

Creating a Mancala Game using App Inventor



Truston Ailende

Table of Contents

- [Introduction](#)
- [Mancala](#)
- [Ise](#)
- [App Inventor](#)
- [Abstraction](#)
- [Algorithms](#)
- [Creating Ise Using App Inventor](#)
- [Getting Started](#)
- [Modelling and Simulation of Seed Distribution](#)
- [Game Algorithm](#)
- [Completing our Game](#)
- [Conclusion](#)
- [Android Development for Everyday People](#)
- [About the Author](#)

Introduction



This guide serves as an introduction to [App Inventor](#) using the creation of a [Mancala](#) game as a point of entry. I was inspired to create this guide after finishing the writing of the draft of the book [Android Development for Everyday People](#). In that book, I created Ise which is a variant of a [Mancala](#) game played by my people the [Ishan people of Edo State in Nigeria](#).

Today the series has been developed into a book that is now available on the [Amazon](#) store. It is my first book as an author and I am extremely proud of it. If you are interested in learning more about [App Inventor](#), do give it a [look](#).

Mancala

Mancala is a family of board games played around the world, sometimes called "sowing" games, or "count-and-capture" games, which describes the gameplay.

Mancala games are especially popular in Africa and are played in one variation or the other. Equipment is typically a board, constructed of various materials, with a series of holes arranged in rows, usually two or four. The materials include clay and other shape-able materials.

Some games are more often played with holes dug in the earth, or carved in stone. The holes may be referred to as "depressions", "pits", or "houses". Sometimes, large holes on the ends of the board, called stores, are used for holding the pieces.

Playing pieces are seeds, beans, stones, cowry shells, half-marbles or other small undifferentiated counters that are placed in and transferred about the holes during play.



Board configurations vary among different games but also within variations of a given game; for example Endodoi is played on boards from 2×6 to 2×10 . The largest are Tchouba (Mozambique) with a board of 160 (4×40) holes requiring 320 seeds; and En Gehé (Tanzania), played on longer rows with up to 50 pits (a total of $2 \times 50 = 100$) and using 400 seeds. The most minimalistic variants are Nano-Wari and Micro-Wari, created by the Bulgarian ethnologue Assia Popova. The Nano-Wari board has eight seeds in just two pits; Micro-Wari has a total of four seeds in four pits.

With a two-rank board, players usually are considered to control their respective sides of the board, although moves often are made into the opponent's side. With a four-rank board, players control an inner row and an outer row, and a player's seeds will remain in these closest two rows unless the opponent captured them.

The objective of most two- and three-row mancala games is to capture more stones than the opponent; in four-row games, one usually seeks to leave the opponent with no legal move or sometimes to capture all counters in their front row.

At the beginning of a player's turn, they select a hole with seeds that will be sown around the board. This selection is often limited to holes on the current player's side of the board, as well as holes with a certain minimum number of seeds.

In a process known as sowing, all the seeds from a hole are dropped one-by-one into subsequent holes in a motion wrapping around the board. Sowing is an apt name for this activity, since not only are many games traditionally played with seeds, but placing seeds one at a time in different holes reflects the physical act of sowing. If the sowing action stops after dropping the last seed, the game is considered a single lap game.

Multiple laps or relay sowing is a frequent feature of mancala games, although not universal. When relay sowing, if the last seed during sowing lands in an occupied hole, all the contents of that hole, including the last sown seed, are immediately re-sown from the hole. The process usually will continue until sowing ends in an empty hole. Another common way to receive "multiple laps" is when the final seed sown lands in your designated hole.

Depending on the last hole sown in a lap, a player may capture stones from the board. The exact requirements for capture, as well as what is done with captured stones, vary considerably among games. Typically, a capture requires sowing to end in a hole with a certain number of stones, ending across the board from stones in specific configurations, or landing in an empty hole adjacent to an opponent's hole that contains one or more pieces.

Another common way of capturing is to capture the stones that reach a certain number of seeds at any moment.

Also, several games include the notion of capturing holes, and thus all seeds sown on a captured hole belong at the end of the game to the player who captured it.

Ise

Ise is a variant of the Mancala game played by the people of [Edo State in Nigeria](#). It is played on a 2×6 board consisting of 4 seeds in each pot.

The game starts with the players agreeing on who gets to play first. Then, a pot is selected by the player and sowing occurs in a counter-clockwise direction. During that first sowing, the state of the board is determined. Once the first sowing takes place, no pot on the board has 4 seeds inside them.

Seed capture occurs when the number of seeds in a pot is 4. In the process of gameplay, the number of seeds in a pot can reach up to 4. Once this happens, the seeds are captured. Capture can be direct or incidental.

With direct capture, the sowing ends in a pot that already has 3 seeds. When this occurs, the seed that would be added makes the number of seeds in that pot equal to 4. As a result of this, the player who did the sowing will get the seeds that are captured assigned to the player.

With incidental capture, the sowing during the course of the game increases the number of seeds in a pot to 4. The seeds that are captured are then assigned to the owner of the pot.

The grand finale occurs when only 8 seeds are left on the entire board. At this point the first person to capture wins the remaining 4 seeds.

The winner is decided by the player with the highest number of seeds.

App Inventor



[App Inventor](#) is a visual, easy to use online Android Application development platform. [App Inventor for Android](#) was originally provided by Google, and is now maintained by the Massachusetts Institute of Technology (MIT).

It allows newcomers to computer programming to create software applications for the Android operating system. It uses a graphical interface which allows users to drag and drop visual objects to create an application that can run on Android devices.

With [App Inventor](#), if you can imagine it, you can create it. Using this free, friendly tool, you can decide what you want your app to do and then click together colourful jigsaw-puzzle blocks to make it happen. App Inventor turns your project into an Android app that you can test on your computer, run on your phone, share with your friends, and even sell in the Google Play store.

When learning how to program, learning the logic of programming and the syntax of a text based language poses a barrier for a new programmer. App Inventor removes the need of a beginner to know the syntax of a programming language and focus on the logic of what they want to create. With App Inventor, the only limit is your imagination.

Abstraction

A critical concept in programming is abstraction. Abstraction is a reduction in the level of detail to allow a grasp of the relevant information. An example of this is a map. A map is a representation of an actual physical space but in order to really represent it, all we do is take the portions we are interested in and ignore the rest.

Below is a map of my location. The level of detail this map shows is different from the view I have from my location.



We abstract because it makes it easier for us to model a problem. By focussing on the details we need to model a problem, we can approach the solution to the problem quicker.

[App Inventor](#) allows us to use an abstraction of programming constructs to create and develop mobile applications. This shortcut makes it easier to create a mobile application.

Don't miss the point to all of this. You are still programming. However, you are not coding. Programming is about getting the computer to do what you want it to do. This means you have to give the computer instructions.

With coding you would have to figure out your instructions as well as create the constructs for them. By using already available constructs in [App Inventor](#), you make the process of creating mobile applications easier.

It is important to note that the computer is a machine. The acronym GIGO (Garbage In, Garbage Out) covers this. For the computer to accurately execute your instructions, you need to give it unambiguous instructions.

A set of unambiguous instructions to carry out a task is called an algorithm. Note the emphasis on

unambiguous because the instructions must be specific and exact. As human beings we tend to make a lot of assumptions whenever we issue instructions. Part of the difficulty in programming is that we have to explicitly describe what we want to do to the computer. The sequence of steps that accomplish this exactly are called an algorithm.

In truth creating an algorithm is the hardest part of programming. Once you have created your algorithm, writing the code to do execute it is much easier. Thankfully with [App Inventor](#), there is no need to write code. Just assemble your constructs.

Algorithms

An algorithm is a set of specific steps that you can follow to solve a problem. I specifically mention steps because in truth algorithms apply to everyday life.

Below is an algorithm to find the average of 3 numbers.

1. Set a running total to 0.
2. Add the first number to the running total.
3. Add the second number to the running total.
4. Add the third number to the running total.
5. Divide the running total by 3.
6. The result is the average of the 3 numbers.

If you followed this algorithm without deviation, you would always get the average of 3 numbers. You don't need a computer to execute the above algorithm; you can do it by hand.

The key to application development is being able to create the algorithm for the problem that you want to solve. Once you can figure out how to solve a problem and write it in a sequence of unambiguous steps, the process of automating the solution to the problem is easy.

In essence that is what application development is really all about: the automation of the solution to a problem.

Creating Ise Using App Inventor

Having covered the concepts of Abstraction and Algorithms, we are ready to proceed on our journey of creating Ise.

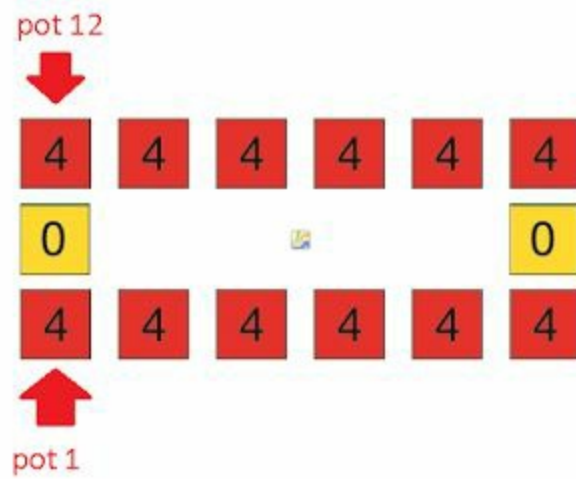
The first step in this journey is to understand the logic of what we want to create. In our case, we are fortunate that we are trying to make a physical artefact become digital.

Our Ise board is made up of 12 pots and 2 holes. The number of seeds in each of the pots is 4. The easiest way to model this is to use a table as shown below:

1	2	3	4	5	6	7	8	9	10	11	12
4	4	4	4	4	4	4	4	4	4	4	4

The above table shows that the number of seeds in the pots labelled 1 to 12 is 4. The labelling is done from the pot at the lower left hand corner of the screen.

The diagram for this is shown below:



Getting Started

The first thing we need to do in our program is create the user interface for our program. The user interface is the part of our program that the user will interact with.

Create a new project in App Inventor and give it an appropriate name. In my case, I decided to call my project IseReview. Change your ScreenOrientation to LandScape.

The key to creating the user interface as we have above is to use the components found in the Drawing and Animation section of the Palette.



Drag the Canvas component on the Screen1 component and set its Width and Height properties to Fill parent.

Drag out 14 ImageSprite components onto the Canvas. Now rename them and set their X and Y coordinates using the table below:

Name	X Coordinate	Y Coordinate
pot1	20	150
pot2	100	150
pot3	180	150
pot4	260	150
pot5	340	150
pot6	420	150
pot7	420	10
pot8	340	10
pot9	260	10
pot10	180	10
pot11	100	10
pot12	20	10
playerOneScore	420	80
playerTwoScore	20	80

The images we intend to use are 58 X 58 pixels. The naming of the images for display on our board should be that the image value for an image that would display zero would be named 0.jpg and the image to display 1 should be named 1.jpg. Naming our images this way will allow us display the

images that we would need as a distribution takes place.

To get the board shown in the pot labelling shown above, we name our images 0.jpg and 4.jpg then we upload them to App Inventor. Rename your image sprites and load the images for the appropriate image sprites.

Since in the creation of the game the images we need will need to be in the App Inventor project, it is a good time to upload our App Inventor images to display the numbers 1 to 48. To make it easy on you, I have uploaded the images that I used to create the prototype for Ise. You can find them [here](#).

Once we can display the image for our board, we can then begin writing the code for our application.

Now that we have finished with the appearance of the board, we have to create its internal representation. For our purposes we will be using the Lists component which is a built-in component in App Inventor.

Think of it this way. A car is made up of an engine and a chassis. The engine powers the car but you don't see it. However, the chassis is the part of the car that people see but it does nothing.

This separation of appearance and functionality is how a car works. We see this in App Inventor. The Designer allows us to create the user interface for our application while the Blocks Editor is where we assemble the components for our code.

The Lists component is similar to a block of flats in a building. Once you get into the apartment, you can access the whole building. This is important in the thought process for this game because it controls how seed distribution will occur.

When I first created this game in 2010, I also created my own data structure. It was a circular linked list. Now experience makes me understand that any list data structure will do. All that is needed is a way to reset the seed distribution to the beginning of the list when the seed distribution gets to the end of the list.

The first thing we need to do is go to our Blocks editor and create a list called pots. For our Ise we have 12 pots so we need to add 12 elements to our list. Assign the value of 4 to each element of our list. You should have the code shown below:



We need 4 variables before we can start working on our program. They are:

1. currentPot which is used to track the pot initially selected by the user
2. nextPot which is used in the timer to get the nextPot from the current pot. This variable allows the program to cycle through the game board
3. seedsHeld which is used to get the number of seeds held by the pot touched by the user
4. potSeeds which is used to get the number of seeds in the next pot in the seed distribution used by the addSeeds procedure

The code that does this is shown below:

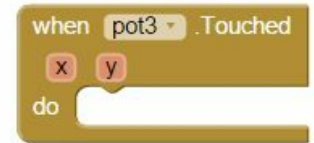
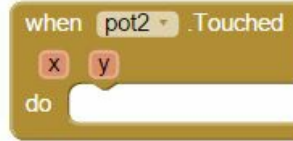
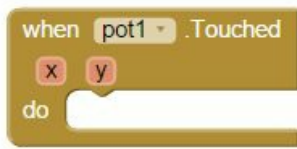


We need a way to identify the pot selected by the user. To do this we will use a procedure that will get the potID of the pot that is clicked by the user. This procedure will be called setPotID and its function will be to assign a value to the currentPot and seedsHeld variables respectively.

Our setPotID procedure has a parameter of potID and it assigns potID to our currentPot variable. The seeds held by the pot are calculated using the seedsHeld function which also takes in potID as an

input parameter.

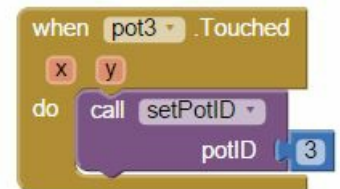
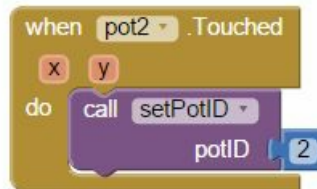
We go to the Blocks section of the Blocks editor and click on pot1. Select the when pot1.Touched event handler and drag it out. Copy and paste them for pots 1 o 12. Our code for the first 3 pots will look like shown below:



Now we need to create the stub for the setPotID procedure with the potID parameter. The code that does this is shown below:



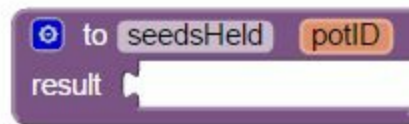
This stub allows us to use the code in the when pot.Touched events for all the pots we have. The code for our first 3 pots is shown below:



Do this for all 12 pots. Now we have a means of identifying the pot that is selected by the user.

Whenever we select a pot, we will be automatically potID to the setPotID procedure. This procedure at the moment is a stub. In order to fill up its components, we need to understand how it will work.

First we will set the currentPot to the potID parameter passed in by the user. Next we will get the seedsHeld by the pot that was selected by the user. If the seedsHeld by the selected pot is 0, then we need to notify the user. The stub for the seedsHeld function is shown below:



The difference between a procedure and a function is that a procedure carries out an action while a function carries out an action and returns a value. This means that whenever we want to compute a value, we need a function.

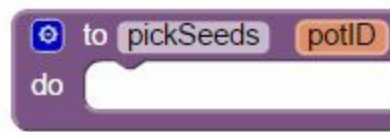
So far we have made no mention of how we would notify the user that they have selected an empty pot. The way to do this is to use a Notifier component which is available from the User Interface drawer of the Palette.

Select it and drag it to your project. You will see it appear under the Non-visible component area in your Viewer area. This simply means that the user will not be able to see this component when the

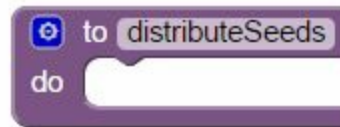
app launches but will appear when we need to use it.

Once we confirm that the seedsHeld by the selected pot are more than 0, we need to pick them from the pot and distribute them around the board.

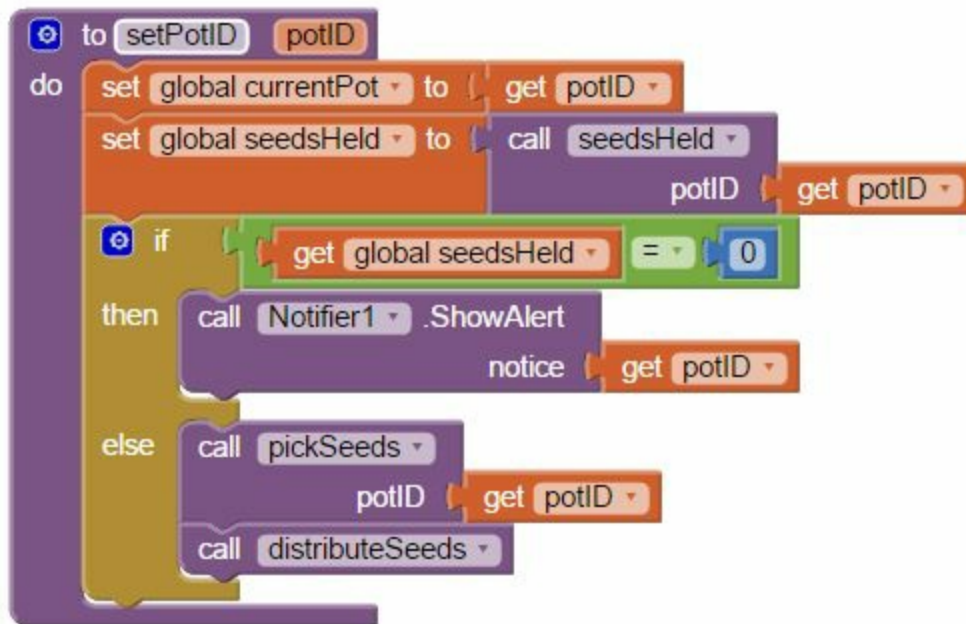
The stub for the pickSeeds function is shown below:



The stub for the distributeSeeds function is shown below:



With the stubs for our requisite functions created, we can now create our setPotID procedure. The code that achieves this is shown below:



Now we can set about filling our functions and procedures.

Our seedsHeld function works by returning the value of the element at the index. In our program we map the potID to the pot number of the selected pot which is also the index of the element of the list at that position. So all we have to do is return the element at the index of the list. The code to do this is shown below:



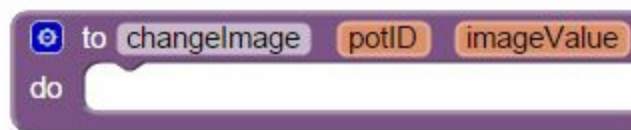
The pickSeeds procedure is used to pick the seeds from the list and also change the visual representation of the list to the user. The first part of this operation means that we have to replace the

value in at the pot selected by the user with 0. Doing this only changes the value of the element in the list externally nothing changes.

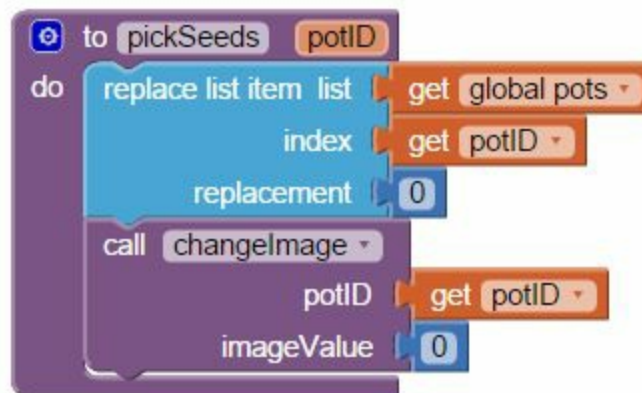
For the value to change externally, we need to use a new procedure called `changeImage` which will take in `potID` for the pot selected by the user and value of the image to be displayed to the user.

If you will recall earlier you uploaded images from 0 to 48 to your App Inventor environment and I recommended that you name the images based on the numbers that the displayed. If you didn't do that, please go back and do that step. It is because of the `changeImage` procedure that we see the images flip when the user touches them and when the seed distribution is ongoing.

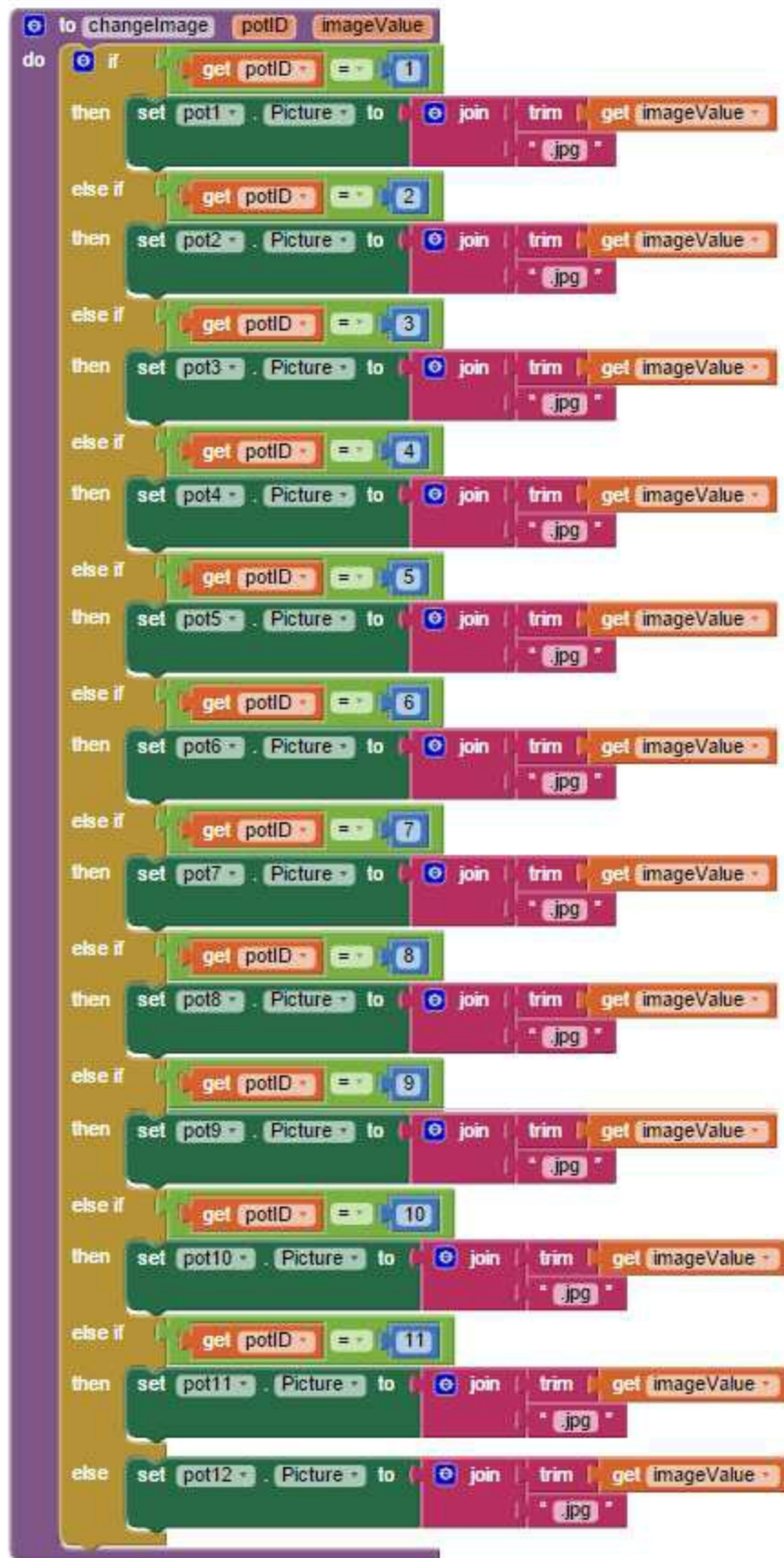
The stub for the `changeImage` procedure is shown below:



The code for the `pickSeeds` procedure can now be written and it is shown below:

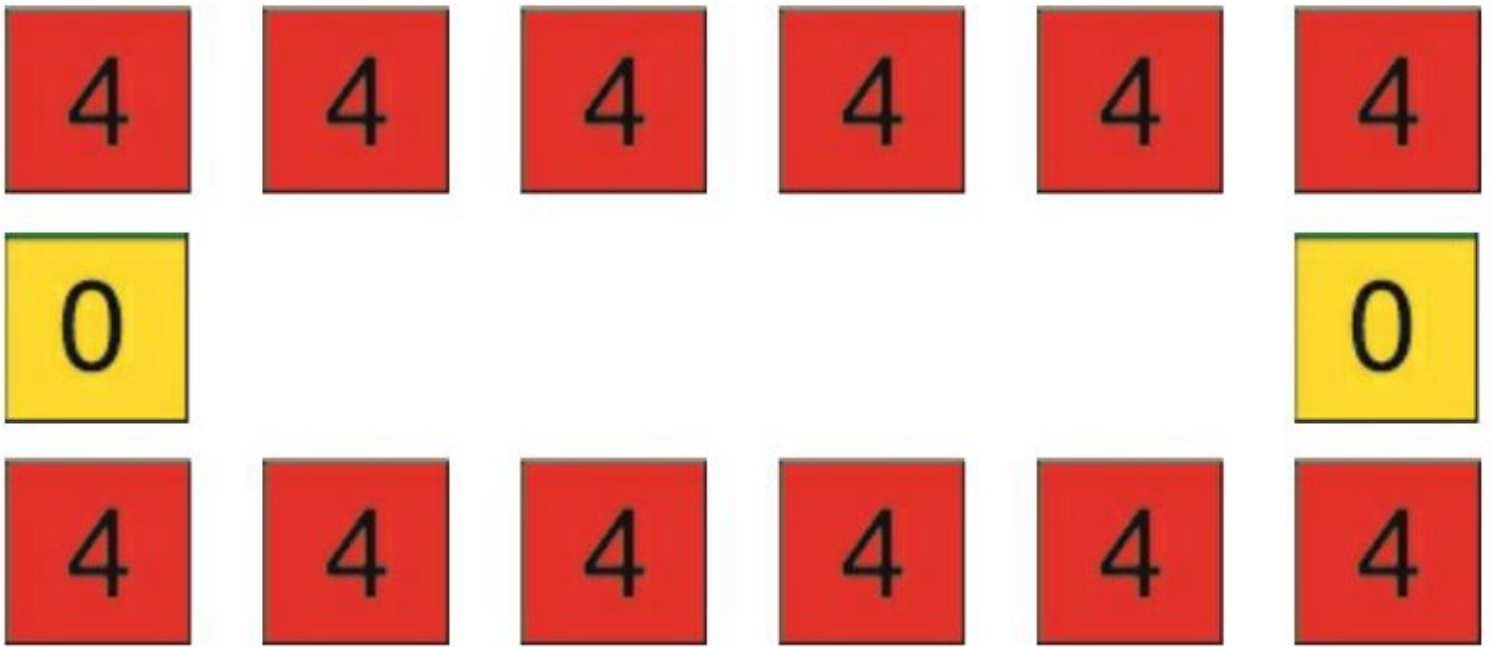


The `changeImage` procedure is used by the `pickSeeds` procedure to change the image for a pot. It works by matching the `imageValue` to the picture in our media library. We use the `trim` function to prevent any surprises. The `changeImage` procedure is shown below:



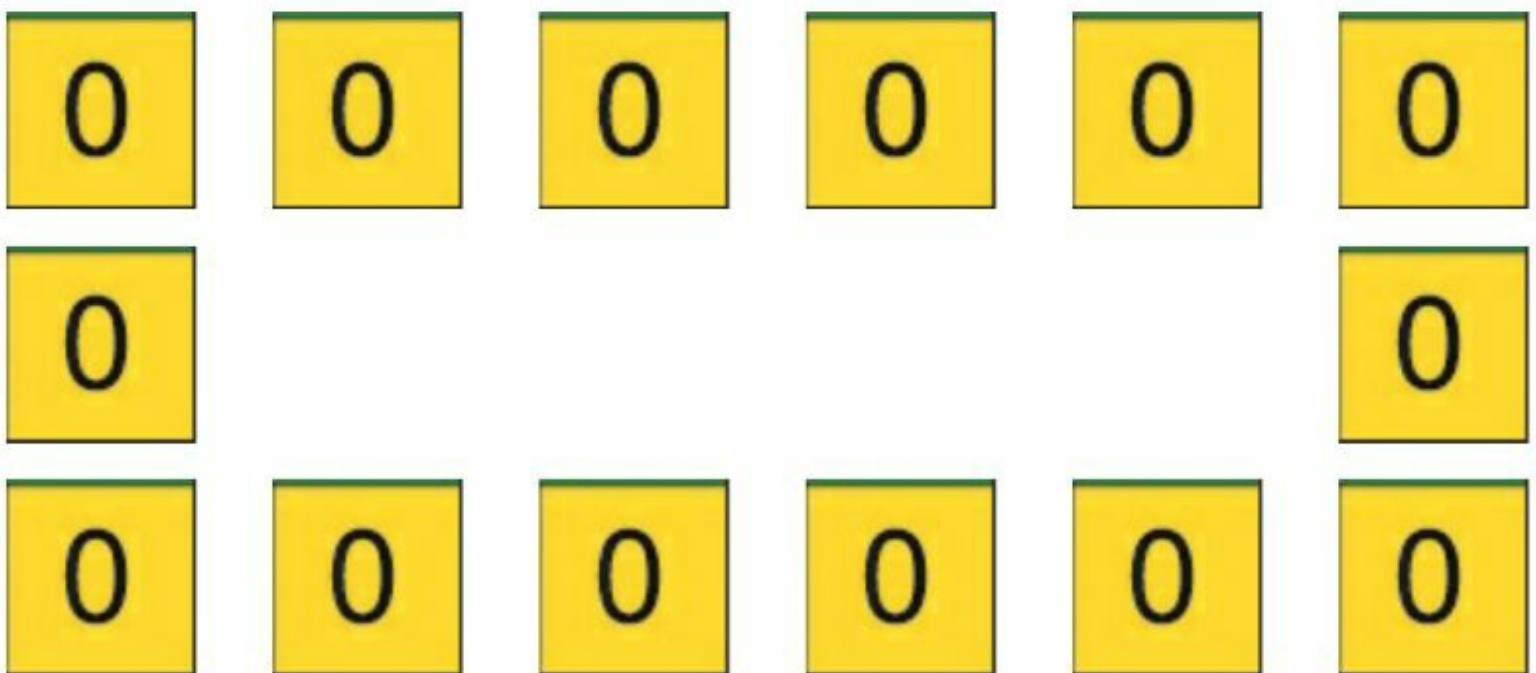
At this point, we can do a build of our project to test if the flipping works. This is a good place to check if the tapping of the pot makes the seed in the tapped pot become zero.

When we launch our app, you should see the display shown below:



Now I will click on pot 1. Doing this the first time should change the image displayed to 0. If I did this the second time, it would display the pot id of the selected pot because we have programmed it to do that when the seedsHeld in a pot is 0. Tapping all the pots in the game board will switch them to 0.

This is shown below:



At this point if you tap any pot, its pot number will be displayed. This shows that all the code we have written thus far works. Try it out.

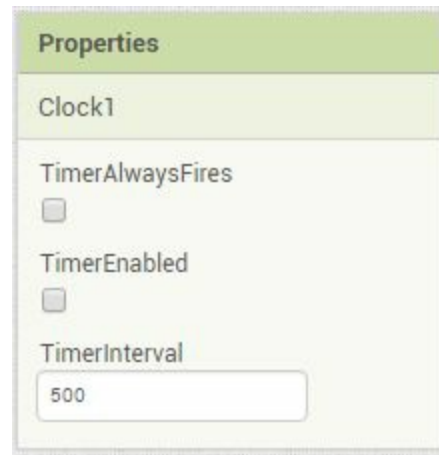
All the code we have written thus far works but we haven't made the seed distribution work. We shall approach this next.

The key to making the seed distribution work is to use a loop. However, if we used a simple while loop our animation would run too quickly for the eye to see it so what do we do?

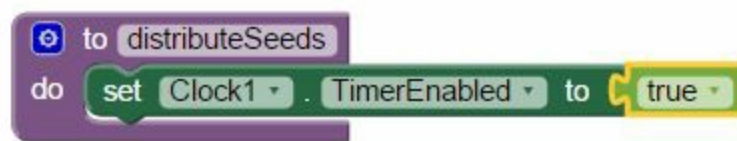
To slow down our animation for it to be seen by the human eye, we will use a Clock. So drag out a Clock component from your Sensors drawer under your Palette.

Once you drag out the Clock component go to the Properties area and disable the TimerAlwaysFires and TimerEnabled properties. Set the TimerInterval to 500. The TimerAlwaysFires property ensures that your timer will continue to run even after you exit application.

You can think of your TimerEnabled property as a switch. The TimerInterval determines the interval at which the timer will execute each instruction. Your properties area for your clock should now look like this:



The task for our distributeSeeds procedure is to switch on the Clock1 component. The code to do this is shown below:



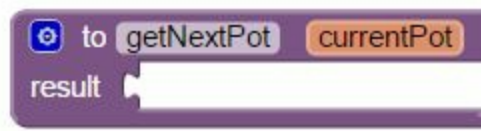
The above code means that your Clock1 component will fire every half of a second once it is turned on. The time interval that we have set will determine how fast our seed distribution will be.

Once our Clock1 component is activated, it will fire every half a second and continue the seed distribution until the number of seedsHeld is 0. In this way our clock acts like a loop but it is a timed loop.

Once the number of seeds held is 0, the Clock will be disabled and the user will be notified that the distribution of seeds has ended.

We will need one new function and procedure our function will be the getNextPot function which gets the next pot that seeds should be distributed to while our procedure is the addSeeds procedure which will adds seeds to the pots that seed distribution has reached. It will also change the image of the seeds in the pot.

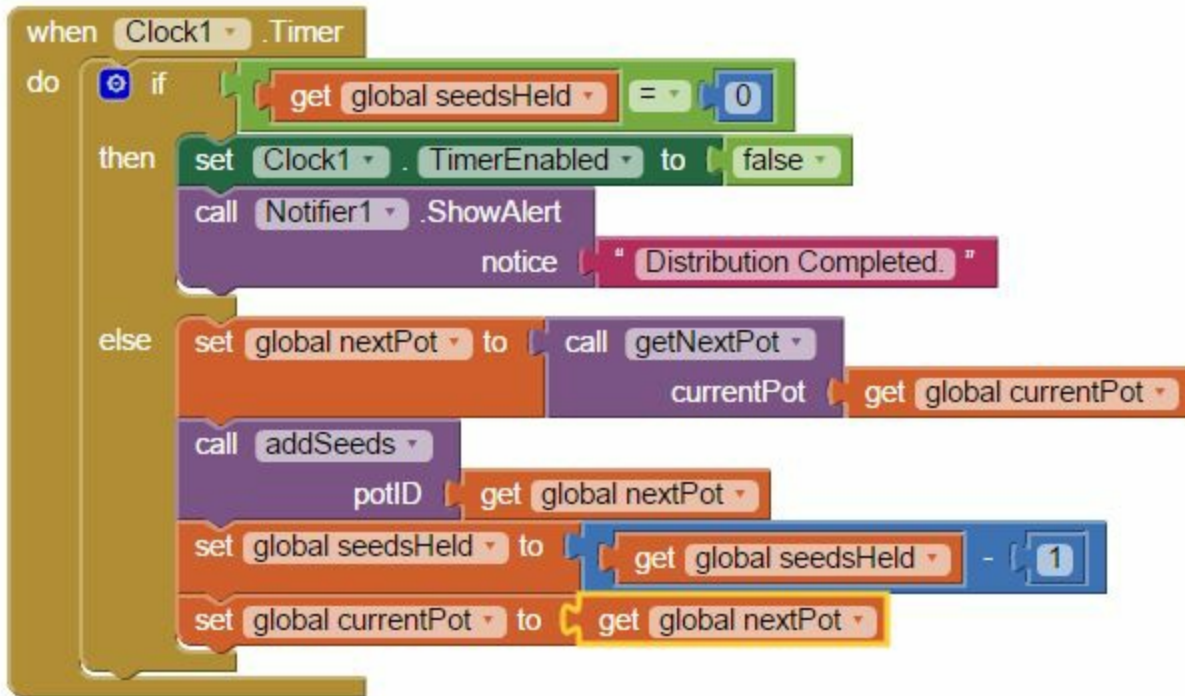
The stub for the getNextPot function is shown below:



The stub for the addSeeds procedure is shown below:



The code for our Clock component is shown below:

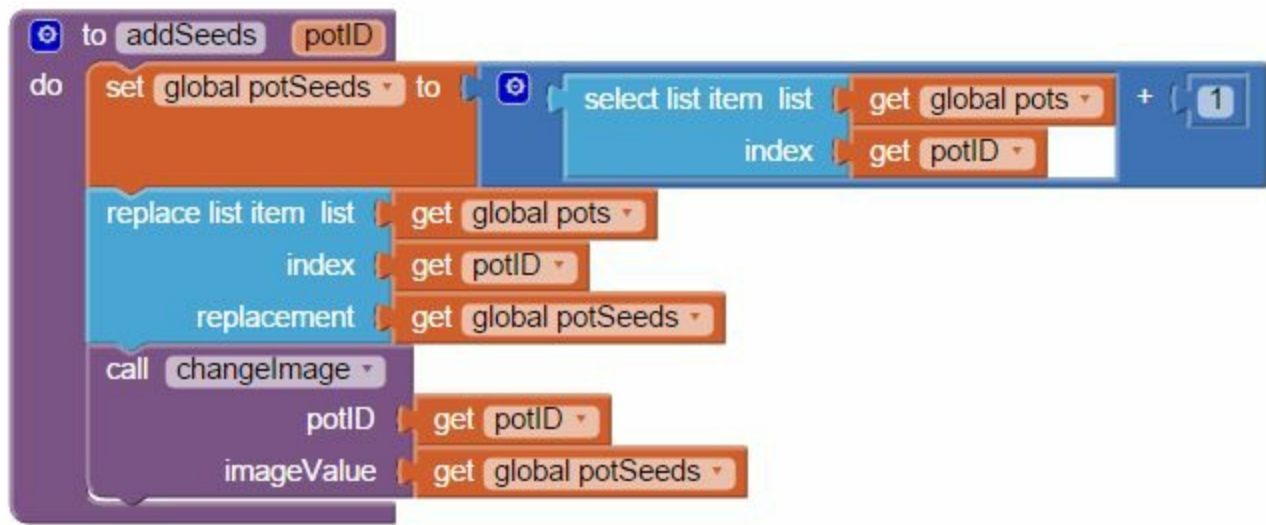


As long as the number of seeds held is greater than 0, we will have to get the next pot for us to distribute our seed to. This is important because the limit of our board is 12 and we must ensure that the distribution loops back to 1 once we get to 12. For our next pot, we add a seed to it and then we subtract 1 from the number of seeds held. Finally, we set our nextPot to our currentPot and continue until our seeds held is 0.

Our getNextPot function uses the modulus operator. The modulus operator is an operator that returns the remainder of the division of two numbers. For example 10 modulus 3 returns 1. A modulus of 0 means that the first number is divisible by the second number. We use this in our program to make our pot distribution cycle back from 12 to 1. The code for this is shown below:

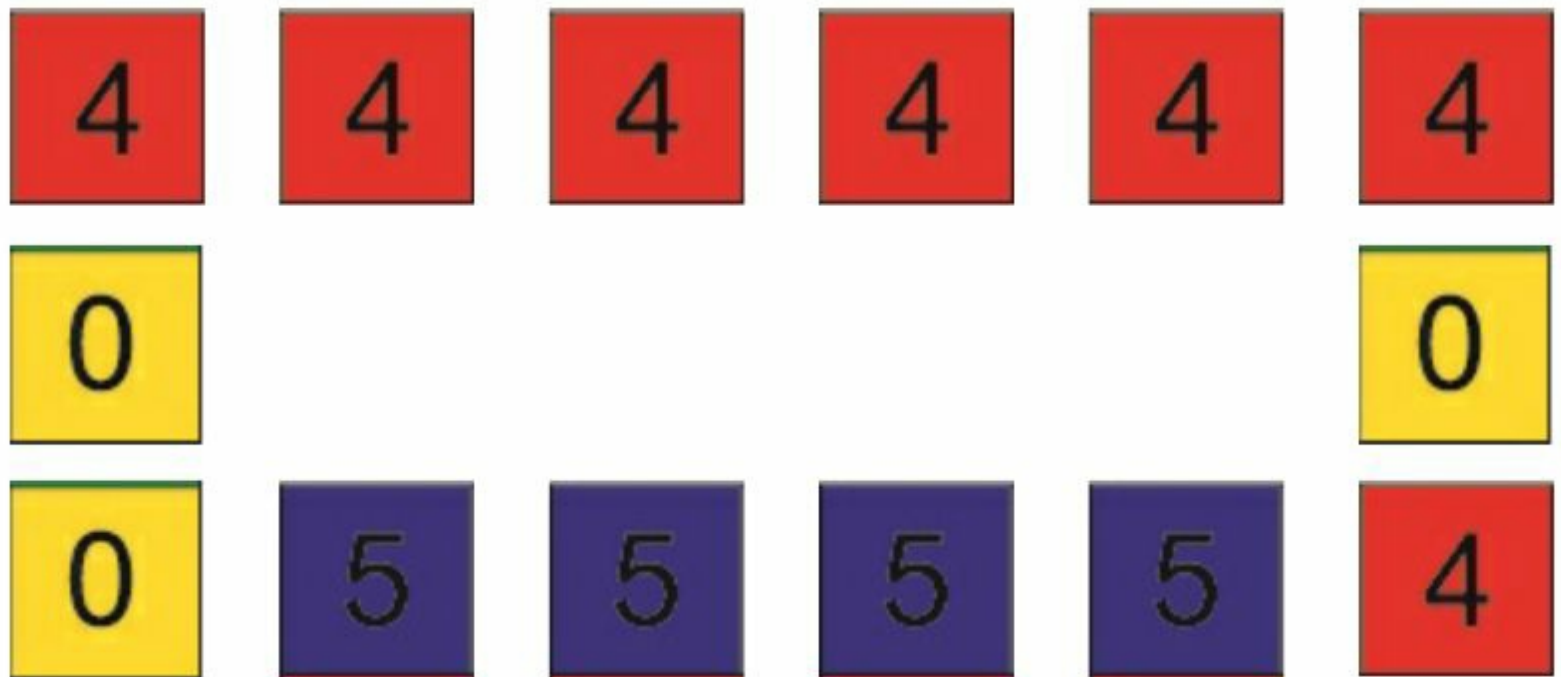


The addSeeds procedure gets number of seeds in a pot and adds 1 to it. It also changes the image for that pot. The code for it is shown below:



This is a great point to test the work we have done so far.

When we tap the first pot, the seed distribution occurs and the notification is shown to the user at the end of the seed distribution. This final state of the board is shown below:



Modelling and Simulation of Seed Distribution

This is a manual process to model seed distribution on the board. I had to do it manually first in order to understand it. I will use the table I introduced before for this purpose.

When the game starts, the user is shown the board the state of the board is shown below:

1	2	3	4	5	6	7	8	9	10	11	12
4	4	4	4	4	4	4	4	4	4	4	4

The moment the user picks the seeds from pot 1, the seeds in that pot change to 0 and the state of our board is as shown below:

1	2	3	4	5	6	7	8	9	10	11	12
0	4	4	4	4	4	4	4	4	4	4	4

The seeds from pot 1 are distributed from the 2 second pot up until the 5 pot where the user has run out of seeds. At this point the state of our board is shown below:

1	2	3	4	5	6	7	8	9	10	11	12
0	5	5	5	5	4	4	4	4	4	4	4

Now at pot 5, we need to pick the seeds again. The state of our board now becomes:

1	2	3	4	5	6	7	8	9	10	11	12
0	5	5	5	0	5	5	5	5	5	4	4

At pot 10, the seed distribution stops and we need to pick the seeds at pot 10 which are 5 in number and distribute them around the pot. The state of our board now becomes:

1	2	3	4	5	6	7	8	9	10	11	12
1	6	6	5	0	5	5	5	5	0	5	5

Now our seed distribution stops at pot 3. Pot 3 contains 6 seeds so we pick them and distribute them around the board. The state of our board now becomes:

1	2	3	4	5	6	7	8	9	10	11	12
1	6	0	6	1	6	6	6	6	0	5	5

At this point our seed distribution stops at pot 9. We pick the seeds from pot 9 and the state our board becomes:

1	2	3	4	5	6	7	8	9	10	11	12
2	7	1	6	1	6	6	6	0	1	6	6

Now seed distribution can stop as we have landed in a pot with no holes.

Do ensure you understand how to model and simulate seed distribution manually before moving on.

Game Algorithm

Having covered how to model and simulate the seed distribution, we can now look at the algorithm for this game.

The algorithm to create the entire game starts with the actions that must be carried out when a pot is clicked by the user and the steps are listed below:

1. Check if the pot is empty or has seed inside it
2. If the pot has seeds inside it, check the number of seeds it has
3. Pick the seeds from the pot the user selected if the pot isn't empty or display a message to the user to select another pot
4. Distribute the seeds picked from the pot to its neighbouring pot in an anticlockwise manner one pot at a time until all the seeds are finished. Cycle back from pots 12 to 1. Drop a seed into each pot
5. If the last seed is placed in an empty pot, stop the seed distribution
6. Else if the last seed is placed in a pot with 3 seeds inside it, capture the seeds, assigned them to the player's score and end the distribution
7. Else pick the seeds from the pot and repeat step 4
8. Assign seeds that reach up to 4 during distribution to the player who owns the pot they belong to and capture them

This is the basic algorithm for our game. It consists of only 8 steps but does not consider the other issues we must take into consideration.

The first issue to consider is that this version of our game is a 2 player game because of this reason, we have to have a way to track whose turn it is.

At the start of our game, we need to notify the users of our game that player 1 should start the game by clicking a pot from 1 to 6.

In addition to this, we need to ensure that our pots cannot be clicked again until seed distribution has stopped. This will prevent the user from clicking a pot while seed distribution is ongoing and messing up our program.

When a player's turn is finished, we have to allow the other player play and disable the pots of the player who has just finished playing. We will have to do this using code.

We must also take seed capture and scoring of the game into consideration. This is the reason why the game needs to keep track of whose turn it is.

In addition, we must know who owns a pot to be able to score them properly.

The game must also be able to skip the turn of the user who has all their pots empty. The ending

condition for the game is that the total number of seeds in the pots must be less than 8 at that point, the game can end.

Completing our Game

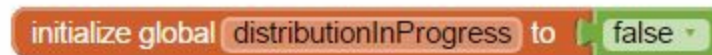
So far we have already completed steps 1 to 4 in our algorithm. In order to complete the game, we need 4 extra variables. They are:

1. `playerID` which is used to identify whose turn it is to play
2. `distributionInProgress` which prevents the user from clicking a pot while seeds are being distributed
3. `playerOneScore` which is the score for the first player
4. `playerTwoScore` which is a score for the second player

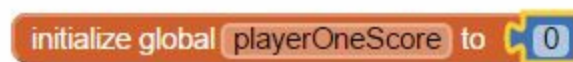
The variables are shown below:



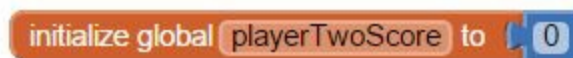
initialize global `playerID` to 1



initialize global `distributionInProgress` to false

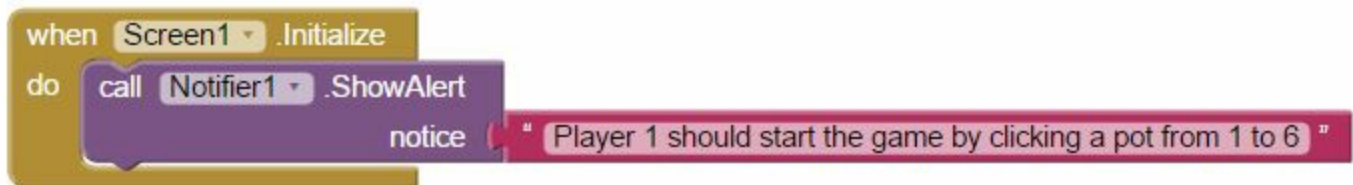


initialize global `playerOneScore` to 0



initialize global `playerTwoScore` to 0

When the game is started, we want to display instructions that player 1 should start gameplay. This is easily done using the Screen component initialize method. The code to do this is shown below:



```
when Screen1 Initialize
do
  call Notifier1 ShowAlert
    notice "Player 1 should start the game by clicking a pot from 1 to 6"
```

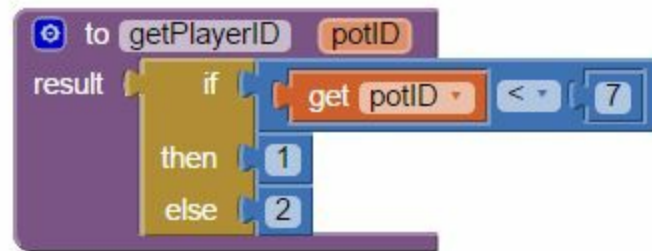
The next thing to do is to ensure that our pots cannot be clicked while seed distribution is ongoing. The way to do this is to test for the value of `distributionInProgress` and display a message to the user to wait for the distribution to end. The code block for all 12 pots will need to be changed to the one for pot 1. The code block for our first pot is shown below:



With the above code, as soon as the seed distribution is ongoing, the player cannot select another pot.

As we are making a two player game, we need to know at each point of the game who turn it is to play. To do this we create a new function called `getPlayerID` which will allow us know the ID for the player whose turn it is.

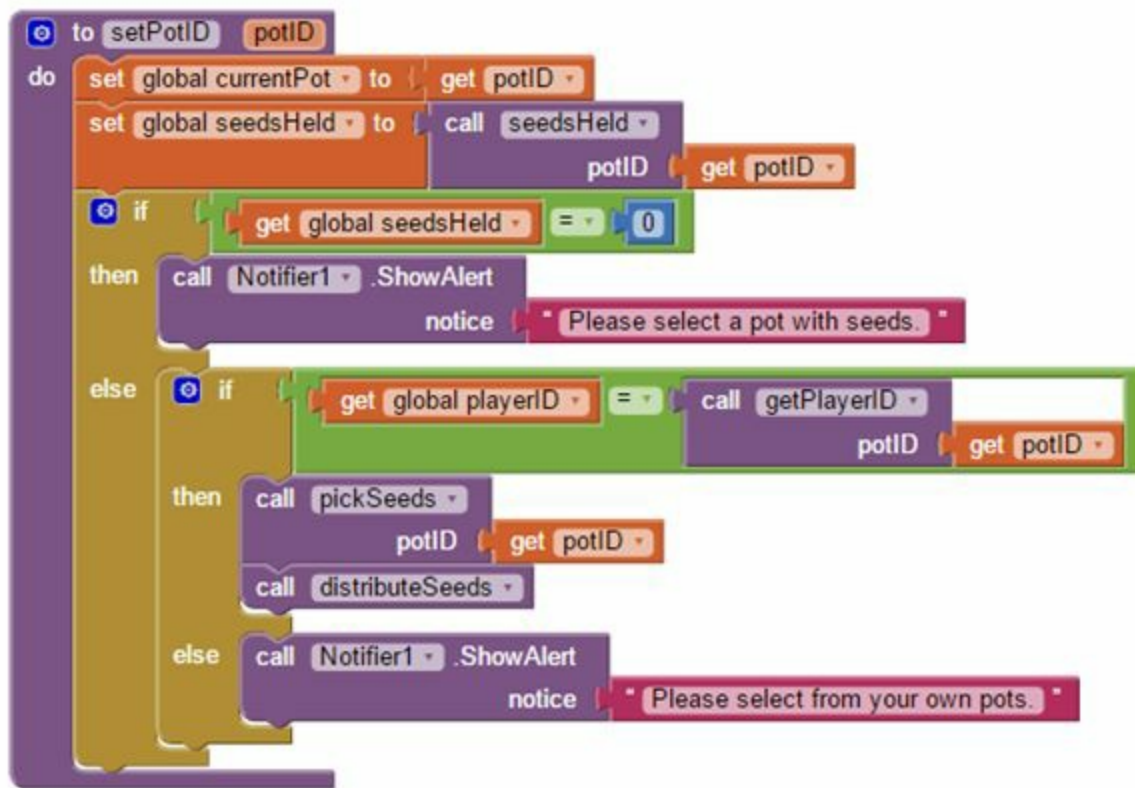
The function is shown below:



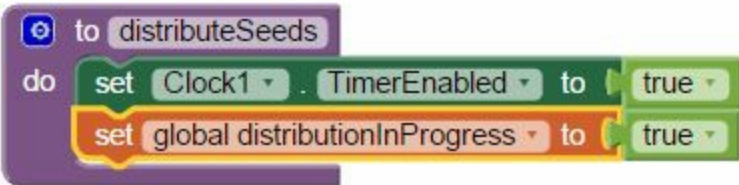
We need to modify our `setPotID` procedure to use the `playerID` to prevent the user from picking other pots apart from their own pots. Do bear in mind that at the beginning of our program, we already have the `playerID` set to 1. This ensures that only our first player can select pots.

With the `getPlayerID` function, we ensure that player 1 can only select pots from pots 1 to 6. This means that in our `setPotID` procedure we have to check that the player picks their own seeds before we can do the seed distribution.

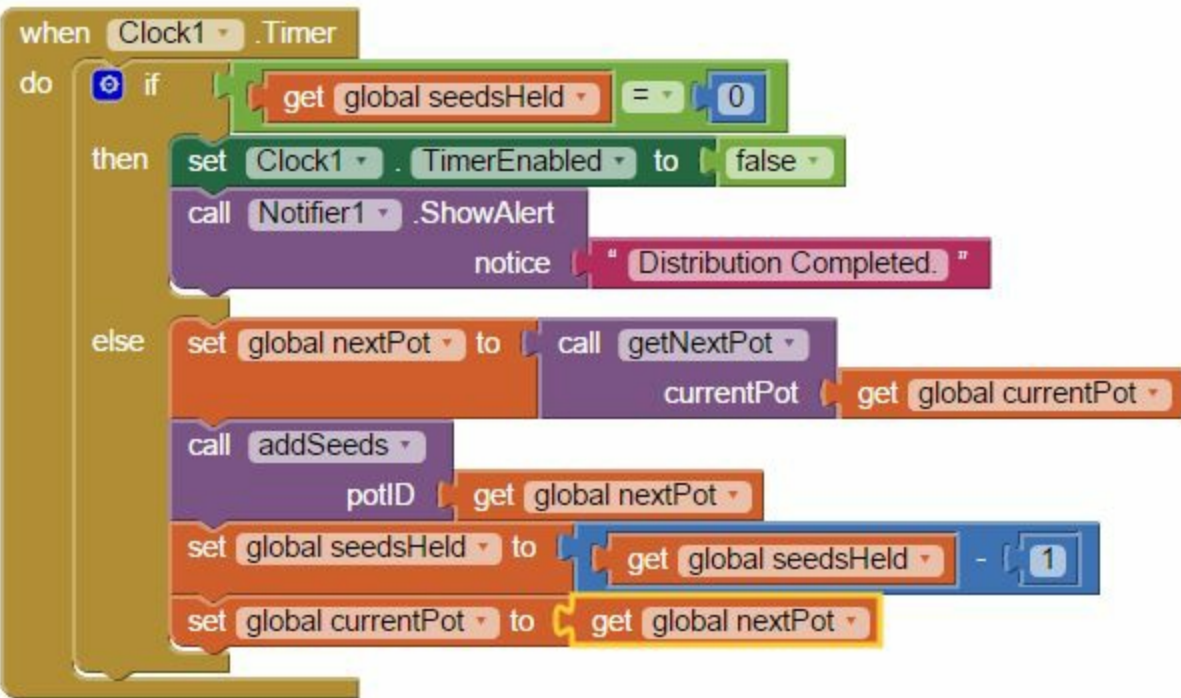
Our modified `setPotID` procedure is shown below:



The moment our setPotID procedure calls the distributeSeeds procedure, the next thing to do is to set the distributionInProgress variable to true. We do this in the body of the code for the distributeSeeds procedure and it becomes:



Now we come to a critical point in the entire program. Our seed distribution in our Clock1 component is currently as follows:



This block of code means that if the number of seedsHeld is equal to 0, the seed distribution will end otherwise, we get the next pot in the seed distribution and we continue the seed distribution until we exhaust the number of seedsHeld.

The problem with the code above is that it runs one off. It doesn't continue the seed distribution past the point where it stops.

The Clock1 component is the engine of the entire application. It executes most of the critical tasks in the execution of this game. Do bear in mind that we have set the interval at which it should execute which is every half a second.

For sowing to continue past the first one, we have to program relay sowing. For this to happen, we must think of a typical relay. At the end of the first lap, the current runner passes on the baton to the next runner who runs a lap and passes the baton to the next runner until the entire race is completed.

That is the same idea behind relay sowing. Until we get to the place where we have an empty pot or we capture seeds, we cannot stop the seed distribution. Along the way, we could cause the number of seeds in our own pots or our opponent's pots to become equal to 4 and they would have to be captured. Our program must take all this into consideration.

In our first if block, once the distribution finally ends, we need to set the distributionInProgress variable to false and switch the playerId. We also have to coordinate gameplay.

A special case in coordinating game play is for the seeds in all the pot of a player to be empty. In the case of our game, this would be when all the seeds from pot 1 to pot 6 are empty for player 1 and all the seeds from pot 7 to pot 12 are empty for pot 2. We have to create functions for this purpose.

The functions we need for this purposes are:

1. sumPlayerOneSeeds
2. sumPlayerTwoSeeds
3. sumTotalSeeds

The code for our sumPlayerOneSeeds function are meant to sum up the seeds in all the pots from pot 1 to pot 6.

```

to sumPlayerOneSeeds
result
  select list item list get global pots + select list item list get global pots + select list item list get global pots +
  index 1 index 2 index 3

```

```

to sumPlayerTwoSeeds
result
  select list item list get global pots + select list item list get global pots + select list item list get global pots +
  index 7 index 8 index 9

```

```

to sumTotalSeeds
result
  call sumPlayerOneSeeds + call sumPlayerTwoSeeds

```

Do bear in mind that sumPlayerOneSeeds extends from 1 to 6 and sumPlayerTwoSeeds extends from 7 to 12. I leave the remainders out because of space to show them. Could this functions be rewritten in another way?

The answer is yes but I leave that to you as an exercise.

The code for our first if block is shown below:

```

when Clock1 . Timer
do
  if
    get global seedsHeld = 0
  then
    set Clock1 . TimerEnabled to false
    set global distributionInProgress to false
    call Notifier1 . ShowAlert
    notice "Distribution completed."
    if
      get global playerID = 1
    then
      if
        call sumPlayerTwoSeeds = 0
      then
        set global playerID to 1
        call Notifier1 . ShowAlert
        notice "Player 1 gets to play again because player 2 has no seeds."
      else
        set global playerID to 2
        call Notifier1 . ShowAlert
        notice "Player 2's turn. Select pots from 7 to 12."
    else
      if
        call sumPlayerOneSeeds = 0
      then
        set global playerID to 2
        call Notifier1 . ShowAlert
        notice "Player 2 gets to play again because player 1 has no seeds."
      else
        set global playerID to 1
        call Notifier1 . ShowAlert
        notice "Player 1's turn. Select pots from 1 to 6."

```

For the else part of our first if block, we need to take into consideration relay sowing. The moment our first sowing ends, we have to check if we can continue. Here our logic becomes tricky because we cannot exit this block.

To exit this block means that our sowing has come to an end. With this in mind, we have to establish the conditions under which our sowing will continue. It is also in this block that our sowing leads to seed capture and scoring.

The pseudo code for this is shown below:

```
if seedsHeld = 0
    if potSeeds > 1 and not equal to 4
        seedsHeld = seedsHeld(currentPot)
    else if potSeeds = 1
        seedsHeld = 0
    else if potSeeds = 4
        seedsHeld = 0
        scorePlayer()
else
    if potSeeds = 4
        scorePotOwner()
```

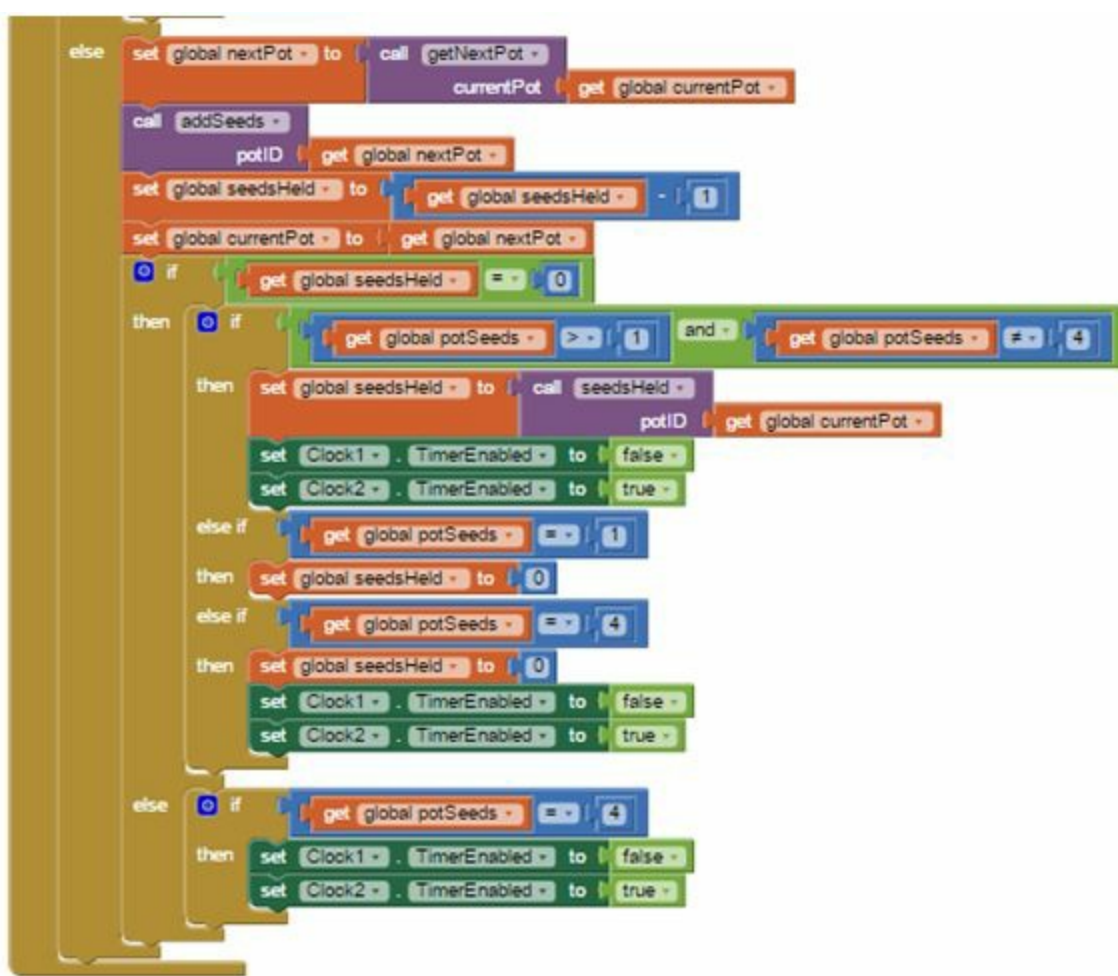
It is critical that you understand the pseudo code before I show you the code. If the number seedsHeld is equal to 0, we need to check for whether our pots seeds are greater than 1 and not equal to 4

. We have to check it this way because the potSeeds variable is used by the addSeeds procedure to increase the number of seeds in the pot visited by the distribution by 1. This means that the increment has already happened by the time we do this test.

If the value of potSeeds is 1, it means that we landed on an empty pot and the distribution can stop. However, if it is 4, it means that we have to capture the seeds in that pot.

The task of displaying this to the user will need a second clock. This is because we need to stop our first clock and let our second clock show the user what is going on at each stage. Our second clock will not only show when the seed distribution ends but when seed capture occurs.

What we need now is a second clock. We need to give it exactly the same properties as we gave to Clock2. Clock2 will also be responsible for scoring. It will be the interplay between the two clocks that will make the animation in our program work. So the code to complete our first else block for Clock1 is shown below:



The program for our second clock should handle seed capture. However, we must bear in mind that the two conditions we must attempt seed capture for are for when the player lands in a pot with 3 seeds and thus captures them and when a player in the course of seed distribution encounters a pot with 3 seeds in the course of seed distribution.

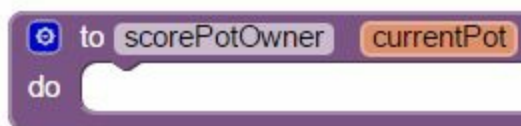
In that moment, because the number of seeds held is not 0, the player has to continue seed distribution. The seeds captured in this manner belong to the player who owns the pots.

Before we can complete the code for the second clock, we need to first create stubs for our `scorePlayer` and `scorePotOwner` procedures respectively.

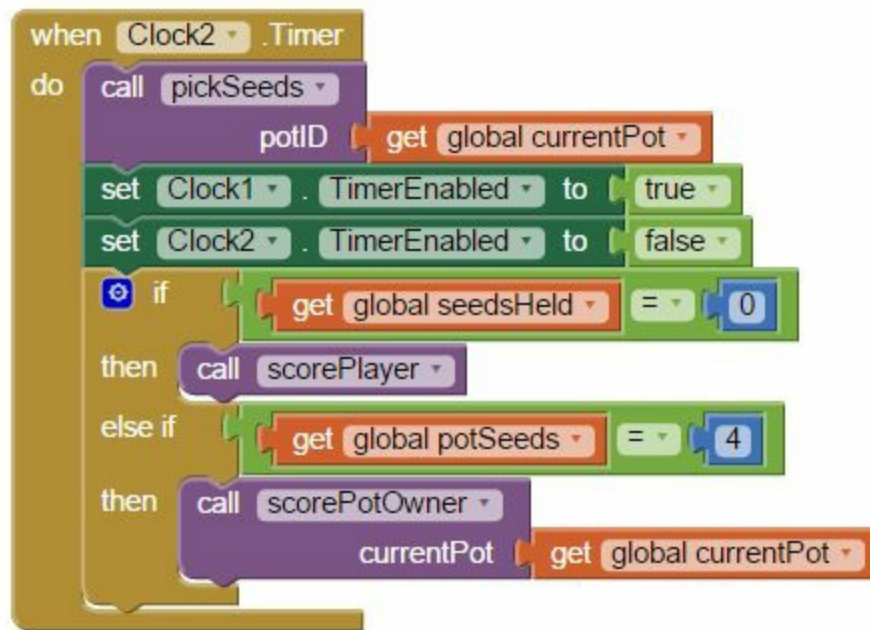
The stub for the `scorePlayer` function is shown below:



The stub for the `scorePotOwner` function is shown below:



The code for our second clock is now shown below:



The first part of the code picks the seeds in the pot that has the ID of the currentPot. Next we activate Clock1 and deactivate Clock2. We then test for the condition we need to follow to do the scoring.

When the seedsHeld variable is 0, it means that the distribution ended and the player who played should have his score updated. However, when the potSeeds is 4, it means that in the course of distribution, the seeds in the pot became 4 and so we simply have to add to the score of the pot owner.

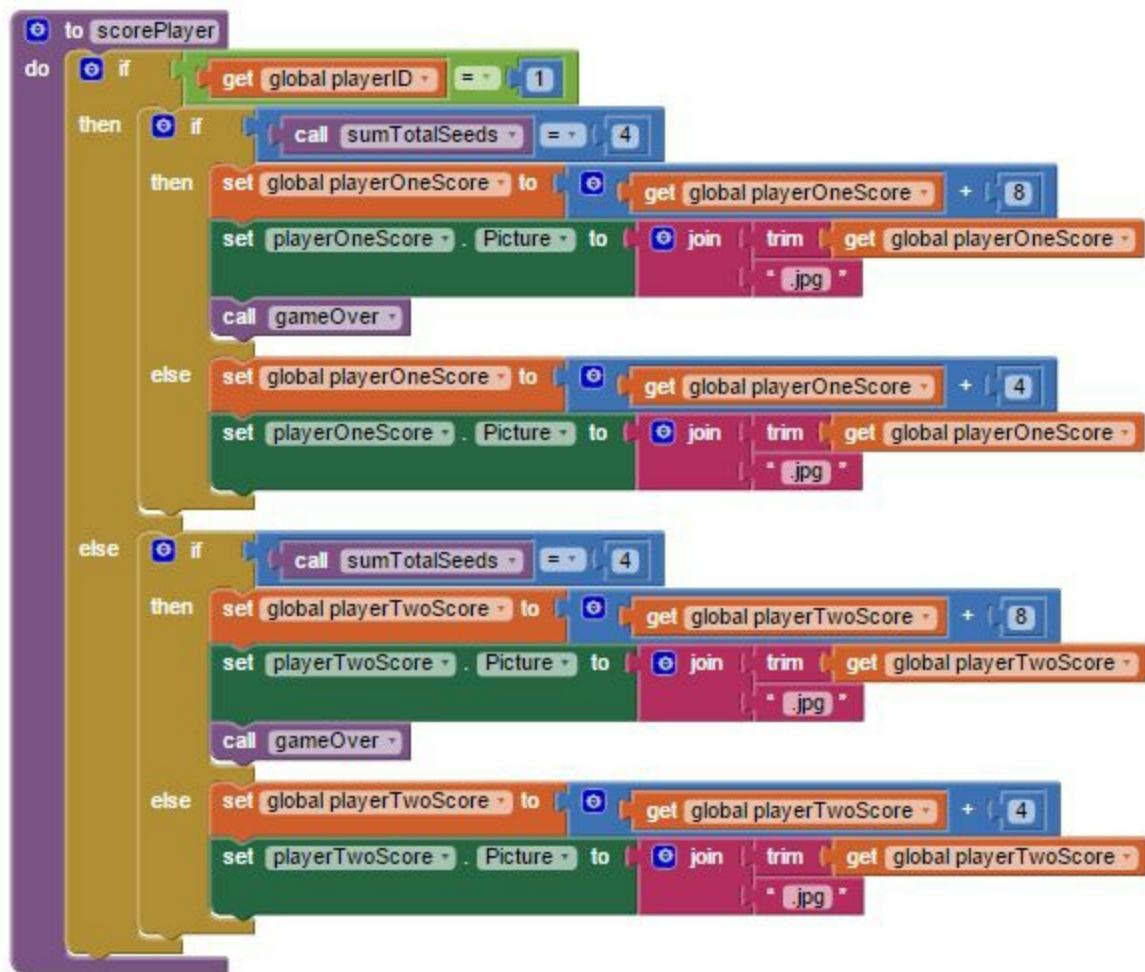
We cannot create the scorePlayer procedure without creating the stub for the gameOver procedure. This is because the very act of scoring in this game leads to depletion of the board, we have to end the game the moment the number of seeds drops critically.

For the game of Ise, this is the point where the total number of seeds on the game board is equal to 4. At this point, the game must end. The first player to score one the total number of seeds in the pots on the game board gets the last 4 seeds.

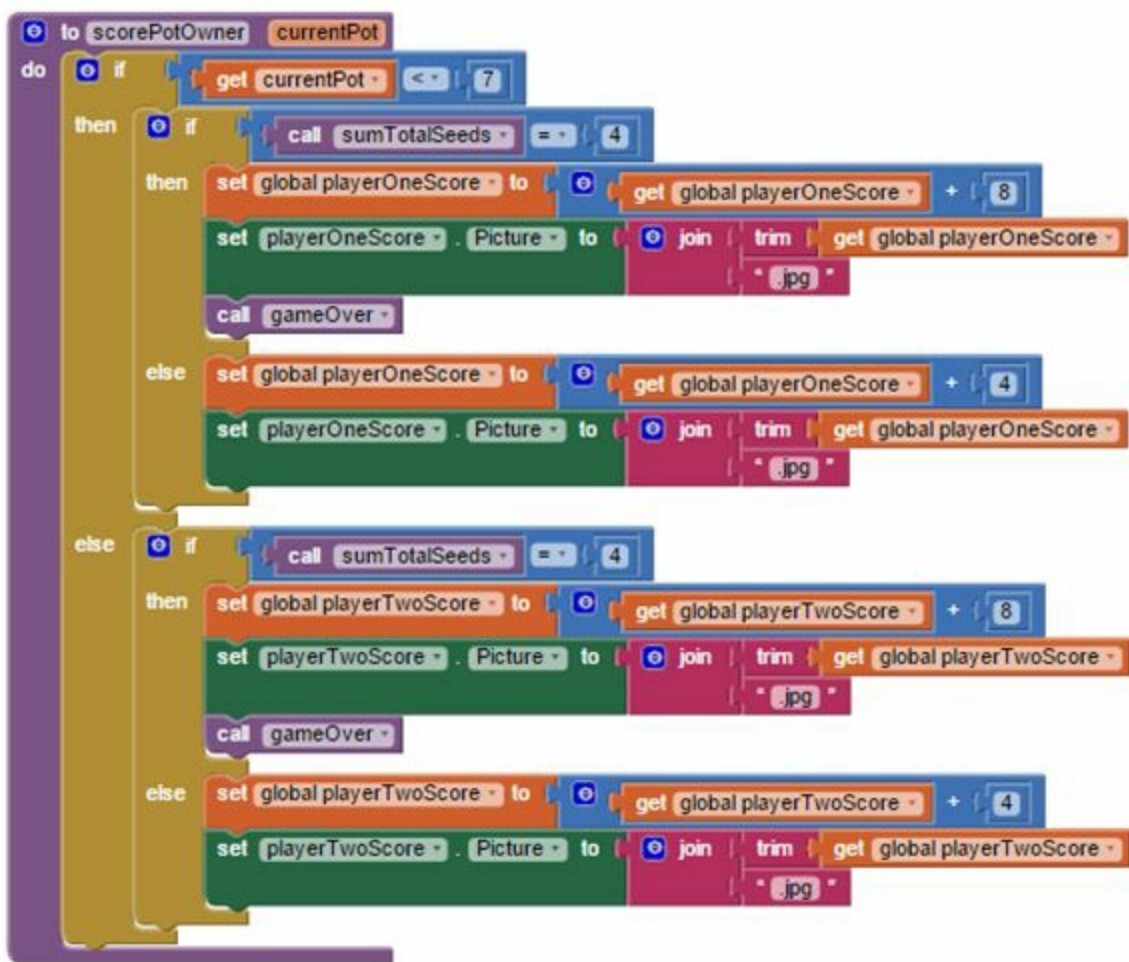
For our purposes, we shall create the stub for the gameOver function. This is shown below:



scorePlayer is a procedure to add to the score of the player who's turn it was that lead to seed capture. The code is self explanatory and is shown below:



scorePotOwner is a procedure to add to the score of the owner of the pot whose seeds have been increased up to 4 during the seed distribution. In this case the distribution would continue but we have to stop seed distribution and score the owner of the pot. The code to do this is shown below:



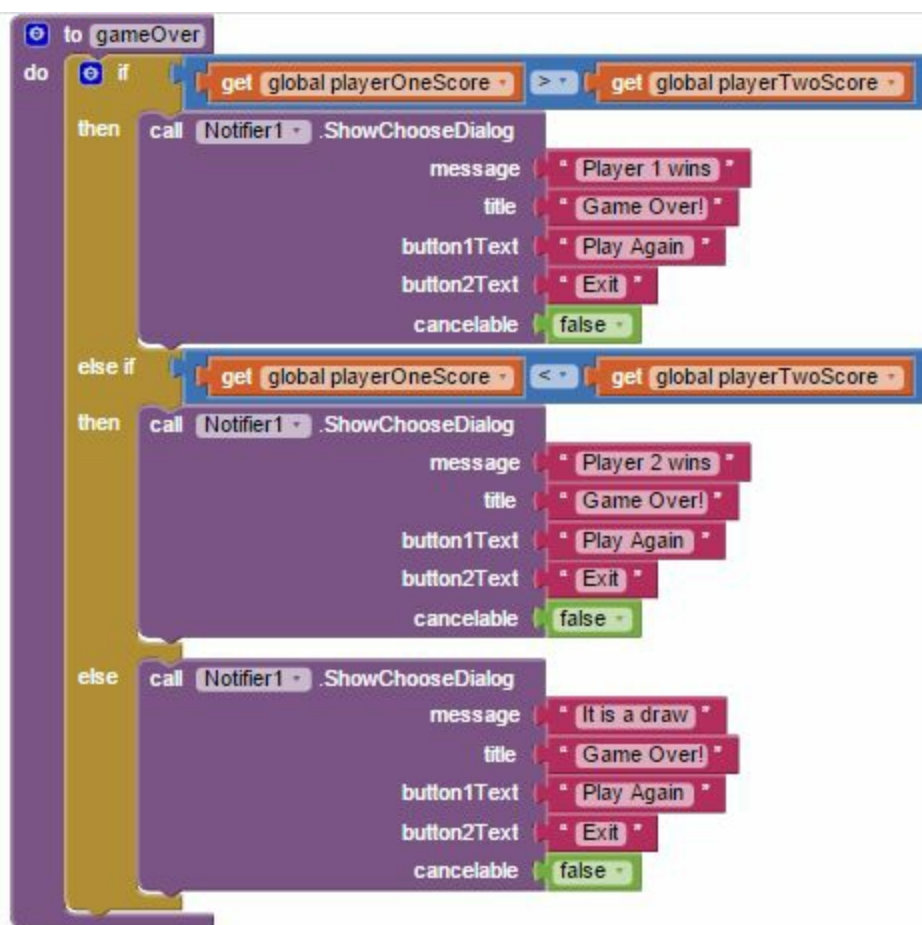
The scorePlayer and scorePotOwner functions are similar and we could refactor the code to make it cleaner. I leave that as an exercise for you.

Our gameOver procedure notifies the player that the game has ended and gives us the option to either play again or to exit the application.

The code to give us the option to play again will be enclosed in a restartGame procedure. The stub of the restartGame procedure is shown below:



Now at last we can assemble the code for our gameOver procedure. This is shown below:



Now when the game ends, we get to choose if we want to play again or not. For this to work, we have to use the AfterChoosing event of the Notifier component. The code for this is shown below:



Lastly we create our restartGame procedure. This procedure will take our game back to the starting condition and display the notification for the first player to start all over again.

The restartGame procedure is our final procedure. It requires us to revisit all the work that we have done so far.

The first thing we are to do is to reset our list to its original condition where every item in the list is 4. We shall use a procedure for this called resetList.

After resetting our list, we must also change the image displayed in each of the pots back to 4 like at the start of the game. The procedure to do this shall be called resetPotImages.

Next we need to reset all our variables. The procedure we shall use for this purpose will be called resetVariables.

After resetting our variables we have to disable our clocks and this can be done using the same restartGame procedure.

Lastly, we shall display the notification for the user to play again. This will be done in the restartGame block.

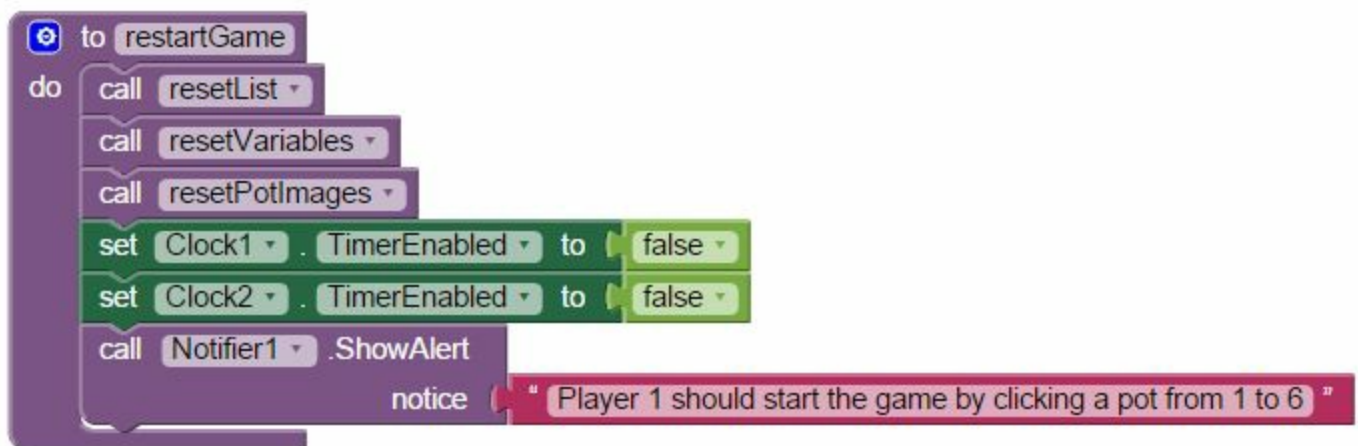
Before we go on, we are at point where we can build our app. So let's take the time to do this. Test it and play with it.

So far all the tests work out. Now let's get back to creating the restartGame procedure.

The procedures we need to work with the restartGame procedure are listed below:

1. resetList
2. resetVariables
3. resetPotImages

I won't go into the creation of stubs for this procedures. The code for the restartGame procedure is shown below:



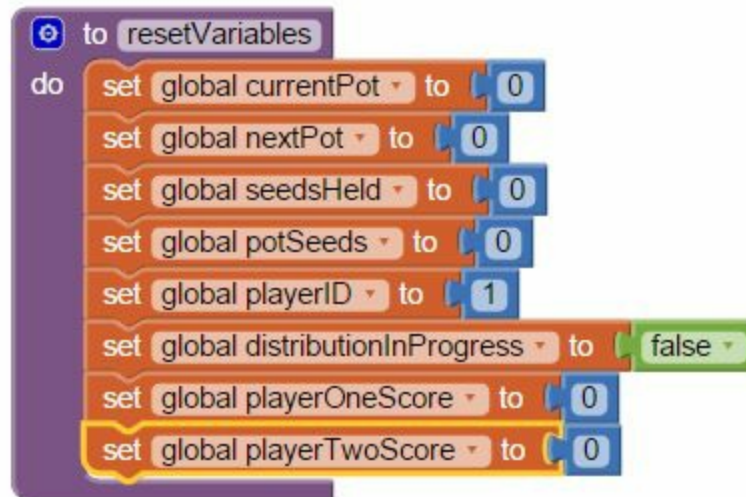
The code for our resetList procedure uses a loop. If you have stuck with me this far, I believe you know what a loop is. We use a for loop that starts at 1 and ends at 12 with increments of 1. The code for this is shown below:



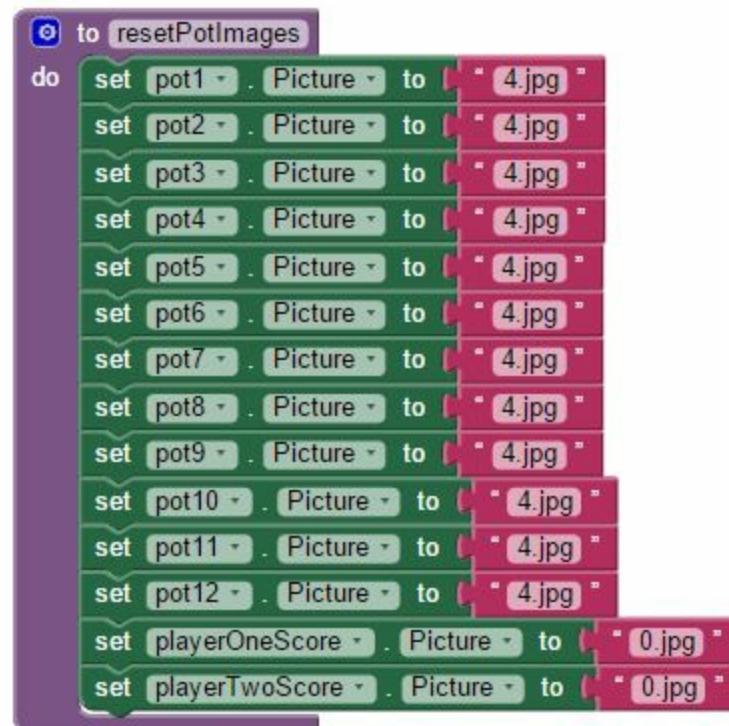
We now move on to create the resetVariables procedure. We have a total of 8 variables in our program. They are:

1. currentPot with a default value of 0
2. nextPot with a default value of 0
3. seedsHeld with a default value of 0
4. potSeeds with a default value of 0
5. playerID with a default value of 1
6. distributionInProgress with a default value of false
7. playerOneScore with a default value of 0
8. playerTwoScore with a default value of 0

All of which will be reset in our resetVariables procedure. The code to do this is shown below:



In order to reset the pot images, we need to set the Picture property of each of our Sprites. The code for this is shown below:



Now we put our game to the test with the goal of finding out if the restart button works.

Conclusion

We have come to the end of this guide. I understand that it was hard so I am glad you made it to the end. You have gone the distance and won.

Do remember that no matter how hard it was for you follow this guide, it was harder for me to write it.

Today with the ubiquity of computers, knowing how to program has become akin to having a superpower. Whatever you do with this knowledge is up to you.

Just remember that with great power comes great responsibility.

Android Development for Everyday People



I hope that you have enjoyed reading this guide. It was a deep dive into [App Inventor](#). If you want a gentler introduction to [App Inventor](#), please check out my book [Android Development for Everyday People](#) on the [Amazon store](#).

It costs \$5 and is a great read for an absolute beginner to programming and application development.

I wish you health, wealth and happiness in this great gift called life.

About the Author



[Truston Ailende](#) is the Founder and CEO of [Dreamquest Multimedia](#), an education technology company with a focus on African heritage and history.

He is a teacher, author and technology entrepreneur with over 10 years of programming experience for a variety of platforms covering desktop, web and mobile.

In 2010 he created his first game and embarked on a journey to develop a games studio in Nigeria. After years of failing, he finally figured out how to build a business around his passion.

Truston studied Systems Engineering at the University of Lagos and holds a Masters in Entrepreneurship from the University of Hard Knocks.

