Table of Contents

Table of Contents
Repeating Blocks
  Repeating Functions on a List: foreach
    foreach mechanics
  A Second foreach example: Displaying a List
    The newline character "\n"
Repeating Blocks with While Do
Processing Multiple Lists with While Do
Summary
Repeating Blocks

An app’s behavior, defined in the blocks editor, consists of event-handlers—an event occurs and the event-handler specifies the functions that should be executed in response. But the response is often not a linear sequence of functions. As we learned in chapter X, the response can contain conditional blocks—functions that are only performed if a test evaluates to true. In this chapter, we’ll see that the response can also contain repeat blocks—functions that should be repeated over an entire list or while some arbitrary condition holds true.

In App Inventor, there are two types of repeat blocks, the foreach and the 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, as shown in this diagram:

![Flowchart Diagram]

One use of the while block is to calculate some mathematical formula— for instance, you could use while to sum the first n numbers, or determine the mathematical function factorial(n). You can also use while when you need to process two lists simultaneously— foreach only processes a single list.

Repeating Functions on a List: foreach
In chapter x, we discussed a “call-a-random-friend” app. Such an app might work out sometimes, but if you have friends like mine they don’t always pickup. A different strategy would be to send a “missing you” text to all your friends, and see who responds first (or more charmingly!).

With such an app, clicking a button should cause a text to be sent to all your friends, not a random one. One way to do this would be to simply copy the blocks for texting a single number, then paste them for each of the friends you want to text.

This “brute-force” copy-paste method isn’t too big a deal if you have just a few friends. But data lists, in general, tend to grow and in general change. Performing the copy-paste each time you wanted to add or remove a friend from your miss-you list would become tiresome.

A better solution is to define a “phoneNumbers” list variable and then use a foreach block to apply the texting functions to all elements of the list. Here is the “foreach” solution:
This code can be read as:

"for each 'item' 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 that will be processed by plugging a reference into the "in list" parameter at the bottom of the block. In this case, the global "phoneNumbers" is plugged in.

At the top of the foreach block, you also provide a name for a placeholder variable. When you drag the foreach out, 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 always holding the item currently being processed. If a list has five items, the inner blocks will be executed five times. The inner blocks are said to be subordinate to or "within" the foreach block. We say that the program "loops" back up when it reaches the bottom block within the foreach.

**foreach mechanics**

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
is done 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 object's 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 app "loops" back up to the top of the foreach
and automatically puts the next item in the list ("222-2222") into item. The two operations within
the foreach are repeated, sending the "miss 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 causing the third text to be sent.

Because the final item in the list-- in this case the third-- has been processed, the foreach
completes. We say that control "pops out of the loop" and the event-handler processing
continues with the blocks below the foreach. In this case there are no blocks below it, so the
event-handler ends.

Note that the foreach solution has the same effect as the "brute force" method of copying then
pasting the texting blocks many times. However, the foreach solution is more maintainable and
can be used even if the data (the phone list) is dynamic.

Maintainable software is software that can be changed easily without introducing bugs. With
the foreach solution, you can change your list of friends by modifying the list variable only--
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
friends is added. Anytime you modify a program’s logic, bugs can be introduced.

Perhaps more importantly, the solution would work even if the data--the phone list-- is dynamic.
Unlike our sample, which has three phone numbers explicit (static) in the code, most apps work
with dynamic data that comes from the end-user or some other source. If you re-designed this
app so that the end-user, instead of the programmer, could enter the phone numbers, you’d
have to use a foreach solution.

**A Second foreach example: Displaying a List**

When you want to display a data list on the phone, you can plug it into the Text property of a
Label, such as in this example:

![Set Label Text](image)

When you plug a list directly into a text property of a label, the items of the list appear within
parenthesis and with spaces separating each:

(1112222 3334444 5556666)

The user can see the data and perhaps comprehend that it’s a list of numbers, but it’s not very
elegant. Lists are more commonly displayed on separate lines or with commas separating each
item.
To display a list more elegantly, you'll serialize it-- write blocks to transform the list items into a text value with the formatting you want. You can display the items on separate lines, add decorative text-- just about anything you can imagine.

As you must “process” each item, a foreach is used. In this case you process each item by adding it and a newline character to the Label.

Here’s a sample that displays the PhoneNumbers list on separate lines.

Above the foreach, PhoneNumberLabel.Text property is initialized to the empty text. Then the foreach block begins. In this sample, the placeholder variable has been named number-- it is the current phone number begin processed on each iteration. The first time the blocks within the foreach are executed, number will have “111-1111” in it. the second time it will have “333-4444”, and the third time it will have 5556666. Each time the make a text block appends the pnumber to the end of BroadcastListLabel.Text, placing a newline chracter, "\n", in between.

**The newline character "\n"**

Text objects generally consist of letters, digits, and punctuation marks. But text can also store special control characters which don't map to a single character. "\n" is such a control character. When it appears in a text block, it means "go down to the next line before you display the next thing." So the text "111-1111\n222-2222\n333-3333" would appear as:

```
1112222
3334444
5556666
```

When the foreach completes, we want PhoneNumberLabel.Text to be equal to the text "111-1111\n222-2222\n333-3333".

Let's trace the blocks to see how this is done. Tracing means to show how each variable/property changes as the blocks are executed. With a foreach, we'll consider the values after each iteration-- each time the blocks within the foreach are executed:
When the foreach begins, the app automatically places the first item of the list (“111-1111”) in number. The blocks in the foreach then make text with PhoneListLabel.Text (nothing), ‘\n’, and number, and set the result into PhoneListLabel.Text. Thus after the first iteration of the foreach, the pertinent variables store the following:

<table>
<thead>
<tr>
<th>number</th>
<th>PhoneListLabel.Text</th>
</tr>
</thead>
<tbody>
<tr>
<td>111-1111</td>
<td>\n111-1111</td>
</tr>
</tbody>
</table>

The next item of the list (“222-2222”) is then put into number and the inner blocks are repeated. This time make text concatenates the value of PhoneListLabel.Text “\n111-1111”, with ‘\n’, and then with number, which is now “222-2222”. After this second iteration, the variables store:

<table>
<thead>
<tr>
<th>number</th>
<th>PhoneListLabel.Text</th>
</tr>
</thead>
<tbody>
<tr>
<td>222-2222</td>
<td>\n111-1111\n222-2222</td>
</tr>
</tbody>
</table>

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:

<table>
<thead>
<tr>
<th>number</th>
<th>PhoneListLabel.Text</th>
</tr>
</thead>
<tbody>
<tr>
<td>333-3333</td>
<td>\n111-1111\n222-2222\n333-3333</td>
</tr>
</tbody>
</table>

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

111-1111
222-2222
333-3333

**Repeating Blocks with While Do**

The while-do is a bit more complicated to use, but it is more general than the for-each: whereas the for-each is specifically for repeating operations on each item of a list, a while-do repeats blocks *while any arbitrary condition is true*.

while-do blocks include a test, just like with the if-blocks. If the test of a while-do evaluates to true, the app executes the inner blocks, then loops back up and rechecks the test again. 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" and continues with the blocks below the while-do.

Let’s look at an example. What do you think the following blocks do?
The blocks within the while will be repeated over and over while the variable number is less than or equal to the variable N. For this app, N is set to a number the end-user enters in a text box (NTextBox). Say the user entered a 3. Then the variables of the app would look like this when the while block is reached:

<table>
<thead>
<tr>
<th>N</th>
<th>number</th>
<th>total</th>
</tr>
</thead>
<tbody>
<tr>
<td>3</td>
<td>1</td>
<td>0</td>
</tr>
</tbody>
</table>

The while block first asks a question: is number <= N? The first time this is asked, it is true, so execution proceeds within the while block. total gets 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:

<table>
<thead>
<tr>
<th>N</th>
<th>number</th>
<th>total</th>
</tr>
</thead>
<tbody>
<tr>
<td>3</td>
<td>2</td>
<td>1</td>
</tr>
</tbody>
</table>

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:

<table>
<thead>
<tr>
<th>N</th>
<th>number</th>
<th>total</th>
</tr>
</thead>
<tbody>
<tr>
<td>3</td>
<td>3</td>
<td>3</td>
</tr>
</tbody>
</table>
The app loops back up again and tests the condition. Once again it is true (3<=3), so the blocks are executed again. Now total is set to itself (3) plus number (3), so it becomes 6. number is incremented to 4:

<table>
<thead>
<tr>
<th>N</th>
<th>number</th>
<th>total</th>
</tr>
</thead>
<tbody>
<tr>
<td>3</td>
<td>4</td>
<td>6</td>
</tr>
</tbody>
</table>

After this third iteration, control then loops back one more time. Now the test number<=total, or 4<=3, evaluates to false. So 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. 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. The app always displays the sum, from 1 to N, where N is the number entered by the user.

**Processing Multiple Lists with While Do**

`foreach` is a high-level way for processing each item of a list. As it is more general, `while-do` can be used instead to process a list. Unlike `foreach`, it can also be used when the app needs to process two or more lists simultaneously, e.g., in a quiz app with separate lists of questions and answers.

To illustrate, let’s first take a look at a `while-do` version of the `textgroup` app:
These blocks do the same thing as the foreach solution earlier in the chapter. With the while-do, the automatic mechanisms of the foreach are the responsibility of the programmer. The index is explicitly initialized to 1 before the while-do begins. To test whether the entire list has been processed, the index is compared to length of list. At the bottom of the while loop, the index is incremented.

All of these operations occur with a foreach, but they are implicit-- foreach keeps its own internal index and relieves the programmer from having to explicitly handle this “bookkeeping”. The while is more general, however. For instance, if you wanted to text every other number, you could increment the index by 2 instead of 1.

Though instructive to compare the two versions, most programmers would choose the foreach over the while-do for text group, as it is simpler and less error prone for the system to take care of the bookkeeping. Sometimes, however, a foreach can’t be used-- if two lists need to be processed in a synchronous fashion, you’ll need to use a while.

The MakeQuiz app (Chapter 10) provides an example. In that app, the questions and answers of the quiz are in different lists. In order to display each question-answer pairs together, you need to iterate through the two lists in a synchronous fashion, grabbing the *index*th item of each. foreach only allows for “walking” through a single list, but with a while loop you can use the index to grab an item from each list. Here are blocks for displaying the question-answer pairs:
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 list of grades processed to compute student’s grade point average.

App Inventor provides two blocks for repeating operations. The foreach applies a set of functions to each element of a list. By using it, you can keep the actual list data out of the “processing” code, thus allowing items to be added and removed from the list without potentially introducing a bug into the processing.

while-do is more general than a foreach— it doesn’t necessarily work on list data. Instead, it executes its inner blocks 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.