CHAPTER 19

Programming Lists of Data

Figure 19-1.

As you've already seen, apps handle events and make decisions; such processing is fundamental to computing. But, the other fundamental part of an app is its data—the information it processes. An app's data is rarely restricted to single memory slots such as the score of a game. More often, it consists of lists of information and complex, interrelated items that must be organized just as carefully as the app's functionality.

In this chapter, we'll examine the way App Inventor handles data. You'll learn the fundamentals of programming both static information, in which the data doesn't change, and dynamic information, in which data is entered by the end user. You'll learn how to work with lists, and then you'll explore a more complex data structure involving lists of lists and a multiple-choice quiz app.

Many apps process lists of data. For example, Facebook processes your list of friends and lists of status reports. A quiz app works with a list of questions and answers. A game might have a list of characters or all-time high scores.

You specify list data in App Inventor with a variable, but instead of naming a single memory cell with the variable, you name a set of related memory cells. You specify that a variable is multi-item by using either the `make a list` or `create empty list` blocks. For instance, the variable `phoneNumbers` in *Figure 19-1* defines a list of three items.

Figure 19-2. phoneNumbers names three memory cells initialized with the numbers shown

Creating a List Variable

You create a list variable in the Blocks Editor by using an `initialize global variable` block and then plugging in a `make a list` block. You can find the `make a list` block in the Lists drawer, and it has only two sockets. But you can specify the number of sockets you want in the list by clicking on the blue icon and adding items, as depicted in *Figure 19-2*.

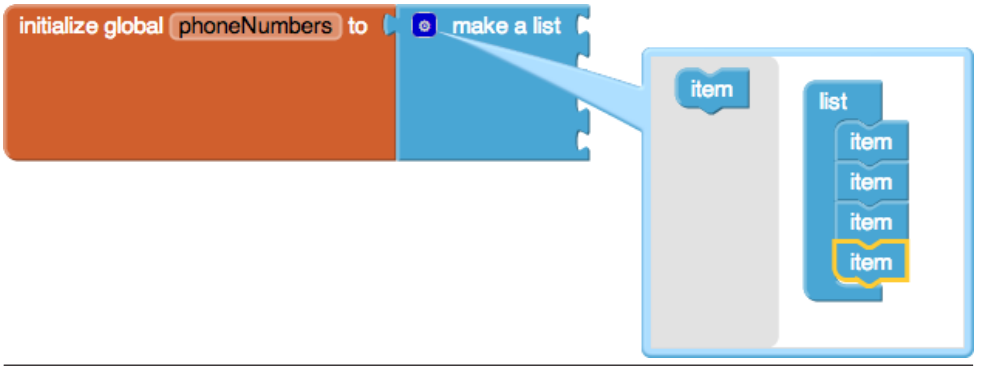

Figure 19-3. Click the blue icon on `make a list` to change the number of items

You can plug any type of data into the "item" sockets of `make a list`. For the phoneNumbers example, the items should be text objects, not numbers, because phone numbers have dashes and other formatting symbols that you can't put in a number object, and you won't be performing any calculations on the numbers (in which case, you would want number objects, instead).

Selecting an Item in a List

As your app runs, you'll need to select items from the list; for example, a particular question as the user traverses a quiz or a particular phone number chosen from a list. You access items within a list by using an index; that is, by specifying a position in the list. If a list has three items, you can access the items by using indices 1, 2, and 3. You can use the `select list item` block to grab a particular item, as shown in *Figure 19-3*.

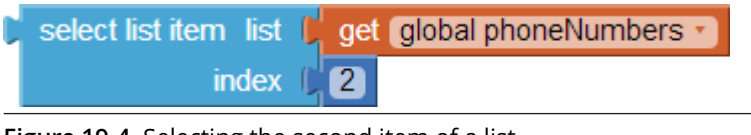

Figure 19-4. Selecting the second item of a list

With `select list item`, you plug in the list you want in the first socket, and the index you want in the second socket. For this phoneNumber sample, the result of selecting the second item is "333–4444."

Using an Index to Traverse a List

In many apps, you'll define a list of data and then allow the user to step through (or *traverse*) it. The Presidents Quiz in *Chapter 8* provides a good example of this: in that app, when the user clicks a Next button, the next item is selected from a list of questions and displayed.

The previous section showed how to select the second item of a list, but how do you select the *next* item? When you traverse a list, the item number you're selecting changes each time; it's your *current* position in the list. Therefore, you need to define a variable to represent that current position. "index" is the common name for such a variable, and it is usually initialized to 1 (the first position in the list), as demonstrated in *Figure 19-4*.

Figure 19-5. Initializing the variable index to 1

When the user does something to move to the next item, you *increment* the index variable by adding a value of 1 to it, and then select from the list by using that incremented value. *Figure 19-5* shows the blocks for doing this.

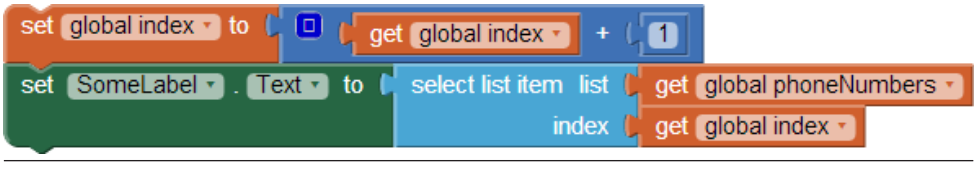

Figure 19-6. Incrementing the index value and using the incremented value to select the next list item

Example: Traversing a List of Paint Colors

Let's consider an example app with which the user can peruse each potential paint color for his house by tapping a "ColorButton." Each time the user taps, the button's color changes. When the user makes it through all of the possible colors, the app goes back to the first one.

For this example, we'll use some basic colors. However, you could customize the code blocks to iterate through any set of colors.

Your first step is to define a list variable for the `colors` list and initialize it with some paint colors as items, as depicted in *Figure 19-6*.

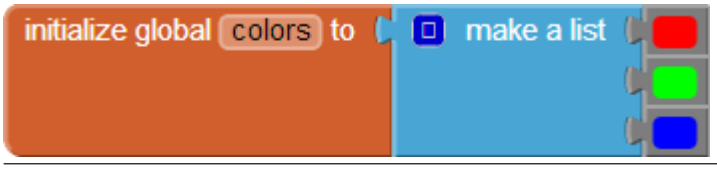

Figure 19-7. Initializing the list colors with a list of paint colors

Next, define an index variable that tracks the current position in the list. It should start at 1. You could give the variable a descriptive name such as `currentColorIndex`, but if you aren't dealing with multiple indexes in your app, you can just name it "index", as in *Figure 19-4*.

The user traverses to the next color in the list by clicking the `ColorButton`. Upon each tap, the index should be incremented and the `BackgroundColor` of the button should change to the currently selected item, as shown in *Figure 19-7*.

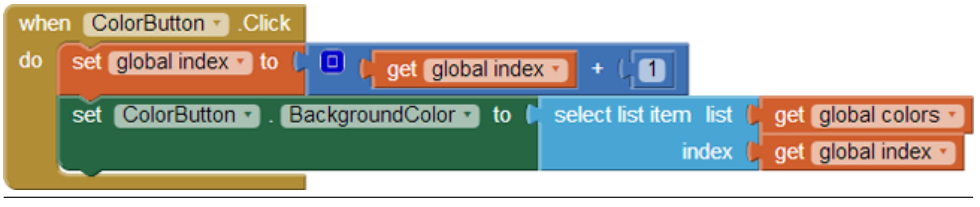

Figure 19-8. Each tap of the button changes its color

Let's assume the button's background is initially set to Red in the Component Designer. The first time the user taps the button, `index` changes from its initial value of 1 to 2, and the button's background color changes to the second item in the list, green. The second time the user taps it, the `index` changes from 2 to 3, and the background color switches to Blue.

But what do you think will happen the next time the user taps it?

If you said there would be an error, you're right! `index` will become 4 and the app will try to select the fourth item in the list, but the list only has three items. The app will *force close*, or quit, and the user will see an error message like the one in *Figure 19-8*.

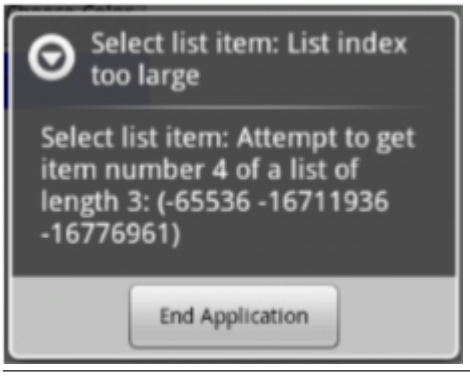

Figure 19-9. The error message displayed when the app tries to select a fourth item from a three-item list

Obviously, that message is not something you want your app's users to see. To avoid that problem, add an `if` block to check whether the last color in the list has been reached. If it has, the `index` can be changed back to 1 so that the first color is again displayed, as illustrated in *Figure 19-9*.

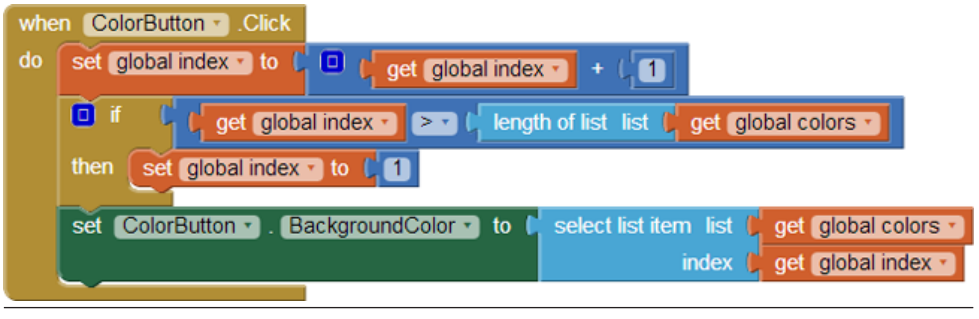

Figure 19-10. Using an if to check whether the index value is larger than the length of the list

When the user taps the button, the `index` is incremented and then checked to see if its value is too large. The `index` is compared to `length of list`, not 3; this way your app will work even if you add items to the list. By checking if the index is greater than your list length (versus checking if it is greater than the specific number 3), you've eliminated a code dependency in your app. A *code dependency* is a programming term that describes code that is defined *too* specifically and lacks flexibility. Thus, if you change something in one place—in our example here, you add items to your list—you'll need to search for every instance where you use that list and change it explicitly.

As you can imagine, these kinds of dependencies can get messy very quickly, and they generally lead to many more bugs for you to chase down, as well. In fact, the

design for our Color app contains another code dependency as it is currently programmed. Can you figure out what it is?

If you changed the first color in your list from red to some other color, the app won't work correctly unless you also remembered to change the initial `Button.BackgroundColor` you set in the Component Designer. The way to eliminate this code dependency is to set the initial `ColorButton.BackgroundColor` to *the first color in the list* rather than to a specific color. Because this change involves behavior that happens when your app first opens, you do this in the `Screen.Initialize` event handler that is invoked when an app is launched, as illustrated in *Figure 19-10*.

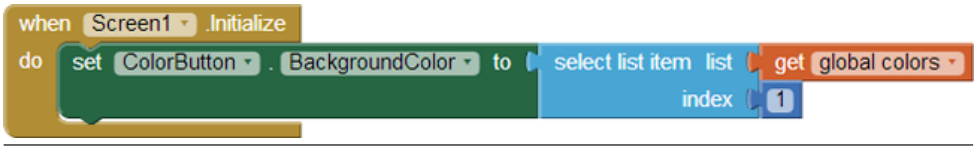

Figure 19-11. Setting the BackgroundColor of the button to the first color in the list when the app launches

Creating Input Forms and Dynamic Data

The previous Color app involved a *static* list: one whose elements are defined by the programmer (you) and whose items don't change unless you change the blocks themselves. More often, however, apps deal with *dynamic* data: information that changes based on the end user entering new items, or new items being loaded in from a database or web information source. In this section, we discuss an example Note Taker app, in which the user enters notes in a form and can view all of her previous notes.

Defining a Dynamic List

Apps such as a Note Taker begin with an empty list. When you want a list that begins empty, you define it with the `create empty list` block, as depicted in *Figure 19-11*.

Figure 19-12. The blocks to define a dynamic list don't contain any predefined items

Adding an Item

The first time someone launches the app, the `notes` list is empty. But when the user types some data in a form and taps Submit, new notes will be added to the list. The form might be as simple as the one shown in *Figure 19-12*.

Figure 19-13. Using a form to add new items to the notes list

When the user types a note and taps the Submit button, the app calls the `add items to list` function to append the new new item to the list, as illustrated in *Figure 19-13*.

Figure 19-14. Calling add items to list to add the new note when the user taps the SubmitButton

You can find the `add items to list` block in the List drawer. Be careful: there is also an `append to list` block, but that one is a fairly rare block used to append one entire list to another.

Displaying a List

The contents of list variables, like all variables, are not visible to the user. The blocks in *Figure 19-13* add items to the list each time `SubmitButton.Click` is invoked, but the user will not receive feedback that the list is growing until you program more blocks to actually display the content of the list.

The simplest way to display a list in your app's user interface is to use the same method you use for displaying numbers and text: put the list in the `Text` property of a `Label` component, as illustrated in *Figure 19-14*.

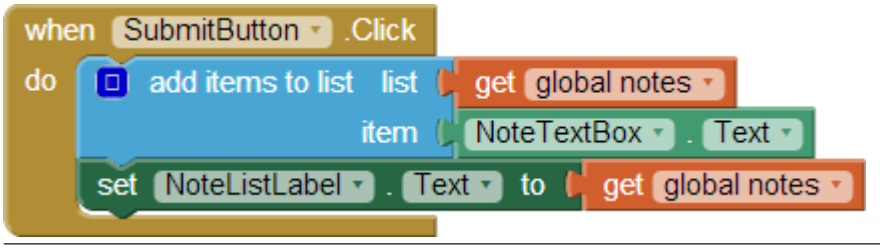

Figure 19-15. Displaying the list to the user by placing it in a label.

Unfortunately, this simple method of displaying a list isn't very elegant; it puts the list within parentheses, with each item separated by a space and not necessarily on the same line. For instance, if the user were to type, "Will I ever finish this book?" as the first note, and "I forget what my son looks like!" as the second, the app would display the notes list similar to what we see in *Figure 19-15*.

Figure 19-16. These entries are listed using default formatting

In *Chapter 20*, you can see a more sophisticated way to display a list.

Removing an Item from a List

You can remove an item from a list by using the `remove list item` block, as shown in *Figure 19-16*.

Figure 19-17. Removing an item from a list

The blocks in *Figure 19-16* remove the second item from the list named `notes`. Generally, however, you won't want to remove a fixed item (e.g., 2), but instead will provide a mechanism for the user to choose the item to remove.

You can use the `ListPicker` component to provide the user with a way to select an item. `ListPicker` comes with an associated button. When the button is tapped, the `ListPicker` displays the items of a list from which the user can choose one. When the user chooses an item, the app can remove it.

`ListPicker` is easy to program if you understand its key events, `BeforePicking` and `AfterPicking`, and its key properties, `Elements`, `Selection`, and `SelectionIndex` (see *Table 19-1*).

Table 19-1. The key events and properties of the ListPicker component

Event	Property
`BeforePicking`: Triggered when button is clicked.	`Elements`: The list of choices.
`AfterPicking`: Triggered when user makes a choice.	`Selection`: The user's choice.
	`SelectionIndex`: Position of choice.

The user triggers the `ListPicker.BeforePicking` event by tapping the `ListPicker`'s associated button. In the `ListPicker.BeforePicking` event handler, you'll set the `ListPicker.Elements` property to a list variable so that the data in the list displays. For the Note Taker app, you'd set `Elements` to the `notes` variable that contains your list of notes, as shown in *Figure 19-17*.

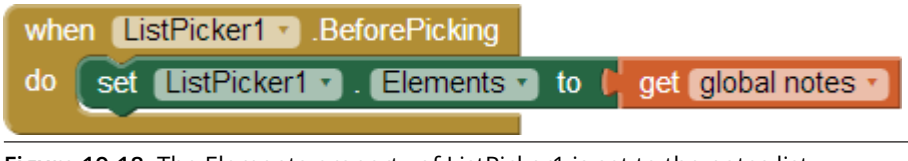

Figure 19-18. The Elements property of ListPicker1 is set to the notes list

With these blocks, the items of the list notes will appear in the `ListPicker`. If there were two notes, it would appear as shown in *Figure 19-18*.

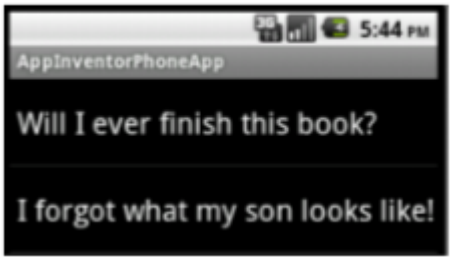

Figure 19-19. The list of notes appears in the ListPicker

When the user chooses an item in the list, it triggers the `ListPicker.AfterPicking` event. In this event handler, you can access the user's selection in the `ListPicker.Selection` property.

However, your goal in this example is to remove an item from the list, and the `remove item from list` block expects an index, not an item. The `Selection` property of the `ListPicker` is the actual data (the note), not the index. Therefore, you need to use the `SelectionIndex` property instead because it provides you with the index of the chosen item. It should be set as the index of the `remove list item` block, as demonstrated in *Figure 19-19*.

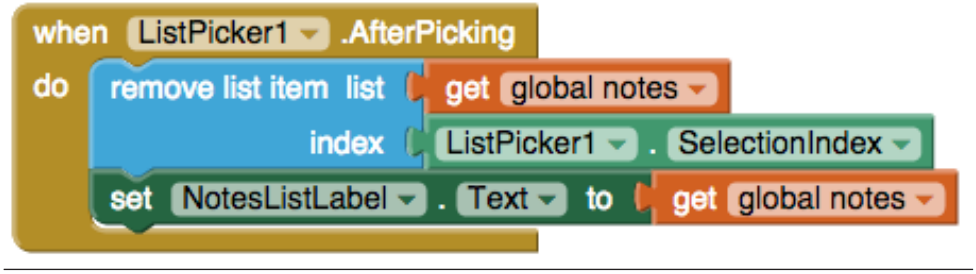

Figure 19-20. Removing an item by using ListPicker.SelectionIndex

Lists of Lists

The items of a list can be of any type, including numbers, text, colors, or Boolean values (true/false). But, the items of a list can also, themselves, be lists. Such complex data structures are common. For example, a list of lists could be used to convert the Presidents Quiz (*Chapter 8*) into a multiple-choice quiz. Let's look again at the basic

structure of the Presidents Quiz, which is a list of questions and a list of answers, as shown in *Figure 19-20*.

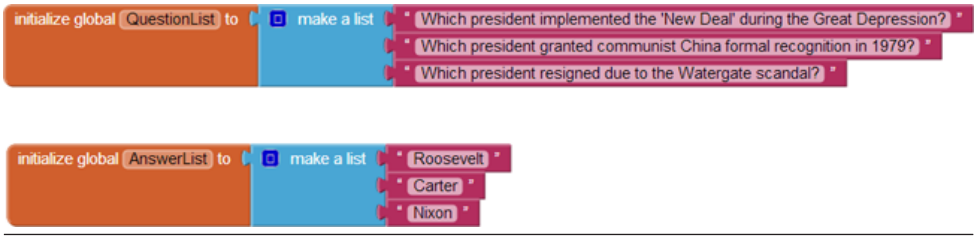

Figure 19-21. A list of questions and a list of answers

Each time the user answers a question, the app checks to see if it is correct by comparing the answer to the current item in the `AnswerList`.

To make the quiz multiple choice, you need to keep an additional list, one which stores the choices for each answer to each question. You specify such data by placing three `make a list` blocks within an inner `make a list` block, as demonstrated in *Figure 19-21*.

Figure 19-22. A list of lists is formed by inserting `make a list` blocks as items within an inner `make a list` block

Each item in the variable `answerChoices` is itself a list containing three items. If you select an item from `answerChoices`, the result is a list. Now that you've populated your multiple-choice answers, how would you display that to the user?

As with the Note Taker app, you could use a `ListPicker` to present the choices to the user. If the index were named `currentQuestionIndex`, the `ListPicker.BeforePicking` event would appear as shown in *Figure 19-22*.

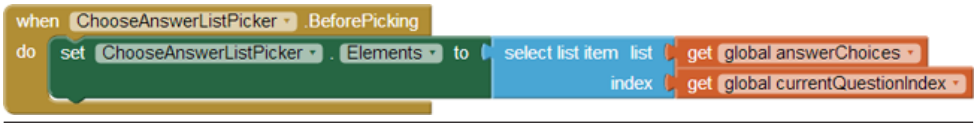

Figure 19-23. Using the List Picker to present one of the list of answer choices to the user

These blocks would take the current sublist of `answerChoices` and let the user choose from it. So, if `currentQuestionIndex` were 1, the `ListPicker` would show a list like the one in *Figure 19-23*.

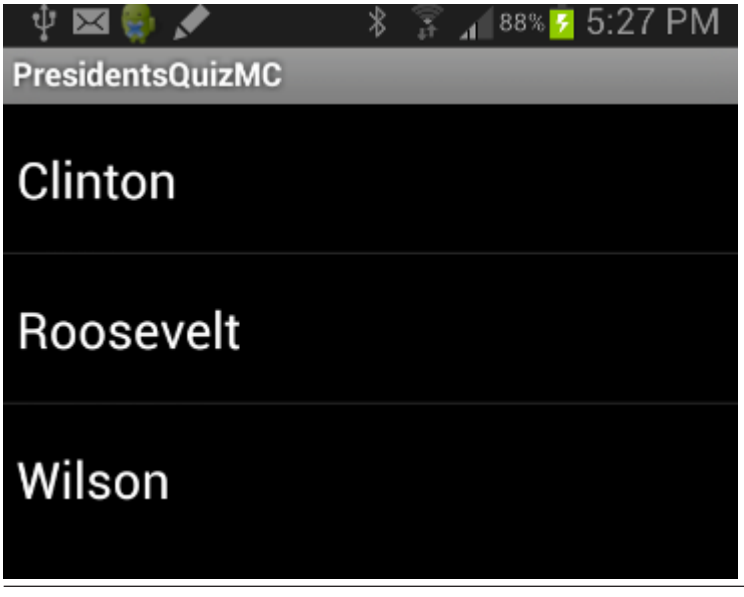

Figure 19-24. The answer choices presented to the user for the first question

When the user chooses, you check the answer with the blocks shown in *Figure 19-24*.

Figure 19-25. Checking whether the user chose the correct answer

In these blocks, the user's selection from the `ListPicker` is compared to the correct answer, which is stored in a different list, `AnswerList` (because `answerChoices` provides only the choices and does not denote the correct answer).

Summary

Lists are used in almost every app you can think of. Understanding how they work is fundamental to programming. In this chapter, we explored one of the most common programming patterns: using an index variable that starts at the beginning of the list and is incremented until each list item is processed. If you can understand and customize this pattern, you are indeed a programmer!

We then covered some of the other mechanisms for list manipulation, including typical forms for letting the user add and remove items. Such programming requires yet another level of abstraction, as you have to envision the dynamic data before it really exists. After all, your lists are empty until the user puts something in them. If you can understand this, you might even think of quitting your day job.

We concluded the chapter by introducing a complex data structure, a list of lists. This is definitely a difficult concept, but we explored it by using fixed data: the answer choices for a multiple-choice quiz. If you mastered that and the rest of the chapter, your final test is this: create an app that uses a list of lists but with dynamic data. One example would be an app with which people can create their own multiple-choice quizzes, extending even further the MakeQuiz app in *Chapter 10*. Good luck!

While you think about how you'll tackle that, understand that our exploration of lists isn't done. In the next chapter, we continue the discussion and focus on list iteration with a twist: applying functions to each item in a list.