

Adventures with Kojo

Exploring Programming, Math, and
Art

Level-2

Play . Create . Learn

www.kogics.net

Credits

By

Lalit Pant

With contributions and feedback from

Anusha Pant, REACHA (www.reacha.org), and the kids at the Kalpana Center

Copyright © 2010–2014 Kogics (www.kogics.net)

This publication is licensed under the

[Creative Commons Attribution-NonCommercial-ShareAlike 3.0 License](https://creativecommons.org/licenses/by-nc-sa/3.0/).

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

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.

If you make a modified version of this publication for Non-Commercial use - this page, including the above notices, must be retained at the front of the publication.

Book version: Aug 11, 2014

Table of Contents

Activity 1 – Commands and Programs.....	7
Self Exploration.....	8
Theory.....	8
Exercise.....	9
Activity 2 – Using the Kojo Environment Effectively.....	10
Self Exploration.....	10
Exercise.....	11
Activity 3 – Pen and Fill Color.....	12
Self Exploration.....	12
Exercise.....	13
Activity 4 – Mixing Colors with a function.....	14
Self Exploration.....	14
Theory.....	15
Exercise.....	16
Activity 5 – Hopping with Speed.....	17
Self Exploration.....	17
Exercise.....	18
Activity 6 – Angles.....	19
Self Exploration.....	19
Exercise.....	20
Activity 6a – Practice.....	21
Activity 7 – Repeating commands.....	22
Self Exploration.....	22
Exercise.....	22
Activity 8 – Analyzing the Repeat command.....	23
Self Exploration.....	23
Theory.....	24
Exercise.....	24
Activity 9 – Repeating to make a pattern.....	25
Self Exploration.....	27
Exercise.....	28
Activity 9a – Pattern Drawing Practice.....	29
Activity 9b – Extra Practice with Patterns.....	30
Activity 9c – Extra Practice with Angle based Patterns.....	31
Activity 10 – Calculations.....	32
Self Exploration.....	32
Exercise.....	33
Activity 11 – Your own commands.....	34
Self Exploration.....	35
Theory.....	35
Exercise.....	37
Activity 12 – Named Values.....	38
Self Exploration.....	39
Theory.....	39
Exercise.....	40

Activity 13 – Your own dynamic commands.....	41
Self Exploration.....	42
Theory.....	42
Exercise.....	43
Activity 14 – Mini Project.....	44
Activity 15 – Break Free.....	45
Activity 16 – Strings and I/O.....	46
Self Exploration.....	46
Theory.....	46
Exercise.....	46
Activity 17 – Conditionals.....	47
Theory.....	47
Activity 18 – Repeat with a counter.....	48
Self Exploration.....	49
Theory.....	49
Exercise.....	51
Activity 19 – Your own functions.....	52
Self Exploration.....	52
Theory.....	53
Exercise.....	54
Activity 20 – Classes.....	55
Self Exploration.....	55
Theory.....	55
Exercise.....	56
Activity 21 – Sequences.....	58
Self Exploration.....	58
Exercise.....	58
Activity 22 – Maps.....	59
Self Exploration.....	59
Exercise.....	59
Activity 23 – Variables.....	60
Self Exploration.....	60
Exercise.....	61
Solutions.....	62
Activity 1 – Commands and Programs.....	62
Activity 2 – Using the Kojo Environment Effectively.....	62
Activity 3 – Pen and Fill Color.....	63
Activity 4 – Mixing Colors with a function.....	63
Activity 5 – Hopping with Speed.....	64
Activity 6 – Angles.....	65
Activity 6a – Practice.....	66
Activity 7 – Repeating commands.....	67
Activity 8 – Analyzing the Repeat Command.....	67
Activity 9 – Repeating to make a pattern.....	68
Activity 9a – Pattern Drawing Practice.....	69
Activity 9b – Extra Practice with Patterns.....	70
Activity 9c – Extra Practice with Angle based Patterns.....	71

Activity x – Repeating to make a pattern.....	72
Activity 21 – Sequences.....	73
Activity 22 – Maps.....	73

Prerequisites

Before you start reading this book, make sure that you have read (or at least browsed through) the Kojo introduction book (*Kojo, An Introduction*) – which can be freely downloaded from:

<http://www.kogics.net/kojo-ebooks>

At the very least, you should be familiar with:

- The different windows within Kojo.
- The buttons in the Script Editor toolbar.
- The actions available within the context menus of the Script Editor and the Drawing Canvas.

Activity 1 – Commands and Programs

This activity involves the following:

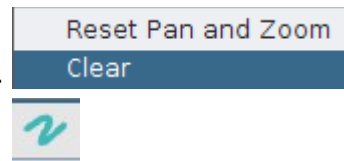
- Learning about commands, actions, and programs.
- Learning the `clear()`, `forward()`, and `right()` commands, and using them to make a square geometrical figure.
- Using the Kojo error recovery feature.

Step 1. Type in the following code within the Script Editor and run it:

```
forward(100)
```

Q1. What does the turtle do?

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:

```
showScale()  
forward(200)
```

Q2a. What does the turtle do?

Q2b. What do you think the `forward(someInput)` command does? What does the input to the command specify (feel free to experiment with different inputs to the command to validate your answer)?

Q2c. What do you think the `showScale()` command does? Does it show you the unit of length for drawing on the turtle canvas? This unit of length is called a pixel.

Step 3. Clear the Drawing Canvas and Script Editor. Then type in the following code and run it:

```
right()
```

Q3a. What does the turtle do?

Q3b. What do you think the `right()` command does?

Step 4. Clear the Script Editor. Then type in the following code and run it:

```
clear()
```

Q4. What does the `clear()` command do? How is it useful?

Step 5. Type in the following incorrect code and run it:

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

Q5. What does Kojo tell you? Observe the kind of message that Kojo shows you when you give it an incorrect command to run. Using this message, can you determine (and go to) the line in your program that has the problem?

Step 6. Type in the following code and run it. **Q6a.** But first guess (before running the code) what figure is made by this program:

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

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.

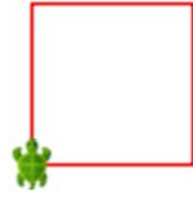
Theory

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.

Exercise

Write a program to make the given figure. The length of the sides of the square is 100 units.



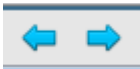


Activity 2 – Using the Kojo Environment Effectively

This activity involves the following:

- Exploring program tracing, error recovery, and history navigation.
- Practicing the selection, copying, cutting, and pasting of program text.
- Exploring code completion.

Step 1. Practice the following based on the square making program from the previous activity.

1. Program Tracing (via the *Trace Script* toolbar button). 
2. Error Recovery (via the *Check Script* toolbar button). 
3. History Navigation – via the *History Previous* and *Next* toolbar buttons or the History Pane. 
4. 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).
5. 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.
6. 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.
7. 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.
8. Code Completion (Ctrl+Space) – to help you to write your code more efficiently.

Q1. What does each of the above features do?

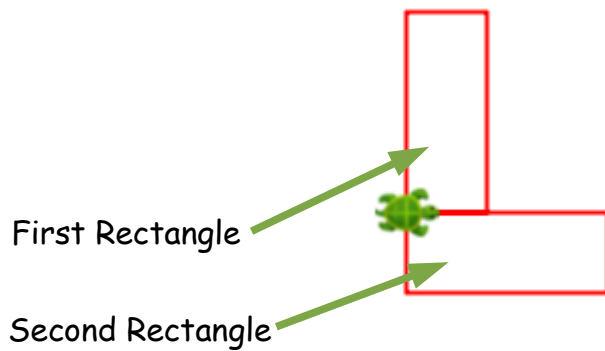
Self Exploration

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

Exercise

Write a program to make the figure below. Use copy-and-paste and code-completion along the way. The dimensions of the two rectangles are:

$$\text{length} = 120 \text{ units} \quad ; \quad \text{breadth} = \frac{1}{3} \text{ of length}$$



Activity 3 – Pen and Fill Color

This activity involves the following:

- Learning about pen and fill colors.
- Learning the `left()` command.
- Applying the idea of ratio and proportion in constructing a figure made out of two rectangles.

Step 1. Type in the following code and run it (use copy-and-paste and code-completion along the way):

```
clear()
setPenColor(blue)
setFillColor(green)
forward(100)
left()
forward(50)
right()
forward(50)
right()
forward(100)
```

Q1a. What do you think the `setPenColor()` command does (*as you answer questions like this, feel free to play around with the given command*)? What does the input to the command specify?

Q1b. What do you think the `setFillColor()` command does? What does the input to the command specify?

Q1c. What do you think the `left()` command does?

Self Exploration

Play with the inputs to the `setPenColor()`, `setFillColor()`, and `forward()` commands in the code above. See how changing these inputs modifies the figure.

Exercise

Write a program to make the given figure. The pen color is blue, and the fill color is green.

To determine the dimensions of the figure, imagine that it is made out of two rectangles – a vertical one and a horizontal one:

- Vertical rectangle – $length=120$, ratio of $breadth : length=1 : 3$
- Horizontal rectangle – $length=90$, ratio of $breadth : length$ is *in the same proportion* as the corresponding ratio for the vertical rectangle.



Activity 4 – 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 numbers related to 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):

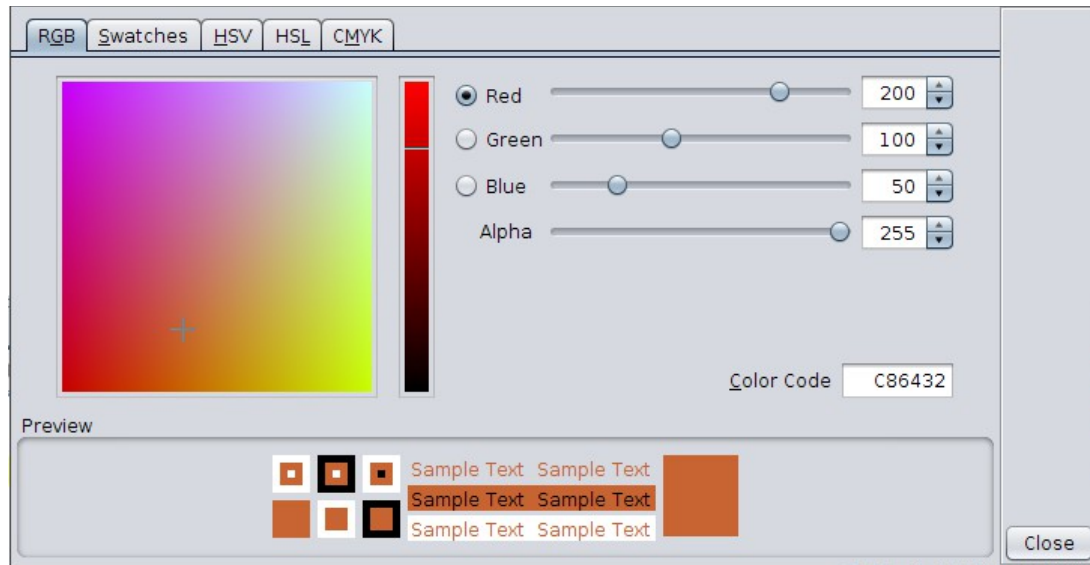
```
clear()
setPenColor(blue)
setFillColor(Color(200, 100, 50, 255))
forward(100)
left()
forward(50)
right()
forward(50)
right()
forward(100)
```

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

Q1b. What do the four different inputs to the `Color` function specify? Play with different input values to try to answer this.

Self Exploration

Play with the inputs to the `Color` function in the code above. See how changing these inputs modifies the figure. Also, click on the word `Color` in your Script Editor. This will bring up a color chooser (shown in the next page). 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. The component values need to be in the range 0-255. Do a Google search for "RGB color" to learn more about how colors are represented by these four components inside computers.

Exercise

Write a program to make the given figure with the following specifications:

Cross arm – length=50 , ratio of breadth:length is 3:5

Fill color – ratio of red:green:blue:opacity is 1:2:3:4 ,
blue=150



Activity 5 – Hopping with Speed

This activity involves the following:

- Learning how to control the speed of the turtle and the thickness of the lines drawn by it.
- Learning how to make the turtle move without drawing lines.
- Learning the `setAnimationDelay`, `setPenThickness`, and `hop` 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)
```

Q1a. What do you think the `setAnimationDelay()` command does? What does the input to the command specify?

Q1b. What do you think the `setPenThickness()` command does? What does the input to the command specify?

Q1c. What do you think the `hop()` command does? What does the input to the command specify?

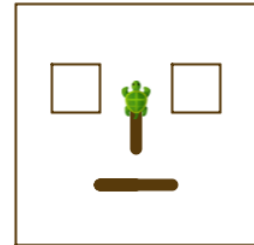
Self Exploration

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 given figure. The pen color is brown. The outline of the face is a square with a length of 200. Use your best judgment to estimate the other dimensions in the figure.



Activity 6 – Angles

This activity involves the following:

- Learning how to make the turtle turn through angles other than 90° .
- Exploring angles.
- Using the idea of supplementary angles.

Step 1. Type in the following code and run it:

```
clear()
showProtractor()
repeat (3) {
  forward(100)
  right(120)
}
```

Q1a. What do you think the input to the `right()` command above specifies?

Q1b. What do you think the `showProtractor()` command does? Does it help you to get an idea of the sizes of different angles?

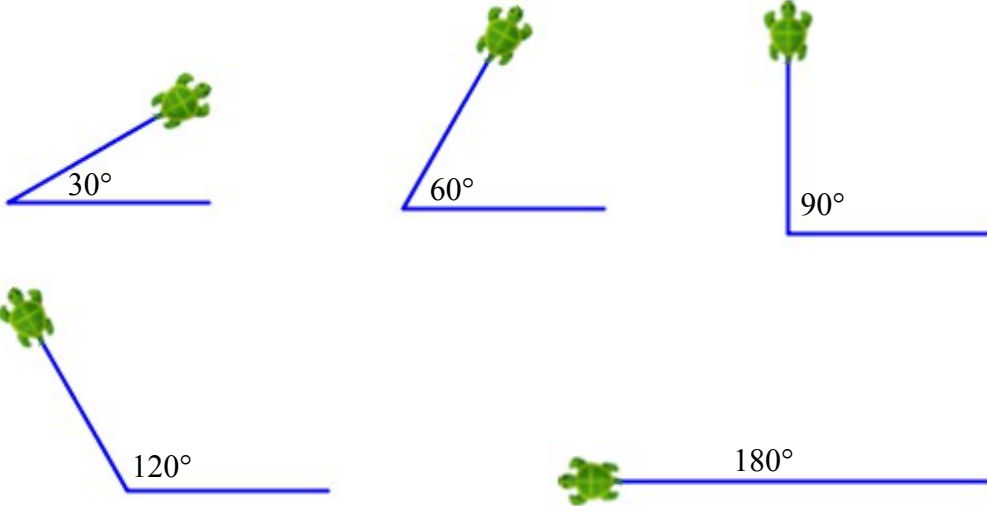
Q1c. The angles of an equilateral triangle are 60° . Why does the turtle turn through 120° to make the above equilateral triangle?

Self Exploration

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

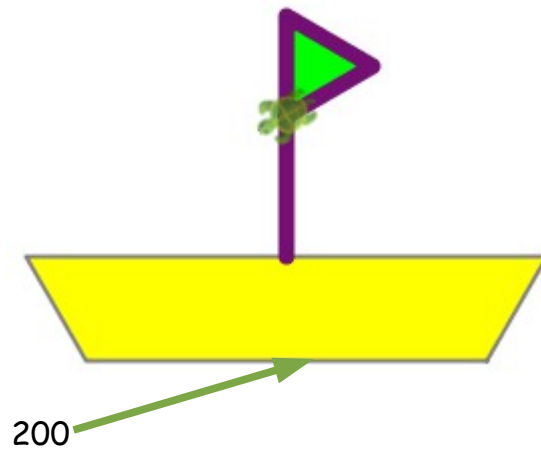
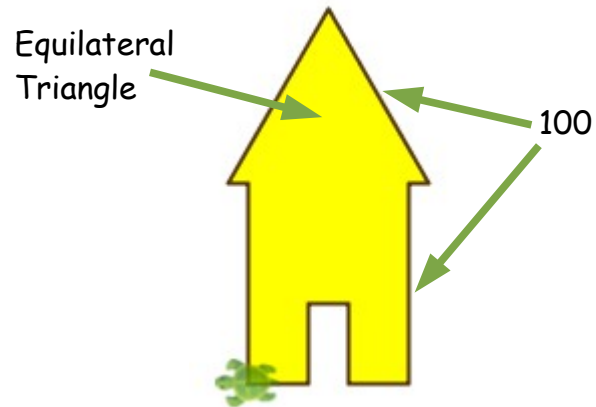
Exercise

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



Activity 6a – Practice

Write programs to make the following figures:



Activity 7 – Repeating commands

This activity involves the following:

- Learning the `repeat` command.
- Learning about removing code duplication/repetition by using the `repeat` command.

Step 1. Type in the following code and run it:

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

Q1a. What do you think the `repeat()` command does?

Q1b. How many inputs does the `repeat()` command take? What do these inputs signify?

Self Exploration

Play with the inputs to the `repeat()` and `forward()` commands in the code above. See how changing these inputs modifies the figure.

Exercise

Write a program, using the `repeat` command, to make the following figure. The length of the square is 100 units. The pen color is gray, and the fill color is orange.



Activity 8 – Analyzing the Repeat command

This activity involves the following:

- Gaining a deeper understanding of the repeat command.

Step 1. Type in and run the following programs:

```
clear()
repeat (4) {
  forward(100)
  right()
}
```

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

Q1a. What's the figure made by the program on the left?

Q1b. What's the figure made by the program on the right?

Q1c. How are these programs similar?

Q1d. How are these programs different?

Q1e. Is there an easy, mechanical way of converting from one to the other?

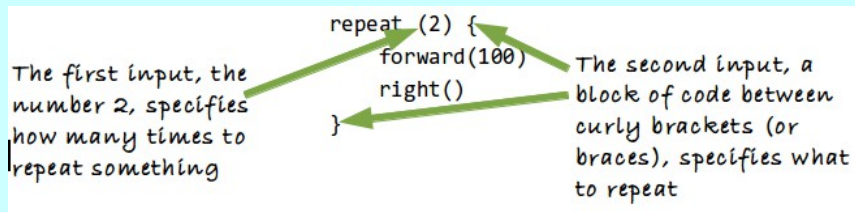
Self Exploration

Play with the inputs to the `repeat()` and `forward()` commands in the code on the left. See how changing these inputs modifies the figure. For changes on the left, think about how you would need to modify the program on the right to achieve a similar effect.

Theory

The `repeat()` command allows you to run other commands for a specified number of times.

Note that `repeat()` takes two inputs:

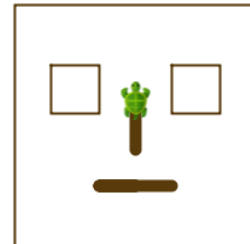


The `repeat()` command has a couple of big benefits:

- It makes your programs shorter by removing repetition.
- It makes your programs easier to understand.

Exercise

Write a program, using the repeat command, to make the figure that you made in activity 4. The only difference from that figure is that the outline of the face is a square with a length of 300 instead of 200.



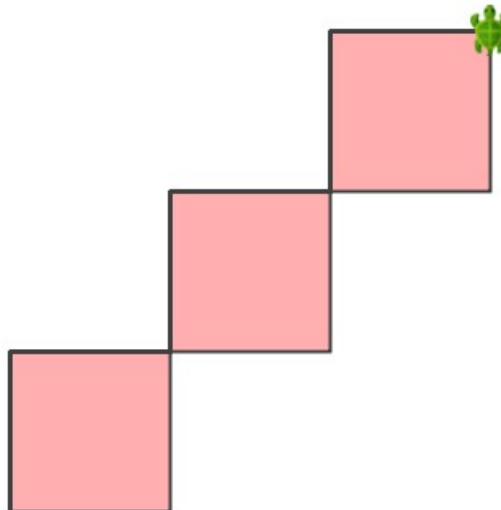
Activity 9 – Repeating to make a pattern

This activity involves the following:

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

Step 1. Learn the procedure for making patterns.

Let's say you want to make a figure like the one below. The pen color is dark gray, and the fill color is pink. The length of the square is 100 units.



Do you see a pattern here?



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, as we'll see...

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 Position and a 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 Position is the *top-right corner* of the square, and Direction is *north*

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 turtle position (marked with a circle on your paper sheet), and a turtle direction (marked with an arrow on your paper sheet).
2. Draw the building-block in Kojo
3. Repeat the building-block, for as many times as the pattern requires, using the repeat command. To do this, you should *Cut* the code for drawing the building-block from Step 2 above – and *Paste* it inside the repeat command curly brackets.

Step 2. 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.

```
clear()
setAnimationDelay(100)
setPenColor(darkGray)
setFillColor(pink)
repeat(3) { // The stuff inside this repeat command is the pattern
  building-block
  // start of building-block
  repeat(4) { // The building-block itself uses the repeat command
    forward(100)
    right()
  }
  hop(100)
  right()
  hop(100)
  left()
  // end of building-block
}
```

Make sure you understand how this program makes the pattern.

Note – in a Kojo program, anything after a // on a line is called a comment, and is ignored by Kojo. Comments are for human consumption.

Q2a. Identify the start of the building-block in the code above.

Q2b. Identify the end of the building-block in the code above.

Q2c. What pieces of information does the building-block contain?

Q2d. What lines in the code above specify the building-block shape?

Q2e. What lines in the code above specify the building-block position?

Q2f. What lines in the code above specify the building-block direction?

Q2g. What is a comment? How many comments do you see in the code above?

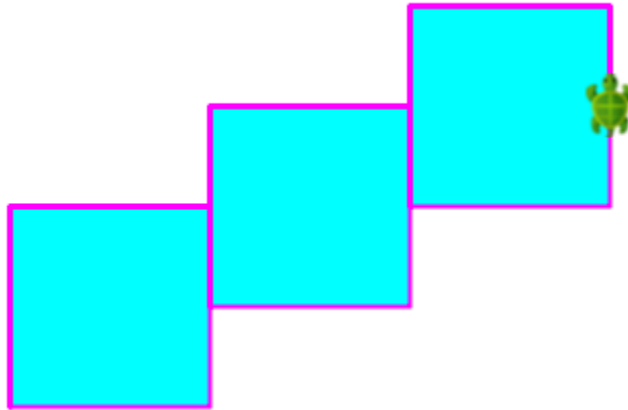
Self Exploration

Play with the code above to try out a few different buildings blocks. For this you need to modify the code between the '// start of building-block' and '// end of building-block' comments.

Exercise

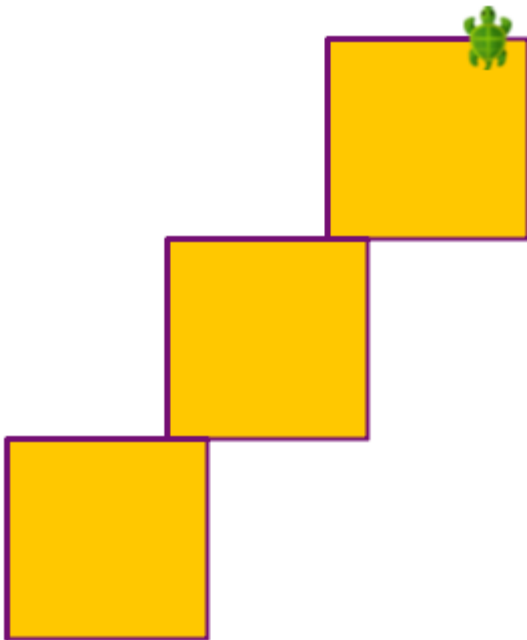
Write a program to make the following pattern. The length of the square is 100 units. Each subsequent square in the pattern starts half-way along the height of the previous square. The pen color is magenta. The fill color is cyan.

Use the three-step pattern-making procedure to make this pattern.



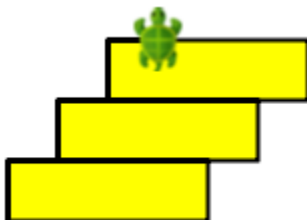
Activity 9a – Pattern Drawing Practice

Write programs to make the following patterns. Use the three-step pattern-making procedure described in Activity 7 to make the patterns. Each program should be no more than 14 lines of code.



The length of the square is 100 units. Each subsequent square in the pattern starts 80% along the width of the previous square.

The pen color is purple. The fill color is orange.



The width of the rectangle is 100 units

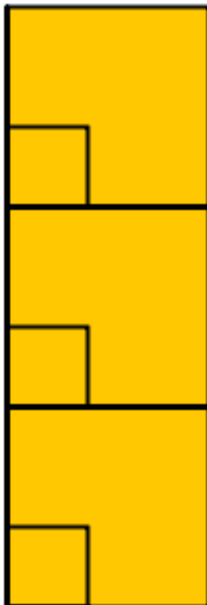
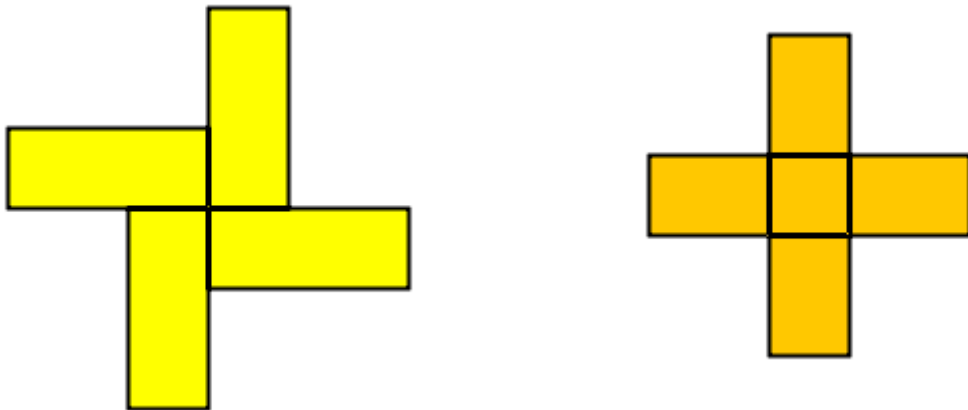
The height of the rectangle is $\frac{3}{10}$ of its width

Each subsequent rectangle in the pattern starts 25% along the width of the previous rectangle.

The pen color is black. The fill color is yellow.

Activity 9b – Extra Practice with Patterns

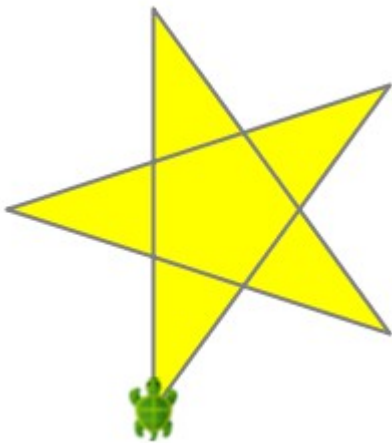
Write programs to make the following patterns. Use the three-step pattern-making procedure described in Activity 7 to make the patterns. Each program should be no more than 14 lines of code, unless specified otherwise. Use dimensions of your own choice such that your patterns look like the ones below.



The program to make this figure can be longer than 14 lines

Activity 9c – Extra Practice with Angle based Patterns

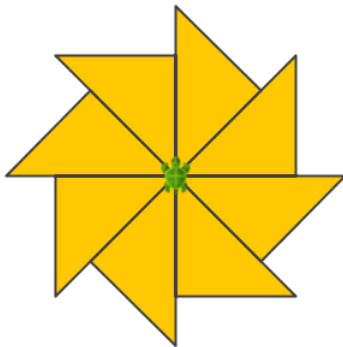
Write programs to make the following patterns. Use the three-step pattern-making procedure described in Activity 7 to make the patterns.



The length of the sides of the star is 100 units. The pen color is grey, and the fill color is yellow.

Make use of the following facts to determine the turning angle for the turtle required to make this figure:

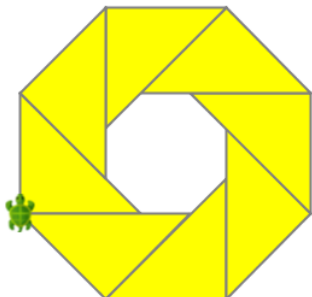
- To turn all the way around in a full circle, you need to turn through 360 degrees.
- The turtle makes a certain number of full circle turns (cumulatively) while making this figure.
- The turtle makes a certain number of actual turns to complete these full circle turns



One of the shorter sides of each triangle is 100 units in length. The pen color is darkGray and the fill color is orange.

Make use of the following facts to determine the turning angles for the turtle and the unknown lengths in the figure:

- Properties of Isosceles triangles.
- Pythagoras theorem.



The pen color is grey, and the fill color is yellow. The triangles have the same dimension as in the previous pattern.

Activity 10 – Calculations

This activity involves the following:

- Learning to do calculations within Kojo
- Learning about expressions.

Step 1. Type in the following code and run it (by using the *Run as Worksheet* button)

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

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 (by 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?

Self Exploration

Play with doing different kinds of calculations.

Theory

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 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 saw was a function (`Color(red, green, blue, alpha)` – for color mixing). A function takes some values as inputs and computes and returns an output value based on the inputs.

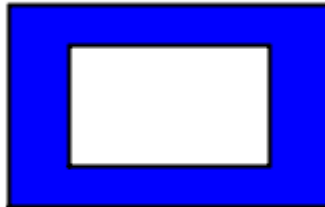
In this activity, you have worked with arithmetic operators like $+$, $-$, $*$, and $/$. These are also functions, but they are called in infix notation (e.g. $1 + 2$) as opposed to prefix notation (e.g. $+(1, 2)$). Functions belong to a category of instructions called expressions. Most expressions are 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.

Exercise

Calculate the area of the given figure with the help of the calculation capability in Kojo. The dimensions of the two rectangles in the figure are:

Outer Rectangle: length = 160, breadth = 100

Inner Rectangle: length = 100, breadth = 60



Activity 11 – Your own commands

This activity involves the following:

- Learning to create new commands within Kojo using the `def` instruction
- Learning about keyword instructions.
- Becoming familiar with the very important ideas of primitives, composition, and abstraction.

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 this 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

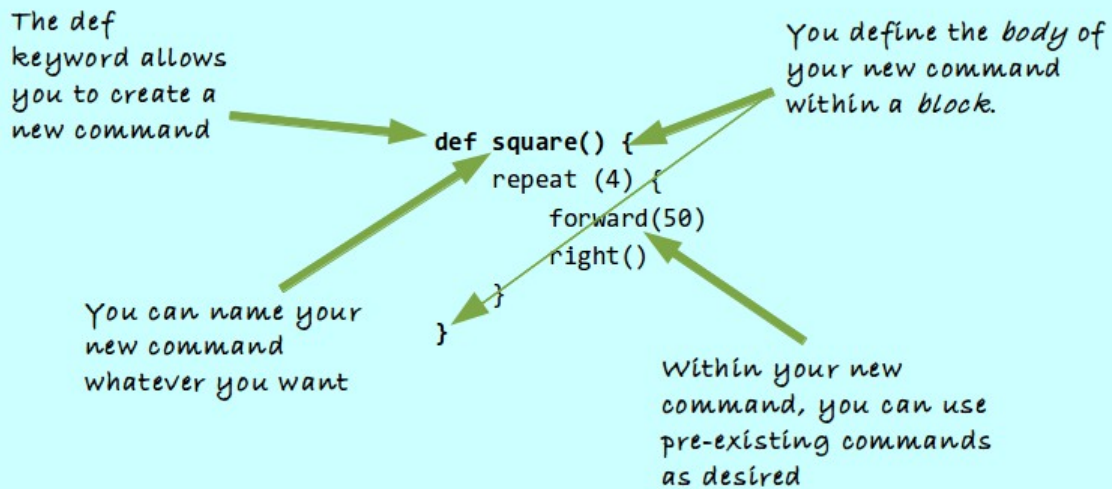
As mentioned earlier, programs are made out of a series of instructions for the computer. You saw two kinds of instructions in the previous activity:

1. Commands, which let you take actions or affect future actions (like moving the turtle forward or setting the pen color).
2. Expressions, which let you do calculations (e.g. 5+9).

You saw a new kind of instruction in this activity – a keyword instruction (**def**). 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 **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, and then reuse them. This reduces code duplication, and makes your programs easier to understand.

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

Computer programming is about three basic things:

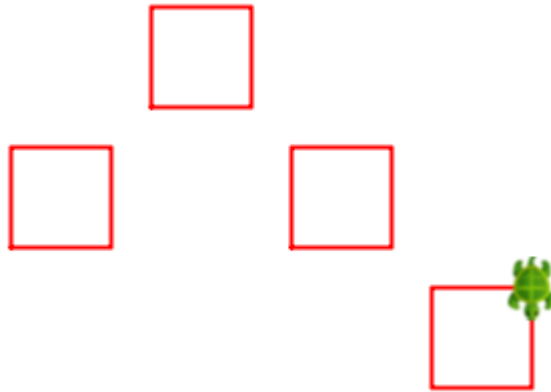
- primitives – these are the instructions already available in our programming environment.
- composition – this is how you combine primitives to do what is required.
- abstraction – this is how you give our compositions a name, so that they can be used as higher level primitives within your programs. 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. pre-existing commands.

Exercise

Write a program to make the given figure. The square size is 50, and the vertical and horizontal distance between the squares is 40% of the square size.



Activity 12 – Named Values

This activity involves the following:

- Learning to assign names to values using the `val` keyword instruction.

Step 1. Type in the following code and run it:

```
clear()
repeat(4) {
    forward(100)
    right()
}
repeat(4) {
    forward(50)
    right()
}
repeat(4) {
    forward(25)
    right()
}
```

Step 2. Now make the figure that you made in the previous step twice as large (how can you do this?).

Step 3. Type in the following code and run it:

```
val size = 100
// You can also use a calculation to determine size
// val size = canvasBounds.height * 0.75
clear()
repeat(4) {
    forward(size)
    right()
}
repeat(4) {
    forward(size/2)
    right()
}
repeat(4) {
    forward(size/4)
```

```

    right()
}

```

Step 4. Now make the figure that you made in the previous step twice as large (how can you do this?).

Q4a. Was it easier to increase the size of the previous figure in Step 2 or was it easier in Step 4?

Q4b. What do you think the val instruction does?

Self Exploration

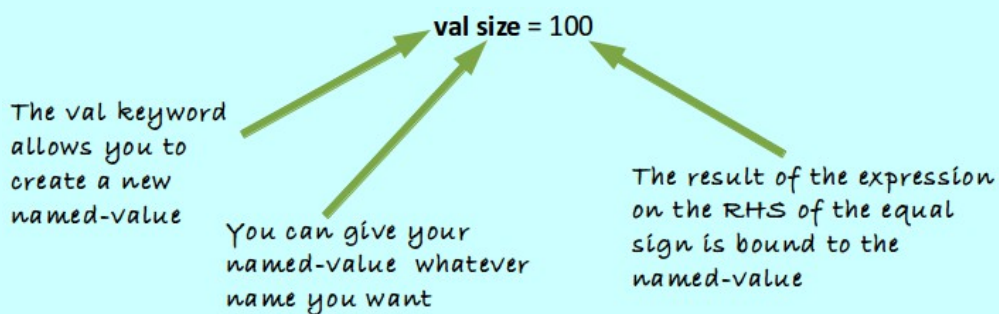
Play with:

- Resizing the figure by assigning different number to the size named-value.
- Making new squares using the size named-value.

Theory

The **val** keyword instruction allows you to create named values of your own. You can then use these named values in different locations in your program.

The first line in the code in Step 3 uses the val keyword instruction. Here's a closer look at that fragment of code:



After this line of code in the program, the name size is available within the program with a value of 100.

Named values give you the ability to refer to numbers by name, as opposed to their values.

Why would you want to do that (i.e. refer to numbers by name)?

For the following reasons:

- To make it easier to make changes in your program. You saw this when you needed

to make a smaller or larger version of the figure above. You just had to change one line at the beginning of your program to change the size of the figure, instead of having to scan through your whole program and making multiple changes.

- To make your program more understandable. Looking at the program in Step 3, it becomes clear right away that the size of the squares made by the program can be controlled.
- To avoid redoing calculations for values that will be used multiple times in your program.

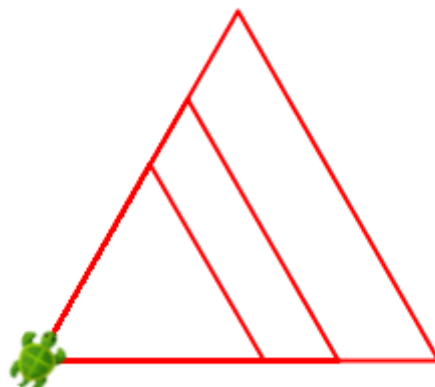
Think about how all of this relates to the idea of primitives, composition, and abstraction.

Exercise

Write a program to make the figure shown below. The figure has three triangles. As we move from the biggest to the smallest triangle, each triangle is 75% of the size of the previous triangle.



Now, make a change in just one line of your program to make the bigger figure shown below:



Activity 13 – Your own dynamic commands

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 this 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)
}
```

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 a detailed picture showing you how to create a new command that takes an input:

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 is provided 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 also have **types** associated with them. The type of an input tells Kojo:

- the permissible values of the input.
- the operators and commands that it can work with.

Telling Kojo the type of the input to your user defined command has a couple of

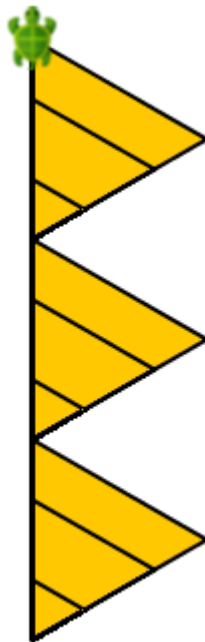
advantages:

- 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 idea of primitives, composition, and abstraction.

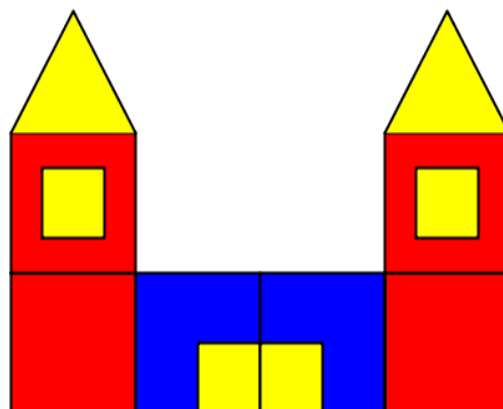
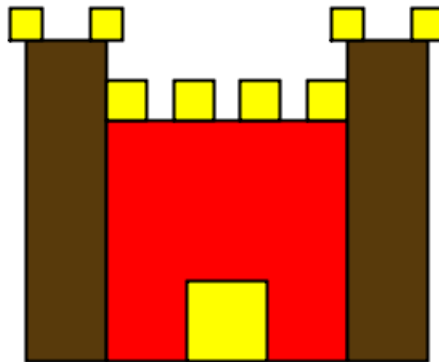
Exercise

Write a program to make the given 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.



Activity 14 – Mini Project

Write programs to make the following figures. Use things like the repeat command and user defined commands to help you in making the figures.



Activity 15 – Break Free

Use everything that you have learnt till now to make your own creation. First, sketch your idea out on paper. Then start making it within Kojo. As you go along, keep refining your idea till you are satisfied with what you have.

Feel free to share your creation with us on the Kojo [Code Exchange](#).



The [Code Exchange](#) is a website to which you can post your Kojo sketches and code - with one click of a button within Kojo.

This *Code Exchange* is a good place for Kojo users to showcase their work, look at and rate the work of others, provide comments and exchange ideas, and learn in a collaborative fashion.

Activity 16 – Strings and I/O

Step 1. Type in the following code and run it:

```
val name = readLn("What's your name?")
val age = readLn("What's your age?")
clear()
write(s"Hello $name, your age is $age")
```

Q1a. What do you think the readLn instruction does?

Q1b. What do you think the readInt instruction does?

Q1c. What do you think is the data between the double quotes ("")

Q1d. Why do you think the input to the write command has an 's' at the beginning – s"Hello \$name, your age is \$age"?

Q1e. Is the program above taking any input? Is it providing any output?

Self Exploration

Play with the code above as you see fit.

Theory

Strings:

- Are used to represent text.
- Are a type within Kojo.
- Are particularly useful for providing Input to a program and generating output from a program.

Exercise

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

Activity 17 – Conditionals

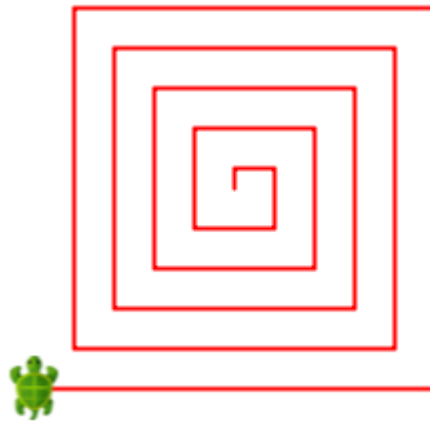
Step 1. Type in the following code and run it:

Theory

Introduce the idea of selection as a fundamental element of programming in addition to primitives, composition and abstraction.

Activity 18 – Repeat with a counter

Till now, you have used the repeat command to do exactly the *same* thing multiple times. But what if you want to do a *similar* (but slightly different) thing multiple times? Look at the following figure:



You can see that, starting from the center, the turtle goes forward and right to make the figure. That's also what the turtle does to make a square. What's different here is that each time the turtle moves forward, it moves forward by a slightly greater amount.

Do you see that?

Step 1. Type in the following code and run it:

```
clear()
setAnimationDelay(100)
repeati(20) { i =>
  forward(10 * i)
  right()
}
```

Q1a. What do you think the repeati command does?

Q1b. How is the repeati command different from the repeat command?

Step 2. Type in the following code and trace it (via the *Trace Script* toolbar button):

```
clear()
setAnimationDelay(100)
```



```
def repCode(i: Int) {
  forward(10 * i)
  right()
}
repeati(20)(repCode)
```

Look at the program trace and answer the following questions:

- Q2a.** How many inputs does the repeati command take?
Q2b. What is the second input to the repeati command?
Q2c. What does the repeati command do with its second input?

Self Exploration

Play with the inputs to the repeati() and forward() commands in the code above (make the input to the forward command depend on 'i'). See how changing these inputs modifies the figure.

Theory

Commands can take multiple inputs in a couple of different ways. Imagine that there's a command to make rectangles on the screen. This command will need two inputs – a length and a breadth. You can define this command in two different ways:

```
def rectangle1(l: Int, b: Int) {
  // body not shown
}
def rectangle2(l: Int)(b: Int) {
  // body not shown
}
```

You can now call these different versions of the rectangle command in the following ways:

The one input list version:

```
rectangle1(100, 50)
```

The multiple input lists version:

```
rectangle2(100)(50)
```

Why do we have two different ways to define commands? The answer to that is beyond the scope of this book, but in case you're curious and want to research the idea further on the

web, this relates to currying and partial application of commands and functions.

The repeat command works (like the second case above) with multiple input lists. So you could call it like this:

```
repeat(3)( forward(100) )
```

Here, the first input tells the repeat command how many times to do something, and the second input tells it what to do. The repeat command calls the supplied code the specified number of times.

This works fine if you want to repeat just one command (like forward(100) above). But what if you want to repeat multiple commands? That's where blocks (the things within curly brackets) come in. A block allows you to combine multiple commands into a group, and pass it as one input into the repeat command:

```
repeat (3) ({
  forward(100)
  right(45)
})
```

But now that you are using curly brackets, the round-brackets are redundant, and you can just write:

```
repeat (3) {
  forward(100)
  right(45)
}
```

The repeat command works in a similar way, except that it gives the specified code some information to allow it to do *similar but slightly different* things. This information is in the form of a repeat counter:

```
repeati(20) { i => // i is the repeat counter
  forward(10 * i)
  right()
}
```

Your job is to map (or convert) the repeat counter into a required value based on the task that you are trying to accomplish.

Exercise

A 6-line figure where the length of each line is 10 more than the length of the previous line and the total length is 270.

A 6-line figure where the length of each line is twice the length of the previous line and the total length is 252.

Activity 19 – Your own functions

Step 1. Type in the following code and run it:

```
clearOutput()
def reqVal(i: Int) = {
  5 + 2 * (i - 1)
}

repeati(5) { i =>
  println(reqVal(i))
}
```

Q1a. Which line in the above code maps (or converts) the repeat counter to the required value (where the required value is the number that we are trying to print in the Output pane)?

Q1b. What do you think the def instruction is doing in the above code?

Q1c. What are the numbers printed in the Output Pane by the above code?

Q1d. How is an arithmetic sequence being used in the above code? What are the a and d parameters of this arithmetic sequence?

Step 2. Type in the following code and run it:

```
clearOutput()
def reqVal(i: Int) = -4 + 3 * (i - 1)
repeati(5) { i =>
  println(reqVal(i))
}
```

Q2a. What are the numbers printed in the Output Pane by the above code.

Q2b. How is the def instruction in Step 2 different from the def instruction in Step 1

Q2c. How is an arithmetic sequence being used in the above code? What are the a and d parameters of this arithmetic sequence?

Self Exploration

Play with how the required value is calculated in Step 2. See how changing the formula in the reqVal function changes the numbers printed in the Output Pane.

Theory

We 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 $+$, $-$, $*$, $/$, `math.max`, `math.sqrt` etc.

In this activity you have seen how you can define your own functions:

```
def reqVal(i: Int) = {  
    5 + 2 * (i - 1)  
}
```

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 reqVal(i: Int) = {
```

Command definitions do not require this `=` sign.

This is meant to signify that functions are *equivalent* to the value that they calculate - and return to the caller - based on the inputs provided to them. To understand this idea better, let's look at the following function call:

```
reqVal(1)
```

Here, `reqVal(1)` is equivalent to the value that it calculates (5) – because it takes an input: 1, and returns this value.

User defined commands, on the other hand, are not equivalent to anything – because they don't calculate and return any values; instead, they just carry out some actions. Hence, we don't put the `=` sign on the first line of their definition.

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

Exercise

Write a command called `numberLine` which takes the following inputs:

`start`: Int – the number where the number line starts.

`end`: Int – the number where the number line ends.

`divisions` – the number of divisions on the number line between the start and end numbers.

The `numberLine` command then draws the specified portion of the number line on the drawing canvas. For example, the following call to `numberLine`:

```
numberLine(2, 10, 12)
```

should draw this:



Note – you can use the `setPenFontSize(n)` command to decrease the size of the font used to write text. The above figure uses a font size of 12.

Activity 20 – Classes

Step 1. Type in the following code and run it in worksheet mode:

```
case class Student(firstName: String, lastName: String, age: Int) {  
  def teenager = age > 12 && age < 20  
  def fullName = firstName + lastName  
}  
  
val student = Student("Anusha", "Pant", 12)  
student.teenager
```

Q1a. What do you think the **case class** keyword does?

Q1b. What is Student? A new value or a new type?

Self Exploration

Play with the code above as you see fit.

Theory

A case class definition allows you to create a new abstraction via the composition of multiple predefined data items and functions/commands that act on this combination of data items. The functions/commands inside a class are called methods.

In concrete terms, a case class defines:

- a new type
- a constructor function for values of that type

A class is just a blueprint/template for values of the type that it defines. These values (called objects or instances of the class) are created via the constructor function.

Exercise

(1) Write a class to describe Rational numbers.

The class should take two inputs: numerator and denominator

It should provide the following methods: add, subtract, multiply, divide

Using this class, you should be able to add two rational numbers like this:

```
val r1 = new Rational(2, 3)
val r2 = new Rational(3, 4)
r1.add(r2)
```

Now comes an interesting idea:

`r1.add(r2)` can be written as `r1 add r2`. If you rename `add` to `+`, that becomes `r1 + r2`

(1b) Rename `add` to `+`, `subtract` to `-`, `multiply` to `*`, and `divide` to `/`. You should now be able to use the usual arithmetic operators with rational numbers just like you use them with the pre-existing integers.

(2) Refine the `numberLine` command that you wrote in the previous Activity. This time, instead of writing decimal fractions next to the tick-marks on the line, write rational numbers (which represent fractions with the form $\frac{m}{n}$). For example, the following call to `numberLine`:

```
numberLine(2, 10, 8)
```

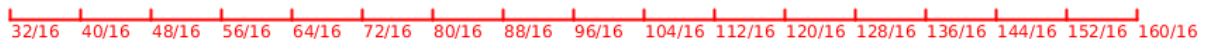
should draw this:



And the following call to `numberLine`:

```
numberLine(2, 10, 16)
```

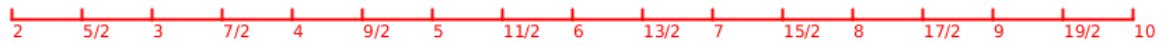
Should draw this:



(3) Refine your `numberLine` command so that the following call to `numberLine`:

```
numberLine(2, 10, 16)
```

draws this:



Note – you can use the following function to help you:

```
def gcd(n1: Int, n2: Int): Int = {  
    if (n2 == 0) n1 else gcd(n2, n1 % n2)  
}
```

Activity 21 – Sequences

Step 1. Type in the following code and run it:

```
clear()
val colors = Seq(blue, green, yellow)

colors.foreach { c =>
  setFillColor(c)
  repeat(4) {
    forward(100)
    right()
  }
  right()
  hop(150)
  left()
}
```

Q1a. What do you think the Seq function does?

Q1b. What do you think the foreach command does?

Self Exploration

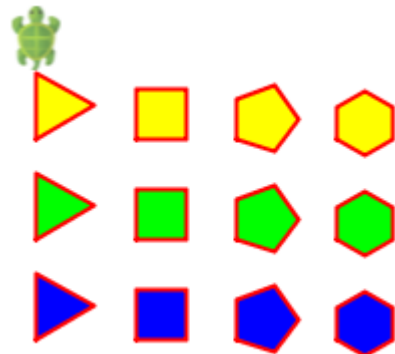
Play with the code above as you see fit.

Exercise

Write a program to make the given figure.

Hints:

- define a command to make regular polygons. The command should take one input – the number of sides of the polygon.
- use the `savePosHe()` and `restorePosHe()` commands to help you with this exercise.



Activity 22 – Maps

Step 1. Type in the following code and run it:

```
clear()
val distances = Map(
  "Small" -> 20,
  "Medium" -> 80,
  "Large" -> 160
)
val d = readln("How much distance should the turtle move (Small, Medium,
Large)?")
forward(distances(d))
```

Q1a. What do you think the Map function does?

Q1b. What's a constructor function?

Q1c. What does distances(d) do in the above code?

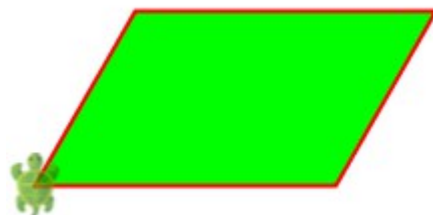
Self Exploration

Play with the code above as you see fit.

Exercise

Write a program that does the following:

- Reads in a color (one of green, blue, or yellow).
- Make a parallelogram (like the one on the right) filled with the given color.



Activity 23 – Variables

Step 1. Type in the following code and run it:

```
val numbers = Seq(1, 2, 3, 4, 5)
var sum = 0
numbers.foreach { n =>
    sum = sum + n
}
clearOutput()
println(s"The sum of the given numbers is: $sum")
```

Q1a. What do you think the var keyword instruction does?

Q1b. How do you think the var instruction is different from the val instruction?

Step 2. Type in the following code and run it:

```
var numbers = Seq.empty[Int]
val msg = "Enter a number, or nothing to finish"
var num = readln(msg)
repeatWhile(num != "") {
    numbers = numbers :+ num.toInt
    num = readln(msg)
}

var sum = 0
numbers.foreach { n =>
    sum = sum + n
}
clearOutput()
println(s"The sum of the given numbers is: $sum")
```

Q2a. What are the different ways in which variables are used (via the var instruction) in this program?

Self Exploration

Play with the code above as you see fit.

Exercise

(1) Write a program that does the following:

- Asks the user a series of questions.
- Makes a bar graph based on the answers.

(2) Write a program that does the following:

- Asks the user a series of questions.
- Makes a histogram based on the answers.

Solutions

Activity 1 – Commands and Programs

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



Activity 2 – Using the Kojo Environment Effectively

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

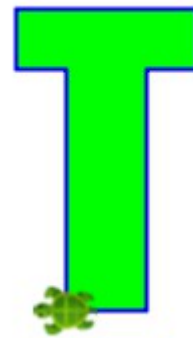


```
right()
```

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

Activity 3 – Pen and Fill Color

```
clear()
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)
```

**Activity 4 – Mixing Colors with a function**

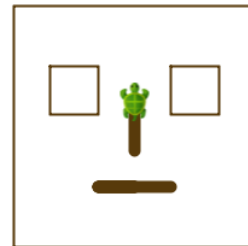
```
clear()
setFillColor(Color(50, 100, 150, 200))
left()
forward(50)
right()
forward(30)
right()
forward(50)
left()
forward(50)
right()
forward(30)
right()
forward(50)
left()
forward(50)
right()
```



```
forward(30)
right()
forward(50)
left()
forward(50)
right()
forward(30)
right()
forward(50)
```

Activity 5 – Hopping with Speed

```
clear()
setAnimationDelay(10)
setPenColor(brown)
forward(200)
right()
forward(200)
right()
forward(200)
right()
forward(200)
right()
hop(150)
right()
hop(30)
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()
right()
hop(100)
right()
setPenThickness(10)
forward(60)
back(30)
right()
hop(30)
forward(40)

```

Activity 6 – Angles

Figure 1

```

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

```

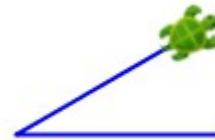


Figure 2

```

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

```

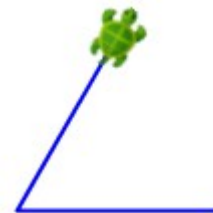
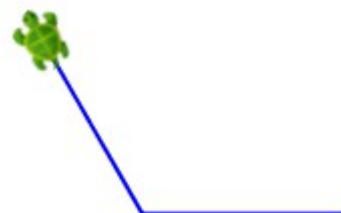


Figure 4

```

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

```



Activity 6a – Practice

Figure 1

```

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)

```

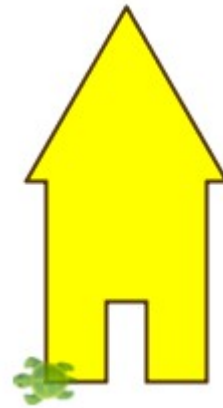


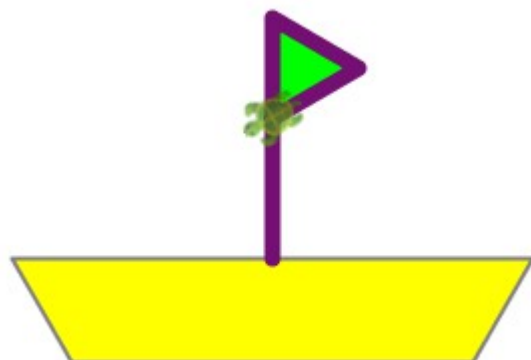
Figure 2

```

clear()
setAnimationDelay(100)
setPenColor(gray)
setFillColor(yellow)
left()
forward(200)
right(60)
forward(60)
right(90 + 30)
forward(260)

```

66



```

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)

```

Activity 7 – Repeating commands

```

clear()
setPenColor(gray)
setFillColor(orange)
repeat (4) {
    forward(100)
    right()
}

```

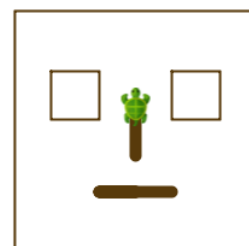


Activity 8 – Analyzing the Repeat Command

```

clear()
setAnimationDelay(10)
setPenColor(brown)
repeat(4) {
    forward(200)
    right()
}
hop(150)
right()
hop(30)
repeat(4) {
    forward(40)
    right()
}

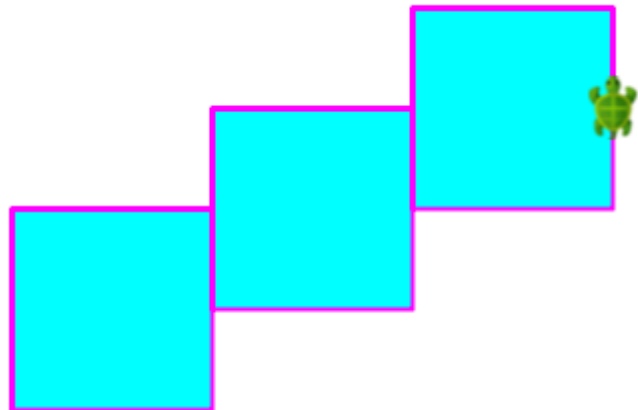
```



```
}  
hop(100)  
repeat(4) {  
    forward(40)  
    right()  
}  
right()  
hop(100)  
right()  
setPenThickness(10)  
forward(60)  
back(30)  
right()  
hop(30)  
forward(40)
```

Activity 9 – Repeating to make a pattern

```
clear()  
setAnimationDelay(100)  
setPenColor(magenta)  
setFillColor(cyan)  
repeat(3) {  
    repeat(6) {  
        forward(100)  
        right()  
    }  
    forward(50)  
    right(180)  
}
```

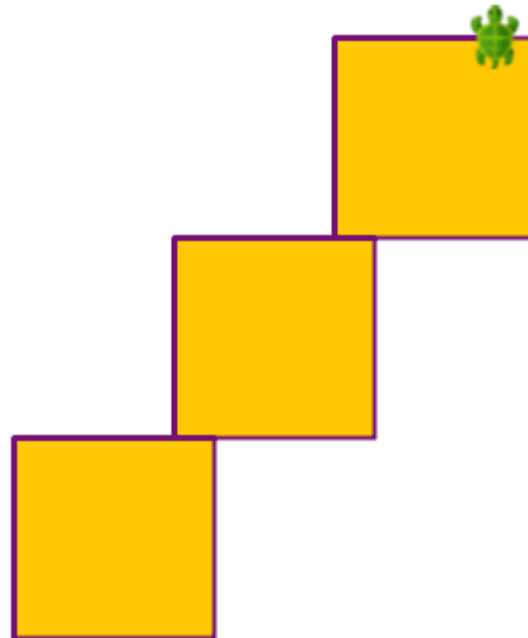


Activity 9a – Pattern Drawing Practice

```

clear()
setAnimationDelay(100)
setPenColor(purple)
setFillColor(orange)
repeat(3) {
  repeat(4) {
    forward(100)
    right()
  }
  forward(100)
  right()
  forward(80)
  left()
}

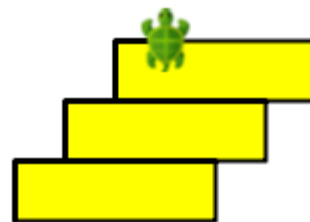
```



```

clear()
setAnimationDelay(100)
setPenColor(black)
setFillColor(yellow)
repeat(3) {
  repeat(2) {
    forward(30)
    right()
  }
  forward(100)
  right()
  forward(30)
  right()
  forward(40)
  left()
}

```

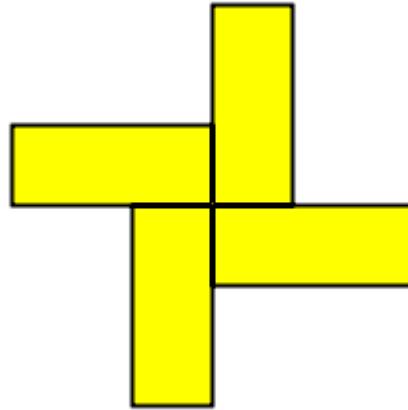


Activity 9b – Extra Practice with Patterns

```

clear()
setAnimationDelay(100)
setPenColor(black)
setFillColor(yellow)
repeat (4) {
  repeat (2) {
    forward(100)
    right()
    forward(40)
    right()
  }
  right()
}
invisible()

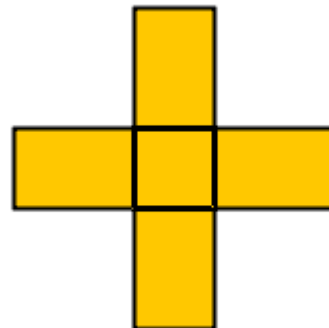
```



```

clear()
setAnimationDelay(100)
setPenColor(black)
setFillColor(orange)
repeat (4) {
  repeat (2) {
    forward(100)
    right()
    forward(40)
    right()
  }
  forward(40)
  right()
}
invisible()

```



```

clear()
setAnimationDelay(100)
setPenColor(black)
setFillColor(yellow)
repeat (4) {
  forward(100)
  right()
}

```

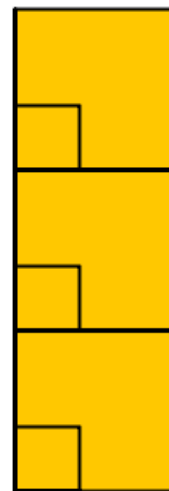


```

        forward(100)
        right()
        forward(100)
        left()
        forward(100)
        left()
    }
invisible()

clear()
setAnimationDelay(100)
setPenColor(black)
setFillColor(orange)
repeat(3) {
    repeat(4) {
        forward(100)
        right()
    }
    repeat(4) {
        forward(40)
        right()
    }
    forward(100)
}
invisible()

```

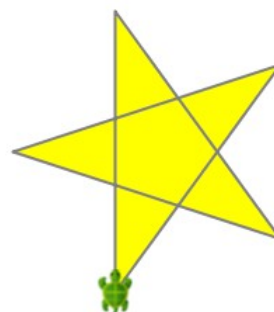


Activity 9c – Extra Practice with Angle based Patterns

```

clear()
setAnimationDelay(100)
setPenColor(gray)
setFillColor(yellow)
repeat (5) {
    forward(200)
    right(144)
}

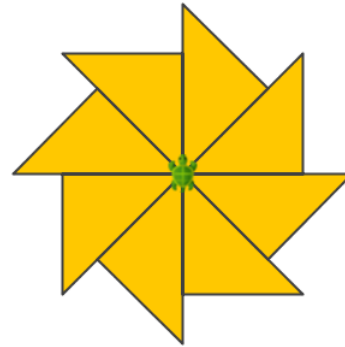
```



```

clear()
setAnimationDelay(100)
setPenColor(darkGray)
setFillColor(orange)
repeat(8) {
  forward(math.sqrt(2 * 100 * 100))
  right(135)
  forward(100)
  right(90)
  forward(100)
  right(180)
}

```

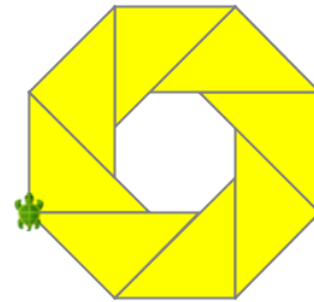


```

clear()
setAnimationDelay(100)
setPenColor(gray)
setFillColor(yellow)

repeat(8) {
  forward(100)
  right(135)
  forward(math.sqrt(2 * 100 * 100))
  right(135)
  forward(100)
  right()
  hop(100)
  right(45)
}

```



Activity x – Repeating to make a pattern

```

clear()
setAnimationDelay(100)
setPenColor(magenta)
setFillColor(cyan)
repeat(3) {
  repeat(6) {
    forward(100)
    right()
  }
}

```



```

    forward(50)
    right(180)
}

```

Activity 21 – Sequences

```

val colors = Seq(blue, green, yellow)

```

```

def rpoly(n: Int) {
  repeat(n) {
    forward(100 / n)
    right(360 / n)
  }
}

```

```

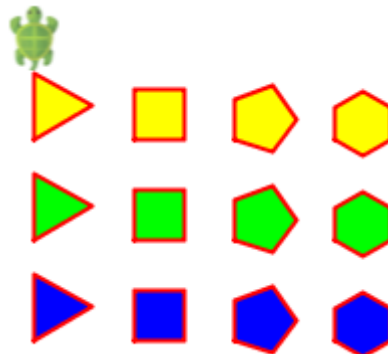
clear()
setAnimationDelay(10)

```

```

colors.foreach { c =>
  setFillColor(c)
  savePosHe()
  repeati(4) { i =>
    savePosHe()
    rpoly(i + 2)
    restorePosHe()
    right()
    hop(50)
    left()
  }
  restorePosHe()
  hop(50)
}

```

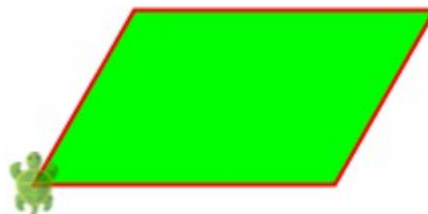


Activity 22 – Maps

```

clear()
val colors = Map(
  "green" -> green,
  "blue" -> blue,
  "yellow" -> yellow
)

```



```
)  
  
val c = readln("What colored parellelogram do you want?")  
  
setFillColor(colors(c))  
  
repeat(2) {  
    right(30)  
    forward(100)  
    right(60)  
    forward(150)  
    right(90)  
}
```