Chapter 13

An Android Social Media App

In This Chapter

- Posting on Twitter with Android code
- Tweeting with your app on a user's behalf
- Using Java exceptions to get out of a jam

A reader from Vancouver (in British Columbia, Canada) writes:

> "Hello, Barry. I just thought I would ask that you include the area that seems to get attention from app developers: programs connecting with social sites. I look forward to reading the new book! Best regards, David."

Well, David, you've inspired me to create a Twitter app. This chapter's example does two things: Post a new tweet, and get a twitter user's timeline. The app can perform many more Twitter tasks — for example, search for tweets, look for users, view trends, check friends and followers, gather suggestions, and do lots of other things that Twitter users want done. For simplicity, though, I have the app perform only two tasks: tweet and display a timeline.

I can summarize the essence of this chapter's Twitter code in two short statements. To post a tweet, the app executes

```
twitter.updateStatus("This is my tweet.");
```

And, to display a user's timeline, the app executes

```
List<twitter4j.Status> statuses =
    twitter.getUserTimeline("allmycode");
```

Of course, these two statements only serve as a summary, and a summary is never the same as the material it summarizes. Imagine standing on the street in Times Square and shouting the statement "Twitter dot update status: 'This is my tweet.'" Nothing good happens because you're issuing the correct command in the wrong context. In the same way, the context surrounding a call to `twitter.updateStatus` in an app matters an awful lot.

This chapter covers all the context surrounding your calls to `twitter.updateStatus` and `twitter.getUserTimeline`. In the process, you can read about Java's exceptions — a vital feature that's available to all Java programmers.

The Twitter App's Files

You can import this chapter's code from my website (http://allmycode.com/Java4Android) by following the instructions in Chapter 2. As is true for any Android app, this chapter's Eclipse project contains about 40 files in about 30 different folders. In this chapter, I concentrate on the project's `MainActivity.java` file. But a few other files require some attention.

The Twitter4J API jar file

Android has no built-in support for communicating with Twitter. Yes, the raw materials are contained in Android's libraries, but to deal with all of Twitter's requirements, someone has to paste together those raw materials in a useful way. Fortunately, several developers have done all the pasting and made their libraries available for use by others. The library that I use in this chapter is Twitter4J. Its website is `http://twitter4j.org`.

Chapter 4 describes the role of `.jar` files in Java program development. For this chapter's example to work, your project must include a `.jar` file containing the Twitter4J libraries. If you've successfully imported this book's code into Eclipse, the 13-01 project contains the necessary `.jar` file.

If you're creating this chapter's example on your own, or if you're having trouble with the project's existing `.jar` files, you can add the Twitter4J libraries by following these steps:

1. **Visit `http://twitter4j.org`.**
2. **Find the link to download the latest stable version of Twitter4J.**

   To run this chapter's example, I use Twitter4J version 3.0.3. If you download a later version, it'll probably work. But I make no promises about the backward compatibility, forward compatibility, or sideward compatibility of the various Twitter4J versions. If my example doesn't run properly for you, you can search the Twitter4J site for a download link to version 3.0.3.
3. **Click the link to download the Twitter4J software.**

   The file that I downloaded is `twitter4j-3.0.3.zip`.

4. **Look for a `twitter4j-core.jar` file inside the downloaded `.zip` file.**

   In the `.zip` file that I downloaded, I found a file named `twitter4j-core-3.0.3.jar`.

5. **Extract the `twitter4j-core.jar` file to a place on your computer's hard drive.**

   Any location on your hard drive is okay, as long as you remember where you put the `twitter4j-core.jar` file.

6. **In Eclipse's Package Explorer, right-click (or Control-click on a Mac) this chapter's project.**

7. **In the resulting context menu, select Properties.**

   Eclipse's Properties dialog box appears.

8. **On the left side of the Properties dialog box, select Java Build Path.**

9. **In the middle of the Properties dialog box, select the Libraries tab.**

10. **On the right side of the Properties dialog box, click the Add External JARs button.**

    Eclipse displays the JAR Selection dialog box.

11. **In the JAR Selection dialog box, navigate to the directory containing your `twitter4j-core.jar` file.**

    What I refer to as your `twitter4j-core.jar` file is probably named `twitter4j-core-3.0.3.jar` or similar.

12. **Select the `twitter4j-core.jar` file and close the JAR Selection dialog box.**

    Doing so adds your `twitter4j-core.jar` file to the list of items on the Libraries tab.

13. **In the middle of the Properties dialog box, switch from the Libraries tab to the Order and Export tab.**

    The Order and Export tab contains a list of items, one of which is your `twitter4j-core.jar` file.

14. **Make sure that the check box next to your `twitter4j-core.jar` file is selected.**

    Doing so ensures that this `.jar` file is uploaded to whatever device you use to test the application. Without the check mark indicating the selection, Eclipse compiles the code using the `.jar` file but doesn't bother sending the `.jar` file to an emulator or to your phone. It's quite frustrating.

15. **Select your** `twitter4j-core.jar` **item on the Order and Export tab. Then, on the right side of the Properties dialog box, click the Up button a few times.**

    Keep clicking the Up button until your `twitter4j-core.jar` file is at the top of the Order and Export tab's list.

16. **Click OK to dismiss the Properties dialog box.**

    If you look at Eclipse's Package Explorer, your project now has a Referenced Libraries branch. When you expand the Referenced Libraries branch, you see a branch for your `twitter4j-core.jar` file.

The manifest file

Every Android app has an `AndroidManifest.xml` file. Listing 13-1 contains the `AndroidManifest.xml` file for this chapter's Twitter app.

Listing 13-1: The AndroidManifest.xml File

```
<?xml version="1.0" encoding="utf-8"?>
<manifest xmlns:android=
    "http://schemas.android.com/apk/res/android"
  package="com.allmycode.twitter"
  android:versionCode="1"
  android:versionName="1.0" >

  <uses-sdk
    android:minSdkVersion="8"
    android:targetSdkVersion="17" />
  <uses-permission android:name=
      „android.permission.INTERNET"/>

  <application
    android:allowBackup="true"
    android:icon="@drawable/ic_launcher"
    android:label="@string/app_name"
    android:theme="@style/AppTheme" >
    <activity
      android:name="com.allmycode.twitter.MainActivity"
      android:label="@string/app_name"
      android:windowSoftInputMode="adjustPan" >
      <intent-filter>
        <action android:name=
            "android.intent.action.MAIN" />
        <category android:name=
            "android.intent.category.LAUNCHER" />
      </intent-filter>
```

```
    </activity>
  </application>

</manifest>
```

When you use Eclipse to create a new Android application project, Eclipse writes most of the code in Listing 13-1 automatically. For this chapter's project, I have to add two additional snippets of code:

- **The `windowSoftInputMode` attribute tells Android what to do when the user activates the onscreen keyboard.**

  The `adjustPan` value tells Android not to squash together all my screen's widgets. (Take my word for it: The app looks ugly without this `adjustPan` value.)

- **The `uses-permission` element warns Android that my app requires Internet connectivity.**

  When a user installs an app that uses the `android.permission.INTERNET` permission, Android warns the user that the app requires full network access. Yes, a large percentage of users ignore this kind of warning. But the app can't access Twitter without the permission. If you forget to add this `uses-permission` element (as I often do), the app doesn't obey any of your Twitter commands. And when your app fails to contact the Twitter servers, Android often displays only cryptic, unhelpful error messages.

The error messages from an unsuccessful run of your Android app range from extremely helpful to extremely unhelpful. One way or another, it never hurts to read these messages. You can find most of the messages in Eclipse's Console view or in Eclipse's LogCat view.

For more information about `AndroidManifest.xml` files, see Chapter 4.

The main activity's layout file

Chapter 4 introduces the use of a layout file to describe the look of an activity on the screen. The layout file for this chapter's example has no extraordinary qualities. I include it in Listing 13-2 for completeness. As usual, you can import this chapter's code from my website (`http://allmycode.com/Java4Android`). But if you're living large and creating the app on your own from scratch, you can copy the contents of Listing 13-2 to the project's `res/layout/activity_main.xml` file. Alternatively, you can use Eclipse's toolset to drag and drop, point and click, or type and tap your way to the graphical layout shown in Figure 13-1.

Listing 13-2: The Layout File

```
<RelativeLayout xmlns:android=
    "http://schemas.android.com/apk/res/android"
  xmlns:tools="http://schemas.android.com/tools"
  android:layout_width="match_parent"
  android:layout_height="match_parent"
  android:paddingBottom="@dimen/activity_vertical_margin"
  android:paddingLeft="@dimen/activity_horizontal_margin"
  android:paddingRight=
      "@dimen/activity_horizontal_margin"
  android:paddingTop="@dimen/activity_vertical_margin"
  tools:context=".MainActivity" >

  <TextView
    android:id="@+id/textView2"
    android:layout_width="wrap_content"
    android:layout_height="wrap_content"
    android:layout_alignBaseline="@+id/editTextUsername"
    android:layout_alignBottom="@+id/editTextUsername"
    android:layout_alignLeft="@+id/editTextTweet"
    android:text="@string/at_sign"
    android:textAppearance=
        "?android:attr/textAppearanceLarge" />

  <EditText
    android:id="@+id/editTextUsername"
    android:layout_width="wrap_content"
    android:layout_height="wrap_content"
    android:layout_above="@+id/timelineButton"
    android:layout_toRightOf="@+id/textView2"
    android:ems="10"
    android:hint="@string/type_username_here" />

  <TextView
    android:id="@+id/textViewTimeline"
    android:layout_width="wrap_content"
    android:layout_height="wrap_content"
    android:layout_alignLeft="@+id/timelineButton"
    android:layout_below="@+id/timelineButton"
    android:maxLines="100"
    android:scrollbars="vertical"
    android:text="@string/timeline_here" />

  <Button
    android:id="@+id/timelineButton"
    android:layout_width="wrap_content"
    android:layout_height="wrap_content"
    android:layout_alignLeft="@+id/textView2"
    android:layout_centerVertical="true"
    android:onClick="onTimelineButtonClick"
    android:text="@string/timeline" />
```

```
  <Button
    android:id="@+id/tweetButton"
    android:layout_width="wrap_content"
    android:layout_height="wrap_content"
    android:layout_above="@+id/editTextUsername"
    android:layout_alignLeft="@+id/editTextTweet"
    android:layout_marginBottom="43dp"
    android:onClick="onTweetButtonClick"
    android:text="@string/tweet" />

  <EditText
    android:id="@+id/editTextTweet"
    android:layout_width="wrap_content"
    android:layout_height="wrap_content"
    android:layout_above="@+id/tweetButton"
    android:layout_alignParentLeft="true"
    android:layout_marginLeft="14dp"
    android:ems="10"
    android:hint="@string/type_your_tweet_here" />

  <TextView
    android:id="@+id/textViewCountChars"
    android:layout_width="wrap_content"
    android:layout_height="wrap_content"
    android:layout_alignBaseline="@+id/tweetButton"
    android:layout_alignBottom="@+id/tweetButton"
    android:layout_toRightOf="@+id/timelineButton"
    android:text="@string/zero" />

</RelativeLayout>
```

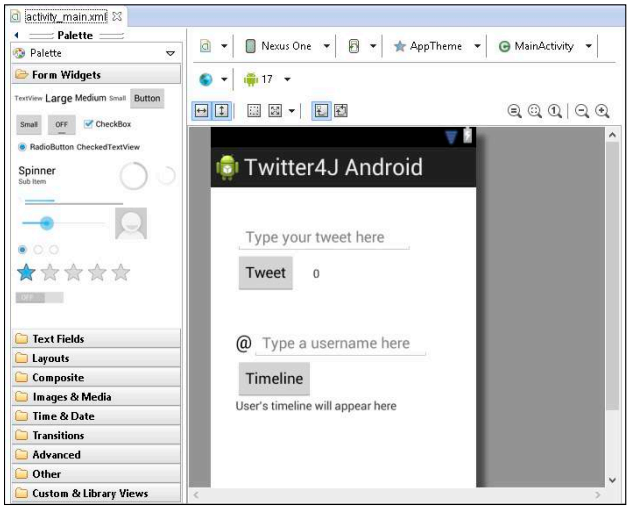

Figure 13-1: The graphical layout of the main activity's screen.

The twitter4j.properties file

This chapter's example involves a file that you don't find in most other apps. It's a file of the kind you see in Listing 13-3.

Listing 13-3: A Fake twitter4j.properties File (Yes, It's Fake!)

```
oauth.consumerKey=01qid0qod5drmwVJIkU1dg
oauth.consumerSecret=TudvMiX1h37WsIvq173SNWnRIhI0ALnGfS1
oauth.accessToken=1385541-ueSEFeFgwQJ8vUpfy6LBv6FibSfm5aXF
oauth.accessTokenSecret=G2FXeXYLSHPI7XlVdMsS2eGfIaKU6nJc
```

The `twitter4j.properties` file lives directly inside your project's `src` directory, as shown in Figures 13-2 and 13-3. Each line of the file gives your app important information for communicating with Twitter.

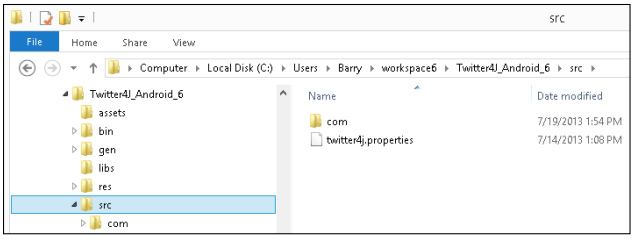

Figure 13-2: The location of `twitter4j.properties` on a Windows computer.

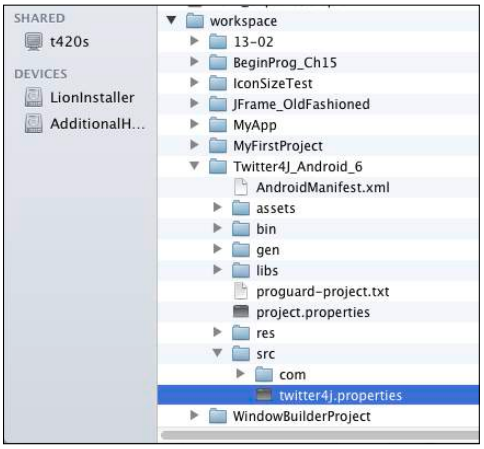

Figure 13-3: The location of `twitter4j.properties` on a Mac.

When you run this chapter's example, the code has to talk to Twitter on your behalf. And normally, to talk to Twitter, you supply a username and password. But should you be sharing your Twitter password with any app that comes your way? Probably not. Your password is similar to the key to your house. You don't want to give copies of your house key to strangers, and you don't want an Android app to remember your Twitter password.

So how can your app post a tweet without having your Twitter password? One answer is *OAuth,* a standardized way to have apps log on to host computers. If the gobbledygook in Listing 13-3 is copied correctly, the app acquires revocable permission to act on behalf of the Twitter user. And the app never gets hold of the user's password.

Now, here come the disclaimers:

- **A discussion of how OAuth works, and why it's safer than using ordinary Twitter passwords, is far beyond the scope of this book.**

  I don't pretend to explain OAuth and its mysteries in this chapter.

- **True app security requires more than a simple `twitter4j.properties` file.**

  The goal of this chapter is to show how an app can talk to a social media site. In the code, I use OAuth and Twitter4J commands to achieve that goal as quickly as I can, without necessarily showing you the "right" way to do it. For more comprehensive coverage of OAuth, visit `http://oauth.net`: the official website for OAuth developers.

- **The codes in Listing 13-3 don't work.**

  I'm not prepared to share my own OAuth codes with the general public, so to create Listing 13-3, I took the general outline of my real `twitter4j.properites` file and then ran my fingers over the keyboard to replace most of the characters.

  To run this chapter's app, you have to create your own set of OAuth keys and copy them to your `twitter4j.properties` file. The next section outlines the steps.

Getting OAuth codes

For your Android app to communicate with Twitter servers, you need your own OAuth codes. To get them, follow this section's steps.

The following instructions apply to the Twitter web pages for developers at the time of this book's publication. Twitter might change the design of its website at any time without notice. (At any rate, it won't notify me!)

1. **Sign in to your Twitter user account (or register for an account if you don't already have one).**

2. **Visit** `https://dev.twitter.com/apps/new`**.**

   If the stars are aligned harmoniously, you should see Twitter's Create an Application page.

3. **On the Create an Application page, fill in all required fields along with the (misleadingly optional) Callback URL field.**

   When I visit the page, I see the Name field, the Description field, the Website field, and the Callback URL field. All but the Callback URL field are listed as being required.

   Typing your name in the Name field is a no-brainer. But what do you use for the other fields? After all, you aren't creating an industrial-strength Android app. You're creating only a test app — an app to help you see how to use Twitter4J.

   The good news is that almost anything you type in the Description field is okay. The same is true for the Website and Callback URL fields, as long as you type things that look like real URLs.

   I've never tried typing a `twitter.com` URL in either the Website or Callback URL fields, but I suspect that typing a `twitter.com` URL doesn't work.

   To communicate with Twitter via an Android app, you need a callback URL. In other words, for this chapter's example, the callback URL isn't optional. Neither the Website field nor the Callback URL field has to point to a real web page. But you must fill in those two fields.

   The Callback URL field isn't marked as being required. Nevertheless, you must type a URL in the Callback URL field.

4. **After agreeing to the terms, and doing the other stuff to prove that you're a good person, click the Create Your Twitter Application button.**

   Doing so brings you to a page where you see some details about your new application — the Details tab, in other words. On this page, you see four important items: your app's access level, consumer key, and consumer secret and a button that offers to create your app's access token.

   In the OAuth world, an app whose code communicates with Twitter's servers is a *consumer*. To identify itself as a trustworthy consumer, an app must send passwords to Twitter's servers. In OAuth terminology, these passwords are called the *consumer key* and the *consumer secret*.

5. **On that same web page, select your application's Settings tab.**

6. **Among the settings, look for a choice of access types. Change your app's access from Read Only (the default) to Read, Write and Access Direct Messages.**

   For this toy application, you select Read, Write and Access Direct Messages — the most permissive access model that's available. This option prevents your app from hitting brick walls because of access problems. But when you develop a real-life application, you do the opposite — you select the least permissive option that suits your application's requirements.

   First change your app's access level, and then create the app's access token (as explained in Step 9). Don't create the access token before changing the access level. If you try to change the access level after you've created the access token, your app won't work. What's worse, the `dev.twitter.com` page won't warn you about the problem. Believe me — I've wasted hours of my life on this Twitter quirk.

7. **Click the button that offers to update your application's settings.**

   Doing so changes your app's access level to Read, Write and Access Direct Messages.

8. **Return to your application's Details tab.**

   I'm not thrilled with the way Twitter's developer site works. A title near the top of the Settings tab reads *Application Details*, and there's no title near the top of the Details tab. Anyway, find the Details tab and click on it.

9. **Click the button that offers to create your access token.**

   After doing so, your app's Details tab now displays your app's access token and the access token secret, in addition to your app's access level, consumer key, and consumer secret.

10. **Copy the four codes (Consumer Key, Consumer Secret, Access Token, and Access Token Secret) from your app's Details tab to the appropriate lines in your `twitter4j.properties` file.**

    Whew! You're done creating your `twitter4j.properites` file!

The Application's Main Activity

What's a *Java Programming For Android Developers For Dummies* book without some Java code? Listing 13-4 contains the Twitter app's Java code.

Listing 13-4: The MainActivity.java file

```
package com.allmycode.twitter;

import java.util.List;

import twitter4j.Twitter;
import twitter4j.TwitterException;
import twitter4j.TwitterFactory;
import android.app.Activity;
import android.os.AsyncTask;
import android.os.Bundle;
import android.text.Editable;
import android.text.TextWatcher;
import android.text.method.ScrollingMovementMethod;
import android.view.View;
import android.widget.EditText;
import android.widget.TextView;

public class MainActivity extends Activity {
  TextView textViewCountChars, textViewTimeline;
  EditText editTextTweet, editTextUsername;

  @Override
  protected void onCreate(Bundle savedInstanceState) {
    super.onCreate(savedInstanceState);
    setContentView(R.layout.activity_main);
    editTextTweet =
        (EditText) findViewById(R.id.editTextTweet);
    editTextTweet.addTextChangedListener
                                  (new MyTextWatcher());
    textViewCountChars =
        (TextView) findViewById(R.id.textViewCountChars);
    editTextUsername =
        (EditText) findViewById(R.id.editTextUsername);
    textViewTimeline =
        (TextView) findViewById(R.id.textViewTimeline);
    textViewTimeline.setMovementMethod
                        (new ScrollingMovementMethod());
  }

  // Button click listeners

  public void onTweetButtonClick(View view) {
    new MyAsyncTaskTweet().execute
                  (editTextTweet.getText().toString());
  }

  public void onTimelineButtonClick(View view) {
    new MyAsyncTaskTimeline().execute
                (editTextUsername.getText().toString());
```

```
}

// Count characters in the Tweet field

class MyTextWatcher implements TextWatcher {

  @Override
  public void afterTextChanged(Editable s) {
    textViewCountChars.setText
              ("" + editTextTweet.getText().length());
  }

  @Override
  public void beforeTextChanged(CharSequence s,
                  int start, int count, int after) {
  }

  @Override
  public void onTextChanged(CharSequence s,
                  int start, int before, int count) {
  }

}

// The AsyncTask classes

public class MyAsyncTaskTweet
            extends AsyncTask<String, Void, String> {

  @Override
  protected String doInBackground(String... tweet) {
    String result = "";

    Twitter twitter = TwitterFactory.getSingleton();
    try {
      twitter.updateStatus(tweet[0]);
      result =
          getResources().getString(R.string.success);
    } catch (TwitterException twitterException) {
      result =
          getResources().getString(R.string.failure);
    }

    return result;
  }

  @Override
  protected void onPostExecute(String result) {
    editTextTweet.setHint(result);
    editTextTweet.setText("");
  }
```

(continued)

Listing 13-4 ***(continued)***

```
  }

  public class MyAsyncTaskTimeline
              extends AsyncTask<String, Void, String> {

    @Override
    protected String doInBackground(String... username) {
      String result = new String("");
      List<twitter4j.Status> statuses = null;

      Twitter twitter = TwitterFactory.getSingleton();
      try {
        statuses = twitter.getUserTimeline(username[0]);

      } catch (TwitterException twitterException) {
        twitterException.printStackTrace();
      }

      for (twitter4j.Status status : statuses) {
        result += status.getText();
        result += "\n";
      }
      return result;
    }

    @Override
    protected void onPostExecute(String result) {
      editTextUsername.setText("");
      textViewTimeline.setText(result);
    }
  }
}
```

Twitter's network protocols require that the device that runs this chapter's app is set to the correct time. I don't know how correct the "correct time" has to be, but I've had lots of trouble running the app on emulators. Either my emulator is set to get the time automatically from the network (and it gets the time incorrectly) or I set the time manually and the *seconds* part of the time isn't close enough. One way or another, the error message that comes back from Twitter (usually specifying a null authentication challenge) isn't helpful. So I avoid lots of hassle by avoiding emulators whenever I test this code. Rather than run an emulator, I set my phone or tablet to get the network time automatically. Then I run this chapter's app on that phone or tablet. I recommend that you do the same.

When you run the app, you see two areas. One area contains a Tweet button; the other area contains a Timeline button, as shown in Figure 13-4.

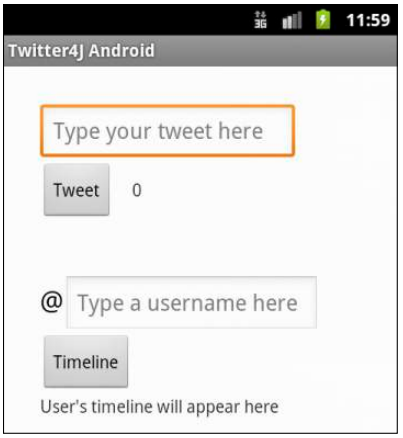

Figure 13-4: The main activity in its pristine state.

In Figure 13-4, the text in both text fields is light gray. This happens because I use `android:hint` attributes in Listing 13-2. A *hint* is a bunch of characters that appear only when a text field is otherwise empty. When the user clicks inside the text field, or types any text inside the text field, the hint disappears.

Type a tweet into the text field on top, and then press the Tweet button, as shown in Figure 13-5. If your attempt to tweet is successful, the message `Success!` replaces the tweet in the text field, as shown in Figure 13-6. If, for one reason or another, your tweet can't be posted, the message `Failed to tweet` replaces the tweet in the text field, as shown in Figure 13-7.

Figure 13-5: The user types a tweet.

Figure 13-6: The app indicates a successful tweet.

Figure 13-7: The app brings bad tidings to the user.

Next, type a username in the lower text field, and then click Timeline. If all goes well, a list of the user's most recent tweets appears below the Timeline button, as shown in Figure 13-8. You can scroll the list to see more of the user's tweets.

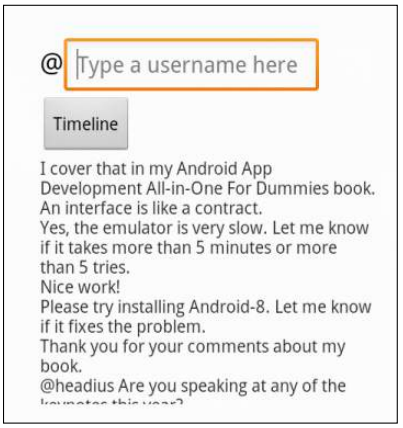

Figure 13-8: A user's timeline.

The onCreate method

The `onCreate` method in Listing 13-4 calls `findViewById` to locate some of the widgets declared in Listing 13-2.

For insight into the workings of Android's `findViewById` method, see Chapter 11.

The `onCreate` method also creates a `MyTextWatcher` instance to listen for changes in the field where the user types a tweet. Android notifies the `MyTextWatcher` instance whenever the user types characters in (or deletes characters from) the app's `editTextTweet` field. Later in Listing 13-4, the actual `TextChangedListener` class's `afterTextChanged` method counts

the number of characters in the `editTextTweet` field. The method displays the count in the tiny `textViewCountChars` field. (With the advent of Twitter, the number 140 has become quite important.)

This chapter's app doesn't do anything special if a user types more than 140 characters into the `editTextTweet` field. In a real-life app, I'd add code to handle 141 characters gracefully, but when I create sample apps, I like to keep the code as uncluttered as possible.

Android actually notifies the `MyTextWatcher` instance three times for each text change in the `editTextTweet` field — once before changing the text, once during the change of the text, and once after changing the text. In Listing 13-4, I don't make `MyTextWatcher` execute any statements before or during the changing of the text. In `MyTextWatcher`, the only method whose body contains statements is the `afterTextChanged` method. Even so, in order to implement Android's `TextWatcher` interface, the `MyTextWatcher` class must provide bodies for the `beforeTextChanged` and the `onTextChanged` methods.

Finally, in the `onCreate` method, the call to `setMovementMethod(new ScrollingMovementMethod())` permits scrolling on the list of items in a user's timeline.

The button listener methods

Listing 13-2 describes two buttons, each with its own `onClick` method. I declare the two methods in Listing 13-4 — the `onTweetButtonClick` method and the `onTimelineButtonClick` method. Each of the methods has a single statement in its body — a call to execute a newly constructed `AsyncTask` of some kind. Believe me, this is where the fun begins!

The trouble with threads

In Chapter 10, I describe the callback as the solution to all your activity's timing problems. Your activity wants to be alerted after a certain number of seconds passes. You can't stall the execution of your activity during those seconds. If you do, your activity is completely unresponsive to user input during those seconds. At best, users give your app a bad rating on Google Play; at worst, users pound on their screens, breaking the glass, blaming you, and sending you the repair bill.

Rather than put your activity to sleep for ten seconds, you create another class that sleeps on your activity's behalf. When the other class's nap

is finished, that other class issues a callback to your original activity. Everything is hunky-dory except for what I say in Chapter 10, in a paragraph with the little Technical Stuff icon on it:

"Well, I must confess that the code in Listings 10-10 through 10-13 doesn't solve the responsiveness problem. To make the program more responsive, you use the interface tricks in Listings 10-10 through 10-13 and, in addition, you put `TimerCommon` in a thread of its own."

Chapter 10 is the wrong place to describe threads. So in Chapter 10, the discussion of activity timing ends with a disappointing thud. (Yes, I'm secure enough to admit it.) Creating a thread means executing several different pieces of code at the same time. For the Java developer, things become complicated in no time at all. Juggling several simultaneous pieces of code is like juggling several raw eggs: One way or another, you're sure to end up with egg on your face.

To help fix all this, the creators of Android developed a multi-threading framework. Within this framework, you bundle all your delicately timed code into a carefully defined box. This box contains all the ready-made structure for managing threads in a well-behaved way. Rather than worry about where to put your `sleep` method calls and how to change a field's text in a timely fashion, you simply plug certain statements into certain places in the box and let the box's ready-made structure take care of all the routine threading details.

This marvelous box, the miracle cure for all your activity-timing ills, belongs to Android's `AsyncTask` classes. To understand these classes, you need a bit of terminology explained:

- **Thread:** A bunch of statements to be executed in the order prescribed by the code
- **Multi-threaded code:** A bunch of statements in more than one thread

  Java executes each thread's statements in the prescribed order. But if your program contains two threads, Java might not execute all the statements in one thread before executing all the statements in the other thread. Instead, Java might intermingle execution of the statements in the two threads. For example, I ran the following code several times:

```
package com.allmycode.threads;

public class TwoThreads {

  public static void main(String[] args) {
    new OneThread().start();
    new AnotherThread().start();
  }
```

```
}

class OneThread extends Thread {
  public void run() {
    System.out.print("1");
    System.out.print("2");
    System.out.print("3");
  }
}

class AnotherThread extends Thread {
  public void run() {
    System.out.print("A");
    System.out.print("B");
    System.out.print("C");
  }
}
```

The first time I ran the code, the output was `1AB23C`. The second time, the output was `123ABC`. The tenth time, the output was `ABC123`. The eleventh time, the output was `12AB3C`. The output `1` always comes before the output `2` because the statements to output `1` and `2` are in the same thread. But you can't predict whether Java will display `1` or `A` first, because the statements to output `1` and `A` are in two different threads.

- **The UI thread:** The thread that displays widgets on the screen

  In an Android program, your main activity runs primarily in the UI thread.

  The "UI" in "UI Thread" stands for "user interface." Another name for the UI thread is the *main thread*. The use of this terminology predates the notion of a main activity in Android.

- **A background thread:** Any thread other than the UI thread

  In an Android program, when you create an `AsyncTask` class, some of that class's code runs in a background thread.

In addition to all the terminology, you should know about two rules concerning threads:

- **Any time-consuming code should be in a background thread — not in the UI thread.**

  If you put time-consuming code in the UI thread, the app responds sluggishly to the user's clicks and keystrokes. Needless to say, users don't like this. In Chapter 10, a call to `sleep` for ten seconds is time-consuming code. In this chapter, any access to data over the Internet (like posting a tweet or getting a Twitter user's timeline) is time-consuming code.

- **Any code that modifies a property of the screen must be in the UI thread.**

  If, in a background thread, you have code that modifies text on the screen, you're either gumming up the UI thread or creating code that doesn't compile. Either way, you don't want to do it.

Android's AsyncTask

A class that extends Android's `AsyncTask` looks like the outline in Listing 13-5.

Listing 13-5: The Outline of an AsyncTask Class

```
public class MyAsyncTaskName
    extends AsyncTask<Type1, Type2, Type3> {

  @Override
  protected void onPreExecute () {
    // Execute statements in the UI thread before the
    // starting background thread. For example, display
    // an empty progress bar.
  }

  @Override
  protected Type3 doInBackground(Type1... param1) {
    // Execute statements in the background thread.
    // For example, get info from Twitter.

    return resultValueOfType3;
  }

  @Override
  protected void onProgressUpdate(Type2... param) {
    // Update a progress bar (or some other kind of
    // progress indicator) during execution of the
    // background thread.
  }

  @Override
  protected void onPostExecute(Type3 resultValueOfType3) {
    // Execute statements in the UI thread after
    // finishing the statements in the background thread.
    // For example, display info from Twitter in the
    // activity's widgets.
  }
}
```

When you create an `AsyncTask` class, Android executes each method in its appropriate thread. In the `doInBackground` method (refer to Listing 13-5), you put code that's too time-consuming for the UI thread. So Android executes the `doInBackground` method in the background thread. (Big surprise!) In Listing 13-5's other three methods (`onPreExecute`, `onProgressUpdate`, and `onPostExecute`), you put code that updates the widgets on the device's screen. So Android executes these methods in the UI thread, as shown in Figure 13-9.

Android also makes your life easier by coordinating the execution of an `AsyncTask` class's methods. For example, `onPostExecute` doesn't change the value of a screen widget until after the execution of `doInBackground`. (Refer to Figure 13-9.) In this chapter's Twitter app, the `onPostExecute` method doesn't update the screen until after the `doInBackground` method has fetched a user's timeline from Twitter. The user doesn't see a timeline until the timeline is ready to be seen.

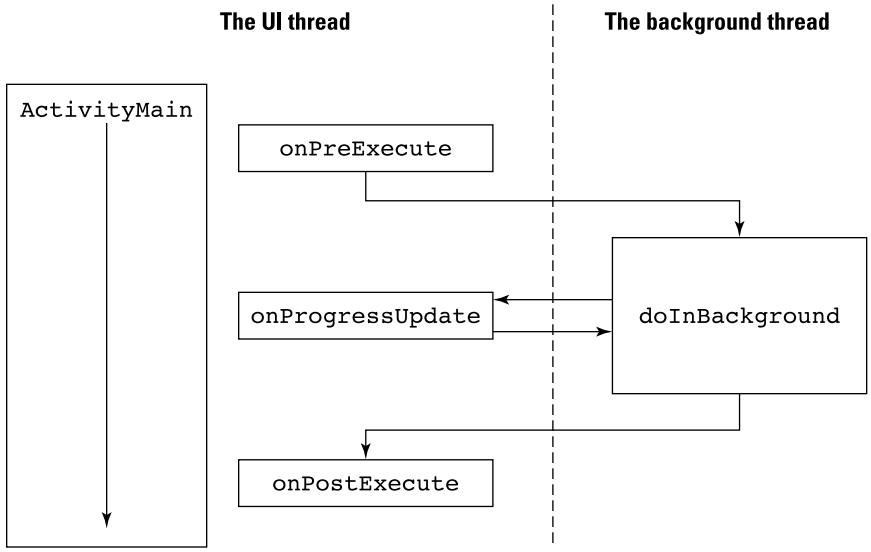

Figure 13-9: The UI thread and the background thread run simultaneously.

You'd think that with all this coordination of method calls, you lose any benefit from having more than one thread. But that's not the case. Because the `doInBackground` method runs outside the UI thread, your activity can respond to the user's clicks and drags while the `doInBackground` method waits for a response from the Twitter servers. It's all good.

My Twitter app's AsyncTask classes

Listing 13-5 contains four methods. But in Listing 13-4, I override only two of the methods — `doInBackground` and `onPostExecute`. The `MyAsyncTaskTweet` and `MyAsyncTaskTimeline` classes in Listing 13-4 inherit the other two methods from their superclass.

Notice (in Listings 13-4 and 13-5) the use of generic type names in an `AsyncTask` class. An `AsyncTask` is versatile enough to deal with all types of values. In Listing 13-4, the first generic parameter of `MyAsyncTaskTweet` has type `String` because a tweet is a string of as many as 140 characters. But someone else's `AsyncTask` might accept an image or a music file as its input. So when you create an `AsyncTask` class, you "fill in the blanks" by putting the following three type names inside the angle brackets:

- **The first type name (*`Type1`* in Listing 13-5) stands for a value (or values) that you pass to the `doInBackground` method.**

  The `doInBackground` method, with its `varargs` parameter, uses these values to decide what has to be done.

- **The second type name (*`Type2`* in Listing 13-5) stands for a value (or values) that mark the background thread's progress in completing its work.**

  This chapter's example has no progress bar, nor a progress indicator of any kind. So in Listing 13-4, the second type name is `Void`.

  In Java, the `Void` class is a wrapper class for the `void` value. Put that in your black hole of nothingness!

- **The third type name (*`Type3`* in Listing 13-5) stands for a value that the `doInBackground` method returns and that the `onPostExecute` method takes as a parameter.**

  In the `doInBackground` method of Listing 13-4, this third type name is `String`. It's `String` because the `doInBackground` method returns the word `"Success!"` or the words `"Failed to tweet"`, and the `onPostExecute` method displays these words in the screen's `editTextTweet` field.

Figure 13-10 summarizes the way generic type names influence the methods' types in Listing 13-4, and Figure 13-11 summarizes how values move from one place to another in the `MyAsyncTaskTweet` class of Listing 13-4.

An `AsyncTask` can be fairly complicated. But when you compare Android's `AsyncTask` to the do-it-yourself threading alternatives, the `AsyncTask` idea isn't bad at all. In fact, when you get a little practice and create a few of your

own `AsyncTask` classes, you get used to thinking that way. The whole business starts to feel quite natural.

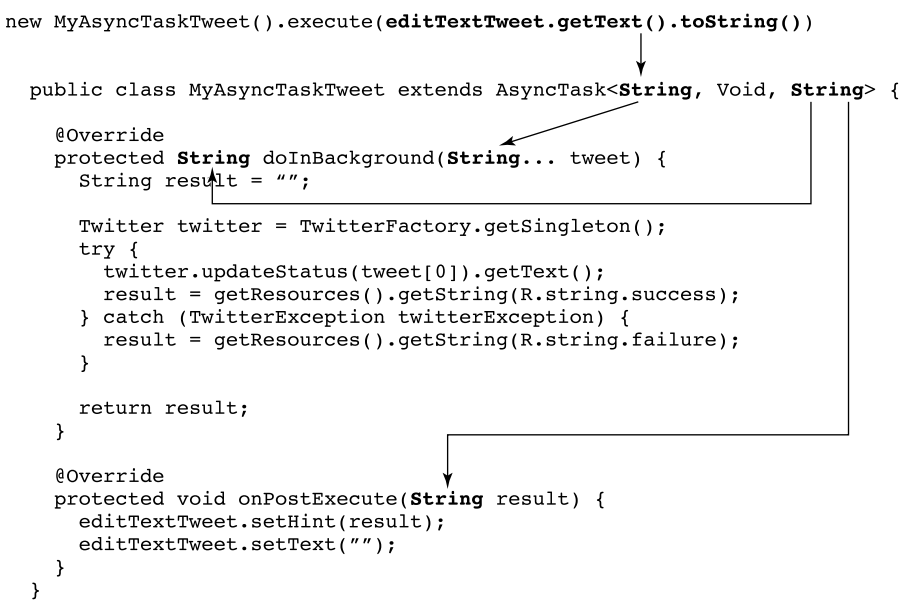

Figure 13-10: The use of types in an `AsyncTask` class.

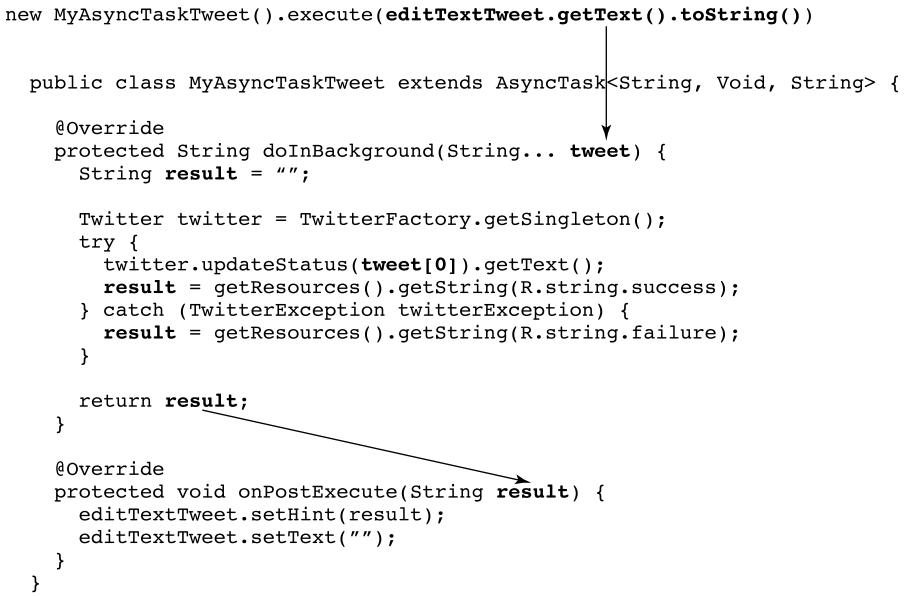

Figure 13-11: The flow of values in an `AsyncTask` class.

Cutting to the chase, at last

At the beginning of this chapter, I promise that a statement like

```
twitter.updateStatus("This is my tweet.");
```

lies at the heart of the code to post a tweet. You can see this by looking at the first `doInBackground` method in Listing 13-4. Here's a quick excerpt from that method:

```
protected String doInBackground(String... tweet) {

  Twitter twitter = TwitterFactory.getSingleton();
  twitter.updateStatus(tweet[0]);
```

In the Twitter4J API,

- **A `Twitter` object is a gateway to the Twitter servers.**
- **`TwitterFactory` is a class that helps you create a new `Twitter` object.**

  In Java, a *factory* class is a class that can call a constructor on your behalf.
- **Calling the `getSingleton` method creates a new `Twitter` object.**

  A factory method, such as `getSingleton`, calls a constructor on your behalf.
- **A call to the `Twitter` class's `updateStatus` method posts a brand-new tweet.**

In Listing 13-4, the parameter to the `updateStatus` method is an array element. That's because, in the `doInBackground` method's header, `tweet` is a `varargs` parameter. You can pass as many values to `doInBackground` as you want. In the body of the method, you treat `tweet` as though it's an ordinary array. The first `tweet` value is `tweet[0]`. If there were a second `tweet` value, it would be `tweet[1]`. And so on.

For the lowdown on varargs parameters, see Chapter 12.

In Listing 13-4, the code to fetch a user's timeline looks something like this:

```
List<twitter4j.Status> statuses = null;

Twitter twitter = TwitterFactory.getSingleton();
statuses = twitter.getUserTimeline(username[0]);
```

A fellow named Yusuke Yamamoto developed Twitter4J (or at least, Yusuke Yamamoto was the Twitter4J project leader), and at some point Mr. Yamamoto decided that the `getUserTimeline` method returns a collection of

twitter4J.Status objects. (Each twitter4J.Status instance contains one tweet.) So, to honor the contract set by calling the getUserTimeline method, the code in Listing 13-4 declares statuses to be a collection of twitter4J.Status objects.

A few lines later in the code, an enhanced for statement steps through the collection of statuses values and appends each value's text to a big result string. The loop adds "\n" (Java's go-to-the-next-line character) after each tweet for good measure. In the onPostExecute method, the code displays the big result string in the screen's textViewTimeline field.

In Listing 13-4, in the second doInBackground method, I use the fully qualified name twitter4j.Status. I do this to distinguish the twitter4J.Status class from Android's own AsyncTask.Status class (an inner class of the AsyncTask class).

For insight into Java's inner classes, refer to Chapter 11.

Java's Exceptions

Have I ever had something go wrong during the run of a program? (***Hint:*** The answer is yes.) Have you ever tried to visit a website and been unable to pull up the page? (Indubitably, the answer is yes.) Is it possible that Java statements can fail when they try to access the Twitter server? (Absolutely!)

In Java, most of the things that go wrong during the execution of a program are *exceptions.* When something goes wrong, your code *throws* an exception. If your code provides a way to respond to an exception, your code *catches* the exception.

Like everything else in Java, an exception is an object. Every exception is an instance of Java's Exception class. When your code tries to divide by zero (which is always a "no-no"), your code throws an instance of the ArithmeticException class. When your code can't read from a stored file, your code throws an instance of the IOException class. When your code can't access a database, your code throws an instance of the SQLException class. And when your Twitter4J code can't access the Twitter servers, your code throws an instance of the TwitterException class.

The classes ArithmeticException, IOException, SQLException, TwitterException, and many, many others are subclasses of Java's Exception class. The classes Exception, ArithmeticException, IOException, and SQLException are each part the Java's standard API library. The class TwitterException is declared separately in the Twitter4J API.

Java has two kinds of exceptions: *unchecked exceptions* and *checked exceptions*. The easiest way to tell one kind of exception from the other is to watch Eclipse's response when you type and run your code.

- **When you execute a statement that can throw an unchecked exception, you don't have to add additional code.**

  For example, an `ArithmeticException` is an unchecked exception. You can write and run the following (awful) Java program:

```
package com.allmycode.exceptions;

public class DoNotDoThis {

  public static void main(String[] args) {
    int i = 3 / 0;
  }

}
```

  When you try to run this code, the program crashes. In Eclipse's Console view, you see the message shown in Figure 13-12.

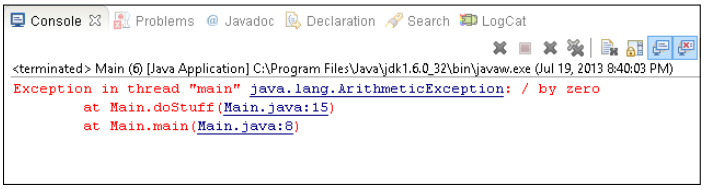

Figure 13-12: Shame on you! You divided by zero.

- **When you execute a statement that can throw a checked exception, you must add code.**

  A `TwitterException` is an example of a checked exception, and a call to `getUserTimeline` can throw a `TwitterException`. To find out what happens when you call `getUserTimeline` without adding code, see a portion of Eclipse's editor in Figure 13-13.

In Figure 13-13, the error message indicates that by calling the `getUserTimeline` method, you run the risk of throwing a `TwitterException`. The word "Unhandled" means that `TwitterException` is one of Java's checked exceptions, and that you haven't provided any code to address the possibility of the exception's being thrown. That is, if the app can't communicate with the Twitter servers, and Java throws a `TwitterException`, your code has no "Plan B."

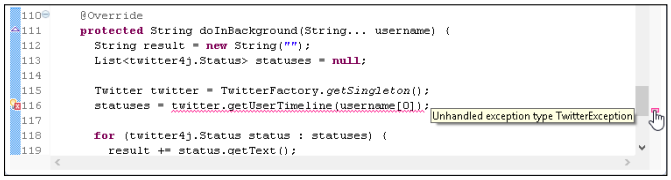

Figure 13-13: Java insists that you add code to acknowledge an exception.

So in Listing 13-4, I add Java's `try` / `catch` statement to my `getUserTimeline` call. Here's the translation of the `try` / `catch` statement:

```
try to execute this statement (or statements): {
  statuses = twitter.getUserTimeline(username[0]);

} If you throw a TwitterException while you're trying, {
  display a stack trace in Eclipse's LogCat view.
}
```

A *stack trace* is the kind of output (refer to Figure 13-12) that tells you which sequence of method calls caused the throwing of the exception. A stack trace can help you diagnose your code's ills.

Catch clauses

A `try` / `catch` statement has only one `try` clause. But a `try` / `catch` statement can have many `catch` clauses, as shown in this example:

```
try {
  count = numberOfTweets / averagePerDay;
  statuses = twitter.getUserTimeline(username[0]);
} catch (TwitterException e) {
  System.out.println("Difficulty with Twitter");
} catch (ArithmeticException a) {
  a.printStackTrace();
} catch (Exception e) {
  System.out.println("Something went wrong.");
}

System.out.println("No longer contacting Twitter");
```

When an exception is thrown inside a `try` clause, Java examines the accompanying list of `catch` clauses. Every `catch` clause has a parameter list, and every parameter list contains a type of exception.

Java starts at whatever `catch` clause appears immediately after the `try` clause and works its way down the program's text. For each `catch` clause, Java asks: Is the exception that was just thrown an instance of the class in this clause's parameter list? If it isn't, Java skips the `catch` clause and moves on to the next `catch` clause in line; if it is, Java executes the `catch` clause and then skips past all other `catch` clauses that come with this try clause. Java goes on and executes whatever statements come after the whole `try` / `catch` statement.

In the sample code with three `catch` clauses, if `averagePerDay` is zero, the code throws an `ArithmeticException`. Java skips past the `getUserTimeline` statement and looks at the `catch` clauses, starting with the topmost `catch` clause. Here's what happens.

The topmost catch clause is for `TwitterException` instances, but dividing by zero doesn't throw a `TwitterException`. So Java marches onward to the next `catch` clause.

The next `catch` clause is for `ArithmeticException` instances. Yes, dividing by zero threw an `ArithmeticException`. So Java executes the statement `a.printStackTrace()` and jumps out of the `try` / `catch` statement.

Java executes the statement immediately after the `try` / `catch` statement, displaying the words *No longer contacting Twitter*. Then Java executes any other statements after that one.

In the sample code with three `catch` clauses, I end the chain of `catch` clauses with an `Exception e` clause. Java's `Exception` class is an ancestor of `TwitterException` and `ArithmeticException` and all the other exception classes. No matter what kind of exception your code throws inside a `try` clause, that exception matches the `Exception e` catch clause. You can always rely on an `Exception e` clause as a last resort for handling a problem.

A finally clause

In addition to tacking on `catch` clauses, you can also tack a `finally` clause onto your `try` / `catch` statement. Java's `finally` keyword says, in effect, "Execute the `finally` clause's statements whether the code threw an exception or not." For example, in the following code snippet, Java always assigns `"Finished"` to the report variable, whether or not the call to `getUserTimeline` throws an exception:

```
String report = "";

try {
  statuses = twitter.getUserTimeline(username[0]);
} catch (TwitterException e) {
  e.printStackTrace();
} finally {
  report = "Finished";
}
```

Passing the buck

Here's a handy response to use whenever something goes wrong: "Don't blame me — tell my supervisor to deal with the problem." (I should have added the Tip icon to this paragraph!) When dealing with an exception, a Java method can do the same thing and say, "Don't expect me to have a `try` / `catch` statement — pass the exception on to the method that called me."

Listing 10-12, over in Chapter 10 calls the `Thread` class's `sleep` method. Execution of the `sleep` method can throw an `InterruptedException`, which is one of Java's checked exceptions. In the listing, I show you how to surround the call to `Thread.sleep` with a `try` / `catch` statement. In Listing 13-6, I show you how to insert that `try` / `catch` statement inside another Java program.

Listing 13-6: Nipping an Exception in the Bud

```
package com.allmycode.naptime;

class GoodNightsSleepA {

  public static void main(String args[]) {
    System.out.println("Excuse me while I nap.");
    takeANap();
    System.out.println("Ah, that was refreshing.");
  }

  static void takeANap() {
    try {
      Thread.sleep(10000);
    } catch (InterruptedException e) {
      System.out.println("Hey, who woke me up?");
    }
  }
}
```

In Listing 13-6, the `takeANap` method says "Try to sleep for 10,000 milliseconds. If your sleep is interrupted, handle it by displaying the question `Hey, who woke me up?`" Normally, no other thread interrupts the `takeANap` method's sleep. So, in Figure 13-14, the output of the code doesn't include `Hey, who woke me up?` (In the figure, you don't see the ten-second pause between the display of the first and second lines of output. To experience the full effect, look at the first line in Figure 13-14, pause for ten seconds, and then look at the second line.)

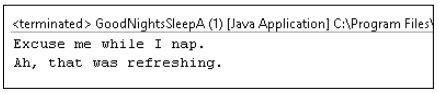

Figure 13-14: Running the code in Listing 13-6.

You can get rid of the `try` / `catch` statement in the `takeANap` method, as long as the next method upstream acknowledges the exception's existence. To see what I mean, look at Listing 13-7.

Listing 13-7: Make the Calling Method Handle the Exception

```
package com.allmycode.naptime;

class GoodNightsSleepB {

  public static void main(String args[]) {
    System.out.println("Excuse me while I nap.");
    try {
      takeANap();
    } catch (InterruptedException e) {
      System.out.println("Hey, who woke me up?");
    }
    System.out.println("Ah, that was refreshing.");
  }

  static void takeANap() throws InterruptedException {
    Thread.sleep(10000);
  }
}
```

In Listing 13-7, the `takeANap` method's header contains a `throws` clause that passes the buck from the `takeANap` method to whichever method calls the `takeANap` method. Because the `main` method calls `takeANap`, Java insists

that the `main` method contain code to acknowledge the possibility of an `InterruptedException`. To fulfill this responsibility, the `main` method surrounds the `takeANap` call with a `try` / `catch` statement.

Of course, the buck doesn't have to stop in the `main` method. You could say, "Don't blame me — tell my supervisor to deal with the problem." And then your supervisor could say, "Don't blame me — tell my supervisor to deal with the problem." The `main` method can avoid having a `try` / `catch` statement with its own `throws` clause (see Listing 13-8).

Listing 13-8: Keep Passing the Hot Potato

```
package com.allmycode.naptime;

class GoodNightsSleepC {

  public static void main(String args[])
                          throws InterruptedException {
    System.out.println("Excuse me while I nap.");
    takeANap();
    System.out.println("Hey, who woke me up?");
    System.out.println("Ah, that was refreshing.");
  }

  static void takeANap() throws InterruptedException {
    Thread.sleep(10000);
  }
}
```

If another thread interrupts this code's sleep time, the `takeANap` method passes the exception to the `main` method, which in turn passes the exception to the Java virtual machine. The Java virtual machine deals with the exception by displaying a stack trace and calling quits on your whole program. It's not the smartest way to handle a problem, but it's a legal alternative in Java.