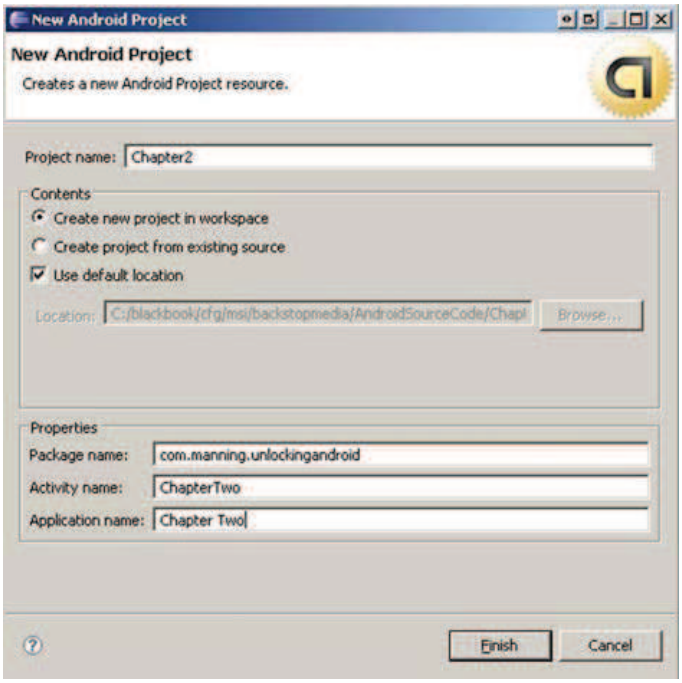

Figure 2.10 Using the Android Project Wizard, it's easy to create an empty Android application, ready for customization.

Figure 2.10 demonstrates the creation of a new project named `Chapter2` using the wizard.

> **TIP** You'll want the package name of your applications to be unique from one application to the next.

Click Finish to create your sample application. At this point, the application compiles and is capable of running on the emulator—no further development steps are required. Of course, what fun would an empty project be? Let's flesh out this sample application and create an Android tip calculator.

2.3.2 Android sample application code

The Android Application Wizard takes care of a number of important elements in the Android application structure, including the Java source files, the default resource files, and the AndroidManifest.xml file. Looking at the Package Explorer view in Eclipse, you can see all the elements of this application. Here's a quick description of the elements included in the sample application:

- The src folder contains two Java source files automatically created by the wizard.
- ChapterTwo.java contains the main `Activity` for the application. You'll modify this file to add the sample application's tip calculator functionality.
- R.java contains identifiers for each of the UI resource elements in the application. Never modify this file directly. It automatically regenerates every time a resource is modified; any manual changes you make will be lost the next time the application is built.

- Android.jar contains the Android runtime Java classes. This reference to the android.jar file found in the Android SDK ensures that the Android runtime classes are accessible to your application.
- The res folder contains all the Android resource folders, including:
  - Drawables contains image files such as bitmaps and icons. The wizard provides a default Android icon named icon.png.
  - Layout contains an XML file called main.xml. This file contains the UI elements for the primary view of your `Activity`. In this example, you'll modify this file but you won't make any significant or special changes—just enough to accomplish the meager UI goals for your tip calculator. We cover UI elements, including `Views`, in detail in chapter 3. It's not uncommon for an Android application to have multiple XML files in the Layout section of the resources.
  - Values contains the strings.xml file. This file is used for localizing string values, such as the application name and other strings used by your application.

AndroidManifest.xml contains the deployment information for this project. Although AndroidManifest.xml files can become somewhat complex, this chapter's manifest file can run without modification because no special permissions are required. We'll visit AndroidManifest.xml a number of times throughout the book as we discuss new features.

Now that you know what's in the project, let's review how you're going to modify the application. Your goal with the Android tip calculator is to permit your user to enter the price of a meal, then tap a button to calculate the total cost of the meal, tip included. To accomplish this, you need to modify two files: ChapterTwo.java and the UI layout file, main.xml. Let's start with the UI changes by adding a few new elements to the primary `View`, as shown in the next listing.

Listing 2.1 main.xml contains UI elements

```
<?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="Chapter 2 Android Tip Calculator"
    />
<EditText
    android:id="@+id/mealprice"
    android:layout_width="fill_parent"
    android:layout_height="wrap_content"
    android:autoText="true"
/>
<Button
android:id="@+id/calculate"
```

```
    android:layout_width="wrap_content"
    android:layout_height="wrap_content"
    android:text="Calculate Tip"
    />
<TextView
    android:id="@+id/answer"
    android:layout_width="fill_parent"
    android:layout_height="wrap_content"
    android:text=""
    />

</LinearLayout>
```

The layout for this application is straightforward. The overall layout is a vertical, linear layout with only four elements; all the UI controls, or *widgets*, are going to be in a vertical arrangement. A number of layouts are available for Android UI design, which we'll discuss in greater detail in chapter 3.

A static `TextView` displays the title of the application. An `EditText` collects the price of the meal for this tip calculator application. The `EditText` element has an attribute of type `android:id`, with a value of `mealprice`. When a UI element contains the `android:id` attribute, it permits you to manipulate this element from your code. When the project is built, each element defined in the layout file containing the `android:id` attribute receives a corresponding identifier in the automatically generated R.java class file. This identifying value is used in the `findViewById` method, shown in listing 2.2. If a UI element is static, such as the `TextView`, and doesn't need to be set or read from our application code, the `android:id` attribute isn't required.

A button named `calculate` is added to the view. Note that this element also has an `android:id` attribute because we need to capture click events from this UI element. A `TextView` named `answer` is provided for displaying the total cost, including tip. Again, this element has an `id` because you'll need to update it during runtime.

When you save the file main.xml, it's processed by the ADT plug-in, compiling the resources and generating an updated R.java file. Try it for yourself. Modify one of the `id` values in the main.xml file, save the file, and open R.java to have a look at the constants generated there. Remember not to modify the R.java file directly, because if you do, all your changes will be lost! If you conduct this experiment, be sure to change the values back as they're shown in listing 2.1 to make sure the rest of the project will compile as it should. Provided you haven't introduced any syntactical errors into your main.xml file, your UI file is complete.

> **NOTE** This example is simple, so we jumped right into the XML file to define the UI elements. The ADT also contains an increasingly sophisticated GUI layout tool. With each release of the ADT, these tools have become more and more usable; early versions were, well, early.

Double-click the main.xml file to launch the layout in a graphical form. At the bottom of the file you can switch between the Layout view and the XML view. Figure 2.11 shows the Layout tool.

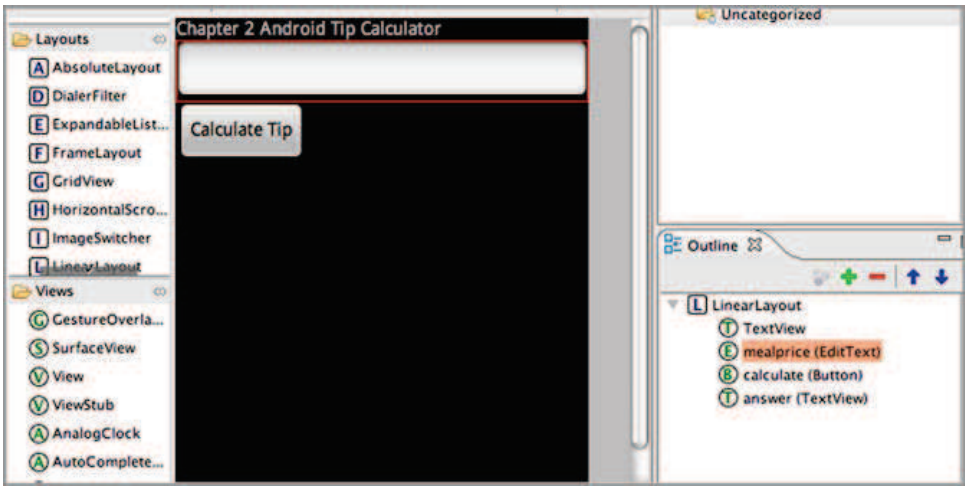

Figure 2.11 Using the GUI Layout tool provided in the ADT to define the user interface elements of your application

It's time to turn our attention to the file ChapterTwo.java to implement the tip calculator functionality. ChapterTwo.java is shown in the following listing. We've omitted some imports for brevity. You can download the complete source code from the Manning website at http://manning.com/ableson2.

Listing 2.2 ChapterTwo.java implements the tip calculator logic

```
package com.manning.unlockingandroid;
import com.manning.unlockingandroid.R;
import android.app.Activity;
import java.text.NumberFormat;
import android.util.Log;
// some imports omitted
public class ChapterTwo extends Activity {
  public static final String tag = "Chapter2";
    @Override
    public void onCreate(Bundle icicle) {
        super.onCreate(icicle);
        setContentView(R.layout.main);
        final EditText mealpricefield =
                    (EditText) findViewById(R.id.mealprice);
        final TextView answerfield =
                    (TextView) findViewById(R.id.answer);
        final Button button = (Button) findViewById(R.id.calculate);
        button.setOnClickListener(new Button.OnClickListener() {
            public void onClick(View v) {
                try {
                  Log.i(tag,"onClick invoked.");
                  // grab the meal price from the UI
                  String mealprice =
                  mealpricefield.getText().toString();
                  Log.i(tag,"mealprice is [" + mealprice + "]");
                  String answer = "";
                  // check to see if the meal price includes a "$"
```

1 Reference EditText for mealprice

2 Log entry

3 Get meal price

```
                    if (mealprice.indexOf("$") == -1) {
                       mealprice = "$" + mealprice;
                    }
                    float fmp = 0.0F;
                    // get currency formatter
                    NumberFormat nf =
                    java.text.NumberFormat.getCurrencyInstance();
                    // grab the input meal price
                    fmp = nf.parse(mealprice).floatValue();
                    // let's give a nice tip -> 20%
                    fmp *= 1.2;
                    Log.i(tag,"Total Meal Price (unformatted) is ["
+ fmp + "]");
                    // format our result
                    answer = "Full Price, Including 20% Tip: "
+ nf.format(fmp);
                    answerfield.setText(answer);
                    Log.i(tag,"onClick complete.");
                  } catch (java.text.ParseException pe) {
                    Log.i(tag,"Parse exception caught");
                    answerfield.setText("Failed to parse amount?");
                  } catch (Exception e)    {
                    Log.e(tag,"Failed to Calculate Tip:" + e.getMessage());
                    e.printStackTrace();
                    answerfield.setText(e.getMessage());
                  }
               }
            });
      }
}
```

4 **Display full price, including tip**

5 **Catch parse error**

Let's examine this sample application. Like all but the most trivial Java applications, this class contains a statement identifying which package it belongs to: `com.manning.unlockingandroid`. This line containing the package name was generated by the Application Wizard.

We import the `com.manning.unlockingandroid.R` class to gain access to the definitions used by the UI. This step isn't required, because the `R` class is part of the same application package, but it's helpful to include this import because it makes our code easier to follow. Newcomers to Android always ask how the identifiers in the `R` class are generated. The short answer is that they're generated automatically by the ADT! Also note that you'll learn about some built-in UI elements in the `R` class later in the book as part of sample applications.

Though a number of imports are necessary to resolve class names in use, most of the import statements have been omitted from listing 2.2 for the sake of brevity. One import that's shown contains the definition for the `java.text.NumberFormat` class, which is used to format and parse currency values.

Another import shown is for the `android.util.Log` class, which is employed to make entries to the log. Calling static methods of the `Log` class adds entries to the log. You can view entries in the log via the LogCat view of the DDMS perspective. When making entries to the log, it's helpful to put a consistent identifier on a group of

related entries using a common string, commonly referred to as the *tag*. You can filter on this string value so you don't have to sift through a mountain of LogCat entries to find your few debugging or informational messages.

Now let's go through the code in listing 2.2. We connect the UI element containing `mealprice` to a class-level variable of type `EditText` (1) by calling the `findViewById` method and passing in the identifier for the `mealprice`, as defined by the automatically generated `R` class, found in R.java. With this reference, we can access the user's input and manipulate the meal price data as entered by the user. Similarly, we connect the UI element for displaying the calculated answer back to the user, again by calling the `findViewById` method.

To know when to calculate the tip amount, we need to obtain a reference to the `Button` so we can add an event listener. We want to know when the button has been clicked. We accomplish this by adding a new `OnClickListener` method named `onClick`.

When the `onClick` method is invoked, we add the first of a few log entries using the static `i()` method of the `Log` class (2). This method adds an entry to the log with an `Information` classification. The `Log` class contains methods for adding entries to the log for different levels, including Verbose, Debug, Information, Warning, and Error. You can also filter the LogCat based on these levels, in addition to filtering on the process ID and tag value.

Now that we have a reference to the `mealprice` UI element, we can obtain the text entered by our user with the `getText()` method of the `EditText` class (3). In preparation for formatting the full meal price, we obtain a reference to the static currency formatter.

Let's be somewhat generous and offer a 20 percent tip. Then, using the formatter, let's format the full meal cost, including tip. Next, using the `setText()` method of the `TextView` UI element named `answerfield`, we update the UI to tell the user the total meal cost (4).

Because this code might have a problem with improperly formatted data, it's a good practice to put code logic into `try/catch` blocks so that our application behaves when the unexpected occurs (5).

Additional boilerplate files are in this sample project, but in this chapter we're concerned only with modifying the application enough to get basic, custom functionality working. You'll notice that as soon as you save your source files, the Eclipse IDE compiles the project in the background. If there are any errors, they're listed in the Problems view of the Java perspective; they're also marked in the left margin with a small red x to draw your attention to them.

TIP Using the command-line tools found in the Android SDK, you can create batch builds of your applications without using the IDE. This approach is useful for software shops with a specific configuration-management function and a desire to conduct automated builds. In addition to the Android-specific build tools found under the tools subdirectory of your Android SDK

installation, you'll also need JDK version 5.0 or later to complete command-line application builds. Creating sophisticated automated builds of Android applications is beyond the scope of this book, but you can learn more about the topic of build scripts by reading *Ant in Action: Second Edition of Java Development with Ant*, by Steve Loughran and Erik Hatcher, found at http://www.manning.com/loughran/.

Assuming there are no errors in the source files, your classes and UI files will compile correctly. But what needs to happen before your project can be run and tested in the Android emulator?

2.3.3 Packaging the application

At this point, your application has compiled and is ready to be run on the device. Let's look more deeply at what happens after the compilation step. You don't need to perform these steps because the ADTs handle these steps for you, but it's helpful to understand what's happening behind the scenes.

Recall that despite the compile-time reliance on Java, Android applications don't run in a Java VM. Instead, the Android SDK employs the Dalvik VM. For this reason, Java bytecodes created by the Eclipse compiler must be converted to the .dex file format for use in the Android runtime. The Android SDK has tools to perform these steps, but thankfully the ADT takes care of all of this for you transparently.

The Android SDK contains tools that convert the project files into a file ready to run on the Android emulator. Figure 2.12 depicts the generalized flow of source files in the Android build process. If you recall from our earlier discussion of Android SDK tools, the tool used at design time is `aapt`. Application resource XML files are processed by `aapt`, with the R.java file created as a result—remember that you need to refer to the `R` class for UI identifiers when you connect your code to the UI. Java source

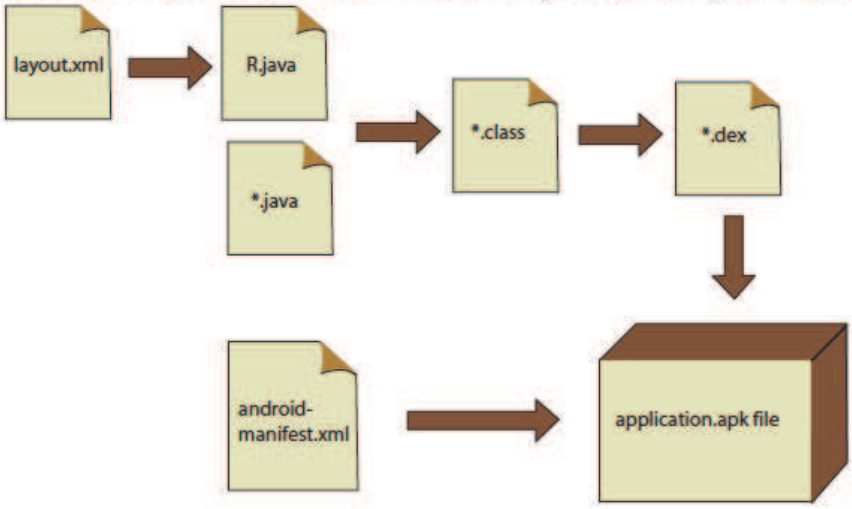

Figure 2.12 The ADT employs tools from the Android SDK to convert source files to a package that's ready to run on an Android device or emulator.