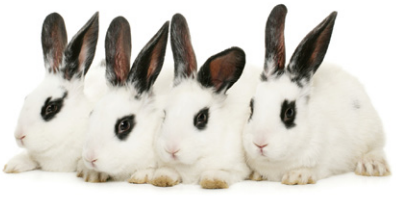
CHAPTER 20

Repeating Blocks: Iteration

One thing computers are good at is repeating operations—like little children, they never tire of repetition. They are also very fast and can do things like process your entire list of Facebook friends in a microsecond.



In this chapter, you'll learn how to program repetition with just a few blocks instead of copying and pasting the same blocks over and over. You'll learn how to do things like send an SMS text to every phone number in a list and sort list items. You'll also learn that repeat blocks can significantly simplify an app.

Controlling an App's Execution: Branching and Looping

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.

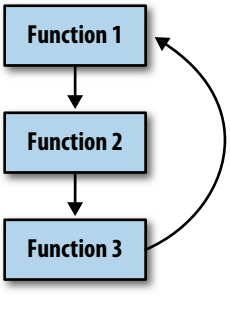


Figure 20-1. Repeat blocks cause a program to loop

Repeat blocks are the other way in which an app behaves nonlinearly. Just as **if** and **ifelse** blocks allow a program to branch, repeat blocks allow a program to *loop*; that is, to perform some 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 repeating functions.

In App Inventor, there are two types of repeat blocks: **foreach** and **while.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 **foreach**. With it, you can program blocks that continually repeat until some arbitrary condition changes. **while** blocks can be used 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; **foreach** processes only a single list at a time.

Repeating Functions on a List Using foreach

In Chapter 18, we discussed a Random Call app. 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 "Missing you" text to *all* of your friends and see who responds first (or more charmingly!).

With such an app, clicking a button sends a text to more than one friend. One way to implement this would be to simply copy the blocks for texting a single number, and then copy and paste them for each friend you want to text, as shown in Figure 20-2.

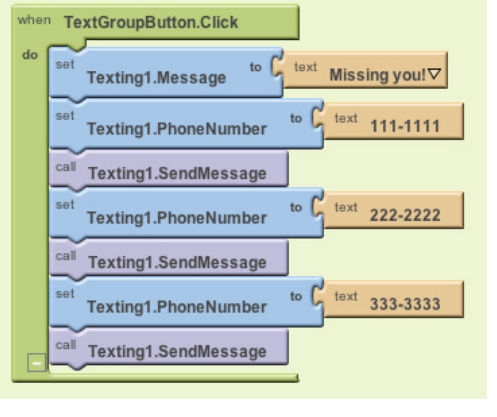


Figure 20-2. Copying and pasting the blocks for each phone number to be texted

This "brute force" copy-paste method is fine if you have just a few blocks to repeat. But data lists, such as the list of your friends, tend to change. You won't want to have to modify your app with the copy-paste method each time you add or remove a phone number from your list.

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

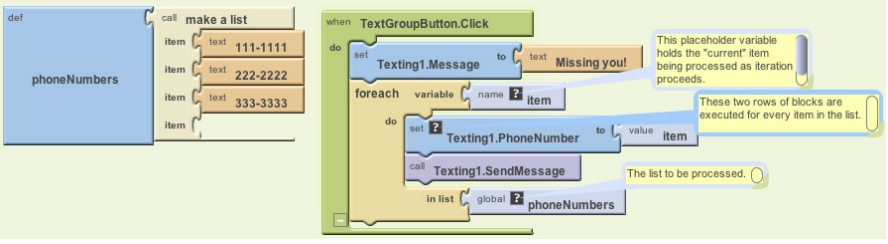


Figure 20-3. Using the foreach block to perform the same blocks for each item in the list

This code can be read 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.*

When you drag out a **foreach** block, you must specify the list to process by plugging a reference into the "in list" parameter at the bottom of the block. In this case, the **global phoneNumbers** block was dragged out of the My Definitions palette and plugged in to provide the list of phone numbers to text.

At the top of the **foreach** block, you also provide a name for a *placeholder* variable that comes with the **foreach**. By default, this placeholder is named "var." You can leave it that way or rename it. One common name for it is "item," as it represents the current item being processed in the list.

The blocks within the **foreach** are repeated for each item in the list, with the placeholder variable (in this example, `item`) always holding the item currently being processed. If a list has three items, the inner blocks will be executed three times. The inner blocks are said to be subordinate to, or within, the **foreach** block. We say that the program counter "loops" back up when it reaches the bottom block within the **foreach**.

A Closer Look at Looping

Let's examine the mechanics of the **foreach** blocks in detail, because understanding loops is fundamental to programming. When the `TextGroupButton` is clicked and the event handler invoked, the first operation executed is the **set Texting1.Message to** block, which sets the message to "Missing you." This block is only executed once.

The **foreach** block then begins. Before the inner blocks of a **foreach** are executed, the placeholder variable `item` is set to the first number in the `phoneNumbers` list (111–1111). This happens automatically; the **foreach** relieves you of having to manually call **select list item**. After the first item is selected into the variable `item`, the blocks within the **foreach** 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 **foreach** (the **Texting.SendMessage** block), the app "loops" back up to the top of the **foreach** and automatically puts the next item in the list (222–2222) into the variable `item`. The two operations within the **foreach** are then repeated, sending the "Missing 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—in this case, the third—has been processed, the **foreach** looping stops at this point. We say that control "pops" out of the loop, which means that the program counter moves on to deal with the blocks below the **foreach**. In this example, there are no blocks below it, so the event handler ends.

Writing Maintainable Code

To the end user, the **foreach** 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 **foreach** 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 **foreach** 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.

Even more important, the **foreach** solution would work even if the phone list was dynamic—that is, one in which the end user, not just the programmer, could 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 **foreach** solution, because when you write the program, you don't know what numbers to put in the brute force solution.

A Second foreach Example: Displaying 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.

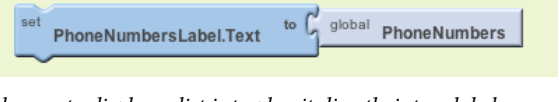


Figure 20-4. The simple way to display a list is to plug it directly into a label

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 may or may 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. But 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 down to the next line before you display the next thing." So 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 **foreach** block and "process" each item by adding it and a newline character to the `PhoneNumberLabel.Text` property, as shown in Figure 20-5.

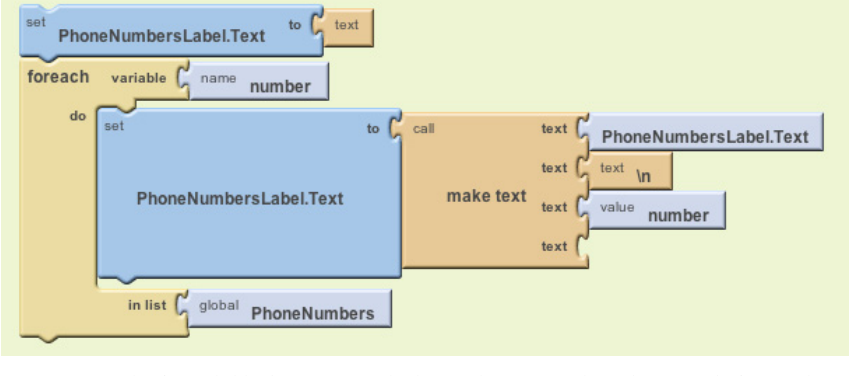


Figure 20-5. Using the foreach block to process the list and put a newline character before each item

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 **foreach**, we consider the values after each *iteration*; that is, each time the program goes through the **foreach** loop.

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

Table 20-1. The values of the variables after the first iteration of foreach

number	PhoneNumbersLabel.Text
111–1111	\n111–1111

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

Table 20-2. The variable values after the second iteration of foreach

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

The third item of the list is then placed in **number**, 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

number	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 **foreach**, PhoneNumbersLabel.Text is set so that the numbers will appear as:

111–1111
222–2222
333–3333

Repeating Blocks with while

The **while** block is a bit more complicated to use than **foreach**. The advantage of the **while** block lies in its generality: **foreach** repeats over a list, but **while** can repeat *while any arbitrary condition is true*. As a trivial example, suppose you wanted to text every *other* person in your phone list. You couldn't do it with **foreach**, but with **while**, you could just increment the index by two instead of one each time.

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 **foreach** block) and continues with the blocks below the **while**.

Using while to Synchronously Process Two Lists

A more instructive example of **while** and its generality involves situations in which you need to process two lists in a synchronous fashion. For example, in the MakeQuiz app (Chapter 10), you keep separate lists of the quiz questions and answers, along with an `index` variable to keep track of the current question number. To display each question-answer pair together, you need to iterate through the two lists in a synchronous fashion, grabbing the `index`*th* item of each. **foreach** only allows for traversing a single list, but with a **while** loop, you can use the index to grab an item from each list. Figure 20-6 illustrates using a **while** block to display the question-answer pairs on separate lines.

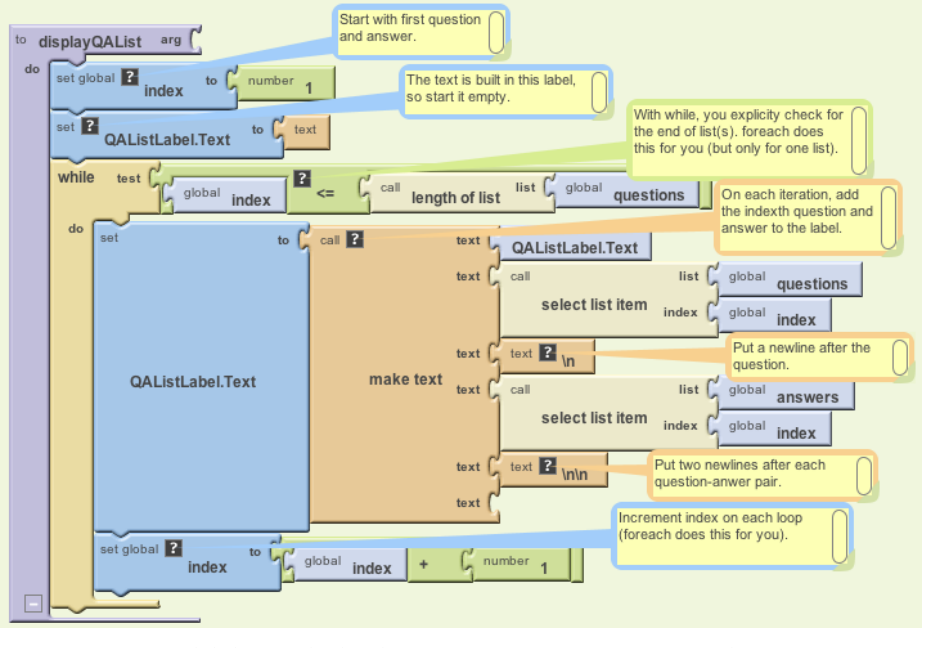


Figure 20-6. Using a while loop to display the question-answer pairs on separate lines

Because a **while** is used instead of a **foreach**, the blocks explicitly initialize the `index`, check for the end of the list, select the items in each loop, and increment the `index`.

Using while to Compute a Formula

Here's another example of **while** that repeats operations but has nothing to do with a list. What do you think the blocks in Figure 20-7 do, at a high level? 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.

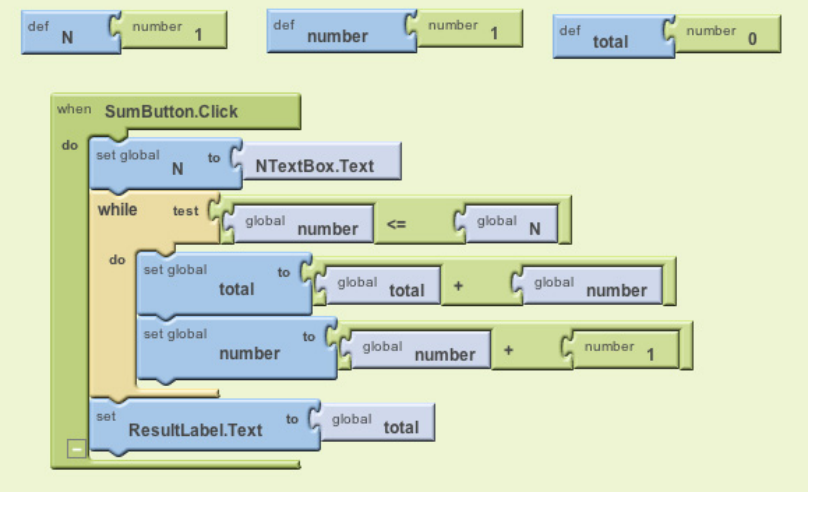


Figure 20-7. Can you figure out what these blocks are doing?

The blocks within the **while** 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 enters in a text box (`NTextBox`). Say the user entered a 3. The variables of the app would look like Table 20-4 when the **while** block is reached.

Table 20-4. This is how the variables look when the while block is reached

N	number	total
3	1	0

The **while** block first asks: is `number` less than or equal to (**<=**) `N`? The first time this question is asked, the test is true, so execution proceeds within the **while** block. `total` is set to itself (0) plus `number` (1), and `number` is incremented. After the first iteration of the blocks within the **while**, 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 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 are as 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, control loops back one more time. Now the test "number<=N", or 4<=3, evaluates to false. Thus, the inner blocks of the **while** 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 enters, the app will report the sum of the numbers 1..*N*, where *N* is the number entered. In this example, we assumed the user had entered 3, so the app came up with a total of 6. If the user had entered 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 where computers use repetition to perform a task.

App Inventor provides two blocks for repeating operations. The **foreach** 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 it's required if the data is dynamic.

Compared to **foreach**, **while** 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**, the inner blocks are performed continuously while 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** block complete.