Getting started with Kojo

Programming. Math. Art. Analysis.

by

Anusha, Aditya, and Lalit Pant

Version: August 6, 2018

A word about Kojo

Kojo is a learning environment where *youngsters* (from ages 8 to 80!) *play, create, and learn*. They play with small Scala programs. They create drawings, animations, games, and Arduino based intelligent circuits (with appropriate additional hardware). And they learn logical and creative thinking, programming, problem solving, math, physics, emotional grit, collaboration (via pair programming), and a lot more. Very importantly, they also learn how to *learn with understanding*. All of this fosters in them a mindset of exploration, innovation, self-reliance, *growth*, mental discipline, and teamwork – with Kojo as the enabler.

A Note for Facilitators and Teachers

This book contains a series of activities for kids to play with.

Most activities contain a fully defined program and a picture of the output of the program. For such activities, ask a kid to type in the instructions of the program into the script editor, run the program, and then check that the actual output of the program matches the output shown in the book. Then, ask the kid to do some reflection, i.e., think about and discuss what was just learned.

Many activities contain new instructions. Ask kids to keep an eye out for this and to figure out what the new instructions do.

Some activities contain an incomplete program, with the incomplete areas marked with `???`, and a picture of the expected output of the (complete) program. For such activities, ask a kid to type in the program into the script editor, fill out the incomplete portions of the program by thinking about and applying what they have learned before, run the program, and then (as before) check that the actual output of the program matches the output shown in the book. This should be followed by some reflection, as before.

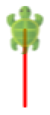
The activities as described above support sequences of (a) guided work, (b) exploration, and (c) challenges (marked with `???`) that need to be carried out . The challenges are very important, as they are the points in the learning material that focus on learning with understanding.

As kids go about doing these sequences of activities, you should encourage the following:

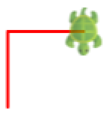
- exploration, discovery, and a sense of play.
- perseverance in the face of unexpected results, and joy in the process of figuring out what went wrong.
- commitment to solving the challenges.
- reflection and discussion about what was learned.
- digressions and diversions from the provided sequence of activities.

It is not important to finish all the activities. But it is vitally important to spend time with, go deep into, enjoy, and learn from each activity!

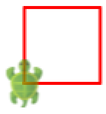
```
clear()
forward(50)
```



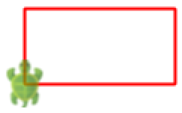
```
clear()
forward(50)
right()
forward(50)
right()
```



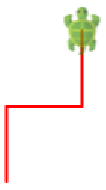
```
clear()
forward(50)
???
```



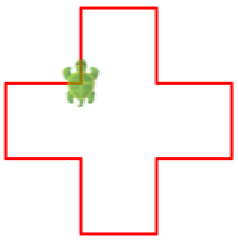
```
clear()
forward(50)
???
```



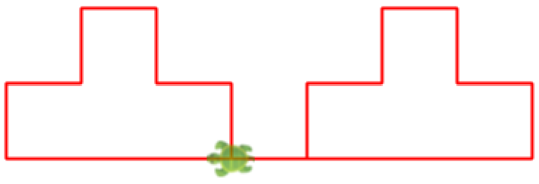
```
clear()
forward(50)
right()
forward(50)
left()
forward(50)
```



```
clear()
???
```



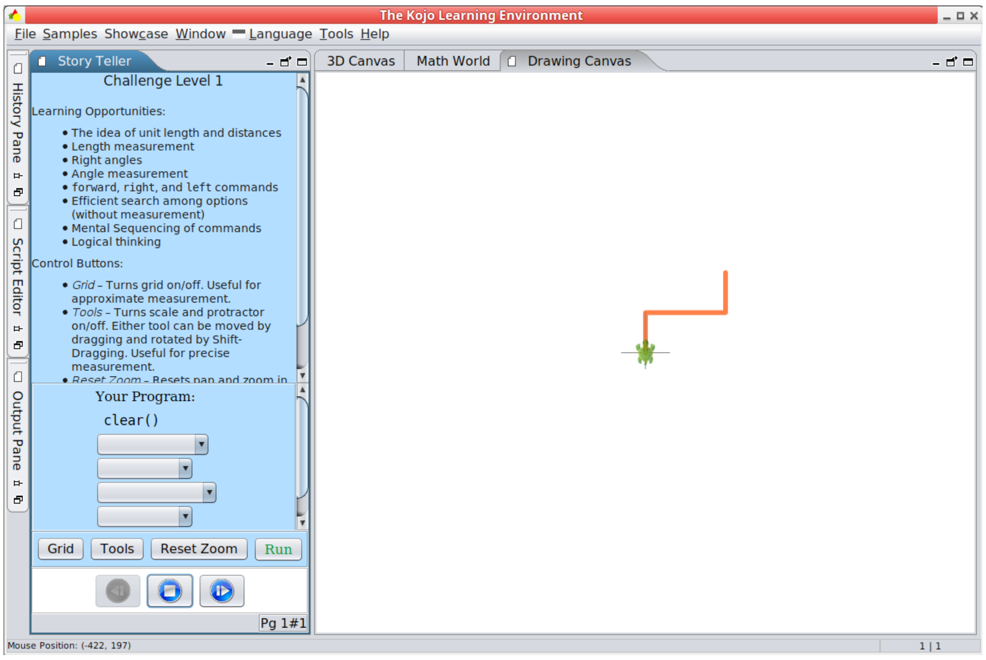
```
clear()
setSpeed(medium)
???
```



A couple of Interactive Activities

Beginner Challenges

At this point, open up the Beginner Challenges by going into *Tools -> Beginner Challenges* in the top level Kojo Menu. Go through the challenges:



Challenge Level 1

Learning Opportunities:

- The idea of unit length and distances
- Length measurement
- Right angles
- Angle measurement
- forward, right, and left commands
- Efficient search among options (without measurement)
- Mental Sequencing of commands
- Logical thinking

Control Buttons:

- *Grid* – Turns grid on/off. Useful for approximate measurement.
- *Tools* – Turns scale and protractor on/off. Either tool can be moved by dragging and rotated by Shift-Dragging. Useful for precise measurement.
- *Reset Zoom* – Resets pan and zoom in

Your Program:

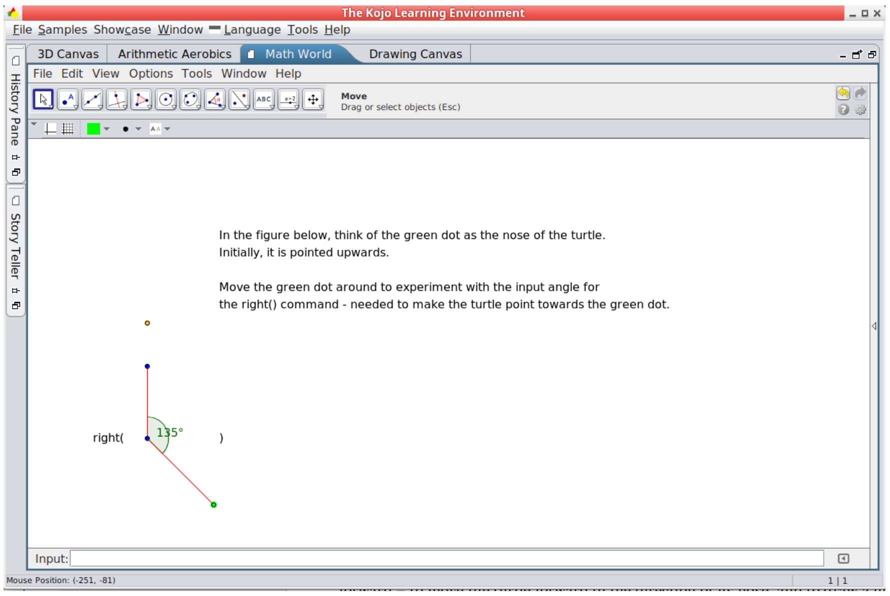
```
clear()
```

Playing with Angles

As you know, for drawing with the turtle, you have two basic commands available to you:

- `forward` – to move the turtle forward in the direction of its nose, and to draw a line as it moves forward.
- `right` (or `left`) – to change the direction (or heading) of the turtle's nose.

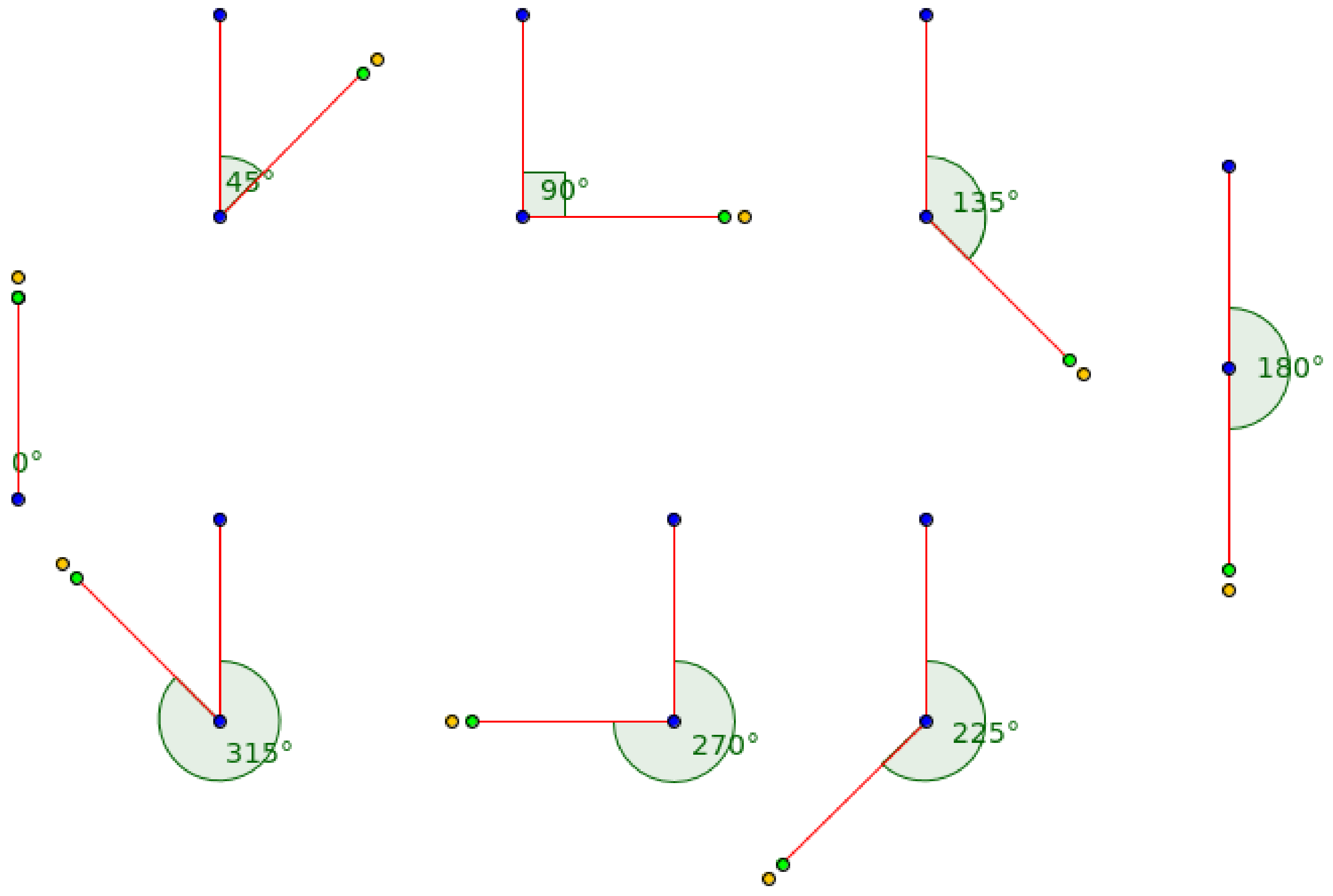
Before you move any further, it is important for you to understand how the `right` (or `left`) command changes the direction of the turtle's nose. To experiment with this, go to *Samples -> Math Learning Modules -> Playing with Angles* in the Kojo Menu (and also look at some well known angles shown on the next page):


The Kojo Learning Environment
File Samples Showcase Window Language Tools Help
History Pane
Story Teller
3D Canvas
Arithmetic Aerobics
Math World
Drawing Canvas
File Edit View Options Tools Window Help
Move
Drag or select objects (Esc)
right(
135°
)
Input:
Mouse Position: (-251, -81)
1 | 1

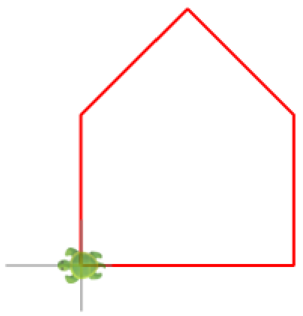
In the figure below, think of the green dot as the nose of the turtle.
Initially, it is pointed upwards.

Move the green dot around to experiment with the input angle for
the right() command - needed to make the turtle point towards the green dot.

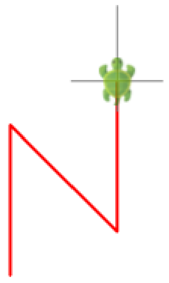
Well Known Angles



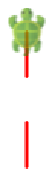
```
clear()
beamsOn()
setSpeed(fast)
forward(100)
right(??)
forward(100)
???
```



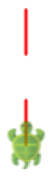
```
clear()
beamsOn()
setSpeed(fast)
forward(100)
???
```



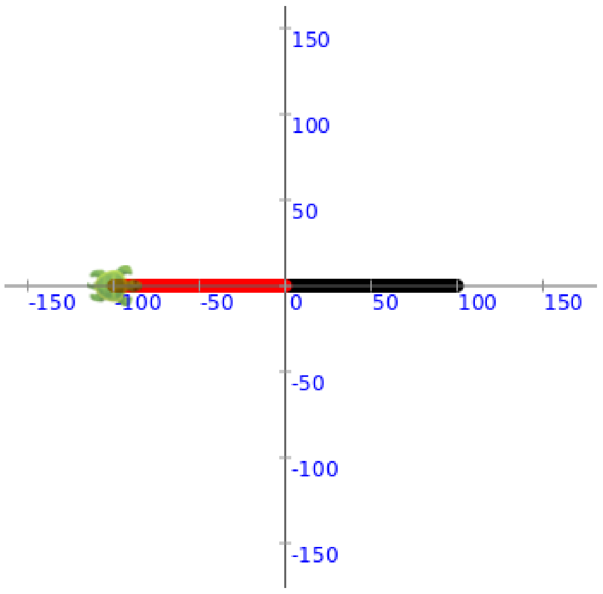
```
clear()
forward(30)
hop(30)
forward(30)
```



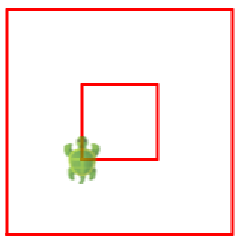
```
clear()
forward(-30)
hop(-30)
forward(-30)
```



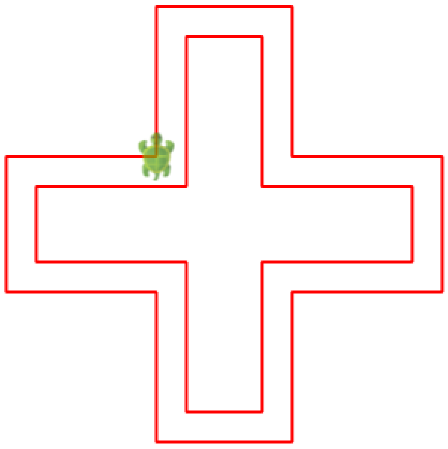
```
clear()
showAxes()
setPenColor(black)
setPenThickness(8)
right()
forward(100)
hop(-100)
setPenColor(red)
forward(-100)
// does anything here remind you of the
   number-line?
```



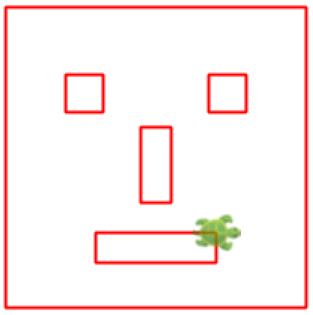
```
clear()
forward(150)
right(90)
???

hop(50)
right(90)
hop(50)
left(90)

forward(50)
right(90)
???
```



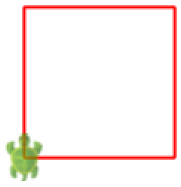
```
clear()
setSpeed(medium)
forward(100)
???
```



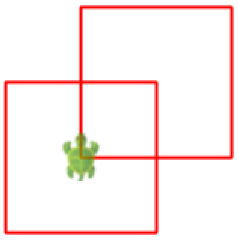
```
clear()
setSpeed(fast)
forward(200)
// pay attention to symmetry
???
```



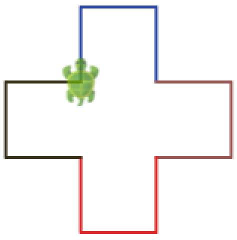
```
clear()
// Note the use of the repeat command
// How does the repeat command help?
repeat(???) {
   forward(100)
   right(90)
}
```



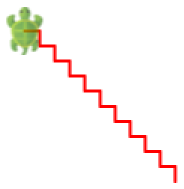
```
clear()
// Use repeat, hop, etc to make the
   figure
???
```



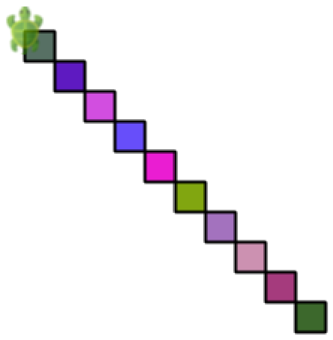
```
clear()
setSpeed(medium)
repeat(4) {
   setPenColor(randomColor)
   forward(50)
   right(90)
   ???
}
```



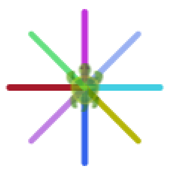
```
clear()
setSpeed(medium)
repeat(10) {
   forward(10)
   ???
}
```



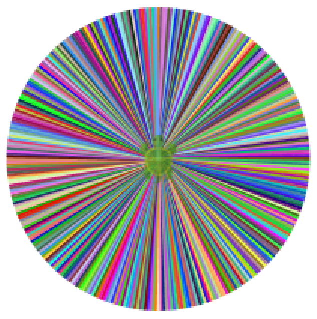
```
clear()
setSpeed(medium)
setPenColor(black)
repeat(10) {
   setFillColor(randomColor)
   repeat(4) {
     ???
   }
   ???
}
```



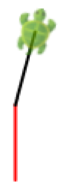
```
clear()
setSpeed(medium)
setPenThickness(4)
repeat(8) {
   setPenColor(randomColor)
   forward(50)
   hop(-50)
   right(45)
}
```



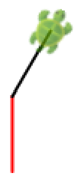
```
clear()
setSpeed(fast)
repeat(???) {
   setPenColor(randomColor)
   forward(100)
   hop(-100)
   right(1)
}
```



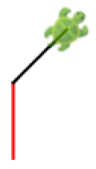
```
clear()
forward(50)
right(15)
setPenColor(black)
forward(50)
```



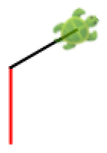
```
clear()
forward(50)
right(30)
setPenColor(black)
forward(50)
```



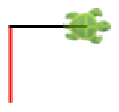
```
clear()
forward(50)
right(45)
setPenColor(black)
forward(50)
```



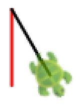
```
clear()
forward(50)
right(60)
setPenColor(black)
forward(50)
```



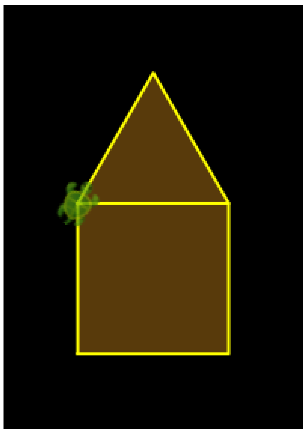
```
clear()
forward(50)
right(90)
setPenColor(black)
forward(50)
```



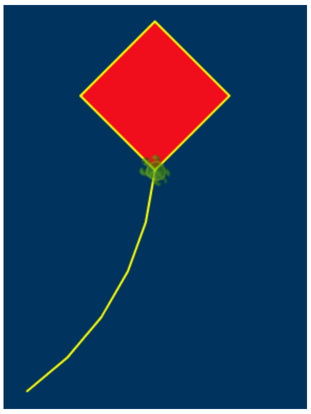
```
clear()
forward(50)
right(90 + 60)
setPenColor(black)
forward(50)
```



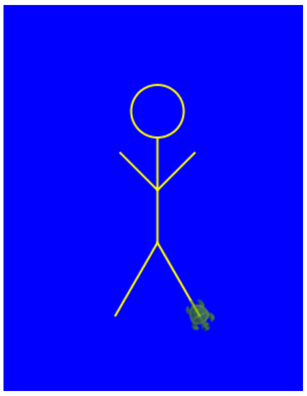
```
clear()
setSpeed(medium)
setBackground(black)
setPenColor(yellow)
setFillColor(brown)
repeat(4) {
   forward(100)
   right(???)
}
hop(100)
right(???)
repeat(3) {
   forward(100)
   right(???)
}
```



```
clear()
setSpeed(medium)
setBackground(blue)
setPenColor(yellow)
right(50)
repeat(5) {
   forward(50)
   left(10)
}
setFillColor(red)
left(???)
repeat(4) {
   forward(???)
   right(90)
}
```



```
clear()
setSpeed(medium)
setBackground(blue)
setPenColor(yellow)
right(90)
circle(25)
right(90)
forward(50)
right(90 + 45)
forward(50)
hop(-50)
right(90)
forward(50)
???
```



Free Thinking Time

You have done a fair bit of guided activity. Now it's time to make a few drawings of your own! Feel free to first sketch your idea on paper and then draw it in Kojo.

Here are some ideas to get you going:

- Flags of a few different countries.
- Patterns made out of geometric shapes like squares, rectangles, triangles, etc.
- Drawings based on real-world objects like houses, buildings, cars, ships, tanks, etc.
- Anything else that you can dream up!

After you are done, move on to the next page of the book...

```
// Within Kojo scripts/programs, lines in green that begin with a
   double slash (like this one) are ignored by Kojo and are meant
   for you, the reader.
// Such lines are program 'comments'; comments are written in
   programs to explain things to humans.
```

```
// You can use Kojo as a (super) calculator. Let's begin with
   some simple calculations.
// Kojo supports two types of numbers - Int (integers - positive
   and negative whole numbers) and Double (decimal fractions).
// Type in the following lines, and then run your script using
   Shift+Enter to see the result values as shown below:

2 + 3 //> res18: Int = 5
9 - 5 //> res19: Int = 4
6 * 7 //> res20: Int = 42
7 / 5 //> res21: Int = 1
7.0 / 5 //> res22: Double = 1.4

// You can use brackets to group calculations as you want; how
   does this relate to BODMAS?
(9 * (4 + 3)) - (2 * 4) //> res23: Int = 55
```

```
// The print and println commands let you write out things to the
   Kojo Output Pane.
// The println command does a print, and then puts a new line in
   the output.
// repeatFor is very similar to repeat, except that it gives you
   a repetition counter that can be used to do something slightly
   different in your repeated code.
// Using these commands, and Kojo's calculating ability, you can
   easily generate multiplication tables:

clearOutput()
// table of six; you can change this to whatever you want.
val t = 6
print("Table of "); println(t)
println("-----------")
repeatFor(1 to 10) { n =>
   val ans = t * n
   print(t); print(" x "); print(n); print(" = "); println(ans)
}

// Some additional things to know:
// the val instruction allows you to give a name to a value.
// you can write multiple commands on a line separated by a
   semi-colon (;).
// Things written withing double quotes - " " - are strings.
   Strings are meant for input and output from programs.
```

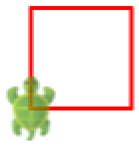
```
Table of 6
-----------
6 x 1 = 6
6 x 2 = 12
6 x 3 = 18
6 x 4 = 24
6 x 5 = 30
6 x 6 = 36
6 x 7 = 42
6 x 8 = 48
6 x 9 = 54
6 x 10 = 60
```

```
// Kojo has a couple of useful functions for calculating the HCF
   and the LCM of numbers:

kmath.hcf(10, 15) //> res14: Int = 5
kmath.hcf(12, 27) //> res15: Int = 3
kmath.lcm(10, 15) //> res16: Int = 30
kmath.lcm(12, 27) //> res17: Int = 108
```

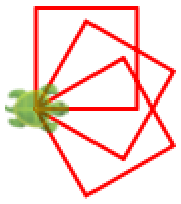
```
// You can do calculations with fractions (or Rational numbers)
   in Kojo by using the .r syntax:

2/3.r + 4/5.r //> res24: builtins.Rational = 22/15
```

Now let’s move back to programming with the Turtle…

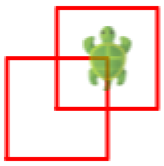
```
// You can teach Kojo new commands using
   the def instruction

def square() {
   repeat(4) {
      forward(50)
      right(90)
   }
}
clear()
square()
```



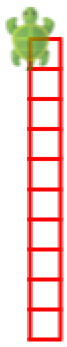
```
def square() {
   // same as before
}
clear()
setSpeed(medium)
repeat(3) {
   square()
   right(30)
}
```



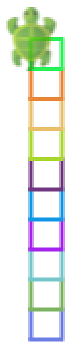
```
def square() { /* same as before */ }
clear()
setSpeed(medium)
repeat(2) {
   square()
   hop(25)
   right(90)
   hop(25)
   left(90)
}
```



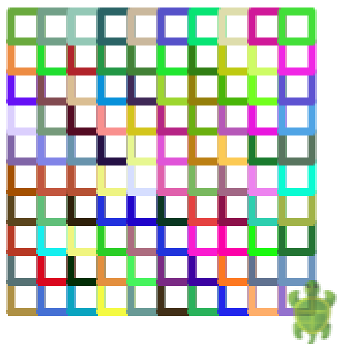
```
def square() {
   // similar to before; size 15
}
clear()
setSpeed(medium)
repeat(10) {
   ???
}
```



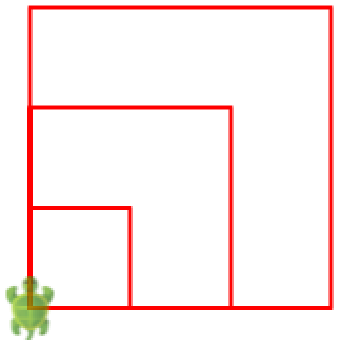
```
def square() { /* same as before */ }
def ladder() {
   repeat(10) {
      setPenColor(randomColor)
      ???
   }
}
clear()
setSpeed(medium)
ladder()
```



```
def square() { /* same as before */ }
def ladder() { /* same as before */ }
clear()
setSpeed(medium)
setPenThickness(4)
repeat(10) {
   ladder()
   ???
}
```

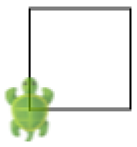


```
// New commands that you teach Kojo can
    also take inputs

def square(n: Int) {
   repeat(4) {
      forward(n)
      right(90)
   }
}
clear()
setSpeed(medium)
square(50)
square(100)
square(150)
```

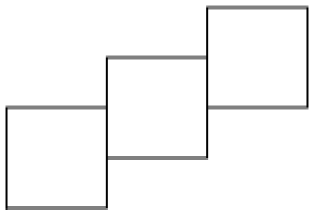


Next, let's work with patterns in a systematic way...

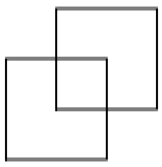
```
// The basic shape for the next few
   patterns:
def square() {
   repeat(4) {
      forward(50)
      right()
   }
}
clear()
setSpeed(medium); setPenColor(black)
square()
```



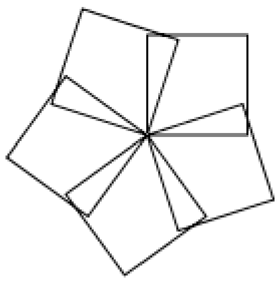
```
def square() { /* same as before */ }

// The building block of the pattern.
   Make the shape, then position the
   turtle to make the next shape in the
   pattern:
def block() {
   square()
   right(90)
   hop(50)
   left(90)
   hop(25)
}
cleari()
setSpeed(medium); setPenColor(black)
// Just repeat the block to make the
   pattern:
repeat(3) {
   block()
}
```



```
def square() { /* same as before */ }

// The building block of the pattern:
def block() {
   square()
   ???
}
cleari()
setSpeed(medium); setPenColor(black)
// Just repeat the block to make the
   pattern:
repeat(2) {
   block()
}
```



```
def square() { /* same as before */ }

// The building block of the pattern:
def block() {
   square()
   ???
}
cleari()
setSpeed(medium); setPenColor(black)
// Just repeat the block to make the
    pattern:
repeat(5) {
   block()
}
```



Each program that you write to make a pattern should contain:

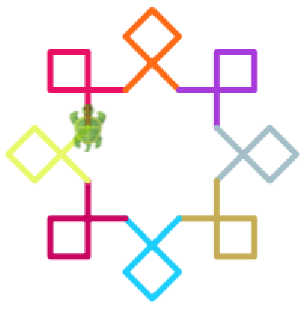
- the definition (via a `def`) of the fundamental **shape** inside the pattern. As you define this shape, make sure that you think about the starting position and direction (or heading) of the turtle within this shape.
- the definition (via a `def`) of the pattern building-**block**. This includes the fundamental shape, and some commands to set the position and direction of the turtle to make the next building block in the pattern. This position and direction relates to the starting position and direction of the fundamental shape.
- A **repeat** command that makes the pattern building-**block** a **certain number of times** to create the full pattern.

Here's a template for the kind of code you should write to make these pattern figures:

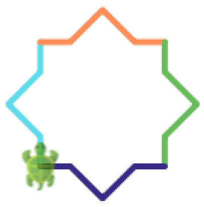
```
// define the fundamental shape present inside the pattern
def shape() {
}

// define the building block of the pattern,
// based on the fundamental shape
def block() {
   shape()
   // position the turtle for the next block
}

cleari()
setSpeed(medium)
repeat(8) {
   block()
}
```

Using the above ideas, write programs to make the following patterns shown on this page. Within each pattern, every building block has a different color – to help you to easily identify it:

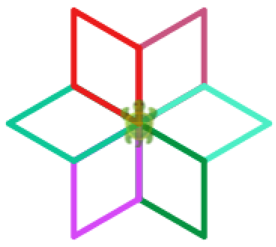
```
def shape() {
    ???
}
def block() {
    shape()
    ???
}
cleari(); setSpeed(medium)
repeat(8) {
    block()
}
```



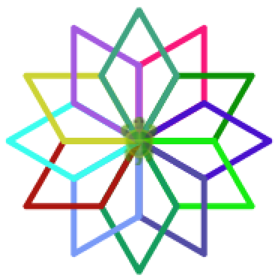
```
def shape() {
    ???
}
def block() {
    shape()
    ???
}
cleari(); setSpeed(medium)
repeat(8) {
    block()
}
```



```
clear()
setSpeed(medium)
setPenThickness(4)
repeat(???) {
    setPenColor(randomColor)
    ???
    right(???)
}
```



```
def shape() {
    ???
}
def block() {
    shape()
    ???
}
cleari(); setSpeed(medium)
repeat(8) {
    block()
}
```



Extended Exercise – Hands on with Forty figures

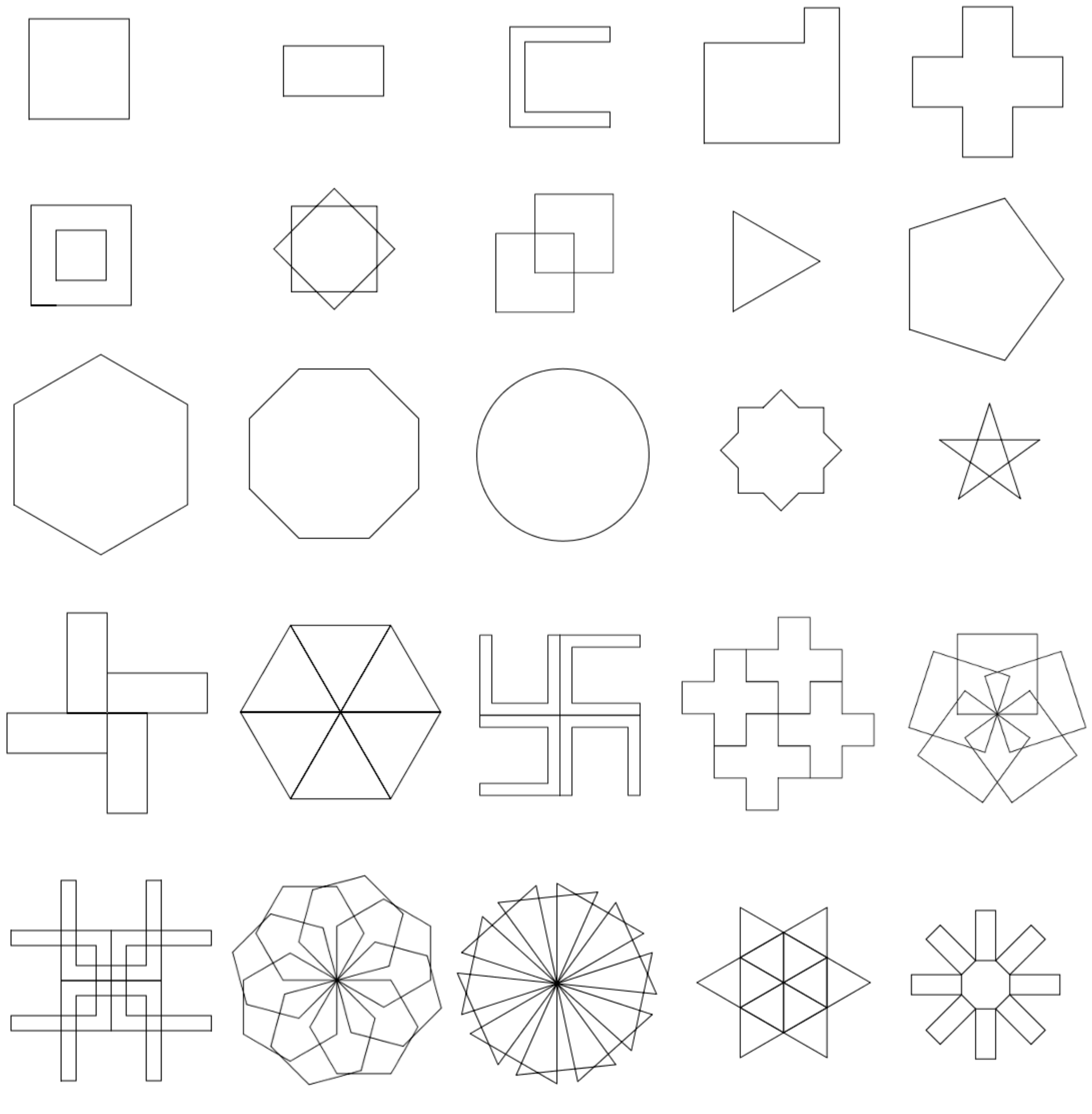
On the next two pages you will see forty different figures (these are Kojo drawings of the figures presented in a wonderful book by Barry Newell called Turtle Confusion). Some of these figures are building blocks for patterns, while others are the patterns themselves. Use the ideas learned while working through the previous pages to make the patterns. As mentioned earlier, each program that you write to make a pattern should contain:

- the definition (via a `def`) of the fundamental **shape** inside the pattern. As you define this shape, make sure that you think about the starting position and direction (or heading) of the turtle within this shape.
- the definition (via a `def`) of the pattern building-**block**. This includes the fundamental shape, and some commands to set the position and direction of the turtle to make the next building block in the pattern. This position and direction relates to the starting position and direction of the fundamental shape.
- A **repeat** command that makes the pattern building-**block** a **certain number of times** to create the full pattern.

Here again is a template (with one additional optional step compared to the previous template) for the kind of code you should write to make the figures:

```
// define the fundamental shape present inside the pattern
def shape() {
}

// define the building block of the pattern,
// based on the fundamental shape
def block() {
   shape()
   // position the turtle for the next block
}

cleari()
setSpeed(medium)
// this optional step is new:
// change the turtle's direction if needed
// and then repeat the block to make the pattern
repeat(8) {
   block()
}
```

Figures 1 to 25



Figures 26 to 40