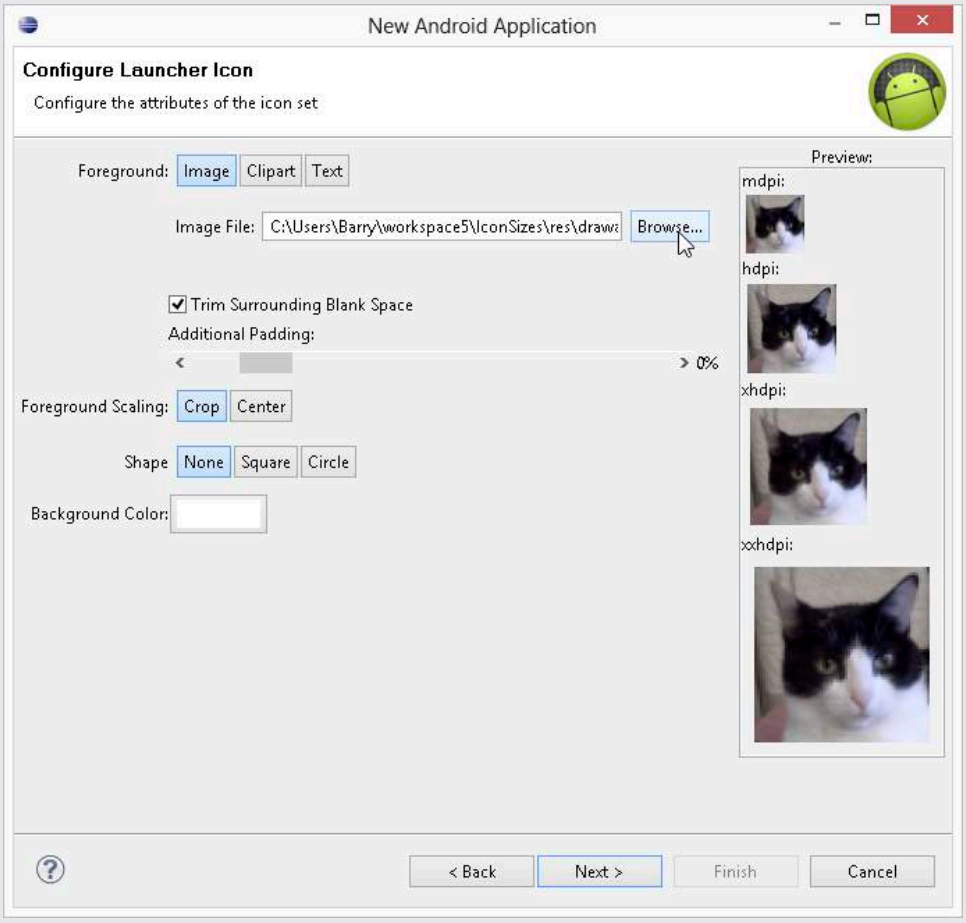
Part IV

Powering Android with Java Code



web extras

Check out the article "Using Android Asset Studio" (and more) online at www.dummies.com/extras/javaprogrammingforandroiddevelopers.

In this part . . .

- Responding to touches, clicks, and pops
- Becoming a collector (in the Java sense)
- Creating an app that uses social media
- Creating an Android game

Chapter 11

A Simple Android Example: Responding to a Button Click

In This Chapter

- How to make a button do something
- Putting a class inside another class
- Using Android's special tricks to avoid programming hassles

In common English usage, an *insider* is someone with information that's not available to most people. An insider gets special information because of her position within an organization.

American culture has many references to insiders. Author John Gunther became famous for writing *Inside Europe* and *Inside Africa* and other books in his *Inside* series. On TV crime shows, an inside job is a theft or a murder committed by someone who works in the victim's own company. So significant is the power of inside information that in most countries, insider stock trading is illegal.

In the same way, a Java class can live inside another Java class. When this happens, the inner class has useful insider information. This chapter explains why.

The First Button-Click Example

Ever heard that wonderful old joke about a circus acrobat jumping over mice? Unfortunately, I'd get sued for copyright infringement if I included it in this book. Anyway, the joke is about starting small and working your way up to bigger things. That's what you do when you read *Java Programming For Android Developers For Dummies*. Most of the programs in this book aren't

Android apps. Instead, the programs are standard Oracle Java apps — apps that run on a desktop or a laptop computer, not on an Android device. In fact, the `JOptionPane.showMessageDialog` method that I use in many of this book's examples runs only on a desktop or laptop, not on Android.

Why does a book with the word *Android* featured prominently in its title contain many examples that don't run on phones or tablet devices? The answer is that you must always start practicing by jumping over small mice. Compare the sets of instructions in Chapters 3 and 4, and notice how much more work is involved in running a Hello World Android app. When you practice creating several Android apps, you become accustomed to the eccentricities of Android's emulator. But when you're learning Java, you don't want an emulator's quirks to get in the way. Java is Java, whether it's a standard Java app to display the words *Hello World* or an Android app to send a spaceship to another world.

Anyway, by the time you reach Chapter 11, you're ready to see some Java features running on a phone or on a phone emulator. So this chapter's example, simple though it might be, is specific to Android.

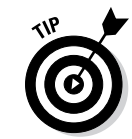
Creating the Android app

You can import this chapter's code from my website (`http://allmycode.com/Java4Android`) by following the instructions in Chapter 2. But if you want to create the example on your own, follow the next several steps:

1. **Follow the instructions in Chapter 4 to create a skeletal Android application.**
2. **Expand your new project's branch in Eclipse's Package Explorer tree, on the left.**
3. **In the project's branch, navigate to the `res/layout` directory.**
4. **In the `res/layout` directory, double-click the `activity_main.xml` item.**

   A graphical layout of your app shows up in Eclipse's editor, as shown in Figure 11-1.

   You can resize this Graphical Layout view of the app by clicking the little magnifying glass icons near the upper-right corner of the editor.



   The `activity_main.xml` file contains a bunch of XML code describing the look (the layout) of your Android activity. To switch between the picture displayed in Figure 11-1 and the actual XML code, select either the Graphical Layout tab or the `activity_main.xml` tab at the bottom of Eclipse's editor.

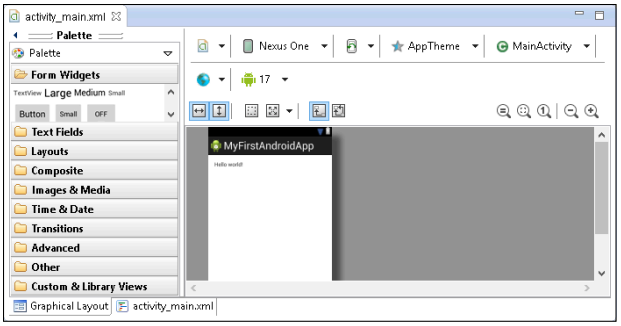


Figure 11-1: A blank layout.

5. **In the palette on the left side of the graphical layout, expand the Form Widgets category.**

   In this Form Widgets category, you find `TextView` elements, buttons, check boxes, and other doodads, as shown in Figure 11-2.

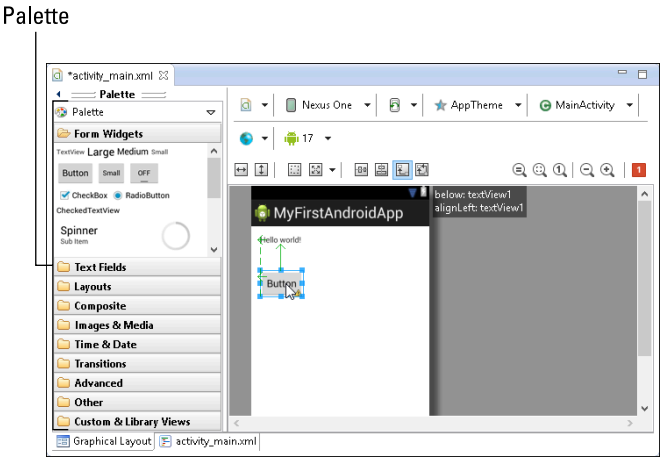


Figure 11-2: Dragging a button onto your app's layout.

6. **Drag a button from the Form Widgets category onto your app's layout (refer to Figure 11-2).**

7. **Notice the names of the items in Eclipse's Outline view.**

   Most likely, the names you see are `RelativeLayout`, `textView1`, and `button1`, as shown in Figure 11-3.

8. **On Eclipse's main menu, select File➪Save.**

   Doing so saves your changes to the `activity_main.xml` file.

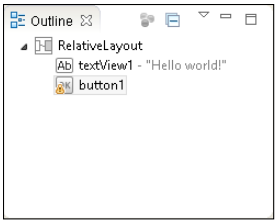


Figure 11-3: Eclipse's Outline view.

9. **In the Package Explorer tree, navigate to your project's `src` directory.**

10. **Inside the `src` directory, expand the branch for the package containing the project.**

    The package name will resemble the name `com.example.myfirstandroidapp`.

11. **Within the package's branch, double-click the `MainActivity.java` file.**

    The activity's Java file appears in Eclipse's editor. Eclipse created all this code for you.

12. **In the editor, add the following fields to the `MainActivity` class's code:**

```
Button button;
TextView textView;
```

    Listing 11-1 shows you exactly where to place the new section of code. (Note that you still have to add some more code; that comes up in Step 13.)

    Android's `Button` class is in the `android.widget` package. To use the short name `Button`, your code needs an import declaration. You can type an import declaration yourself. Alternatively, Eclipse can add the import declaration for you. After adding the `Button button1` field and seeing the nasty red squiggle underneath the name `TextView`, press Ctrl+Shift+O. (That's the letter *O,* not the digit 0.) This shortcut adds the import declaration automatically. If you don't like memorizing shortcut keys, you can achieve the same effect by selecting Source➪Organize Imports from Eclipse's main menu.

13. **In Eclipse's editor, add the following statements immediately after the call to `setContentView`:**

```
button = (Button) findViewById(R.id.button1);
button.setOnClickListener
                    (new MyOnClickListener(this));
textView = (TextView) findViewById(R.id.textView1);
```

    Listing 11-1 has the proper placement of this new code.

The editor displays a red squiggle under the name `MyOnClick Listener` because you haven't yet declared the `MyOnClickListener` class. You do that in the next few steps.

In this step, I assume that the item names you see in Step 7 are `text View1` and `button1`. If you see different names (such as `textView01` and `button01`), use those alternative names after each occurrence of `R.id` in Listing 11-1. For example, rather than use `R.id.button1`, use `R.id.button01`. (No matter what names you see in Step 7, you don't have to change the names `button` and `textView` that you create in Step 12. You can make up any variable names as long as you use these variable names consistently throughout the class's code.)

14. **In Eclipse's main menu, select File➪Save.**

    Doing so saves your changes to the `MainActivity.java` file.

15. **In the Package Explorer, right-click (on a Mac, Control-click) the branch displaying your app's package name.**

    The package name will resemble `com.example.myfirstandroidapp`.

16. **From the resulting contextual menu that appears, select New➪Class.**

    The New Java Class dialog box appears.

17. **In the Name field of the New Java Class dialog box, type `MyOnClick Listener` (the same name you typed in the code in Step 13).**

18. **Click Finish to dismiss the New Java Class dialog box.**

    The New Java Class dialog box goes away, and a minimal `MyOnClickListener` class appears in Eclipse's editor. This class contains (more or less) the following code:

```
package com.example.myfirstandroidapp;

public class MyOnClickListener {

}
```

19. **Add code to the (newly created) `MyOnClickListener` class, as shown in Listing 11-2.**

20. **In Eclipse's main menu, select File➪Save.**

    Doing so saves your changes to the new `MyOnClickListener.java` file.

21. **Run your new Android app.**

When you run the new app, you start with the display shown in Figure 11-4. After clicking the button, you see the display shown in Figure 11-5.

Listing 11-1: Your Main Activity

```
package com.example.myfirstandroidapp;

import android.app.Activity;
import android.os.Bundle;
import android.view.Menu;
import android.widget.Button;
import android.widget.TextView;

public class MainActivity extends Activity {
  Button button;
  TextView textView;

  @Override
  protected void onCreate(Bundle savedInstanceState) {
    super.onCreate(savedInstanceState);
    setContentView(R.layout.activity_main);
    button = (Button) findViewById(R.id.button1);
    button.setOnClickListener
                          (new MyOnClickListener(this));
    textView = (TextView) findViewById(R.id.textView1);
  }

  @Override
  public boolean onCreateOptionsMenu(Menu menu) {
    getMenuInflater().inflate(R.menu.main, menu);
    return true;
  }

}
```

Listing 11-2: A Class Listens for Button Clicks

```
package com.example.myfirstandroidapp;

import android.view.View;
import android.view.View.OnClickListener;

public class MyOnClickListener
                      implements OnClickListener {
  MainActivity caller;

  public MyOnClickListener(MainActivity activity) {
    this.caller = activity;
  }

  public void onClick(View view) {
    caller.textView.setText("You clicked the button!");
  }
}
```

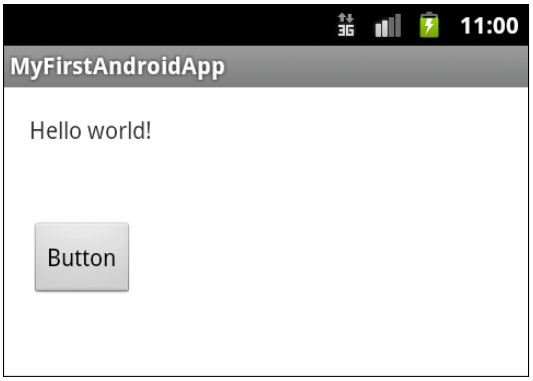


Figure 11-4: Beginning a run of the code in Listings 11-1 and 11-2.

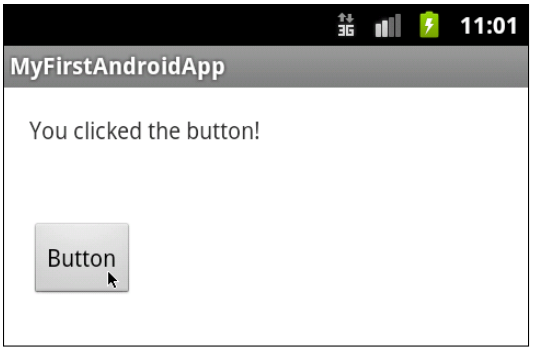


Figure 11-5: What you see after clicking the button in Listings 11-1 and 11-2.

The code in Listings 11-1 and 11-2 performs a callback, much the same as the callback I describe in Chapter 10. In this chapter's callback, the `MyOnClickListener` class calls back to the activity's `textView` object. As in Chapter 10, the callback is possible for two reasons:

- **Android's built-in `setOnClickListener` method expects its parameter to implement Android's `OnClickListener` interface.**

  Here's how it works in Listings 11-1 and 11-2:

  - In Listing 11-1, the call to `setOnClickListener` has, as its parameter, a new `MyOnClickListener` object.
  - As Listing 11-2 shows, the `MyOnClickListener` class implements Android's `OnClickListener` interface.

  If I don't make the `My`*`Whatever`* class implement the `OnClickListener` interface, this call is illegal:

```
button.setOnClickListener(new MyWhatever(this));
```

- **The `MyOnClickListener` object knows how to call back the activity that constructed it.**

  Again, in Listing 11-1, the `MyOnClickListener` constructor call passes `this` to its new `MyOnClickListener` object. ("Call *me* back," says your activity's code in Listing 11-1.) See Figure 11-6.

  Then, in Listing 11-2, the `MyOnClickListener` constructor makes a mental note of who gets called back, by storing a reference to your activity in its own `caller` field. So, when push comes to shove, the code in Listing 11-2 calls back `caller.textView.setText`, which changes the words displayed in the original activity's `textView`.

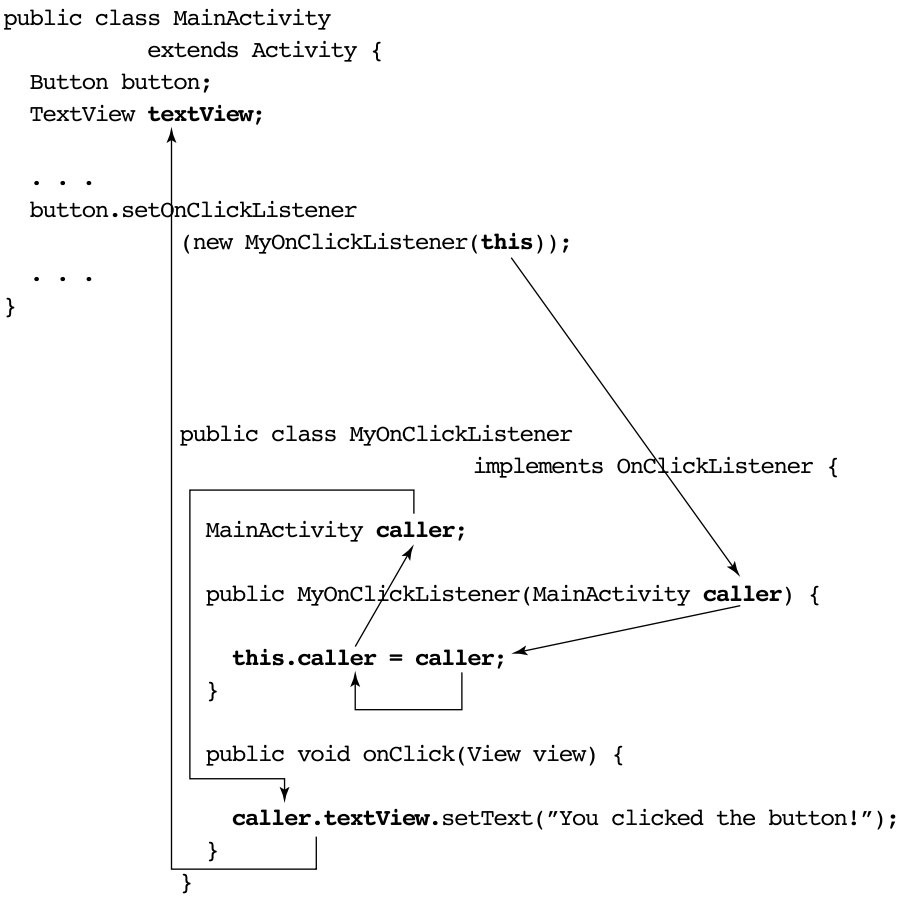


Figure 11-6: The journey of your application's main activity.

Making a view available to your Java code

In Listing 11-2, you execute the statement

```
caller.textView.setText("You clicked the button!");
```

For this statement to work correctly, the `textView` variable must refer to a particular widget on the activity's screen. In particular, the `textView` variable must refer to the widget that displays the words *Hello world!* in Figure 11-4. Presumably, this "widget" that displays those words is an instance of Android's `TextView` class. But who knows? Maybe the code has a mistake in it. (I don't presume to know much about the `textView` variable until later in this section.)

Anyway, there's a problem. That Hello World widget isn't declared anywhere inside Listings 11-1 or 11-2. Instead, it appears because of some stuff in the application's `activity_main.xml` file. You need a way to connect the stuff in the XML file with the `textView` variable in your Java code.

How can you do that? In Chapter 4, I describe the way code numbers stand for strings in Android apps. You put a line such as

```
<string name="hello_world">Hello world!</string>
```

in one of your project's XML files. As a result, Android automatically puts the following lines (and many more lines like them) inside an `R.java` file:

```
public final class R {

  public static final class string {
    public static final int hello_world=0x7f040001;
  }

}
```

Because of this code number mechanism, you can refer to Hello world! in your code with the value `R.string.hello_world`. Under the hood, the name `R.string.hello_world` stands for the hexadecimal number 0x7f040001, though you care only about the name `R.string.hello_world`.

One way or another, this whole code-number mechanism with its `R.java` file allows you to connect values in your XML files with names in your Java code.

All hexadecimal numbers in `R.java` files are arbitrary values. The number to represent the `hello_world` string (the number 0x7f040001 in my example) might be different in someone else's `R.java` file or even in the `R.java` file that Eclipse generates for you tomorrow. You can use the name `R.string.hello_world` in your code, but never use the number 0x7f040001.

Sometimes, I can't resist seeing my code's sordid underbelly. If mysterious hex numbers bother you, find a website that converts hexadecimal numbers to and from decimal numbers. You can type 0x7f040001, for example, into a field on the web page and learn that it has the same value as the ordinary decimal number 2130968577. You can't do much with this information, but it's nice to know that hexadecimal numbers don't involve any special magic.

An Android app has several XML files, one of which describes the layout of the app's main activity. Listing 11-3 contains the `activity_main.xml` file for this section's example.

Listing 11-3: A 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/textView1"
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:text="@string/hello_world" />

    <Button
        android:id="@+id/button1"
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:layout_alignLeft="@+id/textView1"
        android:layout_below="@+id/textView1"
        android:layout_marginTop="46dp"
        android:text="Button" />

</RelativeLayout>
```

The code that you see in your own project's `activity_main.xml` file might not be exactly the same as the code in Listing 11-3. For example, when you drag a button onto your layout in Step 6, you might drop the button in a slightly different place inside your activity's screen. This is no big deal.

Fortunately, you don't have to type the code in Listing 11-3. Eclipse's tools do the typing for you when you create your new Android project and you drop a button into the app's graphical layout.

In Listing 11-3, the lines

```
<TextView
    android:id="@+id/textView1"
 . . .
<Button
    android:id="@+id/button1"
```

tell Android to display a text view and a button on your activity's screen. These lines also tell Eclipse to create code numbers for the new text view and the new button. Finally, these particular lines tell Eclipse to add some code to your project's `R.java` file, as shown in Listing 11-4.

Listing 11-4: A Few Lines from Your Project's R.java File

```
public final class R {
   ...

  public static final class id {
    public static final int button1=0x7f080001;
    public static final int textView1=0x7f080000;
  }

   ...
}
```

The lines in this listing associate the names `R.id.button1` and `R.id.textView1` with the numbers `0x7f080001` and `0x7f080000`. So, indirectly, the lines in Listing 11-4 associate the names `R.id.button1` and `R.id.textView1` with the button and the text view on your main activity's screen.

In your application's main activity (refer to Listing 11-1), Android's `findViewById` method completes the chain of associations. The `findViewById` method takes a number as its parameter (a number such as 0x7f080000 — the value of `R.id.textView1`). The `findViewById` method looks up that number and finds the widget associated with it (a widget from Listing 11-3).

Figure 11-7 illustrates the chain of associations in concrete terms. Using the code snippet in Listing 11-4, the call

```
findViewById(R.id.textView1)
```

sends Android hunting for a view that's connected with the number 0x7f080000. And because of the clever way that Eclipse generates the `R.java` file, the number 0x7f080000 is associated with the appropriate text view widget on the activity's screen. Armed with the intermediate name `R.id.textView1`, your Java code manages to find the appropriate widget in the activity's screen layout.

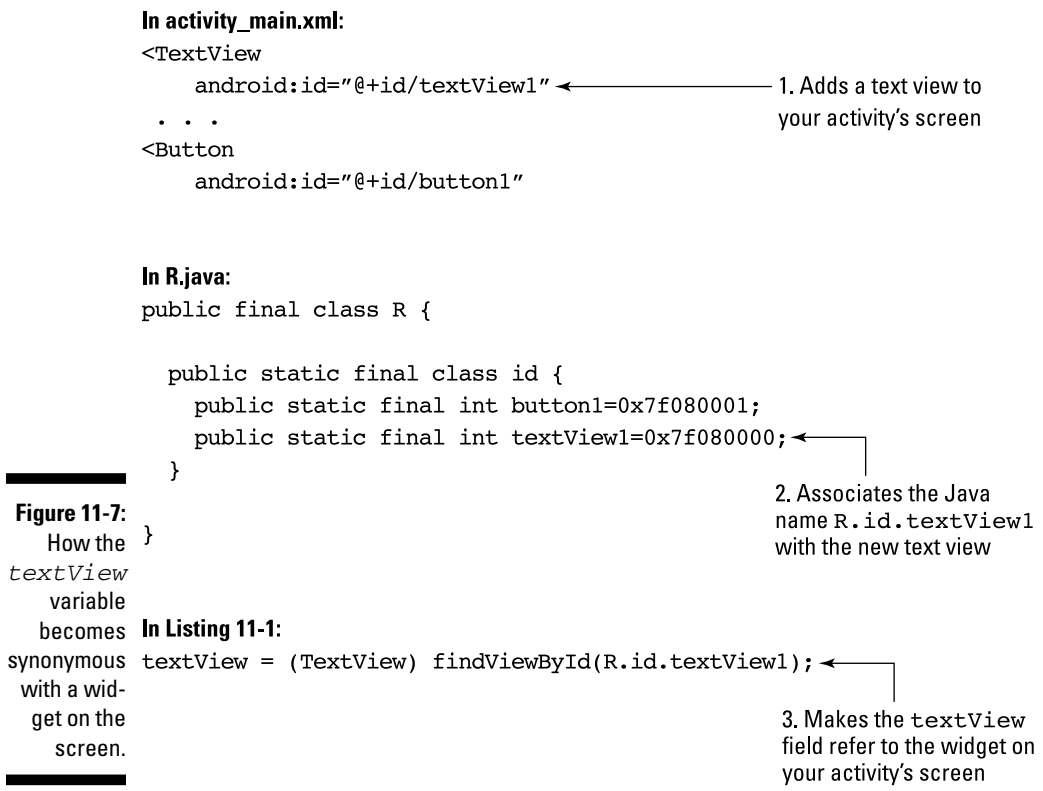


Figure 11-7: How the *textView* variable becomes synonymous with a widget on the screen.

Casting, again

When you call `findViewById`, Java doesn't know what kind of view it will find. The `findViewById` method always returns a `View` instance, but lots of Android's classes extend the `View` class. For example, the classes `Button`, `TextView`, `ImageView`, `CheckBox`, `Chronometer`, and `RatingBar` all extend Android's `View` class. If you type the following code:

```
// DON'T DO THIS!!

TextView textView;

textView = findViewById(R.id.textView1);
```

Java lets out a resounding, resentful roar: "How dare you assume that the object returned by a call to `findViewById` refers to an instance of the `TextView` class!" (Actually, Java quietly and mechanically displays an error message in Eclipse's editor. But I like to personify Java as though it's a stern taskmaster.)

In Listing 11-1, you appease the Java gods by adding a casting operator to the code. You tell Java to convert whatever pops out of the `findViewById` method call into a `TextView` object.

```
textView = (TextView) findViewById(R.id.textView1);
```

While you're typing the code, Java humors you and says, "Your casting operator shows me that you're aware of the difference between a `TextView` and any old `View`. I'll do my best to interpret the `View` object that I find at runtime as a `TextView` object." (Actually, while you're typing the code, Java says nothing. The fact that Java doesn't display any error messages when you use this casting trick is a good sign. Java's casting feature saves the day!)

Casting prevents you from seeing an error message while you develop your code. In that way, casting is quite a useful feature of Java. But casting can't save you if your code contains runtime errors. In Step 7, you verify that the name `textView1` represents a `TextView` widget. When the app runs, Java grabs the `R.id.textView1` widget from the `activity_main.xml` file, and everything works just fine. But you may sometimes forget to check your `R.java` names against the widgets in the XML file. A call to `findViewById` spits out an `ImageView` widget when your casting tells Java to expect a `TextView` widget. When this happens, Java chokes on the casting operator and your app crashes during its run. Back to the drawing board!

For a more complete discussion of casting, see Chapter 7.

Introducing Inner Classes

Does the diagram in Figure 11-6 seem unnecessarily complicated? Look at all those arrows! You might expect to see a few somersaults as the `caller` object bounces from place to place! The `MyOnClickListener` class (refer to Listing 11-2) devotes much of its code to obsessively keeping track of this `caller` object. Is there a simpler way to handle a simple button click?

There is. You can define a class inside another class. When you do, you're creating an *inner class*. It's a lot like any other class. But within an inner class's code, you can refer to the enclosing class's fields with none of the froufrou in Listing 11-2. That's why, at the beginning of this chapter, I sing the praises of insider knowledge.

One big class with its own inner class can replace both Listings 11-1 and 11-2. And the new inner class requires none of the exotic gyrations that you see in the old `MyOnClickListener` class. Listing 11-5 contains this wonderfully improved code.

Listing 11-5: A Class within a Class

```
package com.allmycode.myfirstandroidapp;

import android.app.Activity;
import android.os.Bundle;
import android.view.Menu;
import android.view.View;
import android.view.View.OnClickListener;
import android.widget.Button;
import android.widget.TextView;

public class MainActivity extends Activity {
  Button button;
  TextView textView;

  @Override
  protected void onCreate(Bundle savedInstanceState) {
    super.onCreate(savedInstanceState);
    setContentView(R.layout.activity_main);
    button = (Button) findViewById(R.id.button1);
    button.setOnClickListener(new MyOnClickListener());
    textView = (TextView) findViewById(R.id.textView1);
  }

  @Override
  public boolean onCreateOptionsMenu(Menu menu) {
    getMenuInflater().inflate(R.menu.main, menu);
    return true;
```

```
  }

  class MyOnClickListener implements OnClickListener {

    public void onClick(View view) {
      textView.setText("You clicked the button!");
    }
  }

}
```

When you run the code in Listing 11-5, you see the results shown earlier, in Figures 11-4 and 11-5.

Notice the relative simplicity of the new `MyOnClickListener` class in Listing 11-5. Going from the old `MyOnClickListener` class (refer to Listing 11-2) to the new `MyOnClickListener` inner class (refer to Listing 11-5), you reduce the code's size by a factor of three. But aside from the shrinkage, all the complexity of Figure 11-6 is absent from Listing 11-5. The use of `this`, `caller`, and `textView` in Listings 11-1 and 11-2 feels like a tangled rope. But in Listing 11-5, when you pull both ends of the rope, you find that the rope *isn't* knotted.

An inner class needs no fancy bookkeeping in order to keep track of its enclosing class's fields. Near the end of Listing 11-5, the line

```
textView.setText("You clicked the button!");
```

refers to the `MainActivity` class's `textView` field, which is exactly what you want. It's that straightforward.

No Publicity, Please!

Notice that the code in Listing 11-5 uses the `MyOnClickListener` class only once. (The only use is in a call to `button.setOnClickListener`.) So I ask, do you really need a name for something that's used only once? No, you don't. (If there's only one cat in the house, it's safe to say "Hey, cat!")

When you give a name to your disposable class, you have to type the name twice: once when you call the class's constructor:

```
button.setOnClickListener(new MyOnClickListener());
```

and a second time when you declare the class:

```
class MyOnClickListener implements OnClickListener {
```

To eliminate this redundancy, you can substitute the entire definition of the class in the place where you'd ordinarily call the constructor. When you do this, you have an *anonymous inner class*. Listing 11-6 shows you how it works.

Listing 11-6: A Class with No Name (Inside a Class with a Name)

```
package com.allmycode.myfirstandroidapp;

import android.app.Activity;
import android.os.Bundle;
import android.view.Menu;
import android.view.View;
import android.view.View.OnClickListener;
import android.widget.Button;
import android.widget.TextView;

public class MainActivity extends Activity {
  Button button;
  TextView textView;

  @Override
  protected void onCreate(Bundle savedInstanceState) {
    super.onCreate(savedInstanceState);
    setContentView(R.layout.activity_main);
    button = (Button) findViewById(R.id.button1);
    button.setOnClickListener(new OnClickListener() {
      public void onClick(View view) {
        textView.setText("You clicked the button!");
      }
    });
    textView = (TextView) findViewById(R.id.textView1);
  }

  @Override
  public boolean onCreateOptionsMenu(Menu menu) {
    getMenuInflater().inflate(R.menu.main, menu);
    return true;
  }

}
```

A run of the code from Listing 11-6 is shown in Figures 11-4 and 11-5. In other words, the listing does exactly the same thing as its wordier counterparts in this chapter. The big difference is that, unlike this chapter's previous examples, the listing uses an anonymous inner class.

An anonymous inner class is a lot like an ordinary inner class. The big difference is that an anonymous inner class has no name. Nowhere in Listing 11-6

do you see a name like `MyOnClickListener`. Instead, you see what looks like an entire class declaration inside a call to `button.setOnClick Listener`. It's as though the `setOnClickListener` call says, "The following listener class, which no one else refers to, responds to the button clicks."

As far as I'm concerned, the most difficult aspect of using an anonymous inner class is keeping track of the code's parentheses, curly braces, and other non-alphabetic characters. Notice, for example, the string of closing punctuation characters — `!");}});` — that straddles a few lines in Listing 11-6. The indentation in that listing helps a little bit when you try to read a big *mush* of anonymous inner class code, but it doesn't help a lot. Fortunately, there's a nice correspondence between the code in Listing 11-5 and the anonymized code in Listing 11-6. Figure 11-8 illustrates this correspondence.

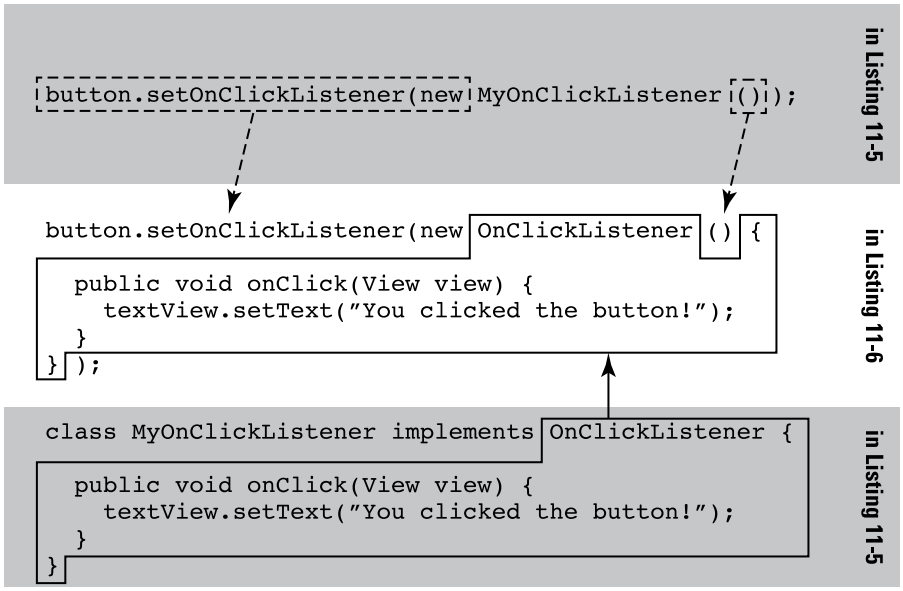


Figure 11-8: Turning ordinary inner class code (refer to Listing 11-5) into anonymous inner class code (refer to Listing 11-6).

I feel obliged to include a written explanation of the material in Figure 11-8. Here goes:

> To go from a named inner class to an anonymous inner class, you replace the named class's constructor call with the entire class declaration. In place of the class name, you put the name of the interface that the inner class implements (or, possibly, the name of the class that the inner class extends).

If you find my explanation helpful, I'm pleased. But if you don't find it helpful, I'm neither offended nor surprised. When I create a brand-new inner class, I find my gut feeling and Figure 11-8 to be more useful than Java's formal grammar rules.

My humble advice: Start by writing code with no inner classes, such as the code in Listing 11-5. Later, when you become bored with ordinary Java classes, experiment by changing some of your ordinary classes into anonymous inner classes.

Doing It the Easy Way

With all the fuss about callbacks and inner classes in this chapter, I'm tempted to end the chapter right here. So this is the last paragraph in Chapter 11. Don't read any further in this chapter. Really, there's nothing more to see here. Move on, everybody!

I warned you to skip the rest of this chapter

Starting with Android 1.6 (code-named Donut, API Level 4), developers can add click-handling code to a button, or to any other Android widget, without creating a separate class. You don't need the extra Java file in Listing 11-2 or the inner class in Listing 11-5 or the anonymous inner class in Listing 11-6.

The `onClick` attribute described in this section (which you shouldn't be reading) lets you handle clicks and other occurrences without coding any additional classes. The `onClick` feature is quite convenient. But the `onClick` feature's existence doesn't mean that you, the Android developer, don't have to understand callbacks and inner classes. Interfaces, callbacks, and inner classes are used implicitly and explicitly in almost every Android application. And, among all the introductory inner class examples, button clicking is one of the easiest to understand. The use of an inner class to handle a button click is an old standby among Java programming examples, so I use this example in Chapter 11, even though Android provides this section's fast, convenient `onClick` attribute work-around.

The "no-hassle" way to click a button

In this section's example, your app's activity class handles the button click on its own. You don't create an additional class to handle the click.

You can follow the instructions in Chapter 2 to import this section's example from my website (`http://allmycode.com/Java4Android`). But if you want to create the example on your own, follow these steps:

1. **Follow Steps 1 through 7 earlier in this chapter, in the section "The First Button-Click Example."**
2. **Right-click (in Windows) or Control-click (on the Mac) the picture of the button in the graphical layout.**
3. **From the contextual menu that appears, select Other Properties➪Inherited from View➪OnClick.**

   A dialog box with a field labeled New onClick Value appears, as shown in Figure 11-9.

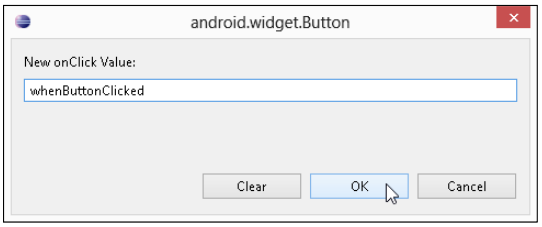


Figure 11-9: The New onClick Value dialog box.

4. **In the New onClick Value field, type the name of a method.**

   In Figure 11-9, I typed the name `whenButtonClicked`. In the next several steps, I assume that you type the same name, `whenButtonClicked`.
5. **Click OK to dismiss the dialog box.**
6. **In Eclipse's main menu, select File➪Save.**

   Doing so saves your changes to the `activity_main.xml` file.

   Your actions in Steps 2 through 6 tell Android to look for a method with the following header when the user clicks the button:

   ```
   public void whenButtonClicked(View view)
   ```

   In the remaining steps, you add that `whenButtonClicked` method to your app's Java code.
7. **In the Package Explorer, navigate to your project's `src` directory.**
8. **Inside the `src` directory, expand the branch for the package containing the project.**

   The package name is probably similar to `com.example.myfirstandroidapp`.

9. **Within the package's branch, double-click the** `MainActivity.java` **file.**

   The activity's Java file appears in Eclipse's editor. Eclipse created all this Java code for you.

10. **In the editor, add the following field to your** `MainActivity` **class's code:**

    ```
    TextView textView;
    ```

    See Listing 11-7.

    After typing the `TextView` declaration, select Source➪Organize Imports from Eclipse's main menu. When you do, Eclipse automatically adds the `TextView` class's import declaration to your code.

11. **In Eclipse's editor, add the following statement immediately after the call to** `setContentView`**:**

    ```
    textView = (TextView) findViewById(R.id.textView1);
    ```

    Refer to Listing 11-7.

12. **Type the** `whenButtonClicked` **method in your code.**

    You can find the method in Listing 11-7.

13. **From Eclipse's main menu, select File➪Save.**

    Doing so saves your changes to the `activity_main.xml` file.

14. **Run your project.**

    The project runs exactly as it did in this chapter's examples. Under the hood, Android creates all the necessary callbacks to have your `whenButtonClicked` method respond to the user's actions.

Listing 11-7: Adding the whenButtonClicked Method to Your Code

```
package com.example.myfirstandroidappnew;

import android.app.Activity;
import android.os.Bundle;
import android.view.Menu;
import android.view.View;
import android.widget.TextView;

public class MainActivity extends Activity {
  TextView textView;

  @Override
  protected void onCreate(Bundle savedInstanceState) {
```

```
    super.onCreate(savedInstanceState);
    setContentView(R.layout.activity_main);
    textView = (TextView) findViewById(R.id.textView1);
  }

  public void whenButtonClicked(View view) {
    textView.setText("You clicked the button!");
  }

  @Override
  public boolean onCreateOptionsMenu(Menu menu) {
    getMenuInflater().inflate(R.menu.main, menu);
    return true;
  }

}
```