Programming the App to Make Decisions: Conditional Blocks
  if and ifelse Blocks
Nested ifelse Sample: Calling a random friend.
Programming Complex Conditions: Location within boundary
Summary
Programming the App to Make Decisions: Conditional Blocks

An app responds to events--its behavior is defined by a set of event-handlers. Each event-handler executes some functions in response to the particular event. But the response is not a linear sequence of functions--some functions are only performed under certain conditions, if a test evaluates to true. An game app might ask if the score has reached 100. A location-aware app might ask if the phone is within the boundaries of some building. The app asks the question, then proceeds down a different program “branch” depending on the answer.

Here’s a flow chart of an event-handler with a conditional:

```
Event
  ↓
  A
    ↓
    test condition
        true  false
          ↓   ↓
          B1  B2
            ↓
            C
```

When the event occurs, function A is executed no matter what. Then a test is performed--a question is asked. If it is true, B1 is executed. If it is false, B2 is executed. In either case, the branches come together and the rest of the the event-handler (C) is completed.

if and ifelse Blocks

App Inventor provides two types of conditional blocks, both of which are found in the Control drawer of the built-in palette:
You’ll fill the then-do part of an if block with function blocks. When an if test is true, these inner blocks are executed. If the test is false, nothing is executed. With an if-else control block, something is always executed, either the blocks within the then-do slot (if the test is true) or the blocks within the else-do (if the test is false).

You can plug any boolean expression into the ‘test’ slot of either of these blocks. A boolean expression is a mathematical equation that evaluates to true or false. The expression tests the value of properties and variables often using the following relational and logical operators:

So for a game, you might plug in a boolean expression concerning the score:

If you wanted to play a song when the score goes over 100, you’d fill the “then-do” slot of the if with a Sound.Play call:

Note that if the test is false (score less than or equal to 100), the action within the if-then block doesn’t happen. If you want a false test to trigger an action, you can use an if-then-else block.
Nested Ifelse Sample: Calling a random friend.

Suppose you wanted an application you could use when you were bored-- you press a button and the phone calls a random friend. One solution is to use a random-integer block to get a random number, then an if-then-else block to set the phone number to call based on the random number:

In this sample, the global variable RandomNum is used to store the random number returned from the random-integer function. This number will be 1 or 2.

After setting RandomNum, the blocks compare it to the number 1 in the ifelse test. If the randomly generated number is 1, the phone number is set to 111-1111. If it is not 1, the test is false, so the else-do blocks are executed and the PhoneNumber is set to 222-2222.

If you wanted to choose between more than two numbers, you could use a nested if-else-- place an if-then or if-then else within the original else clause:
With these blocks, if the first test is true, the first then-do branch is executed and the number 111-1111 is called. If it is false, then the else 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 then-do is executed, and 222-2222 is called. If both tests are false, the else-do branch at the bottom is executed, and the third number (333-3333) is called.

Note that this modification only works because we also modified the *to* parameter of the random integer so that a number between 1 and 3 is generated.

Placing a control construct within another is called nesting-- you’d say that the event-handler above has a "nested if-else". By using multiple levels of nested if-else blocks we can write app behaviors that choose between n choices, instead of just two.

**Programming Complex Conditions: Location within boundary**

Consider an app that vibrating when you leave a building or some boundary-- it might be used by a person on probation or by parents to monitor their children.

Such an app would need to ask a complex question consisting of four tests:

- is the phone’s latitude less than the maximum latitude of the boundary?
- is the phone’s longitude less than the maximum longitude of the boundary?
is the phone’s latitude more than the minimum latitude of the boundary?
is the phone’s longitude more than the minimum longitude of the boundary?

To ask such a complex question, you drag an and block and plug in each test. As you plug in each test into the and block, a new slot opens. The blocks for checking a boundary would look like this one (which roughly checks if the phone is within the University of San Francisco campus):

With these blocks, each time the LocationSensor gets a new reading within the boundary, the phone vibrates. The problem, however, is that if you (the phone) is moving around within the boundary, the phone will vibrate continuously. It might be better for the phone to vibrate only when the boundary is crossed.

One solution is to define a variable “withinBoundary” so we can remember whether the “previous” sensor reading was within the boundary or not. withinBoundary is a boolean variable, meaning that instead of storing a number or text, it stores true or false.

The blocks can now be changed so that the withinBoundary variable is set on each location change, and so that the phone only vibrates when it moves from outside to within the boundary. To put that in terms that we can use for blocks, the phone should vibrate when 1) the location sensor reading is within the boundaries and the variable withinBoundary is false (the previous reading was outside the boundary). Here are the updated blocks: 
Note that withinBoundary is set to true anytime the boundary test evaluates to true, and false anytime it doesn’t. The key change here is that, if withinBoundary was already true, we don’t vibrate on an in-boundary reading.

**Summary**

Conditional blocks allow for the specification of complex behaviors. Using relational operators (e.g., <), logical operators (and, or, not) and nested if-else blocks, you can can program your app to make complex decisions.