Android Programming

with App Inventor for Android

(draft 1, 1/2010)

David Wolber
# Table of Contents

**Chapter 5: Animation**
- Animated Objects
- Timer Events
- Moving an Object Periodically
- Ball/Image Sprite Events
- Reacting to Touch and the Ball Reaching an Edge
- Variables
- Randomness
- Problems

**Chapter 6: Lists**
- Making a List
- Selecting an Item
- Using Length of List
- Iterating Through a List
- Example: Quiz App Version 1
- Iteration 1: Questions and the Next Button
- Iteration 2: Processing Answers
- Summary
- Problems
Chapter 5: Animation

In this chapter, you'll learn how to write a simple animated game and be introduced to timer events, randomness, and global variables.

Animated Objects

App Inventor has two types of animated objects: a Ball and an ImageSprite. A Ball has a restricted appearance, while an ImageSprite can take on any form.

Timer Events

Many of the events that occur in a mobile phone app are end-user initiated, e.g., the user clicks a button. There are other types of events as well, including the Timer event, which is triggered not by the user but by the passing of time.

To add a timer event, you first add a Clock component to the application in the Component Designer. Think of the Clock component as a tiny alarm clock inside your app. You can drag it in from the not ready for primetime pallete, and it is automatically be placed in the 'non-visible component' area, as its not something that will appear on screen.

The Timer component has one important property, its interval. The interval

![Image of Clock component in App Inventor]

The image shows a Clock component placed on the screen. The properties of the Timer component are visible, including its interval set to 2000.
is in milliseconds (1/1000 of a second). If you set the interval to 1000, then a timer event will be triggered every second.

**Moving an Object Periodically**

To illustrate use of the timer, let's consider the most fundamental animation: moving an object across the screen at a consistent pace. Specifically, we'll develop an app in which the ball move horizontally across the screen at 10 pixels a second.

In the Component Designer, drag in a canvas onto the screen, then a ball onto the canvas. The canvas is in the *basic* palette, while the Ball is in the *Animation* palette (along with the Image Sprite). A canvas is required for any animation: you cannot add a ball or image sprite directly on the screen.

After defining the components, open the Blocks Editor, then open the *Clock1* drawer within *My Blocks*. Drag in a *Clock1.Timer* event to specify the activity that should occur on each timer interval:

```
when Clock1.Timer
  do
```

Since the interval for the Clock1 component is set to 1000 milliseconds, the blocks we put into the Clock1.Timer event will be repeated every second.

In our case, we want to move the ball across the screen 10 pixels every second. So we'll open up the palette for *Ball1*, and drag in a *Ball1.MoveTo* operation:

```
call Ball1.MoveTo x y
```

MoveTo moves the ball to a specific place on the canvas, specified with the x and y parameters. How can we use this operation to move the ball right 10 pixels?

Well, another way to characterize, "move the ball right 10 pixels' is, "*change the ball's x property so that it is 10 more than what it was, and keep its y coordinate the same.*" To do this, we'll need to make use of a built-in + block from the Math drawer:
The + block has two slots for its operands. We want to add 10 to Ball1's X property, so we open Ball1's drawer and drag in the X property, then open the Math drawer and drag in a number block, entering 10 as the number. We then slot in the Ball1.X and number 10 into the + block, and the whole + block into the x parameter of the MoveTo block. Finally, to keep the Y property the same, we drag in a Ball1.Y block and place it into the y slot of the MoveTo block. Here's the resulting event-handler:

The blocks read, "On every timer interval, change the x property of the ball to 10 more than it was and leave the y value the same". Because the interval of Clock1 is set to 1000 milliseconds (one second), the ball will move right ten pixels a second.

Using this same scheme, any type of motion is possible. Can you write an app that moves the ball vertically? from bottom to top? Diagonally? All of these can be programmed by setting the initial location of the ball in the component designer, then correctly setting the parameters of the MoveTo.

**Ball/Image Sprite Events**

Both Ball and ImageSprite have a number of events that they respond to:
**EdgeReached** is triggered when an object reaches the edge of its enclosing canvas. Say the canvas has a width of 400 and a height of 300. Then **EdgeReached** is triggered when 1) the object's X property 0 or less or more than 400, or 2) the object's Y property is 0 or less or more than 300.

**CollidedWith** is triggered when when an object runs into another. The parameter other1 tells you which other object is involved. **NoLongerCollidingWith** is triggered when two objects that were touching are no longer in contact.

**Touched** is triggered when the user touches an object on the screen. If the user then drags his finger along the screen, a sequence of **Dragged** events are triggered. **Dragged** is the key event of paint programs.

**Reacting to Touch and the Ball Reaching an Edge**

To illustrate some of the Ball/Image events, consider an app which 1) moves the ball to the center of the screen when the user touches it, and 2) moves the ball to the left-side of the canvas when it reaches an edge.

The first event-handler reacts to the **Touched** event. Since we want the ball to move to the center of the canvas on the touch, we'll need to use the **Canvas.Width** property as well as a division block from the **Math** drawer:

![Diagram](image)

The x parameter of the **MoveTo** is set to 1/2 of the canvas's width. We leave the y parameter (the vertical location) the same.

The second event-handler reacts to the **EdgeReached** event. For this event-handler, we just set the x parameter of **MoveTo** to 1:

![Diagram](image)

With these event-handlers, the ball will move across the screen until it reaches the right-side, at which time it will appear again on the left. If the user touches the ball, it will move to the middle of the screen.
Variables

In programming, the term *variable* refers to a *named memory cell*. It is a place cell in the phone's memory where you can remember a specific data item. These memory cells can be accessed by an app, but they are not directly visible to the end-user of the app.

We've already considered one type of variable-- the properties of a component. As we've seen, these variables have names of the form:

\[
\text{component.property}
\]

For instance, in the last chapter we worked with the PaintColor property of the Canvas component, \textit{Canvas.PaintColor}. This is just a memory cell with a color in it. When the canvas draws something, it checks that memory cell to see what color to draw.

In App Inventor, you can also define variables that are not associated to a particular component. Such variables are called *global variables*. For instance, in many games, you'll need to keep track of the score, so you'll define a global variable named "score". Later, we'll also discuss global variables which are not single numbers but lists of data, and that allow you to remember things like the questions of a quiz.

You define new variables by selecting "def variable" from the \textit{Definition} drawer of the Built-In palette:

If you were setting up the variable to track the score of a game, you'd name it "score" and connect a number to it with an initial setting of 0.

The result of these blocks is to set up a new memory cell within the phone's memory, and put a 0 in it.
When you define a variable, App Inventor adds blocks for getting and modifying its value in the *My Program* drawer of *My Blocks*:

The first block, labeled "global score" allows an app to access the value of a variable. For instance, you could display the score in a Label Component with:

If there was a 0 in the hidden memory cell of score, 0 would be displayed to the end-user in the label component named ScoreLabel.

The block labeled "set global score to" changes the value in the memory cell. For instance, the following:

would change the value in score to one more than it's current value. If it was 0, it would change to 1, 1 to 2, and so on.

In programming, such blocks are called assignment statements, and can be confusing for beginning programmers. The key is that the right-side of the block sequence is evaluated first. The app will first *evaluate* the expression on the right side by grabbing the value of score and adding one to it. The result is then placed back into the variable score (in general, in the variable referenced by the "set global block to" block.

With a variable score, we can extend the ball animation app above so each
Each time the ball is touched, the app moves the ball to the middle of the screen, then adds one to the variable score, and finally updates the text of the visual component `ScoreLabel`.

**Randomness**

Computers can generate random numbers, typically using the computer's real-time clock to seed the generation of the numbers. Such random number generation is essential to many games. For instance, game Mastermind generates a random sequence of colors for every game. The game WhackAMole moves the mole to random spots on the screen.

App Inventor provides a number of blocks for adding randomness to an application. They appear in the Math drawer of the built-in palette:

To write an application that displays a number between one and ten, you
could use the random-integer block:

![Random Integer Block](image)

The random-fraction block returns a decimal number between 0 and 1. Say you wanted to modify the ball animation so that when the ball reaches the right edge, it jumps to a random column on the canvas, instead of the first column. To implement this, you'd multiply a random-fraction block with the width of the canvas:

![Random Fraction Block](image)

If the canvas was 400 pixels, and the random-fraction function happened to return 0.5, the ball would jump to the middle. If the random-fraction function returned .1, the ball would jump to 40 pixels from the left edge.

**Problems**

Write the following programs and document them on your portfolio:

1. Program a ball animation "BallAnimation" application such that:
   - The ball moves diagonally from top-left to bottom-right of the canvas at a rate of 20 pixels per second.
   - When the ball reaches an edge, it appears again in the top-left of the canvas.
   - When the user touches the ball, it is sent to the middle of the canvas.
2. SaveAs BallAnimation into "BallAnimationScore" and:
   - Modify the application so that it keeps score of the number of times the end-user touches the ball. Use a global variable to do it.
   - Modify the application so that when the ball reaches the edge, it jumps to a random column and row on the canvas.
3. Create the game WhackAMole, following the Google tutorial at http://sites.google.com/site/appinventorhelp/tutorials/whackamole. Call yours WhackAnX where X is someone you’d like to whack (use their image).
Chapter 6: Lists

This chapter describes how to work with lists of data in an App Inventor application. You'll use what you learn to build a better text to random friend application, a text to all friends application, and a question and answer application.

Thus far, in working with variables, we've worked with scalar data, that is, atomic data like a single number or text entry.

Many applications deal with lists of data. For instance, we might want to keep track of the list of monsters in a game or a list of some friends' phone numbers:

PhoneNumberList
3219872 4153297878 4592371

App Inventor provides blocks for creating lists, adding elements to lists, selecting a particular item from a list, and applying operations to an entire list.

Making a List

In App Inventor, lists, like numbers and text, are "hidden" variables, ones not seen by the user.

A list is created in the blocks editor using a variable block and a make a list block. Suppose, for instance, that we're writing an app to text a list of friends with one click. We'd create the phone numbers list in the following manner:

1. From the Built-In palette, drag a "define variable" block into the program area and change the name from "variable" to "phoneNumbers".
2. From the Lists palette, drag a "make a list" block in and latch it to the variable. This tells App Inventor that the variable will store a list of data as opposed to a single scalar.
3. Since the elements of our list are phone numbers, drag in some text blocks, enter the desired numbers, and connect them to the "item" slots in the make a list block. Note that a new "item" slot opens up each time you add a new element to the list.
Your blocks should look something like this:

![Diagram of blocks]

Note that this definition block can float by itself in the block editor, just like when you defined numbers or text variables. It does not need to be hooked into an event handler.

With the blocks above, we've defined a hidden variable named "phone numbers" which is of type "list" and which has two elements, each of which is of type text. Now we'll show how you can write event-handling blocks that use this new list variable.

**Selecting an Item**

You can access particular items in a list using an index. If a list has five items, you can access the items with indices 1, 2, 3, 4, and 5. With traditional programming, you'd write code such as:

```python
call phoneNumbers[2]
```

to access the second item in a list. With App Inventor, you use the **select list item** block:

![Diagram of select list item block]

With select list item, you plug in the list you want to choose from in the first slot, and the index you want in the second slot. The blocks above say to select the second element of the list phoneNumbers. The return value of select list item for this sample would be "3779199" as that is the second element of phoneNumbers.

With this, we are ready for a more elegant version of our text random friend application. Recall that in that application, we used random integer function to get a random number, then programmed it so that if the random number was 1, we'd call one friend, if it was 2, another friend, and so on.

Using a list, we can instead just use the random integer we get as an index
We get a random number between 1 and 2, as our list has only two items. We then call select list item with the random number (1 or 2) as the index. The selected phone number is then set as the texting object's phone number, and the message "miss you" is sent to it. If you run this app multiple times, it should text the two numbers with equal likelihood.

**Using Length of List**

One reason the above solution is more elegant is that we don't have to change the event-handler much in order to make the app work for a different set of phone numbers. With the old if-else solution, we'd have to add another if-else block if we wanted a new phone number added. For the new solution, we can just add an element to the make a list block, e.g.,

There is another detail that must be taken care of, however. When we get the random integer, we only get a number between one and two, so the new third element will never be chosen.

A simple solution would be to also change the parameter to random (e.g., change the upper limit to three). However, its better to avoid such dependencies as programmers are notorious for forgetting. We do not want to introduce bugs when we change an app.
The better solution is to make the event-handling code more general. Instead of putting a fixed upper limit in the random call, we can specify that the upper limit for the call to random should be the size of the list: if there are \( n \) items in the list, get a random number between 1 and \( n \). Fortunately, App Inventor provides a list operation to help: `length`. Here is the modified program:

The change is near the right: the "to" parameter to random integer is no longer a fixed number. Instead, it is the result of the function "length of list".

This solution is maintainable-- the programmer can make the app work for more or less numbers by simply modifying the numbers slotted into "make a list". It will also work for apps that have dynamic lists, i.e., apps in which the user (not the programmer) is allowed to add or remove numbers from the list. This is important given that, generally speaking, you want to write apps for others, not just for yourself!

**Iterating Through a List**

Now suppose we wanted another button for the app and that when the user clicked it, the text would be sent to all the numbers in the list, not just a random one.

A brute force method for such an event-handler would look like:
With this method, the blocks to set the phone number and send the message are copied 3 times. For three numbers, this isn't too awful, but what if we wanted to text 100 or 1000 numbers? The program would be quite large. Furthermore, the programmer would have to add three blocks in order to add a new phone number to the one's called.

Fortunately, we can use our "phonenumbers" list and a special block "for each" that allows us to apply some operations to all elements of a list. Here is the more elegant solution:

This code says that "for each phone number 'pnum' in the list 'phonenumbers', set the text object's phone number to pnum and send out the text message".

The foreach block can be found in the control drawer, the same place where the if-else is. When you drag a foreach block into the program area, you
must specify the list that will be processed as the "in list" parameter at the bottom of the for-each block. In this case, the list is the global "phonenumbers".

At the top of the foreach block, you can also provide a name for the placeholder variable. Often, you'll name it "item", as this variable represents an item in the list. In this case, we named it "pnum" as it represents one of the phone numbers in the list.

The foreach block is one method of iterating in a program. To iterate means to repeat some operations over and over. The blocks that are slotted into the foreach block's "belly" will be repeated for each item in the list. In this sample, the blocks that are repeated are:

![Diagram](image)

We say that those blocks are subordinate to or within the foreach block. We say that the program "loops" back up when it reaches the bottom block within the for-each.

The app executes in the following manner: when the TextAllButton is clicked and the event-handler called, the first operation executed is the "set Texting1.Message to" block, which sets the message to "miss you". This block is only executed once.

The for-each block is then entered. The variable pnum is set to the first number in the list phonenumbers ("3594787"), and the blocks within the foreach are executed for the first time. The Texting1 object's PhoneNumber property is set to pnum ("3594787"), and the message is sent.

Because we are within a foreach, the app "loops" back up to the top of the foreach and puts the next item in the list ("3779199") into pnum. The two operations within the foreach are repeated, so the text is sent to "3779199".

The app then loops back up again and sets pnum to the last item in the list ("8143549"). The operations are repeated a third time causing the third text to be sent.

This time, when the app loops back up, it realized there are no more items to process. The app "pops out of the loop" and continues on after the foreach block. In this case there are no blocks below it, so the event-handler ends.
This solution is maintainable because it will work even if new numbers are added to the phonenumber list-- the foreach will process however many items are in the list.

Example: Quiz App Version 1

Specification: The quiz app will display a single question at a time. The user enters an answer and clicks the "submit answer" button. The app responds by either displaying "correct" in the reply label, or displaying "incorrect: the answer is x". The user clicks on a next button to see the next question.

UI Design: We'll begin with a simple user interface: a question label where the question will be displayed, and AnswerBox where the user will answer, a SubmitButton that the user clicks when answer is in, a ResponseLabel that will display "correct" or "incorrect", and a NextQuestion button that the user clicks to get to the next question.

Iterative Development Strategy: We'll first develop a version that just lists a question and responds to the next button by displaying the next question. For now, we'll ignore the user's answers.

Iteration 1: Questions and the Next Button.

In the blocks editor, our first task is to create a list of the questions. We'll drag in a define variable block and a make a list block, and populate the list with some questions:
Next, we want to get the first question in the list to display when the app begins. We'll use an Initialize event block of a component, in this case the QuestionLabel, to define this behavior. Remember that Initialize is called on each component as the app is starting up, so whatever we do here will appear when the user first sees the app. Here are the blocks:

We use the select list item block to grab the first item from the questions list (note that the list parameter is set to "questions" and the index to 1). The result, in this case "who wrote Remix?" will be displayed in the QuestionLabel.

Of course, we could have put the text "who wrote Remix?" directly in the QuestionLabel.Text. But we want to write the behaviors so that they will work on any list, and will work even if the list is rearranged. The more general "select the first item of the list and put it in the QuestionLabel" code allows for this.

Our next task is to program the behavior of the next button. To do this, we need to remember what question we are on at all times. To remember something, we just define a new variable. In this case, we need to remember the question number, i.e., the index into the list "questions":

Note that this variable definition, like all of them, should be placed outside the confines of any event-handler, just floating by itself in the program area.

Now that we have the questionNum variable, we can program the
NextButton event handler. What we want to do is to increment the questionNum variable, e.g., change it from 1 to 2 or from 2 to 3, and then display the next question. Here are the blocks:

This is some sophisticated code. The first operation sets the questionNum variable to itself plus 1. This is typically quite confusing for beginning programmers. The way to conceptualize it is to first realize that the variable questionNum represents a memory cell with some number in it:

```
questionNum
1
```

The user of the program can't see it, but that memory cell is in the phone's memory because of the questionNum variable definition we performed earlier.

The second thing to realize is that you have to read sequences of blocks right-to-left. For the blocks:

```
set global questionNum to global questionNum + number 1
```

the + operation is executed first. It says to take the value in questionNum and add 1 to it. When the program first starts, the value of questionNum is 1, so by adding 1 we'll get 2 as the result of the + operation.

The result of the plus, 2, is then slotted into the `set global questionNum to` block, i.e., 2 is put into questionNum. This causes the hidden memory cell to no longer have a 1, but instead have a 2:

```
questionNum
2
```
Sometimes when we're tracing a program on paper to understand how it works, we'll cross out the old value and place the new one after it:

```
questionNum
1
2
```

and we'll leave off the box around the memory cell. This shows the history of execution, which can be helpful. But always remember that a memory cell holds one thing only, and the 1 is gone as soon as we (re-) set the variable `questionNum`.

So the first operation in the event-handler changes the hidden variable `questionNum` to one bigger. The second operation:

then uses this modified variable `questionNum` as the index into the list questions. Just like we might say, "get the 2nd item from a list", or "get the third item from a list", these blocks read as "get the 'questionNumth' item from the list. The app, when executing the `select list item` operation, must first evaluate the index parameter, which is the non-fixed value "questionNum". There could be anything in `questionNum`, but whatever that number is will be used for the index. If `questionNum` has 2 in it, we'll get the second item from `questions` ("who wrote The Big Switch") and put that into `QuestionLabel`.

The event-handler we have so far would work fine until the user got to the third (last) question. At that point, when the user clicked next, `questionNum` would have a value of three. We would then increment `questionNum`, changing it to four, and the `select item` call would ask for the 4th item in a three item list. This would cause the app to Force Quit, or as programmer say, "bomb out".

So we need to modify the blocks so that, when the NextButton is clicked we do something differently if we are on the last element of the list. Here's a solution:
The if-else block comes to the rescue: if questionNum is equal to the length of the questions list, we reset questionNum back to 1. So if questionNum is 3 and there are three things in the list questions, the test will be true and the app will execute the then-do branch.

If questionNum is 1, 2, or in general any number less than the length of the list, the else-do branch is executed and we increment the questionNum.

Other solutions are of course possible: we could leave the questionNum at its max once its reached or we could gray out (enable=false) the next button once questionNum does reach its limit. The important thing for a programmer is to understand how to convert the specs of an app into blocks like the one in this example.

**Iteration 2: Processing Answers**

This is left as an excercise (see problem 2)

**Summary**

Lists allow us to give a name to related items. Then with list operations we can index into the list to grab particular items and even apply some operations to each element of the list. This ability to group can reduce the size of a program and make it more maintainable.

**Problems**
1. **Sum of a list app**: Create a list "numbers" with at least three integers, then use a foreach block to compute the sum of the numbers in the list. For instance, if the list had the numbers 3,5, and 7, your app should display 15.

2. **Quiz app**: Extend the iteration 1 solution given in the text to create a complete quiz app. You'll need another list, answers, and you'll need another event-handler to handle the AnswerButton.Click event.