

# Programming Your App's Memory

*Just as people need to remember things, so do apps. This chapter examines how you can program an app to remember information.*

*When someone tells you the phone number of a pizza place, your brain stores it in a memory slot. If someone calls out some numbers for you to add, you store the numbers and intermediate results in memory slots. In such cases, you are not fully conscious of how your brain stores information or recalls it.*

*An app has a memory, as well, but its inner workings are far less mysterious than those of your brain. In this chapter, you'll learn how to set up an app's memory, how to store information in it, and how to retrieve that information at a later time.*

Figure 16-1.



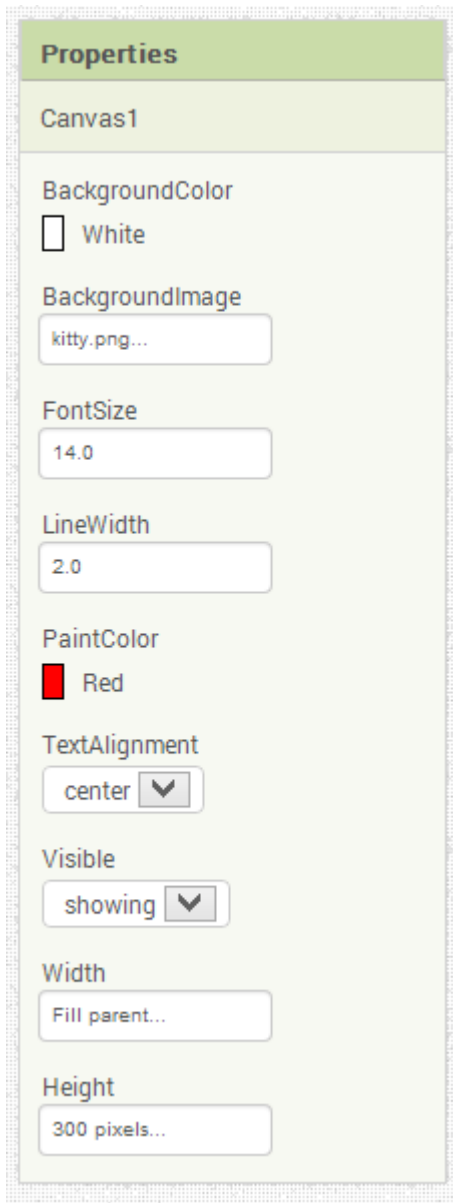
## Named Memory Slots

An app's memory consists of a set of *named memory slots*. Some of these memory slots are created when you drag a component into your app; these slots are called *properties*. You can also define named memory slots that are not associated with a particular component; these are called *variables*. Whereas properties are typically associated with what is visible in an app, variables can be thought of as the app's hidden "scratch" memory.

## Properties

The user of an app can see visible components such as buttons, textboxes, and labels. Internally, however, each component is completely defined by a set of properties. The values stored in the memory slots of each property determine how the component appears.

You set the values of properties directly in the Component Designer. For instance, *Figure 16-1* shows the panel for modifying the properties of a Canvas component.



**Figure 16-2.** You can set component properties in the Designer; you are setting the initial values of the properties (they don't show the current values as an app runs)

The Canvas component has numerous properties of various types. For instance, the `BackgroundColor` and `PaintColor` are memory slots that hold a color. The `BackgroundImage` holds a filename (*kitty.png*). The `Visible` property holds a *Boolean*

value (true or false, depending on whether the box is checked). The width and height slots hold a number or a special designation (e.g., “Fill parent”).

When you set a property in the Component Designer, you are specifying the initial value of the property—its value when the app first begins. Property values also can be changed as the app runs, with blocks. Yet, the values shown in the Component Designer, such as those in *Figure 16-1*, don’t change; these always show only the initial values. This can be confusing when you test an app—the current value of the app’s properties are not visible.

## Defining Variables

Like properties, variables are named memory slots, but they are not associated with a particular component. You define a variable when your app needs to remember something that is not being stored within a component property. For example, a game app might need to remember what level the user has attained. If the level number were going to appear in a Label component, you might not need a variable, because you could just store the level in the Text property of the Label component. But if the level number is not something the user will see, you’d define a variable in which to store it.

The Presidents Quiz (*Chapter 8*) is another example of an app that needs a variable. In that app, only one question of the quiz should appear at a time in the user interface, yet the quiz has many questions (most of which are kept hidden from the user at anytime). Thus, you need to define a variable to store the list of questions.

Whereas properties are created automatically when you drag a component into the Designer, variables are defined explicitly in the Blocks Editor by dragging out an initialize global block. You can name the variable by clicking the text “name” within the block, and you can specify an initial value for it by dragging out a number, text, color, or make a list block and plugging it in. Here are the steps you’d follow to create a variable called score with an initial value of 0:

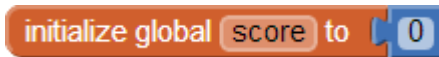
1. In Built-in blocks, open the Variables drawer and drag out the initialize global block.



2. Change the name of the variable by clicking the text “name” and typing “score”.



3. From the Math drawer, drag out a number block and plug it into the open socket of the variable definition to set the initial value.

An orange Scratch block with the text "initialize global score to" and a blue number block containing the value "0".

When you define a variable, you instruct the app to set up a named memory slot for storing a value. These memory slots, as with properties, are not visible to the user when the app runs.

The number block you plug in specifies the value that should be placed in the slot when the app begins. Besides initializing with numbers or text, you can also initialize the variable with a `make a list` or `create empty list` block. This informs the app that the variable will store a list of memory slots instead of a single value. To learn more about lists, see *Chapter 19*.

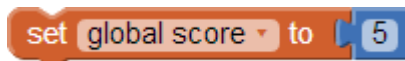
## Setting and Getting a Variable

When you define a variable, App Inventor creates two blocks for it: set and get. You can access these blocks by hovering over the variable name in the initialization block, as shown in *Figure 16-2*.



**Figure 16-3.** The initialization block contains set and get blocks for that variable

The `set global to` block lets you modify the value stored in the variable. For instance, the number block in *Figure 16-3* places the value 5 in the variable `score`. The term “global” in the `set global score to` block refers to the fact that the variable can be used in all of the program’s event handlers and procedures. With the newest version of App Inventor, you can also define variables that are “local” to a particular procedure or event handler—that is, local variables can be used only by the procedure or event with which they’re associated (more on this a little later in the chapter).

An orange Scratch block with the text "set global score to" and a blue number block containing the value "5".

**Figure 16-4.** Placing a number 5 into the variable `score`

You use the block labeled `get global score` to retrieve the value of a variable. For instance, if you wanted to check if the value inside the memory slot was greater than

100, you'd plug the `get global score` block into an `if` test, as demonstrated in *Figure 16-4*.

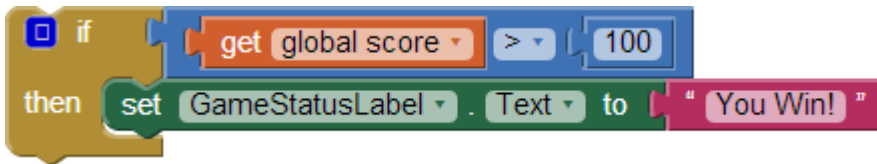


Figure 16-5. Using the global score block to get the value stored in the variable

## Setting a Variable to an Expression

As you've seen, you can put simple values such as 5 into a variable, but often you'll set the variable to a more complex *expression* (expression is the computer-science term for a formula). For example, when the user clicks Next to get to the next question in a quiz app, you'll need to set the `currentQuestion` variable to *one more than its current value*. When someone loses ten points in a game app, you need to modify the score variable to *10 less than its current value*. In a game like MoleMash (*Chapter 3*), you change the horizontal (*x*) location of the mole to *a random position within a canvas*. You'll build such expressions with a set of blocks that plug into a `set global to` block.

## Incrementing a Variable

Perhaps the most common expression is for *incrementing* a variable, or setting a variable based on its own current value. For instance, in a game, when a player scores a point, the variable `score` can be incremented by 5. *Figure 16-5* shows the blocks to implement this behavior.



Figure 16-6. Incrementing the variable score by 5

If you can understand these kinds of blocks, you're well on your way to becoming a programmer. You read these blocks as "set the score to five more than it already is," which is another way to say *increment* your variable. The way it works is that the blocks are interpreted inside out, not left to right. Thus, the innermost blocks—the `get global score` and the number 5 block—are evaluated first. Then, the + block is performed and the result is "set" into the variable score.

Suppose that there were a 10 in the memory slot for score before these blocks; the app would perform the following steps:

1. Retrieve the 10 from score's memory slot (evaluate the get block).
2. Add 5 to it to get 15.
3. Place the result, 15, into score's memory slot (performing the set).

## Building Complex Expressions

In the Math drawer, App Inventor provides a wide range of mathematical functions similar to those you'd find in a spreadsheet or calculator. There are arithmetic operators (e.g., +, -, \*, /), blocks for generating random values, and operators such as sqrt, cosine, and sine.

You can use these blocks to build a complex expression and then plug them in as the *righthand-side expression* of a set global to block. For example, to move an image sprite to a random column within the bounds of a canvas, you'd configure an expression consisting of a multiply (\*) block, a subtract (-) block, a Canvas1.Width property, an ImageSprite1.Width property, and a random fraction block, as illustrated in Figure 16-6.



Figure 16-7. You can use Math blocks to build complex expressions like this one

As with the increment example in the previous section, the blocks are interpreted by the app in an inside-out fashion. Suppose that the Canvas has a width of 300 and the ImageSprite has a width of 50, the app would perform the following steps:

1. Retrieve the 300 and the 50 from the memory slots for Canvas1.Width and ImageSprite.Width, respectively.
2. Subtract:  $300 - 50 = 250$ .
3. Call the random fraction function to get a number between 0 and 1 (say, .5).
4. Multiply:  $250 * .5 = 125$ .
5. Place the 125 into the memory slot for the ImageSprite1.X property.

## Displaying Variables

When you modify a component property, as in the preceding example, the user interface is directly affected. This is not true for variables; changing a variable has no direct effect on the app's appearance. If you just incremented a variable score but didn't modify the user interface in some other way, the user would never know there was a change. It's like the proverbial tree falling in the forest: if nobody was there to hear it, did it really happen?

Sometimes, you do not want to immediately manifest a change to the user interface when a variable changes. For instance, in a game you might track statistics (e.g., missed shots) that will only appear when the game ends.

This is one of the advantages of storing data in a variable as opposed to a component property: you can show just the data you want when you want to show it. You can also separate the computational part of your app from the user interface, making it easier to change that user interface later.

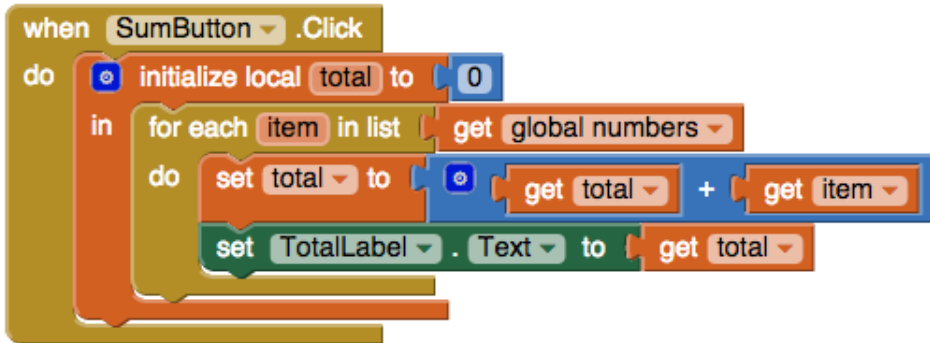
For example, with a game, you could store the score directly in a Label or in a variable. If you store it in a Label, you'd increment the Label's Text property when points were scored, and the user would see the change directly. If you stored the score in a variable and incremented the variable when points were scored, you'd need to include blocks to also move the value of the variable into a label.

However, if you decided to change the app to display the score in a different manner, perhaps with a slider, the variable solution would be easier to change. You wouldn't need to modify all the places that change the score; you'd only need to modify the blocks that display the score.

## Local Variables

The variables described in this chapter thus far are *global* variables and you define them with an `initialize global to` block. The "global" refers to the fact that the variable can be used in all event handlers and procedures. Such variables are said to have *global scope*.

With the latest version of App Inventor, you can now also define `local` variables, that is, variables whose use (scope) is restricted to a single event handler or procedure (see *Figure 16-7*).



**Figure 16-8.** The variable “total” is local; it can only be used in the SumButton.Click event

If the variable is only needed in one place, it is a good idea to define it as a local, as the variable “total” is in *Figure 16-7*. By doing so, you limit the dependencies in your app and ensure that you won’t mistakenly modify a variable. Think of a local variable like the private memory in your brain—you certainly don’t want other brains to have access to it!

## Summary

When an app is launched, it begins executing its operations and responding to events that occur. When responding to events, the app sometimes needs to remember things. For a game, this might be each player’s score or the direction in which an object is moving.

Your app remembers things within component properties, but when you need additional memory slots not associated with a component, you can define variables. You can store values into a variable and retrieve the current value, just like you do with properties.

As with property values, variable values are not visible to the end user. If you want the end user to see the information stored in a variable, you add blocks that display that information in a label or another user interface component.