|  |  |
| --- | --- |
| Paint Pot is a basic “finger painting” app. It simulates the process of dipping your finger in a pot of a paint and then drawing on a canvas. The app uses buttons to simulate dipping your finger in the paint and uses App Inventor’s touch event handlers to draw circles and lines on the canvas.   Objectives: In this lesson you will learn to:  * follow an instructor-led walkthrough to create the *PaintPot* app on a mobile device. * continue navigating the App Inventor programming platform. * develop your understanding of what an App Inventor program is. * deepen your understanding of event-driven programming. | ***[Click to watch Preview Video](http://www.youtube.com/watch?v=7aGbsg1D3no)*** |

## Getting Ready

Open [App Inventor with the Paint Pot Template](http://ai2.appinventor.mit.edu/?repo=templates.appinventor.mit.edu/trincoll/csp/unit3/templates/PaintPotMediaOnly/PaintPotMediaOnly.asc) in a separate tab and follow along with the following tutorial. Once the project opens use *Save As* to rename your project ***PaintPot****.*

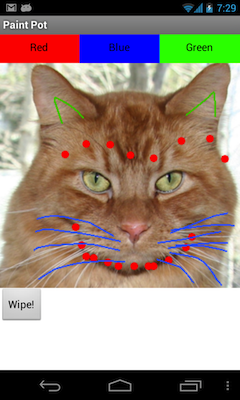
# 

# 

# Paint Pot Tutorial

([Video Tutorial](http://www.youtube.com/watch?v=yvI5Za4DvqA))

## The Paint Pot UI

The UI for our PaintPot app will consist of two types of *Components,* 

**Basic components** -- four *Buttons and a drawing Canvas -- and*

**Screen Arrangement component** --a *HorizontalArrangement*.

As you can see, the Red, Blue, and Green buttons are arranged horizontally along the top of the screen. These are used to set the paint color.

The cat’s image is contained on the canvas component.

And the “Wipe” button is below the canvas.

### 

### Adding Red, Blue, and Green Buttons

By default, when you add components to the App Inventor viewer, they are laid out vertically. To arrange our color buttons horizontally, we need to put them inside a *HorizontalArrangement* component*.*

1. Drag and drop a *HorizontalArrangement* component from the Palette’s *Layout* category to the Viewer.
2. Drag three button components out of the Basic category and place them into the *HorizontalArrangement component*.
3. Select each Button in turn and change its *Text* and *BackgroundColor* properties, so that they match its function. So the red button should be red and should be labeled “Red” as shown in the screenshot.

### 

### 

### Renaming Components with Descriptive Names

So far we’ve been using App Inventor’s default names for our components. Our three buttons are named Button1, Button2, and Button3. But this will be confusing when we switch to the Blocks Editor where you can no longer see that Button1 is red. You’ll want to give your buttons’ names that are more descriptive of their function in the app.

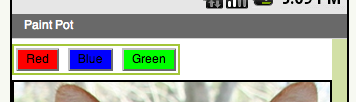
You can give them any names that are meaningful to you. But if you give them names that begin with “Button”, then they’ll all be right next to each other and easier to find in the Blocks Editor’s Toolbox, for example:

* ButtonRed
* ButtonBlue
* ButtonGreen

To rename a component, just select the component by clicking on it in the Viewer or in the Components panel. Then click the *Rename* button in the components panel and type in an appropriate name in the New name textbox.

### Fine Tuning the Horizontal Arrangement (Optional)

You probably noticed that App Inventor puts the buttons right next to each other, from left to right, in the Horizontal Arrangement, and sets their width automatically depending on the size of their labels. So they look like this:



If you prefer, you can adjust the *Width* property of the Buttons as well as the *HorizontalArrangement* so that all buttons have the same size and fill up their container:



To get this latter arrangement, set the *Width* property of all these components, including the HorizontalArrangement to *Fill parent.*

### 

### 

### Adding the Canvas

In App Inventor, all animations, drawing and painting takes place on a *Canvas* component, which is also used as a background for interactive games.

1. Drag a Canvas component out of the Palette’s Drawing and Animation drawer onto the Viewer, placing it just below the row of color buttons.
2. Set the Canvas’ BackgroundImage property to the *kitty.png* image.
3. Set the Canvas’ Width property to Fill Parent.
4. Set the Canvas’ PaintColor property to red, overriding its default color, which is black.

|  |  |
| --- | --- |
| Your Viewer should now look like this.  **Event Driven Programming.** If you recall from the I Have A Dream app, we had one button and we had one block for responding to its “click” events. How many different events will this app have to handle in the code editor view? What would you guess? | PaintPotUI.PNG |

## 

## 

## 

## Coding the App’s Behavior

If you said “six” events, that is correct. There are three buttons, each of which will have its own Button.click event. Plus, we want to be able to draw dots and lines on the canvas. This means will need to respond to a Canvas\_touch event and a Canvas\_drag event. Let’s see how this goes.

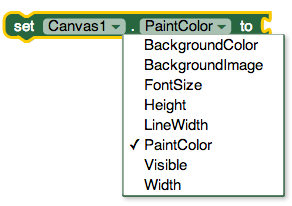
### Setting A Property’s Value

For this app we’ll be using a new kind a block, a *Setter block*, to set, or change, the value of the Canvas’s PaintColor property whenever one of the color buttons is clicked:

setter.png

Note that this block has a *to slot* that looks like a jig-saw piece. This is where you would plug in an appropriate value for the property. In this case we are plugging in the color red (which can be found in the Toolbox’s *Colors* drawer).

Setter blocks are very important and we will be using them in almost our apps. One of the features of App Inventor is that each type of component has a single setter block, which is known as a *mutator* in Computer Science, that can be used to set any of that component’s properties. This is done using the drop-down menu on the block:



In this case we are using the Setter to set the value of the PaintColor property. But this same block can be changed (mutated) to set the value of the *Width* or *Height* and so on.

## 

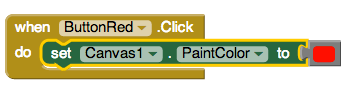
### 

### Handling the Button Clicks

We will need three *when-Button.Click* blocks, one for each button. For the color buttons, when one of them is clicked we want to set the Canvas’s paint color to its color. For example, for *ButtonRed:*

1. Click on the *ButtonRed* component in the Toolbox . Drag its *when ButtonRed.Click* block into the Workspace.
2. Click on the *Canvas1* and drag its *Set Canvas1.BackgroundColor*  block into the ButtonRed’s *do slot.* Then select the *PaintColor* property from its mutator (pull-down menu).
3. Click on the *Colors* in the Toolbox and drag out the color Red and plug it into the *to-slot* on the setter block.

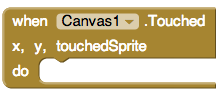
When you’re done, the finished block should look like this:



**You should repeat these same steps for each of the other button.click blocks.**  As a shortcut, you could use the Copy (Ctrl-C) and Paste (Ctrl-V) keys to copy this block and change the names of the components involved. But be careful to name things correctly.

### Responding to a Touch Event

**While the various button events are important for the behavior of the app it is the touch and drag events that give the app its ability to paint on the canvas.** The *Canvas* component has two blocks that enable the app to respond to these kinds of events. The first is the *Canvas1.Touched* block:

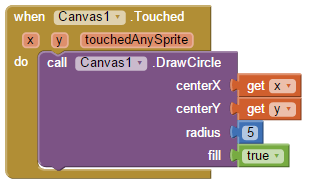


One difference between this event handler and the Button.click handler is the Canvas.Touched event handler has three **pre-set** properties*, x, y,* and, *touchedSprite.* The *x* and *y* properties represent the x- and y-coordinates of the touch event’s location -- i.e., on what pixel, given by its (x,y) coordinates, did the user touch the Canvas. (We won’t worry about the *touchedSprite* property in this app.)

These properties are given values when the event occurs -- that is, in this case, when the Canvas is touched. To access their values, we will use *Getter Blocks,* which are located in either of two places: (1) in the Toolbox’s Variables drawer or (2) by hovering your mouse pointer over the ‘x’ and ‘y’ on the event handler:



For example, here’s how we use these the touch event’s (x,y) values to draw a circle of radius 5 at that same location:



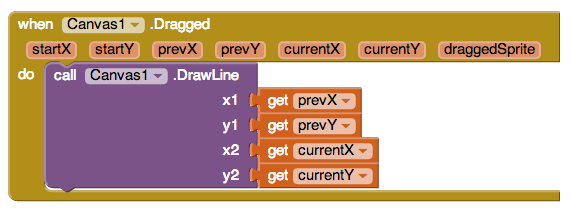
In this case we are using the Canvas’s *DrawCircle* procedure to draw the circle. Note that the *DrawCircle* procedure has three slots, labeled, *x, y,* and *r.* These represent the circle’s (x,y) location and its *radius,* respectively. When we plug the *get-x block* into the DrawCircle’s *x-slot* and the *get-y block* into the DrawCircle’s *y-slot*, we are setting the circle’s location. Similarly, by plugging the value *5* into the DrawCircle’s *r-slot* we are setting its radius.

## 

## 

### Responding to a Drag Event

**Now that you have added the Canvas1.Touched, add in a Canvas1.Dragged to make your app have more capabilities.** The *Canvas.Drag*ged event is very similar to the touch event except that it has more properties:

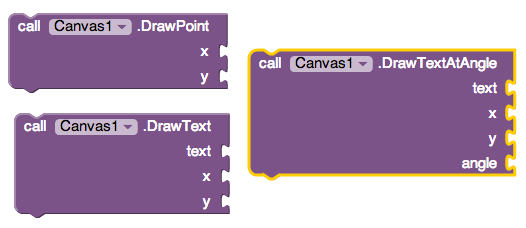


In this example we are using the *Canvas.DrawLine* procedure to draw a line from (x1,y1) to (x2,y2) and we are using the *Canvas.Dragged* block’s *prevX, prevY* and *currentX, currentY* to supply the locations of the line’s endpoints. Don’t forget that the *getters* used in this block can be found in the Toolbox Variables drawer or by hovering your mouse pointer over the property names on the event handler.

(**Just a thought?** If you think about this, when you draw an arc on the kitty’s face, this block is actually drawing many small lines to make up the larger arc. In order to do this, the event must fire many times each second. What do you suppose would happen if instead of *prevX* and *prevY* you put the getters for *startX* and *startY* in the *x1-slot* and *y1-slot*? Take a guess and then try it. You’ll see an interesting result.)

### Canvas Drawing Methods

In addition to *DrawCircle* and *DrawLine,* the *Canvas* component has several other drawing procedures that you can experiment with in your app. Here are some of the others:

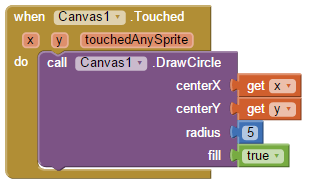


### 

### 

### Testing and Increasing the Dot Size

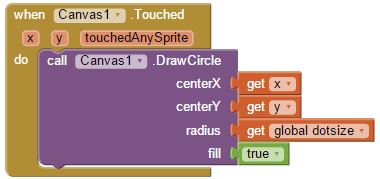
Nice work! Now try out your app and test that it works. Do you notice any limitations? To start, did you notice that all of the paint dots have the same radius -- 5 pixels. Let’s take another look at the *DrawCircle* method that we used to draw a dot whenever the canvas was touched. As you see here, the radius, *r,* is always set to 5. Every dot will have a radius set to 5. The radius will never change!



Programmers refer to the number 5 here as a *constant* or a *literal value* because its value never changes -- it is literally 5. How can we make this more flexible? How can we enable the app to draw dots of different sizes?

## Abstraction to the Rescue

Let’s think about what would happen if we replaced the number 5 in the above block with a symbol, such as *dotsize,* that can represent any value:



Now, when a dot is painted, its radius will be whatever value that *dotsize* represents. If we set dotsize’s value to 5, then it would draw a dot of radius 5. If we set it to 8, it will draw a dot of radius 8. And so on. So, rather than just be a constant, such as 5, *dotsize* is an abstract variable that can stand for any value. This is a simple example of the *abstraction principle.* We will see many other examples in this course. But, we can’t implement this just yet. First, we will need to create a global variable.

### 

### Creating and Using Global Variables

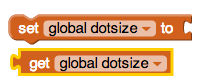
In App Inventor, *dotsize* is an example of a *global variable.* You can think of a variable as a storage container that can store any value and it can be used *globally* throughout the app. Global variables have to be created and given an initial value using a special block that is found in the *Variables* drawer of the Toolbox. (If you look in that drawer you will find that App Inventor also has *local variables,* which we will also learn how to use eventually.) When creating a global variable you should give it a unique, but valid, name. Variables in App Inventor, as in many other computer languages, must be strings of letters and digits (no quotes) and cannot start with a digit. An example of an invalid name would be *10seconds.*

Here’s how we define and initialize our global *dotsize* variable:

initglobaldotsize.png

We have given *dotsize* an initial value of 5. So if we added these two blocks to our app and did nothing else, it would behave just as before -- all dots would have a radius of 5 because that’s the current value of *dotsize.*

Of course, App Inventor also has blocks in the *Variables* drawer to *set* and *get* the value of a global variable:



For the *setter block* we can set the value of *dotsize* to whatever number we put in its *to-slot.* By using a *getter block* we can get *dotsize*’s value and plug it into any slot where it will fit -- just like putting together a jigsaw puzzle.

Now that we’ve created the global variable *dotsize*, we can replace the constant *5* with the *getter block,* for *global dotsize*, as we discussed in the previous section.

## 

## 

## 

## 

## Example: Adding 1 to a Variable

Variables that store numbers, such as *dotsize,* can be used in arithmetic.To see this let’s pull an *addition block* and a *number block* out of the *Math* drawer:



(Don’t be confused by the blue circle on this block. It is a drop-down menu that lets you change the block so you can add more than two values with one block. The white ‘+’ sign in the middle of the two open slots is the plus operator.)

Notice that the open slots the addition block has the correct shape for plugging in either a value (such as 0) or a variable (such as *dotsize*):

dotsizeplus0.png

Thus, we have created an *expression* whose value is (dotsize + 0). Since we have initialized the *dotsize* to 5, then this expression has the value 5. If we change the number block to a 1



then the resulting expression would have the value 6, (dotsize + 1) or (5 +1).

Now, this expression block can itself be plugged into any slot where a value can be plugged. For example, we can plug it into *dotsize’s* setter block:

setdotsizeplus1.png

This is an App Inventor example of an *assignment statement*. The value of dotsize is set to the result of the expression which is its current value + 1. *dotsize* will now have the value 6.

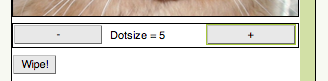
## 

## 

## Drawing Different Sized Dots

How might we use this new found ability to enable our app to draw different sized dots? One way would be to add two Buttons to the app, one labeled ‘+’, which adds 1 to dotsize whenever it is clicked, and the other labeled ‘-’, which subtracts 1 from dotsize whenever it is clicked. It might also be nice to add a *Label Component* that will display the current value of dotsize and to update its value whenever one of the buttons is clicked.

In other words, we want to change our UI so that it has the following additional components:



(NOTE: You could use Labels for this task, one for the prompt, “Dotsize = “, which never changes, and one for the current value of dotsize (e.g., 5), which will vary when the user changes the dotsize. In this example we will show how to use a single label that *concatenates or joins together two or more separate pieces of text called ‘strings’.* In this case, concatenate the prompt and the value.)

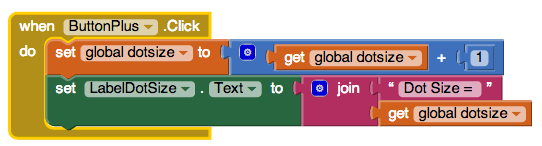
## Coding the Plus Button

Whenever the plus (‘+’) button is clicked it should perform the following operations:

1. Add 1 to the global dotsize.
2. Concatenate the prompt (“Dot Size = “) and the global dotsize using a ***join*** block from the Text drawer:



1. Display dotsize’s value in the *LabelDotsize.* To do this we set the label’s *Text* property.

Here’s what the block should look like:

It has two statements. First it adds 1 to *dotsize --* we saw how to do that above. Then it sets the label’s *Text* property to the string “Dot Size = ***dotsize*** “ where the variable *dotsize* is replaced by its current value. The coding for the (-) button is left as a mini-project.

# Reflection: For Your Portfolio

In your portfolio, create a new page named ***Paint Pot Tutorial*** and answer the following questions:

# Explain the meaning of the statements shown here, both in pseudocode and App Inventor. For example, suppose the variable *X* has the value 10 before the statement is executed. What value would it have after the statement is executed.

|  |  |
| --- | --- |
| Set X to X + 1 |  |

# One aspect of abstraction is that it helps to reduce details to focus on what's relevant. How does the use of a variable, such as *dotsize*, instead of a value, such as '5', help to reduce detail and focus on what is essential in this program.

|  |  |
| --- | --- |
|  |  |

# Exercises

To practice your skills and test your knowledge try some [interactive exercises](https://ram8647.appspot.com/mobileCSP/unit?unit=22&lesson=150).

# Mini Projects

The [next lesson](https://docs.google.com/document/d/1s7PTuvw0fg03iEVUIW11yvHb1TzZdk6T_woT4grvQZY/edit#heading=h.o6tft5swp3bj) provides some ideas and hints for adding enhancements to your app.