marakana

Android Bootcamp Courseware

by Marko Gargenta

Learn to Develop Mobile Apps for Android

marakana.com

Marakana Android Bootcamp

Contents

Chapter 1

Main Building Blocks

In this chapter, you will learn about the big blocks in Android. We'll give you a high-level overview of what Activities are, how Intents work, why Services are cool, how to use Broadcast Receivers and Content Providers to make your app scale, and much more.

By the end of this chapter, you will understand the main Android components for building applications. You should conceptually know when you'd use what component. You will also see how these components relate to a real world application.

1.1 Main Building Blocks

Main building blocks are components that you as application developer use to build Android apps. They are the conceptual items that you put together to create a bigger whole. When you start thinking about your application, it is good to approach it top down. You design your application in terms of screens, features, and interactions between them. You start with conceptual drawing, something that you can represent in terms of "lines and circles". This approach to application development helps you see the big picture - how the components fit together and it all makes sense.

1.2 A Real World Example

Let's say that I want to build a Twitter app. I know that the user should be able to post hers status updates. I also know she should be able to see what her friends are up to. Those are basic features. Beyond that, she should also be able to set her username and password in order to login to her Twitter account. So, now I know I should have these three screens.

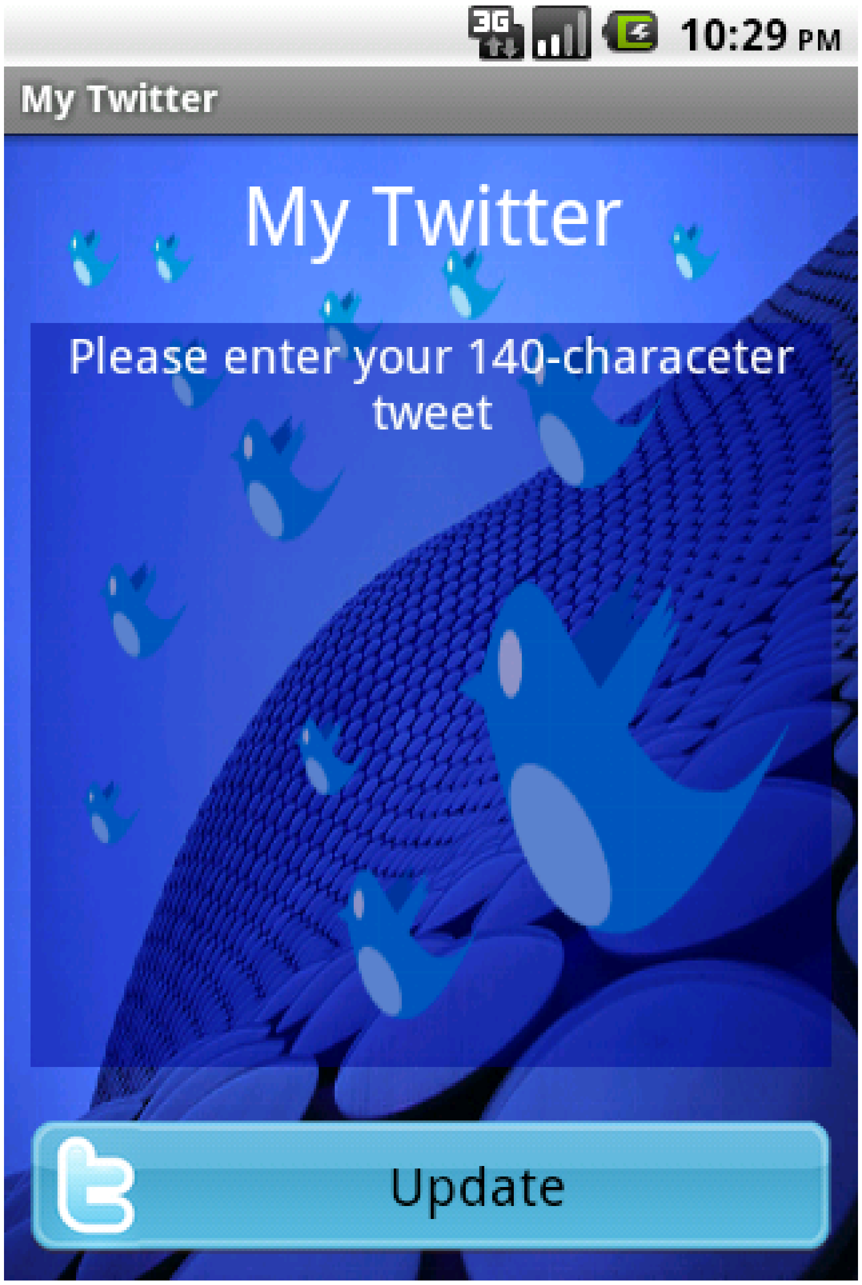
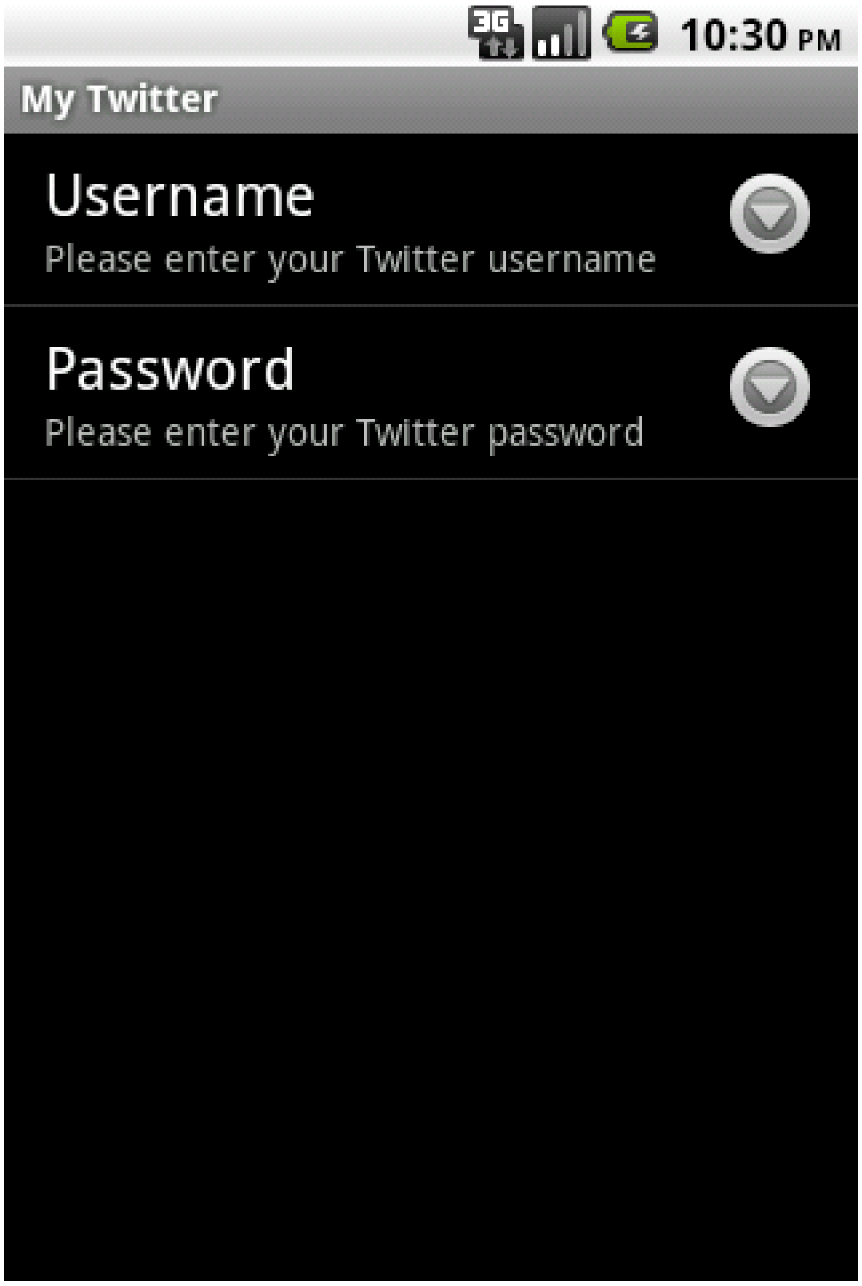


Figure 1.1: Status Activity



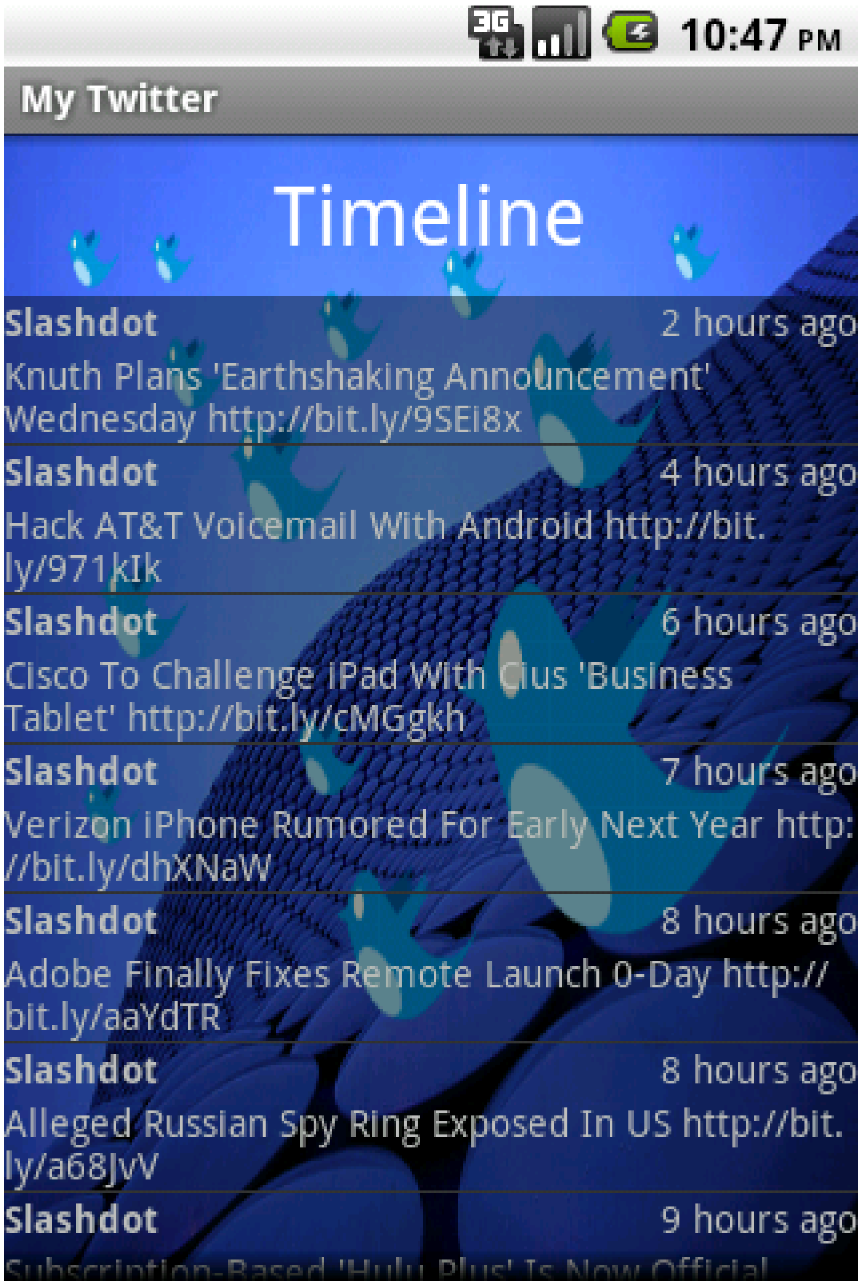
Username

Please enter your Twitter username

Password

Please enter your Twitter password

Figure 1.2: Preference Activity



Slashdot 2 hours ago
Knuth Plans 'Earthshaking Announcement' Wednesday http://bit.ly/9SEi8x

Slashdot 4 hours ago
Hack AT&T Voicemail With Android http://bit.ly/971kIk

Slashdot 6 hours ago
Cisco To Challenge iPad With Cius 'Business Tablet' http://bit.ly/cMGgkh

Slashdot 7 hours ago
Verizon iPhone Rumored For Early Next Year http://bit.ly/dhXNaW

Slashdot 8 hours ago
Adobe Finally Fixes Remote Launch 0-Day http://bit.ly/aaYdTR

Slashdot 8 hours ago
Alleged Russian Spy Ring Exposed In US http://bit.ly/a68JvV

Slashdot 9 hours ago
Subscription-Based 'Hulu Plus' Is Now Official

Figure 1.3: Timeline Activity

Next, I figured I would like my app to work fast regardless of network connection, or lack of. To achieve that, the app has to pull the data from Twitter when it's online, and cache it locally. That will require a service that runs in the background as well as a database.

I also know that I'd like this background service to be started when the device is initially turned on, so by the user first uses the app, there's already up-to-date information on her friends.

So, these are some straight froward requirements. Android building blocks make it easy to break them down into conceptual units so that you can work on them independently, and can also easily put them together into a complete package.

1.3 Activities

An activity is usually a single screen that user sees on the device at one time. An application will typically have multiple activities and user will be flipping back and forth among them. As such, activities are the most visible part of your application.

I usually use a website as an analogy for activities. Just like a website consists of multiple pages, so does an Android application consists of multiple activities. Just like a website has a "home page", an Android app has a "main" activity. It is usually the one that is shown first when you launch the application. And just like a website has to provide some sort of navigation among various page, an Android app should do the same.

On the web, you can jump from a page on one website to a page on another. Similarly, in Android, you could be looking at an activity of one application, but shortly after you could start another activity in a completely separate application. For example, if you are in your Contacts app, and you choose to text a friend, you'd be launching the activity to compose a text message in the Messaging application.

1.3.1 Activity Lifecycle

Launching an activity is quite expensive. It is expensive because it involves a lot of work, such as creating a new linux process, allocating memory for all the UI objects, inflating all the objects from XML layouts, and setting the whole screen up. Since we're doing a lot of work to launch an activity, it's be a waste to just toss it out once user leaves that screen. To avoid this waste, activity lifecycle is managed via Activity Manager.

Activity manager is responsible for creating, destroying, and overall managing activities. For example, when the user wants to start an application first time, the activity manager will create its activity and put it on the screen. Later, when the user switches screens, the activity manger will move that previous activity to a holding place. This way, if the user wants to go back to an older activity, it can be started quicker. Older activities that user hasn't used in a while will be destroyed in order to free more space for the currently active one. This mechanism is designed to help improve the speed of user interface and thus improve the overall user experience.

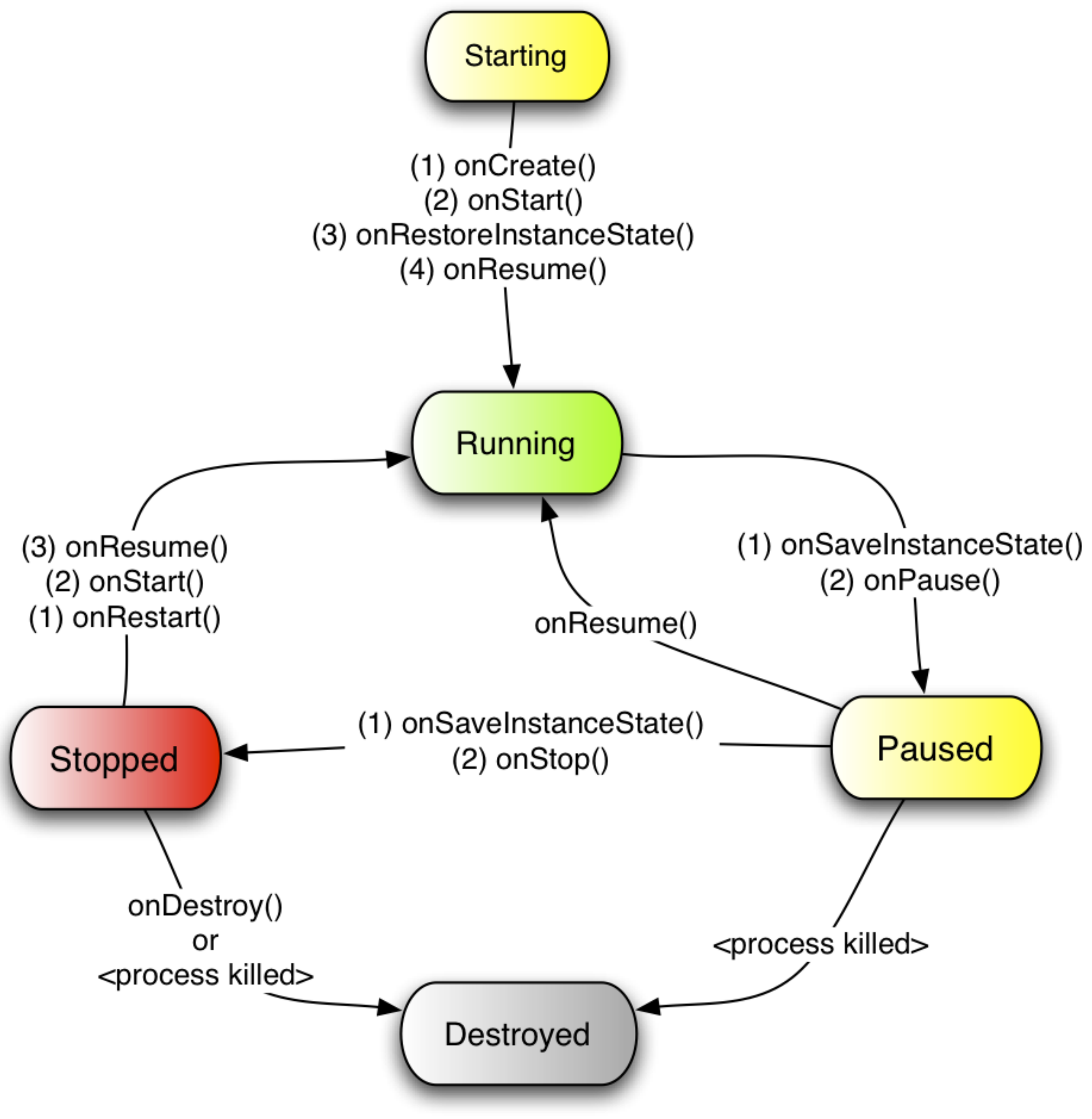


Figure 1.4: Activity Lifecycle

Programming for Android is conceptually different than programming for some other environments. In Android, you find yourself more responding to certain changes in the state of your application rather than driving that change yourself. It is a managed, container-based environment similar to programming for Java applets or servlets. So, when it comes to activity lifecycle, you don't get to say what state the activity is in but you have plenty of opportunity to say what happens on transitions from state to state.

1.3.1.1 Starting Up

When an activity doesn't exist in memory, it is in *Starting* state. While it's starting up, the activity will go through a whole set of callback methods that you as developer have an opportunity to fill out. Eventually, the activity will be in *Running* state.

Keep in mind that this transition from *Starting* to *Running* is one of the most expensive operations in terms of computing time needed. This is the exact reason why we don't automatically destroy activities that are no longer shown. User may want to come back to them, so we keep them around for awhile.

1.3.1.2 Running

A *Running* activity is the one that is currently on the screen interacting with the user. We also say this activity is in focus, meaning that all user interactions, such as typing, touching screen, clicking buttons, are handled by this one activity. As such, there is only one running activity at one time.

The running activity has all the priorities in terms of getting memory and resources needed to run as fast as possible. This is because we want to make sure the running activity is zippy and responsive to user.

1.3.1.3 Paused

When an activity is not in focus (i.e. not interacting with the user) but still visible on the screen, we say it's in *Paused* state. This is not a very typical scenario since the screen is usually small and an activity is either taking the whole screen or not at all. But, non the less, this is a valid state and all activities go through it en route to being stopped.

Paused activities still have high priority in terms of getting memory and other resources. This is because they are visible and cannot be removed from the screen without making it look very strange to the user.

1.3.1.4 Stopped

When an activity is not visible, but still in memory, we say it's in *Stopped* state. Stopped activity could be brought back to front to become *Running* activity again. Or, it could be destroyed and removed from memory.

System keeps activities around in *Stopped* state because it is likely that the user will still want to get back to those activities some time soon. And restarting a stopped activity is far cheaper than starting an activity from scratch. That is because we already have all the objects loaded in memory and just simply have to bring it all up to foreground.

Stopped activities can also, an any point, be removed from memory.

1.3.1.5 Destroyed

A destroyed activity is no longer in memory. The activity manager decided that this activity is no longer needed, and as such has removed it. Before the activity is destroyed, you, the developer, have an opportunity to perform certain actions, such as save any unsaved information.

Note
The fact that an activity is in *Running* state doesn't mean it's doing much. It could be just sitting there and waiting for user input. Similarly, a *Stopped* activity is not necessarily doing nothing - it could be still actively getting sensor updates and processing them, for example. The state names mostly refer to how active the activity is with respect to user input.

1.4 Intents

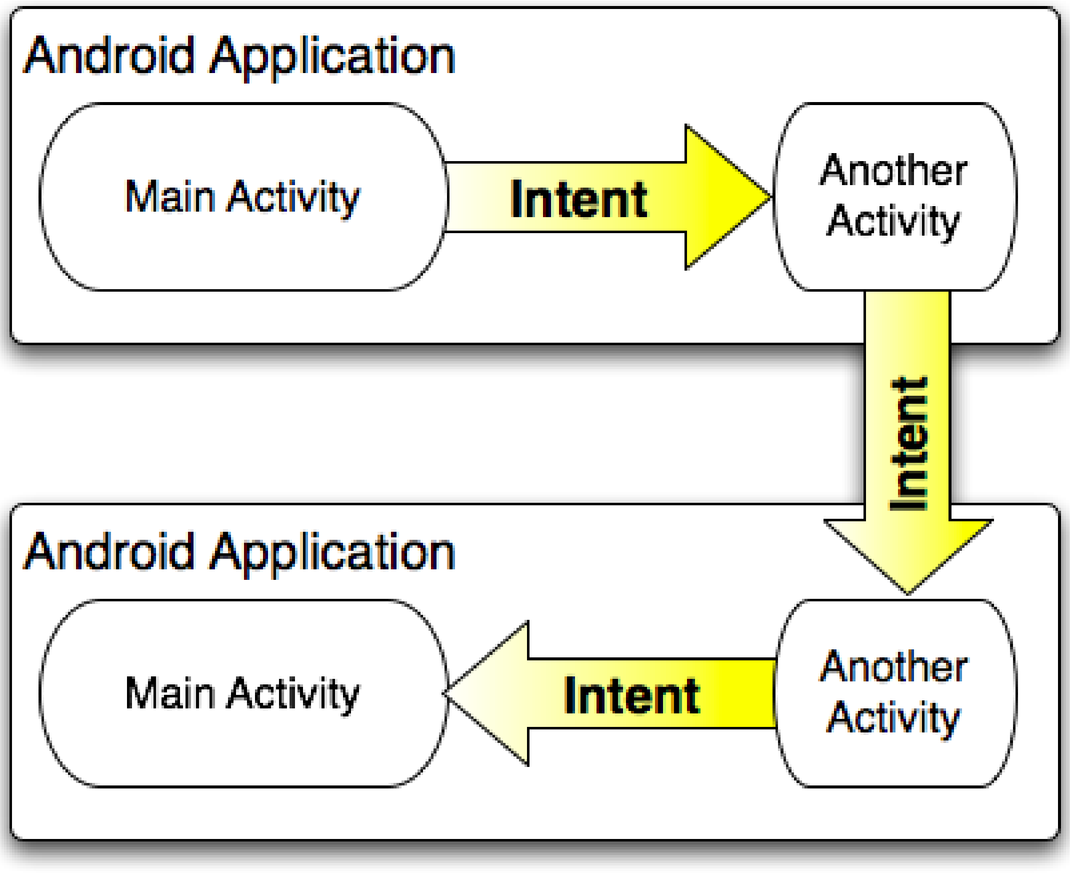


Figure 1.5: Intents

Intents are messages that are sent among major building blocks. They trigger an activity to start up, a service to start or stop, or are simply broadcasts. Intents are asynchronous, meaning the code that is sending them doesn't have to wait for them to be completed.

An intent could be explicit or implicit. In an explicit intent the sender clearly spells out which specific component should be on the receiving end. In an implicit intent, the sender specifies the type of receiver. For example, your activity could send an intent saying it simply wants someone to open up a web page. In that case, any application that is capable of opening a web page could "compete" to complete this action.

What this type of messaging allows for is to allow the user to replace any app on the system with a custom one. For example, you may want to download a different SMS application, or another browser to replace your existing ones.

When you have competing applications, the system will ask you which one you'd like to use to complete a given action. You can also set an app as the default one. This mechanism works very similarly to your desktop environment, when you downloaded Firefox or Chrome to replace your default Internet Explorer or Safari web browsers.

1.5 Services

Services are background processes. They don't have any user interface components. Services are useful for actions that we want to make sure performs for a while, regardless of what is on the screen. For example, you may want to have your music player play music even as you are flipping between other applications. Or, in our Twitter example, we may want to have the Twitter app be checking for the latest updates from our friends even when we are doing something else.

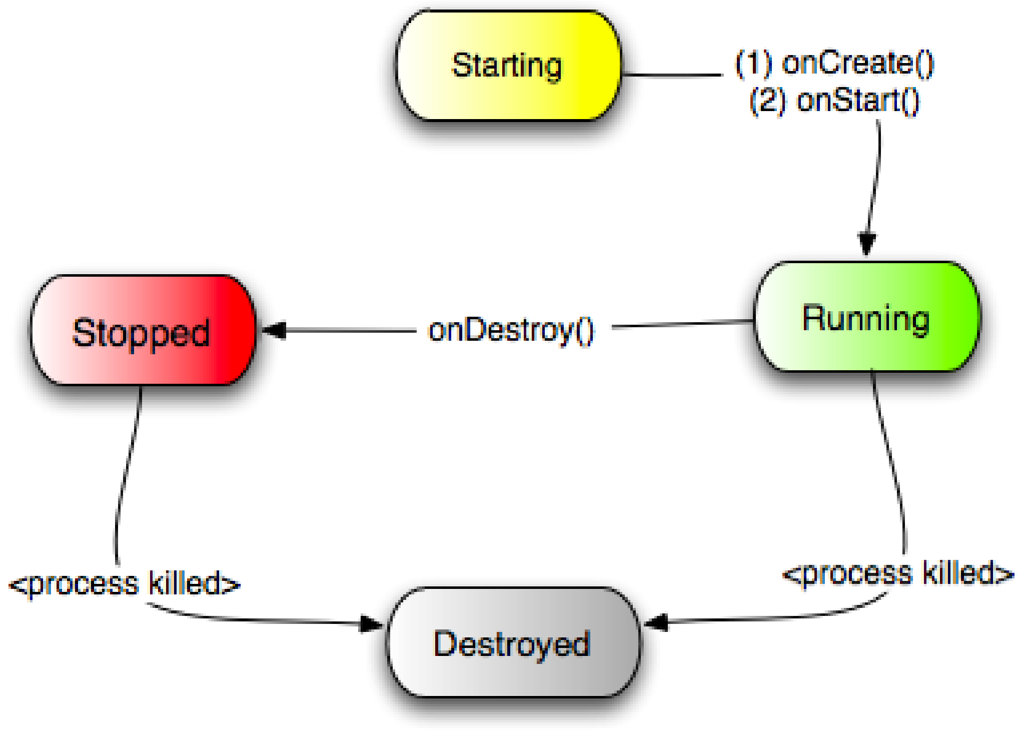


Figure 1.6: Service Lifecycle

Services have a much simpler lifecycle than activities. You start a service, or stop it. Also, the service lifecycle is more or less controlled by the developer, and not so much by the system. The reason for this is that services tend to be easier on resources than activities.

The fact that a service runs in the background doesn't mean it runs on a separate thread. So, if your service is doing some processing that takes a while to complete (such as perform network calls), you may want to consider running it on a separate thread. Otherwise, your user interface will run noticeably slower.

1.6 Content Providers

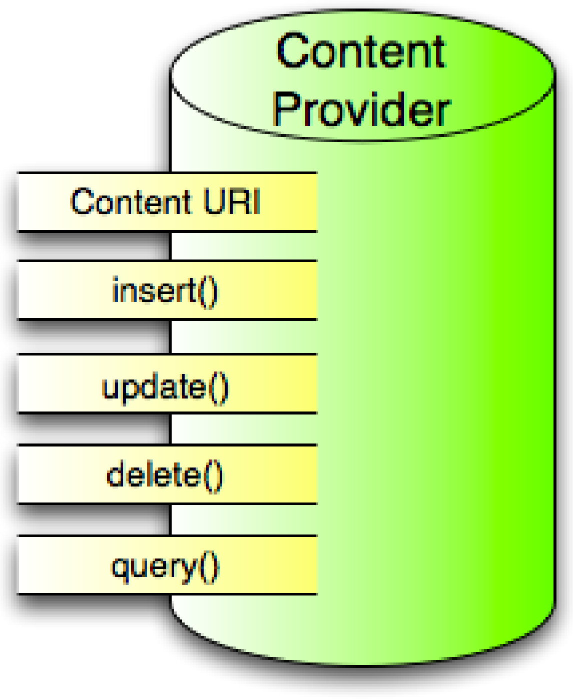


Figure 1.7: Content Provider

Content Providers are interfaces for sharing data between applications. Android by default runs each application in its own sandbox so that all data that belongs to an application is totally isolated from other applications on the system. If you want to expose certain data to another application, Content Providers offer a mechanism to do so.

Android system uses this mechanism all the time. For example, Contacts Provider is a content provider that exposes all users contacts data to various applications. Settings Provider exposes system settings to various applications including the built-in Settings application. Media Store is responsible for storing and sharing all various media, such as photos, and music across various applications. This separation of data storage and the actual user interface application offers greater flexibility to mash up various parts of the system.

Content Providers are a relatively simple interface with the standard `insert()`, `update()`, `delete()` and `query()` methods. These methods look a lot like standard database methods, so it is relatively easy to implement a content provider as a proxy to the database. Having said that, you are much more likely to use content providers than write your own.

1.7 Broadcast Receivers

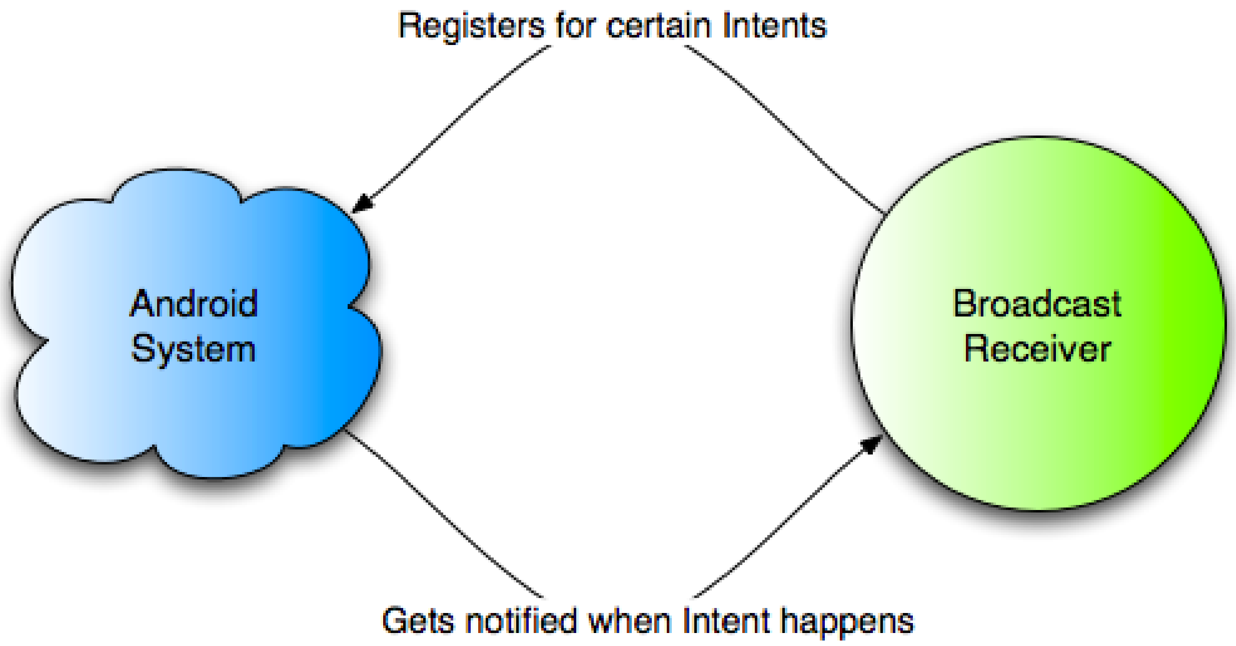

Registers for certain Intents
Android System
Broadcast Receiver
Gets notified when Intent happens

Figure 1.8: Broadcast Receiver

Broadcast Receivers is an Android implementation of system-wide publish/subscribe mechanism. The receiver is simply a dormant code that gets activated once an event it is subscribed to happens.

The system itself broadcasts events all the time. For example, when an SMS arrives, or call comes in, or battery runs low, or system gets booted, all those events are broadcasted and any number of receivers could be triggered by them. In our Twitter app, we want to start the update service once the system starts up. To do that, we can subscribe to the broadcast that tells us the system has completed booting up.

You can also send your own broadcasts from one part of your application to another, or a totally different application.

Broadcast receivers themselves do not have any visual representation. Nor they are actively running in memory. But when triggered, they get to execute some code, such as start an activity, a service, or something else.

Chapter 2

MyTwitter Project Overview

The sample project application for this course is a twitter application. The reason for this particular application is two-fold:

1. Twitter example covers most of the main Android building blocks in a natural way. As such, it's a great sample application to illustrate how various components work individually as well as fit together.
2. Twitter is more or less ubiquitous to most people, so the features of the application do not require much explaining.

2.1 Project Design

This is the design of the entire MyTwitter application, explained in lines and circles.

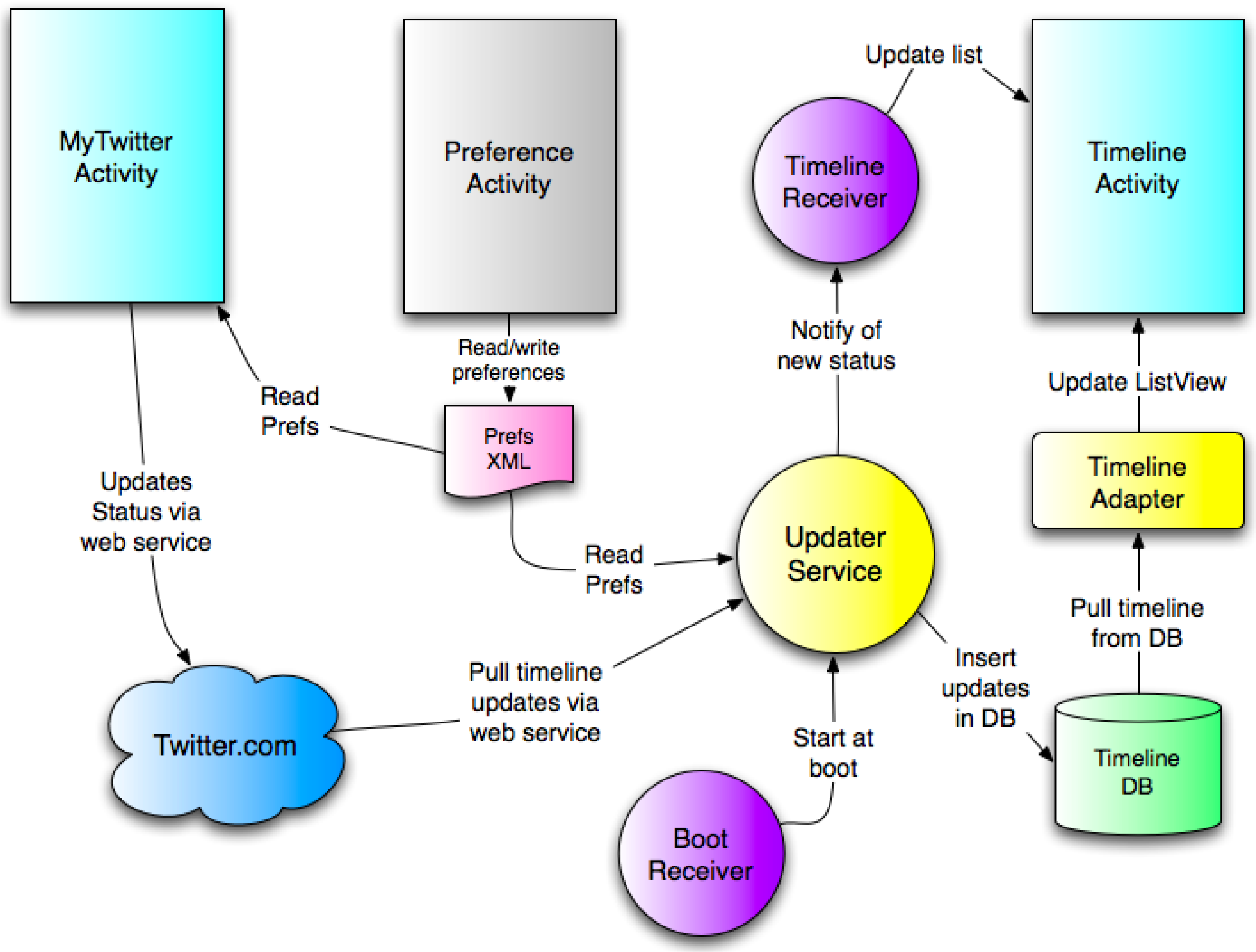


Figure 2.1: MyTwitter Design Diagram

2.2 MyTwitter Part 1

Part one introduces a single activity, StatusActivity. We build a simple layout, and handle the user events. We also add JTwitter external Jar to our project and connect to twitter.com to post a status update.

2.3 MyTwitter Part 2

First, we spice up the user interface a bit, add some graphics and images. Next, we add preferences activity to that user can change her twitter username and password. We also introduce the options menu in Android.

We also learn about Android services and implement UpdaterService.

2.4 MyTwitter Part 3

We introduce SQLite database and the DbHelper. We add TimelineActivity to display the list of friends' statuses. We also create a custom TimelineAdapter to make it all look good.

2.5 MyTwitter Park 4

We finally introduce broadcast receivers, and wrap up the project into a shippable application.

Chapter 3

MyTwitter Part 1

Part one introduces a single activity, StatusActivity. We build a simple layout, and handle the user events. We also add JTwitter external Jar to our project and connect to twitter.com to post a status update.

3.1 StatusActivity Layout

Let's start by designing the user interface for our screen where we'll enter the new status and click a button to update it.

This screen will have four components:

- Title at the top of the screen. This will be a `TextView` widget.
- Big text area to type our 140-character status update. We'll use `EditText` for this purpose.
- Button to click to update the status. This will be a `Button` widget.
- A layout to contain all these widgets and lay them out one after another in vertical fashion. For this screen, we'll use `LinearLayout`, one of the more common ones.

The source code for our StatusActivity layout looks like this:

res/layout/main.xml

```xml
<?xml version="1.0" encoding="utf-8"?>
<LinearLayout xmlns:android="http://schemas.android.com/apk/res/android"
  android:orientation="vertical" android:layout_width="fill_parent"
  android:layout_height="fill_parent">
  <TextView android:layout_width="fill_parent"
    android:layout_height="wrap_content" android:gravity="center"
    android:text="@string/titleMyTwitter" android:textSize="30sp"
    android:layout_margin="10dp" />
  <EditText android:layout_width="fill_parent"
    android:layout_height="fill_parent" android:layout_weight="1"
    android:hint="@string/hintText" android:id="@+id/editText"
    android:gravity="top|center_horizontal"></EditText>
  <Button android:layout_width="fill_parent"
    android:layout_height="wrap_content" android:text="@string/buttonUpdate"
    android:textSize="20sp" android:id="@+id/buttonUpdate"></Button>
</LinearLayout>
```

3.1.1 Important Widget Properties

While each widget can have many different properties, some are more important than others. Table 3.1 show what properties are more important than others.

Table 3.1: Important Widget Properties

Properties	**Description**
`layout_height` and `layout_width`	Defines how much space this widget is asking from its parent layout to display itself. While you could enter a value in pixels, inches, or similar, that is not a good practice. Since your application could run on many different devices with various screen sizes, you want to use relative size for your components, not absolute. So, best practice would be to use either `fill_parent` or `wrap_content` for the value. `fill_parent` means that your widget wants all the available space from its parent. `wrap_content` means that it only requires as much space as it needs to display its own content.
`layout_gravity`	Specifies how this particular widget is positioned within its layout, both horizontally and vertically. Notice the difference between this property and `gravity` below.
`gravity`	Specifies how the content of this widget is positioned within the widget itself. It is commonly confused with `layout_gravity`. Which one to use will depend on size of your widget and desired look.
`id`	`id` is simply the unique identifier for this particular widget in this particular layout resource file. Not every widget needs an id and I recommend removing id's if not needed to minimize clutter. But widgets that we'll later need to manipulate from Java do need id's. Id has the following format: `@+id/someName` where someName is whatever you want to call your widget. My naming convention is to use type followed by name, so `@+id/buttonUpdateStatus` for example.
`text`	Not all widgets have this property, but many do, such as `Button`, `EditText`, and `TextView`. It simply specifies the text for the widget. Now, it is not a good practice to just enter the text because than your layout will only work in one locale/language. Best practice is to define all text in `strings.xml` resource and refer to particular string via this notation: `@string/titleStatusUpdate`.

3.2 Strings Resource

Android tries hard to keep data in separate files. So, layouts are defined in their own resources, and all text values (such as button text, title text, etc.) should be defined in their own file called `strings.xml`. This later allows you to provide multiple versions of strings resources to be used for various languages, such as English, Japanese, or Russian.

Here's what our `strings.xml` file looks like at this point:

res/values/strings.xml

```xml
<?xml version="1.0" encoding="utf-8"?>
<resources>
  <string name="hello">Hello World, MyTwitter!</string>
  <string name="app_name">My Twitter</string>
  <string name="titleMyTwitter">My Twitter</string>
  <string name="hintText">Please enter your 140-characeter tweet</string>
  <string name="buttonUpdate">Update</string>
</resources>
```

That this is simply a set of name/value pairs.

Tip
I use a certain naming convention for my resource names. Let's look at `titleMyTwitter`, for example. First, I prefix the resource with the name of what it is, in this case a title of the activity. Secondly, I give it a name, MyTwitter. This naming convention helps later on keep many different resources sorted in a easy to find way. Finally, I use CamelCase for my names. Again, this is my convention that I see work well. There's no industry standard at this time.

3.3 StatusActivity

src/com/marakana/StatusActivity.java

```java
package com.marakana;

import winterwell.jtwitter.Twitter;
import android.app.Activity;
import android.os.Bundle;
import android.view.View;
import android.view.View.OnClickListener;
import android.widget.Button;
import android.widget.EditText;

public class StatusActivity extends Activity implements OnClickListener { // ❶
  EditText editText;
  Button updateButton;
  Twitter twitter;

  /** Called when the activity is first created. */
  @Override
  public void onCreate(Bundle savedInstanceState) {
    super.onCreate(savedInstanceState);
    setContentView(R.layout.main);

    // Find views
    editText = (EditText) findViewById(R.id.editText); // ❷
    updateButton = (Button) findViewById(R.id.buttonUpdate);

    updateButton.setOnClickListener(this); // ❸

    twitter = new Twitter("androidbootcamp", "pass2009"); // ❹
  }

  // Called when button is clicked // ❺
  public void onClick(View v) {
    twitter.setStatus(editText.getText().toString()); // ❻
  }
}
```

❷ Find views inflated from the XML layout and assign them to Java variables.

❸ Register the button to notify `this` i.e. `StatusActivity` when it gets clicked on.

❶ To make `StatusActivity` capable of being a button listener, it needs to implement `OnClickListener` interface.

❺ The method that is called when button is clicked, as part of `OnClickListener` interface.

❹ Connect to Twitter.com. At this point, we hard code the username and password.

❻ Make the web service API call to Twitter to update our status.

3.4 Update Manifest File for Internet Permission

Before this application can work, we must ask user to grant us right to use the internet. The way Android manages security is by specifying permissions needed for certain dangerous operations. User then must explicitly grant those permissions to each application when she installs the application first time around. User must grant all or no permissions - there's no middle ground. Also, user is not prompted about permissions upon upgrading an existing app.

Note
Since we are running this application in debug mode and installing it via USB cable, Android doesn't prompt us for permissions like it would the end user. However, we still must specify that the application requires certain permissions.

In this case, we want to ask user to grant this application the use of INTERNET permission. We need internet in order to connect to Twitter.com.

AndroidManifest.xml

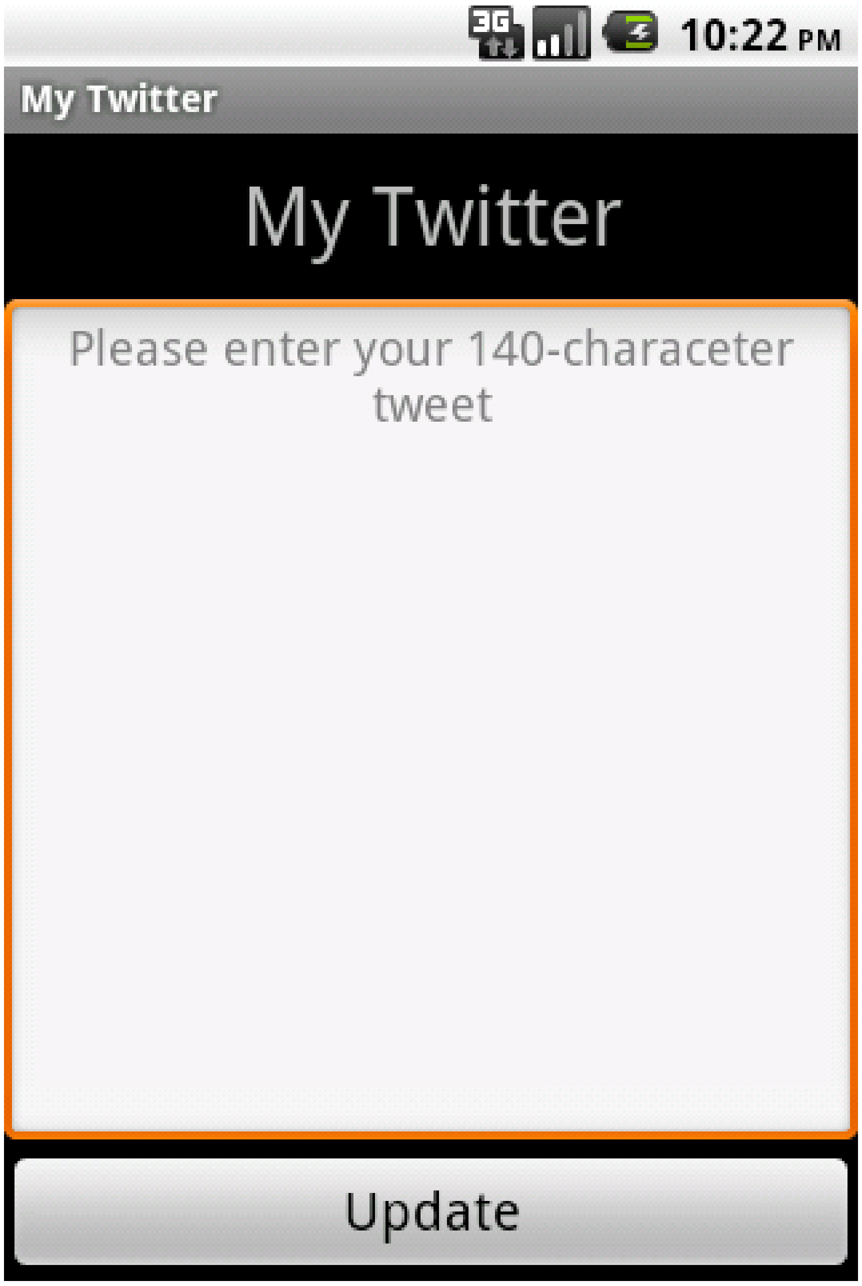
```
<?xml version="1.0" encoding="utf-8"?>
<manifest xmlns:android="http://schemas.android.com/apk/res/android"
  package="com.marakana" android:versionCode="1" android:versionName="1.0">
  <application android:icon="@drawable/icon" android:label="@string/app_name">
    <activity android:name=".StatusActivity" android:label="@string/app_name">
      <intent-filter>
        <action android:name="android.intent.action.MAIN" />
        <category android:name="android.intent.category.LAUNCHER" />
      </intent-filter>
    </activity>

  </application>
  <uses-sdk android:minSdkVersion="8" />

  <uses-permission android:name="android.permission.INTERNET" /> <!-- 1 -->
</manifest>
```

1 Defines the `<uses-permission>` element for `INTERNET` permission.

3.5 Summary

By the end of this section, you should have your application run and look like StatusActivity, Part 1. It should also successfully post your tweets to your twitter account. You can verify it is working by logging into twitter.com using the same username/password as hard coded in the application.

StatusActivity, Part 1



3.5.1 Source Code

Source code can be downloaded from http://marakana.com/static/courseware/android/MyTwitter-1.zip

Chapter 4

MyTwitter Part 2

First, we spice up the user interface a bit, add some graphics and images. Next, we add preferences activity to that user can change her twitter username and password. We also introduce the options menu in Android.

We also learn about Android services and implement UpdaterService.

4.1 Prefs Resource

We need a way for users to specify their individual twitter username and password information. This requires a screen to enter the information, a Java code to validate and process that information, and some kind of storage mechanism to store this information.

While all this sounds like a lot of work, Android provides a framework to help streamline working with user preferences. First, we'll define what our preference data looks like in a `prefs.xml` resource.

res/xml/prefs.xml

```
<?xml version="1.0" encoding="utf-8"?>
<PreferenceScreen xmlns:android="http://schemas.android.com/apk/res/android">
  <EditTextPreference android:title="@string/titleUsername"
    android:summary="@string/summaryUsername" android:key="username"></ ←
        EditTextPreference>
  <EditTextPreference android:title="@string/titlePassword"
    android:summary="@string/summaryPassword" android:key="password"></ ←
        EditTextPreference>
</PreferenceScreen>
```

`<PreferenceScreen>` is the root element that defines our main preference screen. It has two children, both `<EditTextPreference`
This is simply a piece of editable text. Other common elements here could be `<CheckBoxPreference>`, `<ListPreference>`, and so on.

The main property of any of these elements is the `key`. The key is how we'll look up these values later on. Remember, preferences is just a set of name-value pairs at the end of the day.

4.2 PrefsActivity

Now that we have the preferences defined in their own XML resource file, we can create the activity to display these preferences. `PrefsActivity` is a very simple Java file.

src/com/marakana/PrefsActivity.java

```java
package com.marakana;

import android.os.Bundle;
import android.preference.PreferenceActivity;

public class PrefsActivity extends PreferenceActivity { // ❶

  @Override
  protected void onCreate(Bundle savedInstanceState) { // ❷
    super.onCreate(savedInstanceState);
    addPreferencesFromResource(R.xml.prefs); // ❸
  }

}
```

❶ Unlike regular activities, PrefsActivity will subclass (i.e. extend) `PreferenceActivity` class.

❷ Just like any other activity, we override `onCreate()` method to initialize the activity.

❸ Unlike regular activities that usually call `setContentView()`, preference activity will set its content from the `prefs.xml` file via a call to `addPreferencesFromResource()`.

4.3 Update Manifest File

Whenever you create one of these main building blocks (Activities, Services, etc.) you need to define them in `AndroidManifest.xml` file.

AndroidManifest.xml

```xml
<?xml version="1.0" encoding="utf-8"?>
<manifest xmlns:android="http://schemas.android.com/apk/res/android"
  package="com.marakana" android:versionCode="1" android:versionName="1.0">
  <application android:icon="@drawable/icon" android:label="@string/app_name">

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

    <activity android:name=".PrefsActivity" /> <!-- ❶ -->

    <service android:name=".UpdaterService" />  <!-- ❷ -->

  </application>
  <uses-sdk android:minSdkVersion="8" />

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

❶ Defines the new PrefsActivity.

(2) Defines the new UpdaterService which we'll create later in this chapter.

You should now have a preference screen, but no good way of viewing it yet. We need a way to launch this new activity. For that, we use options menu.

4.4 Menu Resource

Our new PrefsActivity is good to go, except we don't have a way of displaying it. We need to somehow tell the app to launch this new activity.

A very Android-like user experience is to setup *options menu*. Options menu is the menu that pops up at the bottom of the application screen when user clicks on *Menu* button on the device itself.m

We start by defining the menus in an XML resource.

res/menu/menu.xml

```
<?xml version="1.0" encoding="utf-8"?>
<menu xmlns:android="http://schemas.android.com/apk/res/android">
  <item android:id="@+id/itemPrefs" android:title="@string/titlePrefs"
    android:icon="@android:drawable/ic_menu_preferences"></item>
  <item android:title="@string/titleServiceStart" android:id="@+id/itemServiceStart"
    android:icon="@android:drawable/ic_media_play"></item>
  <item android:title="@string/titleServiceStop" android:id="@+id/itemServiceStop"
    android:icon="@android:drawable/ic_media_pause"></item>
</menu>
```

An options menu consists of `<menu>` root element. That element can contain multiple `<item>` elements, each specifying a specific menu item.

While menu items can have many different properties, some are more important than others. The `id` property is necessary so later on when can identify in Java which item user clicked on. `title` specifies what text to put on the actual menu button on the screen. As before, you could hard code the text value here, but better practice is to define all your text in `strings.xml` file and reference them here.

Finally, we chose to specify the optional `icon` property as well. Icon specifies the image that the user will see along with the text on the menu item button.

Note
Notice that we refer to actual images as drawables. Also, notice that in this case, the reference looks like this: `@android:drawable/ic_menu_preferences`. The use of *`android:`* is new and it specifies that his resource is not part of our our application but rather part of the Android system-wide set of resources.

4.5 Update StatusActivity to Load Menu

We need to update the StatusActivity to load up the options menu. To do that, add `onCreateOptionsMenu()` method to StatusActivity. This method gets called only first time when user clicks on the *Menu* button.

```
// Called first time user clicks on the menu button
@Override
public boolean onCreateOptionsMenu(Menu menu) {
  MenuInflater inflater = getMenuInflater();   // (1)
  inflater.inflate(R.menu.menu, menu);          // ❷
  return true;
}
```

❶ We get the `MenuInflater` object from the context.

❷ Use the inflater to inflate the menu from the XML resource.

4.6 Update StatusActivity to Handle Menu Events

We also need a way to handle various clicks on the menu items. To do that, we add another method, `onOptionsItemSelected()`. This method is called every time user clicks on a menu item.

```
// Called when an options item is clicked
@Override
public boolean onOptionsItemSelected(MenuItem item) {
  switch (item.getItemId()) {                            // ❶
  case R.id.itemServiceStart:
    startService(new Intent(this, UpdaterService.class));  // ❷
    break;
  case R.id.itemServiceStop:
    stopService(new Intent(this, UpdaterService.class));   // ❸
    break;
  case R.id.itemPrefs:
    startActivity(new Intent(this, PrefsActivity.class));  // ❹
  break;
  }

  return true;
}
```

❶ Since the same method is called regardless which item user clicked on, we need to figure out the id of that item and based on that switch to a specific case to handle each item.

❷ `startService()` method is part of the context and it allows us to send an intent to the system that we'd like a specific service started.

❸ `stopService()` sends an intent to the system to stop a specific service. If service is not started, nothing happens.

❹ `startActivity()` method in context allows us to launch a new activity. In this case, we are creating a new intent specifying to start PrefsActivity class.

4.7 Strings Resource

Our updated `strings.xml` now looks like this:

res/values/strings.xml

```
<?xml version="1.0" encoding="utf-8"?>
<resources>
  <string name="titleServiceStop">Stop Service</string>
  <string name="titleServiceStart">Start Service</string>
  <string name="titlePrefs">Prefs</string>
  <string name="hello">Hello World, MyTwitter!</string>
  <string name="app_name">My Twitter</string>
  <string name="titleMyTwitter">My Twitter</string>
  <string name="hintText">Please enter your 140-characeter tweet</string>
  <string name="buttonUpdate">Update</string>
  <string name="titleUsername">Username</string>
  <string name="titlePassword">Password</string>
  <string name="summaryUsername">Please enter your Twitter username</string>
  <string name="summaryPassword">Please enter your Twitter password</string>
</resources>
```

You should be able to run your application at this point an and see the new PreferenceActivity.

PreferenceActivity images/PreferenceActivity-2.png

4.8 Colors in Android

Android supports standard RGB colors (Red, Green, Blue). The value of each color is between 0 and 255 in decimal system, or 00 to FF in hexadecimal. So, yellow color would be 255,255,0 or #ffff00 in hex. In Android you can also shorten this to #ff0.

Additionally, Android also supports the Alpha channel to specify the transparency of the color. Alpha value is also between 0 and 255, or 00 and FF, where higher number represents more solid color and 0 value would be fully transparent. You can optionally specify Alpha channel either as AARRGGBB, or in short form as ARGB. So, semi translucent yellow would be #7fffff00.

4.9 Units in Android User Interface

Just like in other systems, you can use pixels, points, inches, millimeters, and other typical units to specify sizes of widgets and such. But Android also introduces two new units:

- Dp (synonym for Dip) is a density-independent pixel where at 160dpi screen, 1dp is the same as 1px.
- Sp is used for fonts to denote fonts that should be scalable based on users preference. So a 12sp font may be 18px if user chooses to make all text extra large.

4.10 Spicing Up StatusActivity Layout

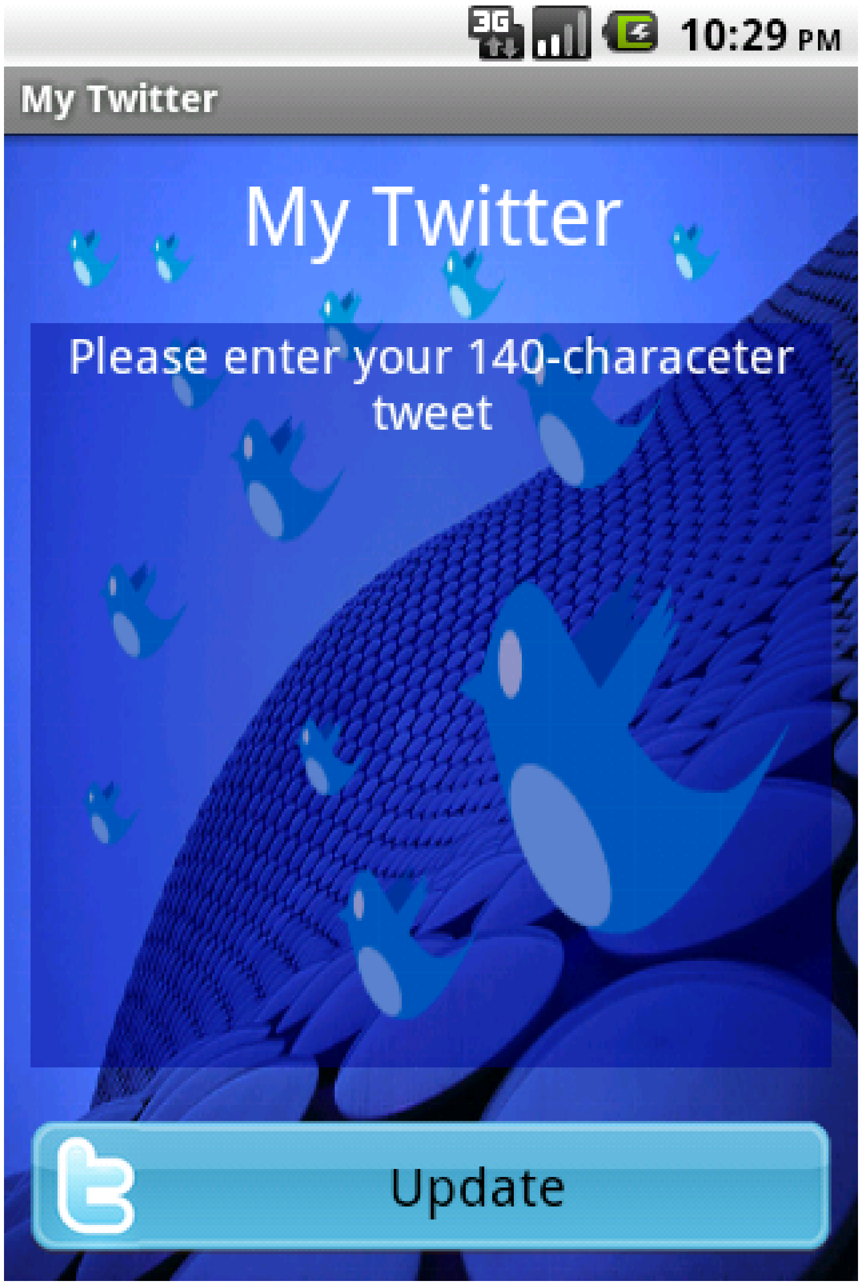
Now that we have the basic screen to post a Twitter update, we can make it look better. Android comes with a very rich support for user interface.

We update our main.xml to add some background images, colors, text sizes, and such.

res/layout/main.xml

```
<?xml version="1.0" encoding="utf-8"?>
<LinearLayout xmlns:android="http://schemas.android.com/apk/res/android"
  android:orientation="vertical" android:layout_width="fill_parent"
  android:layout_height="fill_parent" android:background="@drawable/background">
  <TextView android:layout_width="fill_parent"
    android:layout_height="wrap_content" android:gravity="center"
    android:text="@string/titleMyTwitter" android:textSize="30sp"
    android:layout_margin="10dp" android:textColor="#fff" />
  <EditText android:layout_width="fill_parent"
    android:layout_height="fill_parent" android:layout_weight="1"
    android:hint="@string/hintText" android:id="@+id/editText"
    android:gravity="top|center_horizontal" android:background="#6009"
    android:layout_margin="10dp" android:textColorHint="#fff"
    android:textColor="#fff"></EditText>
  <Button android:layout_width="fill_parent"
    android:layout_height="wrap_content" android:text="@string/buttonUpdate"
    android:textSize="20sp" android:id="@+id/buttonUpdate"
    android:layout_margin="10dp" android:background="@drawable/buttonbg"></Button>
</LinearLayout>
```

You should now have a much better looking StatusActivity.



StatusActivity, Part 2

4.11 UpdaterService

We need to define a service so there's an always-on background process running and pulling latest twitter updates into a local database. Remember, the purpose of this pull mechanism is so that we cache updates locally in order for our app to have data even when its off line.

Just like an activity, a service is implemented by subclassing `Service` class and overriding certain methods. The ones of interest to us are:

- `onCreate()`: Called when the service is created first time.

- `onStart()`: Called when the service is started.
- `onDestroy()`: Called when the service is terminated.

In UpdaterService, we also make a use of a `Handler`. Handler is a responsible for handling jobs posted on a message queue. That's the mechanism for managing various jobs on a single thread.

In our case, we'll define a job as pulling data from Twitter. As this is something that could take a long time due to network latency, we don't want this job to be blocking other processing. This is a good use of handler to post this job for some future execution. Also, once the job is done, it reposts itself back on the same queue to execute again. This is how we accomplish the pull from Twitter to happen over and over again.

For now, we're simply going to print out to the log the fact that the job ran. We'll implement the actual pulling data from Twitter later.

src/com/marakana/UpdaterService.java

```
package com.marakana;

import android.app.Service;
import android.content.Intent;
import android.os.Handler;
import android.os.IBinder;
import android.util.Log;

public class UpdaterService extends Service {   //❶
  static final String TAG = "UpdaterService";
  static final int DELAY = 30000; // 1/2 a min
  UpdaterRunnable updaterRunnable;
  Handler handler;

  @Override
  public IBinder onBind(Intent intent) {  // ❷
    return null;
  }

  @Override
  public void onCreate() {
    super.onCreate();

    // Initialize handler & runnable
    handler = new Handler();    // ❸
    updaterRunnable = new UpdaterRunnable();    // ❹
    handler.post(updaterRunnable);    // ❺

    Log.d(TAG, "onCreate'd");
  }

  @Override
  public void onStart(Intent intent, int startId) {   // ❻
    super.onStart(intent, startId);
    Log.d(TAG, "onStart'd");
  }

  @Override
  public void onDestroy() {   // ❼
    super.onDestroy();
    // Cleanup handler & runnable
    handler.removeCallbacks(updaterRunnable);
    updaterRunnable = null;
    handler = null;

    Log.d(TAG, "onDestroy'd");
  }

  class UpdaterRunnable implements Runnable {   // 8

    public void run() {
      Log.d(TAG, "UpdaterRunnable run'd");

      handler.postDelayed(updaterRunnable, DELAY);    // 9
    }

  }
}
```

1 To create a service, subclass `android.app.Service` class.

2 `onBind()` method is not used for non-bound services, which this one is. For now ignore this method (i.e. implement it by returning `null`).

3 In `onCreate()` we create a new `handler` object.

4 Instantiate a new instance of our "job" represented by the inner class `UpdaterRunnable`.

5 Post the job on the message queue via the handler. This is not a blocking call.

6 `onStart()` doesn't do much in this case.

7 In `onDestroy()` we unwind everything we did in `onCreate()`.

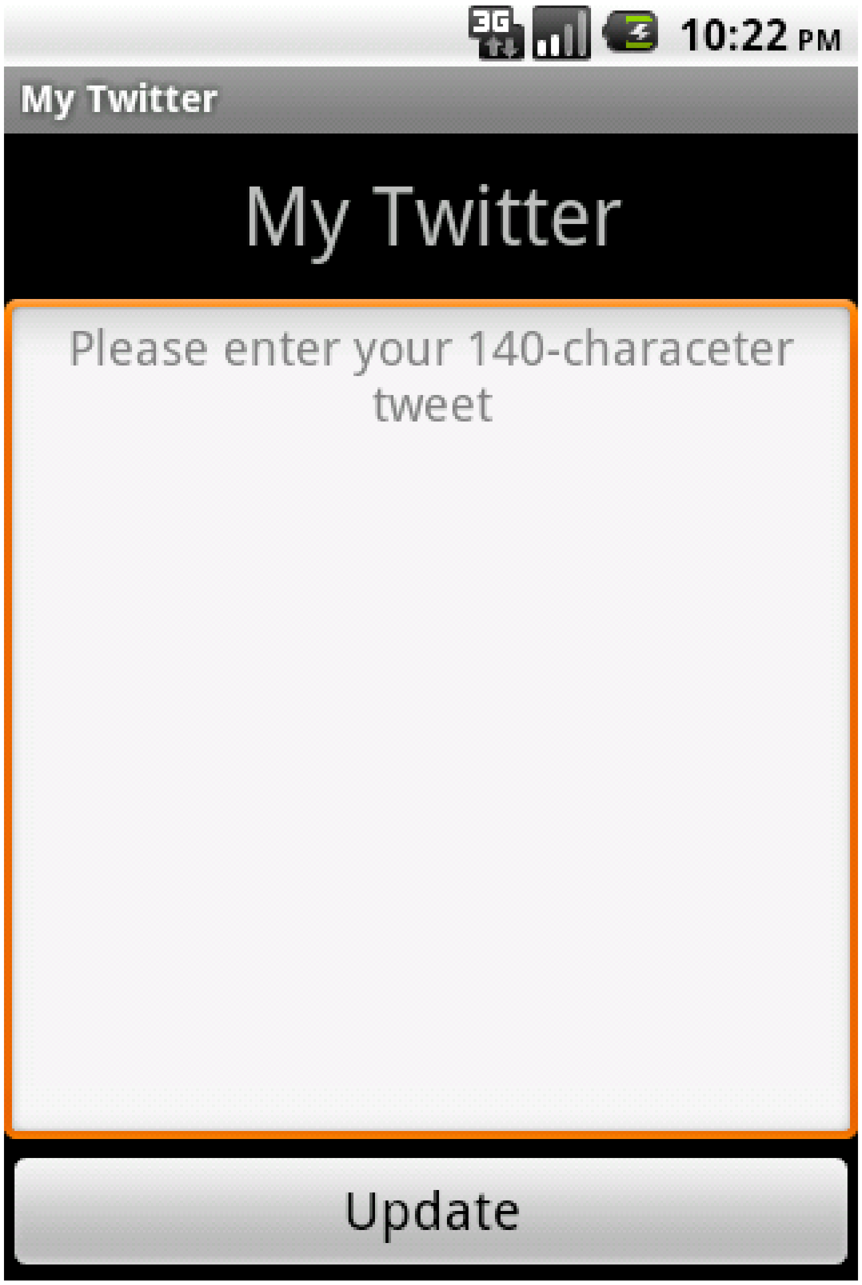
8 `UpdaterRunnable` class represents our job to be done. It needs to implement `Runnable` interface in order for us to post it on the message queue.

9 Once we are done with our work, we re-post it to run again at some future point in time.

4.12 Summary

At this point, our StatusActivity looks like StatusActivity, Part 1. We have also added PrefsActivity which looks like PrefsActivity. You should also have a way to update your Twitter preferences. You can also start and stop the service at this point. The service will run although it currently still doesn't connect to Twitter.

StatusActivity, Part 1



PrefsActivity

images/PrefsActivity-1.png

4.12.1 Source Code

Source code can be downloaded from http://marakana.com/static/courseware/android/MyTwitter-2.zip

Chapter 5

MyTwitter Part 3

We introduce SQLite database and the DbHelper. We add TimelineActivity to display the list of friends' statuses. We also create a custom TimelineAdapter to make it all look good.

5.1 DbHelper

Android provides an elegant interface for your app to store its data in a SQLite database. To get access to the database, you need a helper class that will create your database if it doesn't already exist. It will also provide you with the access to the database via a Java class `SQLiteDatabase`.

src/com/marakana/DbHelper.java

```java
package com.marakana;

import android.content.Context;
import android.database.sqlite.SQLiteDatabase;
import android.database.sqlite.SQLiteOpenHelper;
import android.util.Log;

public class DbHelper extends SQLiteOpenHelper { // ❶
  static final String TAG = "DbHelper";
  static final String DB_NAME = "timeline.db";  // ❷
  static final int DB_VERSION = 3;  // ❸
  static final String TABLE = "timeline"; // ❹
  static final String C_ID = "_id";
  static final String C_CREATED_AT = "created_at";
  static final String C_SOURCE = "source";
  static final String C_TEXT = "txt";
  static final String C_USER = "user";
  Context context;

  // Constructor
  public DbHelper(Context context) {  // ❺
    super(context, DB_NAME, null, DB_VERSION);
    this.context = context;
  }

  // Called only once, first time the DB is created
  @Override
  public void onCreate(SQLiteDatabase db) {
    String sql = context.getString(R.string.sql1);  // ❻

    Log.d(TAG, "onCreat'd sql: " + sql);

    db.execSQL(sql);  // ❼
  }

  // Called whenever newVersion != oldVersion
  @Override
  public void onUpgrade(SQLiteDatabase db, int oldVersion, int newVersion) {  // ❽
    // Typically do ALTER TABLE statements, but...we're just in development,
    // so:

    db.execSQL("drop table if exists " + TABLE); // blow the old database
    // away
    Log.d(TAG, "onUpdate'd");
    onCreate(db); // run onCreate to get new database
  }

}
```

❶ Start by subclassing `SQLiteOpenHelper`.

❷ This is the database file name.

❸ This is the version of our database. Version is important so that later when you change the schema, you can provide existing users with a way to upgrade their database to the latest schema.

❹ The following are some database constants specific to our application. It is handy to define these as constants to that we can refer to them from other classes.

❺ We override the `SQLiteOpenHelper` by passing the constants to the `super` and retaining the local reference to the context.

❻ In this case, I defined the actual SQL to create the database schema in our `strings.xml` resource file. We use `context.getString()` to get the actual string value of this XML resource.

❼ Once we have our SQL to create the database by running `execSQL()` on the database object that was passed into `onCreate()`.

❽ `onUpgrade()` is called whenever user's database version is different than the application version. This will typically happen when you change the schema and release the application update to users who already have older version of your app.

Note

`onCreate()` is called only once, when the database file doesn't exist. You typically run `CREATE TABLE ...` SQL statement in `onCreate()`. `The result of `onCreate()` should be that the actual database is created for the user. `onUpgrade()` will get called when the app is upgraded with a newer database schema. You typically execute `ALTER TABLE ...` SQL statements in `onUpgrade()`.

5.2 Verify Database Got Created

If the database file got created successfully, it will be located in `/data/data/com.marakana/databases/timeline.db` file. You can use Eclipse DDMS and File Explorer to look at the file system of the device, or you can use `adb shell` on your command line, and then `ls /data/data/com.marakana/databases/timeline.db` to make sure the file is there.

5.3 Update UpdaterService

We can now update the UpdaterService to pull the data from Twitter and store it in the database.

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

  // Open the database
  dbHelper = new DbHelper(this);          // ❶
  db = dbHelper.getWritableDatabase();    // ❷

  // Setup preferences
  prefs = PreferenceManager.getDefaultSharedPreferences(this);   // ❸
  prefs.registerOnSharedPreferenceChangeListener(this);          // ❹

  // Initialize handler & runnable
  handler = new Handler();
  updaterRunnable = new UpdaterRunnable();
  handler.post(updaterRunnable);

  Log.d(TAG, "onCreate'd");
}
```

❶ Create the instance of DbHelper and pass `this` as the context for it. DbHelper will figure out if the database needs to be created or upgraded.

❷ Get the writable database so we can insert new statuses into it.

❸ Using `SharedPreferences`, find out the Twitter username and password.

❹ Also register for changes to preferences, since it's likely they may change while the service is running.

We also need to update our `UpdaterRunnable` inner class to call a method that does the actual work of pulling data from Twitter.

```
class UpdaterRunnable implements Runnable {
  public void run() {
    pullFromTwitter();  // ❶

    Log.d(TAG, "UpdaterRunnable run'd");

    // Do this again
    handler.postDelayed(updaterRunnable, DELAY);
  }
}
```

1. Call to method to do the actual pull of data from Twitter.

And finally, we need to connect to Twitter, get latest updates, and insert them into the database. That is done in `pullFromTwitter()` method.

```
private void pullFromTwitter() {
  List<Status> timeline = null;
  try {
    timeline = getTwitter().getFriendsTimeline();   // ❶
  } catch (TwitterException e) {
    e.printStackTrace();
  }

  ContentValues values = new ContentValues();       // ❷
  // Loop over the timeline and print it out
  for (Status status : timeline) {                  // ❸
    // Insert into database
    values.put(DbHelper.C_ID, status.id);           // ❹
    values.put(DbHelper.C_CREATED_AT, status.createdAt.getTime());
    values.put(DbHelper.C_SOURCE, status.source);
    values.put(DbHelper.C_TEXT, status.text);
    values.put(DbHelper.C_USER, status.user.name);
    try {
      db.insertOrThrow(DbHelper.TABLE, null, values);  // ❺
      Log.d(TAG, String.format("%s: %s", status.user.name, status.text));
    } catch (SQLException e) {
    }

  }
}
```

1. `getTwitter()` is our lazy initialization of Twitter object. Then we call the actual Twitter API call `getFriendsTimeline()` to get last 20 statuses from friends in last 24 hours.
2. `ContentValues` is a simple name-value pairs data structure that maps database table names to their respective values.
3. We loop over all the records we got via the Twitter web service API call.
4. For each record, we create a content value.
5. We insert the content value into the database via `insert()` call to `SQLiteDatabase` object. Notice that we are not piecing together a SQL statement here, but rather using a *prepared statement* approach to inserting into the database.

5.4 TimelineActivity Layout

Timeline activity is responsible for displaying all the statuses from our friends. The first iteration of TimelineActivity could be one where we simply out a `TextView` to display all the output from the database. Since there may be quite a bit of data, we may want to wrap `TextView` into `ScollView`.

While this approach would work for smaller data sets, it is not optimal nor recommended. The better approach would be to use `ListView` to represent the list of statuses that we have in the database. `ListView` is much more scalable and efficient.

res/layout/timeline.xml

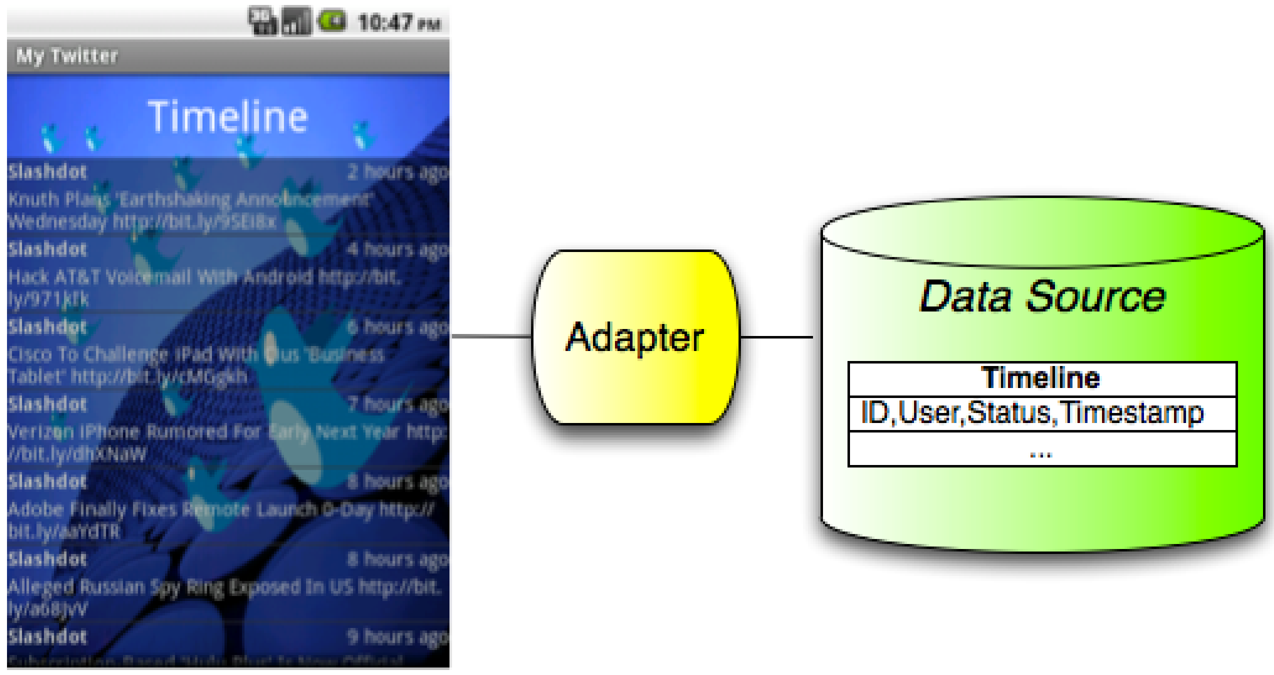
```
<?xml version="1.0" encoding="utf-8"?>
<LinearLayout xmlns:android="http://schemas.android.com/apk/res/android"
  android:orientation="vertical" android:layout_height="fill_parent"
  android:layout_width="fill_parent" android:background="@drawable/background">
  <TextView android:layout_width="wrap_content"
    android:layout_height="wrap_content" android:layout_gravity="center"
    android:layout_margin="10dp" android:text="@string/titleTimeline"
    android:textColor="#fff" android:textSize="30sp" />


  <!-- ❶ -->
  <ListView android:layout_height="fill_parent"
    android:layout_width="fill_parent" android:id="@+id/listTimeline"
    android:background="#6000" />

</LinearLayout>
```

❶ Adding `ListView` to your layout is like adding any other widget. The main attributes are `id`, and `layout_height` and `layout_width`.

5.5 About Adapters

In the first iteration of TimelineActivity, you may just use a `TextView` to print out the data you got from the database. Soon, you'd realize that there's too much data for the size of the screen. You may then wrap your `TextView` into a `ScrollView` so that you can put more text and scroll around the small screen. This will work for a few dozen records. But what if you have 100s or 1,000s of records in the database? Waiting to get them all and print them all would be highly inefficient. User probably doesn't even care about all of the data anyhow.

To address this issue, Android provides adapters. Adapters are a smart way to connect a `View` with some kind of data source. Typically, your view would be `ListView` and the data would come in form of a `Cursor` or `Array`. So, adapters come as subclasses of `CursorAdapter` or `ArrayAdapter`.



Slashdot 2 hours ago
Knuth Plans 'Earthshaking Announcement' Wednesday http://bit.ly/9SEi8x
Slashdot 4 hours ago
Hack AT&T Voicemail With Android http://bit.ly/971kfk
Slashdot 6 hours ago
Cisco To Challenge iPad With Plus 'Business Tablet' http://bit.ly/cMGgkh
Slashdot 7 hours ago
Verizon iPhone Rumored For Early Next Year http://bit.ly/dhXNaW
Slashdot 8 hours ago
Adobe Finally Fixes Remote Launch 0-Day http://bit.ly/aaYdTR
Slashdot 8 hours ago
Alleged Russian Spy Ring Exposed In US http://bit.ly/a68jVV
Slashdot 9 hours ago

Figure 5.1: Adapter

For MyTwitter project, we may want to use `SimpleCursorAdapter` initially. It works well for mapping a piece of data in the database to a single view on the screen.

The row of data in the `ListView` is represented by a custom layout defined in `row.xml` file.

res/layout/row.xml

```
<?xml version="1.0" encoding="utf-8"?>
<!-- Vertical linear layout ❶ -->
<LinearLayout xmlns:android="http://schemas.android.com/apk/res/android"
  android:layout_height="wrap_content" android:orientation="vertical"
  android:layout_width="fill_parent">
  <!-- Horizontal linear layout ❷ -->
  <LinearLayout android:layout_height="wrap_content"
    android:layout_width="fill_parent">
    <!-- User ❸ -->
    <TextView android:layout_height="wrap_content"
      android:layout_width="fill_parent" android:layout_weight="1"
      android:id="@+id/textUser" android:text="Slashdot"
      android:textStyle="bold"></TextView>
    <!-- Timestamp ❹ -->
    <TextView android:layout_height="wrap_content"
      android:layout_width="fill_parent" android:layout_weight="1"
      android:gravity="right" android:id="@+id/textCreatedAt"
      android:text="10 minutes ago"></TextView>
  </LinearLayout>
  <!-- Status ❺ -->
  <TextView android:layout_height="wrap_content"
    android:layout_width="fill_parent" android:id="@+id/textText"
    android:text="Firefox comes to Android"></TextView>
```

```
</LinearLayout>
```

1. The main layout for the entire row. It is vertical because our row consists of two lines.
2. Layout that runs horizontally and represents first line of data, namely user and timestamp.
3. Twitter user that posted this update.
4. Timestamp when it was posted. It should be relative time (e.g. 10 minutes ago).
5. The actual Twitter status, no longer than 140 characters.

5.6 TimelineActivity

Timeline activity will load its layout described above, but it also needs to connect to the database and read all the data. Then it needs to create an adapter to connect the data source to the list view.

src/com/marakana/TimelineActivity.java

```
package com.marakana;

import android.app.Activity;
import android.database.Cursor;
import android.database.sqlite.SQLiteDatabase;
import android.os.Bundle;
import android.widget.ListView;

public class TimelineActivity extends Activity {
  DbHelper dbHelper;
  SQLiteDatabase db;
  Cursor cursor;             // ❶
  ListView listTimeline;     // ❷
  TimelineAdapter adapter;   // ❸

  @Override
  protected void onCreate(Bundle savedInstanceState) {
    super.onCreate(savedInstanceState);
    setContentView(R.layout.timeline);

    // Find your views
    listTimeline = (ListView) findViewById(R.id.listTimeline); // ❹

    // Connect to database
    dbHelper = new DbHelper(this);
    db = dbHelper.getReadableDatabase();
  }

  @Override
  public void onDestroy() {
    super.onDestroy();

    // Close the database
    db.close();
  }
  @Override
  protected void onResume() {                           // ❺
    super.onResume();

    // Get the data from the database
    cursor = db.query(DbHelper.TABLE, null, null, null, null, null,
        DbHelper.C_CREATED_AT + " DESC");               // ❻

    // Create the adapter
    adapter = new TimelineAdapter(this, cursor);        // ❼
    listTimeline.setAdapter(adapter);                   // ❽

  }

}
```

❶ `cursor` represents the data. It could be one row of the data at the time, or pointer to a beginning of a dataset. In this case, `cursor` points to all the status updates that we have in the database.

❷ `listTimeline` is our `ListView` that displays the data.

❸ `adapter` is our custom adapter explained below.

❹ We get the view from the XML layout.

❺ In this case, we read the data from the database in `onResume()` instead of `onCreate()`. This is because we'd like to have the data be refreshed every time we come back to this activity, and not just first time the activity is created.

❻ The actual query to get all the statuses from the timeline database. Notice that we sort them in descending order based on timestamp.

❼ Create a new instance of the `TimelineAdapter` and pass it the context and the data.

❽ Set our `ListView` to connect to the data via the adapter.

At this point, `TimelineActivity` is complete.

5.7 TimelineAdapter

`TimelineAdapter` is our custom adapter. While `SimpleCursorAdapter` did the job of straight forward mapping of data in the database to views on the screen, we had an issue with the timestamp. The timestamp in the database is represented as a number of seconds since January 1, 1070. This is the standard Unix Time and as such is very useful for representing actual points in time. However, it is not very user friendly. For example, "July 5 at 1:37" is stored as 1278293869. So, instead of displaying the actual timestamp, we want to show user the relative time since that event, for example "10 minutes ago". In other words, our mapping from data to screen is not straight forward in this case.

The job of `TimelineAdapter` is to inject some business logic to convert Unix timestamp to relative time.

src/com/marakana/TimelineAdapter.java

```
package com.marakana;

import android.content.Context;
import android.database.Cursor;
import android.text.format.DateUtils;
import android.view.View;
import android.widget.SimpleCursorAdapter;
import android.widget.TextView;

public class TimelineAdapter extends SimpleCursorAdapter { // ❶
  static final String[] from = { DbHelper.C_CREATED_AT, DbHelper.C_USER,
      DbHelper.C_TEXT }; // ❷
  static final int[] to = { R.id.textCreatedAt, R.id.textUser, R.id.textText }; // ❸

  // Constructor
  public TimelineAdapter(Context context, Cursor c) { // ❹
    super(context, R.layout.row, c, from, to);
  }

  // This is where the actual binding of a cursor to view happens
  @Override
  public void bindView(View row, Context context, Cursor cursor) { // ❺
    super.bindView(row, context, cursor);

    // Manually bind created at timestamp to its view
    long timestamp = cursor.getLong(cursor
        .getColumnIndex(DbHelper.C_CREATED_AT)); // ❻
    TextView textCreatedAt = (TextView) row.findViewById(R.id.textCreatedAt); // ❼
    textCreatedAt.setText(DateUtils.getRelativeTimeSpanString(timestamp)); // ❽
  }

}
```

❶ To create our own custom adapter, we subclass one of the Android standard adapters, in this case `SimpleCursorAdapter`.

❷ This constant defines the columns of interest to us in the database.

❸ This constant specifies the id's of views that we'll map those columns to.

❹ Our constructor simply calls the constructor in `super`.

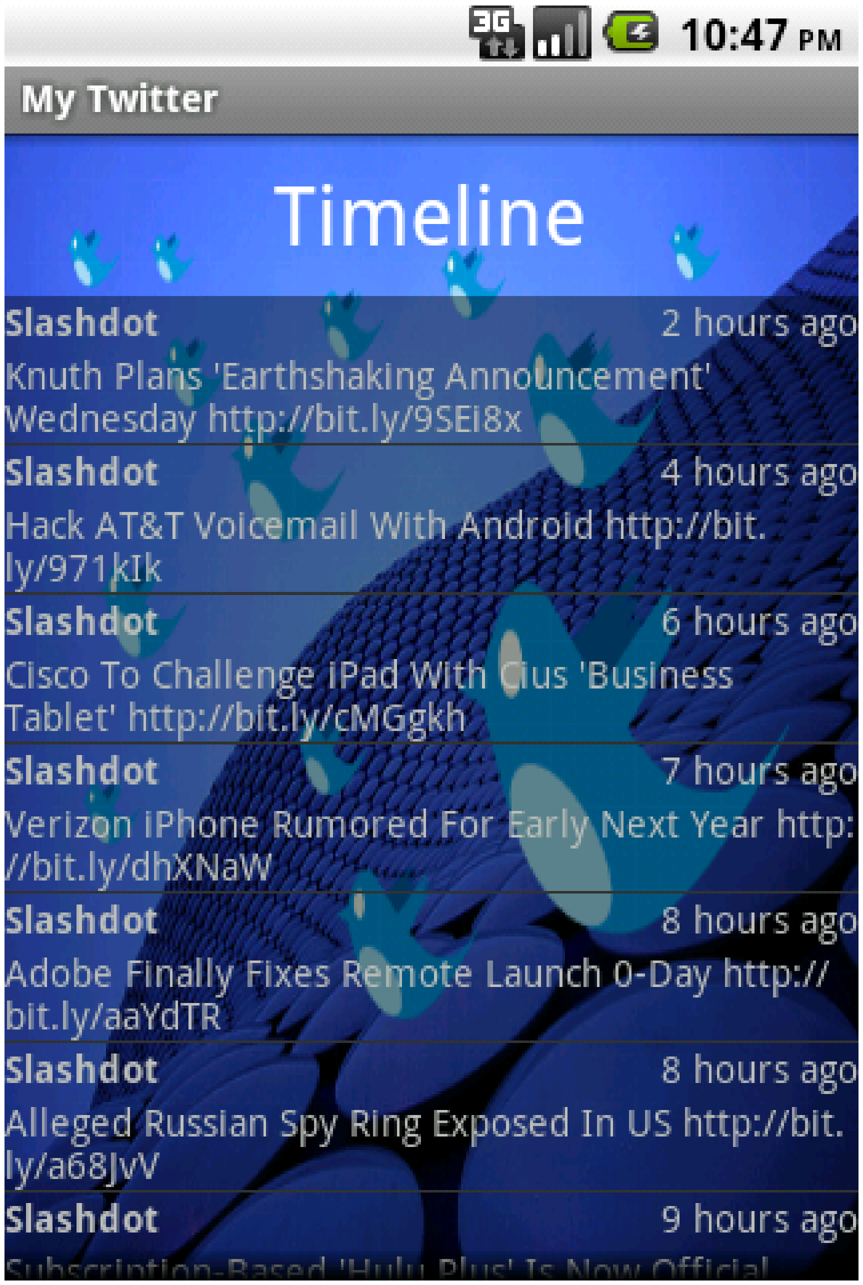
❺ The only method we override is `bindView()` method. This method is called for each row to map its data to its views. This is where the jest of adapter work happens. In order to reuse most of data-to-views mapping that `SimpleCursorAdapter` does, we call `super.bindView()` first.

❻ To override default mapping for timestamp, first we get the actual timestamp value from the database.

❼ Next, we find the specific `TextView` in the `row.xml` file.

❽ Finally, we set the value of `textCreatedAt` to relative time since the timestamp. We use Android SDK method `DateUtils.getRelativeTimeSpanString()` to help us with that.

You should now have a new TimelineActivity with all your twitter data.



Slashdot 2 hours ago
Knuth Plans 'Earthshaking Announcement' Wednesday http://bit.ly/9SEi8x

Slashdot 4 hours ago
Hack AT&T Voicemail With Android http://bit.ly/971kIk

Slashdot 6 hours ago
Cisco To Challenge iPad With Cius 'Business Tablet' http://bit.ly/cMGgkh

Slashdot 7 hours ago
Verizon iPhone Rumored For Early Next Year http://bit.ly/dhXNaW

Slashdot 8 hours ago
Adobe Finally Fixes Remote Launch 0-Day http://bit.ly/aaYdTR

Slashdot 8 hours ago
Alleged Russian Spy Ring Exposed In US http://bit.ly/a68JvV

Slashdot 9 hours ago
Subscription-Based 'Hulu Plus' Is Now Official

TimelineActivity

5.8 Summary

At this point, MyTwitter can post new status as well as list statuses of our friends. Our application is complete and usable.

5.8.1 Source Code

Source code can be downloaded from http://marakana.com/static/courseware/android/MyTwitter-3.zip

Chapter 6

MyTwitter Part 4

At this point MyTwitter is a complete app that does what user would expect it to do. However, there are couple of minor improvements that can help make it even better while also introducing Broadcast Receivers.

6.1 BootReceiver

We have `UpdaterService` responsible for periodically pulling data from Twitter and storing it locally. However, currently user needs to manually start the service. User does this by first starting the application, then clicking on Start Service menu option.

Better approach would be if somehow `UpdaterService` was started automatically by the system, when the device is powered up. To do this, we create `BootReceiver`, a broadcast receiver that will launch our activity and it itself will get launched by the system when the boot is complete.

src/com/marakana/BootReceiver.java

```java
package com.marakana;

import android.content.BroadcastReceiver;
import android.content.Context;
import android.content.Intent;
import android.util.Log;

public class BootReceiver extends BroadcastReceiver { // ❶

  @Override
  public void onReceive(Context context, Intent intent) { // ❷
    context.startService(new Intent(context, UpdaterService.class)); // ❸
    Log.d("BootReceiver", "onReceive'd");
  }

}
```

❶ We create `BootReceiver` by subclassing `BroadcastReceiver` base class for all receivers.

❷ The only method that we need to implement is `onReceive()`. It is this method that gets called when an intent matches this receiver.

3 We launch an intent to start our updater service.

At this point, we have our boot receiver. But, in order for it to get called, we must register it with the system.

6.2 Register BootReceiver with AndroidManifest file

To register `BootReceiver`, we add it to the manifest file. We also add an *intent filter* to it. It is the intent filter that specifies what triggers the receiver to get activated.

AndroidManifest.xml: <application> section

```
...
<receiver android:name=".BootReceiver">
  <intent-filter>
    <action android:name="android.intent.action.BOOT_COMPLETED" />
  </intent-filter>
</receiver>
...
```

In order to get notifications for this particular intent filter, we must specify that we're using a specific permission required by it.

AndroidManifest.xml: <manifest> section

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

Note
If we don't specify the permission we require, we simply won't get notified when this event occurs. We won't even know we won't get notified, so this is potentially a hard bug to find.

6.3 Testing Boot Receiver

At this point, you can reboot your device. Once it comes back up, our UpdaterService should be up and running. You can verify that by either looking at the LogCat for our output, or by using System Settings.

At Home screen, click on Menu button, choose Settings→Applications→Running Services. You should see UpdaterService listed there.

6.4 TimelineReceiver

Currently, if you are viewing TimelineActivity while a new status update comes in, you wouldn't now about it. That's because the UpdaterService doesn't have a way to notify TimelineActivity to refresh itself.

To address this, we create another broadcast receiver, this time as an inner class of `TimelineActivity`.

src/com/marakana/TimelineActivity.java: TimelineReceiver inner class

```
...
class TimelineReceiver extends BroadcastReceiver { // ❶
  @Override
  public void onReceive(Context context, Intent intent) { // ❷
    cursor.requery(); // ❸
    Log.d("TimelineReceiver", "onReceive'd");
  }
}
...
```

❶ Like before, to create a broadcast receiver, we subclass `BroadcastReceiver` class.

❷ The only method we need to override is `onReceive()`. This is where we put the work we want done when this receiver is triggered.

❸ The work we want done is to simply tell `cursor` object to refresh itself. We do this by invoking `requery()` which executes the same query that was executed initially to obtain this cursor object.

At this point, our receiver is ready but not registered. Unlike BootReceiver where we registered our receiver with the system statically via the manifest file, we'll register `TimelineReceiver` programmatically. This is because `TimelineReceiver` only makes sense within `TimelineActivity` (it's refreshing its view).

src/com/marakana/TimelineActivity.java

```
...
@Override
protected void onResume() {
  super.onResume();

  // Get the data from the database
  cursor = db.query(DbHelper.TABLE, null, null, null, null, null,
      DbHelper.C_CREATED_AT + " DESC");
  startManagingCursor(cursor);

  // Create the adapter
  adapter = new TimelineAdapter(this, cursor);
  listTimeline.setAdapter(adapter);

  // Register the receiver
  registerReceiver(receiver, filter);   // ❶
}

@Override
protected void onPause() {
  super.onPause();

  // UNregister the receiver
  unregisterReceiver(receiver);  // ❷
}
...
```

❶ We register the receiver in `onResume()` so it's registered whenever the TimelineActivity is running. Recall that all paths to *Running* state go via `onResume()` as described in Section 1.3.1.2.

❷ Similarly, we unregister the receiver on way to *Stopped* state (recall Section 1.3.1.4). `onPause()` is a good place to do that.

What's missing now is the explanation of what `filter` is. To specify what triggers the receiver, we need an instance of `IntentFilter`. Intent filter is simply a filter for intent actions. In this case, we make up an action string that we filter based on.

src/com/marakana/TimelineActivity.java: onCreate()

```
...
filter = new IntentFilter("Marakana.Twitter.NEW_STATUS"); // ❶
...
```

❶ Create new instance of `IntentFilter` to filter for intent action *"Marakana.Twitter.NEW_STATUS"*

6.5 Broadcasting Intents

Finally, to trigger the filter, we need to broadcast an intent that matches the action that the intent filter is listening for. In case of BootReceiver earlier, we didn't have to do this since the system was already broadcasting the appropriate intent. However, for TimelineReceiver, the broadcast is ours to do since the intent is very specific to our application.

src/com/marakana/UpdaterService.java: UpdaterRunnable inner class

```
...
// The actual work of connecting to twitter and getting latest data
private void pullFromTwitter() {
  boolean hasNewStatus = false;  // ❶

  List<Status> timeline = null;
  try {
    timeline = getTwitter().getFriendsTimeline();
  } catch (TwitterException e) {
    e.printStackTrace();
  }
  if (timeline == null)
    return; // Didn't get anything from twitter

  ContentValues values = new ContentValues();
  // Loop over the timeline and print it out
  for (Status status : timeline) {
    // Insert into database
    values.put(DbHelper.C_ID, status.id);
    values.put(DbHelper.C_CREATED_AT, status.createdAt.getTime());
    values.put(DbHelper.C_SOURCE, status.source);
    values.put(DbHelper.C_TEXT, status.text);
    values.put(DbHelper.C_USER, status.user.name);
    try {
      db.insertOrThrow(DbHelper.TABLE, null, values);
      Log.d(TAG, String.format("%s: %s", status.user.name, status.text));
      hasNewStatus = true; // ❷
    } catch (SQLException e) {
    }
  }// for
  // Do we have new statuses?
  if (hasNewStatus) { // ❸
    // Send a broadcast
    sendBroadcast(new Intent("Marakana.Twitter.NEW_STATUS")); // ❹
  }
}
...
```

❶ We define a boolean flag to represent if we have any new statuses from Twitter. By default, we don't.

❷ Remember that we do our `db.inser()` inside of a try-catch block because it may fail when it encounters duplicate id's from statuses that we already have in the database. This is a normal behavior and that's why we have a try-catch block. But, we can also use it to figure out if a status is new or not by simply flipping the switch *after* the `db.insert()` statement.

❸ Once our `hasNewStatus` flag is true, we can notify TimelineActivity to refresh itself.

❹ We send the broadcast via `sendBroadcast()` call and an intent with the same *"Marakana.Twitter.NEW_STATUS"* message as the one our intent filter is filtering for.

Note
UpdaterService may be sending broadcasts even when the TimelineReceiver is not registered. That is perfectly fine. Those broadcasts will simply be ignored.

At this point, a new status received by UpdaterService causes an intent to be broadcasted over to the TimelineActivity which gets received by the TimelineReceiver which in turn refreshes the ListView of statuses.

6.6 Summary

MyTwitter is now complete and ready for prime time. Our application can now send status updates to Twitter, get latest statuses from our friends as well as get started on boot time and refreshed live when a new status is received.

6.6.1 Source Code

Source code can be downloaded from http://marakana.com/static/courseware/android/MyTwitter-4.zip

Chapter 7

Web Browser

Web Browser example demonstrates how to use WebView widget in order to render HTML content inside of your Android apps. HTML, especially HTML5 is very important. It has the potential to become the common denominator for most mobile apps.

Android uses WebKit open source browser engine. WebKit is extremely fast and powerful and as such is the engine not just for Android but for Safari, OSX Dashboard and Mail apps, Chrome, and so on.

With emergence of HTML5 standard, WebKit becomes even more important for developing portable mobile applications as it supports many features previously done natively on each platform.

Android wraps WebKit into WebView, a widget that can be placed anywhere on Android UI.

7.1 Main Layout

The layout of our web browser is fairly simple. We have a field to enter URL and a Go button to start loading it. The actual HTML content that is loaded is rendered below and takes most of the screen.

res/layout/main.xml

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

  <!-- ❶ -->
  <LinearLayout android:orientation="horizontal"
    android:layout_width="fill_parent" android:layout_height="wrap_content">

    <EditText android:id="@+id/url" android:layout_height="wrap_content"
      android:layout_width="wrap_content" android:lines="1"
      android:layout_weight="1.0" android:hint="http://"
      android:visibility="visible" />

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

  </LinearLayout>
  <!-- ❷ -->
  <WebView android:id="@+id/webview_compontent"
    android:layout_width="fill_parent" android:layout_height="fill_parent"
    android:layout_weight="1.0" />
</LinearLayout>
```

❶ Linear layout running horizontally that contains our URL field as well as the Go button.

❷ WebView is just like any other widget we place into the layout. Main attributes are `id`, `layout_height` and `layout_width`. We also set the `layout_weight` to have it not push out the URL bar, yet still take all the available screen space.

7.2 WebBrowser Activity

This is the main activity of our web browser. For the most part, we setup our UI handling here. The notable exception is a workaround we have to do for WebView to behave properly.

WebView by default will open any subsequent URL request via the default system Browser app. What that means is that if you navigate to, say http://google.com, and Google redirects you to their mobile version of the site at http://m.google.com, the new page will be opened by system Browser application and not our WebBrowser. This is probably not the desired user experience in our case.

To work around this, we create an instance of `WebViewClient` and setup WebView to use our client to handle subsequent URL loading.

src/com/marakana/WebBrowser.java

```
package com.marakana;

import android.app.Activity;
import android.os.Bundle;
import android.view.KeyEvent;
import android.view.View;
import android.view.View.OnClickListener;
import android.view.View.OnKeyListener;
import android.webkit.WebView;
import android.webkit.WebViewClient;
import android.widget.Button;
import android.widget.EditText;

public class WebBrowser extends Activity {
  private WebView webView; // ❶
  private EditText urlField;
  private Button goButton;

  /** Called when the activity is first created. */
  @Override
  public void onCreate(Bundle savedInstanceState) {
    super.onCreate(savedInstanceState);
    setContentView(R.layout.main);

    // Create reference to UI elements
    webView = (WebView) findViewById(R.id.webview_compontent);
    urlField = (EditText) findViewById(R.id.url);
    goButton = (Button) findViewById(R.id.go_button);

    // workaround so that the default browser doesn't take over
    webView.setWebViewClient(new MyWebViewClient()); // ❷

    // Setup click listener
    goButton.setOnClickListener(new OnClickListener() { // ❸
      public void onClick(View view) {
        webView.loadUrl(urlField.getText().toString()); // ❹
      }
    });

    // Setup key listener
    urlField.setOnKeyListener(new OnKeyListener() { // ❺
      public boolean onKey(View view, int keyCode, KeyEvent event) {
        if (keyCode == KeyEvent.KEYCODE_ENTER) {
          webView.loadUrl(urlField.getText().toString());
          return true;
        } else {
          return false;
        }
      }
    });

  }

  // Our WebViewClient, to keep opening URLs within our component by default ❻
  private class MyWebViewClient extends WebViewClient {
    @Override
    public boolean shouldOverrideUrlLoading(WebView webView, String url) {  // ❼
      webView.loadUrl(url);
      return true;
    }
  }
}
```

❶ The local reference to the WebView loaded from XML resource.

❷ We create a new instance of `MyWebViewClient` and set it up as the client for WebView to use.

❸ Button clicks are handled here via this anonymous inner class.

❹ When the button is clicked, we tell the WebView to load the URL that we have in the URL field.

❺ Keyboard entires are handled here. Handling is very similar to button handling.

❻ This is where we define our custom version of WebViewClient, to specify how subsequent URL loading is done.

❼ We change default behavior, we override `shouldOverrideUrlLoading()`.

7.3 Manifest File

The manifest file for WebBrowser is fairly standard. We include it here for completeness.

AndroidManifest.xml

```
<?xml version="1.0" encoding="utf-8"?>
<manifest xmlns:android="http://schemas.android.com/apk/res/android"
  package="com.marakana" android:versionCode="1" android:versionName="1.0.0">
  <uses-permission android:name="android.permission.INTERNET" />
  <application android:icon="@drawable/icon" android:label="@string/app_name">
    <activity android:name=".WebBrowser" android:label="@string/app_name">
      <intent-filter>
        <action android:name="android.intent.action.MAIN" />
        <category android:name="android.intent.category.LAUNCHER" />
      </intent-filter>
    </activity>
  </application>
</manifest>
```

7.4 Summary

At this point, your WebBrowser should work. If you type in a URL such as *http://google.com* and the website redirects you to *http://m.google.com*, you should still see the actual site within your own WebView.

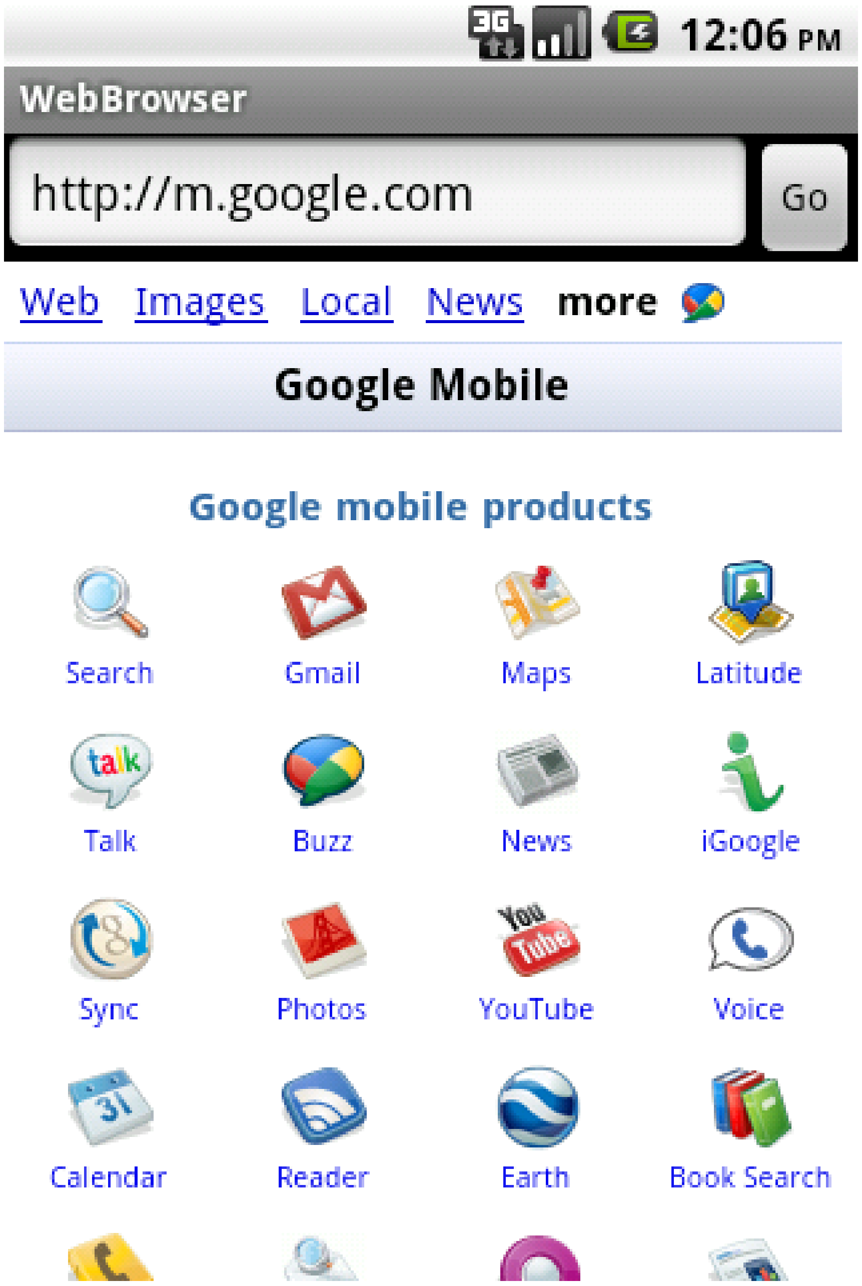


Figure 7.1: WebBrowser

7.4.1 Source Code

Source code can be downloaded from http://marakana.com/static/courseware/android/WebBrowser.zip

Chapter 8

Compass

The Compass example illustrates two main points:

- Sensors - what they are and how to use them
- Custom Widgets - how to create custom UI components

8.1 Compass Main Activity

The main Compass activity sets the Rose as its only widget on the screen. It also registers with SensorManager to listen to sensor events, and updates the Rose orientation accordingly.

src/com/marakana/Compass.java

```
package com.marakana;

import android.app.Activity;
import android.hardware.Sensor;
import android.hardware.SensorEvent;
import android.hardware.SensorEventListener;
import android.hardware.SensorManager;
import android.os.Bundle;
import android.util.Log;
import android.view.Window;
import android.view.WindowManager;

// implement SensorListener
public class Compass extends Activity implements SensorEventListener { // ❶
  SensorManager sensorManager; // ❷
  Sensor sensor;
  Rose rose;

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

    // Set full screen view ❸
    getWindow().setFlags(WindowManager.LayoutParams.FLAG_FULLSCREEN,
        WindowManager.LayoutParams.FLAG_FULLSCREEN);
    requestWindowFeature(Window.FEATURE_NO_TITLE);

    // Create new instance of custom Rose and set it on the screen
    rose = new Rose(this); // ❹
    setContentView(rose); // ❺

    // Get sensor and sensor manager
    sensorManager = (SensorManager) getSystemService(SENSOR_SERVICE); // ❻
    sensor = sensorManager.getDefaultSensor(Sensor.TYPE_ORIENTATION); // ❼

    Log.d("Compass", "onCreate'd");
  }

  // Register to listen to sensors
  @Override
  public void onResume() {
    super.onResume();
    sensorManager.registerListener(this, sensor,
        SensorManager.SENSOR_DELAY_NORMAL); // ❽
  }

  // Unregister the sernsor listener
  @Override
  public void onPause() {
    super.onPause();
    sensorManager.unregisterListener(this); // ❾
  }

  // Ignore accuracy changes
  public void onAccuracyChanged(Sensor sensor, int accuracy) { // ❿
  }

  // Listen to sensor and provide output
  public void onSensorChanged(SensorEvent event) { // ⓫
    int orientation = (int) event.values[0]; // ⓬
    rose.setDirection(orientation); // ⓭
  }
}
```

❶ Since `Compass` listens to sensor events, it needs to implement `SensorEventListener` interface.

❷ We define local variable for Sensor, SensorManager and Rose.

❸ The window manager flags to set the activity into full-screen mode.

❹ We create a new instance of Rose widget, our custom compass rose.

❺ In this case, the activity content is the single Rose widget. This is unlike the usual reference to an XML layout resource.

❻ We get the sensor manager from the system service.

❼ From the sensor manager, we can obtain the actual sensor object that we are interested in.

8 We register to listen to sensor updates in activity's `onResume()` method since all roads to *Running* state (described in Section 1.3.1.2) of the activity lead through `onResume()`. We don't want to register in `onCreate()` because it'd be wasteful to listen and process sensor updates even when our activity is not in the foreground. Recall *Activity Lifecycle* in Section 1.3.1 for details.

9 We unregister from sensor updates in `onPause()` since this is the callback called whenever our activity is about to go into background and thus user cannot see further updates.

10 `onAccuracyChanged()` is called when sensor's accuracy changes. It is not applicable in this example but we still have to implement is as part of `SensorEventListener` interface.

11 `onSensorChanged()` is called whenever sensor changes. The particular information about the change is stored in `SensorEvent`.

12 We are particularly interested in new values reported.

13 Once we have the new orientation, we update our Rose widget to rotate accordingly.

This illustrates all the work related to sensors we need to do. Again, we get the sensor manager, and register for updates. Once sensor changes, it reports values back to our sensor event listener.

Note
Sensor data is erratic. Also, sensors are not supported by the emulator so to really test your application, you'd need a physical device with support for orientation sensor. Most Android phones do have that support.

8.2 Custom Rose Widget

Rose represents our custom widget. It is simply an image of a compass rose that can be rotated as to behave like a real compass.

src/com/marakana/Rose.java

```
package com.marakana;

import android.content.Context;
import android.graphics.Canvas;
import android.widget.ImageView;

public class Rose extends ImageView { // 1
  int direction = 0;

  public Rose(Context context) {
    super(context);

    this.setImageResource(R.drawable.compassrose); // 2
  }

  // Called when component is to be drawn
  @Override
  public void onDraw(Canvas canvas) { // 3
    int height = this.getHeight();  // 4
    int width = this.getWidth();
    canvas.rotate(direction, width / 2, height / 2); // 5
    super.onDraw(canvas); // 6
  }

  // Called by Compass to update the orientation
  public void setDirection(int direction) { // 7
    this.direction = direction;
    this.invalidate(); // request to be redrawn 8
  }

}
```

1 The easiest way to create a new widget is to subclass an existing one. If you are creating a totally custom widget, start by subclassing most basic one in the SDK, namely `View`. Since our widget is a special type of image, we start from `ImageView`.

2 `ImageView` already knows how to set an image as its content. We just specify to `super` what image resource to use. Note that `compassrose.jpg` is in our `/res/drawable` folder.

3 `onDraw()` is the method that layout manager calls to have each widget draw itself. It also passes the `Canvas` to this method. This method is where you typically do any custom drawing to the canvas.

4 Once we have the canvas, we can figure out its size.

5 We simply rotate the entire canvas for some direction (in degrees) around its mid point.

6 We tell `super` to draw the image on this rotated canvas. At this point we have our rose drawn at an angle.

7 `setDirection()` is called by Compass activity to update the direction of the rose based on values that sensor manager reported.

8 Calling `invalidate()` on a View marks it for redraw, which happens later via a call to `onDraw()`.

8.3 Summary

At this point your compass application is complete. You should now have a good understanding of how sensors work from SDK point of view. You should also know how to create a simple custom widget for your app.

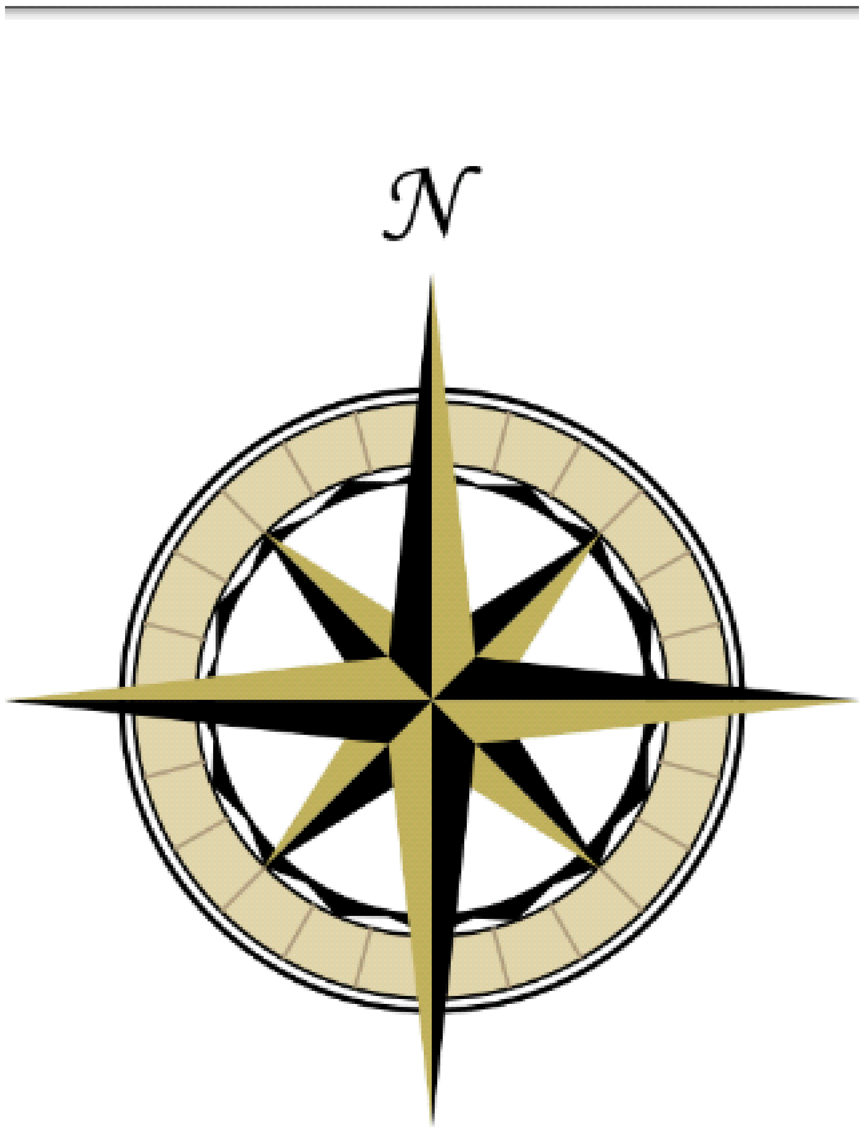


Figure 8.1: Compass App

8.3.1 Source Code

Source code can be downloaded from http://marakana.com/static/courseware/android/Compass.zip

Chapter 9

Where Am I

This example illustrates how to use location based services in Android. First, we use `LocationManager` to figure out our current location based on GPS. Secondly, we use `Geocoder` to convert this location to an address.

9.1 Layout

The layout for this example is trivial. We have a `TextView` for the title, and a `TextView` for the output. Since the output could be longer than the screen size, we wrap the output in `ScrollView`.

res/layout/main.xml

```
<?xml version="1.0" encoding="utf-8"?>
<LinearLayout xmlns:android="http://schemas.android.com/apk/res/android"
  android:layout_height="fill_parent" android:layout_width="fill_parent"
  android:background="#fff" android:orientation="vertical">
  <!-- ❶ -->
  <TextView android:layout_width="wrap_content"
    android:layout_height="wrap_content" android:layout_gravity="center"
    android:textColor="#333" android:textSize="30dp" android:text="@string/title"></ ↩
        TextView>
  <!-- ❷ -->
  <ScrollView android:layout_height="fill_parent"
    android:layout_width="fill_parent">
    <!-- ❸ -->
    <TextView android:textColor="#333" android:layout_gravity="center"
      android:layout_height="fill_parent" android:layout_width="fill_parent"
      android:gravity="center" android:textSize="25dp" android:text="Waiting..."
      android:id="@+id/textOut"></TextView>
  </ScrollView>
</LinearLayout>
```

❶ Title for our application.

❷ `ScrollView` to enable scrolling if the output grows beyond the size of the screen.

❸ `TextView` to represent the output. It will be programmatically set from the WhereAmI activity.

9.2 Activity

The main activity that sets up the screen, connects to `LocationManager` and uses the `Geocoder` to figure out our address.

LocationManager uses location providers, such as GPS or Network, to figure out our current location. The location is expressed as latitude and longitude values.

Geocoder searches an online database for known addresses in the vicinity of the location provided. It may come up with multiple results, some more specific than others.

src/com/marakana/WhereAmI.java

```
package com.marakana;

import java.io.IOException;
import java.util.List;

import android.app.Activity;
import android.location.Address;
import android.location.Geocoder;
import android.location.Location;
import android.location.LocationListener;
import android.location.LocationManager;
import android.os.Bundle;
import android.widget.TextView;

public class WhereAmI extends Activity implements LocationListener { // ❶
  LocationManager locationManager; // ❷
  Geocoder geocoder; // ❸
  TextView textOut; // ❹

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

    textOut = (TextView) findViewById(R.id.textOut);

    locationManager = (LocationManager) getSystemService(LOCATION_SERVICE); // ❺
    geocoder = new Geocoder(this); // ❻

    // Initialize with the last known location
    Location lastLocation = locationManager
        .getLastKnownLocation(LocationManager.GPS_PROVIDER); // ❼
    if (lastLocation != null)
      onLocationChanged(lastLocation);
  }

  @Override
  protected void onResume() { // ❽
    super.onRestart();
    locationManager.requestLocationUpdates(LocationManager.GPS_PROVIDER, 60000,
        10, this);
  }

  @Override
  protected void onPause() { // 9
    super.onPause();
    locationManager.removeUpdates(this);
  }

  // Called when location has changed
  public void onLocationChanged(Location location) { // 10
    String text = String.format(
        "Lat:\t %f\nLong:\t %f\nAlt:\t %f\nBearing:\t %f",
        location.getLatitude(), location.getLongitude(),
        location.getAltitude(), location.getBearing()); // 11
    textOut.setText(text);

    // Perform geocoding for this location
    try {
      List<Address> addresses = geocoder.getFromLocation(
          location.getLatitude(), location.getLongitude(), 10); // 12
      for (Address address : addresses) {
        textOut.append("\n" + address.getAddressLine(0)); // 13
      }
    } catch (IOException e) {
      e.printStackTrace();
    }
  }

  // Methods required by LocationListener 14
  public void onProviderDisabled(String provider) {
  }

  public void onProviderEnabled(String provider) {
  }

  public void onStatusChanged(String provider, int status, Bundle extras) {
  }

}
```

1 Notice that `WhereAmI` implements `LocationListener`. This is the interface that `LocationManager` uses to notify us of changes to location.

2 Local reference to `LocationManager`.

3 Local reference to `Geocoder`.

4 `textOut` is the text view that we print our output to for user to see.

5 We get the local reference to `LocationManager` by asking the context to get the location manager system service.

6 We create a new instance of `Geocoder` and pass the current context to it.

7 Location manager memorizes its last known location. This is useful since it may take a while to get the lock on the new location.

8 As usual, we register in `onResume()`, since that is the method that is called en route to *Running* state. We use location manager's `requestLocationUpdates()` method to register for updates.

(9) We unregister in `onPause()`, just before the activity goes into *Stopped* state. This is important because GPS tends to use a lot of power, so listening to updates even while the app is in the background would waste a lot of battery power.

(10) `onLocationChanged()` is the callback method called by the location manager when it detects that the location has changed.

(11) We get the `Location` object that contains a lot of useful information about current location. We create a human-readable string with this info.

(12) Once we have the location, we can try to "geocode" the location, a process of converting latitude and longitude to a known address.

(13) If we do find known addresses for this location, we print them out.

(14) These are some other callback methods that are part of `LocationListener` interface. We don't use them for purposes of this example.

9.3 Manifest File

The manifest file for this app is fairly standard. Notice that in order to be able to register as location listener, we have to hold the appropriate permissions.

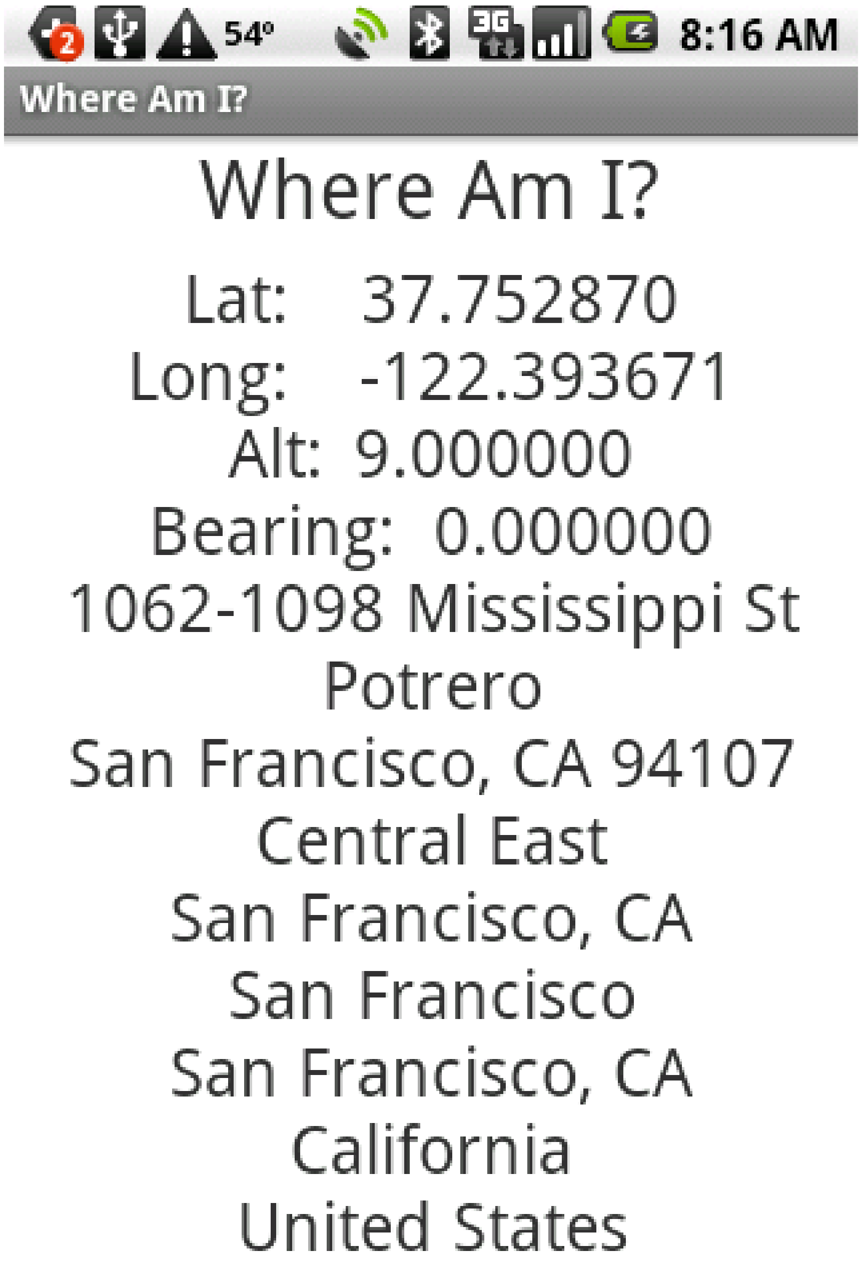
AndroidManifest.xml

```
<?xml version="1.0" encoding="utf-8"?>
<manifest xmlns:android="http://schemas.android.com/apk/res/android"
  package="com.marakana" android:versionCode="1" android:versionName="1.0">
  <application android:icon="@drawable/icon" android:label="@string/app_name">
    <activity android:name=".WhereAmI" android:label="@string/app_name">
      <intent-filter>
        <action android:name="android.intent.action.MAIN" />
        <category android:name="android.intent.category.LAUNCHER" />
      </intent-filter>
    </activity>
  </application>
  <uses-sdk android:minSdkVersion="8" />
  <!-- (1) -->
  <uses-permission android:name="android.permission.ACCESS_FINE_LOCATION" />
</manifest>
```

(1) Declares that this app uses location providers. Location permissions could be `android.permission.ACCESS_FINE_LOCATI` for GPS provider or `android.permission.ACCESS_COARSE_LOCATION` for network provider.

9.4 Summary

At this point, your WhereAmI application is complete. It illustrates how to use LocationManager to get the actual location via a specific location provider (such as GPS or network) and how to convert that location into a known addresses via Geocoder.



Where Am I?

Lat: 37.752870
Long: -122.393671
Alt: 9.000000
Bearing: 0.000000
1062-1098 Mississippi St
Potrero
San Francisco, CA 94107
Central East
San Francisco, CA
San Francisco
San Francisco, CA
California
United States

Figure 9.1: WhereAmI

9.4.1 Source Code

Source code can be downloaded from http://marakana.com/static/courseware/android/WhereAmI.zip

Chapter 10

Unit Testing Android Applications

Unit testing is very important to ensure quality of your code. Every unit of your code should have a corresponding code to test it. Unit testing should also be automated.

Android uses JUnit testing framework.

10.1 About JUnit

JUnit is a testing framework intended to promote structured approach to automated unit testing of software written in Java

- It represents a reusable pattern (structure) for testing Java code
- It was developed by Erich Gamma (co-author of *Design Patterns*) and Kent Back (known for work on Extreme Programming) in 1997
  - It is an instance of xUnit testing architecture that offers instances for testing other languages such as C/C++, Python, and Perl
- JUnit is now an open-source project hosted by Source Forge
  - It is distributed under the Common Public License thereby making it easier to be incorporated into commercial tools
- In the context of JUnit, a "unit test""”"" refers to testing "unit of work"" such as a method, a group of related methods, or even a class
  - The key distinction is isolation of the unit from other units
  - JUnit is not intended for testing unit interaction such as integration testing of components

10.1.1 JUnit 101

JUnit is a combination of two concepts

- Design patterns – the framework in its structure represents an instance of a Command design pattern
- Assertions – the framework makes use of assertions to generate output of individual tests

The JUnit is meant to follow the structure of the code

- For each Java class, typically there is a JUnit Test class
- Each JUnit Test class extends the TestCase base class, which provides the JUnit framework interfaces
- Within the JUnit test class, individual test[UnitTestName] methods are defined to test specific units, such as class methods
- With each test[UnitTestName] method, the assert[Type] assertions are used to check desired testing conditions

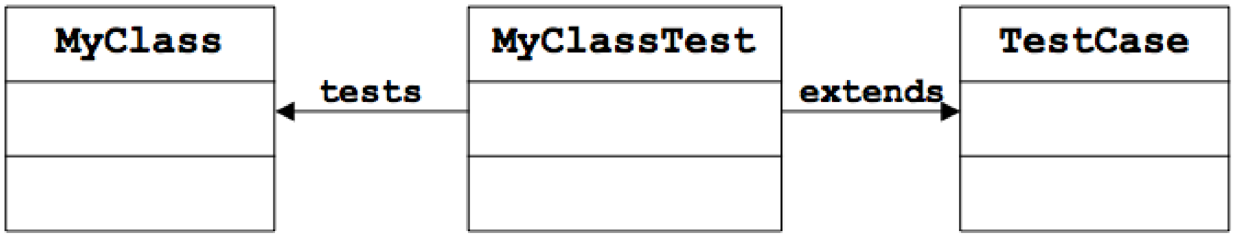


Figure 10.1: JUnit Diagram

10.1.2 Advantages of JUnit

JUnit tests are easy to write since they are written in Java, and they are more reflective of the software requirements

- For each test, typically we need to decide how to define the test, what test variables to instantiate, and how to analyze the output
- With JUnit, we need to focus on the test itself, as the framework resolves questions about the test case instantiation and output (e.g., setUp method for setting up shared test variables)

Unit test cases costs less since much of the creation and maintenance overhead is eliminated through the framework

- The need for defining a new test suite structure for every module is eliminated
- JUnit structure does not change but additional suites can be added or existing ones extended
- Overhead of deciding how to create individual test cases is reduced through a familiar test-suite assertions interface
- Individual test cases run independently of each other so the need for testing coordination is decreased

JUnit tests internally check results and instantly provide feedback

- With the use of assertions, the need for an output language and its analysis is eliminated
- JUnit provides a summary of passed and failed tests, with corresponding details for each one of the tests that failed

The use of JUnit tests leads to increase in software quality

- By promoting simplicity of writing test cases and decreasing cost of creating, running, and maintaining test suites, the appeal of testing to developers should increase
- Given that the JUnit suites can be rerun inexpensively after each change or a set of changes, disruption to the expected software operation can be detected earlier
- Test-driven development promotes focus on satisfying software requirements, and in turn leads to higher quality software

JUnit provides a method for creation of hierarchical test suites

- Individual test classes can be composed into test suits
- Test suits can be composed into other test suits to create a hierarchy of testing modules that reflects the code structure

10.1.3 Disadvantages of JUnit

JUnit does not guarantee better quality software

- It is a framework that promotes the use of testing in software development
- Software systems are inherently complex and their inherent complexity is present even at the lowest levels of abstraction
- Creation of test cases and testing suites in highly complex or critical domains may require a more dedicated approach to testing such as the use of formal methods or simulations
- Moreover, using JUnit by itself will not improve quality unless it is a part of a structured system testing approach

JUnit does not invalidate previous testing methods

- It provides a framework that simplifies test case definition and use, but the analysis algorithms such as boundary-case analysis still apply

10.1.4 Android JUnit Implementation

While Android does adopt JUnit framework, it also makes some changes to it. The following example illustrates that.

10.2 Application to be Tested

First, we need an application that we want to test. In this case, we picked a simple unit converter that converts metric kilograms to imperial pounds. It has to fields, one for kilos and the other one for pounds. As user starts typing into one of those fields, the value in the other unit is automatically calculated by the system.

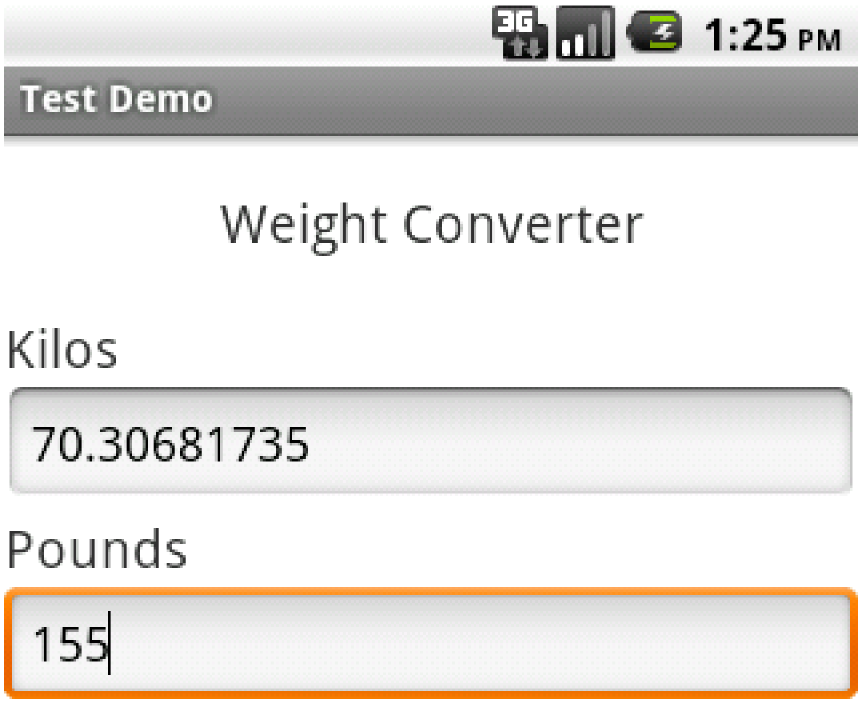


Figure 10.2: TestDemo Screen

10.2.1 Kilo/Pound Converter Layout

The layout for the screen consists of two EditText fields and couple of TextViews. Notice that there are not buttons. In this case, we are listening to keyboard inputs and processing the conversion as user types.

res/layout/main.xml

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

```
<LinearLayout xmlns:android="http://schemas.android.com/apk/res/android"
  android:orientation="vertical" android:layout_width="fill_parent"
  android:layout_height="fill_parent" android:layout_gravity="center">
  <TextView android:layout_width="fill_parent"
    android:layout_height="wrap_content" android:text="@string/titleConverter"
    android:gravity="center" android:textSize="20sp"
    android:layout_margin="20dp" />
  <TextView android:layout_width="wrap_content"
    android:layout_height="wrap_content" android:text="@string/titleKilos"
    android:textSize="20sp"></TextView>
  <EditText android:layout_height="wrap_content" android:hint="@string/titleKilos"
    android:id="@+id/editKilos" android:layout_width="fill_parent"></EditText>
  <TextView android:layout_width="wrap_content"
    android:layout_height="wrap_content" android:text="@string/titlePounds"
    android:textSize="20sp"></TextView>
  <EditText android:layout_height="wrap_content" android:id="@+id/editPounds"
    android:hint="@string/titlePounds" android:layout_width="fill_parent"></EditText>
</LinearLayout>
```

10.2.2 Kilo/Pound Converter Activity

The main activity loads up the XML layout for the view and sets up two key listeners for each of the two input fields. As keys are pressed, it performs the calculations and updates the other field.

src/com/marakana/TestDemo.java

```
package com.marakana;

import android.app.Activity;
import android.os.Bundle;
import android.util.Log;
import android.view.KeyEvent;
import android.view.View;
import android.view.View.OnKeyListener;
import android.widget.EditText;

public class TestDemo extends Activity {
  static final String TAG = "TestDemo";
  EditText editKilos, editPounds;
  public static final String ERROR = "ERROR";

  /** Called when the activity is first created. */
  @Override
  public void onCreate(Bundle savedInstanceState) {
    super.onCreate(savedInstanceState);
    setContentView(R.layout.main);

    // Find views
    editKilos = (EditText) findViewById(R.id.editKilos);
    editPounds = (EditText) findViewById(R.id.editPounds);

    // Setup listener for Kilos to Pounds
    editKilos.setOnKeyListener(new OnKeyListener() { // ❶
          public boolean onKey(View view, int keyCode, KeyEvent event) {
            if (event.getAction() != KeyEvent.ACTION_UP) // ❷
            return false;
          try {
            Log.d(TAG,
                String.format("Kilos: %s", editKilos.getText().toString()));
            double kilos = Double.parseDouble(editKilos.getText().toString());
            double pounds = kilos * 2.20462262; // ❸
            editPounds.setText(new Double(pounds).toString()); // ❹
          } catch (NumberFormatException e) { // ❺
            editPounds.setText(ERROR);
            Log.e(TAG, "e:" + e);
          }
          return true;
        }
      });

    // Setup listener for Pounds to Kilos
    editPounds.setOnKeyListener(new OnKeyListener() {
      public boolean onKey(View view, int keyCode, KeyEvent event) {
        if (event.getAction() != KeyEvent.ACTION_UP)
          return false;
        try {
          Log.d(TAG,
              String.format("Pounds: %s", editPounds.getText().toString()));
          double pounds = Double.parseDouble(editPounds.getText().toString());
          double kilos = pounds * 0.45359237;
          editKilos.setText(new Double(kilos).toString());
        } catch (NumberFormatException e) {
          editKilos.setText(ERROR);
          Log.e(TAG, "e:" + e);
        }
        return true;
      }
    });

  }
}
```

❶ The `OnKeyListener` is implemented as an anonymous inner class at attached to kilos field.

❷ We only care about key event `ACTION_UP` meaning that the key has been released.

❸ 1 lb is about 2.2 kg

❹ Once we have the conversion done, we update the pounds field.

❺ Conversion may fail because user didn't enter valid values. In that case, we simply print *ERROR*.

The explanation for pounds-to-kilos conversion is the same as the one above.

10.3 Application Performing Testing

The application testing the unit converter is a whole other application. So, we have one Android app testing another. The reason for this is to keep the testing separate from the tester code.

10.3.1 TestDemoTests Test Case

TestDemoTests is our unit testing code.

src/com/marakana/test/TestDemoTests.java

```
package com.marakana.test;

import android.test.ActivityInstrumentationTestCase2;
import android.test.TouchUtils;
import android.test.ViewAsserts;
import android.test.suitebuilder.annotation.SmallTest;
import android.widget.EditText;

import com.marakana.TestDemo;

/*
 * Test code to test com.marakana.TestDemo
 *
 * To run on command line:
 * adb -e shell am instrument -w -e class com.marakana.test.TestDemoTests
 *             com.marakana.test/android.test.InstrumentationTestRunner
 */
public class TestDemoTests extends ActivityInstrumentationTestCase2<TestDemo> { // ❶
  EditText editKilos, editPounds;
  TestDemo activity; // ❷

  public TestDemoTests(String name) { // ❸
    super("com.marakana", TestDemo.class);
    setName(name);
  }

  protected void setUp() throws Exception { // ❹
    super.setUp();

    // Find views
    activity = getActivity(); // ❺
    editKilos = (EditText) activity.findViewById(com.marakana.R.id.editKilos); // ❻
    editPounds = (EditText) activity.findViewById(com.marakana.R.id.editPounds); // ❼
  }

  protected void tearDown() throws Exception { // ❽
    super.tearDown();
  }

  @SmallTest
  public void testViewsCreated() {  // ❾
    assertNotNull(getActivity());
    assertNotNull(editKilos);
    assertNotNull(editPounds);
  }

  @SmallTest
  public void testViewsVisible() { // ❿
    ViewAsserts.assertOnScreen(editKilos.getRootView(), editPounds);
    ViewAsserts.assertOnScreen(editPounds.getRootView(), editKilos);
  }
  @SmallTest
  public void testStartingEmpty() { // ⓫
    assertTrue("Kilos field is empty",
        "".equals(editKilos.getText().toString()));
    assertTrue("Pounds field is empty",
        "".equals(editPounds.getText().toString()));
  }

  @SmallTest
  public void testKilosToPounds() { // ⓬
    editKilos.clearComposingText();
    editPounds.clearComposingText();

    TouchUtils.tapView(this, editKilos); // ⓭
    sendKeys("1"); // ⓮

    double pounds;
    try {
      pounds = Double.parseDouble(editPounds.getText().toString());
    } catch (NumberFormatException e) {
      pounds = -1;
    }
    assertTrue("1 kilo is 2.20462262 pounds", pounds > 2.2 && pounds < 2.3); // ⓯
  }

}
```

1. Unlike regular JUnit TestCase, here we subclass `ActivityInstrumentationTestCase2` - a special kind of TestCase for testing activities.

2. `activity` is reference to our unit converter activity that we are testing.

3. The constructor specifies what code we are testing by calling the constructor in `super`.

4. `setUp()` is the method provided by JUnit. It is a good place to setup your test environment.

5. We get the reference to our activity that we are testing.

6. `editKilos` is the kilos text field in our unit converter.

7. `editPounds` is the text field for the pounds in TestDemo activity.

8. `tearDown()` is JUnit placeholder to undo any work previously done in `setUp()`, if needed.

9. `testViewsCreated()` is a simple test that verifies that our objects actually exist. Notice the use of `assertNotNull()` as one of many standard JUNit assert methods. Also, notice the use of `@SmallTest` annotation to communicate that this is a single test case. Prior to Java5, test cases had to be named starting with the word `test`. That's not necessary any more, however still a practice of many.

10. `testViewsVisible()` is another test demonstrating the use of a non-standard assert methods provided by Android. `ViewAsserts.assertOnScreen()` is an example of an Android extension to JUnit specifically for testing UI elements.

11. `testStartingEmpty()` is a simple test that checks that the edit fields are empty when the application is launched.

⓬ `testKilosToPounds()` performs the actual user input and verifies the output.

⓭ Notice the use of `TouchUtils.tapView()` to simulate the touch of the screen. This is another Android extension to JUnit specific to UI.

⓮ `sendKeys()` simulates clicking a key on the keyboard.

⓯ And finally we verify that the output is indeed what we'd expect it to be.

You can keep on adding more tests to this TestDemoTests test case. Typically, you'd have at least one test for each public method but the actual coverage depends on your app and testing requirements.

10.3.2 Android Manifest File

Notice that the manifest file looks different here than in most applications we've seen so far. The key difference is the use of `<uses-library>` and `<instrumentation>` elements

AndroidManifest.xml

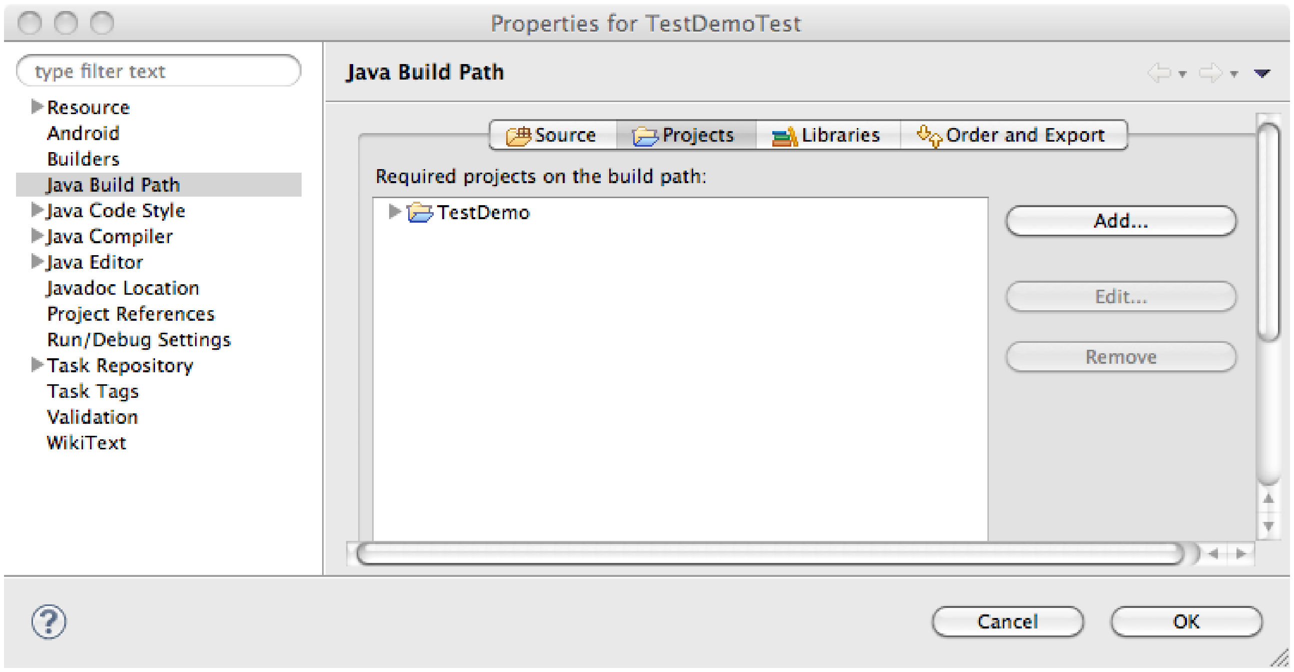
```
<?xml version="1.0" encoding="utf-8"?>
<manifest xmlns:android="http://schemas.android.com/apk/res/android"
  package="com.marakana.test" android:versionCode="1"
  android:versionName="1.0">
  <application android:icon="@drawable/icon" android:label="@string/app_name">
    <uses-library android:name="android.test.runner" /> <!-- ❶ -->
  </application>
  <uses-sdk android:minSdkVersion="8" />
  <instrumentation android:targetPackage="com.marakana"
    android:name="android.test.InstrumentationTestRunner" /> <!-- ❷ -->
</manifest>
```

❶ Specifies that this application uses the code of another application, namely application we are testing.

❷ Specifies that we're using `android.test.InstrumentationTestRunner` to run our tests.

We are now ready almost ready to run our tests. Before we do that, you need to tell Eclipse about the dependency of one project to another. Namely, our TestDemoTest project depends on TestDemo project. To tell Eclipse about that:

- In Eclipse Project Explorer, select TestDemoTest project.
- Open properties window for TestDemoTest project by choosing File → Properties.
- Select *Java Build Path → Projects*.
- Click on *Add* and choose *TestDemo* project.



Required projects on the build path:

TestDemo

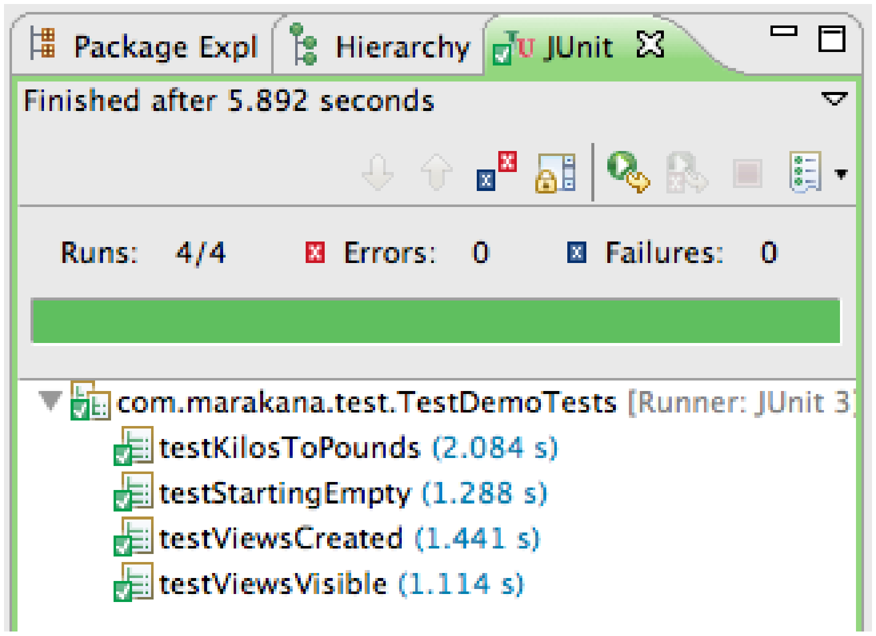
Figure 10.3: TestDemoTest Project Properties

10.4 Running Unit Tests

To run your tests:

- Right-click on TestDemoTest project in Eclipse Project Explorer.
- Choose *Run As* → *Android JUnit Test*.

You should get *JUnit* tab pop up in Eclipse. Your test would actually be performed on the device (e.g. emulator) and you'd see the outcome as it keeps on going.



Finished after 5.892 seconds

Runs: 4/4 Errors: 0 Failures: 0

```
com.marakana.test.TestDemoTests [Runner: JUnit 3]
    testKilosToPounds (2.084 s)
    testStartingEmpty (1.288 s)
    testViewsCreated (1.441 s)
    testViewsVisible (1.114 s)
```

Figure 10.4: Running JUNit Tests in Eclipse

10.5 Summary

Unit testing is very important to maintain the quality and reliability of your Android application. Automated testing based on JUnit framework is a very good solution.

Android SDK integrated the entire JUnit framework, but it also extends it with features necessary to test Activities, Services and other Android-specific components.

You write your test code as a separate application. You can then run your test application in Eclipse and see the results. You can also script running and reporting of the tests using command-line tools.

Appendix A

Android Resources

A.1 Software

To setup your environment, you will need Android SDK and Eclipse vailable at:

- Android SDK http://d.android.com/sdk/index.html
- Eclipse http://www.eclipse.org/downloads/ Choose *Eclipse IDE for Java Developers* (~98 MB)

Alternatively, you may want to choose to run your development inside a virtual machine:

- Marakana Ubuntu VM for Android http://marakana.com/external/VirtualMachines/Marakana-Ubuntu-VM.zip (~1.4GB) Preconfigured Ubuntu virtual machine with everything you need to start developing for Android. Requires VMWare Player.
- VMWare Player is needed to play a virtual image: http://www.vmware.com/products/player/

A.2 Websites

- Android Main Site http://Android.com
- DroidDraw http://droiddraw.org/
- Turbo-charge your UI: How to Make your Android UI Fast and Efficient http://code.google.com/events/io/2009/sessions/-TurboChargeUiAndroidFast.html
- Drawables http://androiddrawableexplorer.appspot.com/
- Sensor Simulator http://code.google.com/p/openintents/wiki/SensorSimulator

A.3 Books

- O'Reilly: Learning Android by Marko Gargenta (coming out Q3 2010)
- Wrox: Professional Android2 Development

A.4 Android Internals Websites

- Linux-Specific Android Portal http://elinux.org/Android_Portal
- Emulator Controls http://developer.android.com/guide/developing/tools/emulator.html
- Android ROMs http://code.google.com/p/android-roms/wiki/TableOfContents
- BusyBox for Android http://benno.id.au/blog/2007/11/14/android-busybox
- Android Platform Development Kit http://pdk.android.com/online-pdk/guide/index.html
- For Building Source on Mac http://www.justinlee.sg/2010/01/08/compiling-android-source-on-mac-os-x-10-6-snow-leopard/
- To Get Android Kernel git clone git://android.git.kernel.org/kernel/common.git kernel

Appendix B

Virtual Machine

You can download the complete Android-ready Ubuntu virtual machine from: http://marakana.com/external/VirtualMachines/-Marakana-Ubuntu-VM.zip (~1.4GB)

You will also need VMWare Player available at: http://www.vmware.com/products/player/

Appendix C

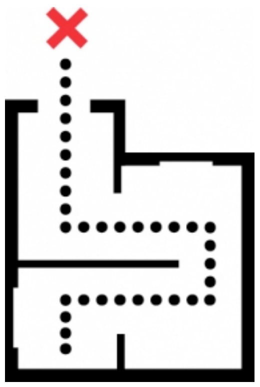
Slides

Attached are various slides for your reference.

Android: A 9,000-foot Overview

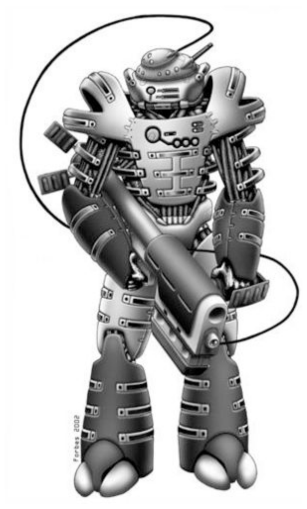
Agenda

- Market Space
- The Stack
- Android SDK
- Hello World!
- Main Building Blocks
- Android User Interface
- Operating System Features
- Debugging
- Summary



History

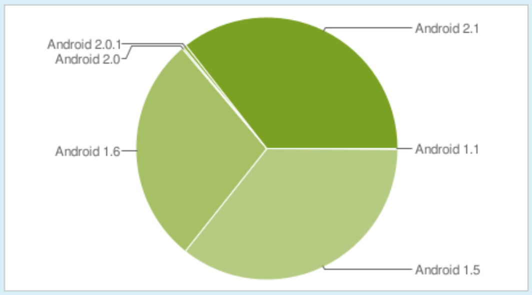
2005	Google buys Android, Inc. Work on Dalvik starts
2007	Open Handset Alliance announced Early Software Development Kit
2008	HTC G1 Announced SDK 1.0 Released
2009	G2 + 20 other phones released Cupcake, Donut, Éclair
2010	Zillion devices FroYo, Gingerbread, JIT



Versions

Version	API Level	Nickname
Android 1.0	1	
Android 1.1	2	
Android 1.5	3	
Android 1.6	4	
Android 2.0	5	
Android 2.01	6	
Android 2.1	7	
Android 2.2	8	

Version Distribution



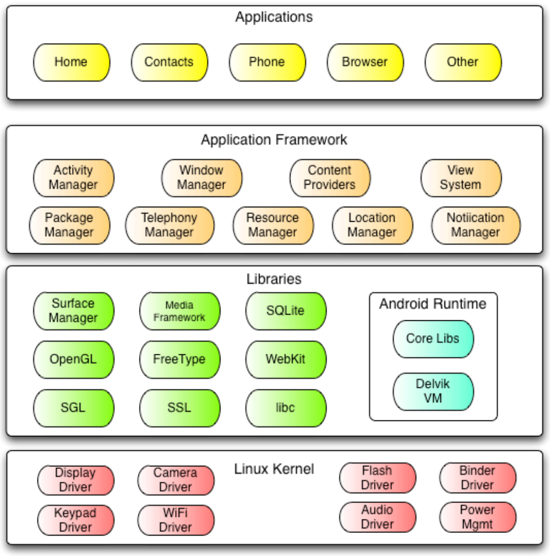
Android Platform	Percent of Devices
Android 1.1	0.1%
Android 1.5	34.1%
Android 1.6	28.0%
Android 2.0	0.2%
Android 2.0.1	0.4%
Android 2.1	37.2%

Data collected during two weeks ending on May 17, 2010

Source: Android.com

ANDROID STACK

The Stack



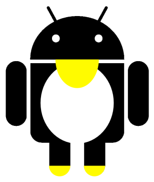
Linux Kernel

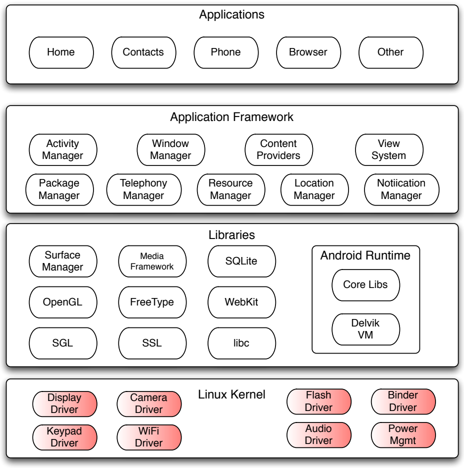
Android runs on Linux.

Linux provides as well as:
- Hardware abstraction layer
- Memory management
- Process management
- Networking

Users never see Linux sub system

The adb shell command opens Linux shell





Native Libraries

Bionic, a super fast and small license-friendly libc library optimized for embedded use

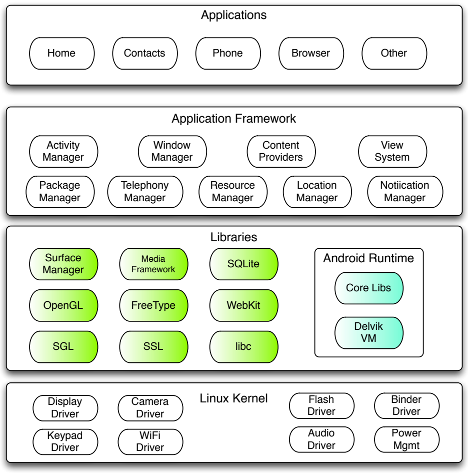
Surface Manager for composing window manager with off-screen buffering

2D and 3D graphics hardware support or software simulation

Media codecs offer support for major audio/video codecs

SQLite database

WebKit library for fast HTML rendering



Dalvik

Now with JIT

Dalvik VM is Google's implementation of Java VM

Optimized for mobile devices

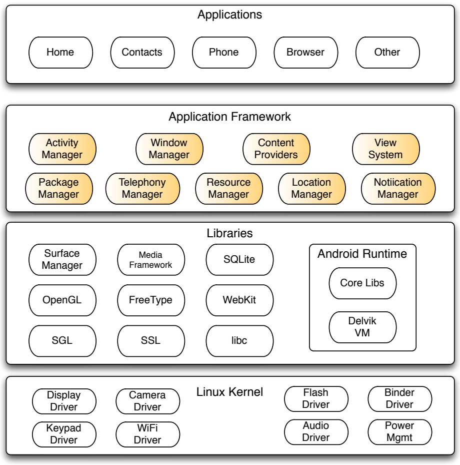
Key Dalvik differences:
- Register-based versus stack-based VM
- Dalvik runs .dex files
- More efficient and compact implementation
- Different set of Java libraries than SDK

Application Framework

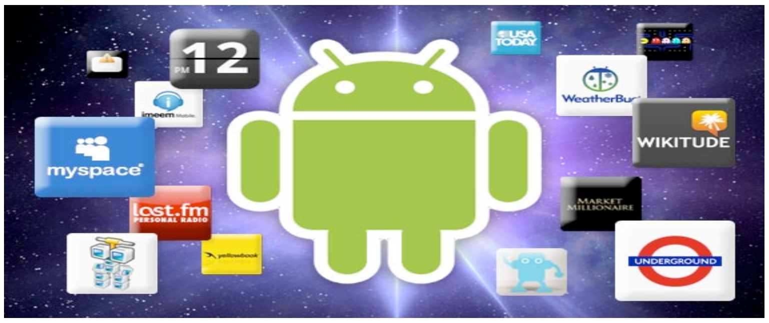
The rich set of system services wrapped in an intuitive Java API.

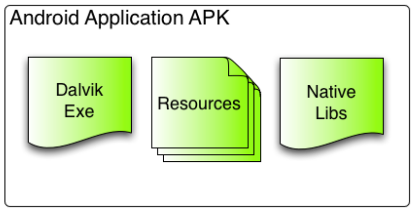
This ecosystem that developers can easily tap into is what makes writing apps for Android easy.

Location, web, telephony, WiFi, Bluetooth, notifications, media, camera, just to name a few.



Applications

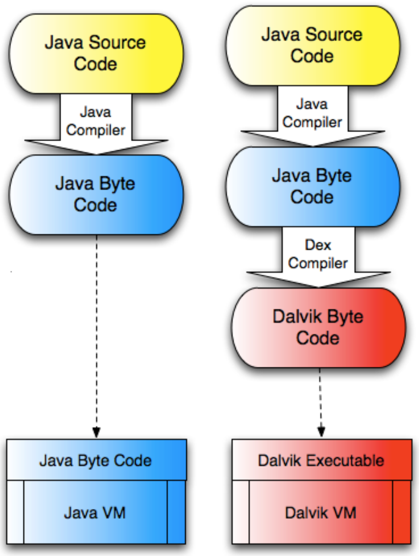




Dalvik Executable + Resources = APK
Must be signed (but debug key is okay for development)
Many markets with different policies

Android and Java

Android Java =
Java SE –
AWT/Swing +
Android API

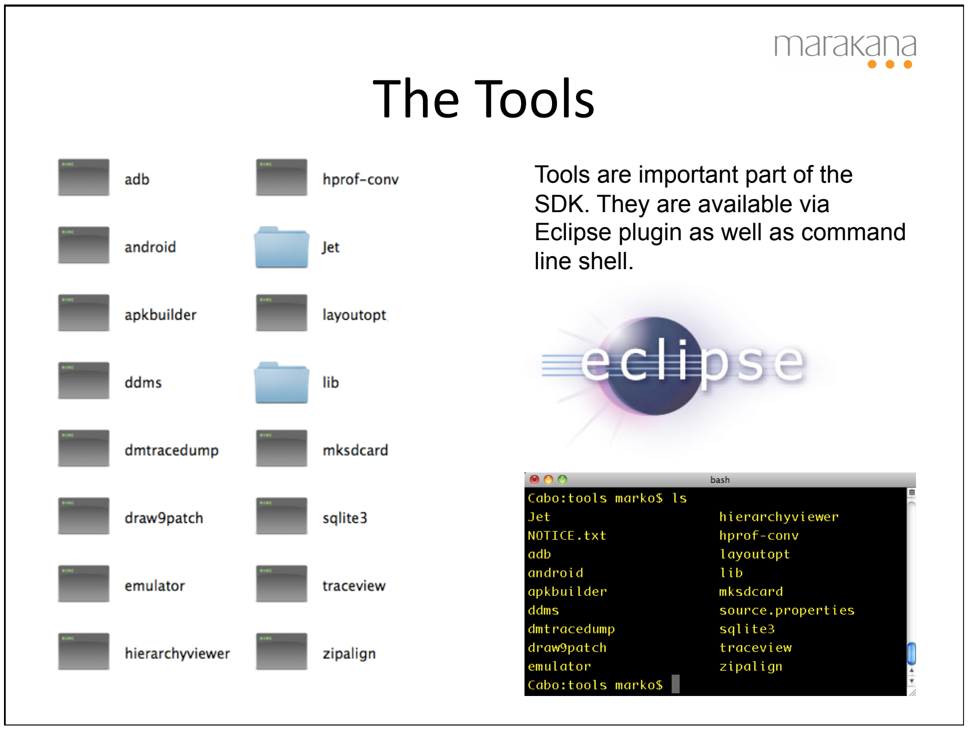


Android SDK - What's in the box

SDK

Tools
Docs
Platforms
- Data
- Skins
- Images
- Samples

Add-ons
- Google Maps

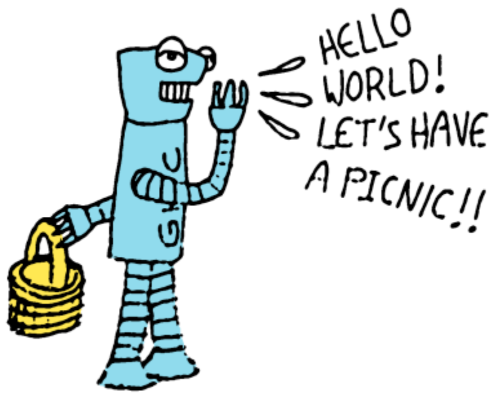


The Tools

adb	hprof-conv
android	Jet
apkbuilder	layoutopt
ddms	lib
dmtracedump	mksdcard
draw9patch	sqlite3
emulator	traceview
hierarchyviewer	zipalign

Tools are important part of the SDK. They are available via Eclipse plugin as well as command line shell.

```
Cabo:tools marko$ ls
Jet                     hierarchyviewer
NOTICE.txt              hprof-conv
adb                     layoutopt
android                 lib
apkbuilder              mksdcard
ddms                    source.properties
dmtracedump             sqlite3
draw9patch              traceview
emulator                zipalign
Cabo:tools marko$
```



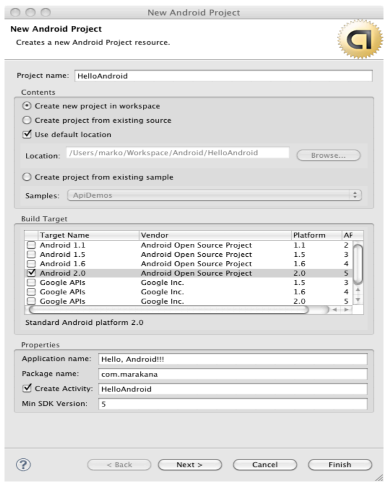
HELLO WORLD!

Create New Project

Use the Eclipse tool to create a new Android project.

Here are some key constructs:

Project	Eclipse construct
Target	minimum to run
App name	whatever
Package	Java package
Activity	Java class



New Android Project

Creates a new Android Project resource.

Project name: HelloAndroid

Contents

(●) Create new project in workspace
( ) Create project from existing source
[✓] Use default location

Location: /Users/marko/Workspace/Android/HelloAndroid

( ) Create project from existing sample

Samples: ApiDemos

Build Target

	Target Name	Vendor	Platform	AF
☐	Android 1.1	Android Open Source Project	1.1	2
☐	Android 1.5	Android Open Source Project	1.5	3
☐	Android 1.6	Android Open Source Project	1.6	4
✓	Android 2.0	Android Open Source Project	2.0	5
☐	Google APIs	Google Inc.	1.5	3
☐	Google APIs	Google Inc.	1.6	4
☐	Google APIs	Google Inc.	2.0	5

Standard Android platform 2.0

Properties

Application name: Hello, Android!!!

Package name: com.marakana

[✓] Create Activity: HelloAndroid

Min SDK Version: 5

The Manifest File

```
<?xml version="1.0" encoding="utf-8"?>
<manifest xmlns:android="http://schemas.android.com/apk/res/android"
      package="com.marakana"
      android:versionCode="1"
      android:versionName="1.0">
    <application android:icon="@drawable/icon"
            android:label="@string/app_name">
        <activity android:name=".HelloAndroid"
                android:label="@string/app_name">
            <intent-filter>
                <action android:name="android.intent.action.MAIN" />
                <category android:name="android.intent.category.LAUNCHER" />
            </intent-filter>
        </activity>
    </application>
    <uses-sdk android:minSdkVersion="5" />
</manifest>
```

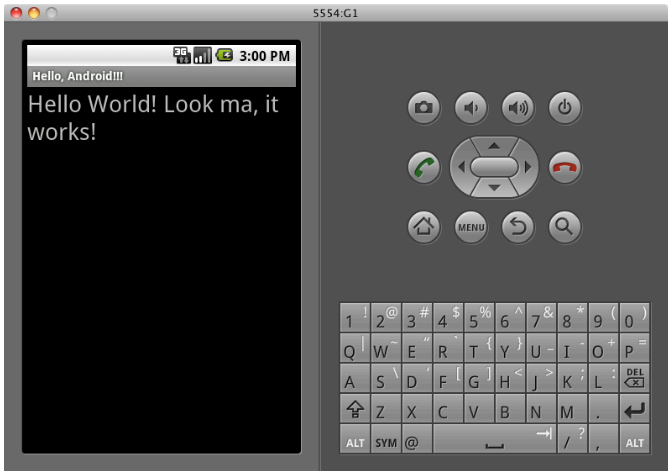
The Layout Resource

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

The Java File

```
package com.marakana;

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

public class HelloAndroid extends Activity {
    /** Called when the activity is first created. */
    @Override
    public void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.main);
    }
}
```

Running on Emulator

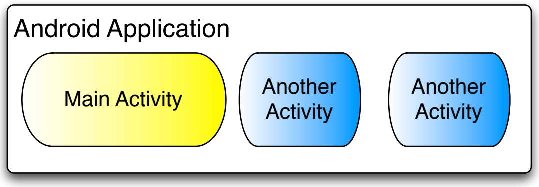


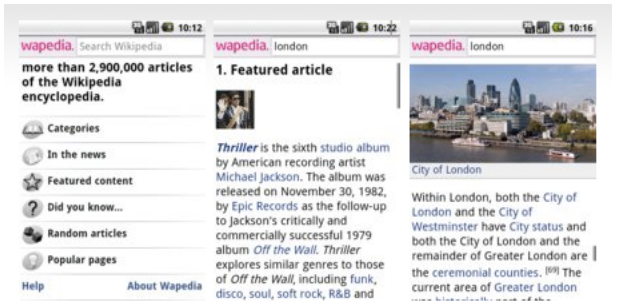
Hello World! Look ma, it works!

MAIN BUILDING BLOCKS

Activities

Activity is to an application what a web page is to a website. Sort of.





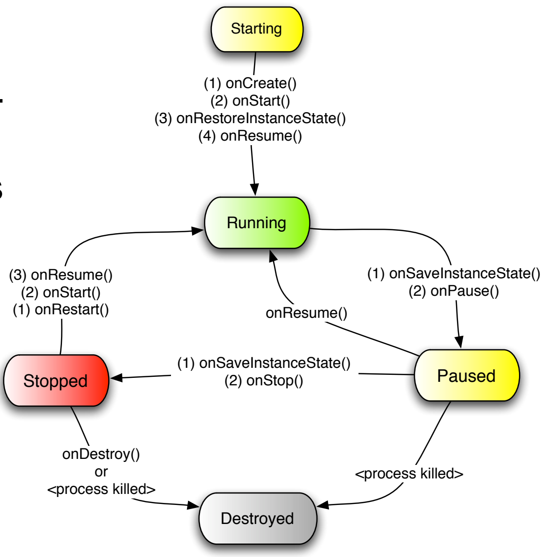
more than 2,900,000 articles of the Wikipedia encyclopedia.

Thriller is the sixth studio album by American recording artist Michael Jackson. The album was released on November 30, 1982, by Epic Records as the follow-up to Jackson's critically and commercially successful 1979 album *Off the Wall*. *Thriller* explores similar genres to those of *Off the Wall*, including funk, disco, soul, soft rock, R&B and

Within London, both the City of London and the City of Westminster have City status and both the City of London and the remainder of Greater London are the ceremonial counties. [69] The current area of Greater London

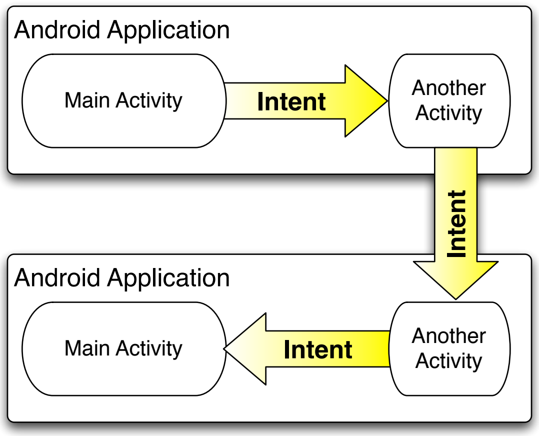
Activity Lifecycle

Activities have a well-defined lifecycle. The Android OS manages your activity by changing its state. You fill in the blanks.



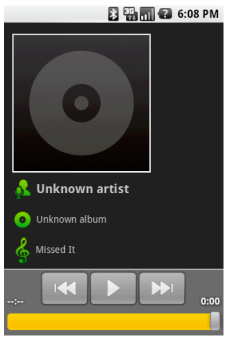
Intents

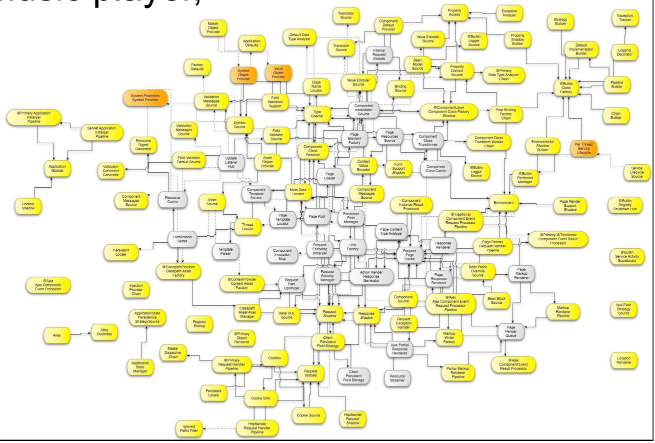
Intents are to Android apps what hyperlinks are to websites. They can be implicit and explicit. Sort of like absolute and relative links.

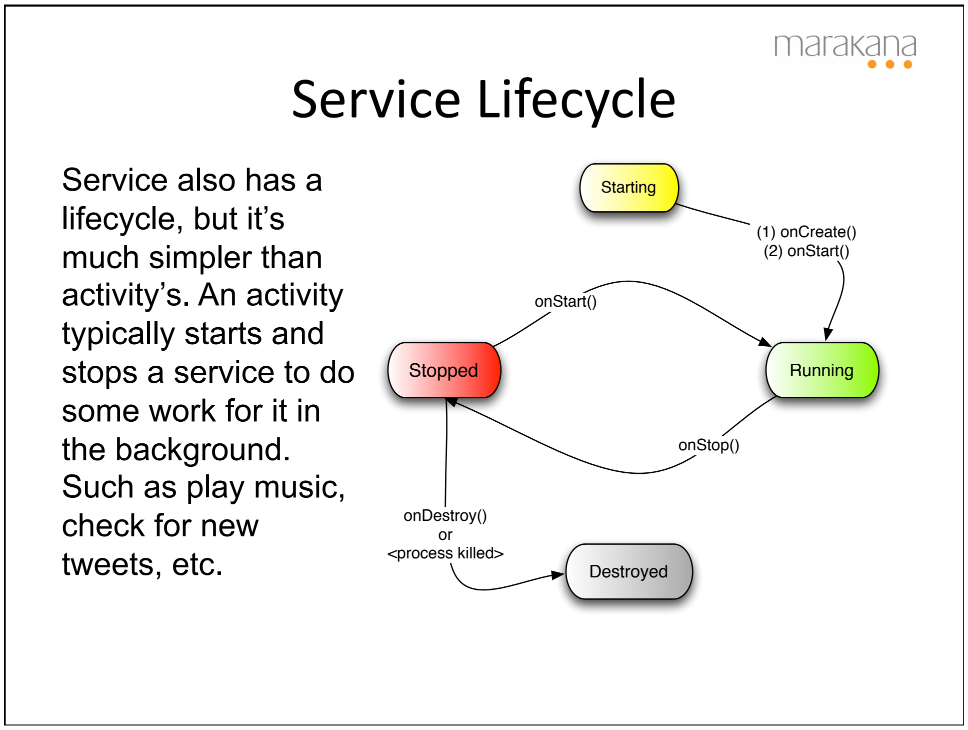


Services

A service is something that can be started and stopped. It doesn't have UI. It is typically managed by an activity. Music player, for example

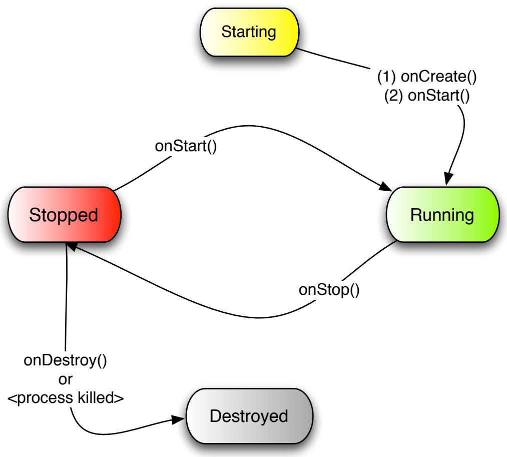


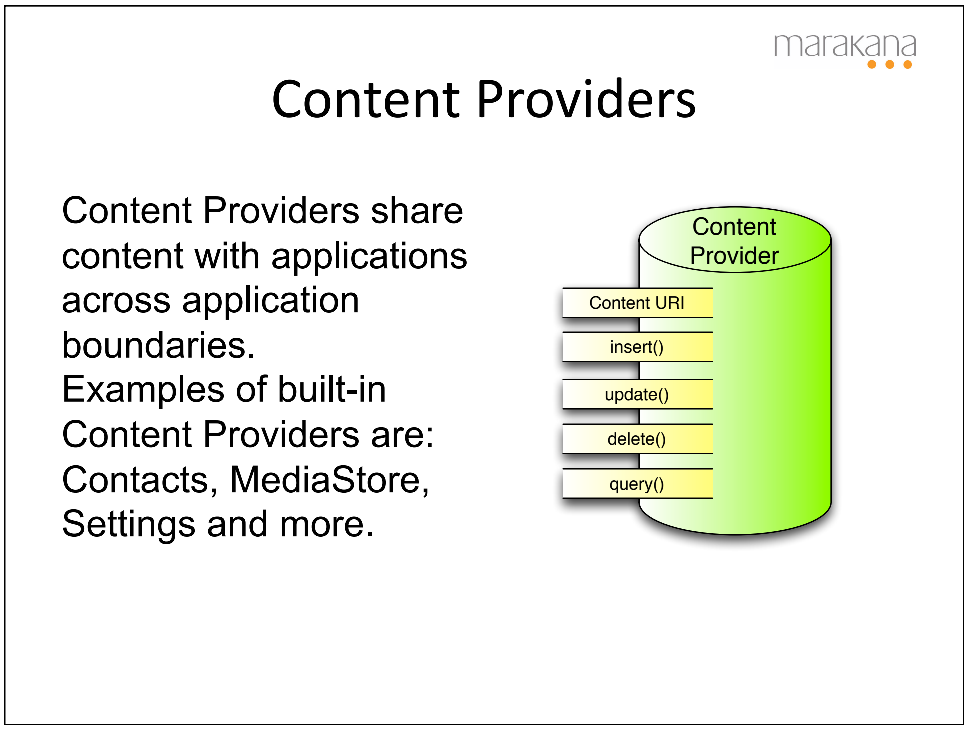




Service Lifecycle

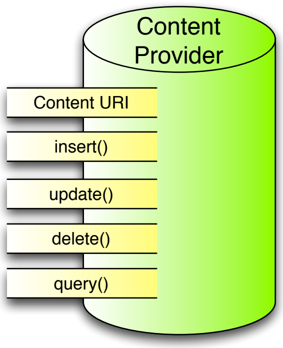
Service also has a lifecycle, but it’s much simpler than activity’s. An activity typically starts and stops a service to do some work for it in the background. Such as play music, check for new tweets, etc.

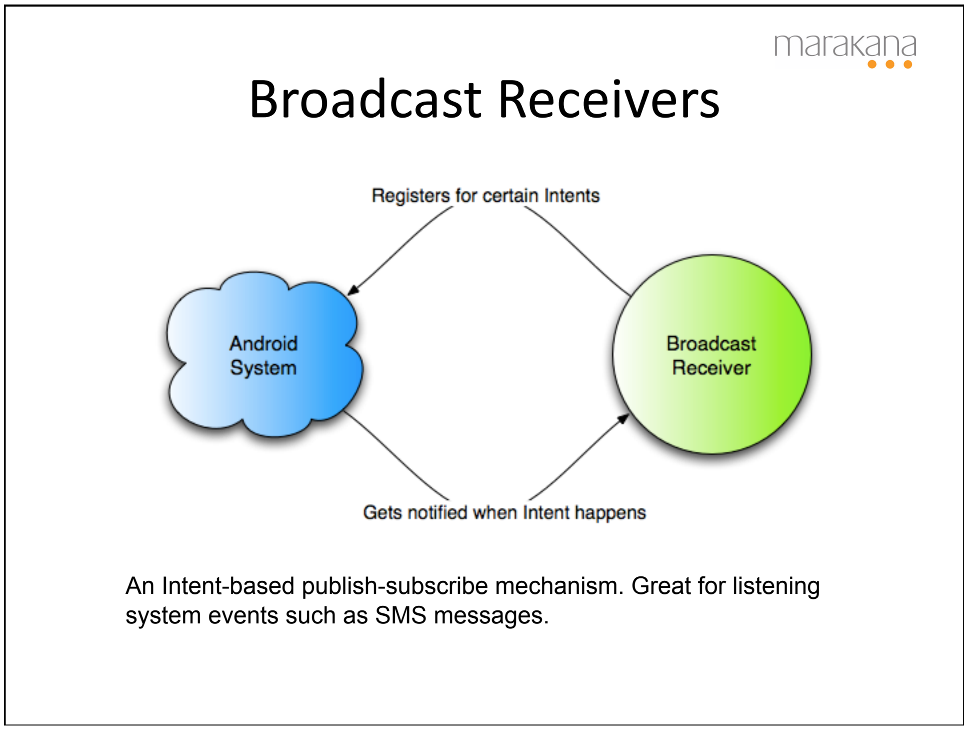




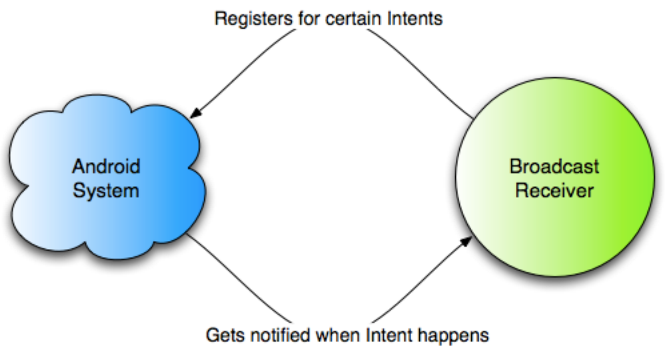
Content Providers

Content Providers share content with applications across application boundaries.
Examples of built-in Content Providers are: Contacts, MediaStore, Settings and more.

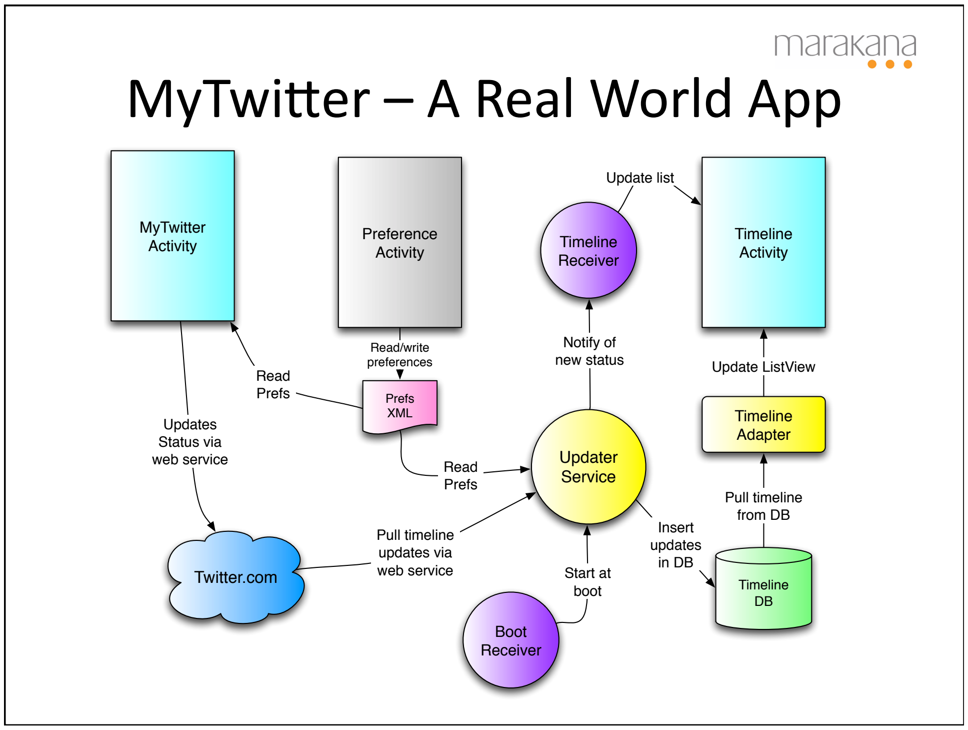




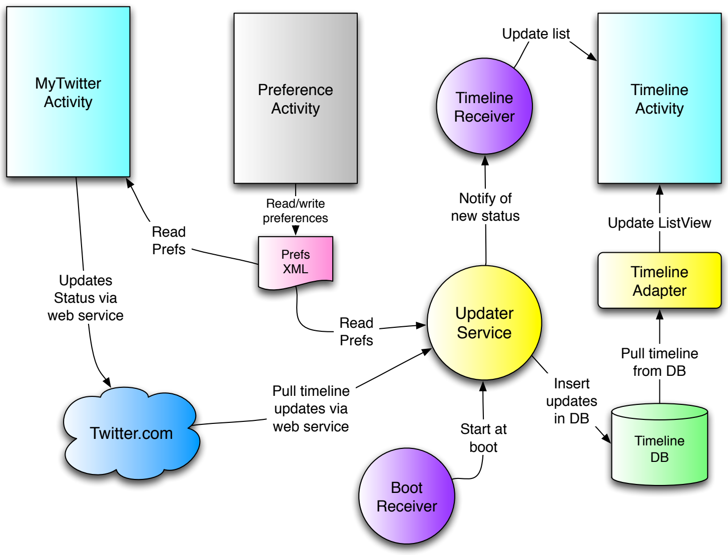
Broadcast Receivers

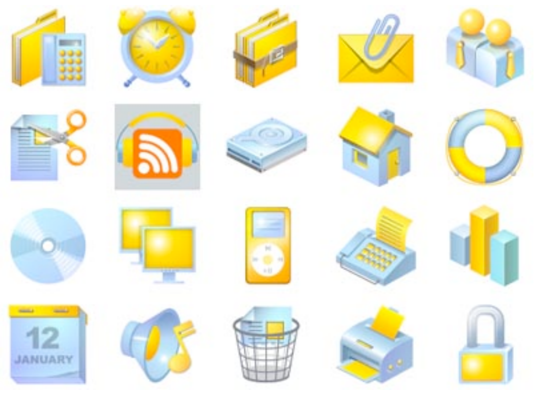

Registers for certain Intents
Android System
Broadcast Receiver
Gets notified when Intent happens

An Intent-based publish-subscribe mechanism. Great for listening system events such as SMS messages.



MyTwitter – A Real World App





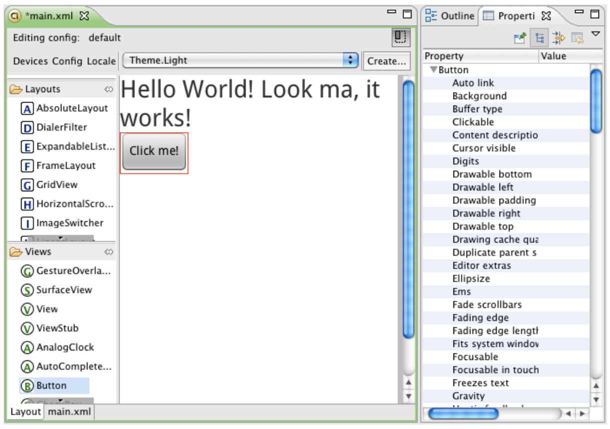
ANDROID USER INTERFACE

Two UI Approaches

Procedural	Declarative
You write Java code Similar to Swing or AWT	You write XML code Similar to HTML of a web page

You can mix and match both styles. Declarative is preferred: easier and more tools

XML-Based User Interface

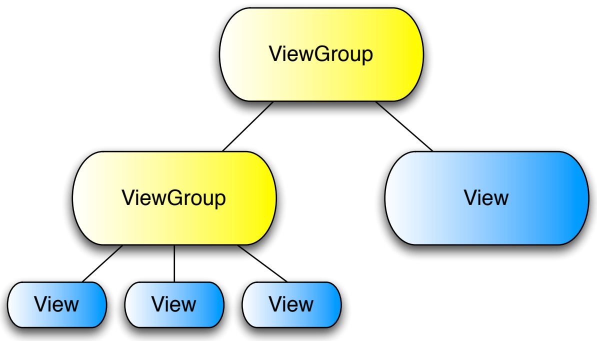


Use WYSIWYG tools to build powerful XML-based UI. Easily customize it from Java. Separate concerns.

Dips and Sps

px (pixel)	Dots on the screen
in (inches)	Size as measured by a ruler
mm (millimeters)	Size as measured by a ruler
pt (points)	1/72 of an inch
dp (density-independent pixel)	Abstract unit. On screen with 160dpi, 1dp=1px
dip	synonym for dp and often used by Google
sp	Similar to dp but also scaled by users font size preference

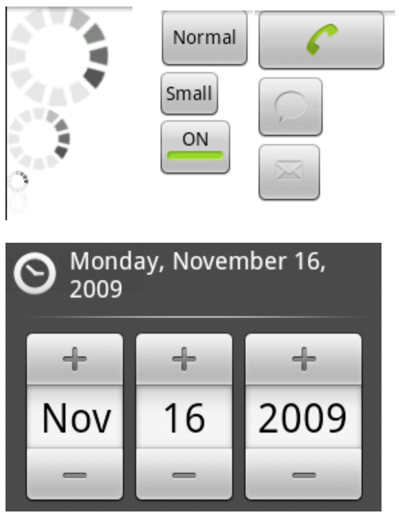
Views and Layouts



ViewGroups contain other Views but are also Views themselves.

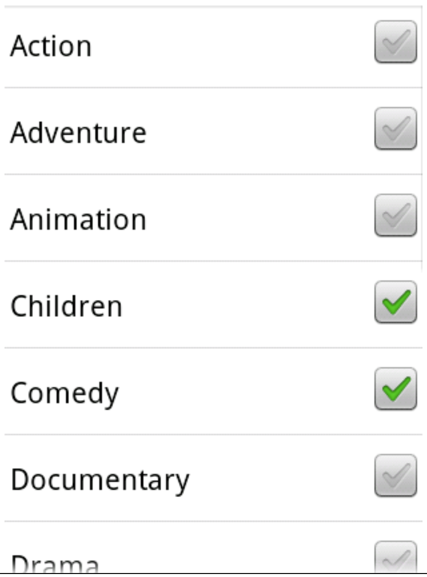
Common UI Components

Android UI includes many common modern UI widgets, such as Buttons, Tabs, Progress Bars, Date and Time Pickers, etc.

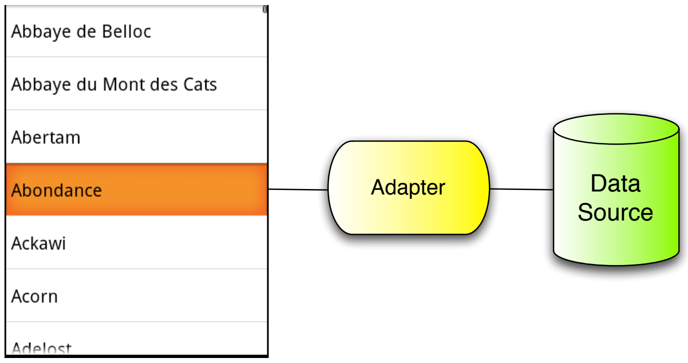


Selection Components

Some UI widgets may be linked to zillions of pieces of data. Examples are ListView and Spinners (pull-downs).



Adapters



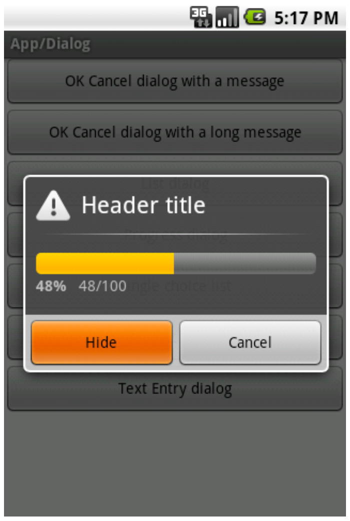
To make sure they run smoothly, Android uses Adapters to connect them to their data sources. A typical data source is an Array or a Database.

Complex Components

Certain high-level components are simply available just like Views. Adding a Map or a Video to your application is almost like adding a Button or a piece of text.

9:52 AM
Tutta Bella Neapolitan Pizza
9:53 AM
Sanjay Mavinkurve
Mountain View, CA
9:55 AM
Transfer to 49 - Mission-Van Ness
At: Van Ness Ave & Market St
Direction: North Point & Van Ness
Departs at 9:01pm

Menus and Dialogs



5:17 PM

App/Dialog

OK Cancel dialog with a message

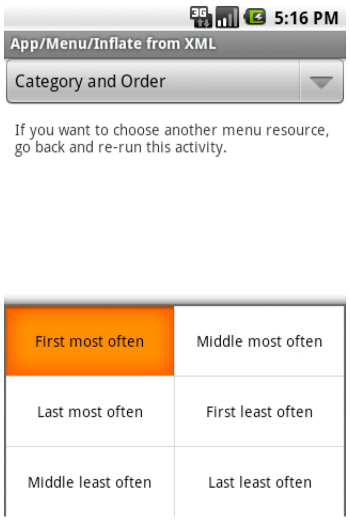
OK Cancel dialog with a long message

Header title

48% 48/100

Hide | Cancel

Text Entry dialog



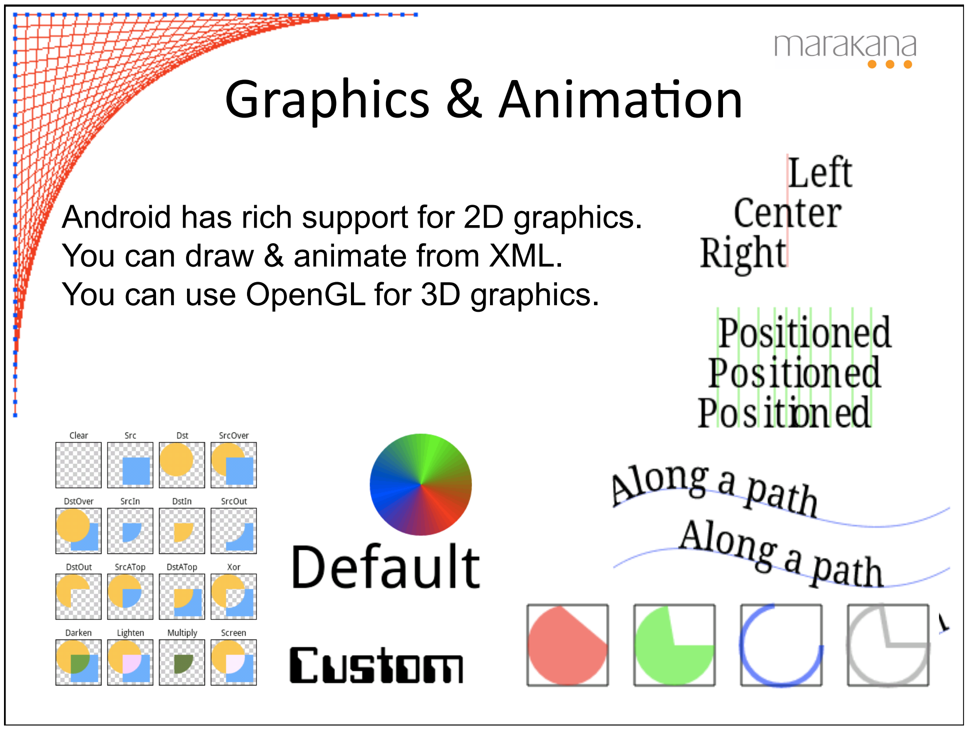
5:16 PM

App/Menu/Inflate from XML

Category and Order

If you want to choose another menu resource, go back and re-run this activity.

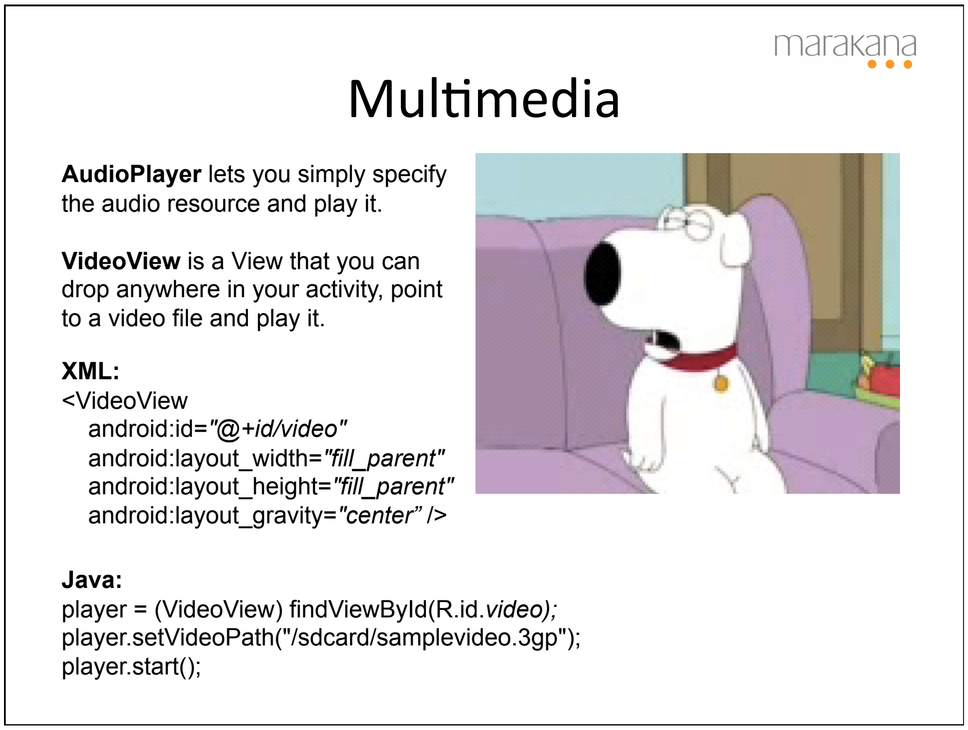
First most often	Middle most often
Last most often	First least often
Middle least often	Last least often



Graphics & Animation

Android has rich support for 2D graphics.
You can draw & animate from XML.
You can use OpenGL for 3D graphics.

marakana
Left
Center
Right
Positioned
Positioned
Positioned
Clear
Src
Dst
SrcOver
DstOver
SrcIn
DstIn
SrcOut
DstOut
SrcATop
DstATop
Xor
Darken
Lighten
Multiply
Screen
Default
Custom
Along a path
Along a path



Multimedia

AudioPlayer lets you simply specify the audio resource and play it.

VideoView is a View that you can drop anywhere in your activity, point to a video file and play it.

XML:

```
<VideoView
   android:id="@+id/video"
   android:layout_width="fill_parent"
   android:layout_height="fill_parent"
   android:layout_gravity="center" />
```

Java:

```
player = (VideoView) findViewById(R.id.video);
player.setVideoPath("/sdcard/samplevideo.3gp");
player.start();
```

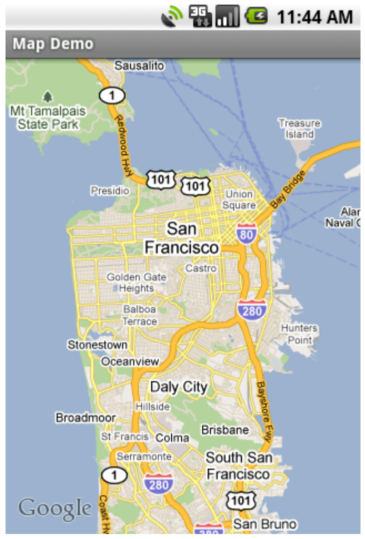
Google Maps

Google Maps is an add-on in Android.
It is not part of open-source project.

However, adding Maps is relatively easy using **MapView**.

XML:

```
<com.google.android.maps.MapView
android:id="@+id/map"
android:clickable="true"
android:layout_width="fill_parent"
android:layout_height="fill_parent"
android:apiKey="0EfLSgdSCWIN…A"
/>
```



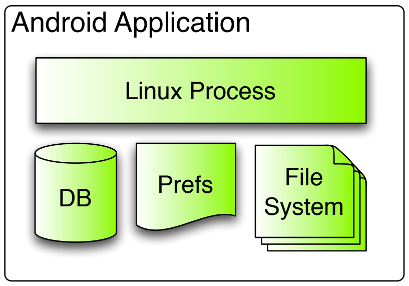
OPERATING SYSTEM FEATURES

Security

Each Android application runs inside its own Linux process.

Additionally, each application has its own sandbox file system with its own set of preferences and its own database.

Other applications cannot access any of its data, unless it is explicitly shared.

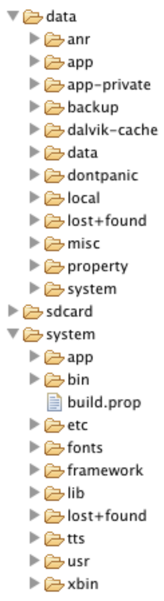


File System

The file system has three main mount points. One for system, one for the apps, and one for whatever.

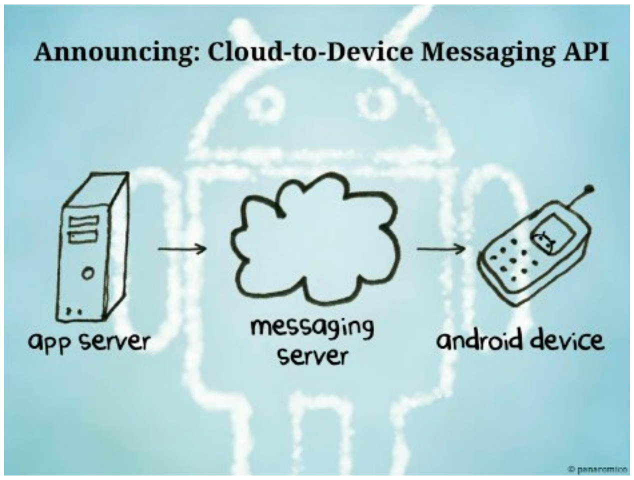
Each app has its own sandbox easily accessible to it. No one else can access its data. The sandbox is in /data/data/com.marakana/

SDCard is expected to always be there. It's a good place for large files, such as movies and music. Everyone can access it.



```
▼ data
    ▶ anr
    ▶ app
    ▶ app-private
    ▶ backup
    ▶ dalvik-cache
    ▶ data
    ▶ dontpanic
    ▶ local
    ▶ lost+found
    ▶ misc
    ▶ property
    ▶ system
▶ sdcard
▼ system
    ▶ app
    ▶ bin
      build.prop
    ▶ etc
    ▶ fonts
    ▶ framework
    ▶ lib
    ▶ lost+found
    ▶ tts
    ▶ usr
    ▶ xbin
```

Cloud to Device Push

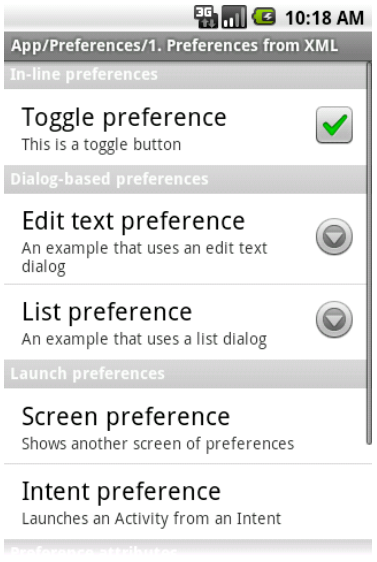


Big deal for many pull-based apps. Will make devices use less battery.

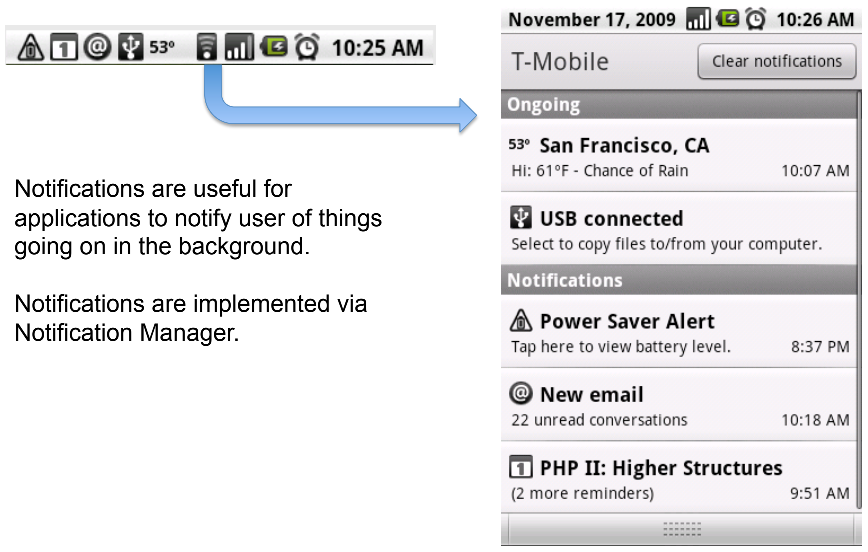
Preferences

Your app can support complex preferences quite easily.

You define your preferences in an XML file and the corresponding UI and data storage is done for free.


10:18 AM
App/Preferences/1. Preferences from XML
In-line preferences
Toggle preference
This is a toggle button ✓
Dialog-based preferences
Edit text preference
An example that uses an edit text dialog
List preference
An example that uses a list dialog
Launch preferences
Screen preference
Shows another screen of preferences
Intent preference
Launches an Activity from an Intent

Notifications



53° San Francisco, CA
Hi: 61°F - Chance of Rain 10:07 AM

USB connected
Select to copy files to/from your computer.

Power Saver Alert
Tap here to view battery level. 8:37 PM

New email
22 unread conversations 10:18 AM

PHP II: Higher Structures
(2 more reminders) 9:51 AM

Notifications are useful for applications to notify user of things going on in the background.

Notifications are implemented via Notification Manager.

SQLite Database

Android ships with SQLite3

SQLite is

Zero configuration
Serverless
Single database file
Cross-Platform
Compact
Public Domain

Database engine.

May you do good and not evil
May you find forgiveness for yourself and forgive others
May you share freely, never taking more than you give.

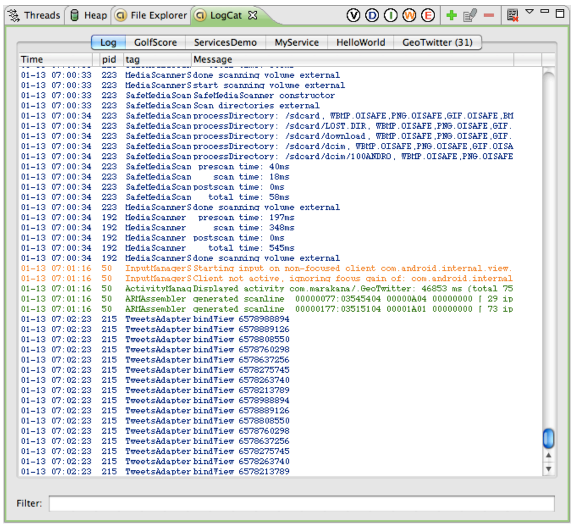
DEBUGGING ANDROID APPS

LogCat

The universal, most versatile way to track what is going on in your app.

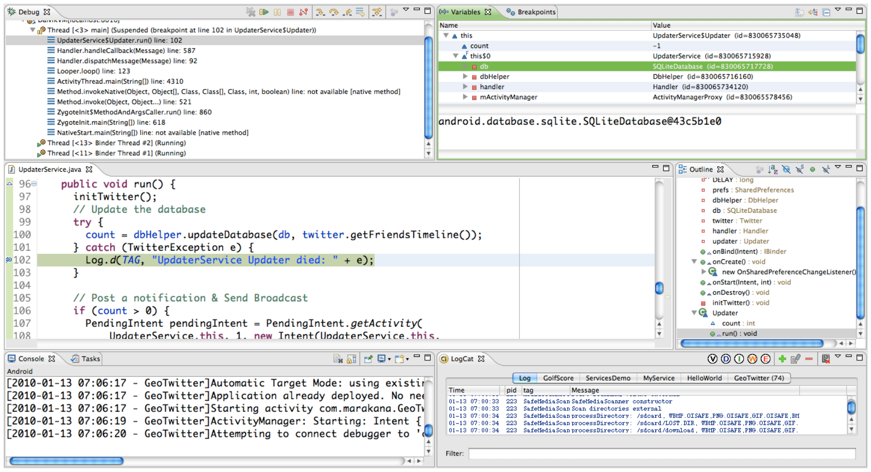
Can be viewed via command line or Eclipse.

Logs can be generated both from SDK Java code, or low-level C code via Bionic libc extension.



```
Time         pid  tag              Message
01-13 07:00:33 223 MediaScannerS done scanning volume external
01-13 07:00:33 223 MediaScannerS start scanning volume external
01-13 07:00:33 223 SafeMediaScan SafeMediaScanner constructor
01-13 07:00:33 223 SafeMediaScan Scan directories external
01-13 07:00:34 223 SafeMediaScan processDirectory: /sdcard, WBMP.OISAFE,PNG.OISAFE,GIF.OISAFE,BM
01-13 07:00:34 223 SafeMediaScan processDirectory: /sdcard/LOST.DIR, WBMP.OISAFE,PNG.OISAFE,GIF.
01-13 07:00:34 223 SafeMediaScan processDirectory: /sdcard/download, WBMP.OISAFE,PNG.OISAFE,GIF.
01-13 07:00:34 223 SafeMediaScan processDirectory: /sdcard/dcim, WBMP.OISAFE,PNG.OISAFE,GIF.OISA
01-13 07:00:34 223 SafeMediaScan processDirectory: /sdcard/dcim/100ANDRO, WBMP.OISAFE,PNG.OISAFE
01-13 07:00:34 223 SafeMediaScan  prescan time: 40ms
01-13 07:00:34 223 SafeMediaScan     scan time: 18ms
01-13 07:00:34 223 SafeMediaScan postscan time: 0ms
01-13 07:00:34 223 SafeMediaScan    total time: 58ms
01-13 07:00:34 223 MediaScannerS done scanning volume external
01-13 07:00:34 192 MediaScanner  prescan time: 197ms
01-13 07:00:34 192 MediaScanner     scan time: 348ms
01-13 07:00:34 192 MediaScanner postscan time: 0ms
01-13 07:00:34 192 MediaScanner    total time: 545ms
01-13 07:00:34 192 MediaScannerS done scanning volume external
01-13 07:01:16 50  InputManagerS Starting input on non-focused client com.android.internal.view.
01-13 07:01:16 50  InputManagerS Client not active, ignoring focus gain of: com.android.internal
01-13 07:01:16 50  ActivityManag Displayed activity com.marakana/.GeoTwitter: 46853 ms (total 75
01-13 07:01:16 50  ARMAssembler generated scanline  00000077:03545404 00000A04 00000000 [ 29 ip
01-13 07:01:16 50  ARMAssembler generated scanline  00000177:03515104 00001A01 00000000 [ 73 ip
01-13 07:02:23 215 TweetsAdapter bindView 6578988894
01-13 07:02:23 215 TweetsAdapter bindView 6578889126
01-13 07:02:23 215 TweetsAdapter bindView 6578808550
01-13 07:02:23 215 TweetsAdapter bindView 6578760298
01-13 07:02:23 215 TweetsAdapter bindView 6578637256
01-13 07:02:23 215 TweetsAdapter bindView 6578275745
01-13 07:02:23 215 TweetsAdapter bindView 6578263740
01-13 07:02:23 215 TweetsAdapter bindView 6578213789
01-13 07:02:23 215 TweetsAdapter bindView 6578988894
01-13 07:02:23 215 TweetsAdapter bindView 6578889126
01-13 07:02:23 215 TweetsAdapter bindView 6578808550
01-13 07:02:23 215 TweetsAdapter bindView 6578760298
01-13 07:02:23 215 TweetsAdapter bindView 6578637256
01-13 07:02:23 215 TweetsAdapter bindView 6578275745
01-13 07:02:23 215 TweetsAdapter bindView 6578263740
01-13 07:02:23 215 TweetsAdapter bindView 6578213789
```

Debugger



Debug

```
Thread [<3> main] (Suspended (breakpoint at line 102 in UpdaterService$Updater))
    UpdaterService$Updater.run() line: 102
    Handler.handleCallback(Message) line: 587
    Handler.dispatchMessage(Message) line: 92
    Looper.loop() line: 123
    ActivityThread.main(String[]) line: 4310
    Method.invokeNative(Object, Object[], Class, Class[], Class, int, boolean) line: not available [native method]
    Method.invoke(Object, Object...) line: 521
    ZygoteInit$MethodAndArgsCaller.run() line: 860
    ZygoteInit.main(String[]) line: 618
    NativeStart.main(String[]) line: not available [native method]
Thread [<13> Binder Thread #2] (Running)
Thread [<11> Binder Thread #1] (Running)
```

Variables

Name	Value
this	UpdaterService$Updater (id=830065735048)
count	-1
this$0	UpdaterService (id=830065715928)
db	SQLiteDatabase (id=830065717728)
dbHelper	DbHelper (id=830065716160)
handler	Handler (id=830065734120)
mActivityManager	ActivityManagerProxy (id=830065578456)

```
android.database.sqlite.SQLiteDatabase@43c5b1e0
```

UpdaterService.java

```java
    public void run() {
      initTwitter();
      // Update the database
      try {
        count = dbHelper.updateDatabase(db, twitter.getFriendsTimeline());
      } catch (TwitterException e) {
        Log.d(TAG, "UpdaterService Updater died: " + e);
      }

      // Post a notification & Send Broadcast
      if (count > 0) {
        PendingIntent pendingIntent = PendingIntent.getActivity(
            UpdaterService.this, 1, new Intent(UpdaterService.this,
```

Outline

```
DELAY : long
prefs : SharedPreferences
dbHelper : DbHelper
db : SQLiteDatabase
twitter : Twitter
handler : Handler
updater : Updater
onBind(Intent) : IBinder
onCreate() : void
  new OnSharedPreferenceChangeListener()
onStart(Intent, int) : void
onDestroy() : void
initTwitter() : void
Updater
  count : int
  run() : void
```

Console

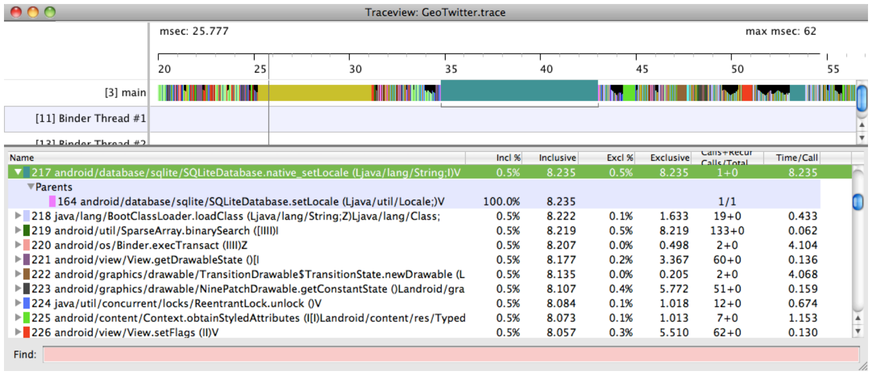
```
Android
[2010-01-13 07:06:17 - GeoTwitter]Automatic Target Mode: using existir
[2010-01-13 07:06:17 - GeoTwitter]Application already deployed. No nee
[2010-01-13 07:06:17 - GeoTwitter]Starting activity com.marakana.GeoTw
[2010-01-13 07:06:19 - GeoTwitter]ActivityManager: Starting: Intent {
[2010-01-13 07:06:20 - GeoTwitter]Attempting to connect debugger to '
```

LogCat

```
Time            pid  tag            Message
01-13 07:00:33  223  SafeMediaScan  SafeMediaScanner constructor
01-13 07:00:33  223  SafeMediaScan  Scan directories external
01-13 07:00:34  223  SafeMediaScan  processDirectory: /sdcard, WBMP.OISAFE,PNG.OISAFE,GIF.OISAFE,BM
01-13 07:00:34  223  SafeMediaScan  processDirectory: /sdcard/LOST.DIR, WBMP.OISAFE,PNG.OISAFE,GIF.
01-13 07:00:34  223  SafeMediaScan  processDirectory: /sdcard/download, WBMP.OISAFE,PNG.OISAFE,GIF.
```

Your standard debugger is included in SDK, with all the usual bells & whistles.

TraceView



Traceview: GeoTwitter.trace

msec: 25.777 max msec: 62

Name	Incl %	Inclusive	Excl %	Exclusive	Calls+Recur Calls/Total	Time/Call
217 android/database/sqlite/SQLiteDatabase.native_setLocale (Ljava/lang/String;I)V	0.5%	8.235	0.5%	8.235	1+0	8.235
Parents						
164 android/database/sqlite/SQLiteDatabase.setLocale (Ljava/util/Locale;)V	100.0%	8.235			1/1	
218 java/lang/BootClassLoader.loadClass (Ljava/lang/String;Z)Ljava/lang/Class;	0.5%	8.222	0.1%	1.633	19+0	0.433
219 android/util/SparseArray.binarySearch ([IIII)I	0.5%	8.219	0.5%	8.219	133+0	0.062
220 android/os/Binder.execTransact (IIII)Z	0.5%	8.207	0.0%	0.498	2+0	4.104
221 android/view/View.getDrawableState ()[I	0.5%	8.177	0.2%	3.367	60+0	0.136
222 android/graphics/drawable/TransitionDrawable$TransitionState.newDrawable (L	0.5%	8.135	0.0%	0.205	2+0	4.068
223 android/graphics/drawable/NinePatchDrawable.getConstantState ()Landroid/gra	0.5%	8.107	0.4%	5.772	51+0	0.159
224 java/util/concurrent/locks/ReentrantLock.unlock ()V	0.5%	8.084	0.1%	1.018	12+0	0.674
225 android/content/Context.obtainStyledAttributes (I[I)Landroid/content/res/Typed	0.5%	8.073	0.1%	1.013	7+0	1.153
226 android/view/View.setFlags (II)V	0.5%	8.057	0.3%	5.510	62+0	0.130

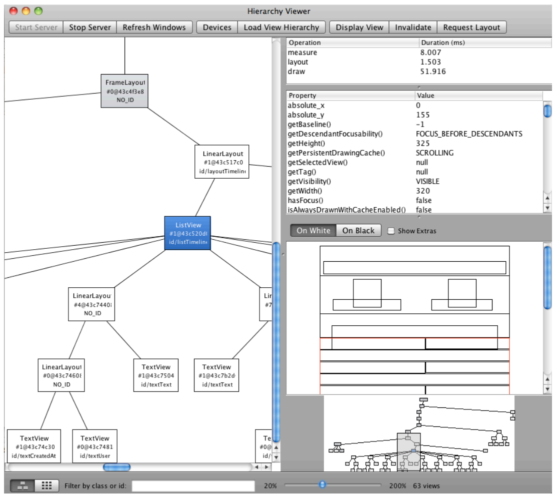
Find:

TraceView helps you profile you application and find bottlenecks. It shows execution of various calls through the entire stack. You can zoom into specific calls.

Hierarchy Viewer

Hierarchy Viewer helps you analyze your User Interface.

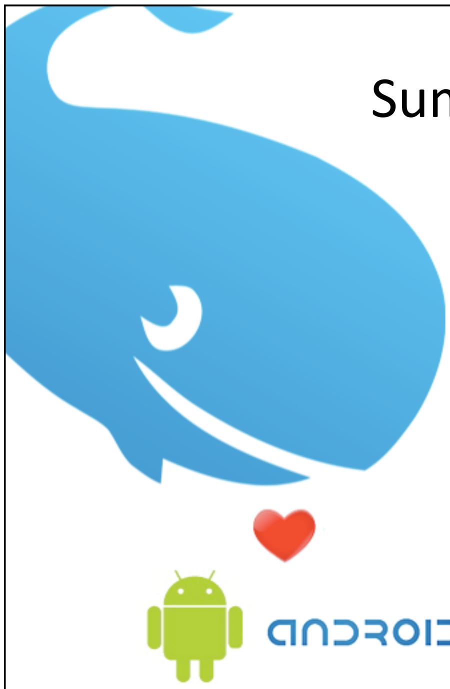
Base UI tends to be the most “expensive” part of your application, this tool is very useful.



Operation	Duration (ms)
measure	8.007
layout	1.503
draw	51.916

Property	Value
absolute_x	0
absolute_y	155
getBaseline()	-1
getDescendantFocusability()	FOCUS_BEFORE_DESCENDANTS
getHeight()	325
getPersistentDrawingCache()	SCROLLING
getSelectedView()	null
getTag()	null
getVisibility()	VISIBLE
getWidth()	320
hasFocus()	false
isAlwaysDrawnWithCacheEnabled()	false

Summary



Android is open and complete system for mobile development. It is based on Java and augmented with XML.

Android is being adopted very quickly both by users, carriers, and manufacturers.

It takes about 3-5 days of intensive training to learn Android application development for someone who has basic Java (or similar) experience.

Marko Gargenta, Marakana.com
marko@marakana.com
+1-415-647-7000