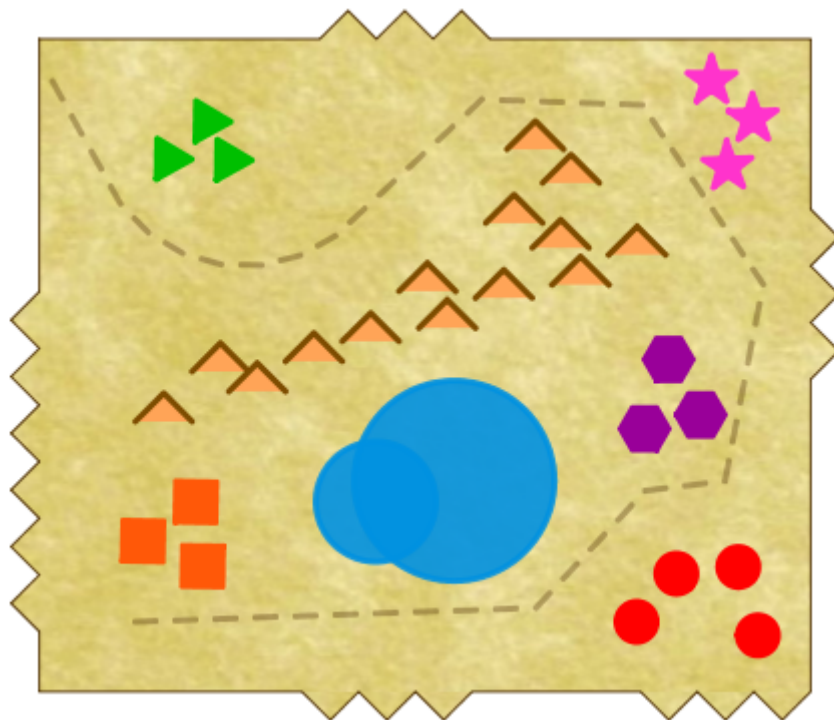


# Explorations

in  
**Math, Art, Programming, Learning, and Science**  
with Kojo



Lalit Pant

---

# Explorations

in

## Math, Art, Programming, Learning, and Science with Kojo

by  
Lalit Pant

with contributions and feedback from  
Vibha Pant, Anusha Pant,  
the kids at the Kalpana Center, and REACHA

Cover art (made in Kojo) by Anusha Pant

**Version:** February 27, 2015



License: Creative Commons *Attribution-NonCommercial-ShareAlike 4.0 International*  
[CC BY-NC-SA 4.0](https://creativecommons.org/licenses/by-nc-sa/4.0/) (see last page for more details)

© 2010–2015 Lalit Pant (lalit@kogics.net)

<http://www.kogics.net>

# Contents

1 A Note for Facilitators and Teachers	6
2 Introduction	8
3 Commands and Programs	10
4 Using Kojo Effectively	14
5 Drawing with Colors	18
6 Mixing Colors With a Function	21
7 Hopping with Speed	25
8 Digging Deeper with Tracing	27
9 Angles	30
10 Practice	32
11 Calculations	33
12 Repeating Commands	37
13 Practice with repeat	39
14 Turning with a radius	40
15 More Fun with Repeat	44
16 Repeat with a sequence	47
17 Art Breakout	50
18 Absolute position and heading	51
19 Exporting your creations	54
20 Random Numbers and Named Values	57
21 Your own Commands	62

<b>22 Your own Commands, with Inputs</b>	<b>66</b>
<b>23 Polygon Art</b>	<b>70</b>
<b>24 Pattern Drawing Practice</b>	<b>76</b>
<b>25 Patterns</b>	<b>77</b>
<b>26 More Practice with Patterns</b>	<b>80</b>
<b>27 Your own Functions</b>	<b>82</b>
<b>28 Mini Project</b>	<b>86</b>
<b>29 Strings and I/O</b>	<b>87</b>
<b>30 Artistic Text in the Canvas</b>	<b>91</b>
<b>31 Conditionals</b>	<b>94</b>
<b>32 Art Breakout</b>	<b>98</b>
<b>33 Recursion</b>	<b>100</b>
<b>34 Practice with Numeric Patterns</b>	<b>103</b>
<b>35 Include Files</b>	<b>104</b>
<b>36 Playing MP3 Music</b>	<b>107</b>
<b>37 Introduction to Electronics and the Arduino platform</b>	<b>109</b>
<b>38 Getting started with Arduino programming in Kojo</b>	<b>111</b>
<b>39 Lights, Music, Action...</b>	<b>113</b>
<b>40 Turtle Commands Quickref</b>	<b>115</b>
<b>41 Exercise Solutions</b>	<b>117</b>
41.3 Commands and Programs . . . . .	117
41.4 Using Kojo Effectively . . . . .	117
41.5 Drawing with Colors . . . . .	118
41.6 Mixing Colors With a Function . . . . .	119
41.7 Hopping with Speed . . . . .	121
41.8 Digging Deeper with Tracing . . . . .	122
41.9 Angles . . . . .	123
41.10 Practice . . . . .	124
41.14 Turning with a radius . . . . .	125
41.15 More Fun with Repeat . . . . .	126

---

41.16 Repeat with a sequence . . . . .	127
41.18 Absolute position and heading . . . . .	128
41.20 Random Numbers and Named Values . . . . .	129
41.21 Your own Commands . . . . .	130
41.22 Your own Commands, with Inputs . . . . .	131
41.23 Polygon Art . . . . .	131
41.27 Your own Functions . . . . .	132
41.28 Mini Project . . . . .	133
41.29 Strings and I/O . . . . .	135
41.30 Artistic Text in the Canvas . . . . .	135
41.31 Conditionals . . . . .	136
41.33 Recursion . . . . .	136
41.35 Include Files . . . . .	137
41.36 Playing MP3 Music . . . . .	137

# 1 A Note for Facilitators and Teachers

This book is about interactively exploring different topics in Math, Art, and Science via Programming. Here's a quick word on these important areas of learning:

- **Math** is the language of structure and pattern, and underlies all of our efforts to understand the world around us in a systematic way.
- **Art** helps us to appreciate beauty and experience the mysterious, and kindles the creative and joyous spark in us.
- **Science** is our way of trying to make sense of nature (using Math).
- **Programming** is a systematic way of representing structure and pattern – in a formalism that can be executed directly on computers by us to do useful things like the following:
  - playing with ideas (in Math, Science, and Art).
  - creating information structures (like the Internet).
  - automating the world around us.

These are crucial subjects for kids to get comfortable with – to be literate, knowledgeable, and happy citizens of the 21st century world. This book helps with the learning of these subject by getting kids to *create* things using them. With the help of this book, they create drawings, algorithms, electronic circuits, and sound effects. Later volumes in this series will help them create animations, games, robots, and more.

The interactive software used to explore the above areas in the book is called the *Kojo Learning Environment*. Kojo is an award winning open-source app that is used by tens of thousands of people around the world. It is powerful, feature rich, and robust, and is available on Linux, Windows, and Mac.

The ideas *used* in this book while *creating* things include:

- **Math** – unit length, distances, integers, rational numbers, fractions, ratio and proportion, percentages, estimation, angles (definition, supplementary, interior and exterior), circles, arcs, polygons, coordinate geometry, LCM, HCF, BODMAS, equations, arithmetic sequences, random numbers, symmetry, simple interest, averages, and visual and numeric patterns.
- **Art** – all kinds of shapes and colors, brought together creatively.
- **Programming** – turtle graphics, values and variables, commands and functions, algorithms, primitives, composition, and abstraction.

- **Science** (Physics) – atomic structure, potential difference, current, resistors, and ohm's law.

This book has a couple of additional, very important, goals – to give kids practice, via Kojo, with the following:

- **different modes of thinking** – inductive, deductive, systematic, analytical, creative, etc.
- **learning how to learn** new things using **exploration, discovery, and feedback**. This includes helping them to learn various problem solving strategies.

These are essential skills for the dynamic, information based work environments of the 21st. century.

## 2 Introduction

The book will help you to:

- **Create** beautiful computer generated art.
- **Make** electronic circuits to do fun things.
- **Apply** math to enrich your creations (drawings and circuits).
- **Learn** computer programming (a very important 21st century skill), math, and physics, along with thinking and problem solving skills – as you go about making your creations.

You will do all of this using a learning environment called *Kojo*. To learn more about Kojo before you begin this journey, you are welcome to take a quick look at the *Kojo, An Introduction* ebook.

The following is a picture of the Kojo workspace:

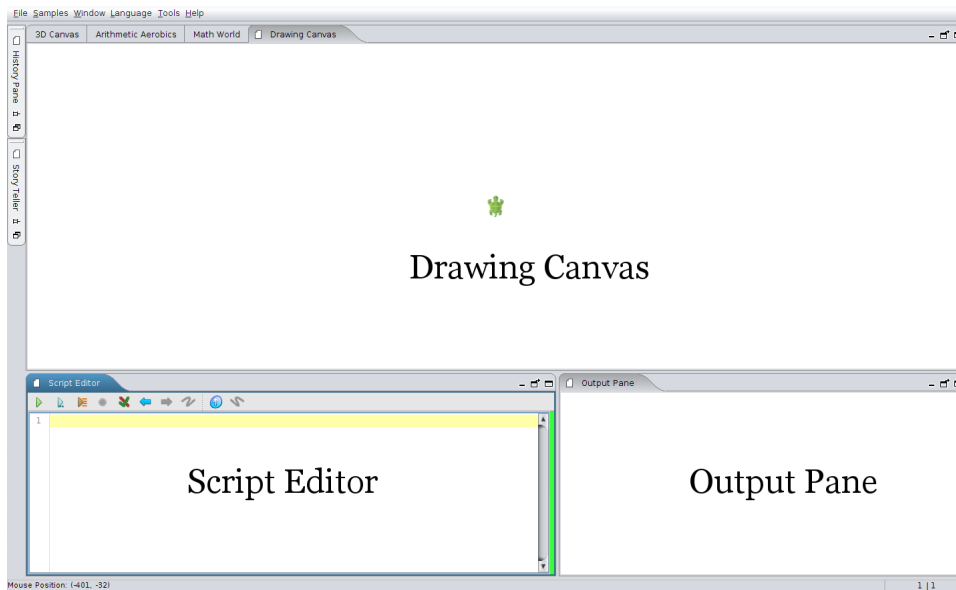


Figure 2.1: The Kojo Workspace

When you start up Kojo, you see a turtle sitting near the center of the screen, within a window called the **drawing canvas**. Your programs instruct this turtle to do things. You can ask it to move forward, drawing a line as it moves. You can ask it to turn left or right. You can ask it to change the color of the lines it draws, and the shapes it fills. All of this can be used to make very interesting drawings.

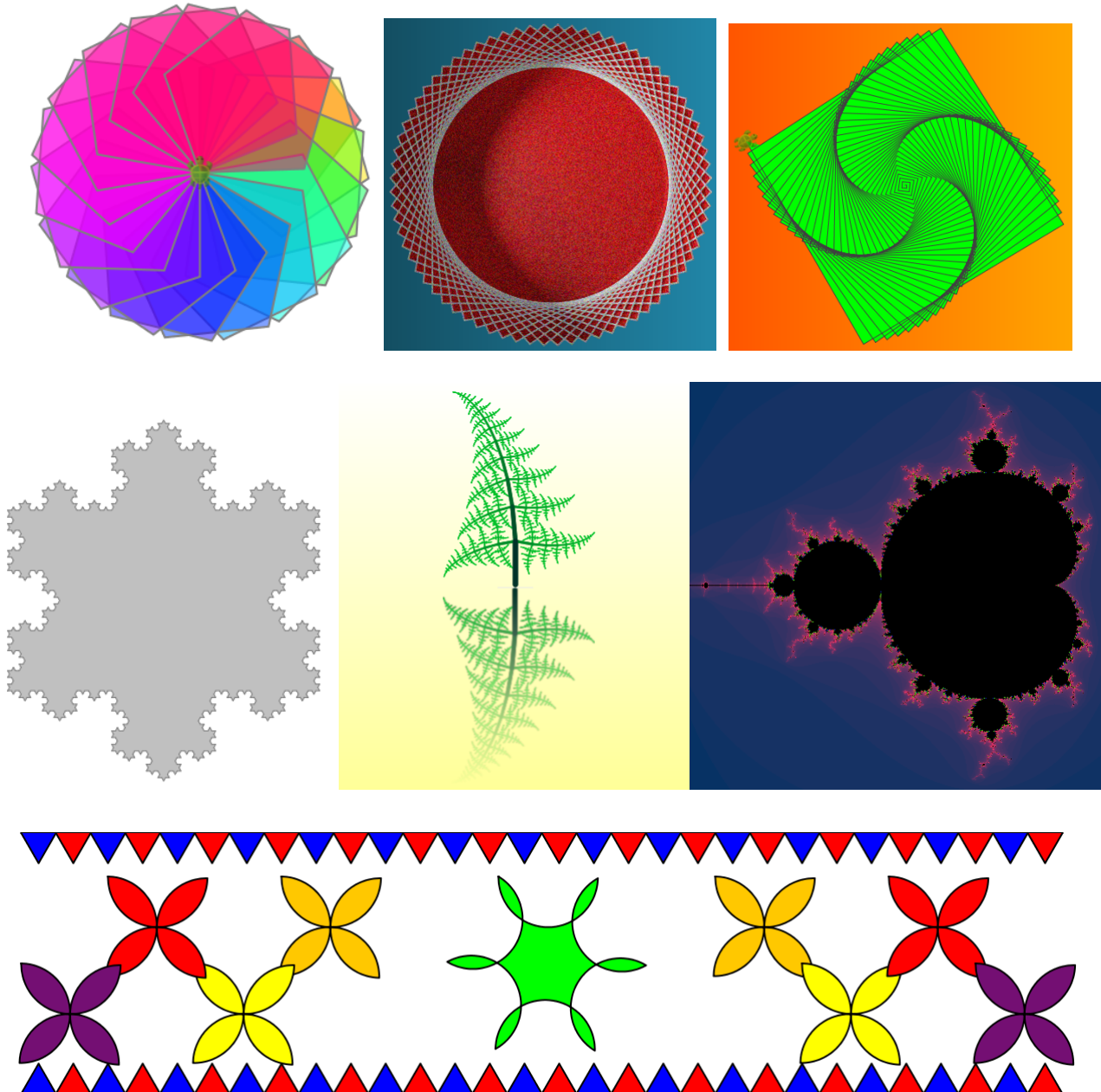
To the bottom-left of the Kojo workspace is the **script editor**, where you write your programs (or scripts, which are small programs). The script editor has a tool-bar with buttons



that quickly let you do most of the core things that you need to do within Kojo – run programs, trace programs, check your programs for errors, etc.

To the bottom-right is the **output pane**, where Kojo gives you informative messages to help you recover from mistakes that you make in your programs. You also write out results from your programs in this area.

To whet your appetite, here are some examples (taken from the Kojo *Samples* and *Show-case* menus) of drawings that you can create within Kojo:



With that brief introduction to Kojo, lets get going...

# 3 Commands and Programs

This activity involves the following:

- Learning about commands, actions, and programs.
- Learning to draw lines using the turtle.
- Learning the `clear`, `forward`, and `right` commands.
- Exploring the ideas of unit length, distances, and right angles, and using them to make a square geometrical figure.
- Using the Kojo error recovery feature.

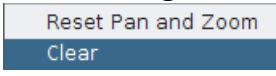

## Step 1

Type in the following code within the Script Editor and run it:

```
forward(100)
```

Q1a What does the turtle do? Does it move? By how much? In what direction?

## Step 2

Clear the line made on the Drawing Canvas in the previous step by right-clicking on the Canvas and pressing `Clear` . Then delete the text in the script editor by pressing the *Clear Editor* toolbar button . Now type in the following code and run it:

```
showAxes()  
forward(200)
```

Q2a What do you think the `showAxes` command does? Does it show you the unit of length (called a pixel) used for drawing on the turtle canvas?

**Q2b** What do you think the `forward` command does? What does the input to the command specify? The input to the `forward` command is the number that you write within round brackets after the command, e.g., `forward(100)` has 100 as its input.

Note – when you are asked to figure out what a command does, feel free to fire up a *Kojo Scratchpad* (using the File -> New Kojo Scratchpad menu item) to experiment with different inputs to the command. A *Kojo Scratchpad* is an instance of Kojo for doing “rough work” and figuring out things – as you work on an activity inside Kojo.

Note – The *Turtle Commands Quickref* chapter contains descriptions of commonly used turtle commands. You should go over to that chapter and validate your understanding of a command after you figure out what it does.

## Step 3

Clear the Drawing Canvas and Script Editor. Then type in the following code and run it:

```
right()
```

**Q3a** What do you think the `right` command does?

## Step 4

Clear the Script Editor (but *not* the drawing canvas). Then type in the following code and run it:

```
clear()
```

**Q4a** What does the `clear` command do?

## Step 5

Clear the Script Editor. Then type in the following code and run it. But first guess (before running the code) what figure is made by this program:

```
clear()
forward(100)
right()
forward(100)
right()
```

**Q5a** How is it useful to have the `clear` command as the first line of your program?

## Step 6

Clear the Script Editor. Then type in the following incorrect code and run it:

```
clear()  
forwardx(100)
```

**Q6a** What does Kojo tell you (in the output pane)? Observe the kind of message that Kojo shows you when you give it an incorrect command to run.

**Q6b** Using this message, can you determine (and go to) the line in your program that has the problem?

Hint – click on *Locate error in script* in the Output pane.

## Self Exploration

Play with the `clear`, `forward`, and `right` commands before you move on to the exercise. Deliberately make a few mistakes (misspelled commands, missing round-brackets) and then try to fix the mistakes with the help of the Kojo error messages.

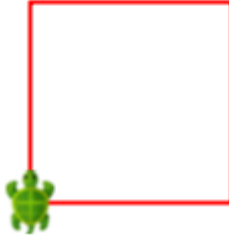
## Theory

- A program is a series of instructions for the computer.
- These instructions can be of a few different kinds. The first kind of instruction (the kind that you have seen in this activity) is a command. A command makes the computer carry out an action (like moving the turtle forward) or indirectly affects future actions (like setting the pen color).
  - Actions are effects produced by your program that you can see, hear, etc. They result in outputs from your program.
- Commands can take inputs. These inputs let you control what the command does, e.g., the input to the `forward` command lets you control the length of the line that the turtle draws. A command with no inputs always does exactly the same thing.

## Exercise

Write a program to make the following figure:

- Sides of the square –  $length = 100$  pixels.



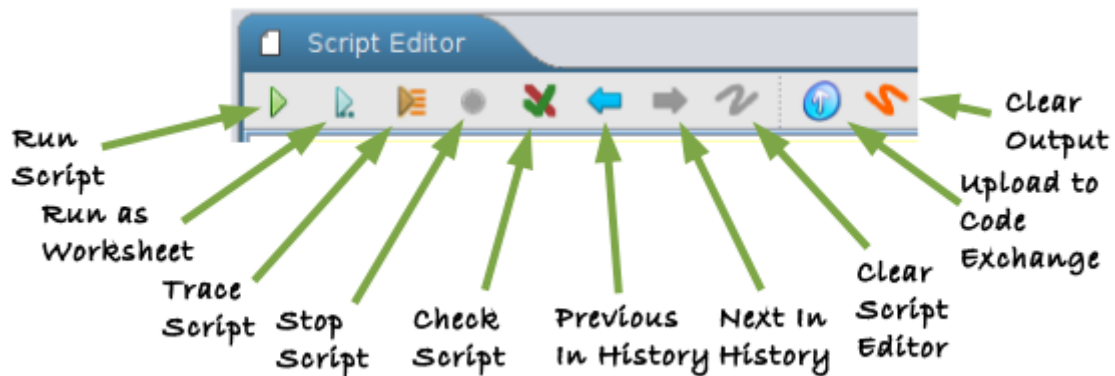
# 4 Using Kojo Effectively

This activity involves the following:

- Becoming familiar with the script editor toolbar.
- Exploring history navigation.
- Exploring code completion.
- Practicing the selection, copying, cutting, and pasting of program text.
- Practicing undo and redo of program text.
- Learning how to pan across the canvas and zoom in and out.
- Using fractions to determine lengths in a figure.

## Step 1

Take a quick look at the script editor toolbar to familiarize yourself with the buttons there:



Here's a brief description of what the buttons do:

- The *Run Script* button – runs your script/program, i.e., the contents of the script editor.
- The *Run as Worksheet* button – runs your script as a worksheet, to let you see expression types and values in-line, right within the Script Editor (don't worry if that does not make sense right now).
- The *Trace Script* button – traces your script, to let you see, line by line, what your program does as it runs.

- The *Stop Script* button – stops a running script. It also allows you to stop runaway scripts that are taking too long to finish.
- The *Check Script for Errors* button – helps you to precisely locate errors in large scripts.
- The *Previous in History* button – calls up, within the Script Editor, the last command/script that you ran.
- The *Next in History* button – along with the previous button, allows you to move back and forth within your command/script history.
- The *Clear Script Editor* button – clears the script editor, making it easy for you to start writing a new script.
- The *Upload to Code Exchange* button – uploads your code to the Kojo Code Exchange, to share it with people around the world.
- The *Clear Output* button – clears the output pane, making it easy for you to look at the output of scripts that you run after that.

## Step 2

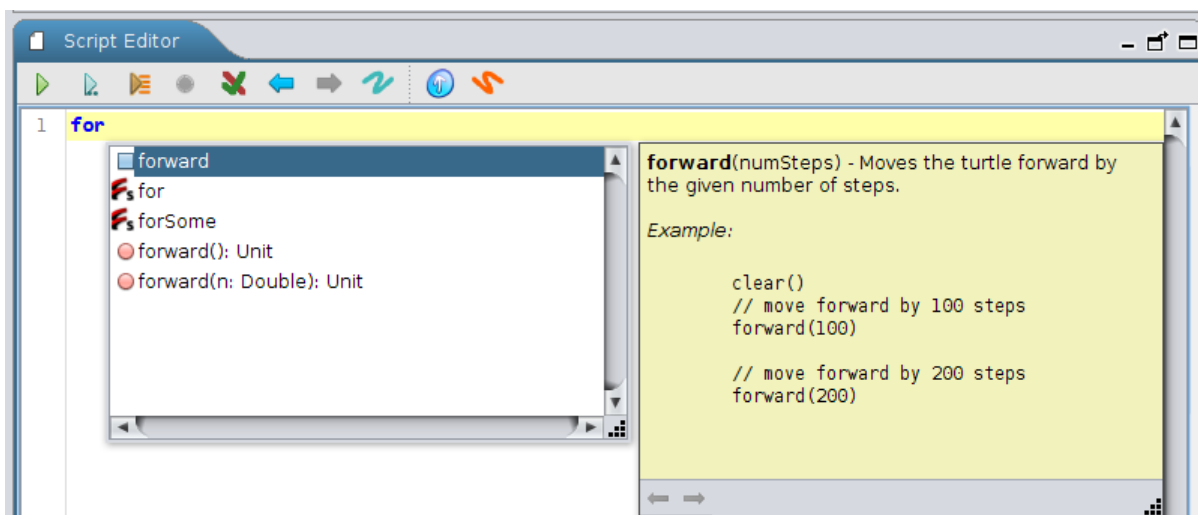
Explore history navigation – by using the History *Previous* and *Next* toolbar buttons .

Q2a What is program history?

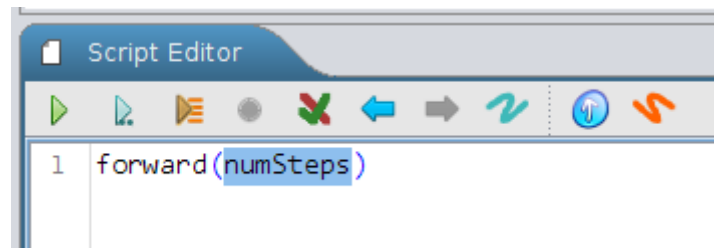
Q2b What do you think is the benefit of being able to navigate history?

## Step 3

Explore code completion. Type in `for` in the script editor and then type in `Ctrl+Space`. A window will pop up and show you all the instructions available within Kojo that start with `for`. The popup window will also show you online help for each instruction:




You can select an instruction from this list and double-click on it or hit *Enter* to have the instruction inserted into the script editor:



If the inserted instruction takes some inputs, your keyboard cursor will automatically be positioned in the input location for the instruction. You can then just type in the desired input, e.g., `100` if you want the turtle to move forward by 100 steps in the case shown above. If the selected instruction takes more than one input, you can move your cursor between these inputs using the *Tab* key.

Now, try code completion with `rig` and `cle` (just like you did it with `for`).

Note – code completion is not available while a program is running. If you don't want to wait for a longish program to finish (to use code completion again or to rerun a slightly modified version of the program), you can stop the currently running program by using the *Stop* toolbar button  (which turns red while a program is running).

**Q3a** How is code completion useful?

## Step 4

Practice the following in the script editor based on the square making program from the previous activity.

1. Text selection – bring your keyboard cursor to the beginning of the text that you want to select, press the Shift key, and then (while keeping the Shift key pressed) press the Arrow Keys to select text.
2. Copying (Ctrl+C) – Press the Control key, and then (while keeping the Control key pressed) press the C key to copy the currently selected text into the Clipboard.
3. Pasting (Ctrl+V) – Move the keyboard cursor to the location where you want to paste text, press the Control key, and then (while keeping the Control key pressed) press the V key to paste the text (from the Clipboard) at the current cursor location.
4. Cutting (Ctrl+X) – Press the Control key, and then (while keeping the Control key pressed) press the X key to cut the currently selected text into the Clipboard. You can now paste this text wherever you want.
5. Undo (Ctrl+Z) – Press the Control key, and then (while keeping the Control key pressed) press the Z key to remove the last piece of text that you added to your program.



6. Redo (Ctrl+Y) – Press the Control key, and then (while keeping the Control key pressed) press the Y key to redo (or roll back) the last undo that you did in your program.

Note – you can also use the mouse to do the above actions, in the following manner:

- Text Selection – drag the mouse from the beginning to the end of the text that you want to select.
- Copy, Paste, Cut, Undo, and Redo – use the script editor context menu, which can be brought up by right-clicking on the script editor.

Q4a How is copying/cutting and pasting text useful?

Q4b How is the undo/redo feature useful?

## Step 5

Play with panning across the drawing canvas, and zooming in and out. Pan by dragging the canvas. Zoom in/out by using the mouse scroll wheel. You can reset pan and zoom level by right clicking on the canvas and clicking *Reset Pan and Zoom*.

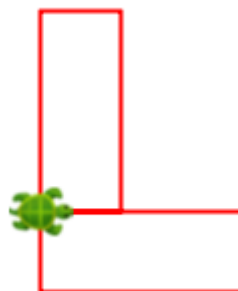
## Self Exploration

Play with the above features before you move on to the exercise.

## Exercise

Write a program to make the following figure. Use copy-and-paste and code-completion along the way:

- The dimensions of the two rectangles are:
  - *length* = 120 pixels
  - *breadth* =  $\frac{1}{3}$  of *length*



# 5 Drawing with Colors

This activity involves the following:

- Learning to set the background color of the canvas.
- Learning to change the color and thickness of the lines drawn by the turtle.
- Learning to fill the shapes drawn by the turtle with color.
- Learning the `setBackground`, `setPenColor`, `setFillColor`, `setPenThickness`, and `left` commands.
- Applying the idea of ratio and proportion in constructing figures.

## Step 1

Type in the following code and run it (use copy-and-paste and code-completion along the way):

```
clear()
setBackground(yellow)
setPenColor(blue)
setFillColor(green)
forward(100)
left()
setPenThickness(5)
forward(50)
right()
forward(50)
right()
forward(100)
right()
forward(150)
right()
forward(50)
```

**Q1a** What do you think the `setBackground` command does? What does the input to the command specify?

**Q1b** What do you think the `setPenColor` command does? What does the input to the command specify?

**Q1c** What do you think the `setFillColor` command does? What does the input to the command specify?

**Q1d** What do you think the `setPenThickness` command does? What does the input to the command specify?

**Q1e** What do you think the `left` command does?

## Self Exploration

Play with the inputs to the `setBackground`, `setPenColor`, `setFillColor`, `setPenThickness`, and `forward` commands in the code above. See how changing these inputs modifies the figure. The predefined colors in Kojo are: `black`, `blue`, `brown`, `cyan`, `darkGray`, `gray`, `green`, `lightGray`, `magenta`, `orange`, `pink`, `purple`, `red`, `white`, and `yellow`. If you don't want any pen or fill color, you can use a special color called `noColor`.

## Theory

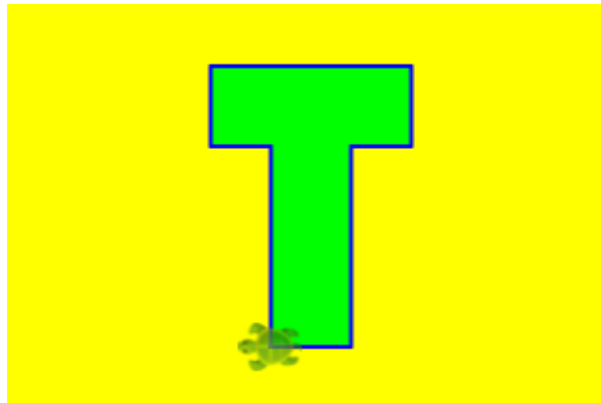
### Math Recap – Ratios and Proportions

- A ratio is a relationship between two quantities. It lets you compare the sizes of the two quantities. For example, the ratio of apples to oranges in a basket might be 3 : 4. That tells you that if the basket has 3 apples, it has 4 oranges. Or if it has 6 apples, it has 8 oranges, etc.
- A ratio can be written as a fraction. You can say that the basket has  $\frac{3}{4}$  as many apples as oranges.
- A ratio also specifies the proportions of the different elements within the ratio. In the fruit basket with a 3 : 4 ratio of apples to oranges,  $\frac{3}{7}$  is the proportion of the apples in the basket, while  $\frac{4}{7}$  is the proportion of the oranges in the basket.
- If two ratios are in proportion, they are equal.

## Exercise

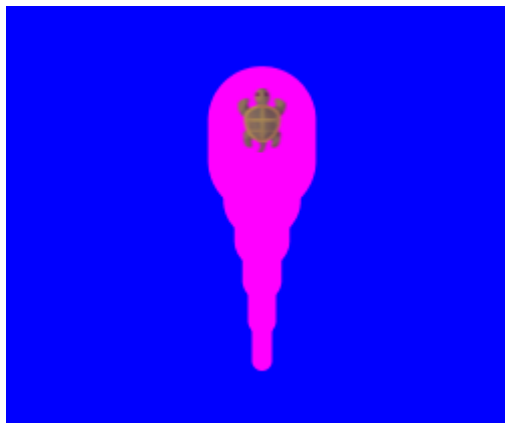
1. Write a program to make the following figure:

- The pen color is blue, and the fill color is green. The background color is yellow.
- To determine the dimensions of the figure, imagine that it is made out of two rectangles – a vertical one and a horizontal one, with the following specifications:
  - Vertical rectangle –  $length = 120$  pixels, ratio of  $breadth : length = 1 : 3$
  - Horizontal rectangle –  $length = 90$  pixels, ratio of  $breadth : length$  is in the same proportion as the corresponding ratio for the vertical rectangle.



2. Write a program to make the following figure:

- The figure is made up of 6 line segments (count them).
- Each segment is 20 pixels long, and has a *magenta* pen color. The background color is blue.
- The thickness of the first segment is 10 pixels. The ratios of the thickness of any segment and its next segment are all in proportion. The third segment is 19.6 pixels thick. Determine the thickness of each of the line segments to make the figure.



# 6 Mixing Colors With a Function

This activity involves the following:

- Learning about functions.
- Learning to use the `Color` function to create new colors.
- Learning about the RGB color model used to represent colors in computers.
- Applying the idea of ratios to determine sizes and colors in a given figure.

## Step 1

Type in the following code and run it (note that just the third line in this code is different from the code in Step 1 of the previous activity. So you can pull up that code in your history, and modify just the third line):

```
clear()
setPenColor(blue)
setFillColor(Color(255, 0, 0, 255))
forward(100)
left()
forward(50)
right()
forward(50)
right()
forward(100)
```

**Q1a** What do you think the `Color` function does?

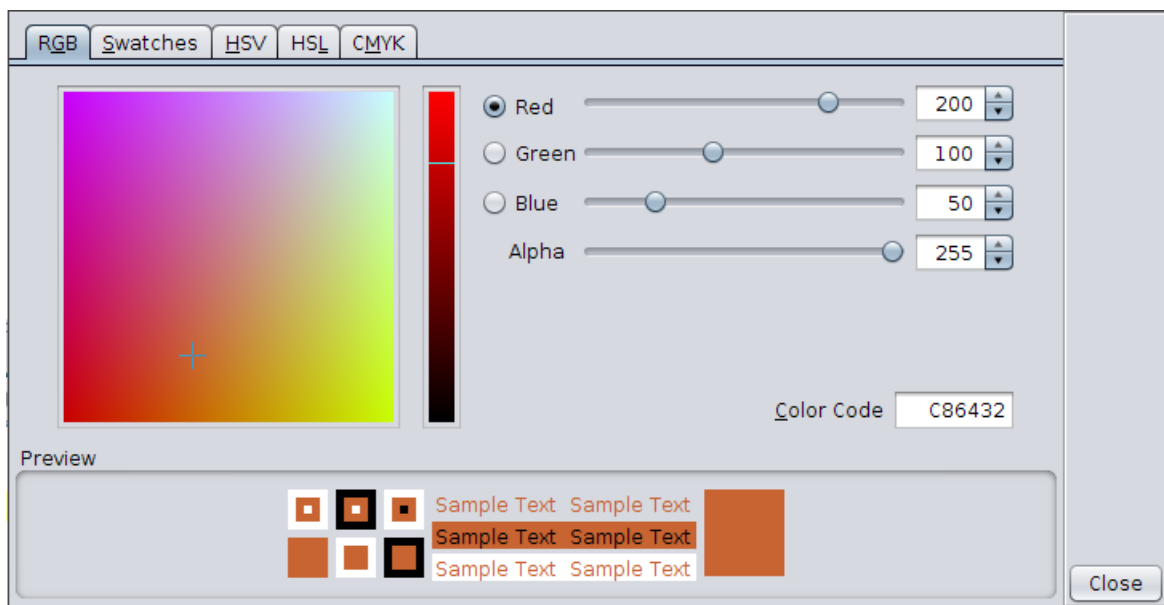
**Q1b** What do the four different inputs to the `Color` function specify? Play with the following fill colors to try to answer this (by replacing the `setFillColor` line of the program above with following lines, one at a time):

1. `setFillColor(Color(0, 255, 0, 255))`
2. `setFillColor(Color(0, 0, 255, 255))`
3. `setFillColor(Color(100, 0, 0, 255))`
4. `setFillColor(Color(0, 100, 0, 255))`

5. `setFillColor(Color(0, 0, 100, 255))`
6. `setFillColor(Color(0, 0, 0, 255))`
7. `setFillColor(Color(0, 0, 0, 150))`
8. `setFillColor(Color(0, 0, 0, 50))`

## Self Exploration

Click on the word `Color` in the `setFillColor` line inside the Script Editor. This will bring up a color chooser (shown below). You can then interactively play with the fill color for the figure.



## Theory

- Before this activity, the programs that you wrote involved using commands with number values as inputs. Let's expand on the definition of a program.
- A program contains a series of instructions. These instructions can be of a few different kinds:
  - The first kind of instruction that you have seen is a command. A command makes the computer carry out an action (e.g. moving the turtle forward) or affects a future action (e.g. setting the turtle pen color). It is said that a command has a side-effect.
  - The second kind of instruction that you have seen (which is the focus of this activity) is a function. A function takes some values as inputs and computes and returns an output value based on the inputs, e.g., `Color(200, 100, 50, 255)` takes four number values as inputs and returns a `Color` value as an output.
- Let's dig into the `Color` function; this function takes four inputs – the red, green, blue, and alpha/opacity components of the color you want to create. Based on these inputs, the `Color` function creates a `Color` value using the RGB+alpha model.
  - The RGB model represents colors in the computer using three values – the red component, the green component, and the blue component. Pretty much all possible colors can be created using a mixture of these three components. The component values need to be in the range 0-255.
  - The Alpha/opacity value allows you to control the transparency of the colors you create; it is the opposite of transparency. An opacity of 0 means perfectly transparent. An opacity of 255 means perfectly opaque. Opacity values between 0 and 255 represent partially transparent colors.

## Exercise

1. Write a program to make the following figure:

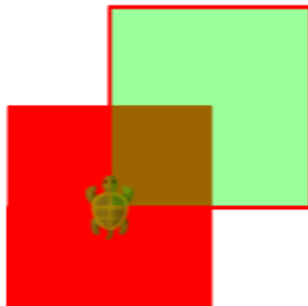
- Each arm of the cross – *length* = 50 pixels, ratio of *breadth* : *length* is 3 : 5.
- Fill color – the ratio of the *red* : *green* : *blue* : *opacity* color components is 1 : 2 : 3 : 4 and the blue component is 150.



2. Write a program to make the following figure:

- Both squares – *length* = 100 pixels.
- Back Square – fill color is *red*.
- Front Square – fill color is *green*, with an opacity of 100.

Notice how a portion of the *red* square is visible behind the transparent *green* square, and how *red* viewed through a transparent *green* looks *brown*.





# 7 Hopping with Speed

This activity involves the following:

- Learning how to control the speed of the turtle.
- Learning how to make the turtle move without drawing lines.
- Learning to hide the turtle.
- Learning the `setAnimationDelay`, `hop`, and `invisible` commands.
- Learning to estimate the dimensions of a figure given the size of one line in the figure.

## Step 1

Type in the following code and run it:

```
clear()
setAnimationDelay(100)
setPenThickness(5)
forward(20)
hop(20)
forward(20)
hop(20)
forward(20)
hop(-100)
right()
hop(20)
forward(20)
hop(20)
forward(20)
invisible()
```

**Q1a** What do you think the `setAnimationDelay` command does? What does the input to the command specify? Play with the following animation delays to try to answer this:

1. `setAnimationDelay(100)`
2. `setAnimationDelay(0)`
3. `setAnimationDelay(1000)`

Q1b What do you think the `hop` command does? What does the input to the command specify?

Q1c What do you think the `invisible` command does?

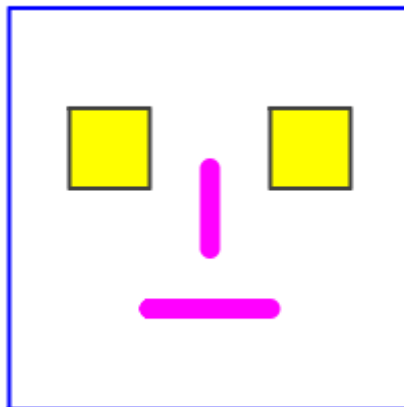
## Self Exploration

Play with the inputs to the `setAnimationDelay`, `setPenThickness` and `hop` commands in the code above. See how changing these inputs modifies the figure and the speed with which the figure is made.

## Exercise

Write a program to make the following figure:

- The outline of the face is a square with  $length = 200$  pixels.
- Use your best judgment to estimate the other dimensions and colors in the figure.



## 8 Digging Deeper with Tracing

This activity involves the following:

- Gaining a deeper understanding of how programs run.
- Learning to determine what portion of a drawing is made by which line in your program.
- Using symmetry and arithmetic to determine the dimensions of a figure.


Look at the following program:

```
clear()  
forward(100)  
right(90)  
forward(50)
```



Figure 8.1: Going forward and right

Do you understand how the program above makes the corresponding figure?

A good way to see how a program does what it does is to trace it, by clicking on the *Trace* button  in the script editor tool-bar. This opens up a program trace window, which contains a step-by-step view (also called a trace) of the running of the program. You can click on any line in the trace to see the corresponding source-code line in the script editor, and the artifact in the drawing canvas that corresponds to that line. Figure 8.2 on the following page shows you how this looks for the program in Figure 8.1.

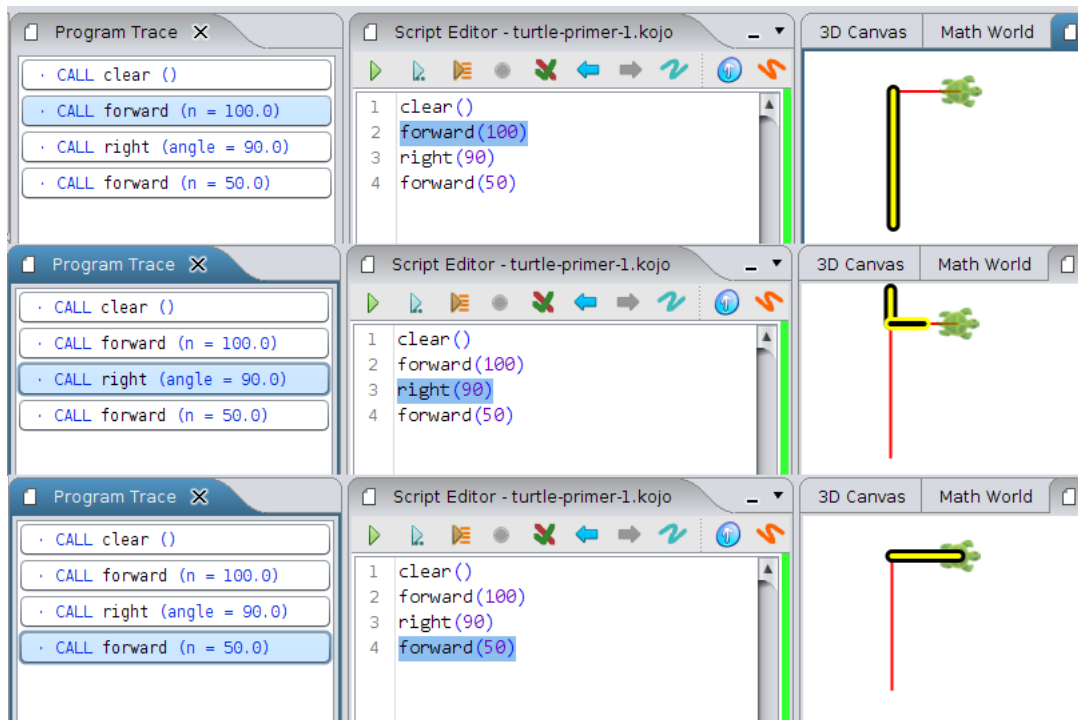


Figure 8.2: Output from tracing the program in Figure 8.1 on the previous page

## Step 1

Type in the following code and run it:

```
clear()
forward(100)
right(90)
forward(50)
right(90)
forward(40)
right(90)
forward(60)
```

Now trace the program by clicking on the *Trace* button .

**Q1a** What's the difference between running and tracing a program?

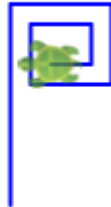
## Self Exploration

Explore the trace of the program from the previous step. Click on different lines of the program trace, and see how the corresponding source line (in the script editor) and portion of the drawing (in the drawing canvas) are highlighted.

## Exercise

Write a program to make the following figure:

- The size of the longest vertical line is 100 pixels.
- Use tracing as much as you can while you write your program to try to understand what the program does as it runs.



# 9 Angles

This activity involves the following:

- Learning how to make the turtle turn through angles other than  $90^\circ$ .
- Exploring angles, and the idea of interior and exterior angles of a triangle.
- Using the idea of supplementary angles.

## Step 1

Type in the following code and run it:

```
clear()
showProtractor()
forward(100)
right(120)
forward(100)
right(120)
forward(100)
right(120)
```

**Q1a** What do you think the `showProtractor` command does?

Hint – look at the bottom-left of the canvas.

**Q1b** How is `showProtractor` command useful? Use the protractor to measure all the angles in the figure before you try to answer that.

Note – You can drag the Protractor to move it around, and Shift-drag to rotate it. And measuring angles is much easier if you zoom in and make the drawing larger.

**Q1c** What do you think the input to the `right` command above specifies?

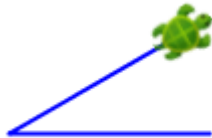
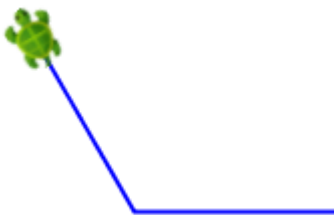
**Q1d** The angles of an equilateral triangle are  $60^\circ$ . Why does the turtle turn through  $120^\circ$  to make the above equilateral triangle? How does that relate to the idea of the interior and exterior angles of a triangle?

## Self Exploration

Play with the inputs to the `right` command in the code above. See how changing these inputs modifies the figure.

## Exercise

Write programs to make the following figures (without the written angle sizes):

 $30^\circ$  $60^\circ$  $90^\circ$  $120^\circ$  $180^\circ$ 

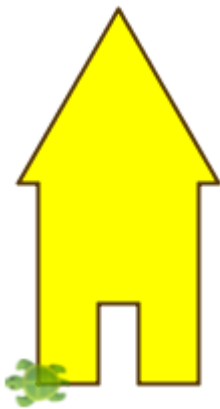
Hint – use the idea of supplementary angles.

# 10 Practice

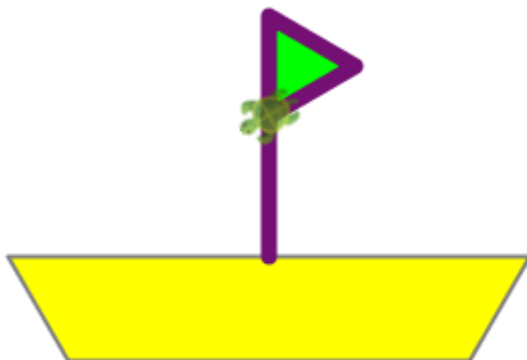
Write programs to make the following figures:



Make creative use of hopping and pen-thickness to make the figure. Use your own estimates for lengths.



The top of the house is an equilateral triangle with sides of 100 pixels.



The bottom of the boat is 200 pixels long.




# 11 Calculations

This activity involves the following:

- Learning to do calculations within Kojo.
- Learning about expressions.
- Exploring operator precedence and associativity (BODMAS) in expressions.
- Learning about some predefined Math functions in Kojo.

## Step 1

Type in the following code and run it using the *Run as Worksheet* button :

```
10 + 2
10 * 2
10 - 2
10 / 2
```

Notice that Kojo shows you the result value and type of each expression on the same line as the expression, after a sequence of separator characters, namely `//>`. This is a consequence of running the program using the *Run as Worksheet* button. When a program is run as a worksheet, the result of each expression within the program is shown at the end of the line containing the expression.

Note – the `//` character sequence has a very specific meaning for Kojo – it is treated as the beginning of a single line comment. A comment in a program is a piece of text that is ignored by Kojo and meant only for human communication. You write a comment in a program so that you or another person can read it later – to better understand the program. The *Run as Worksheet* feature within Kojo makes use of a comment to show you the result of a function.

**Q1a** How can you do the operations of addition, subtraction, multiplication, and division of numbers within Kojo?

## Step 2

Type in the following code and run it using the *Run as Worksheet* button:

```
10 + 2 * 4
(10 + 2) * 4
```

**Q2a** Can you combine multiple operations on numbers within a single expression? If so, in what order are the different operations carried out?

**Q2b** Can you change the default order in which operations are carried out?

### Step 3

Type in the following code and run it using the *Run as Worksheet* button:

```
kmath.hcf(24, 18)
kmath.lcm(24, 18)
math.sqrt(2)
```

**Q3a** Do you recognize the three functions used in the code above (they are common math operations)?

Note – this is the first time that you are seeing the dot notation (e.g., `kmath.lcm(24, 18)`) for calling a function. A call like `kmath.lcm(24, 18)` tells you that the `lcm` function lives inside the `kmath` object. To see all the functions available in the `kmath` object, type in `kmath.` and press *Ctrl+Space*.

### Self Exploration

Play with doing different kinds of calculations.

## Theory

### Expressions

- Let's review the definition of a program. A program contains a series of instructions.
- These instructions can be of a few different kinds:
  - The first kind of instruction that you saw was a command. A command makes the computer carry out an action (e.g., moving the turtle forward) or affects a future action (e.g., setting the turtle pen color). It is said that a command has a side-effect.
  - The second kind of instruction that you saw was a function (that was the function for color mixing – `Color(red, green, blue, alpha)`). A function takes some values as inputs and computes and returns an output value based on the inputs.
- In this activity, you have worked with arithmetic operators like `+`, `-`, `*`, and `/`. These are also functions, but they are used with infix notation (e.g., `1 + 2`) as opposed to prefix notation (e.g., `+(1, 2)`).
- Functions belong to a category of instructions called expressions. Most expressions are functions. The ones that are not (e.g., a number like `2`) are called literals and evaluate to themselves (i.e., the text `2` in your program evaluates to the number `2` when the program runs). In other words, expressions are either functions or literals.

### Types

- Let's dig deeper into the results of expression evaluation via the *Run as Worksheet* button. Here's an example:

```
kmath.lcm(24, 18) //> res3: Int = 72
math.sqrt(2) //> res4: Double = 1.4142135623730951
```

- Notice that every result has this structure – `name: type = value`
- The name is a temporary name assigned by Kojo to the result; the value is the final result of evaluating the expression; the type specifies the set of values that the expression can produce, e.g., `kmath.lcm` can produce Integer results only (belonging to the set of integers), while `math.sqrt` can produce Double results only (belonging to the set of rational numbers written as decimal fractions).
- In general, the type of a value defines:
  - the set of values that it belongs to.
  - the functions and commands that the value can work with.

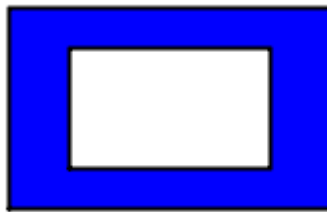
More on this later.

## Exercise

1. Write a program to make the following figure.
2. Calculate the area of the figure with the help of the calculation capability in Kojo (using the *Run as Worksheet* feature).

The dimensions of the two rectangles in the figure are:

- Outer Rectangle –  $length = 160$  pixels,  $breadth = 100$  pixels
- Inner Rectangle –  $length = 100$  pixels,  $breadth = 60$  pixels



# 12 Repeating Commands

This activity involves the following:

- Learning the repeat command.
- Learning about removing code duplication/repetition by using the repeat command.
- Exploring arcs of circles.

## Step 1

Type in the following code and run it:

```
clear()
repeat (2) {
  forward(100)
  right(90)
}
```

**Q1a** What do you think the repeat command does? Make use of tracing to help you understand the repeat command.

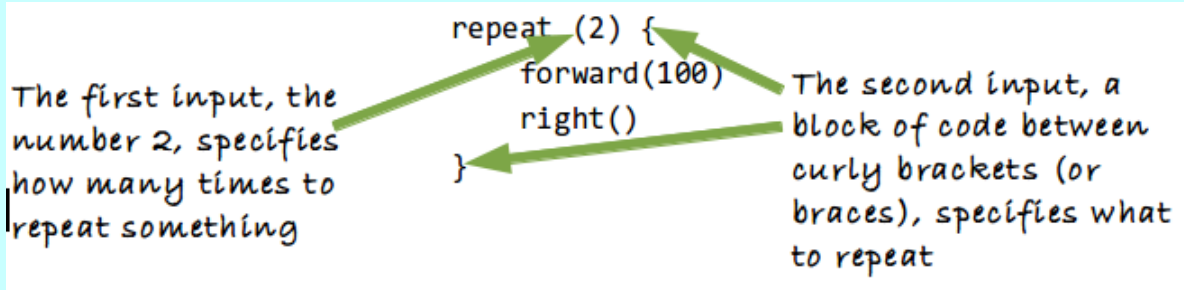
**Q1b** How many inputs does the repeat command take? What do these inputs signify?

## Self Exploration

Play with the inputs to the repeat, forward, and right commands in the code above. See how changing these inputs modifies the figure.

## Theory

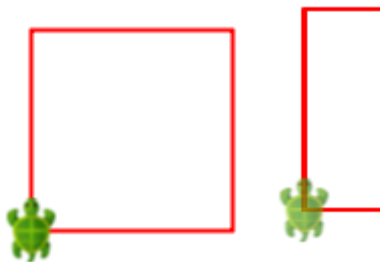
- The repeat command allows you to run other commands for a specified number of times. Note that repeat takes two inputs:



- The repeat command is also known as the repeat loop – because the program loops inside the repeat block for the specified number of times (you can verify this via tracing).
- The repeat command has a couple of big benefits:
  - It makes your programs shorter by removing repetition.
  - It makes your programs easier to understand.

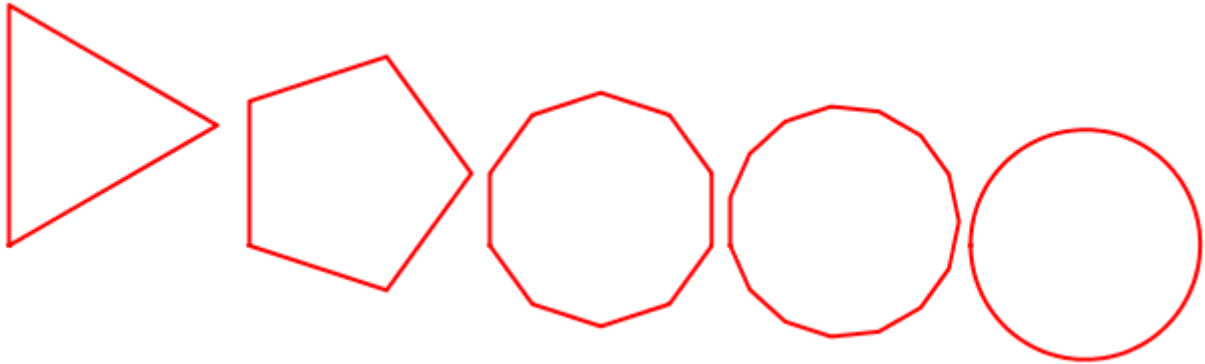
## Exercise

Write programs, using the repeat command, to make the following previously made figures:



## 13 Practice with repeat

Write programs to make the following figures:



Something to think about – the last shape in the sequence of shapes above is a circle? How can you make a circle using a command (forward) that lets you make just straight lines?

# 14 Turning with a radius

This activity involves the following:

- Learning how to make the turtle turn along an arc of a circle.
- Learning about the relationship between angles and the arc of a circle.
- Explorations with geometry and angles.

## Step 1

Type in the following code and run it:

```
clear()
right(150, 50)
```

**Q1a** What do you think the two inputs to the `right` command specify?

**Q1b** Can you identify the circle along which the turtle moves in response to the `right` command above. Where is the center of this circle? What is its radius?

## Step 2

Type in the following code and run it:

```
clear()
setAnimationDelay(100)
setBackground(Color(0, 128, 215))
setPenColor(black)
right(150, 50)
right(90)
right(150, 50)
right(90)
right(150, 50)
```

Make sure you understand how the program generates its output drawing. Tracing can help you with this.

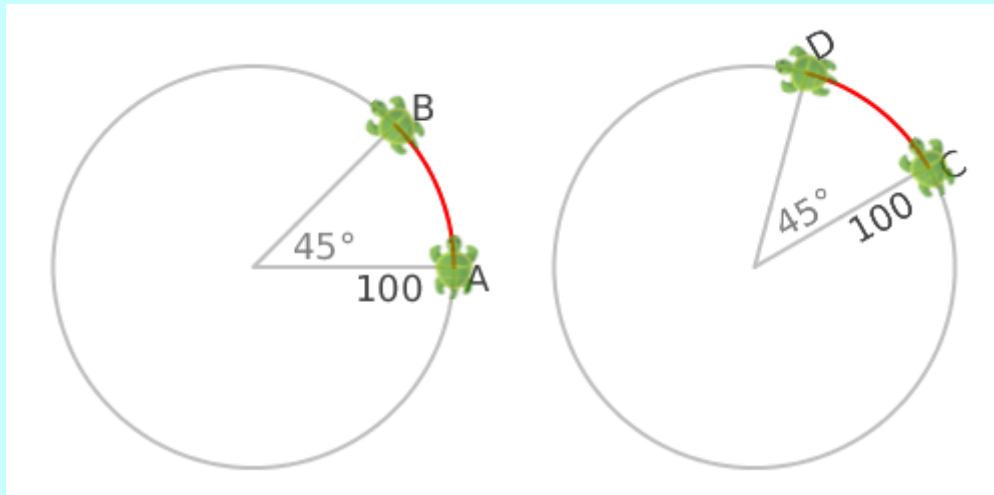


## Self Exploration

Play with the inputs to the `right` command in the code above. See how changing these inputs modifies the figure.

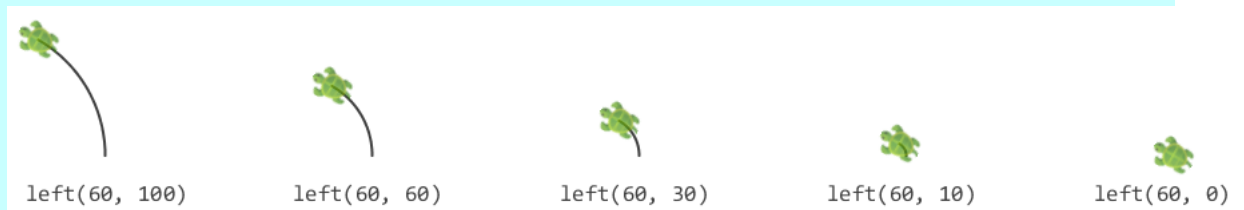
## Theory

- Let's dig deeper with an example – `left(45, 100)`. Take a look at the following figure:



If the turtle is at position A, `left(45, 100)` will move it to position B, along the arc of the shown imaginary circle. The angle moved is 45°, and the radius of the circle is 100. Similarly, if the turtle is at position C, it will move to position D.

- Now take a look at some other examples:



- In each of the scenarios shown above, the turtle moves along the arc of an imaginary circle (which is not shown any more) and turns through an angle of 60 degrees as a result of the `left(60, radius)` command.
  - For the first scenario, the turtle turns along an arc of a circle with a radius of 100. This results in a change of position and direction for the turtle.
  - For the second scenario, the turning radius is 60. The change in direction is the same as in the first scenario, but the change in position is less.
  - This pattern is repeated for the remaining figures.
  - In the last scenario (where the turning radius is zero), the change in direction is the same as the change in direction for all the other scenarios, but there is no change in position. So, a turn with a zero turning radius results in only a change in direction. For this kind of a turn, you can use the `left(angle)` command instead of the `left(angle, radius)` command, i.e., `left(60)` instead of `left(60, 0)`.
- The `right(angle, radius)` command works similar to the `left(angle, radius)` command, except that the turtle turns clockwise instead of anti-clockwise.

### Math Recap – Angles

- The idea of an angle is closely related to the ideas of *a change in direction* and the *arc of a circle*. An angle is defined to be *the length of the arc of a circle*. A couple of different units of length are used to measure angles:
  - degrees, where 1 degree is  $\frac{1}{360}$  of the circumference of a circle.
  - radians, where 1 radian is equal to the radius of a circle.

Note – The circle used to specify an angle can be any circle; all circles are similar, and the ratio of an arc (specifying an angle on a particular circle) to the radius and circumference of the circle is the same for all circles. Look at Samples -> Math Learning Modules -> Angles within Kojo for more information on this.

## Exercise

Write a program to make the following figure:



Hint – use four turns of  $180^\circ$  to make the figure.

Next, write a program to make the following figure (a flock of birds):



# 15 More Fun with Repeat

This activity involves the following:

- Learning to use repeat to make intricate figures.
- Learning the significance of  $360^\circ$  in making closed figures.
- Applying the idea of a lowest common multiple (LCM) to make interesting figures.

## Step 1

Type in the following code and run it:

```
clear()
setAnimationDelay(10)
setPenColor(blue)
repeat(9) { // repeat count is 9
    forward(100)
    right(80) // turn angle is 80 degrees
}
```

## Step 2

Type in the following code and run it:

```
clear()
setAnimationDelay(10)
setPenColor(blue)
repeat(9) { // repeat count is 9
    forward(100)
    right(85) // turn angle is 85 degrees
}
```

**Q2a** The figure in Step 1 is closed, while the figure in Step 2 is not closed. Why? Try to explain this in terms of the total angle turned by the turtle, which is equal to the repeat count multiplied by the turn angle.

## Self Exploration

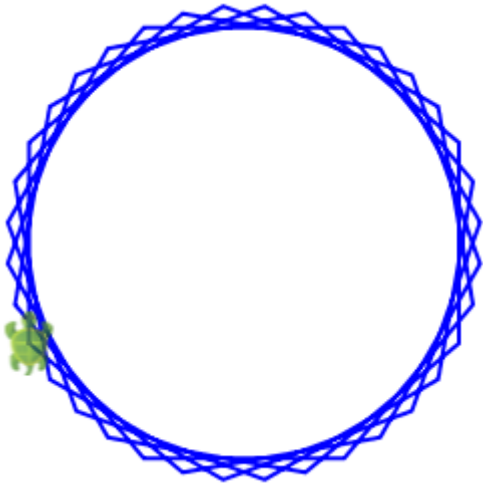
Play with the code in Steps 1 and 2. Try to determine what combinations of repeat count and turn angle make closed figures, and what combinations make open figures.

## Theory

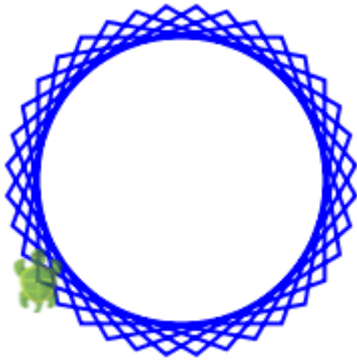
- To make a closed regular (i.e., equal-sided) figure, you need to turn the turtle through an angle of  $360^\circ$  or a multiple of  $360^\circ$  (why?).
- With repeat loops like the ones in steps 1 and 2, the total angle turned is  $repeatCount \times turnAngle$
- So this should hold:  $repeatCount \times turnAngle = n \times 360$
- Here  $n \times 360$  is a multiple of both  $turnAngle$  and 360. We can let it be the LCM of  $turnAngle$  and 360 (that will give us the smallest value of  $repeatCount$ ).
- So  $repeatCount \times turnAngle = lcm(turnAngle, 360)$
- In other words,  $repeatCount = \frac{lcm(turnAngle, 360)}{turnAngle}$
- For example, if the turn angle is  $85^\circ$ , you can calculate the least number of turns to make a closed figure in the following manner:
  - Determine the LCM of 85 and 360.
  - Divide this LCM by 85 to get the required repeat count.

## Exercise

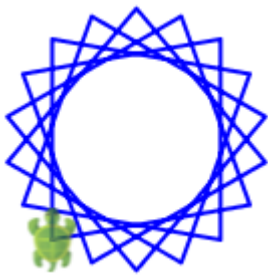
Write programs to make the following figures:



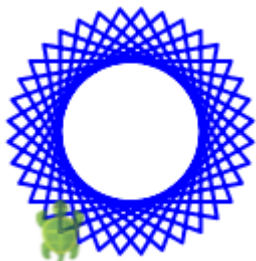
Turn angle =  $50^\circ$



Turn angle =  $70^\circ$



Turn angle =  $100^\circ$



Turn angle =  $110^\circ$

# 16 Repeat with a sequence

This activity involves the following:

- Learning to repeat things based on a sequence.
- Learning to do *similar but not exactly the same* things with repetition.
- Using an equation to determine the lengths in a figure.
- Becoming familiar with an arithmetic sequence.

## Step 1

Type in the following code and run it:

```
clear()
setAnimationDelay(100)
repeatFor(1 to 3) { counter =>
  repeat(4) {
    forward(50 + 50 * counter)
    right()
  }
}
```

**Q1a** What do you think the `repeatFor` command does?

**Q1b** What do you think the expression `1 to 3` evaluates to? Is this a sequence?

**Q1c** The `repeat` command lets you do the *same* thing multiple times. How does the `repeatFor` command let you do *similar but not exactly the same* things – like making squares of different sizes? What role does the `counter` play in this in the code above?

## Self Exploration

Play with the code above as you see fit.

## Theory

- The `repeatFor` command allows you to do *similar but not exactly the same* things.
- How does it do that?

By letting you give it a sequence of values to repeat over, and then giving you a repeat counter that ranges over that sequence, one item per repetition.

- Let's understand this with the help of the code in step 1:

```
repeatFor(1 to 3) { counter =>
  repeat(4) {
    forward(50 + 50 * counter)
    right()
  }
}
```

- Here, you give `repeatFor` a sequence of values (1 to 3), and give it a chunk of code to be repeated. The input sequence has 3 values (1, 2, and 3) so your repeat-code gets called three times. The first time, `counter` is 1. The second time, `counter` is 2. And the third time, `counter` is 3. If the input sequence had been 3 to 7, the values of `counter` would have been 3, 4, 5, 6, and 7.

Does that make sense?

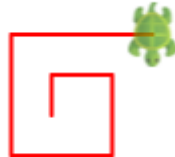
- Also, you can name the counter whatever you want. Here we called it `counter`, but we could have called it `i` or `e` or `idx` or whatever.
- So how does `repeatFor` allow you to do *similar but not exactly the same* things?  
By giving you the counter to work with. Within your repeat-code, you can use the counter to tweak what you do (just like the code in step 1 uses the counter to determine the sizes of the squares that it makes).



## Exercise

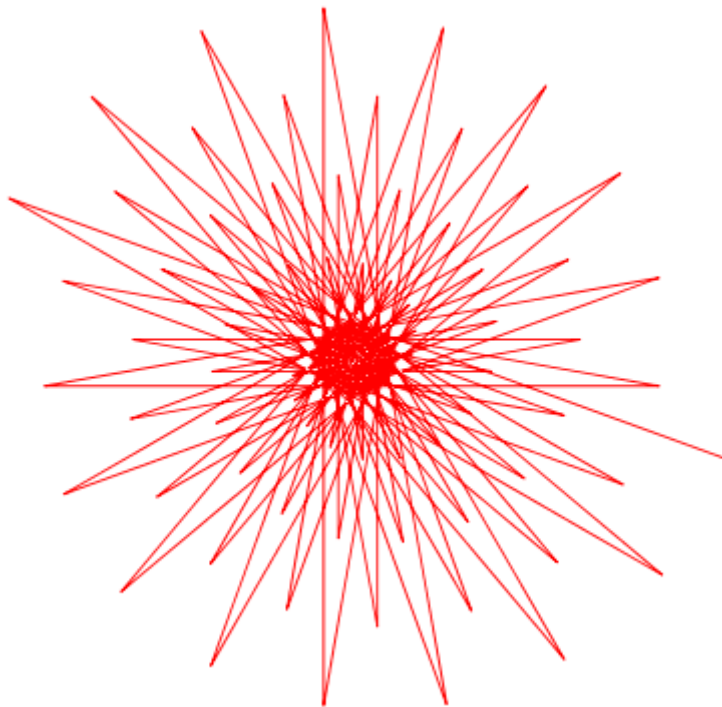
1. Write a program to make the following figure:

- The figure has six lines.
- Each line is 10 pixels longer than the previous line.
- The sum of the lengths of the lines is 270.



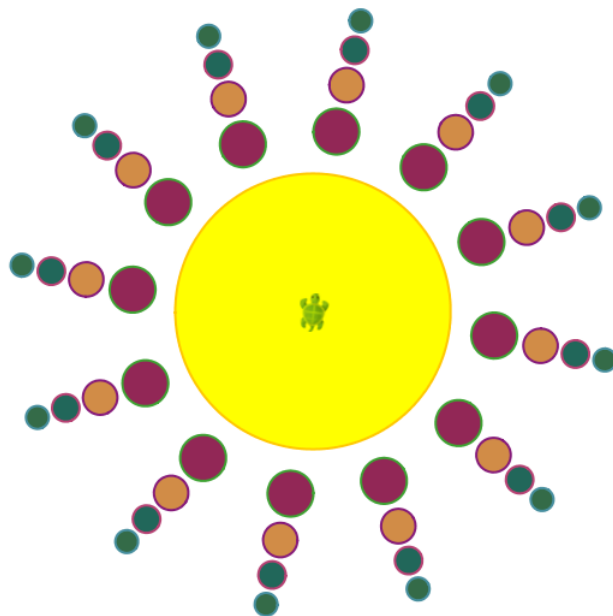
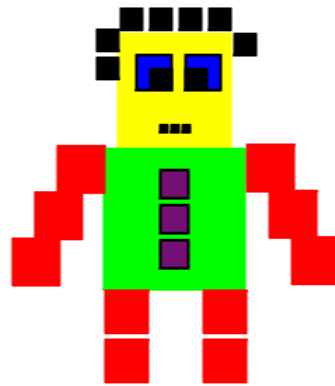
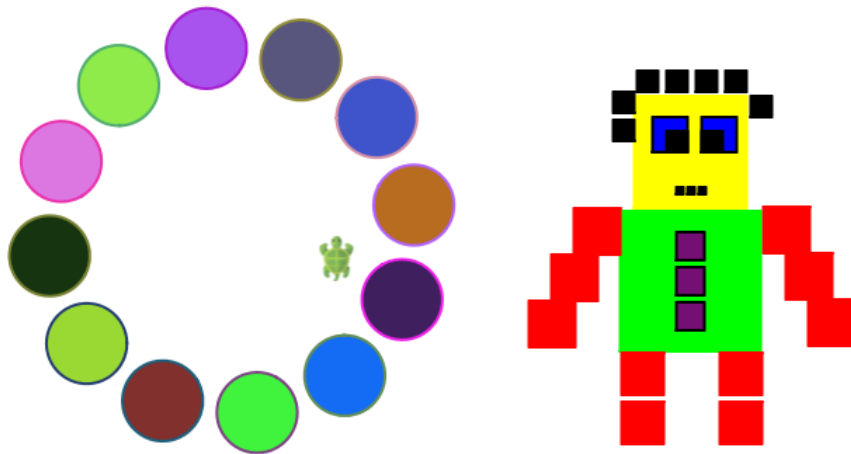
Hint – write an equation and solve it to determine the line sizes. Then use a `repeatFor` loop to make the lines.

2. Take the program that makes the above figure and make slight changes in it (to the number of repetitions and the turning angle) to get it to make the following figure:



# 17 Art Breakout

Write programs to make the following figures:



# 18 Absolute position and heading

This activity involves the following:

- Learning to place the turtle at an absolute position on the canvas.
- Learning to point the turtle along an absolute heading/direction on the canvas.
- Getting introduced to coordinate geometry.
- Learning the `showAxes`, `showGrid`, `setPosition` and `setHeading` commands.

## Step 1

Type in the following code and run it:

```
clear()
showAxes()
showGrid()
setAnimationDelay(100)
setPenColor(Color(0, 100, 200))
setPosition(50, 50)
setHeading(0)
left(90, 100)
left(90)
left(90, 100)
setPosition(-50, 50)
setHeading(90)
left(90, 100)
left(90)
left(90, 100)
setPosition(-50, -50)
setHeading(180)
left(90, 100)
left(90)
left(90, 100)
setPosition(50, -50)
setHeading(270)
left(90, 100)
left(90)
left(90, 100)
```

Q1a What do you think the `showAxes` and `showGrid` commands do?

Q1b What do you think the `setPosition` command does? What do the inputs to the command specify?

Q1c What do you think the `setHeading` command does? What does the input to the command specify?

## Self Exploration

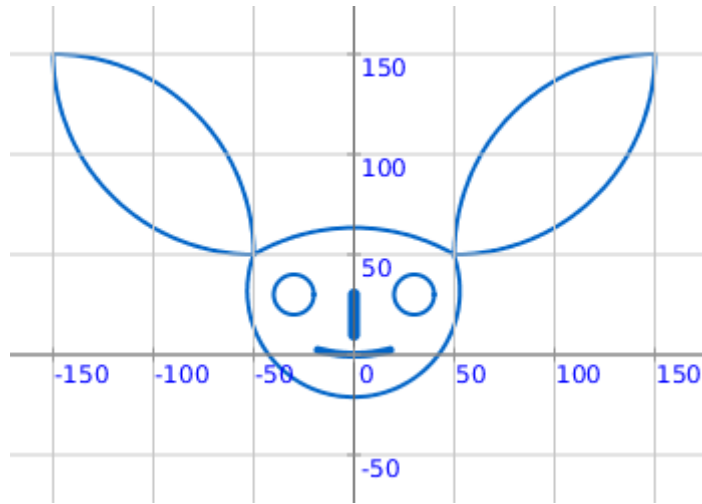
Play with the code above as you see fit.

## Theory

- Before doing this activity, you never really worried about the turtle's position or heading. You just asked it to move forward a certain number of steps or turn right or left by a certain angle.
- In this activity you have seen how you can represent any point on the canvas with two numbers – the point's x coordinate and y coordinate (x, y).
- Any point's x and y coordinates are defined in relation to a special point called the origin, which has x and y coordinates of (0, 0).
- Two special lines through the origin, the x and y axes, help you to visualize the location of any point on the canvas. Grid lines further help with this visualization.
- The x axis runs west to east. The y axis runs south to north. Any point's x coordinate is its distance east of the origin. Any point's y coordinate is its distance north of the origin.
- This idea of representing geometric entities using numbers is the basis for the subject of coordinate geometry.

## Exercise

Write a program to make the following figure:



# 19 Exporting your creations

This activity involves the following:

- Learning how to export a drawing. You might want to do this to:
  - Post your drawing to Facebook.
  - Email your drawing to a friend.
  - Put your drawing on a website.
  - Print your drawing, to hang it up on your wall, or give it to someone else as a greeting card.

## Step 1

Type in the following code and run it:

```
clear()
setAnimationDelay(0)
setBackground(yellow)
setPenThickness(3)
setPenColor(black)
setFillColor(Color(0, 95, 172))
repeat(16) {
  repeat(6) {
    forward(96)
    right(60)
  }
  right(22.5)
}
setFillColor(yellow)
right()
hop(30)
left()
circle(30)
invisible()
```

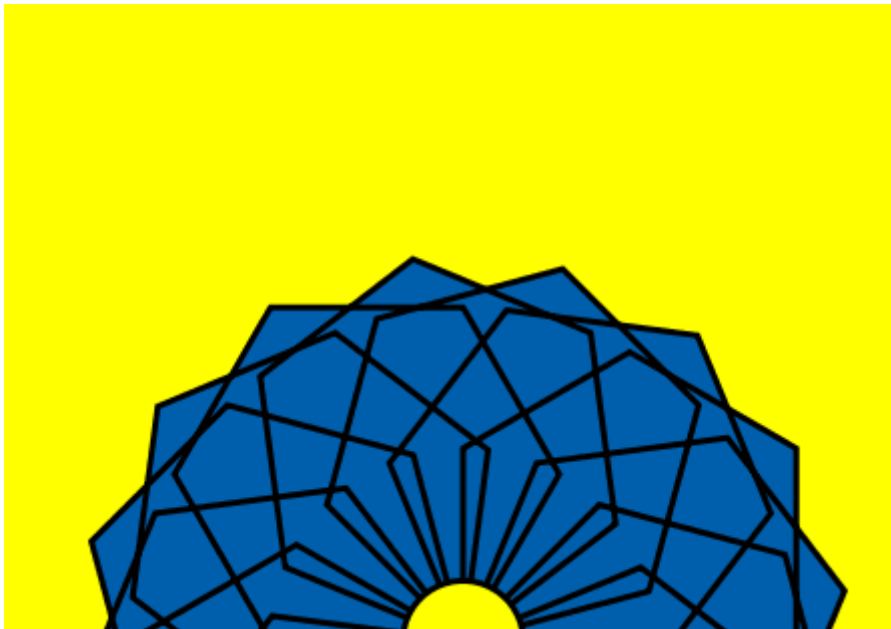
Now, to export (as an image) the drawing made by the above code, do the following:

- Right click on the drawing canvas, and then click on *Save as Image*. Kojo will prompt you for a filename for the exported image file into which the drawing will be written.

- Navigate to any folder that you want, and then save the drawing there under the name `hex-pattern.png`.
- Now go over to that folder using the file manager on your machine, and double click on `hex-pattern.png`. You should see your Kojo drawing in the image that pops up. Notice that the entire drawing canvas is present in the image.
- To control what is displayed in the image, you can resize the canvas, and then pan the drawing in the canvas to get exactly the view that you want before you carry out the *Save as Image* action.

## Step 2

Export something that looks like the following image from the drawing in the canvas:



Note – To resize the canvas, you can dock the drawing canvas to the right of the Kojo workspace by clicking on the title of the canvas and dragging it to the right edge of the workspace. You can then reposition the left and bottom borders of the canvas to get the desired canvas size. Once you are done exporting the image, you can get back to the default Kojo workspace by clicking on *Window -> Default Perspective*.

## Step 3

The image in the previous step is 448 pixels wide and 313 pixels high. This is a perfectly fine image size for putting on a website like Facebook. But printing is another story. If you print the image out as a 6 inch wide drawing on a piece of paper, the number of pixels per inch will be  $\frac{448}{6} = 74.7$ . A printout at that resolution will have jagged lines and will not look good (you need at least 300 pixels per inch for a good printout). Kojo provides a couple of commands to help you export your images at a higher resolution:

- `exportImageH(filePrefix, height)` – saves the contents of the drawing canvas as an image with the given height, in a file located in the temporary directory on your machine. The name of the file starts with `filePrefix`. Kojo prints out the full path of the exported image file in the output pane so that you can easily locate the file.
- `exportImageW(filePrefix, width)` – works similar to the above command, except that you specify the image width instead of height.

So, to generate an image that can be printed with a width of 6 inches at 300 pixels per inch, you can do the following:

```
exportImageW('`hex-pattern`', 1800)
```

## Self Exploration

Play with the above ideas as you see fit.

## Exercise

Export one of your drawings from an earlier activity and print it out.



# 20 Random Numbers and Named Values

This activity involves the following:

- Learning about random numbers.
- Learning to use random numbers to make interesting figures.
- Learning to use the `random` function.
- Learning to save the results of functions for later use in a program.
- Learning about keyword instructions.
- Learning the `val` keyword instruction.
- Learning to nest functions within commands.
- Learning to save the turtle's position and heading, and restoring it later.

## Step 1

Type in the following code and run it using the *Run as Worksheet* button:

```
random(50)
```

Q1a What do you think the random functions does?

## Step 2

Run the code above a few more times (again using the *Run as Worksheet* button).

Q2a Now what do you think the random functions does?

## Step 3

Type in the following code and run it (a few times):

```
clear()
setAnimationDelay(10)
setPenColor(gray)
repeat(20) {
    savePosHe()
    setPenThickness(random(20))
    setPenColor(Color(0, 30, 200, random(255)))
    right(random(360))
    forward(random(100))
    restorePosHe()
}
```

**Q3a** What do you think the `forward(random(100))` instruction does? How does it combine a function with a command?

Note – when the input to a command is the return value of a function, the function is said to be nested within the command. The general idea of nesting in programming relates to putting something inside another thing.

**Q3b** What do you think the `savePosHe` and `restorePosHe` commands do? Why are they needed here?

Hint – the turtle moves forward by a random amount to make a line, and then needs to go back to its starting point before making the next line.

## Step 4

Type in the following code and run it (a few times):

```
clear()
setAnimationDelay(10)
setPenColor(gray)
val len = random(100) + 100
repeat(2) {
    forward(len)
    right()
    forward(20)
    right()
}
```

**Q4a** What do you think the `val` instruction does? Why is it needed here?

Hint – the height of the rectangle is random, and cannot be recalculated. But this height needs to be used twice to make the two vertical sides of the rectangle.

## Self Exploration

Play with the code in steps 3 and 4 above and try to fully understand it.

## Theory

- As mentioned earlier, programs are made out of a series of instructions for the computer. You have seen two kinds of instructions in previous activities:
  - Commands, which let you take actions or affect future actions (like moving the turtle forward or setting the pen color).
  - Expressions, which let you do calculations (e.g.  $5+9$ ). Remember, functions are expressions.
- You saw a new kind of instruction in this activity – a keyword instruction (`val`). A keyword instruction allows you to structure your program better (there's more to it than that, but this is a good working definition).
- The `val` keyword instruction allows you to give a name to a value. Let's look at this line of code from Step 4 to see this in action:

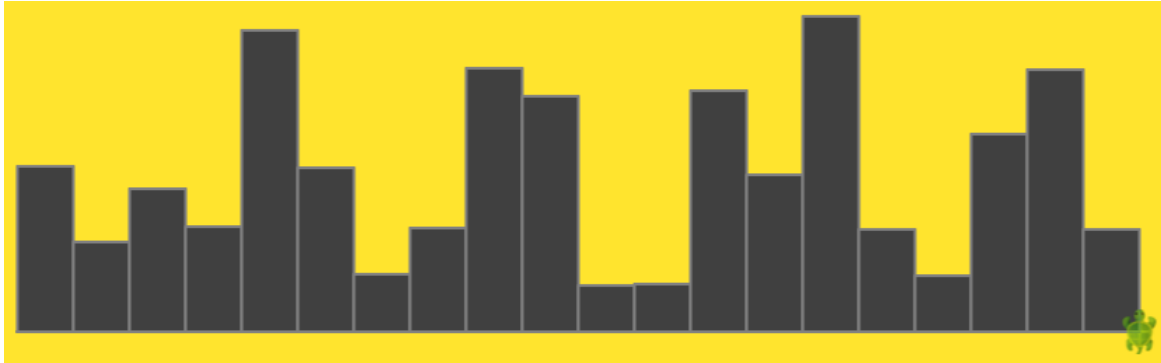
```
val len = random(100) + 100
```

- When Kojo runs this line, it first evaluates the right-hand-side, and then names the resulting value `len`. You can then use these named values in different locations in your program.
- In general, named values are useful because:
  - they let you store values that cannot be recalculated (because they depend on a random component, for example) but are used multiple times in your program – like in Step 4.
  - they let you avoid recalculating values that are used multiple times in your program. Instead of recalculating a value, you just calculate it once, give it a name, and then use it wherever it is required in your program.
  - they make it easier to make changes in your program – because you change the named value in only one location in your program, and that change is picked up wherever the named value is used in the program.
  - they make your programs more understandable – because they let you attach a descriptive name to a value.

## Exercise

Write a program to make a figure that looks something like the following figure:

- The heights of the rectangles in the figure are random.
- All the rectangles have the same width.



Hint – Each rectangle can be made with a repeat command (like in Step 4). There are 20 rectangles in the figure. So you can consider using a repeat (to make a rectangle) inside another repeat (to make 20 rectangles).

# 21 Your own Commands

This activity involves the following:

- Learning to create new commands within Kojo using the `def` instruction.
- Becoming familiar with the very important programming ideas of primitives, composition, and abstraction.
- Applying the idea of percentages to determine the dimensions of geometric figures.

## Step 1

Type in the following code and run it:

```
clear()
setAnimationDelay(100)
repeat(4) {
    forward(50)
    right()
}
hop(60)
repeat(4) {
    forward(50)
    right()
}
right()
hop(60)
left()
repeat(4) {
    forward(50)
    right()
}
hop(60)
repeat(4) {
    forward(50)
    right()
}
```

## Step 2

Now type in the following code and run it:

```
clear()
setAnimationDelay(100)
def square() {
  repeat(4) {
    forward(50)
    right()
  }
}
square()
hop(60)
square()
right()
hop(60)
left()
square()
hop(60)
square()
```

**Q2a** How is the code in Step 1 similar to the code in Step 2? How is it different?

**Q2b** What do you think the `def` instruction does?

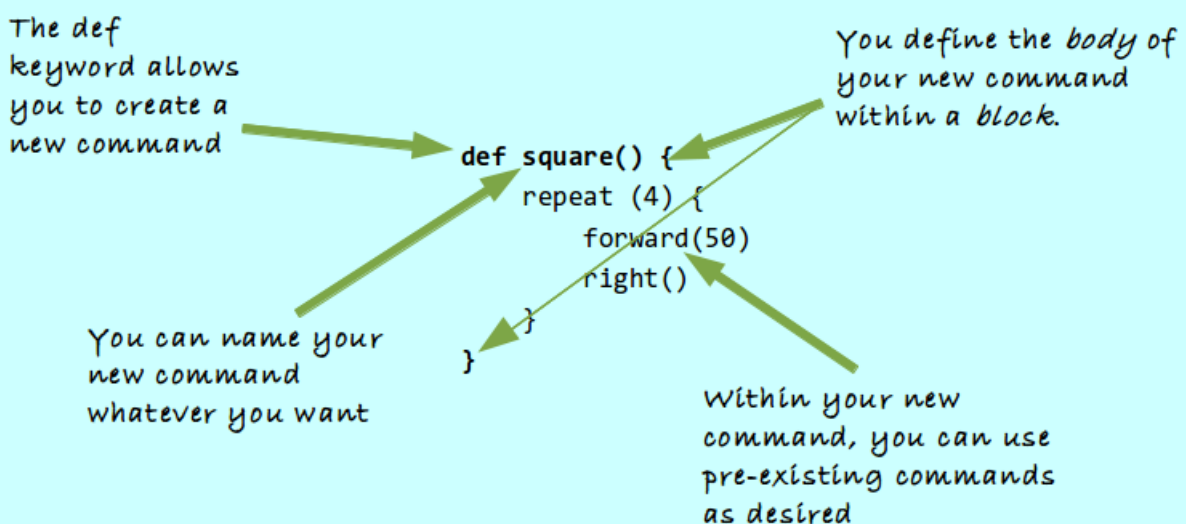
## Self Exploration

Play with:

- Changing the definition of the `square` command to make squares of size other than 50.
- Using the `square` command to create additional squares within the drawing.

## Theory

- Lets do a quick recap of programs. Programs are made out of a series of instructions for the computer. These instructions are of the following three kinds:
  - Commands, which let you take actions or affect future actions (like moving the turtle forward or setting the pen color).
  - Expressions, which let you do calculations (e.g. 5+9).
  - Keyword instructions, which let you structure your programs.
- The `def` keyword instruction lets you create new commands of your own. You can then use or call these commands just like you would call predefined Kojo commands.
- The code in Step 2 defines the square command. Here's a closer look at that fragment of code:



- What's the benefit of creating your own commands? These commands allow you to:
  - capture commonly used patterns of code.
  - give them a name.
  - reuse these patterns.
- This helps by:
  - reducing code duplication.
  - making your programs easier to understand.



- There's also a way of looking at the `def` instruction in terms of a deeper idea. Let's explore that idea...

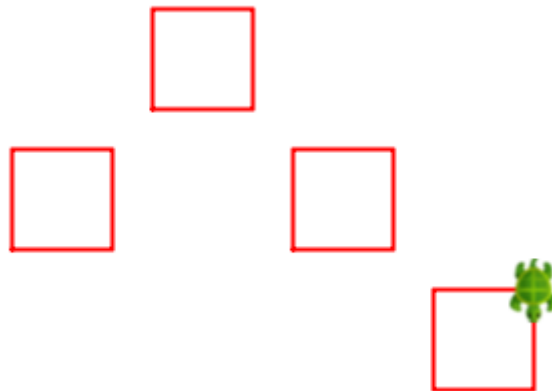
Computer programming is about three basic things:

- primitives – these are the instructions already available in our programming environment.
  - composition – this is how you combine primitives to do what is required.
  - abstraction – this is how you give our compositions a name, so that they can be used as higher level primitives within your programs. These abstractions are used without regard to how they are implemented.
- Seen from this perspective, the `def` instruction allows you to:
    - create a new abstraction, i.e., your new command.
    - implement the abstraction using a combination of primitives, i.e., preexisting commands.

## Exercise

Write a program to make the following figure:

- The size of each square is 50 pixels.
- The vertical and horizontal distances between the squares are 40% of the square size.



## 22 Your own Commands, with Inputs

This activity involves the following:

- Learning to create new commands that take inputs, thus letting them change their behavior based on input values.
- Learning to estimate the dimensions of a figure given the size of one line in the figure.

### Step 1

Type in the following code and run it:

```
clear()
setAnimationDelay(100)
setPenColor(black)
setFillColor(orange)
repeat(3) {
  repeat(4) {
    forward(100)
    right()
  }
  repeat(4) {
    forward(70)
    right()
  }
  repeat(4) {
    forward(30)
    right()
  }
  forward(100)
}
```

### Step 2

Now type in the following code and run it:

```
clear()
setAnimationDelay(100)
setPenColor(black)
```

```
setFillColor(orange)
def square(n: Int) {
  repeat(4) {
    forward(n)
    right()
  }
}
repeat(3) {
  square(100)
  square(70)
  square(30)
  forward(100)
}
```

**Q2a** How is the code in Step 1 similar to the code in Step 2? How is it different?

## Self Exploration

Play with using the square command to create additional different sized squares within the drawing.

## Theory

- Here's what a command that takes an input looks like:

The input to the square command is a 'named value' called 'side'

The 'type' of the input is 'Int'

```
def square(side: Int) {
  repeat (4) {
    forward(side)
    right()
  }
}
```

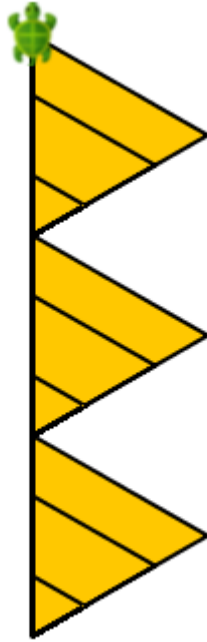
You use the input named value within your command

- Within the round-brackets in the first line of code above, you are telling Kojo that the input to the square command is called *side*, and that its type is *Int* (where, as you have seen earlier, *Int* stands for integer). Now, instead of always drawing squares of the same size, the square command can draw squares of different sizes – based on the input that you provide to it.
- Inputs to commands are *named values* that can be used within the body of a command (do you remember named values from an earlier Activity?).
- Inputs to commands also have *types* associated with them. The type of an input tells Kojo:
  - the permissible values of the input.
  - the functions and commands that the input can work with within the body of the command.
- Telling Kojo the type of the input (to your new command) has a couple of advantages:
  - It makes it easy for Kojo to identify problems with your usage of the input value, and to tell you if you make a mistake.
  - It makes it easier for you (and your friends) to understand what the command does when you (or they) look at it later.
- Think about how all of this relates to the ideas of primitives, composition, and abstraction.

## Exercise

Write a program to make the following figure:

- The size of the vertical black line that runs from the bottom to the top of the figure is 300.
- Use your best judgment to estimate the other dimensions in the figure.



## 23 Polygon Art

This activity involves the following:

- Exploring polygons with different numbers of sides.
- Exploring the interior and exterior angles of polygons.
- Learning about variables.
- Learning to programatically modify colors.
- Learning the `var` keyword instruction, and the `hueMod` function.

### Step 1

Type in the following code and run it:

```
clear()
repeat(3) {
  forward(100)
  right(360 / 3)
}
```

Q1a Is the figure made by the code above a polygon? Why?

Q1b What is the sum of the exterior angles of the figure?

Q1c What is the sum of the interior angles of the figure?

### Step 2

Type in the following code and run it:

```
clear()
setAnimationDelay(10)
repeat(5) {
  forward(100)
  right(360 / 5)
}
```

Q2a What is the sum of the exterior angles of the figure?

Q2b What is the sum of the interior angles of the figure?

## Step 3

Type in the following code and run it:

```
clear()
setAnimationDelay(10)
repeat(6) {
  forward(100)
  right(360 / 6)
}
```

Q3a What is the sum of the exterior angles of the figure?

Q3b What is the sum of the interior angles of the figure?

Q3c Can you derive formulas in terms of  $n$ , the number of sides of a polygon, for the sum of the interior and the sum of the exterior angles of the polygon?

## Step 4

Type in the following code and run it:

```
clear()
setAnimationDelay(10)
var fill = Color(17, 255, 0, 150)
repeat(3) {
  setFillColor(fill)
  repeat(5) {
    forward(100)
    right(360 / 5)
  }
  right(20)
  fill = hueMod(fill, 0.2)
}
```

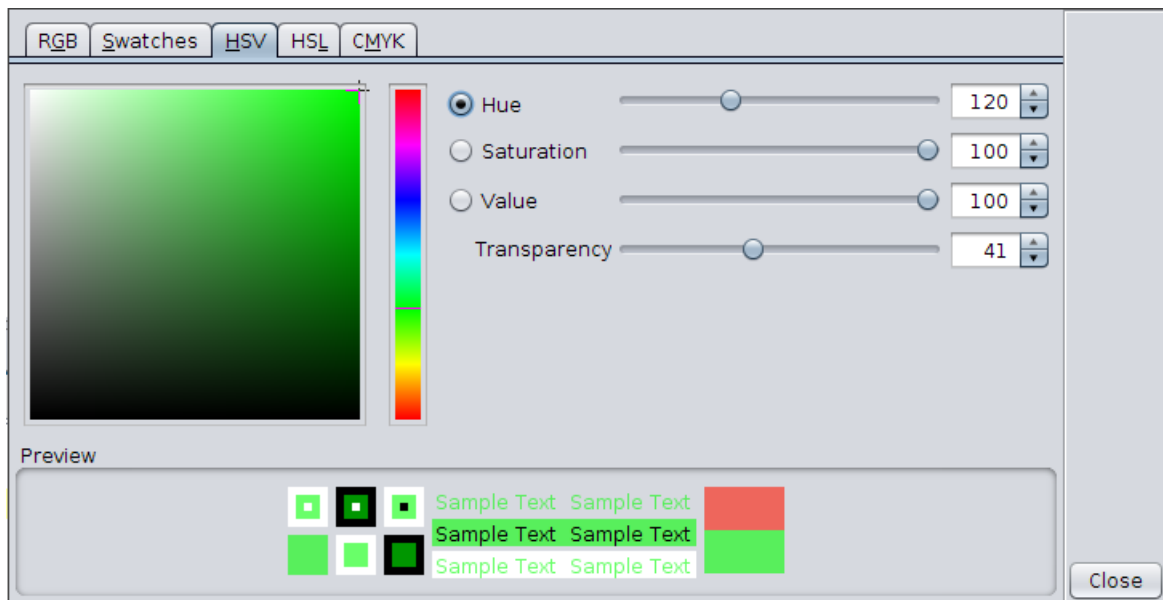
Q4a What do you think the `var` keyword instruction does?

**Q4b** What do you think the `hueMod` function does?

Note – Do you remember the RGB model for representing colors? Another color model (which makes it very easy to modify a color in a controlled way) is the HSV (Hue, Saturation, Value) model (Google it to find out more about it). The word `hue` in the function `hueMod` above relates to the H in the HSV color model.

## Self Exploration

Play with the code in the above steps as you see fit. To explore the idea of color hue, click on the `Color` function in the code for step 4, and then click on the *HSV* tab. You should see the pane shown below. Play around with Hue (and Saturation and Value) values to get a feel for how you can specify a color using the HSV model. You can see the RGB representation of any color that you select by clicking on the RGB tab.





## Theory

### Variables

- The `var` keyword instruction allows you to create a variable and attach it to a value:

```
var fill = Color(17, 255, 0, 150)
```

A variable is like a label that is attached to a value. Here, the label name is on the LHS of the `=` sign, and the value is on the RHS.

- You can then use the label/variable instead of the value in the program:

```
setFillColor(fill)
```

- You can detach the label/variable from its current value and attach it to another value:

```
fill = hueMod(fill, 0.2)
```

- The next time you use the label/variable in your program, it's new value is picked up.

## Color modification

- You can easily modify a color using the `hueMod` function. Shown below is the range of hues available in the HSV color model:



The bottom-most hue has a numeric value of 0, and the top-most hue has a numeric value of 360.

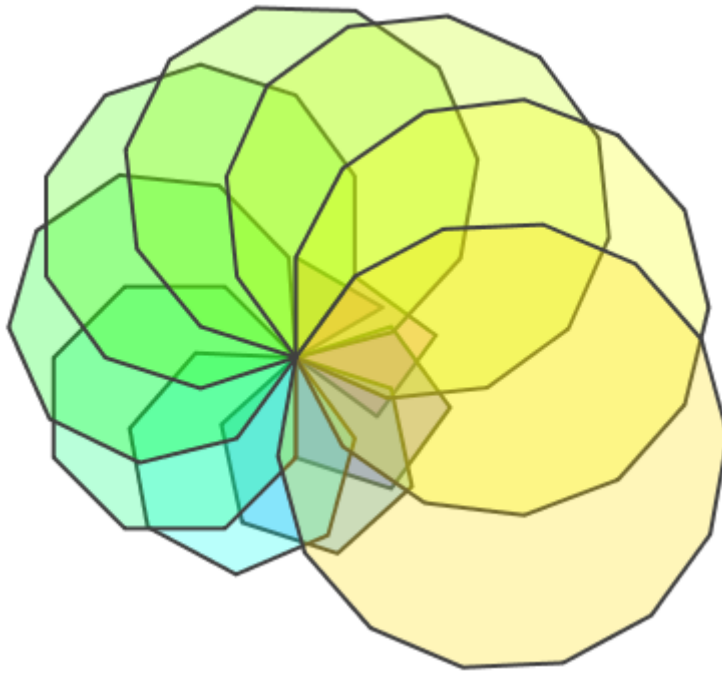
- Let's say that (as in step 4) your current color is stored in a variable called `fill`. The current value of `fill` is green (as shown by the thin horizontal line in the figure above).
- Under these conditions, `hueMod(fill, 0.5)` will return a color that is located exactly half way (i.e. 0.5 of the way) between green and the top of the hue-range shown above. This will be a bluish color.
- `hueMod(fill, -0.5)` will return a color that is located exactly half way (i.e. 0.5 of the way) between green and the bottom (because of the negative sign in front of 0.5) of the hue-range shown above. This will be a yellowish color.
- `hueMod(fill, 0.75)` will return a color that is located exactly three-fourth of the way (i.e. 0.75 of the way) between green and the top of the hue-range shown above. This will be a purplish color.
- `hueMod(fill, -0.75)` will return a color that is located exactly three-fourth of the way (i.e. 0.75 of the way) between green and the bottom (because of the negative sign in front of 0.75) of the hue-range shown above. This will be an orangish color.

That should give you a good idea of how `hueMod` works.

## Exercise

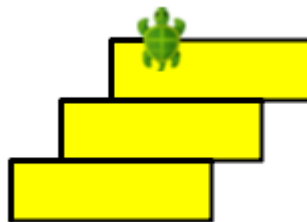
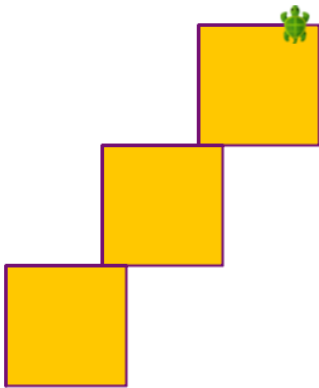
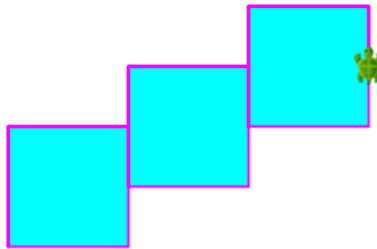
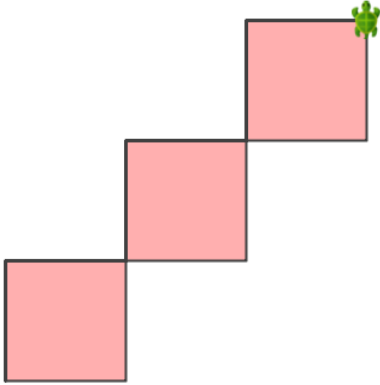
Write a program to make the following figure:

- The figure contains 12 polygons – with 3 to 14 sides.
- The sides of the polygons are 50 pixels long.
- The angle between the polygons is 36 degrees.
- The color of the first polygon has a hue at the top of the hue-range. The colors of the other polygons come down the hue-range. The color of the last polygon has an orangish hue.



# 24 Pattern Drawing Practice

Write programs to make the following patterned figures:



# 25 Patterns

This activity involves the following

- Learning to analyze and identify patterns.
- Learning to make patterns using the repeat command.

So, what's a pattern?

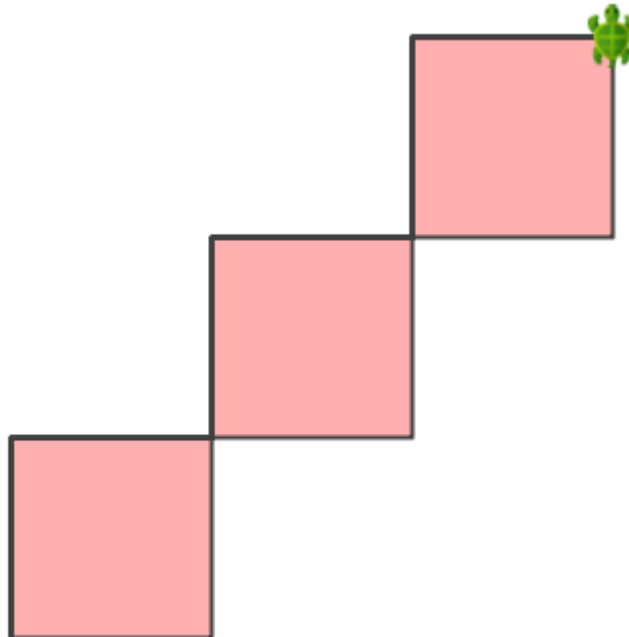
A pattern is something that contains a repeated building-block; the building-block is repeated to make the pattern.

For the current discussion, you can think of a pattern as a figure that contains a smaller building-block shape inside it. This building-block is repeated in a uniform way to make the pattern.

Patterns play a crucial role in Computer Programming and Math.

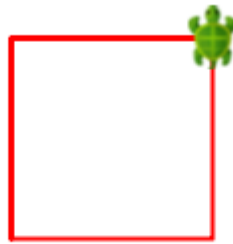
## Step 1

Here, again, is the first figure that you had to draw in the previous exercise:



Try to identify the building-block of the above pattern.

The building-block of the pattern is:



Note that the building-block consists of three things:

- A shape. In this case, the shape is a square.
- A start-position and start-direction within the shape (where the turtle starts drawing the shape). Here, the start-position is the bottom-left corner of the square, and the start-direction is north.
- An end-position and end-direction for the turtle relative to the shape, so that it is ready to draw the next building-block in the pattern. In this case, the end-position is the top-right corner of the square, and the end-direction is north. Note – the end-position and end-direction become the start-position and start-direction for the next building block in the pattern.

## Step 2

Let's come up with a little recipe for making patterns. Given a pattern that you have to make, follow this procedure:

1. Identify the building-block of the pattern and draw it out on paper. The building-block should contain three pieces of information: a shape, a start-position and start-direction (marked with a cross and an arrow), and an end-position and end-direction (marked with a circle and an arrow).
2. Create a user-defined command in Kojo to make the building block.
3. Draw the building-block by using the new command.
4. Repeat the building-block (via the newly created command), for as many times as the pattern requires – using the repeat command.

## Step 3

Using the above procedure, you can come up with a program like the following for making the pattern. Type the program in and run it.

```
def squareBlock() {
  // start of building-block
  repeat(4) {
    forward(100)
    right()
  }
  hop(100)
  right()
  hop(100)
  left()
  // end of building-block
}
clear()
setAnimationDelay(100)
setPenColor(darkGray)
setFillColor(pink)
repeat(3) {
  squareBlock()
}
```

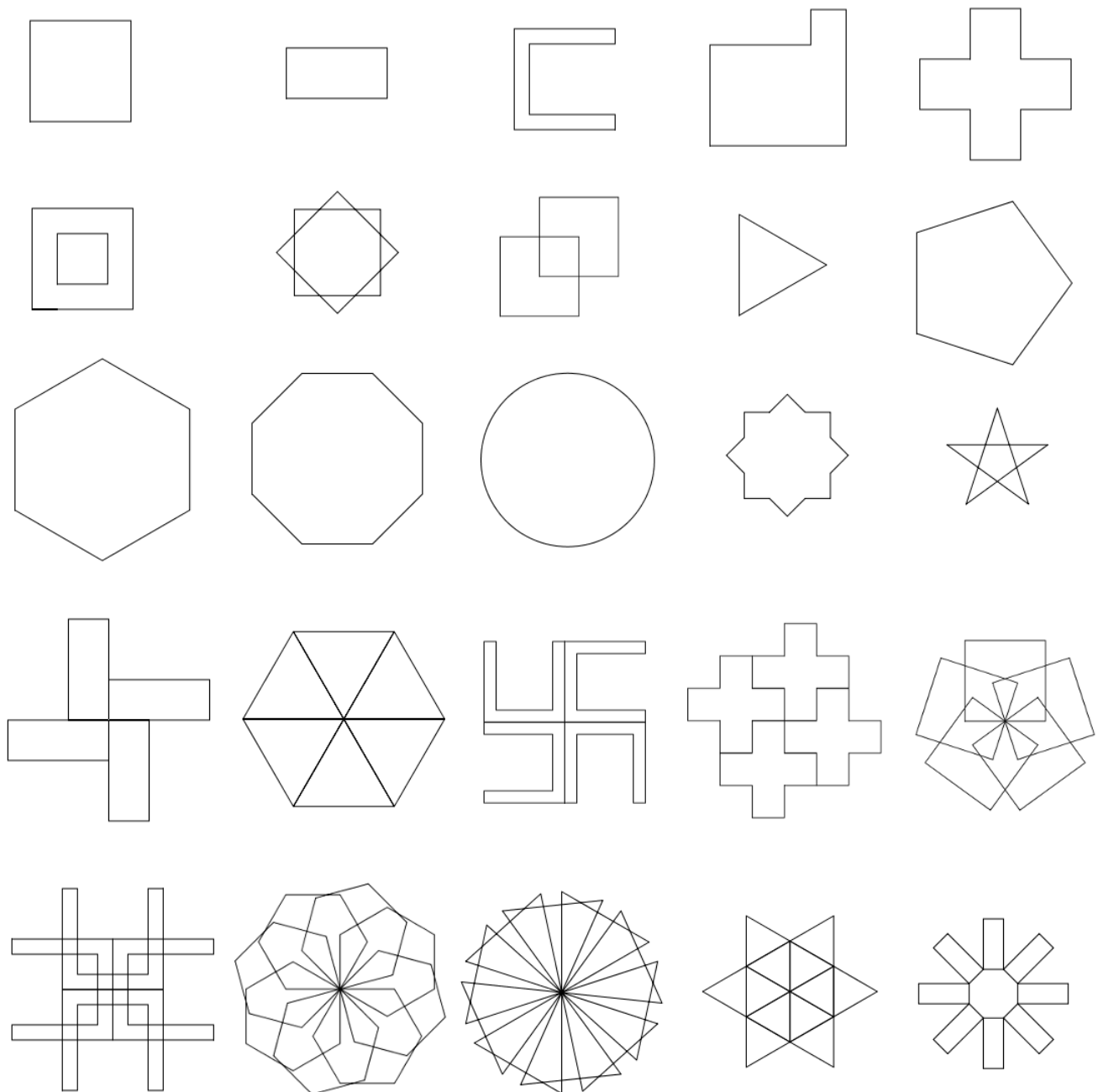
Make sure you understand how this program makes the pattern.

- Q3a Identify the start of the building-block in the code above.
- Q3b Identify the end of the building-block in the code above.
- Q3c What pieces of information does the building-block contain?
- Q3d What lines in the code above make the building-block shape?
- Q3e What lines in the code above move the turtle to the building-block end-position?
- Q3f What lines in the code above turn the turtle towards the building-block end-direction?

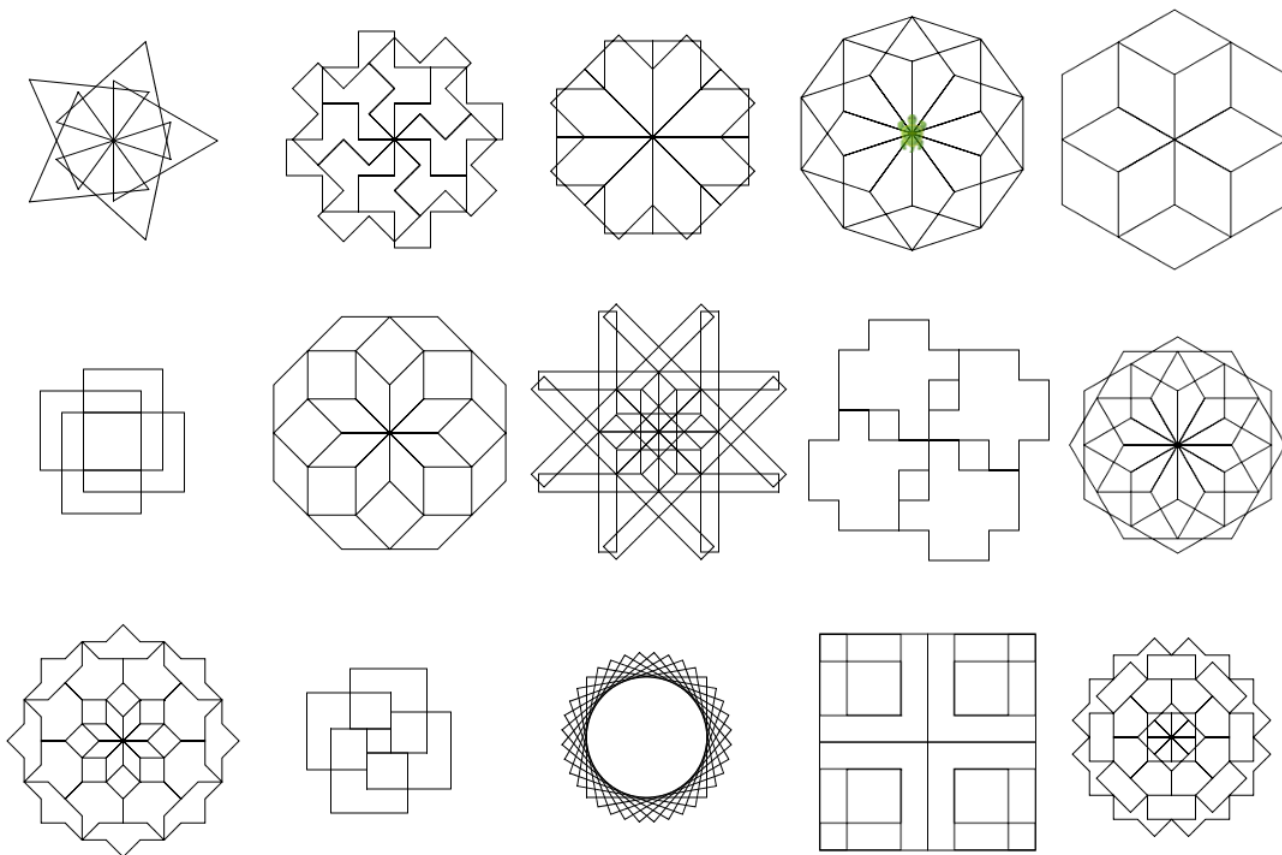
## 26 More Practice with Patterns

Write programs to make the following figures. Make use of the ideas introduced in the previous chapter (involving the repetition of named pattern building blocks) to make the patterns.

Note – these patterns are Kojo drawings of the patterns presented in a wonderful book by Barry Newell called [Turtle Confusion](#).







# 27 Your own Functions

This activity involves the following:

- Learning to create new functions.
- Playing with *simple interest* to make a figure.

## Step 1

Type in the following code and run it using the *Run as Worksheet* button:

```
// Simple interest exploration
// P is the Principal
val P = 100.0
// R is the rate of interest
val R = 10.0
// si is a function that lets you calculate simple interest
// given a principal and a rate of interest
def si(p: Double, r: Double) = p * r / 100
// amt is a function that tells you the amount after interest
// given a principal, a rate of interest, and the number of years
def amt(p: Double, r: Double, t: Int) = {
    p + si(p, r) * t
}
amt(P, R, 0)
amt(P, R, 1)
amt(P, R, 2)
```

Q1a What do you think the code above does?

Q1b Looking at the results of the code, can you tell what the *amount* is after 2 years?

## Self Exploration

Play with the code above as you see fit.

## Theory

- You know that programs are made out of three types of instructions: commands, expressions, and keyword instructions.
- Let's look further at expressions. The simplest expressions are literals – data values (like 3 and 4) that you write down in your code that cannot be simplified any further. All other expressions are built out of functions operating on simpler expressions, e.g.,  $3 + 4$ , where the expression  $3 + 4$  is built out of the function  $+$  and the simpler expressions 3 and 4.
- Some of the predefined functions available within Kojo are  $+$ ,  $-$ ,  $*$ ,  $/$ ,  $\text{math.max}$ ,  $\text{math.sqrt}$  etc. In this activity you have seen how you can define your own functions:

```
def amt(p: Double, r: Double, t: Int) = {
  p + si(p, r) * t
}
```

- Note how similar this is to the way in which you create a new command. The big difference is the use of the equals sign on the first line of the definition:

```
def amt(p: Double, r: Double, t: Int) = {
```

- Command definitions do not require this  $=$  sign. The  $=$  sign in function definitions is meant to signify that functions are equivalent to the values that they calculate (based on inputs) and return to the caller. To understand this idea better, let's look at the following function call:

```
amt(P, R, 1)
```

- Here,  $\text{amt}(P, R, 1)$  is equivalent to the value that it calculates (110). Any occurrence of  $\text{amt}(P, R, 1)$  in the program can safely be replaced by the value 110 without changing what the program does. A command, on the other hand, is not equivalent to anything because it doesn't calculate and return a value; instead, it just carries out some actions. You cannot replace a command with a value in a program and expect the program to do the same thing. Hence we do not put in the  $=$  sign on the first line of command definitions.
- Note that if a function calculates its return value based on just a single expression, there is no need to use curly brackets, and the whole function can be defined on a single line:

```
def si(p: Double, r: Double) = p * r / 100
```

- Think about how defining new functions relates to the idea of primitives, composition, and abstraction.

### Math Recap – Functions and Variables

- From a mathematical point of view, functions are a very fundamental idea. To understand functions, you need to know about variables.
- Variables are mathematical entities that transport you from arithmetic to algebra. In arithmetic, you talk about particular numbers. In algebra, you start using variables to refer to numbers. As opposed to particular numbers, variables let you talk about *any* number(s) or *some* number(s).
- Here's an example of variables talking about *any* numbers:

$$x + y = y + x$$

This says that for any  $x$  and  $y$ , the sum of  $x$  and  $y$  is the same as the sum of  $y$  and  $x$ .

- And here's an example of variables talking about *some* numbers:

$$x^2 + y^2 = 25$$

This says that variables  $x$  and  $y$  are related as per the given equation. Given *some* value of  $x$ ,  $y$  can only take on *some* values that satisfy the equation.

- As you can see, equations allow you to write down relationships between variables.
- Some equations make the relationship between variables explicit:

$$y = 3x + 4$$

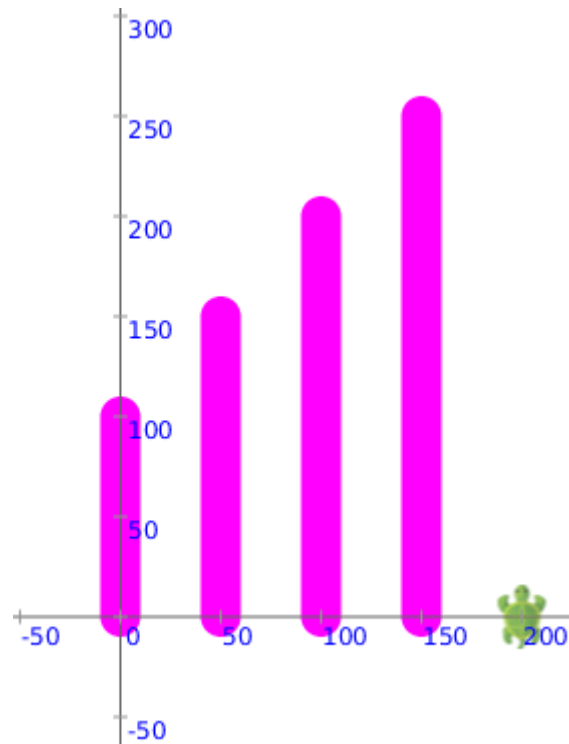
These types of equations allow you to easily find, given the value of one variable ( $x$ ), the value of the related variable ( $y$ ). These equations are called functions.

- In other words, functions allow you to convert, or map, one variable (the input value of the function) into another variable (the result value of the function).
- You can also have functions of many variables, which map multiple variables (the input values) into another variable (the result value). For example, in step 1 above, the function `amt(p: Double, r: Double, t: Int)` maps three numbers – a principal, a rate, and a time, into one number – the amount that the given principal grows to at the given rate in the given time.

## Exercise

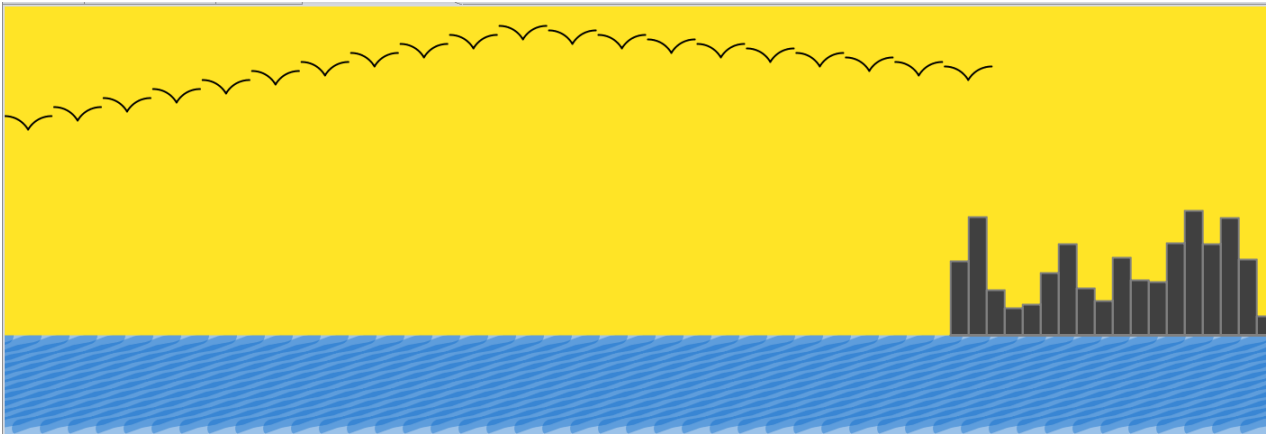
Write a program to make the following figure:

- The figure contains four lines. Each of the lines is 20 pixels thick.
- The lengths of the lines represent the amounts for a simple interest problem at the end of years 0, 1, 2, and 3.
- For the simple interest problem,  $P = 100$  and  $R = 50\%$
- The distance between the lines is 50 pixels.



## 28 Mini Project

Write a program to make the following drawing:



Ideas:

- Use things like the `repeat` command and user defined commands to avoid duplication in your program.
- Use the `random` function to make the buildings.
- Use a pen with a large thickness value and a semi-transparent color to make the water region.

## 29 Strings and I/O

This activity involves the following:

- Learning about impure functions.
- Learning to use the `readln` and `readInt` functions.
- Learning to use the `clearOutput` and `println` commands.
- Learning about program input and output.
- Learning about strings.
- Learning about string interpolation.
- Playing with the idea of number averages.

### Step 1

Type in the following code and run it:

```
clearOutput()  
val name = readln("What's your name?")  
val age = readInt("What's your age?")  
println(s"Hello $name, your age is $age")
```

- Q1a** Is the program above taking any input? Is it providing any output?
- Q1b** What do you think the `clearOutput` command does?
- Q1c** What do you think the `readln` instruction does?
- Q1d** What do you think the `readInt` instruction does?
- Q1e** What do you think the `println` command does?
- Q1f** What do you think is the data between the double quotes, e.g., "What's your name?"?
- Q1g** Why do you think the input to the `println` command has an 's' at the beginning – `s"Hello $name, your age is $age"`?

## Self Exploration

Play with the code above as you see fit.



## Theory

- This activity covers a lot of ground. Let's look at the new ideas introduced in this activity.
- A Program interacts with its user by taking in inputs that the user provides, working with these inputs, and providing outputs that the users can see. The following commands are used for this purpose in this activity:
  - `readln` – takes a String typed in by the user and makes it available to the program.
  - `readInt` – takes a String typed in by the user, converts it to an integer, and makes it available to the program.
  - `println` – takes a String, and prints it out in the output pane.
- But wait a minute. Are `readln` and `readInt` commands? They seem to be commands because they produce an action (a textbox shows up in the output pane where you can type in stuff). But they also return a value (whatever you type in). Since a command never returns anything, they can't really be commands. But they are not functions either (because they carry out an action). So what are they?
- We call them impure functions. They are functions because they return a value, but they are impure because they carry out an action.
- Next, let us look at Strings, which are used by all three instructions mentioned above. Strings are:
  - used to represent text.
  - a type within Kojo.
  - particularly useful for providing input to a program and generating output from a program.
- You create a string by enclosing some text within double quotes, e.g., “an example string”. When you use a string, Kojo does not really care about what is inside the string. For Kojo, strings are just text (that make sense to you). But there is one exception to this. If you create a string like this – `s"my name is: $name, and my age is $age"`, Kojo will plug in the values for the names that you provide into the created string. This kind of string is called an interpolated string. Note the use of the `s` prefix to create the string, and the use of the `$` sign to plug in values into the string.

## Exercise

Write a program that reads in 2 integers provided by the user, and then prints out their average.

# 30 Artistic Text in the Canvas

This activity involves the following:

- Learning to write text on the canvas.
- Learning to determine the fonts that are available in the system.
- Learning to specify the font and size of text written to the canvas.
- Learning to peek inside text strings.

## Step 1

Type in the following code and run it:

```
clear()  
invisible()  
write("Hello There!")
```

**Q1a** What do you think the `write` commands does? What does the input to the command specify?

## Step 2

Type in the following code and run it:

```
cleari()  
setPenFont(Font("Monospaced", 20))  
write("Hello There!")
```

**Q2a** What do you think the `cleari` command does?

**Q2b** Is `Font` a command or a function? How many inputs does it take? Try changing the second input to see what it does? You will explore the first input later.

**Q2c** Is `setPenFont` a command or a function? How many inputs does it take?

## Step 3

Type in the following code and run it:

```
println(availableFontNames)
```

This shows you a list of names of the fonts available on your system. You can use any one of these names while creating a font using the `Font` function that you saw in the previous step.

## Step 4

Type in the following code and run it:

```
clear()
setAnimationDelay(100)
val msg = "Hello There"
repeatFor(0 to msg.length - 1) { idx =>
    left()
    write(msg[idx])
    right()
    right(17, 60)
}
```

This step might be a little difficult to understand. Here are some things you should know as you try to decipher the code above:

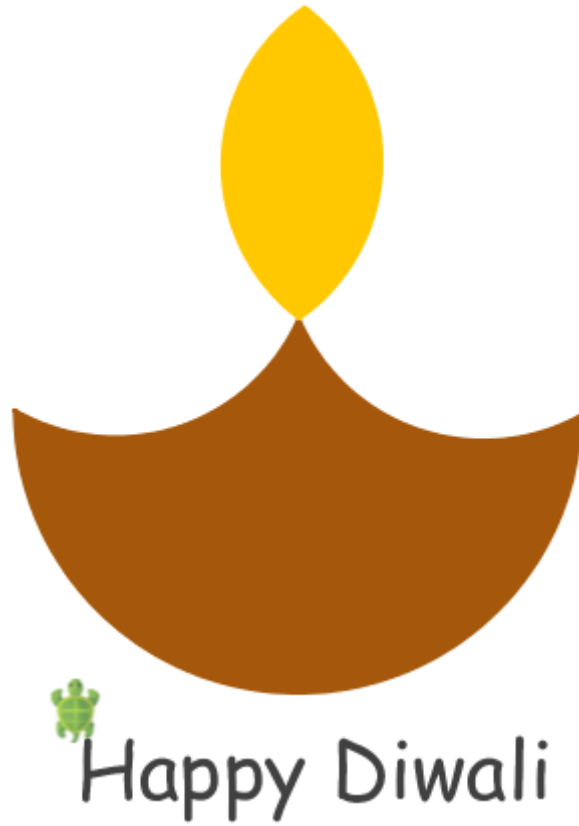
- `val msg = "Hello There"` creates a string and gives it a name – `msg`.
- `msg.length` is a function that gives you the length of the `msg` string (11).
- `msg(0)` is the first character of `msg` (H)
- `msg(10)` is the last character of `msg` (e)

## Self Exploration

Use the list of font names from step 3 to get the “Hello There!” from step 2 to show up in different fonts within the drawing canvas. Play further with the code in the steps above as you see fit.

## Exercise

Write a program to make the following figure:



# 31 Conditionals

This activity involves the following:

- Learning about conditionals.
- Becoming familiar with the important programming idea of selection.
- Learning to work with numeric conditions.

## Step 1

Type in the following code and run it:

```
clear()
val clr = readInt("Square fill color? Enter 1 for green, anything else for blue")
if (clr == 1) {
    setFillColor(green)
}
else {
    setFillColor(blue)
}
repeat(4) {
    forward(100)
    right(90)
}
```

**Q1a** What do you think the `if` keyword instruction does?

## Step 2

Type in the following code and run it:

```
clearOutput()
val n1 = readInt("Enter first number")
val n2 = readInt("Enter second number")
val bigger = if (n1 > n2) n1 else n2
println(s"The bigger number is $bigger")
```

**Q2a** Now what do you think the `if` keyword instruction does? How does it work differently here as compared to step 1.

## Self Exploration

Play with the code above as you see fit.

## Theory

- The if-else keyword instruction allows you to make a choice within your program and select certain portions of your program to run only when certain conditions are true.
- Let's try to be reasonably precise about the definition of the word *condition*. A condition is a function that returns a true or false value. In this chapter, let's focus on conditions that work by comparing two numbers.
- Try the following (in the script editor, via the *Run as Worksheet* button):

```
2 == 3
```

Kojo will respond with something like:

```
res1: Boolean = false
```

- Now try:

```
3 == 3
```

Kojo's response is something like:

```
res2: Boolean = true
```

- What's the type of Kojo's response?

Boolean.

- What's the condition that gives this response?

Something like `2 == 3` (where `==` is an operator/function that takes two Ints as input, and returns a Boolean).

- Boolean is the fourth type you have seen. The others were Int, Double, and Color; true or false values (the results of conditions) are Booleans.

- The following table shows you some conditions:

Operator	Operator name	Condition	Condition Result
<	is less than	4 < 5	true
		5 < 5	false
		5 < 4	false
<=	is less than or equal to	4 <= 5	true
		5 <= 5	true
		5 <= 4	false
==	is equal to	4 == 5	false
		5 == 5	true
>	is greater than	4 > 5	false
		5 > 5	false
		5 > 4	true
>=	is greater than or equal to	4 >= 5	false
		5 >= 5	true
		5 >= 4	true

- You write the if-else instruction like this:

```

if (condition) {
    // do something when condition 1 is true
}
else if (condition2) {
    // do something when condition 2 is true
}
else {
    // do something when neither condition 1 nor condition2 are true
}

```

- The `else-if` and `else` parts above are optional.
- The `if-else` keyword instruction represents selection, and allows you to choose from a set of alternative instructions as you write your programs. Selection is a fundamental element of programming – along with primitives, composition and abstraction.



## Exercise

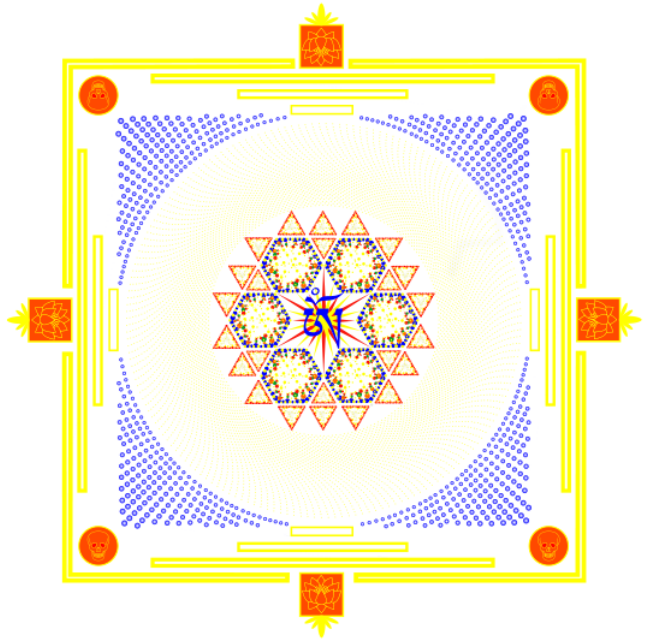
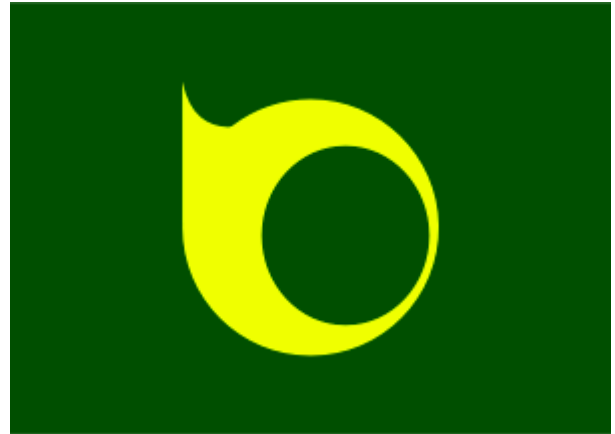
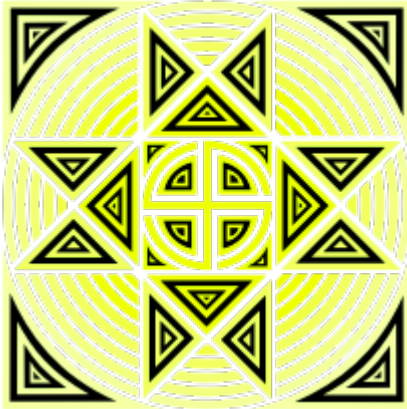
Write a program that asks the user what shape to make. If the user enters 1, make a square. If the user enters 2, make a rectangle.

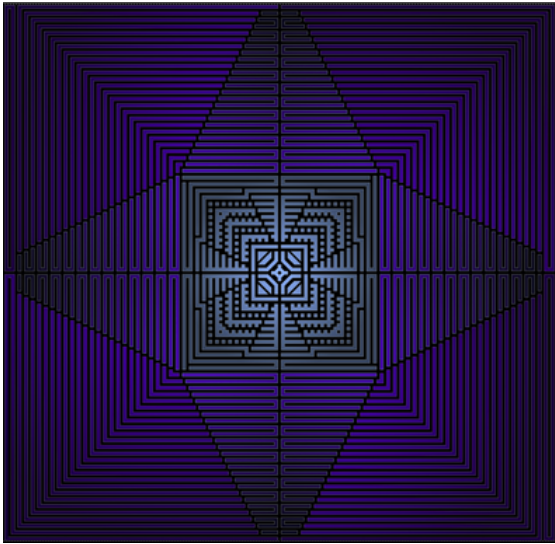
Ideas:

- Use the `readInt` function to show the user a message and retrieve the value entered by the user.
- Define a `rectangle` command that is capable of making both a rectangle and a square. Use this command to make the desired figure.

# 32 Art Breakout

Write programs to make the following figures:





## 33 Recursion

This activity involves the following:

- Learning about self-similar structures.
- Learning to create commands and functions that call themselves.

### Step 1

Type in the following code and run it:

```
def figure(n: Int) {  
    forward(n)  
    right(90)  
    figure(n - 5)  
}  
  
clear()  
figure(100)
```

Q1a How does the above program work? Trace the program to determine the answer.

Q1b What is wrong with the program?

### Step 2

Type in the following code and run it:

```
def figure(n: Int) {  
    if (n < 10) {  
        forward(n)  
    }  
    else {  
        forward(n)  
        right(90)  
        figure(n - 5)  
    }  
}
```

```
clear()
figure(100)
```

**Q2a** How does the program in this step improve upon the previous program?

## Step 3

Type in the following code and run it:

```
def seq(n: Int): Int = {
  if (n == 1) 2 else seq(n-1) + 3
}

clearOutput()
repeatFor(1 to 5) { e =>
  print(s"${seq(e)} ")
}
```

**Q3a** Does this command use recursion with a command or a function?

**Q3b** Trace through the program to figure out how it calculates the required numbers.

## Self Exploration

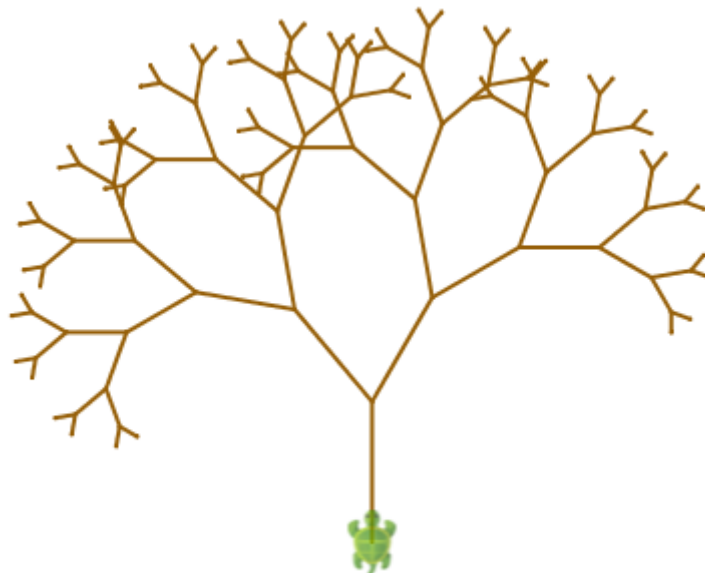
Play with the code above as you see fit.

## Theory

- A recursive command or function is useful when the solution to a problem depends on the solution to another (e.g. smaller) version of the same problem.
- A recursive command or function calls itself.
- Within a recursive command or function, you deal with a couple of different cases:
  - The base case. This handles the version of the problem that does not depend on any another version of the problem. The base case leads to the termination of a recursive solution. Without the base case, a recursive solution will never stop.
  - The recursive case. This handles the version of the problem which depends on another (e.g. smaller) version of the problem.

## Exercise

Write a program to make the something like the following figure:



## 34 Practice with Numeric Patterns

In the activity “More Practice with Patterns”, you created a lot of geometric patterns. In this activity, you will play with numeric patterns. You made geometric patterns using commands. You will make number patterns using functions.

Write programs to print out the following number sequences:

1. 2 4 6 8 10 12

2. 2 7 12 17 22 27

3. 2 4 8 16 32 64

4. 1 2 3 5 8 13 21 34

5. 2 5 9 14 20 27

6. 1 1 2 6 24 120 840

Hint – Write functions that are able to generate the above sequences. Then call the functions to obtain desired values and print these values out. Some functions are best written using recursion. Others can make use of a formula that you discover to capture the sequence.

# 35 Include Files

This activity involves the following:

- Learning to save commonly used code in include files.
- Learning to access the code in include files.
- Playing with the properties of a circle to make an interesting geometric figure.

## Step 1

Type in the following code and run it using the *Run as Worksheet* button:

```
homeDir
```

Q1a What do you think the `homeDir` function does?

## Step 2

Type in the following code and save it (using the `File -> Save As` menu item) in a file called `square.kojo`. Put the file under a directory called `kojo-includes` under your home directory (you saw the location of your home directory in the previous step). If the `kojo-includes` directory does not exist, create it.

```
def square(n: Int) {  
  repeat(4) {  
    forward(n)  
    right(90)  
  }  
}
```

Now, close `square.kojo` (using the `File -> Close` menu item) and type in the following code and run it:

```
// #include ~/kojo-includes/square.kojo  
clear()  
setAnimationDelay(100)  
square(50)  
square(150)
```



**Q2a** The code above does not contain a definition for the square command; how is it still able to use the square command?

**Q2b** What do you think the line with the `// #include` does in the code above?

## Step 3

Change the first line of the code above to:

```
// #include ~/kojo-includes/square-abc.kojo
```

Now, a file named `square-abc.kojo` is not present within the `kojo-includes` directory (we never put it there). So let's see what happens when we run the code. Go ahead and hit the *Run* button.

**Q3a** What error message does Kojo show you? How can you fix this error?

## Self Exploration

Play with the code above as you see fit.

## Theory

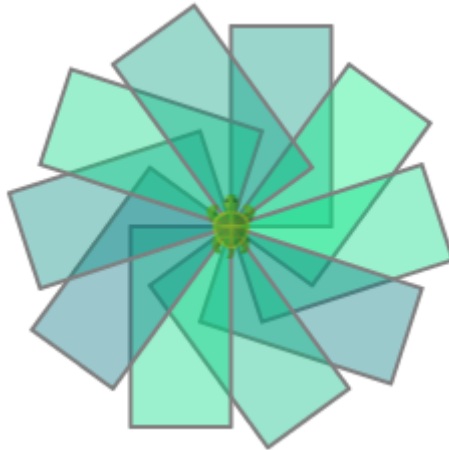
- The `// #include` directive makes Kojo include the specified file in the script editor code, exactly as if the contents of the included file had been present in the script editor.
- You can specify the location of the include file using either an absolute path or a relative path.
- Here are some examples of absolute paths:
  - `c:/users/anusha/kojo-includes/square.kojo` (on Windows)
  - `/home/anusha/kojo-includes/square.kojo` (on Unix/Mac)
- Here are some examples of relative paths:
  - `~/kojo-includes/square.kojo` (relative to the user's home directory)
  - `../kojo-includes/square.kojo` (relative to Kojo's current directory)
- Include files can include other include files.

## Exercise

Define a command that makes rectangles:

```
def rectangle(l: Int, b: Int) {...}
```

Put this command in a file called `rectangle.kojo` in your `kojo-include` folder. Now write a program, which includes this file, to make the following figure:



Hint – make use of the following ideas:

- The figure has a certain number of rectangles that go around and make a complete circle (so  $numRects \times angleBetweenRects = 360$ )
- The fill color of the rectangles has no red component, a random green component, a blue component equal to 128, and an opacity equal to 100.

# 36 Playing MP3 Music

This activity involves the following:

- Learning to play MP3 music.
- Learning to stop MP3 music playback.
- Tracking time using decimal fractions.

## Step 1

To begin playing and controlling MP3 music within Kojo, you need some MP3 music files that you can experiment with. Many such files are available in the Kojo Media bundle. Get these files by doing the following:

- Download the bundle from the [Kojo download page](#). The bundle is a zip file.
- Extract the bundle zip file under your home directory/folder. You will now have a Media folder under your home folder.

## Step 2

Type in the following code and run it:

```
playMp3("~/Media/Sounds/Music Loops/Medieval1.mp3")
```

Q2a What do you think the `playMp3` command does?

## Step 3

Type in the following code and run it:

```
playMp3("~/Media/Sounds/Music Loops/Medieval1.mp3")  
pause(2)  
stopMp3()
```

Q3a What do you think the `pause` command does?

Q3b What do you think the `stopMp3` command does?

## Self Exploration

Play with the code above as you see fit.

## Exercise

Write a program to do the following:

- Play `GuitarChords1.mp3` for 2.6 seconds.
- Play `GuitarChords2.mp3` for 3.4 seconds.
- Repeat the above so that the music plays for a total of 30 seconds.

# 37 Introduction to Electronics and the Arduino platform

For the remaining activities of this book, you are going to write programs that control hardware components – a microcontroller board with attached sensors and actuators. Let's start by looking at what these terms mean, and then move on from there.

- A microcontroller – is an integrated circuit on a chip that contains a CPU, RAM, Flash memory, and programmable input-output lines. A microcontroller runs one program at a time, and its behavior is determined by the program running on it. Here's a brief description of the components of a microcontroller:
  - Flash memory: the microcontroller's "hard disk", where its program is stored.
  - RAM: the microcontroller's working memory, where variables are stored during program execution.
  - CPU: the microcontroller's "brain", which runs the microcontroller's program.
  - Input-output lines: ports that enable the microcontroller to interact with the world.
- A microcontroller board – contains a microcontroller and other related electronic circuitry, including input-output pins that sensors and actuators can plug into.
- Sensors – connect to a microcontroller board's input pins, allowing the microcontroller to sense its environment.
- Actuators – connect to the microcontroller board's output pins, allowing the microcontroller to take action in its environment.

The microcontroller board that you will work with is called the [Arduino](#). You can learn more about Arduino with the help of the [Arduino comic](#).

Here's a brief summary of the information in the comic, along with some related ideas. Don't worry if you don't fully understand the information the first time you read it; keep coming back to this section as you do Arduino based activities – the information here will start to make more and more sense:

- Arduino is an open-source electronics prototyping platform. The platform includes a microcontroller board and a software development environment.
- You can attach sensors or inputs (e.g., switches, light sensors, temperature sensors, pressure sensors, etc.) to an Arduino board via input pins on the board.
- You can attach actuators or outputs (e.g, led bulbs, motors, etc.) to an Arduino board via output pins on the board.

- A program running on an Arduino board takes inputs from attached sensors, decides what to do with the inputs, and then provides outputs using the attached actuators.
- Any input ultimately comes into an Arduino program in the form of a number. Any output is written out by an Arduino program as number. In general, inputs and outputs can be digital or analog. Digital information is discrete (integer based). Analog information is continuous (essentially rational number based).
- An Arduino board has some digital and some analog pins.
- A digital input pin on an Arduino supports the reading, in an Arduino program, of a 0 or 1 value corresponding to a voltage level of 0 or 5 volts on the pin.
- An analog input pin on an Arduino supports the reading, in an Arduino program, of a voltage level in the range of 0-5 volts on the pin, converted to a digital range of 0-1023.
- A digital output pin on an Arduino supports the writing, in an Arduino program, of a 0 or 1 value, which results in a voltage level of 0 or 5 volts on the pin.
- An analog output pin on an Arduino supports the writing, in an Arduino program, of a (simulated) voltage level in the range of 0-5 volts on the pin, using a technique called [PWM](#).
- To do something meaningful with an Arduino board, inputs and outputs are connected to the Arduino pins to form a desired circuit. The connections are made using conducting wires and a breadboard. The atoms in the conducting wires have very loosely bound valence electrons. The moment these electrons encounter a potential difference, they start moving in the form of a current. The Arduino board provides this potential difference, as directed by the program running on the board. These currents cause the inputs to and outputs from the board to work as desired.
- The power and output pins on an Arduino board, in conjunction with a ground pin, are capable of supplying upto 5 volts of potential difference. The Arduino board itself needs to be powered up; one way to do this is by connecting it to a computer with a USB cable.
- Special circuit elements called resistors can be introduced into a circuit to limit the flow of current in the circuit. This might be needed, for example, to prevent a bulb from blowing out with too much current.
- As an example of a useful circuit, you can connect a switch, an LED bulb, and a resistor to Arduino pins, and write a program to make the bulb light up when the switch is pressed.
- As an example of another useful circuit, you can connect a potentiometer (a variable resistance), an LED bulb, and a resistor to Arduino pins, and write a program to make the bulb glow brighter or dimmer as the potentiometer knob is rotated.

# 38 Getting started with Arduino programming in Kojo

## Step 1

Run the Arduino programming story inside Kojo (via the *Tools -> Arduino Programming* menu item). Follow the instructions in the story. By the end of the story, you should have done the following:

1. Uploaded `ka_bridge.ino` to the Arduino board.
2. Saved `ka-bridge.kojo` in the `kojo-includes` directory under your home directory.

## Step 2

Type in the following code and run it:

```
// #include ~/kojo-includes/ka-bridge.kojo

def setup() {
  pinMode(13, OUTPUT)
}

def loop() {
  digitalWrite(13, HIGH)
  delay(1000)
  digitalWrite(13, LOW)
  delay(500)
}
```

**Q2a** What is the result of running the above program? How does the program impact the Arduino board?

**Q2b** What do you think the `pinMode` command does?

**Q2c** What do you think the `digitalWrite` command does?

**Q2d** What do you think the `delay` command does?

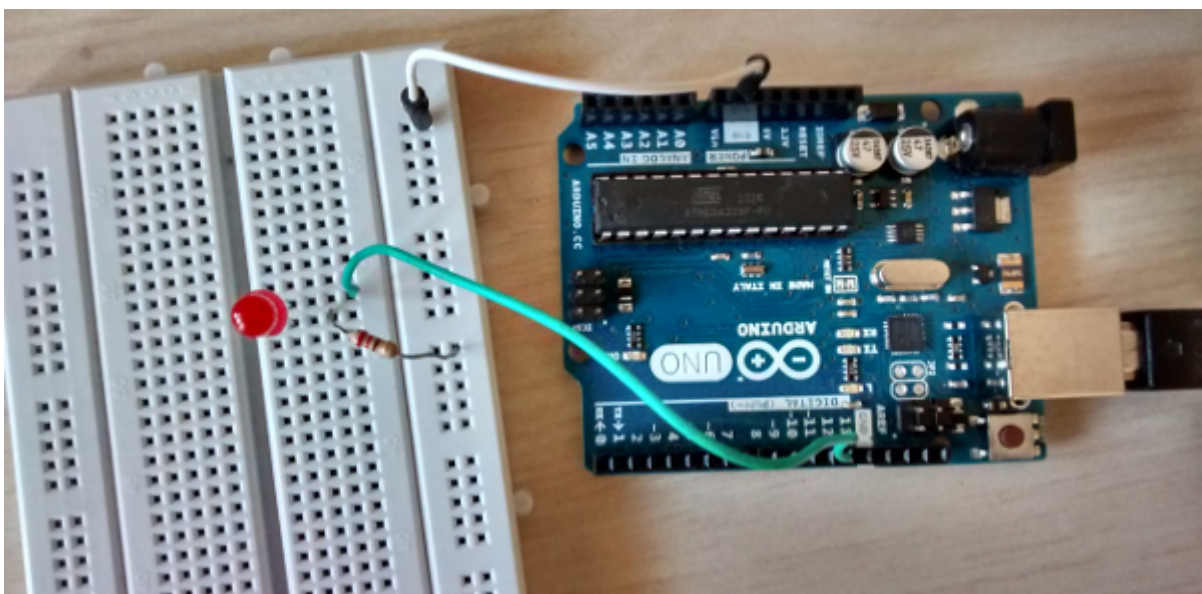
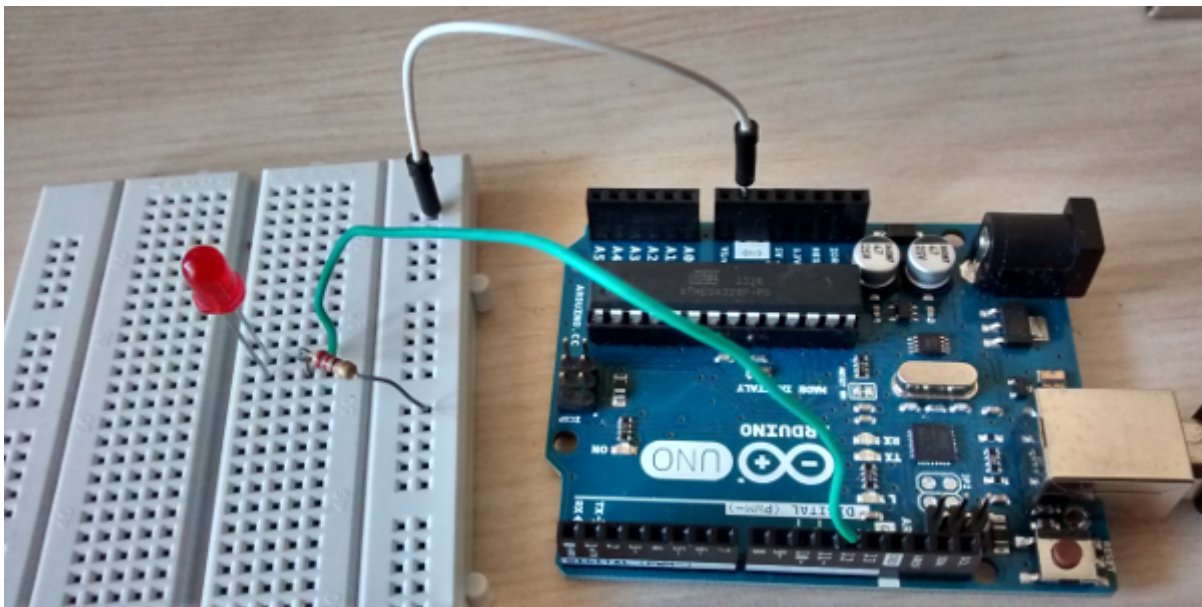
## Self Exploration

Play with the code above as you see fit.

## Exercise

Do the following:

- Create the circuit shown below, as per the following specifications:
  - Hook up an LED to pin 12.
  - Connect it in series with a resistor of  $220\text{ k}\Omega$ .
- Write a program to make the LED blink.





# 39 Lights, Music, Action...

## Step 1

Type in the following code and run it:

```
clear()
var lastClick = epochTimeMillis
onMouseClicked { (x, y) =>
    println(epochTimeMillis - lastClick)
    lastClick = epochTimeMillis
}
```

**Q1a** Click within the drawing canvas. Click again. What do you see in the output pane?

**Q1b** What do the numbers in the output pane represent?

Note – `epochTimeMillis` is a function that returns the number of milliseconds that have elapsed since a reference “epoch” time. The important thing is that the difference between the return values of two calls of `epochTimeMillis` gives you the number of milliseconds that have elapsed *between* those two calls.

## Step 2

Identify an mp3 file on your machine that contains a song (with a nice beat) that you like. Let’s say the file name is `song.mp3`, and it is located in a folder called `Music` under your home folder.

Now, type in the following code and run it:

```
clear()
playMp3("~/Music/song.mp3")
var lastBeat = epochTimeMillis
onMouseClicked { (x, y) =>
    println(epochTimeMillis - lastBeat)
    lastBeat = epochTimeMillis
}
```

Next, do the following:

1. Click on the drawing canvas in time with the beat of the song. You will see numbers getting printed in the output pane.

2. When you have more than ten numbers, stop the program.
3. Take the last ten numbers from output pane and determine their average (using the calculation capability in Kojo).

**Q2a** What does this average number tell you?

## Self Exploration

Play with the code above as you see fit.

## Exercise

1. Do the following:
  - Create a circuit with 2 LEDs (with corresponding  $220\text{ k}\Omega$  resistors) connected to Arduino pins 11 and 12.
  - Write a program to make the LEDs blink in time with the beats of the song that you selected in step 2.
2. Now create a circuit with 4 LEDs. Make these blink to the music in creative ways.

# 40 Turtle Commands Quickref

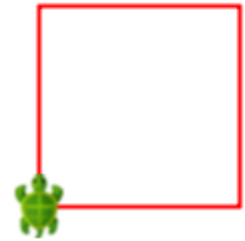
<b>Command</b>	<b>Description</b>
<code>clear()</code>	Clears the turtle canvas, and brings the turtle to the center of the canvas.
<code>forward(steps)</code>	Moves the turtle forward by the given number of steps.
<code>back(steps)</code>	Moves the turtle back by the given number of steps.
<code>right()</code>	Turns the turtle right (clockwise) through ninety degrees.
<code>right(angle)</code>	Turns the turtle right (clockwise) through the given angle in degrees.
<code>right(angle, radius)</code>	Turns the turtle right (clockwise) through the given angle in degrees, along the arc of a circle with the given radius.
<code>left()</code> , <code>left(angle)</code> , <code>left(angle, radius)</code>	These commands work in a similar manner to the corresponding <code>right()</code> commands.
<code>setPosition(x, y)</code>	Places the turtle at the point (x, y) without drawing a line. The turtle's heading is not changed.
<code>changePosition(x, y)</code>	Changes the turtle's position by the given x and y without drawing a line.
<code>dot(diameter)</code>	Makes a dot with the given diameter.
<code>setAnimationDelay(delay)</code>	Sets the turtle's speed. The specified delay is the amount of time (in milliseconds) taken by the turtle to move through a distance of one hundred steps. The default delay is 1000 milliseconds (or 1 second).
<code>setPenColor(color)</code>	Sets the color of the pen that the turtle draws with.
<code>setPenThickness(size)</code>	Sets the width of the pen that the turtle draws with.
<code>setFillColor(color)</code>	Sets the fill color of the figures drawn by the turtle.

<b>Command</b>	<b>Description</b>
<code>setBackground(color)</code>	Sets the canvas background to the specified color. You can use predefined colors for setting the background, or you can create your own colors using the <code>Color</code> , <code>ColorHSB</code> , and <code>ColorG</code> functions.
<code>penUp()</code>	Pulls the turtle's pen up, and prevents it from drawing lines as it moves.
<code>penDown()</code>	Pushes the turtle's pen down, and makes it draw lines as it moves. The turtle's pen is down by default.
<code>hop(steps)</code>	Moves the turtle forward by the given number of steps with the pen up, so that no line is drawn. The pen is put down after the hop.
<code>cleari()</code>	Clears the turtle canvas and makes the turtle invisible.
<code>invisible()</code>	Hides the turtle.
<code>savePosHe()</code>	Saves the turtle's current position and heading, so that they can easily be restored later with a <code>restorePosHe()</code> .
<code>restorePosHe()</code>	Restores the turtle's current position and heading based on an earlier <code>savePosHe()</code> .
<code>saveStyle()</code>	Saves the turtle's current style, so that it can easily be restored later with <code>restoreStyle()</code> . The turtle's style includes: pen color, pen thickness, fill color, pen font, and pen up/down state
<code>restoreStyle()</code>	Restores the turtle's style based on an earlier <code>saveStyle()</code> .
<code>write(obj)</code>	Makes the turtle write the specified object as a string at its current location.
<code>setPenFontSize(n)</code>	Specifies the font size of the pen that the turtle writes with.
<code>Font(name, size)</code> <code>Font(name, size, style)</code>	Creates a font with the give name, size, and style. Permissible styles are <code>PlainFont</code> , <code>BoldFont</code> , and <code>ItalicFont</code> .
<code>setPenFont(font)</code>	Sets the font that the pen writes with. The font can be created using the <code>Font</code> function.
<code>TexturePaint(fileName, x, y)</code>	Creates a paint that can be used as a color (in <code>setFillColor</code> , <code>setPenColor</code> , etc). The paint is based on the texture image in the given filename, and is anchored at the given x and y while filling a shape.

# 41 Exercise Solutions

## 41.3 Commands and Programs

```
clear()  
forward(100)  
right()  
forward(100)  
right()  
forward(100)  
right()  
forward(100)  
right()
```



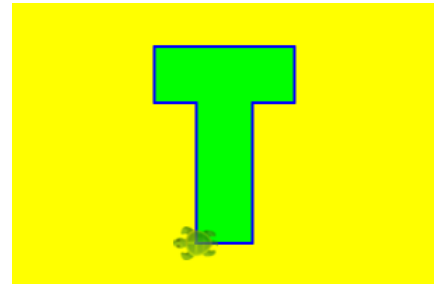
## 41.4 Using Kojo Effectively

```
clear()  
forward(100)  
right()  
forward(40)  
right()  
forward(100)  
right()  
forward(40)  
right()  
  
right()  
  
forward(100)  
right()  
forward(40)  
right()  
forward(100)  
right()  
forward(40)  
right()
```

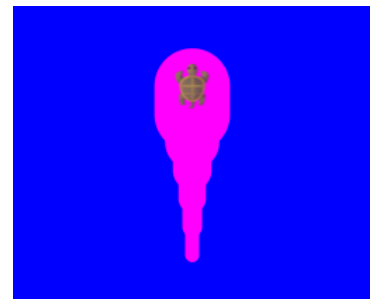


## 41.5 Drawing with Colors

```
clear()
setBackground(yellow)
setPenColor(blue)
setFillColor(green)
forward(100)
left()
forward(30)
right()
forward(40)
right()
forward(100)
right()
forward(40)
right()
forward(30)
left()
forward(100)
right()
forward(40)
```



```
clear()
setBackground(blue)
setPenColor(magenta)
setPenThickness(10)
forward(20)
setPenThickness(14)
forward(20)
setPenThickness(19.6)
forward(20)
setPenThickness(27.4)
forward(20)
setPenThickness(38.4)
forward(20)
setPenThickness(53.8)
forward(20)
```



## 41.6 Mixing Colors With a Function

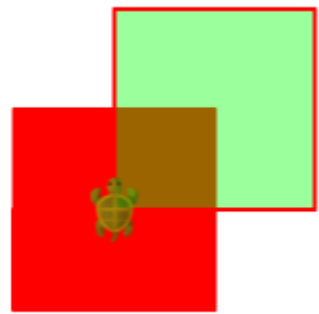
```
clear()
setFillColor(Color(50, 100, 150, 200))
left()
forward(50)
right()
forward(30)
right()
forward(50)
left()
forward(50)
right()
forward(30)
right()
forward(50)
left()
forward(50)
right()
forward(30)
right()
forward(50)
left()
forward(50)
right()
forward(30)
right()
forward(50)
```



```
clear()
setFillColor(red)
repeat(4) {
  forward(100)
  right(90)
}

forward(50)
right(90)
forward(50)
left(90)

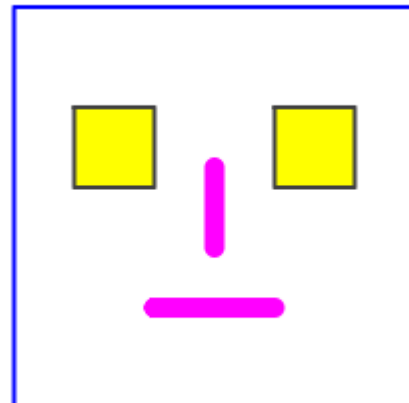
setFillColor(Color(0, 255, 0, 100))
repeat(4) {
  forward(100)
  right(90)
}
```





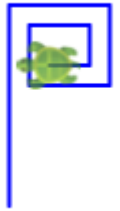
## 41.7 Hopping with Speed

```
clear()
setAnimationDelay(10)
setPenColor(blue)
forward(200)
right()
forward(200)
right()
forward(200)
right()
forward(200)
right()
forward(200)
right()
hop(150)
right()
hop(30)
setPenColor(darkGray)
setFillColor(yellow)
forward(40)
right()
forward(40)
right()
forward(40)
right()
forward(40)
right()
forward(40)
right()
hop(100)
forward(40)
right()
forward(40)
right()
forward(40)
right()
forward(40)
right()
forward(40)
right()
hop(100)
right()
setPenColor(magenta)
setPenThickness(10)
forward(60)
back(30)
right()
hop(30)
forward(40)
invisible()
```



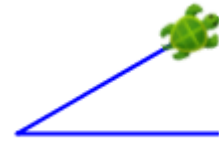
## 41.8 Digging Deeper with Tracing

```
clear()
setAnimationDelay(10)
setPenColor(blue)
forward(100)
right(90)
forward(50)
right(90)
forward(40)
right(90)
forward(40)
right(90)
forward(30)
right(90)
forward(30)
right(90)
forward(20)
right(90)
forward(20)
```



## 41.9 Angles

```
clear()
setAnimationDelay(100)
setPenColor(blue)
left()
forward(100)
right(150)
forward(100)
```



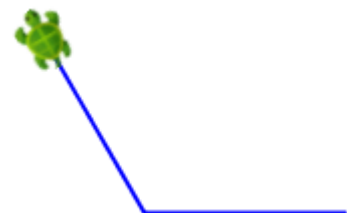
```
clear()
setAnimationDelay(100)
setPenColor(blue)
left()
forward(100)
right(120)
forward(100)
```



```
clear()
setAnimationDelay(100)
setPenColor(blue)
left()
forward(100)
right(90)
forward(100)
```



```
clear()
setAnimationDelay(100)
setPenColor(blue)
left()
forward(100)
right(60)
forward(100)
```

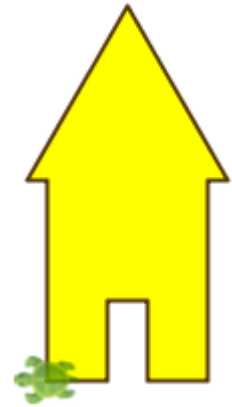


```
clear()
setAnimationDelay(100)
setPenColor(blue)
left()
forward(100)
right(0)
forward(100)
```



## 41.10 Practice

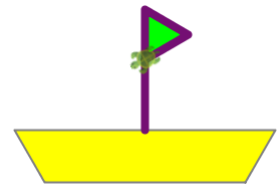
```
clear()
setAnimationDelay(100)
setPenColor(brown)
setFillColor(yellow)
forward(100)
left()
forward(10)
right(120)
forward(100)
right(120)
forward(100)
right(120)
forward(10)
left()
forward(100)
right()
forward(30)
right()
forward(40)
left()
forward(20)
left()
forward(40)
right()
forward(30)
```



```

clear()
setAnimationDelay(100)
setPenColor(gray)
setFillColor(yellow)
left()
forward(200)
right(60)
forward(60)
right(90 + 30)
forward(260)
right(120)
forward(60)
right(60)
hop(100)
right()
hop(60 * 0.866)
setPenThickness(8)
setPenColor(purple)
setFillColor(green)
forward(120)
right(120)
forward(50)
right(120)
forward(50)

```



## 41.14 Turning with a radius

```

clear()
setAnimationDelay(0)
setBackground(Color(0, 72, 199))
setPenThickness(3)
setPenColor(white)
setFillColor(Color(255, 44, 53))
right(180, 50)
right(90)
right(180, 50)
right(90)
right(180, 50)
right(90)
right(180, 50)

```



```

clear()
setAnimationDelay(10)
setBackground(Color(255, 133, 43))
setPenColor(black)
setPenThickness(3)
repeat(9) {
    right(90)
    right(60, 30)
    left(120)
    right(60, 30)
    hop(5)
    left(90)
    hop(5)
}

```

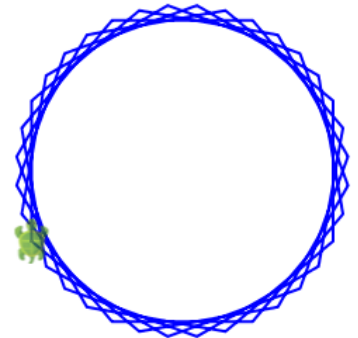


## 41.15 More Fun with Repeat

```

clear()
setAnimationDelay(10)
setPenColor(blue)
repeat(36) {
    forward(100)
    right(50)
}

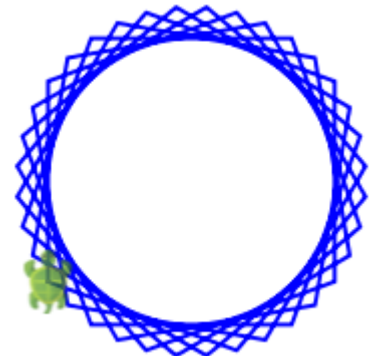
```



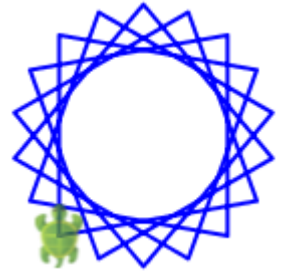
```

clear()
setAnimationDelay(10)
setPenColor(blue)
repeat(36) {
    forward(100)
    right(70)
}

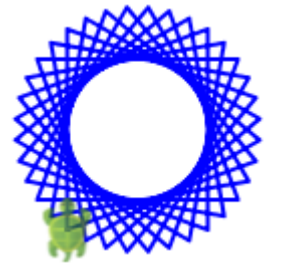
```



```
clear()
setAnimationDelay(10)
setPenColor(blue)
repeat(18) {
  forward(100)
  right(100)
}
```



```
clear()
setAnimationDelay(10)
setPenColor(blue)
repeat(36) {
  forward(100)
  right(110)
}
```



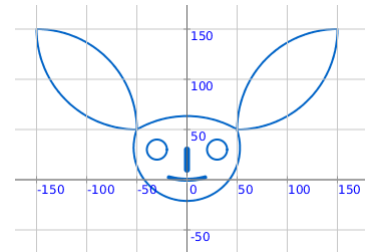
## 41.16 Repeat with a sequence

```
clear()
setAnimationDelay(100)
repeatFor(0 to 5) { counter =>
  forward(20 + counter * 10)
  right()
}
```



## 41.18 Absolute position and heading

```
clear()
showAxes()
showGrid()
setAnimationDelay(10)
setPenColor(Color(0, 100, 200))
setPosition(50, 50)
setHeading(0)
left(90, 100)
left(90)
left(90, 100)
setPosition(-50, 50)
setHeading(90)
left(90, 100)
left(90)
left(90, 100)
setPosition(50, 50)
setHeading(150)
left(60, 100)
setHeading(250)
left(220, 53)
setHeading(90)
setPosition(40, 30)
left(360, 10)
setPosition(-20, 30)
left(360, 10)
setPenThickness(4)
setPosition(0, 0)
setHeading(180)
right(15, 70)
setPosition(0, 0)
setHeading(0)
left(15, 70)
setPenThickness(6)
setPosition(0, 10)
setHeading(90)
forward(20)
invisible()
```





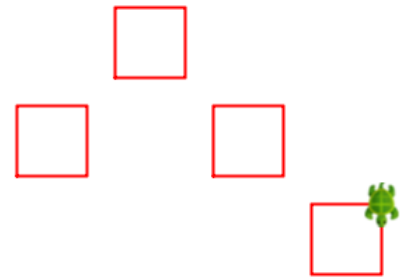
## 41.20 Random Numbers and Named Values

```
clear()
setAnimationDelay(10)
setPenColor(gray)
setFillColor(darkGray)
setBackground(Color(255, 228, 46))
val width = 40
repeat(20) {
    val len = random(200) + 30
    repeat(2) {
        forward(len)
        right()
        forward(width)
        right()
    }
    right()
    hop(width)
    left()
}
```



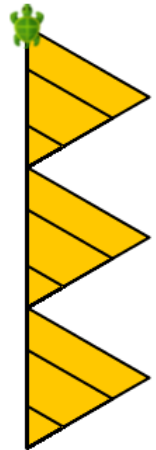
## 41.21 Your own Commands

```
clear()
setAnimationDelay(100)
def square() {
  repeat(4) {
    forward(50)
    right()
  }
}
square()
hop(70)
right()
hop(70)
left()
square()
right()
hop(70)
right()
hop(70)
right(180)
square()
right()
hop(120)
right()
hop(20)
square()
```



## 41.22 Your own Commands, with Inputs

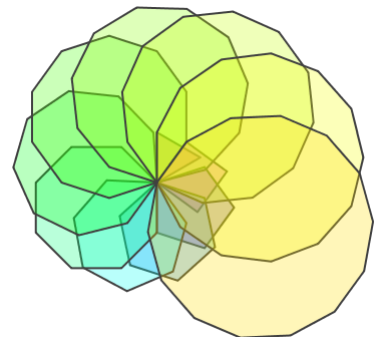
```
clear()
setAnimationDelay(100)
setPenColor(black)
setFillColor(orange)
def triangle(n: Int) {
  repeat(3) {
    forward(n)
    right(120)
  }
}
repeat(3) {
  triangle(100)
  triangle(70)
  triangle(30)
  forward(100)
}
```



## 41.23 Polygon Art

```
def polygon(n: Int) {
  repeat(n) {
    forward(50)
    right(360.0 / n)
  }
}

clear()
setAnimationDelay(10)
setPenColor(darkGray)
var fill = Color(255, 0, 21, 70)
repeatFor(3 to 14) { n =>
  setFillColor(fill)
  polygon(n)
  right(36)
  fill = hueMod(fill, -0.16)
}
invisible()
```



## 41.27 Your own Functions

```
val P = 100
val R = 50
def si(p: Double, r: Double) = p * r / 100
def amt(p: Double, r: Double, t: Int) = {
    p + si(p, r) * t
}

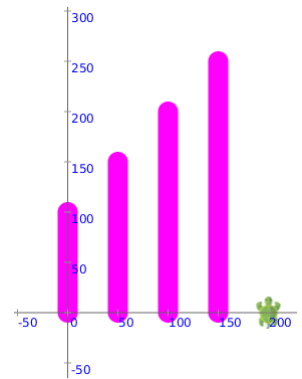
clear()
setAnimationDelay(100)
showAxes()
setPenThickness(20)
setPenColor(magenta)

savePosHe()
forward(amt(P, R, 0))
restorePosHe()
right()
hop(50)
left()

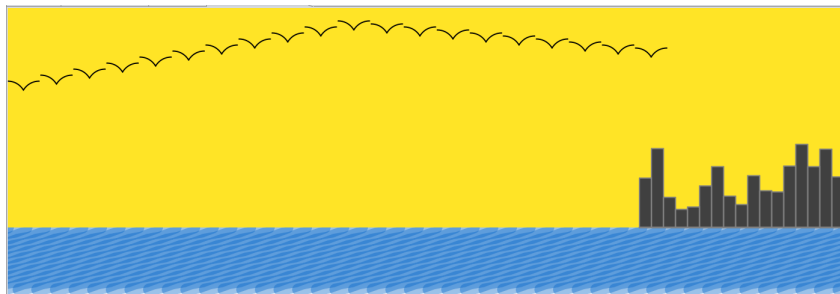
savePosHe()
forward(amt(P, R, 1))
restorePosHe()
right()
hop(50)
left()

savePosHe()
forward(amt(P, R, 2))
restorePosHe()
right()
hop(50)
left()

savePosHe()
forward(amt(P, R, 3))
restorePosHe()
right()
hop(50)
left()
```



## 41.28 Mini Project



```
def bird() {
  right(90)
  right(60, 30)
  left(120)
  right(60, 30)
}

def rect(h: Double, w: Double) {
  repeat(2) {
    forward(h)
    right(90)
    forward(w)
    right(90)
  }
}

def stroke(w: Int, l: Int) {
  savePosHe()
  setPenThickness(w)
  right(90 + 70)
  val angle = 10
  repeat(l / w) {
    forward(w)
    right(angle)
    forward(w)
    left(angle)
  }
  restorePosHe()
}

clear()
setAnimationDelay(0)
val cb = canvasBounds
left(90)
hop(cb.width / 2)
```

```

left(90)
hop(cb.height / 4)
left(90)
savePosHe()
setPenColor(Color(0, 100, 200, 100))
repeat(100) {
    stroke(20, 200)
    hop(20 + 10 + 1)
}
restorePosHe()
left(90)

setPenColor(noColor)
setFillColor(Color(255, 228, 38))
rect(cb.height * 3 / 4, cb.width)

savePosHe()

setPenColor(black)
setPenThickness(2)
hop(cb.height / 2)
repeat(10) {
    bird()
    hop(3)
    left(90)
    hop(10)
}

repeat(10) {
    bird()
    hop(3)
    left(90)
    hop(-5)
}

restorePosHe()
right(90)
hop(cb.width * 3 / 4)
left()
setFillColor(darkGray)
setPenColor(gray)
repeat(20) {
    rect(20 + random(cb.height.toInt / 4), 20)
    right(90)
    forward(20)
}

```

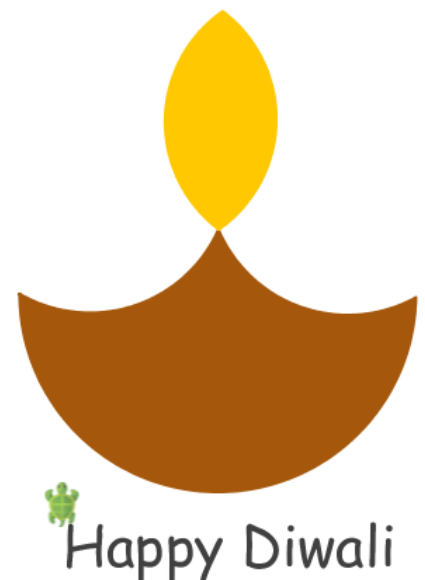
```
    left(90)
}
```

## 41.29 Strings and I/O

```
clearOutput()
val n1 = readInt("What's the first number?")
val n2 = readInt("What's the second number?")
val average = (n1 + n2)/2.0
println("The average of $n1 and $n2 is $average")
```

## 41.30 Artistic Text in the Canvas

```
clear()
setPenColor(Color(165, 88, 11))
setFillColor(Color(165, 88, 11))
setAnimationDelay(10)
left(240)
left(96, 100)
right(133)
left(96, 100)
right(120)
right(179,141)
right(72)
hop(149)
left(72 + 53)
setPenColor(orange)
setFillColor(orange)
right(108, 96)
right(72)
right(108, 96)
setPenColor(darkGray)
setPenFont(Font("Comic Sans MS", 40))
setPosition(30, -150)
setHeading(90)
write("Happy Diwali")
```



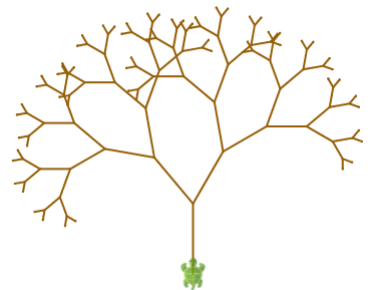
## 41.31 Conditionals

```
def rect(h: Int, w: Int) {
  repeat(2) {
    forward(h)
    right()
    forward(w)
    right()
  }
}
clear()
val shape = readInt("Shape? Enter 1 for square, 2 for rectangle")
if (shape == 1) {
  rect(100, 100)
}
else if (shape == 2) {
  rect(70, 140)
}
```

## 41.33 Recursion

```
def tree(n: Int) {
  savePosHe()
  if (n < 10) {
    forward(n)
  }
  else {
    forward(n)
    right(30)
    tree(n - 10)
    left(70)
    tree(n - 10)
  }
  restorePosHe()
}

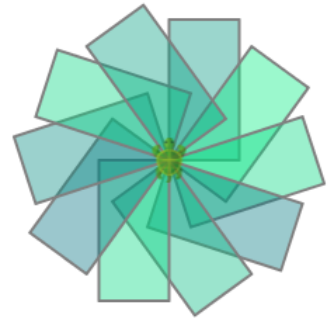
clear()
setAnimationDelay(10)
setPenColor(Color(150, 95, 8))
tree(70)
```





## 41.35 Include Files

```
// #include ~/kojo-includes/rectangle.kojo
clear()
setAnimationDelay(10)
setPenColor(gray)
repeat(10) {
    setFillColor(Color(0,random(256),128,100))
    rectangle(100, 50)
    right(36)
}
```



## 41.36 Playing MP3 Music

```
repeat(5) {
    playMp3("~/Media/Sounds/Music Loops/GuitarChords1.mp3")
    pause(2.6)
    stopMp3()
    playMp3("~/Media/Sounds/Music Loops/GuitarChords2.mp3")
    pause(3.4)
    stopMp3()
}
```

## Licensing Terms



License: Creative Commons *Attribution-NonCommercial-ShareAlike 4.0 International*  
[CC BY-NC-SA 4.0](#)

© 2010–2015 Lalit Pant (lalit@kogics.net)

<http://www.kogics.net>

This publication can be used freely in a Non-Commercial setting as per the above license.

Commercial use of this publication is restricted as per the terms below:

No part of this publication may be reproduced, modified, distributed, stored in a retrieval system, republished, displayed, or performed – for money or compensation of any other kind, either solely or as part of a larger offering or service – without a prior written agreement with Kogics.