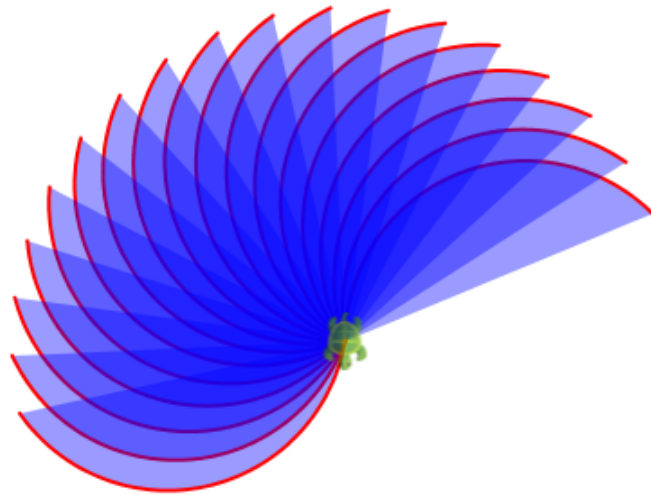


Introduction to Programming

with Kojo



by
Lalit Pant

Version: August 6, 2018



License: Creative Commons *Attribution-NonCommercial-ShareAlike 4.0 International*
CC BY-NC-SA 4.0

Author: Lalit Pant

This book uses ideas from: *Challenges with Kojo*, by Björn Regnell

© 2010–2018 Lalit Pant (lalit@kogics.net) <http://www.kogics.net>

© 2015 Björn Regnell, Lund University <http://lth.se/programmera>

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



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



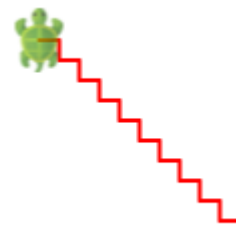
```
clear()
forward(50)
right(90)
forward(50)
left(90)
forward(50)
```



```
clear()
setAnimationDelay(100)
repeat(4) {
  forward(50)
  right(90)
  ???
}
```



```
clear()
setAnimationDelay(100)
repeat(10) {
  forward(10)
  ???
}
```





```
// run this program using the 'Trace Script' button (shown above) in the toolbar  
clear()  
forward(100)  
right(90)  
forward(100)
```

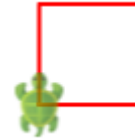
// The trace of the program is shown below. Can you see how tracing a program can help you understand exactly what the program does, step by step?

The image displays three sequential screenshots of a turtle graphics environment, illustrating the execution of a program step by step. Each screenshot consists of three panels: a 'Program Trace' window on the left, a 'Script Editor' window in the middle, and a '3D Canvas' window on the right.

- Top Screenshot:** The 'Script Editor' shows the first line of code, `clear()`, highlighted. The 'Program Trace' window shows a list of calls: `CALL clear ()`, `CALL forward (n = 100.0)`, `CALL right (angle = 90.0)`, and `CALL forward (n = 50.0)`. The '3D Canvas' shows a green turtle at the top right, with a red line extending to the left and a yellow line extending downwards.
- Middle Screenshot:** The 'Script Editor' shows the second line of code, `forward(100)`, highlighted. The 'Program Trace' window shows the same list of calls, but `CALL right (angle = 90.0)` is now highlighted. The '3D Canvas' shows the turtle moving down, with a red line extending to the left and a yellow line extending downwards.
- Bottom Screenshot:** The 'Script Editor' shows the third line of code, `right(90)`, highlighted. The 'Program Trace' window shows the same list of calls, but `CALL forward (n = 50.0)` is now highlighted. The '3D Canvas' shows the turtle moving right, with a red line extending to the left and a yellow line extending downwards.

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

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

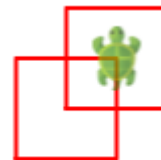


```
// 'call' the new command to use it.
square()
```

```
def square() {
  // same as before
}
clear()
setAnimationDelay(100)
repeat(3) {
  square()
  right(30)
}
```



```
def square() { /* same as before */ }
clear()
setAnimationDelay(100)
repeat(2) {
  square()
  hop(25)
  right(90)
  hop(25)
  left(90)
}
```



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



```

def square() { /* same as before */ }
def ladder() {
  setPenColor(randomColor)
  ???
}
clear()
setAnimationDelay(10)
ladder()

```



```

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

```

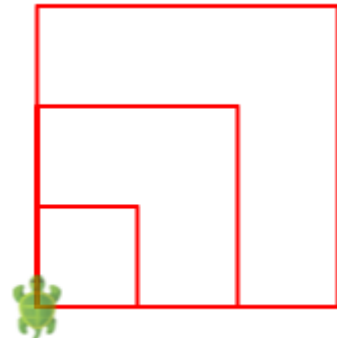


// New commands that you teach Kojo can also take inputs

```

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

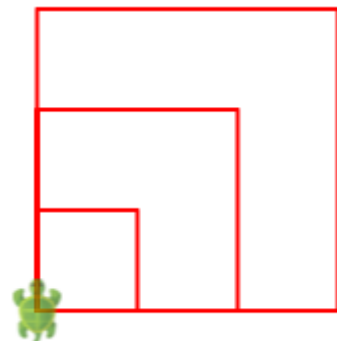
```



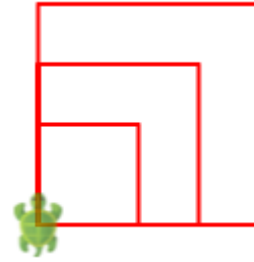
```

def square(n: Int) {
  // same as before
}
clear()
setAnimationDelay(100)
repeatFor(1 to 3) { n =>
  square(n * 50)
}

```



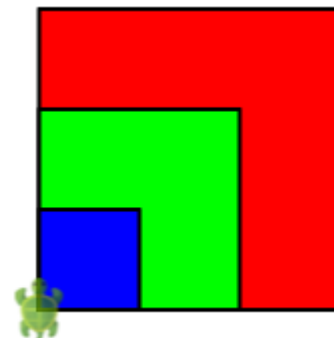
```
def square(n: Int) { /* same as before
  */ }
clear()
setAnimationDelay(100)
// make squares of sizes 50, 80, and 110
repeatFor(1 to 3) { n =>
  ???
}
```



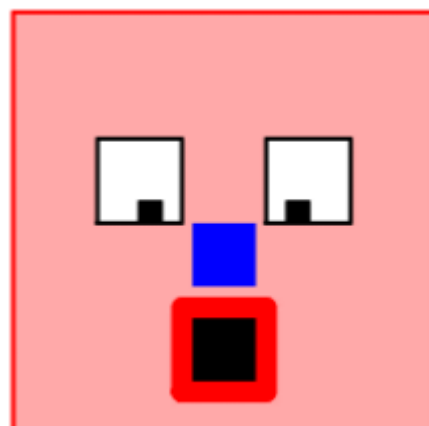
```
def square(n: Int) { /* same as before
  */ }
clear()
setAnimationDelay(100)
setPenThickness(20)
setBackground(yellow)
setPenColor(blue)
repeatFor(1 to 3) { n =>
  square(10 + n * 40)
}
```



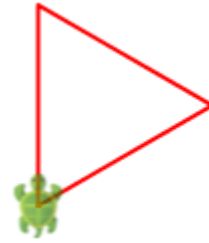
```
def square(n: Int) { /* same as before
  */ }
// A Seq lets you organize/structure
  your data in a sequence
val sizes = Seq(150, 100, 50)
val colors = Seq(red, green, blue)
clear()
setAnimationDelay(100)
setPenColor(black)
repeatFor(0 to 2) { n =>
  // You can access elements in a sequence
  via a 0-based index: seq(idx)
  setFillColor(colors(n))
  square(sizes(n))
}
```



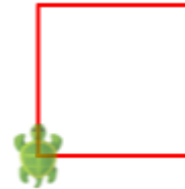
```
def square(n: Int) { /* same as before
  */ }
clear()
setAnimationDelay(100)
???
```



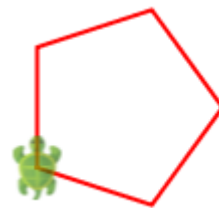
```
clear()
repeat(3) {
  forward(100)
  right(120)
}
```



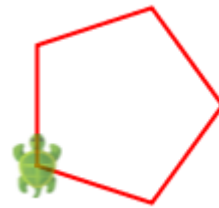
```
clear()
repeat(4) {
  forward(75)
  right(90)
}
```



```
clear()
repeat(5) {
  forward(60)
  ???
}
```



```
def polygon(sides: Int) {
  repeat(sides) {
    ???
  }
}
clear()
polygon(5)
```



```
def polygon(sides: Int) {
  // same as before
}
clear()
polygon(8)
```



```
def polygon(sides: Int) {
  // same as before
}
clear()
setAnimationDelay(100)
polygon(???)
```




```
1 Run Script as a Worksheet - to see expression values and types inline (Shift+Enter)
2
3 2 + 3 //> res1: Int = 5
```

```
1 val x = 10 //> x: Int = 10
2 val y = 5 //> y: Int = 5
3 x + y //> res10: Int = 15
4 x * y //> res11: Int = 50
5 x + y * 2 //> res12: Int = 20
6 (x + y) * 2 //> res13: Int = 30
```

```
1 var x = 10 //> x: Int = 10
2 x = x + 1 //> x: Int = 11
3 x += 1
4 val y = 10 //> y: Int = 10
5 y = y + 1 //> error: reassignment to val
```

```
1 var sum = 0
2 var i = 1
3 while (i < 5) {
4     sum += i
5     i += 1
6 }
7 clearOutput()
8 println(sum)
```

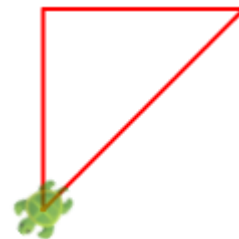
Output Pane: 10

// The val instruction lets you give a name to a value. This name can be used multiple times in the rest of the program.
// The var instruction lets you create a variable.
// The right hand side (after the equal sign) of both the above instructions contain an expression. An expression is any piece of code that can be evaluated to produce a (data) value.

```
Script Editor
1 def twice(n: Int) = n * 2 //> twice: (n: Int)Int
2 twice(5) //> res17: Int = 10
3 def sum(n1: Int, n2: Int) = n1 + n2 //> sum: (n1: Int, n2: Int)Int
4 sum(3, 4) //> res18: Int = 7
5 def diagonal(side1: Double, side2: Double) = {
6     val dsquare = math.pow(side1, 2) + math.pow(side2, 2)
7     math.sqrt(dsquare)
8 } //> diagonal: (side1: Double, side2: Double)Double
9 diagonal(3, 4) //> res19: Double = 5.0
10 diagonal(4, 5) //> res20: Double = 6.4031242374328485
11
```

```
// In the above code - twice, sum, and
// diagonal are functions (and not
// commands).
// So what's the difference between
// commands and functions?
// A commands lets you carry out an
// action or affect a future action.
// Actions are effects produced by your
// program that you can see, hear, etc.
// A function takes one or more input
// values and returns one or more output
// values. Functions carry out
// computations (or calculations) to
// convert inputs to outputs. A function
// call is an expression.
```

```
def diagonal(side1: Double, side2:
    Double) = {
    // same as before
}
clear()
forward(100)
right(90)
forward(100)
right(135)
forward(diagonal(100, 100))
```



```

def diagonal(side1: Double, side2:
  Double) = {
  // same as before
}
var more = "yes"
while (more == "yes") {
  clear(); clearOutput()
  val s1 = readInt("First side of
    triangle")
  val s2 = readInt("Second side of
    triangle")
  val s3 = diagonal(s1, s2)
  val angle = math.atan2(s1,
    s2).toDegrees
  println(s"""
First side is: $s1, second side is: $s2.
The length of the diagonal is: $s3.
The angle between the second side
and the diagonal is: $angle degrees
""")
  )
  forward(s1)
  right(90)
  forward(s2)
  right(180 - angle)
  forward(s3)
  more = readln("More triangles?")
}

```



```

Output Pane
First side is: 120, second side is: 60.
The length of the diagonal is: 134.16407864998737.
The angle between the second side
and the diagonal is: 63.43494882292201 degrees

```

```

// Let's explore a series of functions that will allow us to find all the primes
  below a certain number

def factor(n1: Int, n2: Int) = n2 % n1 == 0

def factors(n: Int) = {
  (2 to n/2).filter { x =>
    factor(x, n)
  }
}

factors(20) //> Vector(2, 4, 5, 10)

def prime(n: Int) = {
  factors(n).size == 0
}

prime(11) //> true

def primesTill(n: Int) = {
  (3 to n).filter { x =>
    prime(x)
  }
}

primesTill(30) //> Vector(3, 5, 7, 11, 13, 17, 19, 23, 29)

// ??? Use the above functions to answer:
// - What are the factors of 27
// - Is 29 a prime number
// - What are all the primes below 40

```

```

// You can use the test command to test your functions. If a test passes, you
  will see green output in the Output Pane for that test. If a test fails, you
  will see red output for that test, and a test FAILED message

// This test should pass
test("primes till 10") {
  primesTill(10) shouldBe Vector(3, 5, 7)
}

// This test should fail
test("primes till 15") {
  primesTill(15) shouldBe Vector(3, 5, 7, 11, 12, 13)
}

// Writing tests to ensure the correctness your functions becomes very important
  as you write bigger pieces of software

```

```
// Objects combine data, and functions that act on that data. You can define your
  // own objects in Kojo using classes.

// First define the structure of your object (via its fields) and it's functions
  // (called methods) using a case class
case class Fraction(num: Int, den: Int) {
  require(den != 0)
  def +(other: Fraction) =
    Fraction(num * other.den + other.num * den,
      den * other.den)
}

// Then create object instances of your class and use them
Fraction(1, 2) + Fraction(1, 2) //> Fraction(4,4)
```

```
// euler problem 2:
// Each new term in the Fibonacci sequence is generated by adding the previous
  // two terms. By starting with 1 and 2, the first 10 terms will be:
// 1, 2, 3, 5, 8, 13, 21, 34, 55, 89, ...
// By considering the terms in the Fibonacci sequence whose values do not exceed
  // four million, find the sum of the even-valued terms.

// A map allows you to store keys and corresponding values
// An lru cache is a map which stores recently used data

// Here we use a cache to store values of the 'fib' function, to avoid
  // recalculation of values
val cache = lruCache[Long, Long](10)

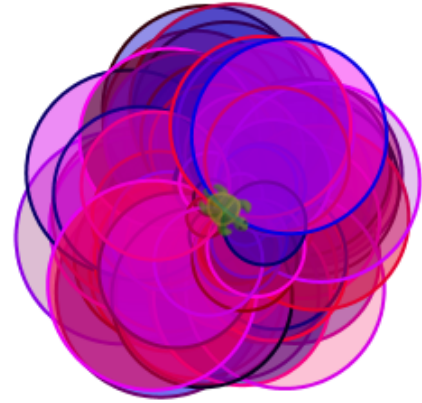
def fib(n: Long): Long = {
  // If required value is in the cache, return it. Else compute it, put it in
  // the cache, and return it.
  cache.getOrElseUpdate(n,
    n match {
      case 1 => 1
      case 2 => 2
      case _ => fib(n - 1) + fib(n - 2)
    })
}

// Streams let you work with lazily generated and conceptually infinitely long
  // data.
val s = Stream.from(1).map(n => fib(n)) //> Stream(1, ?)
s.takeWhile(n => n < 4000000).filter(n => n % 2 == 0).sum //> 4613732
```

```

clear()
setAnimationDelay(10)
repeat(100) {
  setPenColor(Color(random(256), 0,
    random(256)))
  setFillColor(Color(random(256), 0,
    random(256), random(100) + 50))
  left(random(360))
  circle(random(50) + 10)
}

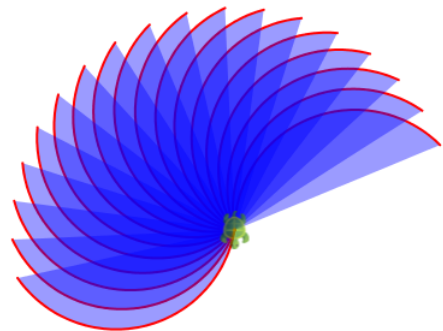
```



```

clear()
setAnimationDelay(10)
setFillColor(Color(0, 0, 255, 100))
repeat(18) {
  savePosHe()
  right(135, 100)
  restorePosHe()
  left(10)
}

```



```

def flower(size: Int) {
  savePosHe()
  ???
  repeat(100) {
    ???
  }
  restorePosHe()
}
clear()
setAnimationDelay(10)
flower(20)

```

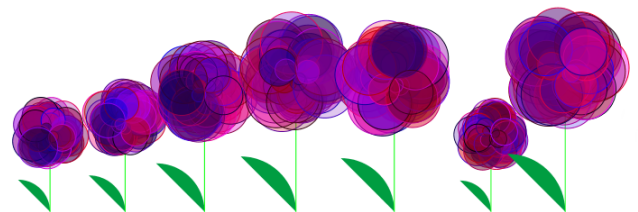


```

def flower(size: Int) {
  // same as before
}

def garden(flowers: Int) {
  repeat(flowers) {
    ???
  }
}
clear()
setAnimationDelay(10)
garden(7)

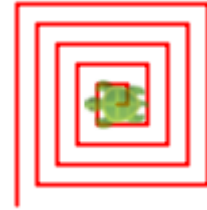
```



```

def figure(n: Int) {
  if (n < 10) {
    forward(n)
  }
  else {
    forward(n)
    right(90)
    figure(n - 5)
  }
}
clear()
figure(100)
// use tracing to understand this program

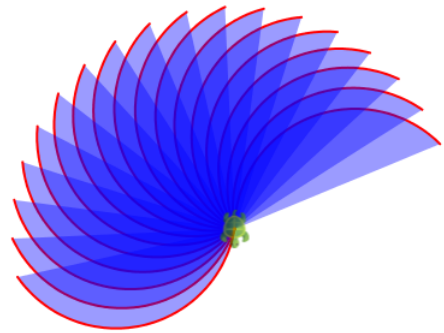
```



```

def pattern(n: Int) {
  if (n > 0) {
    savePosHe()
    right(135, 100)
    restorePosHe()
    left(10)
    pattern(n - 1)
  }
}
clear()
setAnimationDelay(10)
setFillColor(Color(0, 0, 255, 100))
pattern(18)

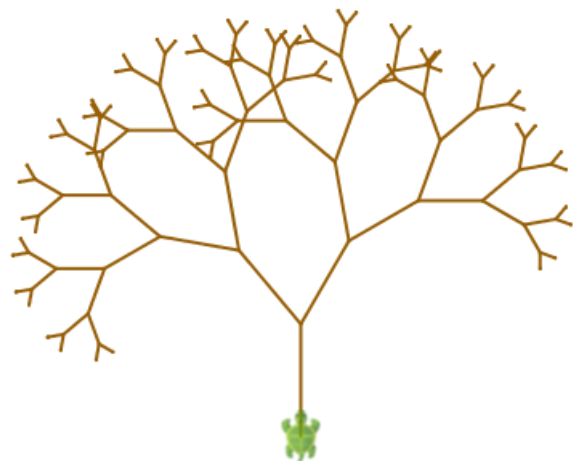
```



```

def tree(n: Int) {
  savePosHe()
  if (n < 10) {
    ???
  }
  else {
    forward(n)
    right(30)
    tree(n - 10)
    left(70)
    tree(n - 10)
  }
  restorePosHe()
}
clear()
setAnimationDelay(10)
setPenColor(Color(150, 95, 8))
tree(70)

```



```
// A traffic light animation
```

```
def light = Picture {
    setPenColor(darkGray)
    circle(20)
}

cleari()

val r = light
val y = light
val g = light
y.translate(0, 40)
r.translate(0, 80)

draw(r, y, g)

r.setFillColor(red)
var lightOn = r

timer(1000){
    if (lightOn == r) {
        r.setFillColor(white)
        y.setFillColor(yellow)
        lightOn = y
    }
    else if (lightOn == y) {
        y.setFillColor(white)
        g.setFillColor(green)
        lightOn = g
    }
    else if (lightOn == g) {
        g.setFillColor(white)
        r.setFillColor(red)
        lightOn = r
    }
}
```



```
// ??? Make the lights blink faster. And
    then slower.
// ??? Put a rectangle around the lights.
```



```
// A simple game. You need to keep the
// rectangle within the canvas. The
// rectangle moves and grows in size.
// Its speed goes up as its size
// increases. You can rotate it by
// pressing the 'P' key. You can make it
// smaller by clicking on it
```

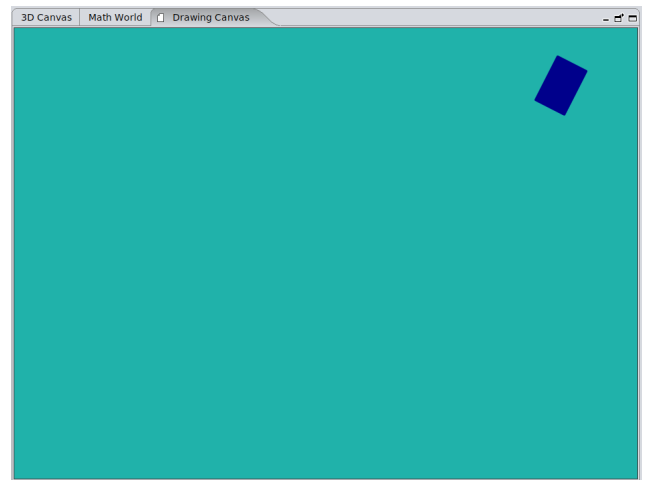
```
switchToDefault2Perspective()
clear()
drawStage(ColorMaker.lightSeaGreen)

val p1 = Picture {
    setPenColor(ColorMaker.darkBlue)
    setFillColor(ColorMaker.darkBlue)
    repeat(2) {
        forward(40)
        right(90)
        forward(60)
        right(90)
    }
}
draw(p1)

animate {
    p1.translate(2, 0)
    p1.scale(1.001)
    if (isKeyPressed(Kc.VK_P)) {
        p1.rotate(1)
    }
    if (p1.collidesWith(stageBorder)) {
        p1.setFillColor(red)
        stopAnimation()
    }
}

p1.onMouseClicked { (x, y) =>
    p1.scale(0.9)
}

activateCanvas()
```



```
// ??? How can you make the game more
// difficult?
// Increase the speed of the rectangle
// at a faster rate?
// Make the rectangle grow bigger faster?
// Try these ideas (and any others that
// you come up with).
```

```
// Another game. The blue square (controlled by
    you) is hunted by the red squares.
```

```
switchToDefault2Perspective()
```

```
clear()
```

```
drawStage(yellow)
```

```
val cb = canvasBounds
```

```
def gameShape(color: Color) = Picture {
    setFillColor(color)
    setPenColor(color)
    repeat(4) {
        forward(40)
        right(90)
    }
}
```

```
val r1 = gameShape(red)
```

```
val r2 = gameShape(red)
```

```
val r3 = gameShape(red)
```

```
val r4 = gameShape(red)
```

```
val player = gameShape(blue)
```

```
r1.setPosition(150, 150)
```

```
r2.setPosition(-150, 150)
```

```
r3.setPosition(0, 150)
```

```
r4.setPosition(250,0 )
```

```
draw(r1, r2, r3,r4, player)
```

```
val playerspeed = 9
```

```
var vel1 = Vector2D(3, 2) * 2
```

```
var vel2 = Vector2D(-3, 2) * 2
```

```
var vel3 = Vector2D(0, 4) * 2
```

```
var vel4 = Vector2D( 4,0) * 2
```

```
animate {
```

```
    r1.transv(vel1)
```

```
    r2.transv(vel2)
```

```
    r3.transv(vel3)
```

```
    r4.transv(vel4)
```

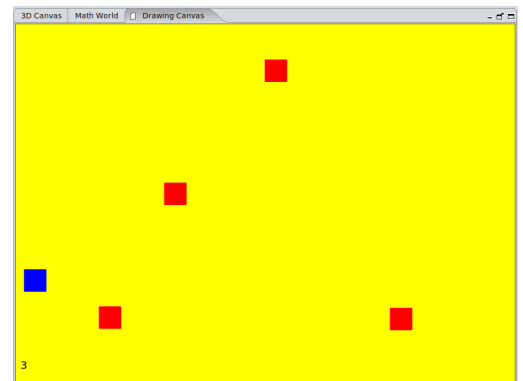
```
    // r1, r2, r3, r4 motion
```

```
    if (r1.collidesWith(stageBorder)) {
        vel1 = bouncePicVectorOffStage(r1, vel1)
    }
```

```
    if (r2.collidesWith(stageBorder)) {
        vel2 = bouncePicVectorOffStage(r2, vel2)
    }
```

```
    if (r3.collidesWith(stageBorder)) {
        vel3 = bouncePicVectorOffStage(r3, vel3)
    }
```

```
}
```



```

if (r4.collidesWith(stageBorder)) {
    vel4 = bouncePicVectorOffStage(r4, vel4)
}

// player keyboard control
if (isKeyPressed(Kc.VK_UP)) {
    player.translate(0, playerspeed)
}
if (isKeyPressed(Kc.VK_DOWN)) {
    player.translate(0, -playerspeed)
}
if (isKeyPressed(Kc.VK_LEFT)) {
    player.translate(-playerspeed, 0)
}
if (isKeyPressed(Kc.VK_RIGHT)) {
    player.translate(playerspeed, 0)
}

// player-r1, r2, r3, r4 collision
if (player.collidesWith(r1)) {
    gameLost()
}
if (player.collidesWith(r2)) {
    gameLost()
}
if (player.collidesWith(r3)) {
    gameLost()
}
if (player.collidesWith(r4)) {
    gameLost()
}

// player-border collision
if (player.collidesWith(stageBorder)) {
    gameLost()
}
}

def gameLost() {
    drawCenteredMessage("You Loose", purple, 20)
    stopAnimation()
    player.setFillColor(red)
    player.scale(1.1)
}

showGameTime(60, "You loose", black)
activateCanvas()

```

```

// ??? Make the game look better by changing the shapes of the hunters and the
// hunted.
// For example, the hunters could be stars, and the hunted could be a pentagon.

```

```

// Here's a slightly better organized version of the previous game. It uses a
// sequence and a map to remove code duplication.

switchToDefault2Perspective()
clear()
drawStage(yellow)
val cb = canvasBounds

def gameShape(color: Color) = Picture {
  setFillColor(color)
  setPenColor(color)
  repeat(4) {
    forward(40)
    right(90)
  }
}

val r1 = gameShape(red)
val r2 = gameShape(red)
val r3 = gameShape(red)
val r4 = gameShape(red)
r1.setPosition(150, 150)
r2.setPosition(-150, 150)
r3.setPosition(0, 150)
r4.setPosition(250, 0)

val player = gameShape(blue)

draw(r1, r2, r3, r4, player)

val playerspeed = 9
var vel1 = Vector2D(3, 2) * 2
var vel2 = Vector2D(-3, 2) * 2
var vel3 = Vector2D(0, 4) * 2
var vel4 = Vector2D(4, 0) * 2

val rs = Seq(r1, r2, r3, r4)
var rsVels = Map(
  r1 -> vel1,
  r2 -> vel2,
  r3 -> vel3,
  r4 -> vel4
)

animate {
  rs.foreach { r =>
    r.translate(rsVels(r))
  }
}

```

```

rs.foreach { r =>
  if (r.collidesWith(stageBorder)) {
    val newVel = bouncePicVectorOffStage(r, rsVels(r))
    rsVels += (r -> newVel)
  }
}

rs.foreach { r =>
  if (player.collidesWith(r)) {
    gameLost()
  }
}

// player keyboard control
if (isKeyPressed(Kc.VK_UP)) {
  player.translate(0, playerspeed)
}

if (isKeyPressed(Kc.VK_DOWN)) {
  player.translate(0, -playerspeed)
}

if (isKeyPressed(Kc.VK_LEFT)) {
  player.translate(-playerspeed, 0)
}

if (isKeyPressed(Kc.VK_RIGHT)) {
  player.translate(playerspeed, 0)
}

// player-border collision
if (player.collidesWith(stageBorder)) {
  gameLost()
}
}

def gameLost() {
  drawCenteredMessage("You Loose", purple, 20)
  stopAnimation()
  player.setFillColor(purple)
  player.scale(1.1)
}

showGameTime(60, "You loose", black)
activateCanvas()

```

```

// ??? Add a fifth hunter to the game. Make use of the rs sequence and rsVels map
so that you don't have to add any code to the 'animate' loop.

```