Chapter 1: Creating Code Quickly with App Inventor

In This Chapter

- **Designing a user interface**
- **Building program logic using visual tools**
- **Creating an Android app without typing any code**

Life is a long sequence of tradeoffs. You enjoy a big gooey dessert, but the dessert isn't healthy. You love playing in a band, but most band members don't earn a decent living. You're blessed with the ability to sleep 10 or 12 hours a day, but with all that lost time, you get behind writing *Android Application Development All-in-One For Dummies*.

And so it goes. Application development also involves tradeoffs. To use the full richness of Android's feature set, you write Java code using Eclipse and the Android API. But if you can forgo some of Android's fancier features, you can save time and effort using Google's App Inventor.

App Inventor is the GUI dessert that I describe in the first paragraph. App Inventor makes Android development easy, using a drag-and-drop paradigm for both the visual interface and for an application's logic. With App Inventor, you can't fine-tune your work the way you can with Java code. You can't make your application do some of the trickier things that Android programs do. But App Inventor hides many of the messy technical details involved in developing mobile code. For a quick, easy way to develop basic Android applications, give App Inventor a try. (In fact, for a quick, easy way to develop *not-so-basic* applications, App Inventor is worth a look.)

Getting Started with App Inventor

To begin your App Inventor experience, do the following:

1. **Visit** `http://java.com`.

   During your visit, you can check to make sure that your computer has the Java Runtime Environment (JRE). To use App Inventor, you don't need the full Java Development Kit (JDK). But you do need the Java Runtime Environment (JRE).

If you've already followed the steps in Book I, Chapter 2, your computer has the JRE. Or if you know, for one reason or another, that your computer has the JRE, you can skip this step and go immediately to Step 4.

To develop Android apps with Eclipse, you need the Java Development Kit. In particular, you need Java SE (the development kit's Standard Edition). For more information about the Java Development Kit and the Java Runtime Environment, see Book I, Chapter 2.

2. **At the Java website, follow the steps to check whether Java is installed on your computer.**

   The website's terminology differs from the official terminology. The website offers to check whether you have "Java" on your computer. To be precise, the website checks whether your computer has the Java Runtime Environment (JRE) and whether the JRE is enabled in your web browser.

   Most computers come with the JRE preinstalled. So if you run the site's "Does my computer have Java?" test, and if the Java web page responds with a `Java is not working` message, check to make sure that Java is enabled in your web browser.

3. **Follow the instructions at `http://java.com` to install Java on your computer.**

   Do this if your computer doesn't have Java, or if you don't want to fiddle with your web browser's settings, or if you may already have Java but you don't care if you install Java again.

   With the JRE installed on your computer, you can proceed to Step 4.

4. **Visit `www.appinventorbeta.com`.**

   By the time you read this book, the URL for App Inventor may have changed. If so, try poking around at `www.media.mit.edu` for a pointer to App Inventor. (The MIT Media Lab took over stewardship of App Inventor from Google in August 2011.)

5. **Sign in with your Google account (or create an account if you don't already have one).**

6. **Find the link to download and install App Inventor's Setup software on your computer.**

   Download links appear in several places on the App Inventor site.

7. **Download and install App Inventor's Setup software.**

   In Windows, the Setup software installs itself in the `c:\Program Files\AppInventor` directory (or on 64-bit systems, in the `C:\Program Files (x86)\AppInventor` directory). On a Mac, the Setup software sits comfortably in the `/Applications/Appinventor` folder. In Linux, the Setup software normally nestles inside the `/usr/google/appinventor` directory.

What manner of beast is the App Inventor?

App Inventor is really three things — a *Designer*, a *Blocks Editor*, and some Setup software:

- With the **Designer,** you specify the appearance of your Android application's user interface.
- With the **Blocks Editor,** you develop your Android application's behavior. ("Here's what happens in this text box when the user clicks that button.")

  Both the Designer and Blocks Editor programs live on the web. Each time you use App Inventor, you download the code for these programs from Google's servers to your computer's hard drive. While you run these programs, the programs communicate frequently with Google's servers.
- The **Setup software** lives locally on your computer's hard drive. This code includes some Android Software Development Kit (SDK) tools and an Android device emulator.

The Designer runs in a web browser, and the Blocks Editor runs in a Java Web Start window. A Java Web Start window is like a Java applet running outside a web browser. A run of the Blocks Editor looks very much like any ordinary program running on your desktop. But unlike most other programs, you download the Blocks Editor's code each time you use App Inventor.

When you download App Inventor's Setup software, you download and install a small portion of the Android SDK with an emulator and a few other gizmos. But the parts of App Inventor that you see most often (the Designer and the Blocks Editor) live on Google's servers, not on your computer's hard drive.

(P.S.: By the time you read this book, someone will have explained to me why the folks at Google split up App Inventor into a browser-based part and a Java Web Start part. When I find out, I'll probably be embarrassed for not having known sooner.)

Everything changes. Or, as the French say, "The more things change, the more things in Barry's books become out of date." I'm guessing that Google's App Inventor will change quickly while this book is in print, so don't take this chapter's instructions as gospel. Google's software people designed App Inventor from the ground up to be friendly and intuitive. So if my instructions don't match precisely what you see on your computer screen, poke around a bit. If you really get stuck, send me an e-mail. (My address is in this book's introduction.)

Creating a Project

Follow these steps to create and test a bare-bones App Inventor project:

1. **Visit** `www.appinventorbeta.com`**.**

A visit to this site brings you to one of several possible web pages. Which page you see depends on the amount of App Inventor stuff that you've already done. If you've already downloaded and installed App Inventor, the site takes you to either App Inventor's Projects page or to App Inventor's Designer page.

The Projects page lists the projects that you've created. (See Figure 1-1.)

The Designer page is the main interface to App Inventor's Designer program. On the Designer page, you lay out your Android application's screen, drag components onto the application's screen, and set the components' properties. (See Figure 1-2.)

You can navigate between the Projects page and the Designer page. When you're on the Projects page, clicking the main toolbar's New button (to create a new project) brings you to the Designer page. Alternatively, you can select the name of one of your existing projects.

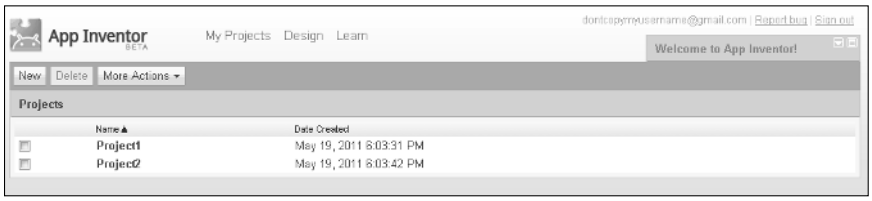

Figure 1-1: The Projects page.

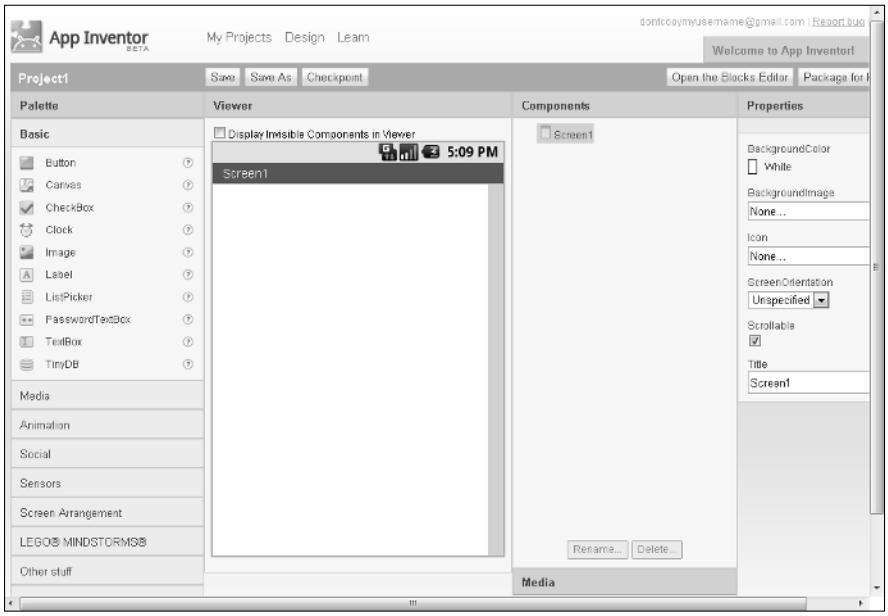

Figure 1-2: The Designer page.

When you're on the Designer page, select the My Projects link to return to the Projects page.

Remember: Neither Ralph Lauren nor Christian Dior has anything to do with the App Inventor's Designer page.

2. **If you're not already on the Projects page, click the My Projects link to go to the Projects page.**

   In Figure 1-2, the My Projects link is at the top of the page on the right side.

3. **On the Projects page, click the New button.**

   The New App Inventor for Android Project dialog box opens.

4. **Type a project name, and click OK.**

   The Designer page opens. (See Figure 1-2.)

   At this point, App Inventor creates a skeletal Android project. You can do a quick reality check by running this project.

5. **On the Designer page, click the Open the Blocks Editor button in the upper-right corner (see Figure 1-2).**

   After you click the Open the Blocks Editor button, your computer downloads and launches the Blocks Editor program from Google's servers.

   The Blocks Editor screen contains lots of interesting thingamajigs, but for this section's minimal app, you can ignore everything except two of the buttons near the top of the screen. (See Figure 1-3.)

6. **Click the New Emulator button. (See the top of the screen in Figure 1-3.)**

   Clicking New Emulator starts the run of an Android virtual device.

7. **Wait for the emulator's startup screen to appear.**

   Android's emulator takes a long time to start running. (On my 2GHz Intel Core™ 2 Duo, the emulator's startup takes minutes, not seconds.) Sometimes, during startup, the emulator stalls, and you have to close the emulator and try launching it again. But after the emulator is fully started, it's usually quite reliable.

8. **In the emulator's screen, do whatever you normally do to unlock a phone or a tablet.**

   With your mouse, slide something from one place on the screen to another. That's usually how it's done.

9. **In the Blocks Editor, click the Connect to Device button. (Again, see the top of the screen in Figure 1-3.)**

   A list of running devices appears in a drop-down list. (See Figure 1-4.)

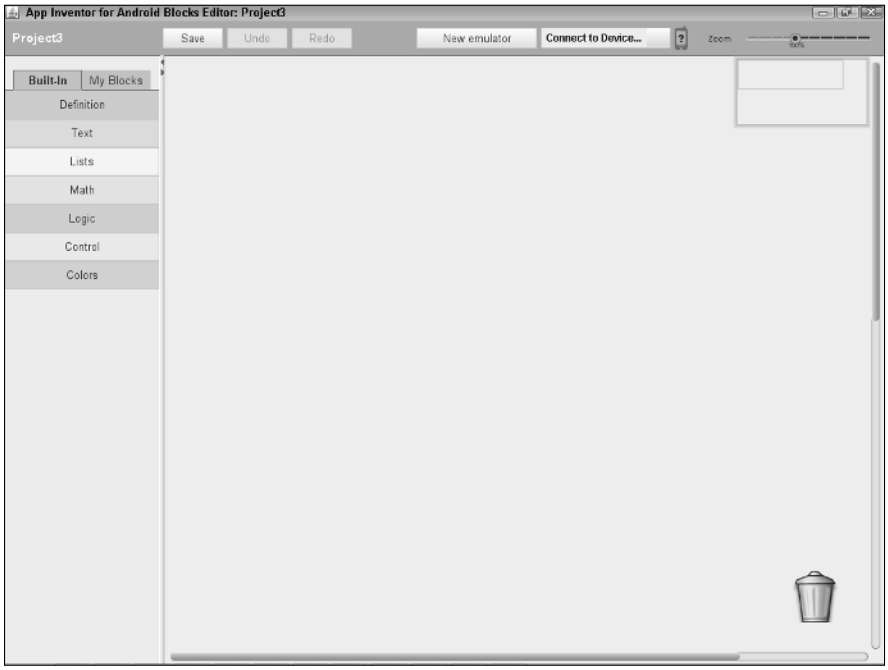

Figure 1-3: The Blocks Editor.

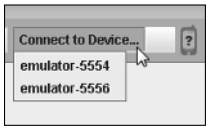

Figure 1-4: The list of running devices (emulators, phones, and tablets).

10. **Choose a device from the drop-down list.**

    In Figure 1-4, the only running devices are emulators.

11. **Wait for your Android application to appear on the emulator screen.**

    Launching an Android emulator requires the patience of a saint.

App Inventor's skeletal application is bland as any application can be. The only items on the emulator's screen are the status bar and the title bar. (See Figure 1-5.)

What's so special about the number 5554?

When you launch an Android device emulator, the new emulator's name is something like emulator-5554 or emulator-5556. The name stems from the fact that each run of an Android emulator uses two *port numbers* (two channels for communicating with the development computer). As you may already know, your web browser normally uses port number 80 to request a web page. Your e-mail program probably uses port 110, port 143, port 585, port 993, or port 995 to retrieve e-mail.

When you launch an emulator on your development computer, you can specify several port numbers for several of the emulator's networking needs. But in most of this book's examples, you start an emulator without explicitly specifying port numbers. When you don't specify port numbers, your emulator relies on default values. If you ever specify a port number other than the default, you do so because you don't want the emulator's communications to conflict with some other program's use of a particular port number. Who knows? Maybe your favorite computer game talks to the web over port 5228, the port number Android uses to obtain apps from the Android Market.

Now imagine that you have no emulators running on your development computer, and you start an emulator without specifying any port numbers. Then the new emulator uses two default port numbers — 5554 and 5555.

- The emulator uses port 5554 to relay its console messages (the text that appears in Eclipse's Console view).
- The emulator uses port 5555 to talk to the Android Debug Bridge (`adb`). For example, when you type **adb install myApp.apk** in your development computer's command window, the Android Debug Bridge installs `myApp.apk` onto your running emulator using port 5555 to handle the communications.

If you type the command **adb devices** in your development computer's command window, you see a list of running emulators. (The list also includes any actual devices that are plugged into your computer via USB or some other fancy connection.) The list probably includes `emulator-5554` because 5554 is the default console port number, and an emulator's name comes from the emulator's console port number (not from emulator's `adb` port number which in this example is 5555).

Time to raise the ante. Imagine that with emulator-5554 running, you go back to your development computer and start a second emulator (again, without explicitly specifying any port numbers). Then Android launches a new emulator with console port 5556 and `adb` port 5557. The `adb` port number is always one more than the console port number. To install `myApp.apk` on the second of the two running emulators, you'd type **adb -s emulator-5556 install myApp.apk** in your development computer's command window. If you close the first emulator, the second emulator's port numbers don't change. So after closing the first of the two emulators, when you type **adb devices**, the list of devices includes emulator-5556 and no longer includes emulator-5554.

The allowable console port numbers for Android emulators are the even numbers from 5554 to 5584 inclusive. So you can simultaneously run emulators named emulator-5554, emulator-5556, emulator-5558, and so on up to emulator-5584. I've never tried to run more than 16 emulators at once, but I'm sure that if I tried, nothing good would come of it.

Figure 1-5: An emulator runs your empty App Inventor project.

Sure, this section's Android app is dull as dirt. But don't slam your laptop's lid shut in frustration! Keep everything running while you step through the next section's instructions.

Using the Designer

The Designer consists of five main panels, plus a few additional menus and buttons. Naturally, each of the main panels has a specific purpose. (For a gander at each of the panels, refer to Figure 1-2.) The five main panels are as follows:

- **The *Designer palette* contains items for you to drag into your Android application's screen.** The items in the Designer palette are called *components.* Some of the components are visible thingies (Android views and such), but other components (such as a LocationSensor) are functional rather than visible.
- **The *Designer viewer* is a preview of your target device's screen.** You drag components from the Designer palette to the Designer viewer.
- **The *Components tree* has a branch for each component that you've dropped into the Designer viewer.** Selections in the Designer viewer and the Components tree stay in sync with one another. That is, when you select a component in the Designer viewer, App Inventor automatically selects the corresponding branch of the Components tree. And when you select a branch of the Components tree, App Inventor automatically selects the corresponding component in the Designer viewer.

The bottom of the Components tree's panel has buttons for renaming and deleting components in the tree.

If you're working on a serious project (instead of goofing around with instructions in this book), rename each component that you drag into the Designer viewer. If you don't do any renaming, App Inventor assigns default names to your components — names such as Button1, Button2, and so on. Later on, when you work with these randomly named components in the Blocks Editor, you'll have trouble remembering each component's role.

- **The *Media panel* displays any images or other media that you've added to your project.** In Figure 1-2, the Media panel is sitting quietly below the Components tree.
- **The *Properties* sheet lists the properties of whatever component you select in the Designer viewer or the Components tree.** You can set a component's properties using the Properties sheet's text fields, drop-downs, and other gizmos.

Adding a component to your project

With the Designer, the Blocks Editor, and the emulator from the previous section still running, try the following trick:

1. **The Designer palette's components are divided into categories, so locate the Basic category.**

   Refer to Figure 1-2. The Basic category is in the upper-left section of the figure.

2. **In the Basic category, locate my favorite component — the Button.**
3. **Drag the Button component from the Designer palette to the Designer viewer.**

   The result is shown in Figure 1-6.

   Now the real fun begins!

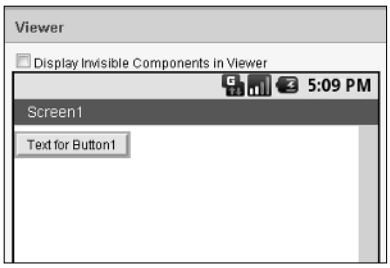

Figure 1-6: A new Button component in the Designer viewer.

4. **Look back at the emulator that you launched in Step 6 of "Creating a Project," earlier in this chapter.**

   Lo and behold! The emulator displays your modified Android app — an app with a button (See Figure 1-7.)

   In Step 9 of "Creating a Project," you forge a connection between App Inventor and an emulator. Later, when you add a button to the Designer viewer, the emulator's screen changes automatically. You don't reload your modified project onto the emulator. The App Inventor reloads for you. Hey! When you connect App Inventor to a device, you *really* connect App Inventor to the device!

5. **Create a checkpoint!**

   The Designer doesn't have an Undo feature. Instead, you create *checkpoints*. When you create a checkpoint, you create a new project containing all the changes you made since the last Save, Save As, or Checkpoint operation. After you create a checkpoint, the Designer continues to display whatever project it displayed before you created the checkpoint.

   The Checkpoint button appears above the Designer viewer in the Designer page. (See Figure 1-2.) Click this button to open the Checkpoint dialog box.

6. **In the Checkpoint dialog box, type a name for your checkpoint and then click OK.**

   The buttons above the Designer viewer have the labels Save, Save As, and Checkpoint:

   - When you click *Save,* you commit the changes you made since the previous Save or Save As operation. The Designer continues to display whatever project you save.
   - When you click *Save As,* you create a new project containing all the changes you made since the previous Save or Save As operation. After the Save As operation, the Designer displays the new project.
   - When you click *Checkpoint,* you create a new project containing all the changes you made since the last Save, Save As, or Checkpoint operation. After you create a checkpoint, the Designer continues to display whatever project it displayed before you created the checkpoint.

     To undo changes that you made to your project, return to the Projects page (by clicking My Projects on the Designer page). From the list on the Projects page, select a project that you created previously. (You created projects using the Designer's Save, Save As, and Checkpoint buttons.)

With App Inventor (as with any other tool that you use), save your work often.

Figure 1-7: The emulator stays in sync with changes in the Designer.

Setting component properties

Using Designer's Properties sheet, you can examine or change all kinds of things about each component that you've placed in the Designer viewer. Try this:

1. **Follow the previous instructions in this chapter.**

   I understand. Asking you to follow all the previous instructions in this chapter is a lot to ask. But believe me, it's worth the effort.

   As a result of following those instructions, you have a Button on the Designer viewer. ("Big deal!" you say. Well, you're on your way to bigger and better things.)

2. **With Screen1 selected in the Components tree, find the ScreenOrientation drop-down in the Properties sheet, and change the selection from Unspecified or Portrait to Landscape.**

   As a result, the Designer viewer's screen orientation changes to Landscape. Congratulations! You've just turned your phone sideways.

3. **With Screen1 still selected, find the BackgroundImage field in the Properties sheet, and select it to reveal a small BackgroundImage dialog box.**

   See Figure 1-8.

Figure 1-8: The BackgroundImage dialog box.

4. **In the BackgroundImage dialog box, click Add.**

   The Upload File dialog box appears. (See Figure 1-9.)

5. **In the Upload File dialog box, click Choose File.**

   As a result, your operating system's Open dialog box appears.

6. **Using your operating system's Open dialog box, look for an image file on your computer's hard drive.**

7. **After choosing an image file, click OK a few times (or whatever you need to click) to back out of the Open dialog box and the Upload File dialog box.**

   As a result, the BackgroundImage field contains the name of your image file, and the Designer viewer's screen displays your image in the background. Figure 1-10 shows the Designer viewer's screen with a background image displaying the Stockton Street Tunnel in San Francisco. I took the photo with my Android phone after attending Google I/O 2011.

Because of the stuff you did in Steps 3–7, your image file appears as an option in the Designer's small Media panel. The Media panel is located below the Components tree.

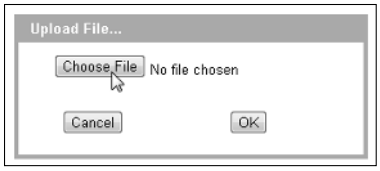

Figure 1-9: The Upload File dialog box.

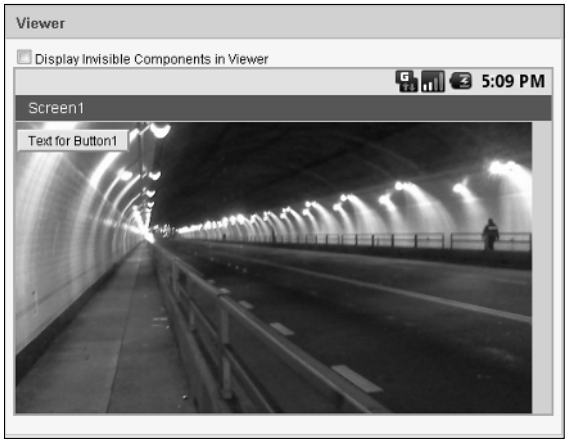

Figure 1-10: There's an image to the Android application's background.

8. **In the Designer viewer (or in the Components tree), select the button that you created previously.**

   In Figure 1-10, the text on the button's face is *Text for Button1*. Select the corresponding button in your own project.

9. **In the Properties sheet, find the Text field, change the words in that field, and then press Enter.**

   In the Designer viewer, the text on the face of the button changes. (See Figure 1-11.)

Figure 1-11: You've changed the text on the face of the button.

Arranging screen elements

The App Inventor doesn't provide all the layout facilities of Android's SDK. For example, the Designer palette has no Relative layout. And in general, the Designer's options aren't as feature-rich as the SDK's layouts. But with the Designer palette's Screen Arrangement components, you can go a long way in customizing the look of your Android app.

It's natural to wonder if you can enjoy the best of both worlds. You can create an Android app in minutes with the App Inventor. Then can you import your app into Eclipse and tweak the code with the Android SDK's high-precision tools? The answer (as of the day I write this sentence) is no. Google doesn't provide a way to translate App Inventor code into ordinary Android SDK code. I've tried one or two third-party translation tools, but none of these tools is reliable. (And for all I know, these tools might not be legal.)

I want to keep this chapter's ongoing example from becoming cluttered. So the next set of instructions starts without the changes from the "Setting component properties" section.

If you followed Steps 5 and 6 in the "Adding a component to your project" section, you can return to your checkpoint. In doing so, you return to the project as it was before changing orientation, adding a background image, and setting the button's text. But if you didn't create a checkpoint (or if you

didn't do anything beyond starting a new project), don't worry. The steps in this section don't build on the steps from previous sections. (Besides, the apps that you create in this chapter are practice apps. You can experiment all you want and not hurt anything.)

For details about using checkpoints, see the "Adding a component to your project" section, earlier in this chapter.

To arrange components on the Designer viewer screen, do the following:

1. **In the Designer palette's Screen Arrangement list, find the VerticalArrangement.**

   See Figure 1-12.

2. **Drag a VerticalArrangement from the palette to the Designer viewer's screen.**

   An empty rectangle appears in the Designer viewer screen. (See Figure 1-13.)

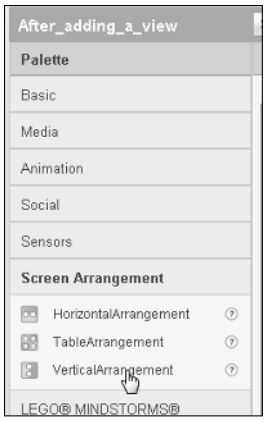

Figure 1-12: The Designer palette's Vertical-Arrangement component.

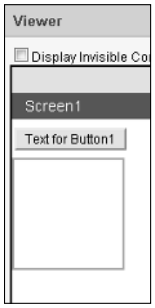

Figure 1-13: The screen contains an empty Vertical-Arrangement.

3. **If you have a button on your Designer viewer's screen, drag that button into the VerticalArrangement square; if you don't already have a button, drag a new button from the Designer palette into the VerticalArrangement square.**

   The VerticalArrangement square shrinks (and maybe moves) to fully enclose the button.

4. **Drag a second button from the Designer palette into the VerticalArrangement square.**

   The VerticalArrangement square grows to accommodate the additional button. The buttons inside the arrangement appear one above the other because this arrangement is a VerticalArrangement. (See Figure 1-14.)

   Look at the quick-and-dirty layout in Figure 1-14. An Android app looks so crude if its buttons are tucked in the upper-left corner! Unlike Android's SDK, the App Inventor doesn't let you change a view's gravity. So to center the components in your application's screen, you need a hack. I describe this hack in the next several steps.

5. **In the Designer viewer or the Component tree, select the VerticalArrangement.**

6. **In the Properties sheet, select the Width field.**

   A small Width dialog box appears. (See Figure 1-15.)

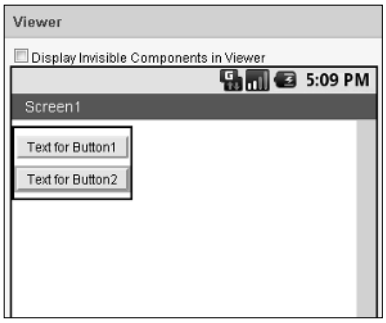

Figure 1-14: Two components inside a Vertical-Arrangement.

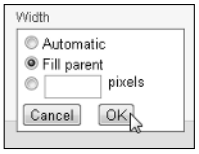

Figure 1-15: The Width dialog box.

7. **In the Width dialog box, select the Fill Parent option.**

   Again, see Figure 1-15.

8. **In the Width dialog box, click OK.**

   As a result, the VerticalArrangement stretches across the entire Designer viewer screen. (See Figure 1-16.)

   So far, so good. But here comes the embarrassing part of the layout hack.

9. **Drag two HorizontalArrangement components from the Designer palette into the VerticalArrangement.**

10. **Drag the buttons on the Designer viewer screen into the HorizontalArrangement components.**

    That is, drag one button into one HorizontalArrangement, and drag the other button into the other HorizontalArrangement. (See Figure 1-17.)

11. **Drag four new labels from the palette into the HorizontalArrangement components, and surround each button with two of the labels. (See Figure 1-18.)**

12. **Set each label's Width property to the Fill Parent option.**

    For details on setting a component's Width property, see Steps 5–8.

13. **Set both HorizontalArrangement components' properties to (yes) the Fill Parent option.**

    At this point, everything stretches across the entire Designer viewer screen. The only sore spot is the text in each of the labels.

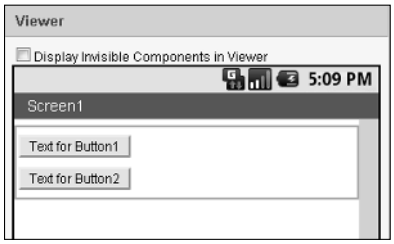

Figure 1-16: The Vertical-Arrangement fills its parent.

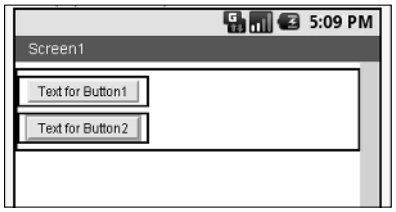

Figure 1-17: A button in each Horizontal-Arrangement.

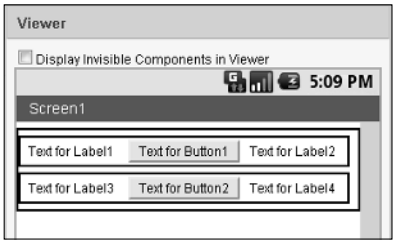

Figure 1-18: Each button is surrounded by labels.

14. **Delete the text in each of the labels, leaving behind just your button text.**

    For help setting a component's text, see the "Setting component properties" section.

App Inventor doesn't treat strings the way Java treats strings. In App Inventor, the notation `" "` doesn't stand for the empty string. Instead, `" "` stands for the two-character string containing two double quotation marks. To put an empty string in one of App Inventor's label components, go to the Properties sheet and delete all characters in the component's Text field.

In the Designer viewer, the final result appears in Figure 1-19. The (beautiful) display that you see in an emulator is pictured in Figure 1-20.

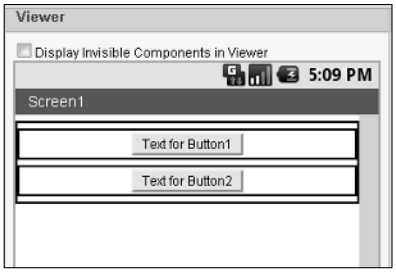

Figure 1-19: At last! The buttons are centered.

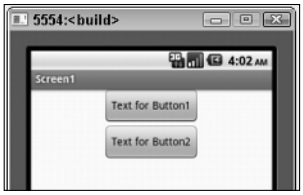

Figure 1-20: Nice!

Using the Blocks Editor

Google's App Inventor has two major parts — a Designer and a Blocks Editor. The Blocks Editor has two big panels. These panels don't seem to have names to speak of. So for the sake of clarity, I hereby christen these panels the *Blocks palette* and the *Blocks viewer.* (In some tutorials, I've seen the Blocks viewer called the *editor,* but that terminology confuses me.)

The Blocks Editor has a palette and a viewer, and the Designer has a palette and a viewer (along with other panels). For the Designer's panels, I use the names *Designer palette* and *Designer viewer;* for the Blocks Editor's panels, I use the names *Blocks palette* and *Blocks viewer.* My terminology isn't standard. But as far as I'm concerned, the standard terminology isn't very helpful. In fact, in the official App Inventor documentation, many parts of the interface are unnamed.

You can see the Blocks palette and the Blocks viewer in Figure 1-3.

- **The *Blocks palette* on the left side contains building blocks for actions — the actions to be performed by your Android application.**

  The Blocks palette has two tabs — Built-In and My Blocks:

  - Most items on the *My Blocks tab* represent the components that you've added to the Designer's viewer — buttons, labels, screen arrangements, and other such things.
  - The items on the *Built-In tab* are fundamental elements of programming logic. This includes text strings, mathematical operators, comparison operators, loops, if-then constructs, and other stuff.

- **The *Blocks viewer* is a preview of your application's actions.** The Blocks viewer, initially empty, consumes most of the Blocks Editor's area. As you might already have guessed, you drag items from the Blocks palette to the Blocks viewer.

The programming model for App Inventor's Blocks Editor is the Scratch development environment. Visit `http://scratch.mit.edu` for details.

Adding event handlers

In previous sections, I describe the kinds of things that most visual editors can do. In fact, Eclipse's Graphical Layout has the same kinds of drag-and-drop facilities as App Inventor's Designer, and the Graphical Layout has an added advantage. With Eclipse's Graphical Layout, you have access to most of the Android SDK features.

So what makes the App Inventor different? Why use the App Inventor instead of Eclipse and the Android SDK? With the App Inventor's Blocks Editor, you can create Android application logic by dragging and dropping things and by fitting things together. Absolutely no coding required!

Okay, what's an advantage in one setting is a disadvantage in another. Along with the Blocks Editor's "no coding required" feature comes the "no coding allowed" limitation. Anyway, App Inventor isn't a magic bullet. To create elaborate Android applications, you still need an IDE, such as Eclipse, and the full Android SDK.

In this section, you experiment with some Blocks Editor techniques:

1. **On the App Inventor's Projects page, create a new project.**

   For details, see the section "Creating a Project," earlier in the chapter.

2. **Drag a button from the Designer palette and drop it onto the Designer viewer.**

   For details, see the section "Adding a component to your project" — again, earlier in the chapter.

3. **Drag a label from the Designer palette and drop it onto the Designer viewer.**

   Now your Application has a button (named Button1) and a label (named Label1).

4. **Open the Blocks Editor.**

   For details, see the "Creating a Project" section. (You know where it is.)

5. **In the Blocks palette, click the My Blocks tab. (See Figure 1-21.)**

Figure 1-21: The My Blocks tab in the Blocks palette.

6. **In the My Blocks tab, click Button1. (See Figure 1-22.)**

Here's where some of my earlier advice shows its real value. Previously in this chapter, I advise you to rename each component that you drag into the Designer viewer. Oddly enough, if you don't follow my advice, life is easier for me. In Step 6, you click Button1, and sure enough, your application has something with the default name *Button1.* But in practice, a Blocks palette with names such as *Button1, Label1, Button2,* and *Button3* is very confusing. Which of your buttons is *Button2?* Is it the Send Mail button or the Receive Mail button? Instead of living with the default *Button2* name, change the name to *SendMail,* or *PlaySong,* or whatever name reminds you of the button's purpose.

Anyway, in this set of steps, you've selected Button1. As a result, the expanded Blocks palette contains a bunch of puzzle pieces. Each puzzle piece (each block) has something to do with Button1. (See Figure 1-22.) Now what?

7. **Click the Button1.Click block.**

(If you read the fine print, the block's label is actually When Button1. Click Do.)

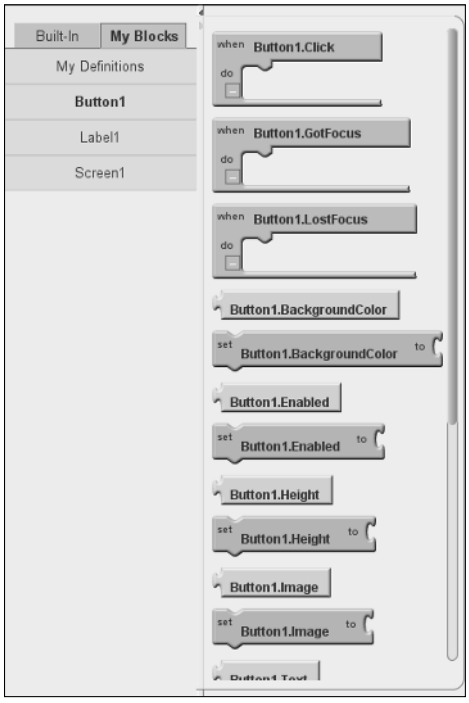

Figure 1-22: The Blocks palette expands when you select Button1.

When you click the Button1.Click block, the other blocks on the expanded palette disappear. The Button1.Click block stands on its own in the Blocks viewer. (See Figure 1-23.)

Everybody does it. At one time or another, you'll reach into the expanded palette and click the wrong block. No problem. Simply drag the unwanted block to the trash can in the Blocks viewer's lower-right corner. (Refer to Figure 1-3.)

The Button1.Click block is an example of an *event handler:*

- An *event* is something that your application might respond to, such as a button click, the pressing of a key, a change in GPS location, or the arrival of a text message.
- An *event handler* (such as the Button1.Click block) is the part of your app that responds to the occurrence of an event.
- When an event occurs, your device automatically invokes the instructions contained in the event handler.

In this example, an event occurs when the user clicks Button1. In the Block1.Click event handler, you'll add instructions to put text in the application's other component (the Label1 component). (In fact, you'll spend the remainder of this step list doing precisely that.)

8. **In the Blocks palette, click Label1.**

Now the expanded Blocks palette contains a bunch of Label1 puzzle pieces. Notice that some of the pieces (the blocks) come in pairs. The expanded palette has a Label1.Text block and a Set Label1.Text To block.

- The *Label1.Text block* is a *getter.*
- The *Set Label1.Text To* block is a *setter.*

A getter gives you access to an existing value. A setter changes an existing value (or sets the value for the first time). Taken together, a getter and a setter allow you to examine and change the value of a component's property.

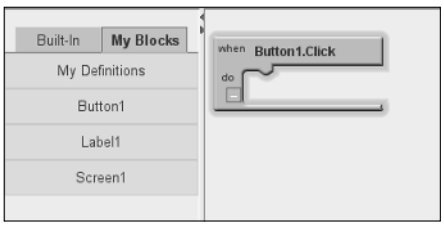

Figure 1-23: The Button1.Click block in the Blocks viewer.

9. **Click the Label1.Text setter block.**

   The Label1.Text setter sits along with the Button1.Click event handler in the Blocks viewer.

   It's time for some jigsaw-puzzle fun! Like two drifting tectonic plates, the two pieces in the Blocks viewer look as if they belong together. The Labe1.Text setter can fit snugly inside the Button1.Click event handler.

10. **Drag-and-drop the Labe1.Text setter block into the gap of the Button1.Click event handler block (see Figure 1-24).**

    To be painfully precise, the name for one of these gaps is actually a *socket*.

    If you drop the label block in the right place, your computer speaker plays a snap sound, and the combination of blocks changes in appearance just a bit. These responses indicate that you've successfully associated one block with the other. In terms of programming logic, you've said, "When the user clicks Button1, set Label1's text to . . .", and you haven't yet specified Label1's new text.

    Near the top of the Button1.Click block, App Inventor displays an exclamation point inside a little box. If you hover over the box, a popup bubble says, `Warning: This clump contains an empty socket and won't be sent to the phone.` That *empty socket* is the gap in the To part of the Label1.Text setter block. The effortless emulator update that I describe in the "Adding a component to your project" section, earlier in this chapter, can't take place.

11. **Click the Built-In tab of the Blocks palette.**

12. **In the Built-In tab, click the Text item.**

    A bunch of text-related blocks appears in the expanded Blocks palette. (See Figure 1-25.)

13. **In the expanded Blocks palette, click the Text item.**

    As a puzzle piece, this item has only one part that connects to another piece. That part is a little knob that fits nicely into the Label1.Text setter block's empty socket.

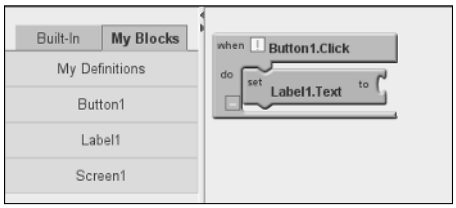

Figure 1-24: The label block snaps into the button block.

14. **Drag and drop the Text block into the gap of the Label1.Text setter block (see Figure 1-26).**

    You're instructing your app to set the Label1's text to something-or-other. The only remaining work is to specify what that something-or-other is.

15. **Click the bold *Text* word in the newly dropped Text piece.**

    The word changes appearance, like the branch whose file you're renaming on an Explorer or Finder tree.

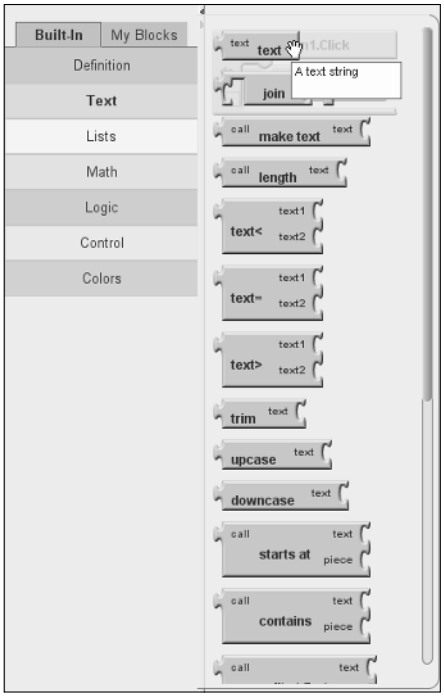

Figure 1-25: The expanded Blocks palette displays blocks related to text.

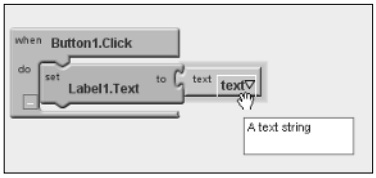

Figure 1-26: A clump of blocks with no empty socket.

16. **Type a few characters in place of the word *Text* and then press Enter (see Figure 1-27).**

17. **If necessary, start an emulator and connect App Inventor to the emulator.**

    For details, see the earlier section "Creating a Project."

    Wait for the emulator to receive the newest changes to your Android application. And then . . .

18. **Click the button on the emulator's screen.**

    The emulator responds by changing the label's text. (See Figure 1-28.)

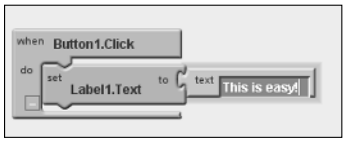

Figure 1-27: The block grows to hold a longer string of characters.

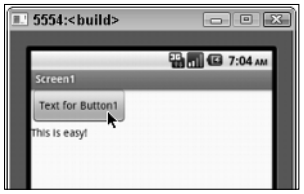

Figure 1-28: Voilà!

Event handlers with parameters

The preceding section's event handler is very stubborn. Whenever you click the button, you get the same message: *This is easy!* If you click again, you see the same message again. This persistent behavior is a result of the work shown in Figure 1-27. In that figure, you type **This is easy!** on a block.

This section's event handler isn't nearly so stubborn. In this section, you do something like the stuff shown in Figure 1-27. But instead of typing **This is easy!** or some other specific text, you add a parameter to the Label1.Text setter block. A *parameter* is a placeholder for any text that the user types. Here's how it works:

1. **On the App Inventor's Projects page, create a new project.**

   For details see the "Creating a Project" section in this chapter.

2. **Drag a button from the Designer palette and drop it onto the Designer viewer.**

3. **Drag a label from the Designer palette and drop it onto the Designer viewer.**

   Now your Application has a button (named Button1) and a label (named Label1).

4. **Drag a Notifier component from the Other Stuff category of the Designer palette and then drop the Notifier onto the Designer viewer.**

   A Notifier is initially invisible. (It's invisible until something interesting happens — something worth notifying the user about.) So you don't see your new Notifier in the Designer viewer's screen. But in the Components tree, you see a branch labeled Notifier1. And if you're lucky, you might peek below the Designer viewer's screen and find an icon representing Notifier1.

5. **Open the Blocks Editor.**

6. **Create the group shown in Figure 1-29.**

   For help grouping blocks, see the preceding section.

   The blocks in Figure 1-29 instruct your app to display a text dialog box. A text dialog box gets text from the user. (See Figure 1-30.)

   In the next few steps, you create a second group of blocks.

7. **In the Blocks palette's My Blocks tab, click the Notifier1 item.**

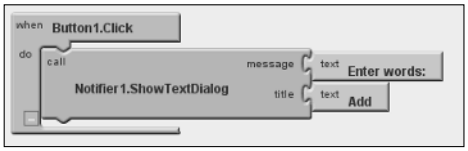

Figure 1-29: When the user clicks the button, the notifier displays a dialog box.

Figure 1-30: A text dialog box.

8. **In the expanded Blocks palette, select the Notifier1.AfterTextInput event handler block.**

   When you make the selection, App Inventor adds two pieces to your Blocks viewer. (Yes, it adds *two* pieces.) Along with the event handler block, App Inventor adds a block displaying Name and Response. (See Figure 1-31.)

   The extra Name Response block is a *parameter,* or placeholder. A user enters **OK, I'm typing** into the field in Figure 1-30. Then, when the user clicks OK, the user's text (*OK, I'm typing,* or whatever else the user entered) has a name. That text's name is Response.

   Because the user's input text has a name, you can use that name in the rest of the handler block. You use the name to help describe an action — the action to be taken in response to the `AfterTextInput` event.

   Why does the text *OK, I'm typing* need a name? The user's input text needs a name because you can't assume that the user always enters **OK, I'm typing** in the field of Figure 1-30. Somehow, you have to instruct the app to do something with whatever the user types. In Figure 1-31, the name for whatever the user types is Response.

   Whatever text the user enters in Figure 1-30 goes by Response. That's fine, but how do you use Response? Here's how:

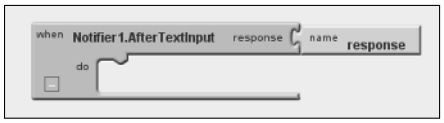

Figure 1-31: Two blocks for the price of one.

9. **In the My Blocks tab, click the My Definitions category.**

   A Value Response block appears (as if by magic) in the extended Blocks palette.

10. **Click the Value Response block, but for now, don't attach the block to anything else in the Blocks viewer.**

    You make use of the parameter Response by plugging this Value Response block into a socket. But first, I want to show you another trick or two.

11. **Add blocks to the Notifier1.AfterTextInput event handler block to form the incomplete group shown in Figure 1-32.**

    The Join block in Figure 1-32 comes from the Text category in the Blocks palette's Built-In tab. When you *join* two pieces of text, you turn the two pieces into one combined piece. (That is, you *concatenate* the two pieces.)

    In this example, you plan to join three pieces of text. So in the next step, you join stuff to an existing Join block.

12. **Add another block to the Notifier1.AfterTextInput handler to form the incomplete group shown in Figure 1-33.**

Figure 1-32: An incomplete (but interesting) group of blocks.

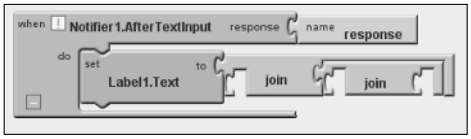

Figure 1-33: A Join block within a Join block.

13. **Fill the empty sockets in the Notifier1.AfterTextInput block as follows:**

    - *Into the leftmost socket — the remaining empty socket of your first Join block — put the Label1.Text getter block.*
    - *Into the middle socket — the first empty socket on your second Join block — put a Text block containing a backslash (\) followed by a lowercase letter n.*

      In many modern programming languages, `\n` stands for an instruction to "go to the next line."
    - *Into the rightmost socket — the remaining empty socket on your second Join block — put the Value Response block that you selected in Step 10.*

    The resulting group of blocks is shown in Figure 1-34.

    When the user finishes typing text, the combination of blocks in Figure 1-34 instructs your app to do the following:

    a. Get whatever text is already on Label1.

    b. Join a line break onto that text.

    c. Join the user's response onto that bundle of text.

    d. Put the whole bunch of joined text back into Label1.

    Now you have two groups of blocks — a group to handle button clicks (in Figure 1-29) and a group to handle text input (in Figure 1-34). You can connect to an emulator and try the app. Some screen shots from a run of the app appear in Figure 1-35.

This chapter covers the general concepts behind App Inventor. Chapter 2 of this minibook describes specific App Inventor projects.

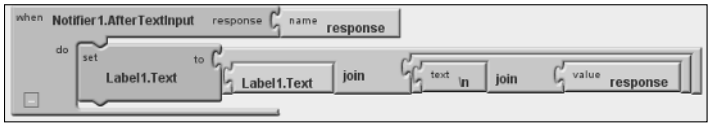

Figure 1-34: At last! A complete event handler!

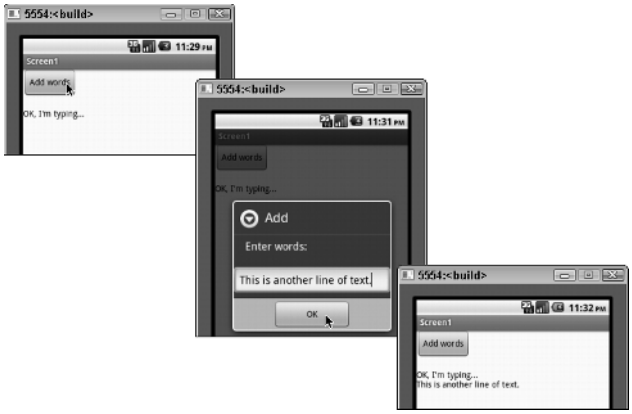

Figure 1-35: Each encounter with the text dialog box adds a line of text to the label.