CHAPTER 20

Repeating Blocks

If there's one thing that computers are good at, it's repeating things—like little children, they never tire of repetition. They are also very fast and can do things such as process your entire list of Facebook friends in a microsecond.

Figure 20-1.

In this chapter, you'll learn how to program repetition with special repeat blocks instead of copying and pasting the same blocks over and over. You'll learn how to send an SMS text to every phone number in a list and how to add up a list of numbers. You'll also learn that repeat blocks can significantly simplify an app.

Controlling an App's Execution: Branching and Looping

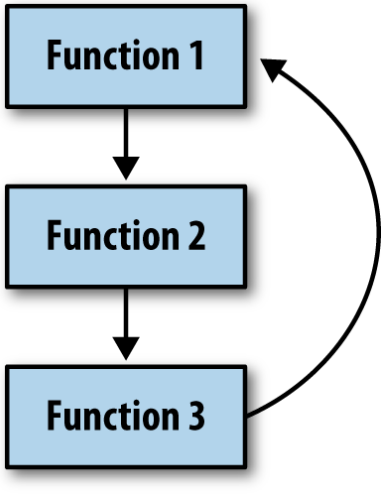

Figure 20-2. Repeat blocks cause a program to loop

In previous chapters, you learned that you define an app's behavior with a set of event handlers—events and the functions that should be executed in response. You also learned that the response to an event is often not a linear sequence of functions and can contain blocks that are performed only under certain conditions.

Repeat blocks are the other way in which an app behaves in a nonlinear fashion. Just as `if` and `else if` blocks allow a program to branch, repeat blocks allow a program to loop; that is, to perform a set of functions and then jump back up in the code and do it again, as illustrated in *Figure 20-1*. When an app executes, a program counter working beneath the hood of the app keeps track of the next operation to be performed. So far, you've examined apps in which the program counter starts at the top of an event handler and (conditionally)

performs operations top to bottom. With repeat blocks, the program counter loops back up in the blocks, continuously performing the same operations.

App Inventor provides a number of repeat blocks, including the `for each` and `while`, which we'll focus on in this chapter. `foreach` is used to specify functions that should be performed on each item of a list. So if you have a list of phone numbers, you can specify that a text should be sent to each number in the list

The `while` block is more general than the `for each`. With it, you can program blocks that continually repeat until some arbitrary condition changes. You can use `while` blocks to compute mathematical formulas such as adding the first *n* numbers or computing the factorial of *n*. You can also use `while` when you need to process two lists simultaneously; `for each` processes only a single list at a time.

Iterating Functions on a List with for each

Chapter 18 demonstrates an app that randomly calls one phone number in a list. Randomly calling one friend might work out sometimes, but if you have friends like mine, they don't always answer. A different strategy would be to send a "Thinking of you!" text to *all* of your friends and see who responds first (or most charmingly!).

One way to implement such an app is to simply copy the blocks for texting a single number and then paste them for each friend who you want to text, as shown in *Figure 20-2*.

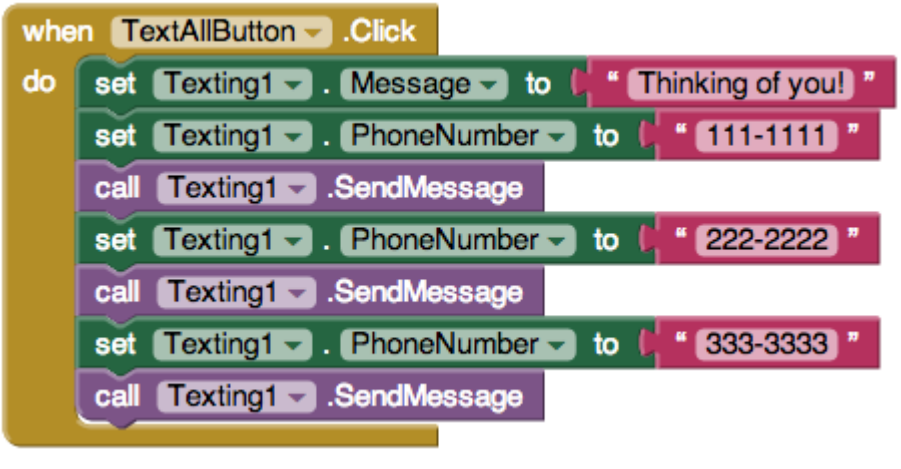

Figure 20-3. Copying and pasting the blocks for each phone number to be texted

This "brute force" copy-paste method is fine if you have just a few blocks to repeat. But, if you're dealing with large amounts of data or data that will change, you won't

want to modify your app with the copy-paste method each time you add or remove a phone number from your list.

The `for each` block provides a better solution. You define a `phoneNumbers` variable with all the numbers and then wrap a `for each` block around a single copy of the blocks that you want to perform. *Figure 20-3* shows the `for each` solution for texting a group.

Figure 20-4. Using the for each block to perform the same blocks for each item in the list

You can read this code as, "For each item (phone number) in the list `phoneNumbers`, set the `Texting` object's phone number to the item and send out the text message."

At the top of the `for each` block, you specify the list that will be processed. The block also has a *placeholder* variable that comes with the `for each`. By default, this placeholder is named "item." You can leave it that way or rename it. This variable represents the *current item* being processed in the list.

If a list has three items, the inner blocks will be executed three times. The inner blocks are said to be subordinate to, or nested within, the `for each` block. We say that the program counter "loops" back up when it reaches the bottom block within the `for each`.

A Closer Look at Looping

Let's examine the mechanics of the `for each` blocks in detail, because understanding loops is fundamental to programming. When the user taps `TextAllButton` and the

event handler is invoked, the first operation executed is the `set Texting1.Message to` block, which sets the message to "Thinking of You!" This block is executed only once.

The `for each` block then begins. Before the nested blocks of a `for each` are executed, the placeholder variable `item` is set to the first number in the `phoneNumbers` list (111–1111). This happens automatically; the `for each` relieves you of having to manually call `select list item`. After the first `item` is selected into the variable item, the blocks within the `for each` are executed for the first time. The `Texting1.PhoneNumber` property is set to the value of `item` (111–1111) and the message is sent.

After reaching the last block within a `for each` (the `Texting.SendMessage` block), the app "loops" back up to the top of the `for each` and automatically puts the next item in the list (222–2222) into the variable `item`. The two operations within the `for each` are then repeated, sending the "Thinking of You!" text to 222–2222. The app then loops back up again and sets `item` to the last item in the list (333–3333). The operations are repeated a third time, sending the third text.

Because the final item in the list has been processed, the `for each` looping stops at this point. In programming lingo, we say that control "pops" out of the loop, which means that the program counter moves on to deal with the blocks below the `for each`. In this example, there are no blocks below it, so the event handler ends.

Writing Maintainable Code

To the app's user, the `for each` solution just described behaves exactly the same as the "brute force" method of copying and then pasting the texting blocks. From a programmer's perspective, however, the `for each` solution is more *maintainable* and can be used even if the data (the phone list) is entered dynamically.

Maintainable software is software that can be changed easily without introducing bugs. With the `for each` solution, you can change the list of friends who are sent texts by modifying *only* the list variable—you don't need to change the logic of your program (the event handler) at all. Contrast this with the brute-force method, which requires you to add new blocks in the event handler when a new friend is added. Anytime you modify a program's logic, you risk introducing bugs.

Equally important, the `for each` solution would work even if the phone list was dynamic; that is, one in which the end user can add numbers to the list. Unlike our sample, which has three particular phone numbers listed in the code, most apps work with dynamic data that comes from the end user or some other source. If you redesigned this app so that the end user could enter the phone numbers, you would *have* to use a `for each` solution, because when you write the program, you don't know what numbers to put in the brute-force solution.

Using for each to Display a List

When you want to display the items of a list on the phone, you can plug the list into the `Text` property of a `Label`, as shown in *Figure 20-4*.

Figure 20-5. The simple way to display a list is to plug it directly into a label

When you plug a list directly into a `Text` property of a `Label`, the list items are displayed in the label as a single row of text, separated by spaces and contained in parentheses:

(111–1111 222–2222 333–3333)

The numbers might or might not span more than one line, depending on how many there are. The user can see the data and perhaps comprehend that it's a list of phone numbers, but it's not very elegant. List items are more commonly displayed on separate lines or with commas separating them.

To display a list properly, you need blocks that transform each list item into a single text value with the formatting you want. Text objects generally consist of letters, digits, and punctuation marks. However, text can also store special *control* characters, which don't map to a character you can see. A tab, for instance, is denoted by \t. (To learn more about control characters, check out the Unicode standard for text representation at *http://www.unicode.org/standard/standard.html*.)

In our phone number list, we want a newline character, which is denoted by \n. When \n appears in a text block, it means "go to the next line before you display the next item." Thus, the text object "111–1111\n222–2222\n333–3333" would appear as:

111–1111
222–2222
333–3333

To build such a text object, we use a `for each` block and "process" each item by adding it along with a newline character to the `PhoneNumberLabel.Text` property, as shown in *Figure 20-5*.

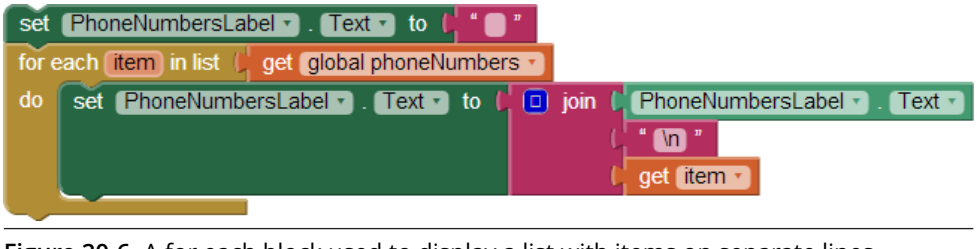

Figure 20-6. A for each block used to display a list with items on separate lines

Let's trace the blocks to see how they work. As discussed in *Chapter 15*, tracing shows how each variable or property changes as the blocks are executed. With a `for each`, we consider the values after each *iteration*; that is, each time the program goes through the `for each` loop.

Before the `for each`, the `PhoneNumbersLabel`, is initialized to the empty text. When the `for each` begins, the app automatically places the first item of the list (111–1111) into the placeholder variable `item`. The blocks in the `for each` then make `join` with `PhoneNumbersLabel.Text` (the empty text), \n, and the item, and set the result into `PhoneNumbersLabel.Text`. Thus, after the first iteration of the `for each`, the pertinent variables store the values shown in *Table 20-1*.

Table 20-1. The values after the first iteration

item	PhoneNumbersLabel.Text
111–1111	\n111–1111

Because the bottom of the `for each` has been reached, control loops back up, and the next item on the list (222–2222) is put into the variable `item`. When the inner blocks are repeated, `text` concatenates the value of `PhoneNumbersLabel.Text` (\n111–1111) with \n, and then with `item`, which is now 222–2222. After this second iteration, the variables store the values shown in *Table 20-2*.

Table 20-2. The values after the second iteration

item	PhoneNumbersLabel.Text
222–2222	\n111–1111\n222–2222

The third item of the list is then placed in `item`, and the inner block is repeated a third time. The final value of the variables, after this last iteration, is shown in *Table 20-3*.

Table 20-3. The variable values after the final iteration

item	PhoneNumbersLabel.Text
333–3333	\n111–1111\n222–2222\n333–3333

So, after each iteration, the label becomes larger and holds one more phone number (and one more newline). By the end of the `for each`, `PhoneNumbersLabel.Text` is set so that the numbers will appear as follows:

111–1111
222–2222
333–3333

The while-do Block

The `while-do` block is a bit more complicated to use than `for each`. The advantage of the `while-do` block lies in its generality: `for each` repeats over a list, but `while` can repeat *so long as any arbitrary condition is true*.

As you learned in *Chapter 18*, a condition tests something and returns a value of either true or false. `while-do` blocks include a conditional test, just like `if` blocks. If the test of a `while` evaluates to true, the app executes the inner blocks, and then loops back up and rechecks the test. As long as the test evaluates to true, the inner blocks are repeated. When the test evaluates to false, the app pops out of the loop (like we saw with the `for each` block) and continues with the blocks below the `while-do`.

Using while-do to Compute a Formula

Here's an example of a `while-do` block that repeats operations. What do you think the blocks in *Figure 20-6* do? One way to figure this out is to trace each block (see *Chapter 15* for more on tracing), tracking the value of each variable as you go.

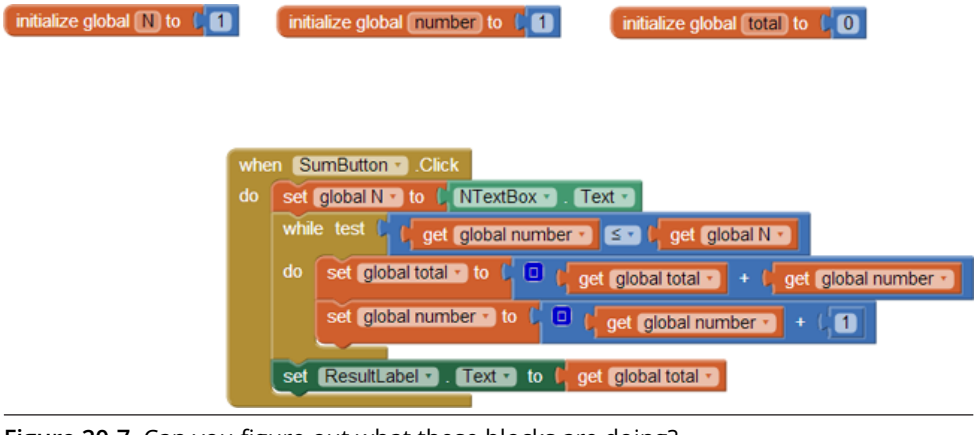

Figure 20-7. Can you figure out what these blocks are doing?

The blocks within the `while-do` loop will be repeated *while the variable number is less than or equal to the variable N*. For this app, `N` is set to a number that the end user types in a text box (`NTextBox`). Suppose that the user types a 3. The variables of the app would look like *Table 20-4* when the `while-do` block is first reached.

Table 20-4. Variable values when while-do is first reached

N	number	total
3	1	0

The `while-do` block first tests the condition: is `number` less than or equal to (≤) N? The first time this question is asked, the test is true, so the blocks nested within the `while-do` block are executed. `total` is set to itself (0) plus `number` (1), and `number` is incremented. After the first iteration of the blocks within the `while-do`, the variable values are as listed in *Table 20-5*.

Table 20-5. The variable values after the first iteration of the blocks within the while block

N	number	total
3	2	1

On the second iteration, the test "number≤N" is still true (2≤3), so the inner blocks are executed again. `total` is set to itself (1) plus `number` (2). `number` is incremented. When this second iteration completes, the variables hold the values listed in *Table 20-6*.

Table 20-6. The variable values after the second iteration

N	number	total
3	3	3

The app loops back up again and tests the condition. Once again, it is true (3≤3), so the blocks are executed a third time. Now, `total` is set to itself (3) plus `number` (3), so it becomes 6. `number` is incremented to 4, as shown in *Table 20-7*.

Table 20-7. The values after the third iteration

N	number	total
3	4	6

After this third iteration, the app loops back one more time to the top of the `while-do`. When the test "number≤N" runs this time, it tests 4≤3, which evaluates to false. Thus, the nested blocks of the `while-do` are not executed again, and the event handler completes.

So what did these blocks do? They performed one of the most fundamental mathematical operations: counting numbers. Whatever number the user types, the app will report the sum of the numbers 1..*N*, where *N* is the number entered. In this example, N is 3, so the app came up with a total of 1+2+3=6. If the user had typed 4, the app would have calculated 10.

Summary

Computers are good at repeating the same function over and over. Think of all the bank accounts that are processed to accrue interest, all the grades processed to compute students' grade point averages, and countless other everyday examples for which computers use repetition to perform a task.

This chapter explored two of App Inventor's repeat blocks. The `for each` block applies a set of functions to each element of a list. By using it, you can design processing code that works on an abstract list instead of concrete data. Such code is more maintainable; and if the data to be processed is dynamic, it's required.

Compared to `for each`, `while-do` is more general: you can use it to process a list, but you can also use it to synchronously process two lists or compute a formula. With `while-do`, the inner blocks are performed continuously for as long as a certain condition is true. After the blocks within the `while` are executed, control loops back up and the test condition is tried again. Only when the test evaluates to false does the `while-do` block complete.