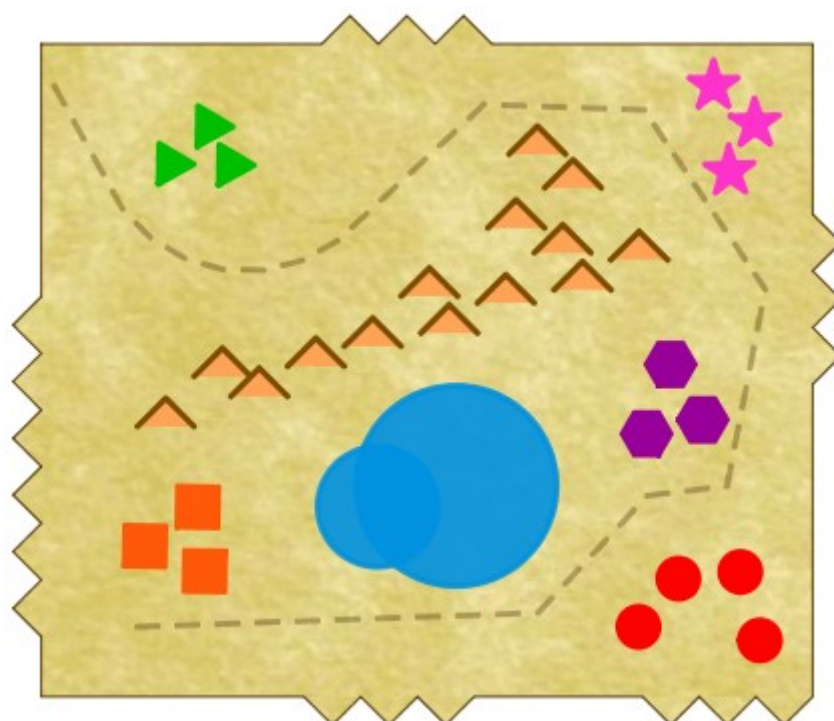


Explorations with Kojo

Level 1



By Lalit Pant

About the author



Lalit is a codecrafter, software architect, and teacher based out of Dehradun, India. He spends his time enjoying teaching computer programming and math to school-children, writing ebooks, and developing educational software. Lalit has worked as a professional programmer for many many years – in roles all the way from junior programmer to CTO – designing and developing software for companies in the United States, Europe, and India.

He has also written articles for popular (in the late 1990s) programming magazines like Dr. Dobbs Journal and Java Report. *Fun fact – the April 1999 issue of Dr. Dobb’s Journal, which features an article by Lalit, also has articles by some very famous computer scientists and authors – Brian Kernighan (co-creator of Unix), Rob Pike (co-creator of Unix and the Go programming language), Willam Stallings (networking, security), Jon Bentley (programming, algorithms), and Al Stevens (programming).*

Lalit holds multiple patents, but is not too proud of that fact given his [open source](#) leanings. His [open source projects](#) include [Kojo](#) (a programming based learning environment), [Tefla](#) (a deep learning AI toolkit), and [Jiva](#) (a genetic algorithms library).

Lalit is a graduate of IIT Kanpur. He has a postgraduate degree from IIT Delhi. He did his schooling at La-Martiniere, Lucknow.

Explorations

with Kojo

Level 1

Version: June 14, 2020



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

© 2010–2020 Lalit Pant (lalit@kogics.net)

<http://www.kogics.net>

Contents

1	A Note for Facilitators and Teachers	6
2	Introduction	9
3	Commands and Programs	12
4	Using Kojo Effectively	17
5	Drawing with Colors	22
6	Hopping with Speed	27
7	More colors with the ColorMaker	30
8	Digging Deeper with Tracing	34
9	Angles	37
10	Practice	40
11	Repeating Commands	41
12	Practice with repeat	43
13	Exporting your artwork	44
14	Art Project 1	46
15	Functions	47
16	Calculations	51
17	Interlude - Computation	56
18	Repeat inside repeat	58
19	Turning with a radius	61
20	More Fun with Repeat	66

21 Repeat with a sequence	70
22 Art Breakout	74
23 Random Numbers and Named Values	75
24 Your own Commands	81
25 Your own Commands, with Inputs	86
26 Polygon Art	91
27 Your own Functions	96
28 Mini Project	101
29 Art Project 2	102
30 Strings and I/O	103
31 Artistic Text in the Canvas	107
32 Conditionals	110
33 Pattern Drawing Practice	115
34 Patterns	116
35 More Practice with Patterns	119
36 Designing and making art	121
37 Final Project	122
38 Programming Quickref	123
39 Turtle Commands Quickref	124

1 A Note for Facilitators and Teachers

This book is meant to help children explore various topics in computer programming, math, art, and science using the Kojo Learning Environment (www.kojo.in).

At a very high level, the goal of this book is to help children gain essential 21st century skills *through sustained practice and constant feedback*. The way this works in Kojo is that children *play, create, and learn*. They *play* with computer programs, they *create* drawings, animations, games, music, and circuits, and they *learn* 21st century skills and more. The 21st century skills that we are talking about here are – computation, critical thinking (defined here as careful goal-directed thinking), creative thinking, collaboration, and communication.

An even deeper goal of this book is to take children on a journey of *learning with understanding* – in the domain of programming, math, art, and science – to begin with, but with ideas that apply across subjects.

The structure of this book is inspired by ideas from the [Understanding by Design](#)¹ pedagogical framework. The basic idea of this framework is to *teach for understanding* by:

1. Identifying desired results or goals for attaining understanding, in terms of **transfer** (the ability to apply what is learnt in a new context), **meaning-making** (the ability to explain in ones own words), and **aquisition** (the learning of core facts).
2. Coming up with a description of assessment evidence – to determine how well children are doing in getting to the desired goals.
3. Planning lessons – to support the above.

In keeping with these ideas, the following are the desired goals and assesment evidence for children using this book:

1. Learn to DESIGN and CREATE using computer programs.
 - Evidence: 3 or 4 self designed artifacts (of printable art) during the year.
Subsequent books will enable children to make fun games, compose interesting music, and create useful electronic circuits. In this book we will focus only on art – which is a rich and rewarding area in itself.
2. Learn to analyse a problem before coming up with a solution for it.
 - Evidence: The ability to look at a complex visual pattern and identify its building blocks before making it (using a computer program).

¹https://www.ascd.org/ASCD/pdf/siteASCD/publications/UbD_WhitePaper0312.pdf

3. Understand the meaning of computation.
 - Evidence: The ability to identify examples of computation in the surrounding environment, and to imagine new possibilities for computation.
4. Learn simple computer programming using turtle graphics.
 - Evidence: The ability to write programs to do given tasks; the ability to define computer programming, relate it to the idea of computation, and use it to create artifacts.
5. Get regular brain exercise along the dimensions of critical and creative thinking in an environment with immediate feedback.
 - Evidence: Creation of original, visually pleasing artifacts on a regular basis. *These are much smaller in scope than the 3 or 4 artifacts for the year mentioned in the first goal.*
6. Apply knowledge from subjects like math and art in creative works.
 - Evidence: The ability to talk, in one's own words, about the math and artistic ideas underlying the works of creation that are carried out.
7. Get practice with the PROCESS of analysis, design, and creation.
 - Evidence: The same as the evidence for items 1, 2, and 5 above.
8. Get practice with COMMUNICATION and COLLABORATION.
 - Evidence: Regular work as part of a team; class discussions; participation in at least one event every year where creations are showcased.
9. Build up an INVENTOR mindset.
 - Evidence: 3 or 4 self designed artifacts (of printable art) during the year. This is the same as the evidence for item 1.

In the context of this book, all of the the above will be accomplished via computer programming within Kojo. 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. More information about the strengths of Kojo as an environment for programming is available at – <https://docs.kogics.net/reference/kojo-env-strengths.html>.

Activity Structure

Most chapters in this book describe activities for children to carry out. Every activity chapter has the following structure:

1. A list of desired goals.

At the end of each goal in the list, there is one or more of three letters – **T**, **A**, and/or **C**. These are meant to identify the *type* of the goal, as per the Understanding by Design framework. Here's a quick recap of what these mean:

- a) Transfer (**T**) – the ability to apply what has been learnt – in a new context.
 - b) Meaning-making (**M**) – the ability to make sense of something so that, for example, it can be explained in one's own words.
 - c) Acquisition (**A**) – the learning of core facts about a subject, based on which meaning-making and transfer can take place.
2. A sequence of guided steps for children to follow.
 - Every step has step questions (which let you check for evidence of **M** and/or **A**).
 3. Time for self exploration

To give children a chance to explore what they have learnt to create meaning for themselves.
 4. A theory section

To help children get the core facts right.
 5. An exercise (which lets you check for evidence of **T**)
 6. A quiz (which lets you check for evidence of **M** and/or **A**)

2 Introduction

The book will help you to:

- **Create** beautiful computer generated art – which you can use as your desktop wallpaper, print on t-shirts, and do many more fun things with.
- **Apply** math to enrich your creations.
- **Learn** computer programming (a very important 21st century skill), and develop 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 the Kojo website – www.kojo.in.

When you start up Kojo, you see a workspace that looks similar to the screenshot below:

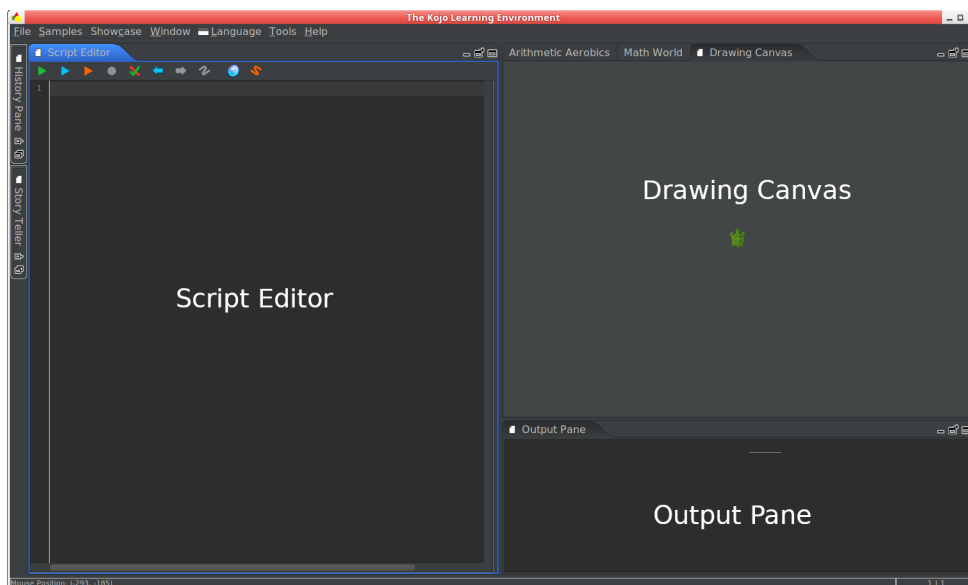


Figure 2.1: The Kojo Workspace

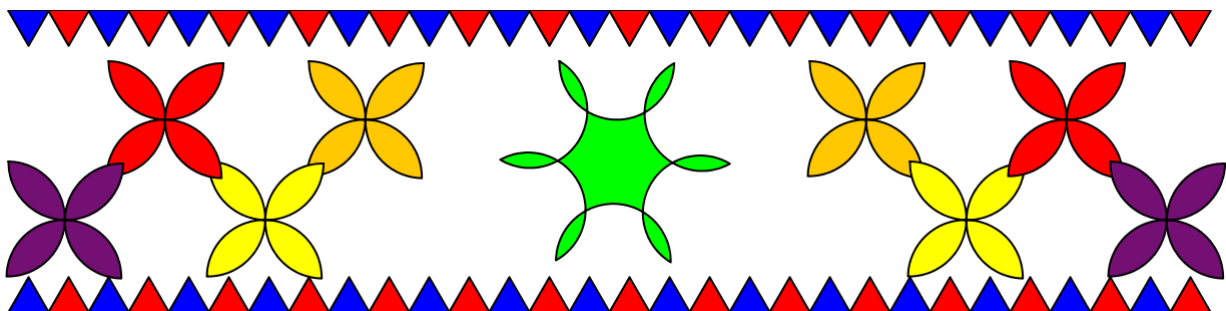
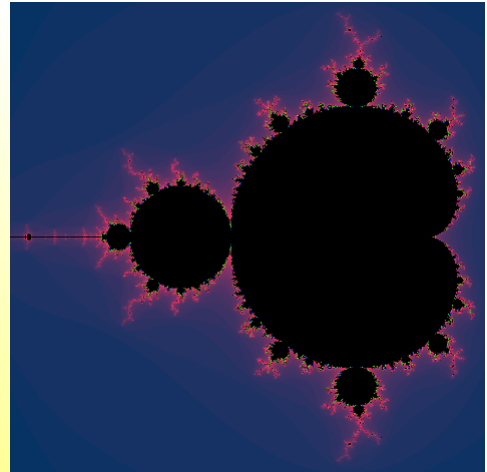
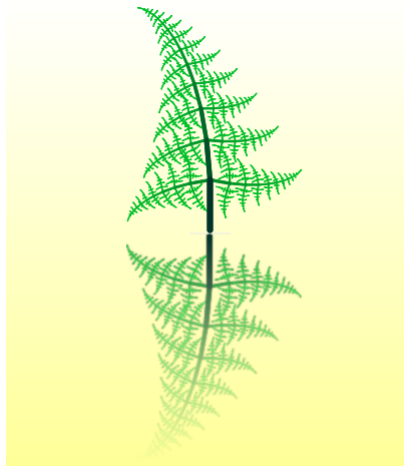
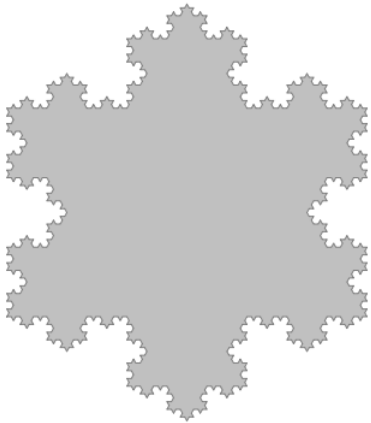
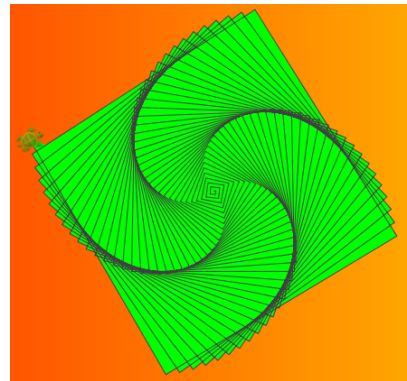
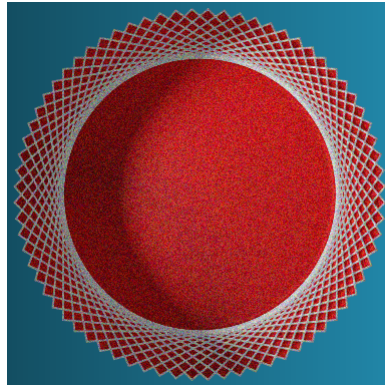
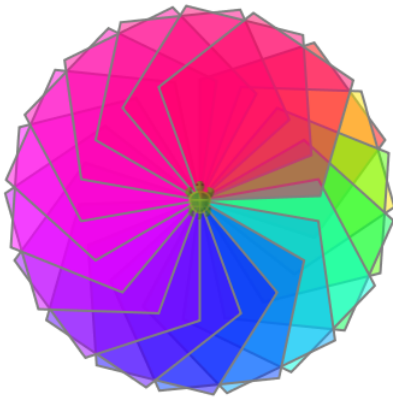
Note - The screenshot above shows Kojo running in *dark* mode. Kojo can switch between a *light* mode and a *dark* mode via the *File* -> *Settings* menu item.

At the left of the workspace is the **script editor**, where you write your computer programs (a program, also known as a script, is a series of instructions for the computer to carry out). The script editor has a tool-bar with buttons that quickly let you do most of the essential things that you need to do within Kojo – things like running a program, tracing a program, checking a programs for errors, etc.

At the top right of the workspace is a window called the **drawing canvas**. You see a turtle sitting at the center of this window. Your programs instruct this turtle to do things. You can ask the turtle 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 makes. All of this can be used to make very interesting drawings.

At 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 *Showcase* menus) of drawings that you can create within Kojo:



Quiz

Qz1 In the drawings shown above, which one do you like the best? Why? Explain to a friend (do this for all your quiz answers).

Qz2 What is Kojo?

Qz3 What is the script editor?

Qz4 What is the drawing canvas?

Qz5 How can you get the turtle to make a drawing?

3 Commands and Programs

This activity has the following desired goals:

- Learning about commands, actions, and programs (**M, A**).
- Learning to draw lines using the turtle (**A**).
- Learning the `clear`, `forward`, and `right` commands (**M, A**).
- Exploring the ideas of unit length, distances, and right angles, and using them to make a square geometrical figure (**M**).
- Using the Kojo error recovery feature (**M, A**).
- Making various geometric figures using turtle commands (**T**).



Step 1

Type in the following code within the Script Editor and run it – by clicking on the *Run* toolbar button  or pressing *Ctrl+Enter*:

```
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 clicking on *Clear* . Then delete the text in the script editor by clicking on 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. You can also find similar information about turtle commands at – <https://docs.kogics.net/reference/turtle.html>

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.

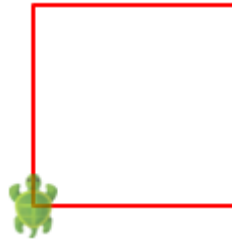
Brief command descriptions:

- `clear()` – clears the drawing canvas.
- `forward(n)`– moves the turtle forward `n` steps in the direction of its nose.
- `right()`– turns the nose of the turtle right by 90 degrees.

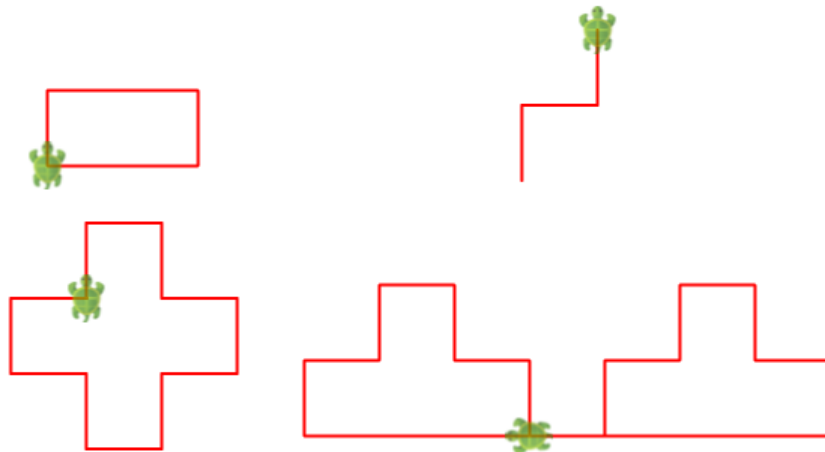
Exercise

Write a program to make the following figure:

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



2. Write programs to make the following figures. Come up with your own estimates for the sizes of the different lines:



Quiz

Qz1 What is a computer program? Explain to a friend (do this for all your quiz answers).

Qz2 What is a command?

Qz3 What does the forward command do?

Qz4 What is a unit length? Measure the length of a book in your bag. What is the unit of this length? How does this unit of length differ from the unit of length used by the forward command?

Qz5 What does the right command do?

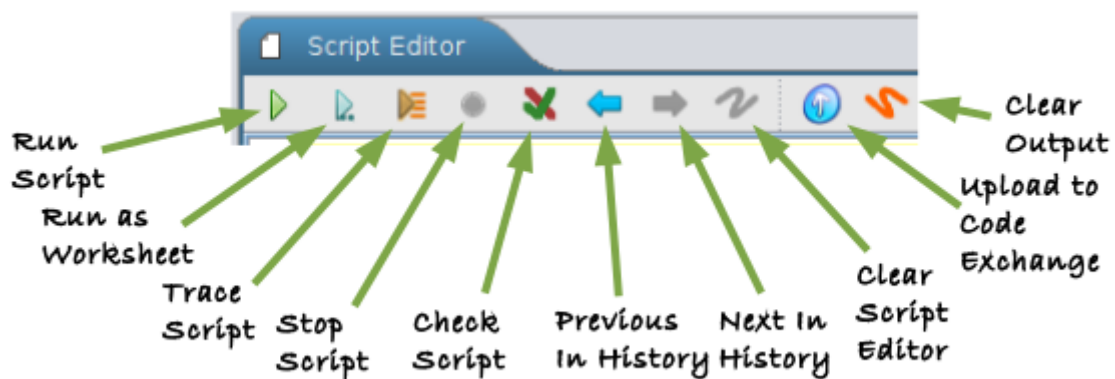
4 Using Kojo Effectively

This activity has the following desired goals:

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

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

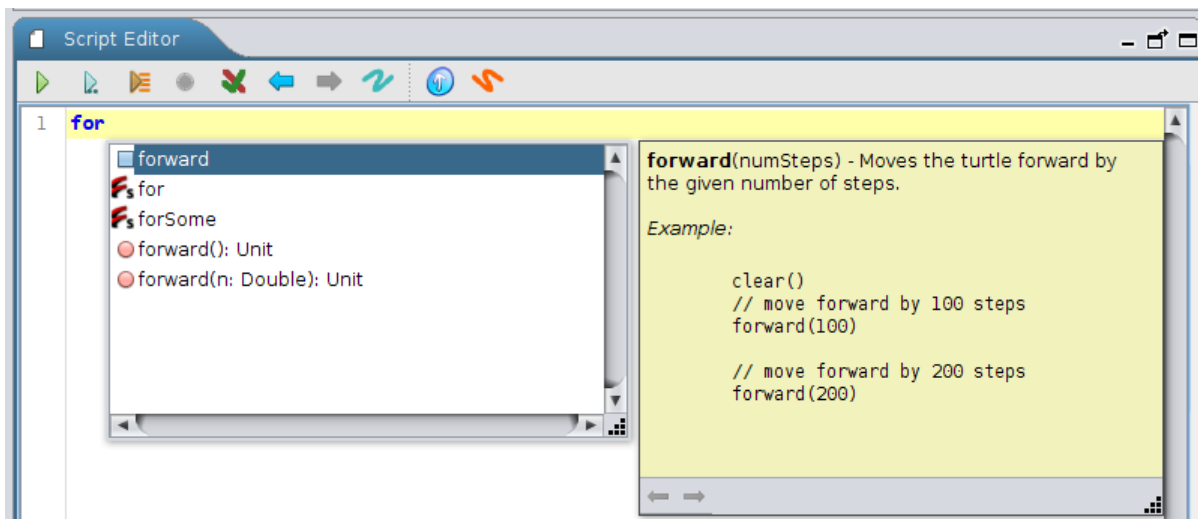
Explore history navigation – by using the History *Previous* and *Next* toolbar buttons .

Q2a What is program history?

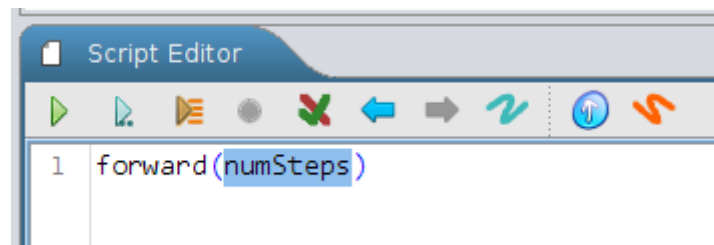
Q2b What do you think is the benefit of being able to navigate history?

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*



Quiz

Qz1 How does code completion help you to write programs? Explain to a friend (do this for all your quiz answers).

Qz2 Which button in the script editor toolbar do you expect to use the most? Why?

Qz3 What is program history in Kojo. How is it useful?

5 Drawing with Colors

This activity has the following desired goals:

- Learning to set the background color of the canvas (**M, A**).
- Learning to change the color and thickness of the lines drawn by the turtle (**M, A**).
- Learning to fill the shapes drawn by the turtle with color (**M, A**).
- Learning the `setBackground`, `setPenColor`, `setFillColor`, `setPenThickness`, and `left` commands (**M, A**).
- Applying the idea of ratio and proportion in constructing figures (**M, T**).
- Making geometric figures using the above ideas (**T**).

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 core 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`.

Note – many more colors are available through the `ColorMaker`, as you will see soon.

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.
-

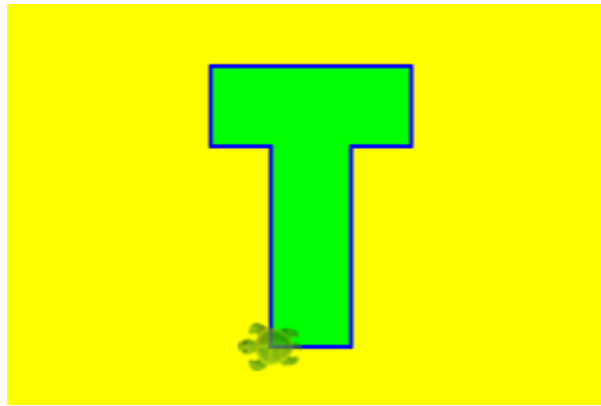
Brief command descriptions:

- `setBackground(color)`– sets the color of the drawing canvas.
- `setPenColor(color)`– sets the color of lines drawn by the turtle after that point in the program.
- `setFillColor(color)`– sets the fill color of enclosed areas drawn by the turtle after that point in the program.
- `setPenThickness(t)`– sets the thickness of lines drawn by the turtle after that point in the program.

Exercise

1. Write a program to make the following figure:

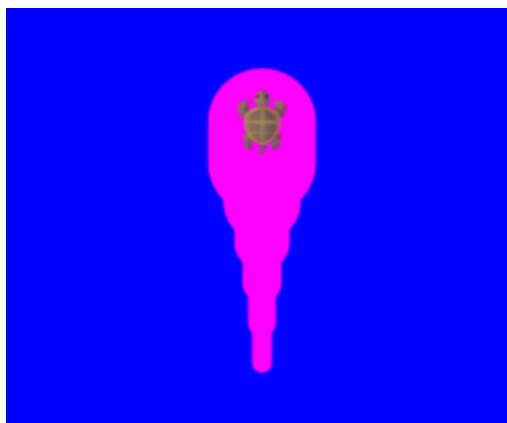
- The pen color is blue, and the fill color is green. The background color is yellow.
- To determine the dimensions of the figure, imagine that it is made out of two rectangles – a vertical one and a horizontal one, with the following specifications:
 - Vertical rectangle – $length = 120$ pixels, ratio 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.

If you find it tough to do the above math, estimate the thickness of each line – by guessing, visual inspection, and trial and error.



Quiz

Qz1 If you make a square using forwards and rights, and then do a `setPenColor(black)`, the square does not become black. Why? Explain to a friend (do this for all your quiz answers).

Qz2 If you do a `setFillColor(green)` and then make a three sided open figure, will the figure be filled with green? Find out by experimentation.

6 Hopping with Speed

This activity has the following desired goals:

- Learning how to control the speed of the turtle (**M, A**).
- Learning how to make the turtle move without drawing lines (**M, A**).
- Learning to hide the turtle (**A**).
- Learning the `setSpeed`, `hop`, and `invisible` commands (**M, A**).
- Learning to estimate the dimensions of a figure given the size of one line in the figure (**M**).
- Making geometric figures using the above ideas (**T**).

Step 1

Type in the following code and run it:

```
clear()
setSpeed(fast) // also try slow instead of fast.
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 `setSpeed` command does? What does the input to the command specify?

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 `setSpeed`, `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. The predefined speeds in `Kojo` are `slow`, `medium`, `fast`, and `superFast`.

Theory

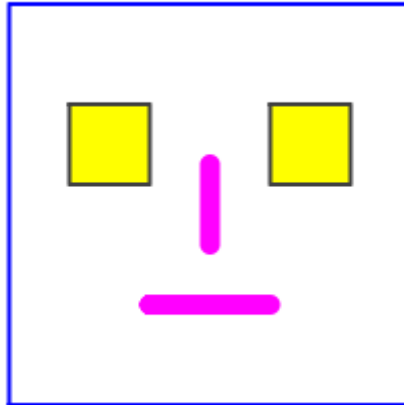
Brief command descriptions:

- `setSpeed(speed)` – sets the speed with which the turtle moves forward.
- `hop(n)` – moves the turtle forward `n` steps in the direction of its nose – without drawing a line.
- `invisible()` – hides the turtle.

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.



Quiz

- Qz1** How is the `hop` command useful? Explain to a friend (do this for all your quiz answers).
- Qz2** When is the `setSpeed` command especially useful?
- Qz3** Why would you want to make the turtle invisible?

7 More colors with the ColorMaker

This activity has the following desired goals:

- Learning to use the ColorMaker to get access to many more predefined colors (**M, A**).
- Using the Color Chooser Window to mix new colors (**A**).
- Learning about the HSL color model used to represent colors in computers (**M, A**).
- Applying the idea of ratios to determine sizes and colors in a given figure (**M**).
- Making geometric figures using the above ideas (**T**).

Step 1

Type in the following code and run it:

```
clear()
setSpeed(superFast)
setPenThickness(4)
setPenColor(ColorMaker.darkMagenta)
// you can use cm as an abbreviation for ColorMaker
setFillColor(cm.lightPink)
repeat(4) {
  forward(100)
  right(90)
}
```

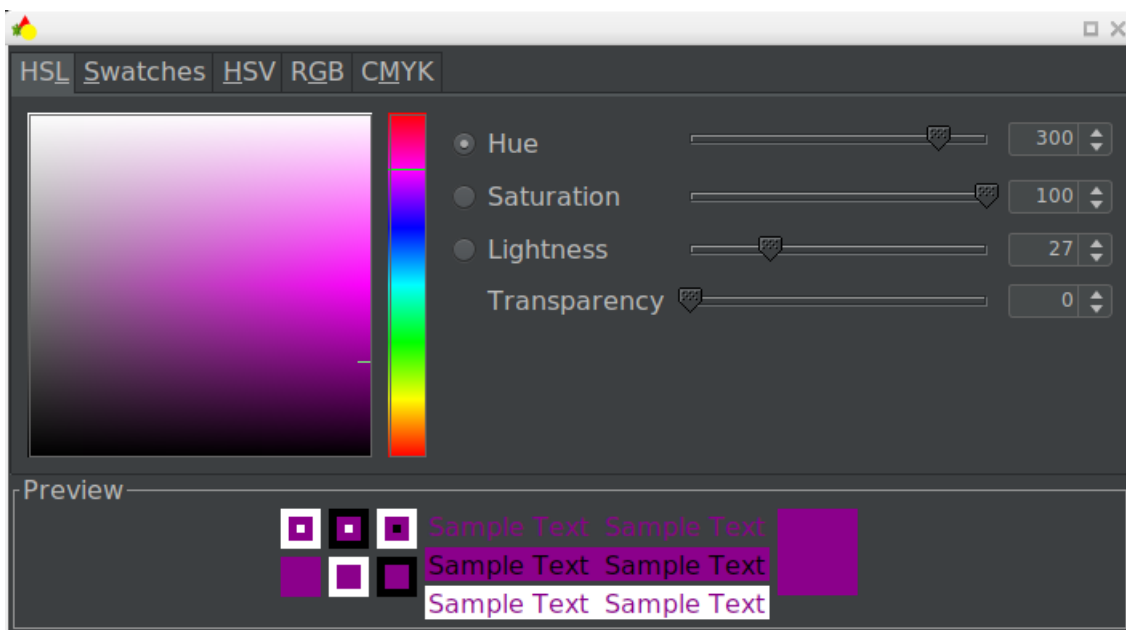
Q1a How many colors does the ColorMaker support? Use code completion to find out.

Self Exploration

Ctrl+Click (i.e., press the *Ctrl* key and then click the mouse) on the word `darkMagenta` in your code inside the Script Editor. This will bring up a color chooser (as shown on the next page). You can then interactively play with the pen color for the figure as described next. You can also do the same thing for the fill color of the figure.

In the color chooser, you can:

- Choose the basic color (a number between 0 and 360) via the Hue slider. 0 is red, 120 is green, 240 is blue, and 360 is again red.
- Add gray to the color via the Saturation slider. 100 is the pure color; 50 is half color and half gray; 0 is fully gray.
- Add white or black to the color via the Lightness slider. 50 is the pure color; numbers greater than 50 add more and more white. Numbers less than 50 add more and more black.
- Increase the transparency of the color via the Transparency slider.



Theory

- The ColorMaker uses the HSL color model for colors by default.
- The HSL color model represents colors in the computer using three values – the hue component, the saturation component, and the lightness component. These three components are described in detail in the Self Exploration section above. Millions of colors can be created using a mixture of these components.
- The Transparency value that you saw in the color chooser window allows you to control the transparency of the colors that you create; it is the opposite of opacity. A transparency of 0 means perfectly opaque. A transparency of 100 means perfectly transparent. Transparency values between 0 and 100 represent partially transparent colors.

Exercise

1. Write a program to make the following figure:

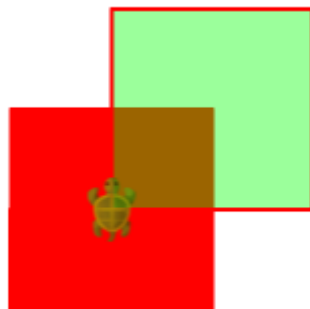
- Each arm of the cross – *length* = 50 pixels, ratio of *breadth* : *length* is 3 : 5.
- Fill color – use the Color Chooser Window to get to the right color.



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 transparency value that you need to determine by trial and error.

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*.



Quiz

Qz1 How many hues are available to you in the color chooser window? Explain to a friend (do this for all your quiz answers).

Qz2 If you want to make a color dull, how can you change it (in terms of changes to its hue, saturation, or lightness)?

Qz3 If you want to make a color dark, how can you change it?

Qz4 If you want to make a color bright, how can you change it?

8 Digging Deeper with Tracing

This activity has the following desired goals:

- Gaining a deeper understanding of how programs run (**M**).
- Learning to determine what portion of a drawing is made by which line in your program (**M**).
- Using symmetry and arithmetic to determine the dimensions of a figure (**M**).
- Making geometric figures using the above ideas (**T**).


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.

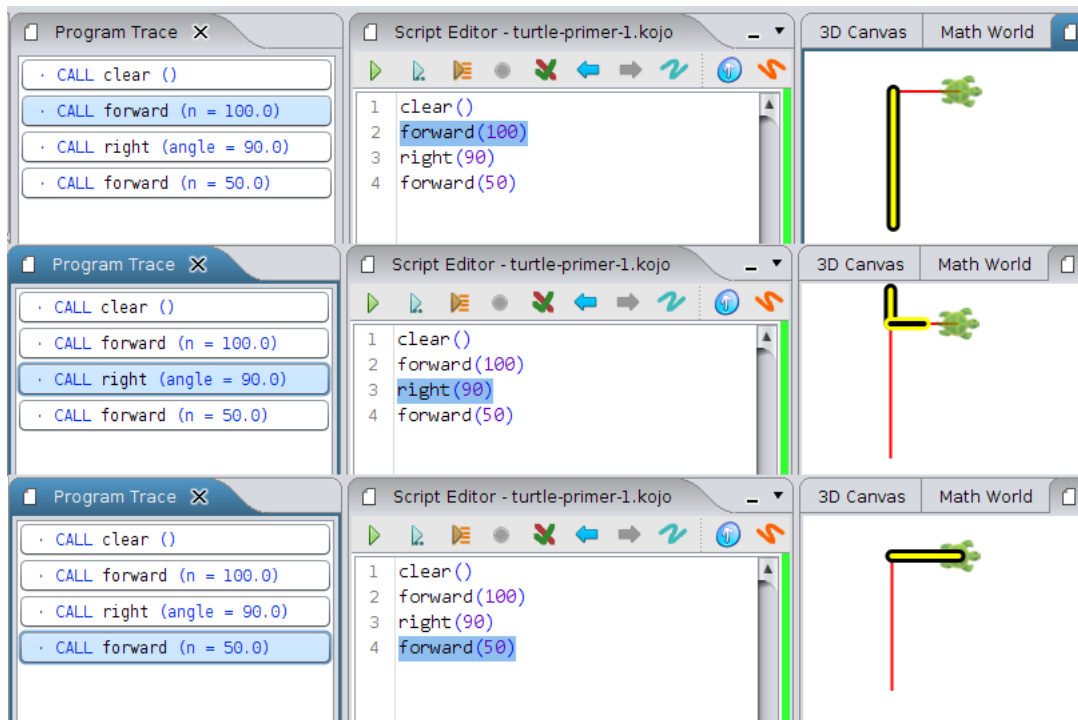


Figure 8.2: Output from tracing the program in Figure 8.1 on the previous page

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?

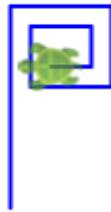
Self Exploration

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. Estimate the sizes of the other lines.
- Use tracing as much as you can while you write your program to try to understand what the program does as it runs.



Quiz

Qz1 Why would you want to trace a program? Explain to a friend (do this for all your quiz answers).

Qz2 When you click on a line in a program trace, what two things does it show you?

9 Angles

This activity has the following desired goals:

- Learning how to make the turtle turn through angles other than 90° (**A**).
- Exploring angles, and the idea of interior and exterior angles of a triangle (**M**).
- Using the idea of supplementary angles (**M**).
- Making geometric figures using the above ideas (**T**).

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° . Why does the turtle turn through 120° to make the above equilateral triangle? How does that relate to the idea of the interior and exterior angles of a triangle?

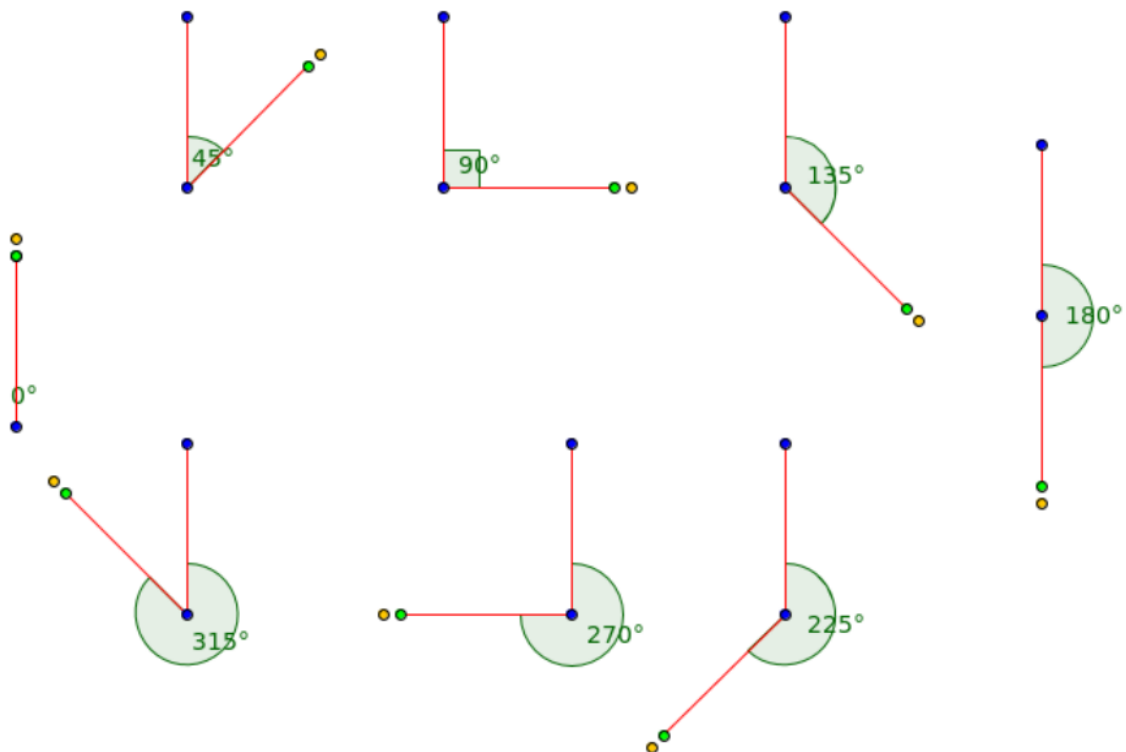
Self Exploration

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

Theory

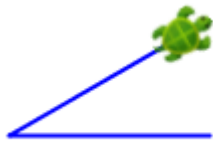
Here are some well known angles:

Well Known Angles



Exercise

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


 30°

 60°

 90°

 120°

 180°

Hint – use the idea of supplementary angles.

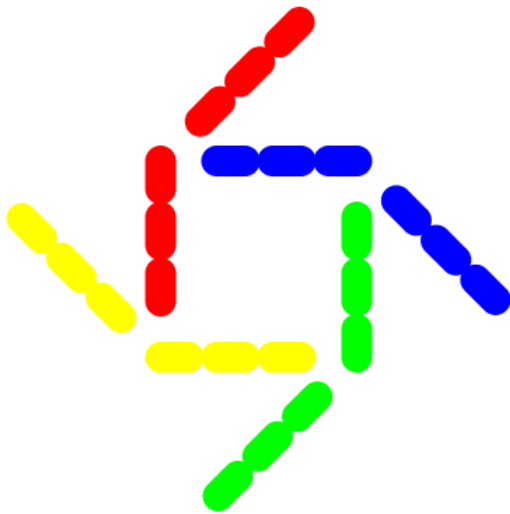
Quiz

Qz1 What is the smallest angle that the turtle can turn through? What is the largest angle? Explain to a friend (do this for all your quiz answers).

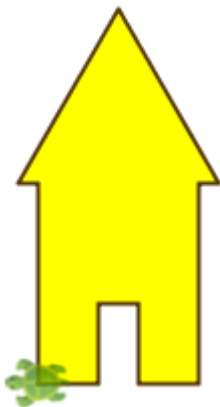
Qz2 Why are the ideas of exterior angles and supplementary angles important when deciding how much the turtle should turn to make a given figure?

10 Practice

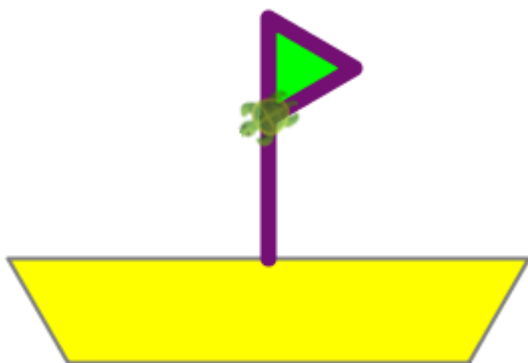
Write programs to make the following figures:



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 Repeating Commands

This activity has the following desired goals:

- Learning the repeat command (**M, A**).
- Learning about removing code duplication/repetition by using the repeat command (**M**).
- Remaking previously made shapes using repeat (**T**).

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.

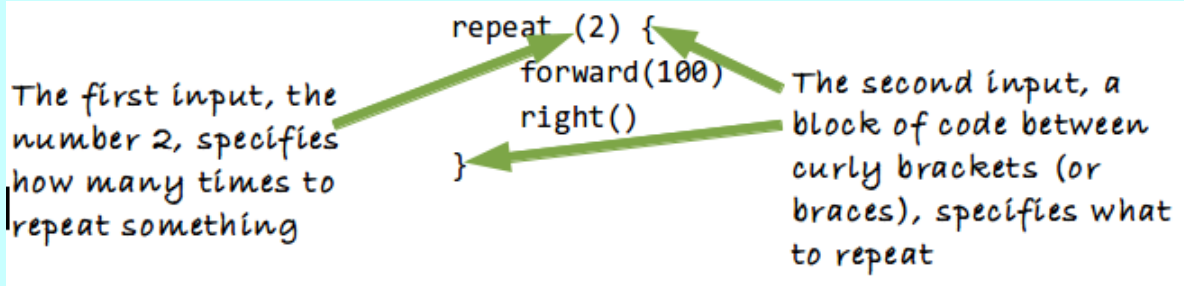
Q1b How many inputs does the repeat command take? What do these inputs signify?

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

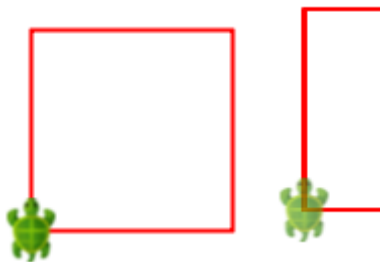
- The repeat command allows you to run other commands for a specified number of times. Note that repeat takes two inputs:



- 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:

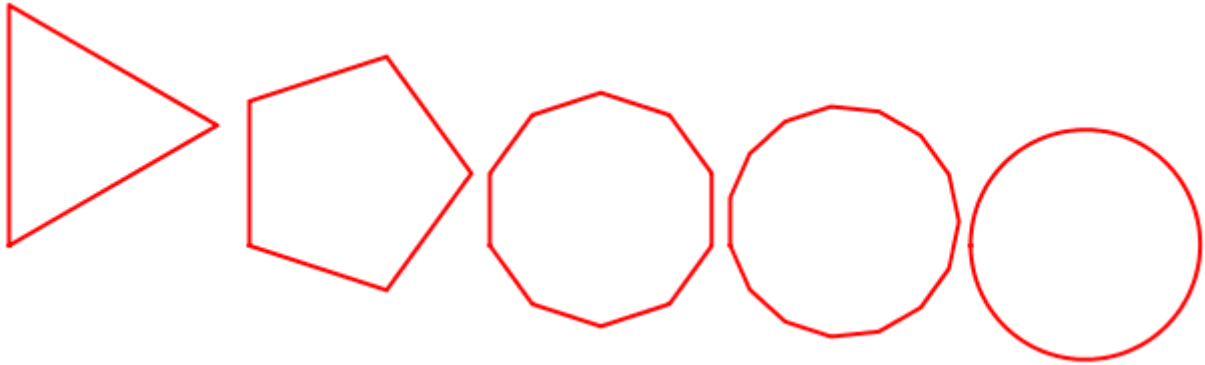


Quiz

Qz1 Why would you want to use the repeat command? Explain to a friend (do this for all your quiz answers).

12 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?

13 Exporting your artwork

You are starting to make very interesting art in Kojo. Wouldn't it be great if you could export the art out of Kojo? Once the art comes out of Kojo (as a PNG image file), there are many things you can do with it:

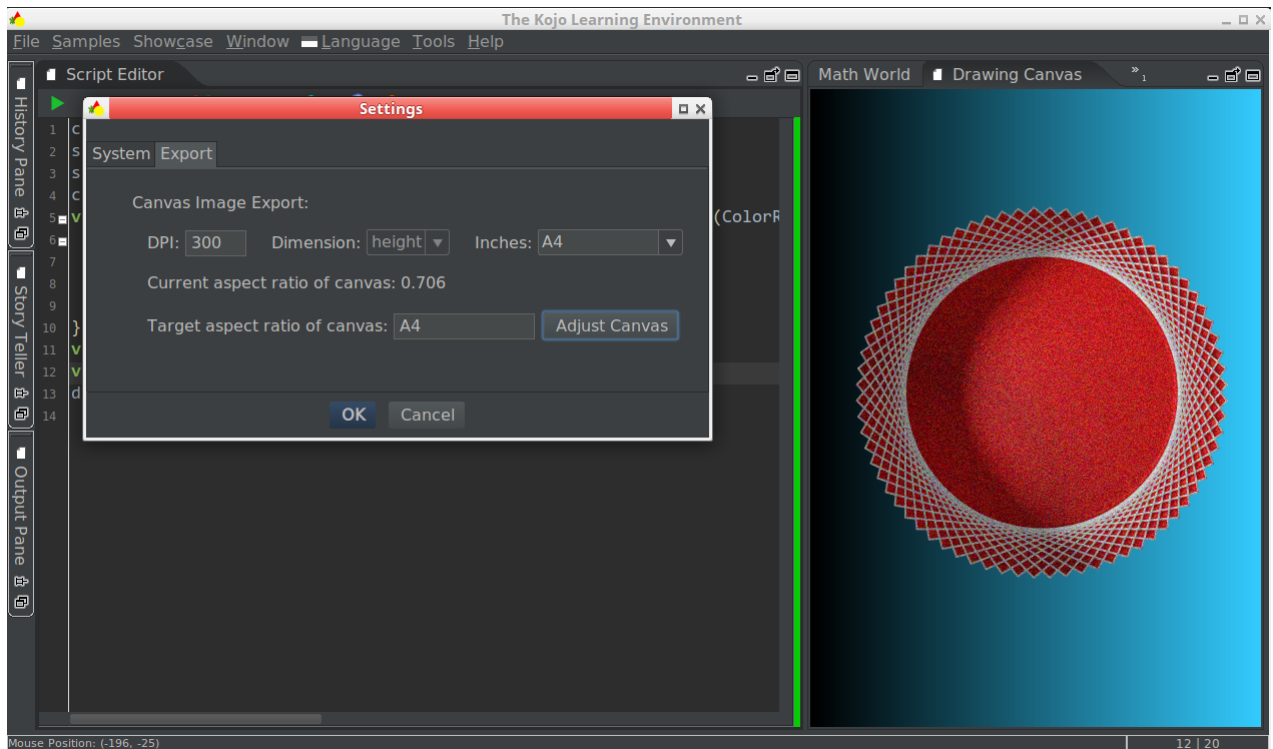
- Use it as desktop wallpaper.
- Print it on photo paper, frame it, and hang it on a wall.
- Print it on a t-shirt.
- Print it on a carry-bag.
- Print it on a mug.
- Do social and entrepreneurial events and projects based on printed artwork.
- And so on – the possibilities are endless.

So how do you get a good quality image of your drawing out of Kojo?

Here's what you need to do:

1. Set the resolution of the exported image (via *File -> Settings | Export*) as per the screenshot on the next page

- a) Set the *DPI* to 300
- b) Set the height to a standard paper size like *A4* (you can choose different paper sizes from the dropdown). Or just write the desired height in inches in the *Inches* text field.

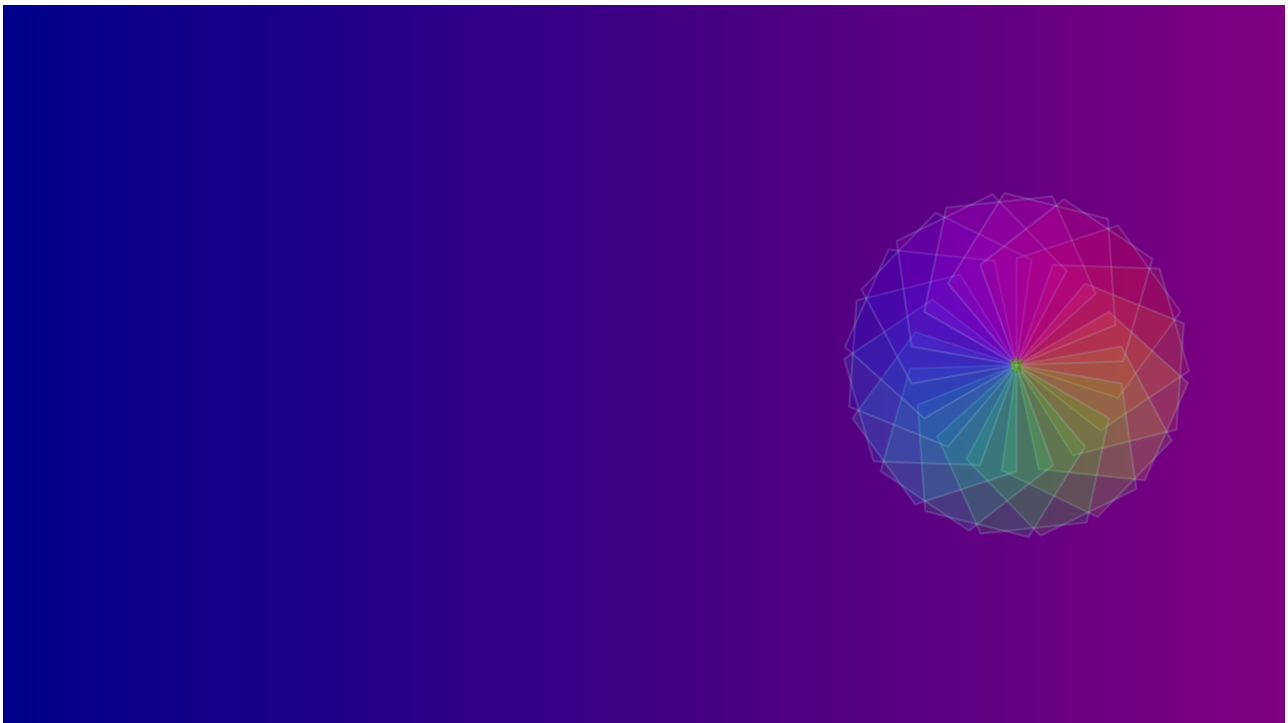


2. Resize the canvas if you are using a standard paper size – using the Adjust Canvas button shown above. This is because whatever is visible in the canvas is exported – with the same aspect ratio (the ratio of width to height) as the canvas. So if you want to export your drawing to a standard paper size, your canvas should have the same aspect ratio as that paper size.
3. Now close the Settings window.
4. Make sure the layout of your drawing (centering in the canvas, etc.) is fine as per the adjust canvas size.
5. Export your drawing (by right-clicking on the canvas and clicking *Export as Image*).

14 Art Project 1

Make a drawing of your own choosing. Export it as an image and make that image your desktop wallpaper.

Here is an example of the kind of wallpaper image that you can create:



15 Functions

This activity has the following desired goals:

- Learning about functions (**M**).
- Learning to use the `ColorMaker.hsl` function to create new colors (**M, A**).
- Making a geometric figure using the above ideas (**T**).

Step 1

Type in the following code and run it:

```
clear()
setSpeed(superFast)
setPenThickness(4)
setPenColor(cm.darkMagenta)
setFillColor(cm.hsl(120, 1, 0.5))
repeat(4) {
    forward(100)
    right(90)
}
```

Q1a Look at the line:

```
setFillColor(cm.hsl(120, 1, 0.5))
```

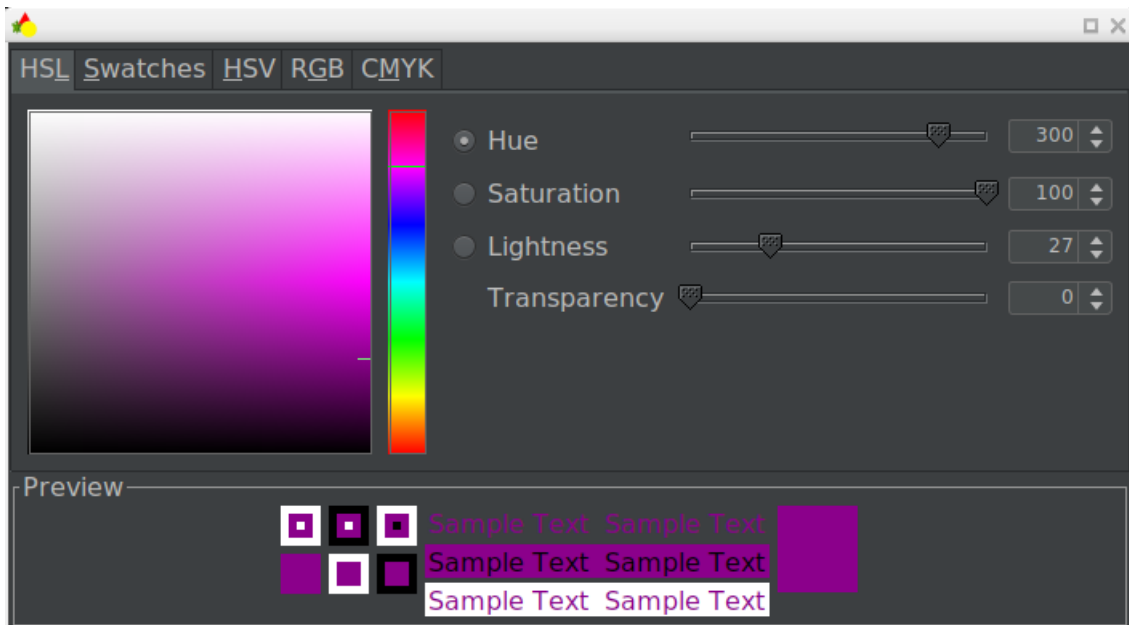
What do you think `cm.hsl(120, 1, 0.5)` does? To help you answer that, *Ctrl+Click* anywhere within the `cm.hsl(120, 1, 0.5)` instruction.

Q1b What do the three inputs to the `cm.hsl` 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(cm.hsl(240, 1, 0.5))`
2. `setFillColor(cm.hsl(240, 0.5, 0.5))`
3. `setFillColor(cm.hsl(240, 1, 0.25))`
4. `setFillColor(cm.hsl(240, 1, 0.75))`

Self Exploration

Bring up the color chooser window and try to relate between the hue, saturation, lightness, and transparency values that you see within the color chooser and the corresponding code within the script editor. For example, the following values in the color chooser correspond to `cm.hsl(300, 1, 0.27)`:



Theory

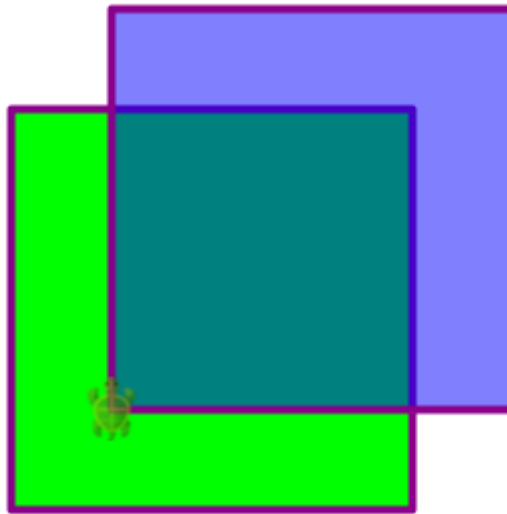
Let's expand the definition of programs.

- 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).
 - 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., `cm.hsl(120, 1, 0.5)` takes three number values as inputs and returns a `Color` value as an output.
Note – for now, be aware that `cm.hsl` refers to the `hsl` function that lives inside the `cm` object. Later on, you will learn more about objects and the functions (called methods) associated with them.
- Let's dig into the `cm.hsl` function; this function takes three inputs – the hue, saturation fraction, and the lightness fraction of the color that you want to create. Based on these inputs, the `cm.hsl` function creates a `Color` value using the HSL color model.
- If you want to create a transparent color, you need to use the `cm.hsla` function. This function takes an additional fourth argument which is the opacity fraction of the color being created.

Exercise

Write a program to make the following figure. Don't use the color chooser window to select the colors in the figure. Instead, use the `cm.hsla` function – based on the following information:

- The first (from the left) rectangle has a hue of 120, a saturation of 1.0, a lightness of 0.5, and an opacity of 1.0.
- The second rectangle has a hue of 240, a saturation of 1.0, a lightness of 0.5, and an opacity of 0.5.



Quiz

Qz1 What is the difference between a command and a function? Explain to a friend (do this for all your quiz answers).

Qz2 What's the hue number for red? For green? For blue?


Qz3 What color do you expect to have the hue number 60? Keep in mind your previous answer and think of a hue that is half-way between two of the hues from that answer (a hue that is halfway between two hues can be obtained by mixing those hues).

16 Calculations

This activity has the following desired goals:

- Learning to do calculations within Kojo (**A, M**).
- Learning about expressions (**A, M**).
- Exploring operator precedence and associativity (BODMAS) in expressions (**A, M**).
- Learning about some predefined Math functions in Kojo (**A, M**).
- Making a geometric figure using the above ideas (**T**).

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:

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

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

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

In Kojo, functions can be associated with objects or can be standalone.

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 (e.g. `cm.hsl(120, 1, 0.5)`). A function takes some values as inputs and computes and returns an output value or result 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:

```
mathx.lcm(24, 18) //> res3: Int = 72
```

```
mathx.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., `mathx.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.

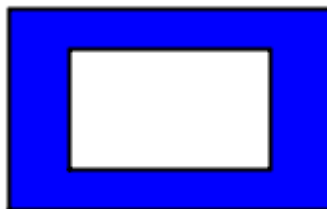
More on this later.

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



Quiz

- Qz1** What's an expression? Explain to a friend (do this for all your quiz answers).
- Qz2** Is `+` a command or an expression? Give reasons for your answer.
- Qz3** How can you change the order of evaluation of mathematical functions in an arithmetic expression? How does this relate to BODMAS?
- Qz4** What's a type?

17 Interlude - Computation

So what's a computer?

A device for doing computations.

And what is a computation?

A process that takes in information and generates new information.

Computations are the foundation of the digital world that's augmenting the physical world to add value (in terms of convenience, productivity, entertainment, and more) for humans. We package computations within computer programs, and that's why we are interested in programs – because they let us access the value-add of computations.

Every time you run a program, the computer creates a *process* to run it. This process then runs and evolves according to the instructions in the program, taking in information and generating new information as specified. The input information going into a program is related to what the user is trying to accomplish. The program then works on this information (by carrying out computations) to generate new, useful information. This useful information is made available to the user.

It's good to be aware of the difference between a program and the computational process that it carries out. To take an analogy - a program is similar to a cooking recipe, while a process is similar to the act of cooking based on the recipe. The computer (and more precisely, the CPU inside it) is the cook.

For much more information about computations and how they relate to programming and the operation of computers, take a look at the [computing essentials](#)¹ page on the kojodocs website.

To make the idea of computations (and the value they provides) more concrete, here are some examples of computations, their delivery, and their effect on the world:

What are some examples of computations that we consume every day?

Google search, Google maps, Amazon/Flipkart, Ola/Uber, Netflix, Banking, etc.,

How are these computations delivered to us?

Via (web)apps.

Exercise - Discuss the above with a friend or in class. Think about the computations that each of the above apps/services do to provide value to us.

¹<https://docs.kogics.net/concepts/computing-essentials.html>

Computations are getting more and more prevalent. Smart cities provide a good context to think about this:

- IoT, AI, Robots, Systems.
- Transportation (e.g. smart traffic lights).
- Health (linked body sensors <-> transportation <-> hospitals).
- Energy (smart buildings, smart road lighting).
- Security (cameras, sensors, social feeds).

Exercise - Discuss the above with a friend or in class. Think about different scenarios in a smart city where all these above services/apps play out.

Exercise - Identify a computation that is happening in your surroundings right now (other than on the device where you are reading this). How is this computation attempting to provide value to you?

18 Repeat inside repeat

This activity has the following desired goals:

- Learning to use repeat inside repeat to make complex figures (**M, A**).
- Learning to use randomColor (**M, A**).
- Learning to fade out a color (via fadeOut(factor)) to make it transparent (**M, A**).
- Making geometric figures using the above ideas (**T**).

Step 1

Type in the following code and run it:

```
clear()
setSpeed(fast)
setPenColor(darkGray)
repeat(18) {
    setFillColor(randomColor.fadeOut(0.5))
    repeat(5) {
        forward(100)
        right(360 / 5)
    }
    right(20)
}
```

- Q1a** What shape is made by the inner repeat in the code above?
- Q1b** How many times does the outer repeat make the inner-repeat shape?
- Q1c** How does the code above ensure that a full circle of (inner-repeat) shapes is made?
- Q1d** What do you think randomColor does? Run the program multiple times to find out.
- Q1e** What do you think fadeOut(factor) does? Play with the fade factor to find out.

Self Exploration

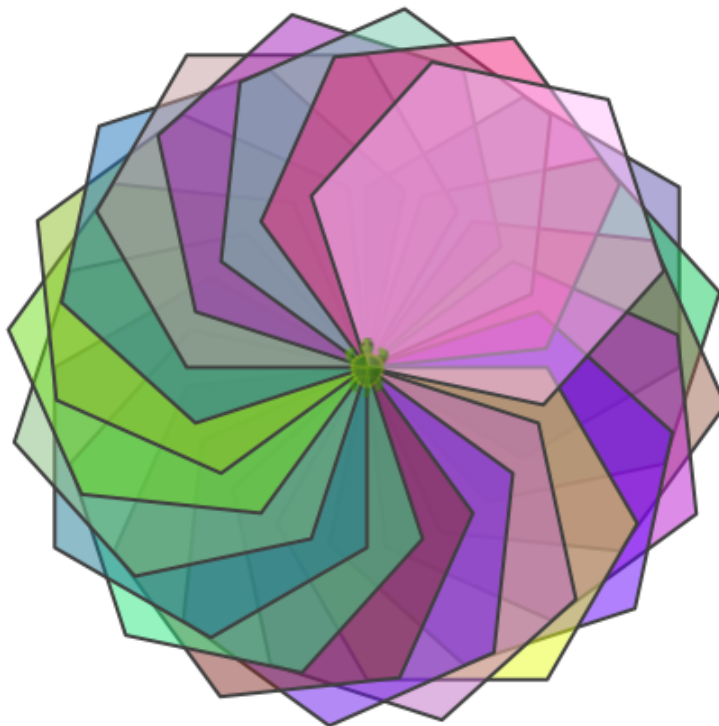
- Play with the inner-repeat to make a different inner shape.
- Play with the outer repeat number and the `right(20)` at the bottom of the code to try out different ways of repeating the inner shape.

Theory

- You can put a repeat inside a repeat inside a repeat as many times as you want.
- `randomColor` is a new kind of color in Kojo that you have not seen before. Every time you run it, it generates a new random color for you.
- If you want a color `c` to become transparent, you can do a `c.fadeOut(factor)`.

Exercise

Write a program to make a figure similar to the following:



Quiz

Qz1 Why does the exercise above ask you to make a figure *similar* to the given one, and not a figure exactly the same (as the given one)? Explain to a friend (do this for all your quiz answers).

Qz2 Is `randomColor` a command or a function?

Qz3 Is `color.fadeOut(factor)` a command or a function?

Qz4 If you want a color to become totally transparent via `color.fadeOut(factor)`, what should the factor be?

Qz5 If you want a color to remain totally opaque even after a call to `color.fadeOut(factor)`, what should the factor be?

19 Turning with a radius

This activity has the following desired goals:

- Learning how to make the turtle turn along an arc of a circle (**M, A**).
- Learning about the relationship between angles and the arc of a circle (**M, A**).
- Explorations with geometry and angles (**M, A**).
- Making geometric figures using the above ideas (**T**).

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)
```

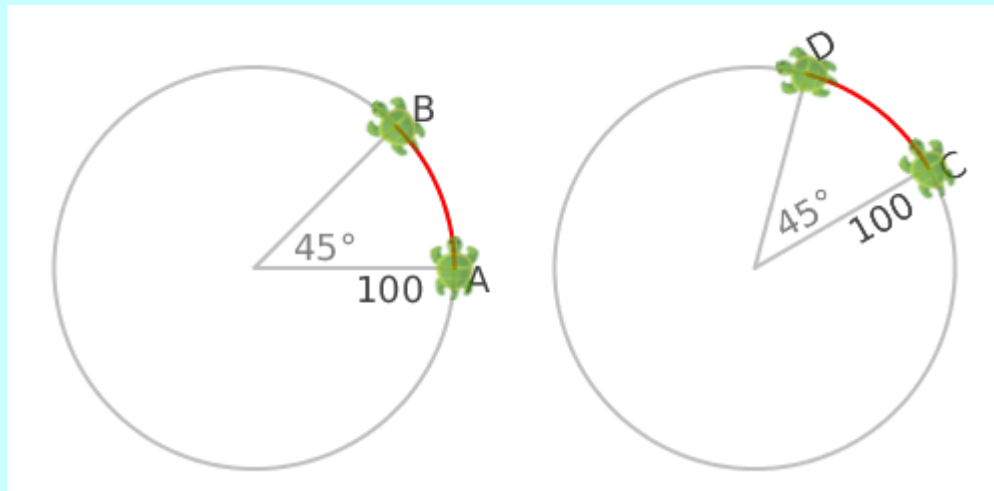
Make sure you understand how the program generates its output drawing. Tracing can help you with this.

Self Exploration

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

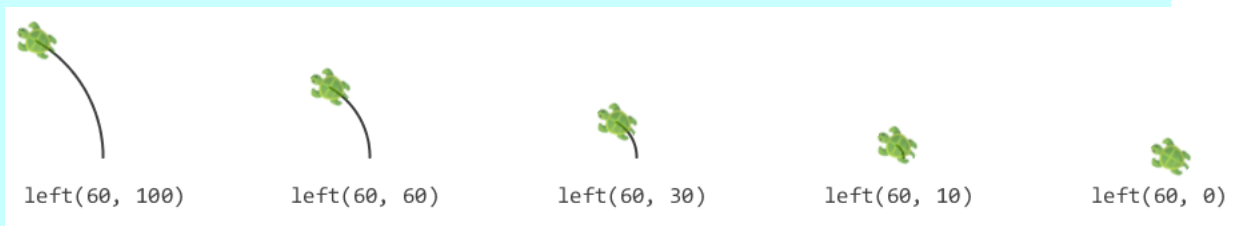
Theory

- 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° to make the figure.

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



Quiz

Qz1 When you use the command `right(90, 100)`, how does the radius of a circle come into the picture?

Qz2 The above command changes both the direction and the position of the turtle. How can you modify the command so that it changes only the direction of the turtle, and not its position?

20 More Fun with Repeat

This activity has the following desired goals:

- Learning to use repeat to make intricate figures (**M, A**).
- Learning the significance of 360° in making closed figures (**M**).
- Applying the idea of a lowest common multiple (LCM) to make interesting figures (**M, T**).

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

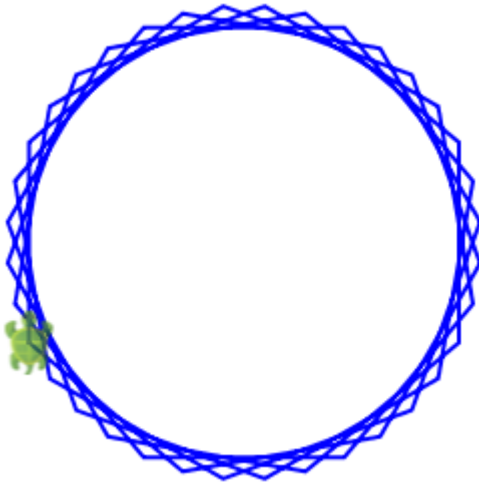
Play with the code in Steps 1 and 2. Try to determine what combinations of repeat count and turn angle make closed figures, and what combinations make open figures.

Theory

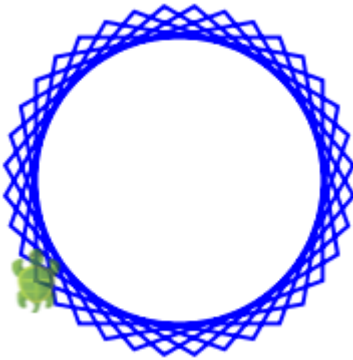
- To make a closed regular (i.e., equal-sided) figure, you need to turn the turtle through an angle of 360° or a multiple of 360° (why?).
- With repeat loops like the ones in steps 1 and 2, the total angle turned is $repeatCount \times turnAngle$
- So this should hold: $repeatCount \times turnAngle = n \times 360$
- 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° , 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

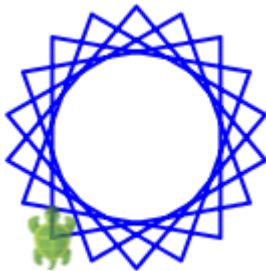
Write programs to make the following figures:



Turn angle = 50°



Turn angle = 70°



Turn angle = 100°



Turn angle = 110°

Quiz

Qz1 For a figure drawn using a repeat of `forward` and `right(turnAngle)`, for any given `turnAngle`, how can you calculate the smallest repeat count that will give you a closed figure?

21 Repeat with a sequence

This activity has the following desired goals:

- Learning to repeat things based on a sequence (**M, A**).
- Learning to do *similar but not exactly the same* things with repetition (**M, A**).
- Using an equation to determine the lengths in a figure (**M, T**).
- Becoming familiar with an arithmetic sequence(**M**).
- Making geometric figures using the above ideas (**T**).

Step 1

Type in the following code and run it:

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

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

Q1b What do you think the function `rangeTo(1, 3)` does?

To see what this function returns, you can open up a Kojo Scratchpad, type in `rangeTo(1, 3).toList` and run this line in worksheet mode. The `.toList` is needed to force the range to compute its values; ranges are lazy by default – they only compute their values when they need to.

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?

Q1d In the above code, change `rangeTo(1, 3)` to `rangeTill(1, 3)`. What do you think `rangeTill(1, 3)` does?

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

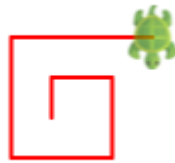
- Here, you give `repeatFor` a range/sequence of values – `rangeTo(1, 3)`, and give it a chunk of code to be repeated. The input range/sequence has three values inside it – (1, 2, and 3), so your repeat-code gets called three times. The first time, `counter` is 1. The second time, `counter` is 2. And the third time, `counter` is 3. If the input sequence had been from 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).
- `repeatFor` works with any sequence, not just a range. You will see more sequences later.

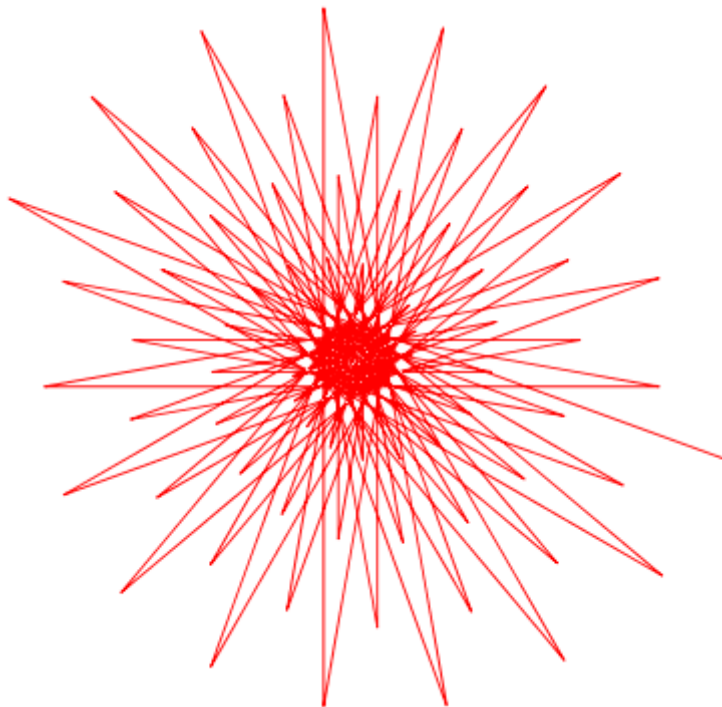
Exercise

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:

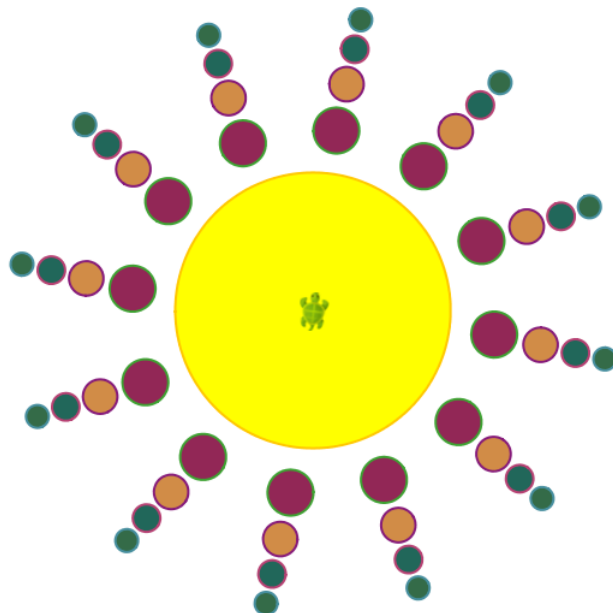
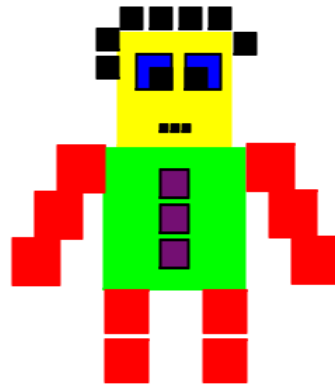


Quiz

Qz1 Why do you need `repeatFor` when you already have `repeat`? Explain to a friend (do this for all your quiz answers).

22 Art Breakout

Write programs to make the following figures:



23 Random Numbers and Named Values

This activity has the following desired goals:

- Learning about random numbers (**M, A**).
- Learning to use random numbers to make interesting figures (**M, T**).
- Learning to use the random function (**M, A**).
- Learning to save the results of functions for later use in a program (**M, A**).
- Learning about keyword instructions (**M, A**).
- Learning the `val` keyword instruction (**M, A**).
- Learning to nest functions within commands (**M, A**).
- Learning to save the turtle's position and heading, and restoring it later (**M, A**).
- Making a geometric figure using some of the above ideas (**T**).

Step 1

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

```
random(10, 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()
setBackground(white)
setSpeed(fast)
setPenColor(gray)
repeat(20) {
    savePosHe()
    setPenThickness(random(1, 20))
    setPenColor(cm.rgb(0, 30, 200, random(1, 255)))
    right(random(1, 360))
    forward(random(5, 100))
    restorePosHe()
}
```

Q3a What do you think the `forward(random(5, 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, 200)
repeat(2) {
    forward(len)
    right()
```

```
    forward(len / 2)
    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. The width of the rectangle is half the random height.

Self Exploration

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

Theory

- `random(l, h)` - returns a number between `l` (inclusive) and `h` (exclusive). Each time you run `random`, it returns a different (random) number. In this respect, it's different from a normal function – which always returns the same value for the same inputs. `random` is an impure function (because it returns a different result each time it is called); such functions are called queries.
- `savePosHe()` - saves the turtle's current position and heading, so that they can easily be restored later with a `restorePosHe()`
- `restorePosHe()` - restores the turtle's current position and heading based on an earlier `savePosHe()`
- 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, 200)
```
- 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).

Quiz

Qz1 What does the random function do? Explain to a friend (do this for all your quiz answers).

Qz2 Can you make the exact same figure as the above? If not, why?

Qz3 What's a keyword instruction?

Qz4 What does the `val` keyword instruction do?

24 Your own Commands

This activity has the following desired goals:

- Learning to create new commands within Kojo using the `def` instruction (**M**, **A**).
- Becoming familiar with the very important programming ideas of primitives, composition, and abstraction (**M**).
- Applying the idea of percentages to determine the dimensions of geometric figures (**M**, **T**).
- Making a geometric figure using the above ideas (**T**).

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?

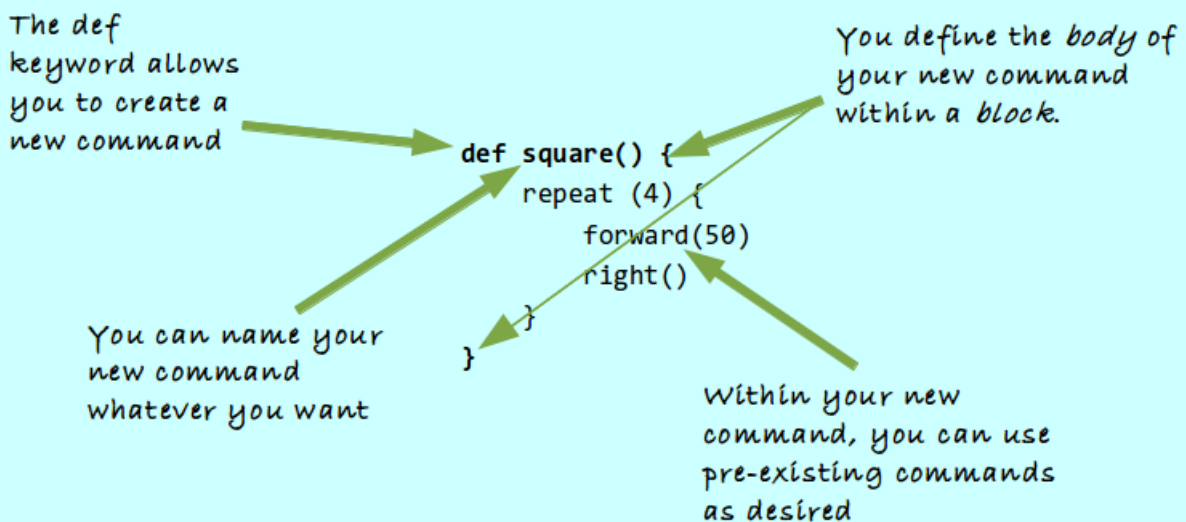
Self Exploration

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...

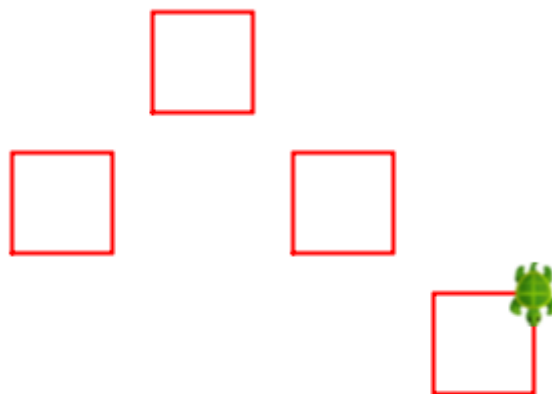
Computer programming is about three basic things:

- primitives – these are the instructions already available in your programming environment.
 - composition – this is how you combine primitives to do what is required.
 - abstraction – this is how you give useful 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 (i.e. once you define an abstraction, you can use it without thinking about how it is 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.



Quiz

Qz1 Why would you want to define your own commands? Explain to a friend (do this for all your quiz answers).

Qz2 What is a primitive?

Qz3 What is composition?

Qz4 What is abstraction?

25 Your own Commands, with Inputs

This activity has the following desired goals:

- Learning to create new commands that take inputs, thus enabling them to change their behavior based on input values (**M**, **A**).
- Learning to estimate the dimensions of a figure given the size of one line in the figure (**M**).
- Making a geometric figure using the above ideas (**T**).

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)
}
```

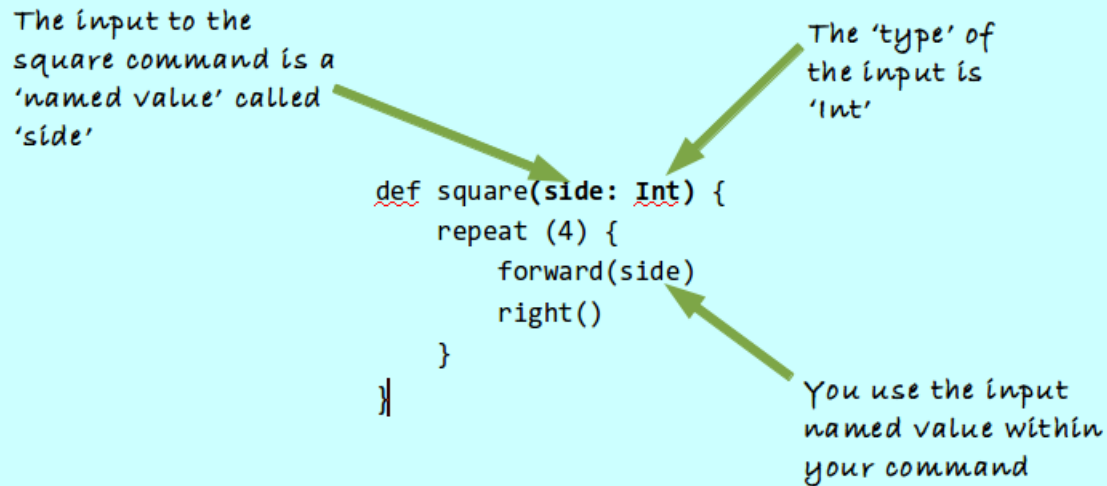
Q2a How is the code in Step 1 similar to the code in Step 2? How is it different?

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:



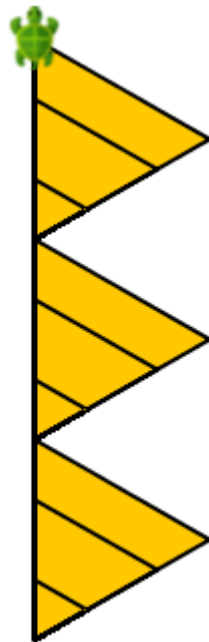
- Within the round-brackets in the first line of code above, you are telling Kojo that the input to the square command is called *side*, and that its type is *Int* (where, as you have seen earlier, *Int* stands for integer). Now, instead of always drawing squares of the same size, the square command 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.



Quiz

Qz1 How does it help if the new commands that you create can take inputs? Explain to a friend (do this for all your quiz answers).

Qz2 Why is the input to a command called a named value?

Qz3 Why is the type of the input an important piece of information that you need to provide to Kojo when you create your new command?

26 Polygon Art

This activity has the following desired goals:

- Exploring polygons with different numbers of sides (**M**).
- Exploring the interior and exterior angles of polygons (**M**).
- Learning about variables (**M, A**).
- Learning to programatically modify colors (**M, A**).
- Learning the `var` keyword instruction, and the `spin` function (**M, A**).
- Making a geometric figure using the above ideas (**T**).

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()
setSpeed(fast)
var fill = cm.hsla(115, 1, 0.5, 0.6)
repeat(7) {
  setFillColor(fill)
  repeat(5) {
    forward(100)
```

```
    right(360 / 5)
  }
  right(20)
  fill = fill.spin(60)
}
```

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

Q4b What do you think the `spin` function does?

Self Exploration

Play with the code in the above steps as you see fit.

Theory

Variables

- The `var` keyword instruction allows you to create a name and bind it to a value:

```
var fill = cm.hsla(115, 1, 0.5, 0.6)
```

The name is on the LHS of the `=` sign, and the value is on the RHS.

- You can then use the name instead of the value in the program (just like you do with named values created with the `val` keyword instruction):

```
setFillColor(fill)
```

- The name that you bind to a value with the `var` sounds very similar to the name that you give to a value with `val`.

How is it different?

You can detach the `var` name from its current value and bind it to a different value.

You cannot do this with names given to values with `val`.

- Here is how you change a `var` binding:

```
fill = fill.spin(60)
```

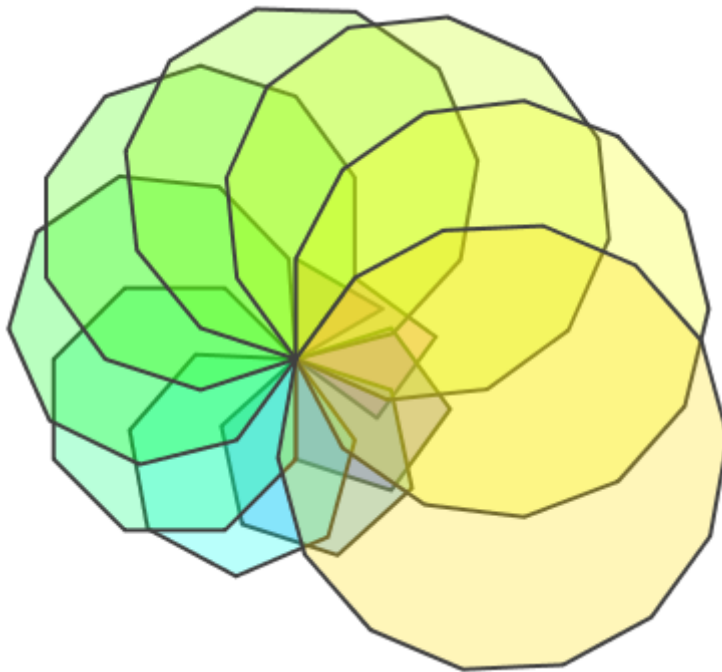
- So we call a name created with `var` a variable, because it can refer to different (variable) values at different places in a program. After changing a variable's value, the next time you use the variable in your program, its new value is picked up.

-
- The `color.spin(angle)` function returns a new color whose hue is the sum of the hue of the given `color` and the given `angle`.

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.



Quiz

Qz1 What's the difference between a `val` and a `var`? Explain to a friend (do this for all your quiz answers).

Qz2 What happens when you spin a color?

27 Your own Functions

This activity has the following desired goals:

- Learning to create new functions (**M, A**).
- Playing with *simple interest* to make a figure (**M**).

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 $+$, $-$, $*$, $/$, `math.max`, `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, `amt(P, R, 1)` is equivalent to the value that it calculates (110). Any occurrence of `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:

$$y = 3x + 4$$

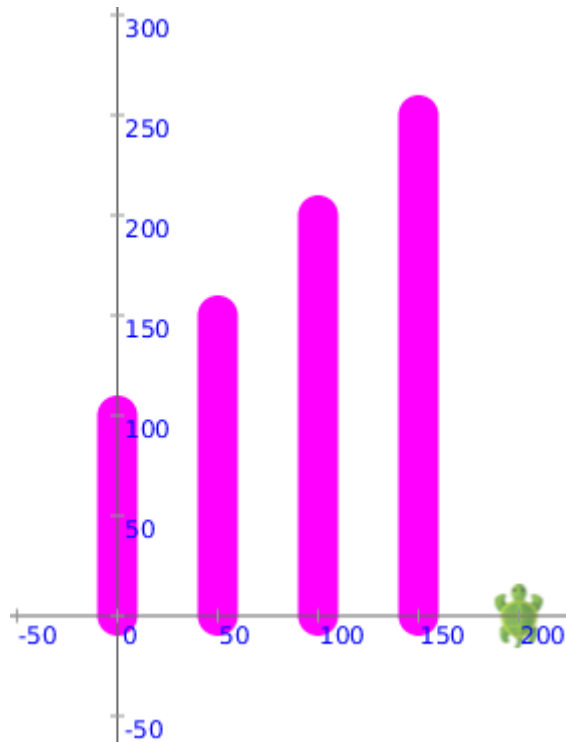
These types of equations allow you to easily find, given the value of one variable (x), the value of the related variable (y). These equations are called functions.

- 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.



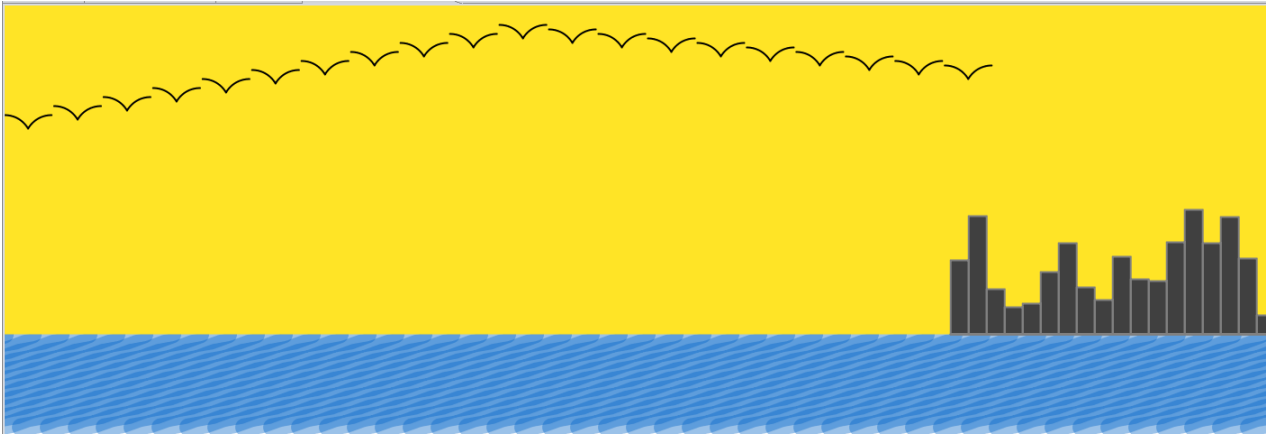
Quiz

Qz1 Why would you want to define your own functions? Explain to a friend (do this for all your quiz answers).

Qz2 What is simple interest?

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 Art Project 2

Make a drawing of your own choosing. Export it as an image and make that image your desktop wallpaper.

30 Strings and I/O

This activity has the following desired goals:

- Learning about queries (**M**)
- Learning to use the `readln` and `readInt` functions (**M, A**).
- Learning to use the `clearOutput` and `println` commands (**M, A**).
- Learning about program input and output (**M**).
- Learning about strings (**M, A**).
- Learning about string interpolation (**M, A**).
- Playing with the idea of number averages (**M, T**).

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 in any input from outside? 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 type of 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"`? Run the program after deleting the 's' to find out.

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 can call them command-queries. They carry out a command, and then get a value from the environment back into the program via a query.
- A query lets you access (a potentially changing) value in the program's environment (e.g. `position`, `heading`, `randomColor`, `random(10)`). A query is like a function in that it returns a value, but is different from a function in that it returns a different output value each time it is used with the same input value(s). You can think of a query as an impure function.
- Next, let us look at Strings, which are used by all three instructions mentioned above. Strings are:
 - used to represent (human language) 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. But there is one exception to this. If you create a string like this – `s"my name is: $name, and my age is $age"`, Kojo will plug in the values for the named-values that you provide into the created string. This kind of string is called an interpolated string. Note the use of the `s` prefix to create the string, and the use of the `$` sign to plug in values into the string.
- You can plug in the value of any expression into an interpolated string by using `${expression}`. Note the use of curly brackets to surround the expression.

Exercise

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

Quiz

Qz1 What's a string?

Qz2 What's a query?

Qz3 How can you put the value of an expression inside a string?

Qz4 What do `readln` and `readInt` do?

Qz5 What do you think `readDouble` does?

Qz6 What does `println` do?

31 Artistic Text in the Canvas

This activity has the following desired goals:

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

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 (`20`) 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()
setSpeed(fast)
val msg = "Hello There"
val msgLen = msg.length
repeatFor(msg) { c =>
    left()
    write(c)
    right()
    right(180.0 / (msgLen), 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.
- A string can be treated as a sequence of characters. When you use `repeatFor` with a string, you get a chance to work with every character of the string in your loop.

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:



Quiz

- Qz1** How do you get the turtle to write text in the canvas?
- Qz2** How can you change the font of this text?
- Qz3** How can you determine what fonts are available on your computer?
- Qz4** How can you get the turtle to write curved text on the canvas?

32 Conditionals

This activity has the following desired goals:

- Learning about conditionals (**M, A**).
- Becoming familiar with the important programming idea of selection (**M, A**).
- Learning to work with numeric conditions (**M, A**).
- Making a geometric figure using the above ideas (**T**).

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. Think in terms of commands vs expressions.

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.
- `if-else` works with both commands and expressions (as you saw in step 1 vs step 2).

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.

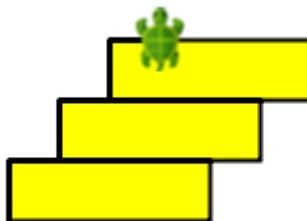
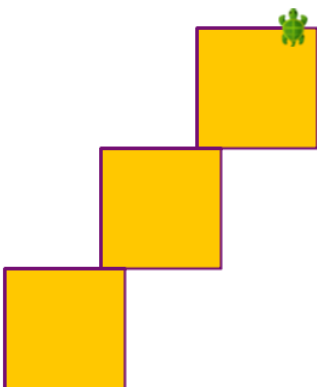
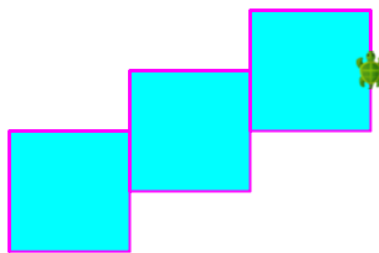
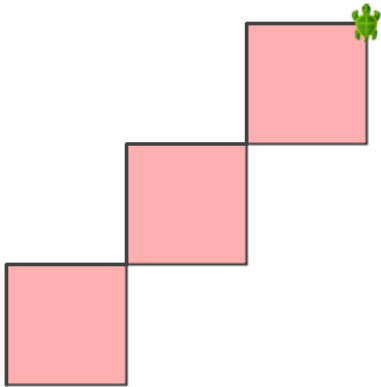
Quiz

Qz1 What is selection?

Qz2 How does `if-else` work in different ways for commands vs expressions?

33 Pattern Drawing Practice

Write programs to make the following patterned figures:



34 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?

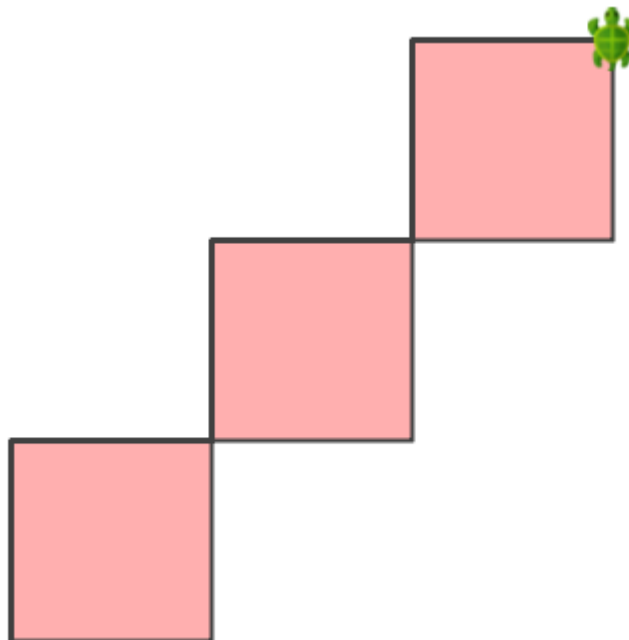
A pattern is something that contains a repeated building-block; the building-block is repeated to make the pattern.

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.

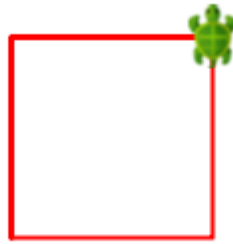
Step 1

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.

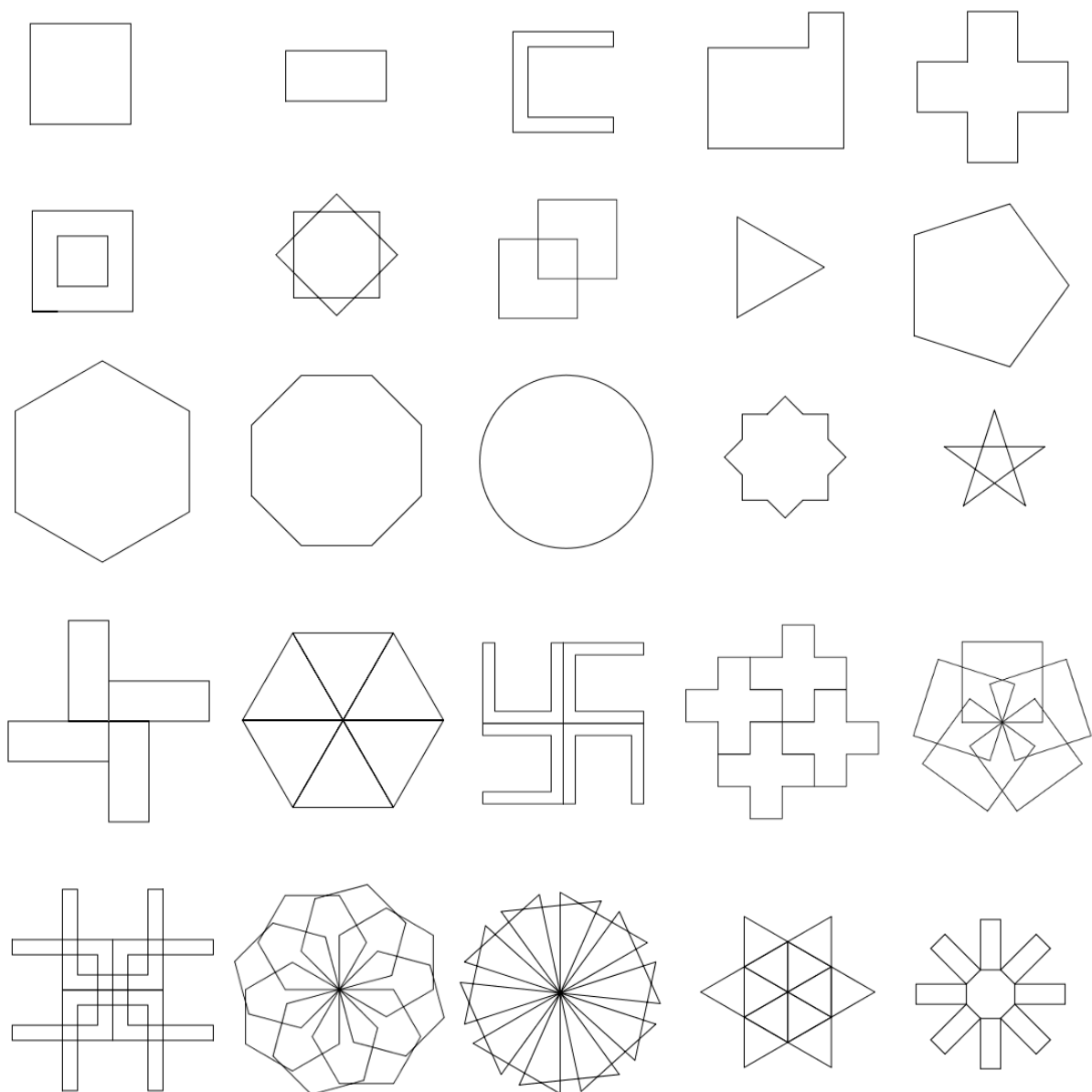
```
def squareBlock() {
    // start of building-block
    repeat(4) {
        forward(100)
        right()
    }
    hop(100)
    right()
    hop(100)
    left()
    // end of building-block
}
clear()
setAnimationDelay(100)
setPenColor(darkGray)
setFillColor(pink)
repeat(3) {
    squareBlock()
}
```

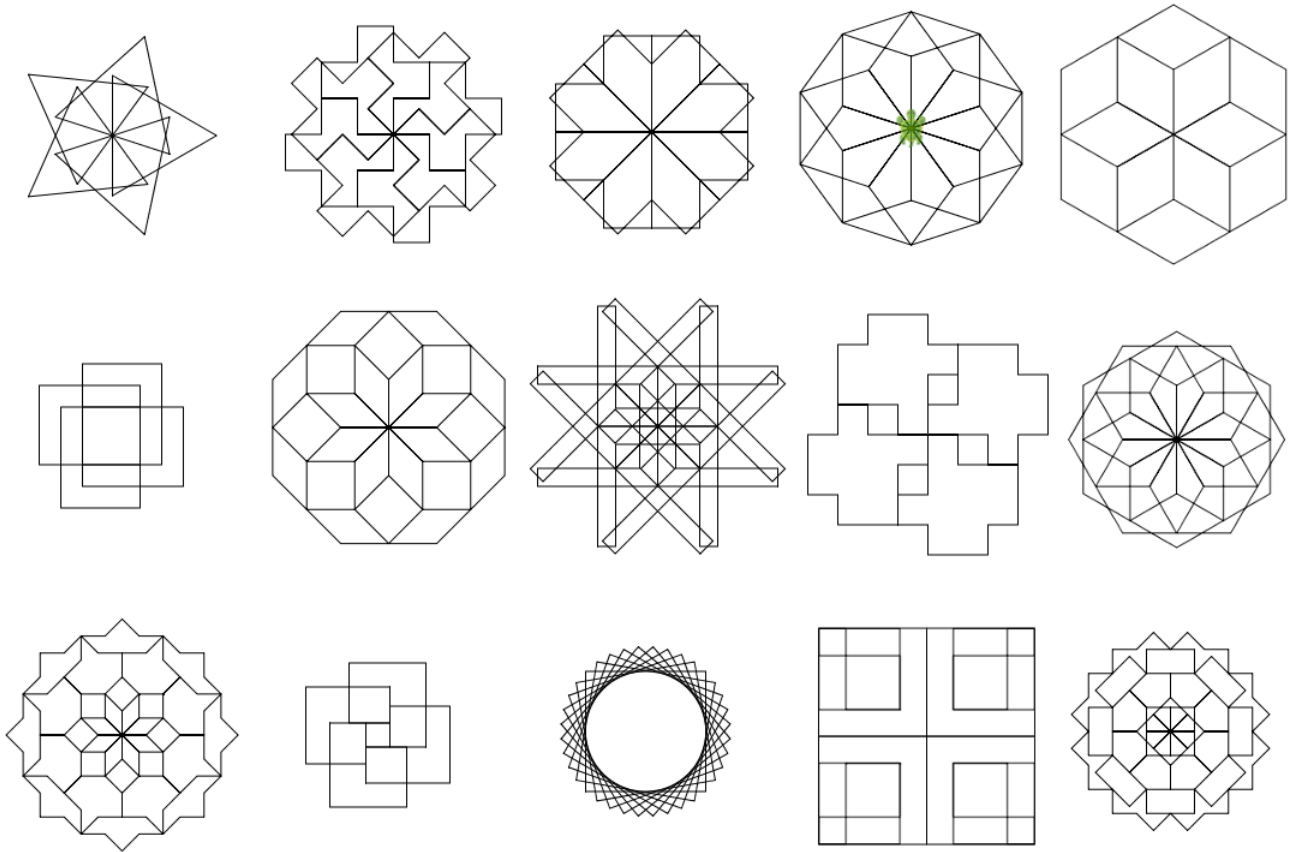
Make sure you understand how this program makes the pattern.

- 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?

35 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*.





36 Designing and making art

The content for this activity is available online.

Become familiar with the shape-block idea for designing and creating art from the following web-page:

<https://docs.kogics.net/ideas/turtle-shape-block.html>

Exercise

Make one drawing each (for a total of four drawings) based on the four methods described in the above web-page.

Quiz

Qz1 What's the shape-block idea for designing an artistic piece?

Qz2 What are the four shape-block methods?

37 Final Project

Make a drawing of your own choosing. Export it as an image and get the image printed on a t-shirt via sublimation printing (do a google search to find a vendor near you who can do this).

38 Programming Quickref

See the programming-quickref online at:

<https://docs.kogics.net/concepts/computing-essentials.html>

39 Turtle Commands Quickref

See the latest version of the [turtle command reference online](#).

Command	Description
<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.

Command	Description
<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.