Kojo Programming Quick-Ref

Lalit Pant

Kojo Programming Quick-Ref

Lalit Pant

February 14, 2014

Contents

1 Introduction

This book is meant for a couple of different audiences:

- Children who have been using Kojo for a while, and want a concise reference to the ideas, terminology, and useful features of Kojo.
- Adults who are familiar with programming, and want a quick way to get productive with Kojo.

This book has many code listings. At the bottom of each listing, you will find a couple of links – one to "Run code online" for that listing, and the other one to "View code online". You are encouraged to run the code for the listings to get the most out of the book. To do this effectively, start up Kojo-Web as you start reading the book. From then on, every time you click on a "Run code online" link for a listing, Kojo will quickly pop-up to the front, load the code for the listing, and run it. You can then study the code and play with it inside Kojo. If you are unable to run Kojo-Web for any reason, you can start up Kojo-Desktop, and make use of the "View code online" links to easily copy and paste the code for the listings into Kojo. If you are not online you can, of course, copy and paste the code directly from the book into Kojo.

1.1 Program building blocks

A program within Kojo is a series of instructions for Kojo to carry out. These instructions can be:

Commands, which let you take actions (like moving the turtle forward) or indirectly affect 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.

Listing 1.1: Using commands

```
clear()
setPenColor(blue) // affects a future action
forward(100) // carries out an action
```

Run code online **View code online**

Expressions, which let you do calculations or computations. Expressions are evaluated to compute data values. Expressions are made out of:

Literals, which evaluate to themselves, e.g., `5, 7.1, List(1, 2, 3), ``road''.`

Functions, which covert (or map) inputs to outputs – a function takes some values as inputs and computes or returns an output value based on the inputs, e.g., `sum(5, 7)`, and `multiply(2, 9)`. Note that a command can also take inputs, but (unlike a function) it does not return an output value that can be used within your program. Instead, a command results in an action or an indirect affect on future actions – it has a *side-effect*.
You *invoke* or *call* commands and functions to make use of them.

Operators, which are functions with names made up of special characters like `+`, `*`, and `!`, e.g., `5 + 7`, `2 * 9`.

Listing 1.2: Using expressions

```
// #worksheet -- run as a worksheet
5
7.1
"road"
List(1, 2, 3)

5 + 7
2 * 9

math.max(5, 10)
math.abs(-30)
```

Run code online **View code online**

Keyword-instructions, which help you to structure your program, e.g., by letting you combine your commands, functions, and data values into higher level building blocks that you can use in your programs (more on this in the next section). A keyword instruction ultimately behaves like a command or an expression.

Listing 1.3: Using keywords

```
// #worksheet
// use def keyword to define new functions
def sum(n1: Int, n2: Int) = n1 + n2
def multiply(n1: Int, n2: Int) = n1 * n2

// use case class keyword to define a new class
case class Rational(num: Int, den: Int) {
    require(den != 0)
    def +(other: Rational) =
        Rational(num * other.den + other.num * den,
            den * other.den)
}

// use the newly defined stuff
sum(5, 7)
```

```
multiply(2, 9)
// create instances of the new class
// and add them using the '+' function from the class
Rational(2, 3) + Rational(4, 5)
```

Run code online **View code online**

Impure-functions and queries, which are instructions that do not fit into the command or expression mold:

Impure-functions are instructions that have a side-effect (like a command) and also return a value (like a function), e.g., an instruction that draws a shape and also returns it. You should avoid impure functions if you can.

Queries are functions that can return different values on different calls, e.g., `turtle.position`.

Listing 1.4: Impure functions and queries

```
// #worksheet
// define an impure function that carries out an action
// and also returns a value
def impure(h: Int, w: Int) = {
    val pic = PicShape.rect(h, w)
    draw(pic)
    pic
}

cleari()
// use the impure function
val pic = impure(40, 60)

// position is a query that returns
// the turtle's current position
position
forward(100)
position
```

Run code online **View code online**

1.2 Fundamental Ideas in Programming

It's helpful to think about programming in Kojo in terms of the following fundamental ideas:

Primitives, which are the building-blocks (commands and functions) already provided by the environment. For example:

Commands – `forward`, `right`, `setPenColor`, etc.

Functions – `Picture`, `trans`, `rot`, `scale`, etc.

Composition, which allows you to combine the available building-blocks as required.

Listing 1.5: Examples of composition

```
// a sequence of commands
forward(100)
right()
forward(50)

// command looping
repeat(4) {
    forward(10)
}

// nesting of functions within commands
forward(math.max(5, 10))

// function nesting
math.abs(math.min(20, -30))

val pic = PicShape.rect(40, 60)
// function chaining
rot(60) * trans(10, 15) * scale(1.5) -> pic

// higher order functions
(1 to 10).filter { n => n % 2 == 0 }
```

Run code online **View code online**

Abstraction, which allows you to give names to your compositions, so that they become available as bigger building-blocks (which can take part in further composition). Hiding of internal (implementation) details is an important aspect of abstraction.

Listing 1.6: Examples of abstration

```
// create abstractions sum, multiply and Rational
// using appropriate keywords
def sum(n1: Int, n2: Int) = n1 + n2
def multiply(n1: Int, n2: Int) = n1 * n2
case class Rational(num: Int, den: Int) {
    require(den != 0)
    def +(other: Rational) =
        Rational(num * other.den + other.num * den,
            den * other.den)
}

// use the newly defined abstractions
sum(5, 7)
```

```
multiply(2, 9)
Rational(2, 3) + Rational(4, 5)
```

Run code online View code online

Selection, which allows you to choose from a series of building blocks during the process of composition.

Listing 1.7: Examples of selection

```
val size = 40
val thershold = 50
// make big shapes red
if (size > thershold) {
    setFillColor(red)
}
else {
    setFillColor(green)
}

// make small shapes move faster
val speed = if (size < thershold) 100 else 60
```

Run code online View code online

1.3 Types, Objects, and Classes

Every data value in Kojo has a (static) type associated with it. This type tells Kojo what commands and functions that value can work with. How is that? For every command, Kojo knows the types of the inputs to the command; for every function, Kojo knows the types of the inputs to the function and the type of the return value of the function. So it's straightforward for Kojo to figure out what data values are compatible with the inputs to a command or function, and what data values are compatible with output of a function. So, for example, if you try to call `setPenColor(10)` or `setPenThickness(blue)`, Kojo will catch your mistake right away and tell you about it. Or, if you are defining a function that returns an `Int`, and you actually return the value `blue` from it, Kojo will again step in to point out your mistake.

Listing 1.8: Types in commands and functions

```
// a command with one input value named n - of type Int
// note that the type of a value is written 'value: type'
def square(n: Int) {
    repeat(4) {
        forward(100)
        right()
    }
}
```

```
// a function with two input values named n1 and n2
// both n1 and n2 are of type Int
// the function returns a value of type Int
def sum(n1: Int, n2: Int): Int = n1 + n2

// most of the time the return type of the function can
// be inferred, so you don't need to explicitly provide it
def sum2(n1: Int, n2: Int) = n1 + n2

clear()
square(sum(40, 60))
```

Run code online **View code online**

Every data value in Kojo is an object. What's an object? It's a data value with commands and functions attached to it. These commands and functions are called methods. You call a method on an object like this – `object.method(inputs)`. So, if you have a picture called `pic`, you can call the translate method on it like this – `pic.translate(10, 20)`. The methods available on an object are determined by the type of the object; a method on an object works within the context of that object.

Kojo has a lot of predefined types (some of these are described in Section 2.3 on page 16). You can create new types using the class keyword (as you saw in Listing 1.6 on page 6).

2 Primitives

> *Note that Kojo includes the Java and Scala runtime environments. Anything that is part of these environments is available out of the box in Kojo.*

This section describes some useful primitives available within Kojo.

2.1 Turtle

Command	Description
`clear()`	Clears the turtle canvas, and brings the turtle to the center of the canvas.
`forward(steps)`	Moves the turtle forward by the given number of steps.
`back(steps)`	Moves the turtle back by the given number of steps.
`right()`	Turns the turtle right (clockwise) through ninety degrees.
`right(angle)`	Turns the turtle right (clockwise) through the given angle in degrees.
`right(angle, radius)`	Turns the turtle right (clockwise) through the given angle in degrees, along the arc of a circle with the given radius.
`left()`, `left(angle)`, `left(angle, radius)`	These commands work in a similar manner to the corresponding `right()` commands.
`setAnimationDelay(delay)`	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).
`setPenColor(color)`	Specifies the color of the pen that the turtle draws with.
`setPenThickness(size)`	Specifies the width of the pen that the turtle draws with.
`setFillColor(color)`	Specifies the fill color of the figures drawn by the turtle.

Command	Description
`setBackground(color)`	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 `Color`, `ColorHSB`, and `ColorG` functions.
`penUp()`	Pulls the turtle's pen up, and prevents it from drawing lines as it moves.
`penDown()`	Pushes the turtle's pen down, and makes it draw lines as it moves. The turtle's pen is down by default.
`hop(steps)`	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.
`cleari()`	Clears the turtle canvas and makes the turtle invisible.
`invisible()`	Hides the turtle.
`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()`.
`write(obj)`	Makes the turtle write the specified object as a string at its current location.
`setPenFontSize(n)`	Specifies the font size of the pen that the turtle writes with.

2.2 Picture

2.2.1 Picture Creation

These functions create pictures. The created picture needs to be drawn (via the `draw(picture)` command) for it to become visible in the canvas.

Function	Description
`Picture { drawingCode }`	Makes a picture out of the given turtle drawing code.
`picRow(pictures)` `HPics(pictures)`	Creates a horizontal row of the supplied pictures.
`picCol(pictures)` `VPics(pictures)`	Creates a vertical column of the supplied pictures.

Function	Description
`picStack(pictures)` `GPics(pictures)`	Creates a stack of the supplied pictures.
`PicShape.hline(length)`	Creates a picture of a horizontal line with the given length.
`PicShape.vline(length)`	Creates a picture of a vertical line with the given length.
`PicShape.rect(height, width)`	Creates a picture of a rectangle with the given height and width.
`PicShape.circle(radius)`	Creates a picture of a circle with the given radius.
`PicShape.arc(radius)`	Creates a picture of an arc with the given radius and angle.
`PicShape.text(content, size)`	Creates a picture out of the given text with the given font-size.
`PicShape.image(fileName)`	Creates a picture out of an image from the file with the given name.
`PicShape.widget(w)`	Creates a picture out of the given widget.

2.2.2 Picture Transformation

These functions transform picture. A transform is applied to a picture using the `->` operator. Transforms are combined using the `*` operator, e.g., `trans(10, 20) * rot(30) -> picture` first translates and then rotates the given picture.

Function	Description
`trans(x, y) -> picture`	Creates a new picture by translating the given picture by the given x and y values.
`offset(x, y) -> picture`	Creates a new picture by offsetting the given picture by the given x and y values with respect to the global (canvas) coordinate system.
`rot(angle) -> picture`	Creates a new picture by rotating the given picture by the given angle.
`scale(factor) -> picture` `scale(xf, yf) -> picture`	Creates a new picture by scaling the given picture by the given scaling factor(s).
`penColor(color) -> picture`	Creates a new picture by setting the pen color for the given picture to the given color.

Function	Description
`trans(x, y) -> picture`	Creates a new picture by translating the given picture by the given x and y values.
`offset(x, y) -> picture`	Creates a new picture by offsetting the given picture by the given x and y values with respect to the global (canvas) coordinate system.
`fillColor(color) -> picture`	Creates a new picture by filling the given picture with the given color.
`penWidth(thickness) -> picture`	Creates a new picture by setting the pen width for the given picture to the given thickness.
`hue(factor) -> picture`	Creates a new picture by changing the hue of the given picture's fill color by the given factor. The factor needs to be between -1 and 1.
`sat(factor) -> picture`	Creates a new picture by changing the saturation of the given picture's fill color by the given factor. The factor needs to be between -1 and 1.
`brit(factor) -> picture`	Creates a new picture by changing the brightness of the given picture's fill color by the given factor. The factor needs to be between -1 and 1.
`opac(factor) -> picture`	Creates a new picture by changing the opacity of the given picture by the given factor.
`axes -> picture`	Creates a new picture by turning on the local axes for the given picture (to help during picture construction).

2.2.3 Picture Interaction

These commands enable a picture to interact with a user.

Command	Description

`picture.react { self =>`
`  // reaction code`
`}`
	The code supplied to react is called many times per second to allow the picture to take part in an animation. The code can use the `isKeyPressed(keyCode)` function to make the animation interactive.

`p2.onMouseClick { (x, y) =>`
`  // on click code`
`}`
	The supplied code is called, with the current mouse position as input, when the mouse is clicked inside the picture.

Command	Description

`picture.react { self =>`
`  // reaction code`
`}`
	The code supplied to react is called many times per second to allow the picture to take part in an animation. The code can use the `isKeyPressed(keyCode)` function to make the animation interactive.

`p2.onMouseDrag { (x, y) =>`
`  // on drag code`
`}`
	The supplied code is called, with the current mouse position as input, when the mouse is dragged inside the picture.

`p2.onMouseEnter { (x, y) =>`
`  // on enter code`
`}`
	The supplied code is called, with the current mouse position as input, when the mouse enters the picture.

`p2.onMouseExit { (x, y) =>`
`  // on exit code`
`}`
	The supplied code is called, with the current mouse position as input, when the mouse exits the picture.

`p2.onMouseMove { (x, y) =>`
`  // on move code`
`}`
	The suppled code is called, with the current mouse position as input, when the mouse is clicked.

`p2.onMousePress { (x, y) =>`
`  // on press code`
`}`
	The supplied code is called, with the current mouse position as input, when the mouse is pressed inside the picture.

`p2.onMouseRelease { (x, y)`
`=>`
`  // on release code`
`}`
	The supplied code is called, with the current mouse position as input, when the mouse is released inside the picture.

2.2.4 Picture game example

Here is a very simple game that shows picture creation, transformation, and interaction in action.

Listing 2.1: A Picture based game

```
// 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
def p = PicShape.rect(40, 60)
val p1 = fillColor(green) * penColor(gray) -> p
val cb = canvasBounds
val walls = GPics(
    trans(-cb.width / 2, -cb.height / 2) -> PicShape.vline(cb.height),
```

```
    trans(cb.width / 2, -cb.height / 2) -> PicShape.vline(cb.height),
    trans(-cb.width / 2, -cb.height / 2) -> PicShape.hline(cb.width),
    trans(-cb.width / 2, cb.height / 2) -> PicShape.hline(cb.width)
)

cleari()
draw(p1)
draw(walls)

p1.react { self =>
    self.translate(1, 0)
    self.scale(1.001)
    if (isKeyPressed(Kc.VK_P)) {
        self.rotate(1)
    }
    if (self.collidesWith(walls)) {
        self.setFillColor(red)
        self.stopReactions()
    }
}

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

activateCanvas()
```

Run code online **View code online**

2.2.5 Picture based widgets

To utilize a user-interface widget as a picture, you need to first create it, and then make a picture out of it by using the `PicShape.widget(w)` function. Once you have a widget picture, you can use it just like any other picture.

So how do you create a widget? You are, of course, free to go directly to the Java Swing API and create any widget/component from there that you want. But Kojo provides high-level support for creating a selected few user interface widgets:

Constructor Function	Description
`Label(text)`	Creates a Label with the given text.
`TextField(default)`	Creates a text field with the given default value.
`Button(text) { // on click code }`	Creates a Button with the given text and the given action code. The action code is called whenever the button is clicked. From within the action code, you can read the value of any other widget by using that widget's `value` function.
`DropDown(elements)`	Creates a drop down list with the given elements
`Slider(min, max, current, step)`	Creates a slider that goes from `min` to `max` in increments of `step`, with `current` as the initial value.

Listing 2.2: Picture based widgets

```
// An example showing the use of user-interface widgets
// (as pictures) in the canvas

val reptf = TextField(5)
val linef = TextField(60)
val angletf = TextField(360 / 5)
val colordd = DropDown("blue", "green", "yellow")
val colors = Map(
    "blue" -> blue,
    "green" -> green,
    "yellow" -> yellow
)

val instructions =
    <html>
      To run the example, specify the required <br/>
      values in the fields below, and click on the <br/>
      <strong><em>Make + animate</em></strong> button at the bottom.
    </html>.toString
```

```
val panel = ColPanel(
    RowPanel(Label(instructions)),
    RowPanel(Label("Line size: "), linef),
    RowPanel(Label("Repeat count: "), reptf),
    RowPanel(Label("Angle between lines: "), angletf),
    RowPanel(Label("Fill color: "), colordd),
    RowPanel(Button("Make + animate figure") {
        val velocity = 50 // pixels per second
        val cbx = canvasBounds.x
        val figure =
            trans(cbx, 0) * fillColor(colors(colordd.value)) -> Picture {
            repeat(reptf.value) {
                forward(linef.value)
                right(angletf.value)
            }
        }
        draw(figure)
        val starttime = epochTime
        figure.react { self =>
            val ptime = epochTime
            val t1 = ptime - starttime
            val d = cbx + velocity * t1
            self.setPosition(d, 0)
        }
    })
)

val pic = PicShape.widget(panel)
cleari()
draw(trans(canvasBounds.x, canvasBounds.y) -> pic)
```

Run code online **View code online**

2.3 Types

As mentioned earlier, Kojo includes the Java and Scala runtime environments. Any type available in these environments can be used in Kojo via the import keyword.

It's useful to know about the following predefined types within Kojo:

Numbers, which are available in a few different types:

Int – integer, e.g., -3, -2, -1, 0, 1, 2, 3, etc. The maximum integer value (based on the fact that integers are stored in 32 bits and are signed) is 2147483647. If you want an integral number bigger than that, you will need to use a long.

Long – long integer, e.g., 2222222222l (note the l at the end that tells Kojo that this is a long integer)

Double – double precision decimal fraction, e.g., 9.5

Rational – rational numbers or fractions. You can create rationals in a couple of different ways:

Literal-values, which are built using r prefixed strings, e.g., `r''2/3''` and `r''5/7''`.

Rational-conversions, using the `r` function available within other numbers, e.g., 1.r, 3.r, etc. This works well with the fact that once you have a rational number in an arithmetic expression, it lifts the other numbers in the expression to make them rationals, resulting in a rational result for the whole expression, e.g., `2.r / 3`, `3.r / 4 + 5.r / 6`, etc.

Listing 2.3: Using numbers

```
// #worksheet
// Part 1 -- Ints and Double
// the usual arithmetic operators are available
// +, -, * (multiplication), and / (division)
2 + 3 * 4 / 7 - 6
2.1 * 3.2 + 7

// Part 2 - Rationals
// In additional to the usual operators,
// ** (exponentiation) is also available
val a = r"9/10"
val b = a * r"10/9"
val x = 3.r / 4 + 4.r / 5
val y = 1.r / 4 + 1.r / 8 * 2 * 3
2.r / 4 ** 2
8.r ** (1.r / 3)
val z = x.toDouble
```

Run code online **View code online**

Booleans, which represent truth and falsity in programs, and are used within conditions. You can create booleans in a few different ways:

Literal-values – `true` and `false`.

Comparison-operators like `==`, `<`, `<=`, `>`, and `>=` working on two values that support the operator, e.g., `2 < 3`, `4 == 5`, `blue == red`, etc.

Colors, which are used for setting background colors, and the pen and fill colors of shapes. Functions for working with colors are described in Section 2.5 on page 20

Strings, which represent language text, and are useful for input to and output from Kojo. You can create strings in few different ways:

Literal-values – text enclosed within double quotes, e.g., ``this is a literal string''.

Multi-line-literals – text enclosed within trip quotes (see Listing 2.4 for an example).

Interpolated-strings – text enclosed within double or triple quotes that can contain external values, e.g., s''this string has $external data''.

HTML, which can be used to create styled text – for use within Label widgets.

Listing 2.4: Booleans, Strings, and HTML

```
// #worksheet
// booleans
val b1 = true
val b2 = false
// the operators && (and), || (or), and !(not) are available
b1 && b2
b1 || b2
!b2
val n = 10
(n > 1) && (n < 15)
(n > 1) || (n < 5)
(n > 1) && (n < 5)


// strings
val string1 = "some text"
val string2 = """Line 1
Line2
Line3
"""

val x = 30
val y = 40
val string3 = s"the value of x is $x"
val string4 = s"""the value of y is $y
and the sum of x and y is ${x + y}
"""

// html/xml
val html1 = <html>
This is some <strong>html</strong> text
</html>
```

Run code online **View code online**

Pictures, which are used for making composable and transformable shapes that can be animated. Functions for working with pictures are described in Section 2.2 on page 10

Widgets, like text fields, buttons, drop-downs, etc., which are used to obtain input from the user. Widgets are further described in Section 2.2.5 on page 15.

Collections, which let you store data values in different ways (sequences, sets, maps, etc.) for future processing. Collections are further described in the next section.

2.4 Collections

Detailed information on Scala collections is available in the Scala collections guide. These collections are available within Kojo, and can be used directly as described in the guide.

2.5 Utility commands and functions

Here are some miscellaneous utility functions available within Kojo:

Function	Description
`Color(red, green, blue, opacity)`	Creates a new color based on the given red, green, blue, and (optional) opacity values. This color can be used with commands like setPenColor, etc.
`hueMod(color, factor)`	Computes and returns a new color made by changing the hue of the given color by the given factor. The factor needs to be between -1 and 1.
`satMod(color, factor)`	Computes and returns a new color made by changing the saturation of the given color by the given factor. The factor needs to be between -1 and 1.
`britMod(color, factor)`	Computes and returns a new color made by changing the brightness of the given color by the given factor. The factor needs to be between -1 and 1.
`canvasBounds`	Returns the bounds of the canvas: the x and y coordinates of its bottom left point, and its width and height.
`textExtent(text, fontSize)`	Computes and returns the size/extent of the given text fragment for the given font size. The extent is returned as a bounding rectangle with the bottom left and the top right points.
`activateCanvas`	Transfers focus to the canvas, so that keyboard events are directed to the canvas.

3 Composition

Composition allows you to combine primitives (and higher level building-blocks) to do useful things. The following are some of the forms of composition available within Kojo:

3.1 Command composition

Sequencing allows you to chain commands together one after the other.

Looping lets you repeat a bunch of commands more than once.

Recursion allows you, while you are defining a command, to get it to call itself.

Listing 3.1: Command composition

```
// examples of command composition
// a sequence of commands
clear()
forward(100)
right()
forward(50)

// a loop
repeat(4) {
    forward(10)
}

// a loop with an index
repeati(4) { i =>
    println(i)
}

// a loop with a condition
var x = 0
repeatWhile(x < 4) {
    println(x)
    x += 1
}

// do the same thing as above with a different kind of loop
x = 0
```

```
repeatUntil(x > 3) {
    println(x)
    x += 1
}

// recursion
def pattern(n: Int) {
    if (n < 10) {
        forward(n)
    }
    else {
        forward(n)
        right()
        pattern(n - 10)
    }
}
clear()
pattern(100)
```

Run code online **View code online**

3.2 Function composition

Nesting/chaining allows you to call many functions one after the other.

Sequencing with stored intermediate results is an alternative to nesting/chaining.

Higher-order-functions let you use functions as inputs to a function, and a function as an output from a function.

Recursion allows you, while you are defining a function, to get it to call itself.

For-comprehensions give you a compact notation for transforming collections of data into other collections. This is based on nested and chained calls to certain well defined methods within the collection types. See the *Programming in Scala* book for more details.

Listing 3.2: Function composition

```
// #worksheet
// function nesting; functions are called from inside out
math.abs(math.min(20, -30))

// sequencing to do the same thing as above
val r1 = math.min(20, -30)
math.abs(r1)
```

```
val pic = PicShape.rect(40, 60)
// function chaining; functions are called from left to right
rot(60) * trans(10, 15) * scale(1.5) -> pic

// higher order functions
(1 to 10).filter { n => n % 2 == 0 }
(1 to 5).map { n => n * 2 }

// a function that uses recursion
// as there is only one expression inside the function,
// the curly brackets are optional
// if the function had a sequence of expressions,
// the curly brackets would be mandatory
def factorial(n: Int): Int = {
    if (n == 0) 1 else n * factorial(n - 1)
}
factorial(5)

// for comprehensions
for (i <- 1 to 5) yield (i * 2)
```

Run code online **View code online**

3.3 Data composition

You compose data values by putting them inside other data values. This is enabled via the following:

Collections – these are predefine classes/types that let you put your data inside them. Collections are described further in Section 2.4 on page 19.

Classes – you can define your own classes that combine data values. This is described further in Section 4 on page 25

Listing 3.3: Data composition

```
// #worksheet
// example of data composition (and abstraction)
// the 'num' and 'den' data values are combined together
// in the Rational class,
// and associated with a '+' function.
case class Rational(num: Int, den: Int) {
    require(den != 0)
    def +(other: Rational) =
        Rational(num * other.den + other.num * den, den * other.den)
}
```

```
// we create two instances of the Rational class and add them
Rational(2, 3) + Rational(4, 5)
```

Run code online **View code online**

4 Abstraction

Abstraction allows you to give names to your compositions. These named elements then become available within your program – for direct use and as building-blocks for further composition. Hiding of internal (implementation) details is an important aspect of abstraction; while using an abstraction, you can focus on what it does without worrying about how it does it. Abstraction within Kojo is done with the help of the following keyword instructions:

val, which lets you create a named value.

def, which lets you define a new command or function.

Listing 4.1: Using val and def

```
// #worksheet
// create a named value
val size = 50

// define a new command
def square(n: Int) {
    repeat(4) {
        forward(n)
        right()
    }
}
clear()
// use the new command along with the named value
square(size)
square(size * 2)


// define a new function
def sum(n1: Int, n2: Int) = n1 + n2

// use the new function
sum(5, 7)
```

Run code online **View code online**

class, which lets you create a new class (and type). Classes help you to compose data values, along with related commands and functions, into named entities that you can use in your programs. Note that there are two versions of the class keyword – just `class` by

itself, and `case class`. The use of case classes is encourage in Kojo, as they are simpler to use and more powerful. See the *Programming in Scala* book for more details.

Listing 4.2: Using classes

```
// #worksheet
// example of abstraction (and data composition)
// a new class called Rational is created
// the 'num' and 'den' data values are combined together
// in this class,
// and associated with a '+' function.
case class Rational(num: Int, den: Int) {
    require(den != 0)
    def +(other: Rational) =
        Rational(num * other.den + other.num * den, den * other.den)
}

// we create two instances of the Rational class and add them
Rational(2, 3) + Rational(4, 5)
```

Run code online **View code online**

trait, which lets you create new traits. Traits help you to separate the interfaces of abstractions from their implementations. They also help with mixin composition, rich interfaces, class decoration, etc. See the *Programming in Scala* book for more details.

implicit, which gives you a flexible and controlled way to extend software. See the *Programming in Scala* book for more details.

5 Selection

Selection allows you to choose from a set of alternative instructions as you write your programs. Selection is based on the following constructs:

If-else allows you to choose between alternatives based on boolean conditions. This works well for a few few alternatives and simple conditions

Pattern-matching also allows you to choose between alternatives based on boolean conditions. But the conditions can be expressed in much more powerful ways (than if-else). Pattern matching works well with a large number of alternatives and complex conditions.

Listing 5.1: Examples of selection

```
// example of if-else
val size = 40
val thershold = 50
// make big shapes red
if (size > thershold) {
    setFillColor(red)
}
else {
    setFillColor(green)
}

// make small shapes move faster
val speed = if (size < thershold) 100 else 60

// example of pattern matching
val data = "abc"
val result = data match {
    case "abc" => "xyz"
    case "def" => "jkl"
    case _ => "Unknown"
}
println(result)
```

Run code online **View code online**

6 Conclusion

This book outlines some core ideas about programming used within Kojo, and provides a reference to commonly used commands, functions, and keywords. It provides examples for all the ideas discussed.

Kojo benefits tremendously from being built on the foundations of the Java and Scala platforms, and from using Scala as its programming language. It benefits due to the rich functionality of these platforms, which is available out of the box in Kojo. It also benefits from all the documentation and books out there that describe the Scala language and the Java libraries. The knowledge available in this reading material can be directly used within Kojo:

- You can read more about the Scala language in the *Programming in Scala* book.
- You can find more information about the Scala API in the official Scaladoc.
- You can find more information about the Java API in the official Javadoc.

If you have any comments on the book, don't hesitate to write to me at pant.lalit@gmail.com.