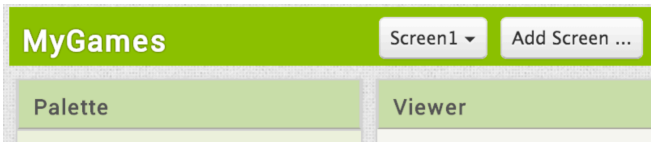
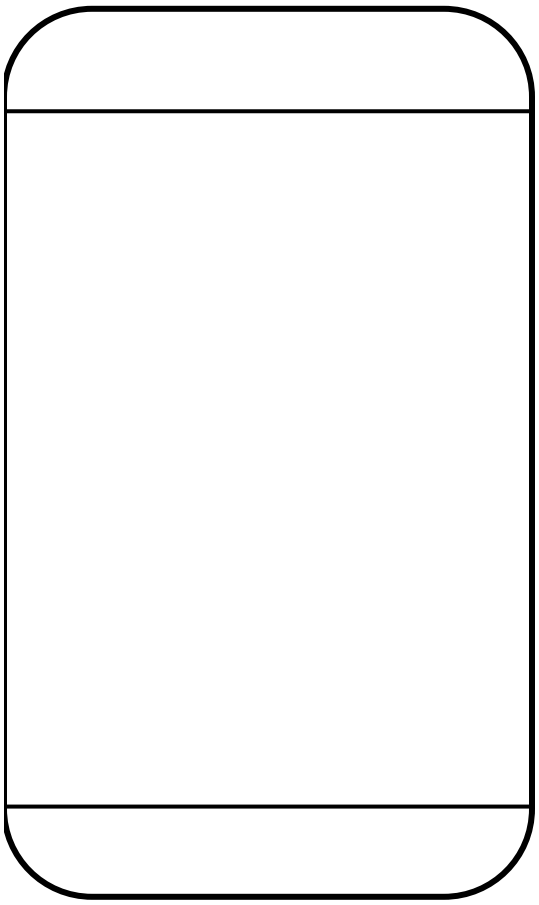
Collection of Games

Let's create an app that contains a couple different games. Let's start with 2 different games: Pong and Paint.

To have 2 different games in this one app, we'll need a minimum of 3 screens. We can have more depending on how we design our games. You can add a new screen using the button that says "Add Screen."





NOTE: The default screen1 will *always* be the first screen that appears when your finished app is loaded.

Design your home screen first.

Home Screen

What do we need to put on the home screen? Before we even ask that question, we should think about what purpose the home screen should serve. One main purpose is to navigate to the different games you have. What are some other purposes of the home screen?

_____

_____

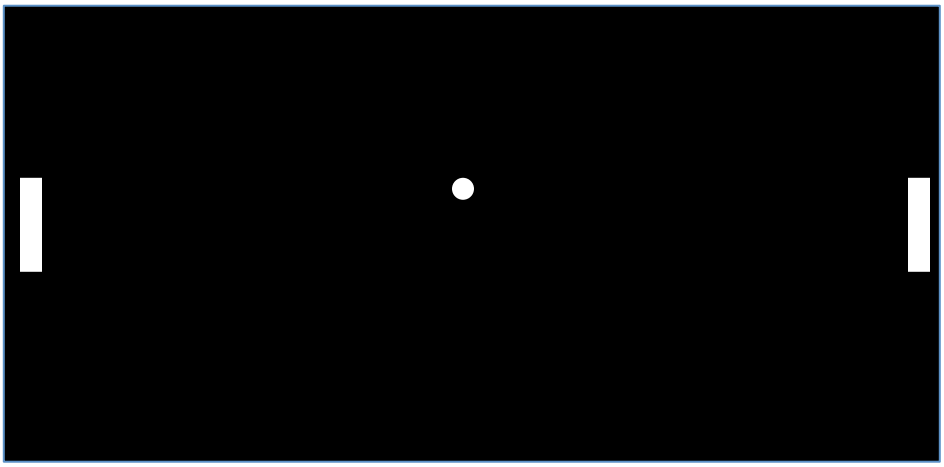
What components will you add to your home screen to navigate to your games? (hint: think about how you navigate to screens in other apps you know)

_____

_____

Pong – Your first game!

Getting Started - What's a pong game?



For those who are unfamiliar, pong is a virtual table tennis (ping pong) game. There is one "paddle" at each end of the screen and a "ball" as depicted above. The objective is for a player to win by sending the ball back to the opponent so that she misses it.

How do we want the game to work?

Before we start programming any part of the game, we must define how we want it to work, known as creating a "spec." The spec is the list of components and properties the game will contain to accomplish the objective stated above.

Let's think about starting the game, playing the game, and ending the game.

The game can start in a few different ways:

- The user clicks a notification prompt and then the game starts.
- The ball starts moving immediately when the game opens.
- The user flings the ball to begin the game.

This tutorial will guide you through the first option (the notification), but you should explore the other options after this tutorial!

Next up, how will the game be played?

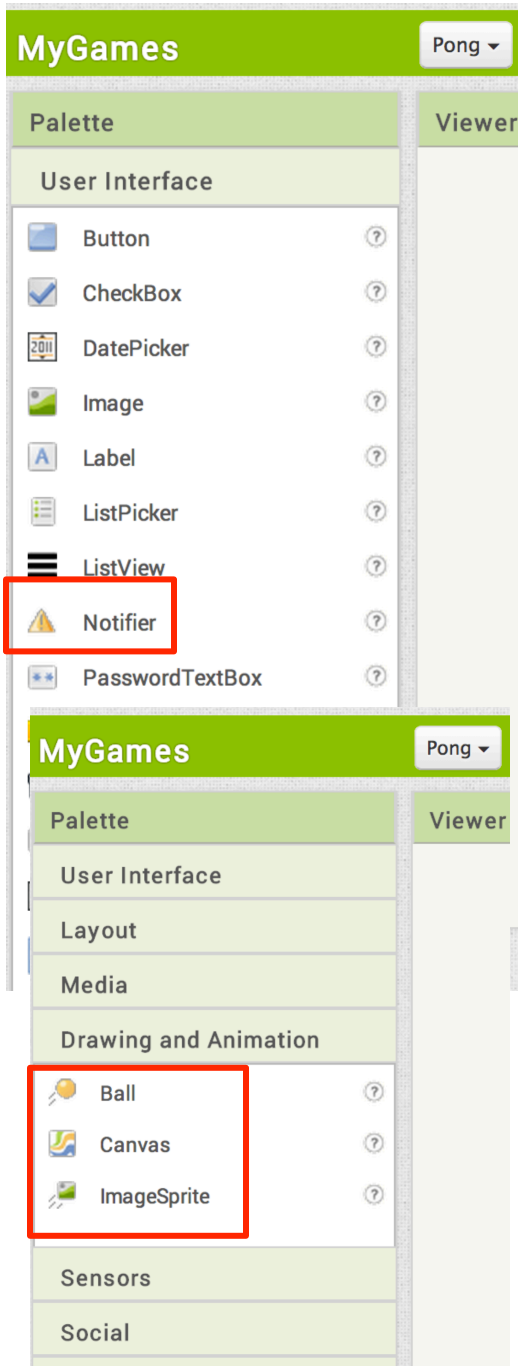
1. Players will slide the paddles up and down to keep the ball from hitting the edges on the left or right of the screen.
2. The ball will bounce when it bumps into the top or bottom of the screen.
3. The ball will bounce when it bumps into one of the paddles.
4. The ball will stop when it hits either the left or the right side of the screen.

Finally, when a player loses, how will the game wrap up? Once again, there are a few different options:

- Have a notification pop up saying who won and who lost the game.
- Reset the game.
- Both of the above.

Now that we've listed out how the game should function, we'll explore the components needed.

- Notifier (User Interface)
- Canvas (Drawing and Animation)
- Image Sprite (Drawing and Animation)
- Ball (Drawing and Animation)



Notifier Component

The Notifier component allows the app to issue alerts to the user while she is using the app. What are some ways that the Notifier can be used in an app?

_____

_____

Canvas Component

The Canvas is the component on which image sprites can be moved or the user can draw. How do you think we'll use the Canvas here?

_____

_____

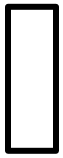
Image Sprite Component

Image sprites are images placed on canvas that can move and react to user input. The ball is also an image sprite. You can upload images that you would like to use for image sprites. What else will we use image sprites for?

_____

_____

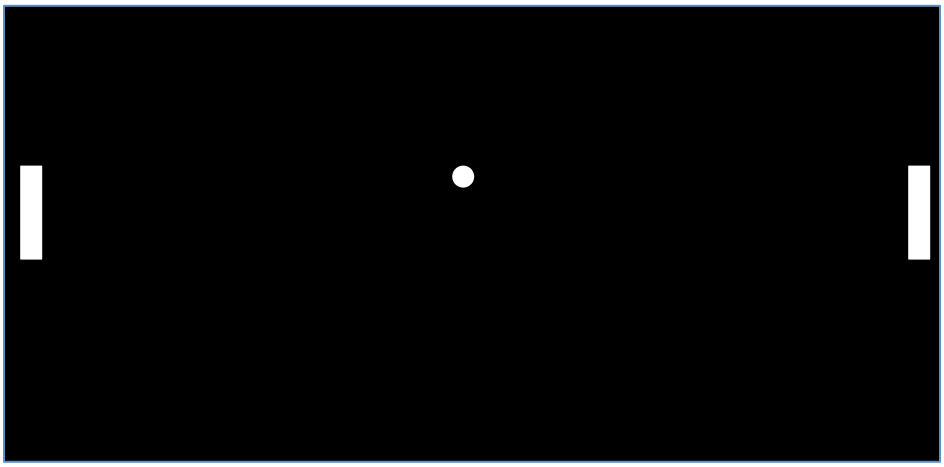
You can create your own paddle images to use for the pong game by drawing a simple rectangle shape and saving it as an image (like the one shown below). Any word processing application on your computer will allow you to do this.



Good Coding Practices

As a general rule, you always want to name your components something that makes sense. If you were to look at your list of components, could you immediately tell what each one is supposed to do? You want to make sure you leave enough “hints” so that you or someone else can come back to your code and understand what’s happening easily. This is called descriptive naming. The name of the component should reflect its function or purpose. You can select a component under the “Components” panel (to the right of the viewer) and then click the “Rename” button at the bottom to make the component name more descriptive.

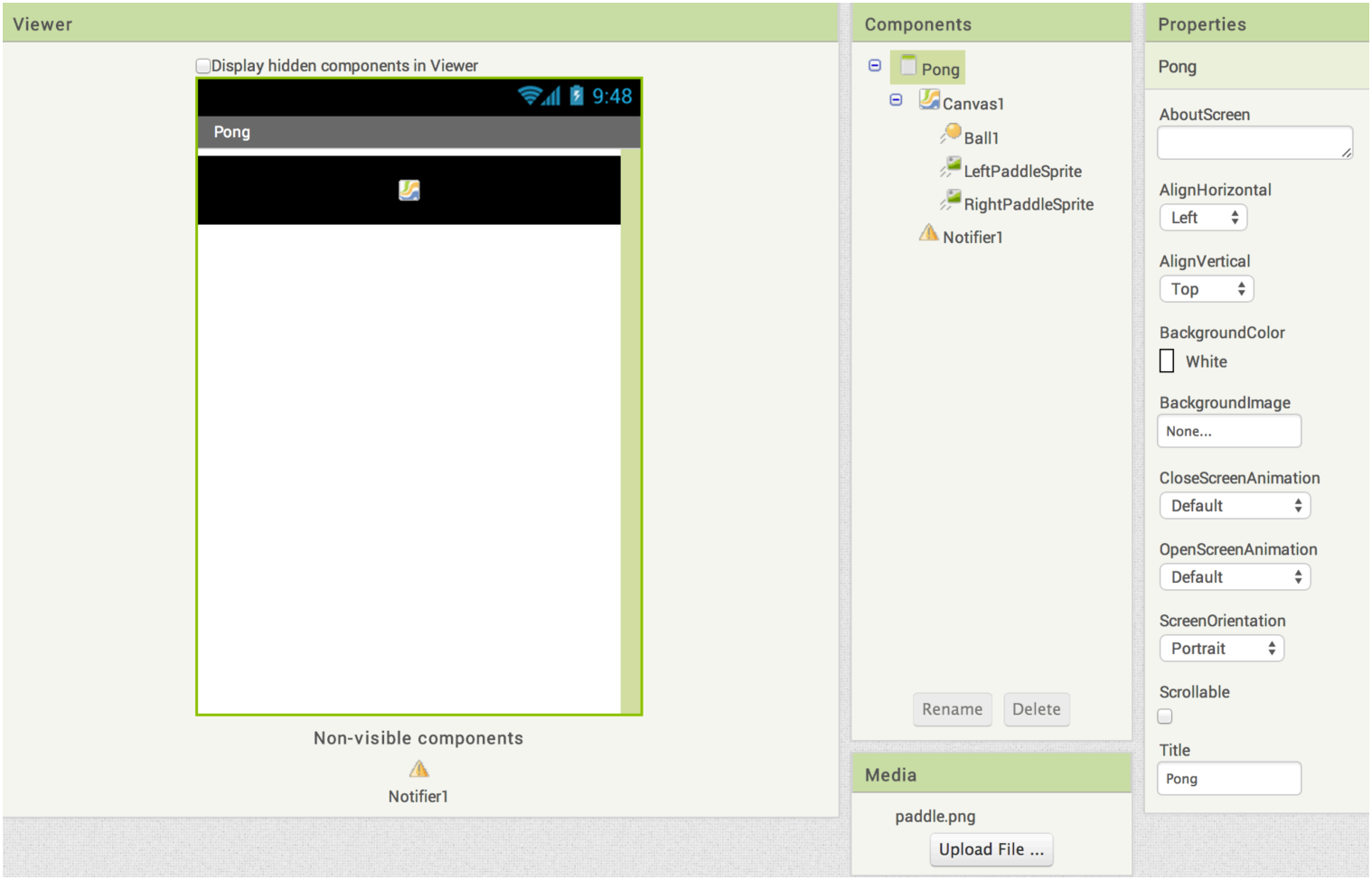
Given everything discussed so far, let’s build the pong screen. Here’s the sample image again. Don’t worry too much about putting the paddles and balls in exactly the same place as the image. We can take care of the positioning in the blocks later.



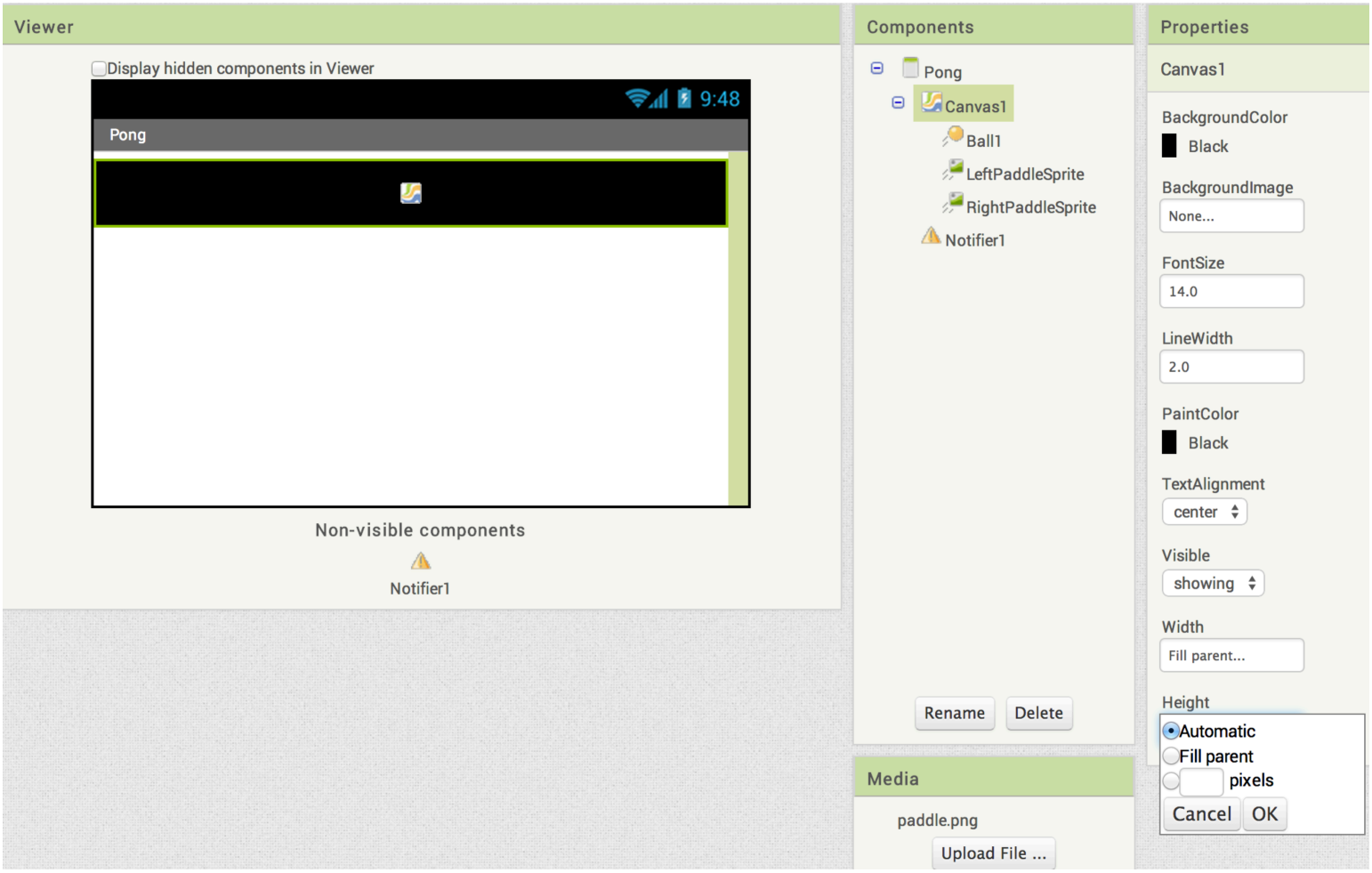
Start by placing the canvas component on the screen. Make sure to do this before placing the image sprite. The image sprite components only work on a canvas.

You should have the same components on your screen as the screenshot below. Can you find where you should upload the paddle image in the screen shot below? And where to change the orientation of the screen?

Also, you'll see that there's a "Scrollable" option in the properties. You'll want to click on the little check mark there so it goes away. Without the little blue check mark, this means the screen will NOT scroll.

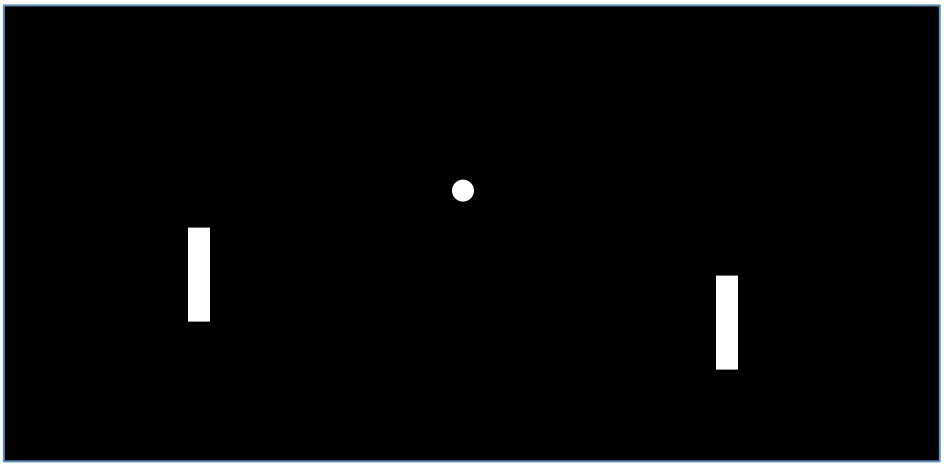


How can you make the canvas fill the entire screen? Under the properties of the canvas, there should be “Width” and “Height” fields. You can change these to “Fill parent.” The “parent” of the canvas is the Pong screen, and you can tell because it’s listed under the Pong screen in the “Components” section. So when we say “Fill parent,” we want the height and width to expand to fill the height and width of the screen.

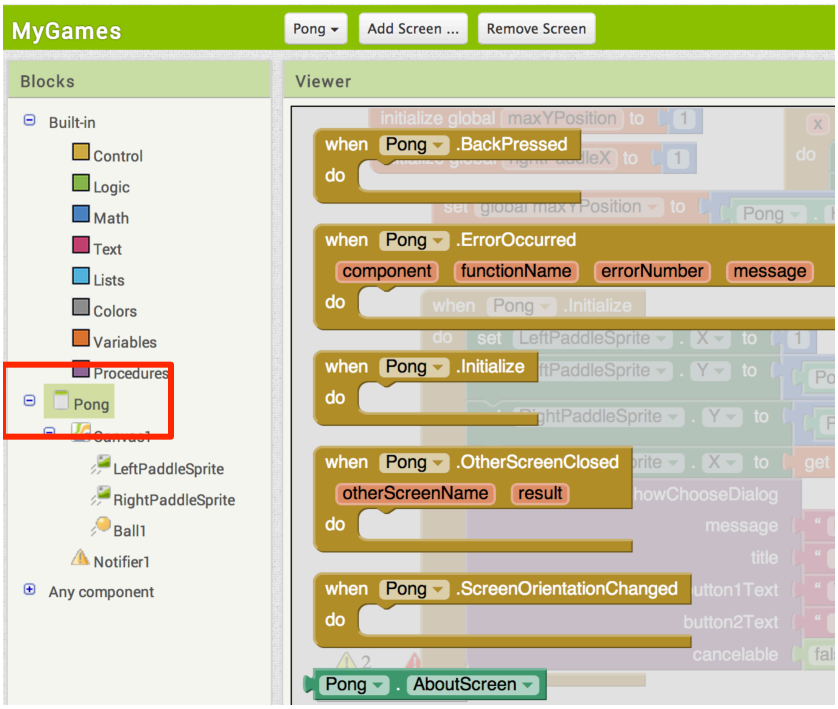


Building your blocks

After dragging out components and uploading our paddle images, our screen looks something like this:



How do we move our paddles to the proper starting positions? Let's explore the blocks available for the screen.

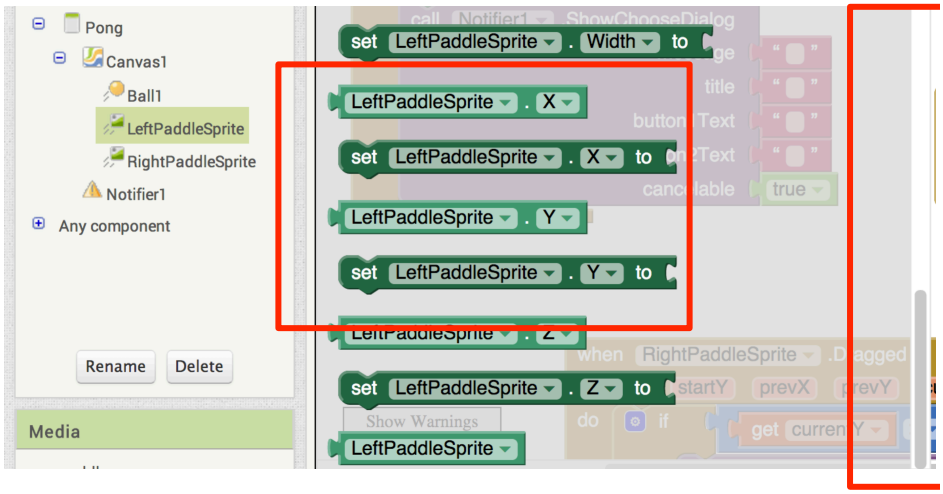


Switch over to your blocks view and select your screen under the blocks on the left.

The "when Pong.Initialize" block is interesting for us. We can use that to change the settings when the screen initializes. Drag this block out.

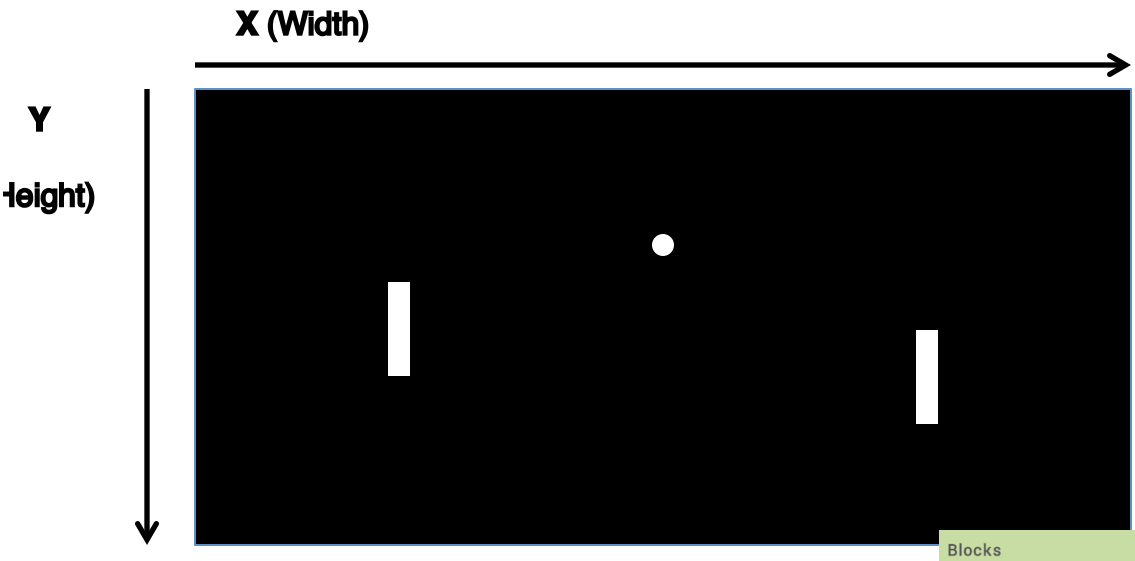
Notice, this block is a tan/brown color – same as the "when Button1.click" block! That indicates that this, too, is an event handler. As a result, when the screen initializes, everything within the event handler block will be executed.

How do we use this to change the placement of our paddles? Let's explore the properties of the paddle sprites. Let's select one of the paddle sprites in the blocks editor and scroll all the way to the bottom of the list of blocks. To our right, we see blocks to set the X and Y values for the left paddle sprite.

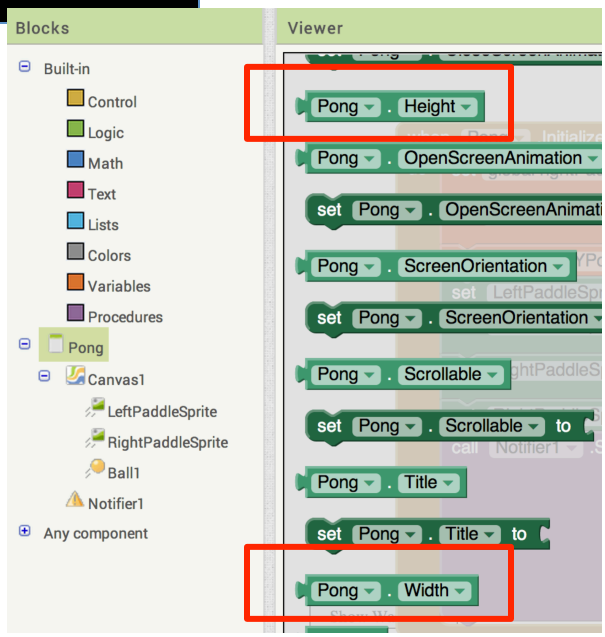


If you don't see the green blocks after selecting one of the paddle sprites, remember that there's a scroll bar to the right of the grey-ed area. Click and drag this to scroll up and down so you can view all of the blocks for a component.

X and Y are the coordinates for where the **top left corner** of the image sprite will be placed on the screen. Here is how the X and Y coordinates are mapped onto the screen.



Given that we want the paddles to be on either end of the screen, what would we want to set the X to what values for the left and right paddles? (Hint: you can get the largest possible X and Y values by using the Height and Width properties from the screen, and X and Y refer to the **top left corner of the image sprite placement.**)

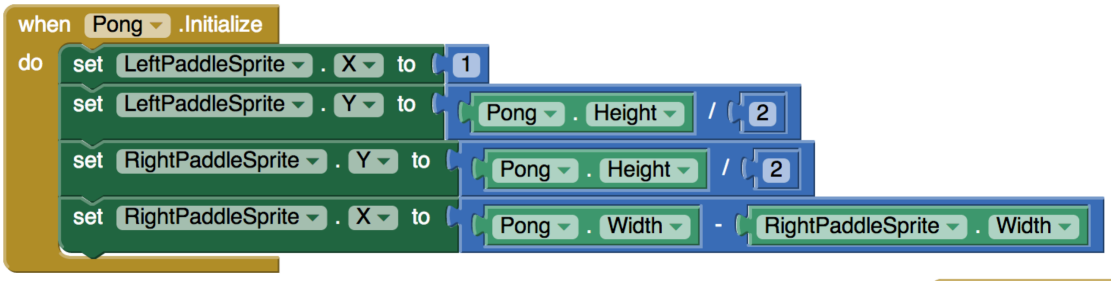


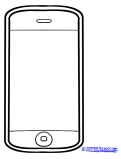
Left Paddle: _____

Right Paddle: _____

What about the Y coordinate property? What value should they be set as?

You should have something that looks like this:





Test it out!

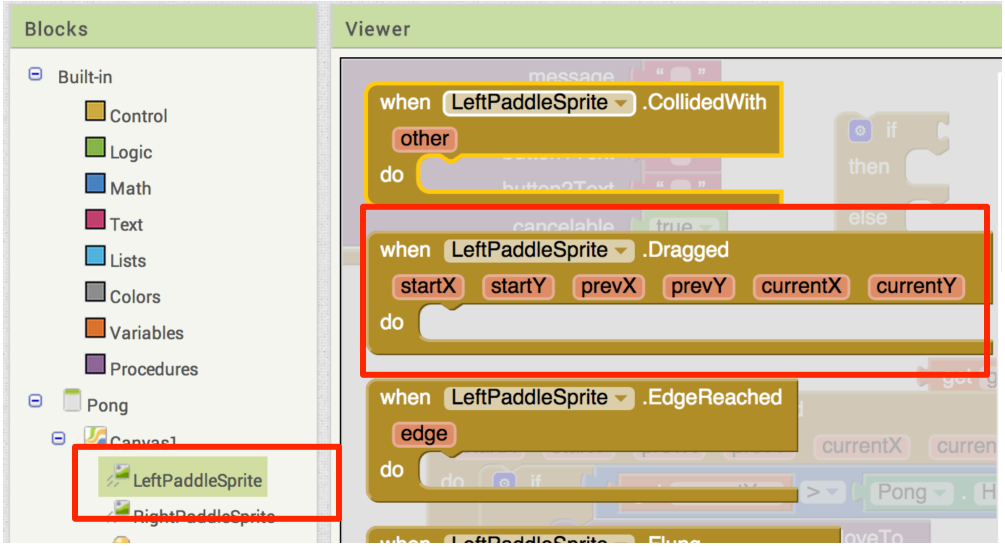
Try out your app and see if the paddles end up on either side of the screen even though that's not where you placed them in the designer.

Making the Paddles Move

First, let's define how we want our paddles to move.

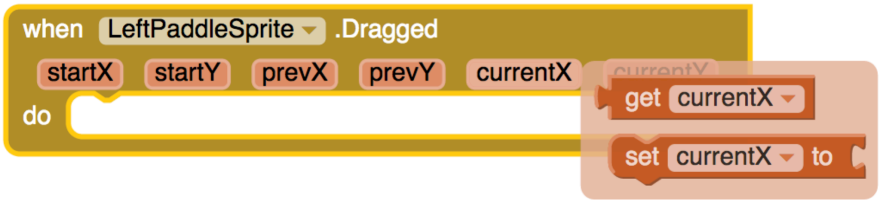
- Paddles should move up and down with a finger drag
- Paddles should stay aligned with their respective sides (the left paddle should stay against the left edge of the screen, the right paddle should stay against the right side of the screen)
- Paddles should not get dragged off the screen

Let's explore what blocks each of our paddle sprites has to offer again.



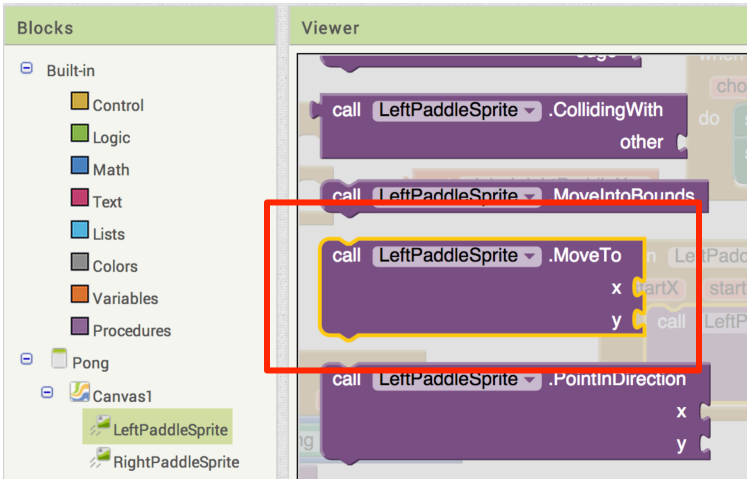
Looking at the LeftPaddleSprite, we see that there is a "when LeftPaddleSprite.Dragged" event handler! This looks precisely like what we need.

"startX" and "startY" will contain the X and Y values for where your finger first touches the screen during a drag event, and "currentX" and "currentY" will contain the X and Y values for where your finger ends during the drag event.

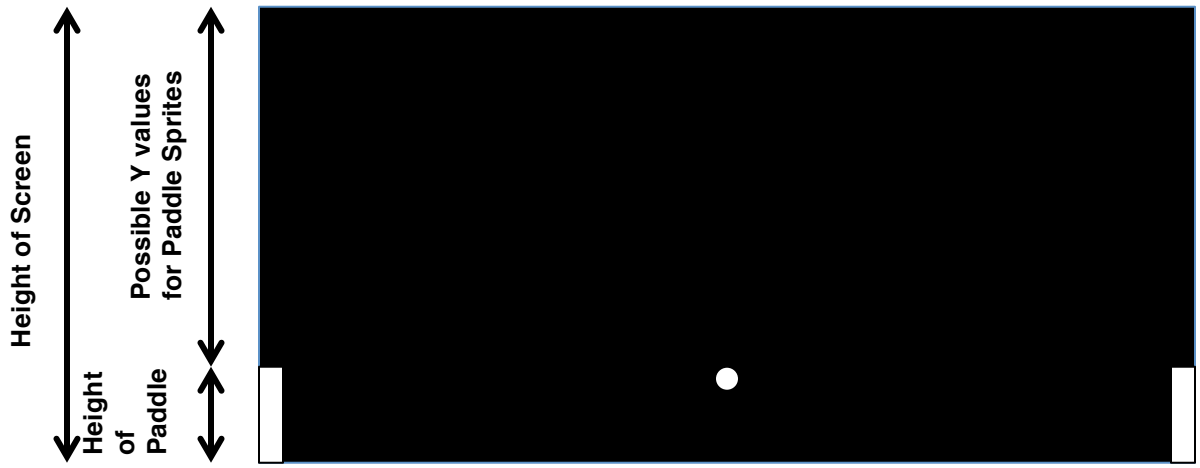


Click and drag this event handler out. If you hold the mouse over any of the little orange boxes, without clicking, two blocks will appear: a "get" block and a "set" block. Remember, a "get" block will allow you to use the value named in the block.

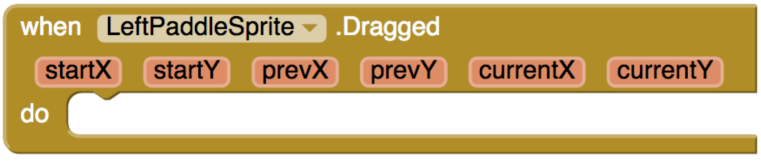
Looking again at the LeftPaddleSprite blocks, we find a procedure called “MoveTo.” This looks useful too!

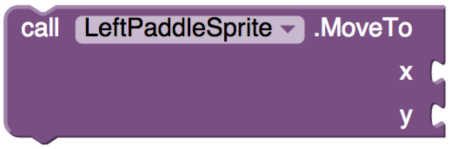


Remember, X and Y refer to the top left corner of the image sprite placement. In that case, the maximum Y value (vertical value) that either of the paddles can be is **the height of the screen minus the height of the paddle.**



So in the case that a user drags the paddle beyond that maximum, we do not want the paddle sprite to continue and move off the screen. Let’s review. These are the two blocks we’ve discovered so far:





We need more blocks than these two since we have a “condition” to satisfy. We need to keep the paddles from moving off the screen.

Setting Up the Paddle Condition

IF *the current Y position of the finger is greater than (screen height minus paddle height)*

THEN *The paddle should move to (screen height minus paddle height) for Y*

and stay at the original X position

ELSE *The paddle should move to (the current Y position)*

and stay at the original X position

So how do we do this using our blocks? Let's take a look in the "Control" group of the blocks editor. These are blocks to help us to specify general behavior (not specific to components).

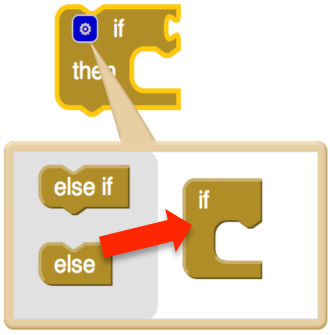
There's a block that matches up with the "If" and "Then" lines we described above, but what about the "Else" part? Let's drag out this block and see.

There's a blue gear in the upper left corner of the block.

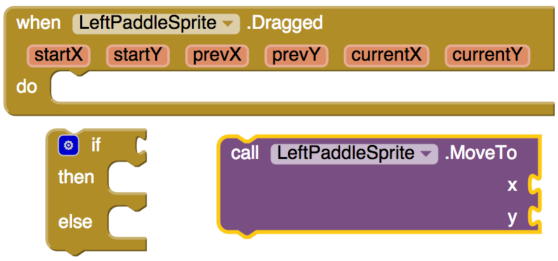
Click on it and some options will appear, just like in the picture to the right.

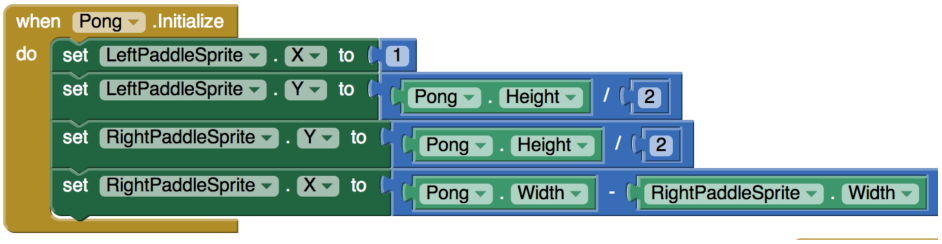
Click and drag the "else" piece into the empty slot like the red arrow shows on the right.

How does the "if-then" block change afterwards?



Now that we've got our "if-then-else" block, how do we fill it in? Let's recall that we have the following blocks picked out and also set the paddles to start at the following values.





So we can modify our original statement to read the following:

IF *the current Y position of the finger is greater than (screen height minus paddle height)*

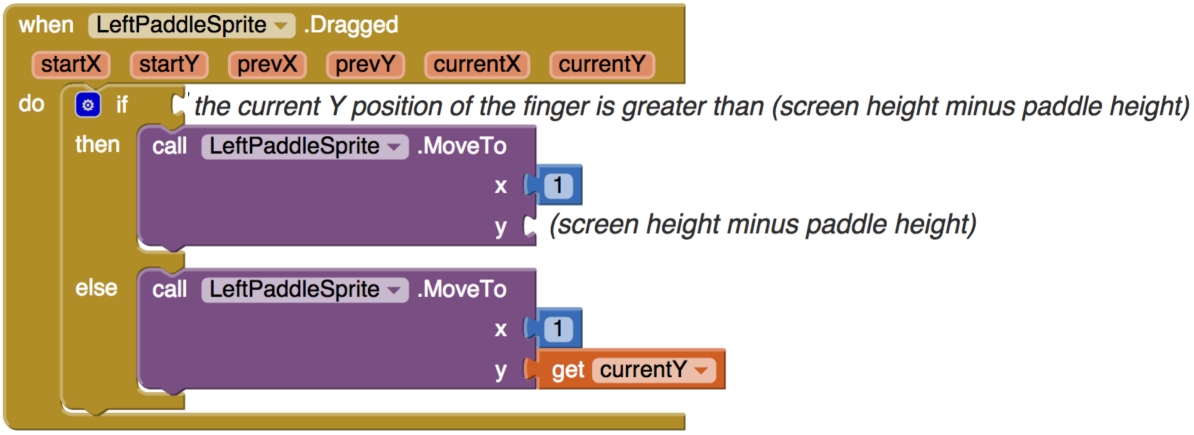
THEN *The paddle should move to (screen height minus paddle height) for Y*

and stay at "1" for X

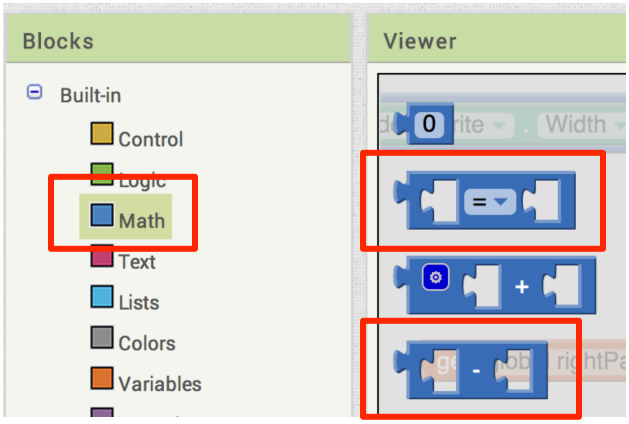
ELSE *The paddle should move to (the current Y position)*

and stay at "1" for X

In both the "then" and "else" case we want the paddle sprite to move to a position, so let's fill in our event handler the best we can so far.



What blocks do we need to make the condition for the “if” part above? We already know how to get the currentY position. What about seeing if one value is greater than another or subtracting one value from another? These are math functions that we’re trying to implement, so let’s take a look at the blocks in “Math.”



Under “Math,” there’s a block that allows us to test if two blocks are equal. This isn’t exactly what we were looking for, but notice the arrow next to the equal sign in the middle of that block. Let’s drag this block out and see what it does.

What do you notice when you drag it out and click on the down arrow in the middle?

_____

_____

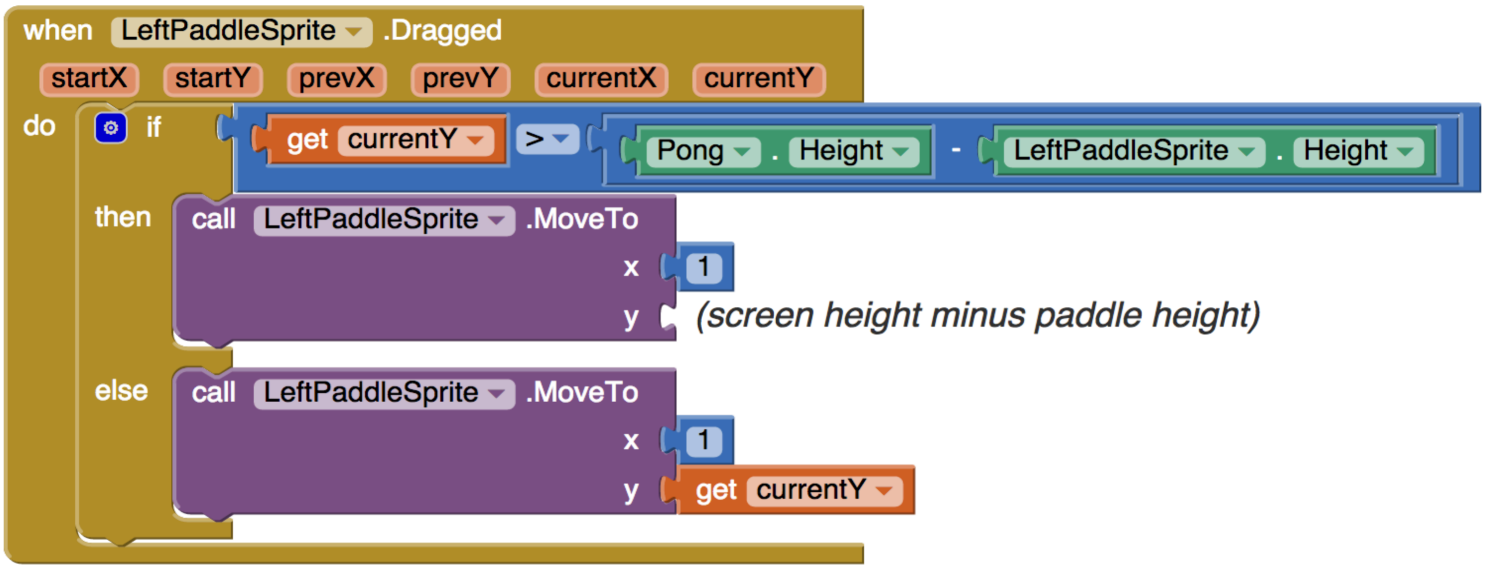
There’s also a block that allows us to do subtraction between 2 other blocks. We need that one too, so let’s drag it out. This is the statement that we want to implement.

> ***IF*** *the current Y position of the finger is greater than (screen height minus paddle height)*

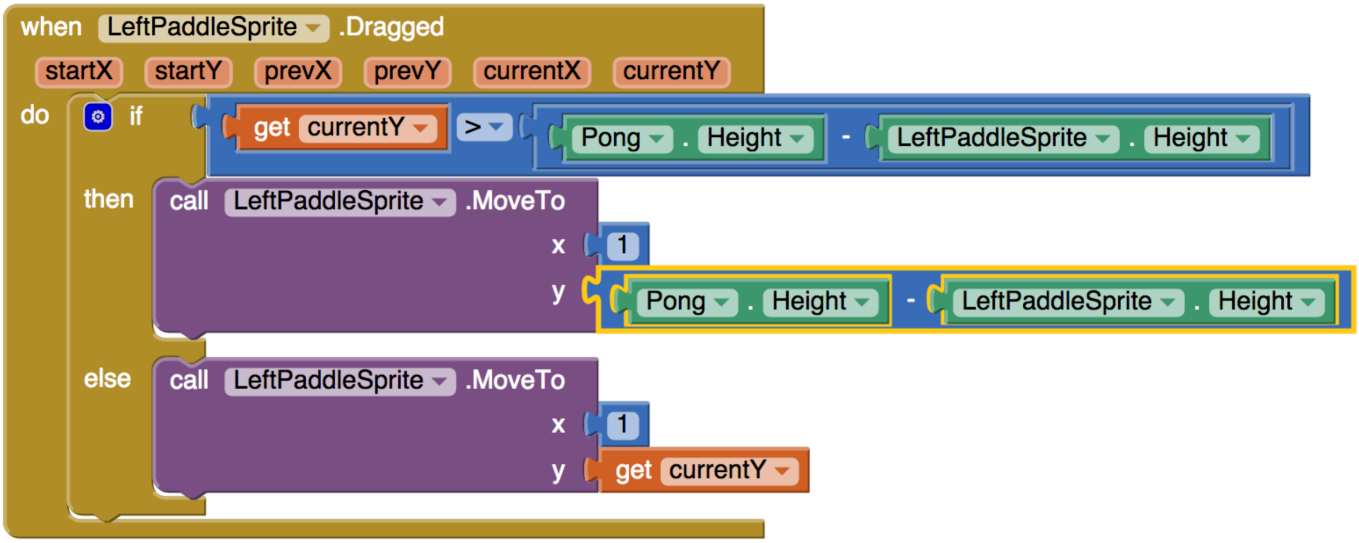
We can rewrite it as this

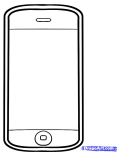
> ***IF*** *the currentY > (Pong.Height – LeftPaddleSprite.Height)*

And as blocks:



Only one more block to fill in here! We already know where to find the subtraction block, so let’s go ahead and finish this off.



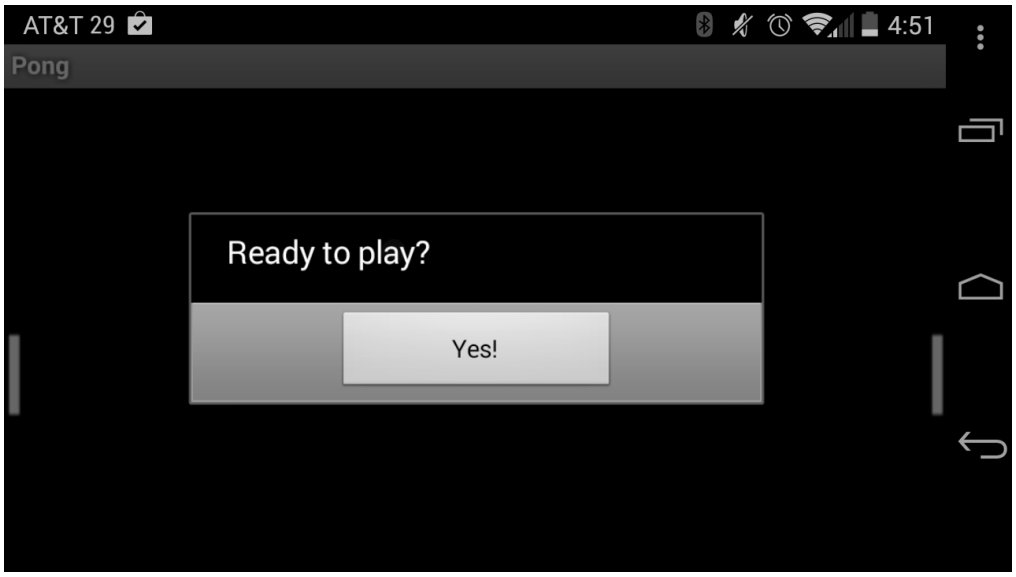


Test it out! Does your LeftPaddleSprite move up and down with your finger?

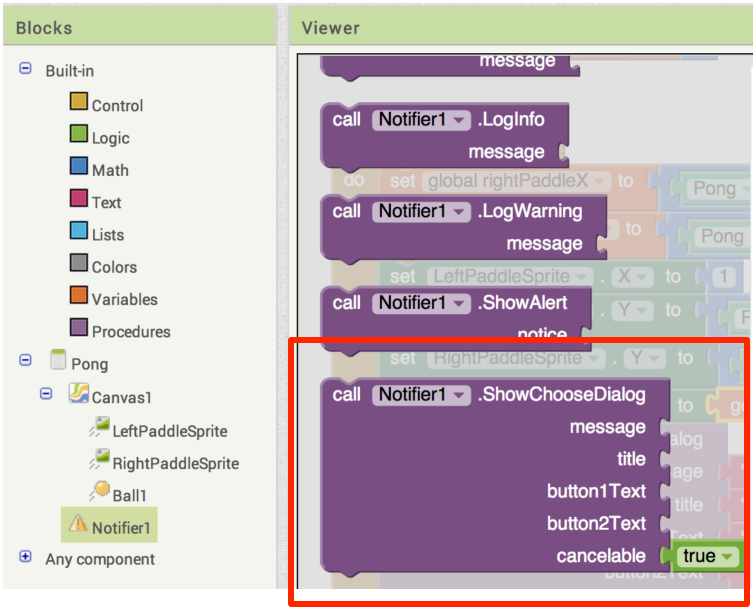
Now that we've completed the blocks needed to drag the left paddle, go ahead and do it for the right paddle as well. (Hint: It should be very similar to the left paddle with only a couple of differences.)

Adding the Notifier

Now that we have the paddles and ball in place, how can we make the ball start moving? This is where the notifier comes in. The notifier can display alerts and messages to the user, and also take responses from the user. Here's a screen shot of what the notifier "Choose Dialogue" box can look like:

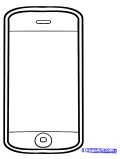


In this case, after the user clicks "Yes!" the game begins. Select the notifier component in the blocks editor. Let's explore how we'll get the above prompt to show up on the screen.



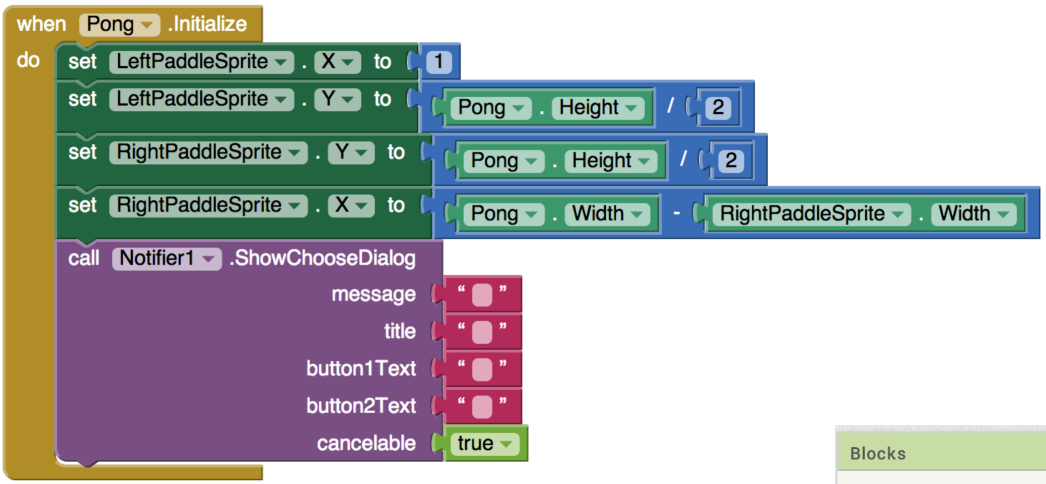
Taking a look through the blocks available for the notifier component, we see there is one called "Notifier1.ShowChooseDialog." This block takes different inputs: message, title, button1Text, button2Text, and cancelable. The first four inputs take text, and the last one takes true or false.

Where should the "ShowChooseDialog" block be placed? (Hint: this is a procedure block and it needs to be placed within an event handler block.)



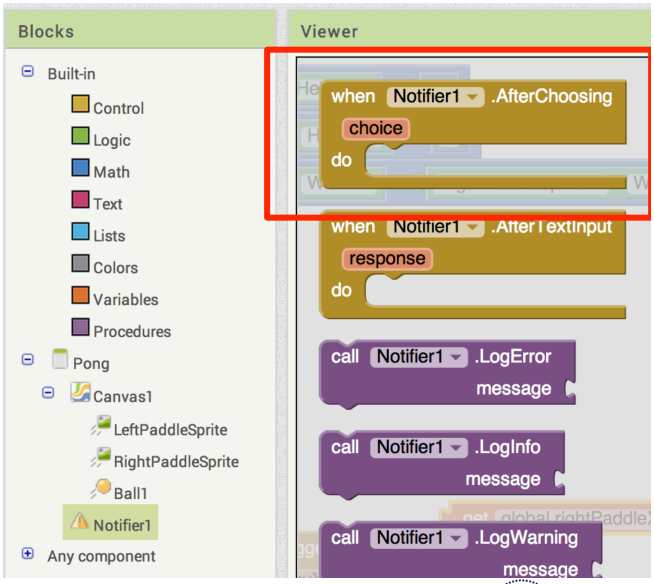
Test it out!

The initialize block should look something like the following now after dragging empty text boxes to the "ShowChooseDialog" inputs. Play around with those inputs and see how that changes the messaging on the Choose Dialog.

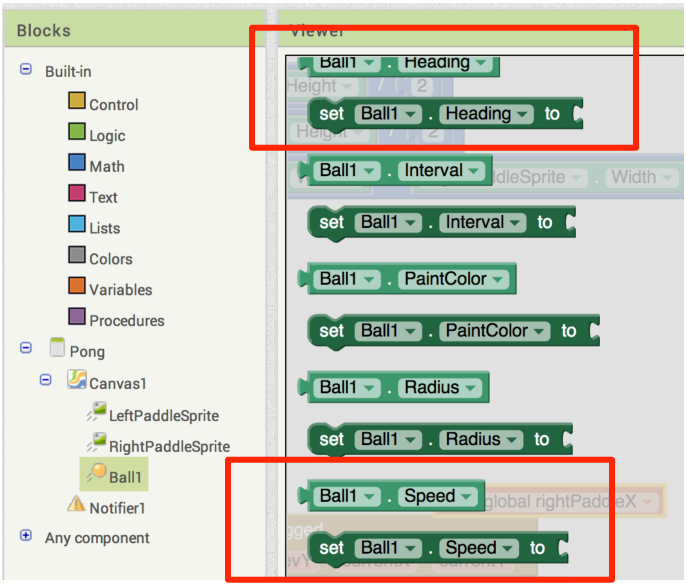


What happens when you click on a button on the notifier dialog? What do we want to happen? Ideally, the game should begin after pressing a button to make the "Choose Dialogue" box go away. In that case, let's explore the blocks available for the notifier again.

The event that occurs after the user chooses an option from the "Choose Dialogue" is called "AfterChoosing." We can use that block to make the game start after the user has chosen a button from the notifier box.



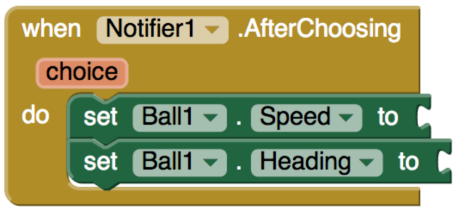
technovation



How can we make the ball start moving? Let's take a look at the ball sprite and its attributes. Select the ball component when in the blocks editor and scroll down.

The ball has properties "Heading" and "Speed." Speed describes how fast the ball is traveling on the screen, and the heading is the direction in which the ball is traveling. Now you have an event handler (Notifier1.AfterChoosing) and 2 "set" procedures. How do you put the two together?

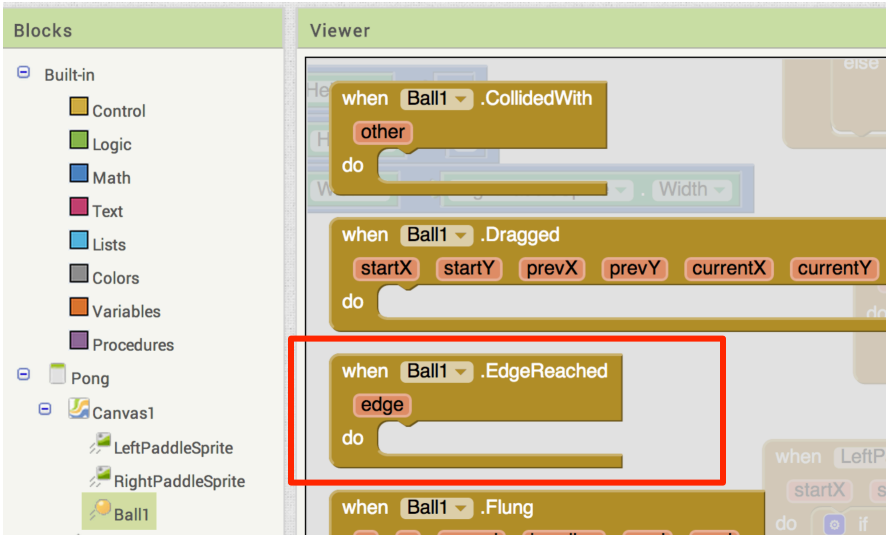
You should have something that looks like the picture on the right. What values to we want to set the Speed and Heading to be? These both take numbers as input. Play around with the values and test it out! (Hint: For the Heading, try "0", "90", "180", and "270" first. What do you observe about the behavior?)



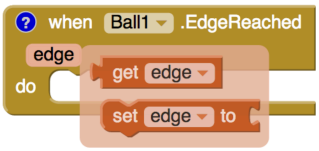
Making the Ball Bounce

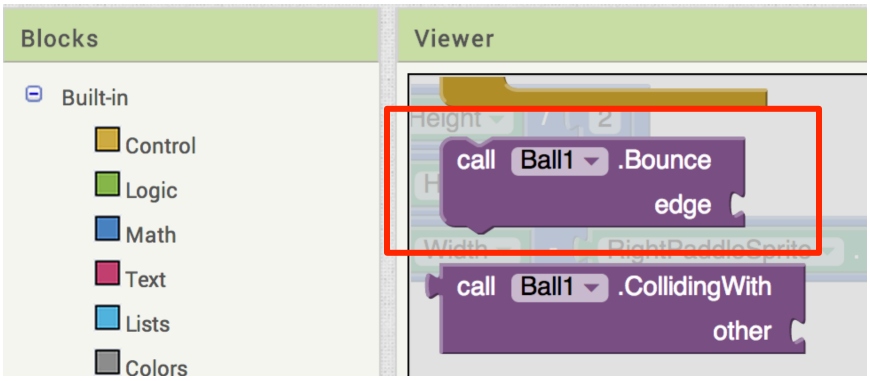
What happens when the ball bumps into the edge of the screen? We want the ball to have a natural movement to bounce off the edge. Let's take a look at the blocks available for the ball again.

There's an event handler for "EdgeReached" and a procedure for the ball to "Bounce."



In the "EdgeReached" event handler, we see "edge" in a light orange box. This will tell us which edge the ball has reached. If you hold your mouse over it (without clicking) a "get" block and a "set" block will pop up. Click and drag out the "get edge" block.





"Edge" can take on the following values.

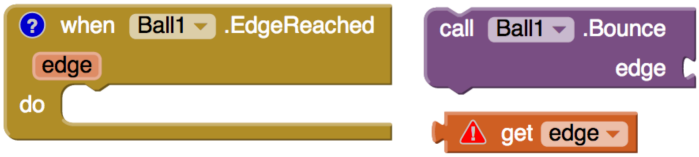
1 → Top of the screen

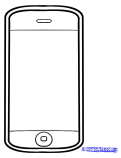
3 → Right side of the screen

-1 → Bottom of the screen

-3 → Left side of the screen

Now that you have these 3 blocks, how do you put them together?





Test it out!

Remember to test frequently and safe often. Let's check to make sure the ball actually bounces off the edges with our new code.

Stopping the ball when it reaches the left or right edge.

What do you notice when the ball reaches the left or right edge? Do we want that to happen? We want the ball to stop when it reaches the left or right edge because it means one of the players has lost! How do we specify that?

Let's break this down first.

What information do we get when the "EdgeReached" event occurs? _____

Remember, the following values tell us which edge the ball has reached.

1 → Top of the screen

3 → Right side of the screen

-1 → Bottom of the screen

-3 → Left side of the screen

Since we know how to tell which edge the ball is at, we can modify the behavior depending on the edge!

We want to say:

> ***IF*** *the edge is "right side of the screen" or "left side of the screen"*
>
> ***THEN*** *The ball should stop*
>
> ***ELSE*** *The ball should bounce off the edge*

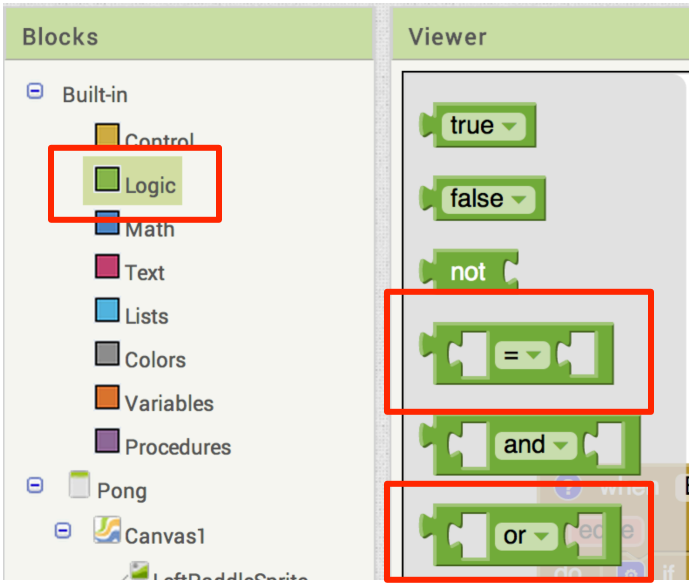
We can change the "if" part of the statement to

> ***IF*** *the edge is "3" or the edge is "-3"*

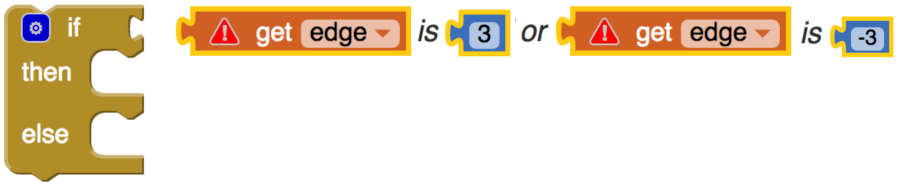
We identified the "if-then-else" block earlier for the paddles, and we'll use it again here. Using these blocks, what can we replace in the above statement? (Hint: You'll also want to use blocks from the "Math" group for your numbers.)

Checking if things are equal or true in programming is called logic, so let's check the "logic" blocks available.

Under the "logic" section, there's a block that we can use to see if one value is equal to another. There's also a block where you can choose between one set of blocks "or" another.

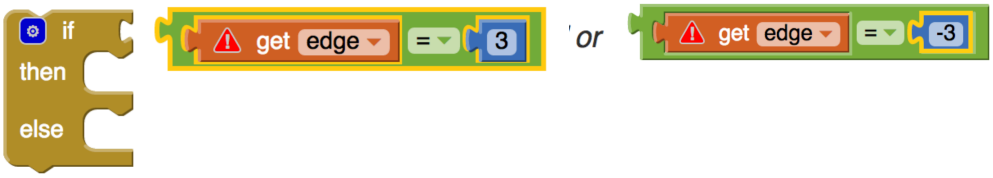


We have this statement that we want to satisfy:

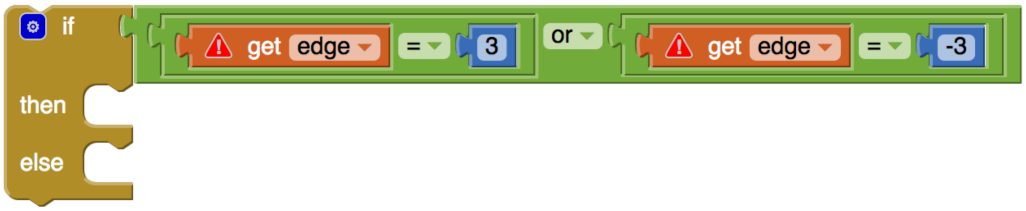


So how do we use the blocks we've just identified in the logic section to modify the statement on the left?

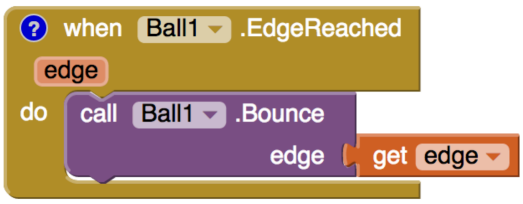
To say the edge "is" 3 is the same as saying the edge "equals" 3, so we can modify the statement above to be the following:



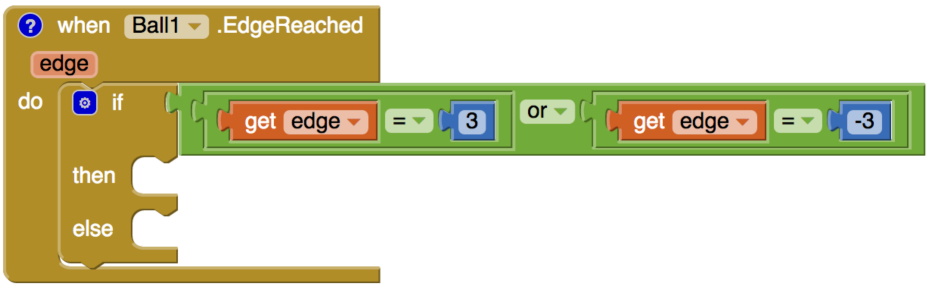
There's also a block that matches up with "or," so let's go ahead and replace that as well.



There we have it! A “condition” for our if-then-else statement. The original blocks we had for our “EdgeReached” event was this.



Now we have the following:

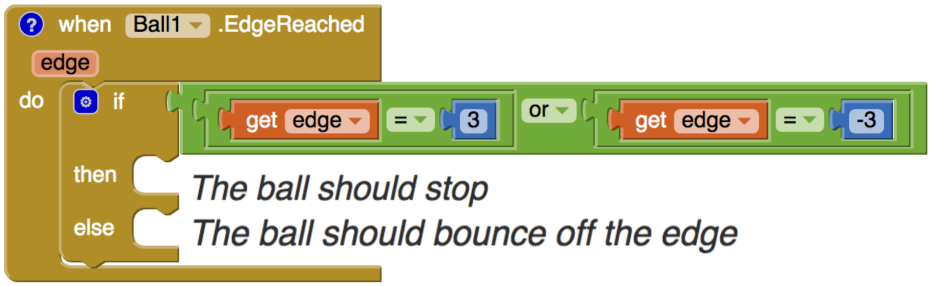


How do we fill in the “then” and “else” parts? Remember this was the original statement we wanted.

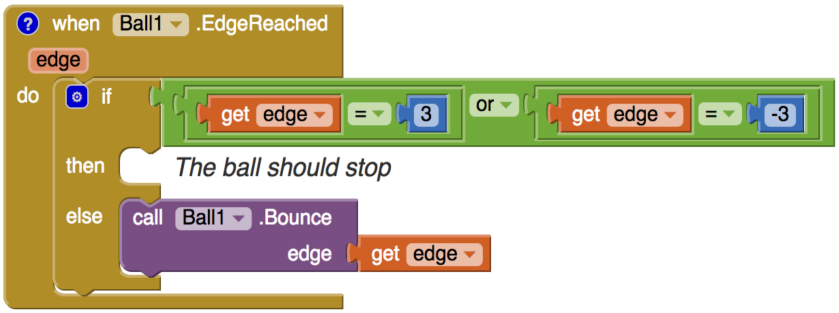
IF *the edge is “right side of the screen” or “left side of the screen”*

THEN *The ball should stop*

ELSE *The ball should bounce off the edge*

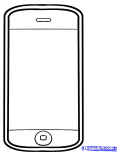


We already know which blocks will make the ball bounce off the edge, so we can go ahead and fill that in.



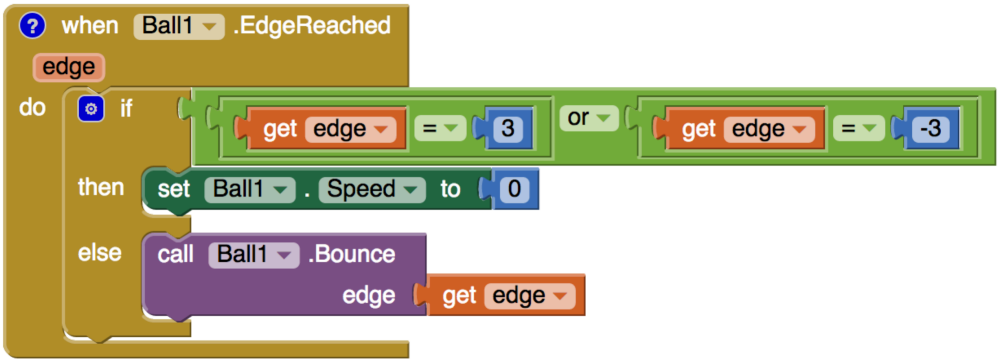
How do we replace the “then” part of this statement?

When the ball stops, it is no longer moving, and therefore its speed is what value? And how do we change the speed to that value? (Hint: look at how we made the ball start moving after the notifier message.)



Test it out! Now that you have a newly constructed set of blocks, try it out. Remember that good coding practices involving saving and testing often.

Does the ball now stop when it reaches the left or right side of the screen?



Bouncing the Ball off the Paddles

The last thing we need to do is make the ball bounce off the paddles! Image sprites have event handlers for when they collide with other image sprites.

When the ball collides with a paddle, we can treat it as a bounce off the wall on the same side of the paddle.

Remember the following to make that happen.

3 → Right side of the screen

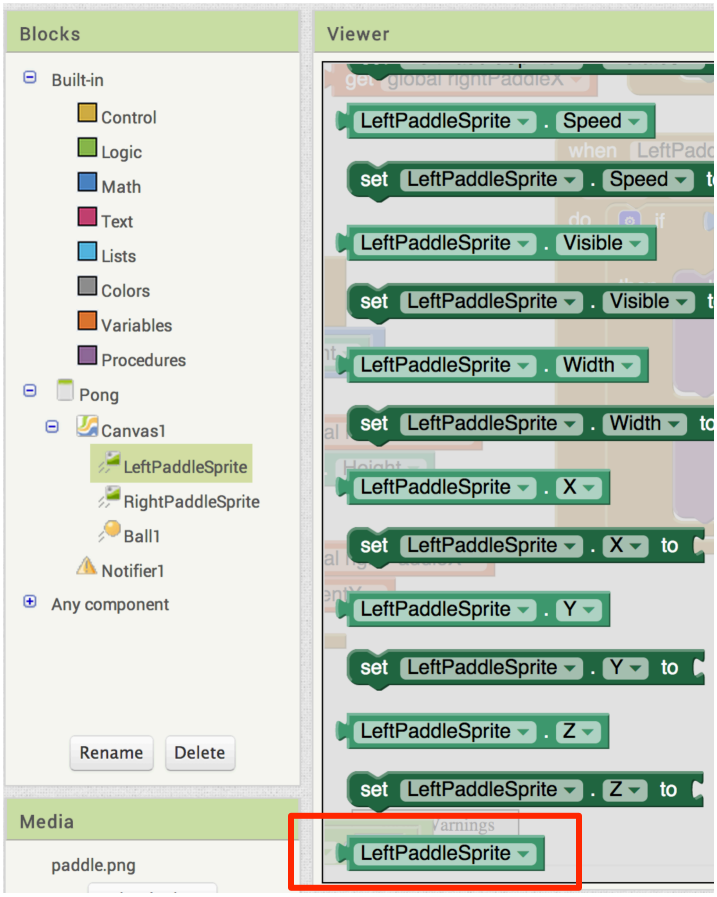
-3 → Left side of the screen

What does “other” refer here? It tells us what other image sprite the ball has collided with in this event.

What are the two possible other image sprites with which the ball can collide here? (Hint: there are only 2 other image sprites on the screen.)

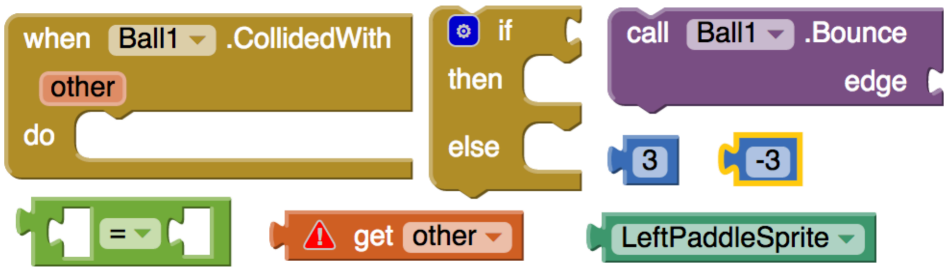
Since there are two different sprites to account for, we’ll need to use our very familiar “if-then-else” block to create different behaviors for the two collisions.

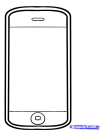
So how do we specify which paddle? Let's take a look at the blocks for the paddle sprites.



If you scroll all the way to the bottom after selecting one of the image sprites, there's a block that just describes the sprite, not any of its properties.

We've already explored all of the blocks needed to make the ball bounce off the paddles, so let's put it together. How can we make the ball bounce off the paddles using the following blocks?





Test it out! Does the ball bounce off the paddles now?

Now how can we restart the game? Think about some of the components we've already used here. Can we reuse anything? (Hint: think about how we started the game at the beginning)

Extra challenge: How can you create difficulty levels for the game?

Game 2: Paint Pot

For the second game, let's build a painting app. If you don't already have a screen for PaintPot, go ahead and add one. You should be able to navigate to this game from your

Historical note:
PaintPot was one of the first programs developed to demonstrate the potential of personal computers, as far back as the 1970s.

What kind of features would you want to build into a Paint app?

- Drawing lines and dots with your finger
- Erasing the screen
- Choosing colors
- Customizing colors
- Anything else?

Setting up the canvas to draw

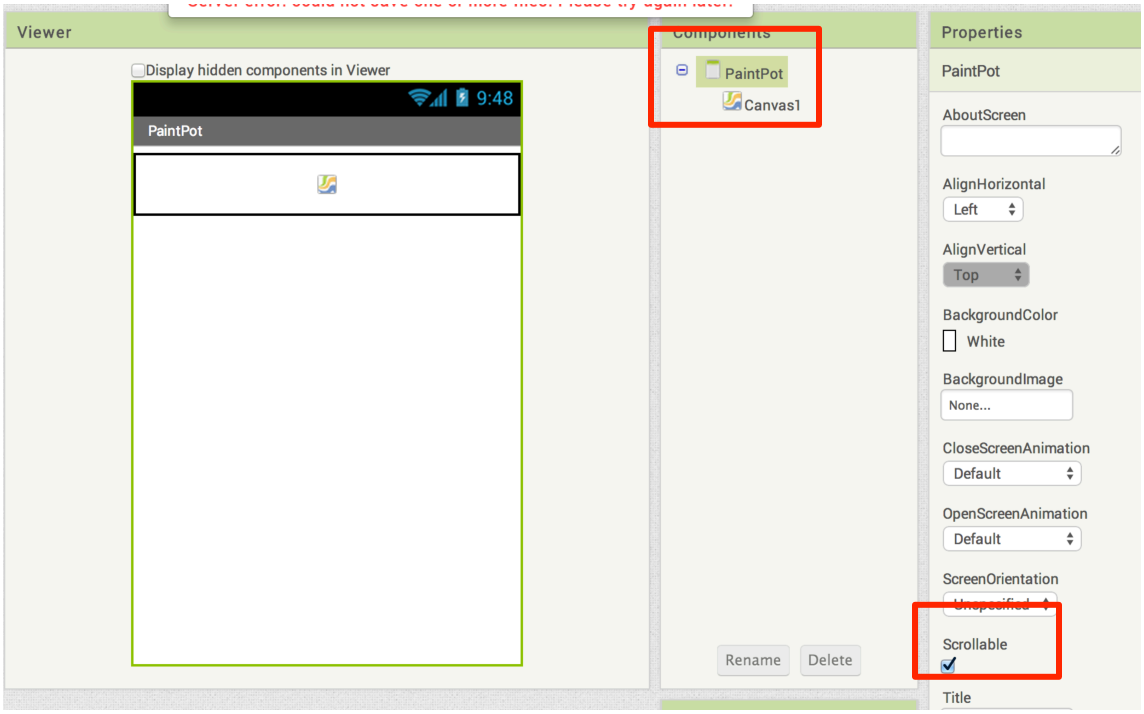
Let's go feature by feature here and start with the most important part: the drawing canvas! We used the canvas to move image sprites in the Pong game, but now we'll use it to do some drawing. Reminder: you can find the canvas component under "Drawing and Animation" in the Palette.

After you've dragged the canvas component onto the screen, take a look at the properties of the canvas. Let's change the height and width of the canvas to "Fill parent." What happens to the height and width when that changes?

What happens to the width that doesn't happen to the height? _____

If you select the screen, under Components, you'll be able to modify the properties for the screen. There's an option to make the screen scrollable. Click on the little check box under "Scrollable" to un-check the box.

What happens to the canvas once you un-check the box?

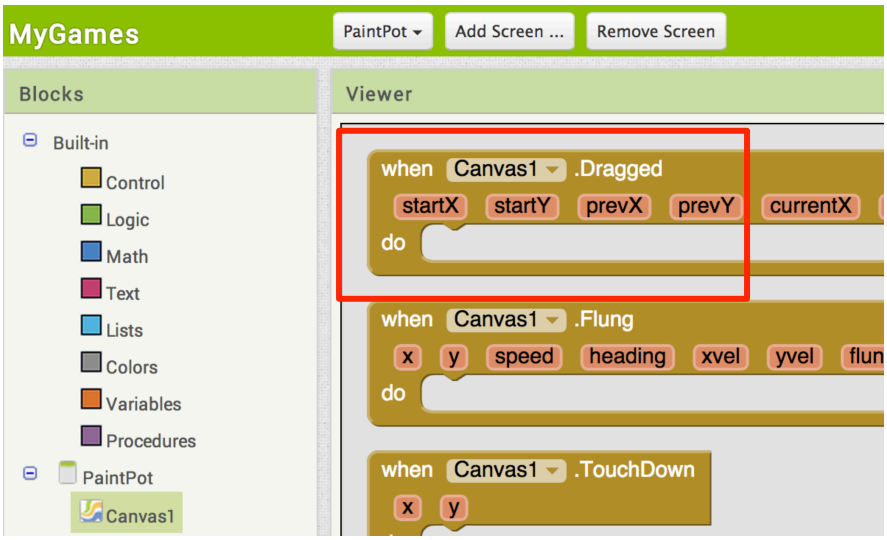


Drawing on the Canvas

Switch over to the blocks view and we can add the blocks that will allow us to draw on the canvas.

Drawing Lines

If you drag your finger slowly along the canvas, it draws a line. A drag is when you place your finger on the canvas and move your finger while keeping it in contact.



Let's take a look at the blocks available for the canvas. Looks like there's a "Dragged" event handler here.

Just like when we dragged an image sprite, we get the starting X and Y positions, the position immediately before the current one, as well as the current X and Y positions from when we're dragging our finger across the canvas.

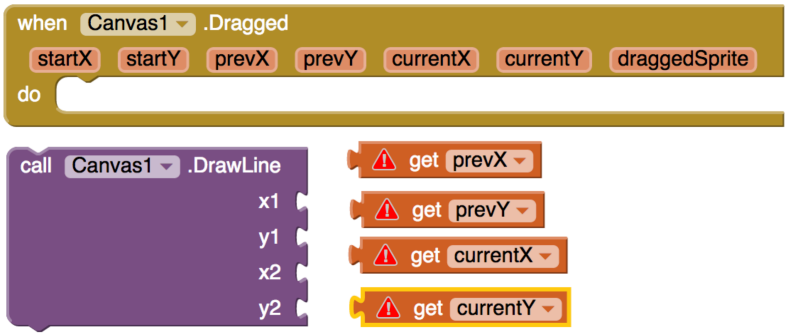
Let's see what procedures for the canvas can help us. There's a DrawLine procedure

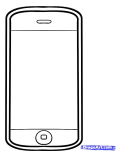
When you drag your finger across the screen, it can appear to draw a giant, curved line where you moved your finger. What you're actually doing is drawing hundreds of tiny straight lines. Because the lines are so small and connected, it appears to be curving in whatever direction you drag your finger.

The DrawLine procedure block here takes four inputs: x1, y1, x2, and y2. x1 and y1 are the X and Y positions for one point of the line, and x2 and y2 are the positions for the second point of the line. Where can we find the values that we need to fill in here?

The "Dragged" event handler contains the values for where your finger starts and where your finger starts in X and Y coordinates, so let's take a look at what blocks we have available.

How can you assemble the blocks you see on the right to draw with your finger?





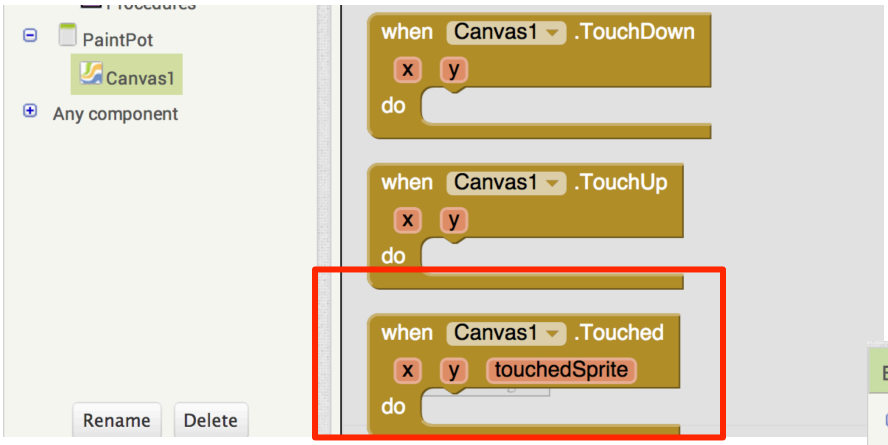
Test it out!

Remember the rules of coding: save often and test often! Try out your assembled blocks and see how they work.

What do you notice about drawing? Are there things that you cannot do?

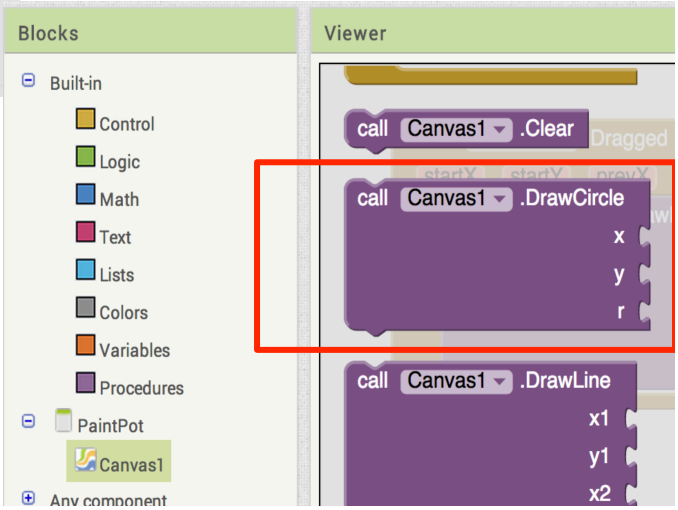
Making Dots

When you touch the canvas, you get a dot at the spot where you touch. Let's take a look at the blocks available again.



There's a "Touched" event handler that we can use.

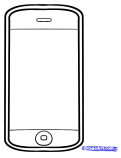
At the top of the event handler, you see **x**, **y**, and **touchedSprite**. The x and y values there tell us where the screen has been touched. What procedure do we need to use with this event handler to draw some dots?



For this touch event, make the canvas draw a small circle at the point with coordinates (x, y). There's a "DrawCircle" block that we can use for the canvas.

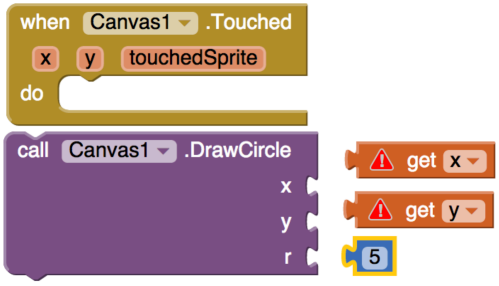
On the right side of the "DrawCircle" block are three sockets where you must specify values for the x and y coordinates where the circle should be drawn, and **r**, which is the radius of the circle. 5 usually works pretty well here. For **x** and **y**, you can use the values from the "Touched" event handler.

How do you assemble the blocks shown on the right?



Test it out!

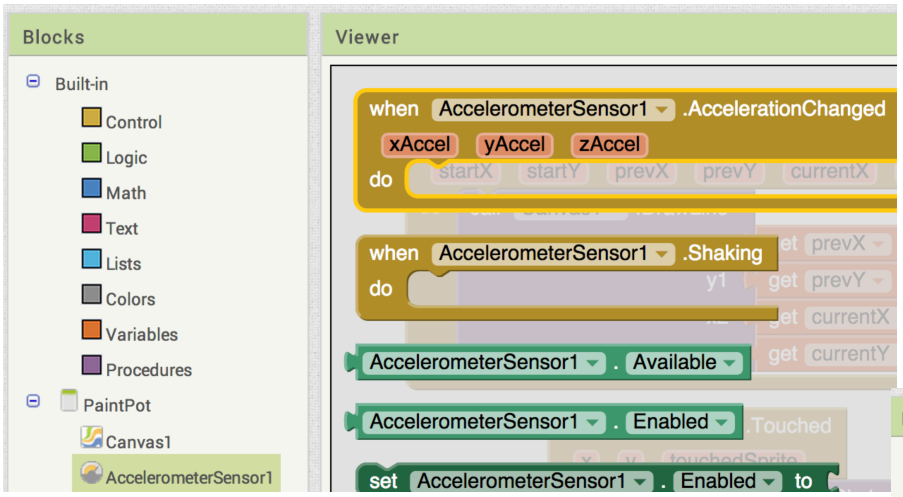
See if you can draw dots as well as lines now? Are you able to erase the screen at all? Let's add that feature next.



technovation

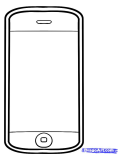
Erasing the Screen

We can erase the screen using buttons (which we learned in the Talk To Me tutorial), but we'll learn a fun new way to do that here. Switch back over to the Designer view and take look under the Sensors section. You'll find a component called AccelerometerSensor there. Accelerometer sensors are in all smartphones, and can tell applications when the phone is moving or shaking. Drag the Accelerometer component onto the screen, then switch back over to the blocks view. Let's see what kind of event handlers we have available.



There's a "Shaking" event handler that we can use. When the accelerometer detects that the phone is shaking, we can call a procedure to erase the canvas.

If we take a look under the canvas again, we see the "Clear" procedure. This will clear, or erase, everything that's currently on the canvas.

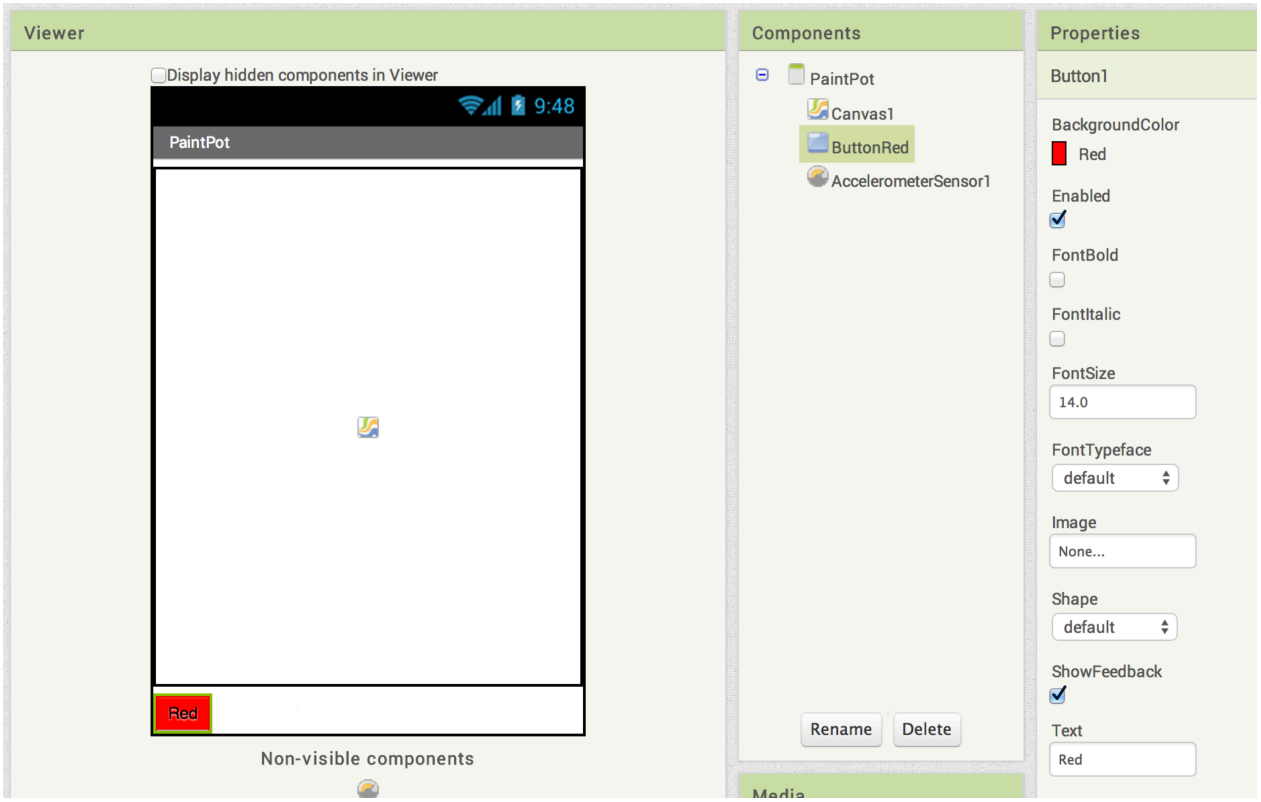


Test it out!

Put those two "Shaking" and "Clear" blocks together and see if it works!

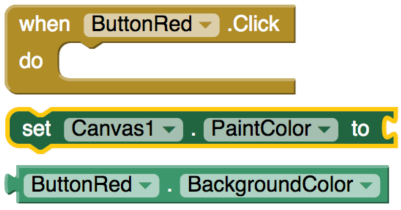
Set Up Color Selection

So far, we've only been able to draw in black. Let's add a way to change the color of the paint. Add a button to select **red** paint. Squeeze it in on the bottom of, then change the button's **Text** attribute to "Red" and make its **BackgroundColor** red.

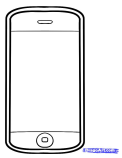


Click on Button1 in the components list in the Viewer to highlight it (it might already be highlighted) and use the "**Rename**" button to change its name from "Button1" to "ButtonRed".

Let's switch over to the blocks view and figure out how to change the paint color based on that button. We've used the button click event handler before. Go ahead and find that block again to use. You can also set the paint color for the canvas since it's a property, and you can retrieve the background color of the button for the same reason.



With the blocks shown on the left, how will you change the paint color?



Test it out!

Try out your button! See if it changes the color as expected

Explore some more!

Now that you know how to change the paint color, add as many as you like! Try out the screen arrangement components to keep things neat on the screen.

You can also program a way for the user to change the size of the dots and make custom colors.