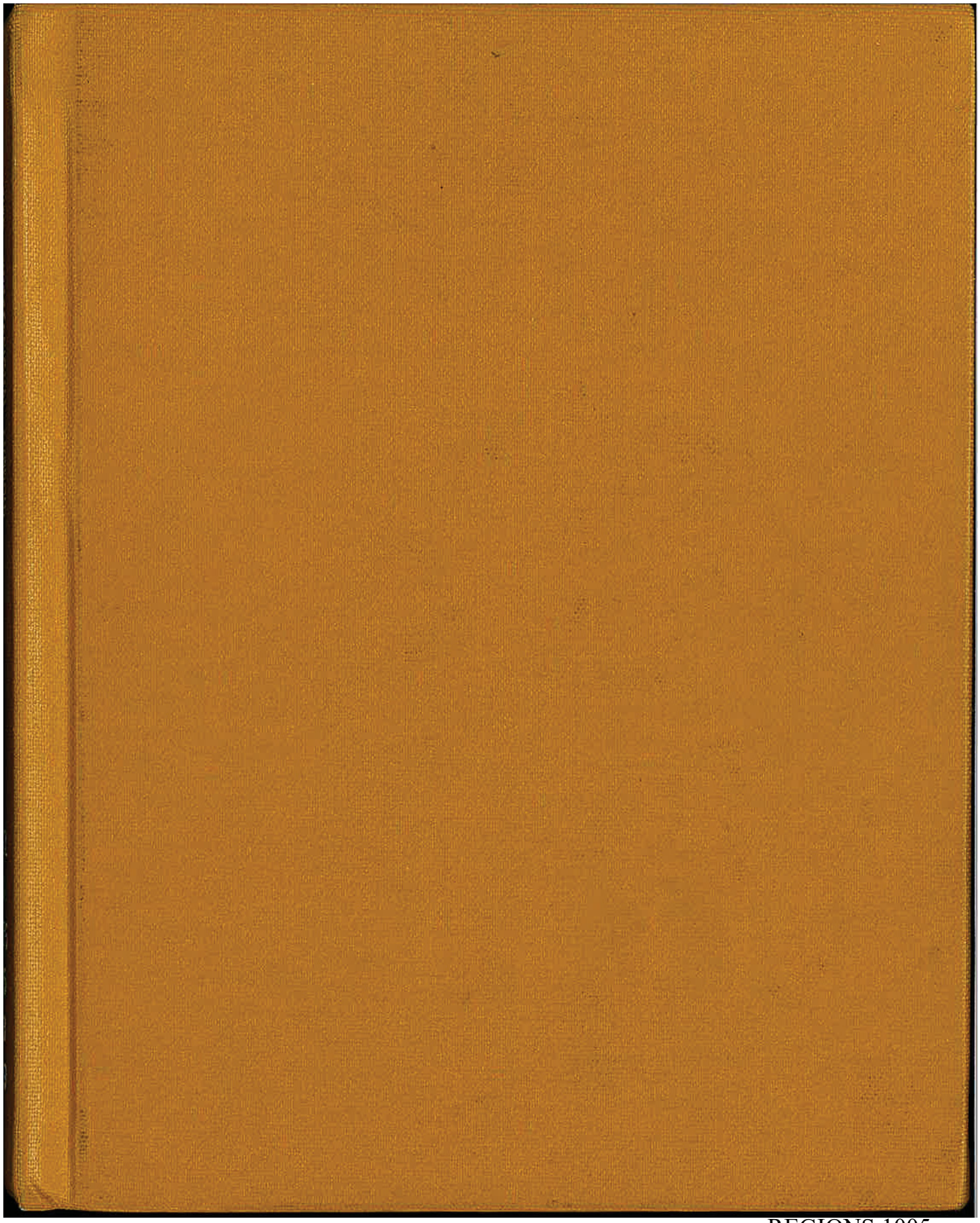


REGIONS 1005
PART 1 OF 3

MEIER PROFESSIONAL ANDROID APPLICATION DEVELOPMENT

AAEL
QA
76.76
.A65
M43
2009


THE UNIVERSITY OF MICHIGAN
LIBRARIES

Wrox Programmer to Programmer™



Professional

Android™ Application Development

Reto Meier

WROX™

Updates, source code, and Wrox technical support at **www.wrox.com**



Programmer to Programmer™

Get more out of WROX.com

Interact

Take an active role online by participating in our P2P forums

Wrox Online Library

Hundreds of our books are available online through Books24x7.com

Wrox Blox

Download short informational pieces and code to keep you up to date and out of trouble!

Chapters on Demand

Purchase individual book chapters in pdf format

Join the Community

Sign up for our free monthly newsletter at newsletter.wrox.com

Browse

Ready for more Wrox? We have books and e-books available on .NET, SQL Server, Java, XML, Visual Basic, C#/ C++, and much more!

UNIVERSITY OF MICHIGAN LIBRARIES

Contact Us.

We always like to get feedback from our readers. Have a book idea? Need community support? Let us know by e-mailing **wrox-partnerwithus@wrox.com**

Professional Android™ Application Development

Introduction xvii
Chapter 1: Hello, Android 1
Chapter 2: Getting Started 19
Chapter 3: Creating Applications and Activities 45
Chapter 4: Creating User Interfaces 75
Chapter 5: Intents, Broadcast Receivers, Adapters, and the Internet 113
Chapter 6: Data Storage, Retrieval, and Sharing 159
Chapter 7: Maps, Geocoding, and Location-Based Services 207
Chapter 8: Working in the Background 249
Chapter 9: Peer-to-Peer Communication 279
Chapter 10: Accessing Android Hardware 315
Chapter 11: Advanced Android Development 353
Index 399

Professional
Android™ Application Development

Reto Meier



Wiley Publishing, Inc.

Professional Android™ Application Development

Published by
Wiley Publishing, Inc.
10475 Crosspoint Boulevard
Indianapolis, IN 46256
www.wiley.com

Copyright © 2009 by Wiley Publishing, Inc., Indianapolis, Indiana

Published simultaneously in Canada

ISBN: 978-0-470-34471-2

Manufactured in the United States of America

10 9 8 7 6 5 4 3 2 1

Library of Congress Cataloging-in-Publication Data is available from the publisher.

No part of this publication may be reproduced, stored in a retrieval system or transmitted in any form or by any means, electronic, mechanical, photocopying, recording, scanning or otherwise, except as permitted under Sections 107 or 108 of the 1976 United States Copyright Act, without either the prior written permission of the Publisher, or authorization through payment of the appropriate per-copy fee to the Copyright Clearance Center, 222 Rosewood Drive, Danvers, MA 01923, (978) 750-8400, fax (978) 646-8600. Requests to the Publisher for permission should be addressed to the Legal Department, Wiley Publishing, Inc., 10475 Crosspoint Blvd., Indianapolis, IN 46256, (317) 572-3447, fax (317) 572-4355, or online at http://www.wiley.com/go/permissions.

Limit of Liability/Disclaimer of Warranty: The publisher and the author make no representations or warranties with respect to the accuracy or completeness of the contents of this work and specifically disclaim all warranties, including without limitation warranties of fitness for a particular purpose. No warranty may be created or extended by sales or promotional materials. The advice and strategies contained herein may not be suitable for every situation. This work is sold with the understanding that the publisher is not engaged in rendering legal, accounting, or other professional services. If professional assistance is required, the services of a competent professional person should be sought. Neither the publisher nor the author shall be liable for damages arising herefrom. The fact that an organization or Web site is referred to in this work as a citation and/or a potential source of further information does not mean that the author or the publisher endorses the information the organization or Web site may provide or recommendations it may make. Further, readers should be aware that Internet Web sites listed in this work may have changed or disappeared between when this work was written and when it is read.

For general information on our other products and services please contact our Customer Care Department within the United States at (800) 762-2974, outside the United States at (317) 572-3993 or fax (317) 572-4002.

Trademarks: Wiley, the Wiley logo, Wrox, the Wrox logo, Wrox Programmer to Programmer, and related trade dress are trademarks or registered trademarks of John Wiley & Sons, Inc. and/or its affiliates, in the United States and other countries, and may not be used without written permission. Android is a trademark of Google, Inc. All other trademarks are the property of their respective owners. Wiley Publishing, Inc., is not associated with any product or vendor mentioned in this book.

Wiley also publishes its books in a variety of electronic formats. Some content that appears in print may not be available in electronic books.

To Kris

About the Author

Originally from Perth, Western Australia, **Reto Meier** now lives in London.

Reto is an experienced software developer with more than 10 years of experience in GUI application architecture, design, and development. He's worked in various industries, including offshore oil and gas, before moving to London and into finance.

Always interested in emerging technologies, Reto has been involved in Android since the initial release in 2007. In his spare time, he tinkers with a wide range of development platforms including WPF and Google's plethora of developer tools.

You can check out Reto's web site, The Radioactive Yak, at `http://blog.radioactiveyak.com`.

About the Tech Editor

Dan Ulery is a software engineer with experience in .NET, Java, and PHP development, as well as in deployment engineering. He graduated from the University of Idaho with a bachelor of science degree in computer science and a minor in mathematics.

Credits

Executive Editor
Chris Webb

Development Editor
William Bridges

Technical Editor
Daniel Ulery

Senior Production Editor
Debra Banninger

Copy Editor
Cate Caffrey

Editorial Manager
Mary Beth Wakefield

Production Manager
Tim Tate

Vice President and Executive Group Publisher
Richard Swadley

Vice President and Executive Publisher
Joseph B. Wikert

Project Coordinator, Cover
Lynsey Stanford

Compositor
James D. Kramer, Happenstance Type-O-Rama

Proofreader
Nancy Carrasco

Indexer
Jack Lewis

Acknowledgments

A big thank you goes to the Android team, particularly those of you who've contributed to the Android developer Google Groups, for creating and supporting an exciting new playground.

I also thank Philipp Lenssen for providing an inspiration, and occasional venue, for my blogging efforts; Chris Webb for reading a blog and seeing an author; and Bill Bridges, Dan Ulery, and the Wrox team for helping me along the way.

Thanks also to Paul, Stu, and Mike: Your friendship and inspiration helped me get to where I am.

Most importantly, I'd like to thank Kristy. For everything.

Contents

Introduction xvii

Chapter 1: Hello, Android 1

A Little Background 2
The Not So Distant Past 2
The Future 3
What It Isn't 3
An Open Platform for Mobile Development 4
Native Android Applications 4
Android SDK Features 5
Access to Hardware including Camera, GPS, and Accelerometer 6
Native Google Maps, Geocoding, and Location-Based Services 6
Background Services 6
SQLite Database for Data Storage and Retrieval 7
Shared Data and Interapplication Communication 7
P2P Services with Google Talk 7
Extensive Media Support and 2D/3D Graphics 8
Optimized Memory and Process Management 8
Introducing the Open Handset Alliance 8
What Does Android Run On? 9
Why Develop for Android? 9
What Will Drive Android Adoption? 10
What Does It Have That Others Don't? 10
Changing the Mobile Development Landscape 11
Introducing the Development Framework 11
What Comes in the Box 12
Understanding the Android Software Stack 12
The Dalvik Virtual Machine 14
Android Application Architecture 14
Android Libraries 15
Advanced Android Libraries 16
Summary 17

Contents

Chapter 2: Getting Started 19

Developing for Android 20
What You Need to Begin 20
Creating Your First Android Activity 24
Types of Android Applications 29
Developing for Mobile Devices 30
Hardware-Imposed Design Considerations 30
Considering the Users' Environment 33
Developing for Android 34
To-Do List Example 37
Android Development Tools 42
The Android Emulator 42
Dalvik Debug Monitor Service (DDMS) 43
The Android Debug Bridge (ADB) 43
Summary 44

Chapter 3: Creating Applications and Activities 45

What Makes an Android Application? 46
Introducing the Application Manifest 46
Using the Manifest Editor 49
The Android Application Life Cycle 50
Understanding Application Priority and Process States 51
Externalizing Resources 52
Creating Resources 53
Using Resources 59
To-Do List Resources Example 62
Creating Resources for Different Languages and Hardware 63
Runtime Configuration Changes 64
A Closer Look at Android Activities 66
Creating an Activity 66
The Activity Life Cycle 68
Android Activity Classes 73
Summary 73

Chapter 4: Creating User Interfaces 75

Fundamental Android UI Design 76
Introducing Views 76
Creating Activity User Interfaces with Views 77
The Android Widget Toolbox 78

Introducing Layouts **79**
Using Layouts 79
Creating New Views **80**
Modifying Existing Views 81
Creating Compound Controls 85
Creating Custom Widgets and Controls 88
Using Custom Controls 98
Creating and Using Menus **99**
Introducing the Android Menu System 99
Defining an Activity Menu 101
Dynamically Updating Menu Items 104
Handling Menu Selections 104
Submenus and Context Menus 105
To-Do List Example Continued 107
Summary **112**

Chapter 5: Intents, Broadcast Receivers, Adapters, and the Internet **113**

Introducing Intents **114**
Using Intents to Launch Activities 114
Using Intent Filters to Service Implicit Intents 121
Using Intent Filters for Plug-ins and Extensibility 130
Using Intents to Broadcast Events 132
Introducing Adapters **136**
Introducing Some Android-Supplied Adapters 136
Using Adapters for Data Binding 136
Using Internet Resources **141**
Connecting to an Internet Resource 142
Leveraging Internet Resources 143
Introducing Dialogs **143**
Introducing the `Dialog` Class 144
Using Activities as Dialogs 147
Creating an Earthquake Viewer **148**
Summary **157**

Chapter 6: Data Storage, Retrieval, and Sharing **159**

Android Techniques for Saving Data **160**
Saving Simple Application Data **160**
Creating and Saving Preferences 161
Retrieving Shared Preferences 161
Saving the Activity State 162
Creating a Preferences Page for the Earthquake Viewer 165

Saving and Loading Files 174
Including Static Files as Resources 174
File Management Tools 175
Databases in Android 175
Introducing SQLite 176
Cursors and Content Values 176
Working with Android Databases 177
Introducing Content Providers 189
Using Content Providers 189
Native Android Content Providers 192
Creating a New Content Provider 194
Creating and Using an Earthquake Content Provider 197
Summary 205

Chapter 7: Maps, Geocoding, and Location-Based Services 207

Using Location-Based Services 208
Setting up the Emulator with Test Providers 208
Updating Locations in Emulator Location Providers 208
Create an Application to Manage Test Location Providers 209
Selecting a Location Provider 212
Finding the Available Providers 212
Finding Providers Based on Requirement Criteria 212
Finding Your Location 213
"Where Am I?" Example 214
Tracking Movement 216
Updating Your Location in "Where Am I?" 217
Using Proximity Alerts 219
Using the Geocoder 220
Reverse Geocoding 221
Forward Geocoding 221
Geocoding "Where Am I?" 222
Creating Map-Based Activities 224
Introducing MapView and MapActivity 224
Creating a Map-Based Activity 224
Configuring and Using Map Views 226
Using the Map Controller 227
Mapping "Where Am I?" 228
Creating and Using Overlays 231
Introducing MyLocationOverlay 239
Introducing ItemizedOverlays and OverlayItems 239
Pinning Views to the Map and Map Positions 240

Mapping Earthquakes Example 242
Summary 247

Chapter 8: Working in the Background 249

Introducing Services 250
Creating and Controlling Services 250
Binding Activities to Services 258
Using Background Worker Threads 259
Creating New Threads 260
Synchronizing Threads for GUI Operations 260
Moving the Earthquake Service to a Background Thread 261
Let's Make a Toast 262
Customizing Toasts 263
Using Toasts in Worker Threads 264
Introducing Notifications 265
Introducing the Notification Manager 266
Creating Notifications 266
Triggering Notifications 267
Adding Notifications to the Earthquake Monitor 267
Advanced Notification Techniques 270
Ongoing and Insistent Notifications 272
Using Alarms 273
Using Alarms to Update Earthquakes 274
Summary 276

Chapter 9: Peer-to-Peer Communication 279

Introducing Android Instant Messaging 280
Using the GTalk Service 280
Binding to the GTalk Service 281
Making a GTalk Connection and Starting an IM Session 282
Introducing Presence and the Contact Roster 283
Managing Chat Sessions 286
Sending and Receiving Data Messages 289
Introducing SMS 291
Using SMS in Your Application 291
Sending SMS Messages 292
Listening for SMS Messages 294
Emergency Responder SMS Example 297
Automating the Emergency Responder 306
Summary 314

Contents

Chapter 10: Accessing Android Hardware 315

Using the Media APIs 316
Playing Media Resources 316
Recording Multimedia 317
Using the Camera 319
Controlling Camera Settings 319
Using the Camera Preview 320
Taking a Picture 320
Introducing the Sensor Manager 321
Using the Accelerometer and Compass 323
Introducing Accelerometers 324
Detecting Acceleration Changes 324
Creating a Speedometer 326
Determining Your Orientation 329
Creating a Compass and Artificial Horizon 330
Android Telephony 333
Making Phone Calls 334
Monitoring Phone State and Phone Activity 334
Monitoring Data Connectivity and Activity 337
Accessing Phone Properties and Status 338
Controlling the Phone 338
Using Bluetooth 339
Introducing the Bluetooth Service 339
Controlling the Local Bluetooth Device 340
Discovering and Bonding with Bluetooth Devices 340
Managing Bluetooth Connections 342
Communication with Bluetooth 342
Using a Bluetooth Headset 344
Managing Network and Wi-Fi Connections 345
Monitoring and Managing Your Internet Connectivity 345
Managing Active Connections 346
Managing Your Wi-Fi 347
Controlling Device Vibration 350
Summary 351

Chapter 11: Advanced Android Development 353

Paranoid Android 354
Linux Kernel Security 354
Introducing Permissions 354
Declaring and Enforcing Permissions 355
Enforcing Permissions with Broadcasting Intents 355

Using AIDL to Support IPC for Services **356**
Implementing an AIDL Interface 356
Using Internet Services **361**
Building Rich User Interfaces **361**
Working with Animations 361
Using Themes to Skin Your Applications 372
Advanced Canvas Drawing 373
Introducing `SurfaceView` 390
Creating Interactive Controls 393
Summary **398**

Index **399**

Introduction

Now is an exciting time for mobile developers. Mobile phones have never been more popular, and powerful smartphones are now a regular choice for consumers. Stylish and versatile phones packing hardware features like GPS, accelerometers, and touch screens are an enticing platform upon which to create innovative mobile applications.

Android hardware will be designed to tempt consumers, but the real win is for developers. With existing mobile development built on proprietary operating systems that restrict third-party applications, Android offers an open and equal alternative. Without artificial barriers, Android developers are free to write applications that take full advantage of increasingly powerful mobile hardware. As a result, developer interest in Android devices has made their 2008 release a hugely anticipated mobile technology event.

Built on an open source framework, and featuring powerful SDK libraries and an open philosophy, Android has opened mobile phone development to thousands of developers who haven't had access to tools for building mobile applications. Experienced mobile developers can now expand into the Android platform, leveraging the unique features to enhance existing products or create innovative new ones.

This book is a hands-on guide to building mobile applications using version 1.0 of the Android software development kit. Chapter by chapter, it takes you through a series of sample projects, each introducing new features and techniques to get the most out of Android. It covers all the basic functionality as well as exploring the advanced features through concise and useful examples.

Since Android is a brand-new, version 1 product, there are only a small number of handsets currently available that support it. As with any early release, there are likely to be regular changes and improvements to the software and development libraries. The explanations and examples included in this book will give the grounding and knowledge you need to write compelling mobile applications using the current SDK, along with the flexibility to quickly adapt to future enhancements.

Whom This Book Is For

This book is for anyone interested in creating applications for the Android mobile phone platform. It includes information that will be valuable, whether you're an experienced mobile developer or making your first foray, via Android, into writing mobile applications.

It will help if readers have used mobile phones (particularly phones running Android), but it's not necessary, nor is prior experience in mobile phone development. It's expected that you'll have some experience in software development and be familiar with basic development practices. While knowledge of Java is helpful, it's not a necessity.

Chapters 1 and 2 introduce mobile development and contain instructions to get you started in Android. Beyond that, there's no requirement to read the chapters in order, although a good understanding of the core components described in Chapters 3 through 6 is important before you venture into the remaining chapters. Chapters 7 through 11 cover a variety of optional and advanced functionality and can be read in whatever order interest or need dictates.

What This Book Covers

Chapter 1 introduces Android, including what it is and how it fits into existing mobile development. What Android offers as a development platform and why it's an exciting opportunity for creating mobile phone applications are then examined in greater detail.

Chapter 2 covers some best practices for mobile development and explains how to download the Android SDK and start developing applications. It also introduces the Android developer tools and demonstrates how to create new applications from scratch.

Chapters 3 through 6 take an in-depth look at the fundamental Android application components. Starting with examining the pieces that make up an Android application and its life cycle, you'll quickly move on to the application manifest and external resources before learning about "Activities," their lifetimes, and their life cycles.

You'll then learn how to create User Interfaces with layouts and Views, before being introduced to the Intent mechanism used to perform actions and send messages between application components. Internet resources are then covered before a detailed look at data storage, retrieval, and sharing. You'll start with the preference-saving mechanism before moving on to file handling and databases. This section finishes with a look at sharing application data using Content Providers.

Chapters 7 to 10 look at more advanced topics. Starting with maps and location-based services, you'll move on to Services, background Threads, and using Notifications.

Android's communication abilities are next, including sending and receiving messages through instant messaging and SMS. Hardware is then covered, starting with media recording and playback, before introducing the camera, accelerometers, and compass sensors. Chapter 10 concludes with a look at phone and networking hardware, starting with telephony APIs and going on to Bluetooth and network management (both Wi-Fi and mobile data connections).

Chapter 11 includes several advanced development topics, among them security, IPC, advanced graphics techniques, and user–hardware interactions.

How This Book Is Structured

This book is structured in a logical sequence to help readers of different development backgrounds learn how to write advanced Android applications.

There's no requirement to read each chapter sequentially, but several of the sample projects are developed over the course of several chapters, adding new functionality and other enhancements at each stage.

Experienced mobile developers with a working Android development environment can skim the first two chapters — which are an introduction to mobile development and instructions for creating your development environment — and dive in at Chapters 3 to 6. These cover the fundamentals of Android development, so it's important to have a solid understanding of the concepts they describe. With this

covered, you can move on to the remaining chapters, which look at maps, location-based Services, background applications, and more advanced topics such as hardware interaction and netwoking.

What You Need to Use This Book

To use the code samples in this book, you will need to create an Android development environment by downloading the Android SDK libraries and developer tools and the Java development kit. You may also wish to download and install Eclipse and the Android Developer Tool plug-in to ease your development, but neither is a requirement.

Android development is supported in Windows, MacOS, and Linux, with the SDK available from the Android web site.

You do not need an Android device to use this book or develop Android applications.

Chapter 2 outlines these requirements in more detail as well as describing where to download and how to install each component.

Conventions

To help you get the most from the text and keep track of what's happening, I've used various conventions throughout the book.

Notes, tips, hints, tricks, and asides to the current discussion are offset and placed in italics like this.

As for styles in the text:

- I show URLs and code within the text like so: `persistence.properties`.
- I present code in two different ways:

```
I use a monofont type with no highlighting for most code examples.
```

```
I use gray highlighting to emphasize code that's particularly important in
the present context.
```

- In some code samples, you'll see lines marked as follows:

```
[… previous code goes here …]
```

or

```
[… implement something here …]
```

This represents an instruction to replace the entire line (including the square brackets) with actual code, either from a previous code snippet in the former case, or your own implementation in the latter.

Source Code

As you work through the examples in this book, you may choose either to type in all the code manually or to use the source code files that accompany the book. All of the source code used in this book is available for download at `www.wrox.com`. Once at the site, simply locate the book's title (either by using the Search box or by using one of the title lists), and click the Download Code link on the book's detail page to obtain all the source code for the book.

Because many books have similar titles, you may find it easiest to search by ISBN; this book's ISBN is 978-0-470-34471-2.

Once you download the code, just decompress it with your favorite compression tool. Alternatively, you can go to the main Wrox code download page at `www.wrox.com/dynamic/books/download.aspx` to see the code available for this book and all other Wrox books.

Errata

We make every effort to ensure that there are no errors in the text or in the code. However, no one is perfect, and mistakes do occur. If you find an error in one of our books, like a spelling mistake or faulty piece of code, we would be very grateful for your feedback. By sending in errata you may save another reader hours of frustration, and at the same time you will be helping us provide even higher quality information.

To find the errata page for this book, go to `www.wrox.com` and locate the title using the Search box or one of the title lists. Then, on the book details page, click the Book Errata link. On this page, you can view all errata that have been submitted for this book and posted by Wrox editors. A complete book list including links to each book's errata is also available at `www.wrox.com/misc-pages/booklist.shtml`.

If you don't spot "your" error on the Book Errata page, go to `www.wrox.com/contact/techsupport.shtml` and complete the form there to send us the error you have found. We'll check the information and, if appropriate, post a message to the book's Errata page and fix the problem in subsequent editions of the book.

p2p.wrox.com

For author and peer discussion, join the P2P forums at `p2p.wrox.com`. The forums are a Web-based system for you to post messages relating to Wrox books and related technologies and interact with other readers and technology users. The forums offer a subscription feature to e-mail you topics of interest of your choosing when new posts are made to the forums. Wrox authors, editors, other industry experts, and your fellow readers are present on these forums.

At `http://p2p.wrox.com`, you will find a number of different forums that will help you not only as you read this book, but also as you develop your own applications. To join the forums, just follow these steps:

1. Go to `p2p.wrox.com` and click the Register link.
2. Read the terms of use and click Agree.
3. Complete the required information to join as well as any optional information you wish to provide, and click Submit.
4. You will receive an e-mail with information describing how to verify your account and complete the joining process.

You can read messages in the forums without joining P2P, but in order to post your own messages, you must join.

Once you join, you can post new messages and respond to messages other users post. You can read messages at any time on the Web. If you would like to have new messages from a particular forum e-mailed to you, click the "Subscribe to This Forum" icon by the forum name in the forum listing.

For more information about how to use the Wrox P2P, be sure to read the P2P FAQs for answers to questions about how the forum software works as well as many common questions specific to P2P and Wrox books. To read the FAQs, click the FAQ link on any P2P page.

1

Hello, Android

Whether you're an experienced mobile engineer, a desktop or web developer, or a complete programming novice, Android represents an exciting new opportunity to write innovative applications for mobile devices.

Despite the name, Android will not help you create an unstoppable army of emotionless robot warriors on a relentless quest to cleanse the earth of the scourge of humanity. Instead, *Android* is an open source software stack that includes the operating system, middleware, and key applications along with a set of API libraries for writing mobile applications that can shape the look, feel, and function of mobile handsets.

Small, stylish, and versatile, modern mobile phones have become powerful tools that incorporate cameras, media players, GPS systems, and touch screens. As technology has evolved, mobile devices have become about more than simply making calls, but their software and development platforms have struggled to keep pace.

Until recently, mobile phones were largely closed environments built on proprietary operating systems that required proprietary development tools. The phones themselves often prioritized native applications over those written by third parties. This has introduced an artificial barrier for developers hoping to build on increasingly powerful mobile hardware.

In Android, native and third-party applications are written using the same APIs and executed on the same run time. These APIs feature hardware access, location-based services, support for background services, map-based activities, relational databases, interdevice peer-to-peer messaging, and 2D and 3D graphics.

Using this book, you will learn how to use these APIs to create your own Android applications. In this chapter, you'll learn some mobile development guidelines and be introduced to the features available from the Android development platform.

Android has powerful APIs, excellent documentation, a thriving developer community, and no development or distribution costs. As mobile devices continue to increase in popularity, this is an exciting opportunity to create innovative mobile phone applications no matter what your development background.

A Little Background

In the days before Twitter and Facebook, when Google was still a twinkle in its founders' eyes and dinosaurs roamed the earth, mobile phones were just that — portable phones small enough to fit inside a briefcase, featuring batteries that could last up to several hours; they offered the freedom to make calls without being physically connected to a landline.

Increasingly small, stylish, and powerful mobile phones are now as ubiquitous as they are indispensable. Hardware advancements have made mobiles smaller and more efficient while including an increasing number of peripherals.

Beginning with cameras and media players, mobiles now include GPS systems, accelerometers, and touch screens. While these hardware innovations should prove fertile ground for software development, the applications available for mobile phones have generally lagged behind the hardware.

The Not So Distant Past

Historically, developers, generally coding in low-level C or C++, have needed to understand the specific hardware they were coding for, generally a single device or possibly a range of devices from a single manufacturer. As hardware technology has advanced, this closed approach has struggled to keep pace.

More recently, platforms like Symbian have been created to provide developers a wider target audience. These systems have proved more successful in encouraging mobile developers to provide rich applications that better leverage the hardware available.

These platforms offer some access to the device hardware, but require writing complex C/C++ code and making heavy use of proprietary APIs that are notoriously difficult to use. This difficulty is amplified when developing applications that must work on different hardware implementations and is particularly true when developing for a particular hardware feature like GPS.

In recent years, the biggest advance in mobile phone development has been the introduction of Java-hosted MIDlets. MIDlets are executed on a Java virtual machine, abstracting the underlying hardware and letting developers create applications that run on the wide variety of hardware that supports the Java run time. Unfortunately, this convenience comes at the price of restricted access to the device hardware.

In mobile development, it's considered normal for third-party applications to receive different hardware access and execution rights compared to native applications written by the phone manufacturers, with MIDlets often receiving few of either.

The introduction of Java MIDlets has expanded developers' audiences, but the lack of low-level hardware access and sandboxed execution have meant that most mobile applications are desktop programs designed to run on a smaller screen rather than take advantage of the inherent mobility of the handheld platform.

The Future

Android sits alongside a new wave of mobile operating systems designed for increasingly powerful mobile hardware. Windows Mobile and Apple's iPhone now provide a richer, simplified development environment for mobile applications. However, unlike Android, they're built on proprietary operating systems that often prioritize native applications over those created by third parties and restrict communication among applications and native phone data. Android offers new possibilities for mobile applications by offering an open development environment built on an open source Linux kernel. Hardware access is available to all applications through a series of API libraries, and application interaction, while carefully controlled, is fully supported.

In Android, all applications have equal standing. Third-party and native Android applications are written using the same APIs and are executed on the same run time. Users can remove and replace any native application with a third-party developer alternative; even the dialer and home screens can be replaced.

What It Isn't

As a disruptive addition to a mature field, it's not hard to see why there has been some confusion about what exactly Android is. Android is **not**:

- **A Java ME implementation** Android applications are written using the Java language, but they are not run within a Java ME virtual machine, and Java-compiled classes and executables will not run natively in Android.
- **Part of the Linux Phone Standards Forum (LiPS) or the Open Mobile Alliance (OMA)** Android runs on an open source Linux kernel, but, while their goals are similar, Android's complete software stack approach goes further than the focus of these standards-defining organizations.
- **Simply an application layer (like UIQ or S60)** While it does include an application layer, "Android" also describes the entire software stack encompassing the underlying operating system, API libraries, and the applications themselves.
- **A mobile phone handset** Android includes a reference design for mobile handset manufacturers, but unlike the iPhone, there is no single "Android Phone." Instead, Android has been designed to support many alternative hardware devices.
- **Google's answer to the iPhone** The iPhone is a fully proprietary hardware and software platform released by a single company (Apple), while Android is an open source software stack produced and supported by the Open Handset Alliance and designed to operate on any handset that meets the requirements. There's been a lot of speculation regarding a Google-branded Android phone, but even should Google produce one, it will be just one company's hardware implementation of the Android platform.

An Open Platform for Mobile Development

Google describes Android as:

> *The first truly open and comprehensive platform for mobile devices, all of the software to run a mobile phone but without the proprietary obstacles that have hindered mobile innovation.*
>
> `http://googleblog.blogspot.com/2007/11/wheres-my-gphone.html`

Android is made up of several necessary and dependent parts including the following:

- A hardware reference design that describes the capabilities required of a mobile device in order to support the software stack
- A Linux operating system kernel that provides the low-level interface with the hardware, memory management, and process control, all optimized for mobile devices
- Open source libraries for application development including SQLite, WebKit, OpenGL, and a media manager
- A run time used to execute and host Android applications, including the Dalvik virtual machine and the core libraries that provide Android specific functionality. The run time is designed to be small and efficient for use on mobile devices.
- An application framework that agnostically exposes system services to the application layer, including the window manager, content providers, location manager, telephony, and peer-to-peer services
- A user interface framework used to host and launch applications
- Preinstalled applications shipped as part of the stack
- A software development kit used to create applications, including the tools, plug-ins, and documentation

> *At this stage, not all of the Android stack has been released as open source, although this is expected to happen by the time phones are released to market. It's also worth noting that the applications you develop for Android do not have to be open source.*

What really makes Android compelling is its open philosophy, which ensures that any deficiencies in user interface or native application design can be fixed by writing an extension or replacement. Android provides you, as a developer, the opportunity to create mobile phone interfaces and applications designed to look, feel, and function exactly as you image them.

Native Android Applications

Android phones will normally come with a suite of preinstalled applications including, but not limited to:

- An e-mail client compatible with Gmail but not limited to it
- An SMS management application
- A full PIM (personal information management) suite including a calendar and contacts list, both tightly integrated with Google's online services

- A fully featured mobile Google Maps application including StreetView, business finder, driving directions, satellite view, and traffic conditions
- A WebKit-based web browser
- An Instant Messaging Client
- A music player and picture viewer
- The Android Marketplace client for downloading thied-party Android applications.
- The Amazon MP3 store client for purchasing DRM free music.

All the native applications are written in Java using the Android SDK and are run on Dalvik.

The data stored and used by the native applications — like contact details — are also available to third-party applications. Similarly, your applications can handle events such as an incoming call or a new SMS message.

The exact makeup of the applications available on new Android phones is likely to vary based on the hardware manufacturer and/or the phone carrier or distributor. This is especially true in the United States, where carriers have significant influence on the software included on shipped devices.

Android SDK Features

The true appeal of Android as a development environment lies in the APIs it provides.

As an application-neutral platform, Android gives you the opportunity to create applications that are as much a part of the phone as anything provided out of the box. The following list highlights some of the most noteworthy Android features:

- No licensing, distribution, or development fees
- Wi-Fi hardware access
- GSM, EDGE, and 3G networks for telephony or data transfer, allowing you to make or receive calls or SMS messages, or to send and retrieve data across mobile networks
- Comprehensive APIs for location-based services such as GPS
- Full multimedia hardware control including playback and recording using the camera and microphone
- APIs for accelerometer and compass hardware
- IPC message passing
- Shared data stores
- An integrated open source WebKit-based browser
- Full support for applications that integrate Map controls as part of their user interface
- Peer-to-peer (P2P) support using Google Talk
- Mobile-optimized hardware-accelerated graphics including a path-based 2D graphics library and support for 3D graphics using OpenGL ES

- Media libraries for playing and recording a variety of audio/video or still image formats
- An application framework that encourages reuse of application components and the replacement of native applications

Access to Hardware including Camera, GPS, and Accelerometer

Android includes API libraries to simplify development involving the device hardware. These ensure that you don't need to create specific implementations of your software for different devices, so you can create Android applications that work as expected on any device that supports the Android software stack.

The Android SDK includes APIs for location-based hardware (such as GPS), camera, network connections, Wi-Fi, Bluetooth, accelerometers, touch screen, and power management. You can explore the possibilities of some of Android's hardware APIs in more detail in Chapter 10.

Native Google Maps, Geocoding, and Location-Based Services

Native map support lets you create a range of map-based applications that leverage the mobility of Android devices. Android lets you create activities that include interactive Google Maps as part of your user interface with full access to maps that you can control programmatically and annotate using Android's rich graphics library.

Android's location-based services manage technologies like GPS and Google's GSM cell-based location technology to determine the device's current position. These services enforce an abstraction from specific location-detecting technology and let you specify minimum requirements (e.g., accuracy or cost) rather than choosing a particular technology. It also means that your location-based applications will work no matter what technology the host handset supports.

To combine maps with locations, Android includes an API for forward and reverse geocoding that lets you find map coordinates for an address, and the address of a map position.

You'll learn the details of using maps, the geocoder, and location-based services in Chapter 7.

Background Services

Android supports applications and services designed to run invisibly in the background.

Modern mobiles are by nature multifunction devices; however, their limited screen size means that generally only one interactive application can be visible at any time. Platforms that don't support background execution limit the viability of applications that don't need your constant attention.

Background services make it possible to create invisible application components that perform automatic processing without direct user action. Background execution allows your applications to become event-driven and to support regular updates, which is perfect for monitoring game scores or market prices, generating location-based alerts, or prioritizing and pre-screening incoming calls and SMS messages.

Learn more about how to get the most out of background services in Chapter 8.

SQLite Database for Data Storage and Retrieval

Rapid and efficient data storage and retrieval are essential for a device whose storage capacity is limited by its compact nature.

Android provides a lightweight relational database for each application using SQLite. Your applications can take advantage of the managed relational database engine to store data securely and efficiently.

By default, each application database is *sandboxed* — its content is available only to the application that created it — but Content Providers supply a mechanism for the managed sharing of these application databases.

Databases, Content Providers, and other data persistence options available in Android are covered in detail in Chapter 6.

Shared Data and Interapplication Communication

Android includes three techniques for transmitting information from your applications for use elsewhere: Notifications, Intents, and Content Providers.

Notifications are the standard ways in which a mobile device traditionally alerts users. Using the API, you can trigger audible alerts, cause vibration, and flash the device's LED, as well as control status bar notification icons as shown in Chapter 8.

Intents provide a mechanism for message passing within and between applications. Using Intents, you can broadcast a desired action (such as dialing the phone or editing a contact) system-wide for other applications to handle. Intents are an important core component of Android and are covered in depth in Chapter 5.

Finally, *Content Providers* are a way to give managed access to your application's private database. The data stores for native applications, such as the Contact Manager, are exposed as Content Providers so you can create your own applications that read or modify these data stores. Chapter 6 covers Content Providers in detail, including the native providers and demonstrating how to create and use providers of your own.

P2P Services with Google Talk

Based on earlier SDK versions, it's expected that in later releases you will once again be able to send structured messages from your application to any other Android mobile using Android's peer-to-peer (P2P) communications service.

The Android P2P service uses a specialized version of XMPP (Extensible Messaging and Presence Protocol). Based on Google's Google Talk instant messaging service, it creates a persistent socket connection between your device and any other online Android handset that ensures communication with low latency and rapid response times.

When made available, you'll be able to use the Google Talk service for conventional instant messaging, or an interface to send data between application instances on separate devices. This is strong sauce for creating interactive applications that involve multiple users, such as real-time multiplayer games or social applications.

The P2P service also offers presence notification, which is used to see if a contact is online. While the P2P service is very attractive in itself, it also plays very well with other Android features. Imagine a background service that transmits locations between friends and a corresponding mapping application that displays these locations or alerts you when friends are nearby.

Owing to security concerns, sending data messages with Google Talk isn't possible in Android 1.0. An instant messaging client is available, and it's expected that XMPP-compatible IM and data messaging will be made available to developers in a future SDK release.

Extensive Media Support and 2D/3D Graphics

Bigger screens and brighter, higher-resolution displays have helped make mobiles multimedia devices. To make the most of the hardware available, Android provides graphics libraries for 2D canvas drawing and 3D graphics with OpenGL.

Android also offers comprehensive libraries for handling still images, video, and audio files including the MPEG4, H.264, MP3, AAC, AMR, JPG, PNG, and GIF formats.

2D and 3D graphics are covered in depth in Chapter 11, while Android media management libraries are covered in Chapter 10.

Optimized Memory and Process Management

Android's process and memory management is a little unusual. Like Java and .NET, Android uses its own run time and virtual machine to manage application memory. Unlike either of these frameworks, the Android run time also manages the process lifetimes. Android ensures application responsiveness by stopping and killing processes as necessary to free resources for higher-priority applications.

In this context, priority is determined depending on the application with which the user is interacting. Ensuring that your applications are prepared for a swift death but are still able to remain responsive and update or restart in the background if necessary, is an important consideration in an environment that does not allow applications to control their own lifetimes.

You will learn more about the Android application life cycle in Chapter 3.

Introducing the Open Handset Alliance

The *Open Handset Alliance (OHA)* is a collection of more than 30 technology companies including hardware manufacturers, mobile carriers, and software developers. Of particular note are the prominent mobile technology companies Motorola, HTC, T-Mobile, and Qualcomm. In their own words, the OHA represents:

> *A commitment to openness, a shared vision for the future, and concrete plans to make the vision a reality. To accelerate innovation in mobile and offer consumers a richer, less expensive, and better mobile experience.*
>
> `http://www.openhandsetalliance.com/oha_faq.html`

The OHA hopes to deliver a better mobile software experience for consumers by providing the platform needed for innovative mobile development at a faster rate and a higher quality without licensing fees for software developers or handset manufacturers.

Ultimately the success of Android as a mobile platform will depend largely on the success of OHA partners in releasing desirable handsets and mobile services that encourage the widespread adoption of Android phones. Developers meanwhile have the opportunity to create innovative new mobile applications for Android to encourage more mobile technology companies to become part of the OHA.

What Does Android Run On?

The first Android mobile handset, the T-Mobile G1, was released in the US in October 2008 and in the UK in November 2008. The Open Handset Alliance has further committed to deploying additional handsets and services that support Android early in 2009.

Rather than a mobile OS created for a single hardware implementation, Android is designed to support a large variety of hardware platforms, from touch-screen phones to devices with no screens at all.

Beyond that, with no licensing fees or proprietary software, the cost to handset manufacturers for providing Android-compatible variations of their handsets is comparatively low. It's hoped that once demand for hardware capable of running popular Android applications reaches a critical mass, more device manufacturers will produce increasingly tailored hardware to meet that demand.

Why Develop for Android?

If you have a background in mobile application development, you don't need me to tell you that:

- A lot of what you can do with Android is already possible.
- But doing it is painful.

Android represents a clean break, a mobile framework based on the reality of modern mobile devices.

With a simple and powerful SDK, no licensing fees, excellent documentation, and a thriving developer community, Android is an excellent opportunity to create software that changes how and why people use their mobile phones.

Android is backed by more than 30 OHA members and is surrounded by significant industry buzz.

In market terms, the growth in portable devices is a worldwide phenomenon, with mobile-phone ownership outstripping computer ownership in many countries. The increasing popularity of *smartphones* — multifunction devices including a phone but featuring cameras, Internet access, media players, Wi-Fi, and GPS services — combined with the increasing availability of mobile broadband and Wi-Fi has created a growth market for advanced mobile applications.

What Will Drive Android Adoption?

Android is targeted primarily at developers, with Google and the OHA betting that the way to deliver better mobile software to consumers is by making it easier for developers to write it themselves.

As a development platform, Android is powerful and intuitive, letting developers who have never programmed for mobile devices create useful applications quickly and easily. It's easy to see how innovative Android applications could create demand for the devices necessary to run them, particularly if developers write applications for Android because they *can't* write them for other platforms.

Open access to the nuts and bolts of the underlying system is what's always driven software development and platform adoption. The Internet's inherent openness and neutrality have seen it become the platform for a multi-billion-dollar industry within 10 years of its inception. Before that, it was open systems like Linux and the powerful APIs provided as part of the Windows operating system that enabled the explosion in personal computers and the movement of computer programming from the arcane to the mainstream.

This openness and power ensure that anyone with the inclination can bring a vision to life at minimal cost. So far, that's not been the case for mobile phones, and that's why there are so few good mobile phone applications and fewer still available for free.

Corporations will also be attracted to Android for the level of control it offers. By using a popular enterprise programming language in Java, no licensing fees, and offering the level of access and control users demand, Android offers an excellent enterprise platform.

What Does It Have That Others Don't?

Many of the features listed previously, such as 3D graphics and native database support, are also available in other mobile SDKs. Here are some of the unique features that set Android apart:

- **Google Map Applications** Google Maps for Mobile has been hugely popular, and Android offers a Google Map as an atomic, reusable control for use in your applications. The MapView widget lets you display, manipulate, and annotate a Google Map within your Activities to build map-based applications using the familiar Google Maps interface.
- **Background Services and Applications** Background services let you create applications that use an event-driven model, working silently while other applications are being used or while your mobile sits ignored until it rings, flashes, or vibrates to get your attention. Maybe it's an application that tracks the stock market, alerting you to significant changes in your portfolio, or a service that changes your ring tone or volume depending on your current location, the time of day, and the identity of the caller.

- **Shared Data and Interprocess Communication** Using Intents and Content Providers, Android lets your applications exchange messages, perform processing, and share data. You can also use these mechanisms to leverage the data and functionality provided by the native Android applications. To mitigate the risks of such an open strategy, each application's process, data storage, and files are private unless explicitly shared with other applications using a full permission-based security mechanism detailed in Chapter 11.
- **All Applications Are Created Equal** Android doesn't differentiate between native applications and those developed by third parties. This gives consumers unprecedented power to change the look and feel of their devices by letting them completely replace every native application with a third-party alternative that has access to the same underlying data and hardware. Every rule needs an exception and this one has two. The "unlock" and "in-call experience" screens can not be replaced in the initial SDK release.
- **P2P Interdevice Application Messaging** Android offers peer-to-peer messaging that supports presence, instant messaging, and interdevice/interapplication communication.

Changing the Mobile Development Landscape

Existing mobile development platforms have created an aura of exclusivity around mobile development. Whether by design or as a side-effect of the cost or complexity involved in developing native applications, most mobile phones will remain nearly identical to what they were when first unwrapped.

In contrast, Android allows, even encourages, radical change. As consumer devices, Android handsets ship with a core set of standard applications that consumers demand on a new phone, but the real power lies in the ability for users to completely change how their device looks, feels, and functions.

Android gives developers a great opportunity. All Android applications are a native part of the phone, not just software that's run in a sandbox on top of it. Rather than writing small-screen versions of software that can be run on low-power devices, you can now write mobile applications that change the way people use their phones.

While Android will still have to compete with existing and future mobile development platforms as an open source developer framework, the strength of use of the development environment is strongly in its favor. Certainly its free and open approach to mobile application development, with total access to the phone's resources, is a giant step in the right direction.

Introducing the Development Framework

With the PR job done, it's time to look at how you can start developing applications for Android. Android applications are written using Java as a programming language but are executed using a custom virtual machine called *Dalvik* rather than a traditional Java VM.

Later in this chapter, you'll be introduced to the framework, starting with a technical explanation of the Android software stack, a look at what's included in the SDK, an introduction to the Android libraries, and a look at the Dalvik virtual machine.

Each Android application runs in a separate process within its own Dalvik instance, relinquishing all responsibility for memory and process management to the Android run time, which stops and kills processes as necessary to manage resources.

Dalvik and the Android run time sit on top of a Linux kernel that handles low-level hardware interaction including drivers and memory management, while a set of APIs provides access to all of the underlying services, features, and hardware.

What Comes in the Box

The Android software development kit (SDK) includes everything you need to start developing, testing, and debugging Android applications. Included in the SDK download are:

- **The Android APIs** The core of the SDK is the Android API libraries that provide developer access to the Android stack. These are the same libraries used at Google to create native Android applications.
- **Development Tools** To turn Android source code into executable Android applications, the SDK includes several development tools that let you compile and debug your applications. You will learn more about the developer tools in Chapter 2.
- **The Android Emulator** The Android Emulator is a fully interactive Android device emulator featuring several alternative skins. Using the emulator, you can see how your applications will look and behave on a real Android device. All Android applications run within the Dalvik VM so that the software emulator is an excellent environment — in fact, as it is hardware-neutral, it provides a better independent test environment than any single hardware implementation.
- **Full Documentation** The SDK includes extensive code-level reference information detailing exactly what's included in each package and class and how to use them. In addition to the code documentation, Android's reference documentation explains how to get started and gives detailed explanations of the fundamentals behind Android development.
- **Sample Code** The Android SDK includes a selection of sample applications that demonstrate some of the possibilities available using Android, as well as simple programs that highlight how to use individual API features.
- **Online Support** Despite its relative youth, Android has generated a vibrant developer community. The Google Groups at `http://code.google.com/android/groups` are active forums of Android developers with regular input from the Android development team at Google.

For those using the popular Eclipse IDE, Android has released a special plug-in that simplifies project creation and tightly integrates Eclipse with the Android Emulator and debugging tools. The features of the ADT plug-in are covered in more detail in Chapter 2.

Understanding the Android Software Stack

The Android software stack is composed of the elements shown in Figure 1-1 and described in further detail below it. Put simply, a Linux kernel and a collection of C/C++ libraries are exposed through an application framework that provides services for, and management of, the run time and applications.

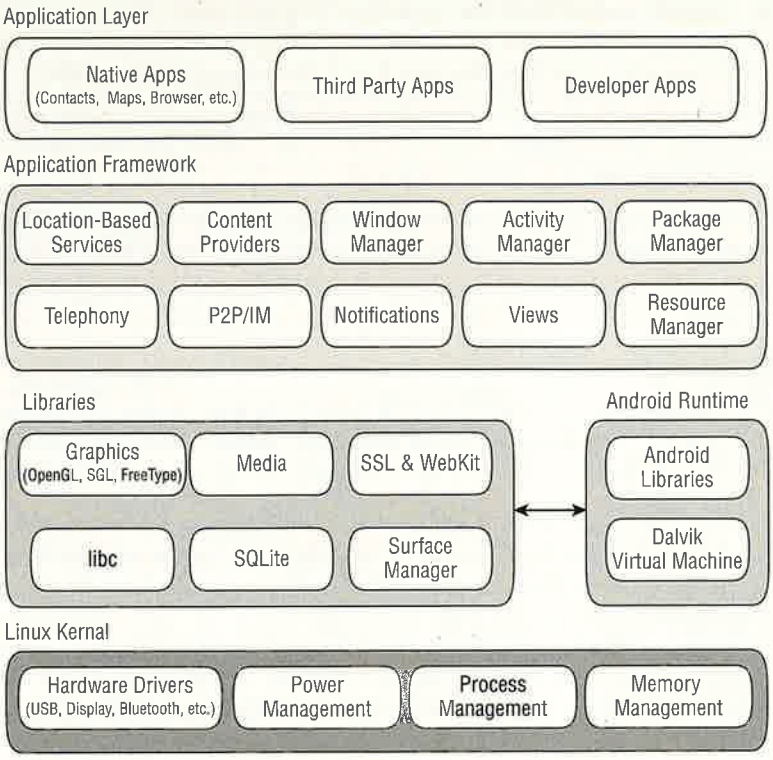


Figure 1-1

- **Linux Kernel** Core services (including hardware drivers, process and memory management, security, network, and power management) are handled by a Linux 2.6 kernel. The kernel also provides an abstraction layer between the hardware and the remainder of the stack.
- **Libraries** Running on top of the kernel, Android includes various C/C++ core libraries such as libc and SSL, as well as:
  - A media library for playback of audio and video media
  - A Surface manager to provide display management
  - Graphics libraries that include SGL and OpenGL for 2D and 3D graphics
  - SQLite for native database support
  - SSL and WebKit for integrated web browser and Internet security
- **Android Run Time** What makes an Android phone an Android phone rather than a mobile Linux implementation is the Android run time. Including the core libraries and the Dalvik virtual machine, the Android run time is the engine that powers your applications and, along with the libraries, forms the basis for the application framework.
  - **Core Libraries** While Android development is done in Java, Dalvik is not a Java VM. The core Android libraries provide most of the functionality available in the core Java libraries as well as the Android-specific libraries.

- **Dalvik Virtual Machine** Dalvik is a register-based virtual machine that's been optimized to ensure that a device can run multiple instances efficiently. It relies on the Linux kernel for threading and low-level memory management.

- **Application Framework** The application framework provides the classes used to create Android applications. It also provides a generic abstraction for hardware access and manages the user interface and application resources.

- **Application Layer** All applications, both native and third party, are built on the application layer using the same API libraries. The application layer runs within the Android run time using the classes and services made available from the application framework.

The Dalvik Virtual Machine

One of the key elements of Android is the Dalvik virtual machine. Rather than use a traditional Java virtual machine (VM) such as Java ME (Java Mobile Edition), Android uses its own custom VM designed to ensure that multiple instances run efficiently on a single device.

The Dalvik VM uses the device's underlying Linux kernel to handle low-level functionality including security, threading, and process and memory management. It's also possible to write C/C++ applications that run directly on the underlying Linux OS. While you *can* do this, in most cases there's no reason you should need to.

> *This book focuses exclusively on writing applications that run within Dalvik. If your inclinations run toward exploring the Linux kernel and C/C++ underbelly of Android, modifying Dalvik, or otherwise tinkering with things under the hood, check out the Android Internals Google Group at* `http://groups.google.com/group/android-internals`

All Android hardware and system service access is managed using Dalvik as a middle tier. By using a VM to host application execution, developers have an abstraction layer that ensures they never have to worry about a particular hardware implementation.

The Dalvik VM executes Dalvik executable files, a format optimized to ensure minimal memory footprint. The `.dex` executables are created by transforming Java language compiled classes using the tools supplied within the SDK. You'll learn more about how to create Dalvik executables in the next chapter.

Android Application Architecture

Android's architecture encourages the concept of component reuse, allowing you to publish and share activities, services, and data with other applications with access managed by the security restrictions you put in place.

The same mechanism that lets you produce a replacement contact manager or phone dialer can let you expose your application components to let other developers create new UI front ends and functionality extensions, or otherwise build on them.

The following application services are the architectural cornerstones of all Android applications, providing the framework you'll be using for your own software:

- **Activity Manager** Controls the life cycle of your activities, including management of the activity stack described in Chapter 3.
- **Views** Are used to construct the user interfaces for your activities as described in Chapter 4.
- **Notification Manager** Provides a consistent and non-intrusive mechanism for signaling your users as described in Chapter 8.
- **Content Providers** Lets your applications share data between applications as described in Chapter 6.
- **Resource Manager** Supports non-code resources like strings and graphics to be externalized as shown in Chapter 3.

Android Libraries

Android offers a number of APIs for developing your applications. The following list of core APIs should provide an insight into what's available; all Android devices will offer support for at least these APIs:

- **android.util** The core utility package contains low-level classes like specialized containers, string formatters, and XML parsing utilities.
- **android.os** The operating system package provides access to basic operating system services like message passing, interprocess communication, clock functions, and debugging.
- **android.graphics** The graphics API supplies the low-level graphics classes that support canvases, colors, and drawing primitives, and lets you draw on canvases.
- **android.text** The text processing tools for displaying and parsing text.
- **android.database** Supplies the low-level classes required for handling cursors when working with databases.
- **android.content** The content API is used to manage data access and publishing by providing services for dealing with resources, content providers, and packages.
- **android.view** Views are the core user interface class. All user interface elements are constructed using a series of Views to provide the user interaction components.
- **android.widget** Built on the View package, the widget classes are the "here's one we created earlier" user-interface elements for you to use in your applications. They include lists, buttons, and layouts.
- **com.google.android.maps** A high-level API that provides access to native map controls that you can use within your application. Includes the MapView control as well as the Overlay and MapController classes used to annotate and control your embedded maps.

- **android.app** A high-level package that provides access to the application model. The application package includes the Activity and Service APIs that form the basis for all your Android applications.
- **android.provider** To ease developer access to certain standard Content Providers (such as the contacts database), the Provider package offers classes to provide access to standard databases included in all Android distributions.
- **android.telephony** The telephony APIs give you the ability to directly interact with the device's phone stack, letting you make, receive, and monitor phone calls, phone status, and SMS messages.
- **android.webkit** The WebKit package features APIs for working with Web-based content, including a WebView control for embedding browsers in your activities and a cookie manager.

In addition to the Android APIs, the Android stack includes a set of C/C++ libraries that are exposed through the application framework. These libraries include:

- **OpenGL** The library used to support 3D graphics based on the Open GL ES 1.0 API
- **FreeType** Support for bitmap and vector font rendering
- **SGL** The core library used to provide a 2D graphics engine
- **libc** The standard C library optimized for Linux-based embedded devices
- **SQLite** The lightweight relation database engine used to store application data
- **SSL** Support for using the Secure Sockets Layer cryptographic protocol for secure Internet communications

Advanced Android Libraries

The core libraries provide all the functionality you need to start creating applications for Android, but it won't be long before you're ready to delve into the advanced APIs that offer the really exciting functionality.

Android hopes to target a wide range of mobile hardware, so be aware that the suitability and implementation of the following APIs will vary depending on the device upon which they are implemented.

- **android.location** The location-based services API gives your applications access to the device's current physical location. Location-based services provide generic access to location information using whatever position-fixing hardware or technology is available on the device.
- **android.media** The media APIs provide support for playback and recording of audio and video media files, including streamed media.
- **android.opengl** Android offers a powerful 3D rendering engine using the OpenGL ES API that you can use to create dynamic 3D user interfaces for your applications.
- **android.hardware** Where available, the hardware API exposes sensor hardware including the camera, accelerometer, and compass sensors as shown in Chapter 10.
- **android.bluetooth, android.net.wifi, and android.telephony** Android also provides low-level access to the hardware platform, including Bluetooth, Wi-Fi, and telephony hardware as shown in Chapter 10.

Summary

This chapter explained that despite significant advances in the hardware features available on modern mobile phones, the software available for them has lagged. A lack of openness, hard-to-use development kits, and hardware-specific APIs have stifled innovation in mobile software.

Android offers an opportunity for developers to create innovative software applications for mobile devices without the restrictions generally associated with the existing proprietary mobile development frameworks.

You were shown the complete Android software stack, which includes not only an application layer and development toolkit but also the Dalvik VM, a custom run time, core libraries, and a Linux kernel; all of which will be available as open source.

The Open Handset Alliance was introduced along with the responsibility that developers — as the primary target audience for Android — have to create applications that will make consumers want Android phones on which to run them.

You also learned:

- How handsets with an expanding range of hardware features have created demand for tools that give developers better access to these features.
- About some of the features available to developers using Android, including peer-to-peer messaging, native map support, hardware access, background services, interprocess and inter-device messaging, shared databases, and 2D and 3D graphics.
- That all Android applications are built equal, allowing users to completely replace one application with another, including the replacement of the core native applications.
- That the Android SDK includes developer tools, APIs, and comprehensive documentation.

The next chapter will help you get started by downloading and installing the Android SDK and setting up an Android development environment in Eclipse.

You'll also learn how to use the Android developer tools plug-in to streamline development, testing, and debugging before creating your first Android application.

After learning about the building blocks of Android applications, you'll be introduced to the different types of applications you can create, and you'll start to understand some of the design considerations that should go into developing applications for mobile devices.

2

Getting Started

All you need to start writing your own Android applications is a copy of the Android SDK and the Java development kit. Unless you're a masochist, you'll probably want a Java IDE — Eclipse is particularly well supported — to make development a little easier.

Versions of the SDK, Java, and Eclipse are available for Windows, Mac OS, and Linux, so you can explore Android from the comfort of whatever OS you favor. The SDK includes an emulator for all three OS environments, and because Android applications are run on a virtual machine, there's no advantage to developing from any particular operating system.

Android code is written using Java syntax, and the core Android libraries include most of the features from the core Java APIs. Before they can be run, though, your projects are first translated into Dalvik byte code. As a result, you get the benefits of using Java, while your applications have the advantage of running on a virtual machine optimized for Android devices.

The SDK download includes all the Android libraries, full documentation, and excellent sample applications. It also includes tools to help you write and debug your applications, like the Android Emulator to run your projects and the Dalvik Debug Monitoring Service (DDMS) to help debug them.

By the end of this chapter, you'll have downloaded the Android SDK, set up your development environment, completed two new applications, and run and debugged them using the emulator and DDMS.

If you've developed for mobile devices before, you already know that their small-form factor, limited power, and restricted memory create some unique design challenges. Even if you're new to the game, it's obvious that some of the things you can take for granted on the desktop or the Web aren't going to work on a mobile.

As well as the hardware limitations, the user environment brings its own challenges. Mobile devices are used on the move and are often a distraction rather than the focus of attention, so your applications need to be fast, responsive, and easy to use.

This chapter examines some of the best practices for writing mobile applications to help overcome the inherent hardware and environmental challenges. Rather than try to tackle the whole topic, we'll focus on using the Android SDK in a way that's consistent with good mobile design principles.

Developing for Android

The Android SDK includes all the tools and APIs you need to write compelling and powerful mobile applications. The biggest challenge with Android, as with any new development toolkit, is learning the features and limitations of its APIs.

If you have experience in Java development, you'll find that the techniques, syntax, and grammar you've been using will translate directly into Android, although some of the specific optimization techniques may seem counterintuitive.

If you don't have experience with Java but have used other object-oriented languages (such as C#), you should find the transition straightforward. The power of Android comes from its APIs, not from Java, so being unfamiliar with all the Java specific classes won't be a big disadvantage.

What You Need to Begin

Because Android applications run within the Dalvik virtual machine, you can write them on any platform that supports the developer tools. This currently includes the following:

- Microsoft Windows (XP or Vista)
- Mac OS X 10.4.8 or later (Intel chips only)
- Linux

To get started, you'll need to download and install the following:

- The Android SDK
- Java Development Kit (JDK) 5 or 6

You can download the latest JDK from Sun at

```
http://java.sun.com/javase/downloads/index.jsp
```

If you already have a JDK installed, make sure that it meets the version requirements listed above, and note that the Java runtime environment (JRE) is not sufficient.

Downloading and Installing the SDK

The Android SDK is completely open. There's no cost to download and use the API, and Google doesn't charge to allow distribution of your finished programs. You can download the latest version of the SDK for your development platform from the Android development home page at

```
http://code.google.com/android/download.html
```

Unless otherwise noted, the version of the Android SDK used for writing this book was version 1.0 r1.

The SDK is presented as a ZIP file containing the API libraries, developer tools, documentation, and several sample applications and API demos that highlight the use of particular API features. Install it by unzipping the SDK into a new folder. (Take note of this location, as you'll need it later.)

The examples and step-by-step instructions provided are targeted at developers using Eclipse with the Android Developer Tool (ADT) plug-in. Neither is required, though — you can use any text editor or Java IDE you're comfortable with and use the developer tools in the SDK to compile, test, and debug the code snippets and sample applications.

If you're planning to use them, the next sections explain how to set up Eclipse and the ADT plug-in as your Android development environment. Later in the chapter, we'll also take a closer look at the developer tools that come with the SDK, so if you'd prefer to develop without using Eclipse or the ADT plug-in, you'll particularly want to check that out.

The examples included in the SDK are well documented and are an excellent source for full, working examples of applications written for Android. Once you've finished setting up your development environment, it's worth going through them.

Developing with Eclipse

Using Eclipse with the ADT plug-in for your Android development offers some significant advantages.

Eclipse is an open source IDE (integrated development environment) particularly popular for Java development. It's available to download for each of the development platforms supported by Android (Windows, Mac OS, and Linux) from the Eclipse foundation homepage:

```
www.eclipse.org/downloads/
```

There are many variations available when selecting your Eclipse download; the following is the recommended configuration for Android:

- Eclipse 3.3, 3.4 (Ganymede)
  - Eclipse JDT plug-in
  - WST

WST and the JDT plug-in are included in most Eclipse IDE packages.

Installing Eclipse consists of uncompressing the download into a new folder. When that's done, run the `Eclipse` executable. When it starts for the first time, create a new workspace for your Android development.

Using the Eclipse Plug-in

The ADT plug-in for Eclipse simplifies your Android development by integrating the developer tools, including the emulator and .class-to-.dex converter, directly into the IDE. While you don't have to use the ADT plug-in, it does make creating, testing, and debugging your applications faster and easier.

The ADT plug-in integrates the following into Eclipse:

- An Android Project Wizard that simplifies creating new projects and includes a basic application template
- Forms-based manifest, layout, and resource editors to help create, edit, and validate your XML resources
- Automated building of Android projects, conversion to Android executables (`.dex`), packaging to package files (.apk), and installation of packages onto Dalvik virtual machines
- The Android Emulator, including control of the emulator's appearance, network connection settings, and the ability to simulate incoming calls and SMS messages
- The Dalvik Debug Monitoring Service (DDMS), which includes port forwarding; stack, heap, and thread viewing; process details; and screen capture facilities
- Access to the device or emulator's filesystem, allowing you to navigate the folder tree and transfer files
- Runtime debugging, so you can set breakpoints and view call stacks
- All Android/Dalvik log and console outputs

Figure 2-1 shows the DDMS perspective within Eclipse with the ADT plug-in installed.

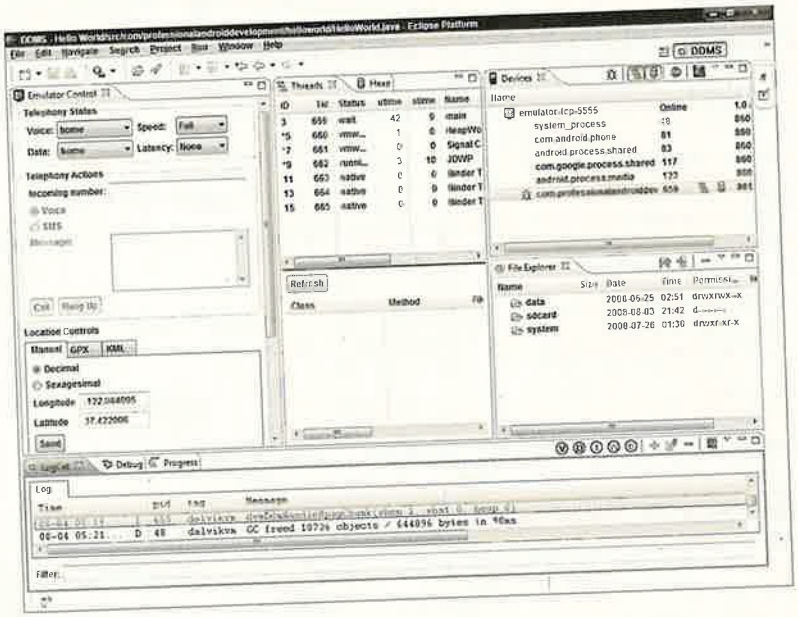


Figure 2-1

Installing the ADT Plug-in

Install the developer tools plug-in by the following steps:

1. Select **Help** ➪ **Software Updates** ➪ **Find and Install** ... from within Eclipse.
2. In the resulting dialog box, choose **Search for new features to install**.
3. Select **New Remote Site**, and enter the following address into the dialog box, as shown in Figure 2-2:

```
https://dl-ssl.google.com/android/eclipse/
```

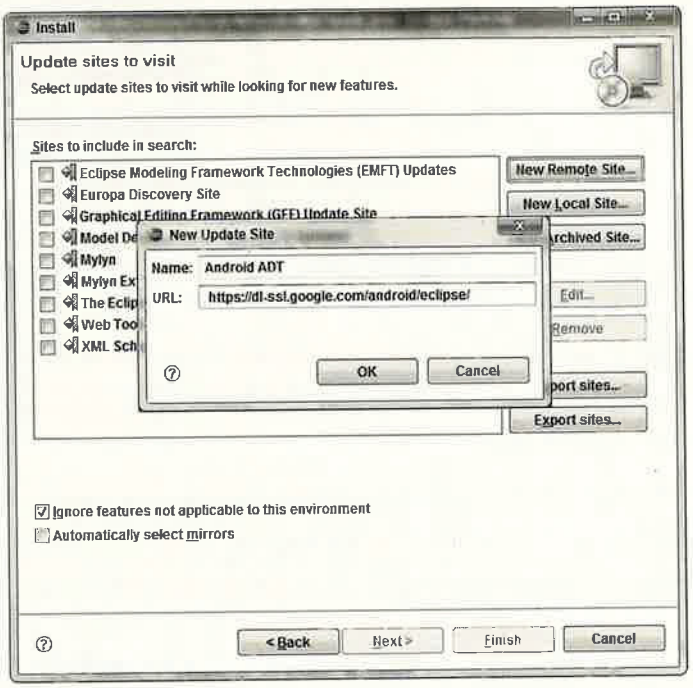


Figure 2-2

4. The new site you entered should now be checked. Click **Finish**.
5. Eclipse will now download the plug-in. When it's finished, select Android **Plugin** ➪ **Developer Tools** from the resulting Search Results dialog box, and click **Next**.
6. Read and then **Accept** the terms of the license agreement, and click **Next** and then **Finish**. As the ADT plug-in is not signed, you'll be prompted before the installation continues.
7. When complete, you'll have to restart Eclipse and update the ADT preferences. Restart and select **Window** ➪ **Preferences** ... (or **Eclipse** ➪ **Preferences** for the Mac OS).
8. Then select **Android** from the left panel.
9. Click **Browse** ..., and navigate to the folder into which you unzipped the Android SDK, as shown in Figure 2-3; then click **Apply** and **OK**.

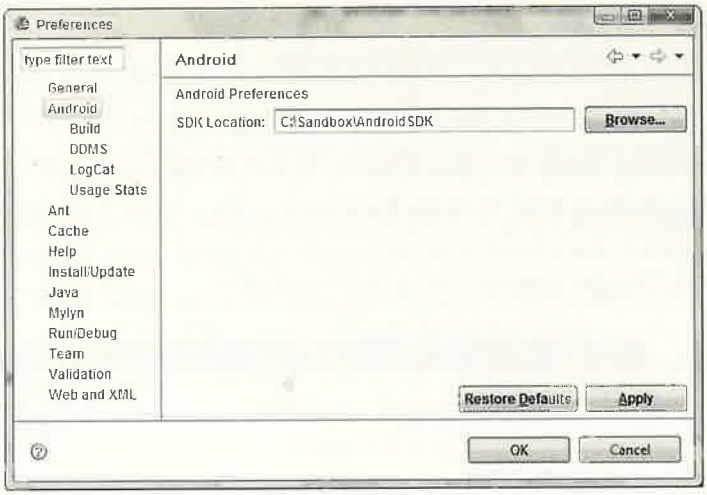


Figure 2-3

If you download a new version of the SDK and place it in a different location, you will need to update this preference to reflect the SDK with which the ADT should be building.

Updating the Plug-in

As the Android SDK matures, there are likely to be frequent updates to the ADT plug-in. In most cases, to update your plug-in, you simply:

1. Navigate to **Help** ➪ **Software Updates** ➪ **Find and Install** ...
2. Select **Search for updates of the currently installed features**, and click **Finish** ...
3. If there are any ADT updates available, they will be presented. Simply select them and choose **Install**.

Sometimes a plug-in upgrade is so significant that the dynamic update mechanism can't be used. In those cases, you may have to remove the previous plug-in completely before installing the newer version as described in the previous section.

Creating Your First Android Activity

You've downloaded the SDK, installed Eclipse, and plugged in the plug-in. You're now ready to start programming for Android. Start by creating a new project and setting up your Eclipse run and debug configurations.

Starting a New Android Project

To create a new Android project using the Android New Project Wizard:

1. Select **File** ➪ **New** ➪ **Project**.
2. Select the **Android Project** application type from the Android folder, and click **Finish**.

3. In the dialog that appears (shown in Figure 2-4), enter the details for your new project. The "Project name" is the name of your project file; the "Package name" specifies its package; the "Activity name" is the name of the class that is your initial Activity; and the "Application name" is the friendly name for your application.

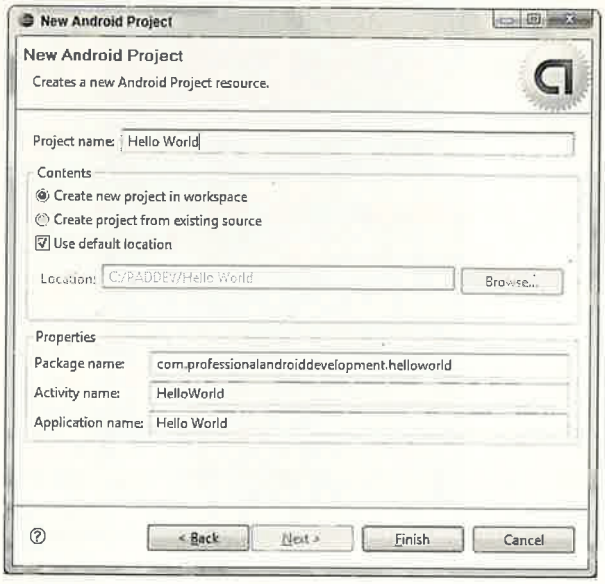


Figure 2-4

4. When you've entered the details, click **Finish**.

The ADT plug-in then creates a new project that includes a new class that extends `Activity`. Rather than being completely empty, the default template implements "Hello World." Before modifying the project, take this opportunity to configure run and debug launch configurations.

Creating a Launch Configuration

Launch configurations let you specify runtime options for running and debugging applications. Using a launch configuration you can specify the following:

- The Project and Activity to launch
- The emulator options to use
- Input/output settings (including console defaults)

You can specify different launch configurations for Run and Debug modes. The following steps show how to create a launch configuration for an Android application:

1. Select **Run ⇨ Open Run Dialog** ... (or **Run ⇨ Open Debug Dialog** ...).
2. Right-click **Android Application** on the project type list, and select **New**.

3. Enter a name for the configuration. You can create multiple configurations for each project, so create a descriptive title that will help you identify this particular setup.
4. Now choose your start-up options. The first (**Android**) tab lets you select the project and Activity that you want to start when you run (or debug) the application. Figure 2-5 shows the settings for the project you created earlier.



Figure 2-5

5. Use the **Target** tab to configure the emulator. There are options to choose the screen size, device skin, and network connection settings. You can also optionally wipe the user data on the emulator and enable or disable the start-up animation. Using the command-line textbox, you can specify additional emulator start-up options if needed.
6. Finally, set any additional properties in the **Common** tab.
7. Click **Apply**, and your launch configuration will be saved.

Running and Debugging Your Android Applications

You've created your first project and created the run and debug configurations for it. Before making any changes, test your installation and configurations by running and debugging the Hello World project.

From the **Run** menu, select **Run** or **Debug** to launch the most recently selected configuration, or select **Open Run Dialog** ... or **Open Debug Dialog** ... to select a configuration to use.

If you're using the ADT plug-in, running or debugging your application:

- Compiles the current project and converts it to an Android executable (`.dex`).
- Packages the executable and external resources into an Android package (.apk).
- Starts the emulator (if it's not already running).
- Installs your application onto the emulator.
- Starts your application.

If you're debugging, the Eclipse debugger will then be attached, allowing you to set breakpoints and debug your code.

If everything is working correctly, you'll see a new Activity running in the emulator, as shown in Figure 2-6.



Figure 2-6

Understanding Hello World

With that confirmed, let's take a step back and have a real look at your first Android application.

`Activity` is the base class for the visual, interactive components of your application; it is roughly equivalent to a Form in traditional desktop development. The following snippet shows the skeleton code for an Activity-based class; note that it extends `Activity`, overriding the `onCreate` method.

```
package com.paad.helloworld;

import android.app.Activity;
import android.os.Bundle;

public class HelloWorld extends Activity {

  /** Called when the activity is first created. */
  @Override
  public void onCreate(Bundle icicle) {
    super.onCreate(icicle);
  }
}
```

What's missing from this template is the layout of the visual interface. In Android, visual components are called *Views*, which are similar to controls in traditional desktop development.

In the Hello World template created by the wizard, the `onCreate` method is overridden to call `setContentView`, which lays out the user interface by inflating a layout resource, as highlighted below:

```
@Override
public void onCreate(Bundle icicle) {
  super.onCreate(icicle);
  setContentView(R.layout.main);
}
```

The resources for an Android project are stored in the `res` folder of your project hierarchy, which includes `drawable`, `layout`, and `values` subfolders. The ADT plug-in interprets these XML resources to provide design time access to them through the `R` variable as described in Chapter 3.

The following code snippet shows the UI layout defined in the `main.xml` file created by the Android project template:

```
<?xml version="1.0" encoding="utf-8"?>
<LinearLayout xmlns:android="http://schemas.android.com/apk/res/android"
  android:orientation="vertical"
  android:layout_width="fill_parent"
  android:layout_height="fill_parent">
  <TextView
    android:layout_width="fill_parent"
    android:layout_height="wrap_content"
    android:text="Hello World, HelloWorld"
  />
</LinearLayout>
```

Defining your UI in XML and inflating it is the preferred way of implementing your user interfaces, as it neatly decouples your application logic from your UI design.

To get access to your UI elements in code, you add identifier attributes to them in the XML definition. You can then use the `findViewById` method to return a reference to each named item. The following XML snippet shows an ID attribute added to the TextView widget in the Hello World template:

```
<TextView
  android:id="@+id/myTextView"
  android:layout_width="fill_parent"
  android:layout_height="wrap_content"
  android:text="Hello World, HelloWorld"
/>
```

And the following snippet shows how to get access to it in code:

```
TextView myTextView = (TextView)findViewById(R.id.myTextView);
```

Alternatively (although it's not considered good practice), if you need to, you can create your layout directly in code as shown below:

```
public void onCreate(Bundle icicle) {
  super.onCreate(icicle);

  LinearLayout.LayoutParams lp;
```

```
        lp = new LinearLayout.LayoutParams(LayoutParams.FILL_PARENT,
                                           LayoutParams.FILL_PARENT);

        LinearLayout.LayoutParams textViewLP;
        textViewLP = new LinearLayout.LayoutParams(LayoutParams.FILL_PARENT,
                                                   LayoutParams.WRAP_CONTENT);

        LinearLayout ll = new LinearLayout(this);
        ll.setOrientation(LinearLayout.VERTICAL);
        TextView myTextView = new TextView(this);
        myTextView.setText("Hello World, HelloWorld");
        ll.addView(myTextView, textViewLP);
        this.addContentView(ll, lp);
    }
```

All the properties available in code can be set with attributes in the XML layout. As well as allowing easier substitution of layout designs and individual UI elements, keeping the visual design decoupled from the application code helps keep the code more concise.

> *The Android web site (`http://code.google.com/android/documentation.html`) includes several excellent step-by-step guides that demonstrate many of the features and good practices you will be using as an Android developer. They're easy to follow and give a good idea of how Android applications fit together.*

Types of Android Applications

Most of the applications you create in Android will fall into one of the following categories:

- **Foreground Activity** An application that's only useful when it's in the foreground and is effectively suspended when it's not visible. Games and map mashups are common examples.
- **Background Service** An application with limited interaction that, apart from when being configured, spends most of its lifetime hidden. Examples of this include call screening applications or SMS auto-responders.
- **Intermittent Activity** Expects some interactivity but does most of its work in the background. Often these applications will be set up and then run silently, notifying users when appropriate. A common example would be a media player.

Complex applications are difficult to pigeonhole into a single category and include elements of all three. When creating your application, you need to consider how it's likely to be used and then design it accordingly. Let's look more closely at some of the design considerations for each application type described above.

Foreground Activities

When creating foreground applications, you need to consider the Activity life cycle (described in Chapter 3) carefully so that the Activity switches seamlessly between the foreground and the background.

Applications have no control over their life cycles, and a backgrounded application, with no Services, is a prime candidate for cleanup by Android's resource management. This means that you need to save the state of the application when the Activity becomes invisible, and present the exact same state when it returns to the foreground.

It's also particularly important for foreground Activities to present a slick and intuitive user experience.

You'll learn more about creating well-behaved and attractive foreground Activities in Chapter 3.

Background Services

These applications run silently in the background with very little user input. They often listen for messages or actions caused by the hardware, system, or other applications, rather than rely on user interaction.

It's possible to create completely invisible services, but in practice, it's better form to provide at least some sort of user control. At a minimum, you should let users confirm that the service is running and let them configure, pause, or terminate it as needed.

Services, the powerhouse of background applications, are covered in depth in Chapter 8.

Intermittent Activities

Often you'll want to create an application that reacts to user input but is still useful when it's not the active foreground Activity. These applications are generally a union of a visible controller Activity with an invisible background Service.

These applications need to be aware of their state when interacting with the user. This might mean updating the Activity UI when it's visible and sending notifications to keep the user updated when it's in the background, as seen in the section on Notifications and Services in Chapter 8.

Developing for Mobile Devices

Android does a lot to simplify mobile-device software development, but it's still important to understand the reasons behind the conventions. There are several factors to account for when writing software for mobile and embedded devices, and when developing for Android, in particular.

> *In this chapter, you'll learn some of the techniques and best practices for writing efficient Android code. In later examples, efficiency is sometimes compromised for clarity and brevity when introducing new Android concepts or functionality. In the best traditions of "Do as I say, not as I do," the examples you'll see are designed to show the simplest (or easiest-to-understand) way of doing something, not necessarily the best way of doing it.*

Hardware-Imposed Design Considerations

Small and portable, mobile devices offer exciting opportunities for software development. Their limited screen size and reduced memory, storage, and processor power are far less exciting, and instead present some unique challenges.

Compared to desktop or notebook computers, mobile devices have relatively:

- ❑ Low processing power
- ❑ Limited RAM
- ❑ Limited permanent storage capacity

- Small screens with low resolution
- Higher costs associated with data transfer
- Slower data transfer rates with higher latency
- Less reliable data connections
- Limited battery life

It's important to keep these restrictions in mind when creating new applications.

Be Efficient

Manufacturers of embedded devices, particularly *mobile* devices, value small size and long battery life over potential improvements in processor speed. For developers, that means losing the head start traditionally afforded thanks to Moore's law. The yearly performance improvements you'll see in desktop and server hardware usually translate into smaller, more power-efficient mobiles without much improvement in processor power.

In practice, this means that you always need to optimize your code so that it runs quickly and responsively, assuming that hardware improvements over the lifetime of your software are unlikely to do you any favors.

Since code efficiency is a big topic in software engineering, I'm not going to try and capture it here. This chapter covers some Android-specific efficiency tips below, but for now, just note that efficiency is particularly important for resource-constrained environments like mobile devices.

Expect Limited Capacity

Advances in flash memory and solid-state disks have led to a dramatic increase in mobile-device storage capacities (although people's MP3 collections tend to expand to fill the available space). In practice, most devices still offer relatively limited storage space for your applications. While the compiled size of your application is a consideration, more important is ensuring that your application is polite in its use of system resources.

You should carefully consider how you store your application data. To make life easier, you can use the Android databases and Content Providers to persist, reuse, and share large quantities of data, as described in Chapter 6. For smaller data storage, such as preferences or state settings, Android provides an optimized framework, as described in Chapter 6.

Of course, these mechanisms won't stop you from writing directly to the filesystem when you want or need to, but in those circumstances, always consider how you're structuring these files, and ensure that yours is an efficient solution.

Part of being polite is cleaning up after yourself. Techniques like caching are useful for limiting repetitive network lookups, but don't leave files on the filesystem or records in a database when they're no longer needed.

Design for Small Screens

The small size and portability of mobiles are a challenge for creating good interfaces, particularly when users are demanding an increasingly striking and information-rich graphical user experience.

Write your applications knowing that users will often only glance at the (small) screen. Make your applications intuitive and easy to use by reducing the number of controls and putting the most important information front and center.

Graphical controls, like the ones you'll create in Chapter 4, are an excellent way to convey dense information in an easy-to-understand way. Rather than a screen full of text with lots of buttons and text entry boxes, use colors, shapes, and graphics to display information.

If you're planning to include touch-screen support (and if you're not, you should be), you'll need to consider how touch input is going to affect your interface design. The time of the stylus has passed; now it's all about finger input, so make sure your Views are big enough to support interaction using a finger on the screen. There's more information on touch-screen interaction in Chapter 11.

Of course, mobile-phone resolutions and screen sizes are increasing, so it's smart to design for small screens, but also make sure your UIs scale.

Expect Low Speeds, High Latency

In Chapter 5, you'll learn how to use Internet resources in your applications. The ability to incorporate some of the wealth of online information in your applications is incredibly powerful.

The mobile Web unfortunately isn't as fast, reliable, or readily available as we'd often like, so when you're developing your Internet-based applications, it's best to assume that the network connection will be slow, intermittent, and expensive. With unlimited 3G data plans and city-wide Wi-Fi, this is changing, but designing for the worst case ensures that you always deliver a high-standard user experience.

This also means making sure that your applications can handle losing (or not finding) a data connection.

The Android Emulator lets you control the speed and latency of your network connection when setting up an Eclipse launch configuration. Figure 2-7 shows the emulator's network connection speed and latency set up to simulate a distinctly suboptimal EDGE connection.

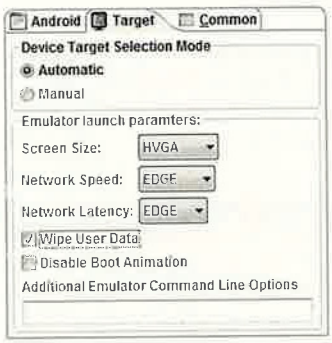


Figure 2-7

Experiment to ensure responsiveness no matter what the speed, latency, and availability of network access. You might find that in some circumstances, it's better to limit the functionality of your application or reduce network lookups to cached bursts, based on the network connection(s) available. Details

on how to detect the kind of network connections available at run time, and their speeds, are included in Chapter 10.

At What Cost?

If you're a mobile owner, you know all too well that some of the more powerful features on your mobile can literally come at a price. Services like SMS, GPS, and data transfer often incur an additional tariff from your service provider.

It's obvious why it's important that any costs associated with functionality in your applications are minimized, and that users are aware when an action they perform might result in them being charged.

It's a good approach to assume that there's a cost associated with any action involving an interaction with the outside world. Minimize interaction costs by the following:

- Transferring as little data as possible
- Caching data and GPS results to eliminate redundant or repetitive lookups
- Stopping all data transfers and GPS updates when your activity is not visible in the foreground if they're only being used to update the UI
- Keeping the refresh/update rates for data transfers (and location lookups) as low as practicable
- Scheduling big updates or transfers at "off peak" times using alarms as shown in Chapter 8

Often the best solution is to use a lower-quality option that comes at a lower cost.

When using the location-based services described in Chapter 7, you can select a location provider based on whether there is an associated cost. Within your location-based applications, consider giving users the choice of lower cost or greater accuracy.

In some circumstances, costs are hard to define, or they're different for different users. Charges for services vary between service providers and user plans. While some people will have free unlimited data transfers, others will have free SMS.

Rather than enforcing a particular technique based on which seems cheaper, consider letting your users choose. For example, when downloading data from the Internet, you could ask users if they want to use any network available or limit their transfers to only when they're connected via Wi-Fi.

Considering the Users' Environment

You can't assume that your users will think of your application as the most important feature of their phones.

Generally, a mobile is first and foremost a phone, secondly an SMS and e-mail communicator, thirdly a camera, and fourthly an MP3 player. The applications you write will most likely be in a fifth category of "useful mobile tools."

That's not a bad thing — it's in good company with others including Google Maps and the web browser. That said, each user's usage model will be different; some people will never use their mobiles

to listen to music, and some phones don't include a camera, but the multitasking principle inherent in a device as ubiquitous as it is indispensable is an important consideration for usability design.

It's also important to consider when and how your users will use your applications. People use their mobiles all the time — on the train, walking down the street, or even while driving their cars. You can't make people use their phones appropriately, but you can make sure that your applications don't distract them any more than necessary.

What does this mean in terms of software design? Make sure that your application:

- **Is well behaved** Start by ensuring that your Activities suspend when they're not in the foreground. Android triggers event handlers when your Activity is suspended or resumed so you can pause UI updates and network lookups when your application isn't visible — there's no point updating your UI if no one can see it. If you need to continue updating or processing in the background, Android provides a Service class designed to run in the background without the UI overheads.
- **Switches seamlessly from the background to the foreground** With the multitasking nature of mobile devices, it's very likely that your applications will regularly switch into and out of the background. When this happens, it's important that they "come to life" quickly and seamlessly. Android's nondeterministic process management means that if your application is in the background, there's every chance it will get killed to free up resources. This should be invisible to the user. You can ensure this by saving the application state and queuing updates so that your users don't notice a difference between restarting and resuming your application. Switching back to it should be seamless with users being shown the exact UI and application state they last saw.
- **Is polite** Your application should never steal focus or interrupt a user's current activity. Use Notifications and Toasts (detailed in Chapter 8) instead to inform or remind users that their attention is requested if your application isn't in the foreground. There are several ways for mobile devices to alert users. For example, when a call is coming in, your phone rings; when you have unread messages, the LED flashes; and when you have new voice mail, a small "mail" icon appears in your status bar. All these techniques and more are available through the notification mechanism.
- **Presents a consistent user interface** Your application is likely to be one of several in use at any time, so it's important that the UI you present is easy to use. Don't force users to interpret and relearn your application every time they load it. Using it should be simple, easy, and obvious — particularly given the limited screen space and distracting user environment.
- **Is responsive** Responsiveness is one of the most important design considerations on a mobile device. You've no doubt experienced the frustration of a "frozen" piece of software; the multifunction nature of a mobile makes it even more annoying. With possible delays due to slow and unreliable data connections, it's important that your application use worker threads and background services to keep your activities responsive and, more importantly, stop them from preventing other applications from responding in a timely manner.

Developing for Android

Nothing covered so far is specific to Android; the design considerations above are just as important when developing applications for any mobile. In addition to these general guidelines, Android has some particular considerations.

To start with, it's worth taking a few minutes to read Google's Android design philosophy at `http://code.google.com/android/toolbox/philosophy.html`.

The Android design philosophy demands that applications be:

- Fast
- Responsive
- Secure
- Seamless

Being Fast and Efficient

In a resource-constrained environment, being fast means being efficient. A lot of what you already know about writing efficient code will be just as effective in Android, but the limitations of embedded systems and the use of the Dalvik VM mean you can't take things for granted.

The smart bet for advice is to go to the source. The Android team has published some specific guidance on writing efficient code for Android, so rather than rehash their advice, I suggest you visit `http://code.google.com/android/toolbox/performance.html` and take note of their suggestions.

> *You may find that some of these performance suggestions contradict established design practices — for example, avoiding the use of internal setters and getters or preferring virtual over interface. When writing software for resource-constrained systems like embedded devices, there's often a compromise between conventional design principles and the demand for greater efficiency.*

One of the keys to writing efficient Android code is to not carry over assumptions from desktop and server environments to embedded devices.

At a time when 2 to 4 GB of memory is standard for most desktop and server rigs, even advanced smartphones are lucky to feature 32 MB of RAM. With memory such a scarce commodity, you need to take special care to use it efficiently. This means thinking about how you use the stack and heap, limiting object creation, and being aware of how variable scope affects memory use.

Being Responsive

Android takes responsiveness very seriously.

Android enforces responsiveness with the Activity Manager and Window Manager. If either service detects an unresponsive application, it will display the unambiguous Application unresponsive (AUR) message, as shown in Figure 2-8.

This alert is modal, steals focus, and won't go away until you hit a button or your application starts responding — it's pretty much the last thing you ever want to confront a user with.

Android monitors two conditions to determine responsiveness:

- An application must respond to any user action, such as a key press or screen touch, within 5 seconds.
- A Broadcast Receiver must return from its `onReceive` handler within 10 seconds.



Figure 2-8

The most likely culprits for causing unresponsiveness are network lookups, complex processing (such as calculating game moves), and file I/O. There are a number of ways to ensure that these actions don't exceed the responsiveness conditions, in particular, using services and worker threads, as shown in Chapter 8.

The AUR dialog is a last resort of usability; the generous 5-second limit is a worst-case scenario, not a benchmark to aim for. Users will notice a regular pause of anything more than half a second between key press and action. Happily, a side effect of the efficient code you're already writing will be faster, more responsive applications.

Developing Secure Applications

Android applications have direct hardware access, can be distributed independently, and are built on an open source platform featuring open communication, so it's not particularly surprising that security is a big concern.

For the most part, users will take responsibility for what applications they install and what permissions they grant them. The Android security model restricts access to certain services and functionality by forcing applications to request permission before using them. During installation, users then decide if the application should be granted the permissions requested. You can learn more about Android's security model in Chapter 11 and at `http://code.google.com/android/devel/security.html`.

This doesn't get you off the hook. You not only need to make sure your application is secure for its own sake, but you also need to ensure that it can't be hijacked to compromise the device. You can use several techniques to help maintain device security, and they'll be covered in more detail as you learn the technologies involved. In particular, you should:

- Consider requiring permissions for any services you create or broadcasts you transmit.
- Take special care when accepting input to your application from external sources such as the Internet, SMS messages, or instant messaging (IM). You can find out more about using IM and SMS for application messaging in Chapter 9.
- Be cautious when your application may expose access to lower-level hardware.

For reasons of clarity and simplicity, many of the examples in this book take a fairly relaxed approach to security. When creating your own applications, particularly ones you plan to distribute, this is an area that should not be overlooked. You can find out more about Android security in Chapter 11.

Ensuring a Seamless User Experience

The idea of a seamless user experience is an important, if somewhat nebulous, concept. What do we mean by *seamless*? The goal is a consistent user experience where applications start, stop, and transition instantly and without noticeable delays or jarring transitions.

The speed and responsiveness of a mobile device shouldn't degrade the longer it's on. Android's process management helps by acting as a silent assassin, killing background applications to free resources as required. Knowing this, your applications should always present a consistent interface, regardless of whether they're being restarted or resumed.

With an Android device typically running several third-party applications written by different developers, it's particularly important that these applications interact seamlessly.

Use a consistent and intuitive approach to usability. You can still create applications that are revolutionary and unfamiliar, but even they should integrate cleanly with the wider Android environment.

Persist data between sessions, and suspend tasks that use processor cycles, network bandwidth, or battery life when the application isn't visible. If your application has processes that need to continue running while your activity is out of sight, use a Service, but hide these implementation decisions from your users.

When your application is brought back to the front, or restarted, it should seamlessly return to its last visible state. As far as your users are concerned, each application should be sitting silently ready to be used but just out of sight.

You should also follow the best-practice guidelines for using Notifications and use generic UI elements and themes to maintain consistency between applications.

There are many other techniques you can use to ensure a seamless user experience, and you'll be introduced to some of them as you discover more of the possibilities available in Android in the coming chapters.

To-Do List Example

In this example, you'll be creating a new Android application from scratch. This simple example creates a new to-do list application using native Android View controls. It's designed to illustrate the basic steps involved in starting a new project.

> *Don't worry if you don't understand everything that happens in this example. Some of the features used to create this application, including ArrayAdapters, ListViews, and KeyListeners, won't be introduced properly until later chapters, where they're explained in detail. You'll also return to this example later to add new functionality as you learn more about Android.*

1. Start by creating a new Android project. Within Eclipse, select **File ⇨ New ⇨ Project** ..., then choose **Android** (as shown in Figure 2-9) before clicking **Next**.

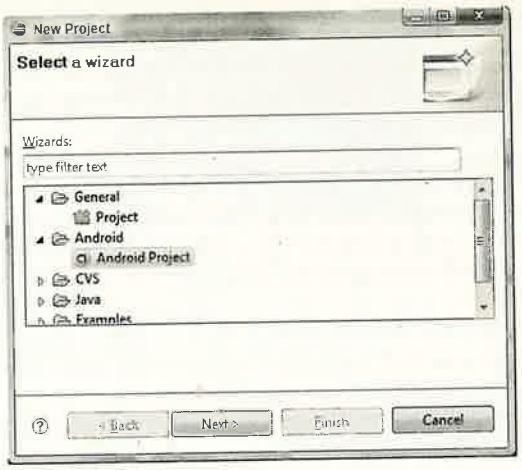


Figure 2-9

2. In the dialog box that appears (shown in Figure 2-10), enter the details for your new project. The "Application name" is the friendly name of your application, and the "Activity name" is the name of your Activity subclass. With the details entered, click **Finish** to create your new project.

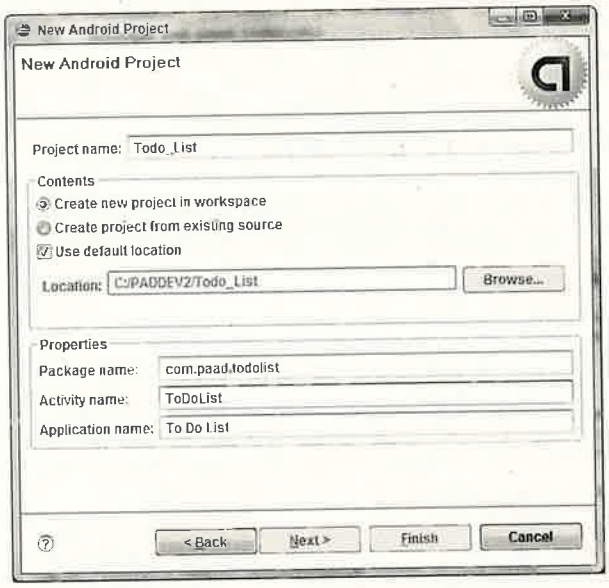


Figure 2-10

3. Take this opportunity to set up debug and run configurations by selecting **Run ⇨ Open Debug Dialog** ... and then **Run ⇨ Open Run Dialog** ..., creating a new configuration for each, specifying the `Todo_List` project. You can leave the launch actions as **Launch Default** Activity or explicitly set them to launch the new `ToDoList` Activity, as shown in Figure 2-11.

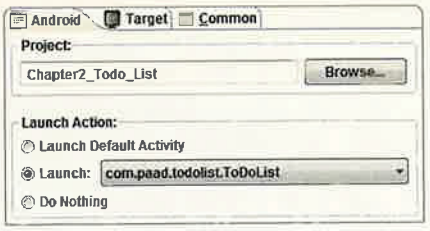


Figure 2-11

4. Now decide what you want to show the users and what actions they'll need to perform. Design a user interface that will make this as intuitive as possible.

In this example, we want to present users with a list of to-do items and a text entry box to add new ones. There's both a list and a text entry control (View) available from the Android libraries. You'll learn more about the Views available in Android and how to create new ones in Chapter 4.

The preferred method for laying out your UI is using a layout resource file. Open the main.xml layout file in the res/layout project folder, as shown in Figure 2-12.

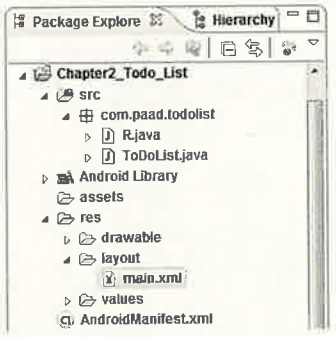


Figure 2-12

5. Modify the main layout to include a `ListView` and an `EditText` within a `LinearLayout`. It's important to give both the `EditText` and `ListView` controls IDs so you can get references to them in code.

```
<?xml version="1.0" encoding="utf-8"?>
<LinearLayout xmlns:android="http://schemas.android.com/apk/res/android"
  android:orientation="vertical"
  android:layout_width="fill_parent"
  android:layout_height="fill_parent">
  <EditText
    android:id="@+id/myEditText"
    android:layout_width="fill_parent"
    android:layout_height="wrap_content"
    android:text="New To Do Item"
  />
  <ListView
```

```
    android:id="@+id/myListView"
    android:layout_width="fill_parent"
    android:layout_height="wrap_content"
  />
</LinearLayout>
```

6. With your user interface defined, open the `ToDoList.java` Activity class from your project's source folder. In this example, you'll make all your changes by overriding the `onCreate` method. Start by inflating your UI using `setContentView` and then get references to the `ListView` and `EditText` using `findViewById`.

```
public void onCreate(Bundle icicle) {
  // Inflate your view
  setContentView(R.layout.main);

  // Get references to UI widgets
  ListView myListView = (ListView)findViewById(R.id.myListView);
  final EditText myEditText = (EditText)findViewById(R.id.myEditText);
}
```

7. Still within `onCreate`, define an `ArrayList` of Strings to store each to-do list item. You can bind a `ListView` to an `ArrayList` using an `ArrayAdapter`, so create a new `ArrayAdapter` instance to bind the to-do item array to the `ListView`. We'll return to `ArrayAdapters` in Chapter 5.

```
public void onCreate(Bundle icicle) {
  setContentView(R.layout.main);

  ListView myListView = (ListView)findViewById(R.id.myListView);
  final EditText myEditText = (EditText)findViewById(R.id.myEditText);

  // Create the array list of to do items
  final ArrayList<String> todoItems = new ArrayList<String>();
  // Create the array adapter to bind the array to the listview
  final ArrayAdapter<String> aa;
  aa = new ArrayAdapter<String>(this,
                                android.R.layout.simple_list_item_1,
                                todoItems);
  // Bind the array adapter to the listview.
  myListView.setAdapter(aa);
}
```

8. The final step to make this to-do list functional is to let users add new to-do items. Add an `onKeyListener` to the `EditText` that listens for a "D-pad center button" click before adding the contents of the `EditText` to the to-do list array and notifying the `ArrayAdapter` of the change. Then clear the `EditText` to prepare for another item.

```
public void onCreate(Bundle icicle) {
  setContentView(R.layout.main);

  ListView myListView = (ListView)findViewById(R.id.myListView);
  final EditText myEditText = (EditText)findViewById(R.id.myEditText);

  final ArrayList<String> todoItems = new ArrayList<String>();
  final ArrayAdapter<String> aa;
  aa = new ArrayAdapter<String>(this,
```

```
                                        android.R.layout.simple_list_item_1,
                                        todoItems);
    myListView.setAdapter(aa);

    myEditText.setOnKeyListener(new OnKeyListener() {
      public boolean onKey(View v, int keyCode, KeyEvent event) {
        if (event.getAction() == KeyEvent.ACTION_DOWN)
          if (keyCode == KeyEvent.KEYCODE_DPAD_CENTER)
          {
            todoItems.add(0, myEditText.getText().toString());
            aa.notifyDataSetChanged();
            myEditText.setText("");
            return true;
          }
        return false;
      }
    });
  }
```

9. Run or debug the application, and you'll see a text entry box above a list, as shown in Figure 2-13.

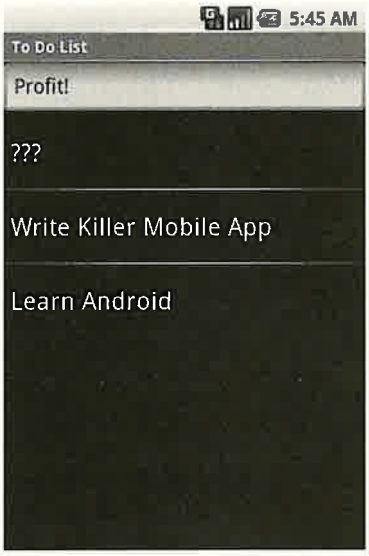


Figure 2-13

10. You've now finished your first "real" Android application. Try adding breakpoints to the code to test the debugger and experiment with the DDMS perspective.

As it stands, this to-do list application isn't spectacularly useful. It doesn't save to-do list items between sessions, you can't edit or remove an item from the list, and typical task list items like due dates and task priority aren't recorded or displayed. On balance, it fails most of the criteria laid out so far for a good mobile application design.

You'll rectify some of these deficiencies when you return to this example in later chapters.

Android Development Tools

The Android SDK includes several tools and utilities to help you create, test, and debug your projects. A detailed examination of each developer tool is outside the scope of this book, but it's worth briefly reviewing what's available. For more detail than is included here, check out the Android documentation at:

```
http://code.google.com/android/intro/tools.html
```

As mentioned earlier, the ADT plug-in conveniently incorporates most of these tools into the Eclipse IDE, where you can access them from the DDMS perspective, including:

- **The Android Emulator** An implementation of the Android virtual machine designed to run on your development computer. You can use the emulator to test and debug your android applications.
- **Dalvik Debug Monitoring Service (DDMS)** Use the DDMS perspective to monitor and control the Dalvik virtual machines on which you're debugging your applications.
- **Android Asset Packaging Tool (AAPT)** Constructs the distributable Android package files (.apk).
- **Android Debug Bridge (ADB)** The *ADB* is a client-server application that provides a link to a running emulator. It lets you copy files, install compiled application packages (.apk), and run shell commands.

The following additional tools are also available:

- **SQLite3** A database tool that you can use to access the SQLite database files created and used by Android
- **Traceview** Graphical analysis tool for viewing the trace logs from your Android application
- **MkSDCard** Creates an SDCard disk image that can be used by the emulator to simulate an external storage card.
- **dx** Converts Java `.class` bytecode into Android `.dex` bytecode.
- **activityCreator** Script that builds Ant build files that you can then use to compile your Android applications without the ADT plug-in

Let's take a look at some of the more important tools in more detail.

The Android Emulator

The emulator is the perfect tool for testing and debugging your applications, particularly if you don't have a real device (or don't want to risk it) for experimentation.

The emulator is an implementation of the Dalvik virtual machine, making it as valid a platform for running Android applications as any Android phone. Because it's decoupled from any particular hardware, it's an excellent baseline to use for testing your applications.

A number of alternative user interfaces are available to represent different hardware configurations, each with different screen sizes, resolutions, orientations, and hardware features to simulate a variety of mobile device types.

Full network connectivity is provided along with the ability to tweak the Internet connection speed and latency while debugging your applications. You can also simulate placing and receiving voice calls and SMS messages.

The ADT plug-in integrates the emulator into Eclipse so that it's launched automatically when you run or debug your projects. If you aren't using the plug-in or want to use the emulator outside of Eclipse, you can telnet into the emulator and control it from its console. For more details on controlling the emulator, check the documentation at `http://code.google.com/android/reference/emulator.html`.

> *At this stage, the emulator doesn't implement all the mobile hardware features supported by Android, including the camera, vibration, LEDs, actual phone calls, the accelerometer, USB connections, Bluetooth, audio capture, battery charge level, and SD card insertion/ejection.*

Dalvik Debug Monitor Service (DDMS)

The emulator lets you see how your application will look, behave, and interact, but to really see what's happening under the surface, you need the DDMS. The Dalvik Debug Monitoring Service is a powerful debugging tool that lets you interrogate active processes, view the stack and heap, watch and pause active threads, and explore the filesystem of any active emulator.

The DDMS perspective in Eclipse also provides simplified access to screen captures of the emulator and the logs generated by LogCat.

If you're using the ADT plug-in, the DDMS is fully integrated into Eclipse and is available from the DDMS perspective. If you aren't using the plug-in or Eclipse, you can run DDMS from the command line, and it will automatically connect to any emulator that's running.

The Android Debug Bridge (ADB)

The *Android debug bridge (ADB)* is a client-service application that lets you connect with an Android Emulator or device. It's made up of three components: a daemon running on the emulator, a service that runs on your development hardware, and client applications (like the DDMS) that communicate with the daemon through the service.

As a communications conduit between your development hardware and the Android device/emulator, the ADB lets you install applications, push and pull files, and run shell commands on the target device. Using the device shell, you can change logging settings, and query or modify SQLite databases available on the device.

The ADT tool automates and simplifies a lot of the usual interaction with the ADB, including application installation and update, log files, and file transfer (through the DDMS perspective).

To learn more about what you can do with the ADB, check out the documentation at `http://code.google.com/android/reference/adb.html`.

Summary

This chapter showed you how to download and install the Android SDK; create a development environment using Eclipse on Windows, Mac OS, or Linux platforms; and how to create run and debug configurations for your projects. You learned how to install and use the ADT plug-in to simplify creating new projects and streamline your development cycle.

You were introduced to some of the design considerations for developing mobile applications, particularly the importance of optimizing for speed and efficiency when increasing battery life and shrinking sizes are higher priorities than increasing processor power.

As with any mobile development, there are considerations when designing for small screens and mobile data connections that can be slow, costly, and unreliable.

After creating an Android to-do list application, you were introduced to the Android Emulator and the developer tools you'll use to test and debug your applications.

Specifically in this chapter, you:

- Downloaded and installed the Android SDK.
- Set up a development environment in Eclipse and downloaded and installed the ADT plug-in.
- Created your first application and learned how it works.
- Set up run and debug launch configurations for your projects.
- Learned about the different types of Android applications.
- Were introduced to some mobile-device design considerations and learned some specific Android design practices.
- Created a to-do list application.
- Were introduced to the Android Emulator and the developer tools.

The next chapter focuses on Activities and application design. You'll see how to define application settings using the Android manifest and how to externalize your UI layouts and application resources. You'll also find out more about the Android application life cycle and Android application states.

3

Creating Applications and Activities

Before you start writing Android applications, it's important to understand how they're constructed and have an understanding of the Android application life cycle. In this chapter, you'll be introduced to the loosely coupled components that make up Android applications (and how they're bound together using the Android manifest). Next you'll see how and why you should use external resources, before getting an introduction to the Activity component.

In recent years, there's been a move toward development frameworks featuring managed code, such as the Java virtual machine and the .NET Common Language Runtime.

In Chapter 1, you learned that Android uses this model, with each application running in a separate process with its own instance of the Dalvik virtual machine. In this chapter, you'll learn more about the application life cycle and how it's managed by the Android run time. This will lead to an introduction of the process states that represent the application priority, which, in turn, determines the likelihood of an application's being terminated when more resources are required.

Mobile devices now come in many shapes and sizes and are used internationally. In this chapter, you'll learn how to give your applications the flexibility to run seamlessly on different hardware, in different countries, and using multiple languages by externalizing resources.

Next you'll examine the Activity component. Arguably the most important of the Android building blocks, the `Activity` class forms the basis for all your user interface screens. You'll learn how to create new Activities and gain an understanding of their life cycles and how they affect the application lifetime.

Finally, you'll be introduced to some of the Activity subclasses that wrap up resource management for some common user interface components such as maps and lists.

What Makes an Android Application?

Android applications consist of loosely coupled components, bound using a project manifest that describes each component and how they interact.

There are six components that provide the building blocks for your applications:

- **Activities** Your application's presentation layer. Every screen in your application will be an extension of the `Activity` class. Activities use Views to form graphical user interfaces that display information and respond to user actions. In terms of desktop development, an Activity is equivalent to a Form. You'll learn more about Activities later in this chapter.
- **Services** The invisible workers of your application. Service components run invisibly, updating your data sources and visible Activities and triggering Notifications. They're used to perform regular processing that needs to continue even when your application's Activities aren't active or visible. You'll learn how to create Services in Chapter 8.
- **Content Providers** A shareable data store. Content Providers are used to manage and share application databases. Content Providers are the preferred way of sharing data across application boundaries. This means that you can configure your own Content Providers to permit access from other applications and use Content Providers exposed by others to access their stored data. Android devices include several native Content Providers that expose useful databases like contact information. You'll learn how to create and use Content Providers in Chapter 6.
- **Intents** A simple message-passing framework. Using Intents, you can broadcast messages system-wide or to a target Activity or Service, stating your intention to have an action performed. The system will then determine the target(s) that will perform any actions as appropriate.
- **Broadcast Receivers** Intent broadcast consumers. By creating and registering a Broadcast Receiver, your application can listen for broadcast Intents that match specific filter criteria. Broadcast Receivers will automatically start your application to respond to an incoming Intent, making them ideal for event-driven applications.
- **Notifications** A user notification framework. Notifications let you signal users without stealing focus or interrupting their current Activities. They're the preferred technique for getting a user's attention from within a Service or Broadcast Receiver. For example, when a device receives a text message or an incoming call, it alerts you by flashing lights, making sounds, displaying icons, or showing dialog messages. You can trigger these same events from your own applications using Notifications, as shown in Chapter 8.

By decoupling the dependencies between application components, you can share and interchange individual pieces, such as Content Providers or Services, with other applications — both your own and those of third parties.

Introducing the Application Manifest

Each Android project includes a manifest file, `AndroidManifest.xml`, stored in the root of the project hierarchy. The manifest lets you define the structure and metadata of your application and its components.

It includes nodes for each of the components (Activities, Services, Content Providers, and Broadcast Receivers) that make up your application and, using Intent Filters and Permissions, determines how they interact with each other and other applications.

It also offers attributes to specify application metadata (like its icon or theme), and additional top-level nodes can be used for security settings and unit tests as described below.

The manifest is made up of a root `manifest` tag with a `package` attribute set to the project's package. It usually includes an `xmlns:android` attribute that supplies several system attributes used within the file. A typical manifest node is shown in the XML snippet below:

```
<manifest xmlns:android=http://schemas.android.com/apk/res/android
          package="com.my_domain.my_app">
          [ ... manifest nodes ... ]
</manifest>
```

The `manifest` tag includes nodes that define the application components, security settings, and test classes that make up your application. The following list gives a summary of the available `manifest` node tags, and an XML snippet demonstrating how each one is used:

- `application` A manifest can contain only one application node. It uses attributes to specify the metadata for your application (including its title, icon, and theme). It also acts as a container that includes the Activity, Service, Content Provider, and Broadcast Receiver tags used to specify the application components.

```
<application android:icon="@drawable/icon"
             android:theme="@style/my_theme">
             [ ... application nodes ... ]
</application>
```

  - `activity` An `activity` tag is required for every Activity displayed by your application, using the `android:name` attribute to specify the class name. This must include the main launch Activity and any other screen or dialogs that can be displayed. Trying to start an Activity that's not defined in the manifest will throw a runtime exception. Each Activity node supports `intent-filter` child tags that specify which Intents launch the Activity.

```
<activity android:name=".MyActivity" android:label="@string/app_name">
  <intent-filter>
    <action android:name="android.intent.action.MAIN" />
    <category android:name="android.intent.category.LAUNCHER" />
  </intent-filter>
</activity>
```

  - `service` As with the `activity` tag, create a new `service` tag for each Service class used in your application. (Services are covered in detail in Chapter 8.) Service tags also support `intent-filter` child tags to allow late runtime binding.

```
<service android:enabled="true" android:name=".MyService"></service>
```

- `provider` Provider tags are used for each of your application's Content Providers. Content Providers are used to manage database access and sharing within and between applications and are examined in Chapter 6.

```
<provider android:permission="com.paad.MY_PERMISSION"
          android:name=".MyContentProvider"
          android:enabled="true"
          android:authorities="com.paad.myapp.MyContentProvider">
</provider>
```

- `receiver` By adding a `receiver` tag, you can register a Broadcast Receiver without having to launch your application first. As you'll see in Chapter 5, Broadcast Receivers are like global event listeners that, once registered, will execute whenever a matching Intent is broadcast by an application. By registering a Broadcast Receiver in the manifest, you can make this process entirely autonomous. If a matching Intent is broadcast, your application will be started automatically and the registered Broadcast Receiver will be run.

```
<receiver android:enabled="true"
          android:label="My Broadcast Receiver"
          android:name=".MyBroadcastReceiver">
</receiver>
```

- `uses-permission` As part of the security model, `uses-permission` tags declare the permissions you've determined that your application needs for it to operate properly. The permissions you include will be presented to the user, to grant or deny, during installation. Permissions are required for many of the native Android services, particularly those with a cost or security implication (such as dialing, receiving SMS, or using the location-based services). As shown in the item below, third-party applications, including your own, can also specify permissions before providing access to shared application components.

```
<uses-permission android:name="android.permission.ACCESS_LOCATION">
</uses-permission>
```

- `permission` Before you can restrict access to an application component, you need to define a permission in the manifest. Use the `permission` tag to create these permission definitions. Application components can then require them by adding the `android:permission` attribute. Other applications will then need to include a `uses-permission` tag in their manifests (and have it granted) before they can use these protected components.

  Within the `permission` tag, you can specify the level of access the permission will permit (`normal`, `dangerous`, `signature`, `signatureOrSystem`), a label, and an external resource containing the description that explain the risks of granting this permission.

```
<permission android:name="com.paad.DETONATE_DEVICE"
            android:protectionLevel="dangerous"
            android:label="Self Destruct"
            android:description="@string/detonate_description">
</permission>
```

- `instrumentation` Instrumentation classes provide a framework for running tests on your Activities and Services at run time. They provide hooks to monitor your application and its

interaction with the system resources. Create a new node for each of the test classes you've created for your application.

```
<instrumentation android:label="My Test"
                 android:name=".MyTestClass"
                 android:targetPackage="com.paad.aPackage">
</instrumentation>
```

A more detailed description of the manifest and each of these nodes can be found at

```
http://code.google.com/android/devel/bblocks-manifest.html
```

The ADT New Project Wizard automatically creates a new manifest file when it creates a new project.

You'll return to the manifest as each of the application components is introduced.

Using the Manifest Editor

The ADT plug-in includes a visual Manifest Editor to manage your manifest, rather than your having to manipulate the underlying XML directly.

To use the Manifest Editor in Eclipse, right-click the `AndroidManifest.xml` file in your project folder, and select **Open With ⇨ Android Manifest Editor**. This presents the Android Manifest Overview screen, as shown in Figure 3-1. This gives you a high-level view of your application structure and provides shortcut links to the Application, Permissions, Instrumentation, and raw XML screens.

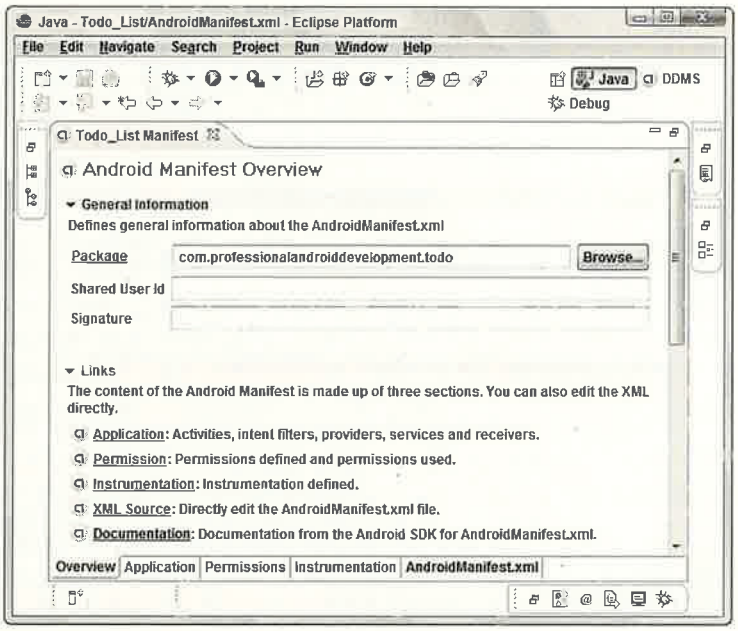


Figure 3-1

Each of the next three tabs contains a visual interface for managing the application, security, and instrumentation (testing) settings, while the last tag (using the manifest's filename) gives access to the raw XML.

Of particular interest is the Application tab, shown in Figure 3-2. Use it to manage the application node and the application component hierarchy, where you specify the application components.

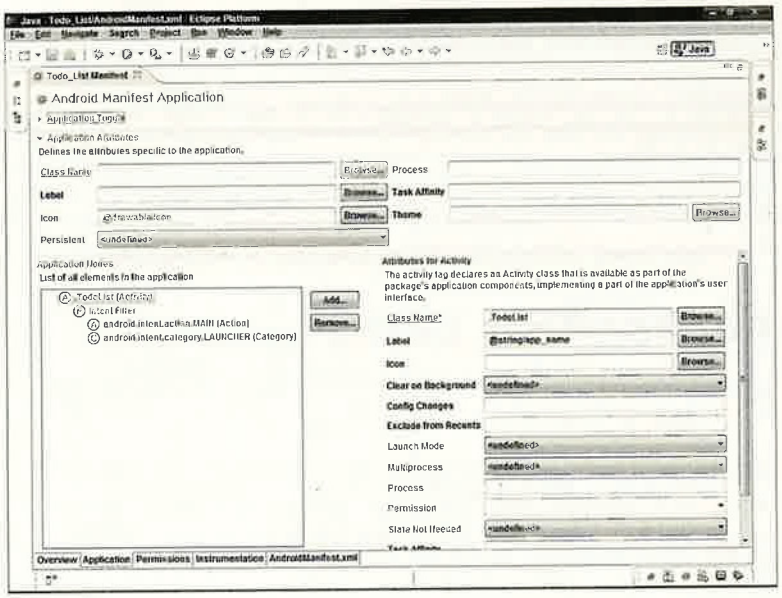


Figure 3-2

You can specify an application's attributes — including its Icon, Label, and Theme — in the Application Attributes panel. The Application Nodes tree beneath it lets you manage the application components, including their attributes and any associated Intent Filter subnodes.

The Android Application Life Cycle

Unlike most traditional environments, Android applications have no control over their own life cycles. Instead, application components must listen for changes in the application state and react accordingly, taking particular care to be prepared for untimely termination.

As mentioned before, by default, each Android application is run in its own process that's running a separate instance of Dalvik. Memory and process management of each application is handled exclusively by the run time.

> *While uncommon, it's possible to force application components within the same application to run in different processes or to have multiple applications share the same process using the* `android:process` *attribute on the affected component nodes within the manifest.*

Android aggressively manages its resources, doing whatever it takes to ensure that the device remains responsive. This means that processes (and their hosted applications) will be killed, without warning if

necessary, to free resources for higher-priority applications — generally those that are interacting directly with the user at the time. The prioritization process is discussed in the next section.

Understanding Application Priority and Process States

The order in which processes are killed to reclaim resources is determined by the priority of the hosted applications. An application's priority is equal to its highest-priority component.

Where two applications have the same priority, the process that has been at a lower priority longest will be killed first. Process priority is also affected by interprocess dependencies; if an application has a dependency on a Service or Content Provider supplied by a second application, the secondary application will have at least as high a priority as the application it supports.

> *All Android applications will remain running and in memory until the system needs its resources for other applications.*

Figure 3-3 shows the priority tree used to determine the order of application termination.

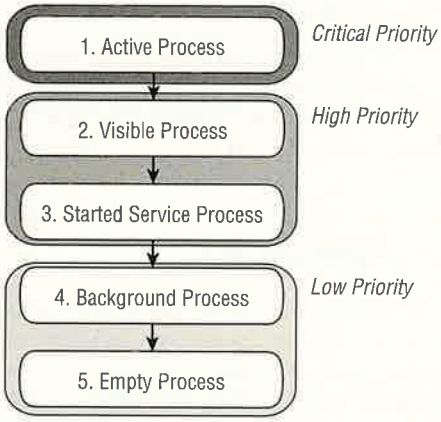


Figure 3-3

It's important to structure your application correctly to ensure that its priority is appropriate for the work it's doing. If you don't, your application could be killed while it's in the middle of something important.

The following list details each of the application states shown in Figure 3-3, explaining how the state is determined by the application components comprising it:

- **Active Processes** Active (foreground) processes are those hosting applications with components currently interacting with the user. These are the processes Android is trying to keep responsive by reclaiming resources. There are generally very few of these processes, and they will be killed only as a last resort.

Active processes include:

- Activities in an "active" state; that is, they are in the foreground and responding to user events. You will explore Activity states in greater detail later in this chapter.
- Activities, Services, or Broadcast Receivers that are currently executing an `onReceive` event handler.
- Services that are executing an `onStart`, `onCreate`, or `onDestroy` event handler.

- **Visible Processes** Visible, but inactive processes are those hosting "visible" Activities. As the name suggests, visible Activities are visible, but they aren't in the foreground or responding to user events. This happens when an Activity is only partially obscured (by a non-full-screen or transparent Activity). There are generally very few visible processes, and they'll only be killed in extreme circumstances to allow active processes to continue.
- **Started Service Processes** Processes hosting Services that have been started. Services support ongoing processing that should continue without a visible interface. Because Services don't interact directly with the user, they receive a slightly lower priority than visible Activities. They are still considered to be foreground processes and won't be killed unless resources are needed for active or visible processes. You'll learn more about Services in Chapter 8.
- **Background Processes** Processes hosting Activities that aren't visible and that don't have any Services that have been started are considered background processes. There will generally be a large number of background processes that Android will kill using a last-seen-first-killed pattern to obtain resources for foreground processes.
- **Empty Processes** To improve overall system performance, Android often retains applications in memory after they have reached the end of their lifetimes. Android maintains this cache to improve the start-up time of applications when they're re-launched. These processes are routinely killed as required.

Externalizing Resources

No matter what your development environment, it's always good practice to keep non-code resources like images and string constants external to your code. Android supports the externalization of resources ranging from simple values such as strings and colors to more complex resources like images (drawables), animations, and themes. Perhaps the most powerful resources available for externalization are layouts.

By externalizing resources, they become easier to maintain, update, and manage. This also lets you easily define alternative resource values to support different hardware and internationalization.

You'll see later in this section how Android dynamically selects resources from resource trees that let you define alternative values based on a device's hardware configuration, language, and location. This lets you create different resource values for specific languages, countries, screens, and keyboards. When an application starts, Android will automatically select the correct resource values without your having to write a line of code.

Among other things, this lets you change the layout based on the screen size and orientation and customize text prompts based on language and country.

Creating Resources

Application resources are stored under the `res/` folder of your project hierarchy. In this folder, each of the available resource types can have a subfolder containing its resources.

If you start a project using the ADT wizard, it will create a `res` folder that contains subfolders for the `values`, `drawable`, and `layout` resources that contain the default layout, application icon, and string resource definitions, respectively, as shown in Figure 3-4.



Figure 3-4

There are seven primary resource types that have different folders: simple values, drawables, layouts, animations, XML, styles, and raw resources. When your application is built, these resources will be compiled as efficiently as possible and included in your application package.

This process also creates an `R` class file that contains references to each of the resources you include in your project. This lets you reference the resources in your code, with the advantage of design time syntax checking.

The following sections describe the specific resource types available within these categories and how to create them for your applications.

In all cases, the resource filenames should contain only lowercase letters, numbers, and the period (.) and underscore (_) symbols.

Creating Simple Values

Supported simple values include strings, colors, dimensions, and string or integer arrays. All simple values are stored within XML files in the `res/values` folder.

Within each XML file, you indicate the type of value being stored using tags as shown in the sample XML file below:

```
<?xml version="1.0" encoding="utf-8"?>
<resources>
  <string name="app_name">To Do List</string>
  <color name="app_background">#FF0000FF</color>
  <dimen name="default_border">5px</dimen>
  <array name="string_array">
```

```
    <item>Item 1</item>
    <item>Item 2</item>
    <item>Item 3</item>
  </array>
  <array name="integer_array">
    <item>3</item>
    <item>2</item>
    <item>1</item>
  </array>
</resources>
```

This example includes all of the simple value types. By convention, resources are separated into separate files for each type; for example, `res/values/strings.xml` would contain only string resources.

The following sections detail the options for defining simple resources.

Strings

Externalizing your strings helps maintain consistency within your application and makes it much easier to create localized versions.

String resources are specified using the `string` tag as shown in the following XML snippet:

```
<string name="stop_message">Stop.</string>
```

Android supports simple text styling, so you can use the HTML tags `<b>`, `<i>`, and `<u>` to apply bold, italics, or underlining to parts of your text strings as shown in the example below:

```
<string name="stop_message"><b>Stop.</b></string>
```

You can use resource strings as input parameters for the `String.format` method. However, `String.format` does not support the text styling described above. To apply styling to a format string, you have to escape the HTML tags when creating your resource, as shown below:

```
<string name="stop_message">&lt;b&gt;Stop&lt;/b&gt;. %1$s</string>
```

Within your code, use the `Html.fromHtml` method to convert this back into a styled character sequence:

```
String rString = getString(R.string.stop_message);
String fString = String.format(rString, "Collaborate and listen.");
CharSequence styledString = Html.fromHtml(fString);
```

Colors

Use the `color` tag to define a new color resource. Specify the color value using a # symbol followed by the (optional) alpha channel, then the red, green, and blue values using one or two hexadecimal numbers with any of the following notations:

- #RGB
- #RRGGBB

- #ARGB
- #ARRGGBB

The following example shows how to specify a fully opaque blue and a partially transparent green:

```
<color name="opaque_blue">#00F</color>
<color name="transparent_green">#7700FF00</color>
```

Dimensions

Dimensions are most commonly referenced within style and layout resources. They're useful for creating layout constants such as borders and font heights.

To specify a dimension resource, use the `dimen` tag, specifying the dimension value, followed by an identifier describing the scale of your dimension:

- `px` Screen pixels
- `in` Physical inches
- `pt` Physical points
- `mm` Physical millimeters
- `dp` Density-independent pixels relative to a 160-dpi screen
- `sp` Scale-independent pixels

These alternatives let you define a dimension not only in absolute terms, but also using relative scales that account for different screen resolutions and densities to simplify scaling on different hardware.

The following XML snippet shows how to specify dimension values for a large font size and a standard border:

```
<dimen name="standard_border">5px</dimen>
<dimen name="large_font_size">16sp</dimen>
```

Styles and Themes

Style resources let your applications maintain a consistent look and feel by specifying the attribute values used by Views. The most common use of themes and styles is to store the colors and fonts for an application.

You can easily change the appearance of your application by simply specifying a different style as the theme in your project manifest.

To create a style, use a `style` tag that includes a `name` attribute, and contains one or more `item` tags. Each `item` tag should include a `name` attribute used to specify the attribute (such as font size or color) being defined. The tag itself should then contain the value, as shown in the skeleton code below:

```
<?xml version="1.0" encoding="utf-8"?>
<resources>
```

```
    <style name="StyleName">
        <item name="attributeName">value</item>
    </style>
</resources>
```

Styles support inheritance using the `parent` attribute on the `style` tag, making it easy to create simple variations.

The following example shows two styles that can also be used as a theme; a base style that sets several text properties and a second style that modifies the first to specify a smaller font.

```
<?xml version="1.0" encoding="utf-8"?>
<resources>
  <style name="BaseText">
    <item name="android:textSize">14sp</item>
    <item name="android:textColor">#111</item>
  </style>
  <style name="SmallText" parent="BaseText">
    <item name="android:textSize">8sp</item>
  </style>
</resources>
```

Drawables

Drawable resources include bitmaps and NinePatch (stretchable PNG) images. They are stored as individual files in the `res/drawable` folder.

The resource identifier for a bitmap resource is the lowercase filename without an extension.

The preferred format for a bitmap resource is PNG, although JPG and GIF files are also supported.

NinePatch (or stretchable) images are PNG files that mark the parts of an image that can be stretched. NinePatch images must be properly defined PNG files that end in `.9.png`. The resource identifier for NinePatches is the filename without the trailing `.9.png`.

A *NinePatch* is a variation of a PNG image that uses a 1-pixel border to define the area of the image that can be stretched if the image is enlarged. To create a NinePatch, draw single-pixel black lines that represent stretchable areas along the left and top borders of your image. The unmarked sections won't be resized, and the relative size of each of the marked sections will remain the same as the image size changes.

NinePatches are a powerful technique for creating images for the backgrounds of Views or Activities that may have a variable size; for example, Android uses NinePatches for creating button backgrounds.

Layouts

Layout resources let you decouple your presentation layer by designing user-interface layouts in XML rather than constructing them in code.

The most common use of a layout is to define the user interface for an Activity. Once defined in XML, the layout is "inflated" within an Activity using `setContentView`, usually within the `onCreate` method.

You can also reference layouts from within other layout resources, such as layouts for each row in a List View. More detailed information on using and creating layouts in Activities can be found in Chapter 4.

Using layouts to create your screens is best-practice UI design in Android. The decoupling of the layout from the code lets you create optimized layouts for different hardware configurations, such as varying screen sizes, orientation, or the presence of keyboards and touch screens.

Each layout definition is stored in a separate file, each containing a single layout, in the `res/layout` folder. The filename then becomes the resource identifier.

A thorough explanation of layout containers and View elements is included in the next chapter, but as an example, the following code snippet shows the layout created by the New Project Wizard. It uses a LinearLayout as a layout container for a TextView that displays the "Hello World" greeting.

```
<?xml version="1.0" encoding="utf-8"?>
<LinearLayout xmlns:android="http://schemas.android.com/apk/res/android"
  android:orientation="vertical"
  android:layout_width="fill_parent"
  android:layout_height="fill_parent">
  <TextView
    android:layout_width="fill_parent"
    android:layout_height="wrap_content"
    android:text="Hello World!"
  />
</LinearLayout>
```

Animations

Android supports two types of animation. *Tweened animations* can be used to rotate, move, stretch, and fade a View; or you can create *frame-by-frame animations* to display a sequence of drawable images. A comprehensive overview of creating, using, and applying animations can be found in Chapter 11.

Defining animations as external resources allows you to reuse the same sequence in multiple places and provides you with the opportunity to present an alternative animation based on device hardware or orientation.

Tweened Animations

Each tweened animation is stored in a separate XML file in the project's `res/anim` folder. As with layouts and drawable resources, the animation's filename is used as its resource identifier.

An animation can be defined for changes in `alpha` (fading), `scale` (scaling), `translate` (moving), or `rotate` (rotating).

Each of these animation types supports attributes to define what the sequence will do:

Alpha	fromAlpha and toAlpha	Float from 0 to 1
Scale	fromXScale/toXScale	Float from 0 to 1
	fromYScale/toYScale	Float from 0 to 1
	pivotX/pivotY	String of the percentage of graphic width/height from 0% to 100%

Translate	fromX/toX	Float from 0 to 1
	fromY/toY	Float from 0 to 1
Rotate	fromDegrees/toDegrees	Float from 0 to 360
	pivotX/pivotY	String of the percentage of graphic width/height from 0% to 100%

You can create a combination of animations using the `set` tag. An **animation** set contains one or more animation transformations and supports various additional tags and **attributes** to customize when and how each animation within the set is run.

The following list shows some of the set tags available:

- **duration** Duration of the animation in milliseconds.
- **startOffset** Millisecond delay before starting this animation.
- **fillBefore** True to apply the animation transformation before it begins.
- **fillAfter** True to apply the animation transformation after it begins.
- **interpolator** Interpolator to set how the speed of this **effect varies** over time. Chapter 11 explores the interpolators available. To specify an **interpolator, reference** the system animation resources at `android:anim/interpolatorName`.

If you do not use the `startOffset` *tag, all the animation effects within a set will execute simultaneously.*

The following example shows an animation set that spins the target 360 degrees while it shrinks and fades out:

```
<?xml version="1.0" encoding="utf-8"?>
<set xmlns:android="http://schemas.android.com/apk/res/android"
     android:interpolator="@android:anim/accelerate_interpolator">
  <rotate
    android:fromDegrees="0"
    android:toDegrees="360"
    android:pivotX="50%"
    android:pivotY="50%"
    android:startOffset="500"
    android:duration="1000" />
  <scale
    android:fromXScale="1.0"
    android:toXScale="0.0"
    android:fromYScale="1.0"
    android:toYScale="0.0"
    android:pivotX="50%"
    android:pivotY="50%"
    android:startOffset="500"
    android:duration="500" />
  <alpha
    android:fromAlpha="1.0"
    android:toAlpha="0.0"
```

```
        android:startOffset="500"
        android:duration="500" />
    </set>
```

Frame-by-Frame Animations

Frame-by-frame animations let you create a sequence of drawables, each of which will be displayed for a specified duration, on the background of a View.

Because frame-by-frame animations represent animated drawables, they are stored in the `res/drawble` folder, rather than with the tweened animations, and use their filenames (without the xml extension) as their resource IDs.

The following XML snippet shows a simple animation that cycles through a series of bitmap resources, displaying each one for half a second. In order to use this snippet, you will need to create new image resources rocket1 through rocket3.

```
<animation-list
  xmlns:android="http://schemas.android.com/apk/res/android"
  android:oneshot="false">
  <item android:drawable="@drawable/rocket1" android:duration="500" />
  <item android:drawable="@drawable/rocket2" android:duration="500" />
  <item android:drawable="@drawable/rocket3" android:duration="500" />
</animation-list>
```

Using Resources

As well as the resources you create, Android supplies several system resources that you can use in your applications. The resources can be used directly from your application code and can also be referenced from within other resources (e.g., a dimension resource might be referenced in a layout definition).

Later in this chapter, you'll learn how to define alternative resource values for different languages, locations, and hardware. It's important to note that when using resources you cannot choose a particular specialized version. Android will automatically select the most appropriate value for a given resource identifier based on the current hardware and device settings.

Using Resources in Code

You access resources in code using the static `R` class. `R` is a generated class based on your external resources and created by compiling your project. The `R` class contains static subclasses for each of the resource types for which you've defined at least one resource. For example, the default new project includes the `R.string` and `R.drawable` subclasses.

> *If you are using the ADT plug-in in Eclipse, the* `R` *class will be created automatically when you make any change to an external resource file or folder. If you are not using the plug-in, use the AAPT tool to compile your project and generate the* `R` *class.* `R` *is a compiler-generated class, so don't make any manual modifications to it as they will be lost when the file is regenerated.*

Each of the subclasses within `R` exposes its associated resources as variables, with the variable names matching the resource identifiers — for example, `R.string.app_name` or `R.drawable.icon`.

THE UNIVERSITY OF MICHIGAN LIBRARIES

The value of these variables is a reference to the corresponding resource's location in the resource table, *not* an instance of the resource itself.

Where a constructor or method, such as `setContentView`, accepts a resource identifier, you can pass in the resource variable, as shown in the code snippet below:

```
// Inflate a layout resource.
setContentView(R.layout.main);
// Display a transient dialog box that displays the
// error message string resource.
Toast.makeText(this, R.string.app_error, Toast.LENGTH_LONG).show();
```

When you need an instance of the resource itself, you'll need to use helper methods to extract them from the resource table, represented by an instance of the `Resources` class.

Because these methods perform lookups on the application's resource table, these helper methods can't be static. Use the `getResources` method on your application context as shown in the snippet below to access your application's Resource instance:

```
Resources myResources = getResources();
```

The `Resources` class includes getters for each of the available resource types and generally works by passing in the resource ID you'd like an instance of. The following code snippet shows an example of using the helper methods to return a selection of resource values:

```
Resources myResources = getResources();
CharSequence styledText = myResources.getText(R.string.stop_message);
Drawable icon = myResources.getDrawable(R.drawable.app_icon);

int opaqueBlue = myResources.getColor(R.color.opaque_blue);

float borderWidth = myResources.getDimension(R.dimen.standard_border);

Animation tranOut;
tranOut = AnimationUtils.loadAnimation(this, R.anim.spin_shrink_fade);

String[] stringArray;
stringArray = myResources.getStringArray(R.array.string_array);

int[] intArray = myResources.getIntArray(R.array.integer_array);
```

Frame-by-frame animated resources are inflated into `AnimationResources`. You can return the value using `getDrawable` and casting the return value as shown below:

```
AnimationDrawable rocket;
rocket = (AnimationDrawable)myResources.getDrawable(R.drawable.frame_by_frame);
```

At the time of going to print, there is a bug in the `AnimationDrawable` class. Currently, AnimationDrawable resources are not properly loaded until some time after an Activity's `onCreate` method has completed. Current work-arounds use timers to force a delay before loading a frame-by-frame resource.

Referencing Resources in Resources

You can also reference resources to use as attribute values in other XML resources.

This is particularly useful for layouts and styles, letting you create specialized variations on themes and localized strings and graphics. It's also a useful way to support different images and spacing for a layout to ensure that it's optimized for different screen sizes and resolutions.

To reference one resource from another, use `@` notation, as shown in the following snippet:

```
attribute="@[packagename:]resourcetype/resourceidentifier"
```

Android will assume you're using a resource from the same package, so you only need to fully qualify the package name if you're using a resource from a different package.

The following snippet creates a layout that uses color, dimension, and string resources:

```
<?xml version="1.0" encoding="utf-8"?>
<LinearLayout
  xmlns:android="http://schemas.android.com/apk/res/android"
  android:orientation="vertical"
  android:layout_width="fill_parent"
  android:layout_height="fill_parent"
  android:padding="@dimen/standard_border">
  <EditText
    android:id="@+id/myEditText"
    android:layout_width="fill_parent"
    android:layout_height="wrap_content"
    android:text="@string/stop_message"
    android:textColor="@color/opaque_blue"
  />
</LinearLayout>
```

Using System Resources

The native Android applications externalize many of their resources, providing you with various strings, images, animations, styles, and layouts to use in your applications.

Accessing the system resources in code is similar to using your own resources. The difference is that you use the native android resource classes available from `android.R`, rather than the application-specific `R` class. The following code snippet uses the `getString` method available in the application context to retrieve an error message available from the system resources:

```
CharSequence httpError = getString(android.R.string.httpErrorBadUrl);
```

To access system resources in XML, specify *Android* as the package name, as shown in this XML snippet:

```
<EditText
  android:id="@+id/myEditText"
  android:layout_width="fill_parent"
  android:layout_height="wrap_content"
  android:text="@android:string/httpErrorBadUrl"
  android:textColor="@android:color/darker_gray"
/>
```

Referring to Styles in the Current Theme

Themes are an excellent way to ensure consistency for your application's UI. Rather than fully define each style, Android provides a shortcut to let you use styles from the currently applied theme.

To do this, you use `?android:` rather than `@` as a prefix to the resource you want to use. The following example shows a snippet of the above code but uses the current theme's text color rather than an external resource:

```
<EditText
  android:id="@+id/myEditText"
  android:layout_width="fill_parent"
  android:layout_height="wrap_content"
  android:text="@string/stop_message"
  android:textColor="?android:textColor"
/>
```

This technique lets you create styles that will change if the current theme changes, without having to modify each individual style resource.

To-Do List Resources Example

In this example, you'll create new external resources in preparation for adding functionality to the To-Do List example you started in Chapter 2. The string and image resources you create here will be used in Chapter 4 when you implement a menu system for the To-Do List application.

The following steps will show you how to create text and icon resources to use for the add and remove menu items, and how to create a theme to apply to the application:

1. Create two new PNG images to represent adding, and removing, a to-do list item. Each image should have dimensions of approximately 16 × 16 pixels, like those illustrated in Figure 3-5.

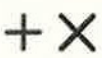


Figure 3-5

2. Copy the images into your project's `res/drawable` folder, and refresh your project. Your project hierarchy should appear as shown in Figure 3-6.



Figure 3-6

3. Open the strings.xml resource from the `res/values` folder, and add values for the "add_new," "remove," and "cancel" menu items. (You can remove the default "hello" string value while you're there.)

```
<?xml version="1.0" encoding="utf-8"?>
<resources>
  <string name="app_name">To Do List</string>
  <string name="add_new">Add New Item</string>
  <string name="remove">Remove Item</string>
  <string name="cancel">Cancel</string>
</resources>
```

4. Create a new theme for the application by creating a new styles.xml resource in the `res/values` folder. Base your theme on the standard Android theme, but set values for a default text size.

```
<?xml version="1.0" encoding="utf-8"?>
<resources>
  <style name="ToDoTheme" parent="@android:style/Theme.Black">
    <item name="android:textSize">12sp</item>
  </style>
</resources>
```

5. Apply the theme to your project in the manifest:

```
<activity android:name=".ToDoList"
          android:label="@string/app_name"
          android:theme="@style/ToDoTheme">
```

Creating Resources for Different Languages and Hardware

One of the most powerful reasons to externalize your resources is Android's dynamic resource selection mechanism.

Using the structure described below, you can create different resource values for specific languages, locations, and hardware configurations that Android will choose between dynamically at run time.

This lets you create language-, location-, and hardware-specific user interfaces without having to change your code.

Specifying alternative resource values is done using a parallel directory structure within the `res/` folder, using hyphen (-) separated text qualifiers to specify the conditions you're supporting.

The example hierarchy below shows a folder structure that features default string values, along with a French language alternative with an additional Canadian location variation:

```
Project/
  res/
    values/
      strings.xml
    values-fr/
      strings.xml
    values-fr-rCA/
      strings.xml
```


THE UNIVERSITY OF MICHIGAN LIBRARIES

The following list gives the available qualifiers you can use to customize your resource files:

- **Language** Using the lowercase two-letter ISO 639-1 language code (e.g., `en`)
- **Region** A lowercase "r" followed by the uppercase two-letter ISO 3166-1-alpha-2 language code (e.g., `rUS`, `rGB`)
- **Screen Orientation** One of `port` (portrait), `land` (landscape), or `square` (square)
- **Screen Pixel Density** Pixel density in dots per inch (dpi) (e.g., `92dpi`, `108dpi`)
- **Touchscreen Type** One of `notouch`, `stylus`, or `finger`
- **Keyboard Availability** Either of `keysexposed` or `keyshidden`
- **Keyboard Input Type** One of `nokeys`, `qwerty`, or `12key`
- **UI Navigation Type** One of `notouch`, `dpad`, `trackball`, or `wheel`
- **Screen Resolution** Screen resolution in pixels with the largest dimension first (e.g., `320x240`)

You can specify multiple qualifiers for any resource type, separating each qualifier with a hyphen. Any combination is supported; however, they must be used in the order given in the list above, and no more than one value can be used per qualifier.

The following example shows valid and invalid directory names for alternative drawable resources.

- **Valid:**

```
drawable-en-rUS
drawable-en-keyshidden
drawable-land-notouch-nokeys-320x240
```

- **Invalid:**

```
drawable-rUS-en (out of order)
drawable-rUS-rUK (multiple values for a single qualifier)
```

When Android retrieves a resource at run time, it will find the best match from the available alternatives. Starting with a list of all the folders in which the required value exists, it then selects the one with the greatest number of matching qualifiers. If two folders are an equal match, the tiebreaker will be based on the order of the matched qualifiers in the above list.

Runtime Configuration Changes

Android supports runtime changes to the language, location, and hardware by terminating and restarting each application and reloading the resource values.

This default behavior isn't always convenient or desirable, particularly as some configuration changes (like screen orientation and keyboard visibility) can occur as easily as a user rotating the device or sliding out the keyboard. You can customize your application's response to these changes by detecting and reacting to them yourself.

To have an Activity listen for runtime configuration changes, add an `android:configChanges` attribute to its manifest node, specifying the configuration changes you want to handle.

The following list describes the configuration changes you can specify:

- `orientation` The screen has been rotated between portrait and landscape.
- `keyboardHidden` The keyboard has been exposed or hidden.
- `fontScale` The user has changed the preferred font size.
- `locale` The user has chosen a different language setting.
- `keyboard` The type of keyboard has changed; for example, the phone may have a 12 keypad that flips out to reveal a full keyboard.
- `touchscreen` or `navigation` The type of keyboard or navigation method has changed. Neither of these events should normally happen.

You can select multiple events to handle by separating the values with a pipe (|).

The following XML snippet shows an activity node declaring that it will handle changes in screen orientation and keyboard visibility:

```
<activity android:name=".TodoList"
          android:label="@string/app_name"
          android:theme="@style/TodoTheme"
          android:configChanges="orientation|keyboard"/>
```

Adding this attribute suppresses the restart for the specified configuration changes, instead, triggering the `onConfigurationChanged` method in the Activity. Override this method to handle the configuration changes using the passed-in `Configuration` object to determine the new configuration values, as shown in the following skeleton code. Be sure to call back to the super class and reload any resource values that the Activity uses in case they've changed.

```
@Override
public void onConfigurationChanged(Configuration _newConfig) {
  super.onConfigurationChanged(_newConfig);

  [ ... Update any UI based on resource values ... ]

  if (_newConfig.orientation == Configuration.ORIENTATION_LANDSCAPE) {
    [ ... React to different orientation ... ]
  }

  if (_newConfig.keyboardHidden == Configuration.KEYBOARDHIDDEN_NO) {
    [ ... React to changed keyboard visibility ... ]
  }
}
```

When `onConfigurationChanged` is called, the Activity's Resource variables will have already been updated with the new values so they'll be safe to use.

Any configuration change that you don't explicitly flag as being handled by your application will still cause an application restart without a call to `onConfigurationChanged`.

A Closer Look at Android Activities

To create user-interface screens for your applications, you extend the `Activity` class, using Views to provide user interaction.

Each Activity represents a screen (similar to the concept of a Form in desktop development) that an application can present to its users. The more complicated your application, the more screens you are likely to need.

You'll need to create a new Activity for every screen you want to display. Typically this includes at least a primary interface screen that handles the main UI functionality of your application. This is often supported by secondary Activities for entering information, providing different perspectives on your data, and supporting additional functionality. To move between screens in Android, you start a new Activity (or return from one).

Most Activities are designed to occupy the entire display, but you can create Activities that are semi-transparent, floating, or use dialog boxes.

Creating an Activity

To create a new Activity, you extend the `Activity` class, defining the user interface and implementing your functionality. The basic skeleton code for a new Activity is shown below:

```
package com.paad.myapplication;

import android.app.Activity;
import android.os.Bundle;

public class MyActivity extends Activity {

  /** Called when the activity is first created. */
  @Override
  public void onCreate(Bundle icicle) {
    super.onCreate(icicle);
  }
}
```

The base Activity class presents an empty screen that encapsulates the window display handling functionality. An empty Activity isn't particularly useful, so the first thing you'll want to do is lay out the screen interface using Views and layouts.

Activity UIs are created using Views. Views are the user-interface controls that display data and provide user interaction. Android provides several layout classes, called *View Groups*, that can contain multiple Views to help you design compelling user interfaces.

Chapter 4 examines Views and View Groups in detail, detailing what's available, how to use them, and how to create your own Views and layouts.

To assign a user interface to an Activity, call `setContentView` from the `onCreate` method of your Activity.

In this first snippet, a simple instance of `MyView` is used as the Activity's user interface:

```
@Override
public void onCreate(Bundle icicle) {
  super.onCreate(icicle);
  MyView myView = new MyView(this);
  setContentView(myView);
}
```

More commonly you'll want to use a more complex UI design. You can create a layout in code using layout View Groups, or you can use the standard Android convention of passing a resource ID for a layout defined in an external resource, as shown in the snippet below:

```
@Override
public void onCreate(Bundle icicle) {
  super.onCreate(icicle);
  setContentView(R.layout.main);
}
```

In order to use an Activity in your application, you need to register it in the manifest. Add new `activity` tags within the `application` node of the manifest; the `activity` tag includes attributes for metadata such as the label, icon, required permissions, and themes used by the Activity. An Activity without a corresponding `activity` tag can't be started.

The following XML snippet shows how to add a node for the `MyActivity` class created in the snippets above:

```
<activity android:label="@string/app_name"
          android:name=".MyActivity">
  <intent-filter>
    <action android:name="android.intent.action.MAIN" />
    <category android:name="android.intent.category.LAUNCHER" />
  </intent-filter>
</activity>
```

Within the `activity` tag, you can add `intent-filter` nodes that specify the Intents your Activity will listen for and react to. Each Intent Filter defines one or more actions and categories that your Activity supports. Intents and Intent Filters are covered in depth in Chapter 5, but it's worth noting that to make an Activity available from the main program launcher, it must include an Intent Filter listening for the `Main` action and the `Launcher` category, as highlighted in the snippet below:

```
<activity android:label="@string/app_name"
          android:name=".MyActivity">
  <intent-filter>
    <action android:name="android.intent.action.MAIN" />
    <category android:name="android.intent.category.LAUNCHER" />
  </intent-filter>
</activity>
```

The Activity Life Cycle

A good understanding of the Activity life cycle is vital to ensure that your application provides a seamless user experience and properly manages its resources.

As explained earlier, Android applications do not control their own process lifetimes; the Android run time manages the process of each application, and by extension that of each Activity within it.

While the run time handles the termination and management of an Activity's process, the Activity's state helps determine the priority of its parent application. The application priority, in turn, influences the likelihood that the run time will terminate it and the Activities running within it.

Activity Stacks

The state of each Activity is determined by its position on the Activity stack, a last-in–first-out collection of all the currently running Activities. When a new Activity starts, the current foreground screen is moved to the top of the stack. If the user navigates back using the Back button, or the foreground Activity is closed, the next Activity on the stack moves up and becomes active. This process is illustrated in Figure 3-7.

As described previously in this chapter, an application's priority is influenced by its highest-priority Activity. The Android memory manager uses this stack to determine the priority of applications based on their Activities when deciding which application to terminate to free resources.

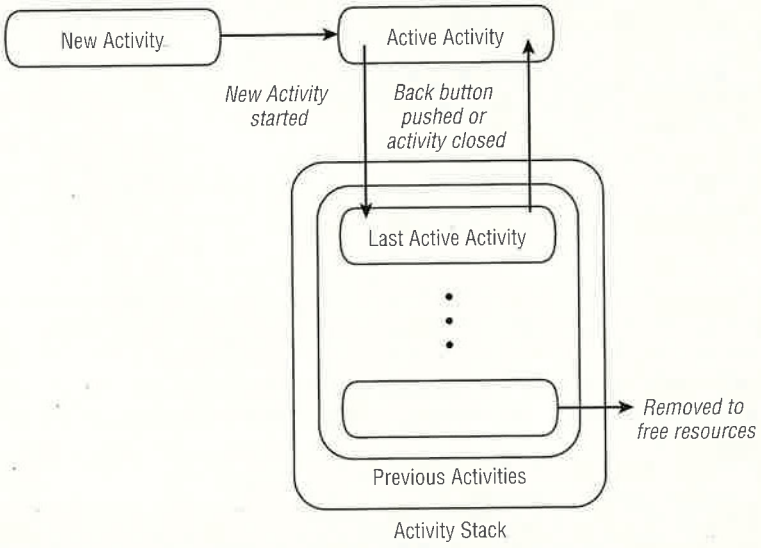


Figure 3-7

Activity States

As activities are created and destroyed, they move in and out of the stack shown in Figure 3-7. As they do so, they transition through four possible states:

- **Active** When an Activity is at the top of the stack, it is the visible, focused, foreground activity that is receiving user input. Android will attempt to keep it alive at all costs, killing Activities further down the stack as needed, to ensure that it has the resources it needs. When another Activity becomes active, this one will be paused.
- **Paused** In some cases, your Activity will be visible but will not have focus; at this point, it's paused. This state is reached if a transparent or non-full-screen Activity is active in front of it. When paused, an Activity is treated as if it were active; however, it doesn't receive user input events. In extreme cases, Android will kill a paused Activity to recover resources for the active Activity. When an Activity becomes totally obscured, it becomes stopped.
- **Stopped** When an Activity isn't visible, it "stops." The Activity will remain in memory retaining all state and member information; however, it is now a prime candidate for execution when the system requires memory elsewhere. When an Activity is stopped, it's important to save data and the current UI state. Once an Activity has exited or closed, it becomes inactive.
- **Inactive** After an Activity has been killed, and before it's been launched, it's inactive. Inactive Activities have been removed from the Activity stack and need to be restarted before they can be displayed and used.

State transitions are nondeterministic and are handled entirely by the Android memory manager. Android will start by closing applications that contain inactive Activities, followed by those that are stopped, and in extreme cases, it will remove those that are paused.

To ensure a seamless user experience, transitions between these states should be invisible to the user. There should be no difference between an Activity moving from paused, stopped, or killed states back to active, so it's important to save all UI state changes and persist all data when an Activity is paused or stopped. Once an Activity does become active, it should restore those saved values.

Monitoring State Changes

To ensure that Activities can react to state changes, Android provides a series of event handlers that are fired when an Activity transitions through its full, visible, and active lifetimes. Figure 3-8 summarizes these lifetimes in terms of the Activity states described above.

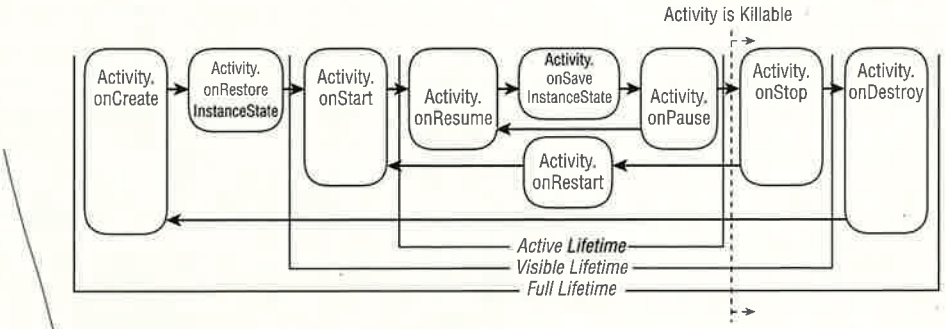


Figure 3-8

The following skeleton code shows the stubs for the state change method handlers available in an Activity. Comments within each stub describe the actions you should consider taking on each state change event.

```
package com.paad.myapplication;

import android.app.Activity;
import android.os.Bundle;

public class MyActivity extends Activity {

  // Called at the start of the full lifetime.
  @Override
  public void onCreate(Bundle icicle) {
    super.onCreate(icicle);
    // Initialize activity.
  }

  // Called after onCreate has finished, use to restore UI state
  @Override
  public void onRestoreInstanceState(Bundle savedInstanceState) {
    super.onRestoreInstanceState(savedInstanceState);
    // Restore UI state from the savedInstanceState.
    // This bundle has also been passed to onCreate.
  }

  // Called before subsequent visible lifetimes
  // for an activity process.
  @Override
  public void onRestart(){
    super.onRestart();
    // Load changes knowing that the activity has already
    // been visible within this process.
  }

  // Called at the start of the visible lifetime.
  @Override
  public void onStart(){
    super.onStart();
    // Apply any required UI change now that the Activity is visible.
  }

  // Called at the start of the active lifetime.
  @Override
  public void onResume(){
    super.onResume();
    // Resume any paused UI updates, threads, or processes required
    // by the activity but suspended when it was inactive.
  }

  // Called to save UI state changes at the
  // end of the active lifecycle.
  @Override
  public void onSaveInstanceState(Bundle savedInstanceState) {
    // Save UI state changes to the savedInstanceState.
```

```
    // This bundle will be passed to onCreate if the process is
    // killed and restarted.
    super.onSaveInstanceState(savedInstanceState);
  }

  // Called at the end of the active lifetime.
  @Override
  public void onPause(){
    // Suspend UI updates, threads, or CPU intensive processes
    // that don't need to be updated when the Activity isn't
    // the active foreground activity.
    super.onPause();
  }

  // Called at the end of the visible lifetime.
  @Override
  public void onStop(){
    // Suspend remaining UI updates, threads, or processing
    // that aren't required when the Activity isn't visible.
    // Persist all edits or state changes
    // as after this call the process is likely to be killed.
    super.onStop();
  }

  // Called at the end of the full lifetime.
  @Override
  public void onDestroy(){
    // Clean up any resources including ending threads,
    // closing database connections etc.
    super.onDestroy();
  }
}
```

As shown in the snippet above, you should always call back to the superclass when overriding these event handlers.

Understanding Activity Lifetimes

Within an Activity's full lifetime, between creation and destruction, it will go through one or more iterations of the active and visible lifetimes. Each transition will trigger the method handlers described previously. The following sections provide a closer look at each of these lifetimes and the events that bracket them.

The Full Lifetime

The full lifetime of your Activity occurs between the first call to `onCreate` and the final call to `onDestroy`. It's possible, in some cases, for an Activity's process to be terminated *without* the `onDestroy` method being called.

Use the `onCreate` method to initialize your Activity: Inflate the user interface, allocate references to class variables, bind data to controls, and create Services and threads. The `onCreate` method is passed a `Bundle` object containing the UI state saved in the last call to `onSaveInstanceState`. You should use this Bundle to restore the user interface to its previous state, either in the `onCreate` method or by overriding `onRestoreInstanceStateMethod`.

Override onDestroy to clean up any resources created in onCreate, and ensure that all external connections, such as network or database links, are closed.

As part of Android's guidelines for writing efficient code, it's recommended that you avoid the creation of short-term objects. Rapid creation and destruction of objects forces additional garbage collection, a process that can have a direct impact on the user experience. If your Activity creates the same set of objects regularly, consider creating them in the onCreate method instead, as it's called only once in the Activity's lifetime.

The Visible Lifetime

An Activity's visible lifetimes are bound between calls to onStart and onStop. Between these calls, your Activity will be visible to the user, although it may not have focus and might be partially obscured. Activities are likely to go through several visible lifetimes during their full lifetime, as they move between the foreground and background. While unusual, in extreme cases, the Android run time will kill an Activity during its visible lifetime without a call to onStop.

The onStop method should be used to pause or stop animations, threads, timers, Services, or other processes that are used exclusively to update the user interface. There's little value in consuming resources (such as CPU cycles or network bandwidth) to update the UI when it isn't visible. Use the onStart (or onRestart) methods to resume or restart these processes when the UI is visible again.

The onRestart method is called immediately prior to all but the first call to onStart. Use it to implement special processing that you want done only when the Activity restarts within its full lifetime.

The onStart/onStop methods are also used to register and unregister Broadcast Receivers that are being used exclusively to update the user interface. It will not always be necessary to unregister Receivers when the Activity becomes invisible, particularly if they are used to support actions other than updating the UI. You'll learn more about using Broadcast Receivers in Chapter 5.

The Active Lifetime

The active lifetime starts with a call to onResume and ends with a corresponding call to onPause.

An active Activity is in the foreground and is receiving user input events. Your Activity is likely to go through several active lifetimes before it's destroyed, as the active lifetime will end when a new Activity is displayed, the device goes to sleep, or the Activity loses focus. Try to keep code in the onPause and onResume methods relatively fast and lightweight to ensure that your application remains responsive when moving in and out of the foreground.

Immediately before onPause, a call is made to onSaveInstanceState. This method provides an opportunity to save the Activity's UI state in a Bundle that will be passed to the onCreate and onRestoreInstanceState methods. Use onSaveInstanceState to save the UI state (such as check button states, user focus, and entered but uncommitted user input) to ensure that the Activity can present the same UI when it next becomes active. During the active lifetime, you can safely assume that onSaveInstanceState and onPause will be called before the process is terminated.

Most Activity implementations will override at least the onPause method to commit unsaved changes, as it marks the point beyond which an Activity may be killed without warning. Depending on your application architecture, you may also choose to suspend threads, processes, or Broadcast Receivers while your Activity is not in the foreground.

The `onResume` method can be very lightweight. You will not need to reload the UI state here as this is handled by the `onCreate` and `onRestoreInstanceState` methods when required. Use `onResume` to re-register any Broadcast Receivers or other processes you may have stopped in `onPause`.

Android Activity Classes

The Android SDK includes a selection of Activity subclasses that wrap up the use of common user interface widgets. Some of the more useful ones are listed below:

- **MapActivity** Encapsulates the resource handling required to support a `MapView` widget within an Activity. Learn more about `MapActivity` and `MapView` in Chapter 7.
- **ListActivity** Wrapper class for Activities that feature a `ListView` bound to a data source as the primary UI metaphor, and exposing event handlers for list item selection
- **ExpandableListActivity** Similar to the List Activity but supporting an `ExpandableListView`
- **ActivityGroup** Allows you to embed multiple Activities within a single screen.

Summary

In this chapter, you learned how to design robust applications using loosely coupled application components: Activities, Services, Content Providers, Intents, and Broadcast Receivers bound together using the application manifest.

You were introduced to the Android application life cycle, learning how each application's priority is determined by its process state, which is, in turn, determined by the state of the components within it.

To take full advantage of the wide range of device hardware available and the international user base, you learned how to create external resources and how to define alternative values for specific locations, languages, and hardware configurations.

Next you discovered more about Activities and their role in the application framework. As well as learning how to create new Activities, you were introduced to the Activity life cycle. In particular, you learned about Activity state transitions and how to monitor these events to ensure a seamless user experience.

Finally, you were introduced to some specialized Android Activity classes.

In the next chapter, you'll learn how to create User Interfaces. Chapter 4 will demonstrate how to use layouts to design your UI before introducing some native widgets and showing you how to extend, modify, and group them to create specialized controls. You'll also learn how to create your own unique user interface elements from a blank canvas, before being introduced to the Android menu system.

THE UNIVERSITY OF MICHIGAN LIBRARIES

4

Creating User Interfaces

It's vital to create compelling and intuitive User Interfaces for your applications. Ensuring that they are as stylish and easy to use as they are functional should be a primary design consideration.

To quote Stephen Fry on the importance of style as *part* of substance in the design of digital devices:

> *As if a device can function if it has no style. As if a device can be called stylish that does not function superbly. ... yes, beauty matters. Boy, does it matter. It is not surface, it is not an extra, it is the thing itself.*
>
> — *Stephen Fry,* The Guardian *(October 27, 2007)*

Increasing screen sizes, display resolutions, and mobile processor power has seen mobile applications become increasingly visual. While the diminutive screens pose a challenge for creating complex visual interfaces, the ubiquity of mobiles makes it a challenge worth accepting.

In this chapter, you'll learn the basic Android UI elements and discover how to use Views, View Groups, and layouts to create functional and intuitive User Interfaces for your Activities.

After being introduced to some of the controls available from the Android SDK, you'll learn how to extend and customize them. Using View Groups, you'll see how to combine Views to create atomic, reusable UI elements made up of interacting subcontrols. You'll also learn how to create your own Views to implement creative new ways to display data and interact with users.

The individual elements of an Android User Interface are arranged on screen using a variety of layout managers derived from `ViewGroup`. Correctly using layouts is essential for creating good interfaces; this chapter introduces several native layout classes and demonstrates how to use them and how to create your own.

THE UNIVERSITY OF MICHIGAN LIBRARIES

Android's application and context menu systems use a new approach, optimized for modern touch-screen devices. As part of an examination of the Android UI model, this chapter ends with a look at how to create and use Activity and context menus.

Fundamental Android UI Design

User Interface design, human–computer interaction, and usability are huge topics that aren't covered in great depth in this book. Nonetheless, it's important that you get them right when creating your User Interfaces.

Android introduces some new terminology for familiar programming metaphors that will be explored in detail in the following sections:

- **Views** *Views* are the basic User Interface class for visual interface elements (commonly known as controls or widgets). All User Interface controls, and the layout classes, are derived from Views.
- **ViewGroups** *View Groups* are extensions of the View class that can contain multiple child Views. By extending the `ViewGroup` class, you can create compound controls that are made up of interconnected child Views. The `ViewGroup` class is also extended to provide the layout managers, such as `LinearLayout`, that help you compose User Interfaces.
- **Activities** *Activities*, described in detail in the previous chapter, represent the window or screen being displayed to the user. Activities are the Android equivalent of a Form. To display a User Interface, you assign a View or layout to an Activity.

Android provides several common UI controls, widgets, and layout managers.

For most graphical applications, it's likely that you'll need to extend and modify these standard controls — or create composite or entirely new controls — to provide your own functionality.

Introducing Views

As described above, all visual components in Android descend from the `View` class and are referred to generically as *Views*. You'll often see Views referred to as *controls* or *widgets* — terms you're probably familiar with if you've done any GUI development.

The `ViewGroup` class is an extension of View designed to contain multiple Views. Generally, View Groups are either used to construct atomic reusable components (widgets) or to manage the layout of child Views. View Groups that perform the latter function are generally referred to as *layouts*.

Because all visual elements derive from Views, many of the terms above are interchangeable. By convention, a *control* usually refers to an extension of Views that implements relatively simple functionality, while a *widget* generally refers to both compound controls and more complex extensions of Views.

The conventional naming model is shown in Figure 4-1. In practice, you will likely see both *widget* and *control* used interchangeably with *View*.

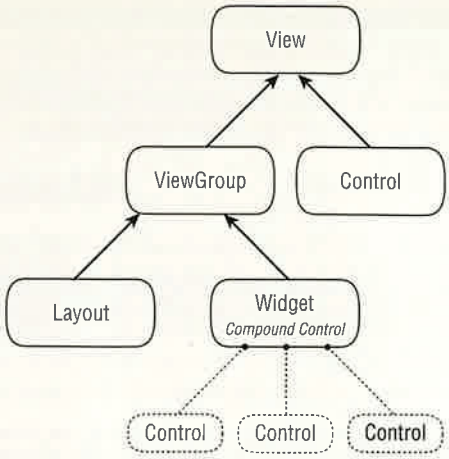


Figure 4-1

You've already been introduced to a layout and two widgets — the `LinearLayout`, a `ListView`, and a `TextView` — when you created the To-Do List example in Chapter 2.

In the following sections, you'll learn how to put together increasingly complex UIs, starting with the Views available in the SDK, before learning how to extend them, build your own compound controls, and create your own custom Views from scratch.

Creating Activity User Interfaces with Views

A new Activity starts with a temptingly empty screen onto which you place your User Interface. To set the User Interface, call `setContentView`, passing in the View instance (typically a layout) to display. Because empty screens aren't particularly inspiring, you will almost always use `setContentView` to assign an Activity's User Interface when overriding its `onCreate` handler.

The `setContentView` method accepts either a layout resource ID (as described in Chapter 3) or a single View instance. This lets you define your User Interface either in code or using the preferred technique of external layout resources.

Using layout resources decouples your presentation layer from the application logic, providing the flexibility to change the presentation without changing code. This makes it possible to specify different layouts optimized for different hardware configurations, even changing them at run time based on hardware changes (such as screen orientation).

The following code snippet shows how to set the User Interface for an Activity using an external layout resource. You can get references to the Views used within a layout with the `findViewById` method. This example assumes that main.xml exists in the project's `res/layout` folder.

```
@Override
public void onCreate(Bundle icicle) {
  super.onCreate(icicle);

  setContentView(R.layout.main);
```


THE UNIVERSITY OF MICHIGAN LIBRARIES

```
    TextView myTextView = (TextView)findViewById(R.id.myTextView);
}
```

If you prefer the more *traditional* approach, you can specify the User Interface in code. The following snippet shows how to assign a new `TextView` as the User Interface:

```
@Override
public void onCreate(Bundle icicle) {
  super.onCreate(icicle);

  TextView myTextView = new TextView(this);
  setContentView(myTextView);
  myTextView.setText("Hello, Android");
}
```

The `setContentView` method accepts a single View instance; as a result, you have to group multiple controls to ensure that you can reference a layout using a single View or View Group.

The Android Widget Toolbox

Android supplies a toolbox of standard Views to help you create simple interfaces. By using these controls (and modifying or extending them as necessary), you can simplify your development and provide consistency between applications.

The following list highlights some of the more familiar toolbox controls:

- `TextView` A standard read only text label. It supports multiline display, string formatting, and automatic word wrapping.
- `EditText` An editable text entry box. It accepts multiline entry and word wrapping.
- `ListView` A View Group that creates and manages a group of Views used to display the items in a List. The standard ListView displays the string value of an array of objects using a Text View for each item.
- `Spinner` Composite control that displays a TextView and an associated ListView that lets you select an item from a list to display in the textbox. It's made from a Text View displaying the current selection, combined with a button that displays a selection dialog when pressed.
- `Button` Standard push-button
- `CheckBox` Two-state button represented with a checked or unchecked box
- `RadioButton` Two-state grouped buttons. Presents the user with a number of binary options of which only one can be selected at a time.

This is only a selection of the widgets available. Android also supports several more advanced View implementations including date-time pickers, auto-complete input boxes, maps, galleries, and tab sheets. For a more comprehensive list of the available widgets, head to `http://code.google.com/android/reference/view-gallery.html`.

It's only a matter of time before you, as an innovative developer, encounter a situation in which none of the built-in controls meets your needs. Later in this chapter, you'll learn how to extend and combine the existing controls and how to design and create entirely new widgets from scratch.

Introducing Layouts

Layout Managers (more generally, "layouts") are extensions of the `ViewGroup` class designed to control the position of child controls on a screen. Layouts can be nested, letting you create arbitrarily complex interfaces using a combination of Layout Managers.

The Android SDK includes some simple layouts to help you construct your UI. It's up to you to select the right combination of layouts to make your interface easy to understand and use.

The following list includes some of the more versatile layout classes available:

- **FrameLayout** The simplest of the Layout Managers, the *Frame Layout* simply pins each child view to the top left corner. Adding multiple children stacks each new child on top of the previous, with each new View obscuring the last.
- **LinearLayout** A *Linear Layout* adds each child View in a straight line, either vertically or horizontally. A vertical layout has one child View per row, while a horizontal layout has a single row of Views. The Linear Layout Manager allows you to specify a "weight" for each child View that controls the relative size of each within the available space.
- **RelativeLayout** Using the *Relative Layout*, you can define the positions of each of the child Views relative to each other and the screen boundaries.
- **TableLayout** The *Table Layout* lets you lay out Views using a grid of rows and columns. Tables can span multiple rows and columns, and columns can be set to shrink or grow.
- **AbsoluteLayout** In an *Absolute Layout*, each child View's position is defined in absolute coordinates. Using this class, you can guarantee the exact layout of your components, but at a price. Compared to the previous managers, describing a layout in absolute terms means that your layout can't dynamically adjust for different screen resolutions and orientations.

The Android documentation describes the features and properties of each layout class in detail, so rather than repeating it here, I'll refer you to `http://code.google.com/android/devel/ui/layout.html`.

Later in this chapter, you'll also learn how to create compound controls (widgets made up of several interconnected Views) by extending these layout classes.

Using Layouts

The preferred way to implement layouts is in XML using external resources. A layout XML must contain a single root element. This root node can contain as many nested layouts and Views as necessary to construct an arbitrarily complex screen.

THE UNIVERSITY OF MICHIGAN LIBRARIES

The following XML snippet shows a simple layout that places a `TextView` above an `EditText` control using a `LinearLayout` configured to lay out vertically:

```
<?xml version="1.0" encoding="utf-8"?>
<LinearLayout xmlns:android="http://schemas.android.com/apk/res/android"
  android:orientation="vertical"
  android:layout_width="fill_parent"
  android:layout_height="fill_parent">
  <TextView
    android:layout_width="fill_parent"
    android:layout_height="wrap_content"
    android:text="Enter Text Below"
  />
  <EditText
    android:layout_width="fill_parent"
    android:layout_height="wrap_content"
    android:text="Text Goes Here!"
  />
</LinearLayout>
```

Implementing layouts in XML decouples the presentation layer from View and Activity code. It also lets you create hardware-specific variations that are dynamically loaded without requiring code changes.

When it's preferred, or required, you can implement layouts in code. When assigning Views to layouts, it's important to apply `LayoutParameters` using the `setLayoutParams` method, or passing them in to the `addView` call as shown below:

```
LinearLayout ll = new LinearLayout(this);
ll.setOrientation(LinearLayout.VERTICAL);

TextView myTextView = new TextView(this);
EditText myEditText = new EditText(this);

myTextView.setText("Enter Text Below");
myEditText.setText("Text Goes Here!");

int lHeight = LinearLayout.LayoutParams.FILL_PARENT;
int lWidth = LinearLayout.LayoutParams.WRAP_CONTENT;

ll.addView(myTextView, new LinearLayout.LayoutParams(lHeight, lWidth));
ll.addView(myEditText, new LinearLayout.LayoutParams(lHeight, lWidth));
setContentView(ll);
```

Creating New Views

The ability to extend existing Views, create composite widgets, and create unique new controls lets you create beautiful User Interfaces optimized for your particular workflow. Android lets you subclass the existing widget toolbox and implement your own View controls, giving you total freedom to tailor your User Interface to maximize the user experience.

When you design a User Interface, it's important to balance raw aesthetics and usability. With the power to create your own custom controls comes the temptation to rebuild all of them from scratch. Resist that urge. The standard widgets will be familiar to users from other Android applications. On small screens with users often paying limited attention, familiarity can often provide better usability than a slightly shinier widget.

Deciding on your approach when creating a new View depends on what you want to achieve:

- Modify or extend the appearance and/or behavior of an existing control when it already supplies the basic functionality you want. By overriding the event handlers and `onDraw`, but still calling back to the superclass's methods, you can customize the control without having to reimplement its functionality. For example, you could customize a `TextView` to display a set number of decimal points.
- Combine controls to create atomic, reusable widgets that leverage the functionality of several interconnected controls. For example, you could create a dropdown combo box by combining a `TextView` and a `Button` that displays a floating `ListView` when clicked.
- Create an entirely new control when you need a completely different interface that can't be achieved by changing or combining existing controls.

Modifying Existing Views

The toolbox includes a lot of common UI requirements, but the controls are necessarily generic. By customizing these basic Views, you avoid reimplementing existing behavior while still tailoring the User Interface, and functionality, of each control to your application's needs.

To create a new widget based on an existing control, create a new class that extends it — as shown in the following skeleton code that extends `TextView`:

```
import android.content.Context;
import android.util.AttributeSet;
import android.widget.TextView;

public class MyTextView extends TextView {

  public MyTextView (Context context, AttributeSet attrs, int defStyle)
  {
    super(context, attrs, defStyle);
  }

  public MyTextView (Context context) {
    super(context);
  }

  public MyTextView (Context context, AttributeSet attrs) {
    super(context, attrs);
  }
}
```

To override the appearance or behavior of your new View, override and extend the event handlers associated with the behavior you want to change.

In the following skeleton code, the onDraw method is overridden to modify the View's appearance, and the onKeyDown handler is overridden to allow custom key press handling:

```
public class MyTextView extends TextView {

  public MyTextView (Context context, AttributeSet ats, int defStyle) {
    super(context, ats, defStyle);
  }

  public MyTextView (Context context) {
    super(context);
  }

  public MyTextView (Context context, AttributeSet attrs) {
    super(context, attrs);
  }

  @Override
  public void onDraw(Canvas canvas) {
    [ ... Draw things on the canvas under the text ... ]

    // Render the text as usual using the TextView base class.
    super.onDraw(canvas);

    [ ... Draw things on the canvas over the text ... ]
  }

  @Override
  public boolean onKeyDown(int keyCode, KeyEvent keyEvent) {
    [ ... Perform some special processing ... ]
    [ ... based on a particular key press ... ]

    // Use the existing functionality implemented by
    // the base class to respond to a key press event.
    return super.onKeyDown(keyCode, keyEvent);
  }
}
```

The User Interface event handlers available within Views are covered in more detail later in this chapter.

Customizing Your To-Do List

The To-Do List example from Chapter 2 uses TextViews (within a List View) to display each item. You can customize the appearance of the list by creating a new extension of the Text View, overriding the onDraw method.

In this example, you'll create a new TodoListItemView that will make each item appear as if on a paper pad. When complete, your customized To-Do List should look like Figure 4-2.



Figure 4-2

1. Create a new `TodoListItemView` class that extends TextView. Include a stub for overriding the `onDraw` method, and implement constructors that call a new `init` method stub.

```
package com.paad.todolist;

import android.content.Context;
import android.content.res.Resources;
import android.graphics.Canvas;
import android.graphics.Paint;
import android.util.AttributeSet;
import android.widget.TextView;

public class TodoListItemView extends TextView {

  public TodoListItemView (Context context, AttributeSet ats, int ds) {
    super(context, ats, ds);
    init();
  }

  public TodoListItemView (Context context) {
    super(context);
    init();
  }

  public TodoListItemView (Context context, AttributeSet attrs) {
    super(context, attrs);
    init();
  }

  private void init() {
  }

  @Override
  public void onDraw(Canvas canvas) {
    // Use the base TextView to render the text.
    super.onDraw(canvas);
  }

}
```

2. Create a new `colors.xml` resource in the `res/values` folder. Create new color values for the paper, margin, line, and text colors.

```
<?xml version="1.0" encoding="utf-8"?>
<resources>
  <color name="notepad_paper">#AAFFFF99</color>
  <color name="notepad_lines">#FF0000FF</color>
  <color name="notepad_margin">#90FF0000</color>
  <color name="notepad_text">#AA0000FF</color>
</resources>
```

3. Create a new `dimens.xml` resource file, and add a new value for the paper's margin width.

```
<?xml version="1.0" encoding="utf-8"?>
<resources>
  <dimen name="notepad_margin">30px</dimen>
</resources>
```

4. With the resources defined, you're ready to customize the `TodoListItemView` appearance. Create new private instance variables to store the `Paint` objects you'll use to draw the paper background and margin. Also create variables for the paper color and margin width values.

   Fill in the `init` method to get instances of the resources you created in the last two steps and create the `Paint` objects.

```
private Paint marginPaint;
private Paint linePaint;
private int paperColor;
private float margin;

private void init() {
  // Get a reference to our resource table.
  Resources myResources = getResources();

  // Create the paint brushes we will use in the onDraw method.
  marginPaint = new Paint(Paint.ANTI_ALIAS_FLAG);
  marginPaint.setColor(myResources.getColor(R.color.notepad_margin));
  linePaint = new Paint(Paint.ANTI_ALIAS_FLAG);
  linePaint.setColor(myResources.getColor(R.color.notepad_lines));

  // Get the paper background color and the margin width.
  paperColor = myResources.getColor(R.color.notepad_paper);
  margin = myResources.getDimension(R.dimen.notepad_margin);
}
```

5. To draw the paper, override `onDraw`, and draw the image using the Paint objects you created in Step 4. Once you've drawn the paper image, call the superclass's `onDraw` method, and let it draw the text as usual.

```
@Override
public void onDraw(Canvas canvas) {
  // Color as paper
  canvas.drawColor(paperColor);

  // Draw ruled lines
```

```
        canvas.drawLine(0, 0, getMeasuredHeight(), 0, linePaint);
        canvas.drawLine(0, getMeasuredHeight(),
                        getMeasuredWidth(), getMeasuredHeight(),
                        linePaint);

        // Draw margin
        canvas.drawLine(margin, 0, margin, getMeasuredHeight(), marginPaint);

        // Move the text across from the margin
        canvas.save();
        canvas.translate(margin, 0);

        // Use the TextView to render the text.
        super.onDraw(canvas);
        canvas.restore();
    }
```

6. That completes the `TodoListItemView` implementation. To use it in the To-Do List Activity, you need to include it in a new layout and pass that in to the Array Adapter constructor.

   Start by creating a new todolist_item.xml resource in the `res/layout` folder. It will specify how each of the to-do list items is displayed. For this example, your layout need only consist of the new `TodoListItemView`, set to fill the entire available area.

```
<?xml version="1.0" encoding="utf-8"?>
<com.paad.todolist.TodoListItemView
    xmlns:android="http://schemas.android.com/apk/res/android"
    android:layout_width="fill_parent"
    android:layout_height="fill_parent"
    android:padding="10dp"
    android:scrollbars="vertical"
    android:textColor="@color/notepad_text"
    android:fadingEdge="vertical"
/>
```

7. Now open the `ToDoList` Activity class. The final step is to change the parameters passed in to the `ArrayAdapter` in `onCreate`. Replace the reference to the default `android.R.layout.simple_list_item_1` with the new `R.layout.todolist_item` layout created in Step 6.

```
final ArrayList<String> todoItems = new ArrayList<String>();
int resID = R.layout.todolist_item;
final ArrayAdapter<String> aa = new ArrayAdapter<String>(this, resID, todoItems);
myListView.setAdapter(aa);
```

Creating Compound Controls

Compound controls are atomic, reusable widgets that contain multiple child controls laid out and wired together.

When you create a compound control, you define the layout, appearance, and interaction of the Views it contains. Compound controls are created by extending a `ViewGroup` (usually a Layout Manager). To

create a new compound control, choose a layout class that's most suitable for positioning the child controls, and extend it as shown in the skeleton code below:

```
public class MyCompoundView extends LinearLayout {

  public MyCompoundView(Context context) {
    super(context);
  }

  public MyCompoundView(Context context, AttributeSet attrs) {
    super(context, attrs);
  }

}
```

As with an Activity, the preferred way to design the UI for a compound control is to use a layout resource. The following code snippet shows the XML layout definition for a simple widget consisting of an Edit Text box and a button to clear it:

```
<?xml version="1.0" encoding="utf-8"?>
<LinearLayout xmlns:android="http://schemas.android.com/apk/res/android"
  android:orientation="vertical"
  android:layout_width="fill_parent"
  android:layout_height="fill_parent">
  <EditText
    android:id="@+id/editText"
    android:layout_width="fill_parent"
    android:layout_height="wrap_content"
  />
  <Button
    android:id="@+id/clearButton"
    android:layout_width="fill_parent"
    android:layout_height="wrap_content"
    android:text="Clear"
  />
</LinearLayout>
```

To use this layout for your new widget, override its constructor to inflate the layout resource using the `inflate` method from the `LayoutInflate` system service. The `inflate` method takes the layout resource and returns an inflated View. For circumstances such as this where the returned View should be the class you're creating, you can pass in a parent and attach the result to it automatically, as shown in the next code sample.

The following code snippet shows the `ClearableEditText` class. Within the constructor it inflates the layout resource created above and gets references to each of the Views it contains. It also makes a call to `hookupButton` that will be used to hookup the *clear text* functionality when the button is pressed.

```
public class ClearableEditText extends LinearLayout {

  EditText editText;
  Button clearButton;

  public ClearableEditText(Context context) {
```

```
        super(context);

        // Inflate the view from the layout resource.
        String infService = Context.LAYOUT_INFLATER_SERVICE;
        LayoutInflater li;
        li = (LayoutInflater)getContext().getSystemService(infService);
        li.inflate(R.layout.clearable_edit_text, this, true);

        // Get references to the child controls.
        editText = (EditText)findViewById(R.id.editText);
        clearButton = (Button)findViewById(R.id.clearButton);

        // Hook up the functionality
        hookupButton();
    }
}
```

If you'd prefer to construct your layout in code, you can do so just as you would for an Activity. The following code snippet shows the `ClearableEditText` constructor overridden to create the same UI as is defined in the XML used in the earlier example:

```
public ClearableEditText(Context context) {
  super(context);

  // Set orientation of layout to vertical
  setOrientation(LinearLayout.VERTICAL);

  // Create the child controls.
  editText = new EditText(getContext());
  clearButton = new Button(getContext());
  clearButton.setText("Clear");

  // Lay them out in the compound control.
  int lHeight = LayoutParams.WRAP_CONTENT;
  int lWidth = LayoutParams.FILL_PARENT;

  addView(editText, new LinearLayout.LayoutParams(lWidth, lHeight));
  addView(clearButton, new LinearLayout.LayoutParams(lWidth, lHeight));

  // Hook up the functionality
  hookupButton();
}
```

Once the screen has been constructed, you can hook up the event handlers for each child control to provide the functionality you need. In this next snippet, the `hookupButton` method is filled in to clear the textbox when the button is pressed:

```
private void hookupButton() {
  clearButton.setOnClickListener(new Button.OnClickListener() {
    public void onClick(View v) {
      editText.setText("");
    }
  });
}
```


THE UNIVERSITY OF MICHIGAN LIBRARIES

Creating Custom Widgets and Controls

Creating completely new Views gives you the power to fundamentally shape the way your applications look and feel. By creating your own controls, you can create User Interfaces that are uniquely suited to your users' needs. To create new controls from a blank canvas, you extend either the `View` or `SurfaceView` classes.

The `View` class provides a `Canvas` object and a series of draw methods and `Paint` classes, to create a visual interface using raster graphics. You can then override user events like screen touches or key presses to provide interactivity. In situations where extremely rapid repaints and 3D graphics aren't required, the `View` base class offers a powerful lightweight solution.

The `SurfaceView` provides a canvas that supports drawing from a background thread and using `openGL` for 3D graphics. This is an excellent option for graphics-heavy controls that are frequently updated or display complex graphical information, particularly games and 3D visualizations.

This chapter introduces 2D controls based on the `View` class. To learn more about the `SurfaceView` class and some of the more advanced Canvas paint features available in Android, see Chapter 11.

Creating a New Visual Interface

The base `View` class presents a distinctly empty 100 × 100 pixel square. To change the size of the control and display a more compelling visual interface, you need to override the `onMeasure` and `onDraw` methods, respectively.

Within `onMeasure`, the new View will calculate the height and width it will occupy given a set of boundary conditions. The `onDraw` method is where you draw on the Canvas to create the visual interface.

The following code snippet shows the skeleton code for a new `View` class, which will be examined further in the following sections:

```
public class MyView extends View {

  // Constructor required for in-code creation
  public MyView(Context context) {
    super(context);
  }

  // Constructor required for inflation from resource file
  public MyView (Context context, AttributeSet ats, int defaultStyle) {
    super(context, ats, defaultStyle );
  }

  //Constructor required for inflation from resource file
  public MyView (Context context, AttributeSet attrs) {
    super(context, attrs);
  }

  @Override
  protected void onMeasure(int wMeasureSpec, int hMeasureSpec) {
    int measuredHeight = measureHeight(hMeasureSpec);
```

```
    int measuredWidth = measureWidth(wMeasureSpec);

    // MUST make this call to setMeasuredDimension
    // or you will cause a runtime exception when
    // the control is laid out.
    setMeasuredDimension(measuredHeight, measuredWidth);
  }

  private int measureHeight(int measureSpec) {
    int specMode = MeasureSpec.getMode(measureSpec);
    int specSize = MeasureSpec.getSize(measureSpec);

     [ ... Calculate the view height ... ]

     return specSize;
  }

  private int measureWidth(int measureSpec) {
    int specMode = MeasureSpec.getMode(measureSpec);
    int specSize = MeasureSpec.getSize(measureSpec);

     [ ... Calculate the view width ... ]

     return specSize;
  }

  @Override
  protected void onDraw(Canvas canvas) {
    [ ... Draw your visual interface ... ]
  }
}
```

Note that the `onMeasure` method calls `setMeasuredDimension`; you must always call this method within your overridden `onMeasure` method or your control will throw an exception when the parent container attempts to lay it out.

Drawing Your Control

The `onDraw` method is where the magic happens. If you're creating a new widget from scratch, it's because you want to create a completely new visual interface. The Canvas parameter available in the `onDraw` method is the surface you'll use to bring your imagination to life.

Android provides a variety of tools to help draw your design on the Canvas using various `Paint` objects. The `Canvas` class includes helper methods to draw primitive 2D objects including circles, lines, rectangles, text, and Drawables (images). It also supports transformations that let you rotate, translate (move), and scale (resize) the Canvas while you draw on it.

Used in combination with `Drawables` and the `Paint` class (which offer a variety of customizable fills and pens), the complexity and detail that your control can render are limited only by the size of the screen and the power of the processor rendering it.

THE UNIVERSITY OF MICHIGAN LIBRARIES

One of the most important techniques for writing efficient code in Android is to avoid repetitive creation and destruction of objects. Any object created in your `onDraw` method will be created and destroyed every time the screen refreshes. Improve efficiency by making as many of these objects (in particular, instances of `Paint` and `Drawable`) class-scoped and moving their creation into the constructor.

The following code snippet shows how to override the `onDraw` method to display a simple text string in the center of the control:

```
@Override
protected void onDraw(Canvas canvas) {
  // Get the size of the control based on the last call to onMeasure.
  int height = getMeasuredHeight();
  int width = getMeasuredWidth();

  // Find the center
  int px = width/2;
  int py = height/2;

  // Create the new paint brushes.
  // NOTE: For efficiency this should be done in
  // the widget's constructor
  Paint mTextPaint = new Paint(Paint.ANTI_ALIAS_FLAG);
  mTextPaint.setColor(Color.WHITE);

  // Define the string.
  String displayText = "Hello World!";

  // Measure the width of the text string.
  float textWidth = mTextPaint.measureText(displayText);

  // Draw the text string in the center of the control.
  canvas.drawText(displayText, px-textWidth/2, py, mTextPaint);
}
```

Android offers a rich drawing library for the Canvas. Rather than diverge too far from the current topic, a more detailed look at the techniques available for drawing more complex visuals is included in Chapter 11.

Android does not currently support vector graphics. As a result, changes to any element of your canvas require repainting the entire canvas; modifying the color of a brush will not change your View's display until it is invalidated and redrawn. Alternatively, you can use OpenGL to render graphics; for more details, see the discussion on `SurfaceView` in Chapter 11.

Sizing Your Control

Unless you conveniently require a control that always occupies 100 × 100 pixels, you will also need to override `onMeasure`.

The `onMeasure` method is called when the control's parent is laying out its child controls. It asks the question, "How much space will you use?" and passes in two parameters — `widthMeasureSpec` and

heightMeasureSpec. They specify the space available for the control and some metadata describing that space.

Rather than return a result, you pass the View's height and width into the setMeasuredDimension method.

The following skeleton code shows how to override onMeasure. Note the calls to local method stubs calculateHeight and calculateWidth. They'll be used to decode the widthHeightSpec and heightMeasureSpec values and calculate the preferred height and width values.

```
@Override
protected void onMeasure(int widthMeasureSpec, int heightMeasureSpec) {

  int measuredHeight = measureHeight(heightMeasureSpec);
  int measuredWidth = measureWidth(widthMeasureSpec);

  setMeasuredDimension(measuredHeight, measuredWidth);
}

private int measureHeight(int measureSpec) {
  // Return measured widget height.
}

private int measureWidth(int measureSpec) {
  // Return measured widget width.
}
```

The boundary parameters, widthMeasureSpec and heightMeasureSpec, are passed in as integers for efficiency reasons. Before they can be used, they first need to be decoded using the static getMode and getSize methods from the MeasureSpec class, as shown in the snippet below:

```
int specMode = MeasureSpec.getMode(measureSpec);
int specSize = MeasureSpec.getSize(measureSpec);
```

Depending on the *mode* value, the *size* represents either the maximum space available for the control (in the case of AT_MOST), or the exact size that your control will occupy (for EXACTLY). In the case of UNSPECIFIED, your control does not have any reference for what the size represents.

By marking a measurement size as EXACT, the parent is insisting that the View will be placed into an area of the exact size specified. The AT_MOST designator says the parent is asking what size the View would like to occupy, given an upper bound. In many cases, the value you return will be the same.

In either case, you should treat these limits as absolute. In some circumstances, it may still be appropriate to return a measurement outside these limits, in which case you can let the parent choose how to deal with the oversized View, using techniques such as clipping and scrolling.

The following skeleton code shows a typical implementation for handling View measurement:

```
@Override
protected void onMeasure(int widthMeasureSpec, int heightMeasureSpec) {
  int measuredHeight = measureHeight(heightMeasureSpec);
```


THE UNIVERSITY OF MICHIGAN LIBRARIES

```
    int measuredWidth = measureWidth(widthMeasureSpec);

    setMeasuredDimension(measuredHeight, measuredWidth);
  }

  private int measureHeight(int measureSpec) {
    int specMode = MeasureSpec.getMode(measureSpec);
    int specSize = MeasureSpec.getSize(measureSpec);

    //  Default size if no limits are specified.
    int result = 500;

    if (specMode == MeasureSpec.AT_MOST) {
      // Calculate the ideal size of your
      // control within this maximum size.
      // If your control fills the available
      // space return the outer bound.
      result = specSize;
    } else if (specMode == MeasureSpec.EXACTLY) {
      // If your control can fit within these bounds return that value.
      result = specSize;
    }
    return result;
  }

  private int measureWidth(int measureSpec) {
    int specMode = MeasureSpec.getMode(measureSpec);
    int specSize = MeasureSpec.getSize(measureSpec);

    //  Default size if no limits are specified.
    int result = 500;

    if (specMode == MeasureSpec.AT_MOST) {
      // Calculate the ideal size of your control
      // within this maximum size.
      // If your control fills the available space
      // return the outer bound.
      result = specSize;
    } else if (specMode == MeasureSpec.EXACTLY) {
      // If your control can fit within these bounds return that value.
      result = specSize;
    }
    return result;
  }
```

Handling User Interaction Events

To make your new widget interactive, it will need to respond to user events like key presses, screen touches, and button clicks. Android exposes several virtual event handlers, listed below, that let you react to user input:

- `onKeyDown` Called when any device key is pressed; includes the D-pad, keyboard, hang-up, call, back, and camera buttons

- `onKeyUp` Called when a user releases a pressed key
- `onTrackballEvent` Called when the device's trackball is moved
- `onTouchEvent` Called when the touch screen is pressed or released, or it detects movement

The following code snippet shows a skeleton class that overrides each of the user interaction handlers in a View:

```
@Override
public boolean onKeyDown(int keyCode, KeyEvent keyEvent) {
  // Return true if the event was handled.
  return true;
}

@Override
public boolean onKeyUp(int keyCode, KeyEvent keyEvent) {
  // Return true if the event was handled.
  return true;
}

@Override
public boolean onTrackballEvent(MotionEvent event ) {
  // Get the type of action this event represents
  int actionPerformed = event.getAction();
  // Return true if the event was handled.
  return true;
}

@Override
public boolean onTouchEvent(MotionEvent event) {
  // Get the type of action this event represents
  int actionPerformed = event.getAction();
  // Return true if the event was handled.
  return true;
}
```

Further details on using each of these event handlers, including greater detail on the parameters received by each method, are available in Chapter 11.

Creating a Compass View Example

In the following example, you'll create a new Compass View by extending the `View` class. It uses a traditional compass rose to indicate a heading/orientation. When complete, it should appear as in Figure 4-3.

A compass is an example of a User Interface control that requires a radically different visual display from the textboxes and buttons available in the SDK toolbox, making it an excellent candidate for building from scratch.

In Chapter 10, you'll use this Compass View and the device's built-in accelerometer to display the user's current bearing. Then in Chapter 11, you will learn some advanced techniques for Canvas drawing that will let you dramatically improve its appearance.

THE UNIVERSITY OF MICHIGAN LIBRARIES

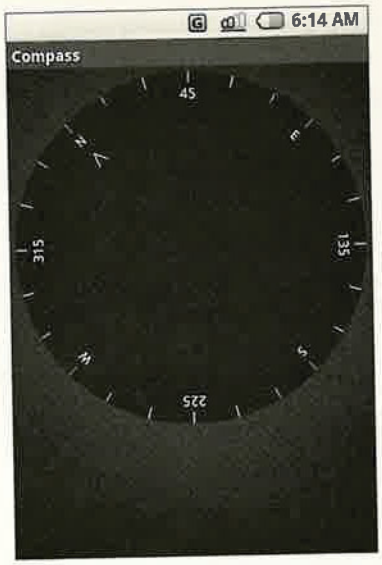


Figure 4-3

1. Create a new `Compass` project that will contain your new Compass View, and an Activity to hold it. Now create a new `CompassView` class that extends `View`. Create constructors that will allow the View to be instantiated in code, or through inflation from a resource layout. Add a new `initCompassView` method that will be used to initialize the control and call it from each constructor.

```
package com.paad.compass;

import android.content.Context;
import android.graphics.*;
import android.graphics.drawable.*;
import android.view.*;
import android.util.AttributeSet;
import android.content.res.Resources;

public class CompassView extends View {

  public CompassView(Context context) {
    super(context);
    initCompassView();
  }

  public CompassView(Context context, AttributeSet attrs) {
    super(context, attrs);
    initCompassView();
  }

  public CompassView(Context context,
                     AttributeSet ats,
                     int defaultStyle) {
    super(context, ats, defaultStyle);
```

```
    initCompassView();
  }

  protected void initCompassView() {
    setFocusable(true);
  }

}
```

2. The compass control should always be a perfect circle that takes up as much of the canvas as this restriction allows. Override the `onMeasure` method to calculate the length of the shortest side, and use `setMeasuredDimension` to set the height and width using this value.

```
@Override
protected void onMeasure(int widthMeasureSpec, int heightMeasureSpec) {
  // The compass is a circle that fills as much space as possible.
  // Set the measured dimensions by figuring out the shortest boundary,
  // height or width.
  int measuredWidth = measure(widthMeasureSpec);
  int measuredHeight = measure(heightMeasureSpec);

  int d = Math.min(measuredWidth, measuredHeight);

  setMeasuredDimension(d, d);
}

private int measure(int measureSpec) {
  int result = 0;

  // Decode the measurement specifications.
  int specMode = MeasureSpec.getMode(measureSpec);
  int specSize = MeasureSpec.getSize(measureSpec);

  if (specMode == MeasureSpec.UNSPECIFIED) {
    // Return a default size of 200 if no bounds are specified.
    result = 200;
  } else {
    // As you want to fill the available space
    // always return the full available bounds.
    result = specSize;
  }
  return result;
}
```

3. Create two new resource files that store the colors and text strings you'll use to draw the compass.

   3.1. Create the text string resource `res/values/strings.xml`.

```
<?xml version="1.0" encoding="utf-8"?>
<resources>
  <string name="app_name">Compass</string>
  <string name="cardinal_north">N</string>
  <string name="cardinal_east">E</string>
  <string name="cardinal_south">S</string>
  <string name="cardinal_west">W</string>
</resources>
```

3.2. Create the color resource `res/values/colors.xml`.

```
<?xml version="1.0" encoding="utf-8"?>
<resources>
  <color name="background_color">#F555</color>
  <color name="marker_color">#AFFF</color>
  <color name="text_color">#AFFF</color>
</resources>
```

4. Now return to the `CompassView` class. Add a new property for the bearing to display and create `get` and `set` methods for it.

```
private float bearing;

public void setBearing(float _bearing) {
  bearing = _bearing;
}
public float getBearing() {
  return bearing;
}
```

5. Next, return to the `initCompassView` method, and get references to each resource created in Step 3. Store the String values as instance variables, and use the color values to create new class-scoped `Paint` objects. You'll use these objects in the next step to draw the compass face.

```
private Paint markerPaint;
private Paint textPaint;
private Paint circlePaint;
private String northString;
private String eastString;
private String southString;
private String westString;
private int textHeight;

protected void initCompassView() {
  setFocusable(true);

  circlePaint = new Paint(Paint.ANTI_ALIAS_FLAG);
  circlePaint.setColor(R.color. background_color);
  circlePaint.setStrokeWidth(1);
  circlePaint.setStyle(Paint.Style.FILL_AND_STROKE);

  Resources r = this.getResources();
  northString = r.getString(R.string.cardinal_north);
  eastString = r.getString(R.string.cardinal_east);
  southString = r.getString(R.string.cardinal_south);
  westString = r.getString(R.string.cardinal_west);

  textPaint = new Paint(Paint.ANTI_ALIAS_FLAG);
  textPaint.setColor(r.getColor(R.color.text_color));

  textHeight = (int)textPaint.measureText("yY");

  markerPaint = new Paint(Paint.ANTI_ALIAS_FLAG);
  markerPaint.setColor(r.getColor(R.color.marker_color));
}
```

6. The final step is drawing the compass face using the `Strings` and `Paint` objects you created in Step 5. The following code snippet is presented with only limited commentary. You can find more detail on how to draw on the Canvas and use advanced Paint effects in Chapter 11.

   6.1. Start by overriding the `onDraw` method in the `CompassView` class.

```
@Override
protected void onDraw(Canvas canvas) {
```

   6.2. Find the center of the control, and store the length of the smallest side as the compass's radius.

```
int px = getMeasuredWidth() / 2;
int py = getMeasuredHeight() /2 ;

int radius = Math.min(px, py);
```

   6.3. Draw the outer boundary, and color the background of the compass face using the `drawCircle` method. Use the `circlePaint` object you created in Step 5.

```
// Draw the background
canvas.drawCircle(px, py, radius, circlePaint);
```

   6.4. This compass displays the current heading by rotating the face, so that the current direction is always at the top of the device. To do this, rotate the canvas in the opposite direction to the current heading.

```
// Rotate our perspective so that the 'top' is
// facing the current bearing.
canvas.save();
canvas.rotate(-bearing, px, py);
```

   6.5. All that's left is to draw the markings. Rotate the canvas through a full rotation, drawing markings every 15 degrees and the abbreviated direction string every 45 degrees.

```
int textWidth = (int)textPaint.measureText("W");
int cardinalX = px-textWidth/2;
int cardinalY = py-radius+textHeight;

// Draw the marker every 15 degrees and text every 45.
for (int i = 0; i < 24; i++) {
  // Draw a marker.
  canvas.drawLine(px, py-radius, px, py-radius+10, markerPaint);

  canvas.save();
  canvas.translate(0, textHeight);

  // Draw the cardinal points
  if (i % 6 == 0) {
    String dirString = "";
    switch (i) {
      case(0)  : {
                   dirString = northString;
                   int arrowY = 2*textHeight;
                   canvas.drawLine(px, arrowY, px-5, 3*textHeight,
                                   markerPaint);
```

```
                    canvas.drawLine(px, arrowY, px+5, 3*textHeight,
                                    markerPaint);
                    break;
                  }
                case(6)  : dirString = eastString; break;
                case(12) : dirString = southString; break;
                case(18) : dirString = westString; break;
              }
              canvas.drawText(dirString, cardinalX, cardinalY, textPaint);
            }

            else if (i % 3 == 0) {
              // Draw the text every alternate 45deg
              String angle = String.valueOf(i*15);
              float angleTextWidth = textPaint.measureText(angle);

              int angleTextX = (int)(px-angleTextWidth/2);
              int angleTextY = py-radius+textHeight;
              canvas.drawText(angle, angleTextX, angleTextY, textPaint);
            }
            canvas.restore();

            canvas.rotate(15, px, py);
          }
          canvas.restore();
        }
```

7. To view the compass, modify the main.xml layout resource and replace the `TextView` reference with your new `CompassView`. This process is explained in more detail in the next section.

```
<?xml version="1.0" encoding="utf-8"?>
<LinearLayout xmlns:android="http://schemas.android.com/apk/res/android"
  android:orientation="vertical"
  android:layout_width="fill_parent"
  android:layout_height="fill_parent">
  <com.paad.compass.CompassView
    android:id="@+id/compassView"
    android:layout_width="fill_parent"
    android:layout_height="fill_parent"
  />
</LinearLayout>
```

8. Run the Activity, and you should see the `CompassView` displayed. See Chapter 10 to learn how to bind the `CompassView` to the device's compass.

Using Custom Controls

Having created you own custom Views, you can use them within code and layouts as you would any other Android control. The snippet below shows how to add the `CompassView` created in the above example to an Activity, by overriding the `onCreate` method:

```
@Override
public void onCreate(Bundle icicle) {
```

```
    super.onCreate(icicle);
    CompassView cv = new CompassView(this);
    setContentView(cv);
    cv.setBearing(45);
  }
```

To use the same control within a layout resource, specify the fully qualified class name when you create a new node in the layout definition, as shown in the following XML snippet:

```
<com.paad.compass.CompassView
  android:id="@+id/compassView"
  android:layout_width="fill_parent"
  android:layout_height="fill_parent"
/>
```

You can inflate the layout and get a reference to the `CompassView` as normal, using the following code:

```
@Override
public void onCreate(Bundle icicle) {
  super.onCreate(icicle);
  setContentView(R.layout.main);
  CompassView cv = (CompassView)this.findViewById(R.id.compassView);
  cv.setBearing(45);
}
```

Creating and Using Menus

Menus offer a way to expose application functions without sacrificing valuable screen space. Each Activity can specify its own Activity menu that's displayed when the device's *menu* button is pressed.

Android also supports context menus that can be assigned to any View within an Activity. A View's context menu is triggered when a user holds the middle D-pad button, depresses the trackball, or long-presses the touch screen for around 3 seconds when the View has focus.

Activity and context menus support submenus, checkboxes, radio buttons, shortcut keys, and icons.

Introducing the Android Menu System

If you've ever tried to navigate a mobile phone menu system using a stylus or trackball, you'll know that traditional menu systems are awkward to use on mobile devices.

To improve the usability of application menus, Android features a three-stage menu system optimized for small screens:

- **The Icon Menu** This compact menu (shown in Figure 4-4) appears along the bottom of the screen when the Menu button is pressed. It displays the icons and text for up to six Menu Items (or submenus).

THE UNIVERSITY OF MICHIGAN LIBRARIES

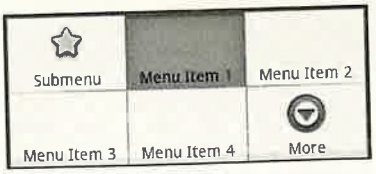


Figure 4-4

This icon menu does *not* display checkboxes, radio buttons, or the shortcut keys for Menu Items, so it's generally good practice not to assign checkboxes or radio buttons to icon menu items, as they will not be available.

If more than six Menu Items have been defined, a *More* item is included that, when selected, displays the expanded menu. Pressing the Back button closes the icon menu.

- **The Expanded Menu** The expanded menu is triggered when a user selects the **More** Menu Item from the icon menu. The expanded menu (shown in Figure 4-5) displays a scrollable list of *only* the Menu Items that weren't visible in the icon menu. This menu displays full text, shortcut keys, and checkboxes/radio buttons as appropriate.

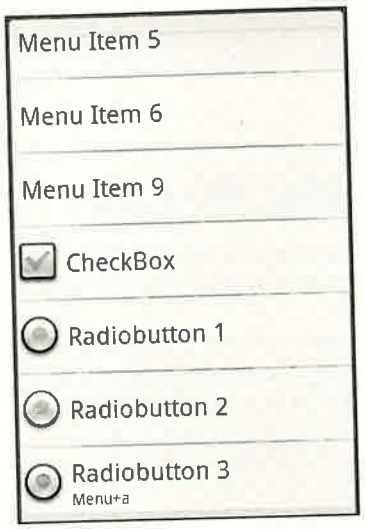


Figure 4-5

It does not, however, display icons. As a result, you should avoid assigning icons to Menu Items that are likely to appear only in the expanded menu.

Pressing Back from the expanded menu returns to the icon menu.

You cannot force Android to display the expanded menu instead of the icon menu. As a result, special care must be taken with Menu Items that feature checkboxes or radio buttons to ensure that they are either available only in the extended menu, or that their state information is also indicated using an icon or change in text.

- **Submenus** The traditional "expanding hierarchical tree" can be awkward to navigate using a mouse, so it's no surprise that this metaphor is particularly ill-suited for use on mobile devices. The Android alternative is to display each submenu in a floating window. For example, when a user selects a submenu such as the creatively labeled *Submenu* from Figure 4-5, its items are displayed in a floating menu Dialog box, as shown in Figure 4-6.



Figure 4-6

Note that the name of the submenu is shown in the header bar and that each Menu Item is displayed with its full text, checkbox (if any), and shortcut key. Since Android does not support nested submenus, you can't add a submenu to a submenu (trying will result in an exception).

As with the extended menu, icons are not displayed in the submenu items, so it's good practice to avoid assigning icons to submenu items.

Pressing the Back button closes the floating window without navigating back to the extended or icon menus.

Defining an Activity Menu

To define a menu for an Activity, override its `onCreateOptionsMenu` method. This method is triggered the first time an Activity's menu is displayed.

The `onCreateOptionsMenu` receives a `Menu` object as a parameter. You can store a reference to, and continue to use, the Menu reference elsewhere in your code until the next time that `onCreateOptionsMenu` is called.

You should always call through to the base implementation as it automatically includes additional system menu options where appropriate.

Use the `add` method on the `Menu` object to populate your menu. For each Menu Item, you must specify:

- A group value to separate Menu Items for batch processing and ordering

THE UNIVERSITY OF MICHIGAN LIBRARIES

- A unique identifier for each Menu Item. For efficiency reasons, Menu Item selections are generally handled by the `onOptionsItemSelected` event handler, so this unique identifier is important to determine which Menu Item was pressed. It is convention to declare each menu ID as a private static variable within the `Activity` class. You can use the `Menu.FIRST` static constant and simply increment that value for each subsequent item.
- An order value that defines the order in which the Menu Items are displayed
- The menu text, either as a character string or as a string resource

When you have finished populating the menu, return True to allow Android to display the menu.

The following skeleton code shows how to add a single item to an Activity menu:

```
static final private int MENU_ITEM = Menu.FIRST;

@Override
public boolean onCreateOptionsMenu(Menu menu) {
  super.onCreateOptionsMenu(menu);

  // Group ID
  int groupId = 0;
  // Unique menu item identifier. Used for event handling.
  int menuItemId = MENU_ITEM;
  // The order position of the item
  int menuItemOrder = Menu.NONE;
  // Text to be displayed for this menu item.
  int menuItemText = R.string.menu_item;

  // Create the menu item and keep a reference to it.
  MenuItem menuItem = menu.add(groupId, menuItemId,
                               menuItemOrder, menuItemText);

  return true;
}
```

Like the `Menu` object, each Menu Item reference returned by a call to `add` is valid until the next call to `onCreateOptionsMenu`. Rather than maintaining a reference to each item, you can find a particular Menu Item by passing its ID into the `Menu.findItem` method.

Menu Item Options

Android supports most of the traditional Menu Item options you're probably familiar with, including icons, shortcuts, checkboxes, and radio buttons, as described below:

- **Checkboxes and Radio Buttons** Checkboxes and radio buttons on Menu Items are visible in expanded menus and submenus, as shown in Figure 4-6. To set a Menu Item as a checkbox, use the `setCheckable` method. The state of that checkbox is controlled using `setChecked`.

  A *radio button group* is a group of items displaying circular buttons, where only one item can be selected at any given time. Checking one of these items will automatically unselect any other item in the same group, that is currently checked. To create a radio button group, assign the

same group identifier to each item, then call `Menu.setGroupCheckable`, passing in that group identifier and setting the exclusive parameter to True.

Checkboxes are not visible in the icon menu, so Menu Items that feature checkboxes should be reserved for submenus and items that appear only in the expanded menu. The following code snippet shows how to add a checkbox and a group of three radio buttons:

```
// Create a new check box item.
menu.add(0, CHECKBOX_ITEM, Menu.NONE, "CheckBox").setCheckable(true);

// Create a radio button group.
menu.add(RB_GROUP, RADIOBUTTON_1, Menu.NONE, "Radiobutton 1");
menu.add(RB_GROUP, RADIOBUTTON_2, Menu.NONE, "Radiobutton 2");
menu.add(RB_GROUP, RADIOBUTTON_3, Menu.NONE,
         "Radiobutton 3").setChecked(true);
menu.setGroupCheckable(RB_GROUP, true, true);
```

- **Shortcut Keys** You can specify a keypad shortcut for a Menu Item using the `setShortcut` method. Each call to `setShortcut` requires two shortcut keys, one for use with the numeric keypad and a second to support a full keyboard. Neither key is case-sensitive.

  The code below shows how to set a shortcut for both modes:

  ```
  // Add a shortcut to this menu item, '0' if using the numeric keypad
  // or 'b' if using the full keyboard.
  menuItem.setShortcut('0', 'b');
  ```

- **Condensed Titles** The icon menu does not display shortcuts or checkboxes, so it's often necessary to modify its display text to indicate its state. The following code shows how to set the icon menu–specific text for a Menu Item:

  ```
  menuItem.setTitleCondensed("Short Title");
  ```

- **Icons** Icon is a drawable resource identifier for an icon to be used in the Menu Item. Icons are only displayed in the icon menu; they are not visible in the extended menu or submenus. The following snippet shows how to apply an icon to a Menu Item:

  ```
  menuItem.setIcon(R.drawable.menu_item_icon);
  ```

- **Menu Item Click Listener** An event handler that will execute when the Menu Item is selected. For efficiency reasons, this is discouraged; instead, Menu Item selections should be handled by the `onOptionsItemSelected` handler as shown later in this section. To apply a click listener to a Menu Item, use the pattern shown in the following code snippet:

  ```
  menuItem.setOnMenuItemClickListener(new OnMenuItemClickListener() {

    public boolean onMenuItemClick(MenuItem _menuItem) {
      [ ... execute click handling, return true if handled ... ]
      return true;
    }

  });
  ```


THE UNIVERSITY OF MICHIGAN LIBRARIES

- **Intents** An Intent assigned to a Menu Item is triggered when clicking a Menu Item isn't handled by either a `MenuItemClickListener` or the Activity's `onOptionsItemSelected` handler. When triggered, Android will execute `startActivity`, passing in the specified Intent. The following code snippet shows how to specify an Intent for a Menu Item:

```
menuItem.setIntent(new Intent(this, MyOtherActivity.class));
```

Dynamically Updating Menu Items

By overriding your activity's `onPrepareOptionsMenu` method, you can modify your menu based on the application state each time it's displayed. This lets you dynamically disable/enable each item, set visibility, and modify text at runtime.

To modify Menu Items dynamically, you can either keep a reference to them when they're created in the `onCreateOptionsMenu` method, or you can use `menu.findItem` as shown in the following skeleton code, where `onPrepareOptionsMenu` is overridden:

```
@Override
public boolean onPrepareOptionsMenu(Menu menu) {
  super.onPrepareOptionsMenu(menu);

  MenuItem menuItem = menu.findItem(MENU_ITEM);

  [ ... modify menu items ... ]

  return true;
}
```

Handling Menu Selections

Android handles all of an Activity's Menu Item selections using a single event handler, the `onOptionsItemSelected` method. The Menu Item selected is passed in to this method as the `MenuItem` parameter.

To react to the menu selection, compare the `item.getItemId` value to the Menu Item identifiers you used when populating the menu, and react accordingly, as shown in the following code:

```
public boolean onOptionsItemSelected(MenuItem item) {
  super.onOptionsItemSelected(item);

  // Find which menu item has been selected
  switch (item.getItemId()) {

    // Check for each known menu item
    case (MENU_ITEM):
      [ ... Perform menu handler actions ... ]
      return true;
  }

  // Return false if you have not handled the menu item.
  return false;
}
```

Submenus and Context Menus

Context menus are displayed using the same floating window as the submenus shown in Figure 4-5. While their appearance is the same, the two menu types are populated differently.

Creating Submenus

Submenus are displayed as regular Menu Items that, when selected, reveal more items. Traditionally, submenus are displayed using a hierarchical tree layout. Android uses a different approach to simplify menu navigation for small-screen devices. Rather than a tree structure, selecting a submenu presents a single floating window that displays all of its Menu Items.

You can add submenus using the `addSubMenu` method. It supports the same parameters as the `add` method used to add normal Menu Items, allowing you to specify a group, unique identifier, and text string for each submenu. You can also use the `setHeaderIcon` and `setIcon` methods to specify an icon to display in the submenu's header bar or the regular icon menu, respectively.

The Menu Items within a submenu support the same options as those assigned to the icon or extended menus. However, unlike traditional systems, Android does not support nested submenus.

The code snippet below shows an extract from an implementation of the `onCreateMenuOptions` code that adds a submenu to the main menu, sets the header icon, and then adds a submenu Menu Item:

```
SubMenu sub = menu.addSubMenu(0, 0, Menu.NONE, "Submenu");
sub.setHeaderIcon(R.drawable.icon);
sub.setIcon(R.drawable.icon);

MenuItem submenuItem = sub.add(0, 0, Menu.NONE, "Submenu Item");
```

Using Context Menus

Context Menus are contextualized by the currently focused View and are triggered by pressing the trackball, middle D-pad button, or the View for around 3 seconds.

You define and populate Context Menus similarly to the Activity menu. There are two options available for creating Context Menus for a particular View.

Creating Context Menus

The first option is to create a generic Context Menu for a `View` class by overriding a View's `onCreateContextMenu` handler as shown below:

```
@Override
public void onCreateContextMenu(ContextMenu menu) {
  super.onCreateContextMenu(menu);
  menu.add("ContextMenuItem1");
}
```

The Context Menu created here will be available from any Activity that includes this `View` class.

THE UNIVERSITY OF MICHIGAN LIBRARIES

The more common alternative is to create Activity-specific Context Menus by overriding the `onCreateContextMenu` method and registering the Views that should use it. Register Views using `registerForContextMenu` and passing them in, as shown in the following code:

```
@Override
public void onCreate(Bundle icicle) {
  super.onCreate(icicle);

  EditText view = new EditText(this);
  setContentView(view);

  registerForContextMenu(view);
}
```

Once a View has been registered, the `onCreateContextMenu` handler will be triggered whenever a Context Menu should be displayed for that View.

Override `onCreateContextMenu` and check which View has triggered the menu creation to populate the menu parameter with the appropriate contextual items, as shown in the following code snippet:

```
@Override
public void onCreateContextMenu(ContextMenu menu, View v,
                                ContextMenu.ContextMenuInfo menuInfo) {
  super.onCreateContextMenu(menu, v, menuInfo);

  menu.setHeaderTitle("Context Menu");
  menu.add(0, menu.FIRST, Menu.NONE,
           "Item 1").setIcon(R.drawable.menu_item);
  menu.add(0, menu.FIRST+1, Menu.NONE, "Item 2").setCheckable(true);
  menu.add(0, menu.FIRST+2, Menu.NONE, "Item 3").setShortcut('3', '3');
  SubMenu sub = menu.addSubMenu("Submenu");
  sub.add("Submenu Item");
}
```

As shown above, the `ContextMenu` class supports the same `add` method as the `Menu` class, so you can populate a Context Menu in the same way as Activity menus — including support for submenus — but icons will never be displayed. You can also specify the title and icon to display in the Context Menu's header bar.

Android supports late runtime population of Context Menus using Intent Filters. This mechanism lets you populate a Context Menu by specifying the kind of data presented by the current View, and asking other Android applications if they support any actions for it. The most common example of this behavior is the cut/copy/paste Menu Items available on EditText controls. This process is covered in detail in the next chapter.

Handling Context Menu Selections

Context Menu Item selections are handled similarly to the Activity menu. You can attach an Intent or Menu Item Click Listener directly to each Menu Item, or use the preferred technique of overriding the `onContextItemSelected` method on the Activity.

This event handler is triggered whenever a Context Menu Item is selected within the Activity. A skeleton implementation is shown below:

```
@Override
public boolean onContextItemSelected(MenuItem item) {
  super.onContextItemSelected(item);

  [ ... Handle menu item selection ... ]

  return false;
}
```

To-Do List Example Continued

In the following example, you'll be adding some simple menu functions to the To-Do List application you started in Chapter 2 and continued to improve previously in this chapter.

You will add the ability to remove items from Context and Activity Menus, and improve the use of screen space by displaying the text entry box only when adding a new item.

1. Start by importing the packages you need to support menu functionality into the `ToDoList` Activity class.

```
import android.view.Menu;
import android.view.MenuItem;
import android.view.ContextMenu;
import android.widget.AdapterView;
```

2. Then add private static final variables that define the unique IDs for each Menu Item.

```
static final private int ADD_NEW_TODO = Menu.FIRST;
static final private int REMOVE_TODO = Menu.FIRST + 1;
```

3. Now override the `onCreateOptionsMenu` method to add two new Menu Items, one to add and the other to remove the to-do item. Specify the appropriate text, and assign icon resources and shortcut keys for each item.

```
@Override
public boolean onCreateOptionsMenu(Menu menu) {
  super.onCreateOptionsMenu(menu);

  // Create and add new menu items.
  MenuItem itemAdd = menu.add(0, ADD_NEW_TODO, Menu.NONE,
                              R.string.add_new);
  MenuItem itemRem = menu.add(0, REMOVE_TODO, Menu.NONE,
                              R.string.remove);

  // Assign icons
  itemAdd.setIcon(R.drawable.add_new_item);
  itemRem.setIcon(R.drawable.remove_item);

  // Allocate shortcuts to each of them.
```


THE UNIVERSITY OF MICHIGAN LIBRARIES

```
    itemAdd.setShortcut('0', 'a');
    itemRem.setShortcut('1', 'r');

    return true;
  }
```

If you run the Activity, pressing the Menu button should appear as shown in Figure 4-7.

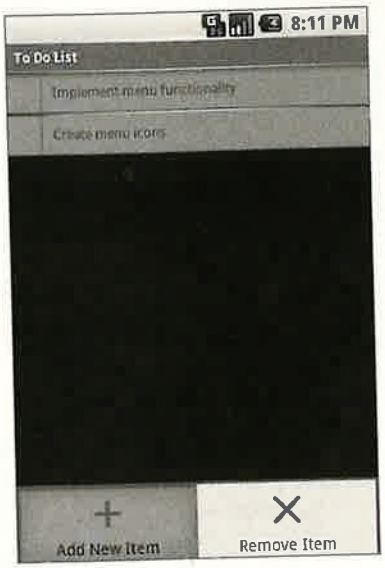


Figure 4-7

4. Having populated the Activity Menu, create a Context Menu. First, modify `onCreate` to register the ListView to receive a Context Menu. Then override `onCreateContextMenu` to populate the menu with a "remove" item.

```
@Override
  public void onCreate(Bundle icicle) {

    [ ... existing onCreate method ... ]

    registerForContextMenu(myListView);
  }

  @Override
  public void onCreateContextMenu(ContextMenu menu,
                                  View v,
                                  ContextMenu.ContextMenuInfo menuInfo) {
    super.onCreateContextMenu(menu, v, menuInfo);

    menu.setHeaderTitle("Selected To Do Item");
    menu.add(0, REMOVE_TODO, Menu.NONE, R.string.remove);
  }
```

5. Now modify the appearance of the menu based on the application context, by overriding the `onPrepareOptionsMenu` method. The menu should be customized to show "cancel" rather than "delete" if you are currently adding a new Menu Item.

```
private boolean addingNew = false;

@Override
public boolean onPrepareOptionsMenu(Menu menu) {
  super.onPrepareOptionsMenu(menu);

  int idx = myListView.getSelectedItemPosition();

  String removeTitle = getString(addingNew ?
                                 R.string.cancel : R.string.remove);

  MenuItem removeItem = menu.findItem(REMOVE_TODO);
  removeItem.setTitle(removeTitle);
  removeItem.setVisible(addingNew || idx > -1);

  return true;
}
```

6. For the code in Step 5 to work, you need to increase the scope of the `todoListItems` and `ListView` control beyond the `onCreate` method. Do the same thing for the `ArrayAdapter` and `EditText` to support the *add* and *remove* actions when they're implemented later.

```
private ArrayList<String> todoItems;
private ListView myListView;
private EditText myEditText;
private ArrayAdapter<String> aa;

@Override
public void onCreate(Bundle icicle) {
  super.onCreate(icicle);

  // Inflate your view
  setContentView(R.layout.main);

  // Get references to UI widgets
  myListView = (ListView)findViewById(R.id.myListView);
  myEditText = (EditText)findViewById(R.id.myEditText);

  todoItems = new ArrayList<String>();
  int resID = R.layout.todolist_item;
  aa = new ArrayAdapter<String>(this, resID, todoItems);
  myListView.setAdapter(aa);

  myEditText.setOnKeyListener(new OnKeyListener() {
    public boolean onKey(View v, int keyCode, KeyEvent event) {
      if (event.getAction() == KeyEvent.ACTION_DOWN)
        if (keyCode == KeyEvent.KEYCODE_DPAD_CENTER)
        {
          todoItems.add(0, myEditText.getText().toString());
          myEditText.setText("");
```


THE UNIVERSITY OF MICHIGAN LIBRARIES

```
            aa.notifyDataSetChanged();
            return true;
          }
        return false;
      }
    });

    registerForContextMenu(myListView);
  }
```

7. Next you need to handle Menu Item clicks. Override the `onOptionsItemSelected` and `onContextItemSelected` methods to execute stubs that handle the new Menu Items.

   7.1. Start by overriding `onOptionsItemSelected` to handle the Activity menu selections. For the *remove* menu option, you can use the `getSelectedItemPosition` method on the List View to find the currently highlighted item.

```
@Override
public boolean onOptionsItemSelected(MenuItem item) {
  super.onOptionsItemSelected(item);

  int index = myListView.getSelectedItemPosition();

  switch (item.getItemId()) {
    case (REMOVE_TODO): {
      if (addingNew) {
        cancelAdd();
      }
      else {
        removeItem(index);
      }
      return true;
    }
    case (ADD_NEW_TODO): {
      addNewItem();
      return true;
    }
  }

  return false;
}
```

   7.2. Next override `onContextItemSelected` to handle Context Menu Item selections. Note that you are using the `AdapterView` specific implementation of `ContextMenuInfo`. This includes a reference to the View that triggered the Context Menu and the position of the data it's displaying in the underlying Adapter.

   Use the latter to find the index of the item to remove.

```
@Override
public boolean onContextItemSelected(MenuItem item) {
  super.onContextItemSelected(item);
  switch (item.getItemId()) {
    case (REMOVE_TODO): {
      AdapterView.AdapterContextMenuInfo menuInfo;
      menuInfo =(AdapterView.AdapterContextMenuInfo)item.getMenuInfo();
```

```
        int index = menuInfo.position;

        removeItem(index);
        return true;
      }
    }
    return false;
}
```

7.3. Create the stubs called in the Menu Item selection handlers you created above.

```
private void cancelAdd() {
}

private void addNewItem() {
}

private void removeItem(int _index) {
}
```

8. Now implement each of the stubs to provide the new functionality.

```
private void cancelAdd() {
  addingNew = false;
  myEditText.setVisibility(View.GONE);
}

private void addNewItem() {
  addingNew = true;
  myEditText.setVisibility(View.VISIBLE);
  myEditText.requestFocus();
}

private void removeItem(int _index) {
  todoItems.remove(_index);
  aa.notifyDataSetChanged();
}
```

9. Next you need to hide the text entry box after you've added a new item. In the `onCreate` method, modify the `onKeyListener` to call the `cancelAdd` function after adding a new item.

```
myEditText.setOnKeyListener(new OnKeyListener() {
  public boolean onKey(View v, int keyCode, KeyEvent event) {
    if (event.getAction() == KeyEvent.ACTION_DOWN)
      if (keyCode == KeyEvent.KEYCODE_DPAD_CENTER)
      {
        todoItems.add(0, myEditText.getText().toString());
        myEditText.setText("");
        aa.notifyDataSetChanged();
        cancelAdd();
        return true;
      }
    return false;
  }
});
```


THE UNIVERSITY OF MICHIGAN LIBRARIES

10. Finally, to ensure a consistent UI, modify the main.xml layout to hide the text entry box until the user chooses to add a new item.

```
<EditText
  android:id="@+id/myEditText"
  android:layout_width="fill_parent"
  android:layout_height="wrap_content"
  android:text=""
  android:visibility="gone"
/>
```

Running the application should now let you trigger the Activity menu to add or remove items from the list, and a Context Menu on each item should offer the option of removing it.

Summary

You now know the basics of creating intuitive User Interfaces for Android applications. You learned about Views and layouts and were introduced to the Android menu system.

Activity screens are created by positioning Views using Layout Managers that can be created in code or as resource files. You learned how to extend, group, and create new View-based controls to provide customized appearance and behavior for your applications.

In this chapter, you:

- Were introduced to some of the controls and widgets available as part of the Android SDK.
- Learned how to use your custom controls within Activities.
- Discovered how to create and use Activity Menus and Context Menus.
- Extended the To-Do List Example to support custom Views and menu-based functions.
- Created a new CompassView control from scratch.

Having covered the fundamentals of Android UI design, the next chapter focuses on binding application components using Intents, Broadcast Receivers, and Adapters. You will learn how to start new Activities and broadcast and consume requests for actions. Chapter 5 also introduces Internet connectivity and looks at the `Dialog` class.

5

Intents, Broadcast Receivers, Adapters, and the Internet

At first glance, the subjects of this chapter may appear to have little in common; in practice, they represent the glue that binds applications and their components.

Mobile applications on most platforms run in their own sandboxes. They're isolated from each other and have strict limits on their interaction with the system hardware and native components. Android applications are also sandboxed, but they can use Intents, Broadcast Receivers, Adapters, Content Providers, and the Internet to extend beyond those boundaries.

In this chapter, you'll look at Intents and learn how to use them to start Activities, both explicitly and using late runtime binding. Using implicit Intents, you'll learn how to request that an action be performed on a piece of data, letting Android determine which application component can service that request.

Broadcast Intents are used to announce application events system-wide. You'll learn how to transmit these broadcasts and consume them using Broadcast Receivers.

You'll examine Adapters and learn how to use them to bind your presentation layer to data sources, and you'll examine the Dialog-box mechanisms available.

Having looked at the mechanisms for transmitting and consuming local data, you'll be introduced to Android's Internet connectivity model and some of the Java techniques for parsing Internet data feeds.

THE UNIVERSITY OF MICHIGAN LIBRARIES

An earthquake-monitoring example will then demonstrate how to tie all these features together. The earthquake monitor will form the basis of an ongoing example that you'll improve and extend in later chapters.

Introducing Intents

Intents are used as a message-passing mechanism that lets you declare your intention that an action be performed, usually with (or on) a particular piece of data.

You can use Intents to support interaction between any of the application components available on an Android device, no matter which application they're part of. This turns a collection of independent components into a single, interconnected system.

One of the most common uses for Intents is to start new Activities, either *explicitly* (by specifying the class to load) or *implicitly* (by requesting an action be performed on a piece of data).

Intents can also be used to broadcast messages across the system. Any application can register a Broadcast Receiver to listen for, and react to, these broadcast Intents. This lets you create event-driven applications based on internal, system, or third-party application events.

Android uses broadcast Intents to announce system events, like changes in Internet connection status or battery charge levels. The native Android applications, such as the phone dialer and SMS manager, simply register components that listen for specific broadcast Intents — such as "incoming phone call" or "SMS message received" — and react accordingly.

Using Intents to propagate actions — even within the same application — is a fundamental Android design principle. It encourages the decoupling of components, to allow the seamless replacement of application elements. It also provides the basis of a simple model for extending functionality.

Using Intents to Launch Activities

The most common use of Intents is to bind your application components. Intents are used to start, stop, and transition between the Activities within an application.

> *The instructions given in this section refer to starting new Activities, but the same rules generally apply to Services as well. Details on starting (and creating) Services are available in Chapter 8.*

To open a different application screen (Activity) in your application, call `startActivity`, passing in an Intent, as shown in the snippet below.

```
startActivity(myIntent);
```

The `Intent` can either explicitly specify the class to open, or include an action that the target should perform. In the latter case, the run time will choose the Activity to open, using a process known as "Intent resolution."

The `startActivity` method finds, and starts, the single Activity that best matches your Intent.

When using `startActivity`, your application won't receive any notification when the newly launched Activity finishes. To track feedback from the opened form, use the `startActivityForResult` method described in more detail below.

Explicitly Starting New Activities

You learned in Chapter 2 that applications consist of several interrelated screens — Activities — that must be included in the application manifest. To connect them, you may want to explicitly specify which Activity to open.

To explicitly select an Activity class to start, create a new `Intent` specifying the current application context and the class of the Activity to launch. Pass this Intent in to `startActivity`, as shown in the following code snippet:

```
Intent intent = new Intent(MyActivity.this, MyOtherActivity.class);
startActivity(intent);
```

After calling `startActivity`, the new Activity (in this example, `MyOtherActivity`) will be created and become visible and active, moving to the top of the Activity stack.

Calling `finish` programmatically on the new Activity will close it and remove it from the stack. Alternatively, users can navigate to the previous Activity using the device's Back button.

Implicit Intents and Late Runtime Binding

Implicit Intents are a mechanism that lets anonymous application components service action requests.

When constructing a new implicit Intent to use with `startActivity`, you nominate an action to perform and, optionally, supply the data on which to perform that action.

When you use this new *implicit* Intent to start an Activity, Android will — at run time — resolve it into the class best suited to performing the action on the type of data specified. This means that you can create projects that use functionality from other applications, without knowing exactly which application you're borrowing functionality from ahead of time.

For example, if you want to let users make calls from an application, rather than implementing a new dialer you could use an implicit Intent that requests that the action ("dial a number") be performed on a phone number (represented as a URI), as shown in the code snippet below:

```
if (somethingWeird && itDontLookGood) {
  Intent intent = new Intent(Intent.ACTION_DIAL,
                             Uri.parse("tel:555-2368"));
  startActivity(intent);
}
```

Android resolves this Intent and starts an Activity that provides the dial action on a telephone number — in this case, the dialler Activity.

Various native applications provide components to handle actions performed on specific data. Third-party applications, including your own, can be registered to support new actions or to provide an alternative provider of native actions. You'll be introduced to some of the native actions later in this chapter.

THE UNIVERSITY OF MICHIGAN LIBRARIES

Introducing `Linkify`

`Linkify` is a helper class that automagically creates hyperlinks within TextView (and TextView-derived) classes through RegEx pattern matching.

Text that matches a specified RegEx pattern will be converted into a clickable hyperlink that implicitly fires `startActivity(new Intent(Intent.ACTION_VIEW, uri))` using the matched text as the target URI.

You can specify any string pattern you want to turn into links; for convenience, the `Linkify` class provides presets for common content types (like phone numbers and e-mail/web addresses).

The Native Link Types

The static `Linkify.addLinks` method accepts the View to linkify, and a bitmask of one or more of the default content types supported and supplied by the `Linkify` class: `WEB_URLS`, `EMAIL_ADDRESSES`, `PHONE_NUMBERS`, and `ALL`.

The following code snippet shows how to linkify a TextView to display web and e-mail addresses as hyperlinks. When clicked, they will open the browser or e-mail application, respectively.

```
TextView textView = (TextView)findViewById(R.id.myTextView);
Linkify.addLinks(textView, Linkify.WEB_URLS|Linkify.EMAIL_ADDRESSES);
```

You can linkify Views from within a layout resource using the `android:autoLink` attribute. It supports one or more (separated by |) of the following self-describing values: `none`, `web`, `email`, `phone`, or `all`.

The following XML snippet shows how to add hyperlinks for phone numbers and e-mail addresses:

```
<TextView
  android:layout_width="fill_parent"
  android:layout_height="fill_parent"
  android:text="@string/linkify_me"
  android:autoLink="phone|email"
/>
```

Creating Custom Link Strings

To define your own linkify strings, you create a new RegEx pattern to match the text you want to display as hyperlinks.

As with the native types, you linkify the target view by calling `Linkify.addLinks`, but this time pass in the new RegEx pattern. You can also pass in a prefix that will be prepended to the target URI when a link is clicked.

The following example shows a View being linkified to support earthquake data provided by an Android Content Provider (that you will create in the next Chapter). Rather than include the entire schema, the linkify pattern matches any text that starts with "quake" and is followed by a number. The content schema is then prepended to the URI before the Intent is fired.

```
int flags = Pattern.CASE_INSENSITIVE;
Pattern p = Pattern.compile("\\bquake[0-9]*\\b", flags);
```

```
Linkify.addLinks(myTextView, p,
                 "content://com.paad.earthquake/earthquakes/");
```

Linkify also supports `TransformFilter` and `MatchFilter` interfaces. They offer additional control over the target URI structure and the definition of matching strings, and are used as shown in the skeleton code below:

```
Linkify.addLinks(myTextView, pattern, prefixWith,
                 new MyMatchFilter(), new MyTransformFilter());
```

Using the Match Filter

Implement the `acceptMatch` method in your `MatchFilter` to add additional conditions to RegEx pattern matches. When a potential match is found, `acceptMatch` is triggered, with the match start and end index (along with the full text being searched) passed in as parameters.

The following code shows a `MatchFilter` implementation that cancels any match that is immediately preceded by an exclamation mark.

```
class MyMatchFilter implements MatchFilter {
  public boolean acceptMatch(CharSequence s, int start, int end) {
    return (start == 0 || s.charAt(start-1) != '!');
  }
}
```

Using the Transform Filter

The Transform Filter gives you more freedom to format your text strings by letting you modify the implicit URI generated by the link text. Decoupling the link text from the target URI gives you more freedom in how you display data strings to your users.

To use the Transform Filter, implement the `transformUrl` method in your Transform Filter. When Linkify finds a successful match, it calls `transformUrl`, passing in the RegEx pattern used and the default URI string it creates. You can modify the matched string, and return the URI as a target suitable to be "viewed" by another Android application.

The following `TransformFilter` implementation transforms the matched text into a lowercase URI:

```
class MyTransformFilter implements TransformFilter {
  public String transformUrl(Matcher match, String url) {
    return url.toLowerCase();
  }
}
```

Returning Results from Activities

An Activity started using `startActivity` is independent of its parent and will not provide any feedback when it closes.

Alternatively, you can start an Activity as a sub-Activity that's inherently connected to its parent. Sub-Activities trigger an event handler within their parent Activity when they close. Sub-Activities are perfect for situations in which one Activity is providing data input (such as a user selecting an item from a list) for another.

THE UNIVERSITY OF MICHIGAN LIBRARIES

Sub-Activities are created the same way as normal Activities and must also be registered in the application manifest. Any manifest-registered Activity can be opened as a sub-Activity.

Launching Sub-Activities

The `startActivityForResult` method works much like `startActivity` but with one important difference. As well as the Intent used to determine which Activity to launch, you also pass in a *request code*. This value will be used later to uniquely identify the sub-Activity that has returned a result.

The skeleton code for launching a sub-Activity is shown below:

```
private static final int SHOW_SUBACTIVITY = 1;

Intent intent = new Intent(this, MyOtherActivity.class);
startActivityForResult(intent, SHOW_SUBACTIVITY);
```

As with regular Activities, sub-Activities can be started implicitly or explicitly. The following skeleton code uses an implicit Intent to launch a new sub-Activity to pick a contact:

```
private static final int PICK_CONTACT_SUBACTIVITY = 2;

Uri uri = Uri.parse("content://contacts/people");
Intent intent = new Intent(Intent.ACTION_PICK, uri);
startActivityForResult(intent, PICK_CONTACT_SUBACTIVITY);
```

Returning Results

When your sub-Activity is ready to close, call `setResult` before `finish` to return a result to the calling Activity.

The `setResult` method takes two parameters: the result code and result payload represented as an Intent.

The result code is the "result" of running the sub-Activity — generally either `Activity.RESULT_OK` or `Activity.RESULT_CANCELED`. In some circumstances, you'll want to use your own response codes to handle application-specific choices; `setResult` supports any integer value.

The Intent returned as a result can include a URI to a piece of content (such as the contact, phone number, or media file) and a collection of Extras used to return additional information.

This next code snippet is taken from a sub-Activity's `onCreate` method and shows how an OK button and a Cancel button might return different results to the calling Activity:

```
Button okButton = (Button) findViewById(R.id.ok_button);
  okButton.setOnClickListener(new View.OnClickListener() {

  public void onClick(View view) {
    Uri data = Uri.parse("content://horses/" + selected_horse_id);

    Intent result = new Intent(null, data);
    result.putExtra(IS_INPUT_CORRECT, inputCorrect);
    result.putExtra(SELECTED_PISTOL, selectedPistol);
```

```
        setResult(RESULT_OK, result);

        finish();
      }
    });

    Button cancelButton = (Button) findViewById(R.id.cancel_button);
    cancelButton.setOnClickListener(new View.OnClickListener() {

      public void onClick(View view) {
        setResult(RESULT_CANCELED, null);

        finish();
      }
    });
```

Handling Sub-Activity Results

When a sub-Activity closes, its parent Activity's `onActivityResult` event handler is fired.

Override this method to handle the results from the sub-Activities. The `onActivityResult` handler receives several parameters:

- **The Request Code** The request code that was used to launch the returning sub-Activity
- **A Result Code** The result code set by the sub-Activity to indicate its result. It can be any integer value, but typically will be either `Activity.RESULT_OK` or `Activity.RESULT_CANCELLED`.

  If the sub-Activity closes abnormally or doesn't specify a result code before it closes, the result code is `Activity.RESULT_CANCELED`.

- **Data** An Intent used to bundle any returned data. Depending on the purpose of the sub-Activity, it will typically include a URI that represents the particular piece of data selected from a list. Alternatively, or additionally, the sub-Activity can return extra information as primitive values using the "extras" mechanism.

The skeleton code for implementing the `onActivityResult` event handler within an Activity is shown below:

```
private static final int SHOW_SUB_ACTIVITY_ONE = 1;
private static final int SHOW_SUB_ACTIVITY_TWO = 2;

@Override
public void onActivityResult(int requestCode,
                             int resultCode,
                             Intent data) {

  super.onActivityResult(requestCode, resultCode, data);

  switch(requestCode) {
    case (SHOW_SUB_ACTIVITY_ONE) : {
      if (resultCode == Activity.RESULT_OK) {
        Uri horse = data.getData();
```


THE UNIVERSITY OF MICHIGAN LIBRARIES

```
        boolean inputCorrect = data.getBooleanExtra(IS_INPUT_CORRECT,
                                                    false);
        String selectedPistol = data.getStringExtra(SELECTED_PISTOL);
      }
      break;
    }
    case (SHOW_SUB_ACTIVITY_TWO) : {
      if (resultCode == Activity.RESULT_OK) {
        // TODO: Handle OK click.
      }
      break;
    }
  }
}
```

Native Android Actions

Native Android applications also use Intents to launch Activities and sub-Activities.

The following noncomprehensive list shows some of the native actions available as static string constants in the `Intent` class. You can use these actions when creating implicit Intents to start Activities and sub-Activities within your own applications.

In the next section you will be introduced to Intent Filters, and you'll learn how to register your own Activities as handlers for these actions.

- `ACTION_ANSWER` Opens an Activity that handles incoming calls. Currently this is handled by the native phone dialer.
- `ACTION_CALL` Brings up a phone dialer and immediately initiates a call using the number supplied in the data URI. Generally, it's considered better form to use the `Dial_Action` if possible.
- `ACTION_DELETE` Starts an Activity that lets you delete the entry currently stored at the data URI location.
- `ACTION_DIAL` Brings up a dialer application with the number to dial prepopulated from the data URI. By default, this is handled by the native Android phone dialer. The dialer can normalize most number schemas; for example, `tel:555-1234` and `tel:(212) 555 1212` are both valid numbers.
- `ACTION_EDIT` Requests an Activity that can edit the data at the URI.
- `ACTION_INSERT` Opens an Activity capable of inserting new items into the cursor specified in the data field. When called as a sub-Activity, it should return a URI to the newly inserted item.
- `ACTION_PICK` Launches a sub-Activity that lets you pick an item from the URI data. When closed, it should return a URI to the item that was picked. The Activity launched depends on the data being picked; for example, passing `content://contacts/people` will invoke the native contacts list.
- `ACTION_SEARCH` Launches the UI for performing a search. Supply the search term as a string in the Intent's extras using the `SearchManager.QUERY` key.
- `ACTION_SENDTO` Launches an Activity to send a message to the contact specified by the data URI.

- `ACTION_SEND` Launches an Activity that sends the specified data (the recipient needs to be selected by the resolved Activity). Use `setType` to set the Intent's type as the transmitted data's mime type.

  The data itself should be stored as an extra using the key `EXTRA_TEXT` or `EXTRA_STREAM` depending on the type. In the case of e-mail, the native Android applications will also accept extras using the `EXTRA_EMAIL`, `EXTRA_CC`, `EXTRA_BCC`, and `EXTRA_SUBJECT` keys.

- `ACTION_VIEW` The most common generic action. View asks that the data supplied in the Intent's URI be viewed in the most reasonable manner. Different applications will handle view requests depending on the URI schema of the data supplied. Natively, `http:` addresses will open in the browser, `tel:` addresses will open the dialer to call the number, `geo:` addresses are displayed in the Google Maps application, and contact content will be displayed in the Contact Manager.

- `ACTION_WEB_SEARCH` Opens an activity that performs a Web search based on the text supplied in the data URI.

As well as these Activity actions, Android includes a large number of Broadcast actions that are used to create Intents that the system broadcasts to notify applications of events. These Broadcast actions are described later in this chapter.

Using Intent Filters to Service Implicit Intents

If an Intent is a request for an action to be performed on a set of data, how does Android know which application (and component) to use to service the request? Intent Filters are used to register Activities, Services, and Broadcast Receivers as being capable of performing an action on a particular kind of data.

Using Intent Filters, application components tell Android that they can service action requests from others, including components in the same, native, or third-party applications.

To register an application component as an Intent handler, use the `intent-filter` tag within the component's manifest node.

Using the following tags (and associated attributes) within the Intent Filter node, you can specify a component's supported actions, categories, and data:

- `action` Use the `android:name` attribute to specify the name of the action being serviced. Actions should be unique strings, so best practice is to use a naming system based on the Java package naming conventions.
- `category` Use the `android:category` attribute to specify under which circumstances the action should be serviced. Each Intent Filter tag can include multiple category tags. You can specify your own categories or use the standard values provided by Android and listed below:
  - `ALTERNATIVE` As you'll see later in this chapter, one of the uses of Intent Filters is to help populate context menus with actions. The `alternative` category specifies that this action should be available as an alternative to the default action performed on an item of this data type. For example, where the default action for a contact is to view it, the alternatives could be to edit or delete it.

THE UNIVERSITY OF MICHIGAN LIBRARIES

- SELECTED_ALTERNATIVE Similar to the alternative category, but where Alternative will always resolve to a single action using the Intent resolution described below, SELECTED_ALTERNATIVE is used when a list of possibilities is required.
- BROWSABLE Specifies an action available from within the browser. When an Intent is fired from within the browser, it will always specify the browsable category.
- DEFAULT Set this to make a component the default action for the data values defined by the Intent Filter. This is also necessary for Activities that are launched using an explicit Intent.
- GADGET By setting the gadget category, you specify that this Activity can run embedded inside another Activity.
- HOME The home Activity is the first Activity displayed when the device starts (the launch screen). By setting an Intent Filter category as home without specifying an action, you are presenting it as an alternative to the native home screen.
- LAUNCHER Using this category makes an Activity appear in the application launcher.

- data The data tag lets you specify matches for data your component can act on; you can include several schemata if your component is capable of handling more than one. You can use any combination of the following attributes to specify the data that your component supports:
  - android:host Specifies a valid host name (e.g., com.google).
  - android:mimetype Lets you specify the type of data your component is capable of handling. For example, <type android:value="vnd.android.cursor.dir/*"/> would match any Android cursor.
  - android:path Valid "path" values for the URI (e.g., /transport/boats/)
  - android:port Valid ports for the specified host
  - android:scheme Requires a particular scheme (e.g., content or http).

The following code snippet shows how to configure an Intent Filter for an Activity that can perform the SHOW_DAMAGE action as either a primary or alternative action. (You'll create earthquake content in the next chapter.)

```
<activity android:name=".EarthquakeDamageViewer"
          android:label="View Damage">
  <intent-filter>
    <action
      android:name="com.paad.earthquake.intent.action.SHOW_DAMAGE">
    </action>
    <category android:name="android.intent.category.DEFAULT"/>
    <category
      android:name="android.intent.category.ALTERNATIVE_SELECTED"
    />
    <data android:mimeType="vnd.earthquake.cursor.item/*"/>
  </intent-filter>
</activity>
```

How Android Resolves Intent Filters

The anonymous nature of runtime binding makes it important to understand how Android resolves an implicit Intent into a particular application component.

As you saw previously, when using `startActivity`, the implicit Intent resolves to a single Activity. If there are multiple Activities capable of performing the given action on the specified data, the "best" of those Activities will be launched.

The process of deciding which Activity to start is called *Intent resolution*. The aim of Intent resolution is to find the best Intent Filter match possible using the following process:

1. Android puts together a list of all the Intent Filters available from the installed packages.
2. Intent Filters that do not match the action or category associated with the Intent being resolved are removed from the list.
   - 2.1. Action matches are made if the Intent Filter either includes the specified action or has no action specified.

     An Intent Filter will only fail the action match check if it has one or more actions defined, where none of them match the action specified by the Intent.
   - 2.2. Category matching is stricter. Intent Filters must include *all* the categories defined in the resolving Intent. An Intent Filter with no categories specified only matches Intents with no categories.
3. Finally, each part of the Intent's data URI is compared to the Intent Filter's `data` tag. If Intent Filter defines the scheme, host/authority, path, or mime type, these values are compared to the Intent's URI. Any mismatches will remove the Intent Filter from the list.

   Specifying no data values in an Intent Filter will match with all Intent data values.
   - 3.1. The mime type is the data type of the data being matched. When matching data types, you can use wild cards to match subtypes (e.g., `earthquakes/*`). If the Intent Filter specifies a data type, it must match the Intent; specifying no data type resolves to all of them.
   - 3.2. The scheme is the "protocol" part of the URI — for example, `http:`, `mailto:`, or `tel:`.
   - 3.3. The host name or "data authority" is the section of the URI between the scheme and the path (e.g., `www.google.com`). For a host name to match, the Intent Filter's scheme must also pass.
   - 3.4. The data path is what comes after the authority (e.g., `/ig`). A path can only match if the scheme and host-name parts of the `data` tag also match.
4. If more than one component is resolved from this process, they are ordered in terms of priority, with an optional tag that can be added to the Intent Filter node. The highest ranking component is then returned.

Native Android application components are part of the Intent resolution process in exactly the same way as third-party applications. They do not have a higher priority and can be completely replaced with new Activities that declare Intent Filters that service the same action requests.

THE UNIVERSITY OF MICHIGAN LIBRARIES

Responding to Intent Filter Matches

When an application component is started through an implicit Intent, it needs to find the action it is to perform and the data upon which to perform it.

Call the `getIntent` method — usually from within the `onCreate` method — to extract the Intent used to launch a component, as shown below:

```
@Override
public void onCreate(Bundle icicle) {
  super.onCreate(icicle);
  setContentView(R.layout.main);

  Intent intent = getIntent();
}
```

Use the `getData` and `getAction` methods to find the data and action of the Intent. Use the type-safe `get<type>Extra` methods to extract additional information stored in its extras Bundle.

```
String action = intent.getAction();
Uri data = intent.getData();
```

Passing on Responsibility

You can use the `startNextMatchingActivity` method to pass responsibility for action handling to the next best matching application component, as shown in the snippet below:

```
Intent intent = getIntent();
if (isAfterMidnight)
  startNextMatchingActivity(intent);
```

This allows you to add additional conditions to your components that restrict their use beyond the ability of the Intent Filter–based Intent resolution process.

In some cases, your component may wish to perform some processing, or offer the user a choice, before passing the Intent on to the native handler.

Select a Contact Example

In this example, you'll create a new sub-Activity that services the `PICK_ACTION` for contact data. It displays each of the contacts in the contact database and lets the user select one, before closing and returning its URI to the calling Activity.

> ***It's worth noting that this*** *example is somewhat contrived. Android already supplies an Intent Filter* ***for picking a contact from*** *a list that can be invoked by using the* `content:/contacts/people/` ***URI in an implicit Intent.*** *The purpose of this exercise is to demonstrate the form, even if this particular implementation isn't overly useful.*

1. Create a new ContactPicker project that includes a `ContactPicker` Activity.

```
package com.paad.contactpicker;

import android.app.Activity;
```

```
import android.content.Intent;
import android.database.Cursor;
import android.net.Uri;
import android.os.Bundle;
import android.provider.Contacts.People;
import android.view.View;
import android.widget.AdapterView;
import android.widget.ListView;
import android.widget.SimpleCursorAdapter;
import android.widget.AdapterView.OnItemClickListener;

public class ContactPicker extends Activity {
  @Override
  public void onCreate(Bundle icicle) {
    super.onCreate(icicle);
    setContentView(R.layout.main);
  }
}
```

2. Modify the main.xml layout resource to include a single `ListView` control. This control will be used to display the contacts.

```
<?xml version="1.0" encoding="utf-8"?>
<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/contactListView"
    android:layout_width="fill_parent"
    android:layout_height="wrap_content"
  />
</LinearLayout>
```

3. Create a new listitemlayout.xml layout resource that includes a single Text View. This will be used to display each contact in the List View.

```
<?xml version="1.0" encoding="utf-8"?>
<LinearLayout
  xmlns:android="http://schemas.android.com/apk/res/android"
  android:orientation="vertical"
  android:layout_width="fill_parent"
  android:layout_height="fill_parent">
  <TextView
    android:id="@+id/itemTextView"
    android:layout_width="fill_parent"
    android:layout_height="wrap_content"
    android:padding="10px"
    android:textSize="16px"
    android:textColor="#FFF"
  />
</LinearLayout>
```

THE UNIVERSITY OF MICHIGAN LIBRARIES

4. Return to the `ContactPicker` Activity. Override the `onCreate` method, and extract the data path from the calling Intent.

```
@Override
public void onCreate(Bundle icicle) {
  super.onCreate(icicle);
  setContentView(R.layout.main);

  Intent intent = getIntent();
  String dataPath = intent.getData().toString();
```

4.1. Create a new data URI for the people stored in the contact list, and bind it to the List View using a `SimpleCursorArrayAdapter`.

The `SimpleCursorArrayAdapter` lets you assign Cursor data, used by Content Providers, to Views. It's used here without further comment but is examined in more detail later in this chapter.

```
final Uri data = Uri.parse(dataPath + "people/");
final Cursor c = managedQuery(data, null, null, null, null);

String[] from = new String[] {People.NAME};
int[] to = new int[] { R.id.itemTextView };

SimpleCursorAdapter adapter = new SimpleCursorAdapter(this,
                                        R.layout.listitemlayout,
                                        c,
                                        from,
                                        to);
ListView lv = (ListView)findViewById(R.id.contactListView);
lv.setAdapter(adapter);
```

4.2. Add an ItemClickListener to the List View. Selecting a contact from the list should return a path to the item to the calling Activity.

```
lv.setOnItemClickListener(new OnItemClickListener() {

  public void onItemClick(AdapterView<?> parent, View view, int pos,
                          long id) {
    // Move the cursor to the selected item
    c.moveToPosition(pos);
    // Extract the row id.
    int rowId = c.getInt(c.getColumnIndexOrThrow("_id"));
    // Construct the result URI.
    Uri outURI = Uri.parse(data.toString() + rowId);
    Intent outData = new Intent();
    outData.setData(outURI);
    setResult(Activity.RESULT_OK, outData);
    finish();
  }
});
```

4.3. Close off the `onCreate` method.

```
}
```

5. Modify the application manifest and replace the `intent-filter` tag of the Activity to add support for the pick action on contact data.

```
<?xml version="1.0" encoding="utf-8"?>
<manifest xmlns:android="http://schemas.android.com/apk/res/android"
  package="com.paad.contactpicker">
  <application android:icon="@drawable/icon">
    <activity android:name="ContactPicker"
              android:label="@string/app_name">
      <intent-filter>
        <action android:name="android.intent.action.PICK"/>
        <category android:name="android.intent.category.DEFAULT"/>
        <data android:path="contacts"
              android:scheme="content">
        </data>
      </intent-filter>
    </activity>
  </application>
</manifest>
```

6. This completes the sub-Activity. To test it, create a new test harness `ContentPickerTester` Activity. Create a new layout resource — `contentpickertester` — that includes a `TextView` to display the selected contact and a `Button` to start the sub-Activity.

```
<?xml version="1.0" encoding="utf-8"?>
<LinearLayout
  xmlns:android="http://schemas.android.com/apk/res/android"
  android:orientation="vertical"
  android:layout_width="fill_parent"
  android:layout_height="fill_parent">
  <TextView
    android:id="@+id/selected_contact_textview"
    android:layout_width="fill_parent"
    android:layout_height="wrap_content"
  />
  <Button
    android:id="@+id/pick_contact_button"
    android:layout_width="fill_parent"
    android:layout_height="wrap_content"
    android:text="Pick Contact"
  />
</LinearLayout>
```

7. Override the `onCreate` method of the `ContentPickerTester` to add a Click Listener to the button so that it implicitly starts a new sub-Activity by specifying the `PICK_ACTION` and the contact database URI (`content://contacts/`).

```
package com.paad.contactpicker;

import android.app.Activity;
import android.content.Intent;
import android.database.Cursor;
import android.net.Uri;
import android.os.Bundle;
import android.provider.Contacts.People;
import android.view.View;
```


THE UNIVERSITY OF MICHIGAN LIBRARIES

```
import android.view.View.OnClickListener;
import android.widget.Button;
import android.widget.TextView;

public class ContentPickerTester extends Activity {

  public static final int PICK_CONTACT = 1;

  @Override
  public void onCreate(Bundle icicle) {
    super.onCreate(icicle);
    setContentView(R.layout.contentpickertester);

    Button button = (Button)findViewById(R.id.pick_contact_button);

    button.setOnClickListener(new OnClickListener() {
      public void onClick(View _view) {
        Intent intent = new Intent(Intent.ACTION_PICK,
                                   Uri.parse("content://contacts/"));
        startActivityForResult(intent, PICK_CONTACT);
      }
    });
  }
}
```

8. When the sub-Activity returns, use the result to populate the Text View with the selected contact's name.

```
@Override
public void onActivityResult(int reqCode, int resCode, Intent data) {
  super.onActivityResult(reqCode, resCode, data);

  switch(reqCode) {
    case (PICK_CONTACT) : {
      if (resCode == Activity.RESULT_OK) {
        Uri contactData = data.getData();
        Cursor c = managedQuery(contactData, null, null, null, null);
        c.moveToFirst();
        String name;
        name = c.getString(c.getColumnIndexOrThrow(People.NAME));
        TextView tv;
        tv = (TextView)findViewById(R.id.selected_contact_textview);
        tv.setText(name);
      }
      break;
    }
  }
}
```

9. With your test harness complete, simply add it to your application manifest. You'll also need to add a `READ_CONTACTS` permission within a `uses-permission` tag, to allow the application to access the contacts database.

```
<?xml version="1.0" encoding="utf-8"?>
<manifest xmlns:android="http://schemas.android.com/apk/res/android"
```