TURNING IDEAS AND SKETCHES INTO **BEAUTIFULLY DESIGNED APPS**

ANDROID™ User Interface Design

Ian G. **CLIFTON**

Praise for *Android User Interface Design*

"*Android User Interface Design* is a truly excellent book, written by one of the most experienced and knowledgeable Android developers. This is a very practical, highly readable guide and a great how-to resource for every Android developer. Each chapter reveals a clear and deep understanding of UI design. I highly recommend this book to anyone wishing to develop Android apps with superior UI."

—Kyungil Kim

Software Engineer, Facebook

"I recommend this book for all Android developers who work alone and want to give a professional look to their apps. The content of the book is excellent and covers all aspects needed to understand how to design Android apps that stand out."

—Gonzalo Benoffi

CEO, Coffee and Cookies, Android Development

"Design was never part of a developer's job until mobile app development started; now it's a must. This book gives a simple yet effective way to design your apps. It's easy for beginners and informative for experienced developers as well. This is the best book I could ever refer to anyone who is in Android development. A one-time read of this book covers the experience you might gain from three years of learning development. I am amazed to see instructions on how to design starting from wireframes, which is something no other book has provided clear enough explanation of. (Some don't even cover it.) I really love it. Thanks to Ian for this wonderful contribution to the Android developer community. Best, simple, and effective!"

—Chakradhar Gavirineni

Android Application Developer, Adeptpros IT Solutions Pvt Ltd.

"Ian's book is an invaluable resource for everything there is to know about designing, creating layouts, and rendering Android applications. The 'Common Task Reference' appendix is an excellent addition that makes this book a must-have. Make sure to keep this one within arm's reach of your desk."

—Josh Schumacher

Software Engineer, HasOffers

"From the first few pages, this book provides a wealth of tips, tricks, and techniques for developing Android user interfaces. If you are grappling with all the various view types, then read this book—it really helps cement when and why you should include the various UI components to great effect (with worked examples!). Well worth a read by anyone looking for inspiration to improve their user interface into a great user experience."

—Richard Sey

PassBx Developers

Android™ User Interface Design

Turning Ideas and Sketches into Beautifully Designed Apps

Ian G. Clifton

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 is on file.

ISBN-13: 978-0-321-88673-6
ISBN-10: 0-321-88673-9

Text printed in the United States on recycled paper at R.R. Donnelley in Crawfordsville, Indiana.

First printing: May 2013

Editor-in-Chief
Mark Taub

Acquisitions Editor
Laura Lewin

Development Editor
Songlin Qiu

Managing Editor
Kristy Hart

Project Editor
Betsy Gratner

Copy Editor
Bart Reed

Indexer
Lisa Stumpf

Proofreader
Jess DeGabriele

Technical Reviewers
Joseph Annuzzi
Cameron Banga
Joshua Jamison

Editorial Assistant
Olivia Basegio

Cover Designer
Chuti Prasertsith

Compositor
Nonie Ratcliff

Dedicated to my family

Contents

This page intentionally left blank

PREFACE

Whether you have been working with the Android SDK since before the first device was released in September of 2008 or you just finished your first "Hello, World" app, you are likely aware of the incredible pace at which Android has been developed. The operating system itself has changed and matured, and the apps have followed suit. That means it is more challenging than ever to stand out. It's no longer enough to create a functional user interface that's "good enough." Now there is enough competition that apps with poor UI and apps that are half-hearted ports from other operating systems are outright rejected by users. Google has shown their commitment to design with the major UI and usability fixes in Android 4.0, Ice Cream Sandwich, and users have learned to expect more from their devices and the apps they download. With the additional work of "Project Butter" in Android 4.1 and continued improvements in Android 4.2, it has become more important than ever to ensure your app is smooth and efficient.

Design has many purposes, but two of the most important are usability and visual appeal. You want brand-new users to be able to jump into your app and get started without any effort because mobile users are more impatient than users of nearly any other platform. Users need to know exactly what they can interact with, and they need to be able to do so in a hurry while distracted. That also means you have to be mindful of what platform conventions are in order to take advantage of learned behavior.

You also want your app to stand out because visual appeal can get users excited about your app and can strengthen your brand. It gives a sense of quality when done right and can immediately lead to a larger user base when your users show the app off to their friends. Comparing your app to a car, you can think of design as the visual appearance and usability as the controls. There is a good bit of flexibility with the appearance of a car, limited only by practicality and the need for it to be usable to the potential owner. If you were to get into a car and not have a steering wheel, you would immediately start looking around and wonder, "How do I control this thing?" The same is true of your app. If the user opens it up and is immediately confused by the controls, your app has failed the most basic usability test.

If you have picked up this book, I probably do not need to go on and on about how important design is. You get it. You want to make the commitment of making beautiful apps.

This book primarily focuses on Android from a developer's perspective, but it also has a large amount of design sensibility built in. This is an attempt to both bridge the gap between designer and developer and to teach you how to implement great designs. We are not here to focus extensively on color theory or Photoshop techniques; we are here to understand what goes into designing an app and how to actually make that app come alive. When you are done with this book, you will be able to communicate your needs and feedback with designers and even do some design on your own.

This book will serve as a tutorial for the entire design and implementation process as well as a handy reference that you can keep using again and again. You will understand how to talk with designers and developers alike in order to make the best applications possible. You will be able to make apps that are visually appealing while still easy to change when those last-minute design requests inevitably come in.

Ultimately, designers and developers both want their apps to be amazing, and I am excited to teach you how to make that happen.

—Ian G. Clifton

ACKNOWLEDGMENTS

Despite having been writing fervently since I was barely old enough to hold a pencil, this book would not have been possible without the help of many individuals. I'd like to thank Executive Editor, Laura Lewin, who kept me on track despite the fact that I am perhaps the worst author to ever estimate chapter sizes and level of effort. Olivia Basegio, the Editorial Assistant, kept track of all the moving pieces despite that some days they moved slowly and others they moved at lightning speed. Songlin Qiu was the Development Editor for this book and managed to make sense out of my 4 a.m. draft chapter postings so that the final chapters could be much improved. I am very appreciative of all the work by the technical reviewers; having done that job myself in the past, I know how much work it can be, so thanks go out to Joseph Annuzzi, Cameron Banga, and Joshua Jamison.

Writing is certainly a full-time job, so balancing it among a full-time job at A.R.O., family (especially during the holidays), and several other projects was quite the challenge. Special thanks goes out to Andy Hickl at A.R.O. for his flexibility and support of my book, along with all my other friends and coworkers for not raising pitchforks when I had to work from home to cut out commute time for this book or even take time off to get chapters done on schedule.

Of course, I have to thank my family for their support. My parents told me I could do anything while growing up, and apparently I misunderstood that to mean I had to do everything. I have never stopped pushing myself to learn more and accomplish more than I ever thought I would, and I am grateful for their ability to believe in me.

ABOUT THE AUTHOR

Ian G. Clifton is the Director of User Experience and lead Android developer at A.R.O. in Seattle, where he develops Saga, an Android and iOS app that learns about you in order to let you live a better life with minimal interaction. He has worked with many designers in the course of his career and has developed several well-known Android apps, such as CNET News, CBS News, Survivor, Big Brother, and Rick Steves' Audio Europe.

Ian's love of technology, art, and user experience has led him along a variety of paths. Besides Android development, he has done platform, web, and desktop development. He served in the United States Air Force as a Satellite, Wideband, and Telemetry Systems Journeyman and has also created quite a bit of art with pencil, brush, and camera.

You can follow Ian G. Clifton on Twitter at http://twitter.com/IanGClifton and see his thoughts about mobile development on his blog at http://blog.iangclifton.com. He also published a video series called "The Essentials of Android Application Development," available at http://my.safaribooksonline.com/video/programming/android/9780132996594.

INTRODUCTION

Audience of This Book

This book is intended primarily for Android developers who want to better understand user interfaces in Android, but it also has a strong design focus, so designers can benefit from it as well. In order to focus on the important topics of Android user interface design, this book makes the assumption that you already have a basic understanding of Android. For example, if you're looking to learn about the development side, this book makes the assumption that you've at least made a "Hello, World" Android app and don't need help setting up your computer for development (if that's not the case, the Android developer site is a good place to start: http://developer.android.com/training/basics/firstapp/index.html). If you're a designer, you may find some of the code examples intimidating, but the book is written to give enough information to be useful for designers as well. For example, Chapter 13, "Working with the Canvas and Advanced Drawing," covers detailed examples of concepts such as PorterDuff compositing. Although most designers haven't heard of these concepts and don't care about the mathematical implementations, they have usually encountered them in other software such as Photoshop, where they are more simply referred to as blending modes (for example, "multiply" and "lighten"). By looking at the sample images and the intro details, designers can understand the capabilities of Android and point developers to the specific details.

Organization of This Book

This book is organized into four parts. Part I, "The Basics of Android User Interface," provides an overview of the Android UI and trends before diving into the specific classes used to create an interface in Android. It also covers the use of graphics and resources. Part II, "The Full Design and Development Process," mirrors the stages of app development, starting with just ideas and goals, working through wireframes and prototypes, and developing complete apps that include efficient layouts, animations, and more. Part III, "Advanced Topics for Android User Interfaces," explores making apps more useful by creating automatically updating ListViews, custom components that combine views, fully custom views, and even advanced techniques such as image compositing. Finally, Part IV, "Helpful Guides and Reference," consists of three appendixes: one that covers Google Play assets, one that covers Amazon Appstore assets, and one that covers a variety of common UI-related tasks that are good to know but don't necessarily fit elsewhere (such as how to dim the onscreen navigation elements).

The emphasis throughout is on implementation in simple and clear ways. You do not have to worry about pounding your head against complex topics such as 3D matrix transformations in OpenGL; instead, you will learn how to create smooth animations, add PorterDuff compositing into your custom views, and efficiently work with touch events. The little math involved will be broken down, making it so simple that you barely realize any math is involved. In addition, illustrations will make even the most complex examples clear, and every example will be practical.

How to Use This Book

This book starts with a very broad overview before going into more specific and more advanced topics. As such, it is intended to be read in order, but it is also organized to make reference as easy as possible. Even if you're an advanced developer, it is a good idea to read through all the chapters because of the wide range of material covered; however, you can also jump directly to the topics that most interest you. For example, if you really want to focus on creating your own custom views, you can jump right to Chapter 12, "Developing Fully Custom Views."

This Book's Website

You can find the source code for the examples used throughout this book at http://auidbook.com and the publisher's website at www.informit.com/title/9780321886736. From there, you can clone the entire repository, download a full ZIP file, and browse through individual files.

Conventions Used in This Book

This book uses typical conventions found in most programming-related books. Code terms such as class names or keywords appear in `monospace font`. When a class is being referred to specifically (for example, "Your class should extend the `View` class"), then it will be in monospace font. If it's used more generally (for example, "When developing a view, don't forget to test on a real device"), then it will not be in a special font.

Occasionally when a line of code is too long to fit on a printed line in the book, a code-continuation arrow (➥) is used to mark the continuation.

note

Notes look like this and are intended to supplement the material in the book with other information you may find useful.

tip

Tips look like this and give you advice on specific topics.

warning

Warnings look like this and are meant to bring to your attention potential issues you may run into or things you should look out for.

This page intentionally left blank

CHAPTER 10

HOW TO HANDLE COMMON COMPONENTS

A lot of different UI components are common to apps. Splash screens and loading indicators, for example, are very common but have several different implementations. There are times when you need to include complex styling for text or even inline images, but do not want to create several views. You may want to improve the user experience by loading content just before the user needs it. In this chapter, you will learn about these common app components and the best way to develop them.

Splash Screen

Splash screens are typically still images that fill the screen of a mobile device. On a desktop computer, they can be full screen (such as for an operating system) or a portion of the screen (for example, Photoshop, Eclipse, and so on). They give feedback that the system has responded to the user's action of opening the application. Splash screens can include loading indicators but are often static images.

Do You Really Need It?

Looking back at the previous examples (operating systems, Photoshop, and Eclipse), you should notice something in common with desktop uses of splash screens: The applications are large. They need to show the user something while they are loading so that the user will not feel like the computer has locked up or failed to respond to the user's actions.

Compared to massive applications such as Photoshop, your Android app is very small and likely loading from flash storage rather than a slower, spinning disk. Many people also have the mistaken conception that because iOS apps require splash screens that Android apps do, too. Related, when an app already exists on iOS and is being built for Android afterward, many people think that it should have a splash screen in both places to be consistent, but you should play to the strengths of each platform and take advantage of how fast Android apps load. So, does your app really need a splash screen?

The correct answer to start with is no, until you have proven it a necessity. Splash screens are often abused as a way of getting branding in front of the user, but they should only be used if loading is going on in the background. An app that artificially displays a splash screen for a few seconds is preventing the user from actually using the app, and that is the whole reason the user has the app in the first place. Mobile apps are especially designed for quick, short uses. You might have someone pushing for more heavy-handed branding, but that can ultimately hurt the app and the brand itself if users find they have two wait a few extra seconds when they open the app but they don't have to wait with a competitor's app.

That said, there are genuine times when a splash screen is needed. If you cannot show any UI until some loading takes place that could take a while, then it makes sense to show a splash screen. A good example of this is a game running in OpenGL that needs to load several textures into memory, not to mention sound files and other resources. Another example is an app that has to load data from the Web. For the first run, the app might not have any content to display, so it can show a splash screen while the web request is made, making sure to give an indication of progress. For subsequent loads, the cached data can be displayed while the request is made. If that cached data takes any significant amount of time, you might opt to show the splash screen there too until the cached content has loaded, then have a smaller loading indicator that shows the web request is still in progress. If you want to display a splash screen because your layout takes a long time to display, you should first consider making your layout more efficient.

Keep in mind that the user might want to take an action immediately and does not care about the main content. You would not want the Google Play app to show a splash screen for five seconds because it is loading the top content when you are actually just opening it to search for a specific app.

Ultimately, you should opt to skip on the splash screen unless you have developed the app and it is absolutely necessary. When in doubt, it is not needed.

Using a `Fragment` for a Splash Screen

If you have determined that your app is one of the few that does truly require a splash screen, one approach for displaying it is to use a `Fragment`. You immediately show it in your `onCreate(Bundle)` method, start your loading on a background thread, and replace the `Fragment` with your actual UI `Fragment` when your loading has completed. That sounds easy enough, but what does it really look like?

First, create the `Fragment` that you will use for the splash screen. You'll probably have some kind of branded background, but this example will just use a simple XML-defined gradient for the background (see Listing 10.1).

Listing 10.1 A Simple Gradient Saved as `splash_screen_bg.xml`

```xml
<?xml version="1.0" encoding="utf-8"?>
<shape xmlns:android="http://schemas.android.com/apk/res/android"
    android:shape="rectangle" >

    <gradient
        android:angle="90"
        android:centerColor="#FF223333"
        android:endColor="@android:color/black"
        android:startColor="@android:color/black" />

</shape>
```

Next, create a layout for the splash screen. This example will use a really simple layout that just shows the text "Loading" and a `ProgressBar`. Notice that the style is specifically set on the `ProgressBar` to make this a horizontal indicator (instead of an indeterminate indicator) in Listing 10.2.

Listing 10.2 The Splash Screen Layout

```xml
<?xml version="1.0" encoding="utf-8"?>
<LinearLayout xmlns:android="http://schemas.android.com/apk/res/
➥android"
    android:layout_width="match_parent"
```

```
        android:layout_height="match_parent"
        android:background="@drawable/splash_screen_bg"
        android:gravity="center"
        android:orientation="vertical" >

        <TextView
            android:layout_width="wrap_content"
            android:layout_height="wrap_content"
            android:text="@string/loading"
            android:textAppearance="@android:style/
➥TextAppearance.Medium.Inverse" />

        <ProgressBar
            android:id="@+id/progress_bar"
            style="?android:attr/progressBarStyleHorizontal"
            android:layout_width="200dp"
            android:layout_height="wrap_content"
            android:max="100" />

</LinearLayout>
```

The last piece of the splash screen portion is to create a simple `Fragment` that displays that layout. The one requirement of this `Fragment` is that it has a way of updating the `ProgressBar`. In this example, a simple `setProgress(int)` method is tied directly to the `ProgressBar` in the layout. See Listing 10.3 for the full class and Figure 10.1 for what this will look like in use.

Listing 10.3 The Fragment That Displays the Splash Screen

```
public class SplashScreenFragment extends Fragment {

    private ProgressBar mProgressBar;

    @Override
    public View onCreateView(LayoutInflater inflater, ViewGroup
➥container, Bundle savedInstanceState) {
        final View view = inflater.inflate(R.layout.splash_screen,
➥container, false);
        mProgressBar = (ProgressBar) view.findViewById
➥(R.id.progress_bar);
        return view;
    }

    /**
     * Sets the progress of the ProgressBar
     *
     * @param progress int the new progress between 0 and 100
     */
    public void setProgress(int progress) {
        mProgressBar.setProgress(progress);
    }
}
```

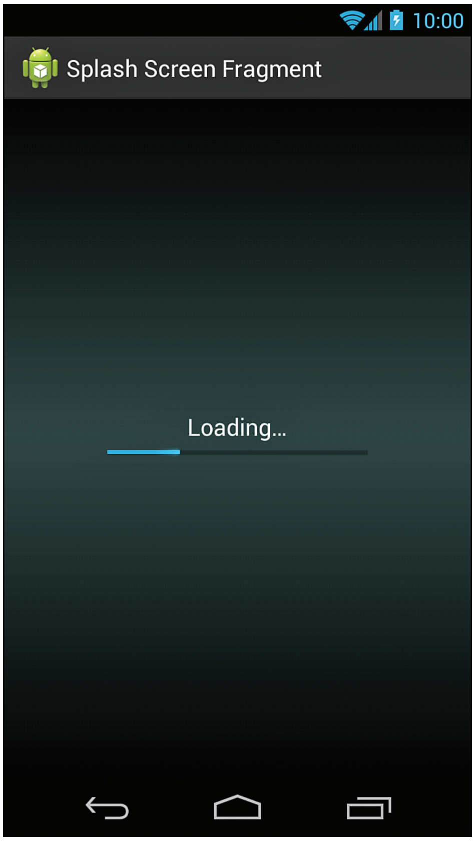


Figure 10.1 The simple splash screen with loading indicator

Now that the easy part is done, you need to handle loading the data. Prior to the introduction of the `Fragment` class, you would do this with a simple `AsyncTask` in your `Activity` that you would save and restore during config changes (attaching and detaching the `Activity` to avoid leaking the `Context`). This is actually easier now with a `Fragment`. All you have to do is create a `Fragment` that lives outside of the `Activity` lifecycle and handles the loading. The `Fragment` does not have any UI because its sole purpose is to manage loading the data.

Take a look at Listing 10.4 for a sample `Fragment` that loads data. First, it defines a public interface that can be used by other classes to be notified of progress with loading the data. When the `Fragment` is attached, it calls `setRetainInstance(true)` so that the `Fragment` is kept across configuration changes. When the user rotates the device, slides out a keyboard, or otherwise affects the device's configuration, the `Activity` will be re-created but this `Fragment` will continue to exist. It has simple methods to check if loading is complete and to get the result of loading the data (which is stored as a `Double` for this example, but it could be anything for your case). It can also set and remove the `ProgressListener` that is notified of updates to the loading.

The `Fragment` contains an `AsyncTask` that does all the hard work. In this case, the background method is combining some arbitrary square roots (to mimic real work) and causing the `Thread` to sleep for 50ms per iteration to create a delay similar to what you might see with real use. This is where you would grab assets from the Web, load them from the disk, parse a complex data structure, or do whatever you needed to finish before the app is ready.

On completion, the `AsyncTask` stores the result, removes its own reference, and notifies the `ProgressListener` (if one is available).

Listing 10.4 A Fragment That Handles Loading Data Asynchronously

```
public class DataLoaderFragment extends Fragment {

    /**
     * Classes wishing to be notified of loading progress/completion
     * implement this.
     */
    public interface ProgressListener {
        /**
         * Notifies that the task has completed
         *
         * @param result Double result of the task
         */
        public void onCompletion(Double result);

        /**
         * Notifies of progress
         *
         * @param value int value from 0-100
         */
        public void onProgressUpdate(int value);
    }

    private ProgressListener mProgressListener;
    private Double mResult = Double.NaN;
    private LoadingTask mTask;

    @Override
    public void onAttach(Activity activity) {
        super.onAttach(activity);
        // Keep this Fragment around even during config changes
        setRetainInstance(true);
    }

    /**
     * Returns the result or {@value Double#NaN}
     *
     * @return the result or {@value Double#NaN}
     */
    public Double getResult() {
        return mResult;
    }

    /**
     * Returns true if a result has already been calculated
     *
     * @return true if a result has already been calculated
     * @see #getResult()
     */
    public boolean hasResult() {
        return !Double.isNaN(mResult);
    }

    /**
     * Removes the ProgressListener
     *
     * @see #setProgressListener(ProgressListener)
     */
    public void removeProgressListener() {
        mProgressListener = null;
    }

    /**
     * Sets the ProgressListener to be notified of updates
     *
     * @param listener ProgressListener to notify
     * @see #removeProgressListener()
     */
    public void setProgressListener(ProgressListener listener) {
        mProgressListener = listener;
    }

    /**
     * Starts loading the data
     */
    public void startLoading() {
        mTask = new LoadingTask();
        mTask.execute();
    }
    private class LoadingTask extends AsyncTask<Void, Integer,
➥Double>
{

        @Override
        protected Double doInBackground(Void... params) {
            double result = 0;
            for (int i = 0; i < 100; i++) {
                try {
                    result += Math.sqrt(i);
                    Thread.sleep(50);
                    this.publishProgress(i);
                } catch (InterruptedException e) {
                    return null;
                }
            }
            return Double.valueOf(result);
        }

        @Override
        protected void onPostExecute(Double result) {
            mResult = result;
            mTask = null;
            if (mProgressListener != null) {
                mProgressListener.onCompletion(mResult);
            }
        }

        @Override
        protected void onProgressUpdate(Integer... values) {
            if (mProgressListener != null) {
                mProgressListener.onProgressUpdate(values[0]);
            }
        }
    }
}
```

To tie everything together, you need an `Activity`. Listing 10.5 shows an example of such an `Activity`. It implements the `ProgressListener` from the `DataLoaderFragment`. When the data is done loading, the `Activity` simply displays a `TextView` with the result (that's where you would show your actual app with the necessary data loaded in). When notified of updates to loading progress, the `Activity` passes those on to the `SplashScreenFragment` to update the `ProgressBar`.

In `onCreate(Bundle)`, the `Activity` checks whether or not the `DataLoaderFragment` exists by using a defined tag. If this is the first run, it won't exist, so the `DataLoaderFragment` is instantiated, the `ProgressListener` is set to the `Activity`, the `DataLoaderFragment` starts loading, and the `FragmentManager` commits a `FragmentTransaction` to add the

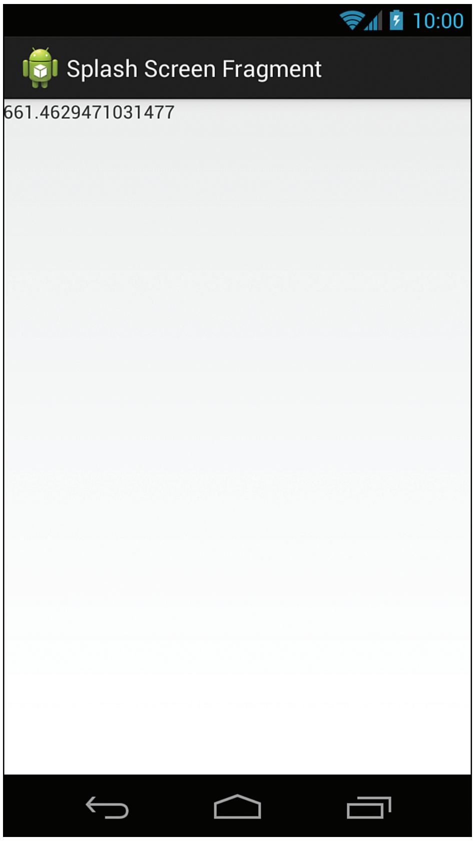
DataLoaderFragment (so it can be recovered later). If the user has rotated the device, the DataLoaderFragment will be found, so the app has to check whether or not the data has already loaded. If it has, the method is done; otherwise, everything falls through to checking if the SplashScreenFragment has been instantiated, creating it if it hasn't.

The onStop() method removes the Activity from the DataLoaderFragment so that your Fragment does not retain a Context reference and the app avoids handling the data result if it's not in the foreground. Similarly, onStart() checks if the data has been successfully loaded.

The last method, checkCompletionStatus(), checks if the data has been loaded. If it has, it will trigger onCompletion(Double) and remove the reference to the DataLoader Fragment. By removing the reference, the Activity is able to ensure that the result is only handled once (which is why onStart() checks if there is a reference to the DataLoaderFragment before handling the result). Figure 10.2 shows what the app looks like once it has finished loading the data.

Listing 10.5 The Activity That Ties Everything Together

```
public class MainActivity extends Activity implements
➥ProgressListener {

    private static final String TAG_DATA_LOADER = "dataLoader";
    private static final String TAG_SPLASH_SCREEN = "splashScreen";

    private DataLoaderFragment mDataLoaderFragment;
    private SplashScreenFragment mSplashScreenFragment;

    @Override
    public void onCompletion(Double result) {
        // For the sake of brevity, we just show a TextView with the
➥result
        TextView tv = new TextView(this);
        tv.setText(String.valueOf(result));
        setContentView(tv);
        mDataLoaderFragment = null;
    }

    @Override
    public void onProgressUpdate(int progress) {
        mSplashScreenFragment.setProgress(progress);
    }

    @Override
    protected void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);

        final FragmentManager fm = getFragmentManager();
        mDataLoaderFragment = (DataLoaderFragment)
➥fm.findFragmentByTag(TAG_DATA_LOADER);
        if (mDataLoaderFragment == null) {
            mDataLoaderFragment = new DataLoaderFragment();
            mDataLoaderFragment.setProgressListener(this);
            mDataLoaderFragment.startLoading();
            fm.beginTransaction().add(mDataLoaderFragment,
➥TAG_DATA_LOADER).commit();
        } else {
            if (checkCompletionStatus()) {
                return;
            }
        }

        // Show loading fragment
        mSplashScreenFragment = (SplashScreenFragment)
➥fm.findFragmentByTag(TAG_SPLASH_SCREEN);
        if (mSplashScreenFragment == null) {
            mSplashScreenFragment = new SplashScreenFragment();
            fm.beginTransaction().add(android.R.id.content,
➥]mSplashScreenFragment, TAG_SPLASH_SCREEN).commit();
        }
    }

    @Override
    protected void onStart() {
        super.onStart();
        if (mDataLoaderFragment != null) {
            checkCompletionStatus();
        }
    }

    @Override
    protected void onStop() {
        super.onStop();
        if (mDataLoaderFragment != null) {
            mDataLoaderFragment.removeProgressListener();
        }
    }

    /**
     * Checks if data is done loading, if it is, the result is handled
     *
     * @return true if data is done loading
     */
    private boolean checkCompletionStatus() {
        if (mDataLoaderFragment.hasResult()) {
                onCompletion(mDataLoaderFragment.getResult());
                FragmentManager fm = getFragmentManager();
                mSplashScreenFragment = (SplashScreenFragment)
➥fm.findFragmentByTag(TAG_SPLASH_SCREEN);
                if (mSplashScreenFragment != null) {
                    fm.beginTransaction().remove(mSplashScreenFragment).
➥commit();
                }
                return true;
            }
            mDataLoaderFragment.setProgressListener(this);
            return false;
        }
    }
```



```
661.4629471031477
```

Figure 10.2 The resulting app after data has loaded

Quite a bit is going on in this small bit of code, so it's a good idea to review it. On a high level, you are using `DataLoaderFragment` to load all the data, and it exists outside of configuration changes. The `Activity` checks `DataLoaderFragment` each time it is created and started to handle the result. If it's not done yet, the `SplashScreenFragment` is shown to indicate progress.

Loading Indication

Immediate feedback is one of the most important parts of a good UI. If a button does not have a touch state and the resulting action takes some time, the user will feel like the app is unresponsive. Unfortunately, whether the app needs to run complex image analysis algorithms or just access web resources, there are times when it will not be able to immediately show the users what they want to see. In these instances, you use a loading indicator to give the user a sense that something is happening. Ideally, you use a loading indicator that can show progress such as when downloading a file, but sometimes you have to fall back on the indeterminate loading indicator, which just tells the user, "Hey, something is happening, but who knows how long it will take."

Dialogs versus Inline

Using dialogs to indicate loading is the go-to solution for a lot of developers. In fact, Android's `ProgressDialog` class makes this extremely easy. Just create an instance using one of the static `show()` methods and then update it if possible. When your task is done, you just `dismiss()` the dialog. Simple enough, right?

The problem is that these dialogs are modal. That means the user can do nothing else in your app while looking at one of these dialogs, so they don't make sense unless there really is nothing else the user can do (for example, the previously explained splash screen). You may have allowed the user to back out of the dialog, but that just results in the user being confused as to whether the task actually stopped or not (and further confused when the UI suddenly changes when it does complete). Instead, consider using inline loading indicators.

An inline loading indicator is basically a loading indicator that is a part of your regular view hierarchy. It goes where the content that is loading will go and serves as a placeholder as well as a visual indication of activity. Not only is this significantly less disruptive than a dialog, it lets the user interact with other content immediately. If you go to Google Play to search for a particular app, you don't want to wait while the front page loads before you can actually search. An additional advantage of inline indicators is that they allow you to load different sections of a screen and display them independently. You might go to someone's profile page in an app and see the basic info. At the same time, one section is loading that displays recent content posted by that person and another section loads people who are similar to that person. Neither of these pieces is dependent on the other.

Using an Inline Loading Indicator

Using inline loading indicators in your app is actually extremely easy. The simplest way is to just include a `ProgressBar` in your layout somewhere and then hide or remove it when the loading is complete and add the new views. There are several ways to make this easier to manage. If the extra content is almost always available (for example, you might go from a list of articles to a detailed article page, and you just need to fetch the body text), then an easy approach is to use a `ViewSwitcher`. A `ViewSwitcher` is a `ViewGroup` that contains two child `Views` and can animate between them. In this case, you use it to display a loading indicator and then switch to the other `View` when it is ready.

First, define a couple of animations in XML. These go in `res/anim`. Listing 10.6 defines a simple fade-in animation, and Listing 10.7 defines a fade-out animation.

Listing 10.6 A Simple Fade-in Animation Saved as `fade_in.xml`

```
<?xml version="1.0" encoding="utf-8"?>
<alpha xmlns:android="http://schemas.android.com/apk/res/android"
    android:duration="300"
    android:fromAlpha="0.0"
    android:interpolator="@android:anim/decelerate_interpolator"
    android:toAlpha="1.0" />
```

Listing 10.7 A Simple Fade-out Animation Saved as `fade_out.xml`

```
<?xml version="1.0" encoding="utf-8"?>
<alpha xmlns:android="http://schemas.android.com/apk/res/android"
    android:duration="300"
    android:fromAlpha="1.0"
    android:interpolator="@android:anim/accelerate_interpolator"
    android:toAlpha="0.0" />
```

With those animations defined, all you need now is a `ViewSwitcher` that is displaying a loading indication and a second child `View` that is the content you have finished loading. You set the animations and then simply call `showNext()`, as shown in Listing 10.8.

Listing 10.8 Using a `ViewSwitcher` to Animate Between Views

```
ViewSwitcher viewSwitcher = (ViewSwitcher) findViewById
➥(R.id.view_switcher);
viewSwitcher.setInAnimation(this, R.anim.fade_in);
viewSwitcher.setOutAnimation(this, R.anim.fade_out);
viewSwitcher.showNext();
```

Sometimes you'll have some chunk of content that is frequently not there or that has a complex view hierarchy. In these cases, it's a good idea to make use of `ViewStub`s. A `ViewStub` is an extremely simple implementation of `View` that essentially acts as a placeholder for other content. It takes up no space and draws nothing, so it has minimal impact on your layout complexity. Think of it like an `include` tag that does not actually include another layout until you say to do so.

Listing 10.9 shows what a `ViewStub` will look like in your XML layout. The regular ID is used for finding the `ViewStub`, but it can also specify an `inflatedID` for finding the layout after it has been inflated. The layout that will be inflated is specified by the `layout` property the same as it is in an `include` tag.

Listing 10.9 An XML `ViewStub`

```
<ViewStub
    android:id="@+id/view_stub"
    android:layout_width="match_parent"
    android:layout_height="wrap_content"
    android:inflatedId="@+id/dynamic_content"
    android:layout="@layout/other_layout" />
```

All that is left to do is to find a reference to your `ViewStub`, inflate it, and do whatever you need to with the resulting layout. See Listing 10.10 for the basic code involved.

Listing 10.10 Inflating a `ViewStub` in Java

```
ViewStub stub = (ViewStub) findViewById(R.id.view_stub);
View otherLayout = stub.inflate();
// Do something with otherLayout...
```

Complex `TextViews`

`TextView`s in Android are extremely powerful. Obviously, they're able to display text, but they can also display several styles of text, different fonts or colors, and even inline images, all within a single `TextView`. You can have specific portions of text respond to click events and really associate any object you want with any portion of text. These ranges of text are generically referred to as "spans," as in a span (range) of bold text or a span of subscript.

Existing Spans

Android has a large number of prebuilt spans you can take advantage of. Because you can assign any object as a span, there isn't an actual span class. That's great in that it gives you a huge amount of flexibility, but it also means you have to dig a little to figure out what is supported.

First, you should know about the two main types of spans: `CharacterStyle` and `ParagraphStyle`. As you can probably guess, these interfaces refer to spans that affect one or more characters and spans that affect entire paragraphs. Most spans will implement one of these two interfaces (although many implement more than just these). See the following list of built-in spans to get an idea about what is already supported:

`AbsoluteSizeSpan`—A span that allows you to specify an exact size in pixels or density independent pixels.

`AlignmentSpan.Standard`—A span that attaches an alignment (from `Layout.Alignment`).

`BackgroundColorSpan`—A span that specifies a background color (the color behind the text, such as for highlighting).

`ClickableSpan`—A span that has an `onClick` method that is triggered. (This class is abstract, so you can extend it with a class that specifies the `onClick` behavior.)

`DrawableMarginSpan`—A span that draws a `Drawable` plus the specified amount of spacing.

`DynamicDrawableSpan`—A span that you can extend to provide a `Drawable` that may change (but the size must remain the same).

`EasyEditSpan`—A span that just marks some text so that the `TextView` can easily delete it.

`ForegroundColorSpan`—A span that changes the color of the text (basically just called `setColor(int)` on the `TextPaint` object).

`IconMarginSpan`—A span that draws a `Bitmap` plus the specified amount of spacing.

`ImageSpan`—A span that draws an image specified as a `Bitmap`, `Drawable`, `URI`, or resource ID.

`LeadingMarginSpan.Standard`—A span that adjusts the margin.

`LocaleSpan`—A span that changes the locale of text (available in API level 17 and above).

`MaskFilterSpan`—A span that sets the `MaskFilter` of the `TextPaint` (such as for blurring or embossing).

`MetricAffectingSpan`—A span that affects the height and/or width of characters (this is an abstract class).

`QuoteSpan`—A span that puts a vertical line to the left of the selected text to indicate it is a quote; by default the line is blue.

`RasterizerSpan`—A span that sets the `Rasterizer` of the `TextPaint` (generally not useful to you).

RelativeSizeSpan—A span that changes the text size relative to the supplied float (for instance, setting a 0.5 float will cause the text to render at half size).

ReplacementSpan—A span that can be extended when something custom is drawn in place of the spanned text (for example, ImageSpan extends this).

ScaleXSpan—A span that provides a multiplier to use when calling the `TextPaint`'s `setTextScaleX(float)` method. (In other words, setting this to 0.5 will cause the text to be scaled to half size along the x axis, thus appearing squished.)

StrikethroughSpan—A span that simply passes `true` to the `TextPaint`'s `setStrikeThruText(boolean)` method, causing the text to have a line through it (useful for showing deleted text, such as in a draft of a document).

StyleSpan—A span that adds bold and/or italic to the text.

SubscriptSpan—A span that makes the text subscript (below the baseline).

SuggestionSpan—A span that holds possible replacement suggestions, such as for a incorrectly spelled word (available in API level 14 and above).

SuperscriptSpan—A span that makes the text superscript (above the baseline).

TabStopSpan.Standard—A span that allows you to specify an offset from the leading margin of a line.

TextAppearanceSpan—A span that allows you to pass in a `TextAppearance` for styling.

TypefaceSpan—A span that uses a specific typeface family (`monospace`, `serif`, or `sans-serif` only).

UnderlineSpan—A span that underlines the text.

URLSpan—A `ClickableSpan` that attempts to view the specified URL when clicked.

Using Spans for Complex Text

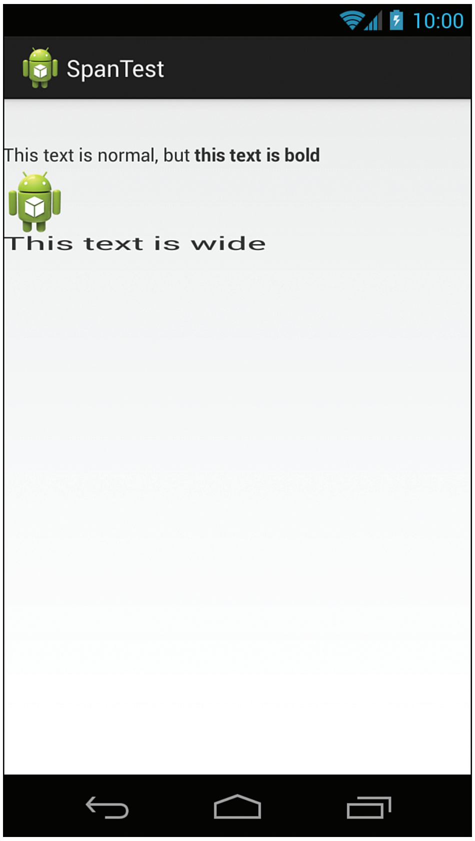
One of the simplest ways to use spans is with the `HTML` class. If you have some HTML in a string, you can simply call `HTML.fromHtml(String)` to get an object that implements the `spanned` interface that will have the applicable spans applied. You can even supply an `ImageGetter` and a `TagHandler`, if you'd like. The styles included in the HTML will be converted to spans so, for example, "b" (bold) tags are converted to `StyleSpan`s and "u" (underline) tags are converted to `UnderlineSpan`s. See Listing 10.11 for a brief example of how to set the text of a `TextView` from an HTML string and enable navigating through and clicking the links.

Listing 10.11 Using HTML in a `TextView`

```
textView.setText(Html.fromHtml(htmlString));
textView.setMovementMethod(LinkMovementMethod.getInstance());
textView.setLinksClickable(true);
```

Another easy method for implementing spans is to use the `Linkify` class. The `Linkify` class allows you to easily create links within text for web pages, phone numbers, email addresses, physical addresses, and so on. You can even use it for custom regular expressions, if you're so inclined.

Finally, you can also manually set spans on anything that implements the `Spannable` interface. If you have an existing `String` or `CharSequence` that you'd like to make `Spannable`, use the `SpannableString` class. If you are building up some text, you can use the `SpannableStringBuilder`, which works like a `StringBuilder` but can attach spans. To the untrained eye, the app in Figure 10.3 is using two `TextView`s and an `ImageView`, but it actually has just a single `TextView`. See Listing 10.12 to understand how you can do this with one `TextView` and a few spans.



This text is normal, but **this text is bold**

This text is wide

Figure 10.3 An app that seemingly uses more views than it really does

Listing 10.12 Using Spans with a `SpannableStringBuilder`

```
public class MainActivity extends Activity {

    @Override
    protected void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);

        final SpannableStringBuilder ssb = new
➥SpannableStringBuilder();
        final int flag = Spannable.SPAN_EXCLUSIVE_EXCLUSIVE;
        int start;
        int end;

        // Regular text
        ssb.append("This text is normal, but ");

        // Bold text
        start = ssb.length();
        ssb.append("this text is bold");
        end = ssb.length();
        ssb.setSpan(new StyleSpan(Typeface.BOLD), start, end, flag);

        // Inline image
        ssb.append('\n');
        start = end++;
        ssb.append('\uFFFC'); // Unicode replacement character
        end = ssb.length();
        ssb.setSpan(new ImageSpan(this, R.drawable.ic_launcher), start,
➥end, flag);

        // Stretched text
        ssb.append('\n');
        start = end++;
        ssb.append("This text is wide");
        end = ssb.length();
        ssb.setSpan(new ScaleXSpan(2f), start, end, flag);

        // Assign to TextView
        final TextView tv = new TextView(this);
        tv.setText(ssb);
        setContentView(tv);
    }
}
```

Autoloading `ListView`s

Using a `ListView` is a great way to display an extensive data set. For instance, you might show a list of news articles. For this to load quickly, you only request the first 10 or 20 news articles to show in the list; however, the user may want to see more. Older user experiences would have a button at the end of the list that the user could tap that would begin loading the next set. That works, but you can do better.

Concept and Reasoning

To improve the user experience, it's a good idea to anticipate the user's actions. Scrolling through a list is a great example where you can easily make reasonable assumptions. If the user has scrolled to the bottom of the list, there is a good chance that the user wants to continue scrolling. Instead of waiting for the user to press a button, you can immediately begin loading more items. So, now you've managed to save the user a little bit of work, but you can go a step further.

When the user has scrolled to near the bottom of the list, you can start loading more. Say you display 20 items and the user has scrolled down to where the device is showing items 12 through 17. The user is very close to the bottom of the list, so you can begin loading more items ahead of time. If your data source and connection are fast, the next 10 or 20 items can be loaded by the time the user gets to the bottom of the list. Now the user can scroll without effort through a large data set, and you can still have the benefits of loading a smaller set of data to speed up the initial user experience.

Autoloading Near the Bottom of a List

Although you are going to start loading before getting to the bottom of a list, you will want to have a loading indicator at the bottom because the user can get there before the loading completes. So, to start, create a simple loading view called `loading_view.xml` like the one in Listing 10.13.

Listing 10.13 A Simple Loading Layout

```
<?xml version="1.0" encoding="utf-8"?>
<LinearLayout xmlns:android="http://schemas.android.com/apk/res/
➥android"
    android:layout_width="match_parent"
    android:layout_height="wrap_content"
    android:gravity="center"
    android:orientation="horizontal" >
    <ProgressBar
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:layout_marginRight="10dp" />

    <TextView
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:text="@string/loading"
        android:textAppearance="?android:attr/textAppearanceMedium" />

</LinearLayout>
```

Next, create an `Adapter` that will provide fake data to test out this technique. Listing 10.14 shows an example. This sample `Adapter` just keeps a count representing the number of fake items to show. You can simulate adding more items with the `addMoreItems(int)` method. For the purposes of testing this autoloading technique, this is adequate.

Listing 10.14 An `Adapter` That Can Mimic Real Content

```
private static class SimpleAdapter extends BaseAdapter {

    private int mCount = 20;
    private final LayoutInflater mLayoutInflater;
    private final String mPositionString;
    private final int mTextViewResourceId;

    /*package*/ SimpleAdapter(Context context, int textViewResourceId)
{
        mLayoutInflater = LayoutInflater.from(context);
        mPositionString = context.getString(R.string.position) + " ";
        mTextViewResourceId = textViewResourceId;
    }

    public void addMoreItems(int count) {
        mCount += count;
        notifyDataSetChanged();
    }

    @Override
    public int getCount() {
        return mCount;
    }

    @Override
    public String getItem(int position) {
        return mPositionString + position;
    }
    @Override
    public long getItemId(int position) {
        return position;
    }

    @Override
    public View getView(int position, View convertView, ViewGroup
➥parent) {
        final TextView tv;
        if (convertView == null) {
            tv = (TextView) mLayoutInflater.inflate
➥(mTextViewResourceId, null);
        } else {
            tv = (TextView) convertView;
        }

        tv.setText(getItem(position));
        return tv;
    }
}
```

The last piece is the actual `Fragment` that does the hard work. It is a `ListFragment` that listens to the position of the `ListView` and loads more items if necessary. The `AUTOLOAD_THRESHOLD` value determines how close to the bottom of the list the user has to be before loading. The `MAXIMUM_ITEMS` value is an arbitrary limit to the size of our list, so you can see how to handle removing the loading `View` when all data has been loaded.

The `mAddItemsRunnable` object simulates adding additional items after a delay, similarly to how you would add more items after fetching them from a data source. In `onActivityCreated(Bundle)`, a `Handler` is created (for posting the `Runnable` with a delay), the `Adapter` is created, a footer view is instantiated and added to the `ListView`, the `Adapter` is set, and the `Fragment` is added as the `OnScrollListener`. It's important that you add the footer before setting your `Adapter` because the `ListView` is actually wrapping your `Adapter` with one that supports adding header and footer `Views`.

When the user scrolls, the `Fragment` checks if data is not currently loading and if there is more data to load. If that is the case, it checks if the `Adapter` has already added at least the maximum number of items (remember, this is arbitrary to simulate having a finite data set like you would in a real use). If there is no more data, the footer is removed. If there is more data, the `Fragment` checks to see if the user has scrolled far enough to load more data and triggers the load.

The `onScrollStateChanged` method has to be implemented as part of `OnScrollListener`, but it is not needed in this code, so it does nothing.

The onStart and onStop methods handle stopping and starting the loading of data, so the load does not continue when the app is no longer visible. This is not particularly necessary in the sample code, but it is useful in the real world when you might be loading or processing a large amount of data and don't want it to be done if the user has changed apps.

For a complete implementation, see Listing 10.15.

Listing 10.15 A ListFragment That Automatically Loads More Content

```
public class AutoloadingListFragment extends ListFragment implements
➥OnScrollListener {

    private final int AUTOLOAD_THRESHOLD = 4;
    private final int MAXIMUM_ITEMS = 52;
    private SimpleAdapter mAdapter;
    private View mFooterView;
    private Handler mHandler;
    private boolean mIsLoading = false;
    private boolean mMoreDataAvailable = true;
    private boolean mWasLoading = false;

    private Runnable mAddItemsRunnable = new Runnable() {
        @Override
        public void run() {
            mAdapter.addMoreItems(10);
            mIsLoading = false;
        }
    };

    @Override
    public void onActivityCreated(Bundle savedInstanceState) {
        super.onActivityCreated(savedInstanceState);
        final Context context = getActivity();
        mHandler = new Handler();
        mAdapter = new SimpleAdapter(context,
➥android.R.layout.simple_list_item_1);
        mFooterView = LayoutInflater.from(context).inflate(
➥R.layout.loading_view, null);
        getListView().addFooterView(mFooterView, null, false);
        setListAdapter(mAdapter);
        getListView().setOnScrollListener(this);
    }

    @Override
    public void onScroll(AbsListView view, int firstVisibleItem,
            int visibleItemCount, int totalItemCount) {
```

```
        if (!mIsLoading && mMoreDataAvailable) {
            if (totalItemCount >= MAXIMUM_ITEMS) {
                mMoreDataAvailable = false;
                getListView().removeFooterView(mFooterView);
            } else if (totalItemCount - AUTOLOAD_THRESHOLD <=
➥firstVisibleItem + visibleItemCount) {
                mIsLoading = true;
                mHandler.postDelayed(mAddItemsRunnable, 1000);
            }
        }
    }

    @Override
    public void onScrollStateChanged(AbsListView view,
➥int scrollState) {
        // Ignore
    }

    @Override
    public void onStart() {
        super.onStart();
        if (mWasLoading) {
            mWasLoading = false;
            mIsLoading = true;
            mHandler.postDelayed(mAddItemsRunnable, 1000);
        }
    }

    @Override
    public void onStop() {
        super.onStop();
        mHandler.removeCallbacks(mAddItemsRunnable);
        mWasLoading = mIsLoading;
        mIsLoading = false;
    }
}
```

Now that you have it all finished, give it a try. Notice that when you are scrolling slowly, the content loads before you ever know that it was loading. If you fling to the bottom quickly, you'll have the experience shown in Figure 10.4, where the list shows the loading footer just before the content loads in. Notice, too, that your position in the list is not changed when new content loads in. The user is not disturbed even if he or she was unaware that content was loading, and the loading view being replaced by content makes it clear that something new has loaded (without the loading view, the user might think he or she is at the bottom of the list, even after new content has been loaded).

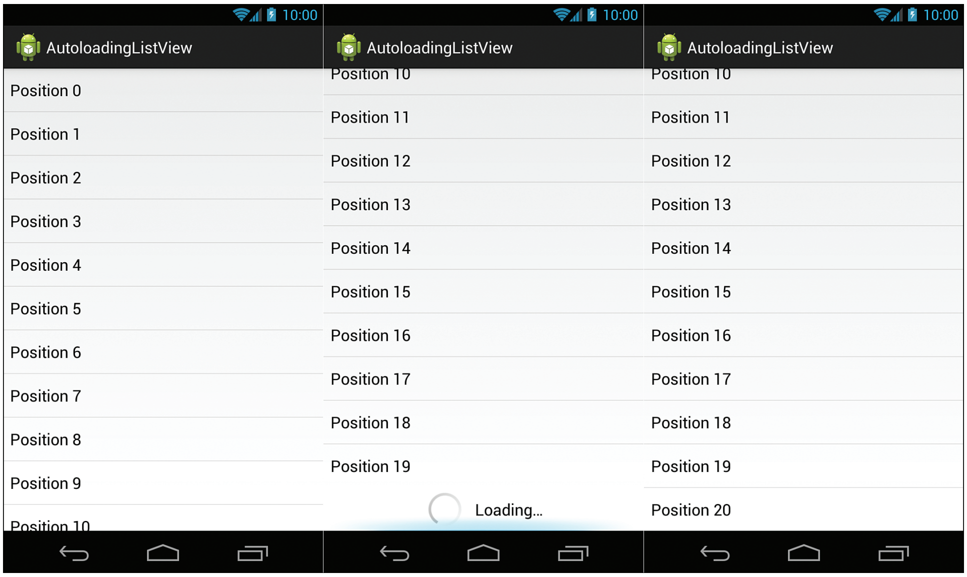


Figure 10.4 Scrolling quickly from the top (left) to the bottom (middle) results in content loading in automatically (right).

Summary

Although there are countless techniques for developing better apps, this chapter has introduced you to a few of the more common ones. By now, you should hate splash screens, but you should also know how to implement them correctly. You should be annoyed by modal loading dialogs and have experience implementing more user-friendly inline dialogs. Your experience with `TextView` spans will allow you to avoid creating several `TextViews` for minor style changes, and you should know how to implement autoloading `ListViews`.

INDEX