CHAPTER 14

Understanding an App's Architecture

This chapter examines the structure of an app from a programmer's perspective. It begins with the traditional analogy that an app is like a recipe and then proceeds to reconceptualize an app as a set of components that respond to events. The chapter also examines how apps can ask questions, repeat, remember, and talk to the Web, all of which will be described in more detail in later chapters.

Figure 14-1.

Many people can tell you what an app is from a user's perspective, but understanding what it is from a programmer's perspective is more complicated. Apps have an internal structure that you must understand in order to create them effectively.

One way to describe an app's internals is to break it into two parts, its *components* and its *behaviors*. Roughly, these correspond to the two main windows you use in App Inventor: you use the Component Designer to specify the objects (components) of the app, and you use the Blocks Editor to program how the app responds to the user and external events (the app's behavior).

Figure 14-1 provides an overview of this app architecture. In this chapter, we'll explore this architecture in detail.

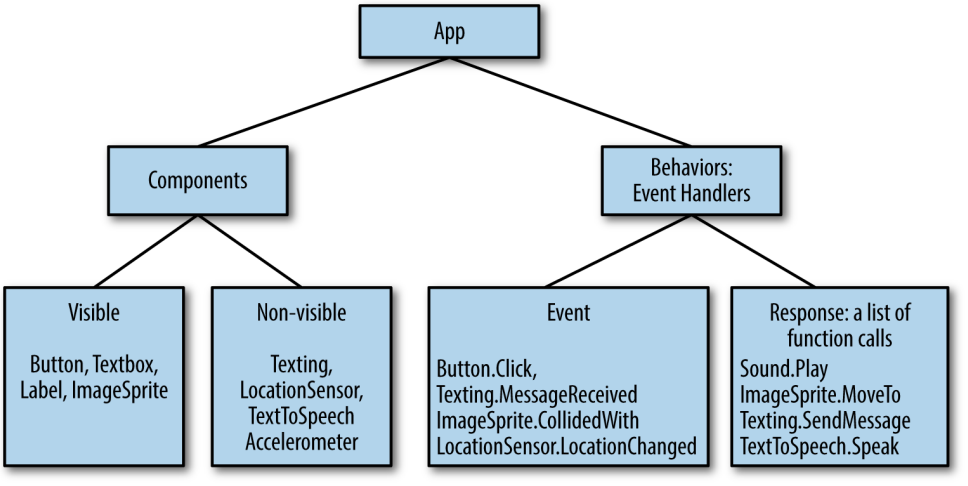


Figure 14-2. The internal architecture of an App Inventor app

Components

There are two main types of components in an app: visible and non-visible. The app's visible components are those that you can see when the app is launched—buttons, text boxes, and labels. These are often referred to as the app's *user interface*.

Non-visible components are those that you can't see, so they're not part of the user interface. Instead, they provide access to the built-in functionality of the device; for example, the `Texting` component sends and processes SMS texts, the `LocationSensor` component determines the device's location, and the `TextToSpeech` component talks. The non-visible components are the technology within the device—little worker bees that do jobs for your app.

Both visible and non-visible components are defined by a set of *properties*. Properties are memory slots for storing information about the component. Visible components like buttons and labels have properties such as `Width`, `Height`, and `Alignment`, which together define how the component looks.

Component properties are like spreadsheet cells: you modify them in the Component Designer to define the *initial* appearance of a component. You can also change the values with blocks.

Behavior

App components are generally straightforward and easy to understand: a text box is for entering information, a button is for clicking, and so on. An app's behavior, on the other hand, is conceptually difficult and often complex. The behavior defines how the

app should respond to events, both user initiated (e.g., a button click) and external (e.g., an SMS text arriving to the phone). The difficulty of specifying such interactive behavior is why programming is so challenging.

Fortunately, App Inventor provides a high-level blocks-based language for specifying behaviors. The blocks make programming behaviors more like plugging puzzle pieces together, as opposed to traditional text-based programming languages, which involve learning and typing vast amounts of code. And App Inventor is designed to make specifying event-response behaviors especially easy. The following sections provide a model for understanding app behavior and how to specify it in App Inventor.

An App as a Recipe

Traditionally, software has often been compared to a recipe. Like a recipe, a traditional app follows a linear sequence of instructions that the computer should perform, such as illustrated in *Figure 14-2*.

A typical app might start a bank transaction (A), perform some computations and modify a customer's account (B), and then print out the new balance on the screen (C).

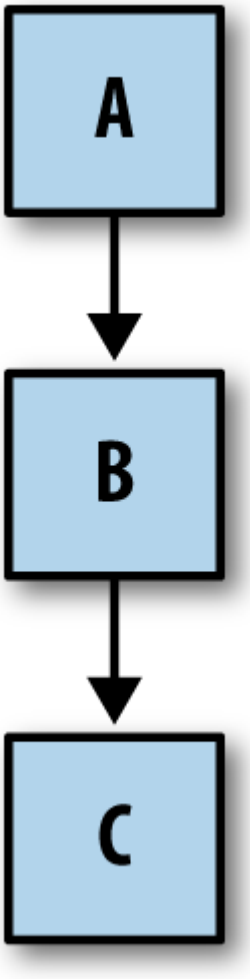


Figure 14-3. Traditional software follows a linear sequence of instructions

An App as a Set of Event Handlers

The *app as a recipe* paradigm fit the early number-crunching computer well, but its not a great fit for mobile phones, the Web, and in general most of the computing done today. Most modern software doesn't perform a bunch of instructions in a predetermined order; instead, it *reacts* to *events*—most commonly, events initiated by the app's end user. For example, if the user taps a button, the app responds by performing some operation (e.g., sending a text message). For touchscreen phones and devices, the act of dragging your finger across the screen is another event. The

app might respond to that event by drawing a line from the point at which your finger first contacts the screen to the point where you lifted it.

Modern apps are better conceptualized as event-response machines. The apps do include recipes—sequences of instructions—but each recipe is only performed in response to some event, as shown in *Figure 14-3*.

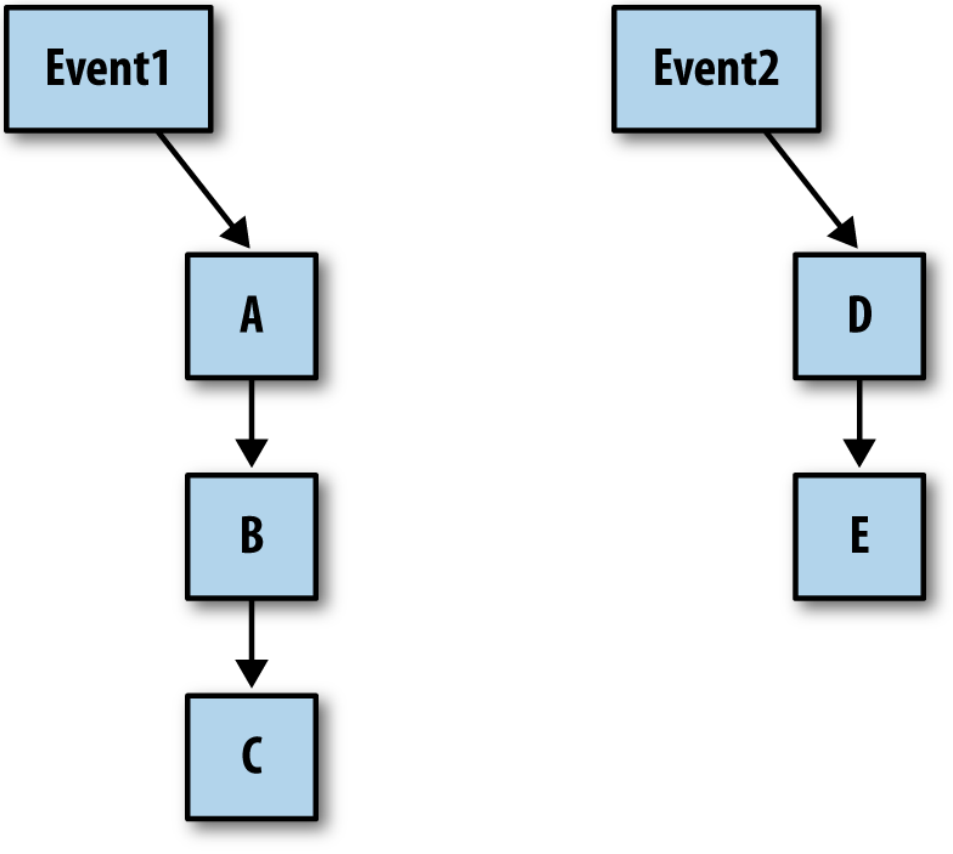


Figure 14-4. An app as multiple recipes hooked to events

As events occur, the app reacts by calling a sequence of *functions*. Functions are things you can do to, or with, a component; these can be operations such as sending an SMS text, or property-changing operations such as changing the text in a label of the user interface. To *call* or *invoke* a function means to carry out the function—to make it happen. We call an event and the set of functions performed in response to it an *event handler*.

Many events are initiated by the end user, but some are not. An app can react to events that happen within the phone, such as changes to its orientation sensor and the clock (i.e., the passing of time), or it can respond to events that originate outside

the phone, such as an incoming text or call from another phone, or data arriving from the Web (see *Figure 14-4*).

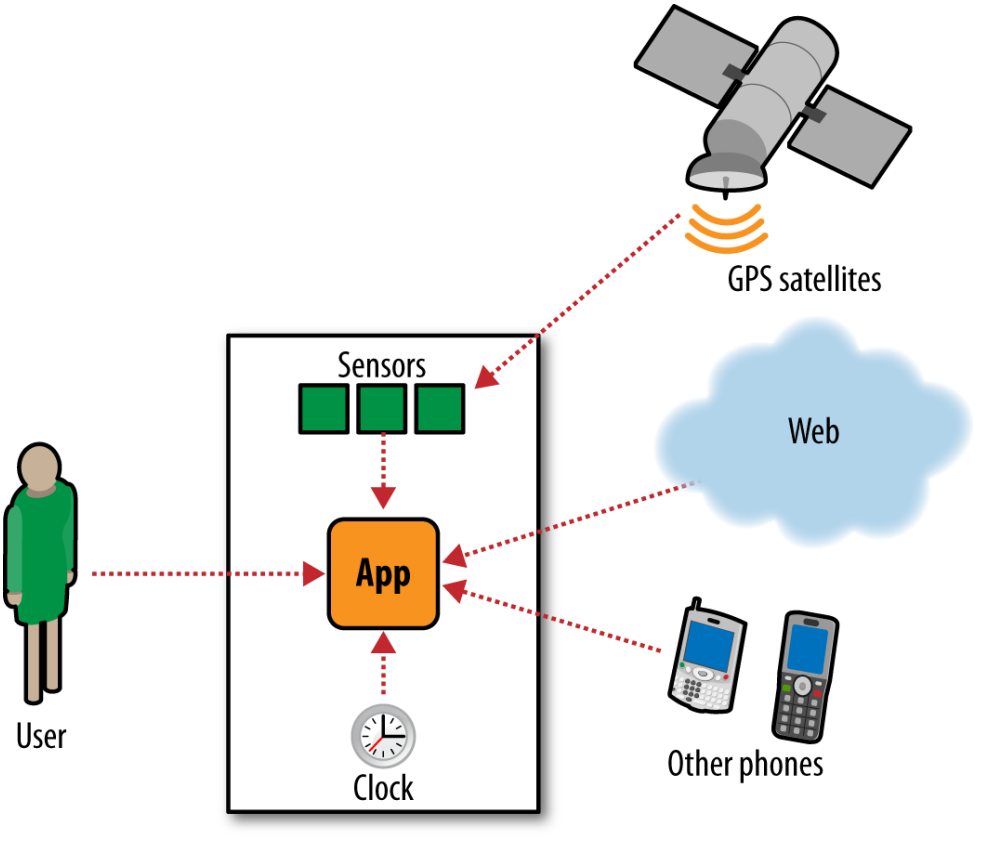


Figure 14-5. An app can respond to both internal and external events

One reason why App Inventor programming is so intuitive is that it's based directly on this event-response paradigm; event handlers are primitives in the language (in many languages, this is not the case). You begin defining a behavior by dragging out an *event block*, which has the form, "When <*event*> do." For example, consider an app, SpeakIt, that responds to button clicks by speaking aloud the text the user has typed in a textbox. This application could be programmed with a single event handler, as demonstrated in *Figure 14-5*.

Figure 14-6. An event handler for a SpeakIt app

These blocks specify that when the user clicks the button named `SpeakItButton`, the `TextToSpeech` component should speak the words the user typed in the text box named `TextBox1`. The response is the call to the function `TextToSpeech1.Speak`. The event is `SpeakItButton.Click`. The event handler includes all the blocks in *Figure 14-5*.

With App Inventor, all activity occurs in response to an event. Your app shouldn't contain blocks outside of an event's `when do` block. For instance, the blocks in *Figure 14-6* accomplish nothing when floating alone.

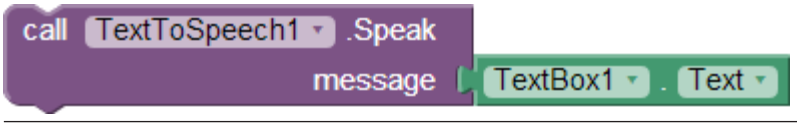


Figure 14-7. Floating blocks won't do anything outside an event handler

Event Types

The events that can trigger activity fall into the categories listed in *Table 14-1*.

Table 14-1. Events that can trigger activity

Event type	Example
User-initiated event	*When* the user clicks **`button1`**, *do*...
Initialization event	*When* the app launches, *do*...
Timer event	*When* 20 milliseconds passes, *do*...
Animation event	When two objects collide, *do*...
External event	*When* the phone receives a text, *do*...

USER-INITIATED EVENTS

User-initiated events are the most common type of event. With input forms, it is typically the user tapping a button that triggers a response from the app. More graphical apps respond to touches and drags.

INITIALIZATION EVENTS

Sometimes, your app needs to perform certain functions immediately upon startup, not in response to any end-user activity or other event. How does this fit into the event-handling paradigm?

Event-handling languages such as App Inventor consider the app's launch as an event. If you want specific functions to be performed as the app opens, you drag out a `Screen1.Initialize` event block and place the pertinent function call blocks within it.

For instance, in the game MoleMash (*Chapter 3*), the `MoveMole` procedure is called upon app startup (see *Figure 14-7*) to randomly place the mole.

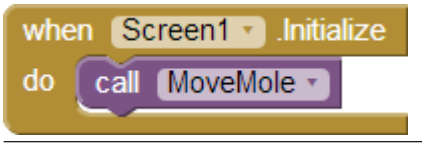


Figure 14-8. Using a Screen1.Initialize event block to move the mole when the app launches

TIMER EVENTS

Some activity in an app is triggered by the passing of time. You can think of an animation as an object that moves when triggered by a *timer event*. App Inventor has a `Clock` component that you can use to trigger timer events. For instance, if you wanted a ball on the screen to move 10 pixels horizontally at a set time interval, your blocks would look like *Figure 14-8*.

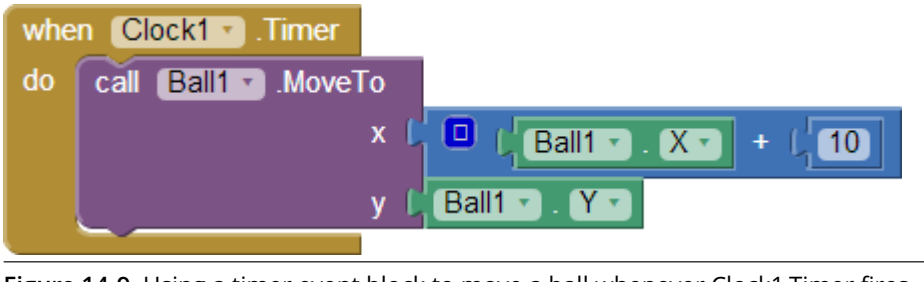


Figure 14-9. Using a timer event block to move a ball whenever Clock1.Timer fires

ANIMATION EVENTS

Activity involving graphical objects (sprites) within canvases will trigger events. So you can program games and other interactive animations by specifying what should occur on events such as an object reaching the edge of the canvas or two objects colliding, as depicted in *Figure 14-9*. For more information, see *Chapter 17*.

Figure 14-10. When the FlyingSaucer sprite hits another object, play a sound

EXTERNAL EVENTS

When your phone receives location information from GPS satellites, an event is triggered. Likewise, when your phone receives a text, an event is triggered (*Figure 14-10*).

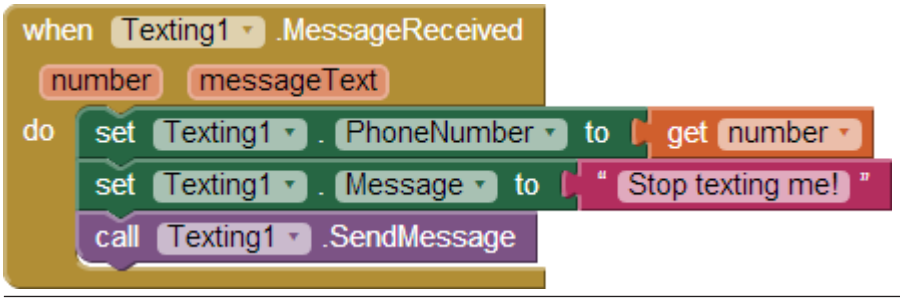


Figure 14-11. The Texting1.MessageReceived event is triggered whenever a text is received

Such external inputs to the device are considered events, no different than the user clicking a button.

Thus, every app you create will be a set of event handlers: one to initialize things, some to respond to the end user's input, some triggered by time, and some triggered by external events. Your job is to conceptualize your app in this way and then design the response to each event handler.

Event Handlers Can Ask Questions

The responses to events are not always linear recipes; they can ask questions and repeat operations. "Asking questions" means to query the data the app has stored and determine its course (branch) based on the answers. We say that such apps have *conditional branches*. *Figure 14-11* illustrates just such a branch.

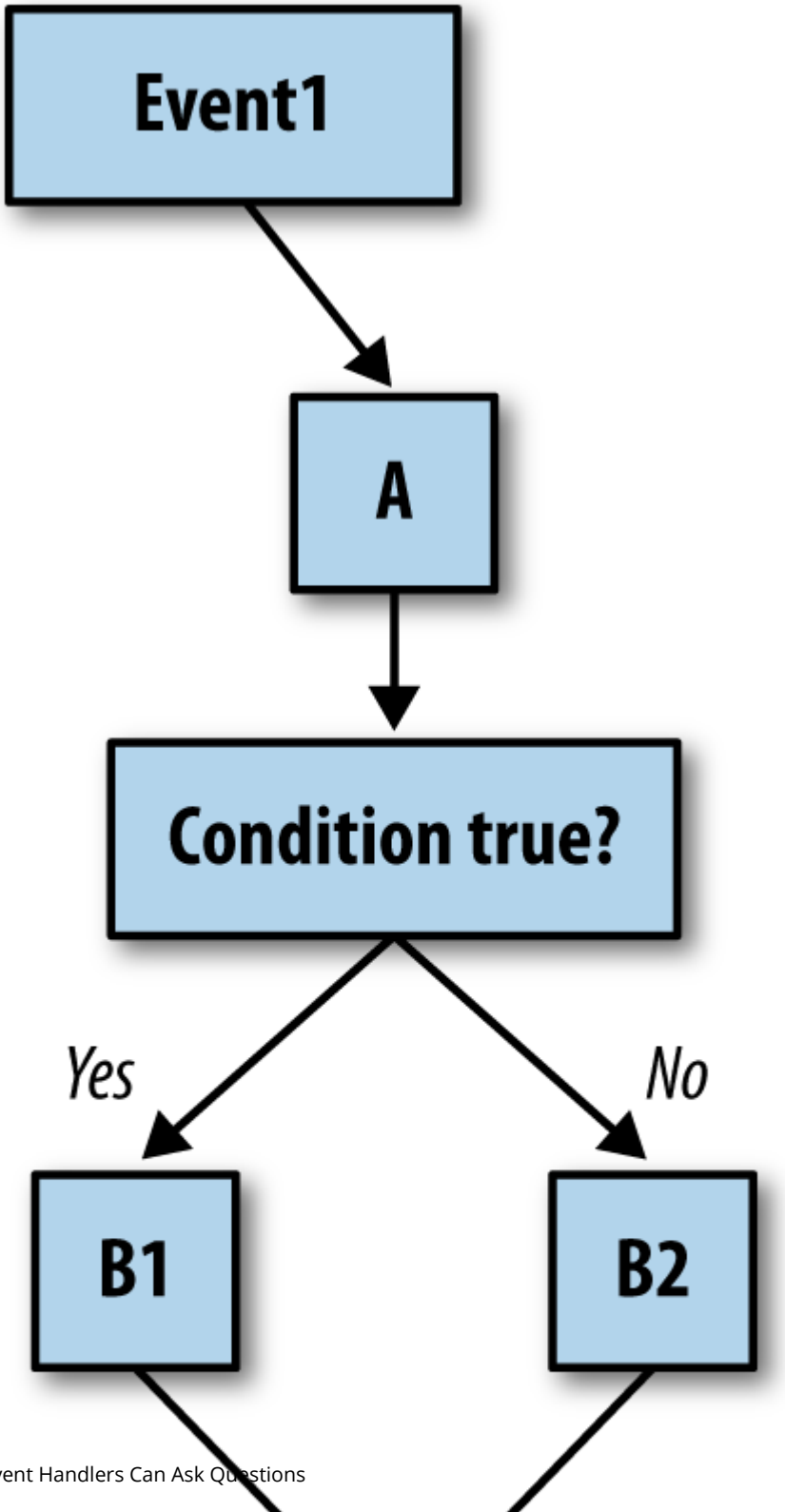
In the diagram, when the event occurs, the app performs operation A and then checks a condition. Function B1 is performed if the condition is true. If the condition is

false, the app instead performs B2. In either case, the app continues on to perform function C.

Conditional tests are questions such as "Has the score reached 100?" or "Did the text I just received come from Joe?" Tests can also be more complex formulas including multiple relational operators (less than, greater than, equal to) and logical operators (and, or, not).

You specify conditional behaviors in App Inventor by using the `if` and `if else` blocks. For instance, the block in *Figure 14-12* would report "You Win!" if the player scored 100 points.

Conditional blocks are discussed in detail in *Chapter 18*.



Event Handlers Can Ask Questions

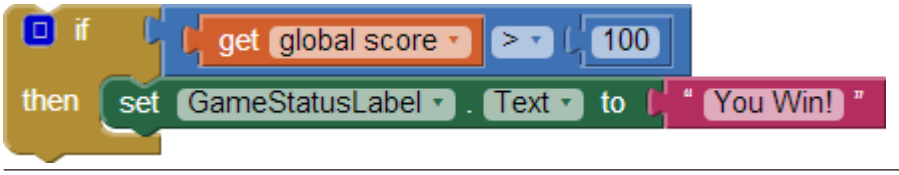


Figure 14-13. Using an if block to report a win when the player reaches 100 points

Event Handlers Can Repeat Blocks

In addition to asking questions and branching based on the answer, a response to an event can also repeat operations multiple times. App Inventor provides a number of blocks for repeating, including the `for each` and the `while do`. Both enclose other blocks. All the blocks within `for each` are performed once for each item in a list. For instance, if you wanted to text the same message to a list of phone numbers, you could use the blocks in *Figure 14-13*.

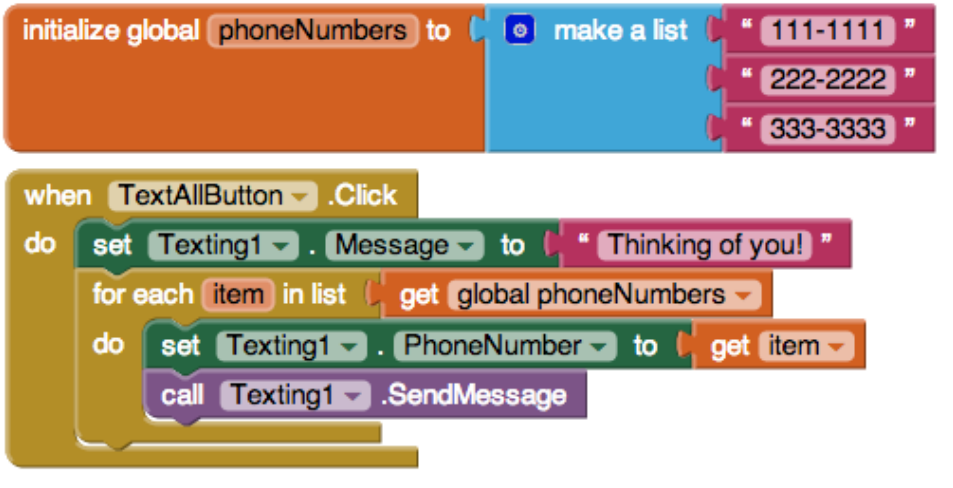


Figure 14-14. The blocks within the for each block are repeated for each item in the list PhoneNumbers

The blocks within the `for each` block are repeated—in this case, three times, because the list `PhoneNumbers` has three items. In this example, the message "Thinking of you..." is sent to all three numbers. Repeating blocks are discussed in detail in *Chapter 20*.

Event Handlers Can Remember Things

Because an event handler executes blocks, it often needs to keep track of information. Information can be stored in memory slots called *variables*, which you define in the Blocks Editor. Variables are like component properties, but they're not associated with any particular component. In a game app, for example, you can define a variable called *score*, and your event handlers would modify its value when the user does something accordingly. Variables store data temporarily while an app is running; when you close the app, the data is lost and no longer available.

Sometimes, your app needs to remember things not just while it runs, but when it is closed and then reopened. If you tracked a high score for the history of a game, for example, you'd need to store this data so that it is available the next time someone plays the game. Data that is retained even after an app is closed is called *persistent data*, and it's stored in some type of a database.

We'll explore the use of both short-term memory (variables) and long-term memory (database data) in *Chapter 16* and *Chapter 22*, respectively.

Event Handlers Can Interact with the Web

Some apps use only the information within the phone or device. But many apps communicate with the Web, either by displaying a web page within the app, or by sending requests to *web service APIs* (application programming interfaces). Such apps are said to be "web-enabled."

Twitter is an example of a web service with which an App Inventor app can talk. You can write apps that request and display your friends' previous tweets and also update your Twitter status. Apps that talk to more than one web service are called *mashups*. We'll explore web-enabled apps in *Chapter 24*.

Summary

An app creator must view his app both from an end-user perspective and from the inside-out perspective of a programmer. With App Inventor, you design how an app looks and you design its behavior—the set of event handlers that make an app behave as you want. You build these event handlers by assembling and configuring blocks representing events, functions, conditional branches, repeat loops, web calls, database operations, and more, and then test your work by actually running the app on your phone. After you write a few programs, the mapping between the internal structure of an app and its physical manifestation becomes clear. When that happens, you're a programmer!