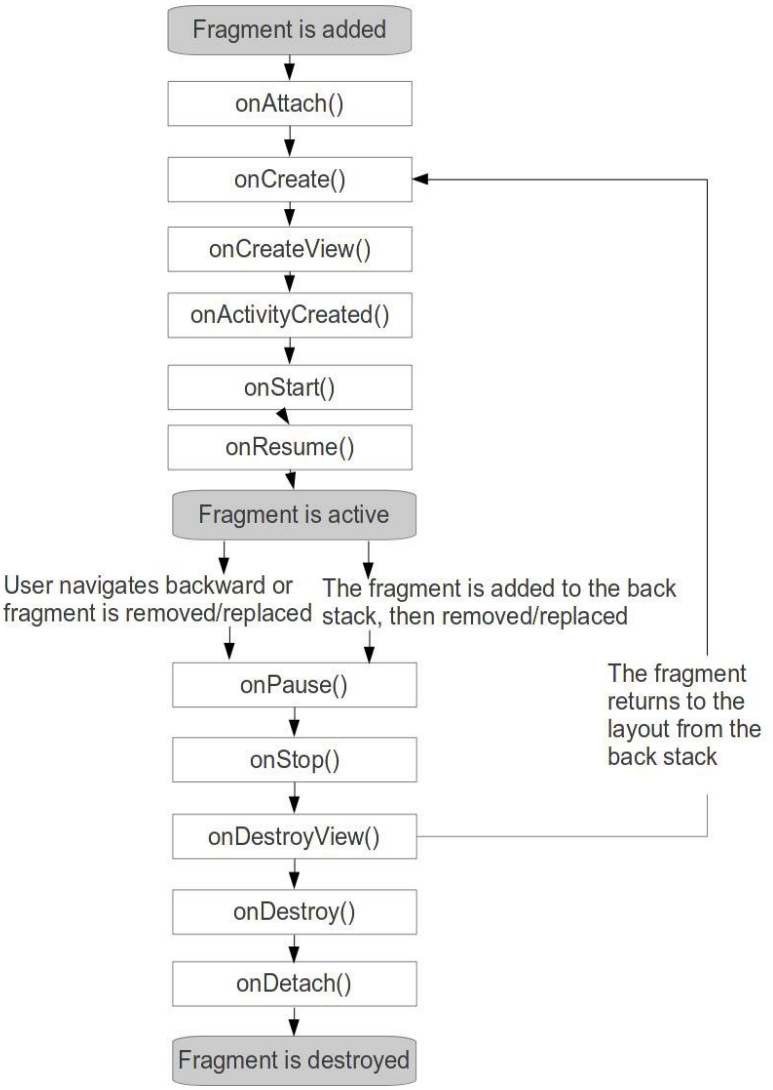

Figure 34.1: The fragment lifecycle

There are some subtle differences between an activity and a fragment. In an activity, you normally set the view for the activity in its **onCreate** method using the **setContentView** method, e.g.

```
protected void onCreate(android.os.Bundle savedInstanceState) {
    super(savedInstanceState);
    setContentView(R.layout.activity_main);
    ...
}
```

In a fragment you normally create a view in its **onCreateView** method. Here is the signature of the **onCreateView** method.

```
public View onCreateView(android.view.LayoutInflater inflater,
        android.view.ViewGroup container,
        android.os.Bundle savedInstanceState);
```

Noticed there are three arguments that are passed to **onCreateView**? The first argument is a **LayoutInflater** that you use to inflate any view in the fragment. The second argument is the parent view the fragment should be attached to. The third argument, a **Bundle**, if not null contains information from the previously saved state.

In an activity, you can obtain a reference to a view by calling the **findViewById** method on the activity. In a fragment, you can find a view in the fragment by calling the **findViewById** on the parent view.

```
View root = inflater.inflate(R.layout.fragment_names,
        container, false);
View aView = (View) root.findViewById(id);
```

Also note that a fragment should not know anything about its activity or other fragments. If you need to listen for an event that occurs in a fragment that affects the activity or other views or fragments, do not write a listener in the fragment class. Instead, trigger a new event in response to the fragment event and let the activity handle it.

You will learn more about creating a fragment in later sections in this chapter.

Fragment Management

To use a fragment in an activity, use the **fragment** element in a layout file just as you would a view. Specify the fragment class name in the **android:name** attribute and an identifier in the **android:id** attribute. Here is an example of a **fragment** element.

```
<fragment
    android:name="com.example.MyFragment"
    android:id="@+id/fragment1"
    ...
/>
```

Alternatively, You can manage fragments programmatically in your activity class using an **android.app.FragmentManager**. You can obtain the default instance of **FragmentManager** by calling the **getFragmentManager** method in your activity class. Then, call the **beginTransaction** method on the **FragmentManager** to obtain a **FragmentTransaction**.

```
FragmentManager fragmentManager = getFragmentManager();
FragmentTransaction fragmentTransaction =
        fragmentManager.beginTransaction();
```

The **android.app.FragmentTransaction** class offers methods for adding, removing, and replacing fragments. Once you're finished, call **FragmentTransaction.commit()** to commit your changes.

You can add a fragment to an activity using one of the **add** method overloads in the **FragmentTransaction** class. You have to specify to which view the fragment should be added to. Normally, you would add a fragment to a **FrameLayout** or some other type of layout. Here is one of the **add** methods in **FragmentTransaction**.

```
public abstract FragmentTransaction add(int containerViewId,
        Fragment fragment, String tag)
```

To use **add**, you would instantiate your fragment class and then specify the ID of

the view to add to. If you pass a tag, you can later retrieve your fragment using the **findFragmentByTag** method on the **FragmentManager**.

If you are not using a tag, you can use this **add** method.

```
public abstract FragmentTransaction add(int containerViewId,
        Fragment fragment)
```

To remove a fragment from an activity, call the **remove** method on the **FragmentTransaction**.

```
public abstract FragmentTransaction remove(Fragment fragment)
```

And to replace a fragment in a view with another fragment, use the **replace** method.

```
public abstract FragmentTransaction replace(int containerViewId,
        Fragment fragment, String tag)
```

As a last step once you are finished managing your fragments, call commit on the **FragmentTransaction**.

```
public abstract int commit()
```

Using A Fragment

The FragmentDemo1 application is a sample application with an activity that uses two fragments. The first fragment lists some cities. Selecting a city causes the second fragment to show the picture of the selected city. Since proper design dictates that a fragment should not know anything about its surrounding, the first fragment rises an event upon receiving user selection. The activity handles this new event and causes the second fragment to change.

Figure 34.2 shows how FragmentDemo1 looks like.

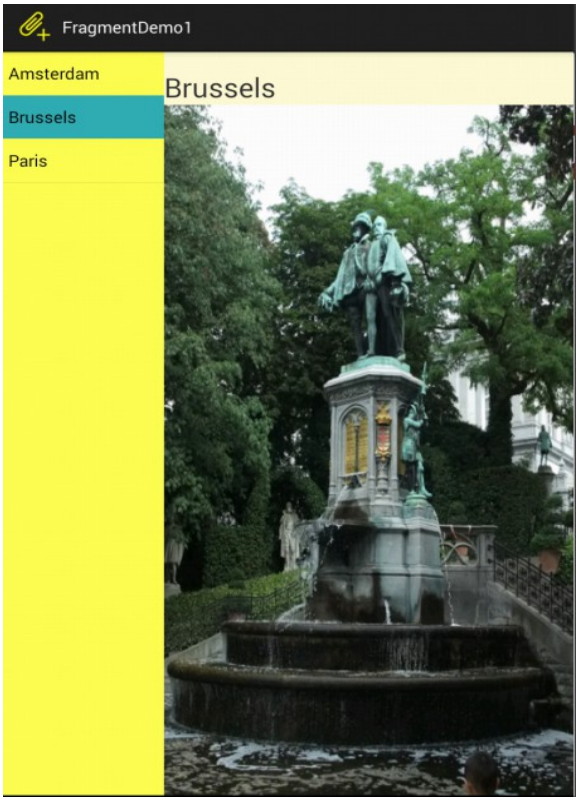

Figure 34.2: Using fragments

The manifest for the application is printed in Listing 34.1.

Listing 34.1: The AndroidManifest.xml file for FragmentDemo1

```
<?xml version="1.0" encoding="utf-8"?>
<manifest
       xmlns:android="http://schemas.android.com/apk/res/android"
    package="com.example.fragmentdemo1"
    android:versionCode="1"
    android:versionName="1.0" >

    <uses-sdk
        android:minSdkVersion="18"
        android:targetSdkVersion="18" />

    <application
        android:allowBackup="true"
        android:icon="@drawable/ic_launcher"
        android:label="@string/app_name"
        android:theme="@style/AppTheme">
        <activity
            android:name="com.example.fragmentdemo1.MainActivity"
            android:label="@string/app_name" >
            <intent-filter>
                <action android:name="android.intent.action.MAIN"/>
```

```
                <category
        android:name="android.intent.category.LAUNCHER"/>
             </intent-filter>
          </activity>
    </application>
</manifest>
```

Nothing extraordinary here. It simply declares an activity for the application.

You use a fragment as you would a view or a widget, by declaring it in an activity's layout file or by programmatically adding one. For FragmentDemo1, two fragments are added to the layout of the application's main activity. The layout file is shown in Listing 34.2.

Listing 34.2: The layout file for the main activity (activity_main.xml)

```
<LinearLayout
       xmlns:android="http://schemas.android.com/apk/res/android"
    android:orientation="horizontal"
    android:layout_width="match_parent"
    android:layout_height="match_parent">
    <fragment
       android:name="com.example.fragmentdemo1.NamesFragment"
            android:id="@+id/namesFragment"
            android:layout_weight="1"
            android:layout_width="0dp"
            android:layout_height="match_parent" />
    <fragment
       android:name="com.example.fragmentdemo1.DetailsFragment"
            android:id="@+id/detailsFragment"
            android:layout_weight="2.5"
            android:layout_width="0dp"
            android:layout_height="match_parent" />
</LinearLayout>
```

The layout for the main activity uses a horizontal **LinearLayout** that splits the screen into two panes. The ratio of the pane widths is 1:2.5, as defined by the **layout_weight** attributes of the **fragment** elements. Each pane is filled with a fragment. The first pane is represented by the **NamesFragment** class and the second pane by the **DetailsFragment** class.

The first fragment, **NamesFragment**, gets its layout from the **fragment_names.xml** file in Listing 34.3. This file is located in the **res/layout** folder.

Listing 34.3: The fragment_names.xml file

```
<ListView
    xmlns:android="http://schemas.android.com/apk/res/android"
    android:id="@+id/listView1"
    android:layout_width="wrap_content"
    android:layout_height="wrap_content"
    android:background="#FFFF55"/>
```

The layout of **NamesFragment** is very simple. It contains a single view that is a **ListView**. The layout is loaded in the **onCreateView** method of the fragment class (See Listing 34.4).

Listing 34.4: The NamesFragment class

```
package com.example.fragmentdemo1;
```

```
import android.app.Activity;
import android.app.Fragment;
import android.os.Bundle;
import android.view.LayoutInflater;
import android.view.View;
import android.view.ViewGroup;
import android.widget.AdapterView;
import android.widget.ArrayAdapter;
import android.widget.ListView;

public class NamesFragment extends Fragment {
    @Override
    public View onCreateView(LayoutInflater inflater,
            ViewGroup container, Bundle savedInstanceState) {
        final String[] names = {"Amsterdam", "Brussels", "Paris"};
        // use android.R.layout.simple_list_item_activated_1
        // to have the selected item in a different color
        ArrayAdapter<String> adapter = new ArrayAdapter<String>(
                getActivity(),
                android.R.layout.simple_list_item_activated_1,
                names);

        View view = inflater.inflate(R.layout.fragment_names,
                container, false);
        final ListView listView = (ListView) view.findViewById(
                R.id.listView1);

        listView.setChoiceMode(ListView.CHOICE_MODE_SINGLE);
        listView.setOnItemClickListener(new
                AdapterView.OnItemClickListener() {
            @Override
            public void onItemClick(AdapterView<?> parent,
                    final View view, int position, long id) {
                if (callback != null) {
                    callback.onItemSelected(names[position]);
                }
            }
        });
        listView.setAdapter(adapter);
        return view;
    }

    public interface Callback {
        public void onItemSelected(String id);
    }

    private Callback callback;

    @Override
    public void onAttach(Activity activity) {
        super.onAttach(activity);
        if (activity instanceof Callback) {
            callback = (Callback) activity;
        }
    }
```

```
    @Override
    public void onDetach() {
        super.onDetach();
        callback = null;
    }
}
```

The **NamesFragment** class defines a **Callback** interface that its activity must implement to listen to the item selection event of its **ListView**. The activity can then use it to drive the second fragment. The **onAttach** method makes sure that the implementing class is an **Activity**.

The second fragment, **DetailsFragment**, has a layout file that is given in Listing 34.5. It contains a **TextView** and an **ImageView**. The **TextView** displays the name of the selected city and the **ImageView** shows the picture of the selected city.

Listing 34.5: The fragment_details.xml file

```
<LinearLayout
       xmlns:android="http://schemas.android.com/apk/res/android"
    android:orientation="vertical"
    android:background="#FAFAD2"
    android:layout_width="match_parent"
    android:layout_height="match_parent">
    <TextView
        android:id="@+id/text1"
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:textSize="30sp"/>
    <ImageView
        android:id="@+id/imageView1"
        android:layout_width="match_parent"
        android:layout_height="match_parent"/>
</LinearLayout>
```

The **DetailsFragment** class in shown in Listing 34.6. It has a **showDetails** method that the containing activity can call to change the content of the **TextView** and **ImageView**.

Listing 34.6: The DetailsFragment class

```
package com.example.fragmentdemo1;
import android.app.Fragment;
import android.os.Bundle;
import android.view.LayoutInflater;
import android.view.View;
import android.view.ViewGroup;
import android.widget.ImageView;
import android.widget.ImageView.ScaleType;
import android.widget.TextView;

public class DetailsFragment extends Fragment {

    @Override
    public View onCreateView(LayoutInflater inflater,
            ViewGroup container, Bundle savedInstanceState) {
        return inflater.inflate(R.layout.fragment_details,
                container, false);
```

```
    }

    public void showDetails(String name) {
        TextView textView = (TextView)
                getView().findViewById(R.id.text1);
        textView.setText(name);

        ImageView imageView = (ImageView) getView().findViewById(
                R.id.imageView1);
        imageView.setScaleType(ScaleType.FIT_XY); // stretch image
        if (name.equals("Amsterdam")) {
            imageView.setImageResource(R.drawable.amsterdam);
        } else if (name.equals("Brussels")) {
            imageView.setImageResource(R.drawable.brussels);
        } else if (name.equals("Paris")) {
            imageView.setImageResource(R.drawable.paris);
        }
    }
}
```

The activity class for FragmentDemo1 is presented in Listing 34.7.

Listing 34.7: The activity class for FragmentDemo1

```
package com.example.fragmentdemo1;
import android.app.Activity;
import android.os.Bundle;

public class MainActivity extends Activity
        implements NamesFragment.Callback {

    @Override
    protected void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.activity_main);
    }
    @Override
    public void onItemSelected(String value) {
        DetailsFragment details = (DetailsFragment)
                getFragmentManager().findFragmentById(
                        R.id.detailsFragment);
        details.showDetails(value);
    }
}
```

The most important thing to note is that the activity class implements **NamesFragment.Callback** so that it can capture the item click event in the fragment. The **onItemSelected** method is an implementation for the **Callback** interface. It calls the **showDetails** method in the second fragment to change the text and picture of the selected city.

Extending ListFragment and Using FragmentManager

FragmentDemo1 showed how you could add a fragment to an activity using the

fragment element in the activity's layout file. In the second sample application, FragmentDemo2, you will learn how to add a fragment to an activity programmatically.

FragmentDemo2 is similar in functionality to its predecessor with a few differences. The first difference pertains to how the name and the picture of a selected city are updated. In FragmentDemo1, the containing activity calls the **showDetails** method in the second fragment, passing the city name. In FragmentDemo2, when a city is selected, the activity creates a new instance of **DetailsFragment** and uses it to replace the old instance.

The second difference is the fact that the first fragment extends **ListFragment** instead of **Fragment**. **ListFragment** is a subclass of **Fragment** and contains a **ListView** that fills its entire view. When subclassing **ListFragment**, you should override its **onCreate** method and call its **setListAdapter** method. This is demonstrated in the **NamesListFragment** class in Listing 34.8.

Listing 34.8: The NamesListFragment class

```
package com.example.fragmentdemo2;
import android.app.Activity;
import android.app.ListFragment;
import android.os.Bundle;
import android.view.View;
import android.widget.AdapterView;
import android.widget.ArrayAdapter;
import android.widget.ListView;

/* we don't need fragment_names-xml anymore */
public class NamesListFragment extends ListFragment {

    final String[] names = {"Amsterdam", "Brussels", "Paris"};

    @Override
    public void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        ArrayAdapter<String> adapter = new ArrayAdapter<String>(
                getActivity(),
                android.R.layout.simple_list_item_activated_1,
                names);
        setListAdapter(adapter);
    }

    @Override
    public void onViewCreated(View view,
            Bundle savedInstanceState) {
        // ListView can only be accessed here, not in onCreate()
        super.onViewCreated(view, savedInstanceState);
        ListView listView = getListView();
        listView.setChoiceMode(ListView.CHOICE_MODE_SINGLE);
        listView.setOnItemClickListener(new
                AdapterView.OnItemClickListener() {
            @Override
            public void onItemClick(AdapterView<?> parent,
                    final View view, int position, long id) {
                if (callback != null) {
                    callback.onItemSelected(names[position]);
```

```
                }
            }
        });

    }

    public interface Callback {
        public void onItemSelected(String id);
    }

    private Callback callback;

    @Override
    public void onAttach(Activity activity) {
        super.onAttach(activity);
        if (activity instanceof Callback) {
            callback = (Callback) activity;
        }
    }
    @Override
    public void onDetach() {
        super.onDetach();
        callback = null;
    }
}
```

Like the **NamesFragment** class in FragmentDemo1, the **NamesListFragment** class in FragmentDemo2 also defines a **Callback** interface that a containing activity must implement to listen to the **ListView**'s **OnItemClick** event.

The second fragment, **DetailsFragment** in Listing 34.9, expects its activity to pass two arguments, a name and an image ID. In its **onCreate** method, the fragment retrieves these arguments and store them in class level variables, **name** and **imageId**. The values of the variables are then used in its **onCreateView** method to populate its **TextView** and **ImageView**.

Listing 34.9: The DetailsFragment class

```
package com.example.fragmentdemo2;
import android.app.Fragment;
import android.os.Bundle;
import android.view.LayoutInflater;
import android.view.View;
import android.view.ViewGroup;
import android.widget.ImageView;
import android.widget.ImageView.ScaleType;
import android.widget.TextView;

public class DetailsFragment extends Fragment {

    int imageId;
    String name;

    public DetailsFragment() {
    }

    @Override
```

```
    public void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        if (getArguments().containsKey("name")) {
            name = getArguments().getString("name");
        }
        if (getArguments().containsKey("imageId")) {
            imageId = getArguments().getInt("imageId");
        }
    }

    @Override
    public View onCreateView(LayoutInflater inflater,
            ViewGroup container, Bundle savedInstanceState) {

        View rootView = inflater.inflate(
                R.layout.fragment_details, container, false);
        TextView textView = (TextView)
                rootView.findViewById(R.id.text1);
        textView.setText(name);

        ImageView imageView = (ImageView) rootView.findViewById(
                R.id.imageView1);
        imageView.setScaleType(ScaleType.FIT_XY); //stretch image
        imageView.setImageResource(imageId);
        return rootView;
    }
}
```

Now that you have looked at the fragments, take a close look at the activity. The layout file is given in Listing 34.10. Instead of two fragment elements like in FragmentDemo1, the activity layout file in FragmentDemo2 has a fragment element and a **FrameLayout**. The latter acts as the container for the second fragment.

Listing 34.10: The activity_main.xml file

```
<LinearLayout
       xmlns:android="http://schemas.android.com/apk/res/android"
    android:orientation="horizontal"
    android:layout_width="match_parent"
    android:layout_height="match_parent">
    <fragment
       android:name="com.example.fragmentdemo2.NamesListFragment"
            android:id="@+id/namesFragment"
            android:layout_weight="1"
            android:layout_width="0dp"
            android:layout_height="match_parent"/>
    <FrameLayout
        android:id="@+id/details_container"
        android:layout_width="0dp"
        android:layout_height="match_parent"
        android:layout_weight="2.5"/>
</LinearLayout>
```

The activity class for FragmentDemo2 is given in Listing 34.11. Like the activity class in FragmentDemo1, it also implements the **Callback** interface. However, its implementation of the **onItemSelected** method is different. First, it passes two

arguments to the **DetailsFragment**. Second, every time **onItemSelected** is called, a new **DetailsFragment** instance is created and passed to the **FrameLayout**.

Listing 34.11: The MainActivity class

```
package com.example.fragmentdemo2;
import android.app.Activity;
import android.app.FragmentManager;
import android.app.FragmentTransaction;
import android.os.Bundle;

public class MainActivity extends Activity
implements NamesListFragment.Callback {

    @Override
    protected void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.activity_main);

    }
    @Override
    public void onItemSelected(String value) {

        Bundle arguments = new Bundle();
        arguments.putString("name", value);
        if (value.equals("Amsterdam")) {
            arguments.putInt("imageId", R.drawable.amsterdam);
        } else if (value.equals("Brussels")) {
            arguments.putInt("imageId", R.drawable.brussels);
        } else if (value.equals("Paris")) {
            arguments.putInt("imageId", R.drawable.paris);
        }
        DetailsFragment fragment = new DetailsFragment();
        fragment.setArguments(arguments);
        FragmentManager fragmentManager = getFragmentManager();
        FragmentTransaction fragmentTransaction =
                fragmentManager.beginTransaction();
        fragmentTransaction.replace(
                R.id.details_container, fragment);
        fragmentTransaction.commit();
    }
}
```

Figure 34.3 shows FragmentDemo2.

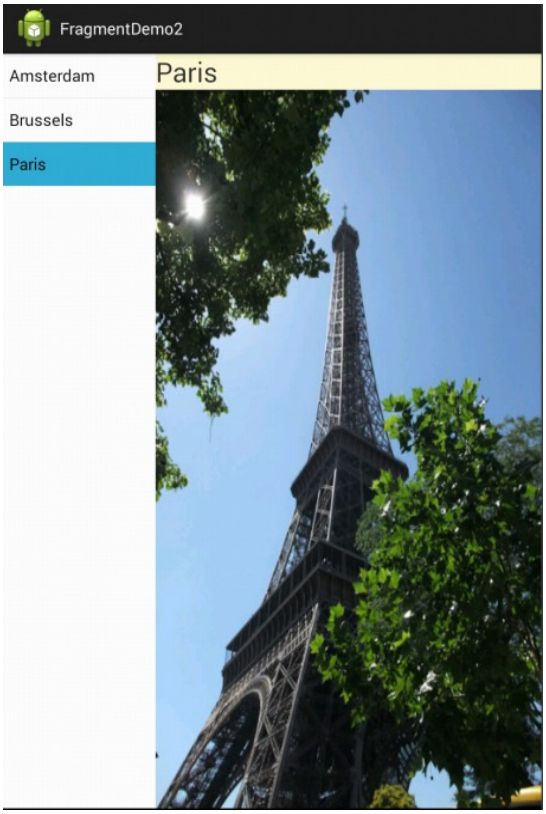

Figure 34.3: FragmentDemo2

Summary

Fragments are components that can be added to an activity. A fragment has its own lifecycle and has methods that get called when certain phases of its life occur. In this chapter you have learn to write your own fragments.

Chapter 35
Multi-Pane Layouts

An Android tablet generally has a larger screen than that of a handset. Many application developers might want to take advantage of the bigger screen in tablets to display more information by using a multi-pane layout.

This chapter discusses multi-pane layouts using fragments that you already learned in Chapter 34, "Fragments."

Overview

A tablet has a larger screen than a handset and you can display more information on a tablet than on a handset. If you are writing an application that needs to look good on both types of devices, a common strategy is to support two layouts. A single-pane layout can be used for handsets and a multi-pane layout for tablets.

Figure 35.1 shows a dual-pane version of an application and Figure 35.2 shows the same application in single-pane mode.

In a single layout, you would display an activity that often contains a single fragment, which in turn often contains a **ListView**. Selecting an item on the **ListView** would start another activity.

In a multi-pane layout, you would have an activity that is big enough for two panes. You would use the same fragment, but this time when an item is selected, it updates a second fragment instead of starting another activity.

The question is, how do you tell the system to pick the right layout? Prior to Android 3.2 (API level 13) a screen may fall into one of these categories depending on its size:

- small, for screens that are at least 426dp x 320dp
- normal, for screens that are at least 470dp x 320dp
- large, for screens that are at least 640dp x 480dp
- xlarge, for screens that are at least 960dp x 720dp

Here, dp stands for density independent pixel. You can calculate the number of pixels (px) from the dp and the screen density (in dots per inch or dpi) by using this formula.

```
px = dp * (dpi / 160)
```

To support a screen category, you would place your layout files in the folder dedicated to that category, that is **res/layout-small** for small screens, **res/layout** for normal screens, **res/layout-large** for large screens, and **res/layout-xlarge** for xlarge screens. To support both normal and large screens, you would have layout files in both **res/layout** and **res/layout-large** directories.

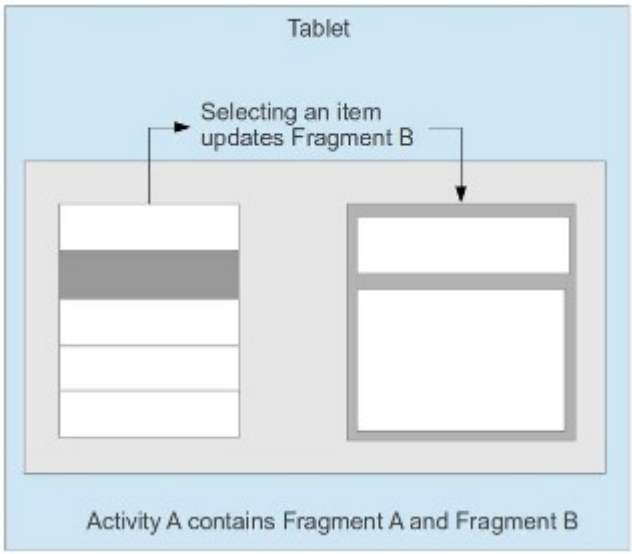

Figure 35.1: Dual-pane layout

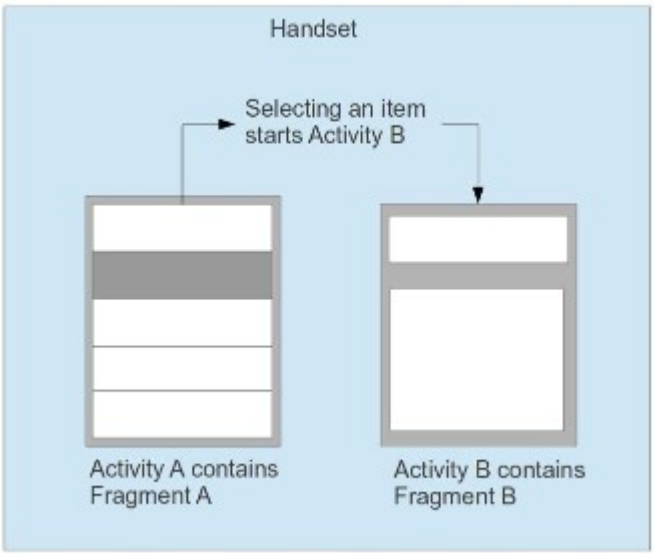

Figure 35.2: Single-pane layout

The system is not without limitations, however. For example, a 7" tablet and a 10" tablet would both fall into the xlarge category, even though they provide different amounts of space. To allow for different layouts for 7" and 10" tablets, Android 3.2 changed the way it worked. Instead of the four screen sizes, Android 3.2 and later employ a new technique that measures the screen based on the amount of space in dp, rather than trying to make the layout fit the generalized size groups.

With the new system, it is easy to provide different layouts for tablets with a 600dp screen width (such as in a typical 7" tablet) and tablets with a 720dp screen width (such as in a typical 10" tablet). A typical handset, by the way, has a 320dp screen width.

Now, to support large screen devices for both pre-3.2 devices and later devices, you need to store layout files in both **res/layout-large** and **res/layout-sw600dp** directories. In other words, for each layout you end up with three files (assuming your layout file is called **main.xml**):

- **res/layout/main.xml** for normal screens
- **res/layout-large/main.xml** for devices running pre-3.2 Android having a large screen
- **res/layout-sw600dp/main.xml** for devices running Android 3.2 or later having a large screen

In addition, if your application has a different screen for 10" tablets, you will also need a **res/layout-sw720dp/main.xml** file.

The **main.xml** files in the **layout-large** and **layout-sw600dp** directories are identical and having duplicates that both have to be changed if one of them was updated is certainly a maintenance nightmare.

To get around it, you can use references. With references, you only need two layout files, one for normal screens and one for large screens, both in the **res/layout** directory. Assuming the names of your layout files are **main.xml** and **main_large.xml**, to reference the latter, you need to have a **refs.xml** file in both **res/values-large** and **res/values-sw600dp**. The content of **refs.xml** would be as follows.

```
<resources>
    <item name="main" type="layout">@layout/main_large</item>
</resources>
```

Figure 35.3 shows the content of the **res** directory.

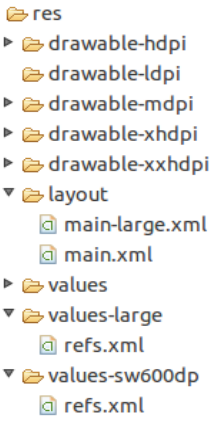

Figure 35.3: The structure of the res directory that supports layout references

This way, you still have two identical files, the **refs.xml** file in the **values-large** directory and the **refs.xml** file in the **values-sw600dp** directory. However, these are reference files that do not need to be updated if the layout changes.

A Multi-Pane Example

FragmentDemo3 is an application that supports small and large screens. For large screens it shows an activity that uses a multi-pane layout consisting of two

fragments. For smaller screens, another activity will be shown that contains only one fragment.

The easiest way to create a multi-pane application is by using ADT Eclipse. As usual, you would use the New Android Application wizard as described in Chapter 22, "Your First Android Application." However, instead of creating a blank activity as in Chapter 22, you should select Master/Detail Flow, as shown in Figure 35.4.

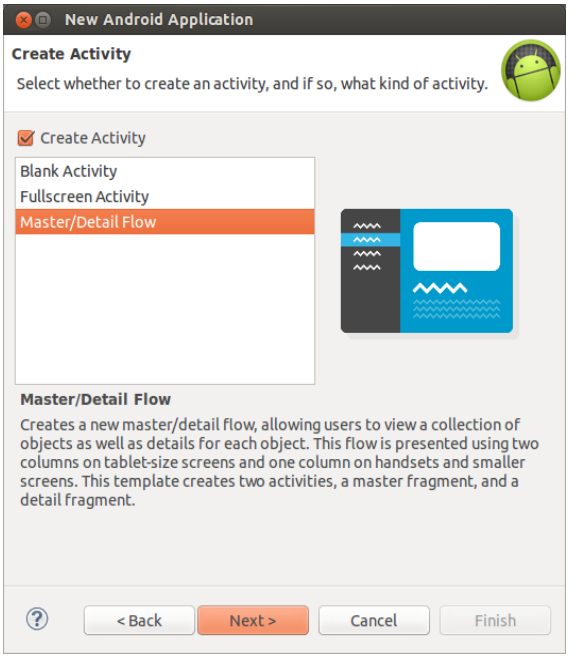

Figure 35.4: Selecting Master/Detail Flow activity

After you reach the window in Figure 35.4, click **Next**. In the window that appears next (See Figure 35.5), select the name for your item(s) and click **Finish**.

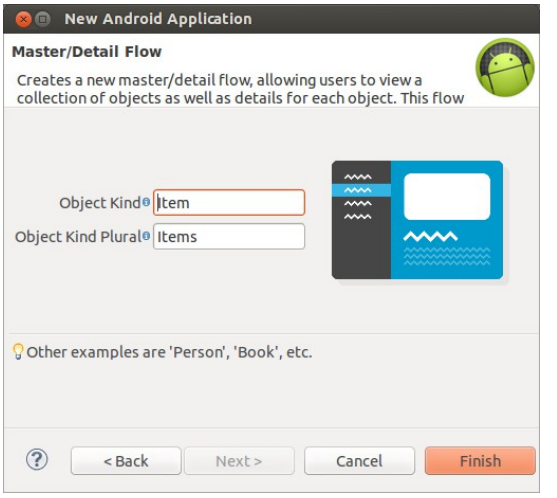

Figure 35.5: Choosing names for the items

Note that ADT Eclipse creates a multi-pane layout that supports Android 3.0 and

later as well as pre-3.0 Android. If you don't need to support older devices, however, you can remove the support classes. The advantage is you will have an apk file that is about 30KB lighter.

The **AndroidManifest.xml** file for the application is given in Listing 35.1.

Listing 35.1: The AndroidManifest.xml file

```
<?xml version="1.0" encoding="utf-8"?>
<manifest
       xmlns:android="http://schemas.android.com/apk/res/android"
    package="com.example.fragmentdemo3"
    android:versionCode="1"
    android:versionName="1.0" >

    <uses-sdk
        android:minSdkVersion="18"
        android:targetSdkVersion="18" />

    <application
        android:allowBackup="true"
        android:icon="@drawable/ic_launcher"
        android:label="@string/app_name"
        android:theme="@style/AppTheme" >
        <activity

       android:name="com.example.fragmentdemo3.ItemListActivity"
            android:label="@string/app_name" >
            <intent-filter>
                <action
       android:name="android.intent.action.MAIN" />
                <category
android:name="android.intent.category.LAUNCHER" />
            </intent-filter>
        </activity>
        <activity

       android:name="com.example.fragmentdemo3.ItemDetailActivity"
            android:label="@string/title_item_detail"
            android:parentActivityName=".ItemListActivity" >
            <meta-data
                android:name="android.support.PARENT_ACTIVITY"
                android:value=".ItemListActivity" />
        </activity>
    </application>

</manifest>
```

The application has two activities. The main activity is used in both single-pane and multi-pane environments. The second activity is used in the single-pane environment only.

The Layouts and Activities

As you can see in the manifest, the **ItemListActivity** class is the activity class that will be instantiated when the application is launched. This class is shown in Listing 35.2.

Listing 35.2: The ItemListActivity class

```
package com.example.fragmentdemo3;
import android.content.Intent;
import android.os.Bundle;
import android.support.v4.app.FragmentActivity;

public class ItemListActivity extends FragmentActivity implements
        ItemListFragment.Callbacks {

    private boolean mTwoPane;

    @Override
    protected void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.activity_item_list);

        if (findViewById(R.id.item_detail_container) != null) {
            mTwoPane = true;
           ((ItemListFragment)  getSupportFragmentManager().
                    findFragmentById( R.id.item_list)).
                    setActivateOnItemClick(true);
        }
    }

    /**
     * Callback method from ItemListFragment indicating that
     * the item with the given ID was selected.
     */
    @Override
    public void onItemSelected(String id) {
        if (mTwoPane) {
            // In two-pane mode, show the detail view in this
            // activity by adding or replacing the detail fragment
            // using a fragment transaction.
            Bundle arguments = new Bundle();
            arguments.putString(ItemDetailFragment.ARG_ITEM_ID,
       id);
            ItemDetailFragment fragment = new ItemDetailFragment();
            fragment.setArguments(arguments);
            getSupportFragmentManager().beginTransaction()
                    .replace(R.id.item_detail_container,
       fragment).commit();

         } else {
            // In single-pane mode, simply start the detail
       activity
            // for the selected item ID.
            Intent detailIntent = new Intent(this,
                    ItemDetailActivity.class);
            detailIntent.putExtra(ItemDetailFragment.ARG_ITEM_ID,
                    id);
            startActivity(detailIntent);
         }
    }
}
```

The **onCreate** method in **ItemListActivity** loads the layout indicated by layout identifier **R.layout.activity_item_list**.

```
    protected void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.activity_item_list);
    ...
```

In devices with smaller screens, the **res/layout/activity_item_list.xml** will be loaded. In devices with larger screens, the system will try to locate the **activity_item_list.xml** file in either the **res/layout-large** or **res/layout-sw600dp** directory. However, none of these directories exists so the system searches the **values-large** and **values-sw600dp** directories. In these directories, it finds two files named **refs.xml** that are given in Listing 35.3. The **refs.xml** files are identical and define an item named **activity_item_list** (the layout ID being sought) that points to **layout/activity_item_twopane**. As a result, the **layout/activity_item_twopane.xml** file is loaded when the application is deployed to a device with a large screen.

Listing 35.3: The refs.xml file

```
<resources>
    <item name="activity_item_list"
       type="layout">@layout/activity_item_twopane</item>
</resources>
```

The **activity_item_twopane.xml** file (in Listing 35.4) is used in devices with a large screen.

Listing 35.4: The activity_item_twopane.xml file (multi-pane)

```
<LinearLayout
       xmlns:android="http://schemas.android.com/apk/res/android"
    android:layout_width="match_parent"
    android:layout_height="match_parent"
    android:layout_marginLeft="16dp"
    android:layout_marginRight="16dp"
    android:baselineAligned="false"
    android:divider="?android:attr/dividerHorizontal"
    android:orientation="horizontal"
    android:showDividers="middle">

    <fragment
        android:id="@+id/item_list"
        android:name="com.example.fragmentdemo3.ItemListFragment"
        android:layout_width="0dp"
        android:layout_height="match_parent"
        android:layout_weight="1"/>

    <FrameLayout
        android:id="@+id/item_detail_container"
        android:layout_width="0dp"
        android:layout_height="match_parent"
        android:layout_weight="3" />
</LinearLayout>
```

The **activity_item_twopane.xml** layout file features a horizontal **LinearLayout** that splits the screen into two panes. The left pane consists of a fragment that contains a **ListView**. The right pane contains a **FrameLayout** to which instances

of another fragment called **ItemDetailFragment** can be added. Listing 35.5 shows the layout for **ItemDetailFragment**.

Listing 35.5: The fragment_item_detail.xml file

```
<TextView
       xmlns:android="http://schemas.android.com/apk/res/android"
    xmlns:tools="http://schemas.android.com/tools"
    android:id="@+id/item_detail"
    style="?android:attr/textAppearanceLarge"
    android:layout_width="match_parent"
    android:layout_height="match_parent"
    android:padding="16dp"
    android:textIsSelectable="true" />
```

For smaller screens, two activities will be used. The main activity will load the **activity_item_list.xml** layout file in Listing 35.6. This layout contains the same fragment used by the left pane in the multi-pane layout.

Listing 35.6: The activity_item_list.xml file (single-pane)

```
<fragment
       xmlns:android="http://schemas.android.com/apk/res/android"
    xmlns:tools="http://schemas.android.com/tools"
    android:id="@+id/item_list"
    android:name="com.example.fragmentdemo3.ItemListFragment"
    android:layout_width="match_parent"
    android:layout_height="match_parent"
    android:layout_marginLeft="16dp"
    android:layout_marginRight="16dp"
    tools:context=".ItemListActivity"
    tools:layout="@android:layout/list_content"/>
```

The Fragment Classes

The two fragment classes are given in Listing 35.7 and Listing 35.8, respectively.

Listing 35.7: The ItemListFragment class

```
package com.example.fragmentdemo3;
import android.app.Activity;
import android.os.Bundle;
import android.support.v4.app.ListFragment;
import android.view.View;
import android.widget.ArrayAdapter;
import android.widget.ListView;
import com.example.fragmentdemo3.dummy.DummyContent;
/**
 * A list fragment representing a list of Items. This fragment
 * also supports tablet devices by allowing list items to be given
 * an 'activated' state upon selection. This helps indicate which
 * item is currently being viewed in a {@link ItemDetailFragment}.
 * <p>
 * Activities containing this fragment MUST implement the
 * {@link Callbacks} interface.
 */
public class ItemListFragment extends ListFragment {

    /**
```

```
 * The serialization (saved instance state) Bundle key
 * representing the activated item position. Only used on
 * tablets.
 */
private static final String STATE_ACTIVATED_POSITION =
        "activated_position";

/**
 * The fragment's current callback object, which is notified
 * of list item clicks.
 */
private Callbacks mCallbacks = sDummyCallbacks;

/**
 * The current activated item position. Only used on tablets.
 */
private int mActivatedPosition = ListView.INVALID_POSITION;

/**
 * A callback interface that all activities containing this
 * fragment must implement. This mechanism allows activities
 * to be notified of item selections.
 */
public interface Callbacks {
    /**
     * Callback for when an item has been selected.
     */
    public void onItemSelected(String id);
}

/**
 * A dummy implementation of the {@link Callbacks} interface
 * that does nothing. Used only when this fragment is not
 * attached to an activity.
 */
private static Callbacks sDummyCallbacks = new Callbacks() {
    @Override
    public void onItemSelected(String id) {
    }
};

/**
 * Mandatory empty constructor for the fragment manager to
 * instantiate the fragment (e.g. upon screen orientation
 * changes).
 */
public ItemListFragment() {
}

@Override
public void onCreate(Bundle savedInstanceState) {
    super.onCreate(savedInstanceState);

    // TODO: replace with a real list adapter.
    setListAdapter(new ArrayAdapter<DummyContent.DummyItem>(
```

```
            getActivity(),
            android.R.layout.simple_list_item_activated_1,
            android.R.id.text1, DummyContent.ITEMS));
}

@Override
public void onViewCreated(View view,
        Bundle savedInstanceState) {
    super.onViewCreated(view, savedInstanceState);

    // Restore the previously serialized activated item
    // position.
    if (savedInstanceState != null
            && savedInstanceState
                    .containsKey(STATE_ACTIVATED_POSITION)) {
        setActivatedPosition(savedInstanceState
                .getInt(STATE_ACTIVATED_POSITION));
    }
}

@Override
public void onAttach(Activity activity) {
    super.onAttach(activity);

    // Activities containing this fragment must implement
    // its callbacks.
    if (!(activity instanceof Callbacks)) {
        throw new IllegalStateException(
                "Activity must implement fragment's " +
                "callbacks.");
    }
    mCallbacks = (Callbacks) activity;
}

@Override
public void onDetach() {
    super.onDetach();
    // Reset the active callbacks interface to the dummy
    // implementation.
    mCallbacks = sDummyCallbacks;
}

@Override
public void onListItemClick(ListView listView, View view,
        int position, long id) {
    super.onListItemClick(listView, view, position, id);

    // Notify the active callbacks interface (the activity,
    // if the fragment is attached to one) that an item
    // has been selected.
    mCallbacks.onItemSelected(
              DummyContent.ITEMS.get(position).id);
}

@Override
```

```
    public void onSaveInstanceState(Bundle outState) {
        super.onSaveInstanceState(outState);
        if (mActivatedPosition != ListView.INVALID_POSITION) {
            // Serialize and persist the activated item position.
            outState.putInt(STATE_ACTIVATED_POSITION,
                    mActivatedPosition);
        }
    }

    /**
     * Turns on activate-on-click mode. When this mode is on,
     * list items will be given the 'activated' state when
     * touched.
     */
    public void setActivateOnItemClick(
            boolean activateOnItemClick) {
        // When setting CHOICE_MODE_SINGLE, ListView will
        // automatically give items the 'activated' state when
        // touched.
        getListView().setChoiceMode(
                activateOnItemClick ? ListView.CHOICE_MODE_SINGLE
                        : ListView.CHOICE_MODE_NONE);
    }

    private void setActivatedPosition(int position) {
        if (position == ListView.INVALID_POSITION) {
            getListView()
                    .setItemChecked(mActivatedPosition, false);
        } else {
            getListView().setItemChecked(position, true);
        }
        mActivatedPosition = position;
    }
}
```

The **ItemListFragment** class extends **ListFragment** and gets the data for its **ListView** from a **DummyContent** class. It also provides a **Callbacks** interface that any activity using this fragment must implement to handle the **ListItemClick** event of the **ListView**. In the **onAttach** method, the fragment makes sure the activity class implements Callbacks and replaces the content of mCallbacks with the activity, in effect delegating the event handling to the activity.

Listing 35.8: The ItemDetailFragment class

```
package com.example.fragmentdemo3;
import android.os.Bundle;
import android.support.v4.app.Fragment;
import android.view.LayoutInflater;
import android.view.View;
import android.view.ViewGroup;
import android.widget.TextView;
import com.example.fragmentdemo3.dummy.DummyContent;

/**
 * A fragment representing a single Item detail screen. This
 * fragment is either contained in a {@link ItemListActivity}
 * in two-pane mode (on tablets) or a {@link ItemDetailActivity}
```

```
 * on handsets.
 */
public class ItemDetailFragment extends Fragment {
    /**
     * The fragment argument representing the item ID that
     * this fragment represents.
     */
    public static final String ARG_ITEM_ID = "item_id";

    /**
     * The dummy content this fragment is presenting.
     */
    private DummyContent.DummyItem mItem;

    /**
     * Mandatory empty constructor for the fragment manager to
     * instantiate the fragment (e.g. upon screen orientation
     * changes).
     */
    public ItemDetailFragment() {
    }

    @Override
    public void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);

        if (getArguments().containsKey(ARG_ITEM_ID)) {
            // Load the dummy content specified by the fragment
            // arguments. In a real-world scenario, use a Loader
            // to load content from a content provider.
            mItem = DummyContent.ITEM_MAP.get(
                    getArguments().getString(ARG_ITEM_ID));
        }
    }

    @Override
    public View onCreateView(LayoutInflater inflater,
            ViewGroup container, Bundle savedInstanceState) {
        View rootView = inflater.inflate(
                R.layout.fragment_item_detail, container, false);
        // Show the dummy content as text in a TextView.
        if (mItem != null) {
            ((TextView) rootView.findViewById(R.id.item_detail))
                    .setText(mItem.content);
        }
        return rootView;
    }
}
```

Running the Application

Figure 35.6 and Figure 35.7 show the FragmentDemo3 application on a tablet and a handset, respectively.

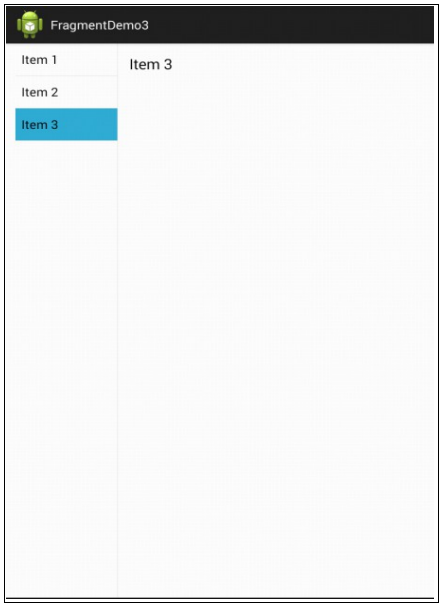

Figure 35.6: Multi-pane layout on a large screen

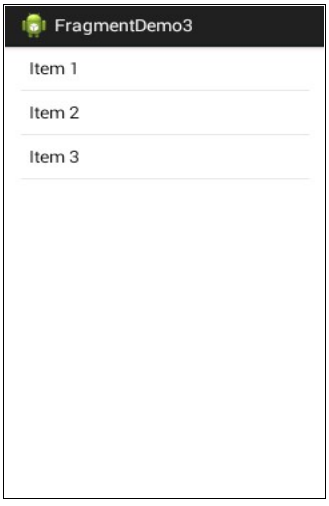

Figure 35.7: Single-pane layout on a small screen

Summary

To give your users the best experience, you may want to use different layouts for different screen sizes. In this chapter, you learned that a good strategy to achieve that is to use a multi-pane layout for tablets and a single-pane layout for handsets.

Chapter 36
Animation

Animation is an interesting feature in Android and animation capabilities have been available since the very beginning (API Level 1). In this chapter you will learn to use an Animation API called property animation, which was added to Honeycomb (API Level 11). The new API is more powerful than the previous animation technology called view animation. You should use property animation in new projects.

Overview

The Property Animation API consists of types in the **android.animation** package. The old animation API, called view animation, resides in the **android.view.animation** package. This chapter focuses on the new animation API and does not discuss the older technology. It also does not discuss drawable animation, which is the type of animation that works by loading a series of images, played one after another like a roll of film. For more information on drawable animation, see the documentation for **android.graphics.drawable.AnimationDrawable**.

Property Animation

The powerhouse behind property animation is the **android.animation.Animator** class. It is an abstract class, so you do not use this class directly. Instead, you use one of its subclasses, either **ValueAnimator** or **ObjectAnimator**, to make an animation. In addition, the **AnimatorSet** class, another subclass of **Animator**, is designed to run multiple animations in parallel or sequentially.

All these classes reside in the same package and this section looks at these classes.

Animator

The **Animator** class is an abstract class that provides methods that are inherited by subclasses. There is a method for setting the target object to be animated (**setTarget**), a method for setting the duration (**setDuration**), and a method for starting the animation (**start**). The **start** method can be called more than once on an **Animator** object.

In addition, this class provides an **addListener** method that takes an **Animator.AnimatorListener** instance. The **AnimatorListener** interface is defined inside the **Animator** class and provides methods that will be called by the system upon the occurrence of certain events. You can implement any of these methods if you want to respond to a certain event.

The methods in **AnimatorListener** are as follows.

```
void onAnimationStart(Animator animation);
void onAnimationEnd(Animator animation);
void onAnimationCancel(Animator animation);
void onAnimationRepeat(Animator animation);
```

For example, the **onAnimationStart** method is called when the animation starts and the **onAnimationEnd** method is called when it ends.

ValueAnimator

A **ValueAnimator** creates an animation by calculating a value that transitions from a start value and to an end value. You specify what the start value and end value should be when constructing the **ValueAnimator**. By registering an **UpdateListener** to a **ValueAnimator**, you can receive an update at each frame, giving you a chance to update your object(s).

Here are two static factory methods that you can use to construct a **ValueAnimator**.

```
public static ValueAnimator ofFloat(float... values)
public static ValueAnimator ofInt(int... values)
```

Which method you should use depends on whether you want to receive an **int** or a **float** in each frame.

Once you create a **ValueAnimator**, you should create an implementation of **AnimationUpdateListener**, write your animation code under its **onAnimationUpdate** method, and register the listener with the **ValueAnimator**. Here is an example.

```
valueAnimator.addUpdateListener(new
        ValueAnimator.AnimatorUpdateListener() {
    @Override
    public void onAnimationUpdate(ValueAnimator animation) {
        Float value = (Float) animation.getAnimatedValue();
        // use value to set a property or multiple properties
        // Example: view.setRotationX(value);
    }
});
```

Finally, call the **ValueAnimator**'s **setDuration** method to set a duration and its **start** method to start the animation. If you do not call **setDuration**, the default method (300ms) will be used.

More on using **ValueAnimator** is given in the example below.

ObjectAnimator

The **ObjectAnimator** class offers the easiest way to animate an object, most probably a **View**, by continually updating one of its properties. To create an animation, create an **ObjectAnimator** using one of its factory methods, passing a target object, a property name, and the start and end values for the property. In recognition of the fact that a property can have an **int** value, a **float** value, or another type of value, **ObjectAnimator** provides three static methods: **ofInt**, **ofFloat**, and **ofObject**. Here are their signatures.

```
public static ObjectAnimator ofInt(java.lang.Object target,
```

```
        java.lang.String propertyName, int... values)
public static ObjectAnimator ofFloat(java.lang.Object target,
        java.lang.String propertyName, float... values)
public static ObjectAnimator ofObject(java.lang.Object target,
        java.lang.String propertyName, java.lang.Object... values)
```

You can pass one or two arguments to the *values* argument. If you pass two arguments, the first will be used as the start value and the second the end value. If you pass one argument, the value will be used as the end value and the current value of the property will be used as the start value.

Once you have an **ObjectAnimator**, call the **setDuration** method on the **ObjectAnimator** to set the duration and the **start** method to start it. Here is an example of animating the **rotation** property of a **View**.

```
ObjectAnimator objectAnimator = ObjectAnimator.ofFloat(view,
        "rotationY", 0F, 720.0F); // rotate 720 degrees.
objectAnimator.setDuration(2000); // 2000 milliseconds
objectAnimator.start();
```

Running the animation will cause the view to make two full circles within two seconds.

As you can see, you just need two or three lines of code to create a property animation using **ObjectAnimator**. You will learn more about **ObjectAnimator** in the example below.

AnimatorSet

An **AnimatorSet** is useful if you want to play a set of animations in a certain order. A direct subclass of **Animator**, the **AnimatorSet** class allows you to play multiple animations together or one after another. Once you're finished deciding how your animations should be called, call the **start** method on the **AnimatorSet** to start it.

The **playTogether** method arranges the supplied animations to play together. There are two overrides for this method.

```
public void playTogether(java.util.Collection<Animator> items)
public void playTogether(Animator... items)
```

The **playSequentially** method arranges the supplied animations to play sequentially. It too has two overrides.

```
public void playSequentially(Animator... items)
public void playSequentially(java.util.List<Animator> items)
```

Example

The AnimationDemo project uses the **ValueAnimator**, **ObjectAnimator**, and **AnimatorSet** to animate an **ImageView**. It provides three buttons to play different animations.

The manifest for the application is given in Listing 36.1.

Listing 36.1: The manifest for AnimationDemo

```
<?xml version="1.0" encoding="utf-8"?>
```

```
<manifest
       xmlns:android="http://schemas.android.com/apk/res/android"
    package="com.example.animationdemo"
    android:versionCode="1"
    android:versionName="1.0" >

    <uses-sdk
        android:minSdkVersion="11"
        android:targetSdkVersion="18" />

    <application
        android:allowBackup="true"
        android:icon="@drawable/ic_launcher"
        android:label="@string/app_name"
        android:theme="@style/AppTheme" >
        <activity
            android:name="com.example.animationdemo.MainActivity"
            android:label="@string/app_name" >
            <intent-filter>
                <action
       android:name="android.intent.action.MAIN" />

                <category
       android:name="android.intent.category.LAUNCHER" />
            </intent-filter>
        </activity>
    </application>

</manifest>
```

Note that the minimum SDK level is 11 (Honeycomb).

The application has one activity, whose layout is printed in Listing 36.2

Listing 36.2: The activity_main.xml file

```
<LinearLayout
    xmlns:android="http://schemas.android.com/apk/res/android"
    xmlns:tools="http://schemas.android.com/tools"
    android:layout_width="match_parent"
    android:layout_height="match_parent"
    android:paddingBottom="@dimen/activity_vertical_margin"
    android:paddingLeft="@dimen/activity_horizontal_margin"
    android:paddingRight="@dimen/activity_horizontal_margin"
    android:paddingTop="@dimen/activity_vertical_margin"
    tools:context=".MainActivity" >

    <ImageView
        android:id="@+id/imageView1"
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:layout_gravity="top|center"
        android:src="@drawable/photo1" />

    <Button
        android:id="@+id/button1"
        android:text="@string/button_animate1"
```

```
        android:textColor="#ff4433"
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:onClick="animate1"/>
    <Button
        android:id="@+id/button2"
        android:text="@string/button_animate2"
        android:textColor="#33ff33"
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:onClick="animate2"/>
    <Button
        android:id="@+id/button3"
        android:text="@string/button_animate3"
        android:textColor="#3398ff"
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:onClick="animate3"/>
</LinearLayout>
```

The layout defines an **ImageView** and three **Button**s.

Finally, Listing 36.3 shows the **MainActivity** class for the application. There are three event-processing methods (**animate1**, **animate2**, and **animate3**) that each uses a different method of animation.

Listing 36.3: The MainActivity class

```
package com.example.animationdemo;
import android.animation.AnimatorSet;
import android.animation.ObjectAnimator;
import android.animation.ValueAnimator;
import android.app.Activity;
import android.os.Bundle;
import android.view.Menu;
import android.view.View;

public class MainActivity extends Activity {

    @Override
    protected void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.activity_main);
    }

    @Override
    public boolean onCreateOptionsMenu(Menu menu) {
        getMenuInflater().inflate(R.menu.main, menu);
        return true;
    }

   public void animate1(View source) {
        View view = findViewById(R.id.imageView1);
        ObjectAnimator objectAnimator = ObjectAnimator.ofFloat(
                view, "rotationY", 0F, 720.0F);
        objectAnimator.setDuration(2000);
        objectAnimator.start();
```

```
    }

    public void animate2(View source) {
        final View view = findViewById(R.id.imageView1);
        ValueAnimator valueAnimator = ValueAnimator.ofFloat(0F,
                7200F);
        valueAnimator.setDuration(15000);

        valueAnimator.addUpdateListener(new
                ValueAnimator.AnimatorUpdateListener() {
            @Override
            public void onAnimationUpdate(ValueAnimator animation)
       {
                Float value = (Float) animation.getAnimatedValue();
                view.setRotationX(value);
                if (value < 3600) {
                    view.setTranslationX(value/20);
                    view.setTranslationY(value/20);
                } else {
                    view.setTranslationX((7200-value)/20);
                    view.setTranslationY((7200-value)/20);
                }
            }
        });
        valueAnimator.start();
    }
    public void animate3(View source) {
        View view = findViewById(R.id.imageView1);
        ObjectAnimator objectAnimator1 =
                ObjectAnimator.ofFloat(view, "translationY", 0F,
                        300.0F);
        ObjectAnimator objectAnimator2 =
                ObjectAnimator.ofFloat(view, "translationX", 0F,
                        300.0F);
        objectAnimator1.setDuration(2000);
        objectAnimator2.setDuration(2000);
        AnimatorSet animatorSet = new AnimatorSet();
        animatorSet.playTogether(objectAnimator1, objectAnimator2);

        ObjectAnimator objectAnimator3 =
                ObjectAnimator.ofFloat(view, "rotation", 0F,
                        1440F);
        objectAnimator3.setDuration(4000);
        animatorSet.play(objectAnimator3).after(objectAnimator2);
        animatorSet.start();
    }
}
```

Run the application and click the buttons to play the animations. Figure 36.1 shows the application.

Summary

In this chapter you learned about the new Animation API in Android, the Property

Animation system. In particular, you learned about the **android.animation.Animator** class and its subclasses, **ValueAnimator** and **ObjectAnimator**. You also learned to use the **AnimatorSet** class to perform multiple animations.

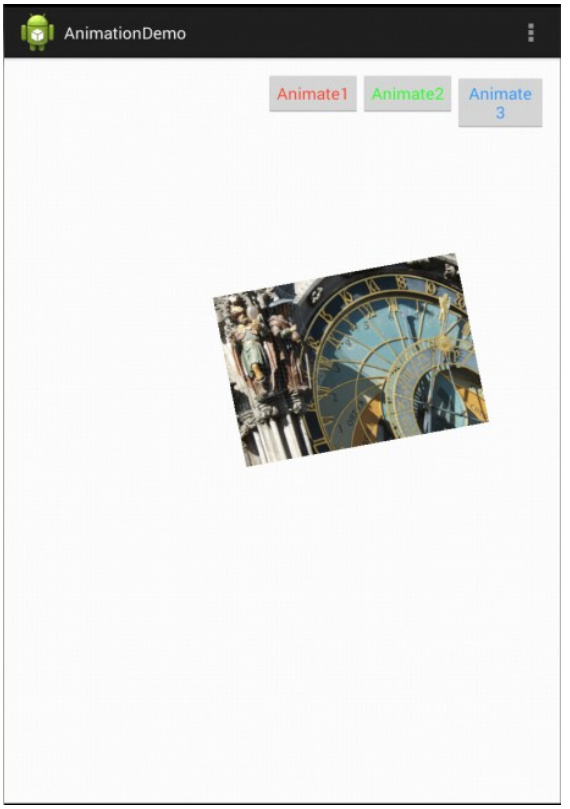

Figure 36.1: Animation demo

Chapter 37
Preferences

Android comes with the **SharedPreferences** interface that can be used to manage application settings as key/value pairs. **SharedPreferences** also takes care of the writing of data to a file. In addition, Android provides the Preference API with user interface (UI) classes that are linked to the default **SharedPreferences** instance so that you can easily create a UI for modifying application settings.

This chapter discusses **SharedPreferences** and the Preference API in detail.

SharedPreferences

The **android.content.SharedPreferences** interface provides methods for storing and reading application settings. You can obtain the default instance of **SharedPreferences** by calling the **getDefaultSharedPreferences** static method of **PreferenceManager**, passing a **Context**.

```
PreferenceManager. getDefaultSharedPreferences(context);
```

To read a value from the **SharedPreferences**, use one of the following methods.

```
public int getInt(java.lang.String key, int default)
public boolean getBoolean(java.lang.String key, boolean default)
public float getFloat(java.lang.String key, float default)
public long getLong(java.lang.String key, long default)
public int getString(java.lang.String key, java.lang.String default)
public java.util.Set<java.lang.String> getStringSet(
    java.lang.String key, java.util.Set<java.lang.String> default)
```

The **get*XXX*** methods return the value associated with the specified key if the pair exists. Otherwise, it returns the specified default value.

To first check if a **SharedPreferences** contains a key/value pair, use the **contains** method, which returns **true** if the specified key exists.

```
public boolean contains(java.lang.String key)
```

On top of that, you can use the **getAll** method to get all key-value pairs as a **Map**.

```
public java.util.Map<java.lang.String, ?> getAll()
```

Values stored in a **SharedPreferences** are persisted automatically and will survive user sessions. The values will be deleted when the application is uninstalled.

The Preference API

To store a key-value pair in a **SharedPreferences**, you normally use the Android Preference API to create a user interface that the user can use to edit settings. The **android.preference.Preference** class is the main class for this. Some of its subclasses are

- **CheckBoxPreference**
- **EditTextPreference**
- **ListPreference**
- **DialogPreference**

An instance of a **Preference** subclass corresponds to a setting.

You could create a **Preference** at runtime, but the best way to create one is by using an XML file to lay out your preferences and then use a **PreferenceFragment** to load the XML file. The XML file must have a **PreferenceScreen** root element and is commonly named **preferences.xml** and should be saved to an **xml** directory under **res**.

Note

Prior to Android 3.0, the **PreferenceActivity** was often used to load a preference xml file. This class is now deprecated and should not be used. Use **PreferenceFragment**, instead.

You will learn how to use **Preference** in the following example.

Using Preferences

The PreferenceDemo1 application shows you how you can use **SharedPreferences** and the Preference API. It has two activities. The first activity shows the values of three application settings by reading them when the activity is resumed. The second activity contains a **PreferenceFragment** that allows the user to change each of the settings.

Figures 37.1 and 37.2 show the main activity and the second activity, respectively.

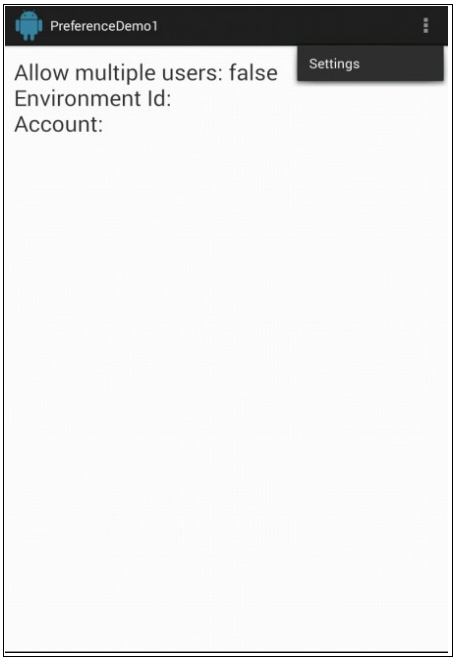

Figure 37.1: The main activity of PreferenceDemo1

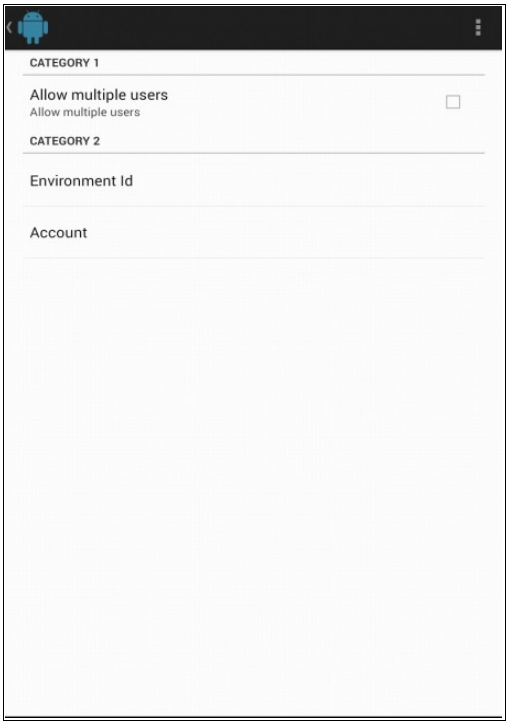

Figure 37.2: The SettingsActivity activity

The **AndroidManifest.xml** file for the application, which describes the two

activities, is shown in Listing 37.1.

Listing 37.1: The AndroidManifest.xml file

```
<?xml version="1.0" encoding="utf-8"?>
<manifest
       xmlns:android="http://schemas.android.com/apk/res/android"
    package="com.example.preferencedemo1"
    android:versionCode="1"
    android:versionName="1.0" >

    <uses-sdk
        android:minSdkVersion="19"
        android:targetSdkVersion="19" />

    <application
        android:allowBackup="true"
        android:icon="@drawable/ic_launcher"
        android:label="@string/app_name"
        android:theme="@style/AppTheme" >
        <activity
            android:name="com.example.preferencedemo1.MainActivity"
            android:label="@string/app_name">
            <intent-filter>
                <action android:name="android.intent.action.MAIN"/>
                <category
android:name="android.intent.category.LAUNCHER" />
            </intent-filter>
        </activity>
        <activity

       android:name="com.example.preferencedemo1.SettingsActivity"
            android:parentActivityName=".MainActivity"
            android:label="">
        </activity>
    </application>
</manifest>
```

The first activity has a very simple layout that features a sole **TextView** as is presented by the **activity_main.xml** file in Listing 37.2.

Listing 37.2: The layout file for the first activity (activity_main.xml)

```
<RelativeLayout
       xmlns:android="http://schemas.android.com/apk/res/android"
    android:layout_width="match_parent"
    android:layout_height="match_parent"
    android:paddingBottom="@dimen/activity_vertical_margin"
    android:paddingLeft="@dimen/activity_horizontal_margin"
    android:paddingRight="@dimen/activity_horizontal_margin"
    android:paddingTop="@dimen/activity_vertical_margin">
    <TextView
        android:id="@+id/info"
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:textSize="30sp"/>
</RelativeLayout>
```

The **MainActivity** class, shown in Listing 37.3, is the activity class for the first

activity. It reads three settings from the default **SharedPreferences** in its **onResume** method and displays the values in the **TextView**.

Listing 37.3: The MainActivity class

```
package com.example.preferencedemo1;
import android.app.Activity;
import android.content.Intent;
import android.content.SharedPreferences;
import android.os.Bundle;
import android.preference.PreferenceManager;
import android.view.Menu;
import android.view.MenuItem;
import android.widget.TextView;

public class MainActivity extends Activity {

    @Override
    protected void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.activity_main);
    }


    @Override
    public void onResume() {
        super.onResume();
        SharedPreferences sharedPref = PreferenceManager.
                getDefaultSharedPreferences(this);
        boolean allowMultipleUsers = sharedPref.getBoolean(
                SettingsActivity.ALLOW_MULTIPLE_USERS, false);
        String envId = sharedPref.getString(
                SettingsActivity.ENVIRONMENT_ID, "");
        String account = sharedPref.getString(
                SettingsActivity.ACCOUNT, "");
        TextView textView = (TextView) findViewById(R.id.info);
        textView.setText("Allow multiple users: " +
                allowMultipleUsers + "\nEnvironment Id: " + envId
                + "\nAccount: " + account);
    }

    @Override
    public boolean onCreateOptionsMenu(Menu menu) {
        getMenuInflater().inflate(R.menu.main, menu);
        return true;
    }

    @Override
    public boolean onOptionsItemSelected(MenuItem item) {
        switch (item.getItemId()) {
            case R.id.action_settings:
                startActivity(new Intent(this,
                        SettingsActivity.class));
                return true;
            default:
                return super.onOptionsItemSelected(item);
```

```
        }
    }
}
<RelativeLayout
```

In addition, the **MainActivity** class overrides the **onCreateOptionsMenu** and **onOptionsItemSelected** methods so that a Settings action appears on the action bar and clicking it will start the second activity, **SettingsActivity**.

SettingsActivity, presented in Listing 37.4, contains a default layout that is replaced by an instance of **SettingsFragment** when the activity is created. Pay attention to the **onCreate** method of the class. If the last lines of code in the method looks foreign to you, please first read Chapter 34, "Fragments."

Listing 37.4: The SettingsActivity class

```
package com.example.preferencedemo1;
import android.app.Activity;
import android.os.Bundle;
import android.view.Menu;

public class SettingsActivity extends Activity {

    public static final String ALLOW_MULTIPLE_USERS =
            "allowMultipleUsers";
    public static final String ENVIRONMENT_ID = "envId";
    public static final String ACCOUNT = "account";

    @Override
    protected void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        getActionBar().setDisplayHomeAsUpEnabled(true);
        getFragmentManager()
                .beginTransaction()
                .replace(android.R.id.content,
                        new SettingsFragment()).commit();
    }

    @Override
    public boolean onCreateOptionsMenu(Menu menu) {
        getMenuInflater().inflate(R.menu.settings, menu);
        return true;
    }
}
```

Note that the **SettingsActivity** class declares three public static finals that define setting keys. The fields are used internally as well as from other classes.

The **SettingsFragment** class is a subclass of **PreferenceFragment**. It is a simple class that simply calls **addPreferencesFromResource** to load the XML document containing the layout for three Preference subclasses. The **SettingsFragment** class is shown in Listing 37.5 and the XML file in Listing 37.6.

Listing 37.5: The SettingsFragment class

```
package com.example.preferencedemo1;
import android.os.Bundle;
import android.preference.PreferenceFragment;

public class SettingsFragment extends PreferenceFragment {
```

```
    @Override
    public void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);

        // Load the preferences from an XML resource
        addPreferencesFromResource(R.xml.preferences);
    }
}
```

Listing 37.6: The res/preferences.xml file

```
<PreferenceScreen
        xmlns:android="http://schemas.android.com/apk/res/android">

    <PreferenceCategory
            android:title="Category 1">
        <CheckBoxPreference
                android:key="allowMultipleUsers"
                android:title="Allow multiple users"
                android:summary="Allow multiple users" />

    </PreferenceCategory>

    <PreferenceCategory
            android:title="Category 2">
        <EditTextPreference
                android:key="envId"
                android:title="Environment Id"
                android:dialogTitle="Environment Id"/>
        <EditTextPreference
                android:key="account"
                android:title="Account"/>
        </PreferenceCategory>
</PreferenceScreen>
```

The **preferences.xml** file groups the **Preference** subclasses into two categories. In the first category is a **CheckBoxPreference** linked to the **allowMultipleUsers** key. In the second category are two **EditTextPreference**s linked to **envId** and **account**.

Summary

An easy way to manage application settings is by using the Preference API and the default **SharedPreferences**. In this chapter you learned how to use both.

Chapter 38
Working with Files

Reading from and writing to a file are some of the most common operations in any type of application, including Android. In this chapter, you will learn how Android structures its storage areas. You will also learn to use the Java File API in an Android application.

Overview

Android devices offer two storage areas, internal and external. The internal storage is private to the application. The user and other applications cannot access it.

The external storage is where you store files that will be shared with other applications or accessible to the user. For example, the built-in Camera application stores digital image files in the external storage so the user can easily copy them.

Internal Storage

All applications can read from and write to internal storage. The location of the internal storage is **/data/data/[*app package*]**, so if your application package is **com.example.myapp**, the internal directory for this application is **/data/data/com.example.myapp**. The **Context** class provides various methods to access the internal storage from your application. You should use these methods to access files you store in the internal storage and should not hardcode the location of the internal storage. (Recall that **Activity** is a subclass of **Context**, so you can call public and protected methods in **Context** from your activity class). Here are methods in **Context** for working with files and streams in the internal storage.

`public java.io.File getFilesDir()`
: Returns the path to the directory dedicated to your application in internal storage.

`public java.io.FileOutputStream openFileOutput(`
`java.lang.String` *`name`*`, int` *`mode`*`)`
: Opens a **FileOutputStream** in the application's section of internal storage.

`public java.io.FileInputStream openFileInput(java.lang.String` *`name`*`)`
: Opens a **FileInputStream** for reading. The name argument is the name of the file to open and cannot contain path separators.

`public java.io.File getFilesDir()`
: Obtains the absolute path to the file system directory where internal files are saved.

`public java.io.File getDir(java.lang.String` *`name`*`, int` *`mode`*`)`
: Creates or retrieves an existing directory within the application's internal storage space. The **name** argument is the name of the directory to retrieve and the **mode** argument should be given one of these:

MODE_PRIVATE for the default operation or **MODE_WORLD_READABLE** or **MODE_WORLD_WRITEABLE** to control permissions.

`public boolean deleteFile(java.lang.String fileName)`
Deletes a file saved on the internal storage. The method returns **true** if the file was successfully deleted.

`public java.lang.String[] FileList()`
Returns an array of strings naming the files associated with this Context's application package.

External Storage

There are two types of files that can be written to external storage, private files and public files. Private files are private to the application and will be deleted when the application is uninstalled. Public files, on the other hand, are meant to be shared with other applications or accessible to the user.

External storage may be removable. As such, there is a difference between files stored in internal storage and files stored in external storage as public files. Files in internal storage are secure and cannot be accessed by the user or other applications. Public files in external storage do not enjoy the same level of security as the user can remove the storage and use some tool to access the files.

Since external storage can be removed, when you try to read from or write to it, you should first test if the external storage is available. Trying to access external storage when it is unavailable may crash your application.

To inquire if external storage is available, use one of these methods.

```
public boolean isExternalStorageWritable() {
    String state = Environment.getExternalStorageState();
    return Environment.MEDIA_MOUNTED.equals(state);
}

public boolean isExternalStorageReadable() {
    String state = Environment.getExternalStorageState();
    return (Environment.MEDIA_MOUNTED.equals(state) ||
            Environment.MEDIA_MOUNTED_READ_ONLY.equals(state));
}
```

You can use the **getExternalFilesDir** method on the **Context** to get the directory for storing private files on the external storage.

Public files can be stored in the directory returned by the **getExternalStoragePublicDirectory** method of the **android.os.Environment** class. Here is the signature of this method.

```
public static java.io.File getExternalStoragePublicDirectory(
        java.lang.String type)
```

Here, *type* is a directory under the root directory. The **Environment** class provides the following fields that you can use for various file types.

- **Directory_ALARMS**
- **Directory_DCIM**
- **Directory_DOCUMENTS**
- **Directory_DOWNLOADS**
- **Directory_MOVIES**

- **Directory_MUSIC**
- **Directory_NOTIFICATIONS**
- **Directory_PICTURES**
- **Directory_PODCASTS**
- **Directory_RINGTONES**

For example, music files should be stored in the directory returned by this code.

```
File dir = new File(Environment.getExternalStoragePublicDirectory(
        Environment.DIRECTORY_PICTURES))
```

Writing to external storage requires user permission. To ask the user to grant you read and write access to external storage, write this in your manifest.

```
<uses-permission
        android:name="android.permission.WRITE_EXTERNAL_STORAGE"/>
```

Currently if your application only needs to read from external storage, you do not need special permissions. However, this will change in the future so you should declare this **uses-permission** element in your manifest if you need to read from external storage so that your application will keep working after the change takes effect.

```
<uses-permission
        android:name="android.permission.READ_EXTERNAL_STORAGE"/>
```

Creating a Notes Application

The FileDemo1 application is a simple application for managing notes. A note has a title and a body and each note is stored as a file, using the title as the file name. The user can view a list of notes, view a note, create a new note, and delete a note.

The application has two activities, **MainActivity** and **AddNoteActivity**. The **MainActivity** activity uses a **ListView** that lists all note titles in the system. The main activity contains a **ListView** that lists all note titles. Selecting a note title from the **ListView** shows the note in the **TextView** beside the **ListView**.

Figures 38.1 and 38.2 show the **MainActivity** activity and **AddNoteActivity** activity, respectively. The **MainActivity** activity contains a **ListView** that lists all note titles and a **TextView** that shows the body of the selected note. Its action bar also contains two buttons, Add and Delete. Add starts the **AddNoteActivity** activity. Delete deletes the selected note.

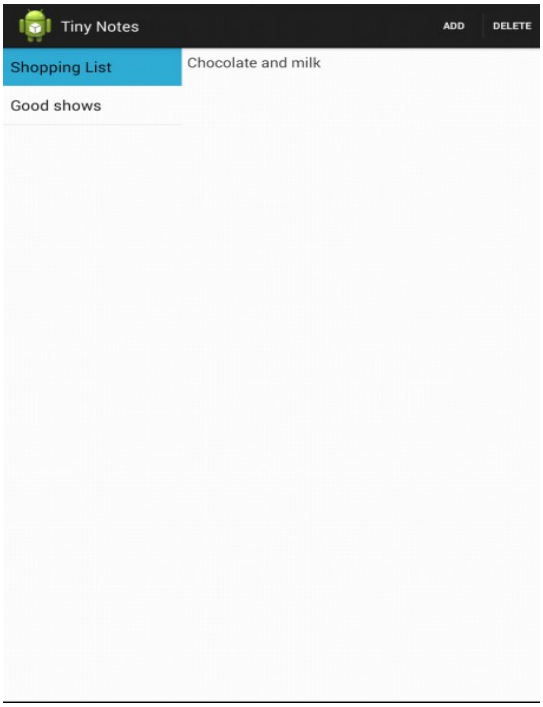

Figure 38.1: MainActivity

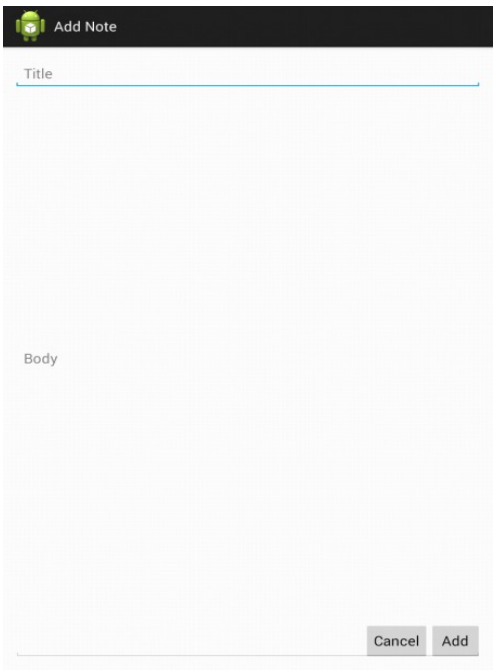

Figure 38.2: AddNoteActivity

Let's now take a look at the application code. Listing 38.1 shows the

AndroidManifest.xml file for this application.

Listing 38.1: The AndroidManifest.xml file

```
<?xml version="1.0" encoding="utf-8"?>
<manifest
       xmlns:android="http://schemas.android.com/apk/res/android"
    package="com.example.filedemo1"
    android:versionCode="1"
    android:versionName="1.0" >

    <uses-sdk
        android:minSdkVersion="19"
        android:targetSdkVersion="19" />

    <application
        android:allowBackup="true"
        android:icon="@drawable/ic launcher"
        android:label="@string/app name"
        android:theme="@style/AppTheme" >
        <activity
            android:name="com.example.filedemo1.MainActivity"
            android:label="@string/app_name" >
            <intent-filter>
                <action android:name="android.intent.action.MAIN"/>
                <category
android:name="android.intent.category.LAUNCHER"/>
            </intent-filter>
        </activity>
        <activity
            android:name="com.example.filedemo1.AddNoteActivity"
            android:label="@string/title_activity_add_note" >
        </activity>
    </application>
</manifest>
```

The manifest declares the two activities in the application. The activity class of the main activity, **MainActivity**, is shown in Listing 38.2.

Listing 38.2: The MainActivity class

```
package com.example.filedemo1;
import java.io.BufferedReader;
import java.io.File;
import java.io.FileReader;
import java.io.IOException;
import android.app.Activity;
import android.content.Intent;
import android.os.Bundle;
import android.view.Menu;
import android.view.MenuItem;
import android.view.View;
import android.widget.AdapterView;
import android.widget.AdapterView.OnItemClickListener;
import android.widget.ArrayAdapter;
import android.widget.ListView;
import android.widget.TextView;
```

```
public class MainActivity extends Activity {
    private String selectedItem;

    @Override
    protected void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.activity main);
        ListView listView = (ListView) findViewById(
                R.id.listView1);
        listView.setChoiceMode(ListView.CHOICE_MODE_SINGLE);
        listView.setOnItemClickListener(
                new OnItemClickListener() {
            @Override
            public void onItemClick(AdapterView<?> adapterView,
                    View view, int position, long id) {
                readNote(position);
            }
        });
    }

    @Override
    public void onResume() {
        super.onResume();
        refreshList();
    }

    @Override
    public boolean onCreateOptionsMenu(Menu menu) {
        getMenuInflater().inflate(R.menu.main, menu);
        return true;
    }
    @Override
    public boolean onOptionsItemSelected(MenuItem item) {
        // Handle presses on the action bar items
        switch (item.getItemId()) {
            case R.id.action add:
                startActivity(new Intent(this,
                        AddNoteActivity.class));
                return true;
            case R.id.action delete:
                deleteNote();
                return true;
            default:
                return super.onOptionsItemSelected(item);
        }
    }

    private void refreshList() {
        ListView listView = (ListView) findViewById(
                R.id.listView1);
        String[] titles = fileList();
        ArrayAdapter<String> arrayAdapter =
                new ArrayAdapter<String>(
                this,
                android.R.layout.simple_list_item_activated_1,
```

```
                titles);
        listView.setAdapter(arrayAdapter);
    }

    private void readNote(int position) {
        String[] titles = fileList();
        if (titles.length > position) {
            selectedItem = titles[position];
            File dir = getFilesDir();
            File file = new File(dir, selectedItem);
            FileReader fileReader = null;
            BufferedReader bufferedReader = null;
            try {
                fileReader = new FileReader(file);
                bufferedReader = new BufferedReader(fileReader);
                StringBuilder sb = new StringBuilder();
                String line = bufferedReader.readLine();
                while (line != null) {
                    sb.append(line);
                    line = bufferedReader.readLine();
                }
                ((TextView) findViewById(R.id.textView1)).
                        setText(sb.toString());
            } catch (IOException e) {

            } finally {
                if (bufferedReader != null) {
                    try {
                        bufferedReader.close();
                    } catch (IOException e) {
                    }
                }
                if (fileReader != null) {
                    try {
                        fileReader.close();
                    } catch (IOException e) {
                    }
                }
            }
        }
    }

    private void deleteNote() {
        if (selectedItem != null) {
            deleteFile(selectedItem);
            selectedItem = null;
            ((TextView) findViewById(R.id.textView1)).setText("");
            refreshList();
        }
    }
}
```

The **MainActivity** class contains a **ListView** and its **onCreate** method sets the **ListView**'s choice mode and passes a listener to it.

```
        ListView listView = (ListView) findViewById(
```

```
            R.id.listView1);
        listView.setChoiceMode(ListView.CHOICE_MODE_SINGLE);
        listView.setOnItemClickListener( ... )
```

No list adapter is passed to the **ListView** in the **onCreate** method. Instead, the **onResume** method calls the **refreshNotes** method, which passes a new **ListAdapter** to the **ListView** every time **onResume** is called. The reason why a new **ListAdapter** needs to be created every time **onResume** is called is because the main activity can call the **AddNoteActivity** for the user to add a note. If the user does add a note and leave the **AddNoteActivity**, the main activity needs to include the new note, hence the need to refresh the **ListView**.

Note

Automatic refresh in a **ListView** can be done using a Cursor. See Chapter 39, "Working with the Database."

The **readNote** method, which gets called when a list item is selected, starts by getting all file names in the internal storage.

```
String[] titles = fileList();
```

It then retrieves the note title and uses it to create a file, using the directory returned by **getFilesDir** as the parent.

```
if (titles.length > position) {
    selectedItem = titles[position];
    File dir = getFilesDir();
    File file = new File(dir, selectedItem);
```

The **readNote** method then uses a **FileReader** and a **BufferedReader** to read the note, one line at a time, and sets the value of the **TextView**.

```
        FileReader fileReader = null;
        BufferedReader bufferedReader = null;
        try {
            fileReader = new FileReader(file);
            bufferedReader = new BufferedReader(fileReader);
            StringBuilder sb = new StringBuilder();
            String line = bufferedReader.readLine();
            while (line != null) {
                sb.append(line);
                line = bufferedReader.readLine();
            }
            ((TextView) findViewById(R.id.textView1)).
                    setText(sb.toString());
```

The **AddNoteActivity** class is shown in Listing 38.3.

Listing 38.3: The AddNoteActivity class

```
package com.example.filedemo1;
import java.io.File;
import java.io.PrintWriter;
import android.app.Activity;
import android.app.AlertDialog;
import android.os.Bundle;
import android.view.View;
import android.widget.EditText;

public class AddNoteActivity extends Activity {
```

```
    @Override
    protected void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.activity_add_note);
    }

    public void cancel(View view) {
        finish();
    }

    public void addNote(View view) {
        String fileName = ((EditText)
                findViewById(R.id.noteTitle))
                .getText().toString();
        String body = ((EditText) findViewById(R.id.noteBody))
                .getText().toString();
        File parent = getFilesDir();
        File file = new File(parent, fileName);
        PrintWriter writer = null;
        try {
            writer = new PrintWriter(file);
            writer.write(body);
            finish();
        } catch (Exception e) {
            showAlertDialog("Error adding note", e.getMessage());
        } finally {
            if (writer != null) {
                try {
                    writer.close();
                } catch (Exception e) {

                }
            }
        }
    }

    private void showAlertDialog(String title, String message) {
        AlertDialog alertDialog = new
                AlertDialog.Builder(this).create();
        alertDialog.setTitle(title);
        alertDialog.setMessage(message);
        alertDialog.show();
    }
}
```

The **AddNoteActivity** class has two public methods that act as click listeners to the two buttons in its layout, cancel and addNote. The **cancel** method simply closes the activity. The **addNote** method reads the values in the **TextView**s and create a file in the internal storage using a **PrintWriter**.

Accessing the Public Storage

The second example in this chapter, FileDemo2, shows how you can access the public storage. FileDemo2 is a file browser that shows the content of a standard

directory. There is only one activity in FileDemo2 and it is shown in Figure 38.3.

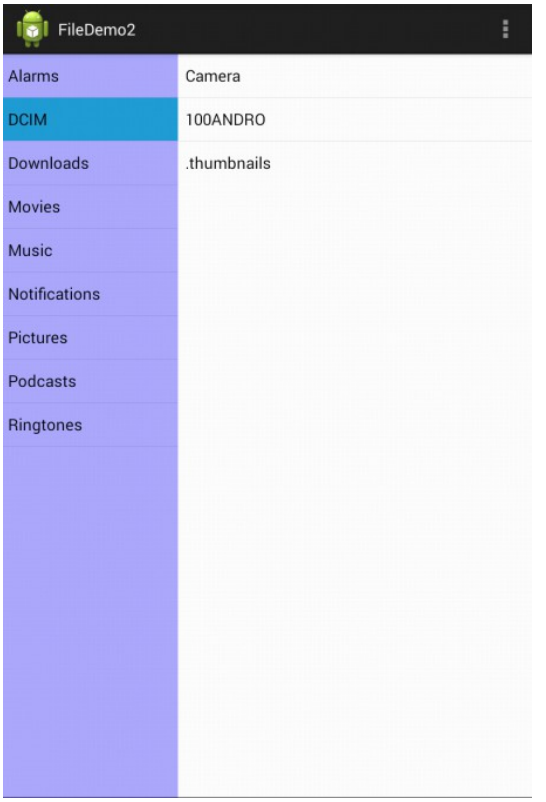

Figure 38.3: FileDemo2

The layout file for the activity is presented in Listing 38.4. It is a **LinearLayout** that contains two **ListView**s. The **ListView** on the left lists several frequently used directories. The **ListView** on the right shows the content of the selected directory.

Listing 38.4: The layout file for FileDemo2's activity

```
<LinearLayout
       xmlns:android="http://schemas.android.com/apk/res/android"
    android:layout_width="match_parent"
    android:layout_height="match_parent"
    android:orientation="horizontal">
    <ListView
        android:id="@+id/listView1"
        android:layout_width="0sp"
        android:layout_weight="1"
        android:layout_height="match_parent"
        android:background="#ababff"/>
    <ListView
        android:id="@+id/listView2"
        android:layout_width="0sp"
        android:layout_height="wrap_content"
        android:layout_weight="2"/>
</LinearLayout>
```

The activity class for FileDemo2 is shown in Listing 38.5.

Listing 38.5: The MainActivity class

```
package com.example.filedemo2;
import java.io.File;
import java.util.Arrays;
import java.util.List;

import android.app.Activity;
import android.os.Bundle;
import android.os.Environment;
import android.view.Menu;
import android.view.View;
import android.widget.AdapterView;
import android.widget.AdapterView.OnItemClickListener;
import android.widget.ArrayAdapter;
import android.widget.ListView;

public class MainActivity extends Activity {
    class KeyValue {
        public String key;
        public String value;
        public KeyValue(String key, String value) {
            this.key = key;
            this.value = value;
        }
        @Override
        public String toString() {
            return key;
        }
    }

    @Override
    protected void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.activity_main);
        final List<KeyValue> keyValues = Arrays.asList(
            new KeyValue("Alarms", Environment.DIRECTORY_ALARMS),
            new KeyValue("DCIM", Environment.DIRECTORY_DCIM),
            new KeyValue("Downloads",
                    Environment.DIRECTORY_DOWNLOADS),
            new KeyValue("Movies", Environment.DIRECTORY_MOVIES),
            new KeyValue("Music", Environment.DIRECTORY_MUSIC),
            new KeyValue("Notifications",
                    Environment.DIRECTORY_NOTIFICATIONS),
            new KeyValue("Pictures",
                    Environment.DIRECTORY_PICTURES),
            new KeyValue("Podcasts",
                    Environment.DIRECTORY_PODCASTS),
            new KeyValue("Ringtones",
                    Environment.DIRECTORY_RINGTONES)
        );
        ArrayAdapter<KeyValue> arrayAdapter = new
                ArrayAdapter<KeyValue>(this,
                    android.R.layout.simple_list_item_activated_1,
                    keyValues);
        ListView listView1 = (ListView)
```

```
                findViewById(R.id.listView1);
        listView1.setChoiceMode(ListView.CHOICE_MODE_SINGLE);
        listView1.setAdapter(arrayAdapter);
        listView1.setOnItemClickListener(new
                OnItemClickListener() {
            @Override
            public void onItemClick(AdapterView<?> adapterView,
                    View view, int position, long id) {
                KeyValue keyValue = keyValues.get(position);
                listDir(keyValue.value);
            }
        });
    }

    @Override
    public boolean onCreateOptionsMenu(Menu menu) {
        getMenuInflater().inflate(R.menu.main, menu);
        return true;
    }

    private void listDir(String dir) {
        File parent = Environment
                .getExternalStoragePublicDirectory(dir);
        String[] files = null;
        if (parent == null || parent.list() == null) {
            files = new String[0];
        } else {
            files = parent.list();
        }
        ArrayAdapter<String> arrayAdapter = new
                ArrayAdapter<String>(this,
                    android.R.layout.simple_list_item_activated_1,
                    files);
        ListView listView2 = (ListView)
                findViewById(R.id.listView2);
        listView2.setAdapter(arrayAdapter);
    }
}
```

The first thing to note is the **KeyValue** class in the activity class. This is a simple class to hold a pair of strings. It is used in the **onCreate** method to pair selected keys with directories defined in the **Environment** class.

```
            new KeyValue("Alarms", Environment.DIRECTORY_ALARMS),
            new KeyValue("DCIM", Environment.DIRECTORY_DCIM),
            new KeyValue("Downloads",
                    Environment.DIRECTORY_DOWNLOADS),
            new KeyValue("Movies", Environment.DIRECTORY_MOVIES),
            new KeyValue("Music", Environment.DIRECTORY_MUSIC),
            new KeyValue("Notifications",
                    Environment.DIRECTORY_NOTIFICATIONS),
            new KeyValue("Pictures",
                    Environment.DIRECTORY_PICTURES),
            new KeyValue("Podcasts",
                    Environment.DIRECTORY_PODCASTS),
            new KeyValue("Ringtones",
```

```
                Environment.DIRECTORY_RINGTONES)
```

These **KeyValue** instances are then used to feed the first **ListView**.

```
        ArrayAdapter<KeyValue> arrayAdapter = new
                ArrayAdapter<KeyValue>(this,
                    android.R.layout.simple_list_item_activated_1,
                    keyValues);
        ListView listView1 = (ListView)
                findViewById(R.id.listView1);
        listView1.setChoiceMode(ListView.CHOICE_MODE_SINGLE);
        listView1.setAdapter(arrayAdapter);
```

The **ListView** also gets a listener that listens for its **OnItemClick** event and calls the **listDir** method when one of the directories in the **ListView** is selected.

```
        listView1.setOnItemClickListener(new
                OnItemClickListener() {
            @Override
            public void onItemClick(AdapterView<?> adapterView,
                    View view, int position, long id) {
                KeyValue keyValue = keyValues.get(position);
                listDir(keyValue.value);
            }
        });
```

The **listDir** method list all files in the selected directory and feed them to an **ArrayAdapter** that in turn gets passed to the second **ListView**.

```
    private void listDir(String dir) {
        File parent = Environment
                .getExternalStoragePublicDirectory(dir);
        String[] files = null;
        if (parent == null || parent.list() == null) {
            files = new String[0];
        } else {
            files = parent.list();
        }
        ArrayAdapter<String> arrayAdapter = new
                ArrayAdapter<String>(this,
                    android.R.layout.simple_list_item_activated_1,
                    files);
        ListView listView2 = (ListView)
                findViewById(R.id.listView2);
        listView2.setAdapter(arrayAdapter);
    }
```

Summary

You use the Java File API to work with files in Android applications. In addition to mastering this API, in order to work with files effectively in Android, you need to know how Android structures its storage system and the file-related methods defined in the **Context** and **Environment** classes.

Chapter 39
Working with the Database

Android has its own technology for working with databases and it has nothing to do with Java Database Connectivity (JDBC), the technology Java developers use for accessing data in a relational database. In addition, Android ships with SQLite, an open source database.

This chapter shows how to work with the Android Database API and the SQLite database.

Overview

Android comes with its own Database API. The API consists of two packages, **android.database** and **android.database.sqlite**. Android ships with SQLite, an open source relational database that partially implements SQL-92, the third revision of the SQL standard.

Currently at version 3, SQLite offers a minimum number of data types: Integer, Real, Text, Blob, and Numeric. One interesting feature of SQLite is that an integer primary key is automatically auto-incremented when a row is inserted without passing a value for the field.

More information on SQLite can be found here:

`http://sqlite.org`

The Database API

The **SQLiteDatabase** and **SQLiteOpenHelper** classes, both part of **android.database.sqlite**, are the two most frequently used classes in the Database API. In the **android.database** package, the **Cursor** interface is one of the most important types.

The three types are discussed in detail in the following subsections.

The SQLiteOpenHelper Class

To use a database in your Android application, extend **SQLiteOpenHelper** to help with database and table creation as well as connecting to the database. In a subclass of **SQLiteOpenHelper**, you need to do the following.

- provide a constructor that calls its super, passing, among others, the **Context** and the database name.
- override the **onCreate** and **onUpgrade** methods.

For example, here is a constructor for a subclass of **SQLiteOpenHelper**.

```
public SubClassOfSQLiteOpenHelper(Context context) {
    super(context,
        "mydatabase", // database name
        null,
        1             // db version
    );
}
```

The **onCreate** method that needs to be overriden has the following signature.

```
public void onCreate(SQLiteDatabase database)
```

The system will call **onCreate** the first time access to one of the tables is required. In this method implementation you should call the **execSQL** method on the **SQLiteDatabase** and pass an SQL statement for creating your table(s). Here is an example.

```
@Override
public void onCreate(SQLiteDatabase db) {
    String sql = "CREATE TABLE " + TABLE_NAME
        + " (" + ID_FIELD + " INTEGER, "
        + FIRST_NAME_FIELD + " TEXT,"
        + LAST_NAME_FIELD + " TEXT,"
        + PHONE_FIELD + " TEXT,"
        + EMAIL_FIELD + " TEXT,"
        + " PRIMARY KEY (" + ID_FIELD + "));";
     db.execSQL(sql);
}
```

SQLiteOpenHelper automatically manages connections to the underlying database. To retrieve the database instance, call one of these methods, both of which return an instance of **SQLiteDatabase**.

```
public SQLiteDatabase getReadableDatabase()
```

```
public SQLiteDatabase getWritableDatabase()
```

The first time one of these methods is called a database will be created if none exists. The difference between **getReadableDatabase** and **getWritableDatabase** is the former can be used for read-only whereas the latter can be used to read from and write to the database.

The SQLiteDatabase Class

Once you get a **SQLiteDatabase** from a **SQLiteOpenHelper**'s **getReadableDatabase** or **getWritableDatabase** method, you can manipulate the data in the database by calling the **SQLiteDatabase**'s **insert** or **execSQL** method. For example, to add a record, call the **insert** method whose signature is as follows.

```
public long insert (String table, String nullColumnHack,
        ContentValues values)
```

Here, *table* is the name of the table and *values* is an **android.content.ContentValues** that contains pairs of field names/values to be inserted to the table. This method returns the row identifier for the new row.

For instance, the following code inserts a record into the **employees** table passing three field values.

```
SQLiteDatabase db = this.getWritableDatabase();
// this is an instance of SQLiteOpenHelper
ContentValues values = new ContentValues();
```

```
values.put("first_name", "Joe");
values.put("last_name", "Average");
values.put("position", "System Analyst");
long id = db.insert("employees", null, values);
db.close();
```

To update or delete a record, use the **update** or **delete** method, respectively. The signatures of these methods are as follows.

```
public int delete (java.lang.String table,
        java.lang.String whereClause, java.lang.String[] whereArgs)
```

```
public int update (java.lang.String table,
        android.content.ContentValues values,
        java.lang.String whereClause, java.lang.String[] whereArgs)
```

Examples of the two methods are shown in the accompanying application.

To execute a SQL statement, use the **execSQL** method.

```
public void execSQL (java.lang.String sql)
```

Finally, to retrieve records, use one of the **query** methods. One of the method overloads has this signature.

```
public android.database.Cursor query(java.lang.String table,
        java.lang.String[] columns, java.lang.String selection,
        java.lang.String[] selectionArgs,
        java.lang.String groupBy,
        java.lang.String having,
        java.lang.String orderBy, hava.lang.String limit)
```

You can find an example on how to use this method in the sample application accompanying this chapter.

One thing to note: The data returned by the query method is contained in an instance of **Cursor**, an interesting type explained in the next section.

The Cursor Interface

Calling the **query** method on a **SQLiteDatabase** returns a **Cursor**. A **Cursor**, an implementation of the **android.database.Cursor** interface, provides read and write access to the result set returned by a database query.

To read a row of data through a **Cursor**, you first need to point the **Cursor** to a data row by calling its **moveToFirst**, **moveToNext**, **moveToPrevious**, **moveToLast**, or **moveToPosition** method. **moveToFirst** moves the **Cursor** to the first row and **moveToNext** to the next row. **moveToLast**, you may have guessed correctly, moves it to the last record and **moveToPrevious** to the previous row. **moveToPosition** takes an integer and moves the **Cursor** to the specified position.

Once you move the **Cursor** to a data row, you can read a column value in the row by calling the **Cursor**'s **getInt**, **getFloat**, **getLong**, **getString**, **getShort**, or **getDouble** method, passing the column index.

An interesting aspect of **Cursor** is that it can be used as the data source for a **ListAdapter**, which in turn can be used to feed a **ListView**. The advantage of using a **Cursor** for a **ListView** is that the **Cursor** can manage your data. In other words, if the data is updated, the **Cursor** can self-refresh the **ListView**. This is a very useful feature as you then have one fewer thing to worry about.

Example

The DatabaseDemo1 application is an application for managing contacts in a SQLite database. A contact is a data structure that contains a person's contact details. The application has three activities, **MainActivity**, **AddContactActivity**, and **ShowContactActivity**.

The main activity shows the list of contacts and is shown in Figure 39.1.

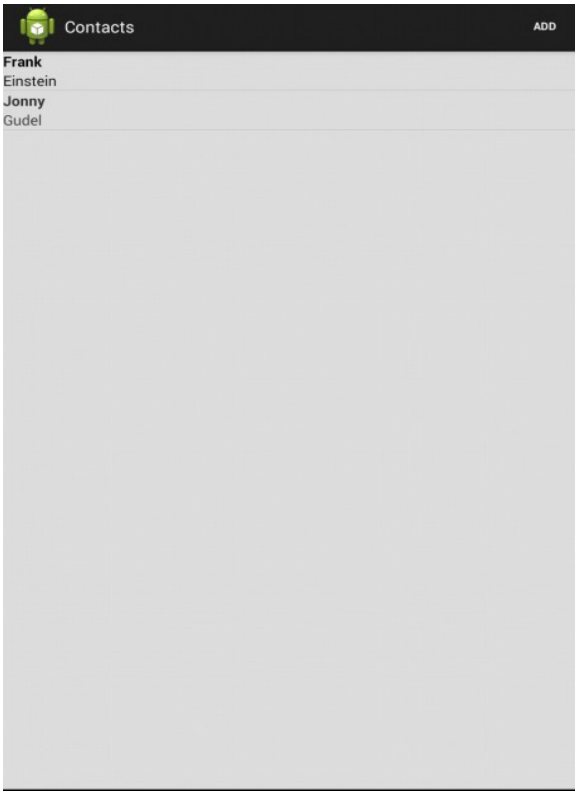

Figure 39.1: The main activity

The main activity offers an Add button on its action bar that will start the **AddContactActivity** activity if pressed. The latter activity contains a form for adding a new contact and is shown in Figure 39.2.

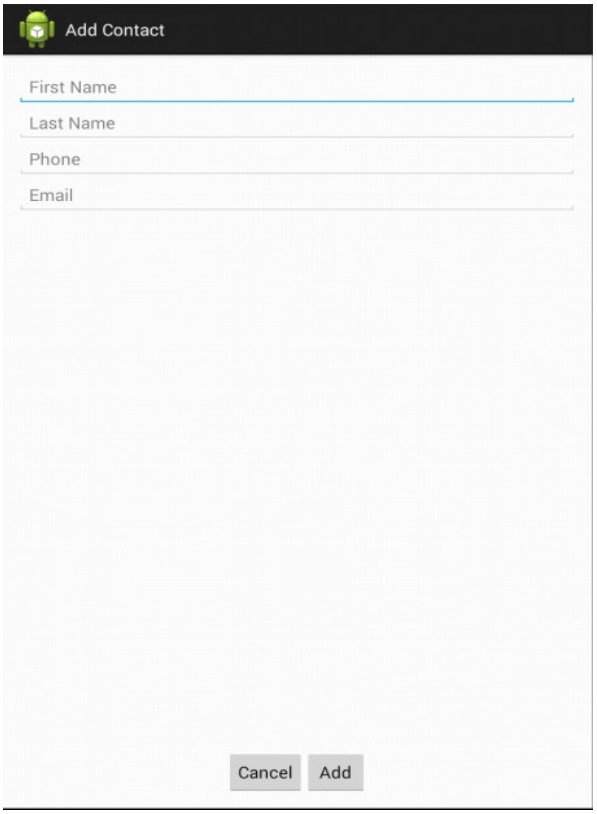

Figure 39.2: AddContactActivity

The main activity also uses a **ListView** to display all contacts in the database. Pressing an item on the list activates the **ShowContactActivity** activity, which is shown in Figure 39.3.

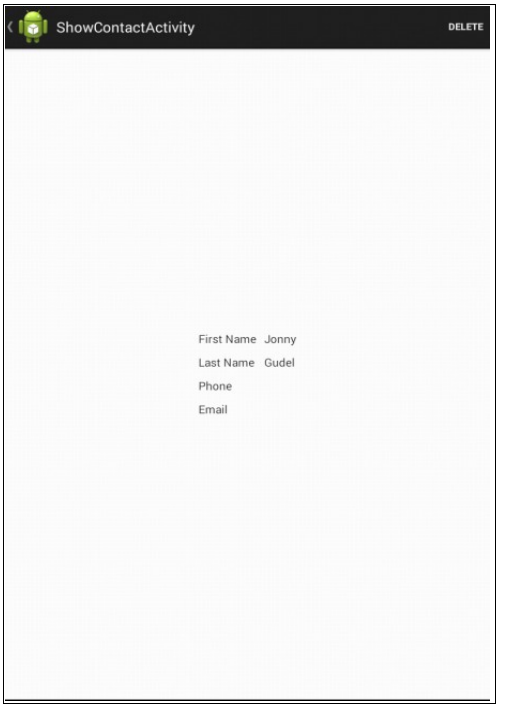

Figure 39.3: ShowContactActivity

The **ShowContactActivity** activity allows the user to delete the shown contact by pressing the Delete button on the action bar. Pressing the button prompts the user to confirm if he or she really wishes to delete the contact. The user can press the activity label to go back to the main activity.

The three activities in the application are declared in the manifest presented in Listing 39.1.

Listing 39.1: The AndroidManifest.xml file

```
<?xml version="1.0" encoding="utf-8"?>
<manifest
       xmlns:android="http://schemas.android.com/apk/res/android"
    package="com.example.databasedemo1"
    android:versionCode="1"
    android:versionName="1.0" >

    <uses-sdk
        android:minSdkVersion="11"
        android:targetSdkVersion="18" />

    <application
        android:allowBackup="true"
        android:icon="@drawable/ic launcher"
        android:label="@string/app name"
        android:theme="@style/AppTheme" >
        <activity
            android:name="com.example.databasedemo1.MainActivity"
            android:label="@string/app_name" >
            <intent-filter>
```

```
                <action
       android:name="android.intent.action.MAIN" />
                <category
android:name="android.intent.category.LAUNCHER"/>
            </intent-filter>
        </activity>
        <activity
android:name="com.example.databasedemo1.AddContactActivity"
            android:parentActivityName=".MainActivity"
            android:label="@string/title_activity_add_contact">
        </activity>
        <activity
android:name="com.example.databasedemo1.ShowContactActivity"
            android:parentActivityName=".MainActivity"
            android:label="@string/title_activity_show_contact" >
        </activity>
    </application>
</manifest>
```

DatabaseDemo1 is a simple application that features one object model, the **Contact** class in Listing 39.2. This is a POJO with five properties, **id**, **firstName**, **lastName**, **phone**, and **email**.

Listing 39.2: The Contact class

```
package com.example.databasedemo1;
public class Contact {
    private long id;
    private String firstName;
    private String lastName;
    private String phone;
    private String email;

    public Contact() {
    }

    public Contact(String firstName, String lastName,
            String phone, String email) {
        this.firstName = firstName;
        this.lastName = lastName;
        this.phone = phone;
        this.email = email;
    }
    // get and set methods not shown to save space
}
```

Now comes the most important class in this application, the **DatabaseManager** class in Listing 39.3. This class encapsulates methods for accessing data in the database. The class extends **SQLiteOpenHelper** and implements its **onCreate** and **onUpdate** methods and provides methods for managing contacts, **addContact**, **deleteContact**, **updateContact**, **getAllContacts**, and **getContact**.

Listing 39.3: The DatabaseManager class

```
package com.example.databasedemo1;
import java.util.ArrayList;
import java.util.List;
import android.content.ContentValues;
import android.content.Context;
```

```
import android.database.Cursor;
import android.database.sqlite.SQLiteDatabase;
import android.database.sqlite.SQLiteOpenHelper;
import android.util.Log;

public class DatabaseManager extends SQLiteOpenHelper {
    public static final String TABLE NAME = "contacts";
    public static final String ID FIELD = " id";
    public static final String FIRST NAME FIELD = "first name";
    public static final String LAST NAME FIELD = "last_name";
    public static final String PHONE FIELD = "phone";
    public static final String EMAIL FIELD = "email";
    public DatabaseManager(Context context) {
        super(context,
                /*db name=*/ "contacts db2",
                /*cursorFactory=*/ null,
                /*db version=*/1);
    }
    @Override
    public void onCreate(SQLiteDatabase db) {
        Log.d("db", "onCreate");
        String sql = "CREATE TABLE " + TABLE NAME
                + " (" + ID FIELD + " INTEGER, "
                + FIRST NAME FIELD + " TEXT,"
                + LAST NAME FIELD + " TEXT,"
                + PHONE FIELD + " TEXT,"
                + EMAIL FIELD + " TEXT,"
                + " PRIMARY KEY (" + ID_FIELD + "));";
        db.execSQL(sql);

    }

    @Override
    public void onUpgrade(SQLiteDatabase db, int arg1, int arg2) {
        Log.d("db", "onUpdate");
        db.execSQL("DROP TABLE IF EXISTS " + TABLE_NAME);
        // re-create the table
        onCreate(db);
    }

    public Contact addContact(Contact contact) {
        Log.d("db", "addContact");
        SQLiteDatabase db = this.getWritableDatabase();
        ContentValues values = new ContentValues();
        values.put(FIRST NAME FIELD, contact.getFirstName());
        values.put(LAST NAME FIELD, contact.getLastName());
        values.put(PHONE FIELD, contact.getPhone());
        values.put(EMAIL FIELD, contact.getEmail());
        long id = db.insert(TABLE_NAME, null, values);
        contact.setId(id);
        db.close();
        return contact;
    }

    // Getting single contact
```

```
Contact getContact(long id) {
    SQLiteDatabase db = this.getReadableDatabase();
    Cursor cursor = db.query(TABLE NAME, new String[] {
            ID FIELD, FIRST NAME FIELD, LAST NAME FIELD,
            PHONE FIELD, EMAIL FIELD }, ID FIELD + "=?",
            new String[] { String.valueOf(id) }, null,
            null, null, null);
    if (cursor != null) {
        cursor.moveToFirst();
        Contact contact = new Contact(
                cursor.getString(1),
                cursor.getString(2),
                cursor.getString(3),
                cursor.getString(4));
        contact.setId(cursor.getLong(0));
        return contact;
    }
    return null;
}

// Getting All Contacts
public List<Contact> getAllContacts() {
    List<Contact> contacts = new ArrayList<Contact>();
    String selectQuery = "SELECT  * FROM " + TABLE_NAME;

    SQLiteDatabase db = this.getWritableDatabase();
    Cursor cursor = db.rawQuery(selectQuery, null);

    while (cursor.moveToNext()) {
        Contact contact = new Contact();
        contact.setId(Integer.parseInt(cursor.getString(0)));
        contact.setFirstName(cursor.getString(1));
        contact.setLastName(cursor.getString(2));
        contact.setPhone(cursor.getString(3));
        contact.setEmail(cursor.getString(4));
        contacts.add(contact);
    }
    return contacts;
}

public Cursor getContactsCursor() {
    String selectQuery = "SELECT  * FROM " + TABLE NAME;
    SQLiteDatabase db = this.getWritableDatabase();
    return db.rawQuery(selectQuery, null);
}

public int updateContact(Contact contact) {
    SQLiteDatabase db = this.getWritableDatabase();

    ContentValues values = new ContentValues();
    values.put(FIRST NAME FIELD, contact.getFirstName());
    values.put(LAST NAME FIELD, contact.getLastName());
    values.put(PHONE FIELD, contact.getPhone());
    values.put(EMAIL_FIELD, contact.getEmail());
```

```
        return db.update(TABLE NAME, values, ID FIELD + " = ?",
                new String[] { String.valueOf(contact.getId()) });
    }

    public void deleteContact(long id) {
        SQLiteDatabase db = this.getWritableDatabase();
        db.delete(TABLE NAME, ID FIELD + " = ?",
                new String[] { String.valueOf(id) });
        db.close();
    }
}
```

The **DatabaseManager** class is used by all the three activity classes. The **MainActivity** class employs a **ListView** that gets its data and layout from a **ListAdapter** that in turn gets its data from a cursor. The **AddContactActivity** class receives the details of a new contact and inserts it into the database by calling the **DatabaseManager** class's **addContact** method. The **ShowContactActivity** class retrieves the details of the pressed contact item in the main activity and uses the **DatabaseManager** class's **getContact** method to achieve this. If the user decides to delete the shown contact, **ShowContactActivity** will resort to **DatabaseManager** to delete it.

The **MainActivity**, **AddContactActivity**, and **ShowContactActivity** classes are given in Listing 39.4, Listing 39.5, and Listing 39.6, respectively.

Listing 39.4: The MainActivity class

```
package com.example.databasedemo1;
import android.app.Activity;
import android.content.Intent;
import android.database.Cursor;
import android.os.Bundle;
import android.support.v4.widget.CursorAdapter;
import android.view.Menu;
import android.view.MenuItem;
import android.view.View;
import android.widget.AdapterView;
import android.widget.AdapterView.OnItemClickListener;
import android.widget.ListAdapter;
import android.widget.ListView;
import android.widget.SimpleCursorAdapter;

public class MainActivity extends Activity {
    DatabaseManager dbMgr;

    @Override
    protected void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.activity main);
        ListView listView = (ListView) findViewById(
                R.id.listView);
        dbMgr = new DatabaseManager(this);

        Cursor cursor = dbMgr.getContactsCursor();
        startManagingCursor(cursor);

        ListAdapter adapter = new SimpleCursorAdapter(
```

```
            this,
            android.R.layout.two_line_list_item,
            cursor,
            new String[] {DatabaseManager.FIRST NAME FIELD,
                    DatabaseManager.LAST NAME FIELD},
            new int[] {android.R.id.text1, android.R.id.text2},
            CursorAdapter.FLAG_REGISTER_CONTENT_OBSERVER);

        listView.setAdapter(adapter);
        listView.setChoiceMode(ListView.CHOICE_MODE_SINGLE);
        listView.setOnItemClickListener(
                new OnItemClickListener() {
            @Override
            public void onItemClick(AdapterView<?> adapterView,
                    View view, int position, long id) {
                Intent intent = new Intent(
                        getApplicationContext(),
                        ShowContactActivity.class);
                intent.putExtra("id", id);
                startActivity(intent);
            }
        });
    }

    @Override
    public boolean onCreateOptionsMenu(Menu menu) {
        getMenuInflater().inflate(R.menu.main, menu);
        return true;
    }

    @Override
    public boolean onOptionsItemSelected(MenuItem item) {
        switch (item.getItemId()) {
            case R.id.action add:
                startActivity(new Intent(this,
                        AddContactActivity.class));
                return true;
            default:
                return super.onOptionsItemSelected(item);
        }
    }
}
```

Listing 39.5: The AddContactActivity class

```
package com.example.databasedemo1;
import android.app.Activity;
import android.os.Bundle;
import android.view.Menu;
import android.view.View;
import android.widget.TextView;

public class AddContactActivity extends Activity {

    @Override
    protected void onCreate(Bundle savedInstanceState) {
```

```
        super.onCreate(savedInstanceState);
        setContentView(R.layout.activity_add_contact);
    }

    @Override
    public boolean onCreateOptionsMenu(Menu menu) {
        getMenuInflater().inflate(R.menu.add_contact, menu);
        return true;
    }

    public void cancel(View view) {
        finish();
    }
    public void addContact(View view) {
        DatabaseManager dbMgr = new DatabaseManager(this);
        String firstName = ((TextView) findViewById(
                R.id.firstName)).getText().toString();
        String lastName = ((TextView) findViewById(
                R.id.lastName)).getText().toString();
        String phone = ((TextView) findViewById(
                R.id.phone)).getText().toString();
        String email = ((TextView) findViewById(
                R.id.email)).getText().toString();
        Contact contact = new Contact(firstName, lastName,
                phone, email);
        dbMgr.addContact(contact);
        finish();
    }
}
```

Listing 39.6: The ShowContactActivity class

```
package com.example.databasedemo1;
import android.app.Activity;
import android.app.AlertDialog;
import android.content.DialogInterface;
import android.os.Bundle;
import android.util.Log;
import android.view.Menu;
import android.view.MenuItem;
import android.widget.TextView;

public class ShowContactActivity extends Activity {
    long contactId;

    @Override
    protected void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.activity show contact);
        getActionBar().setDisplayHomeAsUpEnabled(true);
        Bundle extras = getIntent().getExtras();
        if (extras != null) {
            contactId = extras.getLong("id");
            DatabaseManager dbMgr = new DatabaseManager(this);
            Contact contact = dbMgr.getContact(contactId);
            if (contact != null) {
```

```
            ((TextView) findViewById(R.id.firstName))
                    .setText(contact.getFirstName());
            ((TextView) findViewById(R.id.lastName))
                    .setText(contact.getLastName());
            ((TextView) findViewById(R.id.phone))
                    .setText(contact.getPhone());
            ((TextView) findViewById(R.id.email))
                    .setText(contact.getEmail());
        } else {
            Log.d("db", "contact null");
        }
    }
}

@Override
public boolean onCreateOptionsMenu(Menu menu) {
    getMenuInflater().inflate(R.menu.show_contact, menu);
    return true;
}

@Override
public boolean onOptionsItemSelected(MenuItem item) {
    switch (item.getItemId()) {
    case R.id.action_delete:
        deleteContact();
        return true;
    default:
        return super.onOptionsItemSelected(item);
    }
}

private void deleteContact() {
    new AlertDialog.Builder(this)
        .setTitle("Please confirm")
        .setMessage(
            "Are you sure you want to delete " +
            "this contact?")
        .setPositiveButton("Yes",
            new DialogInterface.OnClickListener() {
                public void onClick(
                        DialogInterface dialog,
                        int whichButton) {
                    DatabaseManager dbMgr =
                            new DatabaseManager(
                            getApplicationContext());
                    dbMgr.deleteContact(contactId);
                    dialog.dismiss();
                    finish();
                }
            })
        .setNegativeButton("No",
            new DialogInterface.OnClickListener() {
                public void onClick(
                        DialogInterface dialog,
                        int which) {
```

```
                        dialog.dismiss();
                    }
                })
            .create()
            .show();
    }
}
```

Summary

The Android Database API makes it easy to work with relational databases. The **android.database** and **android.database.sqlite** packages contains classes and interfaces that support access to a SQLite database, which is the default database shipped with Android. In this chapter you learned how to use the three most frequently used types in the API, the **SQLiteOpenHelper** class, the **SQLiteDatabase** class, and the **Cursor** interface.

Chapter 40
Taking Pictures

Almost all Android handsets and tablets come with one or two cameras. You can use a camera to take still pictures by activating the built-in Camera application or use the Camera API.

This chapter shows how to use both approaches.

Overview

An Android application can call another application to use one or two features offered by the latter. For example, to send an email from your application, you can use the default Email application rather than writing your own app. In the case of taking a picture, the easiest way to do this is by using the Camera application. To activate Camera, use the following code.

```
int requestCode = ...;
Intent intent = new Intent(MediaStore.ACTION_IMAGE_CAPTURE);
startActivityForResult(intent, requestCode);
```

Basically, you need to create an **Intent** by passing **MediaStore.ACTION_IMAGE_CAPTURE** to the **Intent** class's constructor. Then, you need to call **startActivityForResult** from your activity passing the **Intent** and a request code. The request code can be any integer your heart desires. You will learn shortly the purpose of passing a request code.

To tell Camera where to save the taken picture, you can pass a **Uri** to the **Intent**. Here is the complete code.

```
int requestCode = ...;
Intent intent = new Intent(MediaStore.ACTION_IMAGE_CAPTURE);
Uri uri = ...;
intent.putExtra(MediaStore.EXTRA_OUTPUT, uri);
startActivityForResult(intent, requestCode);
```

When the user closes Camera after taking a picture or canceling the operation, Android will notify your application by calling the **onActivityResult** method in the activity that called Camera. This gives you the opportunity to save the picture taken using Camera. The signature of **onActivityResult** is as follows.

```
protected void onActivityResult(int requestCode, int resultCode,
        android.content.Intent data)
```

The system calls **onActivityResult** by passing three arguments. The first argument, **requestCode**, is the request code passed when calling **startActivityForResult**. The request code is important if you call other activities from your activity, passing a different request code each time. Since you can only have one **onActivityResult** implementation in your activity, all calls to

startActivityForResult will share the same **onActivityResult**, and you need to know which activity caused **onActivityResult** to be called by checking the request code.

The second argument to **onActivityResult** is a result code. The value can be either **Activity.RESULT_OK** or **Activity.RESULT_CANCELED** or a user defined value. **Activity.RESULT_OK** indicates that the operation succeeded and **Activity.RESULT_CANCELED** indicates that the operation was canceled.

The third argument to **onActivityResult** contains data from the called activity if the operation was successful.

Using Camera is easy. However, if Camera does not suit your needs, you can also use the Camera API directly. This is not as easy as using Camera, but the API lets you configure many aspects of the camera.

The samples accompanying this chapter show you both methods.

Using Camera

The CameraDemo application shows how to use the built-in intent to activate the Camera application and use it to take a picture. CameraDemo has only one activity, which sports two button on its action bar, Show Camera and Email. The Show Camera button starts Camera and the Email button emails the picture. The application is shown in Figure 40.1.

Let's start dissecting the code, starting from the menu file (the **main.xml** file in Listing 40.1) that contains two menu items for the action bar.

Listing 40.1: The menu file (main.xml)

```
<menu xmlns:android="http://schemas.android.com/apk/res/android">
    <item
        android:id="@+id/action_camera"
        android:orderInCategory="100"
        android:showAsAction="ifRoom"
        android:title="@string/action_show_camera"/>
    <item
        android:id="@+id/action_email"
        android:orderInCategory="200"
        android:showAsAction="ifRoom"
        android:title="@string/action_email"/>
</menu>
```

The layout file for the main activity is presented in Listing 40.2. It contains an **ImageView** for showing the taken picture. The activity class itself is shown in Listing 40.3.

Figure 40.1: CameraDemo

Listing 40.2: The activity_main.xml file

```
<RelativeLayout
       xmlns:android="http://schemas.android.com/apk/res/android"
    android:layout_width="match_parent"
    android:layout_height="match_parent"
    android:paddingBottom="@dimen/activity_vertical_margin"
    android:paddingLeft="@dimen/activity_horizontal_margin"
    android:paddingRight="@dimen/activity_horizontal_margin"
    android:paddingTop="@dimen/activity_vertical_margin">

    <ImageView
        android:id="@+id/imageView"
        android:layout_width="match_parent"
        android:layout_height="match_parent"
       />
</RelativeLayout>
```

Listing 40.3: The MainActivity class

```
package com.example.camerademo;
import java.io.File;
import android.app.Activity;
import android.content.Intent;
import android.net.Uri;
import android.os.Bundle;
import android.os.Environment;
import android.provider.MediaStore;
import android.util.Log;
import android.view.Menu;
import android.view.MenuItem;
import android.widget.ImageView;
import android.widget.Toast;

public class MainActivity extends Activity {
```

```
private static final int CAPTURE_IMAGE_ACTIVITY_REQUEST_CODE =
   100;
File pictureDir = new
   File(Environment.getExternalStoragePublicDirectory(
        Environment.DIRECTORY_PICTURES), "CameraDemo");
private static final String FILE_NAME = "image01.jpg";

private Uri fileUri;

@Override
protected void onCreate(Bundle savedInstanceState) {
    super.onCreate(savedInstanceState);
    setContentView(R.layout.activity_main);
    if (!pictureDir.exists()) {
        pictureDir.mkdirs();
    }
}

@Override
public boolean onCreateOptionsMenu(Menu menu) {
    getMenuInflater().inflate(R.menu.main, menu);
    return true;
}

@Override
public boolean onOptionsItemSelected(MenuItem item) {
    switch (item.getItemId()) {
        case R.id.action_camera:
            showCamera();
            return true;
        case R.id.action_email:
            emailPicture();
            return true;
        default:
            return super.onContextItemSelected(item);
    }
}

private void showCamera() {
    Intent intent = new Intent(
            MediaStore.ACTION_IMAGE_CAPTURE);
    File image = new File(pictureDir, FILE_NAME);
    fileUri = Uri.fromFile(image);
    intent.putExtra(MediaStore.EXTRA_OUTPUT, fileUri);
    // check if the device has a camera:
    if (intent.resolveActivity(getPackageManager()) != null) {
        startActivityForResult(intent,
                CAPTURE_IMAGE_ACTIVITY_REQUEST_CODE);
    }
}

@Override
protected void onActivityResult(int requestCode,
        int resultCode, Intent data) {
    if (requestCode ==
```

```
                CAPTURE_IMAGE_ACTIVITY_REQUEST_CODE) {
            if (resultCode == RESULT_OK) {
                ImageView imageView = (ImageView)
                        findViewById(R.id.imageView);
                File image = new File(pictureDir, FILE_NAME);
                fileUri = Uri.fromFile(image);
                imageView.setImageURI(fileUri);
            } else if (resultCode == RESULT_CANCELED) {
                Toast.makeText(this,  "Action cancelled",
                        Toast.LENGTH_LONG).show();
            } else {
                Toast.makeText(this,  "Error",
                        Toast.LENGTH_LONG).show();
            }
        }
    }

    private void emailPicture() {
        Intent emailIntent = new Intent(
                android.content.Intent.ACTION_SEND);
        emailIntent.setType("application/image");
        emailIntent.putExtra(android.content.Intent.EXTRA_EMAIL,
                new String[]{"me@example.com"});
        emailIntent.putExtra(android.content.Intent.EXTRA_SUBJECT,
                "New photo");
        emailIntent.putExtra(android.content.Intent.EXTRA_TEXT,
                "From My App");
        emailIntent.putExtra(Intent.EXTRA_STREAM, fileUri);
        startActivity(Intent.createChooser(emailIntent,
                "Send mail..."));
    }
}
```

The Show Camera button in **MainActivity** calls the **showCamera** method. This method starts Camera by calling **startActivityForResult**. The **emailPicture** method starts another activity that in turn activates the default Email application.

The Camera API

At the center of the Camera API is the **android.hardware.Camera** class. A **Camera** represents a digital camera.

Every camera has a viewfinder, through which the photographer can see what the camera is seeing. A viewfinder can be optical or electronic. An analog camera normally offers an optical viewfinder, which is a reversed telescope mounted on the camera body. Some digital cameras have an electronic viewfinder and some have an electronic one plus an optical one. On an Android tablet and handset, the whole screen or part of the screen is normally used as a viewfinder.

In an application that uses a camera, the **android.view.SurfaceView** class is normally used as a viewfinder. **SurfaceView** is a subclass of **View** and, as such, can be added to an activity by declaring it in a layout file using the **SurfaceView** element. The area of a **SurfaceView** will be continuously updated with what the camera sees. You control a **SurfaceView** through its **SurfaceHolder**, which you can obtain by calling the **getHolder** method on the **SurfaceView**. **SurfaceHolder**

is an interface in the **android.view** package.

Therefore, when working with a camera, you need to manage an instance of **Camera** as well as a **SurfaceHolder**.

Managing A Camera

When working with the Camera API, you should start by checking if the device does have a camera. You must also determine which camera to use if a device has multiple cameras. You do it by calling the **open** static method of the **Camera** class.

```
Camera camera = null;
try {
    if (Build.VERSION.SDK_INT >= Build.VERSION_CODES.GINGERBREAD) {
        camera = Camera.open(0);
    } else {
        camera = Camera.open();
    }
} catch (Exception e) {
    e.printStackTrace();
}
```

For pre-Gingerbread Android (Android version 2.3), use the no-argument method overload. For Android version 2.3, use the overload that takes an integer.

```
public static Camera open(int cameraId)
```

Passing 0 to the method gives you the first camera, 1 the second camera, and so on.

You should enclose the call to **open** in a try block as it may throw an exception.

Once you obtain a **Camera**, pass a **SurfaceHolder** to the **setPreviewDisplay** method on the **Camera**.

```
public void setPreviewDisplay(android.view.SurfaceHolder holder)
```

If **setPreviewDisplay** returns successfully, call the camera's **startPreview** method and the **SurfaceView** controlled by the **SurfaceHolder** will start displaying what the camera sees.

To take a picture, call the camera's **takePicture** method. After a picture is taken, the preview will stop so you will need to call **startPreview** again to take another picture.

When you are finished with the camera, call **stopPreview** and **release** to release the camera.

Optionally, you can configure the camera after you call **open** by calling its **getParameters** method, modifying the parameters, and passing them back to the camera using the **setParameters** method.

With the **takePicture** method you can decide what to do to the resulting raw and JPEG images from the camera. The signature of **takePicture** is as follows.

```
public final void takePicture(Camera.ShutterCallback shutter,
    Camera.PictureCallback raw, Camera.PictureCallback postview,
    Camera.PictureCallback jpeg)
```

The four parameters are these.

- *shutter*. The callback for image capture moment. For example, you can pass

code that plays a click sound to make it more like a real camera.
- *raw*. The callback for uncompressed image data.
- *postview*. The callback with postview image data.
- *jpeg*. The callback for JPEG image data.

You will learn how to use **Camera** in the **CameraAPIDemo** application.

Managing A SurfaceHolder

A **SurfaceHolder** communicates with its user through a series of methods in **SurfaceHolder.Callback**. To manage a **SurfaceHolder**, you need to pass an instance of **SurfaceHolder.Callback** to the **SurfaceHolder**'s **addCallback** method.

SurfaceHolder.Callback exposes these three methods that the **SurfaceHolder** will call in response to events.

```
public abstract void surfaceChanged(SurfaceHolder holder,
        int format, int width, int height)
```
Called after any structural changes (format or size) have been made to the surface.

```
public abstract void surfaceCreated(SurfaceHolder holder)
```
Called after the surface is first created.

```
public abstract void surfaceDestroyed(SurfaceHolder holder)
```
Called before a surface is being destroyed.

For instance, you might want to link a **SurfaceHolder** with a **Camera** right after the **SurfaceHolder** is created. Therefore, you might want to override the **surfaceCreated** method with this code.

```
@Override
public void surfaceCreated(SurfaceHolder holder) {
    try {
        camera.setPreviewDisplay(holder);
        camera.startPreview();
    } catch (Exception e){
        Log.d("camera", e.getMessage());
    }
}
```

Using the Camera API

The CameraAPIDemo application demonstrates the use of the Camera API to take still pictures. It uses a **SurfaceView** as a viewfinder and a button to take a picture. Clicking the button takes the picture and emits a beep sound. After a picture is taken, the **SurfaceView** freezes for two seconds to give the user the change to inspect the picture and restart the camera preview to allow the user to take another picture. All pictures are given a random name and stored in the external storage.

The application has one activity, whose layout is shown in Listing 40.4.

Listing 40.4: The layout file (activity_main.xml)

```
<LinearLayout
       xmlns:android="http://schemas.android.com/apk/res/android"
    android:orientation="vertical"
    android:layout_width="fill_parent"
```

```
    android:layout_height="fill_parent">

    <Button
        android:id="@+id/button1"
        android:layout width="wrap content"
        android:layout height="wrap content"
        android:layout gravity="center"
        android:onClick="takePicture"
        android:text="@string/button_take"/>

    <SurfaceView
        android:id="@+id/surfaceview"
        android:layout width="match parent"
        android:layout_height="match_parent" />
</LinearLayout>
```

The layout features a **LinearLayout** containing a button and a **SurfaceView**. The activity class is presented in Listing 40.5.

Listing 40.5: The MainActivity class

```
package com.example.cameraapidemo;
import java.io.File;
import java.io.FileNotFoundException;
import java.io.FileOutputStream;
import java.io.IOException;
import android.app.Activity;
import android.hardware.Camera;
import android.hardware.Camera.PictureCallback;
import android.hardware.Camera.ShutterCallback;
import android.media.AudioManager;
import android.media.SoundPool;
import android.net.Uri;
import android.os.Build;
import android.os.Bundle;
import android.os.Environment;
import android.os.Handler;
import android.provider.Settings;
import android.util.Log;
import android.view.Menu;
import android.view.SurfaceHolder;
import android.view.SurfaceView;
import android.view.View;
import android.widget.Button;
import android.widget.Toast;

public class MainActivity extends Activity
        implements SurfaceHolder.Callback {

    private Camera camera;
    SoundPool soundPool;
    int beepId;
    File pictureDir = new File(Environment
            .getExternalStoragePublicDirectory(
                    Environment.DIRECTORY_PICTURES),
                    "CameraAPIDemo");
    private static final String TAG = "camera";
```

```
@Override
public void onCreate(Bundle savedInstanceState) {
    super.onCreate(savedInstanceState);
    setContentView(R.layout.activity_main);
    pictureDir.mkdirs();

    soundPool = new SoundPool(1,
            AudioManager.STREAM NOTIFICATION, 0);
    Uri uri = Settings.System.DEFAULT RINGTONE URI;
    beepId = soundPool.load(uri.getPath(), 1);
    SurfaceView surfaceView = (SurfaceView)
            findViewById(R.id.surfaceview);
    surfaceView.getHolder().addCallback(this);
}

@Override
public boolean onCreateOptionsMenu(Menu menu) {
    getMenuInflater().inflate(R.menu.main, menu);
    return true;
}

@Override
public void onResume() {
   super.onResume();
   try {
       if (Build.VERSION.SDK INT >=
               Build.VERSION CODES.GINGERBREAD) {
           camera = Camera.open(0);
       } else {
           camera = Camera.open();
       }
   } catch (Exception e) {
       e.printStackTrace();
   }
}

@Override
public void onPause() {
    super.onPause();
    if (camera != null) {
        try {
            camera.release();
            camera = null;
        } catch (Exception e) {
            e.printStackTrace();
        }
    }
}

private void enableButton(boolean enabled) {
    Button button = (Button) findViewById(R.id.button1);
    button.setEnabled(enabled);
}
```

```
public void takePicture(View view) {
    enableButton(false);
    camera.takePicture(shutterCallback, null,
            pictureCallback);
}

private ShutterCallback shutterCallback =
        new ShutterCallback() {
    @Override
    public void onShutter() {
        // play sound
        soundPool.play(beepId, 1.0f, 1.0f, 0, 0, 1.0f);
    }
};

private PictureCallback pictureCallback =
        new PictureCallback() {
    @Override
    public void onPictureTaken(byte[] data,
            final Camera camera) {
        Toast.makeText(MainActivity.this, "Saving image",
                Toast.LENGTH_LONG)
                .show();
        File pictureFile = new File(pictureDir,
                System.currentTimeMillis() + ".jpg");

        try {
            FileOutputStream fos = new FileOutputStream(
                    pictureFile);
            fos.write(data);
            fos.close();
        } catch (FileNotFoundException e) {
            Log.d(TAG, e.getMessage());
        } catch (IOException e) {
            Log.d(TAG, e.getMessage());
        }

        Handler handler = new Handler();
        handler.postDelayed(new Runnable() {
            @Override
            public void run() {
                try {
                    enableButton(true);
                    camera.startPreview();
                } catch (Exception e) {
                    Log.d("camera",
                            "Error starting camera preview: "
                                    + e.getMessage());
                }
            }
        }, 2000);
    }
};

@Override
```

```
    public void surfaceCreated(SurfaceHolder holder) {
        try {
            camera.setPreviewDisplay(holder);
            camera.startPreview();
        } catch (Exception e){
            Log.d("camera", e.getMessage());
        }
    }

    @Override
    public void surfaceChanged(SurfaceHolder holder,
            int format, int w, int h3) {
        if (holder.getSurface() == null){
            Log.d(TAG, "surface does not exist, return");
            return;
        }

        try {
            camera.setPreviewDisplay(holder);
            camera.startPreview();
        } catch (Exception e){
            Log.d("camera", e.getMessage());
        }
    }

    @Override
    public void surfaceDestroyed(SurfaceHolder holder) {
        Log.d(TAG, "surfaceDestroyed");
    }
}
```

The **MainActivity** class uses a **Camera** and a **SurfaceView**. The latter continuously displays what the camera sees. Since a **Camera** takes a lot of resources to operate, the **MainActivity** releases the camera when the application stops and re-opens it when the application resumes.

The **MainActivity** class also implements **SurfaceHolder.Callback** and passes itself to the **SurfaceHolder** of the **SurfaceView** it employs as a viewfinder. This is shown in the following lines in the **onCreate** method.

```
    SurfaceView surfaceView = (SurfaceView)
            findViewById(R.id.surfaceview);
    surfaceView.getHolder().addCallback(this);
```

In both **surfaceCreated** and **surfaceChanged** methods that **MainActivity** overrides, the class calls the camera's **setPreviewDisplay** and **startPreview** methods. This makes sure when the camera is linked with a **SurfaceHolder**, the **SurfaceHolder** has already been created.

Another important point in **MainActivity** is the **takePicture** method that gets called when the user presses the button.

```
    public void takePicture(View view) {
        enableButton(false);
        camera.takePicture(shutterCallback, null,
                pictureCallback);
    }
```

The **takePicture** method disables the button so that no more picture can be taken

until the picture is saved and calls the **takePicture** method on the **Camera**, passing a **Camera.ShutterCallback** and a **Camera.PictureCallback**. Note that calling **takePicture** on a **Camera** also stops previewing the image on the **SurfaceHolder** linked to the camera.

The **Camera.ShutterCallback** in **MainActivity** has one method, **onShutter**, that plays a sound from the sound pool.

```
@Override
public void onShutter() {
    // play sound
    soundPool.play(beepId, 1.0f, 1.0f, 0, 0, 1.0f);
}
```

The **Camera.PictureCallback** also has one method, **onPictureTaken**, whose signature is this.

```
public void onPictureTaken(byte[] data, final Camera camera)
```

This method is called by the **Camera**'s **takePicture** method and receives a byte array containing the photo image.

The **onPictureTaken** method implementation in **MainActivity** does three things. First, it displays a message using the **Toast**. Second, it saves the byte array into a file. The name of the file is generated using **System.currentTimeMillis()**. Finally, the method creates a **Handler** to schedule a task that will be executed in two seconds. The task enables the button and calls the camera's **startPreview** so that the viewfinder will start working again.

Figure 40.2 shows the CameraAPIDemo application.

Summary

Android offers two options for applications that need to take still pictures: use a built-in intent to start Camera or use the Camera API. The first option is the easier one to use but lacks the features that the Camera API provides.

This chapter showed how to use both methods.

Figure 40.2: The CameraAPIDemo application

Chapter 41
Making Videos

The easiest way to provide video-making capability in your application is to use a built-in intent to activate an existing activity. However, if you need more than what the default application can provide, you need to get your hands dirty and work with the API directly.

This chapter shows how to use both methods for making videos.

Using the Built-in Intent

If you choose to use the default Camera application for making video, you can activate the application with these three lines of code.

```
int requestCode = ...;
Intent intent = new Intent(MediaStore.ACTION_VIDEO_CAPTURE);
startActivityForResult(intent, requestCode);
```

Basically, you need to create an **Intent** object by passing **MediaStore.ACTION_VIDEO_CAPTURE** to its constructor and pass it to the **startActivityForResult** method in your activity class. You can choose any integer for the request code that you will pass as the second argument to **startActivityForResult**. This method will pause the current activity and start Camera and make it ready to capture a video.

When you exit from Camera, either by canceling the operation or when you are done making a video, the system will resume your original activity (the activity where you called **startActivityForResult**) and call its **onActivityResult** method. If you are interested in saving or processing the captured video, you must override **onActivityResult**. Its signature is as follows.

```
protected void onActivityResult(int requestCode, int resultCode,
        android.content.Intent data)
```

The system calls **onActivityResult** by passing three arguments. The first argument, **requestCode**, is the request code passed when you called **startActivityForResult**. The request code is important if you are calling other activities from your activity, passing a different request code for each activity. Since you can only have one **onActivityResult** implementation in your activity, all calls to **startActivityForResult** will share the same **onActivityResult**, and you need to know which activity caused **onActivityResult** to be called by checking the value of the request code.

The second argument to **onActivityResult** is a result code. The value can be either **Activity.RESULT_OK** or **Activity.RESULT_CANCELED** or a user defined value. **Activity.RESULT_OK** indicates that the operation succeeded and **Activity.RESULT_CANCELED** indicates that the operation was canceled.

The third argument to **onActivityResult** contains data from Camera if the operation was successful.

In addition, you need the following **uses-feature** element in your manifest to indicate that your application needs to use the camera hardware of the device.

```
<uses-feature android:name="android.hardware.camera"
        android:required="true" />
```

As an example, consider the VideoDemo application that has an activity with a button on its action bar. You can press this button to activate Camera for the purpose of making a video. The VideoDemo application is shown in Figure 41.1.

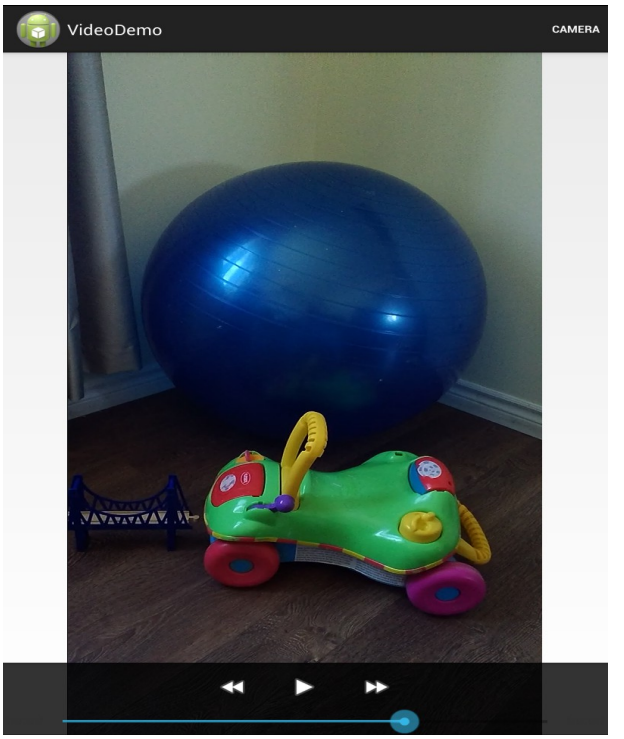

Figure 41.1: VideoDemo

The **AndroidManifest.xml** file in Listing 41.1 shows the activity used in the application as well as a **use-feature** element.

Listing 41.1: The AndroidManifest.xml file

```
<?xml version="1.0" encoding="utf-8"?>
<manifest
       xmlns:android="http://schemas.android.com/apk/res/android"
    package="com.example.videodemo"
    android:versionCode="1"
    android:versionName="1.0" >

    <uses-sdk
        android:minSdkVersion="16"
        android:targetSdkVersion="19" />

    <uses-feature android:name="android.hardware.camera"
                  android:required="true" />
```

```
    <application
        android:allowBackup="true"
        android:icon="@drawable/ic_launcher"
        android:label="@string/app_name"
        android:theme="@style/AppTheme" >
        <activity
            android:name="com.example.videodemo.MainActivity"
            android:label="@string/app_name">
            <intent-filter>
                <action
      android:name="android.intent.action.MAIN" />
                <category
      android:name="android.intent.category.LAUNCHER" />
            </intent-filter>
        </activity>
    </application>
</manifest>
```

There is only one activity in the application, the **MainActivity** activity. The activity reads the menu file in Listing 41.2 to populate its action bar.

Listing 41.2: The menu file (main.xml)

```
<menu xmlns:android="http://schemas.android.com/apk/res/android">
    <item
        android:id="@+id/action_camera"
        android:orderInCategory="100"
        android:showAsAction="always"
        android:title="@string/action_camera"/>
</menu>
```

The activity also uses the layout file in Listing 41.3 to set its view. There is only a **FrameLayout** with a **VideoView** element that is used to display the video file.

Listing 41.3: The activity_main.xml file

```
<FrameLayout
      xmlns:android="http://schemas.android.com/apk/res/android"
    android:layout_width="match_parent"
    android:layout_height="match_parent" >
    <VideoView
        android:id="@+id/videoView"
        android:layout_width="match_parent"
        android:layout_height="match_parent"
        android:layout_gravity="center">
    </VideoView>
</FrameLayout>
```

Note that I use a **FrameLayout** to enclose the **VideoView** to center it. For some reason, a **LinearLayout** or a **RelativeLayout** will not center it.

Finally, the **MainActivity** class is presented in Listing 41.4. You should know by now that the **onOptionsItemSelected** method is called when the a menu item is pressed. In short, pressing the Camera button on the action bar calls the **showCamera** method. **showCamera** constructs a built-in **Intent** and passes it to **startActivityForResult** to activate the video making feature in Camera.

Listing 41.4: The MainActivity class

```
package com.example.videodemo;
```

```
import android.app.Activity;
import android.content.Intent;
import android.net.Uri;
import android.os.Bundle;
import android.provider.MediaStore;
import android.view.Menu;
import android.view.MenuItem;
import android.widget.MediaController;
import android.widget.Toast;
import android.widget.VideoView;

public class MainActivity extends Activity {
    private static final int REQUEST_CODE = 200;

    @Override
    protected void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.activity_main);
    }

    @Override
    public boolean onCreateOptionsMenu(Menu menu) {
        getMenuInflater().inflate(R.menu.main, menu);
        return true;
    }

    @Override
    public boolean onOptionsItemSelected(MenuItem item) {
        switch (item.getItemId()) {
        case R.id.action_camera:
            showCamera();
            return true;
        default:
            return super.onContextItemSelected(item);
        }
    }

    private void showCamera() {
        // cannot set the video file
        Intent intent = new Intent(
                MediaStore.ACTION_VIDEO_CAPTURE);
        // check if the device has a camera:
        if (intent.resolveActivity(getPackageManager()) != null) {
            startActivityForResult(intent, REQUEST_CODE);
        } else {
            Toast.makeText(this, "Opening camera failed",
                    Toast.LENGTH_LONG).show();
        }
    }

    @Override
    protected void onActivityResult(int requestCode,
            int resultCode, Intent data) {
        if (requestCode == REQUEST_CODE) {
            if (resultCode == RESULT_OK) {
```

```
                if (data != null) {
                    Uri uri = data.getData();
                    VideoView videoView = (VideoView)
                            findViewById(R.id.videoView);

                    videoView.setVideoURI(uri);
                    videoView.setMediaController(
                            new MediaController(this));
                    videoView.requestFocus();
                }
            } else if (resultCode == RESULT_CANCELED) {
                Toast.makeText(this, "Action cancelled",
                        Toast.LENGTH_LONG).show();
            } else {
                Toast.makeText(this, "Error", Toast.LENGTH_LONG)
                        .show();
            }
        }
    }
}
```

What is interesting is the implementation of the **onActivityResult** method, which gets called when the user leaves Camera. If the result code is **RESULT_OK** and **data** is not null, the method calls the **getData** method on **data** to get a **Uri** pointing to the location of the video. Next, it finds the **VideoView** widget and set its **videoURI** property and calls two other methods on the **VideoView**, **setMediaController** and **requestFocus**.

```
    protected void onActivityResult(int requestCode,
            int resultCode, Intent data) {
        if (requestCode == REQUEST_CODE) {
            if (resultCode == RESULT_OK) {
                if (data != null) {
                    Uri uri = data.getData();
                    VideoView videoView = (VideoView)
                            findViewById(R.id.videoView);

                    videoView.setVideoURI(uri);
                    videoView.setMediaController(
                            new MediaController(this));
                    videoView.requestFocus();
                }
    ...
```

Passing a **MediaController** decorates the **VideoView** with a media controller that can be used to play and stop the video. Calling **requestFocus()** on the VideoView sets focus on the widget.

MediaRecorder

If you choose to deal with the API directly rather than using the Camera to provide your application with video-making capability, you need to know the details of **MediaRecorder**.

The **android.media.MediaRecorder** class can be used to record audio and

video. Figure 41.2 shows the various states a **MediaRecord** can be in.

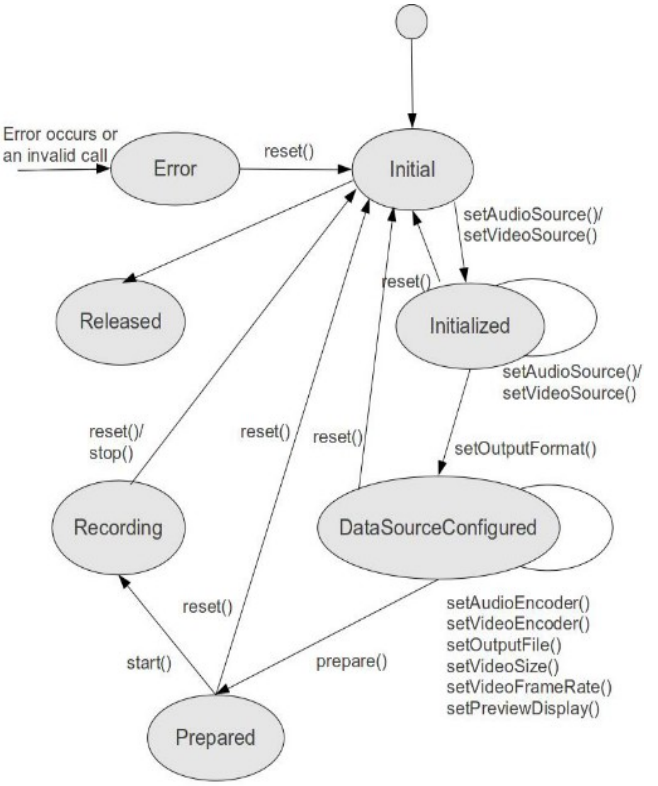

Figure 41.2: The MediaRecorder state diagram

To capture a video with a **MediaRecorder**, of course you need an instance of it. So, the first thing to do is to create a **MediaRecorder**.

```
MediaRecorder mediaRecorder = new MediaRecorder();
```

Then, as you can see in Figure 41.2, to record a video, you have to bring the **MediaRecorder** to the **Initialized** state, followed by the **DataSourceConfigured** and **Prepared** states by calling certain methods.

To transition a **MediaRecorder** to the **Initialized** state, call the **setAudioSource** and **setVideoSource** methods to set the audio and video sources. The valid value for **setAudioSource** is one of the fields defined in the **MediaRecorder.AudioSource** class, which are **CAMCORDER**, **DEFAULT**, **MIC**, **REMOTE_SUBMIX**, **VOICE_CALL**, **VOICE_COMMUNICATION**, **VOICE_DOWNLINK**, **VOICE_RECOGNITION**, and **VOICE_UPLINK**.

The valid value for **setVideoSource** is one of the fields in the **MediaRecorder.VideoSource** class, which are **CAMERA** and **DEFAULT**.

Once the **MediaRecorder** is in the **Initialized** state, call its **setOutputFormat** method, passing one of the file formats in the **MediaRecorder.OutputFormat** class. The following fields are defined: **AAC_ADTS**, **AMR_NB**, **AMR_WB**, **DEFAULT**, **MPEG_4**, **RAW_AMR**, and **THREE_GPP**.

Successfully calling **setOutputFormat** brings the **MediaRecorder** to the **DataSourceConfigured** state. You just need to call **prepare** to prepare the **MediaRecorder**.

To start recording, call the **start** method. It will keep recording until **stop** is called or an error occurs. An error may occur if the **MediaRecorder** runs out of space to store video or if a specified maximum record time is exceeded.

Once you stop a **MediaRecorder**, it goes back to the initial state. You must take it through the previous three states again to record another video.

Also note that a **MediaRecorder** uses a lot of resources and it is prudent to release the resources by calling its **release** method if the **MediaRecorder** is not being used. For example, you should release the **MediaRecorder** when the activity is paused. Once a **MediaRecorder** is released, the same instance cannot be reused to record another video.

Using MediaRecorder

The **VideoRecorder** application demonstrates how to use the **MediaRecorder** to record a video. It has one activity that contains a button and a **SurfaceView**. The button is used to start and stop recording whereas the **SurfaceView** for displaying what the camera sees. **SurfaceView** was explained in detail in Chapter 30, "Taking Pictures."

The layout file for the activity is shown in Listing 41.5 and the activity class in Listing 41.6.

Listing 41.5: The layout file (activity_main.xml)

```
<LinearLayout
       xmlns:android="http://schemas.android.com/apk/res/android"
    android:orientation="vertical"
    android:layout_width="fill_parent"
    android:layout_height="fill_parent">

    <Button
        android:id="@+id/button1"
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:layout_marginLeft="33dp"
        android:layout_marginTop="22dp"
        android:onClick="startStopRecording"
        android:text="@string/button_start" />
     <SurfaceView
        android:id="@+id/surfaceView"
        android:layout_width="match_parent"
        android:layout_height="match_parent" />
</LinearLayout>
```

Listing 41.6: The MainActivity class

```
package com.example.videorecorder;
import java.io.File;
import java.io.IOException;
import android.app.Activity;
import android.media.MediaRecorder;
import android.os.Bundle;
import android.os.Environment;
import android.view.SurfaceHolder;
import android.view.SurfaceView;
```

```
import android.view.View;
import android.widget.Button;

public class MainActivity extends Activity {
    private MediaRecorder mediaRecorder;
    private File outputDir;
    private boolean recording = false;

    @Override
    public void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        File moviesDir = Environment
                .getExternalStoragePublicDirectory(
                        Environment.DIRECTORY_MOVIES);
        outputDir = new File(moviesDir,
                "VideoRecorder");
        outputDir.mkdirs();
        setContentView(R.layout.activity_main);
    }

    @Override
    protected void onResume() {
        super.onResume();
        mediaRecorder = new MediaRecorder();
        initAndConfigureMediaRecorder();
    }

    @Override
    protected void onPause() {
        super.onPause();
        if (recording) {
            try {
                mediaRecorder.stop();
            } catch (IllegalStateException e) {
            }
        }
        releaseMediaRecorder();
        Button button = (Button) findViewById(R.id.button1);
        button.setText("Start");
        recording = false;
    }

    private void releaseMediaRecorder() {
        if (mediaRecorder != null) {
            mediaRecorder.reset();
            mediaRecorder.release();
            mediaRecorder = null;
        }
    }

    private void initAndConfigureMediaRecorder() {
        mediaRecorder.setAudioSource(
                MediaRecorder.AudioSource.CAMCORDER);
        mediaRecorder
                .setVideoSource(MediaRecorder.VideoSource.CAMERA);
```

```
        mediaRecorder.setOutputFormat(
                MediaRecorder.OutputFormat.MPEG_4);
        mediaRecorder.setVideoFrameRate(10);// make it very low
        mediaRecorder.setVideoEncoder(
                MediaRecorder.VideoEncoder.MPEG_4_SP);
        mediaRecorder.setAudioEncoder(
                MediaRecorder.AudioEncoder.AMR_NB);
        String outputFile = new File(outputDir,
                System.currentTimeMillis() + ".mp4")
                    .getAbsolutePath();

        mediaRecorder.setOutputFile(outputFile);
        SurfaceView surfaceView = (SurfaceView)
                findViewById(R.id.surfaceView);
        SurfaceHolder surfaceHolder = surfaceView.getHolder();
        mediaRecorder.setPreviewDisplay(surfaceHolder
                .getSurface());
    }



    public void startStopRecording(View view) {
        Button button = (Button) findViewById(R.id.button1);
        if (recording) {
            button.setText("Start");
            try {
                mediaRecorder.stop();
            } catch (IllegalStateException e) {

            }
            releaseMediaRecorder();
        } else {
            button.setText("Stop");
            if (mediaRecorder == null) {
                mediaRecorder = new MediaRecorder();
                initAndConfigureMediaRecorder();
            }
            // prepare MediaRecorder
            try {
                mediaRecorder.prepare();
            } catch (IllegalStateException e) {
                e.printStackTrace();
            } catch (IOException e) {
                e.printStackTrace();
            }
            mediaRecorder.start();
        }
        recording = !recording;
    }
}
```

Let's start with the **onCreate** method. It does an important job which is to create a directory for all videos captured under the default directory for movie files.

```
        File moviesDir = Environment
                .getExternalStoragePublicDirectory(
```

```
                Environment.DIRECTORY_MOVIES);
        outputDir = new File(moviesDir,
                "VideoRecorder");
        outputDir.mkdirs();
```

The other two important methods are **onResume** and **onPause**. In **onResume** you create a new instance of **MediaRecorder** and initialize and configure it by calling **initAndConfigureMediaRecorder**. Why a new instance every time? Because once used, a **MediaRecorder** cannot be reused.

In **onPause**, you stop the **MediaRecorder** if it is recording and call the **releaseMediaRecorder** method to release the **MediaRecorder**.

Now, let's have a look at the **initAndConfigureMediaRecorder** and **releaseMediaRecorder** methods.

As the name implies, **initAndConfigureMediaRecorder** initializes and configures the **MediaRecorder** created by the **onResume** method. It calls various methods in **MediaRecorder** to transition it to the **Initialized** and **DataSourceConfigured** states. It also passes the **Surface** of the **SurfaceView** to display what the camera sees.

```
        SurfaceView surfaceView = (SurfaceView)
                findViewById(R.id.surfaceView);
        SurfaceHolder surfaceHolder = surfaceView.getHolder();
        mediaRecorder.setPreviewDisplay(surfaceHolder
                .getSurface());
```

In this state, the **MediaRecorder** just waits until the user presses the Start button. When it happens, the **startStopRecording** method is called, which in turn calls the **prepare** and **start** methods on the **MediaRecorder**. It also changes the Start button to a Stop button.

When the user presses the Stop button, the **MediaRecorder**'s **stop** method is called and the **MediaRecorder** is released. The Stop button is changed back to a Start button, waiting for another turn.

Summary

Two methods are available if you want to equip your application with video-making capability. The first, the easy one, is by creating the default intent and passing it to **startActivityForResult**. The second method is to use **MediaRecorder** directly. This method is harder but brings with it the full features of the device camera.

This chapter showed how to use both methods to make video.

Chapter 42
The Sound Recorder

The Android platform ships with a multitude of APIs, including one for recording audio and video. In this chapter you learn how to use the **MediaRecorder** class to sample sound levels. This is the same class you used for making videos in Chapter 41, "Making Videos."

The MediaRecorder Class

Support for multimedia is rock solid in Android. There are classes that you can use to play audio and video as well as record them. In the **SoundMeter** project discussed in this chapter, you will use the **MediaRecorder** class to sample sound or noise levels.**MediaRecorder** is used to record audio and video. The output can be written to a file and the input source can be easily selected. It is relatively easy to use too. You start by instantiating the **MediaRecorder** class.

```
MediaRecorder mediaRecorder = new MediaRecorder();
```

Then, configure the instance by calling its **setAudioSource**, **setVideoSource**, **setOutputFormat**, **setAudioEncoder**, **setOutputFile**, or other methods. Next, prepare the **MediaRecorder** by calling its **prepare** method:

```
mediaRecorder.prepare();
```

Note that **prepare** may throw exception if the **MediaRecorder** is not configured property or if you do not have the right permissions.

To start recording, call its **start** method. To stop recording, call **stop**.

When you're done with a **MediaRecorder**, call its **reset** method to return it to its initial state and its **release** method to release resources it currently holds.

```
mediaRecorder.reset();
mediaRecorder.release();
```

Example

Now that you know how to use the **MediaRecorder**, let's take a look at the SoundMeter project. The application samples sound amplitudes at certain intervals and displays the current level as a bar.

As usual, let's start by looking at the manifest (the **AndroidManifest.xml** file) for the project. It is given in Listing 42.1.

Listing 42.1: The manifest for SoundMeter

```
<?xml version="1.0" encoding="utf-8"?>
```

```
<manifest
       xmlns:android="http://schemas.android.com/apk/res/android"
    package="com.example.soundmeter"
    android:versionCode="1"
    android:versionName="1.0" >

    <uses-sdk
        android:minSdkVersion="8"
        android:targetSdkVersion="17" />

    <uses-permission android:name="android.permission.RECORD_AUDIO"
       />

    <application
        android:allowBackup="true"
        android:icon="@drawable/ic_launcher"
        android:label="@string/app_name"
        android:theme="@style/AppTheme" >
        <activity
            android:name="com.example.soundmeter.MainActivity"
            android:label="@string/app_name" >
            <intent-filter>
                <action android:name="android.intent.action.MAIN"/>
                <category
       android:name="android.intent.category.LAUNCHER" />
            </intent-filter>
        </activity>
    </application>
</manifest>
```

One thing to note here is the use of the **uses-permission** element in the manifest to ask for the user's permission to record audio. If you don't include this element, your application will not work. Also, if the user does not consent, the application will not install.

There is only one activity in this project as can be seen in the manifest.

Listing 42.2 shows the layout file for the main activity. A **RelativeLayout** is used for the main display and it contains a **TextView** for displaying the current sound level and a button that will act as a sound indicator.

Listing 42.2: The res/layout/activity_main.xml file in SoundMeter

```
<RelativeLayout
       xmlns:android="http://schemas.android.com/apk/res/android"
    xmlns:tools="http://schemas.android.com/tools"
    android:layout_width="match_parent"
    android:layout_height="match_parent"
    android:paddingBottom="@dimen/activity_vertical_margin"
    android:paddingLeft="@dimen/activity_horizontal_margin"
    android:paddingRight="@dimen/activity_horizontal_margin"
    android:paddingTop="@dimen/activity_vertical_margin"
    tools:context=".MainActivity" >

    <TextView
        android:id="@+id/level"
        android:layout_width="wrap_content"
        android:layout_height="wrap_content" />
```

```
    <Button
        android:id="@+id/button1"
        style="?android:attr/buttonStyleSmall"
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:layout_alignLeft="@+id/level"
        android:layout_below="@+id/level"
        android:background="#ff0000"
        android:layout_marginTop="30dp" />

</RelativeLayout>
```

There are two Java classes in this application. The first one, given in Listing 42.3, is a class called **SoundMeter** that encapsulates a **MediaRecorder** and exposes three methods to manage it. The first method, **start**, creates an instance of **MediaRecorder**, configures it, and starts it. The second method, **stop**, stops the **MediaRecorder**. The third method, **getAmplitude**, returns a **double** indicating the sampled sound level.

Listing 42.3: The SoundMeter class

```
package com.example.soundmeter;
import java.io.IOException;
import android.media.MediaRecorder;

public class SoundMeter {

    private MediaRecorder mediaRecorder;
    boolean started = false;

    public void start() {
        if (started) {
            return;
        }
        if (mediaRecorder == null) {
            mediaRecorder = new MediaRecorder();

            mediaRecorder.setAudioSource(
                    MediaRecorder.AudioSource.MIC);
            mediaRecorder.setOutputFormat(
                    MediaRecorder.OutputFormat.THREE_GPP);
            mediaRecorder.setAudioEncoder(
                    MediaRecorder.AudioEncoder.AMR_NB);
            mediaRecorder.setOutputFile("/dev/null");
            try {
                mediaRecorder.prepare();
            } catch (IllegalStateException e) {
                e.printStackTrace();
            } catch (IOException e) {
                e.printStackTrace();
            }
            mediaRecorder.start();
            started = true;
        }
    }
```

```
    public void stop() {
        if (mediaRecorder != null) {
            mediaRecorder.stop();
            mediaRecorder.release();
            mediaRecorder = null;
            started = false;
        }
    }
    public double getAmplitude() {
        return mediaRecorder.getMaxAmplitude() / 100;
    }
}
```

The second Java class, **MainActivity**, is the main activity class for the application. It is presented in Listing 42.4.

Listing 42.4: The MainActivity class in SoundMeter

```
package com.example.soundmeter;
import android.app.Activity;
import android.os.Bundle;
import android.os.Handler;
import android.view.Menu;
import android.widget.Button;
import android.widget.TextView;

public class MainActivity extends Activity {
    Handler handler = new Handler();
    SoundMeter soundMeter = new SoundMeter();

    @Override
    protected void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.activity_main);
    }

    @Override
    public boolean onCreateOptionsMenu(Menu menu) {
        // Inflate the menu; this adds items to the action bar if
       it
        // is present.
        getMenuInflater().inflate(R.menu.main, menu);
        return true;
    }

    @Override
    public void onStart() {
        super.onStart();
        soundMeter.start();
        handler.postDelayed(pollTask, 150);
    }

    @Override
    public void onPause() {
        soundMeter.stop();
        super.onPause();
    }
```

```
    private Runnable pollTask = new Runnable() {
        @Override
        public void run() {
            double amplitude = soundMeter.getAmplitude();
            TextView textView = (TextView)
       findViewById(R.id.level);
            textView.setText("amp:" + amplitude);
            Button button = (Button) findViewById(R.id.button1);
            button.setWidth((int) amplitude * 10);
            handler.postDelayed(pollTask, 150);
        }
    };
}
```

The **MainActivity** class overrides two activity lifecycle methods, **onStart** and **onPause**. You may recall that the system calls **onStart** right after an activity was created or after it was restarted. The system calls **onPause** when the activity was paused because another activity was started or because an important event occurred. In the **MainActivity** class, the **onStart** method starts the **SoundMeter** and the **onPause** method stops it. The **MainActivity** class also uses a **Handler** to sample the sound level every 150 milliseconds.

Figure 42.1 shows the application. The horizontal bar shows the current sound amplitude.

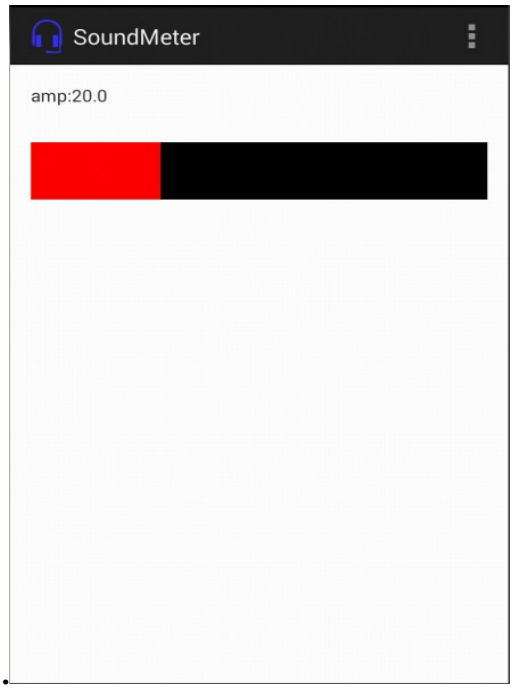

Figure 42.1: The SoundMeter application

Summary

In this chapter you learned to use the **MediaRecorder** class to record audio. You also created an application for sampling noise levels.

Chapter 43
Handling the Handler

One of the most interesting and useful types in the Android SDK is the **Handler** class. Most of the time, it is used to process messages and schedule a task to run at a future time.

This chapter explains what the class is good for and offers examples.

Overview

The **android.os.Handler** class is an exciting utility class that, among others, can be scheduled do execute a **Runnable** at a future time. Any task assigned to a **Handler** will run on the **Handler**'s thread. In turn, the **Handler** runs on the thread that created it, which in most cases would be the UI thread. As such, you should not schedule a long-running task with a **Handler** because it would make your application freeze. However, you can use a **Handler** to handle a long-running task if you can be split the task into smaller parts, which you learn how to achieve in this section.

To schedule a task to run at a future time, call the **Handler** class's **postDelayed** or **postAtTime** method.

```
public final boolean postDelayed(Runnable task, long x)
```

public final boolean postAtTime(Runnable *task*, long *time*)**postDelayed** runs a task *x* milliseconds after the method is called. For example, if you want a **Runnable** to start five seconds from now, use this code.

```
Handler handler = new Handler();
```

handler.postDelayed(*runnable*, 5000);**postAtTime** runs a task at a certain time in the future. For example, if you want a task to run six seconds later, write this.

```
Handler handler = new Handler();
handler.postAtTime(runnable, 6000 + System.currentTimeMillis());
```

Example

As an example, consider the **HandlerTest** project that uses **Handler** to animate an **ImageView**. The animation performed is simple: show an image for 400 milliseconds, then hide it for 400 milliseconds, and repeat this five times. The entire task would take about four seconds if all the work is done in a **for** loop that sleeps for 400 milliseconds at each iteration. Using the **Handler**, however, you can split this into 10 smaller parts that each takes less than one millisecond (the exact time would depend on the device running it). The UI thread is released during each 400ms wait so that it can cater for something else.

Note

Android offers animation APIs that you should use for all animation tasks. This example uses **Handler** to animate a control simply to illustrate the use of **Handler**.

Listing 43.1 shows the manifest (the **AndroidManifest.xml** file) for the project.

Listing 43.1: The manifest for HandlerTest

```
<?xml version="1.0" encoding="utf-8"?>
<manifest
       xmlns:android="http://schemas.android.com/apk/res/android"
    package="com.example.handlertest"
    android:versionCode="1"
    android:versionName="1.0" >

    <uses-sdk
        android:minSdkVersion="8"
        android:targetSdkVersion="17" />

    <application
        android:allowBackup="true"
        android:icon="@drawable/ic_launcher"
        android:label="@string/app_name"
        android:theme="@style/AppTheme" >
        <activity
            android:name="com.example.handlertest.MainActivity"
            android:label="@string/app_name" >
            <intent-filter>
                <action android:name="android.intent.action.MAIN"/>
                <category
                  android:name="android.intent.category.LAUNCHER"/>
            </intent-filter>
        </activity>
    </application>
</manifest>
```

Nothing spectacular in the manifest. It shows that there is one activity named **MainActivity**. The layout file for the activity is given in Listing 43.2.

Listing 43.2: The res/layout/activity_main.xml file in HandlerTest

```
<RelativeLayout
       xmlns:android="http://schemas.android.com/apk/res/android"
    xmlns:tools="http://schemas.android.com/tools"
    android:layout_width="match_parent"
    android:layout_height="match_parent"
    android:paddingBottom="@dimen/activity_vertical_margin"
    android:paddingLeft="@dimen/activity_horizontal_margin"
    android:paddingRight="@dimen/activity_horizontal_margin"
    android:paddingTop="@dimen/activity_vertical_margin"
    tools:context=".MainActivity" >

    <ImageView
        android:id="@+id/imageView1"
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:layout_alignParentLeft="true"
        android:layout_alignParentTop="true"
```

```
        android:layout_marginLeft="51dp"
        android:layout_marginTop="58dp"
        android:src="@drawable/surprise" />

    <Button
        android:id="@+id/button1"
        style="?android:attr/buttonStyleSmall"
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:layout_alignRight="@+id/imageView1"
        android:layout_below="@+id/imageView1"
        android:layout_marginRight="18dp"
        android:layout_marginTop="65dp"
        android:onClick="buttonClicked"
        android:text="Button"/>
</RelativeLayout>
```

The main layout for **MainActivity** is a **RelativeLayout** that contains an **ImageView** to be animated and a button to start animation.

Now look at the **MainActivity** class in Listing 43.3. This is the main core of the application.

Listing 43.3: The MainActivity class in HandlerTest

```
package com.example.handlertest;
import android.app.Activity;
import android.os.Bundle;
import android.os.Handler;
import android.view.Menu;
import android.view.View;
import android.widget.ImageView;

public class MainActivity extends Activity {

    int counter = 0;
    Handler handler = new Handler();

    @Override
    protected void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.activity_main);
        getUserAttention();
    }

    @Override
    public boolean onCreateOptionsMenu(Menu menu) {
        // Inflate the menu; this adds items to the action bar if
       it
        // is present.
        getMenuInflater().inflate(R.menu.main, menu);
        return true;
    }

    public void buttonClicked(View view) {
        counter = 0;
        getUserAttention();
```

```
    }

    private void getUserAttention() {
        handler.post(task);
    }

    Runnable task = new Runnable() {
        @Override
        public void run() {
            ImageView imageView = (ImageView)
                    findViewById(R.id.imageView1);
            if (counter % 2 == 0) {
                imageView.setVisibility(View.INVISIBLE);
            } else {
                imageView.setVisibility(View.VISIBLE);
            }
            counter++;
            if (counter < 8) {
                handler.postDelayed(this, 400);
            }
        }
    };
}
```

The brain of this activity are a **Runnable** called **task**, which animates the **ImageView**, and the **getUserAttention** method that calls the **postDelayed** method on a **Handler**. The **Runnable** sets the **ImageView**'s visibility to **Visible** or **Invisible** depending on whether the value of the **counter** variable is odd or even.

If you run the **HandlerTest** project, you'll see something similar to the screenshot in Figure 43.1. Note how the **ImageView** flashes to get your attention. Try clicking the button several times quickly to make the image flash faster. Can you explain why it goes faster as you click?

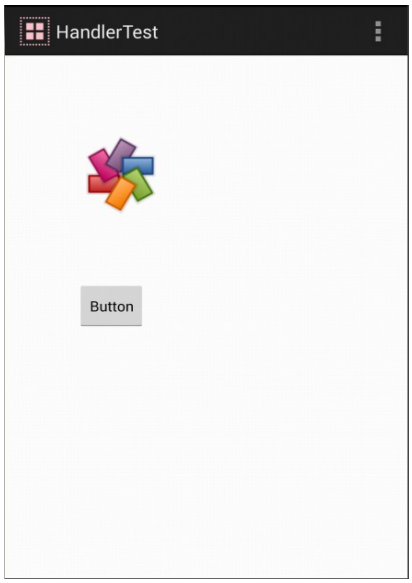

Figure 43.1: The HandlerTest application

Summary

In this chapter you learned about the **Handler** class and write an application that makes use of the class.

Chapter 44
Asynchronous Tasks

This chapter talks about asynchronous tasks and how to handle them using the **AsyncTask** class. It also presents a photo editor application that illustrates how this class should be used.

Overview

The **java.os.AsyncTask** class is a utility class that makes it easy to handle background processes and publish progress updates on the UI thread. This class is meant for short operations that last at most a few seconds. For long-running background tasks, you should use the Java Concurrency Utilities framework.

The **AsyncTask** class comes with a set of public methods and a set of protected methods. The public methods are for executing and canceling its task. The **execute** method starts an asynchronous operation and **cancel** cancels it. The protected methods are for you to override in a subclass. The **doInBackground** method, a protected method, is the most important method in this class and provides the logic for the asynchronous operation.

There is also a **publishProgress** method, also a protected method, which is normally called multiple times from **doInBackground**. Typically, you will write code to update a progress bar or some other UI component here.

Then there are two **onCancelled** methods for you to write what should happen if the operation was canceled (i.e. if the **AsyncTask**'s **cancel** method was called).

Example

As an example, the PhotoEditor application that accompanies this book uses the **AsyncTask** class to perform image operations that each takes a few seconds. **AsyncTask** is used so as not to jam the UI thread. Two image operations, invert and blur, are supported.

The application manifest (the **AndroidManifest.xml** file) is printed in Listing 44.1.

Listing 44.1: The manifest for PhotoEditor

```
<?xml version="1.0" encoding="utf-8"?>
<manifest
      xmlns:android="http://schemas.android.com/apk/res/android"
    package="com.example.photoeditor"
    android:versionCode="1"
    android:versionName="1.0" >
```

```
    <uses-sdk
        android:minSdkVersion="8"
        android:targetSdkVersion="17" />

    <application
        android:allowBackup="true"
        android:icon="@drawable/ic_launcher"
        android:label="@string/app_name"
        android:theme="@style/AppTheme" >
        <activity
            android:name="com.example.photoeditor.MainActivity"
            android:label="@string/app_name" >
            <intent-filter>
                <action
       android:name="android.intent.action.MAIN" />
                <category
       android:name="android.intent.category.LAUNCHER" />
            </intent-filter>
        </activity>
    </application>

</manifest>
```

The layout file, printed in Listing 44.2, shows that the application uses a vertical **LinearLayout** to house an **ImageView**, a **ProgressBar**, and two buttons. The latter are contained in a horizontal **LinearLayout**. The first button is used to start the blur operation and the second to start the invert operation.

Listing 44.2: The res/layout/activity_main.xml file in PhotoEditor

```
<LinearLayout
       xmlns:android="http://schemas.android.com/apk/res/android"
    xmlns:tools="http://schemas.android.com/tools"
    android:layout_width="fill_parent"
    android:layout_height="fill_parent"
    android:orientation="vertical"
    android:paddingLeft="16dp"
    android:paddingRight="16dp" >

    <LinearLayout
        android:layout_height="wrap_content"
        android:layout_width="fill_parent"
        android:orientation="horizontal" >

        <Button
            android:id="@+id/blurButton"
            android:layout_width="wrap_content"
            android:layout_height="wrap_content"
            android:onClick="doBlur"
            android:text="@string/blur_button_text" />

        <Button
            android:id="@+id/button2"
            android:layout_width="wrap_content"
            android:layout_height="wrap_content"
            android:onClick="doInvert"
            android:text="@string/invert_button_text" />
```

```
    </LinearLayout>

    <ProgressBar
        android:id="@+id/progressBar1"
        style="?android:attr/progressBarStyleHorizontal"
        android:layout_width="fill_parent"
        android:layout_height="10dp" />

    <ImageView
        android:id="@+id/imageView1"
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:layout_gravity="top|center"
        android:src="@drawable/photo1" />

</LinearLayout>
```

Finally, the **MainActivity** class for this project is given in Listing 44.3.

Listing 44.3: The MainActivity class in PhotoEditor

```
package com.example.photoeditor;
import android.app.Activity;
import android.graphics.Bitmap;
import android.graphics.drawable.BitmapDrawable;
import android.os.AsyncTask;
import android.os.Bundle;
import android.view.Menu;
import android.view.View;
import android.widget.ImageView;
import android.widget.ProgressBar;

public class MainActivity extends Activity {
    private ProgressBar progressBar;

    @Override
    protected void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.activity_main);
        progressBar = (ProgressBar)
       findViewById(R.id.progressBar1);
    }

    @Override
    public boolean onCreateOptionsMenu(Menu menu) {
        // Inflate the menu; this adds items to the action bar if
       it
        // is present.
        getMenuInflater().inflate(R.menu.main, menu);
        return true;
    }

    public void doBlur(View view) {
        BlurImageTask task = new BlurImageTask();
        ImageView imageView = (ImageView)
                findViewById(R.id.imageView1);
        Bitmap bitmap = ((BitmapDrawable)
```

```
                imageView.getDrawable()).getBitmap();
        task.execute(bitmap);
    }

    public void doInvert(View view) {
        InvertImageTask task = new InvertImageTask();
        ImageView imageView = (ImageView)
                findViewById(R.id.imageView1);
        Bitmap bitmap = ((BitmapDrawable)
        imageView.getDrawable()).getBitmap();
        task.execute(bitmap);
    }

    private class InvertImageTask extends AsyncTask<Bitmap,
       Integer,
            Bitmap> {
        protected Bitmap doInBackground(Bitmap... bitmap) {
            Bitmap input = bitmap[0];
            Bitmap result = input.copy(input.getConfig(),
                    /*isMutable'*/true);
            int width = input.getWidth();
            int height = input.getHeight();
            for (int i = 0; i < height; i++) {
                for (int j = 0; j < width; j++) {
                    int pixel = input.getPixel(j, i);
                    int a = pixel & 0xff000000;
                    a = a | (~pixel & 0x00ffffff);
                    result.setPixel(j, i, a);
                }
                int progress = (int) (100*(i+1)/height);
                publishProgress(progress);
            }
            return result;
        }

        protected void onProgressUpdate(Integer... values) {
            progressBar.setProgress(values[0]);
        }

        protected void onPostExecute(Bitmap result) {
            ImageView imageView = (ImageView)
                    findViewById(R.id.imageView1);
            imageView.setImageBitmap(result);
            progressBar.setProgress(0);
        }
    }

    private class BlurImageTask extends AsyncTask<Bitmap, Integer,
            Bitmap> {
        protected Bitmap doInBackground(Bitmap... bitmap) {
            Bitmap input = bitmap[0];
            Bitmap result = input.copy(input.getConfig(),
                    /*isMutable=*/ true);
            int width = bitmap[0].getWidth();
            int height = bitmap[0].getHeight();
```

```
            int level = 7;
            for (int i = 0; i < height; i++) {
                for (int j = 0; j < width; j++) {
                    int pixel = bitmap[0].getPixel(j, i);
                    int a = pixel & 0xff000000;
                    int r = (pixel >> 16) & 0xff;
                    int g = (pixel >> 8) & 0xff;
                    int b = pixel & 0xff;
                    r = (r+level)/2;
                    g = (g+level)/2;
                    b = (b+level)/2;
                    int gray = a | (r << 16) | (g << 8) | b;
                    result.setPixel(j, i, gray);
                }
                int progress = (int) (100*(i+1)/height);
                publishProgress(progress);
            }
            return result;
        }

        protected void onProgressUpdate(Integer... values) {
            progressBar.setProgress(values[0]);
        }

        protected void onPostExecute(Bitmap result) {
            ImageView imageView = (ImageView)
                    findViewById(R.id.imageView1);
            imageView.setImageBitmap(result);
            progressBar.setProgress(0);
        }
    }
}
```

The **MainActivity** class contains two private classes, **InvertImageTask** and **BlurImageTask**, which extend **AsyncTask**. The **InvertImageTask** task is executed when the **Invert** button is clicked and the **BlurImageTask** when the **Blur** button is clicked.

The **doInBackground** method in each task processes the **ImageView** bitmap in a **for** loop. At each iteration it calls the **publishProgress** method to update the progress bar.

Figure 44.1 shows the initial bitmap and Figure 44.2 shows the bitmap after an invert operation.

Figure 44.1: The ImageEditor application

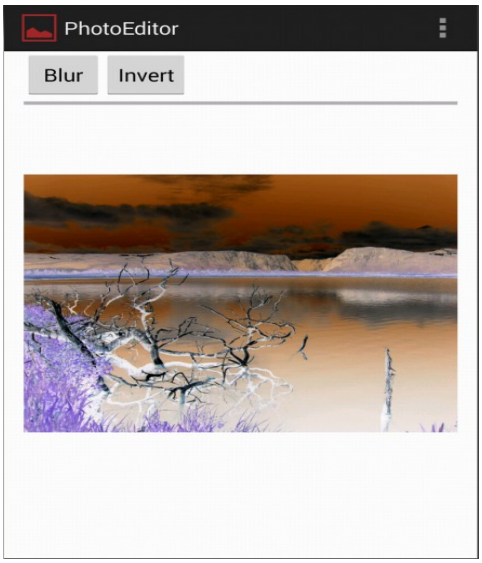

Figure 44.2: The bitmap after invert

Summary

In this chapter you learned to use the **AsyncTask** class and created a photo editor application that uses it.

Appendix A
javac

javac is a Java compiler for compiling Java programs to bytecode. Source files must have a **java** extension and be organized in a directory structure that reflects the package tree. The results are **class** files in a directory structure reflecting the package tree.

javac has the following syntax.

`javac [options] [sourceFiles] [@argFiles]`

where *options* are command-line options, *sourceFiles* one or more Java source files, and *@argFiles* one or more files that list options and source files.

You can pass source code file names to **javac** in one of two ways:

- List the file names on the command line. This method is suitable if the number of source files is small.
- List the file names in a file, separated by blanks or line breaks, then pass the path to the list file to the **javac** command line prefixed by an **@** character. This is appropriate for a large number of source files.

Options

Options are used to pass instructions to **javac**. For example, you can tell **javac** where to find classes referenced in the source files, where to put the generated class files, etc. There are two types of options, standard and nonstandard. Nonstandard options start with -X.

Here are the lists of standard and nonstandard options.

Standard Options

`-classpath classpath`

If in your source files you reference other Java types than those packaged in the Java standard libraries, you need to tell **javac** how to find these external types by using the **–classpath** option. The value should be the path to a directory containing referenced Java types or a jar file containing them. The path can be absolute or relative to the current directory. Two paths are separated by a semicolon in Windows and by a colon in Unix/Linux.
For example, the following Windows command line compiles **MyClass.java** that references the **primer.FileObject** class located in the **C:\program\classes** directory.

```
javac -classpath C:/program/classes/ MyClass.java
```

Note that the **FileObject** class is in the **primer** package, so you pass the directory containing the package.

The following Linux command line compiles **MyClass.java** that references the **primer.FileObject** class located in the **/home/user1/classes** directory.

```
javac -classpath /home/user1/classes/ MyClass.java
```

To reference class files packaged in a jar file, pass the full path to the jar file. For instance, here is how you compile **MyClass.java** that reference **primer.FileObject** in the **MyLib.jar** file located in **C:\temp** in a Windows system.

```
javac -classpath C:/temp/MyLib.jar MyClass.java
```

This example compiles **MyClass.java** that references classes located in the **/home/user1/lib** directory and packaged in the **Exercises.jar** file located in the **/home/jars** directory in Linux:

```
javac -classpath /home/user1/lib/:/home/user1/Exercises.jar
MyClass.java
```

If you are referencing a class whose root is the same as the class being compiled, you can pass **./** as the value for the classpath. For example, the following command line compiles **MyClass.java** that references both **C:\temp** and the current directory:

```
javac -classpath C:/temp/;./ MyClass.java
```

The alternative to the **classpath** option is to assign the value to the **CLASSPATH** environment variable. However, if the **classpath** option is present, the value of the **CLASSPATH** environment variable will be overridden.
If the **-sourcepath** option is not specified, the user class path is searched for both source files and class files.

`-Djava.ext.dirs=`*directories*
: Override the location of installed extensions.

`-Djava.endorsed.dirs=`*directories*
: Override the location of endorsed standards path.

`-d` *directory*
: Specify the target directory for class files. The target directory must already exist. **javac** puts the class files in a directory structure that reflects the package name, creating directories as needed.
By default, **javac** creates class files in the same directory as the source file.

`-deprecation`
: List each use or override of a deprecated member or class. Without this option, **javac** shows the names of source files that use or override deprecated members or classes. **-deprecation** is shorthand for **-Xlint:deprecation**.

`-encoding` *encoding*
: Specify the source file encoding name, such as UTF-8. By default, **javac** uses the platform default converter.

`-g`
: Print debug information, including local variables. By default, only line number and source file information is generated.

`-g:none`
: Prevent **javac** from generating debug information.

`-g:{`*keyword list*`}`

Generate only some kinds of debug information, specified by a comma separated list of keywords. Valid keywords are:
- **source**. Source file debug information
- **lines**. Line number debug information
- **vars**. Local variable debug information

`-help`
Print a description of standard options.

`-nowarn`
Disable warning messages. This has the same effect as **-Xlint:none**.

`-source` *release*
Specifies the version of source code accepted. The values allowed are **1.3**, **1.4**, **1.5**, **5**, **1.6**, **6**, 1.7, and 7.

`-sourcepath` *sourcePath*
Set the source code path to search for class or interface definitions. As with the user class path, source path entries are separated by semicolons (in Windows) or colons (in Linux/Unix) and can be directories, jar archives, or zip archives. If packages are used, the local path name within the directory or archive must reflect the package name.
Note: Classes found through the classpath are subject to automatic recompilation if their sources are found.

`-verbose`
Include information about each class loaded and each source file compiled.

`-X`
Display information about nonstandard options.

Nonstandard Options

`-Xbootclasspath/p:`*path*
Prepend to the bootstrap class path.

`-Xbootclasspath/a:`*path*
Append to the bootstrap class path.

`-Xbootclasspath/:`*path*
Override location of bootstrap class files.

`-Xlint`
Enable all recommended warnings.

`-Xlint:none`
Disable all warnings not mandated by the Java Language Specification.

`-Xlint:-`*xxx*
Disable warning *xxx*, where *xxx* is one of the warning names supported for **-Xlint:xxx**.

`Xlint:unchecked`
Provide more detail for unchecked conversion warnings that are mandated by the Java Language Specification.

`-Xlint:`*path*
Warn about nonexistent path directories specified in the **classpath**, **sourcepath**, or other option.

`-Xlint:serial`
Warn about missing serialVersionUID definitions on serializable classes.

`-Xlint:finally`
: Warn about **finally** clauses that cannot complete normally.

`-Xlint:fallthrough`
: Check switch blocks for fall-through cases and provide a warning message for any that are found. Fall-through cases are cases in a **switch** block, other than the last case in the block, whose code does not include a **break** statement.

`-Xmaxerrors` *number*
: Specify the maximum number of errors that will be reported

`-Xmaxwarns` *number*
: Specify the maximum number of warnings to be reported.

`-Xstdout` *filename*
: Send compiler messages to the named file. By default, compiler messages go to System.err.

The -J Option

-Joption
: Pass option to the java launcher called by javac. For example, **-J-Xms48m** sets the startup memory to 48 megabytes. Although it does not begin with **-X**, it is not a `standard option' of **javac**. It is a common convention for -J to pass options to the underlying VM executing applications written in Java.

Command Line Argument Files

If you have to pass long arguments to **javac** again and again, you will save a lot typing if you save those arguments in a file and pass the file to **javac** instead. An argument file can include both **javac** options and source filenames in any combination. Within an argument file, you can separate arguments using a space or separate them as new lines. The **javac** tool even allows multiple argument files.

For example, the following command line invokes **javac** and passes the file **MyArguments** to it:

```
javac @MyArguments
```

The following passes two argument files, **Args1** and **Args2**:

```
javac @Args1 @Args2
```

Appendix B
java

The **java** program is a tool for launching a Java program. Its syntax has two forms.

```
java [options] class [argument ...]
java [options] -jar jarFile [argument ...]
```

where *options* represents command-line options, *class* the name of the class to be invoked, *jarFile* the name of the jar file to be invoked, and *argument* the argument passed to the invoked class's **main** method

---

Options

There are two types of options you can pass to **java**, standard and nonstandard.

Standard Options

`-client`
: Select the Java HotSpot Client VM.

`-server`
: Select the Java HotSpot Server VM.

`-agentlib:`*`libraryName[=options]`*
: Load native agent library libraryName. Example values of *libraryName* are **hprof**, **jdwp=help**, and **hprof=help**.

`-agentpath:pathname[=options]`
: Load a native agent library by full pathname.

`-classpath` *`classpath`*
: The same as the **-cp** option.

`-cp` *`classpath`*
: Specify a list of directories, jar archives, and zip archives to search for class files. Two class paths are separated by a colon in Unix/Linux and by a semicolon in Windows. For examples on using -cp and –classpath, see the description of the **javac** tool's **classpath** option in Appendix A.

`-Dproperty=`*`value`*
: Set a system property value.

`-d32`
: See the description of the **–d64** option.

`-d64`
: Specify whether the program is to be run in a 32-bit or a 64-bit environment if available.
Currently only the Java HotSpot Server VM supports 64-bit operation, and the **-server** option is implicit with the use of -d64. This is subject to change

in a future release.
If neither **-d32** nor **-d64** is specified, the default is to run in a 32-bit environment, except for 64-bit only systems. This is subject to change in a future release.

`-enableassertions[:<package name>"..." | :<class name> ]`
See the description for the **–ea** option.

`-ea[:<package name>"..." | :<class name> ]`
Enable assertions. Assertions are disabled by default.

`-disableassertions[:<package name>"..." | :<class name> ]`
See the description for the **–da** option.

`-da[:<package name>"..." | :<class name> ]`
Disable assertions. This is the default.

`-enablesystemassertions`
See the description for the **–esa** option.

`-esa`
Enable asserts in all system classes (set the default assertion status for system classes to **true**).

`-disablesystemassertions`
See the description for the **–dsa** option.

`-dsa`
Disables asserts in all system classes.

`-jar`
Execute a Java class in a jar file. The first argument is the name of the jar file instead of a startup class name. To tell **java** the class to invoke, the manifest of the jar file must contain a line of the form **Main-Class: *classname***, where *classname* identifies the class having the public static void **main(String[] args)** method that serves as your application's starting point.

`-javaagent:jarpath[=options]`
Load a Java programming language agent.

`-verbose`
See the description for the **–verbose:class** option.

`-verbose:class`
Display information about each class loaded.

`-verbose:gc`
Report on each garbage collection event.

`-verbose:jni`
Report information about use of native methods and other Java Native Interface activity.

`-version`
Display the JRE version information and exit.

`-showversion`
Display the version information and continue.

`-?`
See the description for the **–help** option.

`-help`

Display usage information and exit.

`-X`
Display information about nonstandard options and exit.

Nonstandard Options

`-Xint`
Operate in interpreted-only mode. Compilation to native code is disabled, and all bytecodes are executed by the interpreter. You will not be able to enjoy the performance benefits offered by the Java HotSpot VMs' adaptive compiler.

`-Xbatch`
Disable background compilation so that compilation of all methods proceeds as a foreground task until it completes. Without this option, the VM will compile the method as a background task, running the method in interpreter mode until the background compilation is finished.

`-Xdebug`
Start with support for JVMDI enabled. JVMDI has been deprecated and is not used for debugging in Java SE 5 and 6, so this option isn't needed for debugging in Java SE 5 and 6.

`-Xbootclasspath:`*`bootclasspath`*
Specify a list of directories, jar archives, and zip archives to search for boot class files. Entries are separated by colons (in Linux/Unix) or by semicolons (in Windows). These are used in place of the boot class files included in Java 5 and 6.

`-Xbootclasspath/a:`*`path`*
Specify a list of directories, jar archives, and zip archives to append to the default bootstrap class path. Entries are separated by colons (in Linux/Unix) or by semicolons (in Windows).

`-Xbootclasspath/p:`*`path`*
Specify a list of directories, jar archives, and zip archives to prepend in front of the default bootstrap class path. Entries are separated by colons (in Linux/Unix) or by semicolons (in Windows).

`-Xcheck:jni`
Perform additional checks for Java Native Interface (JNI) functions. Specifically, the Java Virtual Machine validates the parameters passed to the JNI function as well as the runtime environment data before processing the JNI request. Any invalid data encountered indicates a problem in the native code, and the JVM will terminate with a fatal error in such cases. Using this option imposes a performance penalty.

`-Xfuture`
Perform strict class-file format checks. For backwards compatibility, the default format checks performed by the Java 2 SDK's virtual machine are no stricter than the checks performed by 1.1.x versions of the JDK software. This flag turns on stricter class-file format checks that enforce closer conformance to the class-file format specification. Developers are encouraged to use this flag when developing new code because the stricter checks will become the default in future releases of the Java application launcher.

`-Xnoclassgc`
Disable class garbage collection.

`-Xincgc`
: Enable the incremental garbage collector. The incremental garbage collector, which is off by default, will reduce the occasional long garbage-collection pauses during program execution. The incremental garbage collector will at times execute concurrently with the program and during such times will reduce the processor capacity available to the program.

`-Xloggc:`*`file`*
: Report on each garbage collection event, as with **-verbose:gc**, but record this data to file. In addition to the information **-verbose:gc** gives, each reported event will be preceded by the time (in seconds) since the first garbage-collection event.
Always use a local file system for storage of this file to avoid stalling the JVM due to network latency. The file may be truncated in the case of a full file system and logging will continue on the truncated file. This option overrides the **-verbose:gc** option if both are present.

`-Xms`*`n`*
: Specify the initial size of the memory allocation pool in bytes. The value must be a multiple of 1024 greater than 1MB. Append the letter **k** or **K** to indicate kilobytes, or **m** or **M** to indicate megabytes. The default value is 2MB. For example:

```
-Xms6291456
-Xms6144k
-Xms6m
```

`-Xmx`*`n`*
: Specify the maximum size of the memory allocation pool in bytes. The value must a multiple of 1024 greater than 2MB. Append **k** or **K** to indicate kilobytes, or **m** or **M** to indicate megabytes. The default value is 64MB. For instance:

```
-Xmx83880000
  -Xmx8192k
  -Xmx86M
```

`-Xprof`
: Profile the running program and send profiling data to standard output. This option is provided as a utility that is useful in program development and should not be used in production.

`-Xrunhprof[:`*`help`*`][:<`*`suboption`*`>=<`*`value`*`>,...]`
: Enable cpu, heap, or monitor profiling. This option is typically followed by a list of comma-separated "<suboption>=<value>" pairs. You can display the list of suboption and their default values by running the command **java -Xrunhprof:help**.

`-Xrs`
: Reduce the use of operating-system signals by the Java virtual machine (JVM).

`-Xss`*`n`*
: Set thread stack size.

`-XX:+UseAltSigs`
: The JVM uses SIGUSR1 and SIGUSR2 by default, which can sometimes conflict with applications that signal-chain SIGUSR1 and SIGUSR2. This option will cause the JVM to use signals other than SIGUSR1 and SIGUSR2 as the default.

Appendix C
jar

jar, short for Java archive, is a tool for packaging Java class files and other related resources into a jar file. The **jar** tool is included in the JDK and initially the reason for its creation was so that an applet class and its related resources could be downloaded with a single HTTP request. Over time, **jar** became the preferred way of packaging any Java classes, not only applets.

The jar format is based on the zip format. As such, you can change the extension of a jar file to **.zip** and view it using a ZIP viewer, such as WinZip. A jar file can also include the META-INF directory for storing package and extension configuration data, including security, versioning, extension and services. jar is also the only format that allows you to digitally sign your code.

This appendix provides the syntax of the **jar** tool and examples of how to use it.

Syntax

You can use **jar** to create, update, extract, and list the content of a jar file. A **jar** command can be used with options, which are explained in the section "Options." Here is the syntax of the **jar** program commands.

To create a jar file, use this syntax.

```
jar c[v0M]f jarFile [-C dir] inputFiles [-Joption]
jar c[v0]mf manifest jarFile [-C dir] inputFiles [-Joption]
jar c[v0M] [-C dir] inputFiles [-Joption]
jar c[v0]m manifest [-C dir] inputFiles [-Joption]
```

To update a jar file, use this syntax.

```
jar u[v0M]f jarFile [-C dir] inputFiles [-Joption]
jar u[v0]mf manifest jarFile [-C dir] inputFiles [-Joption]
jar u[v0M] [-C dir] inputFiles [-Joption]
jar u[v0]m manifest [-C dir] inputFiles [-Joption]
```

To extract a jar file, use this.

```
jar x[v]f jarFile [inputFiles] [-Joption]
jar x[v] [inputFiles] [-Joption]
```

To list the contents of a jar file, use the following syntax.

```
jar t[v]f jarFile [inputFiles] [-Joption]
jar t[v] [inputFiles] [-Joption]
```

And, to add index to a jar file, use this syntax.

```
jar i jarFile [-Joption]
```

The arguments are as follows.

`cuxtiv0Mmf`
: Options that control the **jar** command. These will be detailed in the section "Options."

`jarFile`
: The jar file to be created, updated, extracted, have its contents viewed, or add index to. The absence of the **f** option and *jarFile* indicates that we are accepting input from the standard input (when extracting and viewing the contents) or sending output to the standard output (for creating and updating).

`inputFiles`
: Files or directories, separated by spaces, to be packaged into a jar file (when creating and updating), or to be extracted or listed from *jarFile*. All directories are processed recursively. The files are compressed unless option O (zero) is used.

`manifest`
: Pre-existing manifest file whose **name: value** pairs are to be included in **MANIFEST.MF** in the jar file. The options **m** and **f** must appear in the same order that manifest and jarFile appear.

`-C dir`
: Temporarily changes directories to dir while processing the following *inputFiles* argument. Multiple -C *dir inputFiles* sets are allowed.

`-Joption`
: Option to be passed into the Java runtime environment. (There must be no space between -J and *option*).

Options

The options that can be used in a jar command is as follows.

`c`
: Indicates that the **jar** command is invoked to create a new jar file.

`u`
: Indicates that the **jar** command is invoked to update the specified jar file.

`x`
: Indicates that the **jar** command is invoked to extract the specified jar file. If *inputFiles* is present, only those specified files and directories are extracted. Otherwise, all files and directories are extracted.

`t`
: Indicates that the **jar** command is invoked to list the contents of the specified jar file. If *inputFiles* is present, only those specified files and directories are listed. Otherwise, all files and directories are listed.

`i`
: Generate index information for the specified *jarFile* and its dependent jar files.

`f`
: Specifies the file *jarFile* to be created, updated, extracted, indexed, or viewed.

`v`

Generates verbose output to standard output.

`0`
: This is a zero that indicates that files should be stored without being compressed.

`M`
: Indicates that a manifest file entry should not be created for creation and update. This option also instructs the **jar** tool to delete any manifest during update.

`m`
: Includes **name: value** attribute pairs from the specified manifest file manifest in the file at **META-INF/MANIFEST.MF**. A **name: value** pair is added unless one with the same name already exists, in which case its value is updated.

`-C` *dir*
: Temporarily changes directories (cd dir) during execution of the **jar** command while processing the following *inputFiles* argument.

`-Joption`
: Pass option to the Java runtime environment, where option is one of the options described on the reference page for the java application launcher. For example, **-J-Xmx32M** sets the maximum memory to 32 megabytes.

Examples

The following are examples of how to use **jar**.

Create

This **jar** command packages all directories and files in the current directory into a jar file named **MyJar.jar**.

```
jar cf MyJar.jar *
```

The following, with the **v** option, does the same but outputs all messages to the console:

```
jar cvf MyJar.jar *
```

The following packages all class files in the **com/brainysoftware/jdk/** directory into the **MyJar.jar** file.

```
jar cvf MyJar.jar com/brainysoftware/jdk/*.class
```

Update

This command adds **MathUtil.class** to **MyJar.jar**.

```
jar uf MyJar.jar MathUtil.class
```

This command updates the **MyJar.jar** manifest with the name: value pairs in manifest.

```
jar umf manifest MyJar.jar
```

The following command adds **MathUtil.class** in the **classes** directory to **MyJar.jar**.

```
jar uf MyJar.jar -C classes MathUtil.class
```

List

The following command lists the contents of **MyJar.jar**:

```
jar tf MyJar.jar
```

Extract

The following command extracts all files in **MyJar.jar** to the current directory.

```
jar xf MyJar.jar
```

Index

This command generates in **MyJar.jar** an **INDEX.LIST** file that contains location information for each package in **MyJar.jar** and all the jar files specified in the **Class-Path** attribute of **MyJar.jar**.

```
jar i MyJar.jar
```

Setting an Application's Entry Point

The **java** tool, explained in Appendix B, allows you to invoke a class in a jar file. Here is the syntax:

```
java -jar jarFile
```

For **java** to be able to invoke the correct class, you need to include in the jar file a manifest that has the following entry:

```
Main-Class: className
```

Appendix D
NetBeans

Sun Microsystems launched the NetBeans open source project in 2000. The name NetBeans came from Netbeans Ceska Republika, a Czech company that Sun bought over. The new project was based on the code Sun acquired as the result of the purchase.

This appendix provides a quick tutorial to using NetBeans to build Java applications. NetBeans requires a JDK to work.

Download and Installation

You can download NetBeans free from http://netbeans.org. The latest version at the time of writing is 7.4. You need version 7 or later to enjoy the new features in Java 7. NetBeans is written in Java and, as such, can run on any platform where Java is available. Each distribution includes an installer for easy installation. Make sure you download the correct version for your operating system. The installer guides you through step-by-step instructions that are easy to follow. You will be prompted to agree on the terms and conditions of use, specify the installation directory, and select the JDK version to use if your computer has more than one.

Once installed, you can run the NetBeans IDE just like you would other applications.

Creating a Project

NetBeans organizes resources in projects. Therefore, before you can create a Java class, you must first create a project. To do so, follow these steps.

1. Click **File**, **New Project**. The **New Project** dialog will be displayed (See Figure D.1)

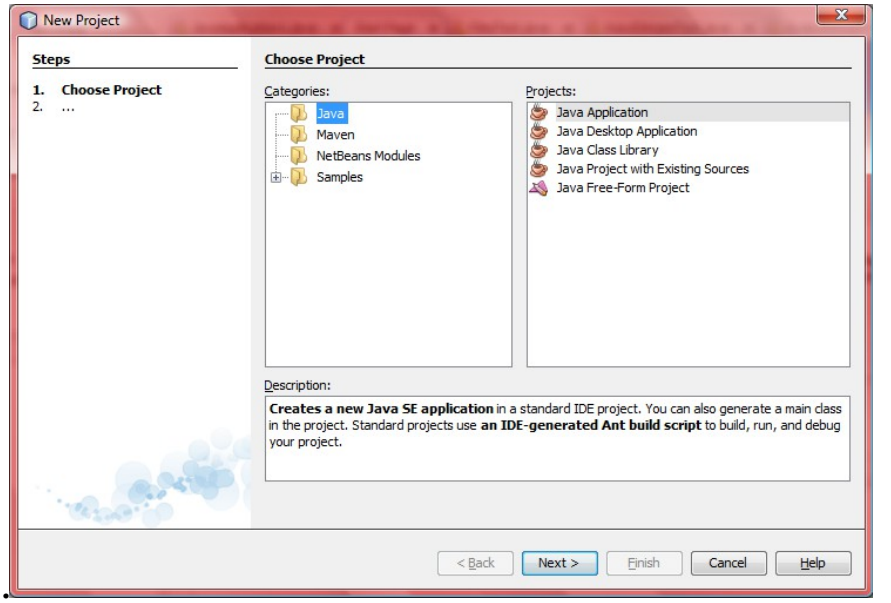

Figure D.1: The New Project dialog

2. Click **Java** from the **Categories** box and **Java Application** from the **Projects** box. And then, click **Next**. The next screen will be displayed, as shown in Figure D.2.
3. Enter a project name in the **Project Name** box and browse to the directory where you want to save the project's resources. Afterwards, click **Finish**. NetBeans will create a new project plus the first class in the project. This is depicted in Figure D.3.

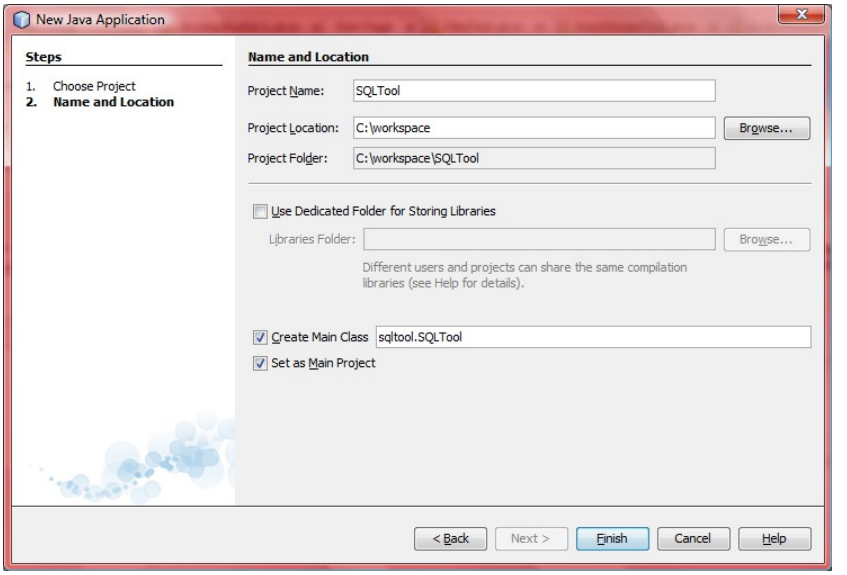

Figure D.2: Select a project name

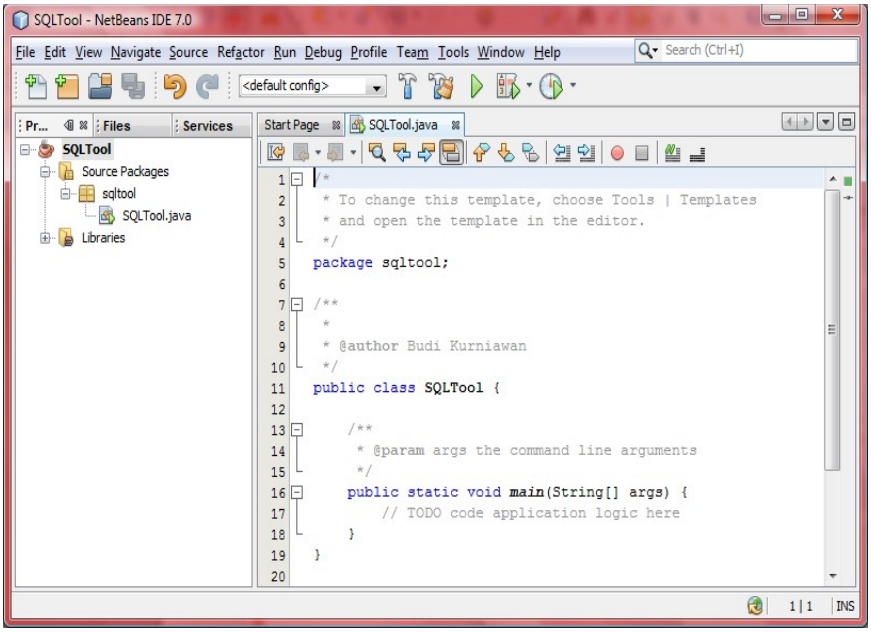

Figure D.3: A Java Project

Figure D.3 shows two windows, the **Projects** window on the left and the source file window on the right. You are ready to write your code.

Creating a Class

To create a class other than that created by default by NetBeans, right-click the project icon in the **Projects** window, then click **New**, **Java Class**. You will see the **New Java Class** dialog like that in Figure D.4.

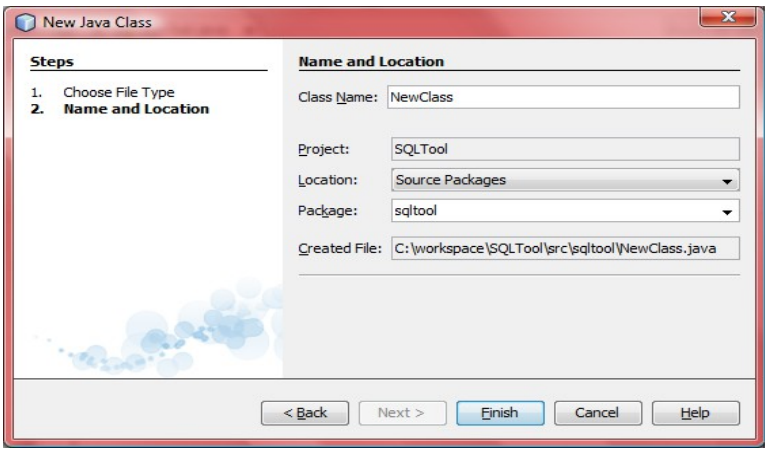

Figure D.4: The New Java Class dialog

Type in the class name and the package for this class, then click **Finish**. A new Java class will be created for you. You can see the new class listed on the **Projects** window.

You can now write your code. As you type, NetBeans will check and correct the syntax of your Java code. You can save your code by clicking **Ctrl+S** and NetBeans automatically compiles it as you save.

Running a Java Class

Once you are finished with a class, you can run it to test it. To run a class, click **Run**, **Run File**, then select the Java class you want to run. Any result will be displayed in the Console window. Another way to run a Java class is to right-click on the source code and click **Run File**.

Also, to run the last run class, press **Shift+F6**.

Adding Libraries

Oftentimes your classes or interfaces reference types in other projects or in a jar file. To compile these classes/interfaces, you need to tell NetBeans where to find the referenced library by adding a reference to it. You do this by right-clicking the **Libraries** icon in the **Projects** window, and then clicking **Add JAR/Folder**. A navigation window will then appear that lets you select your library file.

Debugging Code

A powerful feature offered by many IDE's is support for debugging. In NetBeans, you can step through a program line by line. The steps for debugging a program are as follows.

1. Add a breakpoint. You do this by clicking on the line on your code and click **Toggle Breakpoint**.
2. Execute the program by clicking **Run**, **Run File**, and then select the Java class to debug.

After selecting a class to debug in Step 2, these windows will open: **Watches**, **Call Stack**, and **Local Variables**. They allow you to monitor the progress of your code. The **Local Variables** window, for instance, allows you to inspect the value of a local variable.

To continue, click the **Run** menu and select whether to step into, step over, continue, or pause the program.

Appendix E
Eclipse

IBM launched Eclipse in 2001 after buying Object Technology International, a Canadian company. Including the purchase, IBM spent $40 million on the code that it finally released as an open source project. Written in Java, Eclipse ships with its own compiler, so it does not rely on Oracle's Java compiler. As a result, you don't need a JDK to run Eclipse, just a JRE. In fact, Eclipse comes with compilers for other languages as well, such as C++ and C# because its developers have the ambition to make Eclipse the ultimate IDE.

Another thing to note, even though Eclipse is written in Java, it does not use the Swing technology. It created its own graphics library called the Standard Widget Toolkit in order to make Eclipse look and feel more like a native application. You can still use Eclipse to write Swing applications, though.

This appendix provides a quick tutorial to using Eclipse to build Java applications.

Download and Installation

You can download Eclipse free from http://www.eclipse.org. Only version 3.7 SR 1 and later support Java 7. You should download the latest **release** version. Release versions tend to be more bug-free, even though stable versions offer more new features. Make sure you download the version that will run on your operating system. Currently Eclipse is available on Windows, Linux, Solaris, AIX, Mac OSX, and HP-UX. You can even download the source code.

Eclipse distributions are packaged in a zip or gz file. In addition, if your Internet connection is slow, you can download Eclipse in torrent format.

Installation is a matter of extracting the distribution zip or gz file into a directory. No other steps are necessary. Once you extract, you can run it right away. Windows users can simply double click the **eclipse.exe** program icon. Figure E1 shows Eclipse's main page.

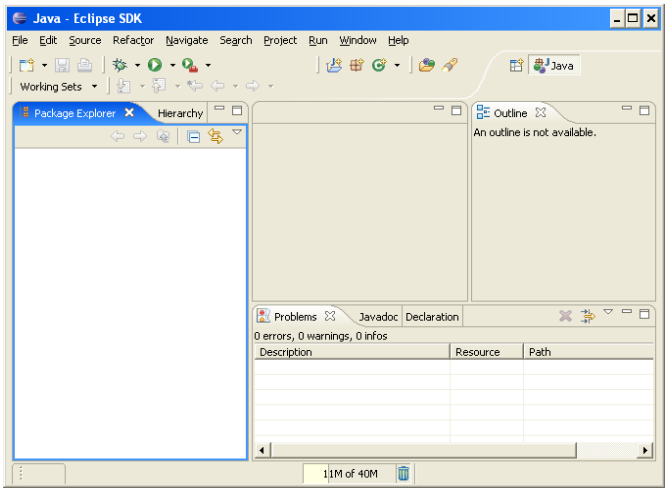

Figure E.1: Eclipse Main Page

Creating a Project

Eclipse organizes resources in projects. Therefore, before you can create a Java class, you must first create a project. To do so, follow these steps.

1. Click **File**, **New**, and Project. The New Project dialog will be displayed (See Figure E.2).

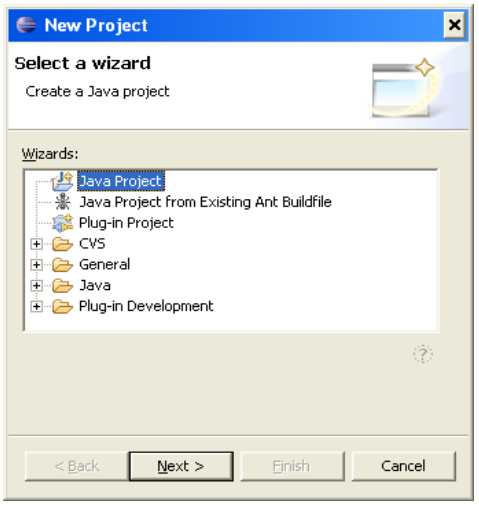

Figure E.2: The New Project dialog

2. Make sure **Java Project** is highlighted in the **New Project** dialog, then click **Next**. You will see the next screen that prompts you for a project name. This screen is shown in Figure E.3.

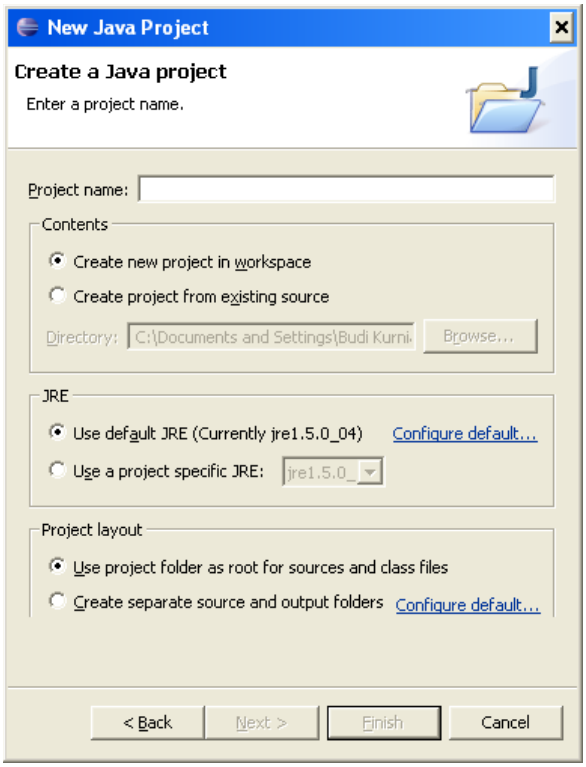

Figure E.3: Select a project name

3. Supply a name for your project. Once you type something in the **Project name** box, the **Next** and **Finish** buttons will become active. Note that by default your project will be created in the default workspace. However, you can choose a different location by clicking the **Create project from existing source** radio button and browsing to a directory in your file system.
4. To proceed you can either click **Next** or **Finish**. Clicking **Finish** uses default settings to create the project, and clicking **Next** allows you to select directories for your source and class files. For now, simply click **Finish**. A project will be created for you. Figure E.4 shows a project named **SQL Tool**.

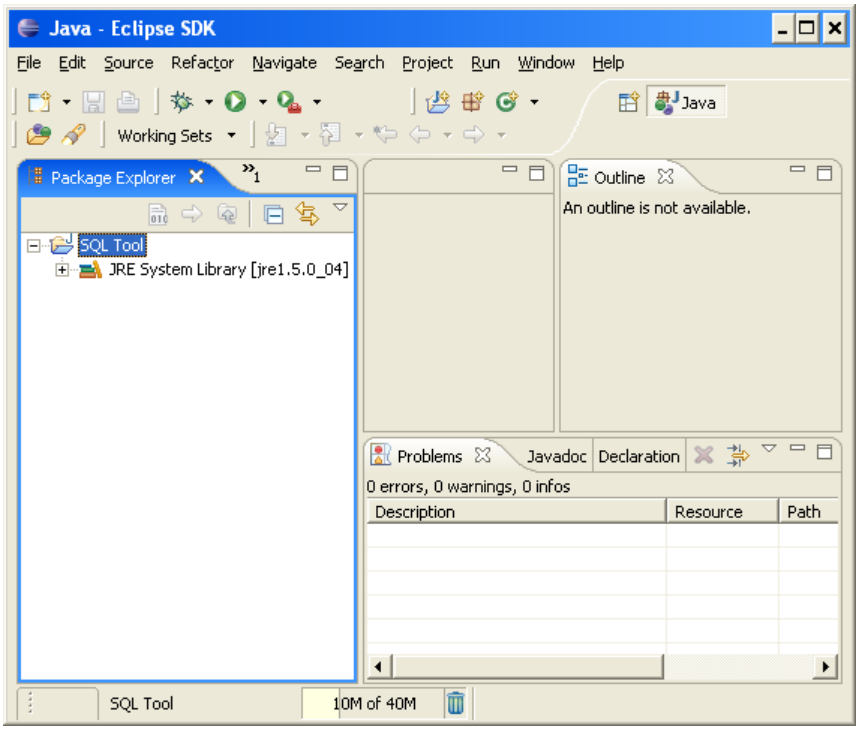

Figure E.4: The Java Perspective

What you see in Figure E.4 is called the Java perspective. A perspective is the combination of views that are suitable for performing a certain task. The Java perspective is used for writing Java code. It consists of the Package Explorer View on the left, the Outline view on the right, and the Problems view at the bottom. The location of each view is changeable by dragging the header of the view. Figure E.4 shows the default position of each view. There are many other views, all of which can be seen by clicking **Window**, **Show View**.

The other perspectives are Java browsing and Debug. You can select a perspective by clicking **Window**, **Open Perspective**.

Creating a Class

To create a class, right-click the project icon in the Package Explorer view, then click **New**, **Class**. You will see the **New Java Class** dialog like the one in Figure E.5.

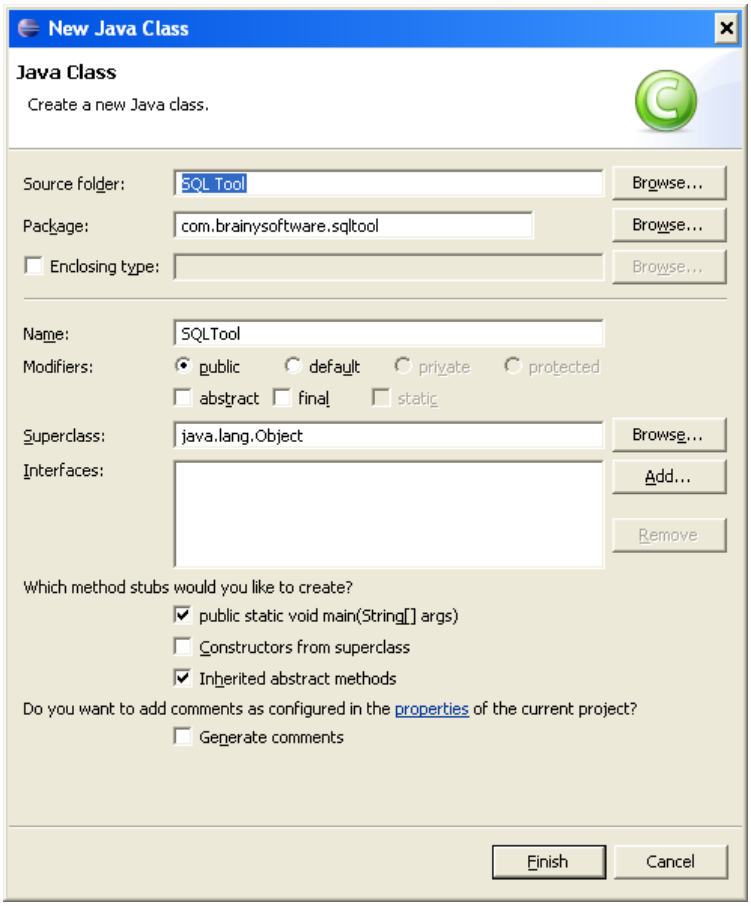

Figure E.5: The New Java Class dialog

Enter the package and the class name, then click **Finish** to create a class. The Java perspective will display the class code in a new pane, as shown in Figure E.6.

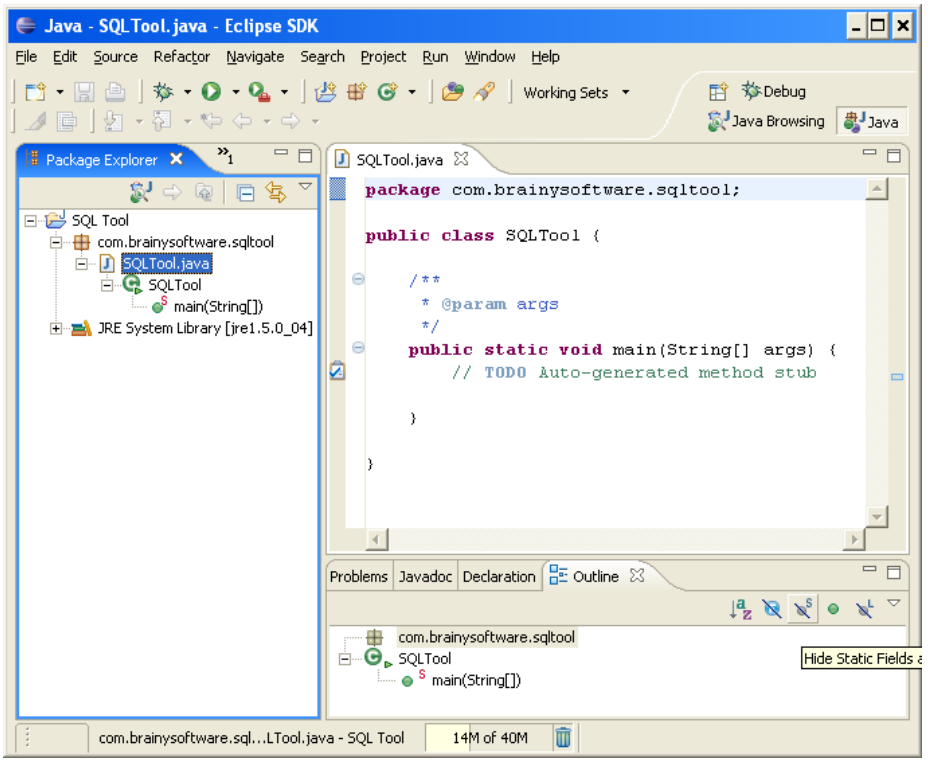

Figure E.6: Editing the Java class

You can now write your code. As you type, Eclipse will check and correct the syntax of your Java code. You can save by clicking **Ctrl+S** and Eclipse automatically compiles it as you save.

Running a Java Class

Once you are finished with a class, you can run it to test it. To run a class, click **Run**, **Run As**, **Java Application**. Any result will be displayed in the Console view. Another way to run a Java class is to right-click on the class pane and click **Run As**, **Java Application**.

Also, to run the last run class, press **Ctrl+F11**.

Adding Libraries

Oftentimes your classes or interfaces reference types in a jar file or in another project. To compile these classes/interfaces, you need to tell Eclipse where to find the library by adding a reference to it. You do this by clicking **Project**, **Properties**. The Properties window will appear.

Click **Java Build Path** on the right pane and then click the **Libraries** tab on the left. Then, click **Add External JARs** and navigate to select the jar file. If the referenced types are in another project, click the **Projects** tab and add the required project.

Debugging Code

A powerful feature offered by many IDE's is support for debugging. In Eclipse, you can step through a program line by line. The steps for debugging a program are as follows.

1. Add a breakpoint. You do this by clicking on the line on your code and click **Run**, **Toggle Line Breakpoint**.
2. Execute the program by clicking **Run**, **Debug As**, **Java Application**.

Debugging requires the Debug perspective be open. After clicking **Java Application** in Step 2, a window will appear that asks you if you want to switch to the Debug perspective. Click **Yes**, and you will see the Debug perspective like that in Figure E.7.

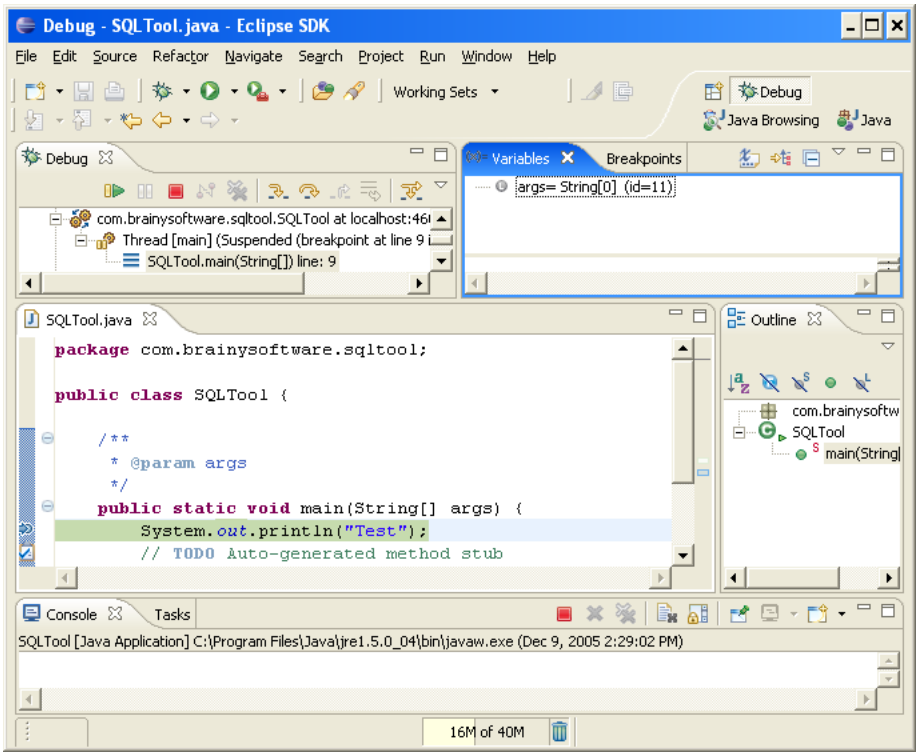

Figure E.7: The Debug Perspective

A useful view that appears is the **Variables** view. It displays the list of variables in your program and lets you inspect their values.

To continue, click the **Run** menu and select whether to step into, step over, resume, or terminate the program.

Index