Defining Procedures and Reusing Blocks

A procedure is a named sequence of blocks that an app can “call” from other places in your program.

When I tell my little boy to brush his teeth, I really mean “take your brush and toothpaste from the cabinet, squeeze out some toothpaste on the brush, swivel the brush on each of your teeth for 60 seconds (hal) and so on. “Brush your teeth” is an abstraction, a name for a sequence of lower-level operations. When I say to him, “get off the Wii and brush your teeth”, I am invoking a procedure, asking him to perform the operations that we’ve agreed to mean, “Brush your teeth”

In programming, you can create such named sequences of operations. Some programming languages call them functions or sub-programs. In App Inventor, they’re called procedures. Here’s an example of a procedure whose job is to find the distance, in miles, between two GPS coordinates:

You’ll learn how to create such a procedure in this chapter, but it’s important to first know why organizing an app with procedures is important. First, procedures let you extend the language by which you design and build programs. If I had to explain the steps of “brush your teeth” to my son each night, we’d have a tough time! Similarly, being able to refer to a procedure, “distanceBetweenPoints”, allows for the details of it to be ignored during high-level discusssions. This type of abstraction is key to a groups ability to solve a complex problem at a high-level. Most useful software tends to be very large— thousands of lines of code or blocks. The only way to understand the big picture is to use abstraction, to break it down into parts.

Second, procedures reduce errors because they eliminate redundancy in your code. With procedures you can put a chunk of code in one place, then call it from various places in your app. So maybe you have an app that needs to know the minimum distance between your current location and ten other spots. With the procedure, you don’t have to copy the code blocks
ten times, you just call it whenever you need it. The alternative—copying and pasting blocks—is error prone because when you make a change, you have to find all the copies of those blocks and change each of them. Imagine finding the 5-10 places you pasted a particular snippet of code in a million line (or block) app! Instead of copy-paste, a procedure lets you encapsulate blocks in one place.

Finally, with procedures you can build up a library of code that can be reused in many apps. Even when programming apps, experienced programmers are always thinking of building reusable parts that can be used in other apps. Some programmers never even create apps, but focus solely on building reusable code libraries.

So let’s take a look at how procedures are created, and how you can make them as general purpose as possible.

Consider the following code, and identify the redundant code blocks:

![Diagram showing code blocks]

The redundant blocks are the ones involving a foreach to display the list items. The app needs to display the notes list when a new item is added, when an item is removed, and when the list is loaded from the database at application launch.
Such redundant code will ring a bell for an experienced programmer-- probably even before they’ve copied and pasted the blocks in the first place. They know that its best to encapsulate such redundancy in one place-- a procedure-- both to make the program more understandable and so that changes will be easier later.

**Defining a Procedure**

You define a procedure using a block from the Definitions drawer, just as with variables. With procedures, you drag out either a “to procedure” block or a “to procedure with result” block. The latter is used if your procedure should calculate some value and return it.

After dragging out one of the procedure blocks, change its name by clicking on the word “procedure” and typing the new name:

![Procedure Block](image)

Next, drag one copy of the original blocks out of its event-handler, and place it within the procedure block:

![Procedure Block with Blocks](image)

You now have a procedure for displaying the notes list.

**Calling a Procedure**

Procedures, like displayList and “brush your teeth”, are entities with the potential to perform a task. However, they’ll only perform that task if they are called upon to do so. In our example, we created a procedure but never called it (just as I often forget to call my son to duty!).

You call a procedure by dragging out a call block from the My Definitions drawer. Recall that the
My Definitions drawer is empty when you first begin an app. Each time you define something, new blocks appear in it. When you define a variable, blocks to set and access the variable’s value are added. When you define a procedure, a call block is added:

You use call blocks all the time to call built-in functions of App Inventor. Here you are calling a procedure you’ve created yourself.

For the notes sample, you’ll drag out three call displayList blocks and use them to replace the redundant code in the three event handlers. For instance, the ListPicker1.AfterPicking event-handler (for deleting a note) should look like this:

To understand how the call block works, think of an app as having a pointer that steps through the blocks performing their functions. In programming, this pointer is called the program counter.

When the program counter is performing the blocks within an event-handler and it reaches a call block, it jumps over to the procedure and executes the blocks in it. When the procedure completes, the program counter jumps back to its previous location (the call block) and proceeds from there.

**Parameterizing a Procedure**

DisplayList serves a purpose by allowing redundant code to be refactored into a single place. The app is easier to understand because one can read it at a high-level and at times ignore the details of displayList. It is also helpful because you may decide to modify how you display the list, and the procedure allows you to make such a modification in a single place (instead of three).
The DisplayList procedure has limits in terms of its *general* usefulness, however. The procedure works for a specific list (noteList) and displays that list in a specific label (notesLabel). You couldn’t use it to display a different data list, say a list of users using the app, because it is defined too specifically.

App Inventor and other languages provide a mechanism for making procedures more general: parameters. Parameters are the information a procedure needs to do its job—specifics about how the procedure should be performed. When I ask my son to brush his teeth, I might tell him what toothpaste to use, or how long to brush. We say that toothpaste type and brushing time are parameters of the procedure brush teeth.

You define parameters for a procedure by dragging out a name block from the Definitions drawer and plugging it into the procedure slots labeled “arg”. For the displayList procedure, we can define a parameter named “list”:

The name list block defines a *formal parameter* for the procedure displayList.

The next step is to replace the reference to the specific list *notes* with a reference to the parameter *list*:
The new version of the procedure is now more generic— the caller can now send it any list and it will display it. When you add the parameter to the procedure, App Inventor automatically puts a corresponding slot in the call block. So when the parameter list was added to displayList, the call blocks to displayList became:

The list slot within the procedure definition is called a “formal parameter”. The corresponding slot within the call block is called an “actual parameter”, as it refers to the actual data that is used in each call to the procedure.

You must fill in the slots for the actual parameter, or the app will have an error. For the original code blocks, you’d add a reference to the notesList as the actual parameter. Here’s how ListPicker.AfterSelection should be modified:
Now when displayList is called, the list notes is sent over to the procedure and placed in the parameter list. The program counter proceeds to execute the blocks in the procedure, referring to the parameter list but really working with the variable notes.

The key is that now the procedure displayList can be used with any list, not just notes. So if the had a list of users, or a list of comments on notes, you could reuse the procedure for that as well, e.g.:

### Returning Values from a Procedure

There is still one issue with the displayList procedure in terms of its reusability-- it always displays data in the label NotesLabel. What if you wanted the user list or some other list to be displayed in a different user interface object.

One solution is to re-conceptualize the procedure, to change its job from displaying a list in a particular label to simply returning a text object that the “caller” can display anywhere. To do this, you’ll use a procedureWithResult block instead of the procedure block:
The procedure with return block has an extra slot at the bottom-- you place a variable in this slot and it’s returned to the caller. So just as the caller can send data to a procedure with a parameter, a procedure can send data back with a return value.

Below is the reworked version of the displayList procedure. Note that, because it is now doing a different job, its name is changed to serializeList:

As you can see, a variable displayText is defined to hold the text as the procedure iterates through each item of the list-- it replaces the too specific NotesLabel component that was being used. When the foreach completes, the variable displayText has the list items in it, with each item separated by a ‘\n’ new-line character. displayText is thus plugged into the return value slot.

When a procedureWithResult is defined, its corresponding call blocks are different. Compare the call to serializeList with the call to the procedure displayList:

The difference is that the serializeList call has a plug in the back. The reason is that when the call is executed, the procedure will run through its task and then return a value to the call block. That return value-- in this case, a text object-- must be plugged into something.

In this case, the callers can plug that return value into a set Label.Text to block-- they can display the serialized list in any label they want. For the notes sample, the three event-handlers that need to display a list will call the procedure like this:
Now serializeList is completely generic-- it doesn't refer to any lists or labels specifically. Thus, the part of the app dealing with users could display the userList in a different label:

Reuse between Apps
The reuse of code blocks through procedures need not be restricted to a single app. There are many procedures, like serializeList, that have utility in just about any app you create. Organizations will also build up code libraries for procedures with utility in a particular domain, e.g., a code library of animation procedures.

Typically, programming languages provide an “import” utility that allows for library code to be included in any app. App Inventor doesn't yet have such a utility, but one is being developed. In the mean time, you can create procedures in a special “library app” and begin new app development with “Save As” of that app.

A Second Example: DistanceBetweenPoints
With the displayList (serializeList) example, we characterized procedure definition as a way to eliminate redundant code. Generally, however, a software developer or team will design an app in the first place with procedures (and no redundancy) in mind.

Consider an app to determine the closest local hospital to one's current location, something that would come in very handy in case of an emergency. The high-level design of the app reads something like the paragraph below-- can you determine the procedures that should be defined for this app?

when the app launches, find the distance, in miles, between the first hospital and the current location. Then find it for the second hospital, and so on. When you have the distances, determine the minimum distance and display the address (or a map) to that location.

The procedures an app should define can often be gleaned from such text. In this case, finding the distance between two points and determining the minimum of some distances are two pieces of functionality that this app will need.

Let's begin with a distanceBetweenPoints procedure. In designing a procedure you must determine the inputs and outputs of the procedure-- the parameters the caller will send to the procedure for it to do its job, and the result value the procedure will send back to the caller. In this case, the caller needs to send the latitude and longitude of both points to the procedure.
The procedures job is to send the distance, in miles, back.

Here's the procedure, using a well-known formula for approximating the mileage between two GPS coordinates:

Here are two calls to the procedure, each of which finds the distance from the current location to a particular hospital:
For the first call, the actual parameters for point1 are the GPS coordinates for St. Mary's Hospital, while point2 uses the current readings from the LocationSensor. The result value is placed in the variable distanceStMarys. The second call is similar, only the data for CPMC Hospital is used.

Summary

Programming languages, like App Inventor, provide a base set of functionality-- the built-in functionality. Procedures let app creators extend that language with new abstractions. App Inventor doesn't provide a block for displaying a list, so you build one. You Need a block for computing the distance between GPS coordinates? Create one.

This ability to define higher-level procedure blocks is the key to engineering understandable, maintainable apps. Complex problems (and programs) are overwhelming if you have to think about all the details all of the time. Procedures let you encapsulate code blocks and give those blocks a name. While you program the procedure, you focus on the details of those blocks. But in programming the rest of the app, you now have an abstraction-- a name-- that you can refer to at a high-level.