CHAPTER 2

PaintPot

This tutorial introduces the Canvas *component for creating simple, two-dimensional (2D) graphics. You'll build PaintPot, an app that lets the user draw on the screen in different colors, and then update it to allow him to take his own picture and draw on that instead. On a historical note, PaintPot was one of the first programs developed to demonstrate the potential of personal computers, as far back as the 1970s. Back then, making something like this simple drawing app was a very complex undertaking, and the results were pretty unpolished. But now with App Inventor, anyone can quickly put together a fairly cool drawing app, which is a great starting point for building 2D games.*

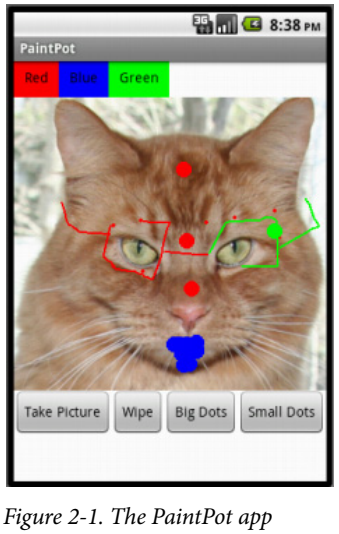

Figure 2-1. The PaintPot app

With the PaintPot app shown in Figure 2-1, you can:

- Dip your finger into a virtual paint pot to draw in that color.
- Drag your finger along the screen to draw a line.
- Poke the screen to make dots.
- Use the button at the bottom to wipe the screen clean.
- Change the dot size to large or small with the buttons at the bottom.
- Take a picture with the camera and then draw on that picture.

What You'll Learn

This tutorial introduces the following concepts:

- Using the `Canvas` component for drawing.
- Handling touch and drag events on the phone's surface.
- Controlling screen layout with arrangement components.
- Using event handlers that take arguments.
- Defining variables to remember things like the dot size the user has chosen for drawing.

Getting Started

Make sure your computer and your phone are set up to use App Inventor, and browse to the App Inventor website at *http://appinventor.googlelabs.com*. Start a new project in the Component Designer window and name it "PaintPot". Open the Blocks Editor, click "Connect to Device," and make sure the phone has started the App Inventor app.

To get started, go to the Properties panel on the right of the Designer and change the screen title to "PaintPot" (no more `Screen1` here!). You should see this change on the phone, with the new title displayed in the title bar of your app.

If you're concerned about confusing your project name and the screen name, don't worry! There are three key names in App Inventor:

- The name you choose for your project as you work on it. This will also be the name of the application when you package it for the phone. Note that you can click Save As in the Component Designer to start a new version or rename a project.
- The component name `Screen1`, which you'll see in the panel that lists the application's components. You can't change this name in the current version of App Inventor.
- The title of the screen, which is what you'll see in the phone's title bar. This starts out being Screen1, which is what you used in HelloPurr. But you can change it, as we just did for PaintPot.

Designing the Components

You'll use these components to make the app:

- Three `Button` components for selecting red, blue, or green paint, and a `HorizontalArrangement` component for organizing them.

- One `Button` component for wiping the drawing clean, and two for changing the size of the dots that are drawn.
- A `Canvas` component, which is the drawing surface. `Canvas` has a `BackgroundImage` property, which we'll set to the *kitty.png* file from the HelloPurr tutorial in Chapter 1. Later in this chapter, you'll modify the app so the background can be set to a picture the user takes.

Creating the Color Buttons

First, create the three color buttons using the following instructions:

1. Drag a `Button` component onto the viewer and change its `Text` attribute to "Red" and make its `BackgroundColor` red.
2. Click `Button1` in the components list in the Viewer to highlight it (it might already be highlighted) and click Rename to change its name from `Button1` to `RedButton`. Note that spaces aren't allowed in component names, so it's common to capitalize the first letter of each word in the name.
3. Similarly, make two more buttons for blue and green, named `BlueButton` and `GreenButton`, placing them under the red button vertically. Check your work up to this point against Figure 2-2.

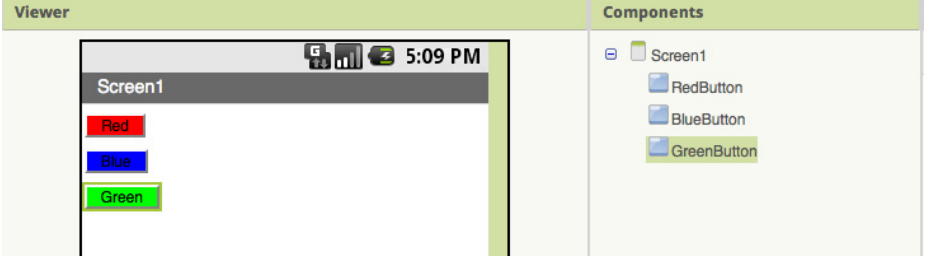

Figure 2-2. The Viewer showing the three buttons created

Note that in this project, you're changing the names of the components rather than leaving them as the default names as you did with HelloPurr. Using more meaningful names makes your projects more readable, and it will really help when you move to the Blocks Editor and must refer to the components by name. In this book, we'll use the convention of having the component name end with its type (for example, `RedButton`).

Test your app. *If you haven't clicked "Connect to Device," do so now and check how your app looks on either your phone (if it's plugged in) or in the emulator.*

Using Arrangements for Better Layouts

You should now have three buttons stacked on top of one another. But for this app, you want them all lined up next to one another at the top of the screen, as shown in Figure 2-3. You do this using a `HorizontalArrangement` component:

1. From the Palette's Screen Arrangement category, drag out a `HorizontalArrangement` component and place it under the buttons.
2. In the Properties panel, change the `Width` of the `HorizontalArrangement` to "Fill parent" so that it fills the entire width of the screen.
3. Move the three buttons one by one into the `HorizontalArrangement` component. *Hint*: You'll see a blue vertical line that shows where the piece you're dragging will go.

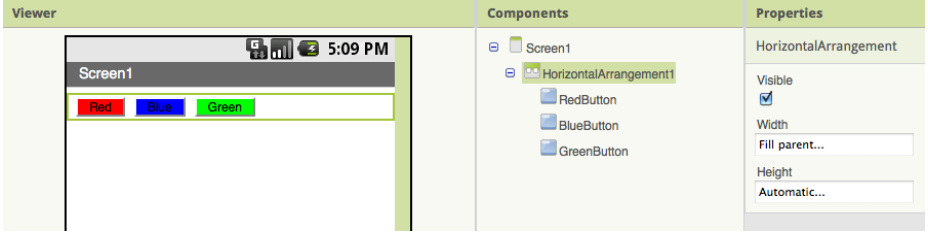

Figure 2-3. The three buttons within a horizontal arrangement

If you look in the list of project components, you'll see the three buttons indented under the `HorizontalArrangement` component to show that they are now its subcomponents. Notice that all the components are indented under `Screen1`.

Test your app. *You should also see your three buttons lined up in a row on the phone screen, although things might not look exactly as they do on the Designer. For example, the outline around* `HorizontalArrangement` *appears in the Viewer but not on the phone.*

In general, you use screen arrangements to create simple vertical, horizontal, or tabular layouts. You can also create more complex layouts by inserting (or *nesting*) screen arrangement components within each other.

Adding the Canvas

The canvas is where the user will draw circles and lines. Add it, and add the *kitty.png* file from HelloPurr as the `BackgroundImage`:

1. From the Palette's Basic category, drag a `Canvas` component onto the Viewer. Change its name to `DrawingCanvas`. Set its `Width` to "Fill parent." Set its `Height` to 300 pixels.
2. If you've completed the HelloPurr tutorial (Chapter 1), you have already downloaded the *kitty.png* file. If you haven't, you can download it from *http://examples.oreilly.com/0636920016632/*.
3. Set the `BackgroundImage` of the `Canvas` to the *kitty.png* file. In the Property editor, the `BackgroundImage` will be set to None. Click the field and choose Add to upload the *kitty.png* file.
4. Set the `PaintColor` of the `Canvas` to red so that when the user starts the app but hasn't clicked on a button yet, his drawings will be red. Check to see that what you've built looks like Figure 2-4.

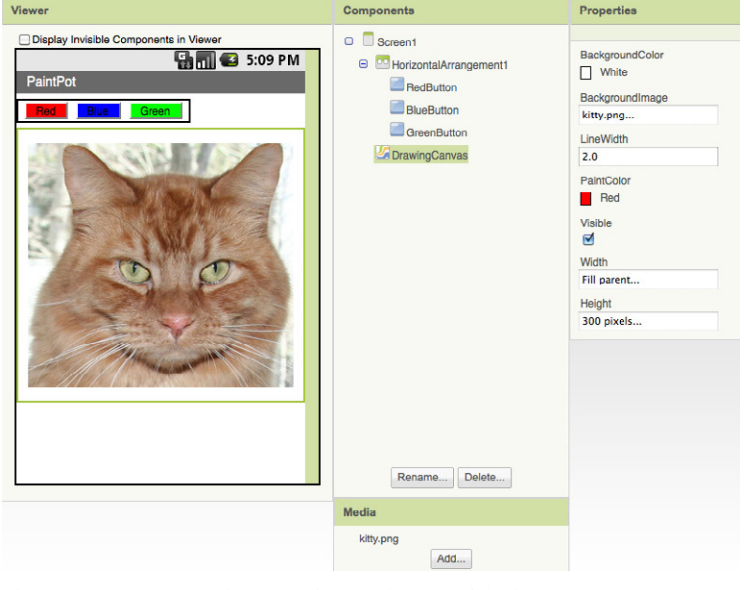

Figure 2-4. The Canvas component has a BackgroundImage of the kitty picture

Arranging the Bottom Buttons and the Camera Component

1. From the Palette, drag out a second `HorizontalArrangement` and place it under the canvas. Then drag two more `Button` components onto the screen and place them in this bottom `HorizontalArrangement`. Change the name of the first button to `TakePictureButton` and its `Text` property to "Take Picture". Change the name of the second button to `WipeButton` and its `Text` property to "Wipe".
2. Drag two more `Button` components from the Palette into the `HorizontalArrangement`, placing them next to `WipeButton`.

3. Name the buttons `BigButton` and `SmallButton`, and set their `Text` to "Big Dots" and "Small Dots", respectively.
4. From the Media Palette, drag a `Camera` component into the Viewer. It will appear in the non-visible component area.

You've now completed the steps to set the appearance of your app as shown in Figure 2-5.

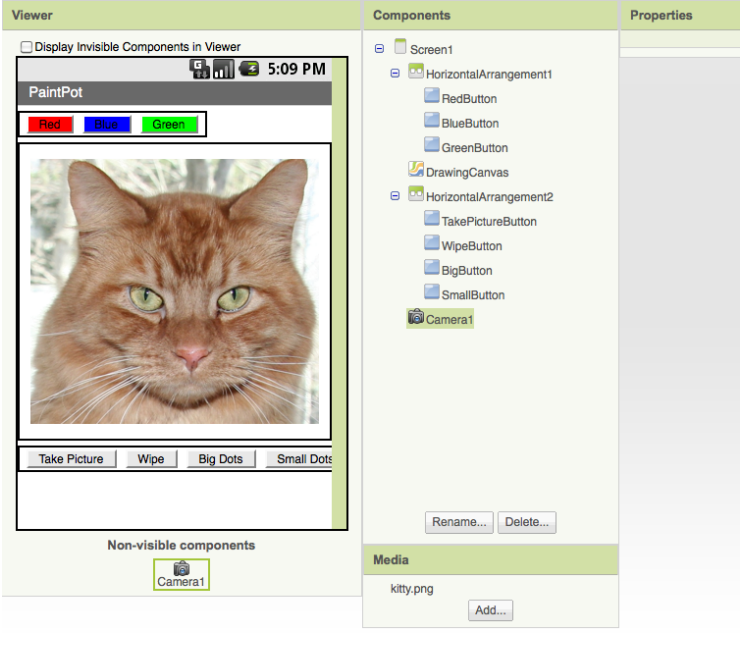

Figure 2-5. The complete user interface for PaintPot

Test your app. *Check the app on the phone. Does the kitty picture now appear under the top row of buttons? Does the bottom row of buttons appear?*

Adding Behaviors to the Components

The next step is to define how the components behave. Creating a painting program might seem overwhelming, but rest assured that App Inventor has done a lot of the heavy lifting for you: there are easy-to-use blocks for handling the user's touches and drags, and for drawing and taking pictures.

In the Designer, you added a Canvas component named DrawingCanvas. Like all canvas components, DrawingCanvas has a Touched event and a Dragged event. You'll program the **DrawingCanvas.Touched** event so that it calls **DrawingCanvas.DrawCircle**. You'll program the **DrawingCanvas.Dragged** event to call **DrawingCanvas.DrawLine**. You'll then program the buttons to set the DrawingCanvas.PaintColor property, clear the DrawingCanvas, and change the BackgroundImage to a picture taken with the camera.

Adding the Touch Event to Draw a Dot

First, you'll arrange things so that when you touch the DrawingCanvas, you draw a dot at the spot you touch:

1. In the Blocks Editor, click My Blocks, select the drawer for the DrawingCanvas, and drag the **DrawingCanvas.Touched** block to the workspace. As soon as you drag the block out, the three plugs on the right automatically fill in with name blocks for x, y, and touchedSprite, as shown in Figure 2-6.

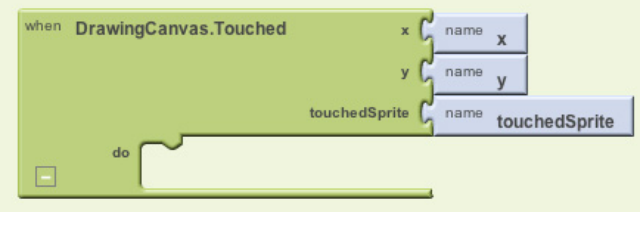

Figure 2-6. The event comes with information about where the screen is touched

Note. *If you've completed the HelloPurr app in Chapter 1, you're familiar with* ***Button.Click*** *events, but not with* Canvas *events.* ***Button.Click*** *events are fairly simple because there's nothing to know about the event other than that it happened. Some event handlers, however, come with information about the event called arguments. The* ***DrawingCanvas.Touched*** *event tells you the x and y coordinates of the touch within the canvas. It also tells you if an object within the* DrawingCanvas *(in App Inventor, this is called a* sprite*) was touched, but we won't need that until Chapter 3. The x and y coordinates are the arguments we'll use to note where the user touched the screen, so we can then draw the dot at that position.*

2. Drag out a **DrawingCanvas.DrawCircle** command from the DrawingCanvas drawer and place it within the **DrawingCanvas.Touched** event handler, as shown in Figure 2-7.

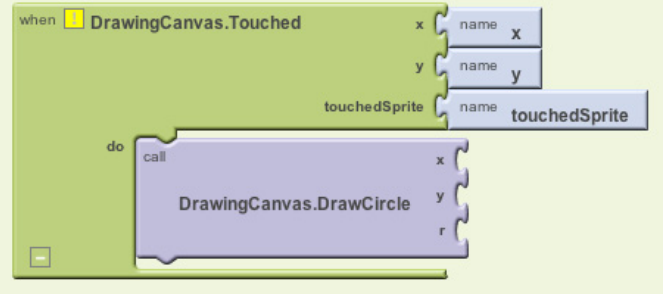

Figure 2-7. When the user touches the canvas, the app draws a circle

On the right side of the **DrawingCanvas.DrawCircle** block, you'll see three slots for the arguments we need to fill in: x, y, and `r`. The x and y arguments specify the location where the circle should be drawn, and `r` determines the radius (or size) of the circle. The yellow warning box with the exclamation point at the top of the **DrawingCanvas.Touched** event handler denotes that these slots haven't yet been filled. We'll build the blocks to do that next.

This event handler can be a bit confusing because the **DrawingCanvas.Touched** event also has x and y slots; just keep in mind that the x and y for the **DrawingCanvas.Touched** event tell you where the user touched, while the x and y for the **DrawingCanvas.DrawCircle** event are open slots for you to specify where the circle should be drawn.

Because you want to draw the circle where the user touched, plug in the x and y values from **DrawingCanvas.Touched** as the values of the x and y parameters in **DrawingCanvas.DrawCircle**.

Note. *Do not grab the arguments of the* Touched *event directly, even though this might seem logical! The fact that the arguments can even be grabbed is an unfortunate design aspect of App Inventor. Instead, you want to grab these values from the My Definitions drawer, as shown in Figure 2-8.*

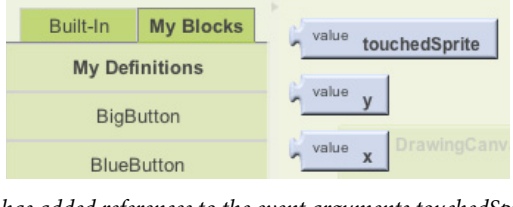

Figure 2-8. The system has added references to the event arguments touchedSprite, y, and x

3. Open the My Definitions drawer within My Blocks and find the blocks for **value x** and **value y**.

   The blocks were automatically created for you by App Inventor when you dragged out the **DrawingCanvas.Touched** event handler block: they are *references* to the x and y arguments (or names) of that event. Drag out the **value x** and **value y** blocks and plug them into the corresponding sockets in the **DrawingCanvas .DrawCircle** block so they resemble what is shown in Figure 2-9.

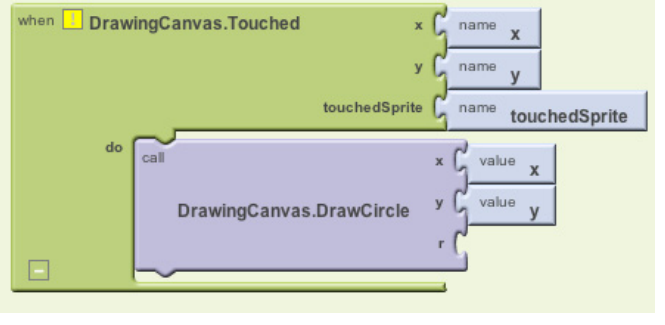

Figure 2-9. The app knows where to draw (x,y), but we still need to specify how big the circle should be

4. You'll also need to specify the radius, `r`, of the circle to draw. The radius is measured in pixels, which is the tiniest dot that can be drawn on the screen. For now, set it to 5: click in a blank area of the screen to bring up the shortcut menu, and then select the Math folder. Select 123 from the drop-down list to create a number block. Change the 123 to 5 and plug that in for the `r` slot. When you do, the yellow box in the top-left corner will disappear as all the slots are filled. Figure 2-10 illustrates how the final **DrawingCanvas.Touched** event handler should look.

Note. *Note that you could have created the **number 5** block by simply typing a 5 in the Blocks Editor, followed by Return. This is an example of typeblocking: if you start typing, the Blocks Editor shows a list of blocks whose names match what you are typing; if you type a number, it creates a number block.*

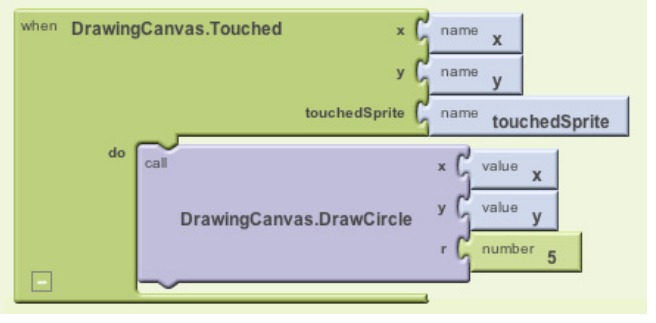

Figure 2-10. When the user touches the canvas, a circle of radius 5 will be drawn at (x,y)

Test your app. *Try out what you have so far on the phone. Touch the canvas—your finger should leave a dot at each place you touch. The dots will be red if you set the* `Canvas.PaintColor` *property to red in the Component Designer (otherwise, it's black, as that's the default).*

Adding the Drag Event That Draws a Line

Next, you'll add the drag event handler. Here's the difference between a touch and a drag:

- A *touch* is when you place your finger on the canvas and lift it without moving it.
- A *drag* is when you place your finger on the canvas and move it while keeping it in contact with the screen.

In a paint program, dragging your finger across the screen appears to draw a giant, curved line along your finger's path. What you're actually doing is drawing hundreds of tiny, straight lines; each time you move your finger, even a little bit, you extend the line from your finger's last position to its new position.

1. From the DrawingCanvas drawer, drag the **DrawingCanvas.Dragged** block to the workspace. You should see the event handler as it is shown in Figure 2-11.

   The **DrawingCanvas.Dragged** event comes with seven arguments:

   `startx`, `starty`
   : The position of your finger back where the drag started.

   `currentx`, `currenty`
   : The current position of your finger.

   `prevx`, `prevy`
   : The immediately previous position of your finger.

`draggedSprite`
: The argument that will be true if the user drags directly on an image sprite. We won't use this argument in this tutorial.

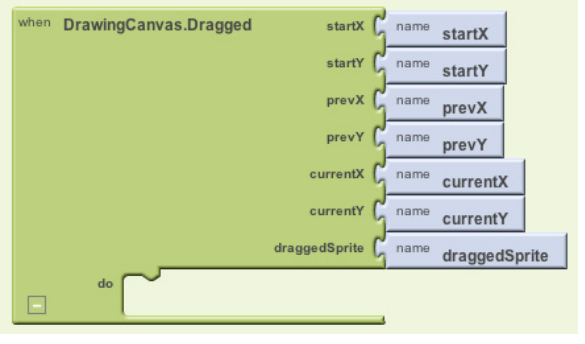

Figure 2-11. A Dragged event has even more arguments than Touched

2. From the DrawingCanvas drawer, drag the **DrawingCanvas.DrawLine** block into the **DrawingCanvas.Dragged** block, as shown in Figure 2-12.

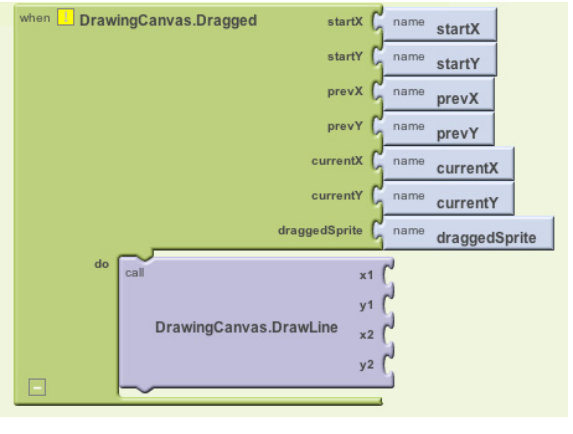

Figure 2-12. Adding the capability to draw lines

The **DrawingCanvas.DrawLine** block has four arguments, two for each point that determines the line: (`x1`,`y1`) is one point, while (`x2`,`y2`) is the other. Can you figure out what values need to be plugged into each argument? Remember, the **Dragged** event will be called many times as you drag your finger across the canvas: the app draws a tiny line each time your finger moves, from (`prevx`,`prevy`) to (`currentX`,`currentY`). Let's add those to our **DrawingCanvas.DrawLine** block:

3. Click the My Definitions drawer. You should see the blocks for the arguments you need. Drag the corresponding value blocks to the appropriate slots in `DrawingCanvas.Dragged`. **value prevX** and **value prevY** should be plugged into the `x1` and `y1` slots. **value currentX** and **value currentY** should be plugged into the `x2` and `y2` slots, as shown in Figure 2-13.

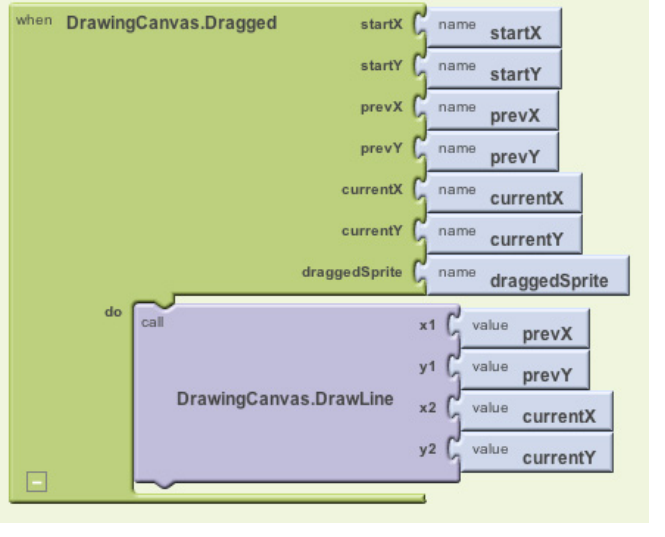

Figure 2-13. As the user drags, the app will draw a line from the previous spot to the current one

Test your app. *Try this behavior on the phone: drag your finger around on the screen to draw lines and curves. Touch the screen to make spots.*

Adding Button Event Handlers

The app you've built lets the user draw, but it always draws in red. Next, add event handlers for the color buttons so users can change the paint color, and another for `WipeButton` so they can clear the screen and start over.

In the Blocks Editor:

1. Switch to the My Blocks column.
2. Open the drawer for `RedButton` and drag out the **RedButton.Click** block.
3. Open the DrawingCanvas drawer. Drag out the **set DrawingCanvas.PaintColor to** block (you may have to scroll through the list of blocks in the drawer to find it) and place it in the "do" section of **RedButton.Click**.

4. Switch to the Built-In column. Open the Colors drawer and drag out the block for the color red and plug it into the **set DrawingCanvas.PaintColor to** block.
5. Repeat steps 2–4 for the blue and green buttons.
6. The final button to set up is `WipeButton`. Switch back to the My Blocks column and drag out a **WipeButton.Click** from the ButtonWipe drawer. From the DrawingCanvas drawer, drag out **DrawingCanvas.Clear** and place it in the **WipeButton.Click** block. Confirm that your blocks show up as they do in Figure 2-14.

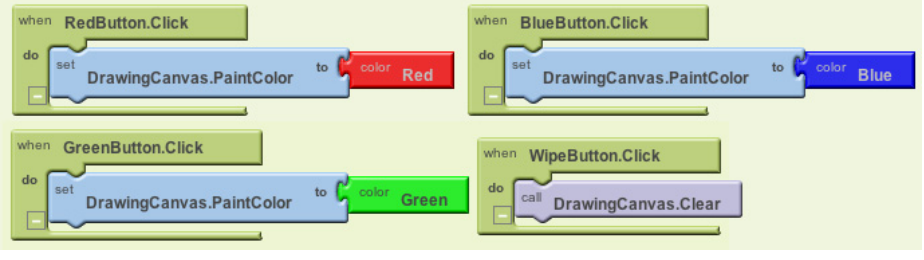

Figure 2-14. Clicking the color buttons changes the canvas's PaintColor; clicking Wipe clears the screen

Letting the User Take a Picture

App Inventor apps can interact with the powerful features of an Android device, including the camera. To spice up the app, we'll let the user set the background of the drawing to a picture she takes with the camera.

1. The `Camera` component has two key blocks. The **Camera.TakePicture** block launches the camera application on the device. The event **Camera.AfterPicture** is triggered when the user has finished taking the picture. You'll add blocks in the **Camera.AfterPicture** event handler to set the `DrawingCanvas.BackgroundImage` to the just-taken Switch to the My Blocks column and open the TakePictureButton drawer. Drag the **TakePictureButton.Click** event handler into the workspace.
2. From `Camera1`, drag out **Camera1.TakePicture** and place it in the **TakePictureButton.click** event handler.
3. From `Camera1`, drag the **Camera1.AfterPicture** event handler into the workspace.
4. From `DrawingCanvas`, drag the **set DrawingCanvas.BackgroundImage to** block and place it in the **Camera1.AfterPicture** event handler.
5. **Camera1.AfterPicture** has an argument named `image`, which is the picture just taken. You can get a reference to it, **value image**, in the My Definitions palette; drag it out and plug it into **DrawingCanvas.BackgroundImage**.

The blocks should look like Figure 2-15.

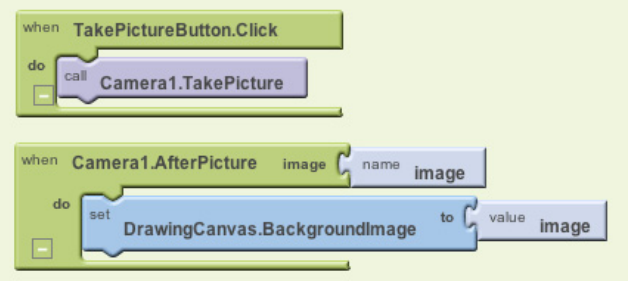

Figure 2-15. When the picture is taken, it's set as the canvas's background image

Test your app. *Try out this behavior by clicking Take Picture on your phone and taking a picture. The cat should change to the picture you take, and then you can draw on that picture. (Drawing on Professor Wolber is a favorite pastime of his students, as exemplified in Figure 2-16.)*

Changing the Dot Size

The size of the dots drawn on the canvas is determined in the call to **DrawingCanvas.DrawCircle** when the radius argument `r` is set to 5. To change the thickness, you can put in a different value for `r`. To test this, try changing the 5 to a 10 and testing it out on the phone to see how it looks.

The catch here is that whatever size you set in the radius argument is the only size the user can use. What if he wants to change the size of the dots? Let's modify the program so that the user, not just the programmer, can change the dot size. We'll change it so that when the user clicks a button labeled "Big Dots," the dot size is 8, and when he clicks a button labeled "Small Dots," it is 2.

To use different values for the radius argument, the app needs to know which one we want to apply. We have to tell it to use a specific value, and it has to store (or remember) that value somehow so it can keep using it. When your app needs to remember

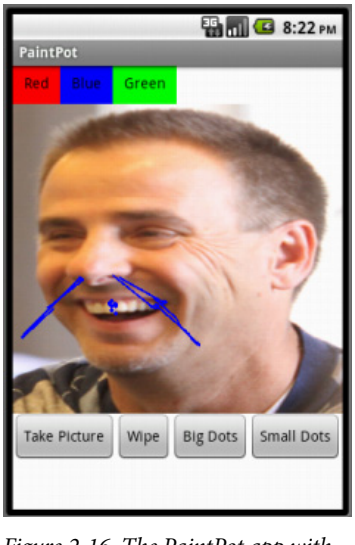

Figure 2-16. The PaintPot app with an "annotated" picture of Professor Wolber

something that's not a property, you can define a *variable*. A variable is a *memory cell*; you can think of it like a bucket in which you can store data that can vary, such as the current dot size (for more information about variables, see Chapter 15).

Let's start by defining a variable `dotSize`:

1. In the Blocks Editor, open the Definitions drawer in the Built-In column. Drag out a **def variable** block. Change the text "variable" to "dotSize".
2. Notice that the **def dotSize** block has an open slot. This is where you can specify the initial value for the variable, or the value that it defaults to when the app begins. (This is often referred to as "initializing a variable" in programming terms.) For this app, initialize the `dotSize` to 2 by creating a **number 2** block (by either starting to type the number 2 or dragging a **number 123** block out of the Math drawer) and plugging it into **def dotSize**, as shown in Figure 2-17.

Figure 2-17. Initializing the variable dotSize with a value of 2

Using variables

Next, we want to change the argument of **DrawingCanvas.DrawCircle** in the **DrawingCanvas.Touched** event handler so that it uses the value of **dotSize** rather than always using a fixed number. (It may seem like we've "fixed" `dotSize` to the value 2 because we initialized it that way, but you'll see in a minute how we can change the value of `dotSize` and therefore change the size of the dot that gets drawn.)

1. In the Blocks Editor, switch to the My Blocks column and open the My Definitions drawer. You should see two new blocks: (1) a **global dotSize** block that provides the value of the variable, and (2) a **set global dotSize to** block that sets the variable to a new value. These blocks were automatically generated for you when you created the `dotSize` variable, in the same way that value blocks for the arguments `x` and `y` were created when you added the **DrawingCanvas.Touched** event handler earlier.
2. Go to the **DrawingCanvas.Touched** event handler and drag the **number 5** block out of the `r` slot and place it into the trash. Then replace it with the **global dotSize** block from the My Definitions drawer (see Figure 2-18). When the user touches the canvas, the app will now determine the radius from the variable `dotSize`.

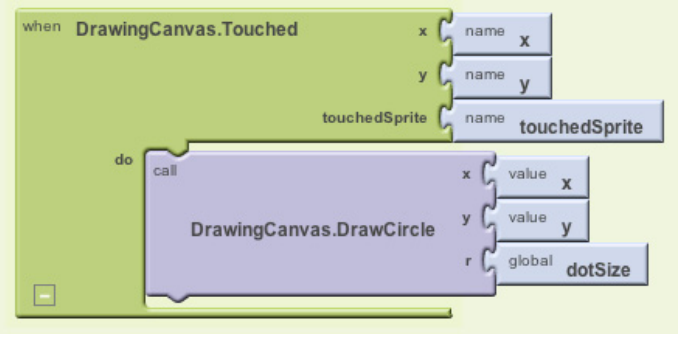

Figure 2-18. Now the size of each circle is dependent on what is stored in the variable dotSize

Changing the values of variables

Here's where the magic of variables really comes into play—the variable `dotSize` allows the user to choose the size of the circle, and your event handler will draw the circle accordingly. We'll implement this behavior by programming the **SmallButton.Click** and **BigButton.Click** event handlers:

1. Drag out a **SmallButton.Click** event handler from the SmallButton drawer of My Blocks. Then drag out a **set global dotSize to** block from My Definitions and plug it into **SmallButton.Click**. Finally, create a **number 2** block and plug it into the **set global dotSize to** block.
2. Make a similar event handler for **BigButton.Click**, but set `dotSize` to 8. Both event handlers should now show up in the Blocks Editor, as shown in Figure 2-19.

Note. *The "global" in the **set global dotSize to** refers to the fact that the variable can be used in all the event handlers of the program (globally). Some programming languages allow you to define variables that are "local" to a particular part of the program; App Inventor currently does not.*

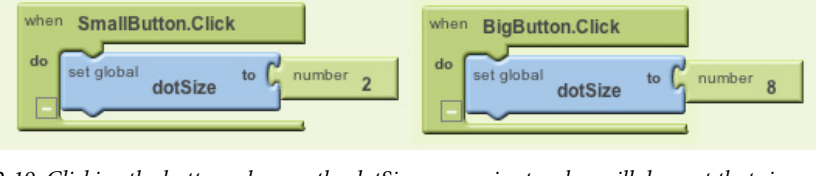

Figure 2-19. Clicking the buttons changes the dotSize; successive touches will draw at that size

Test your app. *Try clicking the size buttons and then touching the canvas. Are the circles drawn with different sizes? Are the lines? The line size shouldn't change because you programmed* `dotSize` *to only be used in the* ***DrawingCanvas.DrawCircle*** *block. Based on that, can you think of how you'd change your blocks so users could change the line size as well? (Note that Canvas has a property named LineWidth.)*

The Complete App: PaintPot

Figure 2-20 illustrates our completed PaintPot app.

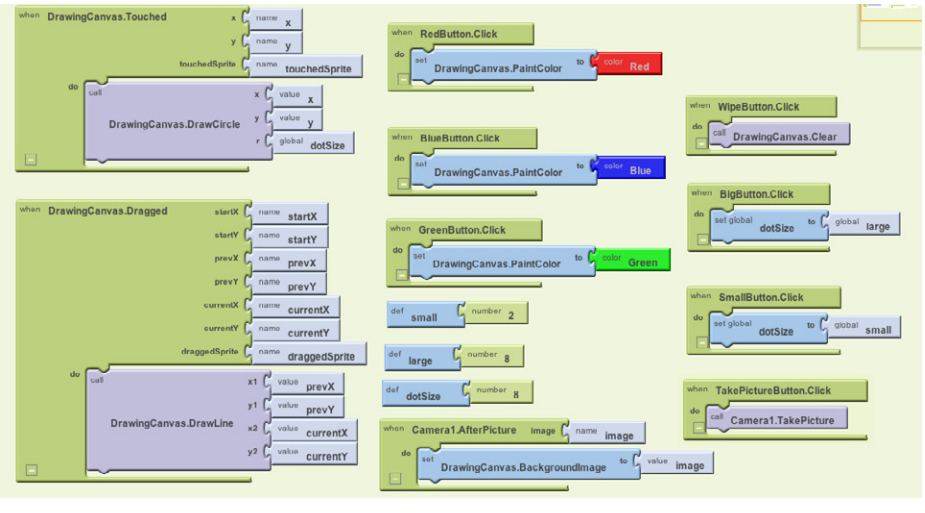

Figure 2-20. The final set of blocks for PaintPot

Variations

Here are some variations you can explore:

- The app's user interface doesn't provide much information about the current settings (for example, the only way to know the current dot size or color is to draw something). Modify the app so that these settings are displayed to the user.
- Let the user enter the dot size within a `TextBox` component. This way, she can change it to other values besides 2 and 8. For more information on input forms and the `TextBox` component, see Chapter 4.

Summary

Here are some of the ideas we've covered in this chapter:

- The Canvas component lets you draw on it. It can also sense touches and drags, and you can map these events to drawing functions.
- You can use screen arrangement components to organize the layout of components instead of just placing them one under the other.
- Some event handlers come with information about the event, such as the coordinates of where the screen was touched. This information is represented by arguments. When you drag out an event handler that has arguments, App Inventor creates value blocks for them and places them in the My Definitions drawer.
- You create variables by using **def variable** blocks from the Definitions drawer. Variables let the app remember information, like dot size, that isn't stored in a component property.
- For each variable you define, App Inventor automatically supplies a **global value** block that gives the value of the variable, and a **set global variable to** block for changing the value of the variable. These blocks can be found in the My Definitions drawer. To learn more about variables, see Chapter 16.

This chapter showed how the Canvas component can be used for a painting program. You can also use it to program animations such as those you'd find in 2D games. To learn more, check out the Ladybug Chase game in Chapter 5 and the discussion of animation in Chapter 17.