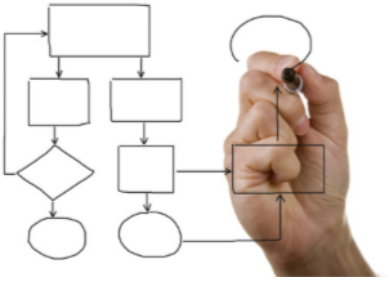
CHAPTER 18

Programming Your App to Make Decisions: Conditional Blocks

Computers, even small ones like the phone in your pocket, are good at performing millions of operations in a single second. Even more impressively, they can also make decisions based on the data in their memory banks and logic specified by the programmer. This decision-making capability is probably the key ingredient of what people think of as artificial intelligence, and it's definitely a very important part of creating smart, interesting apps! In this chapter, we'll explore how to build this decision-making logic into your apps.

Figure 18-1.



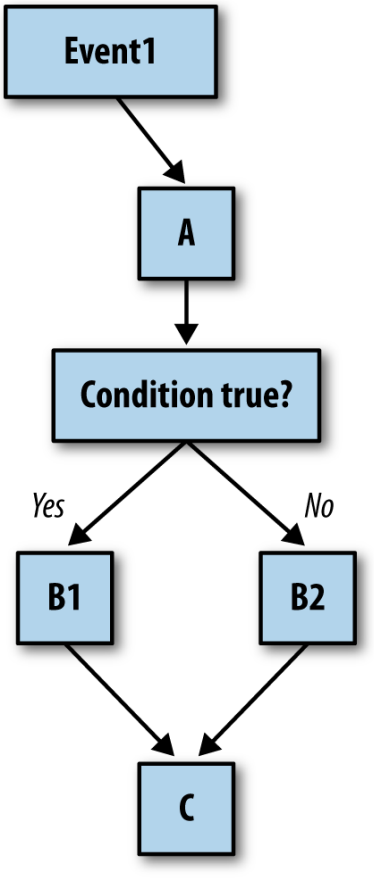


Figure 18-2. An event handler that tests for a condition and branches accordingly

Chapter 14 discusses how an app's behavior is defined by a set of event handlers. Each event handler executes specific functions in response to a particular event. The response need not be a linear sequence of functions, however; you can specify that some functions be performed only under certain conditions. For example, a game app might check if a player's score has reached 100, or a location-aware app might ask if the phone is within the boundaries of some building. Your app can ask such questions and, depending on the answer, proceed accordingly.

Consider the diagram in *Figure 18-1*. When the event occurs, function (block) A is performed. Then, a decision test is performed. If the test is true, B1 is performed. If it is false, B2 is performed. In either case, the rest of the event handler (C) is completed.

Because app decision diagrams like this one look something like trees, we say that the app "branches" one way or the other depending on the test result. So, in this instance, you'd say, "If the test is true, the branch containing B1 is performed."

Testing Conditions with if and else if Blocks

To allow conditional branching, App Inventor provides an `if-then` conditional block in the Control drawer. You can extend the block with as many `else` and `else if` branches as you'd like by clicking the blue icon, as shown in *Figure 18-2*.

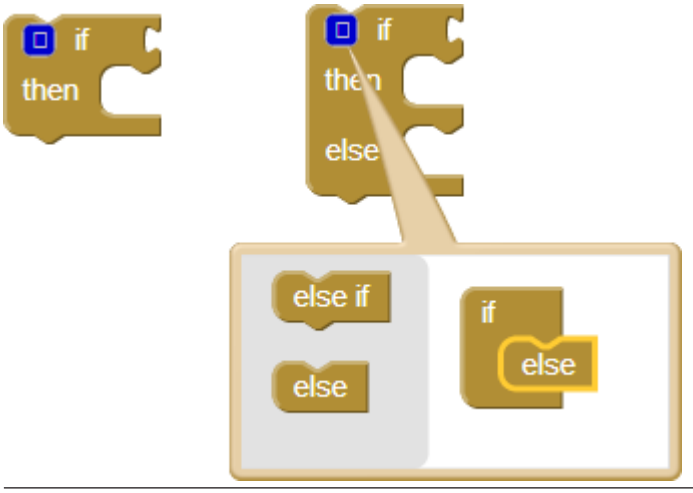


Figure 18-3. The if and else if conditional blocks

You can plug any *Boolean expression* into the *test* sockets of the `if` and `else if` blocks. A Boolean expression is a mathematical equation that returns a result of either true or false. The expression tests the value of properties and variables by using relational and logical operators such as those shown in *Figure 18-3*.

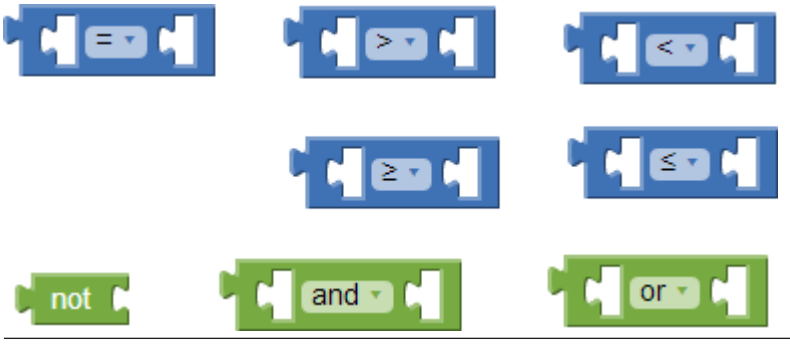


Figure 18-4. Relational and logical operator blocks used in conditional tests

The blocks you put within the "then" socket of an `if` block will only be executed if the test is true. If the test is false, the app moves on to the ensuing blocks.

For a game, you might plug in a Boolean expression for checking a player's score, as shown in *Figure 18-4*.

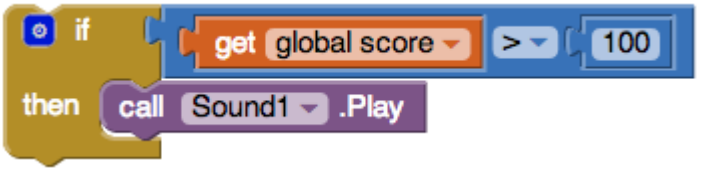


Figure 18-5. A Boolean expression used to test the value of the variable score

In this example, a sound file is played if the score is greater than 100. In this example, if the test is false, the sound isn't played and the app jumps below the entire `if-then` block and moves on to the next block in your app. If you want a false test to trigger an action, you can use an `else` or `else if` block.

Programming an Either/Or Decision

Consider an app that you could use when you're bored: you press a button on your phone, and it calls a random friend. In *Figure 18-5*, a `random integer` block is used to generate a random number and then an `if else` block calls a particular phone number based on that random number.

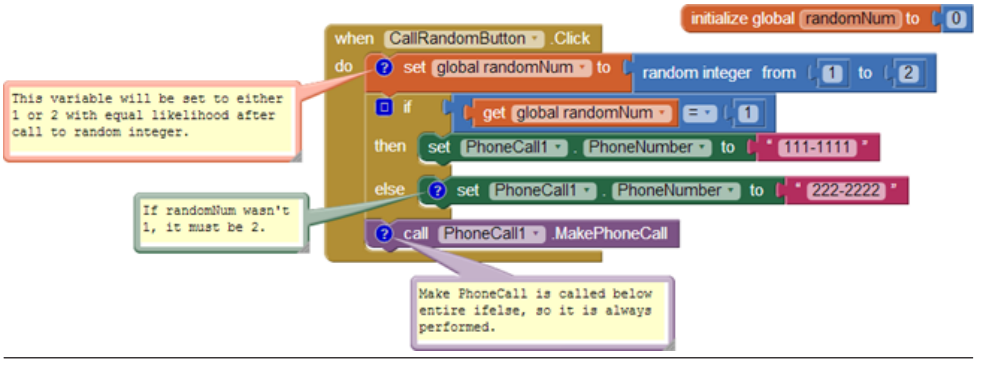


Figure 18-6. This else if block calls one of two numbers based on the randomly generated integer

In this example, `random integer` is called with arguments 1 and 2, meaning that the returned random number will be 1 or 2 with equal likelihood. The variable `RandomNum` stores the random number returned.

After setting `RandomNum`, the blocks compare it to the number 1 in the `if` test. If the value of `RandomNum` is 1, the app takes the first branch (`then`), and the phone number is set to 111–1111. If the value is not 1, then the test is false, in which case the app takes the second branch (`else`), and the phone number is set to 222–2222. The app makes

the phone call either way because the call to `MakePhoneCall` is below the entire `if else` block.

Programming Conditions Within Conditions

Many decision situations have more than just two outcomes from which to choose. For example, you might want to choose between more than two friends in your Random Call program. To do this, you could place an `else if` prior to the original `else` branch, as demonstrated in *Figure 18-6*.

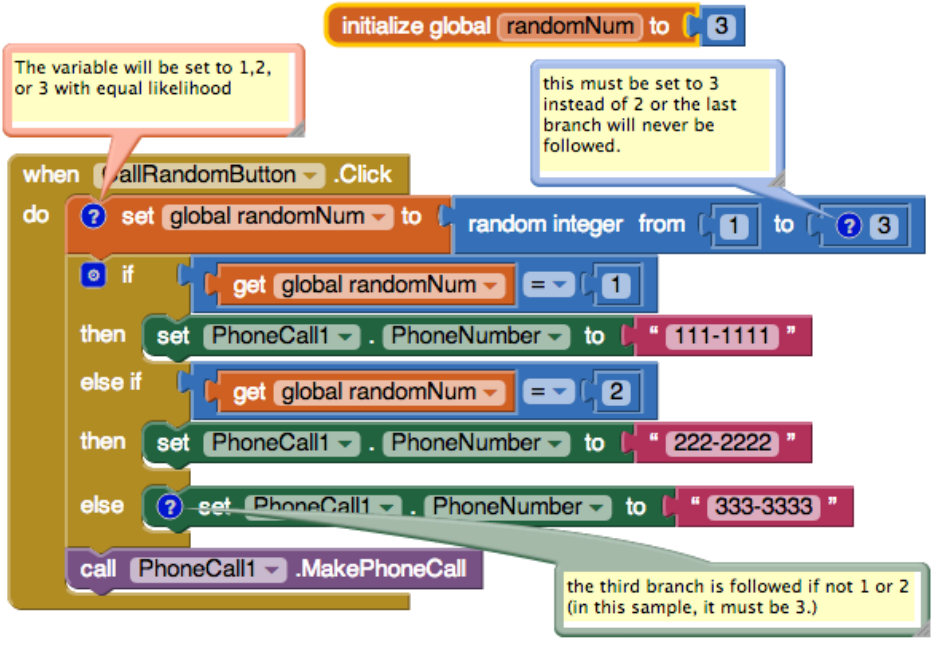


Figure 18-7. `if, else if` and `else` provide three possible branches

With these blocks, if the first test is true, the app executes the first then-do branch and calls the number 111–1111. If the first test is false, the `else if` branch is executed, which immediately runs another test. So, if the first test (`RandomNum=1`) is false and the second (`RandomNum=2`) is true, the second branch is executed and 222–2222 is called. If both tests are false, `else` branch at the bottom is executed and the third number (333–3333) is called.

Note that this modification only works because the `to` parameter of the `random integer` call was changed to 3 so that 1, 2, or 3 is generated with equal likelihood.

You can add as many `else if` branches as you'd like. You can also nest conditionals within conditionals. When conditional tests are placed within branches of another conditional test, we say they are *nested*. You can nest conditionals and other *control constructs* such as `for each` loops to arbitrary levels in order to add complexity to your app.

Programming Complex Conditions

Besides nesting conditionals, you can also specify single conditional tests that are more complex than a simple equality test. For example, consider an app that vibrates when your phone (and presumably you!) leave a building or some boundary. Such an app might be used by a person on probation to warn him when he strays too far from his legal boundaries. Parents might use it to monitor their children's whereabouts. A teacher might use it to automatically take roll (if all her students have an Android phone!).

For this example, let's ask this question: is the phone within the boundary of Harney Science Center at the University of San Francisco? Such an app would require a complex test consisting of four different questions:

- Is the phone's latitude less than the maximum latitude (37.78034) of the boundary?
- Is the phone's longitude less than the maximum longitude (–122.45027) of the boundary?
- Is the phone's latitude more than the minimum latitude (37.78016) of the boundary?
- Is the phone's longitude more than the minimum longitude (–122.45059) of the boundary?

You need the `LocationSensor` component for this example. You should be able to follow along here even if you haven't been exposed to `LocationSensor`, but you can learn more about it in *Chapter 23*.

You can build complex tests by using the logical operators `and`, `or`, and `not`, which you can find in the Logic drawer. In this case, you drag out an `if` block and some `and` blocks, place one of the `and` blocks within the "test" socket of the `if`, and the others within the first `and` block, as illustrated in *Figure 18-7*.

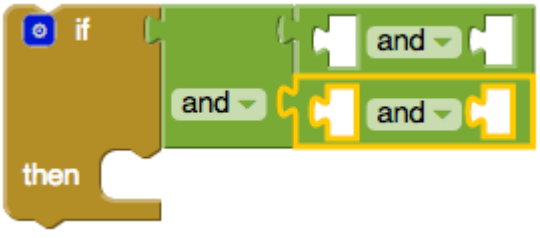


Figure 18-8. An `if` test can test many conditions using `and`, `or`, and other relational blocks

You'd then drag out blocks for the first question and place them into the first block's "test" socket, as shown in *Figure 18-8*.

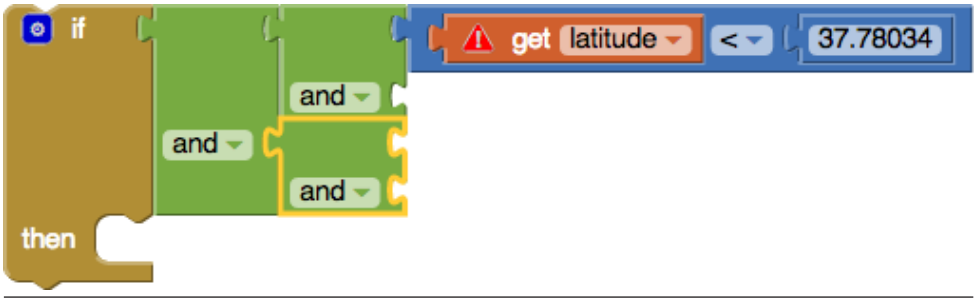


Figure 18-9. Blocks for the first test are placed into the `and` block

You can then fill the other sockets with the other tests and place the entire `if` within a `LocationSensor.LocationChanged` event. You now have an event handler that checks the boundary, as illustrated in *Figure 18-9*.

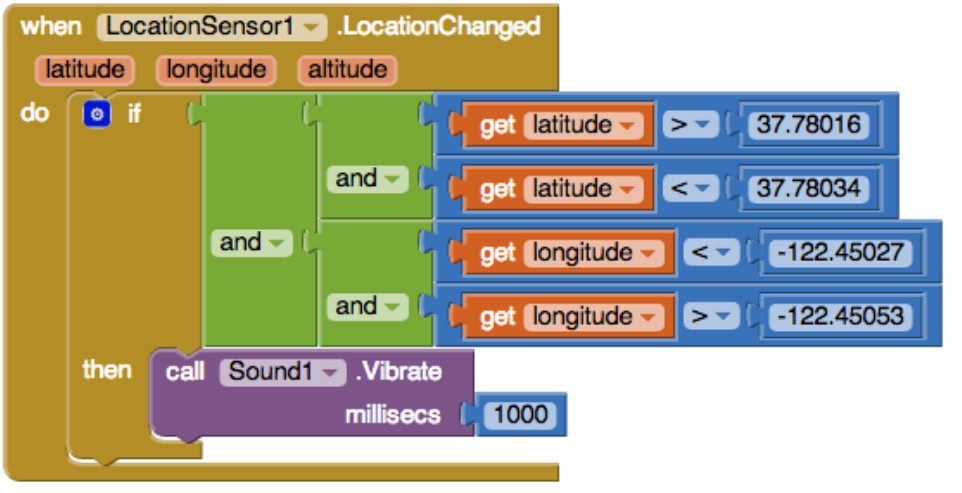


Figure 18-10. This event handler checks the boundary each time the location changes

With these blocks, each time the `LocationSensor` gets a new reading and its location is within the boundary, the phone vibrates.

OK, so far this is pretty cool, but now let's try something even more complicated to give you an idea of the full extent of the app's decision-making powers. What if you wanted the phone to vibrate only when the boundary was crossed from inside to outside? Before moving ahead, think about how you might program such a condition.

Our solution is to define a variable `withinBoundary` that remembers whether the *previous* sensor reading was within the boundary or outside of it, and then compares that to each successive sensor reading. `withinBoundary` is an example of a *Boolean variable*—instead of storing a number or text, it stores true or false. For this example, you'd initialize it to false, as shown in *Figure 18-10*, meaning that the device is not within USF's Harney Science Center.

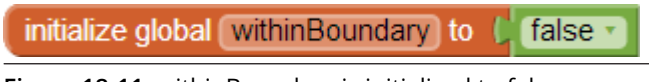


Figure 18-11. withinBoundary is initialized to false

The blocks can now be modified so that the `withinBoundary` variable is set on each location change, and so that the phone vibrates only when it moves from inside to outside the boundary. To put that in terms we can use for blocks, the phone should vibrate when 1) the variable `withinBoundary` is true, meaning the previous reading was inside the boundary, and 2) the new location sensor reading is outside the boundary. *Figure 18-11* shows the updated blocks.

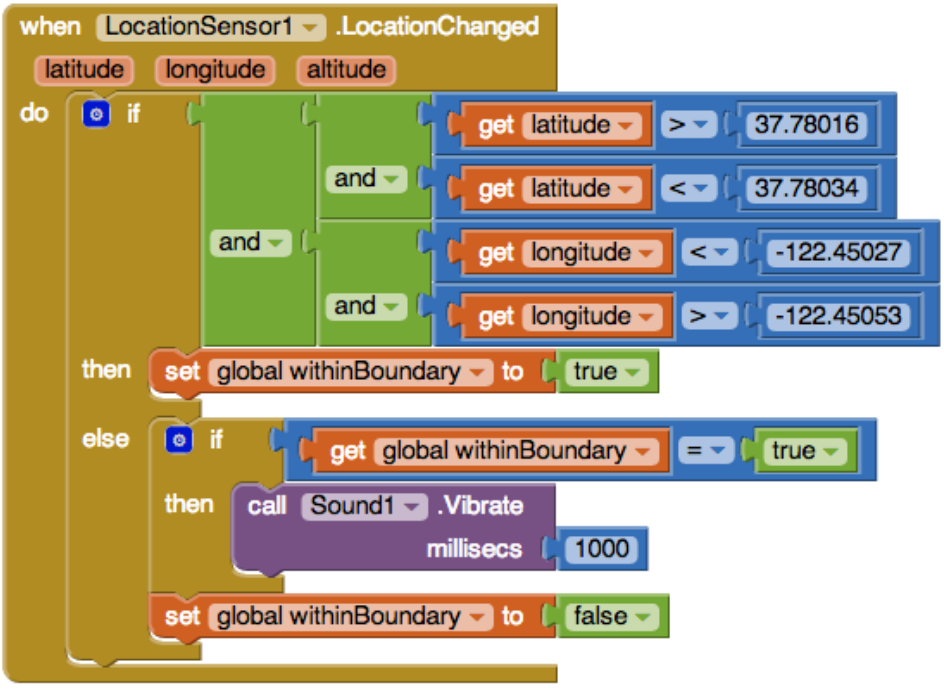


Figure 18-12. These blocks cause the phone to vibrate only when it moves from within the boundary to outside the boundary

Let's examine these blocks more closely. When the `LocationSensor` gets a reading, it first checks if the new reading is within the boundary. If it is, `LocationSensor` sets the `withinBoundary` variable to true. Because we want the phone to vibrate only when we are outside the boundary, no vibration takes place in this first branch.

If we get to the `else`, we know that the new reading is outside the boundary. At that point, we need to check the previous reading: if we're outside the boundary, we want the phone to vibrate only if the previous reading was *inside* the boundary. `withinBoundary` gives us the previous reading, so we can check that. If it is true, we vibrate the phone.

There's one more thing we need to do after we've confirmed that the phone has moved from inside to outside the boundary—can you think of what it is? We also need to reset `withinBoundary` to false so that the phone won't vibrate again on the next sensor reading.

One last note on Boolean variables: check out the two `if` tests in *Figure 18-12*. Are they equivalent?

Figure 18-13. Can you tell whether these two `if` tests are equivalent?

The answer is "yes!" The only difference is that the test on the right is actually the more sophisticated way of asking the question. The test on the left compares the value of a Boolean variable with true. If `withinBoundary` contains `true`, you compare `true` to `true`, which is true. If the variable contains `false`, you compare `false` to `true`, which is false. However, just testing the value of `withinBoundary`, as in the test on the right, gives the same result and is easier to code.

Summary

Is your head spinning? That last behavior was quite complex! But, it's the type of decision making that sophisticated apps need to perform. If you build such behaviors part by part (or branch by branch) and test as you go, you'll find that specifying complex logic—even, dare we say, *artificial intelligence*—is doable. It will make your head hurt and exercise the logical side of your brain quite a bit, but it can also be lots of fun.