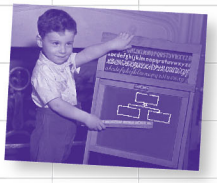
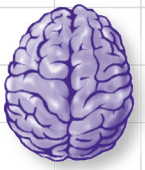
A Brain-Friendly Guide

Head First Android Development



Learn what matters, when it matters



Load important concepts directly into your brain

Bend your mind around d
puzzles an

Avoid embarrassing mistakes

Master out-of-this-world concepts

O'REILLY®

Jonathan Simon

2 working with feeds

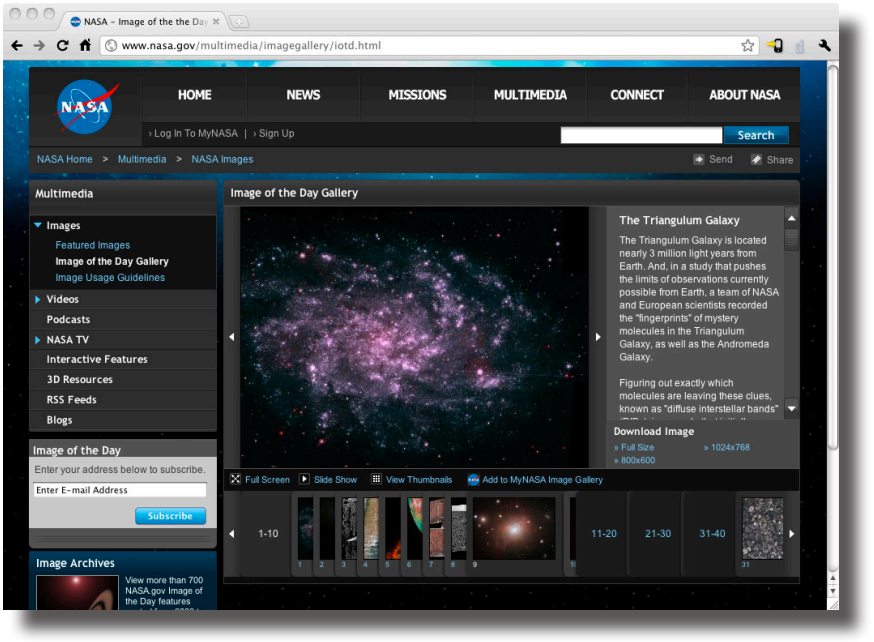
Pictures from space!

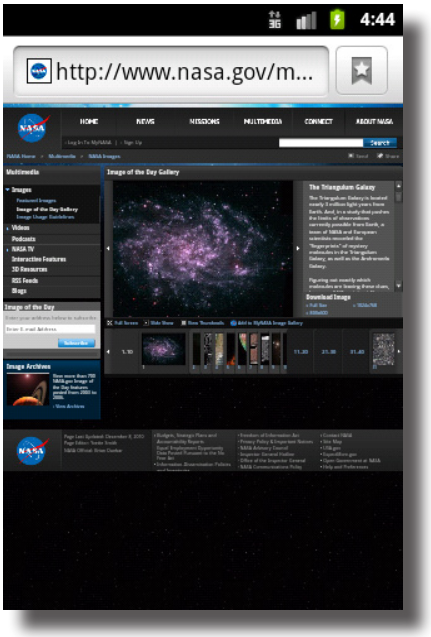
RSS feeds are everywhere! From weather and stock information to news and blogs, huge amounts of content are distributed in RSS feeds and just waiting to be used in your apps. In fact, the RSS feed publishers want you to use them! In this chapter, you'll learn how to build your own app that incorporates **content** from a public RSS feed on the Web. Along the way, you'll also learn a little more about **layouts**, **permissions,** and **debugging**.

I really wanted a telescope, but all I can find are these binoculars. Since I can't see space, I've been checking out the NASA image of the day web site instead. It's got a cool new picture of something about space every day!

Bobby

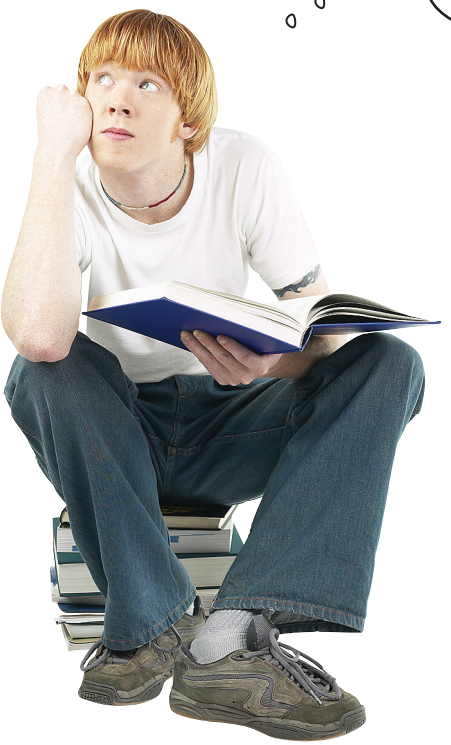
NASA's image of the day site





The image of the day site looks pretty good on a big computer, but not so hot on a phone. It technically works, but not without a ton of scrolling and zooming. There has to be something better ...

I saw an RSS feed on NASA's site. Could you use that feed and build an Android app that reads it and displays the picture? That would be way cooler than hitting the website from my phone...



Yes! We can write an app for that!

Let's put your newly developed Android skills to use and build an app that will let Bobby see the NASA daily image on his phone. He's going to love it!

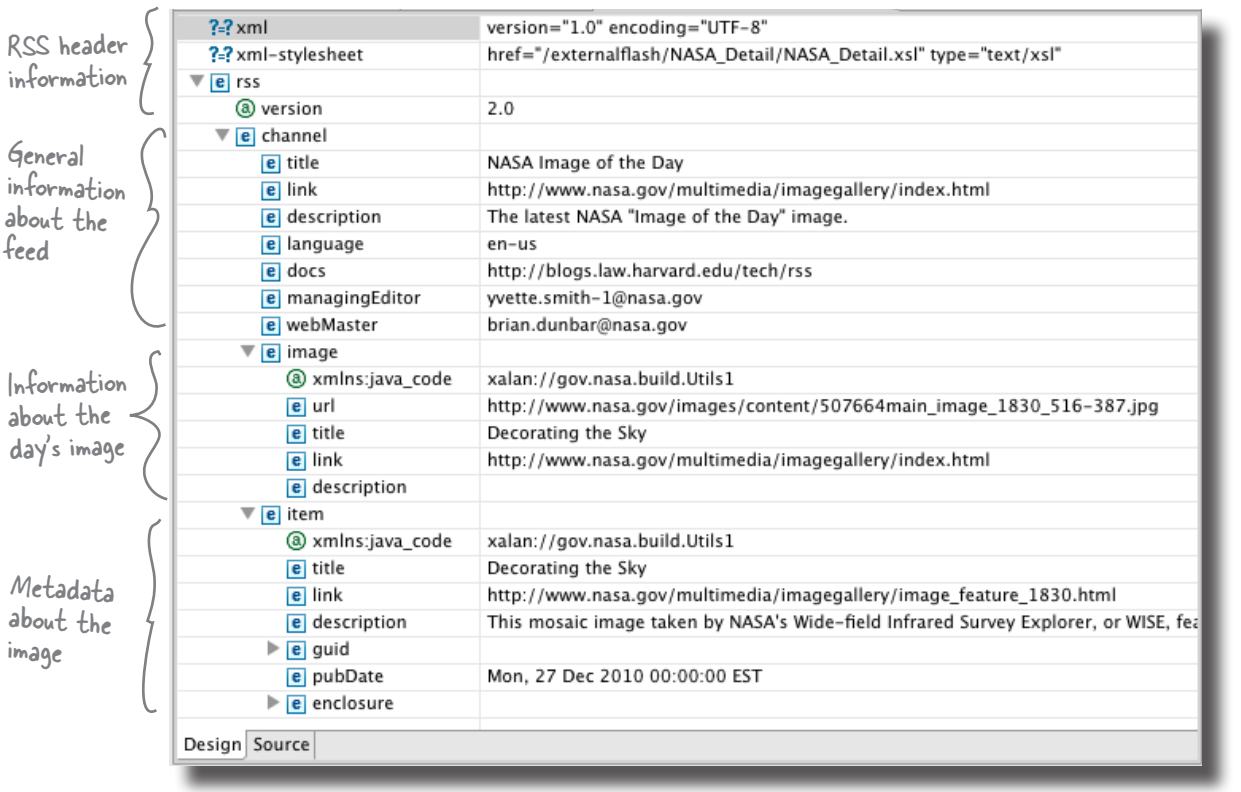
Plan out your app

Before starting on your brand-new app, take a minute to plan it out. Since you'll be building the app from he image feed from NASA, start by taking a look at the feed to get a feeling of what you have to work with.

The feed is an **RSS** (Really Simple Syndication) feed. You can find out more about RSS feeds with a quick search of the Web, but for this app, just think of it as **pure XML**.

Eclipse has a built-in XML editor that really helps to ***visualize*** the format of feeds like this. Go to **http://www.nasa.gov/rss/image_of_the_day.rss** and save the content locally on your computer as an XML file. Then you can open the XML file in Eclipse (which will automatically open the built-in XML editor) and view away!

Image of the day feed saved locally as an XML file and opened in Eclipse's XML editor



Sharpen your pencil

There's a whole bunch of stuff in that feed! If you show it all, you're going to overload your users with information and miss the point of building a specialized mobile app for viewing the image of the day. At the same time, just showing the image would be pretty boring.

Take a look at the XML view of the feed and pick a few things you think you should show. And make sure to say why you picked it. The first one is filled in for you. Add a few more on your own.

Property to include	Why include it?
image URL	I definitely want display the image, so I'll include the image URL. This is an image of the day app, after all!

Sharpen your pencil Solution

There's a whole bunch of stuff in that feed! If you show it all, you're going to overload your users with information and miss the point of building a specialized mobile app for viewing the image of the day. At the same time, just showing the image would be pretty boring.

You were to look at the XML view of the feed, pick a few things you think you should show, and make say why you picked it.

Property to include	Why include it?
image URL	I definitely want display the image, so I'll include the image URL. This is an image of the day app, after all!
image title	The image title will help users quickly tell what the image is about.
item description	If the image is cool, users will want to read more about it. This isn't the most important information, but it's great to know.
item pubDate	NASA doesn't publish a new image every day (not on weekends, for example), so it helps to know when they did publish the image being displayed.

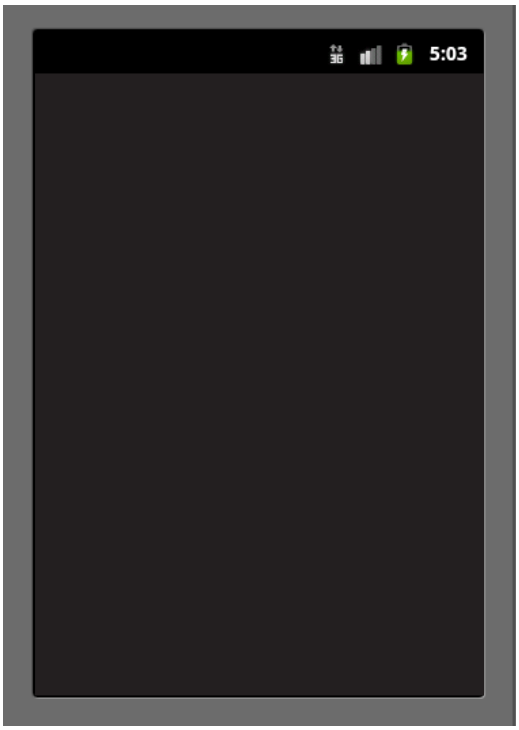
The XML feed doesn't include the binary image data. But using the image URL, you'll be able to doanload the image and diaply it on the screen.

You'll need to make sure you get the correct title and description, because the example feed contains many of each. In the example feed, the image description is blank, but the item description is populated correctly.

Your answers may be slightly different and you may have picked a different field or two (and that's perfectly OK). We'll use the properties here, but there are several other perfectly good ways you could build this app.

Screen Design Magnets

To build your interface, add the `View` magnets at the bottom of this page to the screen. There is one `View` for each of the properties you picked from the RSS feed.



Put the Views on the screen here

Image title in a TextView

Decorating the Sky

Item pubDate in a TextView

Mon, 27 Dec 2010 00:00:00 EST

Item description in a TextView

This mosaic image taken by NASA's Wide-fie
Infrared Survey Explorer, or WISE, features
nebulae that are part of the giant Orion
Molecular Cloud--the Flame nebula, the
Horsehead nebula and NGC 2023. Despite i
name, there is no fire roaring in the Flame
nebula. What makes this nebula shine is the
bright blue star seen to the right of the cen

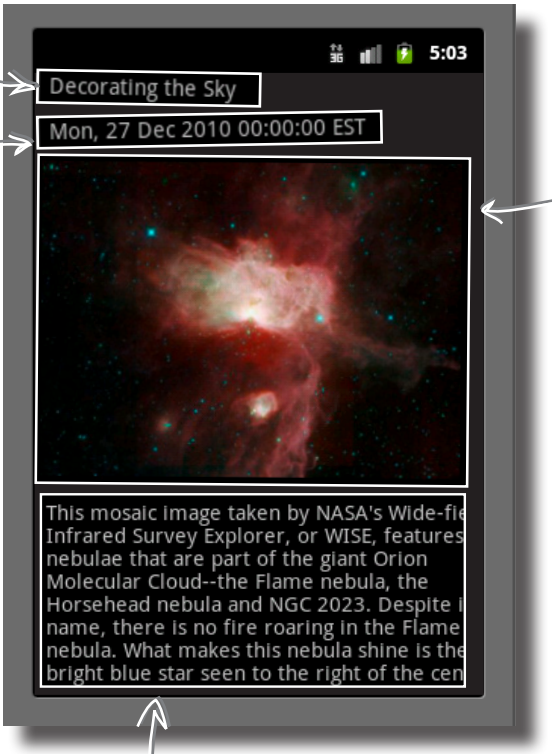
The image at the URL displayed in an ImageView (This is a new component but don't worry, you'll learn how to use it in a bit.)

Screen Design Magnet Solution

You were to add the `View` magnets at the bottom of this page to the screen to build your interface. There is one `View` for each of the properties you picked from the RSS feed.

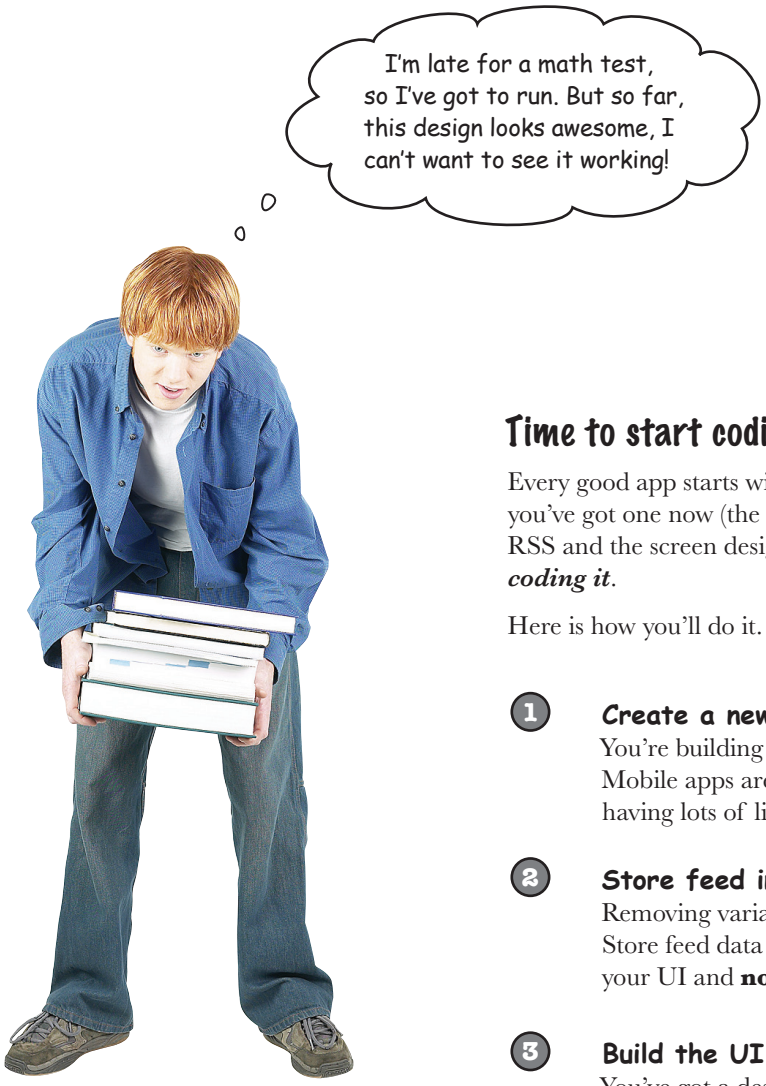
the title is at the top so you know what you're looking at.

The date really could go anywhere, but it's kind of a nice subheader isn't it?



The image is front and center, stretched to the size of the screen.

The description is nice to have, but it's definitely not the most important piece of data. It's also really big! Best to keep it at the bottom of the screen, out of the way.



Time to start coding!

Every good app starts with **a good plan**, and you've got one now (the selected fields from the RSS and the screen design). Now it's time to ***start coding it***.

Here is how you'll do it.

1. **Create a new project**
   You're building a new app, so start a new project. Mobile apps are small and concise, so get used to having lots of little apps (and projects) around!

2. **Store feed information locally**
   Removing variables from development is a good thing. Store feed data locally, so you can focus on building your UI and **not** connecting to the feed.

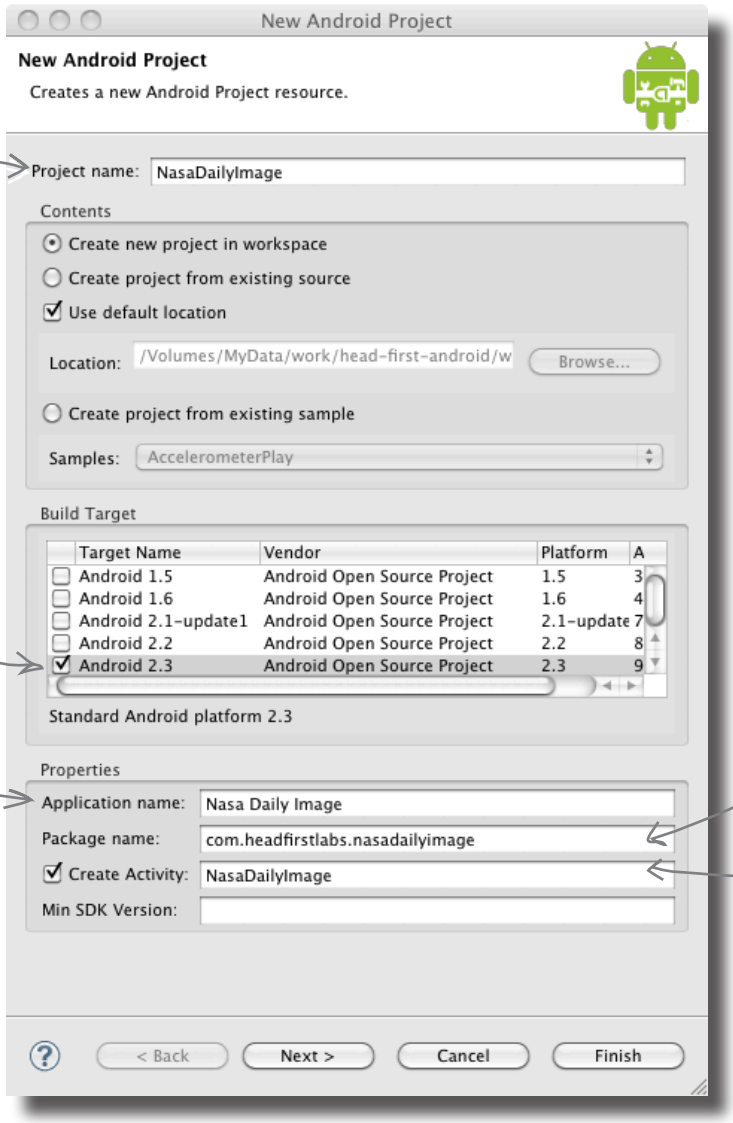
3. **Build the UI using the stored feed data**
   You've got a design for the UI; now it's time to execute it. Create layouts, implement UI functionality, and get the app up and running!

4. **Connect the app to the XML RSS feed**
   Once the app is up and running, just plug it into the XML feed and get the live data. It really ***is*** going to be that easy. Promise!

Create a new project

Now that you're ready to start coding, make a new Android Eclipse project. Launch the new Android project wizard in Eclipse by going to File -> New -> Android Project.

The project name can have spaces or not. But It's better leave out spaces, because a directory is created with the project name in your workspace, and command-line navigation is usually easier without spaces.



Select the latest platform you have installed (2.3 at the timeof this writing)

The application name has spaces. This is shown to your users, so format it to be human readable.

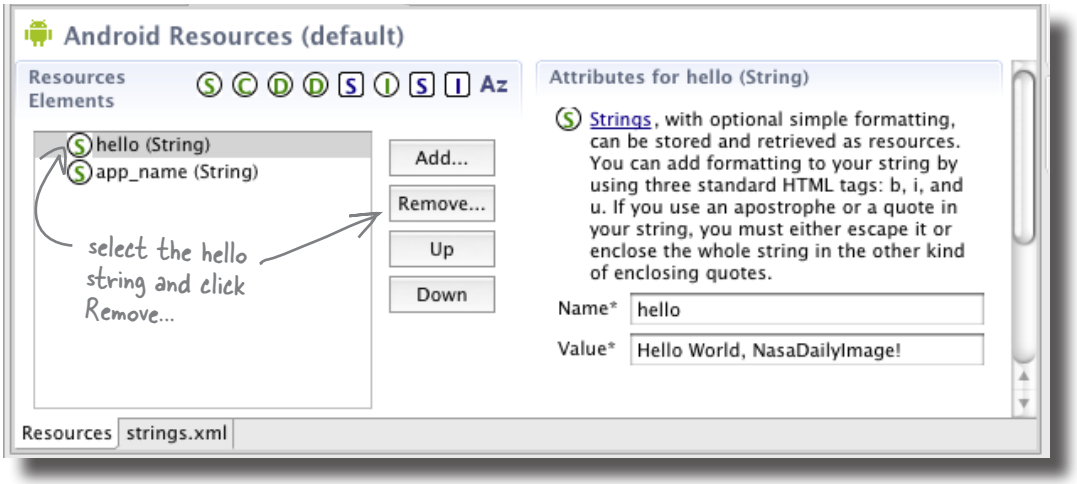
Web site plus application name is a pretty safe bet for a package name.

Make a default activity. Naming the activity to match the project name is a good rule for single-screen apps.

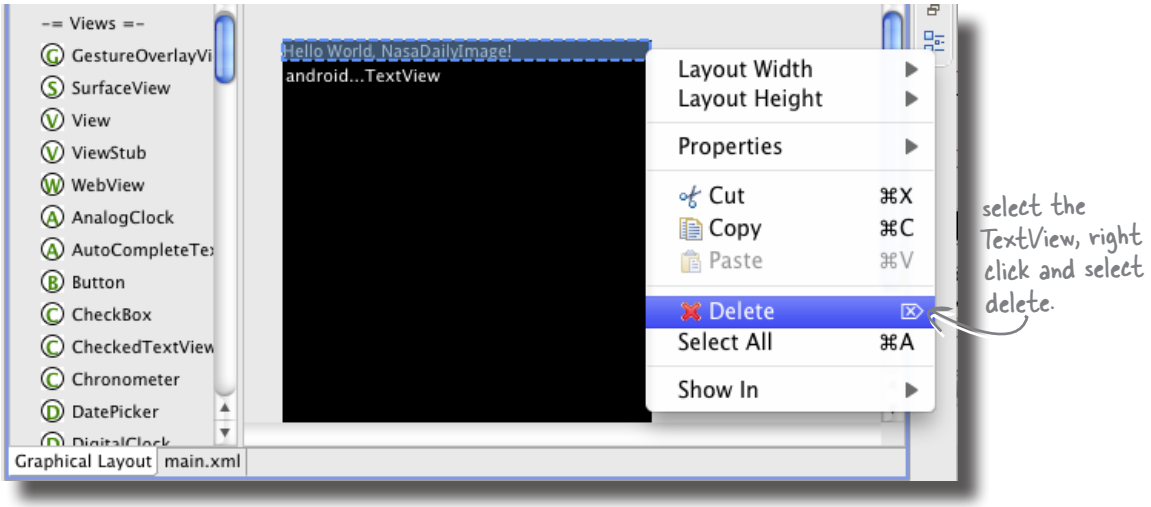
Get rid of the autogenerated 'Hello' stuff

You're not going to need the autogenerated `TextView` showing the default "Hello World, NasaDailyImage" text. So before you get going, delete the `TextView` and the the `String`.

1. Open *strings.xml* (under `res/values`) and delete the `hello String`.



2. Open *main.xml* (under `res/layout`) and delete the `hello TextView`.



3. Save your files. You now have a nice, **clean** app, without the boilerplate ***hello app*** content.

Store feed information locally

Start by saving text values as string resources. Open *strings.xml* and add three new strings for the image title, date, and description. The easiest way to do this is to copy the values directly from the sample XML feed file you saved at the beginning of the chapter.

strings.xml with the new test information added

```
<resources>
    <string name="app_name">NASA Daily Image</string>
    <string name="test-image_title">
            Decorating the Sky
    </string>
    <string name="test-image_date">
            Mon, 27 Dec 2010 00:00:00 EST
    </string>
    <string name="test-image_description">
            This mosaic image taken by NASA\'s Wide-field
            Infrared Survey Explorer, or WISE, features
            three nebulae that are part of the giant Orion
            Molecular Cloud--the Flame nebula, the Horsehead
            nebula and NGC 2023. Despite its name, there is
```

Values from a RSS feed sample

Watch out for places where you need to add escape characters.

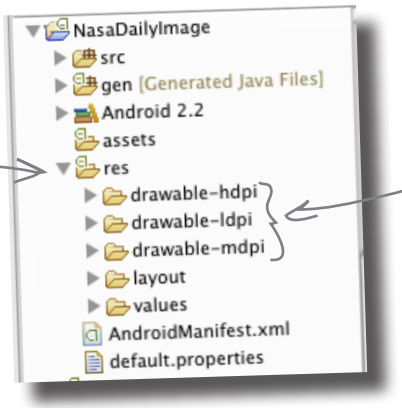
Watch out for escape characters

Some of the characters in the XML file (usually `, ", and \) need to be escaped, to let Java know they aren't control characters. Do this by preceding these characters with a \.

Save the image in your project

Images are stored in your Android project as resources in the *res* directory. Can you find a folder called *drawable* inside your project's *res* directory?

Here's the res directory, the same place your layouts and string resources are located.



Hmm. There are three different drawable directories here ...

There are **three** different drawable directories under res. What gives?

Ah yes, the folders are for different screen sizes.

One of the great things about Android is how many devices it runs on... and how many devices your apps can run on! The price for that versatility is the need to support a whole bunch of different devices with a wide range of resolutions and screen sizes.

You'll learn more about supporting different screen sizes and devices later. For now, just add images to the *drawable-hdpi* directory. The default emulator will use the images in this directory.

Do this!

Open up a browser and navigate to the URL for the image in the RSS XML file. Save the file to your project in the *res/drawable_hdmi* directory. Call it *test_image.jpg*.

Now that you have stored your data locally, let's build the layout!

Exercise

Below are magnets with the XML layout declarations for the `Views` in your layout along with the the `Views` they represent. Drag the the `View` XML magnets onto the layout on the next page of the exercise. This will complete the layout for the app.

View visuals here, just for reference.

Image

View XML declaration magnets

```
<ImageView
        android:id="@+id/imageDisplay"
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:src="@drawable/test_image"/>
```

Decorating the Sky

Title

```
<TextView
        android:id="@+id/imageTitle"
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:text="@string/test_image_title"/>
```

This mosaic image taken by NASA's Wide-field Infrared Survey Explorer, or WISE, features three nebulae that are part of the giant Orion Molecular Cloud--the Flame nebula, the Horsehead nebula and NGC 2023. Despite its name, there is no fire roaring in the Flame nebula. What makes this nebula shine is the bright blue star seen to the right of the central

Description

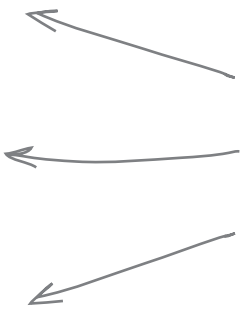
```
<TextView
        android:id="@+id/imageDescription"
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:text="@string/test_image_description"/>
```

Mon, 27 Dec 2010 00:00:00 EST

Date

```
<TextView
        android:id="@+id/imageDate"
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:text="@string/test_image_date"/>
```

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

</LinearLayout>
```



Put the widget magnets here to complete the layout. You're using LinearLayout, so you just need to arrange them with the component at the top of the screen as the first in the layout and continuing down.

Below are magnets with the XML layout declarations for the `Views`. You were to arrange the the `View` XML magnets to complete the layout for the app.

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

<TextView
        android:id="@+id/imageTitle"
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:text="@string/test_image_title"/>
```

Title

```
<TextView
        android:id="@+id/imageDate"
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:text="@string/test_image_date"/>
```

Date

```
<ImageView
        android:id="@+id/imageDisplay"
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:src="@drawable/test_image"/>
```

Image

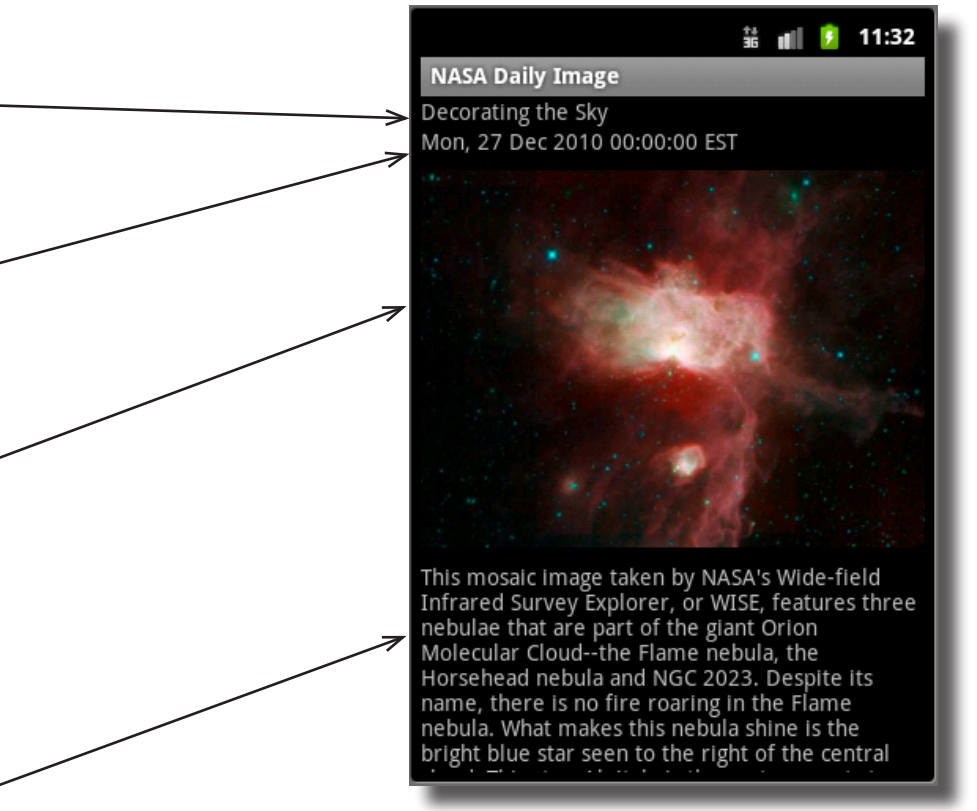
```
<TextView
        android:id="@+id/imageDescription"
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:text="@string/test_image_description"/>
```

Description

```
</LinearLayout>
```

Test Drive

Run the app by selecting the project in the Exclipse explorer view and selecting run. You'll have to select Android Application in the "Run as" pop-up that displays.



Nice! The screen is looking good!

The running screen matches your design. Excellent work.

Actually, scrolling *would* be a good idea!

You never know how long the description might be. NASA could throw a whole book in there, for all we know! After all, they are in control of the feed. The best we can do is make our app ***visually scalable***. A good way to do that is just to make the entire screen scroll.


Wouldn't it be dreamy if I could just wrap up my entire layout into some kind of View that would automatically scroll? But I know it's just a fantasy...

Use ScrollView to show more content

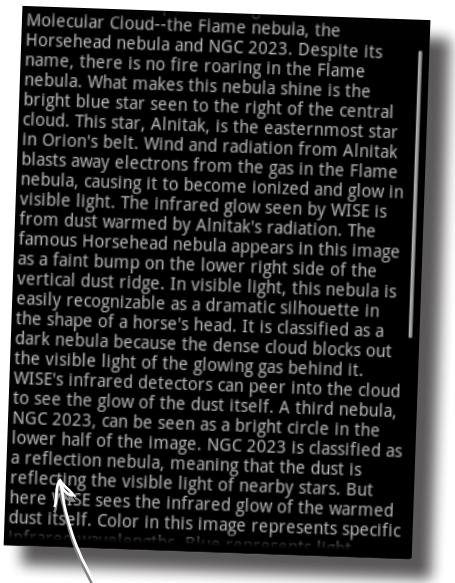
`ScrollView` is a `View` you can add to your screens to make content scroll. `ScrollView` is a `ViewGroup` (Android's name for layout manager). Use `ScrollView` by adding a child component to it, and the `ScrollView` will automatically scroll.

This ScrollView wraps the description TextView ...

```
<ScrollView
        android:layout_width="fill_parent"
        android:layout_height="fill_parent" >
        <TextView
            android:text="@string/image_description"
            android:layout_width="fill_parent"
            android:layout_height="wrap_content" />
</ScrollView>
```

The description TextView is big enough to overfill the screen by itself!

... and here's that TextView scrolling inside the ScrollView!



LOTS of scrolling text!

How much should scroll?

You can put one or more of the existing `Views` into the `ScrollView`. Any `Views` you add to the `ScrollView` will scroll, and the views not in the scrollview won't. Since your goal is visual scalability, just make the ***entire layout scroll***. This way, you can be gauranteed to have a **scalable** UI, even if unepxected information comes through the feed (like a *really* long title, for example).

One catch using `ScrollView` is that it can have only a single child `View`. In the example on this page, the `TestView` is added directly as a child to the `ScrollView`. But for the *whole* screen to scroll, you need multiple `Views` to scroll. The solution is to add a complete `LinearLayout` (with multiple child `Views`) as the `ScrollView`'s child.

Sharpen your pencil

Add and amend the following code to use the `ScrollView` to make the entire screen scroll. You'll need to make the `ScrollView` the main layout. And since the `ScrollView` can hold only one `View`, you need to add the entire `LinearLayout` as the one `ScrollView` child `View`.

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

<TextView
        android:id="@+id/imageTitle"
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:text="@string/test_image_title"/>
<TextView
        android:id="@+id/imageDate"
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:text="@string/test_image_date"/>
<ImageView
        android:id="@+id/imageDisplay"
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:src="@drawable/test_image"/>
<TextView
        android:id="@+id/imageDescription"
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:text="@string/test_image_description"/>

</LinearLayout>
```

Quick Tip: This needs to be in the root layout. If you add this layout to a ScrollView, you'll need to move this to the ScrollView.

Wrap this entire layout in a ScrollView.

Sharpen your pencil Solution

You were to used the `ScrollView` to make the entire screen scroll. You needed to make the `ScrollView` the main layout. And since the `ScrollView` can hold only one `View`, you should have added the entire `LinearLayout` as the one `ScrollView` child `View`.

```
<ScrollView xmlns:android="http://schemas.android.com/apk/res/android"
        android:layout_width="fill_parent"
        android:layout_height="fill_parent" >
    <LinearLayout xmlns:android="http://schemas.android.com/apk/res/android"
        android:orientation="vertical"
        android:layout_width="fill_parent"
        android:layout_height="fill_parent" >
    <TextView
        android:id="@+id/imageTitle"
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:text="@string/test_image_title"/>
    <TextView
        android:id="@+id/imageDate"
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:text="@string/test_image_date"/>
    <ImageView
        android:id="@+id/imageDisplay"
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:src="@drawable/test_image"/>
    <TextView
        android:id="@+id/imageDescription"
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:text="@string/test_image_description"/>
    </LinearLayout>
</ScrollView>
```

Beginning of the ScrollView

Did you remember to move the xmlns:android attribute from the LinearLayout to the ScrollView (the root view)

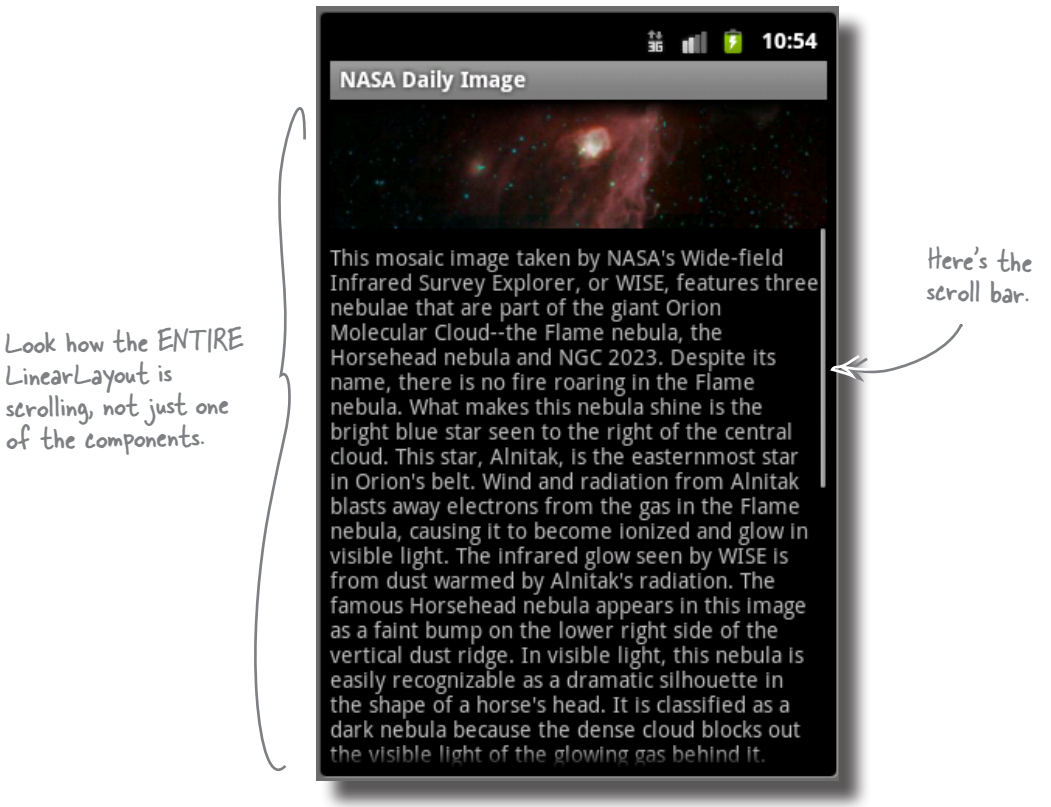
The complete non-scrolling layout

The inner widgets remain untouched inside the LinearLayout.

end of the ScrollView

Test Drive

Run your app to check the scrolling you just added. You should see the entire screen scrolling.



Everything is scrolling as expected.

The scrolling is working properly. See how the ***entire screen*** content scrolls up and down together? That's because you added the entire `LinearLayout` as the child to the `ScrollView`.

Let's show it to Bobby and see what he thinks!

Oops! Almost forgot about the actual feed.

Things are going really well with the design and layout. The screen ***looks*** like you want. Now it's time to make it ***work*** the way you want... parsing the feed data in real time.

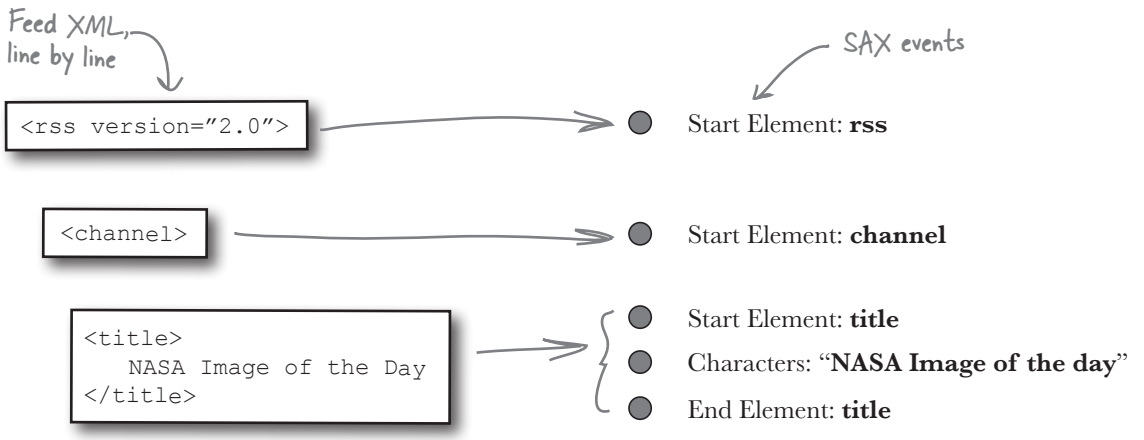
Choose a parser

There are plenty of XML parsers out there, and Android has built-in support for three of them: **DOM** (Document Object Model), **SAX** (Simple API for XML), and **XMLPULL**. They each take a different approach to parsing the XML and each has benefits and drawbacks. We're going to skip the big XML parser smackdown here (don't worry, though, you can find plenty on the Web) and just pick one.

Let's keep it simple and start with SAX.

SAX Parsing Up Close

SAX works by firing events while parsing the XML. There is no random access with SAX. The parser begins at the beginning of the XML, fires appropriate messages, and exits. Here's a quick sample of a few events that get fired in the first three lines of the NASA image feed.



The parser for the NASA feed will need to listen for the SAX start element messages for the fields in the app (the title, image URL, description, and date) and cache the values. *That's it!*

Let's review some Ready Bake parser code to keep you moving!

SAX-based feed parsers look pretty much the same. Now that you understand how the SAX parser ***conceptually*** works, here is a parser packaged up as Ready Bake code that you can just drop into your app. Don't worry about understanding everything; just add it to your project. But feel free to explore it!

```
public class IotdHandler extends DefaultHandler {
        private String url = "http://www.nasa.gov/rss/image_of_the_day.rss";
        private boolean inUrl = false;
        private boolean inTitle = false;
        private boolean inDescription = false;
        private boolean inItem = false;
        private boolean inDate = false;
        private Bitmap image = null;
        private String title = null;
        private StringBuffer description = new StringBuffer();
        private String date = null;

        public void processFeed() {
                try {
                        SAXParserFactory factory =
                                SAXParserFactory.newInstance();
                        SAXParser parser = factory.newSAXParser();
                        XMLReader reader = parser.getXMLReader();
                        reader.setContentHandler(this);
                        InputStream inputStream = new URL(url).openStream();
                        reader.parse(new InputSource(inputStream));
                } catch (Exception e) {
        }
        private Bitmap getBitmap(String url) {
                try {
                        HttpURLConnection connection =
                                (HttpURLConnection)new URL(url).openConnection();
                        connection.setDoInput(true);
                        connection.connect();
                        InputStream input = connection.getInputStream();
                        Bitmap bitmap = BitmapFactory.decodeStream(input);
                        input.close();
                        return bitmap;
                } catch (IOException ioe) { return null; }
        }
```

Since the events get called separately (like starting elements and their contents), keep track of what element you're in ...

Configuring the reader and parser

Make an input stream from the feed URL

Start the parsing!

```
public void startElement(String uri, String localName, String qName,
                Attributes attributes) throws SAXException {
        if (localName.equals("url")) { inUrl = true; }
        else { inUrl = false; }

        if (localName.startsWith("item")) { inItem = true; }
        else if (inItem) {
                if (localName.equals("title")) { inTitle = true; }
                else { inTitle = false; }

                if (localName.equals("description")) { inDescription = true; }
                else { inDescription = false; }

                if (localName.equals("pubDate")) { inDate = true; }
                else { inDate = false; }
        }
}

public void characters(char ch[], int start, int length) {
        String chars = new String(ch).substring(start, start + length);
        if (inUrl && url == null) { image = getBitmap(chars); }
        if (inTitle && title == null) { title = chars; }
        if (inDescription) { description.append(chars); }
        if (inDate && date == null) { date = chars; }
}
}
```

.... and if you're in an element that you are interested in, cache the characters.

Here are a few accessors. so you can get the cached variables back from the parser...

```
public String getImage() { return image; }
public String getTitle() { return title; }
public StringBuffer getDescription() { return description; }
public String getDate() { return date; }
```

Download the `IotdHandler` code from the *Head First Android Development* site and add it to your project.

Connect the handler to the activity

Now that you've added the feed parser code to your project, you need to use it in your activity. Start by instantiating the handler in your Activities `onCreate` method.

```
public void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.main);
        IotdHandler handler = new IotdHandler();
        handler.processFeed();
}
```

Create the handler ...

... and start parsing ...

The app's not going to work with the parser yet. You're parsing the feed, but you're not setting the values cached in the feed on the Views.

True, the values are cached in the handler, but never displayed.

Let's make a method called `resetDisplay` that will set all of the view data on screen. Then you can call that method in `onCreate()` after`processFeed()` returns.

Code Magnets

Complete the `resetDisplay()` method below by retrieving references to the on-screen `Views` (using `findViewById`) and setting the values on those `Views` with the values passed in. Once this method is complete, you can use it to pass in the values from the feed.

```
private void resetDisplay(String title, String date,
        String imageUrl, String description) {
```

Get a reference to each on screen View. Then set the values on those Views to the cached values from the parser.

Here are your magnets.

```
(TextView)findViewById(R.id.imageDate);
(TextView)findViewById(R.id.imageTitle);
titleView.setText(title);
TextView descriptionView =
(ImageView)findViewById(R.id.imageDisplay);
}
(TextView)findViewById(R.id.imageDescription);
dateView.setText(date);
ImageView imageView =
imageView.setImageBitmap(image);
descriptionView.setText(description);
TextView dateView =
TextView titleView =
```

Code Magnets

You were to complete the `resetDisplay()` method below by retrieving references to the on screen `Views` (using `findViewById`) and setting the values on those `Views` with the values passed in. With this method complete, you can use it to pass in the values from the feed.

```
private void resetDisplay(String title, String date,
        String imageUrl, String description) {
```

```
TextView titleView =
(TextView)findViewById(R.id.imageTitle);
titleView.setText(title);
```

Get a reference to the title view and set the text to the cached value from the handler.

```
TextView dateView =
(TextView)findViewById(R.id.imageDate);
dateView.setText(date);
```

Same deal with date View: get the View reference and set the text to the value from the parser.

Get a reference to the ImageView.

```
ImageView imageView =
(ImageView)findViewById(R.id.imageDisplay);
imageView.setImageBitmap(image);
```

Use the image from the feed parser and set it on the ImageView.

```
TextView descriptionView =
(TextView)findViewById(R.id.imageDescription);
descriptionView.setText(description);
```

Finish up by getting the description View reference and setting the text with the cached description value.

```
}
```

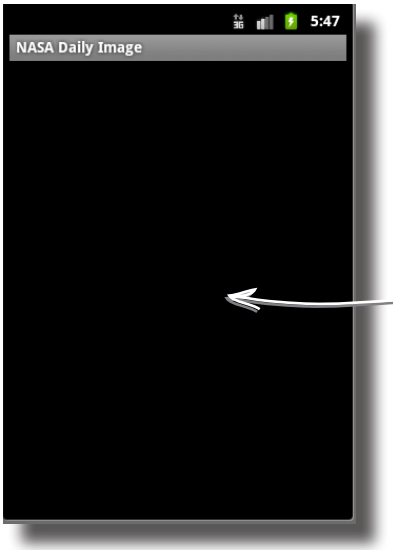
Now you can finish connecting the handler in the `onCreate()` method. Add a call to `resetDisplay()` after `handler.processFeed()`. This will take the cached values in the parser and set them in the `Views` screen.

```
resetDisplay(iotdHandler.getTitle(), iotdHandler.getDate(),
    iotdHandler.getImage(), iotdHandler.getDescription());
```

The resetDisplay method is a helper method you're about to write to populate the fields on screen with the parsed data.

Test Drive

Everything is plugged in with the parser. The parser is integrated with the activity, and the results from the parsing are displayed on the screen. You should be good to go. Go ahead and run the app.



Hmm, a blank screem...

Uh oh! The screen is gone!

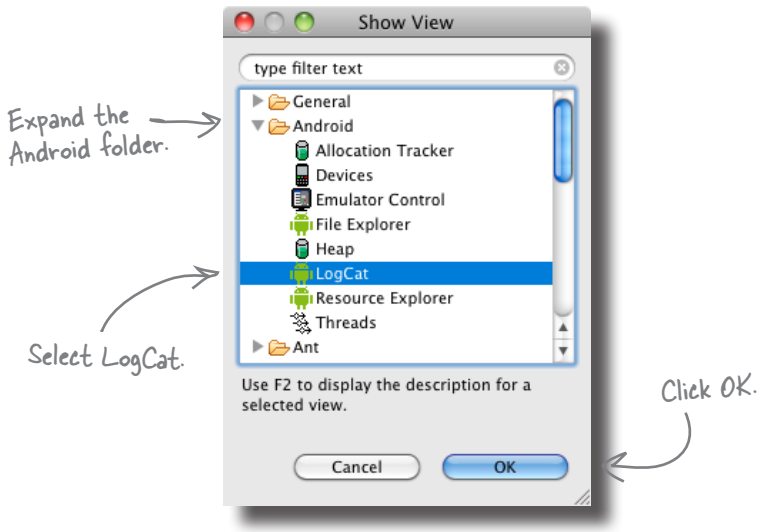
Brain Barbell

Clearly, something broke along the way. What broke? Where would you look to find out what's broken?

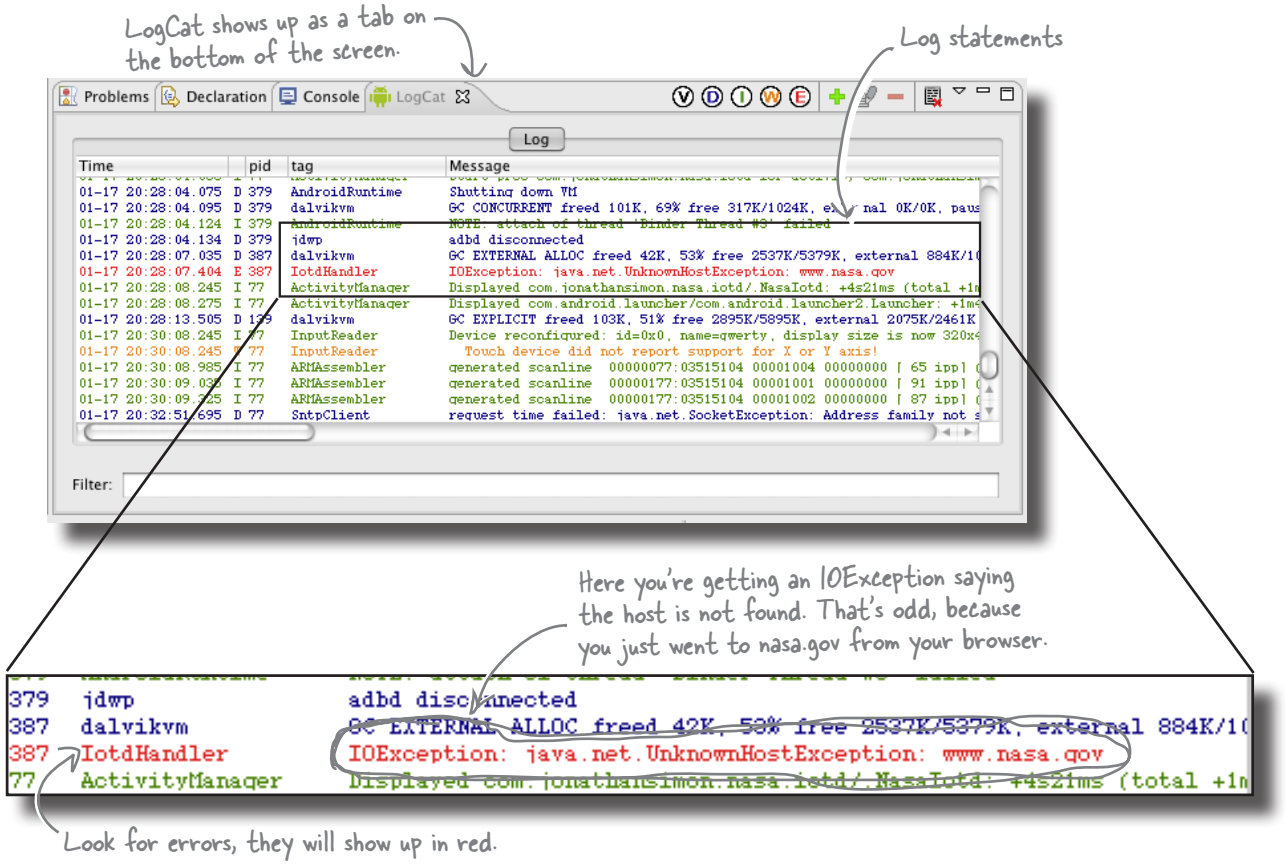
Find errors with LogCat

It's OK; errors happen! The important thing is knowing where to go to find out what's happening with your application, so you can fix things when they break. Android uses a built-in logging mechanism that outputs to a screen included in the Android Development Tools (ADT) called *LogCat*.

Open LogCat by going to Window -> Show View -> Other, which will bring up the Eclipse Show View dialog. Expand the Android folder, select LogCat, and press OK.



After you click OK, you'll see the new LogCat view in your Eclipse workspace.



Use permissions to gain restricted access

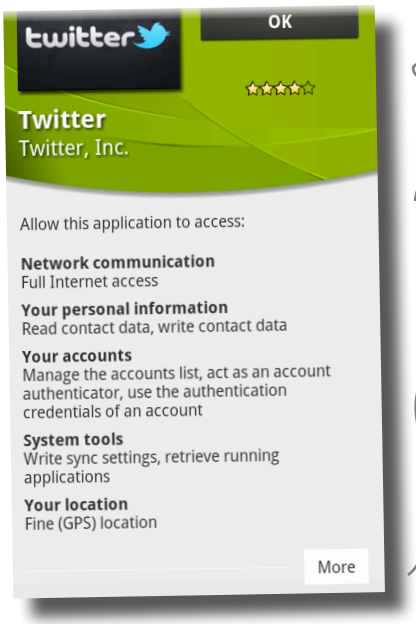
The `UnknownHostException` is thrown here because you need ***permission*** to access the Internet.

With all the cool stuff you can do with Android devices, it's hard to remember that they are **mobile devices**. And because of this, Android is built to be super careful about making sure each app has rights only to the system resources it ***absolutely needs***. The only way for your app to get those permissions is to request them.

How do permissions work?

You can specity the permissions your app needs using a group of permission constants in *AndroidManifest.xml*. When users install your app from the Android market, they are prompted with a list of permissions that your app requsts. If they agree, they accept the permissions and the app installs.

As an example, let's take a look at the Android market install page for the official Twitter app.



The market install page for the Twitter app

All of the permissions the app requests

The permissions on the market page are generated from these constants.

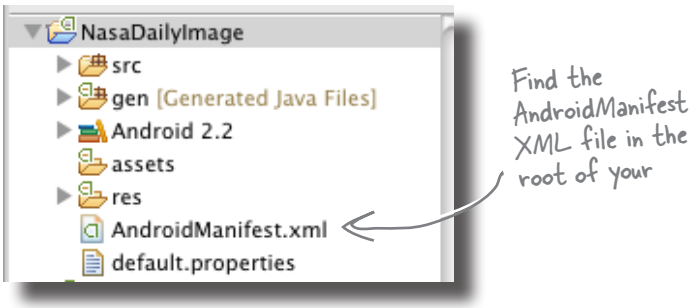
```
android.permission.INTERNET

android.permission.READ_CONTACTS
android.permission.WRITE_CONTACTS

android.permission.MANAGE_ACCOUNTS
android.permission.AUTHENTICATE_ACCOUNTS
android.permission.USE_CREDENTIALS

android.permission.WRITE_SYNC_SETTINGS
android.permission.GET_TASKS

android.permission.ACCESS_FINE_LOCATION
```

Enough about Twitter. Let's add permission to your app!

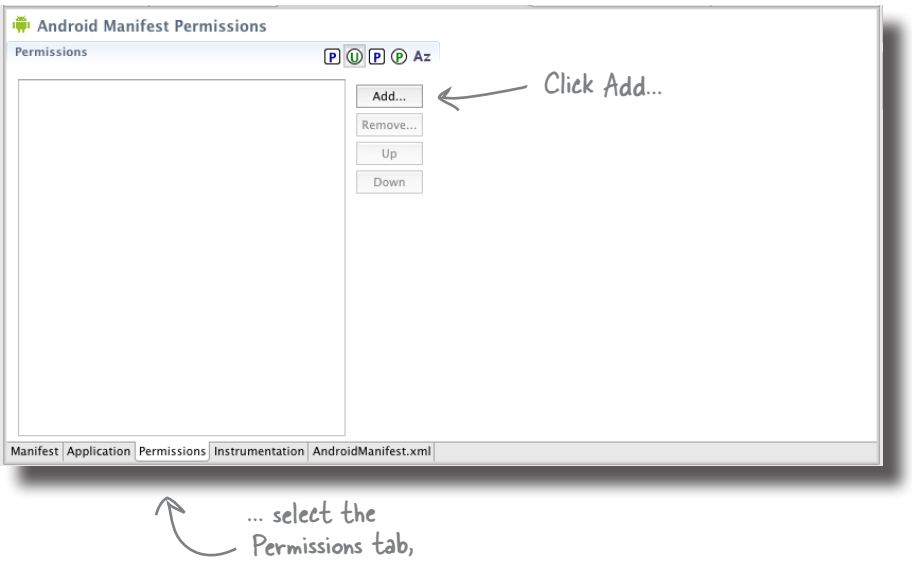
Add a permission to access the internet

The Twitter app had a lot of permissions, but your app just needs permission to access the Internet. Follow these instructions to add the Internet access permission.

1 **Open AndroidManifest.xml**
The *AndroidManifest* file is generated by the new app wizard. You can find it in the root of your project. Double-click the file to open it.

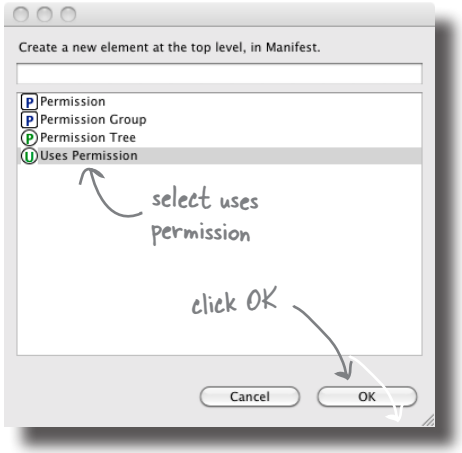


2 **Add a new permission to the manifest**
Just like all of the other Eclipse XML editors you've been working with, there's a custom editor for *AndroidManifest* file. Click on the Permissions tab and press the Add button to add a new permission.



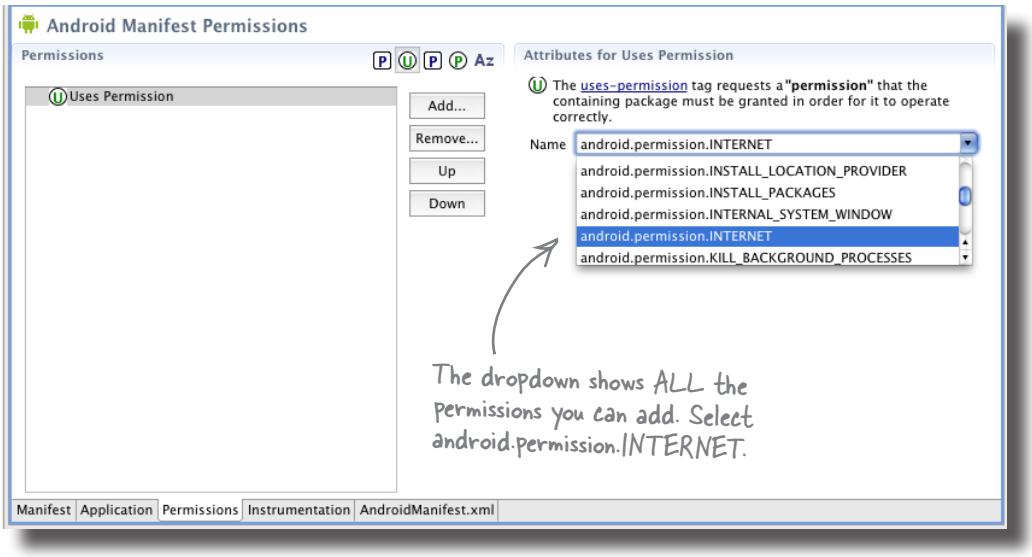
3 Select the permission type

When the dialog opens, select Uses Permission and click OK. This tells Android that you want to use a permission in your application.



4 Select the permission

There are a bunch of different permissions that you can ad to your application. Since you're accessing the Internet to get the feed and the image, select the `android.permission.INTERNET` permission.



To apply the changes, save the file when you're done.

Tonight's talk: **Permissions**

Android App:

What, seriously? I have to ask permission to do everything? Don't you trust me at all? This is ridiculous!

Unsupervised?!? Look, I'm not a child!

OK, well I kind of see that. But really, I have to tell you everything I do? Like *everything*? That's lame!

Why can't I just ask them myself?

Hey man, that's low.

You're right, I probably wouldn't. BUT ...

Mffft! Well, I suppose I don't really have a choice, do I?

Harsh.

Android Operating System:

No, it isn't ridiculous. I just need to be *really* careful about what I let you do unsupervised.

Well, listen, my user (who is also your user I might add) expects us all to work together to keep the whole phone ***secure***. We can't allow any viruses, unauthorized data access, unecessary Internet access, or other security no-nos to spoil their experience. Then we all lose!

Sorry, but you do. That way, I can tell our user what you're planning on doing and they can decide if they will let you do it.

How can I trust that if the user says no to you you'll actually listen? You wouldn't even listen to me if I couldn't kill your process!

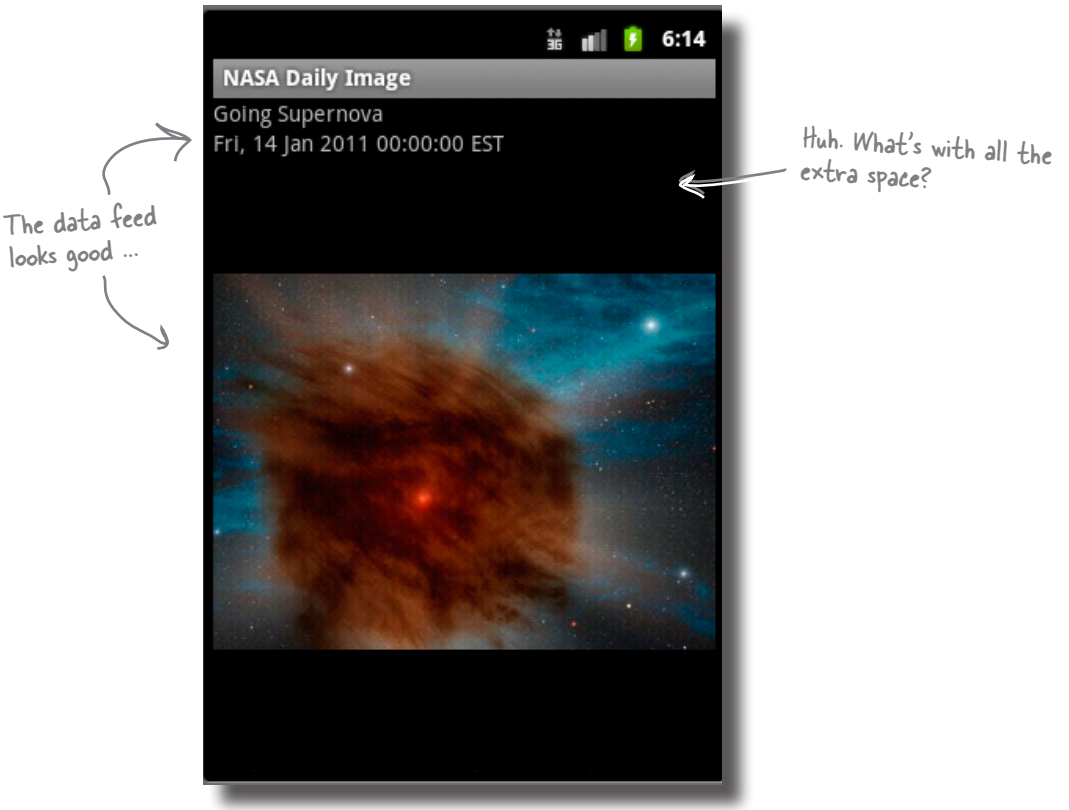
Well would you?

I rest my case!

Nope! You don't have a choice. My way, or the highway, buddy.

Test Drive

Now that the permissions are properly set, the app should run correctly, parsing the feed and displaying everything on the screen. Go ahead and run your app!



Better, but not done yet!

The feed is working (***fantastic!***), and fresh data is bing displayed on the screen. This is all great, but something is going wrong with the formatting.

How do you find out what's wrong?

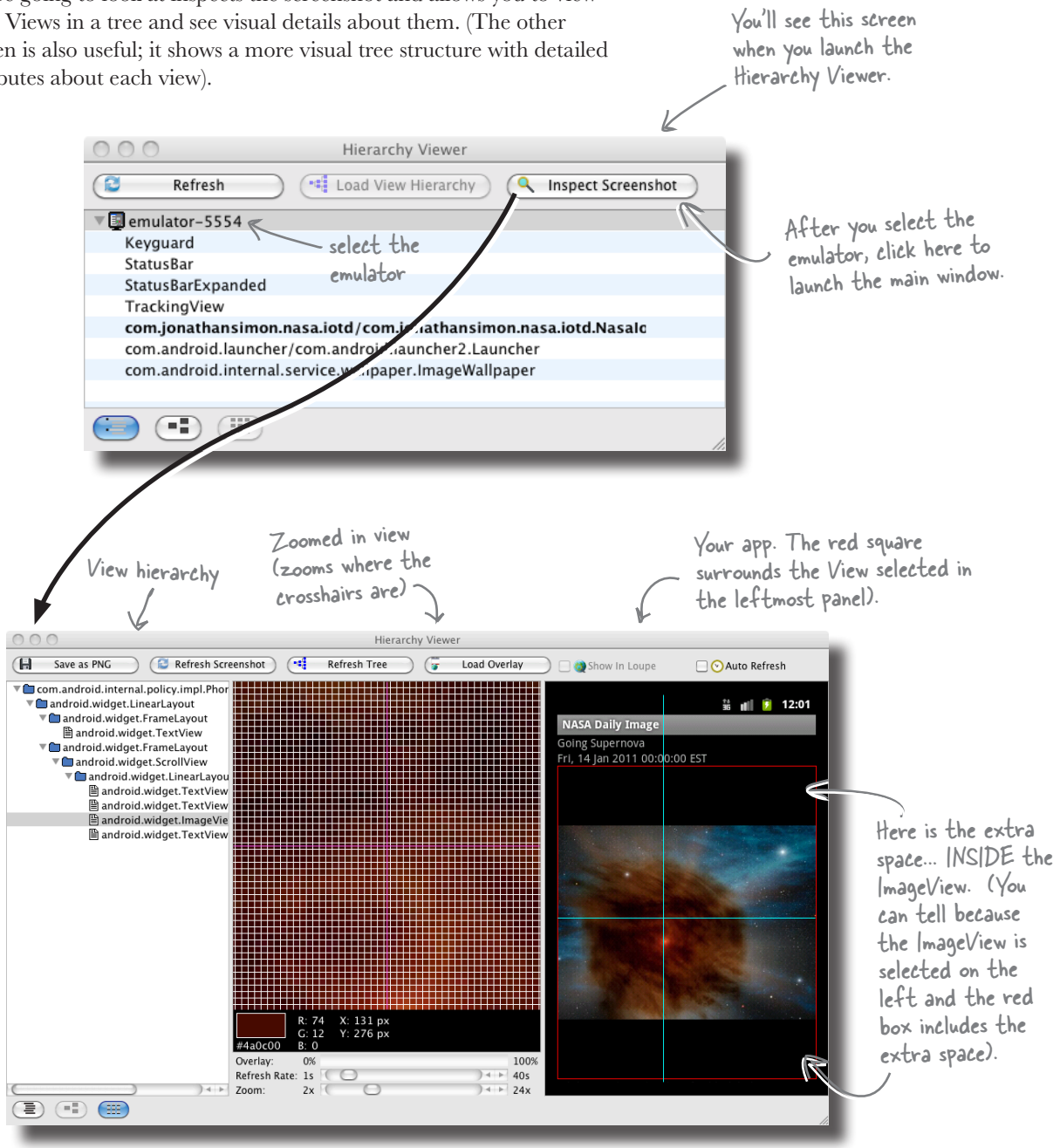
In fact, there is a built-in tool.

That tool is rhe Android **Hierarchy Viewer**. This cool little tool from the **Android SDK** lets you do all kinds of introspection on your layouts and Views to get to the bottom what's going on.

Launch the Hierarchy Viewer by opening a terminal, going to your `<SDK>/tools` directory, and executing **`hierarchyviewer`** at the command line.

Find layout problems with HierarchyViewer

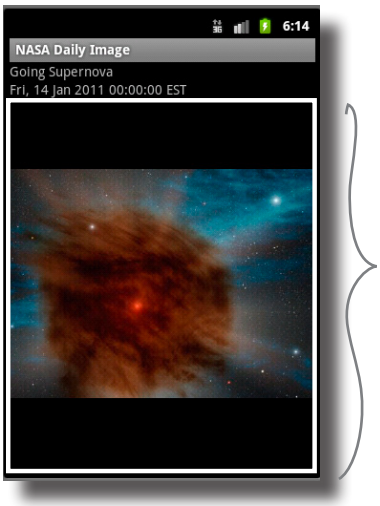
When you launch the **Hierarchy Viewer**, the first thing you'll see is the selection screen below. There are two main views; the view you're going to look at inspects the screenshot and allows you to view your Views in a tree and see visual details about them. (The other screen is also useful; it shows a more visual tree structure with detailed attributes about each view).



Set the adjustViewBounds property

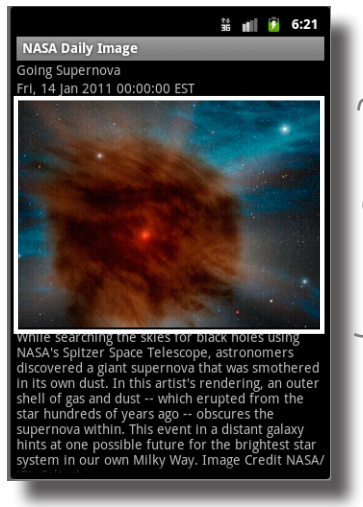
You can see from the Hierarchy Viewer that the `ImageView` is too big. But why? The cause is actually that the apsect ratio is not preserved when the `Bitmap` from the Web is displayed. The **aspect ratio** is what keeps the width to height ***proptionally*** the same when you resize an image, and the image is being resized by the internal layout code to fill the screen width.

`adjustViewBounds = false`



Without keeping the aspect ratio the same, the image stretches and takes up too much space.

`adjustViewBounds = true`



When set to true, the image stretches to the edges of the screen, and sets a height proprtional to that width.

If you set the `adjustViewBounds` property to **true** in your layout XML, the extra space will go away.

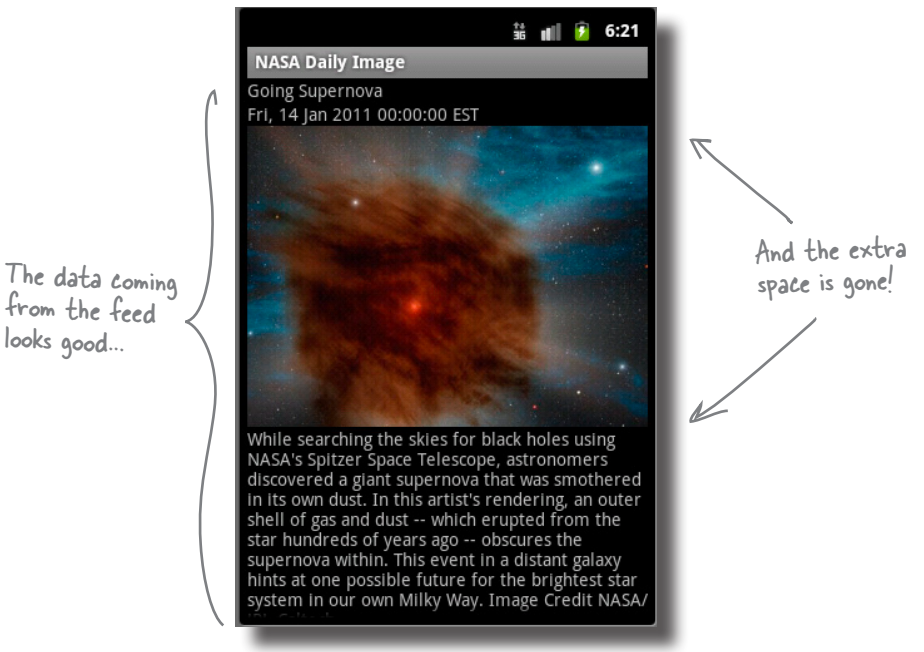
main.xml

```
<ImageView
        android:id="@+id/imageDisplay"
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:adjustViewBounds="true" />
```

Set this property in your layout XML.

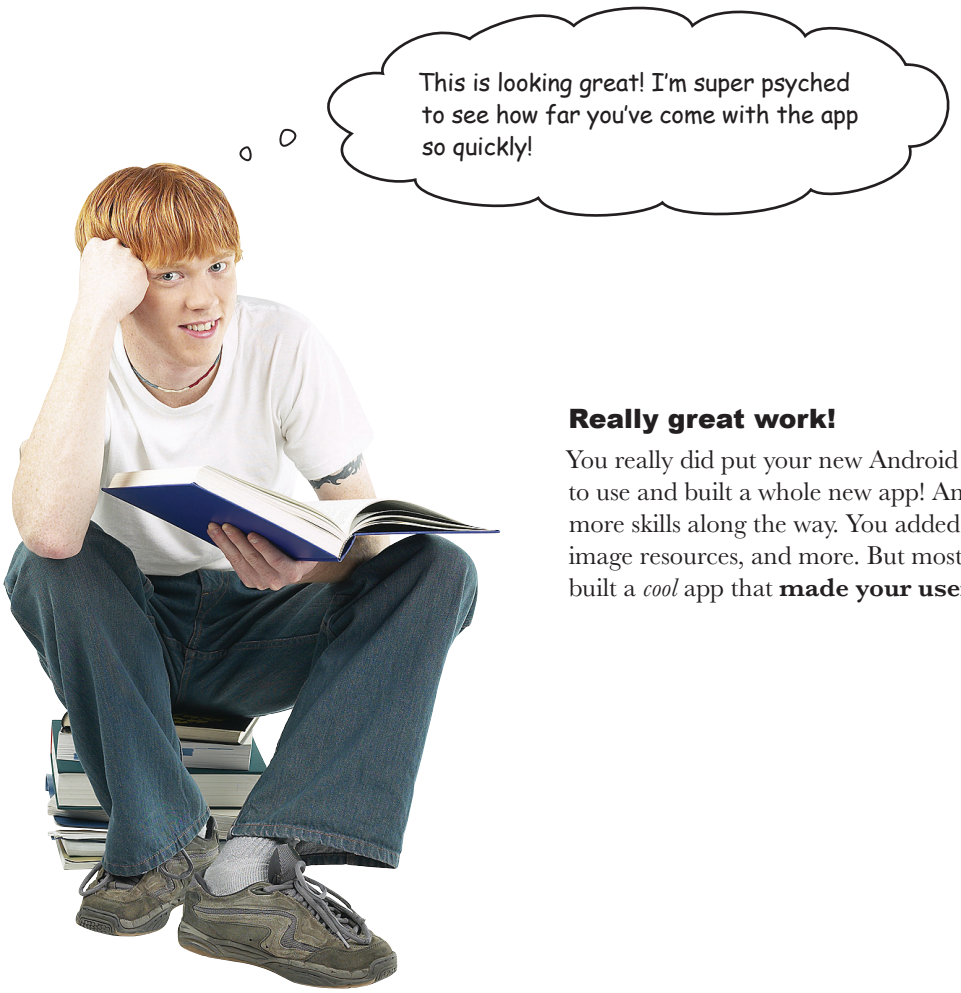
Test Drive

With the `adjustViewBounds` properts upated in your layout, run the app again. This time, you should see the image resized correctly in the layout.



It's all coming together!

The layout works just like you designed it, the feed parsing is up and running, and the layout issue with the `ImageView` is fixed.



Really great work!

You really did put your new Android development skills to use and built a whole new app! And you learned even more skills along the way. You added scrolling layouts, image resources, and more. But most importantly, you built a *cool* app that **made your users happy**!

NASA Daily Image Toolbox

Now that you have a cool RSS feed-parsing app in your toolbox, you can build all kinds of your own cool feed-based apps!

Built-in Problem Solvers

- Use LogCat to view code log statements and errors from your apps.
- Use HierarchyViewer to analyze your views and layouts. This can be extremely helpful when layouts or views aren't behaving as you might expect them to.

View Roundup

- Use TextView to display text. You can use it for small text like labels, or really big text like the Image descriptions.
- Use ImgaeView to display images. You can add your own images to the res directory and display them in an ImageView.
- Use ScrollView to make your content scroll on screen. ScrollView can have only one chid View, so wrap multiple child views in a layout to make them all scroll.

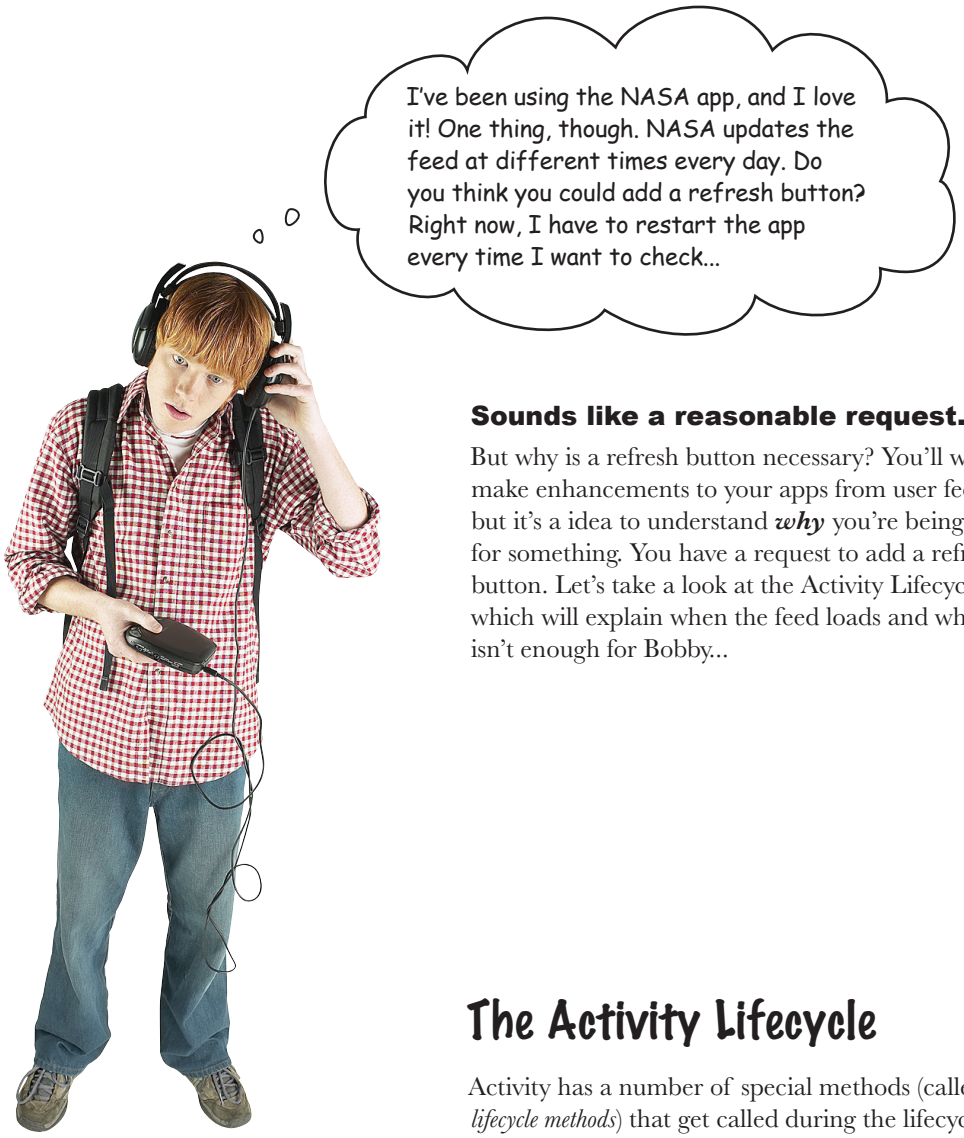
BULLET POINTS

- When working with RSS feeds, download a sample of the feed and decide what content in the feed you want to use in your app.
- Start with `SAX` parsing, but explore the `DOM` and `XMLPULL` parsers to see if they will work better in your app.
- It's a good practice to break your app down into small development pieces. For RSS feed apps that rely on the Internet, it's perfectly acceptable (and even a good idea) to build out your app with test data and plug in the Internet services later.
- Add image resources to the *res/drawable-hdmi* directory (for now). These will get picked up by the Android compiler and the images will be available to your application.
- Use `ImageView` to display images in your app.
- Use `ScrollView` when your app's content is too big for the screen. (Just remember that `ScrollView` can have only one child).
- When things go wrong, use `LogCat` to look at Android errors and log statements.
- Make sure your app has the proper permissions configured in *AndroidManifest.xml*.
- Use `HierarchyViewer` to debug your layouts when your app isn't displaying correctly.

3 long-running processes

When things take time

It would be great if everything happened instantly. Unfortunately, some things just take time. This is especially true on mobile devices, where network latency and the occasionally slow processors in phones can cause things to take a ***bit*** longer. You can make your apps faster with optimizations, but some things just take time. But you ***can*** learn how to **manage long-running processes better**. In this chapter, you'll learn how to show active and passive status to your users. You'll also learn how to perform expensive operations off the UI thread to guarantee your app is always responsive.

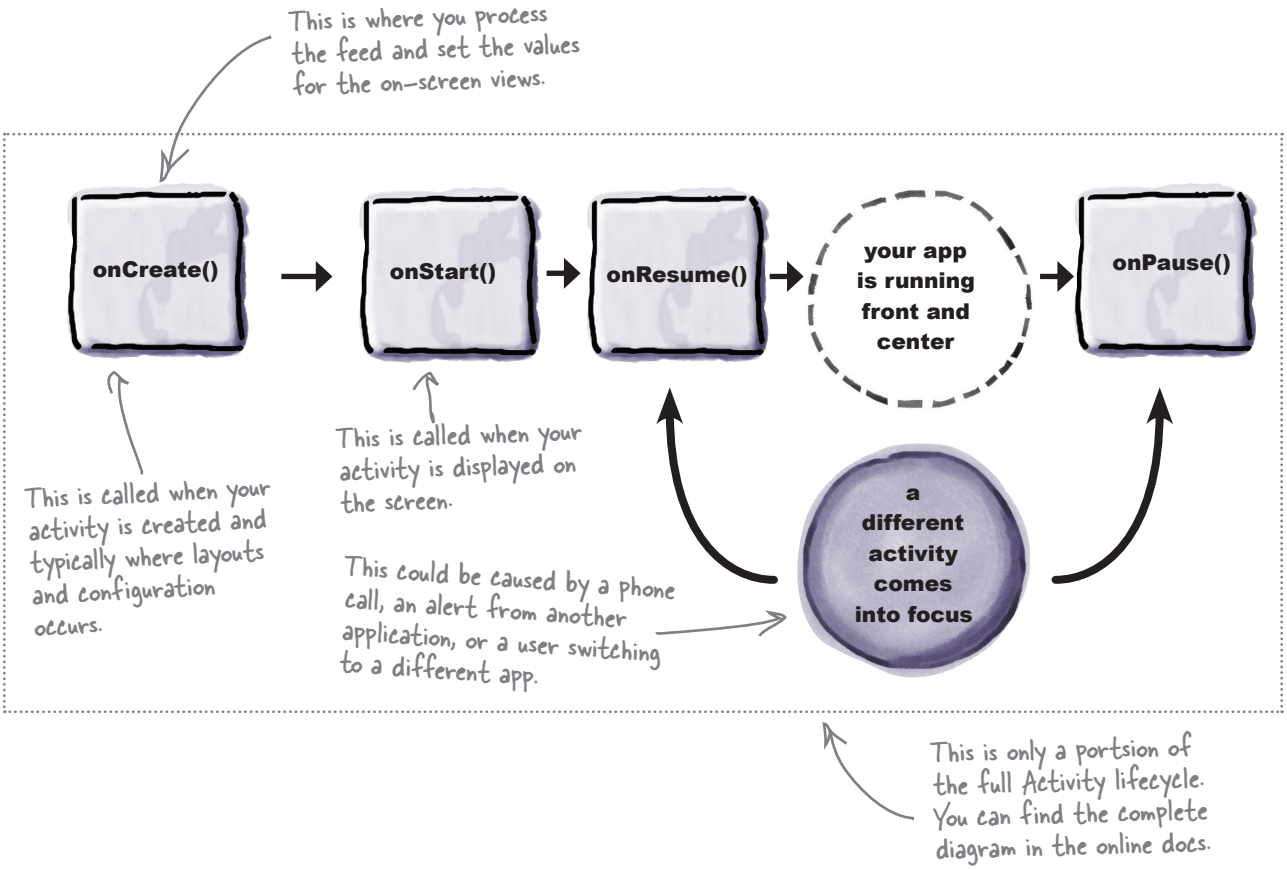


Sounds like a reasonable request...

But why is a refresh button necessary? You'll want to make enhancements to your apps from user feedback, but it's a idea to understand ***why*** you're being asked for something. You have a request to add a refresh button. Let's take a look at the Activity Lifecycle which will explain when the feed loads and why it isn't enough for Bobby...

The Activity Lifecycle

Activity has a number of special methods (called *lifecycle methods*) that get called during the lifecycle of the activity. The `onCreate()` method where you set the layout is one of these methods, and there are ***many*** more. A few of these methods are shown here, so you can see where the feed is (and *is not*) refreshed.



When does the feed refresh?

The feed refreshes ***only*** when the activity starts and the `onCreate()` method is called. The feed will never refresh once the app starts. Currently, the only way to get the app to refresh the feed is to exit the app and then restart it.

You could override more of the lifecycle methods like `onResume()`, but that would only cover the case where the app is paused and restarted. You could also build some sort of auto-refresh mechanism, but that is very processor and battery intensive. Looks like the refresh button *is* a good idea after all.

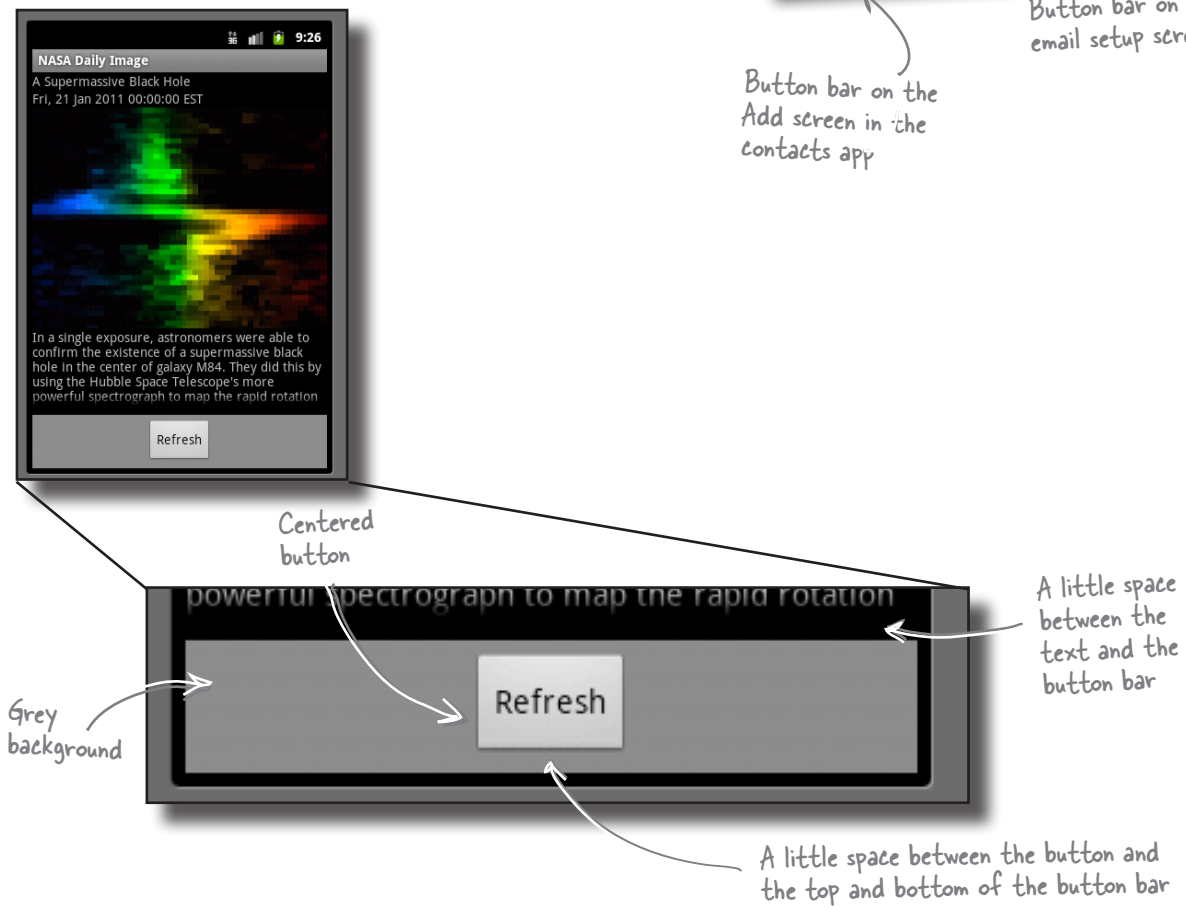
Update the user interface

A recurring Android user interface design pattern for on-screen actions, the **button bar** is a gray panel on the bottom of the screen holding one ore more buttons. This will work perfectly for the refresh button placement.

Let's build the **button bar** as a standalone layout and then add it the app's current layout. Encapsulating parts of your fullscreen layout into separate *smaller* layouts can be a good way to organize layouts. And since `LinearLayout` extends `ViewGroup`, which itself extends `View`, you can add your entire new `LinearLayout` you're making for the **button bar** as a child to your original `ViewGroup`.

Button bar on an email setup screen

Button bar on the Add screen in the contacts app



Start with a basic LinearLayout

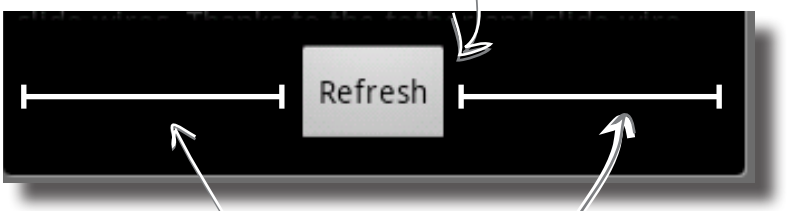
`LinearLayout` is a surprisingly functional layout manager for basic screen designs. You've already built a few screens using `LinearLayout`, and you're going to build the button bar with it too. You will learn more about `LinearLayout` in the process, and don't worry; you will also learn about other layout managers later in the book.

The key to using `LinearLayout` for the Button Bar is to center the refresh button using the `android:gravity` attribute. Then you can fine-tune the layout.

The beginnings of the button bar layout: right now, just a LinearLayout with a centered button.

```
<LinearLayout
        android:orientation="horizontal"
        android:layout_width="fill_parent"
        android:layout_height="wrap_content"
        android:gravity="center"
        >
        <Button android:text="@string/refresh"
                android:layout_width="wrap_content"
                android:layout_height="wrap_content" />
</LinearLayout>
```

This is overkill, because horizontal is the default, but it's good to be safe.



The button is centered.

You're off to a great start! Now start fine-tuning the layout ...

Use properties to fune-tune the layout

With the button properly centered in the layout, let's focus on fine-tuning the layout to get the colors and spacing looking like the button bar examples. Use these properties to get the layout looking the way you want.

padding

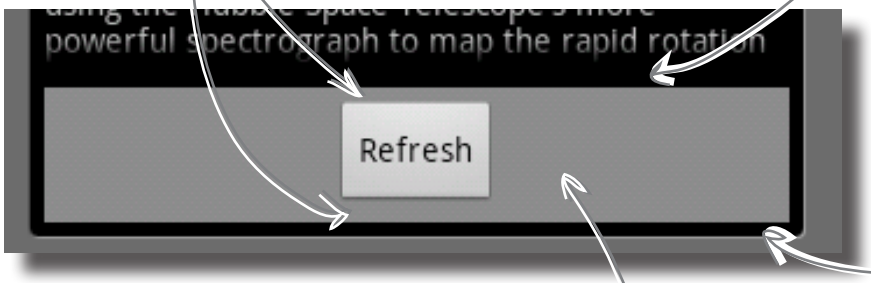
Padding controls the spacing between Views **within** a layout. Use ***Density Independent Pixels*** (DIP) to specify spacing rather than raw pixels to make your layouts ***really*** flexible.

```
android:padding="5dp"
```

margin

Margin controls the spacing between this View and the Views **outside** this layout. Use ***Density Independent Pixels*** (DIP) to specify spacing rather than raw pixels to make your layouts ***really*** flexible.

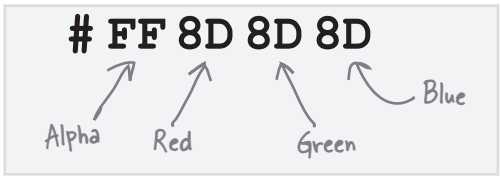
```
android:margin-top="5dp"
```



background

The background property can be set to an image resource, a color, and a few additional Android graphics types. Use a solid color for the button panel background, which is defined in **8-digit hexadecimal** format (two digits each for *alpha*, *red*, *green*, and *blue*).

```
android:background="#ff8D8D8D"
```



layout-width and layout-height

Layout width and height can be set to predefined values of **wrap_content** and **fill_parent**, as well as raw size values in pixels and DIPs. Using **wrap_content** makes the view just as big as it needs to be, while using **fill_parent** sizes the view to fill all of the space it can.

```
android:layout_height="wrap_content"
```

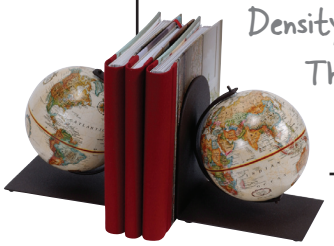
Use wrap_content to size the button. This way, it will be just as big as it needs to fit the "refresh" text.

```
android:layout_height="fill_parent"
```

Use fill_parent for the button bar's LinearLayout width. This will make sure that the layout stretches to the edges of the screen.

the Scholar's Corner

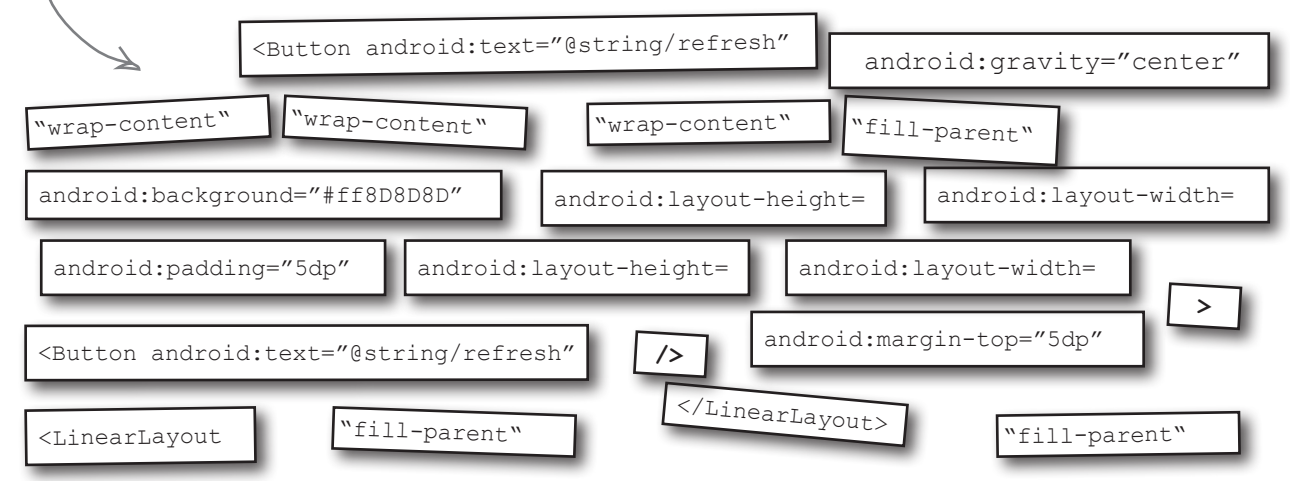
Density Independent Pixels (DIP) Android supports too many screen sizes to keep track! Using raw pixel dimensions in layouts might make your layout look good on one device and terrible on others. Android provides an ABSTRACT sizing measurement called Density Indepent Pixels that is derived from device attributes. This means that you can define layout attributes in DIPs that will look great on all Android devices. Thanks, Android!



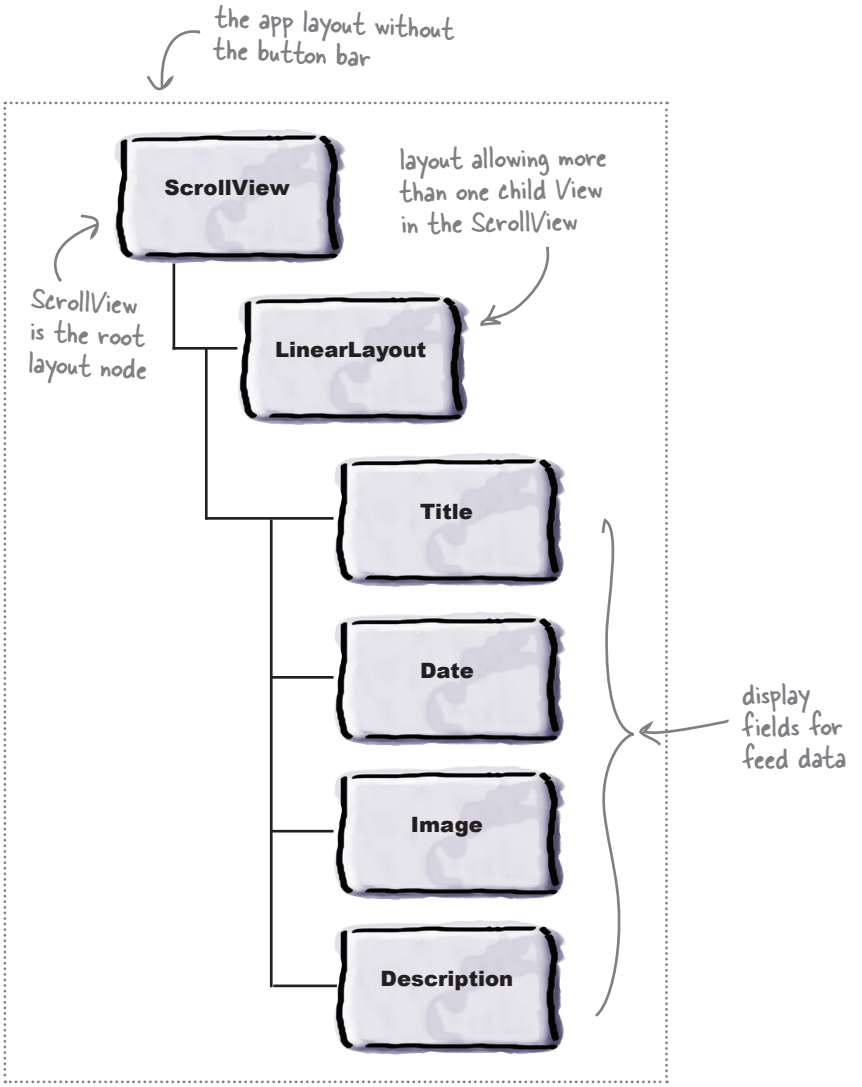
Button Bar Layout Magnets

Construct the button bar layout using the magnets below. Think about the width and height for each the button and the `LinearLayout`. And don't worry; you'll have a few extra magnets left over for widths and heights you didn't use.

Here are your magnets.



Now that you have the button bar layout, you need to add it to your screen. Below is a graphical representation of your current View/Layout hierarchy. Draw new views and layouts for the button bar Views (and any other views and layouts you need) to complete your layout. Also, remember, just like `ScrollView` that can have only one child, there can be only one root layout.



Button Bar Layout Magnets Solution

You were to construct the button bar layout using the magnets below. Think about the width and height for each the button and the `LinearLayout`. you should have a few extra magnets left over for widths and heights you didn't use.

```
<LinearLayout
    android:layout-width=    "fill-parent"
    android:layout-height=   "wrap-content"
    android:background="#ff8D8D8D"
    android:margin-top="5dp"
    android:padding="5dp"
    android:gravity="center"
>

    <Button android:text="@string/refresh"
        android:layout-width=    "wrap-content"
        android:layout-height=   "wrap-content"
    />

</LinearLayout>
```

The width is set to fill parent, so it fills the width of the screen.

The height is set to wrap content; it shouldn't be the full height (since there is also the scrollpane).

Set the background color to a medium grey.

Add some spacing between the button panel and scroll pane.

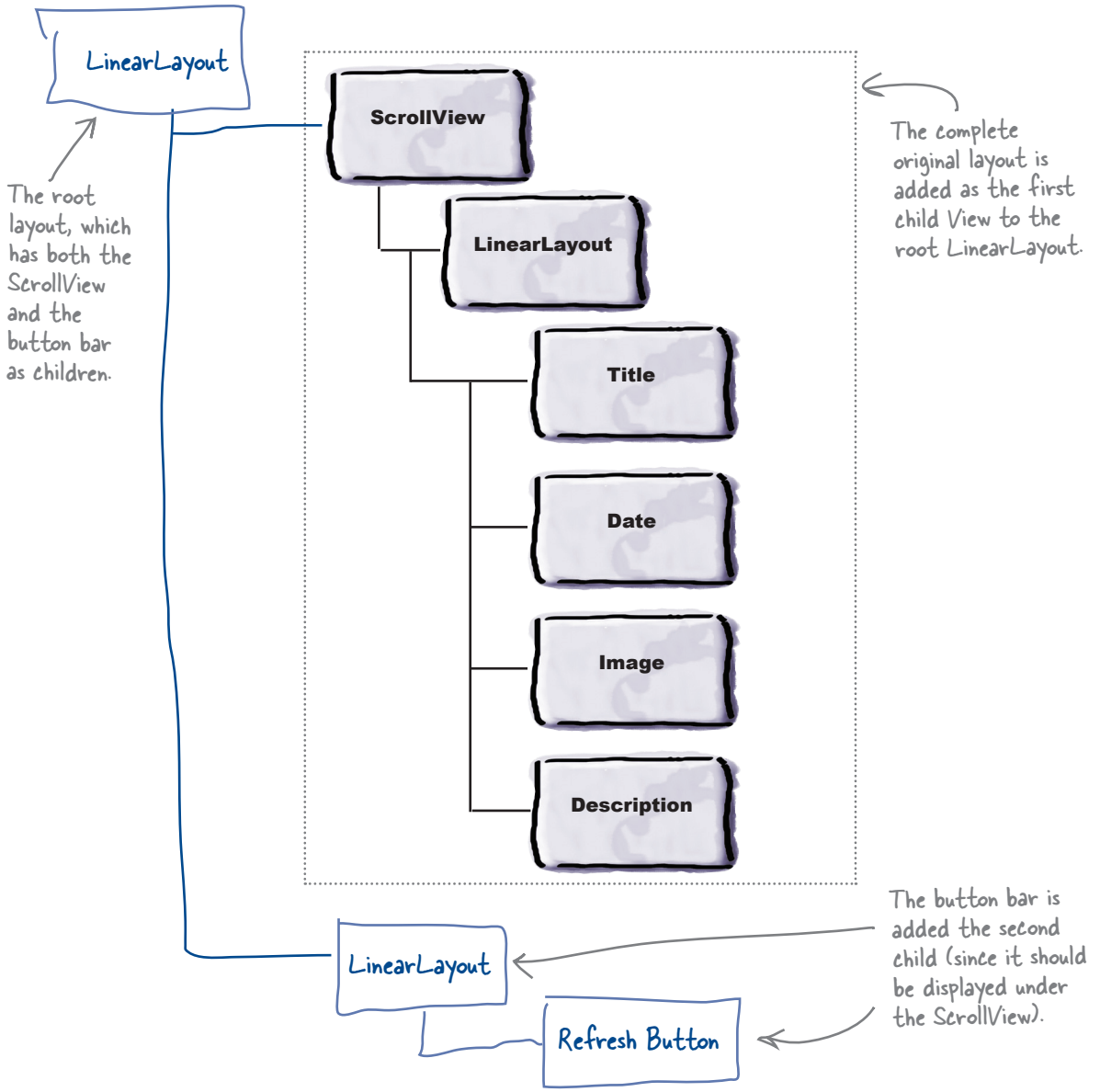
Add some spacing around the button inside the layout.

Center the button.

Both the width and height are set to wrap content, so the button will size as it needs to based on the button text

Exercise Solution

Now that you have the Button Bar layout, you needed to add it to your screen. Below is a graphical representation of your current View/Layout hierarchy. You were to draw new views and layouts for the button bar Views (and any other views and layouts you need) to complete your layout.



Update your app layout

Add the button bar to the app layout in *main.xml*. Also, add the wrapper `LinearLayout` in the root, and add the button bar and the `ScrollView` to that layout.

Update your layout in *main.xml*, adding the code for the button bar and the wrapper `LinearLayout`.

Beginning of wrapper layout

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

xmlns moved to root layout

Existing layout

```
<ScrollView xmlns:android="http://schemas.android.com/apk/res/android"
        android:layout_width="fill_parent"
        android:layout_height="fill_parent" "wrap_content"
        >
<LinearLayout
```

Height changed to wrap_content; otherwise, it would fill the screen, leaving no room for the button bar.

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

The complete button bar layout

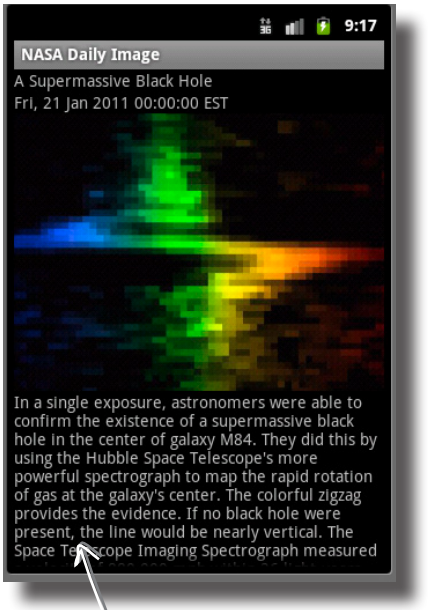
```
<LinearLayout
        android:layout_width="fill_parent"
        android:layout_height="wrap_content"
        android:background="#ff8D8D8D"
        android:layout_marginTop="5dp"
        android:padding="5dp" >
        <Button android:text="@string/refresh"
                android:onClick="onRefresh"
                android:layout_width="wrap_content"
                android:layout_height="wrap_content" />
</LinearLayout>
```

```
</LinearLayout>
```

End of wrapper layout

Test Drive

After you update your layout in *main.xml*, run the app to verify your layout updates.



Brain Barbell

There has ***got*** to be something going on here. The widths and height look OK, and the `LinearLayout` should be resizing everything... right? What could be wrong?

Use LinearLayout's weight property

`LinearLayout` lets you assign a weight property that controls the resizing behavior of its child Views. For the button bar, you want the button bar to be ***just*** as big as it needs to and then have the `ScrollView` **fill** the *entire rest of the screen*.

Weights are defined using the `android:layout_weight` XML attribute and have a number value of **0** or **1**. Using a weight of **1** makes the `View` stretch, while using **0** will make that `View` just as big as needed.

ScrollView definition

```
<ScrollView
        xmlns:android="http://schemas.android.com/apk/res/android"
        android:layout_width="fill_parent"
        android:layout_height="wrap_content"
        android:layout_weight="1"
```

A weight of 1 fills the screen with just enough space left for the button bar.

Button bar LinearLayout definition

```
<LinearLayout
        android:layout_width="fill_parent"
        android:layout_height="wrap_content"
        android:layout_weight="0"
```

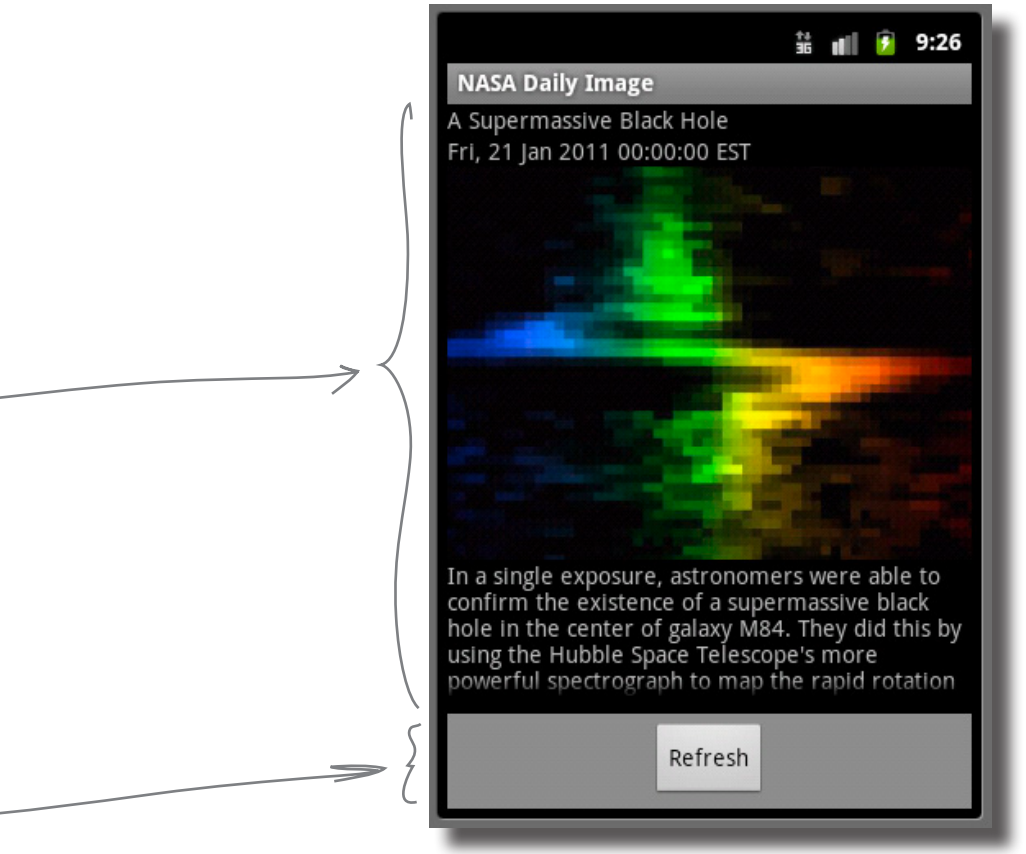
A weight of 0 makes the button bar just as big as needed.

Where do you find out about these properties?

All of the properties used here (and many, many more) are documented in the Android online documentation. To learn about more of these properties, look at the documentation for your specific layout as well as the layout tutorials. Do a quick search at **developer.android.com**, and you'll get right to it.

Test Drive

Run the app again, and check that the layout weight modifications made the desired layout changed.



Great work!

The app is looking fantastic. Now just wire up the refresh button and you can show it to Bobby.

Connect the refresh button

You already have the feed-handling code working from Chapter 2. To keep your code clean and concise (and without duplicate code), move the feed-handling code to a new method called `refreshFromFeed()`. Then you can call the same feed-processing method from `onRefresh()` ***and*** `onCreate()`.

```
public void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.activity_main);
        iotdHandler = new IotdHandler();
        iotdHandler.processFeed();
        resetDisplay(iotdHandler.getTitle(),
                iotdHandler.getDate(),
                iotdHandler.getUrl(),
                iotdHandler.getDescription());

        refreshFromFeed();
}
```

Move this code to a new method called refreshFromFeed().

Call refreshFromFeed from onCreate().

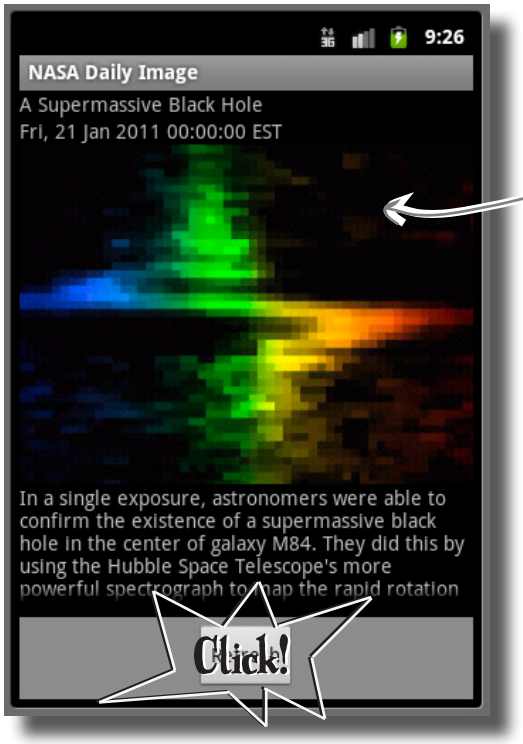
```
private void refreshFromFeed() {
        iotdHandler = new IotdHandler();
        iotdHandler.processFeed();
        resetDisplay(iotdHandler.getTitle(),
                iotdHandler.getDate(),
                iotdHandler.getUrl(),
                iotdHandler.getDescription())
}
```

```
public void onRefresh(View view) {
        refreshFromFeed();
}
```

Call the same refreshFromFeed from the button's onRefresh() method.

Test Drive

Run the app again, and click refresh. This will update the app from the feed.



It's not clear what's going on here...

Did the refresh work? Was the feed successfully processed? It's totally unclear what exactly happens here when the user clicks on the refresh button.

Use the debugger

The debugger is an incredibly useful tool for figuring out what's happening while your application is running. The Android Eclipse plugin includes tools to seamlessly use the built-in Eclipse debugger to debug your Android apps, either in the emulator or even on a device. Follow these steps to debug the app and see whether `refreshFromFeed()` is getting

This isn't intended to be a detailed debugger tutorial.

There is just enough detail here to debug the NASA app. Take a look at the Android and Eclipse documentation for more tips on using the Eclipse debugger.

1 **Set a breakpoint**

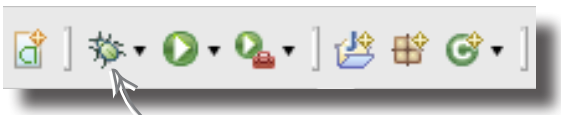
The debugger works by setting stopping points in your app called **breakpoints**. A breakpoint is like a scenic stop on a nice drive where you stop and take a look at what's going on in that spot.

```
            input.close();
            connection.disconnect();
            return bitmap;
        } catch (IOException ioe) {
            return null;
        }
    }

    public void refreshFromFeed() {
        iotdHandler = new IotdHandler();
        iotdHandler.processFeed();
        resetDisplay(iotdHandler.getTitle(), iotdHandler.getDate(), iotdHandler.getU
    }

    public void onRefresh(View view) {
         refreshFromFeed();
```

Double-click in the gray margin to set a breakpoint.

2 **Launch the debugger**

The debug button is just to the left of the play button in the Eclipse toolbar. It uses the Android launch configurations you already set up. Press it to launch the debugger.



Click this button to launch the debugger.

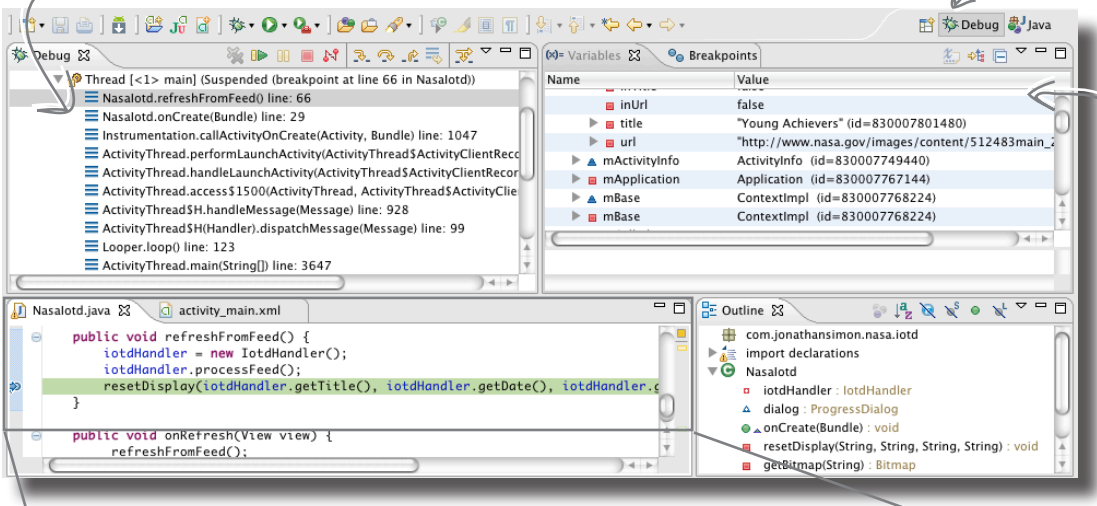
The debugger automatically deploys and attaches to your app (on the emulator or a device).

3 Monitor your app in the debug perspective

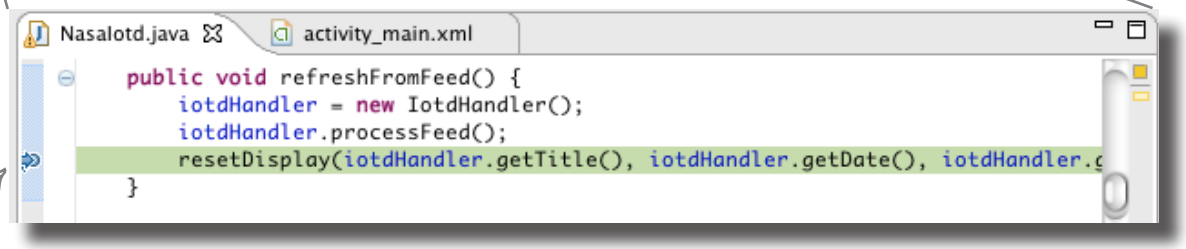
The debug **perspective** is where you can see the state of your app running. (A **perspective** is Eclipse's name for a stored collection of panels for specific work.) When you launch your app with the debugger, it will immediately hit a breakpoint, because `onCreate()` calls `refreshFromFeed()`, which is where you set your **breakpoint**.

This selector switched from Java to Debug, letting you know you're in the debug perspective. Click Java to take you back to the standard code perspective.

This view shows thread stack traces.



This view shows you the values of variables that are in scope.



The arrow next to the breakpoint indicator lets you know the line was reached.

So the line *was* reached... but how does the *user know*?

Add a progress dialog

The `ProgressDialog` is a utility that shows a **modal** progress pop-up with customized information for your app. `ProgressDialog` is perfect here, because you can show your users status, but you also keep them from repeatedly pressing refresh and successively triggering refresh after refresh.

Geek Bits

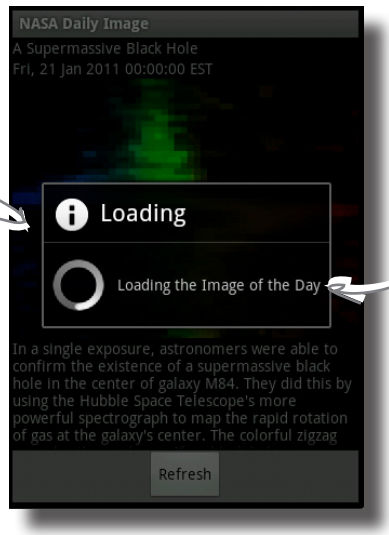
Modal means users can't interact with the application at all. All user input will be ignored.

How do you show a progress dialog?

Show a `ProgressDialog` by calling the static method **show** on `ProgressDialog`. The `show` method returns a reference to a `ProgressDialog` isntance. Make sure to cache the reference, as you'll need it to dismiss the dialog when you're done with it.

This is the code to show a progress dialog. Change the title and detail text as needed.

```
ProgressDialog dialog = ProgressDialog.show(
        this,
        "Loading",
        "Loading the image of the Day");
```



Call dismiss on the dialog when you've completed all of your work and the dialog will go away.

```
dialog.dismiss();
```

Call this to dismiss the dialog.

Sharpen your pencil

Below is the `refreshFromFeed` method with long-running code. Add the necessary code to show the `ProgressDialog` before the long-running work is shown. And remember to dismiss the dialog once the work is completed.

```
public void refreshFromFeed() {

Show the dialog here. →

        iotdHandler = new IotdHandler();
        iotdHandler.processFeed();
        resetDisplay(iotdHandler.getTitle(),
                iotdHandler.getDate(),
                iotdHandler.getUrl(),
                iotdHandler.getDescription());

The long-running work of the feed processing.

Dismiss the dialog here, now that all the work is done. →

}
```

Sharpen your pencil Solution

Below is the refreshFromFeed method with long running code. You were to add the necessary code to show the `ProgressDialog` before the long running work is shown. You should have also dismissed when dialog once the work is completed.

```
public void refreshFromFeed() {

    ProgressDialog dialog = ProgressDialog.show(
            this,
            "Loading",
            "Loading the image of the Day");

    iotdHandler = new IotdHandler();
    iotdHandler.processFeed();
    resetDisplay(iotdHandler.getTitle(),
            iotdHandler.getDate(),
            iotdHandler.getUrl(),
            iotdHandler.getDescription());

    dialog.dismiss();

}
```

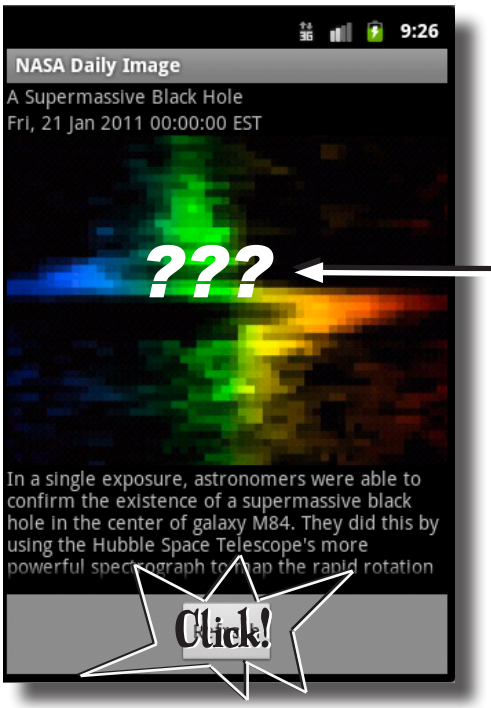
Show the progress dialog.

The feed and UI update code remains untouched.

Dismiss the progress screen, now that the work is done.

Test Drive

Run the app and click Refresh to verify that the `ProgressDialog` is working correctly.



***What*? no dialog after clicking refresh ...**

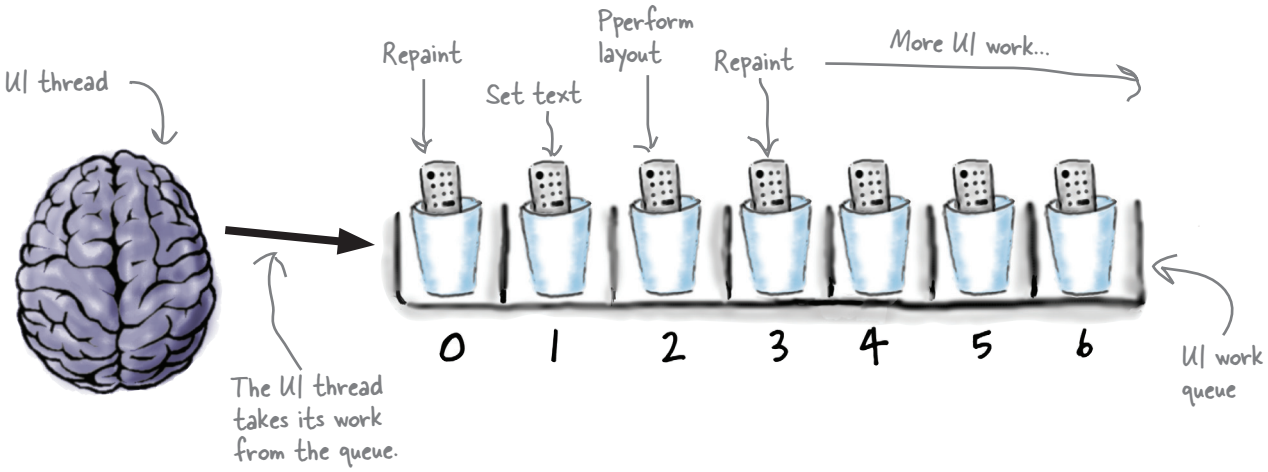
Well that's not good.

The whole point of putting in the `ProgressDialog` was to have it show while the long-running feed-processing work is occurring. The dialog code is in the right place, but for some reason it's not showing. ***What could be happening?***

The problem is in the threading...

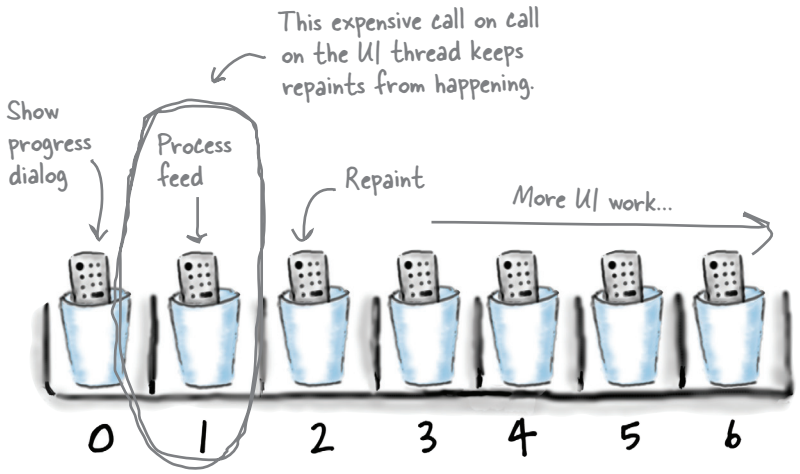
Dedicated UI thread

Android has a dedicated thread for updating the **user interface** (UI). It is responsible for repaints, layouts, and other graphical processing that helps keep the UI responsive and keeps animations smooth. The UI thread has a queue of work, and it continually gets the mot important chunk of work to process.



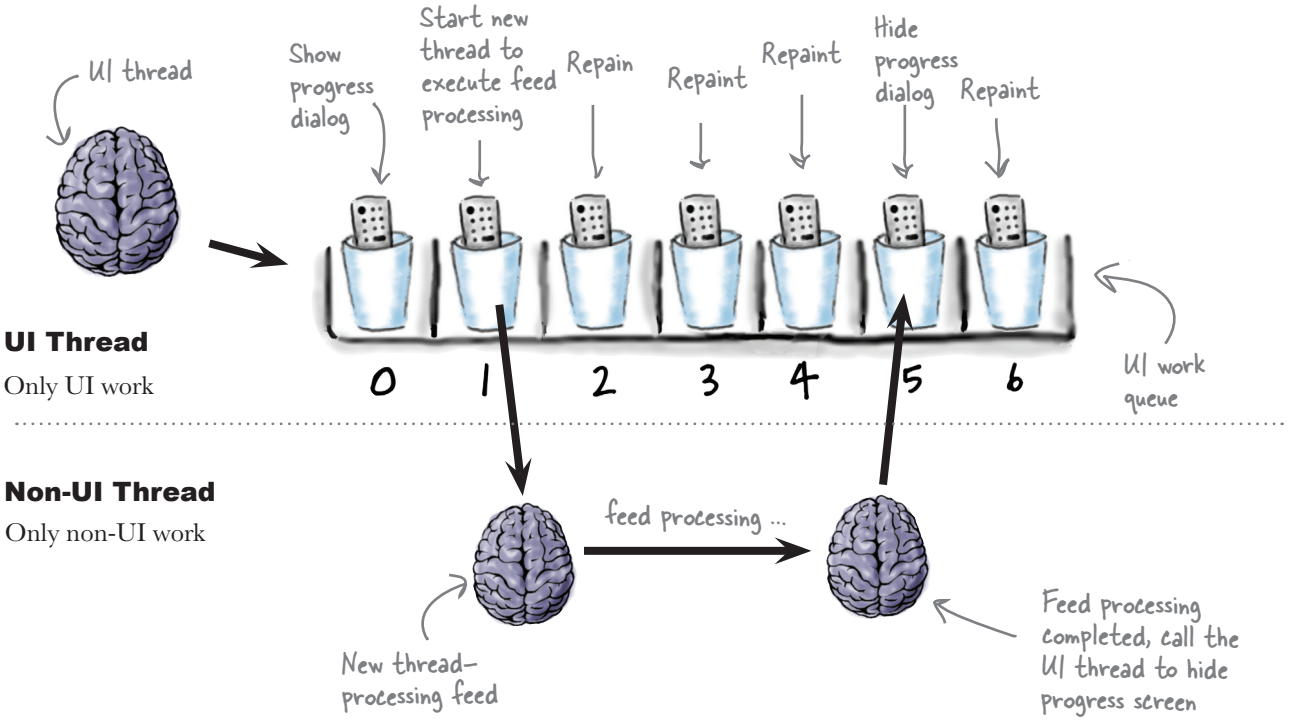
Why didn't the progress dialog display?

The button action occurs in the UI thread by default. When the progress screen is shown, successive calls to repaint the screen are made to support the animation effect. But the process feed code also runs in the UI thread, which occupies the UI thread. By the time the UI thread could run the repaint code, the dialog was hidden.



How do you fix it?

The solution is to keep non-UI work off the UI thread and all UI work on the UI thread.



Moving the feed processing work off the UI thread and onto a separate thread allows the UI thread to focus on repaints. The first repaint shows the progress dialog, and the successive repaints make the animation happen. Then, when the feed processing is completed, the new thread puts an item in the UI queue to hide the progress screen. This switch back to the UI thread is important, because the non-UI thread can't hide the dialog, which is a UI component.

Keep the UI thread free of expensive processing for a responsive UI.

Spawn a new thread for the long process

The most straightforward way to get your long-running processing code on a different thread than the UI thread is to make an inner class extending `Thread` and implementing the `run` method inline.

There are about a million different ways to structure your code to deal with threads. The goal here isn't to debate them, but to understand how to work with the Android UI thread.

```
public void refreshFromFeed() {
        dialog = ProgressDialog.show(
                this,
                "Loading",
                "Loading the Image of the Day");

        Thread th = new Thread() {
                public void run() {
                        if (iotdHandler == null) {
                                iotdHandler = new IotdHandler();
                        }
                        iotdHandler.processFeed();
                        resetDisplay(
                                iotdHandler.getTitle(),
                                iotdHandler.getDate(),
                                iotdHandler.getUrl(),
                                iotdHandler.getDescription());
                        dialog.dismiss();
                }
        };
        th.start();
}
```

Leave this code on the UI thread.

Extend thread.

Implement run.

All of the feed-processing goes on the new thread.

Don't forget to start your new thread.

Test Drive

Run the app again, now with the expensive feed-processing code moved to the new thread. The dialog *should* show... but when you run the app, you will see an error.

You'll see this error dialog when you run the app.

The error description

```
FATAL EXCEPTION: Thread-11
android.view.ViewRoot$CalledFromWrongThreadException: Only the original
thread that created a view hierarchy can touch its views.
        at android.view.ViewRoot.checkThread(ViewRoot.java:2932)
        at android.view.ViewRoot.requestLayout(ViewRoot.java:629)
        at android.view.View.requestLayout(View.java:8267)
```

What's the problem?

The problem is the dismissing of the `ProgressDialog`. Properly managing your work on and off the UI thread means ***not only*** getting expensive work **off** the UI thread, but also making sure that all necessary UI code occurs **on** the UI thread.

```
iotdHandler.processFeed();

resetDisplay(
        iotdHandler.getTitle(),
        iotdHandler.getDate(),
        iotdHandler.getUrl(),
        iotdHandler.getDescription());
dialog.dismiss();
```

This needs to occur on the UI thread.

Use Handler to get code on the UI thread

The `dialog.dismiss()` call needs to get back on the UI thread. Getting off of the UI thread is a cinch by creating a new thread. But that thread doesn't have a reference to the UI thread to get code to execute back on the UI thread after the expensive work. That's where **`Handler`** comes in.

`Handler` works by keeping a reference to the thread it was created by. You can pass it work and `Handler` ensures that the code is executed on the instantiated thread. (`Handler` actually works for ***more*** than just the UI thread.)

Start by instantiating a handler from the UI thread

The `onCreate()` method is called from the UI thread. Instantiate the `Handler` there, so you can get work back on the UI thread later.

onCreate method from the Nasalotd Activity

```
Handler handler;

public void onCreate(Bundle savedInstanceState) {
       super.onCreate(savedInstanceState);
       setContentView(R.layout.activity_main);

       handler = new Handler();
       refreshFromFeed();
}
```

Cache a Handler reference as a member variable, so you don't have to create Handlers over and over again.

onCreate() executes on the UI thread.

Since onCreate() executes in the UI thread, creating the Handler here makes a handler with the ability to execute code on the UI thread.

Pass work to the Handler using *post*

Once you have a `Handler` instance, you can call `post`, passing it a `Runnable` to execute on the desired thread.

```
handler.post(Runnable runnable)
```

This is a standard Runnable, nothing Android specific.

Get ready to fix refreshFromFeed() with correct threading...

Handler Magnets

Use the magnets below to complete `refreshFromFeed()` with all of the necessary threading changes. The expensive feed-processing code needs to execute on a new thread, and the call to dismiss the dialog has to happen on the UI thread using `Handler`. Assume the `Handler` was already instantiated for you in `onCreate()`.

Here are your magnets.

```
resetDisplay(iotdHandler.getTitle(),
        iotdHandler.getDate(), iotdHandler.getUrl(),
        iotdHandler.getDescription());

handler.post(

dialog.dismiss();

dialog = ProgressDialog.show(this,
        "Loading", "Loading the Image of the Day");

new Runnable () {

public void run() {

iotdHandler.processFeed();

public void run() {

});

if (iotdHandler == null) {
        iotdHandler = new IotdHandler();
}

Thread th = new Thread() {

th.start();
```

Handler Magnet Solution

You were to use the magnets below to complete `refreshFromFeed()` with all of the necessary threading changes. The expensive feed-processing code should be executing on a new thread, and the call to dismiss the dialog should be executing on the UI thread using `Handler`. Assume the `Handler` was already instantiated for you in `onCreate()`.

```
dialog = ProgressDialog.show(this,
       "Loading", "Loading the Image of the Day");
```

The dialog is called from the UI thread (where refreshFromFeed is called from).

Start a new thread for the actual feed code.

```
Thread th = new Thread() {
    public void run() {
        if (iotdHandler == null) {
            iotdHandler = new IotdHandler();
        }
        iotdHandler.processFeed();
        handler.post(
            new Runnable () {
                public void run() {
                    resetDisplay(iotdHandler.getTitle(),
                        iotdHandler.getDate(), iotdHandler.getUrl(),
                        iotdHandler.getDescription());
                    dialog.dismiss();
                }});
th.start();
```

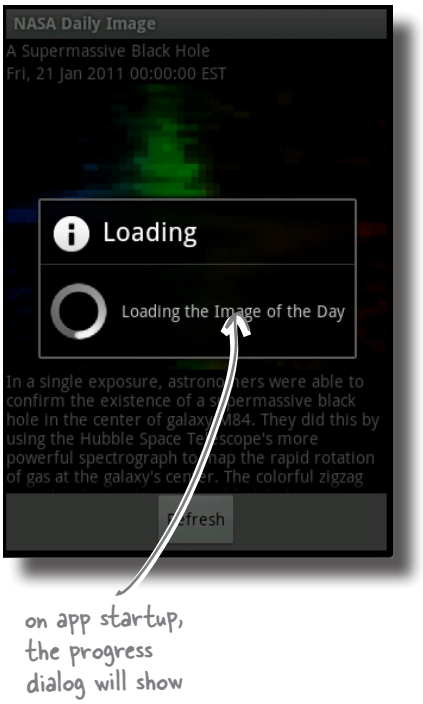
Post a new Runnable to the Handler.

Call resetDisplay and dismiss the dialog from the UI thread.

Test Drive

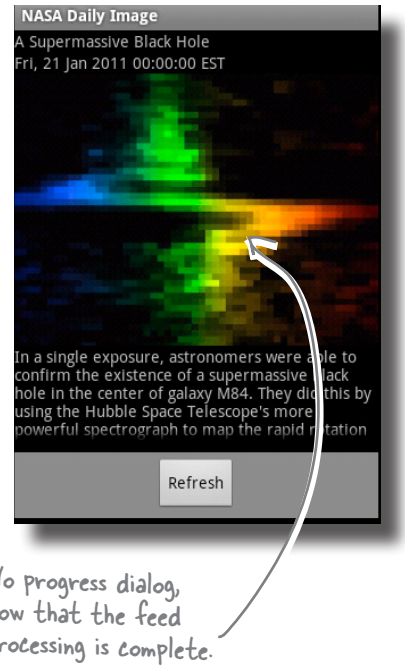
Now run the app and you'll see the progress screen show while the app loads from the feed during `onCreate()`. You'll also see the the progress screen show when you click the refresh button.

1. Start the app.



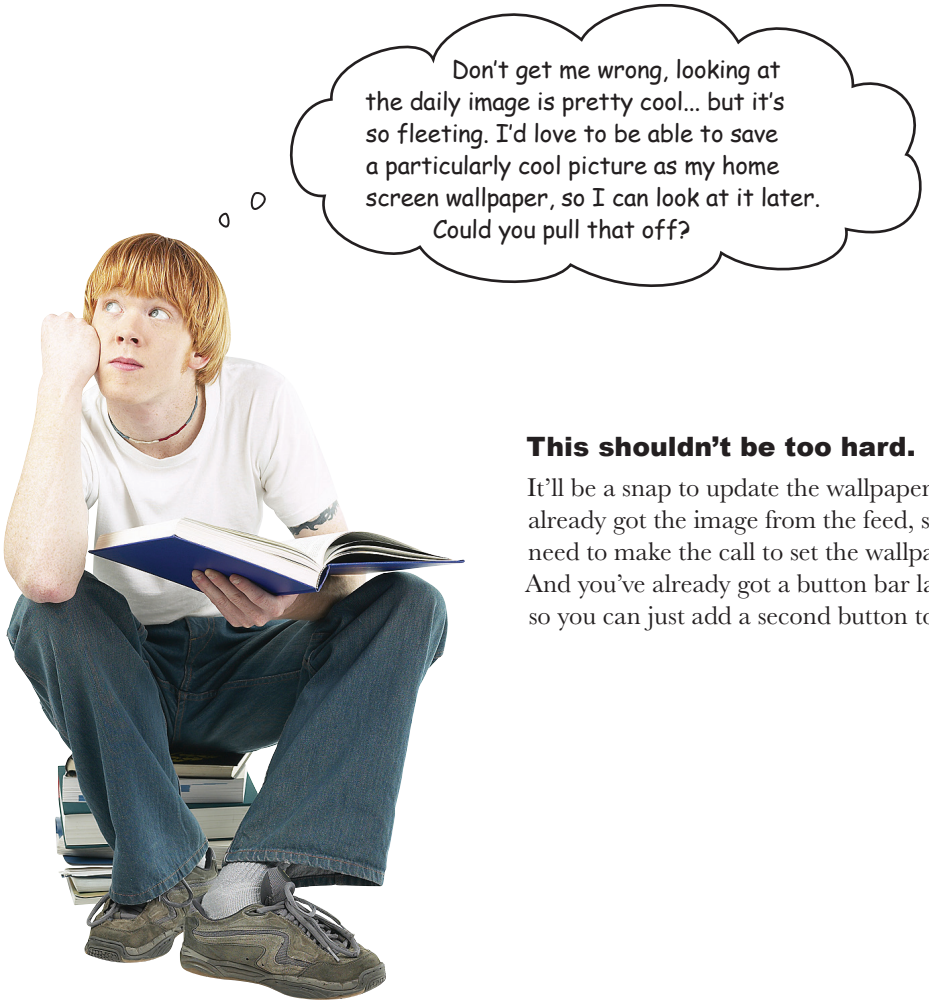
2. Give the app a few seconds to load the feed.

3. Watch the progress dialog get hidden.



Great work!

Now your users know that the app is doing something. Positive reinforcement goes a long way!



This shouldn't be too hard.

It'll be a snap to update the wallpaper. You've already got the image from the feed, so you just need to make the call to set the wallpaper using that. And you've already got a button bar layout in place, so you can just add a second button to the bar.

The code

You can set the wallpaper by retrieving the `WallpaperManager` and setting the wallpaper by `Bitmap`. You've already got a reference to the `Bitmap` coming from the feed, so this should be a piece of cake.

Ready Bake Code

This is the code to actually set the wallpaper.

```
WallpaperManager wallpaperManager =
      WallpaperManager.getInstance(this);
wallpaperManager.setBitmap( bitmap );
```

You can pass in the bitmap you decoded from the feed here.

"This" refers to the current activity.

The design

You already built the button bar to house the refresh button. And that is an ideal place to add a button to set the wallpaper. (More than two buttons in the button bar could be a problem if the button text is two long, but these two work great.)

The new set wallpaper button

Bobby's going to love this! Let's get started ...

Add the "Set Wallpaper" button

The button bar is built with a `LinearLayout`, so you can just add the new Set Wallpaper button directly to the button bar layout. `LinearLayouts` are horizontal by default, so you can add the **`android:orientation="horizontal"`** or simply rely on the default.

Add the new button to the buton bar layout in *main.xml*:

```
<LinearLayout
        android:orientation="horizontal"
        android:layout_width="fill_parent"
        android:layout_height="wrap_content"
        android:layout_weight="0"
        android:gravity="center_horizontal"
        android:background="#ff8D8D8D" >

        <Button android:text="@string/refresh"
                android:onClick="onRefresh"
                android:layout_width="wrap_content"
                android:layout_height="wrap_content" />

        <Button android:text="@string/setwallpaper"
                android:onClick="onSetWallpaper"
                android:layout_width="wrap_content"
                android:layout_height="wrap_content" />
</LinearLayout>
```

LinearLayout defaults to horizontal orientation, but it's a good idea to include the orientation attribute anyway. It makes your code easier to understand later and proptects you in case defaults change.

Add the "Set Wallpaper" button as the second child in the button bar layout. This will add it to the right of the refresh button.

Update *strings.xml* adding the new string for the Set Wallpaper button:

```
<string name="setwallpaper">Set Wallpaper</string>
```

Update the activity for the button action

The feed-processing code already downloads the image from the URL and creates a `Bitmap` from the web resource. To complete `onSetWallpaper` (the `onClick` call declared in the layout), cache the `Bitmap` once decoded and pass that image to the `WallpaperManager`.

```
public class NasaIotd extends Activity {
    private IotdHandler iotdHandler;
    ProgressDialog dialog;
    Handler handler;
    Bitmap image;
```

Make a member variable for the bitmap.

In refreshFromFeed()

```
iotdHandler.processFeed();
image = getBitmap(
        iotdHandler.getUrl());
```

Store the bitmap in the image member variable after processing the feed.

Add the `onSetWallpaper` method to your activity in `NasaIotd.java`:

```
public void onSetWallpaper(View view) {
    Thread th = new Thread() {
        public void run() {
            WallpaperManager wallpaperManager =
                    WallpaperManager.getInstance(NasaIotd.this);
            try {
                wallpaperManager.setBitmap(image);
            }
            catch (Exception e) {
                e.printStackTrace();
            }
        }
    };
    th.start();
}
```

Setting the wallpaper can take a while, so kick off a new thread to get it off the UI thread.

Since the current scope is an inner class; you can get a reference to "this" by preceding it with the class name.

This will do a default dump of the exception to LogCat.

Setting the wallpaper requires a `uses-permission` element with **`android.permission.SET_WALLPAPER`**. Set this now in *AndroidManifest.xml* before you run the app.

Test Drive

Run the app to make sure the Set Wallpaper button is correctly configured in the layout.

First check that the button displays correctly...

The button is displaying correctly, horizontally positioned next to the Refresh button.

The button looks good. Now check and see how it works!

What? Nothing happened. I clicked Set Wallpaper and now the app is just sitting there!

The button did actually work, *but*...

If you go to the home screen, you'll see that the wallpaper was in fact set to the feed image. That said, the **user experience** is *awful*! Remember that getting your app working is just one part of bigger picture. In order to make successful apps that people want to use (and that will make you bags of money on the Android market!), you need to have a fantastic user experience.

The issue here in setting the the wallpaper is that the change is happening *off screen* away from the user's view. What you need is some **positive reinforcement** so your users ***know*** it worked.

You could just show a ProgressDialog while the wallpaper is being set, but there's a better way...

Click on the home screen and you will see that the wallpaper was in fact set to the NASA feed image. Now to deal with the user experience...

Use toast to give users reinforcement

You could show a progress screen while the wallpaper is being set. But one of the inherent features of the progress screen is that it blocks users from doing anything. This is great when the feed is loading, because you want to block your users from interacting with the app. (This is what keeps users from repeatedly clicking on refresh.)

But setting the wallpaper is different. You want to make sure to notify your users when the wallpaper is set, but you don't want to keep them from doing something else in the app. For example, it would be perfectly acceptable for the user to set the wallpaper and to scroll down to view the long description while the wallpaper is being set in the background. This wouldn't be possible if you used a progress dialog, because it blocks all user interaction.

Toast

= passive notifications

Progress Dialog

= active, blocking notifications

Android provides Toast for just such occasions

Toast is a passive, non-blocking user notification that shows a simple message at the bottom of the user's screen. A toast typically displays for a few seconds and disappears. Meanwhile, the user can still completely interact with the application. Here is what the app would look like with a Toast message when the wallpaper is set and the code to make it happen.

```
Toast.makeText(this,
               "Wallpaper set",
               Toast.LENGTH_SHORT).show();
```

Pass in your activity.

Message text

Time to display the toast.

Complete the `onSetWallpaper` method below adding two `Toast` notifications: one for success and one in case of failure in the `catch` block. The `Toast` call must be made from the UI thread. Use the `Handler` reference cached previously to make both of the toast calls on the UI thread.

```
public void onSetWallpaper(View view) {
       Thread th = new Thread() {
              public void run() {
                     WallpaperManager wallpaperManager =
                            WallpaperManager.getInstance(NasaIotd.this);
                     try {
                            wallpaperManager.setBitmap(image);
```

Add the code here to create the toast confirmation message for setting the wallpaper.

```
                     } catch (Exception e) {
                            e.printStackTrace();
```

Add a toast message in the catch block with the message "Error setting wallpaper."

```
       }}};
       th.start();
}
```

You were to complete the `onSetWallpaper` method below adding two `Toast` notifications: one for success and one in case of failure in the `catch` block. The `Toast` call must be made from the UI thread. You should have used the `Handler` reference cached previously to make both of the toast calls on the UI thread.

```
public void onSetWallpaper(View view) {
        Thread th = new Thread() {
                public void run() {
                        WallpaperManager wallpaperManager =
                                WallpaperManager.getInstance(NasaIotd.this);
                        try {
                                wallpaperManager.setBitmap(image);
                                handler.post(
                                        new Runnable () {
                                                public void run() {
                                                        Toast.makeText(NasaIotd.this,
                                                                "Wallpaper set",
                                                                Toast.LENGTH_SHORT).show();
                                        }});
                        } catch (Exception e) {
                                e.printStackTrace();
                                handler.post(
                                        new Runnable () {
                                                public void run() {
                                                        Toast.makeText(NasaIotd.this,
                                                                "Error setting wallpaper",
                                                                Toast.LENGTH_SHORT).show();
                                        }});
        }}};
        th.start();
}
```

Use Nasaiotd.this to get a reference to the Activity from the inner class.

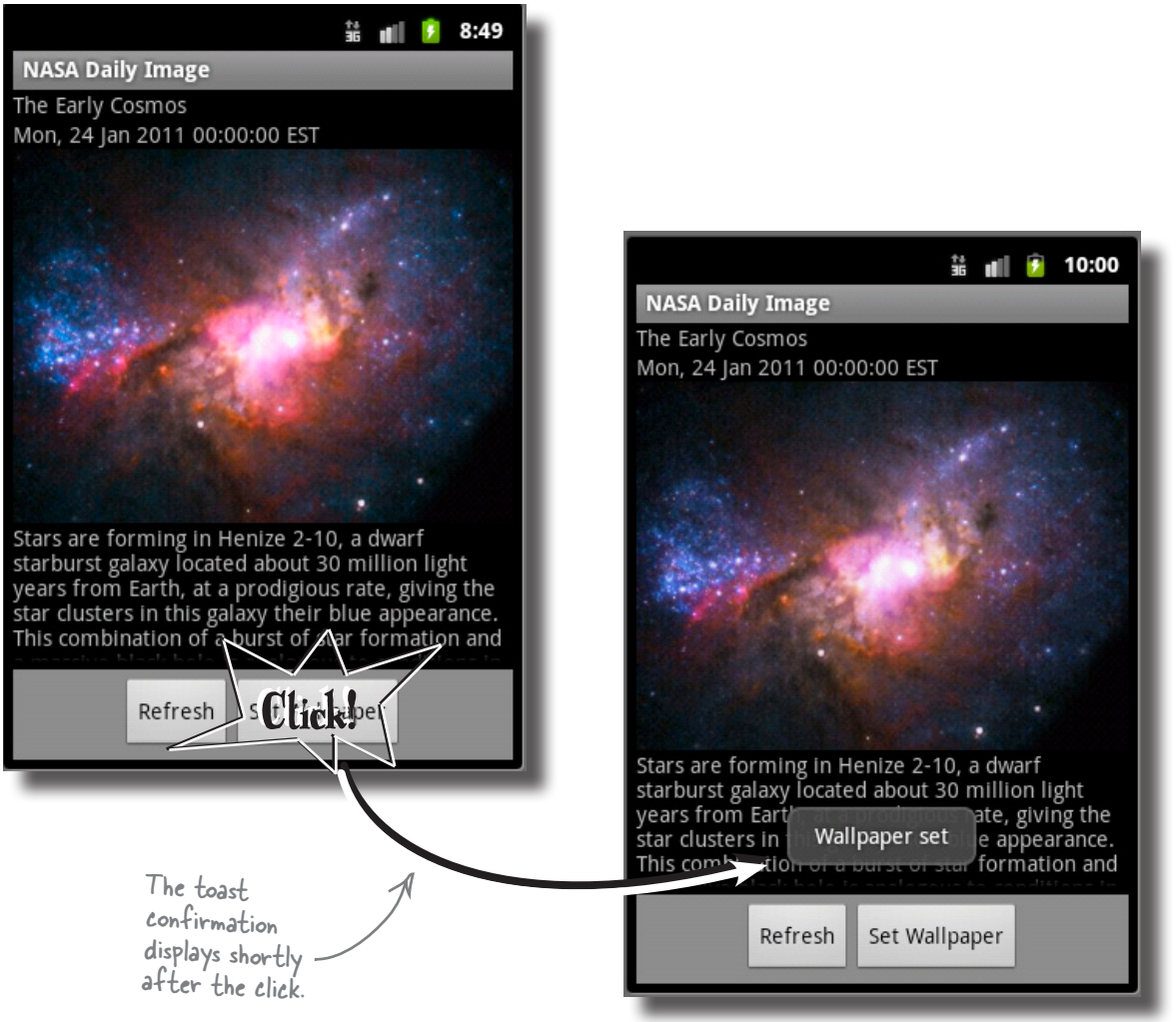
Make a confirmation toast

Use the handler to post runnables to the UI thread.

Show another toast if an exception is caught.

Test Drive

Run the app and click the Set Wallpaper button. Now you will see the wallpaper set *and* a nice toast conformation that lets you and your users know.



Fantastic work! Bobby and all of his friends are going to love this!


The app rocks... you totally came through! I think it's time to share this with more people than just my friends, though. Can you get it on the Market?
Sounds great! Next stop... the Android Market!
You'll have the NASA app up on the market in the next chapter. Stay tuned!
android
market

Your Android Toolbox

With proper threading and user feedback, you can guarantee your users a responsive app with a rock solid user experience.

The UI Thread

- Keep expensive work off the UI thread; otherwise, the responsiveness of the UI will suffer.
- Make sure all UI work occurs only on the UI thread. Calling UI code from non-UI threads will throw exceptions throughout your code.

Give your users feedback

- Toast: Use toast to passively display a message to your users
- ProgressDialog: Use a ProgressDialog when you want to block user input and display a message and progress on the screen.

BULLET POINTS

- Use extended properties of `LinearLayout` to fine-tune your screens (`padding`, `margin`, `background`, `gravity`, and more).
- Define layout width and height using `fill_parent` and `wrap_content`. Use `fill_parent` to maximize the size to fill the parent. Use `wrap_content` to make a View just as big as it needs to be.
- Use Density Independent Pixels (DIPs) when you need to define sizing or dimensions. This will ensure your layouts work on the most possible number of devices .
- Layouts can nest (you can add layouts as Views to other layouts). Just remember that too much nesting will slow down the layout and rendering of your screens. So use nested layouts with caution. (You'll learn strategies for this in later chapters.)
- Use the debugger to trace code in the emulator or a device.
- Use a `ProgressDialog` to block users and display progress.
- Use `Toast` to passively notify users of progress.
- Both `Toast` and `ProgressDialog` can be extensively customized for your app.
- Keep expensive work off the UI thread, and UI work only on the UI thread
- Use `Handler` to add UI work to the UI thread's queue from non-UI threads.