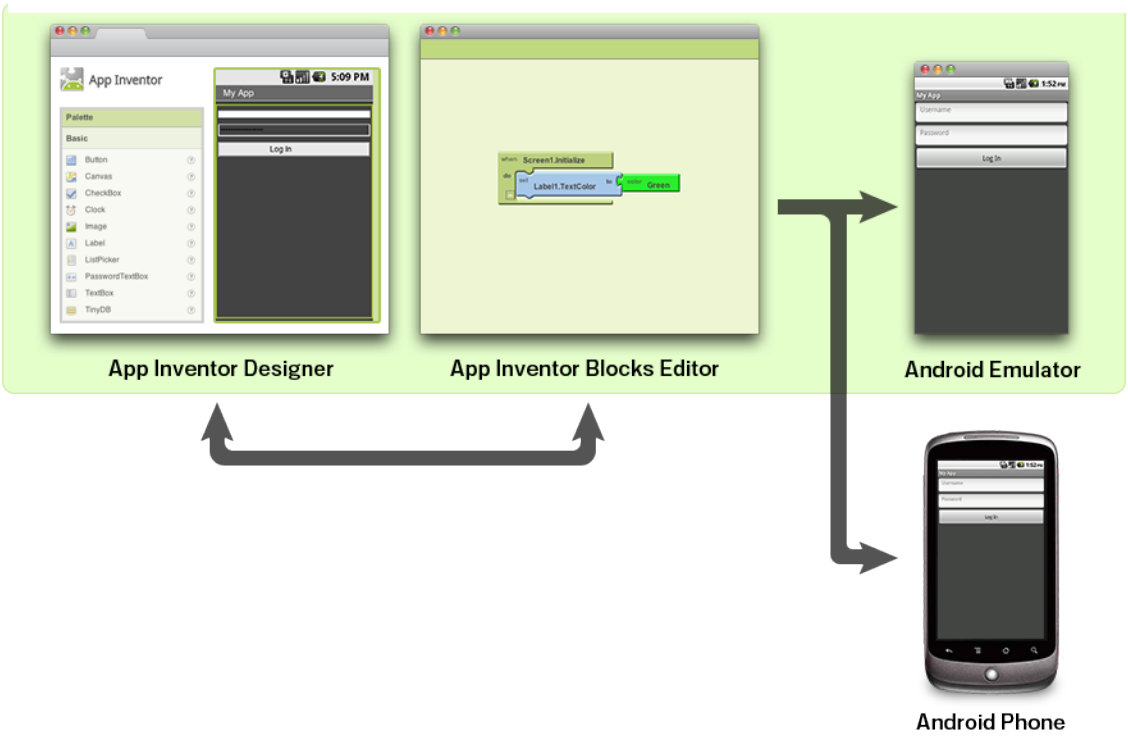
MIT App Inventor Getting Started Guide

What is App Inventor?

App Inventor lets you develop applications for Android phones using a web browser and either a connected phone or an on-screen phone emulator. The MIT App Inventor servers store your work and help you keep track of your projects.



You build apps by working with:

- The *App Inventor Designer*, where you select the components for your app.
- The *App Inventor Blocks Editor*, where you assemble program blocks that specify how the components should behave. You assemble programs visually, fitting pieces together like pieces of a puzzle.

Your app appears on the phone step-by-step as you add pieces to it, so you can test your work as you build. If you don't have an Android phone, you can build your apps using the *Android emulator*, software that runs on your computer and behaves just like the phone.

The App Inventor development environment is supported for Mac OS X, GNU/Linux, and Windows operating systems, and several popular Android phone models. Applications created with App Inventor can be installed on any Android phone.

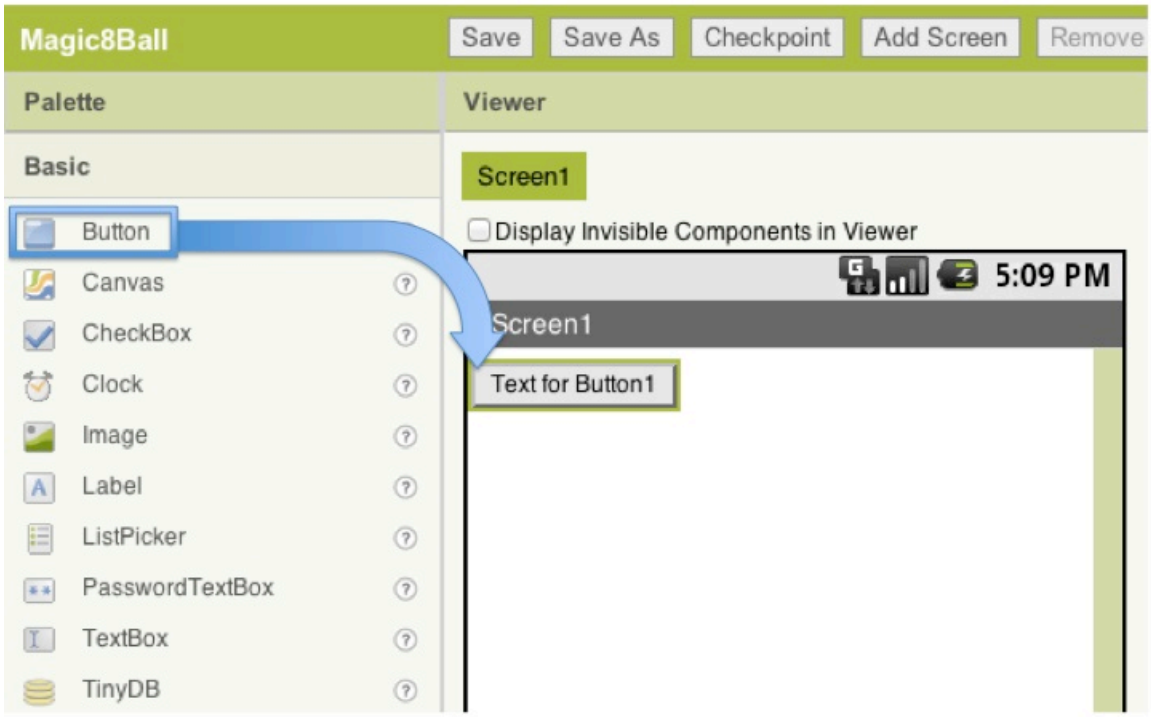
Before you can use App Inventor, you need to set up your computer and install the *App Inventor Setup* package on your computer. See: http://explore.appinventor.mit.edu/content/setup

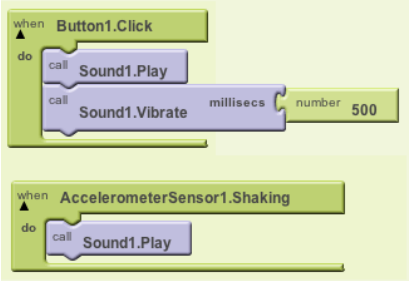
What can I do with App Inventor?

You can build many different types of apps with App Inventor. Often people begin by building games like MoleMash or games that let you draw funny pictures on your friend's faces. You can even make use of the phone's sensors to move a ball through a maze based on tilting the phone.

But app building is not limited to simple games. You can also build apps that inform and educate. You can create a quiz app to help you and your classmates study for a test. With Android's text-to-speech capabilities, you can even have the phone ask the questions aloud.

To use App Inventor, you do not need to be a professional developer. This is because instead of writing code, you visually design the way the app looks and use blocks to specify the app's behavior.





The App Inventor team has created blocks for just about everything you can do with an Android phone, as well as blocks for doing "programming-like" stuff-- blocks to store information, blocks for repeating actions, and blocks to perform actions under certain conditions. There are even blocks to talk to services like Twitter.

Simple but Powerful!

App Inventor is simple to use, but also very powerful. Apps you build can even store data created by users in a database, so you can create a make-a-quiz app in which the teachers can save questions in a quiz for their students to answer.

Because App Inventor provides access to a GPS-location sensor, you can build apps that know where you are. You can build an app to help you remember where you parked your car, an app that shows the location of your friends or colleagues at a concert or conference, or your own custom tour app of your school, workplace, or a museum.

You can write apps that use the phone features of an Android phone. You can write an app that periodically texts "missing you" to your loved ones, or an app "No Text While Driving" that responds to all texts automatically with "sorry, I'm driving and will contact you later". You can even have the app read the incoming texts aloud to you (though this might lure you into responding).

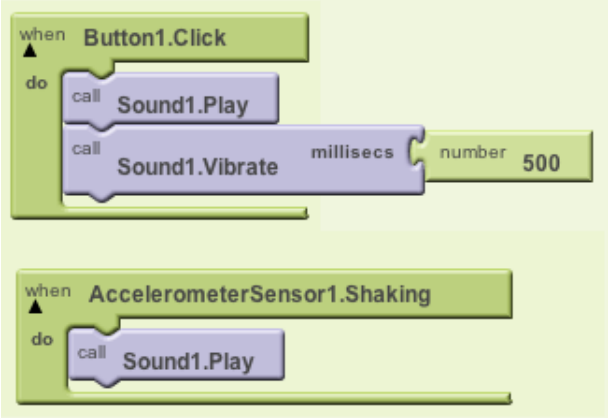
App Inventor provides a way for you to communicate with the web. If you know how to write web apps, you can use App Inventor to write Android apps that talk to your favorite web sites, such as Amazon and Twitter.

Understanding App Inventor Programming

Event Handlers

App Inventor programs describe how the phone should respond to certain events: a button has been pressed, the phone is being shaken, the user is dragging her finger over a canvas, etc. This is specified by **event handler** blocks, which used the word when. E.g., when Button1.Click and when AccelerometerSensor1.Shaking in HelloPurr.

Most event handlers are in green color and stored at the top part of each drawer. Here are the example of event handlers.

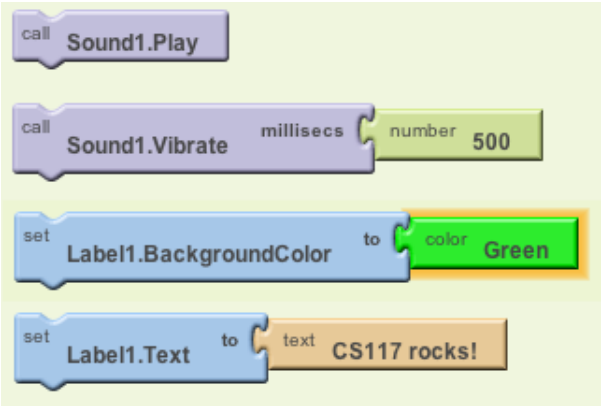


When an event occurs on a phone, the corresponding event handler is said to **fire**, which means it is executed.

Commands and Expressions

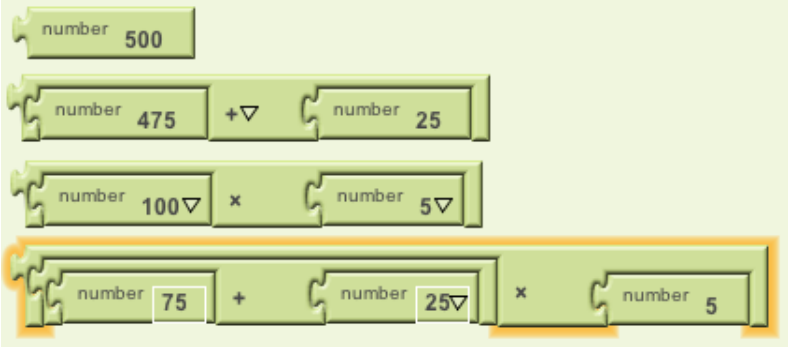
When an event handler fires, it executes a sequence of commands in its **body**. A **command** is a block that specifies an action to be performed on the phone (e.g., playing sounds). Most command blocks are in purple or blue color.

Here are some sample commands available in HelloPurr:

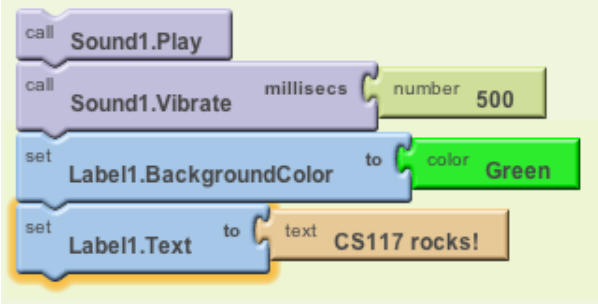


Some commands require one or more **input values** (also known as **parameters** or **arguments**) to completely specify their action. For example, call Sound1.Vibrate needs to know the number of milliseconds to vibrate, set Label1.BackgroundColor needs to know the new background color of the label, and set Label1.text needs to know the new text string for the label. The need for input values is shown by sockets on the right edge of the command.

These sockets can be filled with **expressions**, blocks that denote a value. Expression blocks have **leftward-pointing plugs** that you can imagine transmit the value to the socket. Larger expressions can be built out of simpler ones by horizontal composition. E.g., all of the following expressions denote the number 500:



Commands are shaped so that they naturally compose vertically into a **command stack**, which is just one big command built out of smaller ones. Here's a stack with four commands:



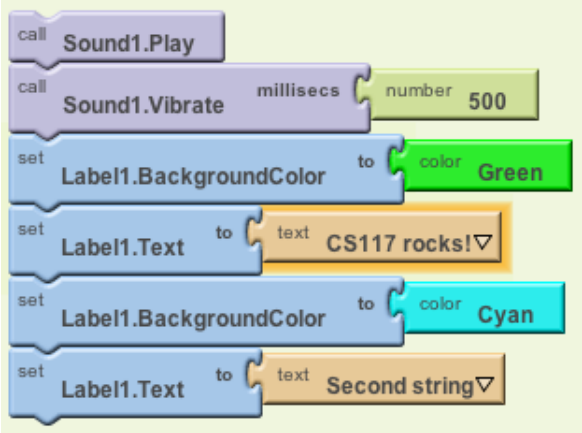
When this stack of commands are placed in a body of an event handler (e.g., the ***when.Button1.Click*** event handler), the command will be executed from the top to the bottom. If the stack of command above is executed, then the phone will first play the sound, then vibrate, then change the label's color to be green, and then label will show the text "CS117 rocks!"

However, the execution works very fast: you would see all the actions happen at the same time.

Control Flow

When an event handler fires, you can imagine that it creates a karaoke-like **control dot** that flows through the command stack in its body. The control dot moves from the top of the stack to the bottom, and when it reaches a command, that command is **executed** -- i.e, the action of that command is performed. Thinking about control "flowing" through a program will help us understand its behavior.

The order of the commands, or the ***control flow*** is important when you make an app. You need to make sure which action should come first.



Arranging Components on the Screen

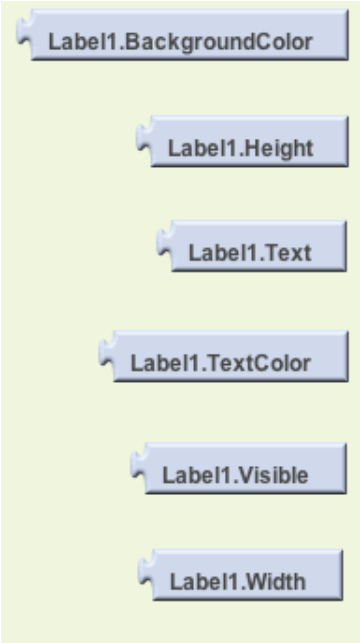
App components are organized vertically by default. In the Designer palette, create a Screen Arrangement>HorizontalArrangement component to organize the three buttons of LabelSize horizontally. (VerticalArrangement and TableArrangement can also be used to control positioning.) Warning: the Designer window Viewer is only an approximation of how the components will look on the phone.

Manipulating Component State

Every component is characterized by various **properties**. What are some properties of a Label component?

The current values of these properties are the **state** of the component. You can specify the initial state of a component in the Properties pane of the Designer window.

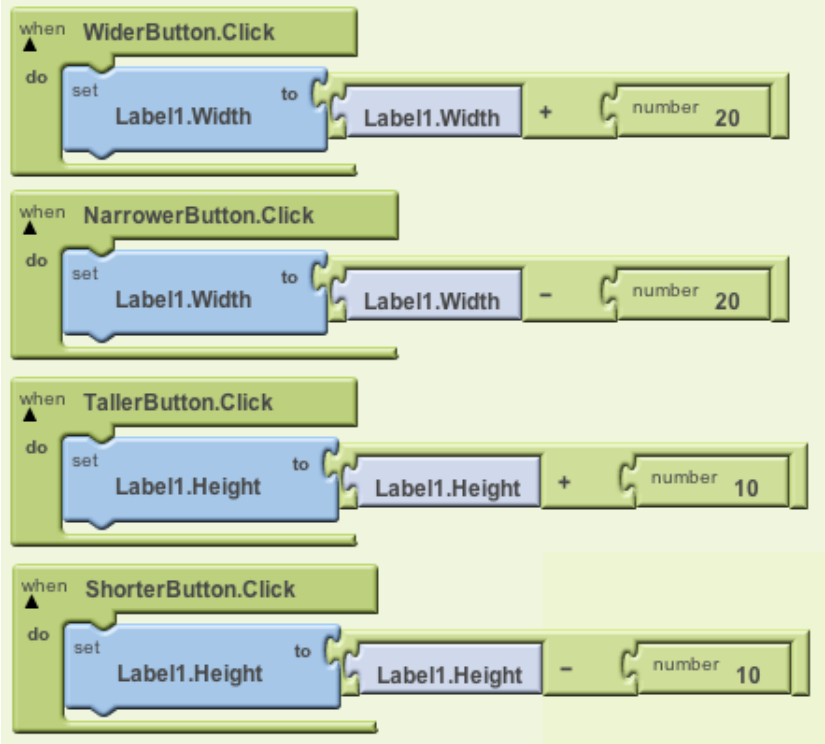
App Inventor programs can get and set most component properties via blocks. E.g., here are blocks for manipulating the state of Label1.

Getter blocks	Setter blocks
	

Getter blocks are expressions that get the current value of the property. Setter blocks are commands that change the value associated with the property.

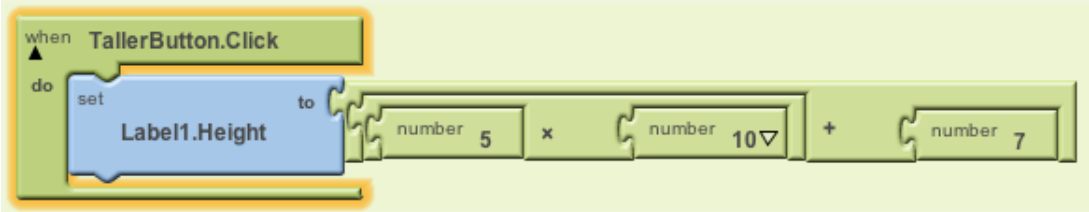
Some Label properties cannot be manipulated by blocks. Which ones?

As an example of manipulating Label properties, open the LabelSize program, which has 4 event handlers. What do these do?

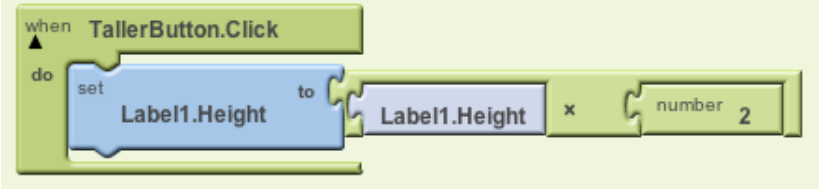


Predict what happens if we change the when TallerButton.Click handler as follows:

Modification 1:



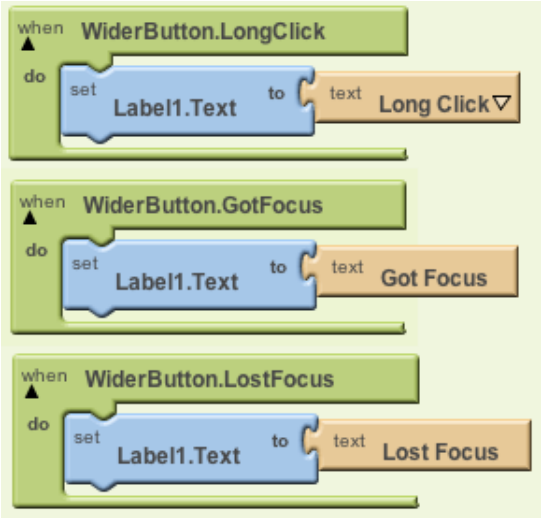
Modification 2:



Modification 3:

Other Button Events

Other button events are when LongClick, when GotFocus, and when LostFocus. Experiment with these by creating the following handlers:



Note: many components get focus when touched, and lose it when the touch is removed. But buttons are special, because touching them fires the Click event. However, they can get/lose focus through the G1 track ball or IDEOS navigator.
Renaming Components

Programs can be easier to read if you change the default name of components.
E.g., NarrowerButton is more meaningful than Button2. In the Components pane of the Designer window, use the Rename button to rename Label1to MyLabel. What happens in the Blocks Editor to blocks that used to mention Label1?

The TextBox Component

Many apps expect users to enter input like numbers or text strings. The TextBox component is used for this purpose.

Let's add two text boxes to the LabelSize program:

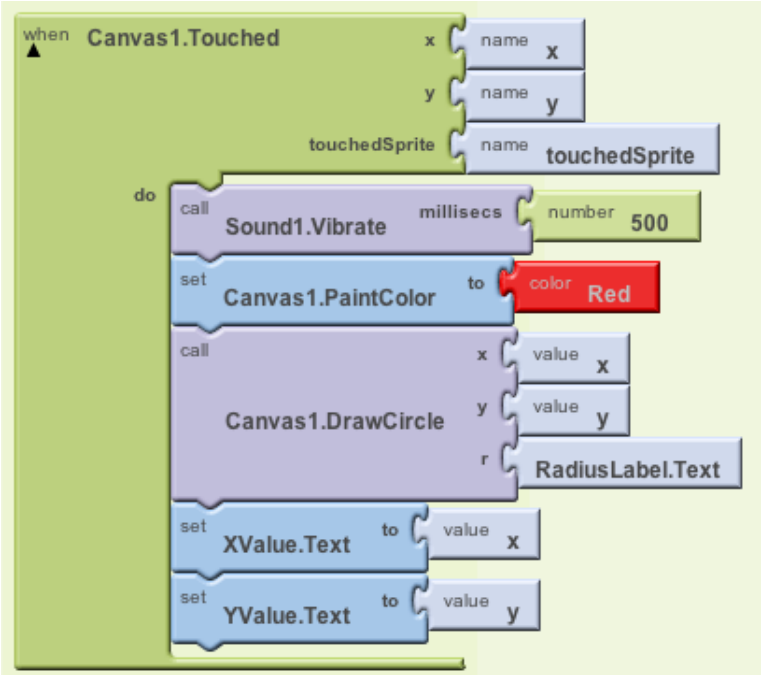
- The first one should specify the amount by which the label should become wider/narrower.
- The second one should specify a string to be concatenated with the current label. (Use the join operator for this.)

Note: Sometimes it is useful to have text strings with multiple lines. In a string, the notation \n stands for the newline character. For example, the text string one\ntwo\nthree has three lines.

The Canvas Component

In PaintPot, you met a new component, Canvas, that is used for drawing and animation.
Upload CanvasTest2.zip (attached at the bottom of this page: https://sites.google.com/site/wellesleycs117fall11/lecture-notes/lecture-04-animation-components) to create the CanvasTest2 app.

This app illustrates the two kinds of event handlers for canvases. The first is the when Touched handler:



Notes:

Canvas1.DrawCircle draws a filled-in circle in the current paint color of the canvas (which is set to red by set Canvas1.PaintColor).

The first two arguments of Canvas1.DrawCircle (annotated x and y) are the x and y coordinates of the upper left corner of the square enclosing the circle. X coordinates grow from 0 (left edge) rightward. Y coordinates grow from 0 (top edge) downward. Coordinates are measure in pixels. Get a sense for the coordinate system by touching the canvas in different spots and observing the x and y coordinates that are reported above the canvas in the CanvasTest2 app.

The last argument of Canvas1.DrawCircle (annotated r) is the radius of the circle. Although RadiusLabel is technically a text string, App Inventor automatically treats any string of digits as a number.

The three occurrences of each of x and y have different meanings:

- The annotation x to the left of the socket is a hint for the kind of entity that will fill the socket.
- name x is a **formal parameter** that names the x-coordinate where the canvas was touched. It is created automatically along with the event handler block.
- value x is a **variable reference expression** that stands for the value of the x-coordinate. It is created from the My Definitions drawer in My Blocks.

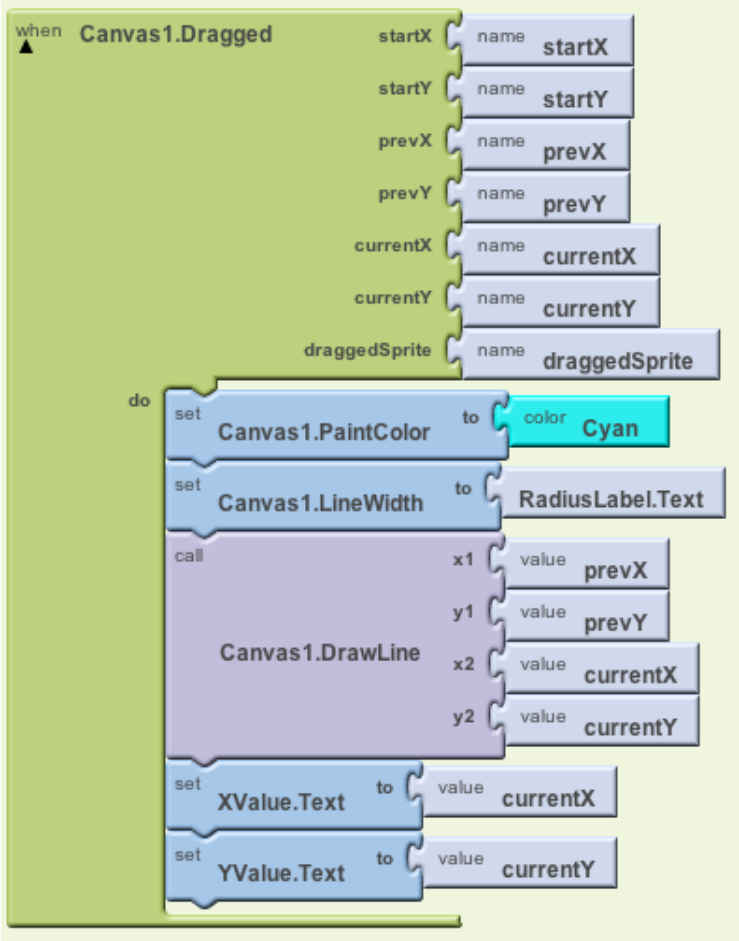
App Inventor makes this much more confusing than it has to be. Here's what you have to know:

- formal parameters like name x can only appear in annotated slots at the top of event handlers and procedures. They cannot be used as general expression blocks! In fact, if you try to use one as an expression, App Inventor will complain with a pop-up error window (try it and see).
- variable reference expressions like value x can be used as general expression blocks, and can

be used whereever a value is expected inside the body of the event handler or procedure in which it was defined. However, any attempt to use this value block in a different event handler will result in marking the block with a yellow exclamation point, indicating an error (try it and see).

- Renaming a name block automatically renames all associated value blocks, demonstrating the connection between them.

The second handler is the when Dragged handler:



Notes:

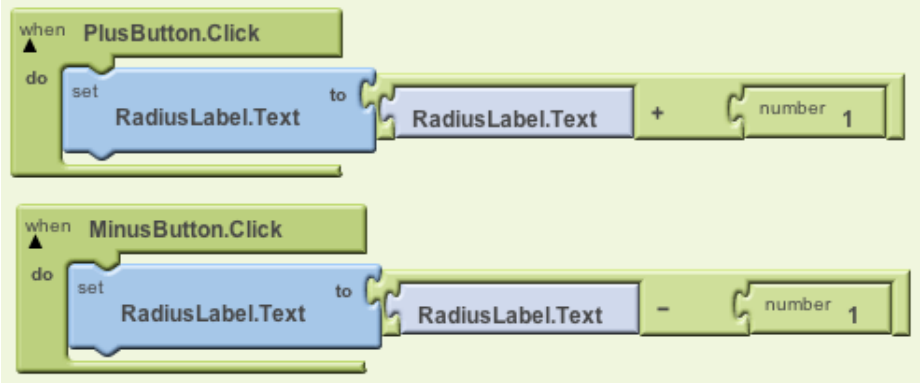
The LineWidth property controls the width of the drawn line. The PaintPot program in the book has a bug (program error): it fails to set LineWidth when drawing the line.

Each segment of a line is drawn as a rectangle whose thickness is LineWidth. For thick lines, there can be unaesthetic gaps between adjacent line segments.

To understand the distinction between prevX/prevY and startX/startY, replace the former by the latter in Canvas1.DrawLine.

In CanvasTest2, the circle radius and line width are specified by the contents of RadiusLabel. This

value can be changed by the **minus** and **plus** buttons:



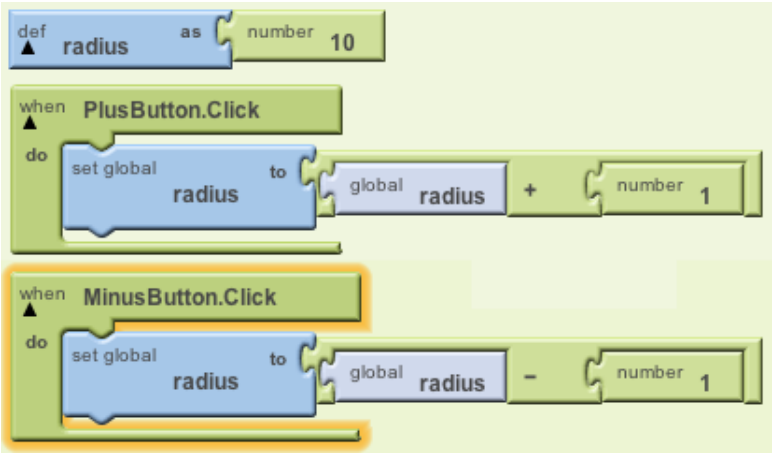
Global Variables

In many programs, there is no app component property that remembers a piece of state information that we need in the program.

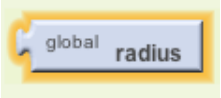
For example, suppose we remove the RadiusLabel component from CanvasTest. Where can we store the current line width?

Well, in this case, we could store it in the LineWidth property of the canvas. But let's disallow that for the time being. Another option is to store it in a **global variable**. This is just a named slot in the program whose value can change over time.

Here we use a global variable named radius to remember the current value of the radius:



Of course, we also need to change every RadiusLabel.Text expression in the Canvas1 event handlers to instead refer to the current value in the global variable:



The state of global variables is normally "hidden" within an application. But you can show the

current value in a global variable within the BlocksEditor window by right-clicking (Control-clicking on a Mac) on a global variable block and selecting **Watch**.

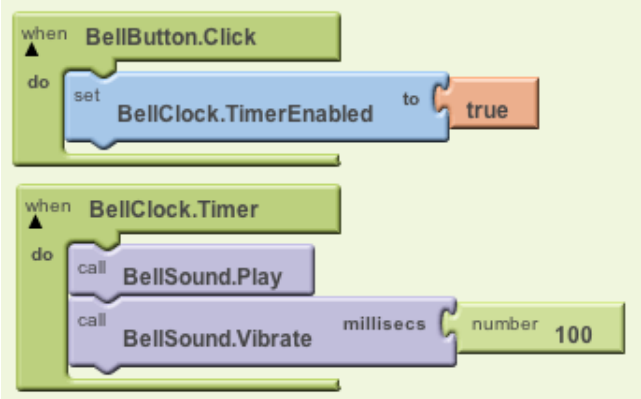
Rather than using RadiusLabel or the global variable radius, a third possibility is to use the LineWidth property of Canvas1. How would we change the CanvasTest2 program to do this?

Clocks and Timer Events

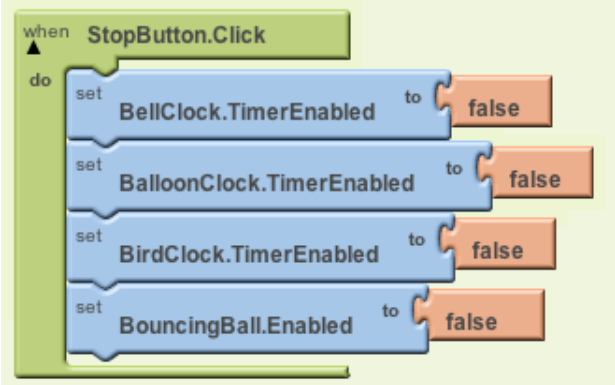
Begin by uploading the TimerTest app attached at the bottom of these notes.

The Clock component has a timer feature that can perform actions at a specified rate. When a Clock's TimerEnabled property is set to true, its when ... Timer event will fire every TimerInterval milliseconds.

For example, in TimerTest, clicking the BellButton enables the BellClock's timer, and that timer plays a sound and vibrates the phone. The TimerInterval in this case is controlled by the value (1000 milliseconds = 1 second) specified in the Properties pane fo the BellClock.



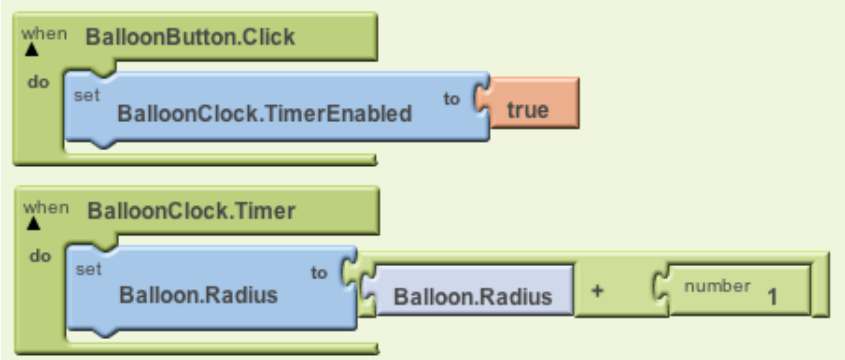
The timer can be disabled by setting the TimerEnabled property to false. In TimerTest, the StopButton disables all timers:



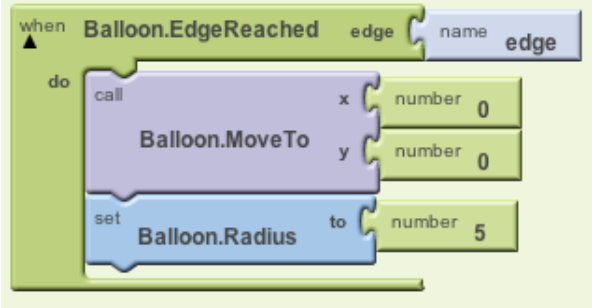
Balls, Sprites, and Simple Animations

A sprite is an creature that can move around the canvas in which it lives. In App Inventor, a ball is a simple sprite that appears as a circle.

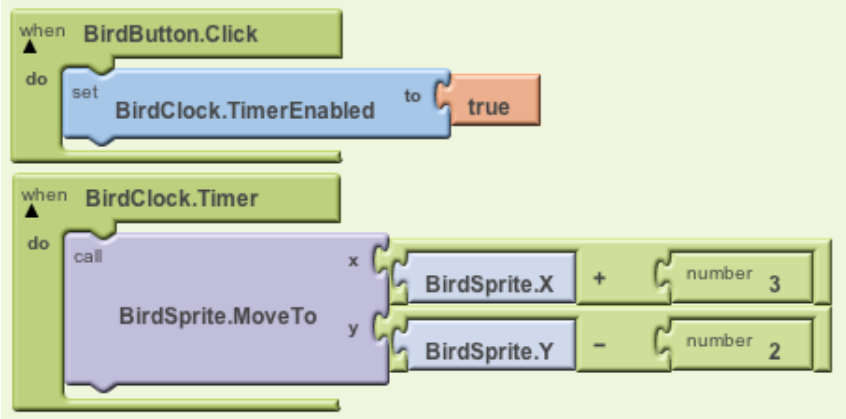
We can get the effect of an inflating red balloon by having a red ball whose radius grows every time an associated timer fires:



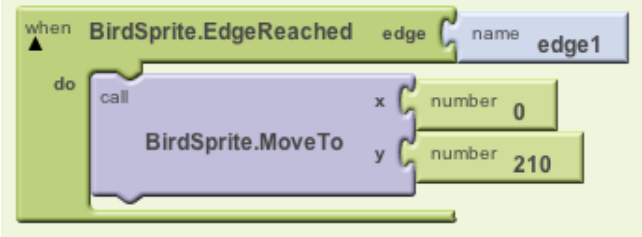
We can specify that the balloon "pops" when it hits the edge of the canvas by resetting it to its initial radius.



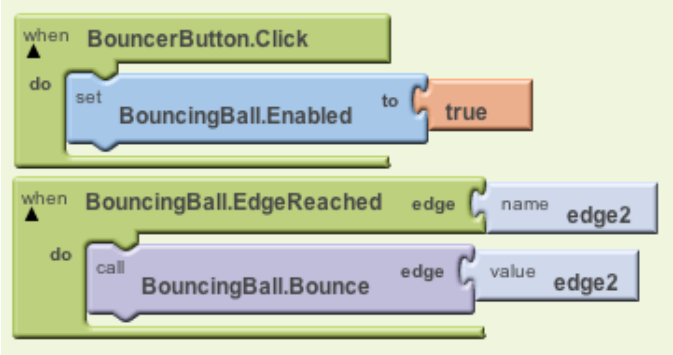
We can add a bird sprite that "flies" from the lower left to upper right corner of the canvas. Here are the blocks that specify the flight:



When the bird hits an edge, we reset it to its initial position:



Every sprite (including balls) come equipped with their own internal timer and properties that control their speed and heading. Once we've specified the speed and heading of the yellow BouncingBall in TimerTest, here are the only blocks we need to turn it on and make it bounce:



DRAFT 7/29/2012
Many thanks to Franklyn Turbak at Wellesley University for providing the [basis for this material](https://sites.google.com/site/wellesleycs117fall11/lecture-notes/lecture-04-animation-components).
https://sites.google.com/site/wellesleycs117fall11/lecture-notes/lecture-04-animation-components