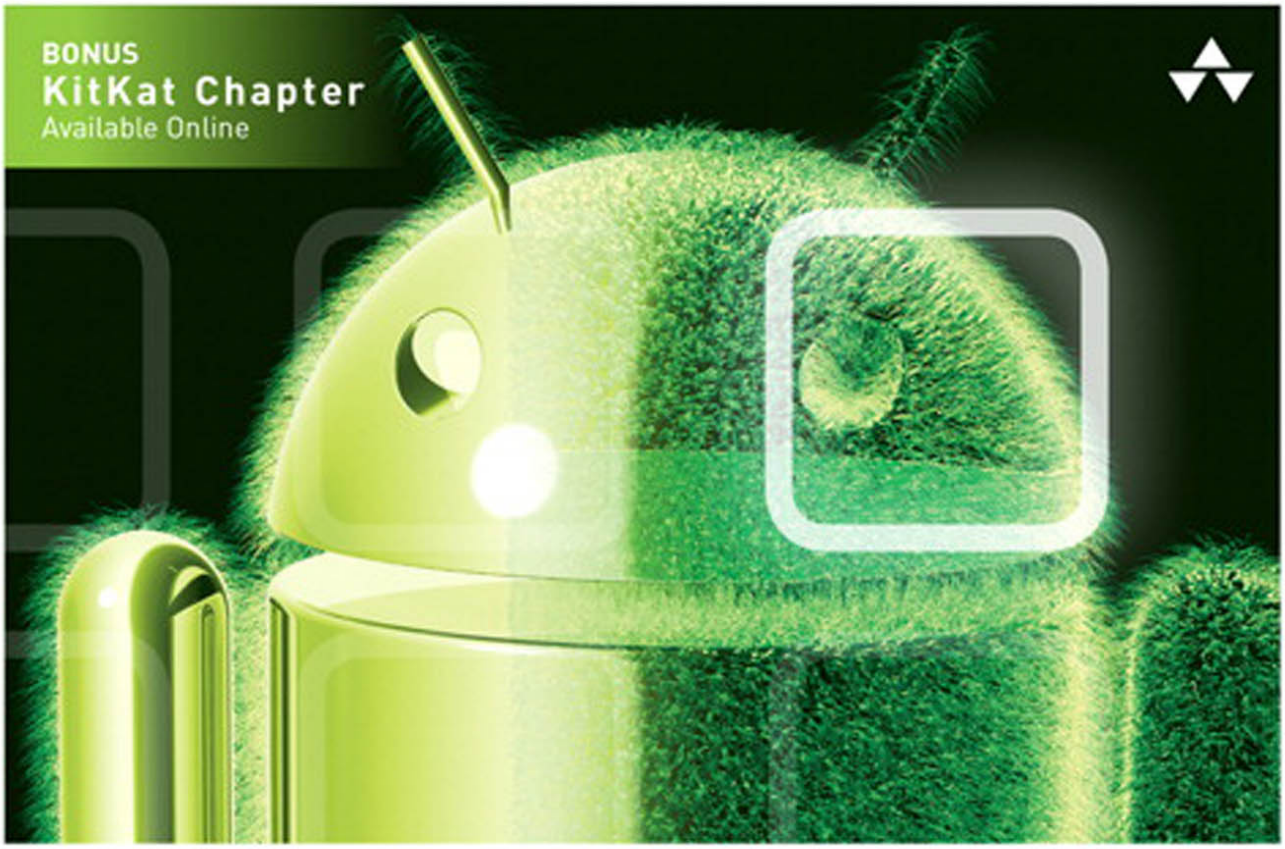



# LEARNING Android™ Application PROGRAMMING

A Hands-On Guide to Building Android Applications

JAMES TALBOT
JUSTIN McLEAN

# Praise for *Learning Android™ Application Programming*

"James Talbot and Justin McLean have done excellent work creating this beginner's tutorial. The hands-on focus of the book takes the reader from installing the development environment to writing the app and finally publishing the app. Topics include what most developers would want to know, from basic app structure and function, styling the app, testing and optimization, to using social media."
*—Matthew Boles, learning specialist, Brightcove Inc.*

"*Learning Android Application Programming* is a treasure trove of holistic information on developing applications for Android devices. It should satisfy those who prefer detailed descriptions as well as those who enjoy a book chock full of high-quality code samples. This book is a one-stop shop to take the reader from zero to app publish via a most relevant sample application. Throughout the book, the sample app is built, refactored, and optimized as the reader picks up all the necessary concepts and skills needed to become a true Android developer. Outstanding work, Justin and James!"
*—Jun Heider, senior architect and development manager, RealEyes Media*

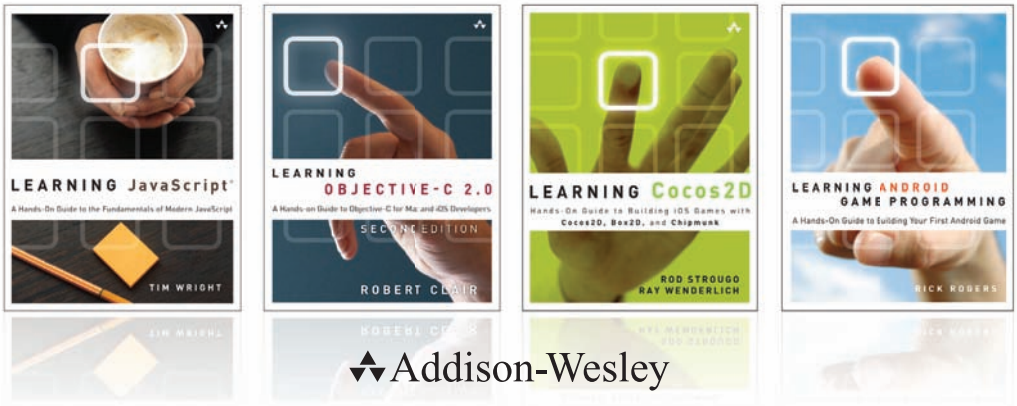
"A unique book that iteratively covers every aspect—requirements, design, developing, testing, and publishing—of a production-grade Android application."
*—Romin Irani, owner, Mind Storm Software*

"*Learning Android Application Programming* covers a rich variety of commonly encountered scenarios when approaching the Android development platform. Newcomers can step through the provided examples in an easily approachable format, while those who are more familiar with Android will find many useful nuggets scattered throughout. Everything is written in an understandable way and demonstrated through concrete examples, which can be immediately applied to a multitude of projects—great stuff!"
*—Joseph Labrecque, senior interactive software engineer, University of Denver*

*This page intentionally left blank*

# Learning Android™ Application Programming

# Addison-Wesley Learning Series




Visit informit.com/learningseries for a complete list of available publications.

---

The Addison-Wesley Learning Series is a collection of hands-on programming guides that help you quickly learn a new technology or language so you can apply what you've learned right away.

Each title comes with sample code for the application or applications built in the text. This code is fully annotated and can be reused in your own projects with no strings attached. Many chapters end with a series of exercises to encourage you to reexamine what you have just learned, and to tweak or adjust the code as a way of learning.

Titles in this series take a simple approach: they get you going right away and leave you with the ability to walk off and build your own application and apply the language or technology to whatever you are working on.

# Learning Android™ Application Programming

## A Hands-On Guide to Building Android Applications

James Talbot
Justin McLean

Addison-Wesley

Upper Saddle River, NJ • Boston • Indianapolis • San Francisco
New York • Toronto • Montreal • London • Munich • Paris • Madrid
Capetown • Sydney • Tokyo • Singapore • Mexico City

*Library of Congress Cataloging-in-Publication Data*

Talbot, James.
Learning Android application programming : a hands-on guide to building Android applications / James Talbot, Justin McLean.
pages cm
Includes index.
ISBN 978-0-321-90293-1 (pbk. : alk. paper)—ISBN 0-321-90293-9 (pbk. : alk. paper)
1. Android (Electronic resource) 2. Application software—Development. 3. Mobile computing. I. McLean, Justin. II. Title.
QA76.76.A65T35 2014
004.16—dc23

2013037213



ISBN-13: 978-0-321-90293-1
ISBN-10: 0-321-90293-9
Text printed in the United States on recycled paper at RR Donnelly in Crawfordsville, Indiana.
First printing, December 2013

*Copyright page continues on page 395.*

**Editor-in-Chief**
Mark L. Taub

**Executive Editor**
Laura Lewin

**Development Editor**
Michael Thurston

**Managing Editor**
John Fuller

**Full-Service Production Manager**
Julie B. Nahil

**Copy Editor**
Betsy Hardinger

**Indexer**
John S. Lewis

**Proofreader**
Rebecca Rider

**Technical Reviewers**
Romin Irani
Douglas Jones
Prashant Thakkar

**Editorial Assistant**
Olivia Basegio

**Cover Designer**
Chuti Prasertsith

**Compositor**
Kim Arney

❖

*I'd like to thank my family and friends,*
*as well as my colleagues at Adobe Systems.*
*This book is dedicated to my brand new niece, Lenora Talbot,*
*who is entering a world that is forever changed*
*by the mobile revolution.*
*—James Talbot*

*I'd like to thank my family, friends,*
*and all the new people I've met over the last year while traveling,*
*speaking at conferences, and writing this book.*
*Life would be a boring place without you.*
*Parts of this book were written in New York City; Los Angeles;*
*San Francisco; Portland; Gloucester, MA; Denver; St. Louis; Sydney;*
*Hobart; Perth; Melbourne; Brisbane; up Bostock Road (near Tucabia);*
*Cologne; Berlin; outside Arklow (near Dublin); London;*
*Ammanford in Wales; and Edinburgh.*
*It's been a fun adventure, and I hope you enjoy the book*
*as much as I've enjoyed working on it.*
*—Justin McLean*

❖

*This page intentionally left blank*

# Contents at a Glance

*This page intentionally left blank*

# Contents

*This page intentionally left blank*

# Preface

This is a book about learning how to program an Android application from start to finish. It assumes that you have some web development or programming experience but may not be familiar with the Java language or the Android operating system or have working knowledge of the Android API/SDK. This book teaches you best practices for programming Android applications and explains how to solve real-world issues such as device fragmentation. You'll learn how to code your application to work on the widest range of Android OSs while still taking advantage of the latest Android features, and you'll explore how to use (often inaccurate) data from sensors. You'll discover how to preserve the battery life of your device and how to make your application easily work in multiple countries and languages.

Each chapter builds upon the preceding chapter, step by step, until you have a complete working application. This book is best read in order, but you can skip around if you already understand the content in a chapter, because the code for each chapter can be found on the book's website and on GitHub. However, remember that the goal of this book is to learn by doing, and, if you follow each chapter, you will learn some useful best practices.

This book is aimed at web developers or programmers who may have little or no Android or Java experience and want to know how to write an Android application from start to finish. This book is not an API reference, and it isn't filled with small snippets of unconnected code. Instead, it's a hands-on, learn-as-you-go tutorial that helps you avoid the common traps and pitfalls that new Android developers get themselves into. As you go through each chapter, you'll build the *On Your Bike* Android application, a handy tool for bicycle riders. When you've finished the book, you'll have a complete application, and you will have learned enough to create your own application and publish it in Google Play and the Amazon Appstore.

While working through this book, it's recommended that you have access to an Android device. Although it's possible to work through most of the book using only a computer and the Android emulator, there are some things that will work only on a real device.

The color code in the printed book is meant to be representative of what you will see when you are programming in Eclipse. Colors do not match exactly but are close approximations of what you will see in the Eclipse Development Environment.

## Code Examples

The code listings for each chapter can be found at the book's website:

http://www.androiddevbook.com/code.html

They are also available on GitHub:

https://github.com/androiddevbook/onyourbike

The application can also be found in Google Play:

https://play.google.com/store/apps/details?id=com.androiddevbook.onyourbike.book

If you have any questions about the book or the code, please contact the authors at james@androiddevbook.com or justin@androiddevbook.com. You can follow the book on Twitter at @androiddevbook. The code and more information are on http://www.androiddevbook.com.

# Acknowledgments

The authors would like to thank the following:

Jorge Hernández, who proposed the initial idea of a cycle computer application and helped us write and format the initial chapters.

Romin Irani, Douglas Jones, and Prashant Thakkar for being the technical reviewers of the book. They picked up many issues, both big and small, during the review process, and the book would not be the quality it is without the time and effort they put into reviewing both the content and the code.

Betsy Hardinger, an absolutely amazing editor who caught many things we did not even think of. We appreciate her professionalism and diligence.

Michael Thurston, the development editor, who suffered through our bad grammar and spelling, inconsistent formatting, and confusing language and structure, and managed to get the draft document into a state fit for publishing.

Olivia Basegio, an incredible asset, who always stepped in and made us feel comfortable with the daunting process of publishing a book.

Laura Lewin, the editor of the project, who kept us on schedule and was tireless in her research and assistance. We owe a lot to her and really appreciate her professionalism.

We would also like to thank Julian Ledger for designing the Android *On Your Bike* icon and for general design guidance in the production of this book.

Finally, for the internationalization chapter we'd like to thank several people who provided translations of the application resource strings into their own language: Kai König (German), Carlos Rovira (Spanish), Frédéric Thomas (French), Christophe Herreman (Dutch), and John Koch (Japanese).

*This page intentionally left blank*

# About the Authors

**James Talbot** has been with Adobe for more than a decade, on the sales engineering, professional services, and training teams, and has many years of experience in working with object-oriented programming and web applications. He is currently working on constructing exciting web, mobile web, and native Android applications built on top of a Java Content Repository (JCR) based on open source standards. He cowrote *Object-Oriented Programming with ActionScript 2.0* (New Riders Press, 2004) and *Adobe Flex 2: Training from the Source* (Adobe Press, 2006), as well as *Adobe Flex 3: Training from the Source* (Adobe Press, 2008). He has also recorded training videos for Lynda.com and Total Training and has spent extensive time teaching in the classroom. He has deep knowledge of all Adobe web products and has spoken at numerous conferences.

**Justin Mclean** has been writing code since the early days of the web. For 15 years he has managed his own consulting company, Class Software, and during that time he has worked on hundreds of browser, desktop, and mobile applications. He has seen significant changes of technology in the industry, surviving the browser wars and the dot-com bubble. He is an Apache Flex committer, board member, and release manager and an Adobe Community Professional. He teaches training courses and has spoken at numerous conferences all over the world. In his spare time he tinkers about with open source electronics.

*This page intentionally left blank*

# We Want to Hear from You!

As the reader of this book, you are our most important critic and commentator. We value your opinion and want to know what we're doing right, what we could do better, what areas you'd like to see us publish in, and any other words of wisdom you're willing to pass our way.

You can email or write us directly to let us know what you did or didn't like about this book—as well as what we can do to make our books stronger.

*Please note that we cannot help you with technical problems related to the topic of this book, and that due to the high volume of mail we receive, we might not be able to reply to every message.*

When you write, please be sure to include this book's title and author as well as your name and phone or email address.


Email: laura.lewin@pearson.com
Mail: Reader Feedback
Addison-Wesley Learning Series
800 East 96th Street
Indianapolis, IN 46240 USA


## Reader Services

Visit our website and register this book at **informit.com/register** for convenient access to any updates, downloads, or errata that might be available for this book.

*This page intentionally left blank*

# 4

# Going for Your First Ride: Creating an Android User Interface

*Life is like a riding a bicycle,*
*you don't fall off unless you stop pedaling.*

—Claude Pepper

Now it's time to begin coding the **On Your Bike** application. This Android app will act as a bicycle computer—a device, usually clipped on the handlebars, that helps you keep track of the length and time of your ride. By creating this application, you will learn more about how to code with the Android activity lifecycle, how to code a simple user interface, and how to specify user preferences.

## Refactoring Your Code

Because of project time pressures, you often need to make quick changes to code. Over time, these little changes add up, and, as a result, you need to revisit the code before the project is complete. This is known as technical debt. The code base becomes fragile, and it's easy to introduce bugs and more difficult to maintain the code. It's important to have a spring cleaning every now and then to fix the most obvious issues.

It makes sense to rearrange the code at a time when you're not trying to change its functionality, a process referred to as refactoring. Of course, it's also much easier to change functionality when you have clean, refactored code.

When you're undertaking a major refactoring, don't forget to back up your code first, or, better still, keep your code under version control. But don't despair if you get lost and make a mistake with your code: You can always download the code for the

chapter from the **On Your Bike** website (http://www.androiddevbook.com) or from GitHub (https://github.com/androiddevbook/onyourbike).

The simplest form of refactoring is to rename packages, classes, methods, and variables. You might do this for several reasons.

- Renaming a class, method, or variable will increase the readability or understanding of the existing code.
- Naming wasn't consistent across the application.
- A method's functionality has changed, and it now does something a little different from what its original name indicated. It makes sense to rename the method to something more descriptive.
- You can move duplicate blocks of code into a single new method. This can help implement the Don't Repeat Yourself (DRY) principle, whose primary purpose is to prevent the repetition of information.
- You can break larger methods into several smaller methods so that they can be reused. This will also make the code more readable.

Always remember, your code should be human readable first, and machine readable second. If you've ever had to work on other people's code or returned to code you wrote months ago, you'll be thankful for that readability. If you don't follow this principle, it can result in substantial frustration. You may end up cursing yourself—or the original developer.

Now let's refactor your ongoing project to better describe it. Follow these steps.

1. In the **Package Explorer** view, do the following.
   - Expand the `/src` directory.
   - Right-click the `com.androiddevbook.onyourbike.chapter3` package.
   - Select **Refactor > Rename**.
   - Change the end of the package name from `chapter3` to `chapter4`, as shown in Figure 4.1. Keep the **Update references** checkbox checked.
   - Click **Preview** to check the changes that will take place. You will see that the import statements in **MainActivity** will change and that the package will be renamed.
   - Click **OK** to apply the changes. Ignore any compiler errors that are shown.
2. Perform the same procedure (by right-clicking the filename and selecting **Refactor > Rename**) with the **MainActivity** class, and rename it **TimerActivity**.
3. Locate the `\res\layout\activity _ main.xml` file, and rename it **activity_timer.xml**.

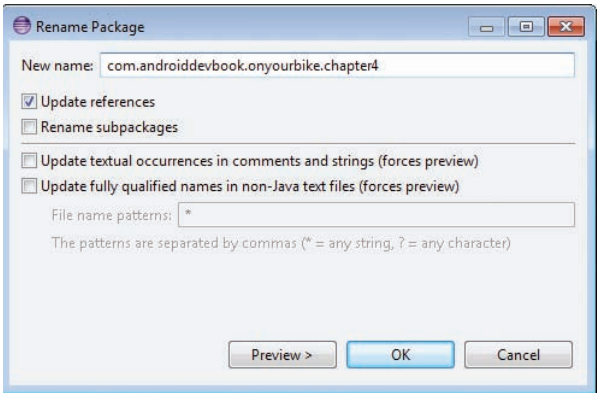



Figure 4.1 Rename Package dialog box in Eclipse

4. Change the call to the `setContentView` method in **TimerActivity** to pass the new activity identifier:

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

5. After you save the **TimerActivity.java** file, the compilation error will be resolved.
6. Open `\res\values\strings.xml`, and change the following lines to reflect a new application name and a new title.
   - Change the value of the string node with an attribute `app_name` to the following:

```
<string name="app_name">On Your Bike - Chapter 4</string>
```

   - Change the name attribute `title_activity_main` to `title_activity_timer`, and the node value to the following:

```
<string name="title_activity_timer">Timer</string>
```

7. Double-click on the error in the *Problem* view to open the `AndroidManifest.xml` file. Change the following.
   - Change the package name to match the new package:

```
package="com.androiddevbook.onyourbike.chapter4"
```

   - Change the activity name to match the new activity class:

```
android:name=".TimerActivity"
```

   - Change the activity label to match the new string resource:

```
android:label="@string/title_activity_timer"
```

8. From the **Refactor** menu, select **Rename**, and rename the `className` constant in **TimerActivity**. It is better practice to define a variable treated as a constant with uppercase letters and make it private so that it is not visible outside the class:

   ```
   private static String CLASS_NAME;
   ```

   Eclipse will automatically rename all references to the constant.

9. Rename the project **On Your Bike Chapter 4** by right-clicking on the project name, selecting **Refactor -> Rename**, entering the new name, and clicking **OK**. It is a good idea to clean your project after making all the changes to make sure that everything has been recompiled and to double-check that there are no errors. You do this by selecting **Project > Clean**.

# Implementing Strict Mode

When you're first programming for Android, you need to be aware of several gotchas that may trip you up. For example, it's common to accidentally block the user interface thread and cause your application to perform badly or, even worse, to become unresponsive. Strict mode was added to the Android SDK to identify issues like this. It's a good idea, especially when you're starting out, to always turn on Strict mode.

Strict mode is flexible in that you can filter issues so that it reports only the ones you're interested in and, when those issues occur, what sort of action should be taken.

You can take the following actions:

- Logging the issue to LogCat
- Flashing the device's screen
- Stopping the application
- Opening a dialog box

Setting up Strict mode in your application is straightforward.

1. To enable Strict mode, add the code in Listing 4.1 after the call to `Log.d` in the `onCreate` method of `TimerActivity`.

   Listing 4.1 **Turning On Strict Mode in `onCreate`**

   ```
   if (BuildConfig.DEBUG) {
       StrictMode.setThreadPolicy(new StrictMode.ThreadPolicy.Builder()
           .detectAll().penaltyLog().build());
       StrictMode.setVmPolicy(new StrictMode.VmPolicy.Builder()
           .detectAll().penaltyLog().penaltyDeath().build());
   }
   ```

This code will detect all issues with threading and display them to the *LogCat* view. It will also detect common memory leaks, log them, and stop the application. Note that the `Builder` constructor and all the various detect and penalty methods return the current instance of `builder`. This is known as function chaining. In this way, methods can be called together one after another to make the code more readable and concise.

2. A few errors will show in the *Problem* view. Run **Quick fix** `StrictMode` to add the import statement:

```
import android.os.StrictMode;
```

## Creating a Simple User Interface

At this point, your `activity _ timer` activity is using as its base tag the `RelativeLayout` view group. By using the `RelativeLayout` class, you're telling the app to position the views in relation to how other views are positioned. For example, the position of views could be determined by whether the views are to the right or left of another view, below or above another view, centered in the view group, aligned to the left or right of each other, or even aligned to the bottom or top of the view group.

The values for the layout properties are either a Boolean or an ID that references another view. In the XML layout, they can be declared in any order. For example, if `android:layout _ centerVertical` is set to `true`, then the top edge of the view will match the top edge of the parent. If `android:layout _ below` is set, then the top edge of the view will be below the view specified with a resource ID—for example, `android:layout _ below="@id/name"`. If `android:layout _ toRightOf` is set, then the left edge of the view will be to the right of the view with the resource ID.

Once you have indicated the position of the views in a view group, you can then specify the layout width and layout height. These measurements can be an exact number and a unit.

Possible units of measurement include the following.

- Density-independent pixels (dp): Use to make UI elements the same size on different screen densities.
- Pixels (px): Try to use dp instead.
- Scale-independent pixels (sp): Use for font sizes that scale according to the user preference and the screen density.
- Points (pt): Try to use sp instead.
- Millimeters (mm) and inches (in): Avoid if possible.

You can also specify the height and width in terms of the view's actual size or the view group's size; to do this, set the width or height layout attribute to

`wrap _ content` or `match _ parent`. This gives you even more flexibility in designing layouts for devices of various sizes. (Note that `match _ parent` was called `fill _ parent` in earlier versions of the SDK, so you may come across this in old code.) The `wrap _ content` attribute makes the view as big as it needs to be, so the view group layout may include gaps; `match _ parent` also makes the view resize, so there are no gaps in the view group's layout except for the padding.

There are other concepts that come into play when you're laying out views. Weight describes how the total width or height is shared between multiple views. For example, each child view is given a proportion of its weight over the total weight of all views. If all child views have the same weight, then all of them will have the same height and width. However, if a child view has a weight of 2 and other child views have a weight of 1, then the first child view will be twice as high and twice as wide as the other views.

If any of the child views also has a width or height, then the remaining space is divided by the weights; in the preceding example, the first child would be proportionally wider but not twice as wide as the other views. You often need to experiment with the right combination of width, height, and weight to get something that works for each view.

It makes sense to do one of two things: either (1) express weight in terms of how much bigger or smaller a view is compared to its siblings or (2) make the weights add up to 100 so that the weight can be thought of as a percentage. You should use whatever makes sense in the layout.

Another view group is `LinearLayout`. A `LinearLayout` enables you to position views vertically (one view above the other on the screen) or horizontally (the views side by side). To control whether the views inside a `LinearLayout` are positioned horizontally or vertically, set the orientation attribute to `horizontal` or `vertical`.

Note that layouts are defined in this way so that the screen size of an activity is mostly irrelevant and activities scale and resize to fit on a wide range of screen sizes and densities. In this way, your app can display correctly on all the different Android devices out there.

Other layout view groups include `GridView`, `ListView`, and `WebView`. As you might expect, `Gridview` displays items in a grid, `ListView` displays views in a vertical list, and `WebView` displays web pages. Laying out a UI is a complex topic, and you will learn much more about it as you begin to build the application.

## Using Linear Layouts

The basic display on a bicycle computer includes a timer that tracks how long you've been riding. You will build this functionality in this section. The first step is to build a user interface that will include a Start button as well as a Stop button for the timer. The timer output will appear on the text view you have already created. To build this functionality, follow these steps.

1. Edit the existing TextView in the activity _ timer.xml file. Remove the line that sets the android:text attribute. (The text will no longer be hard-coded but instead will be dynamic and changed through code you will add later.)
2. Change the android:id to the value @+id/timer:

```
<TextView
    android:id="@+id/timer"
```

3. Change the toolContext to be the TimerActivity class by assigning it a value .TimerActivity:

```
tools:context=".TimerActivity"
```

4. Add a LinearLayout below the TextView:

```
<LinearLayout>
</LinearLayout>
```

5. In the linear layout you just added, you will add two buttons. The buttons need to stretch horizontally. To do this, change the android:layout _ width to match _ parent. For the buttons to be as high as they need to be, set the android:layout _ height to wrap _ content. Set the android:orientation to horizontal so that the buttons are side by side:

```
<LinearLayout
    android:layout _ width="match _ parent"
    android:layout _ height="wrap _ content"
    android:orientation="horizontal">
```

6. Still inside the LinearLayout tag, align the buttons at the bottom of the screen by assigning the android:layout _ alignParentBottom to true:

```
android:layout _ alignParentBottom ="true"
```

7. Also inside the LinearLayout tag, add the Start button inside the linear layout, give it an android:id of @+id/start _ button, and set the android:layout _ width and android:layout _ height to wrap _ content:

```
<Button
    android:id="@+id/start _ button"
    android:layout _ width="wrap _ content"
    android:layout _ height="wrap _ content"/>
```

8. Still inside the LinearLayout tag, add the Stop button after the Start, give it an android:id of @+id/stop _ button, and set the android:layout _ width and android:layout _ height to wrap _ content:

```
<Button
    android:id="@+id/stop _ button"
    android:layout _ width="wrap _ content"
    android:layout _ height="wrap _ content"/>
```

9. The two buttons need to be the same size, so set the `android:layout _ weight` on both to 1:

```
android:layout _ weight="1"
```

10. Click on the **Graphical Layout** view to check that there are no errors; that the view consists of a `TextView` in the center of the layout; and that there are two buttons of equal size at the bottom of the layout, as shown in Figure 4.2.
11. To the first button, add an `android:text` value of `@string/start _ button`:

```
android:text="@string/start _ button"
```

12. Add the same attribute to the second button with a value of `@string/stop _ button`:

```
android:text="@string/stop _ button"
```

13. Add the two created resources to `values/strings.xml`:

```
<string name="start _ button">Start</string>
<string name="stop _ button">Stop</string>
```

Your layout code should now look like Listing 4.2.

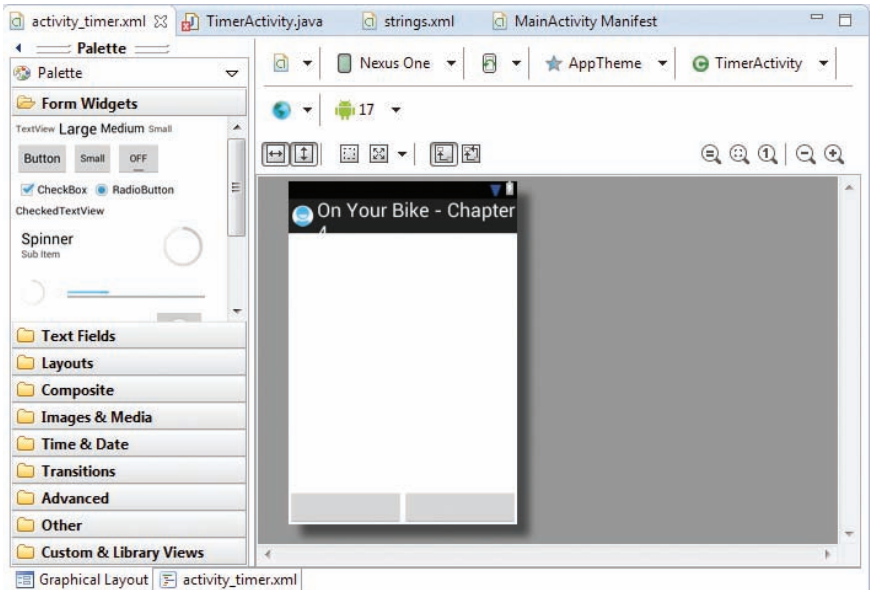



Figure 4.2 Graphical Layout view showing two blank buttons

Listing 4.2 **Linear Layout Containing Two Buttons**

```
<RelativeLayout xmlns:android="http://schemas.android.com/apk/res/android"
    xmlns:tools="http://schemas.android.com/tools"
    android:layout_width="match_parent"
    android:layout_height="match_parent" >

    <TextView
        android:id="@+id/timer"
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:layout_centerHorizontal="true"
        android:layout_centerVertical="true"
        tools:context=".TimerActivity" />

    <LinearLayout
        android:layout_width="match_parent"
        android:layout_height="wrap_content"
        android:layout_alignParentBottom="true"
        android:orientation="horizontal">

        <Button
            android:id="@+id/start_button"
            android:layout_width="wrap_content"
            android:layout_height="wrap_content"
            android:layout_weight="1"
            android:text="@string/start_button" />

        <Button
            android:id="@+id/stop_button"
            android:layout_width="wrap_content"
            android:layout_height="wrap_content"
            android:layout_weight="1"
            android:text="@string/stop_button" />
    </LinearLayout>

</RelativeLayout>
```

14. Open `TimerActivity.java` and either correct or run **Quick fix** to address the error by changing `hello` to `timer`.
15. **Debug** your application. The activity should be displayed, as shown in Figure 4.3. You can click both buttons, but they don't do anything yet.

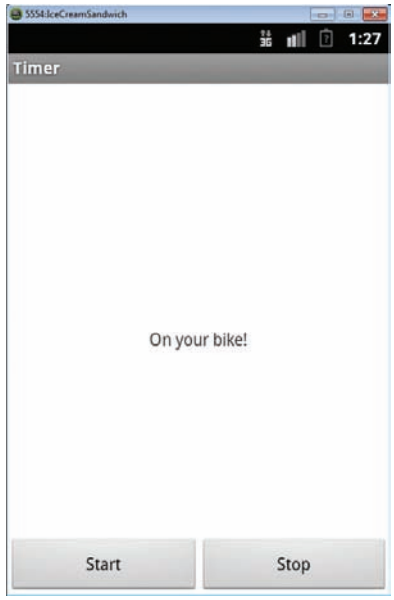



Figure 4.3 Debugging application showing buttons

## Creating Button Event Handlers

To make the buttons do something, you need to add event handlers to the buttons that detect when they are clicked, and you need to supply the method to be called. There are several ways of doing this with the Android SDK, but first let's take the simple approach and add the handlers to the layout.

1. Open the `activity_timer.xml` layout file, and locate the two buttons you added earlier. Add the two click handlers to the appropriate buttons by setting `android:onClick` to the name of the methods you want called when the buttons are clicked. Call the two methods `clickedStart` and `clickedStop`, as shown in Listing 4.3.

Listing 4.3 **Adding Click Handlers to Two Buttons**

```
<Button
    android:id="@+id/start_button"
    android:layout_width="wrap_content"
    android:layout_height="wrap_content"
    android:layout_weight="1"
    android:text="@string/start_button"
    android:onClick="clickedStart" />
```

```
<Button
    android:id="@+id/stop _ button"
    android:layout _ width="wrap _ content"
    android:layout _ height="wrap _ content"
    android:layout _ weight="1"
    android:text="@string/stop _ button"
    android:onClick="clickedStop" />
```

2. Add the `clickedStart` and `clickedStop` methods to the `TimerActivity` class, logging that the methods have been called. Run **Quick fix** to import the `View` class. See Listing 4.4.

Listing 4.4 **Adding Click Handlers Methods**

```
public void clickedStart(View view) {
    Log.d(CLASS _ NAME, "Clicked start button.");
}

public void clickedStop(View view) {
    Log.d(CLASS _ NAME, "Clicked stop button.");
}
```

3. **Debug** the application. Click each button to make sure the click log messages are displayed in the *LogCat* view, as shown in Figure 4.4.

   Note that if the method names are incorrect (if they don't match what is in the layout XML), then the application will compile and run with no warnings or errors, but you will get a run time exception (RTE) when clicking on the button. This is the downside of specifying handlers this way, but it's easy enough to avoid with a little care and testing.

4. Add the following class properties at the top of the `TimerActivity` class declaration:

```
protected TextView counter;
protected Button start;
protected Button stop;
```

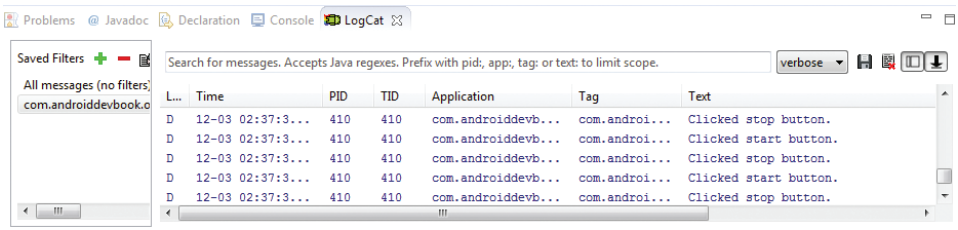



Figure 4.4 LogCat view showing Start and Stop button logs

5. Change the `onCreate` method to assign each of these variables to match the corresponding view in the layout. To do this in each case, call `findViewById`, passing the automatically generated identifier for that view. This must be done after the `setContentView` call; otherwise, you get an RTE when the application is run. Also, change the text view `findViewById` to refer to the new timer variable:

```
counter = (TextView) findViewById(R.id.timer);
start = (Button) findViewById(R.id.start_button);
stop = (Button) findViewById(R.id.stop_button);
```

6. Remove the `hello.setText` line. The text of this text view will now be set through code.
7. Create a new class property called `timerRunning` to store the state of the timer and whether or not it has been started. This in turn determines whether the buttons are enabled or disabled.

```
protected boolean timerRunning;
```

8. Add a new method called `enableButtons` to toggle which button (Start or Stop) is enabled depending on the value of `timerRunning`:

```
protected void enableButtons() {
    Log.d(CLASS_NAME, "Set buttons enabled/disabled.");
    start.setEnabled(!timerRunning);
    stop.setEnabled(timerRunning);
}
```

9. Call `enableButtons` after the calls to `findViewById` in `onCreate` and in the `clickedStart` and `clickedStop` methods.
10. Before the call to `enableButtons`, set the property `timerRunning` to true in `clickedStart`, and to false in `clickedStop`. Your two event handlers should now look like Listing 4.5.

Listing 4.5 **Button `onClick` Event Handlers**

```
public void clickedStart(View view) {
    Log.d(CLASS_NAME, "Clicked start button.");
    timerRunning = true;
    enableButtons();
}

public void clickedStop(View view) {
    Log.d(CLASS_NAME, "Clicked stop button.");
    timerRunning = false;
    enableButtons();
}
```

11. **Run** the application. The buttons should now toggle to the one that is enabled when it is clicked, as shown in Figure 4.5.

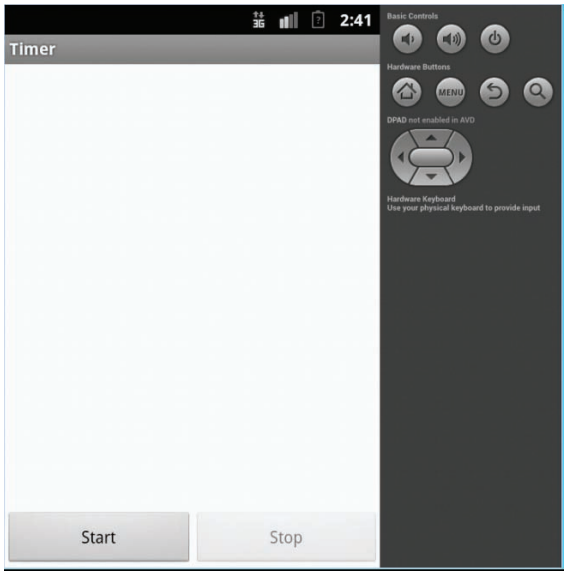



Figure 4.5 Debugging application showing enabled button

## Updating the Timer Display

For the application to work as a bicycle computer, the counter needs to be updated frequently. This update is based on how much time has passed since the timer was started. There are two parts to solving this issue: updating the display and doing it at regular intervals. Let's first update the display.

1. Create two new properties of type `long` in the `TimerActivity` class called `startedAt` and `lastStopped`:

   ```
   protected long startedAt;
   protected long lastStopped;
   ```

2. In the `clickedStart` method, set `startedAt` to contain the current time in milliseconds:

   ```
   startedAt = System.currentTimeMillis();
   ```

   And in the `clickedStop` method, set `lastStopped` to contain the current time in milliseconds:

   ```
   lastStopped = System.currentTimeMillis();
   ```

   In this way, you can determine how long the timer has been running between a start click and a stop click.

3. Create a new method called `setTimeDisplay` that sets the counter's text to the elapsed time. The method should look like Listing 4.6.

Listing 4.6 **Method for Displaying the Elapsed Time**

```
protected void setTimeDisplay() {
    String display;
    long timeNow;
    long diff;
    long seconds;
    long minutes;
    long hours;

    Log.d(CLASS_NAME, "Setting time display");

    if (timerRunning) {
        timeNow = System.currentTimeMillis();
    } else {
        timeNow = lastStopped;
    }

    diff = timeNow - startedAt;

    // no negative time
    if (diff < 0) {
        diff = 0;
    }

    seconds = diff / 1000;
    minutes = seconds / 60;
    hours = minutes / 60;
    seconds = seconds % 60;
    minutes = minutes % 60;

    display = String.format("%d", hours) + ":"
            + String.format("%02d", minutes) + ":"
            + String.format("%02d", seconds);

    counter.setText(display);
}
```

The first section of Listing 4.6, after the local variable declarations and log call, checks to see whether the timer is running. If it is, it gets the current time; otherwise, it gets the time when the Stop button was last clicked.

The difference between the time the Start button was clicked (stored in `startedAt`) and the current time (stored in `timeNow`) is then calculated. This

gives the number of milliseconds that the counter has been running. Make sure that the difference is a positive number. You wouldn't want to display a negative time value.

The time difference is in milliseconds and needs to be converted to a more human-friendly representation of time. From the number of milliseconds, you can calculate the number of seconds, minutes, and hours through integer division and modulo arithmetic (the remainder after a number is divided by another). This reflects the way minutes and seconds normally wrap around on a clock.

Once the time is calculated, you can create and format a time string by using `String.format`. Notice the use of the format `String` `%02d`, which pads the minutes and seconds with an initial zero if needed.

Then the counter text can set to the value of the time-formatted string stored in `display`.

4. Add a call to the `setTimeDisplay` method at the end of the `clickedStart` and `clickedStop` methods.
5. **Run** the application. Click the Start button, and the timer will display 0:00:00. Wait a few seconds, and then click the Stop button. The timer will now display something different, such as 0:00:03.

## Displaying a Running Timer

Next, you need to update the display at regular intervals so that the current time is displayed. On Android this is not as straightforward as it may seem.

The activity's user interface runs in a single thread. If you block that thread for too long, the Android OS thinks your application has frozen, and you will get the Application Not Responding (ANR) dialog box. Strict mode (which you added earlier) will tell you about potential issues that could cause your application to become unresponsive.

One solution is to create an extra thread and do all the work in that thread; in this way, you would not block the main UI thread and would stop any ANRs. Unfortunately, though, simply using standard Java timers or threads is not the answer. That's because the Android SDK is not thread safe, and any thread you create in this manner will not be able to update the display. Only the UI thread can update the display.

The solution? You can create a timer by using the `Runnable` interface and the `Handler` class.

The `Runnable` interface defines a single method called `run` that you implement. (It takes no parameters and returns void.) This `run` method is called once when the new thread is started.

The `Handler` class allows you to queue calls to the `run` method (and a few other things) in a `Runnable` class. You can use this class to make a timer that fires at regular intervals.

There are a couple of other ways of implementing this—for instance, using `AsyncTask` or `Services`—but using `Runnable` and `Handler` is the most straightforward way. In the following, you will create a timer using the `Runnable` and `Handler` process.

1. Open `TimerActivity.java`, and, at the top of the class, create a `static long` called `UPDATE _ EVERY`. Set it to a value of 200; this is how often you want the screen counter to update. If you set it to 1000, it may not exactly match every second, and the timer display may miss seconds. You might want to play with this value to see what works best.

   ```
   private static long UPDATE_EVERY = 200;
   ```

2. Create a new class called `UpdateTimer` that implements `Runnable` and has a single `run` method. In the `run` method, log that it has been called.

   ```
   class UpdateTimer implements Runnable {

       public void run() {
           Log.d(CLASS_NAME, "run");
       }
   }
   ```

3. Add a `handler` property and an `updateTimer` property to the class:

   ```
   protected Handler handler;
   protected UpdateTimer updateTimer;
   ```

   Run **Quick fix** to add the import statement for the `Handler` class, making sure it is the `android.os.Handler` class that you import. The `UpdateTimer` class doesn't need an import, because it's in the same package as `Handler`.

4. At the end of the `clickedStart` method in the `TimerActivity` class, create a new instance of both properties, and call the handler's `postDelayed` method. This will cause the `run` method of `UpdateTimer` to be called in 200 milliseconds.

   ```
   handler = new Handler();
   updateTimer = new UpdateTimer();
   handler.postDelayed(updateTimer, UPDATE_EVERY);
   ```

5. **Debug** the application. Check that the `run` method is logged when you click the Start button.

6. At the end of `clickedStop`, stop any pending call to the `run` method by calling `removeCallbacks` and set the `handler` to `null`.

   ```
   handler.removeCallbacks(updateTimer);
   handler = null;
   ```

7. In the `run` method, comment out the log call (otherwise, the *LogCat* view will be flooded with messages). Add calls to set the timer display and call the `run` method again in another 200 milliseconds (via a call to `postDelayed`).

```
setTimeDisplay();
if (handler != null) {
    handler.postDelayed(this, UPDATE_EVERY);
}
```

   The null check is to make sure that the handler exists and the Start button has been clicked.

8. **Run** the application again. You should now see the timer counting up when the Start button is pressed, and the timer stopping when the Stop button is pressed (see Figure 4.6).

The application seems as though it is now working. Not quite. Run the application on a USB-connected device, start the timer, wait a while, and rotate the screen. What happened? If you're running in an emulator, you can rotate the screen via **Ctrl + F12** on Windows and **Ctrl + fn + F12** on Mac. The activity lifecycle, discussed briefly in Chapter 3, is the reason the application did not function. In the next section, you will examine the activity lifecycle in more detail to get to the bottom of this.

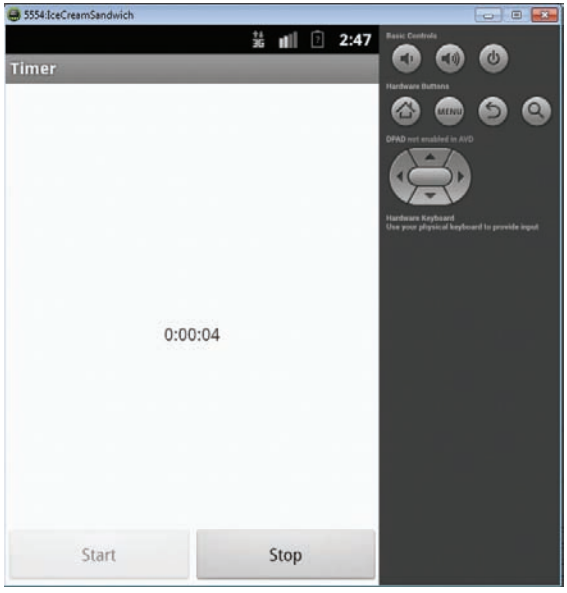



Figure 4.6 Debugging application showing timer

## Understanding the Activity Lifecycle

As you have seen, an activity is simply a screen or user interface in an Android application—either a full screen or a floating window that a user interacts with. An Android app is made up of different activities that interact with the user as well as one another. For example, a simple calculator would use one single activity. If you enhanced the calculator app to switch between a simple version and a scientific version, you would then use two activities.

Every Android application runs inside its own process. Processes are started and stopped to run an application and also can be killed to conserve memory and resources. Activities, in turn, are run inside the main UI thread of the application's process.

Once an activity is launched, it goes through a **lifecycle**, a term that refers to the steps the activity progresses through as the user (and OS) interacts with it. There are specific method callbacks that let you react to the changes during the activity lifecycle.

The activity lifecyle has four states.

- When the activity is on the foreground of the application, it is the *running* activity. Only one activity can be in the running state at a given time.
- If the activity loses focus but remains visible (because a smaller activity appears on top), the activity is *paused*.
- If the activity is completely covered by another running activity, the original activity is *stopped*. When an activity stops, you will lose any state and will need to re-create the current state of the user interface when the activity is restarted.
- While the activity is paused or stopped, the system can kill it if it needs to reclaim memory. The user can restart the activity.

While the application moves through the different states, the `android.app.Activity` lifecycle methods (or callbacks) get called by the system. These callbacks are as follows.

- `onCreate(Bundle savedInstanceState)` is called when the activity is created for the first time. You should initialize data, create an initial view, or reclaim the activity's frozen state if previously saved (this is covered later). The `onCreate` callback is always followed by `onStart`.
- `onStart()` is called when the activity is becoming visible. This is an ideal place to write code that affects the UI of the application, such as an event that deals with user interaction. This callback is normally followed by `onResume` but could be followed by `onStop` if the activity becomes hidden.
- `onResume()` is called when the activity is running in the foreground and the user can interact with it. It is followed by `onPause`.

- `onPause()` is called when another activity comes to the foreground. The implementation needs to be quick, because the other activity cannot run until this method returns. The `onPause` callback is followed by `onResume` if the activity returns to the foreground, or by `onStop` if the activity becomes invisible.
- `onStop()` is called when the activity is invisible to the user; either a new activity has started, an existing activity has resumed, or this activity is getting destroyed. The `onStop` callback is followed by `onRestart` if the activity returns to the foreground.
- `onRestart()` is called when the activity is being restarted, as when the activity is returning to the foreground. It is always followed by `onStart`.
- `onDestroy()` is called by the system before the activity is destroyed, either because the activity is finishing or because the system is reclaiming the memory the activity is using.

Figure 4.7 illustrates the various states the activity goes through and the order in which the callback methods get invoked.

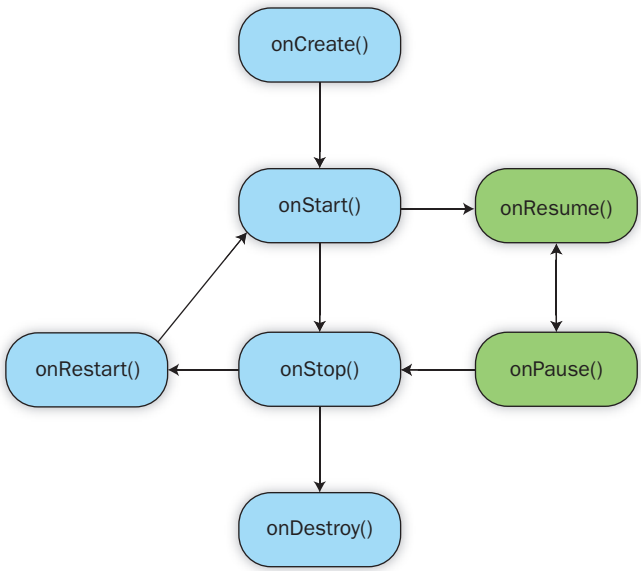



Figure 4.7 Activity lifecycle showing activity states

## Exploring the Android Activity Lifecycle

Now let's look at how the Android activity lifecycle works. In Chapter 3, you overrode the `onCreate` method. Now you'll override the remaining lifecycle methods in your `TimerActivity` class by following these steps.

1. Open the **TimerActivity.java** file in the project, and override the existing `onStart` method, which is called when the activity is first viewed. Call the `onStart` method of the parent class, and log a debug message:

```
@Override
public void onStart(){
    super.onStart();
    Log.d(CLASS_NAME, "onStart");
}
```

2. Override the existing `onPause` method, which is called when another activity is called to the foreground. Call the `onPause` method of the parent and log a debug message:

```
@Override
public void onPause(){
    super.onPause();
    Log.d(CLASS_NAME, "onPause");
}
```

3. Override the existing `onResume` method, which is called when the activity is running in the foreground and the user can interact with it. Call the `onResume` method of the parent class, and log a debug message:

```
@Override
public void onResume(){
    super.onResume();
    Log.d(CLASS_NAME, "onResume");
}
```

4. Override the existing `onStop` method, which is called when the activity is invisible to the end user. Call the `onStop` method of the parent class, and log a debug message:

```
@Override
public void onStop(){
    super.onStop();
    Log.d(CLASS_NAME, "onStop");
}
```

5. Override the existing `onDestroy` method, which is called when the activity is removed from the system and can no longer be interacted with. Call the `onDestroy` method of the parent class, and log a debug message:

```
@Override
public void onDestroy(){
    super.onDestroy();
    Log.d(CLASS_NAME, "onDestroy");
}
```

6. Override the existing `onRestart` method, which is called when the activity is started again and returns to the foreground. Call the `onRestart` method of the parent class and log a debug message:

```
@Override
public void onRestart(){
    super.onRestart();
    Log.d(CLASS_NAME, "onRestart");
}
```

7. Now **debug** your application on a device, and look at the debug messages (in the *LogCat* view) that show the changes of state in the application, as shown in Figure 4.8. Experiment with the application to see which state changes occur.
   - Turn your device on its side to see if the state changes. The activity is re-created when you do this, and in that process it loses all state.
   - Navigate to another application, and see which methods are called.
   - Let your device go to sleep, and then unlock the screen to see your application again.


Figure 4.8 LogCat showing activity lifecycle

### Fixing Activity Lifecycle Issues

As you've seen, when the application is not running there is no need to have the timer display update, and when the timer activity is re-created you need to refresh the display to put it into the correct state.

To fix these issues you need to update the screen at the correct time.

1. When `onStart` is called and the timer is still running, start calling the `run` method of `UpdateTimer` again. Add this code to the `onStart` method:

```
if (timerRunning) {
    handler = new Handler();
    updateTimer = new UpdateTimer();
    handler.postDelayed(updateTimer, UPDATE_EVERY);
}
```

2. When `onStop` is called, you no longer need to update the display. Add this code to the `onStop` method:

```
if (timerRunning) {
    handler.removeCallbacks(updateTimer);
    updateTimer = null;
    handler = null;
}
```

3. When `onResume` is called, you need to refresh the display. Add these two lines of code:

```
enableButtons();
setTimeDisplay();
```

4. Debug the application on a device, and rotate the screen when the timer is running. You should now see that the application behaves as you would expect.

## Making an Android Device Vibrate

Sometimes a device's screen may not be visible (for example, if it's in someone's pocket), so you need to indicate that time has passed in a nonvisual way. Making the device vibrate is a good way to do this.

Let's set up the code to vibrate once every 5 minutes, twice every 15 minutes, and three times every hour while the timer is running.

1. Add a property called `vibrate` of type `Vibrator` to the `TimerActivity` class:

```
protected Vibrator vibrate;
```

2. Add a property called `lastSeconds` of type `long`. This is needed because the `run` method is called several times a second, and you want the device to vibrate only once.

```
protected long lastSeconds;
```

3. In the `onStart` method, set up the `vibrate` property by calling `getSystemService`. Not all devices can vibrate (and most tablets can't), so you need to check and log when a device doesn't support the feature:

```
vibrate = (Vibrator) getSystemService(VIBRATOR_SERVICE);

if (vibrate == null) {
    Log.w(CLASS_NAME, "No vibration service exists.");
}
```

4. Add a new method called `vibrateCheck`, which should look like Listing 4.7. This method uses a similar approach as `setTimeDisplay`'s to work out the time difference, but you need only calculate the current minutes and seconds.

   To vibrate the device, you call the `vibrate` method, passing it an array of numbers. The numbers represent a vibration pattern, with the first number being the number of milliseconds to wait before starting. This is followed by how long it should vibrate and how long it should pause between each vibration.

Listing 4.7 **Method for Vibrating a Number of Times at Regular Intervals**

```
protected void vibrateCheck() {
    long timeNow = System.currentTimeMillis();
    long diff = timeNow - startedAt;
    long seconds = diff / 1000;
    long minutes = seconds / 60;

    Log.d(CLASS_NAME, "vibrateCheck");

    seconds = seconds % 60;
    minutes = minutes % 60;

    if (vibrate != null && seconds == 0 && seconds != lastSeconds) {
        long[] once = { 0, 100 };
        long[] twice = { 0, 100, 400, 100 };
        long[] thrice = { 0, 100, 400, 100, 400, 100 };

        // every hour
        if (minutes == 0) {
            Log.i(CLASS_NAME, "Vibrate 3 times");
            vibrate.vibrate(thrice, -1);
```

```
            }
            // every 15 minutes
            else if (minutes % 15 == 0) {
                Log.i(CLASS_NAME, "Vibrate 2 time");
                vibrate.vibrate(twice, -1);
            }
            // every 5 minutes
            else if (minutes % 5 == 0) {
                Log.i(CLASS_NAME, "Vibrate once");
                vibrate.vibrate(once, -1);
            }
        }

        lastSeconds = seconds;
    }
```

Once the minutes and seconds have been calculated, the code needs to check whether it is on one of the three vibration boundaries. If it is, it should vibrate the required number of times. Note the check `seconds != lastSeconds`. This makes sure you don't vibrate more than once per second, because this method could be called multiple times in a single second.

5. Inside the `run` method, add a check (before the handler check and `postDelayed` call) to see whether the timer is running and, if it is, to call the `vibrateCheck` method:

```
if (timerRunning) {
    vibrateCheck();
}
```

6. Debug the application in the emulator, and see that `vibrateCheck` is being called in the *LogCat* view.

7. Debug the application via USB debugging. An error will occur. Correct this error by adding the vibrate permission to the Android manifest file just after `<uses-sdk>`:

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

## Saving User Preferences

Because an activity's state is not saved automatically during its lifecycle, you need to save user preferences so that you can redisplay an activity in the correct state. Let's see how to do that.

## Creating a New Activity

Applications often consist of more than one activity. Let's create a new Settings activity to enable and disable vibration and create the best possible experience for the user.

1. Create a new activity called **activity_settings** via the Android New Activity wizard. Select **BlankActivity** as the template, **Settings** as the activity name, and **activity_settings** as the layout file. Type **Settings** as the title.
2. Open the `activity_settings` file. Change the `RelativeLayout` to a `LinearLayout` with a vertical orientation:

```
<LinearLayout xmlns:android=http://schemas.android.com/apk/res/android
    xmlns:tools=http://schemas.android.com/tools
    android:layout_width="match_parent"
    android:layout_height="match_parent"
    android:orientation="vertical" >
```

3. Add a new checkbox view inside the linear layout. Give it a new `id` of `vibrate_check`, and set the `layout_width` and `layout_height` to `wrap_content`. Set a resource text to the value `@string/vibrate_checkbox`:

```
<CheckBox
    android:id="@+id/vibrate_checkbox"
    android:layout_width="wrap_content"
    android:layout_height="wrap_content"
    android:text="@string/vibrate_checkbox" />
```

4. Add the new resource string `vibrate_checkbox` to the `strings.xml` file:

```
    <string name="vibrate_checkbox">Vibrate</string>
```

## Showing a New Activity

To show a new activity, you first need to create an intent. Intents, in their simplest form, are a description of an activity that you want to occur. (You can also start activities in other applications, as covered later in the book.)

Next, you'll create a new intent to display the Settings activity.

1. Open the `activity_timer` layout. To launch the new activity, add a new button to the linear layout. Give the button an ID of `settings_button`, and a click handler to call the method `clickedSettings` when the button is pressed:

```
<Button
    android:id="@+id/settings_button"
    android:layout_width="wrap_content"
    android:layout_height="wrap_content"
    android:layout_weight="1"
```

```
    android:text="@string/settings_button"
    android:onClick="clickedSettings" />
```

2. Add the new resource string for the Settings button:

```
<string name="settings_button">Settings</string>
```

3. In the `TimerActivity.java` file, add a new `clickedSettings` method:

```
public void clickedSettings(View view) {
    Log.d(CLASS_NAME, "clickedSettings");
}
```

4. **Debug** the application, and check that the `clickedSettings` call is logged in the *LogCat* view. If an RTE occurs, double-check that the `onClick` contains exactly the same method name as the new method just added.
5. In the `clickedSettings` method, create a new `Intent`. Then pass the application context and the class property of the `SettingsActivity`. Run **Quick fix** to add the import statement for the `Intent` class:

```
Intent settingsIntent = new Intent(getApplicationContext(),
    SettingsActivity.class);
```

6. Display the new activity by calling `startActivity`, passing the intent you just created:

```
startActivity(settingsIntent);
```

7. **Run** the application again, and click the Settings button. The setting activity (displaying a checkbox) with a single checkbox will replace the timer activity, as shown in Figure 4.9.

## Saving an Application's State

Application state can be stored in many ways, either as static properties stored globally in the application or through the use of the singleton pattern. This pattern is designed to control object creation, limiting the number of objects to one. Because there is only ever one instance of the application class, you can use that to act as a singleton.

Here's how to create a class to save and retrieve the application settings.

1. Create a new Java class called `Settings`. Add a `private static` (of type `String`) `CLASS_NAME`, and assign the class name in the class constructor:

```
public class Settings {
    private static String CLASS_NAME;

    public Settings() {
        CLASS_NAME = getClass().getName();
    }
}
```

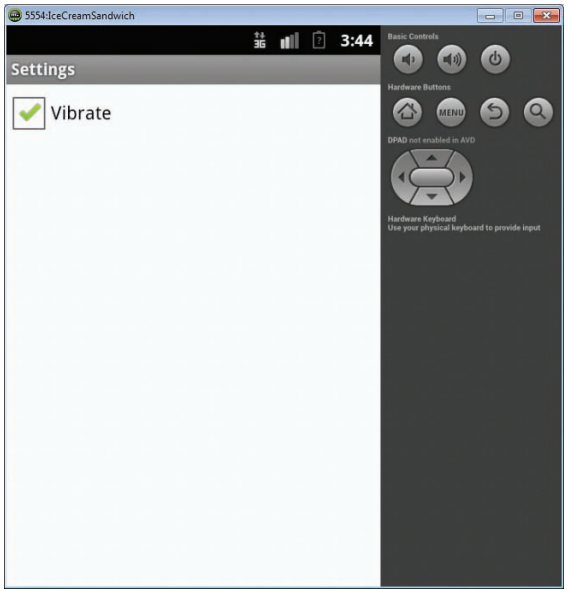



Figure 4.9 The new Settings activity

2. Create a private property to store whether or not the vibrate setting is turned on:

```
protected boolean vibrateOn;
```

3. Create a method to return this property. Run **Quick fix** to import the `Log` class:

```
public boolean isVibrateOn() {
    Log.d(CLASS_NAME, "isVibrateOn");
    return vibrateOn;
}
```

4. Create a method to set the value of the property:

```
public void setVibrate(boolean vibrate) {
    Log.d(CLASS_NAME, "setVibrate");
    vibrateOn = vibrate;
}
```

5. Create a new class called `OnYourBike` that extends `Application`. Add a `settings` property of type `Settings` to this class:

```
public class OnYourBike extends Application {
    protected Settings settings;
}
```

6. Add a method named `getSettings` that creates an instance of `Settings` if it hasn't already been created, and return the `settings` property:

```
public Settings getSettings() {
    if (settings == null) {
        settings = new Settings();
    }
    return settings;
}
```

7. Add a method named `setSettings` that changes the `settings` property to the `settings` value passed in:

```
public void setSettings(Settings settings) {
    this.settings = settings;
}
```

8. Change the Android manifest file so that the application uses this class as its application by setting the `android:name` attribute to `".OnYourBike"`:

```
<application android:name=".OnYourBike"
    android:allowBackup="true"
    android:icon="@drawable/ic_launcher"
    android:label="@string/app_name"
    android:theme="@style/AppTheme" >
```

9. Open **SettingActivity.java**, and add a vibrate checkbox property. Run **Quick fix** to import the `CheckBox` class:

```
private CheckBox vibrate;
```

10. In the `onCreate` method, after the call to `setContentView`, obtain access to the checkbox by calling `findViewById`:

```
vibrate = (CheckBox)
    findViewById(R.id.vibrate_checkbox);
```

11. Obtain the settings by calling the `getSettings` method just created:

```
Settings settings = ((OnYourBike)getApplication()).getSettings();
```

12. Just after that, set the state of the checkbox according to the setting:

```
vibrate.setChecked(settings.isVibrateOn());
```

13. Override the `onStop` method to save the settings:

```
@Override
public void onStop() {
    super.onStop();
    Settings settings = ((OnYourBike) getApplication()).getSettings();
    settings.setVibrate(vibrate.isChecked());
}
```

14. **Run** the application, click the **Settings** button, change the settings checkbox, and press the back button. Go back into the setting activity again by clicking the **Settings** button. The vibrate checkbox should still be ticked.

Notice that there was no need to add a handler to the checkbox for the state to be saved when the activity was stopped. Depending on how the activity is used in your application, you may want to save the setting right away rather than wait until the activity is stopped.

## Using Shared Preferences

The settings class you created saves the application's state only while it is running. If the application is stopped and restarted, it won't remember the previous state. To fix that, you need to use shared preferences to save the application's state. Shared preferences allow you to save key value pairs on a device.

You can save the vibration setting—whether it's turned on or off—as a preference:

1. Open **Settings.java**, and add a `private static` string called `VIBRATE`:

```
private static String VIBRATE = "vibrate";
```

2. In the `isVibrateOn` method, obtain an instance of shared preferences by calling `activity.getPreferences`:

```
SharedPreferences preferences
    = activity.getPreferences(Activity.MODE_PRIVATE);
```

   Run **Quick fix** to import the `SharedPreferences` and `Activity` classes.

3. Check whether the `VIBRATE` keys exist, and, if they do, set `vibrateOn` to be the saved value:

```
if (preferences.contains(VIBRATE)) {
    vibrateOn = preferences.getBoolean(VIBRATE, false);
}
```

4. Change the `isVibrateOn` method to take a single parameter of type `Activity`:

```
public boolean isVibrateOn(Activity activity)
```

5. In the `setVibrate` method, after the existing code, save the `vibrate` property by getting access to the shared preferences, creating an editor, saving the property by calling `putBoolean`, and committing the changes by calling `apply`:

```
SharedPreferences preferences
    = activity.getPreferences(Activity.MODE_PRIVATE);
Editor editor = preferences.edit();
editor.putBoolean(VIBRATE, vibrate);
editor.apply();
```

   Run **Quick fix** to import the `Editor` class.

6. Change the `setVibrate` method to take an additional parameter of type `Activity`:

```
public void setVibrate(Activity activity, boolean vibrate)
```

7. Open **SettingsActivity.java**, and fix the two errors by passing `this` to the `isVibrateOn` and `setVibrate` methods:

```
vibrate.setChecked(settings.isVibrateOn(this));
settings.setVibrate(this, vibrate.isChecked());
```

8. **Run** the application, click **Settings**, check the **vibrate** checkbox, and press the **back** button. Click **Menu**, and select **all apps**. Select your application, and click **force stop**. **Run** the application again, and click the **Settings** button. The **vibrate** checkbox should still be checked.

## Summary

It's important to refactor and keep your code clean, as you've learned in this chapter. Android gives you a way to lay out child views in relation to each other and to their parent view group, and adding event handlers to your code lets your app react to button clicks.

Looking further into the activity lifecycle identifies a few issues with the application you're building. To fix these issues, you implement simple threading by using the `Runnable` interface and the `Handler` class. (Remember it's important to not hold up the main thread of the UI, or you'll get the dreaded Application Not Responding dialog box.) You can display a new activity by creating an intent and calling `startActivity`.

To store your application's state, you can create a data model and extend the `Application` class. In this data model, you store a simple user preference to control whether or not the device will vibrate.

# Index

## B

## C

## D



## E

## F



## G

## H



## I

## J

## K



## L

## M

## N



## O

## P

## Q



## R



## S

## T

## U



## V



## W

## X

## Trademark Notices

Oracle and Java are registered trademarks of Oracle Corporation and/or its affiliates.

Apple, Mac, and OS X are trademarks of Apple Inc., registered in the U.S. and other countries.

Windows, Windows XP, Windows Visa, Windows 7, and Exchange are registered trademarks of Microsoft Corporation in the United States and other countries.

“Eclipse” is a trademark of Eclipse Foundation, Inc.

IntelliJ® is a registered trademark owned by Jetbrains s.r.o.

SQLite is a registered trademark owned by Hwaci.

OpenGL is a registered trademark of Silicon Graphics International Corp. in the U.S. and/or other countries worldwide.

Apache Ant, Apache Subversion, and Subversion are trademarks of The Apache Software Foundation.

GitHub is a trademark of Github, LLC.

SourceTree by Atlassian is a trademark of Atlassian.

Twitter is a registered trademark of Twitter, Inc.

Facebook® and Instagram™ are trademarks and registered trademarks of Facebook Inc.

Apkudo is a trademark of Apkudo LLC.

Adobe and Flash are either registered trademarks or trademarks of Adobe Systems Incorporated in the United States and/or other countries.

PhoneGap is a trademark of Adobe Systems Incorporated.

Sencha Touch™ are registered trademarks of Sencha, Inc.

Titanium™ is a trademark of Appcelerator, Inc., registered in the U.S and /or other countries and is used under license.

Kendo UI is a trademark of Telerik AD.

All of the Tumblr trademarks displayed in this book are the property of Tumblr, Inc.

Amazon, Amazon Appstore for Android, Kindle, and Kindle Fire are trademarks of Amazon.com, Inc., or its affiliates.

BlackBerry®, RIM®, Research In Motion® and related trademarks, names, and logos are the property of Research In Motion Limited and are registered and/or used in the U.S. and countries around the world. Used under license from Research In Motion Limited.

MOTOROLA and MOTOBLUR are trademarks or registered trademarks of Motorola Trademark Holdings, LLC.

HTC is a trademark of HTC Corporation.

LG is a registered trademark of LG Electronics, Inc.

Samsung, Galaxy S3, Galaxy S4, Galaxy Note, Galaxy Camera, Pocket, and TouchWiz are trademarks of Samsung Electronics Co., Ltd.

OUYA is a trademark of OUYA, Inc.

Verizon Wireless is a trademark of Verizon Trademark Services, LLC.

AT&T is a trademark of AT&T Intellectual Property or AT&T affiliated company (“AT&T Marks”).

T-Mobile is registered and/or unregistered trademark of Deutsche Telekom AG.

The Bluetooth® word mark and logos are registered trademarks owned by Bluetooth SIG, Inc.

Wi-Fi® is a registered trademark of Wi-Fi Alliance.

USB is a trademark or registered trademark of USB Implementers Forum corporation in the United States and/or other countries.

SD is a trademark or registered trademark of SD-3C, LLC, in the United States, other countries, or both.