Lists
Defining a List Variable
Selecting an Item
Text a Random Number in a List
Using Length of List
Indexing Through a List of Colors
Adding an item to a list
Removing an Item from a List
Summary
Programming with Lists of Data

Many apps process lists of data. Facebook is essentially a list of users each of whom has a list of friends. A quiz app has a list of questions and answers. A game has a list of scores.

List variables work like the atomic text and number variables you’ve worked with, but instead of the variable representing a “named memory cell”, the variable represents a set of contiguous memory cells.

<table>
<thead>
<tr>
<th>PhoneNumberList</th>
</tr>
</thead>
<tbody>
<tr>
<td>3219872</td>
</tr>
<tr>
<td>4153297878</td>
</tr>
<tr>
<td>4592371</td>
</tr>
</tbody>
</table>

Each element in such a list is accessed using an index. PhoneNumberList with an index of 1 refers to 3219872. An index of 2 gives 4153297878 and index 3 give 4592371.

App Inventor provides blocks for creating lists, adding elements to lists, selecting a particular item from a list, and applying operations to an entire list.

Defining a List Variable

A list is created in the Blocks Editor using a def variable block and a make list block. Suppose, for instance, that you are writing an app to text a list of phone numbers with one click. You create the phone numbers list in the following manner:

1. From the Built-In palette, drag a "def variable" block into the program area and change the name from "variable" to "phoneNumbers":

```
def variable
```

2. From the Lists palette, drag a "make a list" block in and plug it to the def variable block. This tells App Inventor that the variable will store a list of data as opposed to a single scalar value.

```
def phoneNumbers
```

3. Since the elements of your list are phone numbers, drag in some text blocks, enter the desired phone numbers, and connect them to the "item" slots in the make a list block. Put the phone numbers in as text objects, not math numbers, as math numbers can’t handle dashes and other formatting symbols, and you won’t be performing any calculations on the numbers anyway. Note that a new "item" slot opens up each time you add a new element to the list.
Your blocks should look this, but with the particular numbers you chose:

With the blocks above, you’ve defined a variable named "phoneNumbers". The list will be created when the app launches and any event-handlers you create can then access the list data.

**Selecting an Item**

Your app can access particular items in a list using an index-- a position in the list. If a list has three items, you can access the items with indices 1, 2, and 3. You access particular items with the select list item block, as in this one that choose the second item:

With select list item, you plug in the list you want to choose from in the first slot, and the index you want in the second slot. The blocks above say to select the second element of the list phoneNumbers. If you were selecting from the phoneNumbers list defined above, the result of Select List Item would be: "333-4444".

**Text a Random Number in a List**

Consider an app, discussed in Chapter N, that calls a random phone number from a list-- if you were single you might use it to choose who you want to date on a particular weekend. As shown in the previous chapter, you could program such an app using the random integer function to get a random number, and then a nested if-else blocks to call one of the numbers:
With these blocks, if the random number is 1, the number 11-2222 is called, if it is 2, another 333-4444 is called, and if it is 3, 555-6666 is called.

The app works fine, but it isn’t very maintainable. If you get some new “digits” and want to add them to the choices, you have to add a number of blocks including another if-else block. The scheme also only works on static data-- if you wanted the end-user of the app to dynamically add and remove numbers, the nested if-else approach, with the specific phone numbers in the blocks controlling the logic, doesn’t work.

Using a list, you can simplify things by eliminating the if-else blocks and using the random integer as an index into the list. Here are the blocks:
The event-handler first calls for a random number between 1 and 3, as the list has three items. It then calls select list item with the random number (1, 2, or 3) as the index. The randomly chosen phone number is set as the PhoneCall1 component's phone number, and the call is made. If you run this app multiple times, it should call your three numbers with equal likelihood.

**Using Length of List**

One advantage of the list solution above, compared to the if-else solution, is that you don't have to change the event-handler much in order to make the app work for a different set of phone numbers.

The goal is to just be able to modify the list and have the program still work. Say you added a fourth phone number to the make a list block, e.g.,

![Diagram](image)

You’d want the event-handler to still work. Do you think it would?

The answer is ‘no’. When you call for the random integer, the blocks specifically request a number between 1 and 3. Thus, no matter how many times you run the updated app the new fourth element will never be chosen.

An obvious solution is to change the "to" parameter of the random block from 3 to 4:

![Diagram](image)

However, there is an inherent dependency amongst blocks-- you must always change the number you send to the random call when you change the size of the list. As programmers are
notorious for forgetting things, it's better to avoid such dependencies. The last thing you want to do is introduce a bug when making a change in an app.

The better solution is to make the event-handling code more general. Instead of putting a fixed upper limit in the random call, you can specify the length of the list as the upper limit for the call to random. If there are n items in the list, get a random number between 1 and n. Fortunately, App Inventor provides a list operation specifically for this—length of list. Here is the modified event-handler:

The change is near the right: the "to" parameter to random integer is no longer a fixed number. Instead, it is the result of the call to "length of list".

Note how this solution is more maintainable— you can make the app work with more or less phone numbers by simply modifying the entries in "make a list". Perhaps more importantly, the solution will also work for apps that have dynamically changing lists, i.e., apps in which the user (not the programmer) is allowed to add or remove numbers from the list. This is important given that, generally speaking, you want to write apps for others, not just for yourself. Later in this chapter, we'll discuss dynamic lists.

**Indexing Through a List of Colors**

The choose a random item blocks provide a good example of using select list item with an index that can change. An even more common pattern involves using an index variable that begins at one (the front of the list) and is incremented so that each list item is processed.

As an example, consider an app that lets the user choose a color by clicking on a button, something you might find in a drawing app, or a game, such as Mastermind, that lets players choose an array of colors. Each time the user clicks, the button’s color changes to a different color. At some point, it goes through all the possible colors and sequences back to the first one. [show a pic of the button changing color on each click]

First, consider this solution that uses a nested if-else (the button is named ColorButton1):
As with the random phone call sample above, we can eliminate the nested-if-else and use a list to make this color sequencer more maintainable. By basing the logic on a list, the app will work even if we change the colors or number of colors that should appear.

The first step is to define a list variable for the colors:

Next, define a variable for the current color index. This is the variable that will change each time the user clicks on ColorButton1:

Now you're ready to (re-) code the ColorButton1.Click event handler. It should increment the index to get to the next color, then call select list item to get the color for the new background:
Assume the button’s background is initially set to Red in the Component Designer. Then the first time the button is clicked, currentColorIndex will change from its initial value, 1, to 2, and the button’s background color will change to the second item in the list, Green.

The next time the user clicks, the currentColorIndex will change from 2 to 3 and the background color to Blue. But what do you think will happen on the next click?

If you answered “an error”, you are correct. currentColorIndex will become 4 and then the app will try to select the 4th item in the list, but the list only has 3 items. The app will “force close” and an error message will be displayed:

The solution is to add an if-then block to handle the case when the last color in the list is reached. In this case, the app can just set the currentColorIndex back to 1:
When the button is clicked, the index is incremented, then checked to see if it’s too large. Note that currentColorIndex is compared to length of list, not 3, so the solution will work for lists of different sizes.

The event-handler above and the nested if-else solution both give the same behavior, just in different ways. The first uses a nested if-else to choose the next color, while the latter uses a list and an index variable to choose the next color.

The advantage of this latter method is that the app works no matter how many items are in the list. If the programmer adds a new color to the list, the app will still work.

Adding an item to a list

The examples in this chapter so far have involved lists with static data-- elements defined by the programmer in the make a list variable definitions. In practice, many apps deal with dynamic lists-- lists that change based on the end-user entering new items or new items being loaded in from the database or a web information source.

The add items to list block is used to add elements after a list’s initial definition. For instance, the following blocks add a new number to the phoneNumbers list:

You define a dynamic list with make a list, as in the previous examples, but without adding any pre-fixed items at definition time. For example, consider an app for taking notes. You’d define a dynamic list with the following definition:
When the app begins the notes list is empty. The items are added dynamically, most typically when the user enters some data in a form and clicks submit. For a note taking app, the form might be as simple as:

![Note Taker](image)

When the user enters a note and clicks the button, the button click event-handler adds the new item to the list:

![SubmitButton.Click](image)

The add item to list block appends the item to the end of the list. Each time the submit button is clicked, a new note will be added.

You’ll find the add item to list block in the lists drawer. Be careful-- there is also an append to list block, but that one actually appends a list to another list.

**Removing an Item from a List**

You can remove an item from a list with the remove list item block:

![remove list item](image)

The blocks above remove the second item from the list named notes.

A common user interface component used for removing items is the ListPicker. ListPicker will display a list as choosable items. Once the user chooses, the app can obtain the chosen index and call remove list item with it (replacing the 2 above with an index variable).
The ListPicker component has a property named elements. You want to set this property to your list so the list’s items will appear.

ListPicker also comes with a button that triggers the options being listed. When the user clicks the button, the ListPicker.BeforeSelecting event is triggered. This is where you should set the elements property to your list:

When the user chooses, the ListPicker.AfterSelection event is triggered. This is where you can access the user’s selection in the ListPicker.Selection property. The Selection property is the data, not the index. But you can get the index of the chosen element using the position in list function within the list drawer. The entire event-handler for removing after a ListPicker selection is:

**Summary**

Lists are used in almost every app you can think of. Understanding how they work, and how to use an index to sequence through them, is fundamental to programming. This chapter introduced the basic mechanisms for list manipulation, including typical user interfaces for letting the user add and remove items. In the following chapter, we’ll continue the discussion and focus on list iteration—applying functions to each item of a list.