|  |  |
| --- | --- |
| [Pong](http://en.wikipedia.org/wiki/Pong) was one of the first computer games. It consists of a paddle controlled by the user and a ball. In its simplest form the ball bounces off the four walls and the user tries to keep the ball from hitting the bottom wall. When the ball hits the bottom wall, the game is over. Variations of the game include keeping score, multiple lives, increasing the ball’s speed, sound effects, and so on.  **Objectives:** In this lesson you will learn to :   * make incremental additions to an existing program; * understand that if/else is a selection control structures in algorithms; * define and use a procedure with a parameter; * define an if/else statement to evaluate more than one condition. | ***[Click to watch Preview Video](http://www.youtube.com/watch?v=7aGbsg1D3no)*** |

# [Getting R](http://www.youtube.com/watch?v=7aGbsg1D3no)eady

# Open [App Inventor with the Pong Basic Template](http://ai2.appinventor.mit.edu/?repo=templates.appinventor.mit.edu/trincoll/csp/unit5/templates/PongBasic/PongBasic.asc) in a separate tab and follow along with the following tutorial. Once the project opens use *Save As* to rename your project ***Pong****.*

# Pong Tutorial

## Objective

You are provided with a template that implements a very basic Pong game, in which the ball will bounce off any of the 4 walls and the paddle. There is no score kept and there is no way to end the game. After the walkthrough your goal will be to use various programming techniques to add the following enhancements to the app:

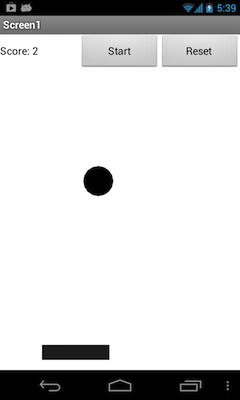
* Keep score.
* End the game when the ball touches the bottom edge.
* Add sound effects.
* Allow the user to turn the sound effects on/off.

Programming constructs you will use in this lesson are:

* If, if/else, and procedures.

## Pong Walkthrough

This walkthrough will go over the app’s main elements.



The UI for our Pong app consists of several *components* that you have used before: one *Label* (Score), two *Buttons* (Start and Reset)*,* a *HorizontalArrangement* that contains those three components, and a drawing *Canvas*.

The *Canvas* contains two components, a *Ball* and an *ImageSprite,* both of which are *animation components.* If you’ve done the *MoleMash* project, you’ve already encountered ImageSprites.

Balls and ImageSprites have many common features that we will learn about. Both components can respond to touch events and collisions with other balls or sprites or with the edges of the canvas. So they are ideal for creating a mobile Pong game.

The main difference is that an ImageSprite can have an image associated with it, in this case a black rectangle, which represents the “Paddle”.

### Canvas Dimensions

Let’s first look at some important features of the *Canvas* component:

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
| App Inventor assigns numbers to the four edges of the canvas, as shown in this diagram. These numbers are used to detect when the ball has hit the bottom edge (edge = -1).  Also, notice how the ball’s and the sprite’s *heading* property is defined: a heading of 90 degrees means that the ball or sprite is aimed toward the top of the canvas. A heading of 0 means that the ball is heading toward the right edge.  Notice also that you can change the heading by the simple expression:  (360 - heading):   |  |  | | --- | --- | | Current Heading | New Heading | | 90 | 360-90 = 270 | | 240 | 360-240 = 120 |   We’ll use this expression to make the ball *bounce* or *ricochet* in the opposite direction when it hits the paddle. | The *heading* property is based on the coordinate plane system: |

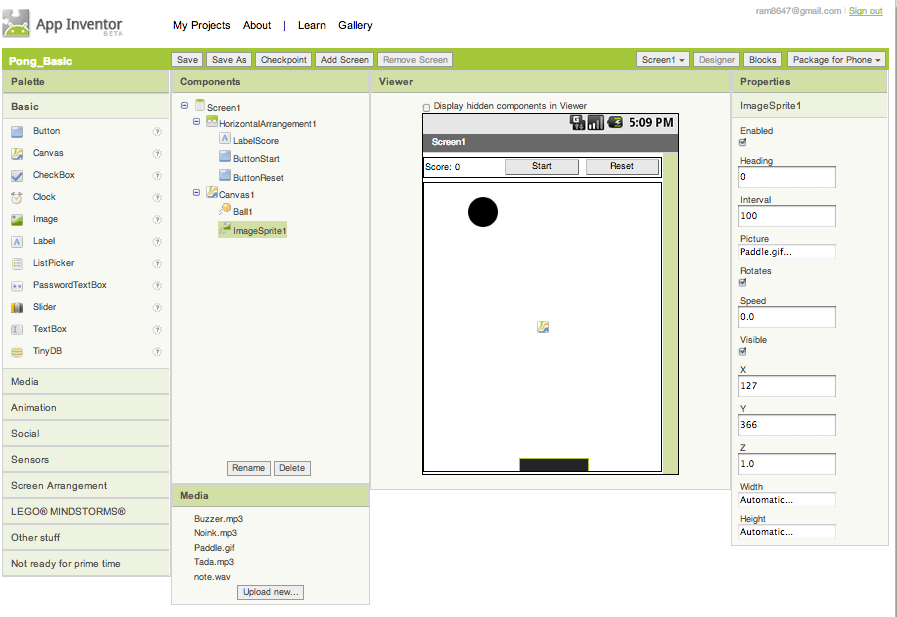
### 

### 

### 

### Description of the User Interface

Here’s a screenshot of the viewer with the ImageSprite selected:



Here are some important points to notice:

* The buttons and label are contained within the *HorizontalArrangement*  component and that *Ball1* and *ImageSprite1* are contained within the *Canvas.*
* We’ve used ***descriptive names***for the buttons and labels to make it easier to distinguish them in the blocks editor.
* We’ve used the default names, *Ball1* and *ImageSprite1*. This is okay because the app will only contain one instance of each.
* Note the many different properties of the ImageSprite. You’ll find that the Ball has the same properties except for the ***Picture*** property. The value of this property is the name of an image file (“Paddle.gif”), which you can find listed among the project’s media files.
* A main part of our programming task for this app will be controlling the values of these properties -- i.e., the Ball’s heading and speed and whether it is enabled (moving) or not.

### Ball and Sprite Events

The Ball and ImageSprite have exactly the same event blocks, procedures, setters, and getters. As we noted, the only difference is that the *ImageSprite* has a *Picture* property.

For the Pong game, there are three events that we must handle:

|  |  |
| --- | --- |
| **When the Ball reaches an edge.** This event has one property, *edge,* whose value is the edge that was reached. If edge = -1, that means the ball reached the bottom edge. | edgereached.png |
| **When the Ball collides with the paddle (or another Ball).**  This event has one property, *other,* whose value is the ball or sprite (paddle) that was collided with. For this app, we don’t care about this value of this property. It will always be the Paddle. | collidedwith.png |
| **When the ImageSprite is dragged.** This event has six properties. The most important for our app are *currentX* and *currentY.* These represent the paddle’s location when the user stops dragging it. | dragged.png |

### Ball and Sprite Procedures

There are two ball and sprite procedures that we will use in this app:

|  |  |
| --- | --- |
| **Bounce the Ball.** This procedure has one input slot, *edge,* which is the edge that you want the ball to bounce off of. Given the ball’s current *heading*, it will bounce automatically at a fixed angle given by (360 - heading). |  |
| **Move the Ball/Sprite to (x,y).** Balls and sprites can be moved to any point on the canvas. The *x* and *y* input slots represent the point’s (x,y) coordinates. |  |

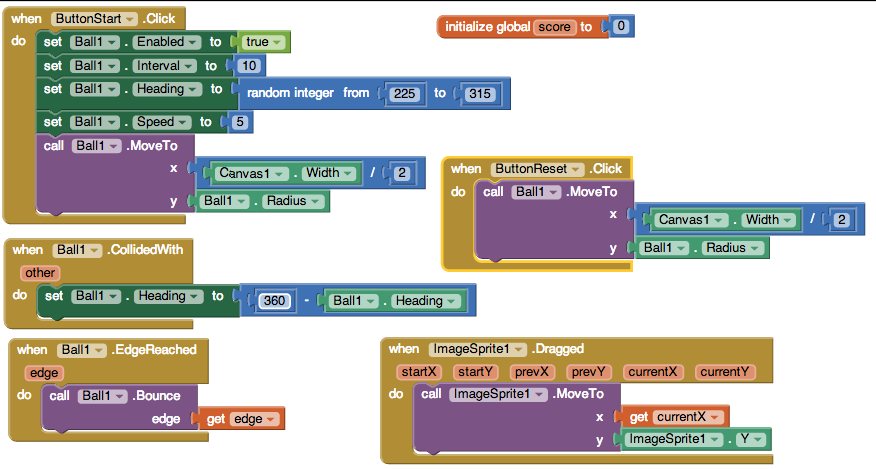
## Play with the App

Before proceeding you should connect the app to your device and play the pong game. You can also use the emulator, but it’s more fun on the device. Some things to notice:

* The user can only drag the paddle horizontally across the bottom of the canvas.
* The reset button puts the ball at the top-middle of the canvas and resets the score.
* The start button places the ball in motion at some random downward motion.

## Pong Code

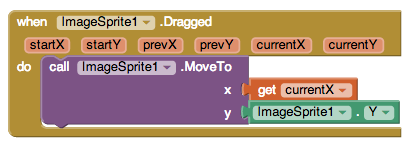
Here are the blocks used in this simple version of the Pong game:



## NOTE: THESE BLOCKS HAVE ALREADY BEEN CODED FOR YOU AND ARE EXPLAINED BELOW.

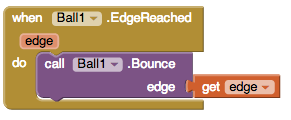
### Controlling the Paddle

To control the user’s dragging of the paddle we use the *when ImageSprite1.Dragged* block. Because the paddle can only move horizontally, not vertically, its *Y*-coordinate is always going to be the same -- i.e., *ImageSprite1.Y.* However, its X-coordinate will be changed to whatever value it has for its *currentX* location. Thus, we use a *getter* block to get the paddle’s currentX coordinate, which is one of the *Dragged* block’s inputs.



### Bouncing off the Edge

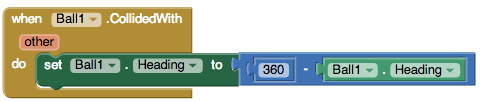
In the current version of the game whenever the ball reaches an edge -- including the bottom edge -- it simply bounces off that same edge. To do this we call the *Ball1.Bounce* procedure.



Notice how we use a *getter block* to give the procedure the value of the *EdgeReached* edge input. (Don’t be confused by the various uses of ‘edge’ in this example. The one on the yellow *EdgeReached* block is the property of the edge reached event. Its value is provided by App Inventor. It represents the edge that was reached. The one on the purple *Bounce* block is an *input parameter.* We need to input a value in that slot that represents the edge that we want to bounce off of. The *getter block* gets the value of the event property and puts it into the Bounce slot*.* Of course, its value is the edge that was reached in the event.)

### Colliding with the Paddle

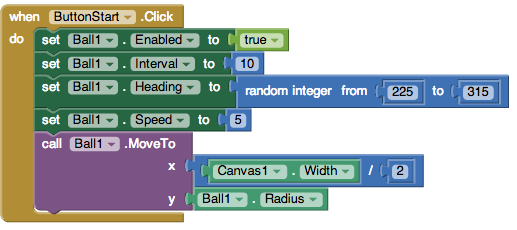
Finally, we have to manage the collisions between the ball and the paddle. All that happens here is the ball bounces off the paddle by heading in the reverse direction. Here is where we use the (360 - heading) expression to reverse the ball’s direction:



### 

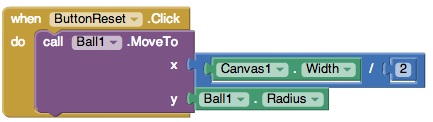
### Starting and Resetting the Game

To start the game, the Ball’s speed,heading, position, and interval properties must be set and it must be *enabled* in order to start moving:



Note how the *Heading* property is set to a random value between 225 and 315. This will make the ball move downward -- recall that a heading of 90 is straight up and 0 is toward the right edge. The *Speed* and *Interval* properties control how fast the ball moves. In this case it moves 5 pixels every 10 milliseconds. Finally notice how we position the Ball in the top-center of the canvas.

The reset button simply repositions the Ball at the top-center of the canvas:



# 

# Enhancements

### 1. Keeping Score: Add a procedure and other statements to the code to perform score keeping tasks.

HINTS: Here is a one minute video lesson on [how to define a procedure](https://www.youtube.com/watch?v=xWp0JxmD314) and here’s one on [how to define a procedure with a parameter](https://www.youtube.com/watch?v=NCh6BuiQrhE&feature=youtu.be). Also, The [video tutorial (part 5)](https://www.youtube.com/watch?time_continue=292&v=4IU9qdtwMpQ) shows how to write a procedure to keep and report the score. You could follow that approach for this enhancement.

Keeping score involves the following tasks:

1. Every time the user successfully hits the ball with the paddle (preventing it from hitting the bottom wall), the score should be increased by 1 point.
2. Whenever the score changes, the new score should be displayed in the score label.
3. **Optional**. At the end of the game call your procedure to display “Game Over” plus the final score. (HINT: You should use a parameter to distinguish this case from the case (#1) where the score is just incremented. The parameter could be a text with values such as “Hit” and “Game Over” or a number, such as 1 for “Hit” and 2 for “over”. NOTE: If you follow the video tutorial, part 5, to keep score, you might have to use a second parameter to keep track of when the game is over.)

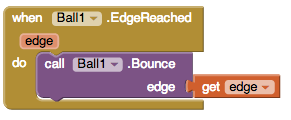
Note the use of the *join block* (from the *Text* drawer) to join together the text “Score: “ and the score value*.* This will produce “Score: 0” as the value for *LabelScore’s Text* and eliminates the need for two separate labels.



**Before moving on, test your enhancement. Does it work as intended?**

### 2. Ending the Game: Modify the *when Ball1.EdgeReached* block to end the game when the ball reaches the bottom edge.

The current game never ends. No matter what edge the Ball touches, it just bounces off:



To fix this block, when an edge is reached we need to check whether it is the bottom edge or not. If it is the bottom edge (edge = -1), then we should end the game. If it is not the bottom edge, then the ball should bounce. In other words, we’ll need a block here that implements the following *if/else algorithm:*

if (the bottom edge is reached):

do:

stop the game

else:

bounce off the edge

And if/else statement is an example of a *control statement* -- i.e., a statement that helps control the app’s behavior. The *if-clause* contains a condition that is either True or False. Thus, the if/else block is a *selection* control statement tells the app to take one or other of two paths in the code -- i.e., stop the game or bounce -- depending on the truth (or falsity) of the if-clause:

Bounce

Game Over!

Bottom edge?

do

else

The App Inventor if/else block can be found in the *Control* drawer:



This version of the block only has a *do-slot,* which means it doesn’t do anything when the if-clause is false (except continue on to the next statement):

Notice that the if/block has a *mutator* (the blue symbol mutator.png).

Something

True?

do

Pressing the mutator will bring up a mini-editor that allows you to add clauses to the if-block.

Drag the else block into the in block (left) to create an if/else block.

As shown in this diagram, to add an *else-clause* to the block, click on the mutator and then drag the else-block on the left-hand-side of the mini-editor into the *if-block* on the right-hand-side of the editor. Then click on the mutator again.

In order to end the game, there are two tasks that you should perform:

* Disable the Ball to stop it from moving.
* Set LabelScore’s Text to “Game Over!”

Here’s the algorithm you should implement in the *when Ball1.EdgedReached(edge)* block:

if (edge = -1):

do:

set Ball1.enabled to false

set LabelScore.Text to “Game Over!”

else:

call Ball1.Bounce(edge)

**Before moving on, test your enhancement. Does it work as intended?**

### 3. Add Sound Effects.

The template for this app includes a collection of sound files that can be used as sound effects:

|  |  |
| --- | --- |
| **Action** | **Sound Effect** |
| Ball hits bottom wall (game over). | Buzzer.mp3 |
| Ball hits an edge. | note.wav |
| Ball hits paddle. | Noink.mp3 |

To use these sound files you will need to add a *Sound component* to your app (in the Designer’s *Media* drawer). Then to play the sound, you would use the following two blocks:



## Before moving on, test your enhancement. Does it work as intended?

### 

### 

### 4. ADVANCED: Allow the user turn the sound on/off.

Add a feature to the app that allows the user to turn the sound effects on/off. You may use any if or if/else blocks and procedures that you need to solve this problem.

We want to give the user the ability to turn the sound effects on and off -- so they can play the game in the library or on a crowded bus. There are lots of ways to do this, but we suggest using a new component, a *CheckBox.*

### 

HINT: The Checkbox component has a *Checkbox.Checked* block:



This block will be ***true*** if the CheckBox1 is checked and ***false*** if it is not.

Depending on your approach to solving this problem, you may need to use another block: the *Checkbox.Changed* block:



This block controls what happens when the CheckBox is checked or unchecked.

**Fun fact:** This problem *can* be solved usingonly **one** additional if or if/else block in the app**.** Can you figure out how to do this?

## Test your enhancement. Does it work as intended?

## Acknowledgements

This tutorial borrows ideas and materials from the [MIT Pong Tutorial](http://appinventor.mit.edu/explore/resources/visual-pong-tutorial.html).

***Nice work! Complete the Self-Check Exercises and Portfolio Reflection Questions as directed by your instructor.***