|  |  |
| --- | --- |
| In this lesson the template provides a standard Logo platform with procedures such as *forward(N)* and *turn(M).* Using these procedures you will design, implement, and test algorithms to geometrical figures, such as triangles and pentagons, and then you’ll use these shapes in loops to draw even more interesting concoctions.    **Objectives**: In this lesson you will learn to:     * use *Logo* commands to draw shapes; * understand how procedures,parameters, and loop work; * define your own procedures, your own abstractions, to draw more complex shapes. | ***[Click to watch Preview Video](http://www.youtube.com/watch?v=U0X5xan1XWM)*** |

## Getting Ready

Open [App Inventor with the Logo Part 2 Template](http://ai2.appinventor.mit.edu/?repo=templates.appinventor.mit.edu/trincoll/csp/unit5/templates/LogoWParamsTemplate/LogoWParamsTemplate.asc) in a separate tab and follow along with the following tutorial. Once the project opens use *Save As* to rename your project ***Logo2****.*

Note: If the blocks don’t appear well in the Blocks Editor, right-click on the background and use the *Arrange Vertically* option.

# Logo 2 : Algorithms with Repetition and Selection

|  |  |
| --- | --- |
|  | The template you load will provide a running app that lets you draw using simple Logo drawing commands.  The commands are implemented in buttons:   * Fwd -- make the Android go forward 10 pixels. * Right -- make the Android turn right 90 degrees. * Up/Down -- a toggle that takes the Android off the canvas (no drawing mode) or puts it on the canvas (drawing mode). * Hide/Show -- a toggle that makes the Android disappear or reappear. * Draw -- this button will execute whatever algorithm you put into the *draw* procedure as part of this lesson’s exercises. The final exercise in this lesson asks you to draw a face, similar to the one shown here -- details below. * Reset -- this button clears the canvas and puts the Android back at the middle. |

# Introduction

In our [previous Logo lesson](https://docs.google.com/document/d/1JI5Twi34YIPRvIizC8wRe7ItkV8zQhwfLt4E60QVnng/edit#heading=h.qk106ns2ykxr) you developed algorithms for drawing simple shapes. But it was hard because the commands you used were very weak and inflexible. For example, the *forward* command could only be used to move the Android forward by 10 pixels. The *turn* command could only turn the Android by 90 degrees. With those commands drawing a square with sides of 100 pixels was very tedious and it was impossible to draw a simple triangle.

In this lesson we’ve improved our set of commands by making them ***more general.*** The primary improvements are in the *forward(N)* and *turn(D)* commands:

* The *forward(N)* command moves the Android **forward by *N* pixels**.
* The *turn(D)* command causes the Android to **turn right by *D* degrees**.

The *N* and *D* here are ***parameters***  or ***arguments.*** And a simple example should suffice to show how they are more general and, hence, more powerful. In the previous version to move forward by 40 pixels would require 4 statements, each of which moved the Android forward by 10 pixels:

|  |  |
| --- | --- |
| *forward*  *forward*  *forward*  *forward* |  |

With this new set of commands to move forward by 40 pixels (or any number of pixels) requires only one command:

|  |  |
| --- | --- |
| *forward(40)* |  |

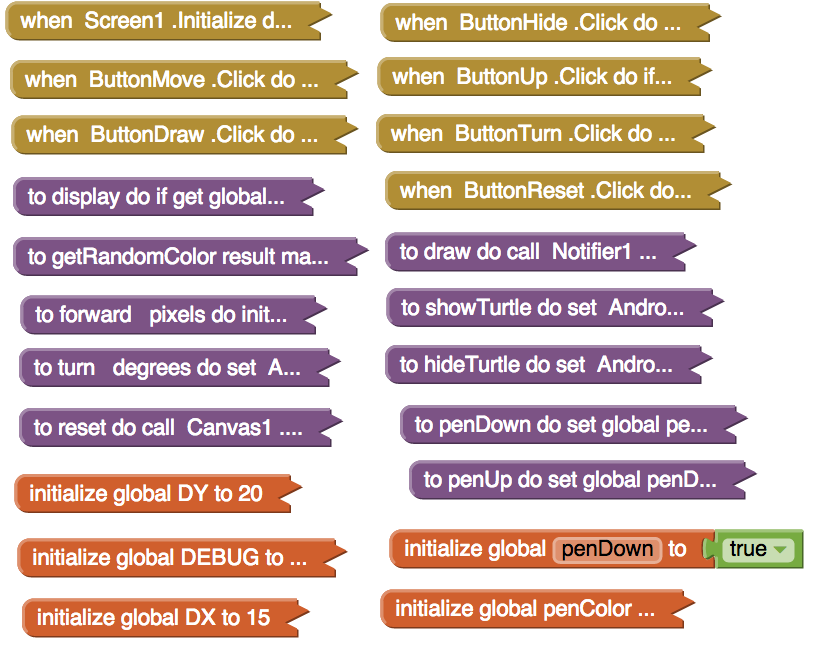
The earlier version of *forward()* was very ***specific*** whereas the new version, with a parameter, is more ***general*** and it is the presence of the parameter that gives it its generality. Instead of always going forward by 10 pixels, we can now go forward by any number of pixels with one procedure call. The same observations would apply to the *turn()* procedure. The earlier abstraction was too specific, allowing us only to turn by 90 degrees. The new one, because it involves a parameter, lets us turn by any number of degrees.

**NOTE:** The old version and the new version of Logo procedures are both ***abstractions***. But clearly, the new set of abstractions are much more powerful. As a rule of thumb, ***the more general a procedure (or abstraction) the better.***

The rest of the Logo commands are the same as in the previous version:

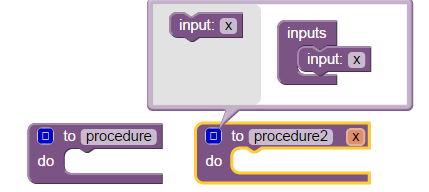
* The *penUp* command pulls the pen off the canvas so nothing is drawn when the Android moves. The *penDown* command puts the down on the canvas so it will draw.
* The *showTurtle* command makes the Android visible. The *hideTurtle* command hides the Android invisible.
* The *draw* command moves the Android according to the code you specify. Here is where you will put your algorithms.
* The *reset* command clears the canvas and moves the Android back to its starting position.

As in the earlier lesson, these commands are implemented in App Inventor procedures:



# Defining a Procedure with Parameters

In the previous lesson, you defined procedures without parameters. Later in this lesson, you will need to define procedures *with* parameters. To do this, you will need get a procedure block from the Procedures drawer. As always, you should give your procedure an appropriate name. To specify that the procedure you have defined requires a parameter when called, click the blue plus sign and drag an input block from the left into the inputs block on the right (Similar to how you would add an else to an If block.)

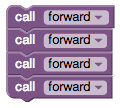


Replace x with a useful and helpful parameter name. Click the blue minus sign when you have finished adding the parameters needed for the procedure.

# Algorithms with Repetition and Selection

In designing algorithms there are three basic types of ***control structures:***  sequence, selection, and repetition. Just about any algorithm you can think of can be built using these three types of controls.

**Sequence.** You are already familiar with ***sequence***, which just means a sequence of steps. In App Inventor we arrange blocks in a sequence by placing them top to bottom:



**Selection.** You are also familiar with ***selection***, which is just the term we use for *if-else* structures:

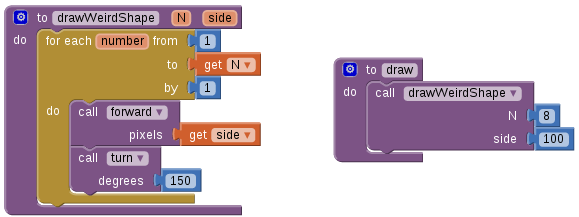


**Repetition.** So far we haven’t used ***repetition*** or ***looping*** in our algorithms. But here’s a simple example of where a loop can be very helpful. Compare the algorithms on the left and right:

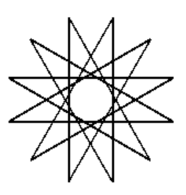
|  |  |
| --- | --- |
|  | forEachSquare.png  **How to read the *for-each* block:**  For each number (the loop counter) from 1 to 4 varying by 1, do the statements in the do-slot. So i will go from 1 to 2 to 3 to 4 and each time it changes the statements in the do-slot will be done.  In other words, repeat the statements in the do-slot 4 times. |

The algorithm on the left uses a simple sequence with copies of the *forward* and *turn* blocks to draw a square, whereas the algorithm on the right uses a ***for-each loop****,* a much more practical and general approach. The *for-each block* in this case ***repeats*** the statements in its *do-slot* 4 times. There are other uses of the *for-each* block that we will explore in future lessons. But in this example and for many of the exercises in this lesson it can be used as shown here to count from ***start*** to ***end*** by ***steps*** of size 1.

In this example we placed constants for the ***start,* *end,*** and ***step*** slots. But you can also place variables in those slots. For example, here’s a procedure that draws a weird star-like shape. Notice that the variable *N*, which represents the number of repetitions, is used as the value for the ***end*** of the loop. When we call *drawWeirdShape(8,100)* it does 8 repetitions of going forward by *100* pixels and turning right by 150 degrees.



Try copying that procedure into your app and playing around with the number of repetitions. See if you can figure out how many repetitions are needed to draw this star:



# Exercises

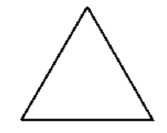
**0.** Download and print [**this graph paper**](https://drive.google.com/file/d/0B5ZVxaK8f0u9NjNuaTZ5S0Z4OUE/edit?usp=sharing) to use when designing your algorithms for the following exercises.

**1.** Using the *for-each* algorithm given above for drawing a square, define a procedure called *drawSquare(L)* that will draw a L x L square where *L* is the length of the side.

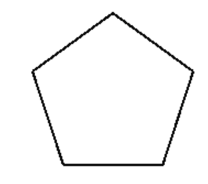
NOTE and HINT: *In App Inventor and other programming languages the name of the parameter doesn’t matter so you can use names that are descriptive of the parameter’s purpose.* For example, either of these procedure definition blocks could be used as the basis of your *drawSquare* procedure. The key is to use parameter names that are meaningful to you and other programmers.

|  |  |
| --- | --- |
| 5.4_Exercise1_1.png | 5.4_Exercise1_2.png |

To test your algorithm, you have to call it from the *draw()* procedure because that’s the procedure that will be run when you click on the Draw button.

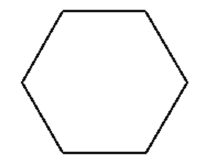
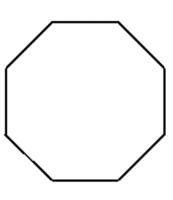
**2.** Design an algorithm for drawing an equilateral triangle -- i.e., a triangle with equal sides and equal angles. First design it by hand -- how much does the Android have to turn? Because this is another example of a ***repetition*,** you can use the *for-each* statement in your algorithm. How many repetitions are necessary? 

Once you’ve got the algorithm figured out, implement it in App Inventor and test it. Because you might want to use your triangle algorithm again, define it into a procedure with a parameter. What should the parameter represent?

**3.** Draw a pentagon -- i.e., a 5-sided figure with equal sides and angles. Again, first design it by hand -- how much does the Android have to turn to draw a pentagon? Since this is another example of a ***repetition*,** use the *for-range* statement in your algorithm. How many repetitions are necessary? 

HINT: To draw a square the Android had to turn 4 times each by 90 degrees meaning it turned a total of 360 degrees.

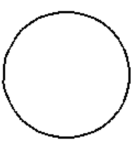
Once you’ve got the algorithm figured out, implement it in App Inventor and test it. Because you might want to use your pentagon algorithm again, define it into a procedure with a parameter. What should the parameter represent?

**4. (Advanced)** Squares and pentagons are both examples of a more general shape, a polygon. A ***polygon*** is an multi-sided figure. So a square is a polygon with 4 sides and a pentagon is a polygon with 5 sides. If you could design a *polygon(N)* procedure, then you could use it to draw a square or a pentagon or hexagon (6 sides) or octagon (8 sides) or even approximate a circle (36 sides?). So give it a try. There’s quite a payoff if you can do it.

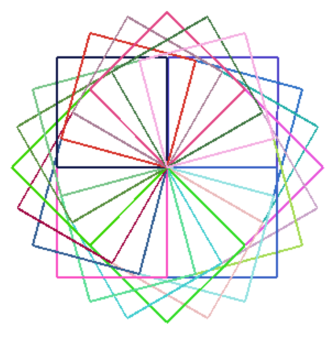
HINT: Your procedure will need 2 parameters, N, and L, where *N* is the number of sides (e.g., 4, 5, 6, etc.) and *L* is the length of each side.

HINT: A 4-sided figure has 4 sides and turns by 360/4 degrees. A 5-sided figure has 5 sides and turns by 360/5 degrees.

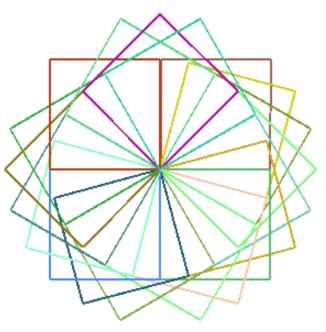
Test your polygon() procedure by using it to draw a hexagon (6 sides) and a octagon (8 sides). Again, you will have to call your procedures from the *draw()* procedure.

**5.** Use your *drawPolygon()* procedure to draw a circle. This exercise might require some trial and error to get the the number of sides and the length of the sides right. Does a 36-sided polygon look like a circle?

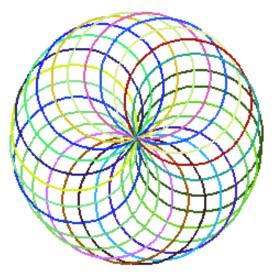
**6.** Draw a flower by repeatedly drawing a square and turning right by some number of degrees. (***NOTE***: To change the color of the drawing pen you need to set the *Canvas.PaintColor* property. If you want a random color you can use the *getRandomColor()* block that is provided. Setting the global *penColor* variable won’t have any effect on the Canvas.)



**7.** Draw a flower with some missing petals -- i.e., use if/else and randomness.



**8.** Design and draw your own shapes, including flowers, spirals, stars. For example, here’s an interesting flower-like shape that was made by rotating a circle:



# Conclusion

## The lesson here is that our choice of abstractions, in this case the use of parameters in our Logo commands, affects the kinds of problems we can solve and how we solve them. That is, our choice of abstractions have an enormous impact on our algorithms. In addition, procedural abstraction (both with and without parameters) makes algorithms easier by raising the level of abstraction

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