Creative Scala

Dave Gurnell and Noel Welsh

July 2015

underscore

Creative Scala

July 2015

Published by Underscore Consulting LLP, Brighton, UK.

Copies of this, and related topics, can be found at http://underscore.io/training.

Team discounts, when available, may also be found at that address.

Contact the author regarding this text at: hello@underscore.io.

Our courses, workshops, and other products can help you and your team create better software and have more fun. For more information, as well as the latest Underscore titles, please visit http://underscore.io/training.

Contents

Foreword

Creative Scala is aimed at developers who have no prior experience in Scala. It is designed to give you a fun introduction to functional programming. We assume you have some familiarity with another programming language but little or no experience with Scala or other functional languages.

Our goal is to demonstrate the building blocks that Scala developers use to create programs in a clear, succinct, declarative manner. Working through the exercises in the book should take a few hours, after which we hope you will have a feel of what Scala can do for your applications.

Although this book will give you the basic mental model required to become competent with Scala, you won't finish knowing *everything* you need to be self-sufficient. For further advancement we recommend considering one of the many excellent Scala textbooks out there, including our own Essential Scala.

If you are working through the exercises on your own, we highly recommend joining our Gitter chat room to provide get help with the exercises and provide feedback on the book.

The text of Creative Scala is open souce, as is the source code for the Doodle drawing library used in the exercises. You can grab the code from our Github account. Contact us on Gitter or by email if you would like to contribute.

Thanks for downloading and happy creative programming!

—Dave and Noel

Notes on the Early Access Edition

> **Warning**
>
> This is an *early access* release of Creative Scala. This means there are unfinished aspects as detailed below. There may be typos and errata in the text and examples.
>
> If you spot any mistakes or would like to provide feedback, please let us know via our Gitter chat room or by email:
>
> - Dave Gurnell (dave@underscore.io)
> - Noel Welsh (noel@underscore.io)

Acknowledgements

Creative Scala was written by Dave Gurnell and Noel Welsh. Many thanks to Richard Dallaway, Jonathan Ferguson, and the team at Underscore for their invaluable contributions and extensive proof reading.

Thanks also to the many people who pointed out errors or made suggestions to improve the book: Neil Moore, and many more for whom we didn't record the name.

Chapter 1

Expressions, Values, and Types

Scala programs have three fundamental building blocks: *expressions*, *values*, and *types*. An *expression* is a fragment of Scala code that we write in an a text editor. Valid expressions have a *type* and calculate a *value*. For example, this expression has the type `String` and calculates the value `HELLO WORLD!`

```
"Hello world!".toUpperCase
// res0: String = "HELLO WORLD!"
```

A Scala program goes through two distinct stages. First it is *compiled*; if compiles successfully it can then be *executed*. The most important distinction between types and values is that types are determined at compile time, whereas values can only be determined at run time. Values can change each time we run the code, whereas types are fixed. For example, the following expression is certainly of type `String`, but its value depends on the the user input each time it is run:

```
readLine.toUpperCase
```

We are used to thinking of types that refer to sets of literals such as `Int`, `Boolean`, and `String`, but in general a type is defined as *anything we can infer about a program without running it*. Scala developer use types to gain assurances about our programs before we put them into production.

1.1 Simple Scala Expressions

Let's look at some of the basic kinds of expressions we can write in Scala:

1.1.1 Literals

The simplest kinds of expressions are *literals*. These are fragments of code that "stand for themselves". Scala supports a similar set of literals to Java:

```
// Integers:
1
// res0: Int = 1

// Floating point numbers:
0.1
// res1: Double = 0.1

// Booleans:
true
// res2: Boolean = true
```

```
// Strings:
"Hello world!"
// res3: String = Hello world!

// And so on...
```

1.1.2 Method Calls

Scala is a completely object-oriented programming language, so all values are objects with methods that we can call. Method calls are another type of expression:

```
123.4.ceil
// res4: Double = 124.0

true.toString
// res5: String = "true"
```

1.1.3 Constructor Calls

Scala only has literals for a small set of types (`Int`, `Boolean`, `String`, and so on). To create values of other types we either need to call a method, or we need to call a *constructor* using the new keyword. This behaves similarly to new in Java:

```
import java.util.Date
// import java.util.Date

new Date()
// res4: java.util.Date = Tue Feb 10 10:30:21 GMT 2015
```

The new operator tends to be a distraction when writing larger, more complex expressions. For this reason, Scala libraries typically provide *factory methods* to wrap constructor calls. The effective difference is that we can create many Scala data types without writing new:

```
List(1, 2, 3)
// res6: List[Int] = List(1, 2, 3)
```

Other than the lack of a new keyword, the semantics here are similar to the `Date` example above:

- `List(1, 2, 3)` is an expression that returns a value;
- `List[Int]` is its type.

1.1.4 Operators

Operators in Scala are actually method calls under the hood. Scala has a set of convenient syntactic shorthands to allow us to write normal-looking code without excessive numbers of parentheses. The most common of these is *infix syntax*, which allows us to write any expression of the form `a.b(c)` as `a b c`, without the full stop or parentheses:

```
1 .+(2).+(3).+(4) // the space prevents `1` being treated as a double
// res0: Int = 10

1 + 2 + 3 + 4
// res1: Int = 10
```

1.1.5 Conditionals

Many other syntactic constructs are expressions in Scala, including some that are statements in Java. Conditionals ("if expressions") are a great example:

```
// Conditionals ("if expressions"):
if(123 > 456) "Higher!" else "Lower!"
// res6: String = Lower!
```

1.1.6 Blocks and Side-Effects

Blocks are another type of expression in Scala. Running a block runs each contained expression in order. The type and return value of the block are determined by the *last* contained expression:

```
{
  println("First line")
  println("Second line")
  1 + 2 + 3
}
// res0: Int = 6
```

In functional programming we make the distiction between "pure expressions" and expressions that have "side effects":

- *pure expressions* do nothing more than calculate a value;
- expressions with *side effects* do something else aside from calculate their result—for example, `println` prints a message to the console.

Because the results of intermediate expressions in a block are thrown away, it doesn't make sense to use pure expressions there. The Scala console even warns us when we try this:

```
{
  1 + 2 + 3
  4 + 5 + 6
}
// <console>:9: warning: a pure expression does nothing in statement position;
//              you may be omitting necessary parentheses
//                1 + 2 + 3
//                    ^
// res0: Int = 15
```

The message here is warning us that the intermediate expression `1 + 2 + 3` is wasted computation. All it does is calculate the value 6. We immediately throw the result away and calculate `4 + 5 + 6` instead. We might as well simply write `4 + 5 + 6` and get rid of the block.

Side-effecting expressions, by contrast, make perfect sense within a block. `println` expressions are a great example of this—they do something useful even though they don't return a useful value:

```
{
  println("Intermediate result: " + (1 + 2 + 3))
  4 + 5 + 6
}
// Intermediate result: 6
// res0: Int = 15
```

Scala developers tend to prefer pure expressions to side-effects because they are easier to reason about. See the section on Substitution for more information. We won't use blocks in anger until the next chapter when we start declaring intermediate values and re-use them in later expressions.

1.2 Images

Numbers and text are boring. Let's switch to something more interesting—images! Grab the *Doodle* project from https://github.com/underscoreio/doodle. This toy drawing library will provide the basis for most of the exercises in this course. Start a Scala console to try Doodle out:

```
bash$ git clone https://github.com/underscoreio/doodle.git
# Cloning ...

bash$ cd doodle

bash$ ./sbt.sh console
[info] Loading project definition from /.../doodle/project
[info] Set current project to doodle (in build file:/.../doodle/)
[info] Compiling 1 Scala source to /.../doodle/jvm/target/scala-2.11/classes...
[info] Starting scala interpreter...
[info]
import doodle.core._
import doodle.syntax._
import doodle.jvm.Java2DCanvas._
import doodle.backend.StandardInterpreter._
import doodle.examples._
Welcome to Scala version 2.11.7 (Java HotSpot(TM) 64-Bit Server VM, Java 1.8.0_20-ea).
Type in expressions to have them evaluated.
Type :help for more information.

scala>
```

1.2.1 Primitive Images

The Doodle console gives us access to some useful drawing tools as well as the regular Scala standard library. Try creating a simple shape:

```
Circle(10)
// res0: doodle.core.Circle = Circle(10.0)
```

> **Tip**
>
> *How To Run This Example*
>
> When you see an example like the one above, enter the line of Scala code at the `scala>` prompt in the console. You should see the text in the comment as output:
>
> ```
> scala> Circle(10)
> res0: doodle.core.Circle = Circle(10.0)
> ```
>
> We haven't written the `scala>` prompts in the examples in this book because they make it difficult to copy and paste text into the console. We've written the console output as comments for the same reason.

Notice the type and value of the expression we just entered. The type is `doodle.core.Circle` and the value is `Circle(10.0)`—a circle with a 10 pixel radius.

We can draw this circle (and other images) using the `draw` method. Try drawing the circle now:

```
res0.draw
```

Figure 1.1: A circle

A window should appear containing the following:

Doodle supports a handful “primitive” images: circles, rectangles, and triangles. Let’s try drawing a rectangle:

```
Rectangle(50, 100).draw
```

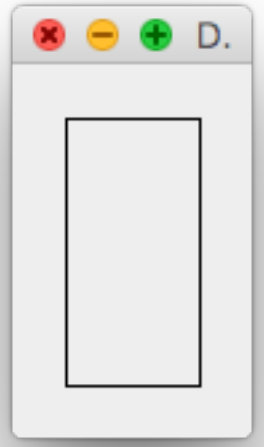

Figure 1.2: A rectangle

Finally let’s try a triangle:

```
Triangle(60, 40).draw
```

1.2.2 Layout

We can write expressions to combine images producing more complex images. Try the following code—you should see a circle and a rectangle displayed beside one another:

Figure 1.3: A triangle

```
(Circle(10) beside Rectangle(10, 20)).draw
```

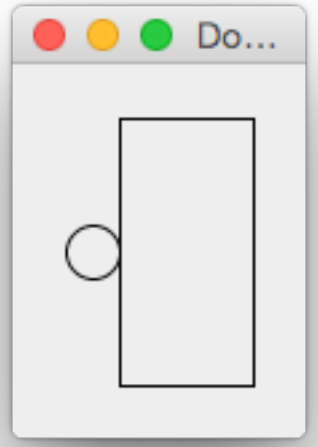

Figure 1.4: A circle beside a rectangle

Doodle contains several layout operators for combining images. Try them out now to see what they do:

Operator	Type	Description	Example
`Image beside Image`	`Image`	Places images horizontally next to one another.	`Circle(10) beside Circle(20)`
`Image above Image`	`Image`	Places images vertically next to one another.	`Circle(10) above Circle(20)`
`Image below Image`	`Image`	Places images vertically next to one another.	`Circle(10) below Circle(20)`

Operator	Type	Description	Example
`Image on Image`	`Image`	Places images centered on top of one another.	`Circle(10) on Circle(20)`
`Image under Image`	`Image`	Places images centered on top of one another.	`Circle(10) under Circle(20)`

Exercise: Compilation Target

Create a line drawing of an archery target with three concentric scoring bands:

For bonus credit add a stand so we can place the target on a range:

See the solution

1.2.3 Colour

In addition to layout, Doodle has some simple operators to add a splash of colour to our images. Try these out to see how they work:

Operator	Type	Description	Example
`Image fillColor Color`	`Image`	Fills the image with the specified colour.	`Circle(10) fillColor Color.red`
`Image lineColor Color`	`Image`	Outlines the image with the specified colour.	`Circle(10) lineColor Color.blue`
`Image lineWidth Int`	`Image`	Outlines the image with the specified stroke width.	`Circle(10) lineWidth 3`

Doodle has various ways of creating colours. The simplest are the predefined colours in `shared/src/main/scala/doodle/core/Comm`
Here are a few of the most important:

Color	Type	Example
`Color.red`	`Color`	`Circle(10) fillColor Color.red`
`Color.blue`	`Color`	`Circle(10) fillColor Color.blue`
`Color.green`	`Color`	`Circle(10) fillColor Color.green`
`Color.black`	`Color`	`Circle(10) fillColor Color.black`
`Color.white`	`Color`	`Circle(10) fillColor Color.white`
`Color.gray`	`Color`	`Circle(10) fillColor Color.gray`
`Color.brown`	`Color`	`Circle(10) fillColor Color.brown`

Exercise: Stay on Target

Colour your target red and white, the stand in brown (if applicable), and some ground in green:

See the solution

1.3 Take Home Points

1.3.1 Substitution

In the absence of side-effects, an expression will always evaluate to the same value. For example, `3 + 4` will always evaluate to 7 no matter how many times we compile or run the code.

Figure 1.5: Simple archery target

Figure 1.6: Archery target with a stand

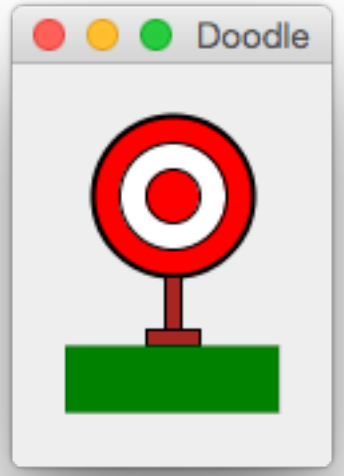

Figure 1.7: Colour archery target

Given these restrictions, the expressions `3 + 4` and `7` become interchangeable from a user's point of view. This is known as the *substitution model* of evaluation, although you may remember it as "simplifying formulae" from your maths class at school.

As programmers we must develop a mental model of how our code operates. In the absence of side-effects, the substitution model always works. If we know the types and values of each component of an expression, we know the type and value of the expression as a whole.

Functional programmers aim to avoid side-effects for this reason: it makes our programs easy to reason about without having to look beyond the current block of code.

1.3.2 Types in Scala

We've seen several types so far, including primitive Scala types such as `Int`, `Boolean`, and `String`, the `Date` type from the Java standard library, `List` from the Scala standard library, and the `Circle`, `Rectangle`, `Image`, and `Color` types from Doodle. Let's take a moment to see how all of these fit together:

All types in Scala fall into a single *inheritance hierarchy*, with a grand supertype called `Any` at the top. `Any` has two subtypes, `AnyVal` and `AnyRef`.

- `AnyVal` is a supertype of the JVM's fixed set of "value types", all of which which we know from Java: `Int` is `int`, `Boolean` is `boolean`, and so on. `AnyVal` is also the supertype of `Unit`, which we will discuss in a moment.
- `AnyRef` is the supertype of all JVM "reference types". It is an alias for Java's `Object` type. `AnyRef` is the supertype of all Java and Scala classes.

The `Unit` type is Scala's equivalent of `void` in Java or C—we use it to write code that evaluates to "no interesting value":

```
val uninteresting = println("Hello world!")
// Hello world!
// uninteresting: Unit = ()
```

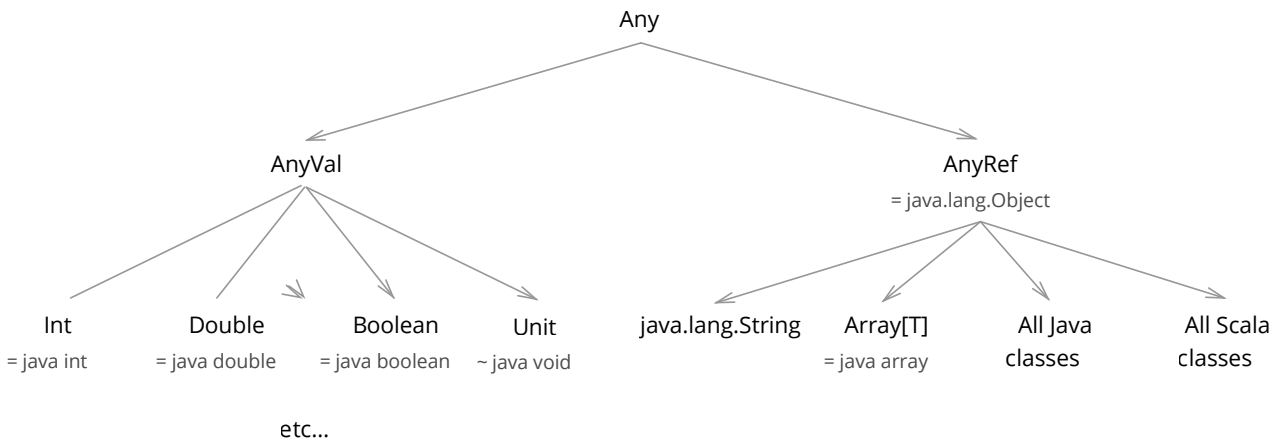

Figure 1.8: Scala's type hierarchy

While `void` is simply a syntax, `Unit` is an actual type with a single value, `()`. Having an concrete type for `Unit` and value allows us to reason about side-effecting code with the same principles as functional code. This is essential for a language like Scala that bridges the worlds of the imperative and the functional.

We have so far seen two imperative-style methods that return `Unit`: the `println` method from the Scala standard library and Doodle's `draw` method. Each of these methods does something useful but neither returns a useful result:

```
val alsoUninteresting = Circle(10).draw
// alsoUninteresting: Unit = ()
```

Designing programs in a functional way involves limiting the side-effects spread throughout our code. Doodle is a classic example of functional design—we assemble a *representation* of the scene we want in a purely functional manner, and then *interpret* the scene to produce output. The `draw` method—our interpreter—can use imperative libraries and mutable state without them intruding into the rest of our application.

Chapter 2

Declarations

Not all programs are single expressions. Sometimes it is useful to bind expressions to a name and re-use them later. These constructs are called *declarations*. Declarations themselves don't have a type or a value. However, they do bind names that can be used as expressions. There are several types of declaration in Scala as we shall see below.

2.1 Value Declarations

The simplest type of declaration binds a name to the result an expression. These are called *value declarations*. They are similar to *variable declarations*, except we cannot assign new values to them after declaration:

```
val blackSquare = Rectangle(30, 30) fillColor Color.black
// blackSquare: doodle.Image = // ...

val redSquare = Rectangle(30, 30) fillColor Color.red
// redSquare: doodle.Image = // ...
```

Exercise: Chess Board

Create an 8x8 square chess board without loops or comprehensions, using the definitions of `redSquare` and `blackSquare` above. Use intermediate value declarations to make your code as compact as possible:

See the solution

2.2 Method Declarations

Sometimes we want to repeat a process, but each time we do it we change some part of what we're doing. For example, imagine creating chess boards where each chess board has a different combination of colors. It would be extremely tedious to declare each chess board as we have above for each choice of colors. What we'd like is to be able to define some process for making chess boards that allows the user to specify the color choice for the particular chess board we're making. This is what *methods* allow us to do.

We have already seen method calls. In this section we are going to see how we can declare our own methods. Like value declarations, method declarations define a name. Instead of giving a name to a value, a method declaration gives a name to a process for creating values:

```
def twoByTwo(color1: Color, color2: Color): Image = {
  val square1 = Rectangle(30, 30) fillColor color1
  val square2 = Rectangle(30, 30) fillColor color2

  (square1 beside square2) above
  (square2 beside square1)
}
```

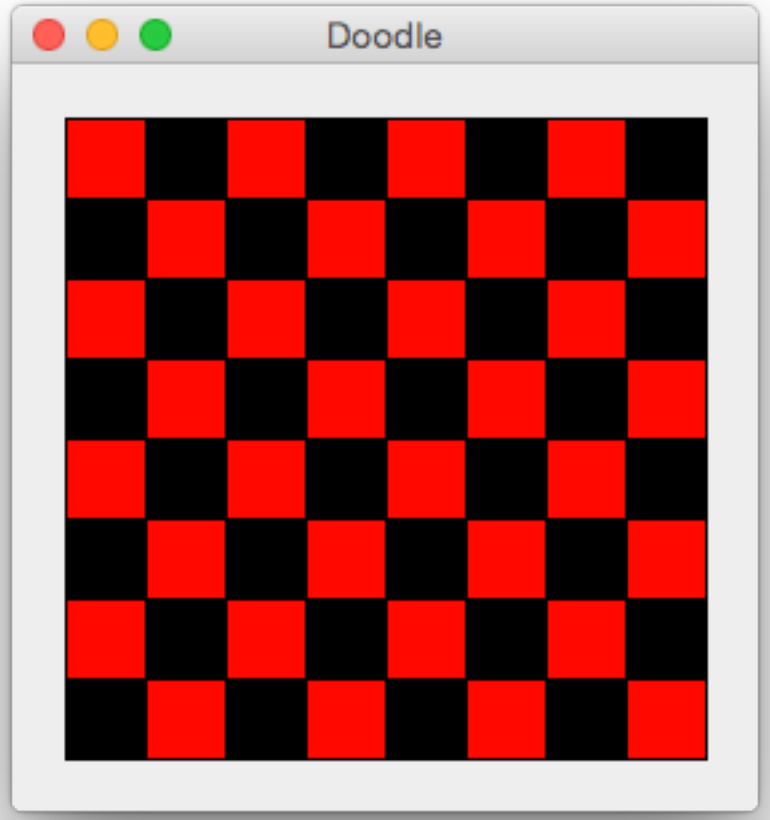

Figure 2.1: Chess board

This declares a method called `twoByTwo`. The method has two parameters, called `color1` and `color2`, which we have declared to have type `Color`. We have also declared the type of the value returned—`Image`. The body of the method, which is enclosed with optional braces (the { and } pair) defines how we create the `Image`. This mirrors the process for creating a two by two chess board that we saw above, but in this case we are using the colors we are passed as parameters.

Exercise: Chess Board

Declare a method called `fourByFour` that constructs a four-by-four chess board using `twoByTwo` declared above. The method should have two parameters, both `Colors`, that it passes on to `twoByTwo`.

You should be able to call `fourByFour` like so

```
fourByFour(Color.cornflowerBlue, Color.seaGreen) beside
fourByFour(Color.chocolate, Color.darkSalmon)
```

to create this picture:

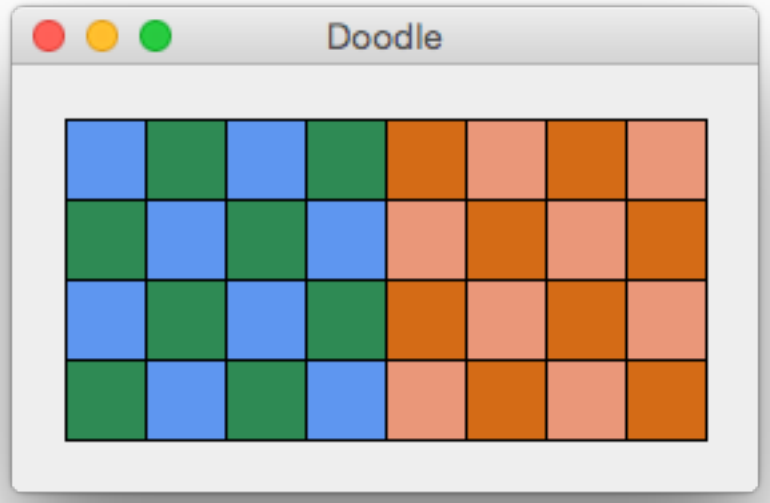

Figure 2.2: Two Chessboards

See the solution

> **Tip**
>
> *Syntax Summary*
>
> We've seen quite a lot of Scala syntax so far. If you can't remember everything we've covered, don't panic! There's a syntax quick reference in the appendices at the end of the book.

2.3 Extended Exercise: Color Palettes

In this exercise we will explore the creation of color palettes. An attractive picture must make good choices for color. Color theory has developed to explain combinations of color that go together. We will use color theory to create programs that can automatically construct attractive color palettes.

2.3.1 Color Representation

In Doodle we can represent a color in one of two ways:

1. as triples containing red, green, and blue values (RGB); or
2. as hue, saturation, and lightness (HSL).

We will use the HSL representation as it better corresponds to our perception of color. If we arrange colors in the familiar color wheel, distance from the center corresponds to lightness and steps around the outside correspond to changes in hue:

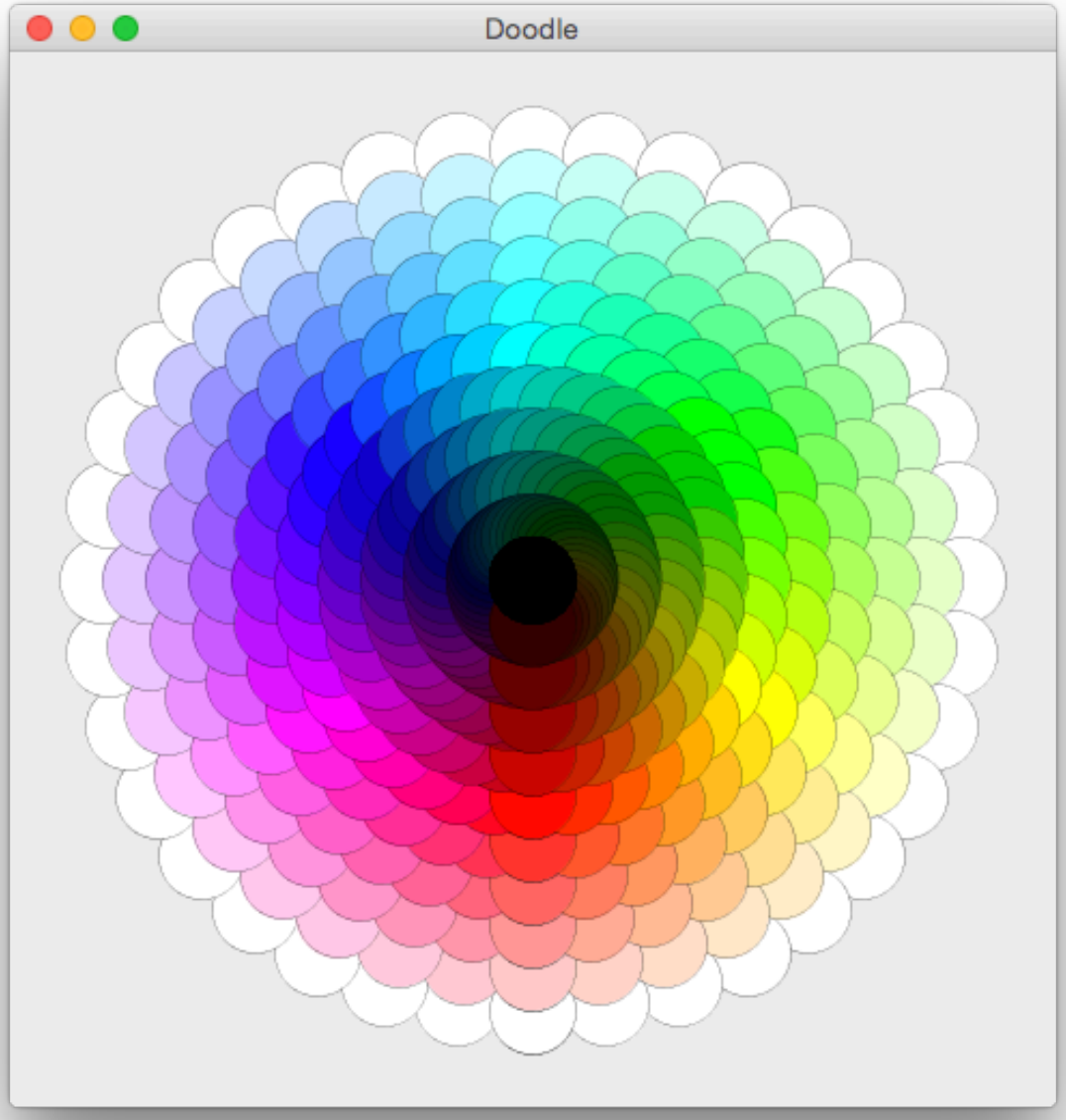

Figure 2.3: A color wheel. A full turn around the wheel represents a 360 degree change in hue.

Saturation, the third dimension, corresponds to intensity of color. The strip of colors below shows the effect of varying saturation from 0.0 to 1.0, for fixed hue (170 degrees) and lightness (0.5). As you can see, changing saturation goes from a dull gray to a bright and vibrant color.

Figure 2.4: The effect of changing saturation while keeping hue and lightness fixed. Saturation increases from left to right, starting at zero and finishing at one.

2.3.2 The Color API

Before we can create color schemes we need to know how to create and manipulate colors.

2.3.2.1 Creating Colors

There are two main methods to create colours:

```
Color.hsl(hue: Angle, saturation: Normalized, lightness: Normalized)
Color.rgb(red: UnsignedByte, green: UnsignedByte, blue: UnsignedByte)
```

These methods use types — `Angle`, `Normalized`, and `UnsignedByte` — that have not seen before. They all represent numbers with some special characteristics. A `Normalized` is a number between 0 and 1. An `UnsignedByte` is an integer between 0 and 255. An `Angle` is unrestricted in value but there are several operations that only make sense on angles (sine, cosine, and so on) and several representations (angles, radians) in common use.

The `Normalized` and `UnsignedByte` types make it explicit that some conversion is necessary from raw number types like `Int` and `Double`. There are many different ways to handle inputs that are out of range, such as clipping them or raising an error, and we require the programmer to be explicit about the approach they want.

For `Normalized` and `UnsignedByte` Doodle provides a default conversion of clipping. For example, if we are creating a `Normalized` (value between 0.0 and 1.0), any input less than 0.0 is set to 0.0 and greater than 1.0 becomes 1.0. To use these conversions import `doodle.syntax.normalized._` or `doodle.syntax.uByte._` and then numbers are *enriched* with methods `normalized` and `uByte` respectively. Here's a quick example. Notice how values out of range are set to the closest valid value.

```
import doodle.syntax.normalized._

0.5.normalized
//res: doodle.core.Normalized = Normalized(0.5)
0.0.normalized
//res: doodle.core.Normalized = Normalized(0.0)
-0.5.normalized
//res: doodle.core.Normalized = Normalized(0.0)
1.5.normalized
//res: doodle.core.Normalized = Normalized(1.0)

import doodle.syntax.uByte._

128.uByte
//res: doodle.core.UnsignedByte = UnsignedByte(0)
```

```
0.uByte
//res: doodle.core.UnsignedByte = UnsignedByte(-128)
255.uByte
//res: doodle.core.UnsignedByte = UnsignedByte(127)
-127.uByte
//res: doodle.core.UnsignedByte = UnsignedByte(-128)
512.uByte
//res: doodle.core.UnsignedByte = UnsignedByte(127)
```

For `Angle` we ask the programmer to specify if the raw number represents a value in degrees, radians, or turns (fractions of a circle, with a full circle being one turn). For `Angles` the import is `doodle.syntax.angle._` which enriches numbers with methods `degrees`, `radians`, and `turns`. Here's an example:

```
import doodle.syntax.angle._

0.degrees
//res: doodle.core.Angle = Angle(0.0)
180.degrees
//res: doodle.core.Angle = Angle(3.141592653589793)
360.degrees
//res: doodle.core.Angle = Angle(6.283185307179586)

math.Pi
//res: Double = 3.141592653589793
math.Pi.radians
//res: doodle.core.Angle = Angle(3.141592653589793)

0.5.turns
//res: doodle.core.Angle = Angle(3.141592653589793)
1.0.turns
//res: doodle.core.Angle = Angle(6.283185307179586)
```

We can now create some colors:

```
Color.hsl(170.degrees, 1.0.normalized, 0.5.normalized)
// res: doodle.core.HSLA = HSLA(Angle(2.9670597283903604),Normalized(1.0),Normalized(0.5),Normalized(1.0))
```

Note that the color we created has four fields. The fourth field is the `alpha` value, which specifies the opacity of the color. There are four parameter methods `Color.hsla` and `Color.rgba` that can be used to specify the `alpha` when creating a color.

2.3.2.2 Modifying Colors

There are several methods to modify colors. These methods all create a new `Color`. No `Color` is ever actually changed after it is created, as doing so breaks substitution.

New `hue`, `saturation`, `lightness`, and `alpha` values can all be set with methods of the same name. Notice how the original color is unchanged.

```
val c = Color.hsl(170.degrees, 1.0.normalized, 0.5.normalized)
//c: doodle.core.HSLA = HSLA(Angle(2.9670597283903604),Normalized(1.0),Normalized(0.5),Normalized(1.0))

c.hue(220.degrees)
//res: doodle.core.Color = HSLA(Angle(3.839724354387525),Normalized(1.0),Normalized(0.5),Normalized(1.0))

c.saturation(0.5.normalized)
//res: doodle.core.Color = HSLA(Angle(2.9670597283903604),Normalized(0.5),Normalized(0.5),Normalized(1.0))
```

```
c.lightness(0.25.normalized)
//res: doodle.core.Color = HSLA(Angle(2.9670597283903604),Normalized(1.0),Normalized(0.25),Normalized(1.0))

c.alpha(0.5.normalized)
//res: doodle.core.Color = HSLA(Angle(2.9670597283903604),Normalized(1.0),Normalized(0.5),Normalized(0.5))

c
//res: doodle.core.HSLA = HSLA(Angle(2.9670597283903604),Normalized(1.0),Normalized(0.5),Normalized(1.0))
```

There are also methods to adjust the existing hue, `saturation`, `lightness`, and `alpha`. These methods all create a new color by adding or subtracting from the existing value of the parameter of interest.

```
val c = Color.hsl(170.degrees, 1.0.normalized, 0.5.normalized)
//c: doodle.core.HSLA = HSLA(Angle(2.9670597283903604),Normalized(1.0),Normalized(0.5),Normalized(1.0))

c.spin(220.degrees)
//res: doodle.core.HSLA = HSLA(Angle(6.806784082777885),Normalized(1.0),Normalized(0.5),Normalized(1.0))

c.lighten(0.2.normalized)
//res: doodle.core.HSLA = HSLA(Angle(2.9670597283903604),Normalized(1.0),Normalized(0.7),Normalized(1.0))

c.darken(0.2.normalized)
//res: doodle.core.HSLA = HSLA(Angle(2.9670597283903604),Normalized(1.0),Normalized(0.3),Normalized(1.0))

c.saturate(0.2.normalized)
//res: doodle.core.HSLA = HSLA(Angle(2.9670597283903604),Normalized(1.0),Normalized(0.5),Normalized(1.0))

c.desaturate(0.2.normalized)
//res: doodle.core.HSLA = HSLA(Angle(2.9670597283903604),Normalized(0.8),Normalized(0.5),Normalized(1.0))

c.fadeIn(0.2.normalized)
//res: doodle.core.HSLA = HSLA(Angle(2.9670597283903604),Normalized(1.0),Normalized(0.5),Normalized(1.0))

c.fadeOut(0.2.normalized)
//res: doodle.core.HSLA = HSLA(Angle(2.9670597283903604),Normalized(1.0),Normalized(0.5),Normalized(0.8))
```

2.3.3 Complementary Colors

A simple way to generate colors that look good together is to use *complementary colors*. Given a color, it's complement is the one opposite it on the color wheel. In other words, it has hue rotated by 180 degrees. Complementary pairs have high contrast and make for striking compositions:

Exercise: Complementary Colors

Create a method `complement` that takes a `Color` as input and returns its complement. You can use the method `spin` on a `Color` to rotate its hue by a given `Angle`.

See the solution

Exercise: Complementary Chess Boards

Using `complement` write a method `complementaryChessBoard` that creates a four-by-four chess board using a complementary color scheme. This method should take a `Color` input. Here's the method signature:

```
def complementaryChessBoard(color: Color): Image = ???
```

You should end up with a picture like the below.

See the solution

Figure 2.5: Aubergines by Estaban Cavrico CC BY-NC-ND 2.0. The green and purple of the aubergins are near complements.

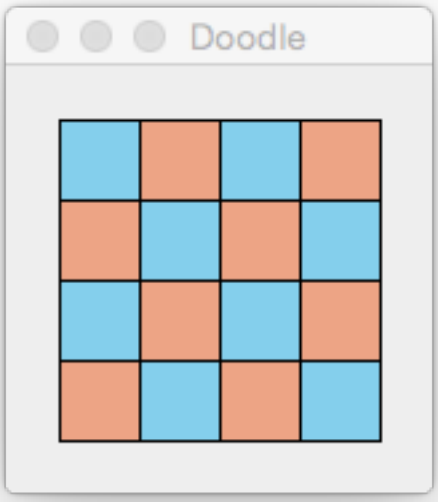

Figure 2.6: A chess board colored using complementary colors

2.3.4 Analogous Colors

Complementary colors can be quite harsh on the eyes. We can play around with saturation and lightness to decrease the contrast but ultimately this color scheme is quite limited. Let's explore another color scheme, *analogous colors*, that gives us more flexibility.

In analogous color is simply one that is close on the color wheel to a given color. We can generate an analogous color by spinning hue, say, fifteen degrees.

Exercise: Analogous Colors

Create a method `analogous` that takes a `Color` as input and returns an analogous color.

See the solution

Exercise: Analogous Chess Boards

Now create a method `analogousChessBoard` that creates a four-by-four chess board with an analogous color scheme. You should get a result like the below.

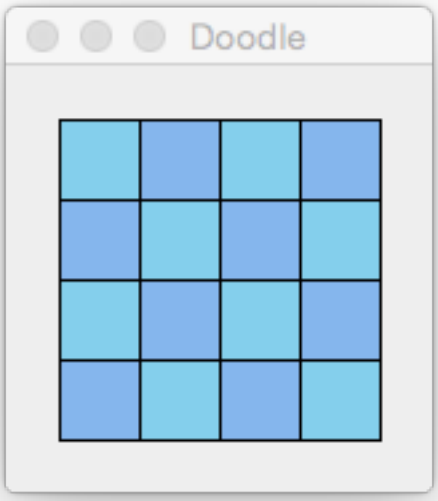

Figure 2.7: A chess board colored using analogous colors

See the solution

2.3.5 Beyond Two-Color Palettes

We have seen how we can build very simple color palettes from complementary and analogous colors. Now let's combine these ideas to build more complex palettes. A *tetrad color scheme* consists of two analogous colors and their complements.

Define a method `tetradChessBoard` that creates a chess board colored with a tetradic color scheme as illustrated. Use the following skeleton

```
def tetradChessBoard(color: Color) = ???
```

Hint: You will have to call `twoByTwo`, not `fourByFour`, within the body of `tetradChessBoard`.

See the solution

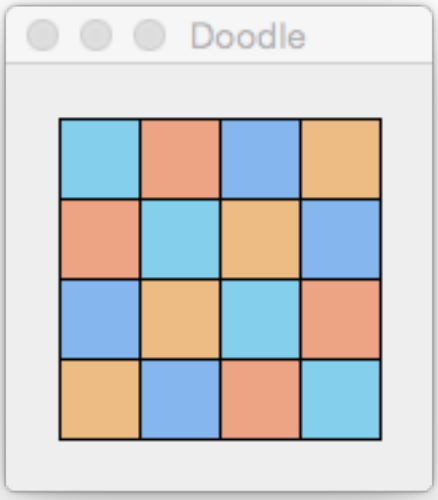

Figure 2.8: A chess board colored using a tetradic color scheme

2.4 Take Home Points

Some programs cannot be written in a single expression!

Scala provides *value declarations* as a means to capture and re-use values in the rest of our code. Because Doodle is based on immutable data structures, we can re-use single values multiple times without worrying about unintended side-effects. The chessboard example demonstrates this nicely—it re-uses the `fourByFour` value four times and the `twoByTwo` value sixteen times, resulting in a compact memory-efficient representation.

Method declarations do a different job. They allow us to *abstract over parameters*, creating blocks of code that work with a variety of inputs. Functional programming places emphasis on writing methods that *return useful values*, effectively turning methods into high-level constructors.

For example, we can view a method like `tetradChessBoard()` as a constructor for a chess board. Even though the method creates many objects internally, the substitution model allows us to ignore the implementation details and treat the method as a black box.

Chapter 3

(Functional) Programming

So far we have seen how to write literals, expressions, value declarations, and method declarations. We now have the tools to write non-trivial programs that generate complex images. In this chapter we will introduce some more tools to our functional programming toolbox and create even more complex images: recursion, first-class functions, and higher-order functions.

3.1 Recursive Algorithms

Recursion is a natural part of functional programming. The classic functional data structure—the single linked list—is recursive in nature. We can similarly create interesting drawings using recursion.

Let's start with a simple example—a set of concentric circles. We can create this image by recursing over the natural numbers. Each number corresponds to the next layer of the image:

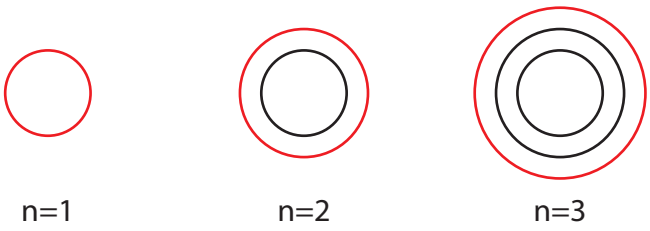

Figure 3.1: Concentric circles (n = 1 to 3)

Note the recursive pattern here:

- the `n == 1` case is the *base case*;
- every other case is the `n - 1` case (shown in black) surrounded by an extra circle (shown in red).

Given these two rules, we can generate a picture for any value of `n >= 1`. We can even model the rules directly in Scala:

```
def concentricCircles(n: Int): Image =
  if(n == 1) {
    // Return a circle
  } else {
    // Return a circle superimposed on the image from n - 1
  }
```

Exercise: Concentric circles

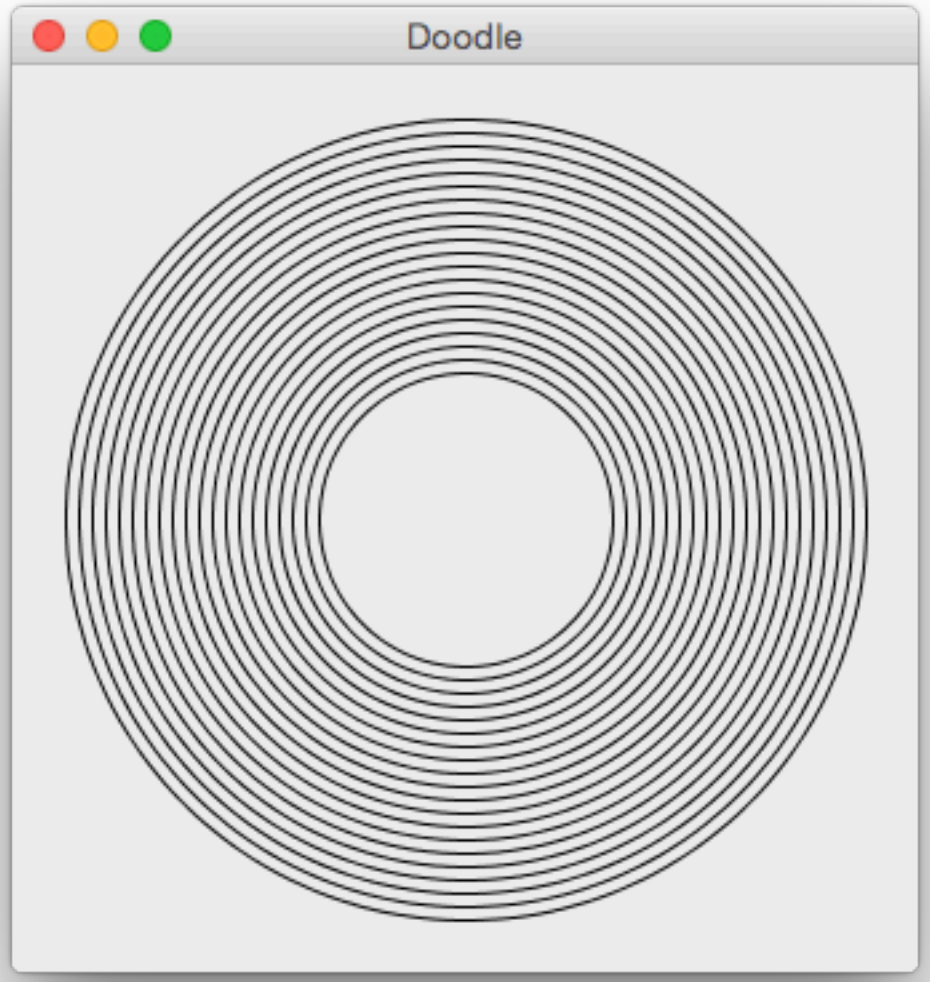

Figure 3.2: Concentric circles (n = 20)

Create an image containing 20 concentric circles using the approach described above:

For extra credit, give each circle its own hue or opacity by gradually changing the colour at each level of recursion:

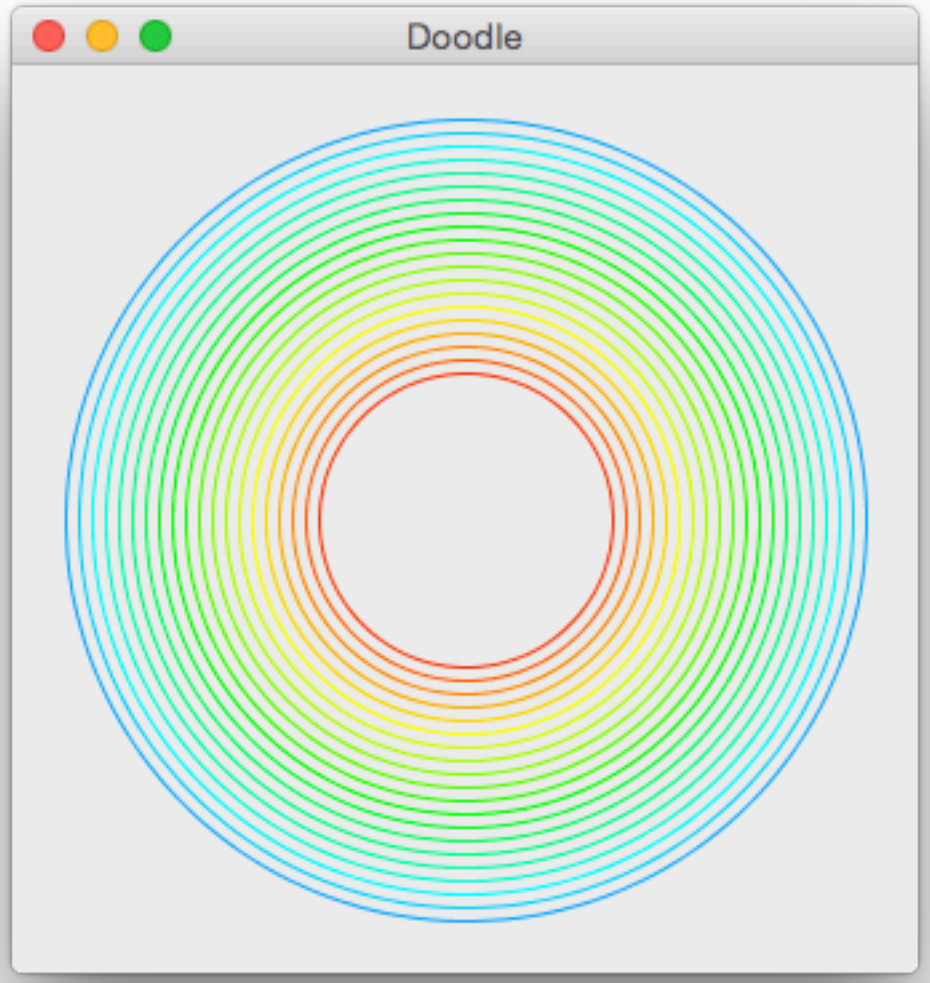

Figure 3.3: Concentric circles IN COLOUR! (n = 20)

See the solution

Exercise: Sierpinski triangle

Sierpinski triangles are a more interesting example of a recursive drawing algorithm. The pattern is illustrated below:

Here is an English description of the recursive pattern:

- The base case for `n = 1` is an equilateral triangle. We can draw this in Doodle as follows:

```
Triangle(10, 10)
```

- Every other case involves three copies of the `n - 1` case arranged in a triangle.

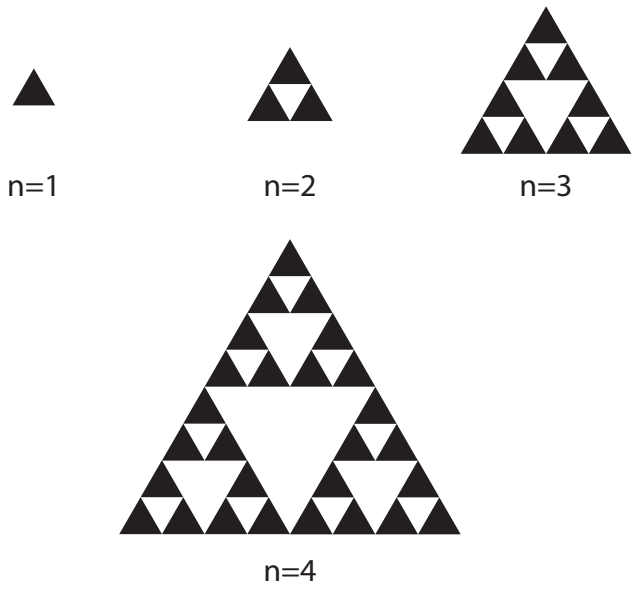

Figure 3.4: Sierpinski triangles (n = 1 to 4)

Use this description to write Scala code to draw a Sierpinski triangle. Start by dealing with the `n = 1` case, then solve the `n = 2` case, then generalise your code for any value of n. Finish by drawing the `n = 10` *Sier**pink**ski* triangle below:

You may notice that the final result is extremely large! For extra credit, rewrite your code so you can specify the size of the triangle up front:

```
def sierpinski(n: Int, size: Double): Image = ???
```

Finally, for double extra credit, answer the following questions:

1. How many pink triangles are there in your drawing?
2. How many `Triangle` objects is your code creating?
3. Is this the answer to question 2 necessarily the same as the answer to question 1?
4. If not, what is the minimum number of `Triangles` needed to draw the `n = 10` Sierpinski?

See the solution

3.2 Functions as Values

The defining feature of a functional programming programming language is the ability to define *functions that are first class values*. Scala has special syntax for functions and function types. Here's a function that calculates

```
(a: Double, b: Double) => math.sqrt(a*a + b*b)
// res0: (Double, Double) => Double = <function2>

res0(3, 4)
// res1: Double = 5.0
```

Because Scala is an object oriented language, all first class values are objects. This means functions are objects, not methods! In fact, functions themselves have useful methods for composition:

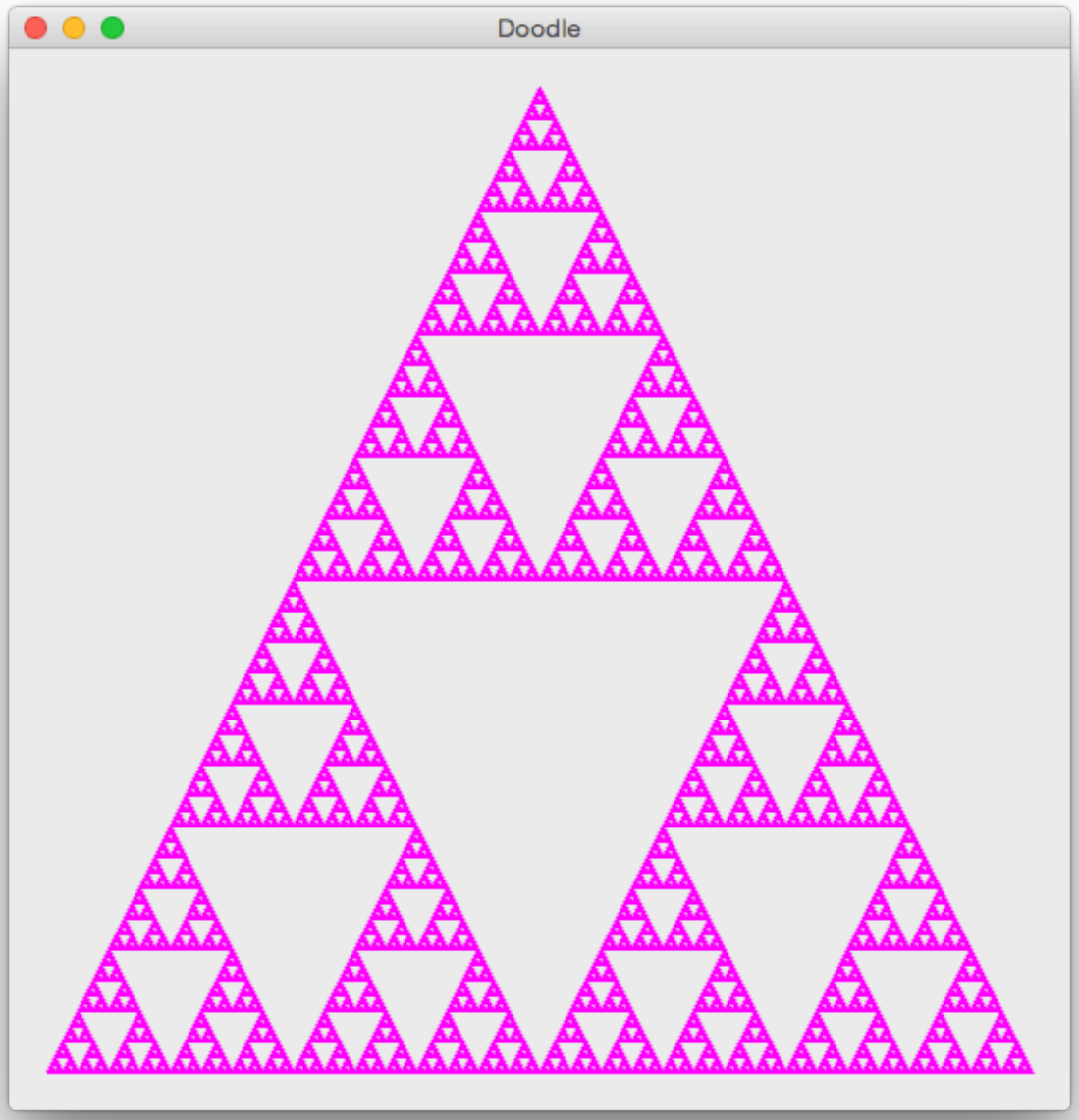

Figure 3.5: Sierpinski triangle (n = 10)

```
(a: Int) => a + 10
// res0: Int => Int = <function1>

(a: Int) => a * 2
// res1: Int => Int = <function1>

res0 andThen res1 // this composes the two functions
// res2: Int => Int = <function1>

res2(5)
// res3: Int = 30
```

It may seem surprising and restrictive that Scala methods are not values. We can prove this by attempting to refer to a method without invoking it:

```
Color.rgb
// <console>:20: error: missing arguments for method rgb in object Color;
// follow this method with `_' if you want to treat it as a partially applied function
//               Color.rgb
//                     ^
```

Fortunately, as the error message above suggests, we can convert any method to a function using the _ operator and call it with the same parameters:

```
Color.rgb _
// res4: (Int, Int, Int) => doodle.core.Color = <function3>

res4(255, 0, 0)
// res5: doodle.core.Color = ...
```

3.3 Higher Order Methods and Functions

Why are functions useful? We can already use methods to package up and name reusable fragments of code. What other advantages do we get from treating code as values?

- we can pass functions as parameters to other functions and methods;
- we can create methods that return functions as their results.

Let's consider the pattern from the concentric circles exercise as an example:

```
def manyShapes(n: Int): Image =
  if(n == 1) {
    singleShape
  } else {
    singleShape on manyShapes(n - 1)
  }

def singleShape: Image = ???
```

This pattern allows us to create many different images by changing the definition of `singleShape`. However, each time we provide a new definition of `singleShape`, we also need a new definition of `manyShapes` to go with it.

We can make `manyShapes` completely general by supplying `singleShape` as a parameter:

```
def manyShapes(n: Int, singleShape: Int => Image): Image =
  if(n == 1) {
    singleShape(n)
  } else {
    singleShape(n) on manyShapes(n - 1, singleShape)
  }
```

Now we can re-use the same definition of `manyShapes` to produce plain circles, circles of different hue, circles with different opacity, and so on. All we have to do is pass in a suitable definition of `singleShape`:

```
// Passing a function literal directly:

val blackCircles: Image =
  manyShapes(10, (n: Int) => Circle(50 + 5*n))

// Converting a method to a function:

def redCircle(n: Int): Image =
  Circle(50 + 5*n) lineColor Color.red

val redCircles: Image =
  manyShapes(10, redCircle _)
```

> **Tip**
>
> *Function Syntax*
>
> We're introducing a lot of syntax here! There's a dedicated section on function syntax in the quick reference if you get lost!

Exercise: The Colour and the Shape

Starting with the code below, write color and shape functions to produce the following image:

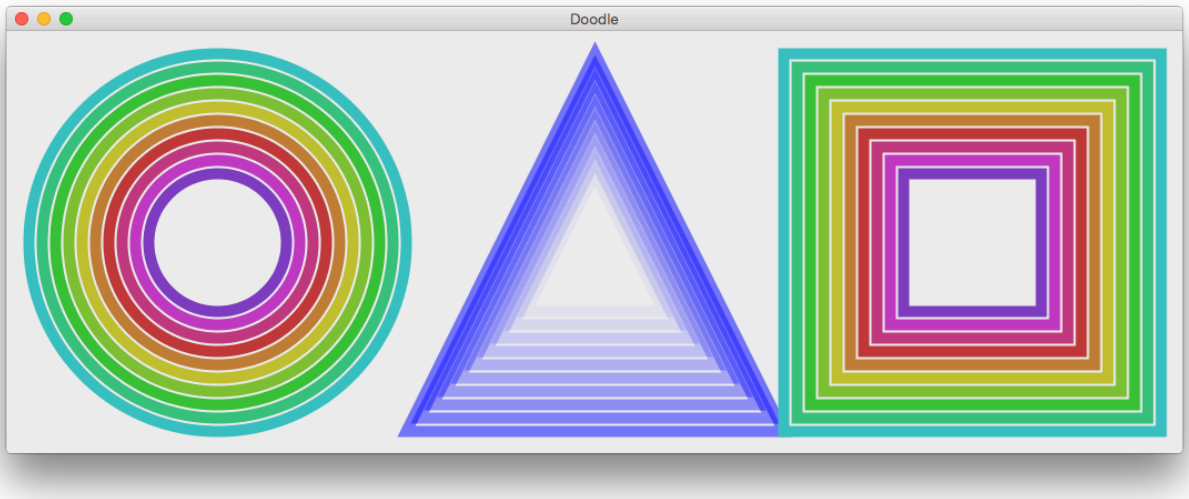

Figure 3.6: Colours and Shapes

```
def manyShapes(n: Int, singleShape: Int => Image): Image =
  if(n == 1) {
    singleShape(n)
```

```
  } else {
    singleShape(n) on manyShapes(n - 1, singleShape)
  }
```

The `manyShapes` method is equivalent to the `concentricCircles` method from previous exercises. The main difference is that we pass in the definition of `singleShape` as a parameter.

Let's think about the problem a little. We need to do two things:

1. write an appropriate definition of `singleShape` for each of the three shapes in the target image;
2. call `manyShapes` three times, passing in the appropriate definition of `singleShape` each time and putting the results `beside` one another.

Let's look at the definition of the `singleShape` parameter in more detail. The type of the parameter is `Int => Image`, which means a function that accepts an `Int` parameter and returns an `Image`. We can declare a method of this type as follows:

```
def outlinedCircle(n: Int) =
  Circle(n * 10)
```

We can pass a reference to this method to `manyShapes` to create an image of concentric black outlined circles:

```
manyShapes(10, outlinedCircle).draw
```

The rest of the exercise is just a matter of copying, renaming, and customising this function to produce the desired combinations of colours and shapes:

```
def circleOrSquare(n: Int) =
  if(n % 2 == 0) Rectangle(n*20, n*20) else Circle(n*10)

(manyShapes(10, outlinedCircle) beside manyShapes(10, circleOrSquare)).draw
```

For extra credit, when you've written your code to create the sample shapes above, refactor it so you have two sets of base functions—one to produce colours and one to produce shapes. Combine these functions using a *combinator* as follows, and use the result of the combinator as an argument to `manyShapes`

```
  def colored(shape: Int => Image, color: Int => Color): Int => Image =
    (n: Int) => ???
```

See the solution

3.4 Take Home Points

Scala is both functional and object oriented. In this chapter we were introduced to some of its functional aspects.

The most important aspect of a functional language is that *functions are first class values*. This allows us to parameterize one method or function by another, which is an enormously powerful abstraction tool that we saw in action with our revisions to the `manyShapes` method:

```
def manyShapes(n: Int, singleShape: Int => Image): Image =
  ???
```

So far we have mainly written code using recursion. However, don't be fooled into think that Scala is all about recursion. In the next chapter we will look at Scala's collections library, which provides many convenient high level transformation methods that take functions as parameters.

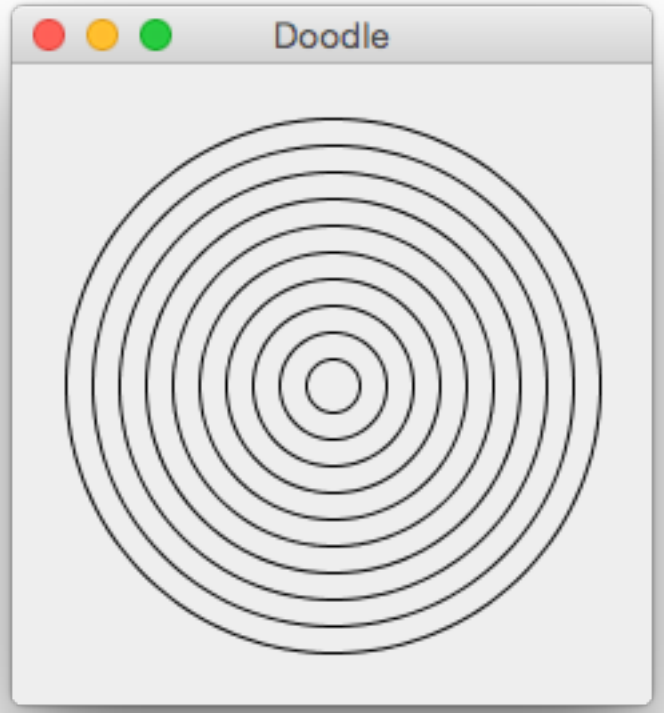

Figure 3.7: Many outlined circles

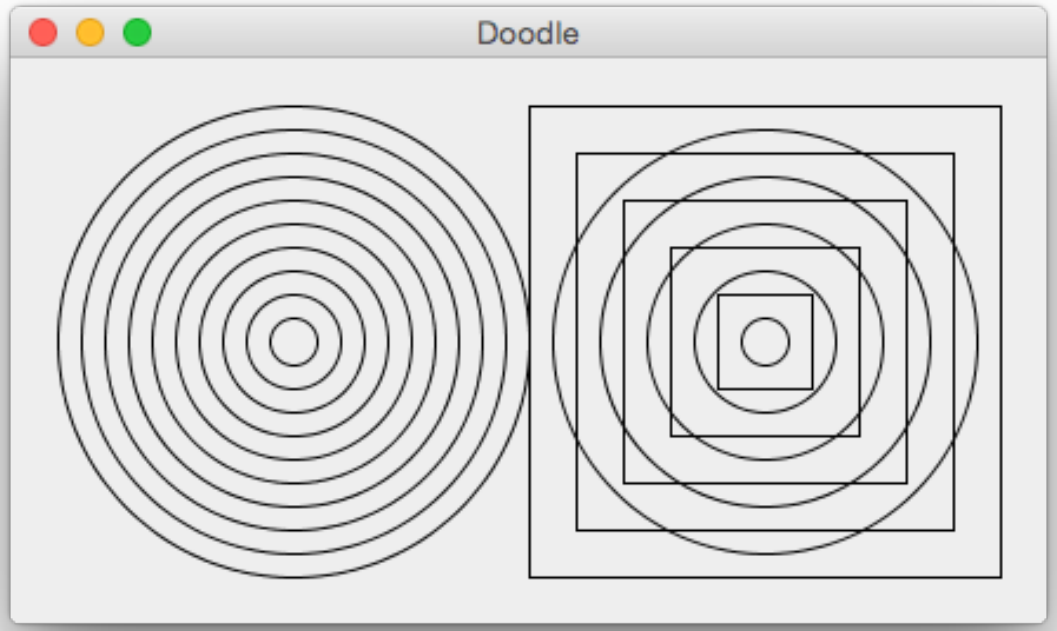

Figure 3.8: Many outlined circles beside many circles and squares

Chapter 4

Collections

An introduction to functional programming wouldn't be complete without discussing transformations on collections. In this chapter we will look at the `Lists`, which are without a doubt the best known data type in functional programming.

4.1 Creating Sequences

The standard library in Scala contains many types of sequence: mutable and immutable, lazy and eager, parallel and sequential. In this course we will use two types of sequence: `Lists` and `Ranges`. Both are simple, immutable, eager data types. Let's see them in action.

We can create a list in Scala by calling the `List` factory method as follows:

```
List(1, 2, 3, 4, 5)
// res0: List[Int] = List(1, 2, 3, 4, 5)
```

The result of the expression is of type `List[Int]`, which we read as "list of integers". If we call the factory method with `String` arguments, the type of the result changes accordingly:

```
List("a", "b", "c", "d", "e")
// res1: List[String] = List(a, b, c, d, e)
```

`Lists` are useful for storing short sequences of values. If we want to create long sequences of numbers, however, we are better off using `Ranges`. We can create these using the `until` method of `Int` or `Double`:

```
0 until 10
// res0: scala.collection.immutable.Range.Inclusive =
//   Range(0, 1, 2, 3, 4, 5, 6, 7, 8, 9)

0.0 until 5.0
// res1: scala.collection.immutable.Range.Inclusive =
//   Range(0.0, 1.0, 2.0, 3.0, 4.0)
```

`Ranges` have a `by` method that allows us to change the step between consecutive elements of the range:

```
0 until 10 by 2
// res1: scala.collection.immutable.Range.Inclusive =
//   Range(0, 2, 4, 6, 8)

0.0 until 1.0 by 0.3
// res2: scala.collection.immutable.Range.Inclusive =
//   Range(0.0, 0.3, 0.6, 0.9)
```

Many methods in Doodle are designed to work with `Lists` and `Ranges`, but you can use the `toList` of any `Range` to convert it to a `List` if you run into problems:

```
(0 until 10).toList
// res0: List[Int] =
//   List(0, 1, 2, 3, 4, 5, 6, 7, 8, 9)
```

Let's see what we can do with these sequences.

4.2 Transforming Sequences

In imperative programs we use loops to iterate over lists and transform them to produce new values. For example, here is a Java 7 method to double the numbers in an `ArrayList`:

```
List<Integer> doubleAll(List<Integer> numbers) {
  List<Integer> ans = new ArrayList<Integer>();

  for(int i : numbers) {
    ans.push(i * 2);
  }

  return ans;
}

List<Integer> result = doubleAll(new ArrayList<Integer>(1, 2, 3, 4, 5));
```

There are a lot of lines in this example, many of which aren't to do with the desired operation of doubling numbers. We have to allocate a temporary list and push numbers onto it before returning it, all of which ought to be be handled by library code.

We can't easily abstract away the temporary list allocation in Java 7 because we have no direct way of abstracting the doubling operation. In Scala and Java 8 we can represent doubling succinctly using a function literal, aka a "closure":

```
(x: Int) => x * 2
// res2: Int => Int = <function1>

res2(10)
// res3: Int = 20
```

Scala's `List` class has a method called `map` that allows us to exploit functions to remove all of the boilerplate from our Java 7 example. `map` accepts a function as a parameter and returns a new `List` created by applying the function to every item:

```
List(1, 2, 3, 4, 5).map(i => i * 2)
// res4: List[Int] = List(2, 4, 6, 8, 10)
```

We can use the `map` method to convert `Lists` of values to `Lists` of `Images`:

```
val radii = List(10, 20, 30, 40, 50)
// radii: List[Int] = List(10, 20, 30, 40, 50)

val circles = radii.map(i => Circle(i * 10))
// circles: List[doodle.core.Circle] = // ...
```

Doodle contains a handful of convenient methods to convert values of type `List[Image]` to single `Images`. One of these is `allBeside`, which lays a list of images out beside one another:

```
allBeside(circles).draw
```

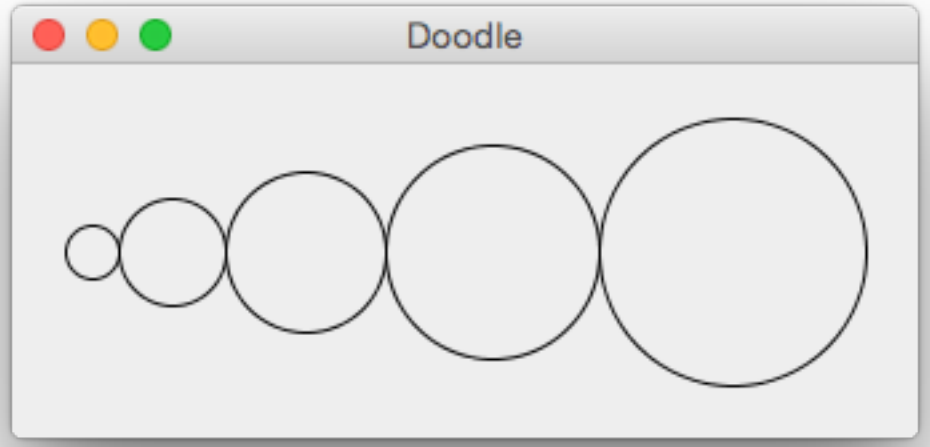

Figure 4.1: A row of circles

Other combinators are listed below:

Operator	Type	Description	Example
`allBeside(listOfImages)`	`Image`	Places images horizontally next to one another.	`allBeside(Circle(10), Circle(20))`
`allAbove(listOfImages)`	`Image`	Places images vertically above one another.	`allAbove(Circle(10), Circle(20))`
`allBelow(listOfImages)`	`Image`	Places images vertically below one another.	`allBelow(Circle(10), Circle(20))`
`allOn(listOfImages)`	`Image`	Places images centered on top of one another.	`allOn(Circle(10), Circle(20))`
`allUnder(listOfImages)`	`Image`	Places images centered underneath one another.	`allUnder(Circle(10), Circle(20))`

We can recreate our concentric circles example trivially using `allOn` or `allUnder`. Much simpler than writing a recursive method!

```
val radii = List(10, 20, 30, 40, 50)
// radii: List[Int] = List(10, 20, 30, 40, 50)

val circles = radii.map(i => Circle(i))
// circles: List[Circle] = // ...

allOn(circles).draw
```

Exercise: Colour Palette

Create an application to show the range of colours you can get in HSL space. Create a two-dimensional grid of rectangles, with hue varying from 0 to 360 degrees on the x-axis and lightness varying from 0.0 to 1.0 on the y-axis.

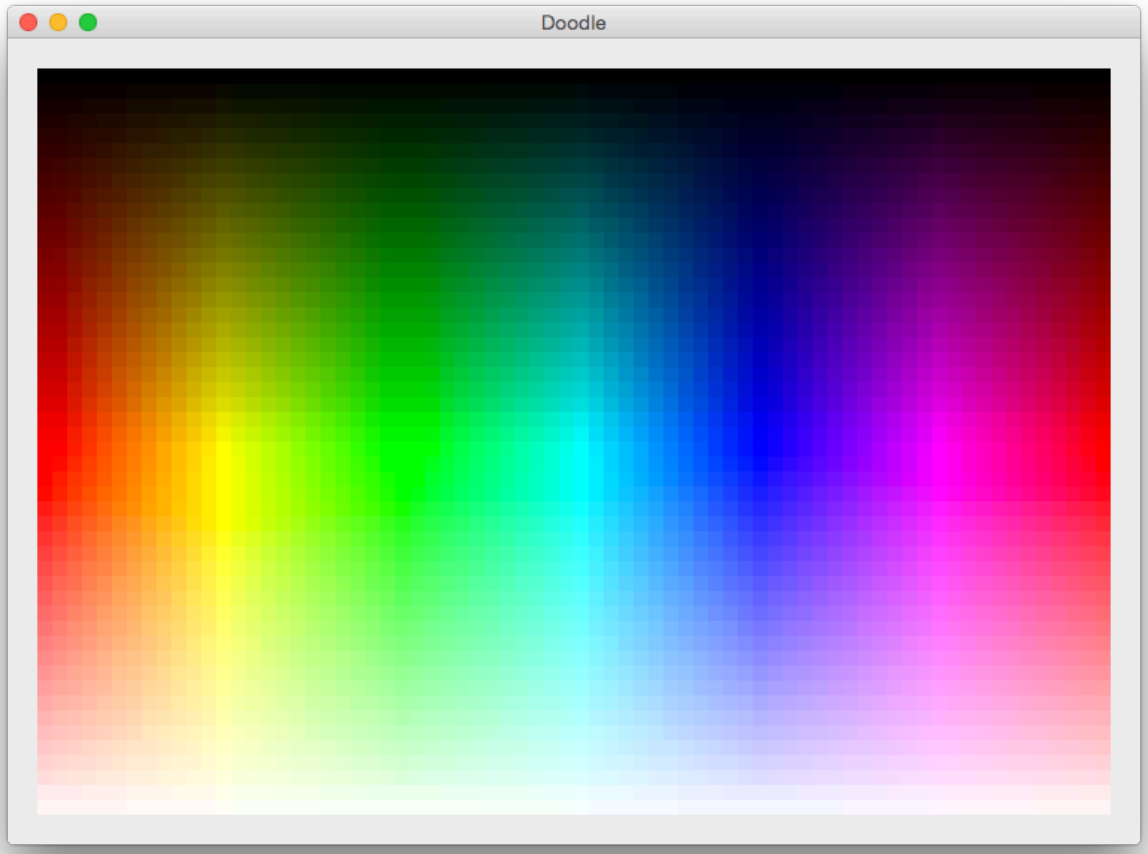

Figure 4.2: HSL Colour Palette

Here are some tips:

1. Think about the structure of your image and build it from the bottom up. Start by building a single square of constant hue and lightness, then compose a sequence of squares into a single column, then compose a sequence of columns together to form the palette.
2. You can create an HSL colour as follows:

```
val hue = 180.degrees

val lightness = 0.5.normalized

val color = Color.hsl(hue, 1.0.normalized, lightness)
```

For extra credit, allow the user to specify parameters for the step size along each axis and the basic shape used in each cell:

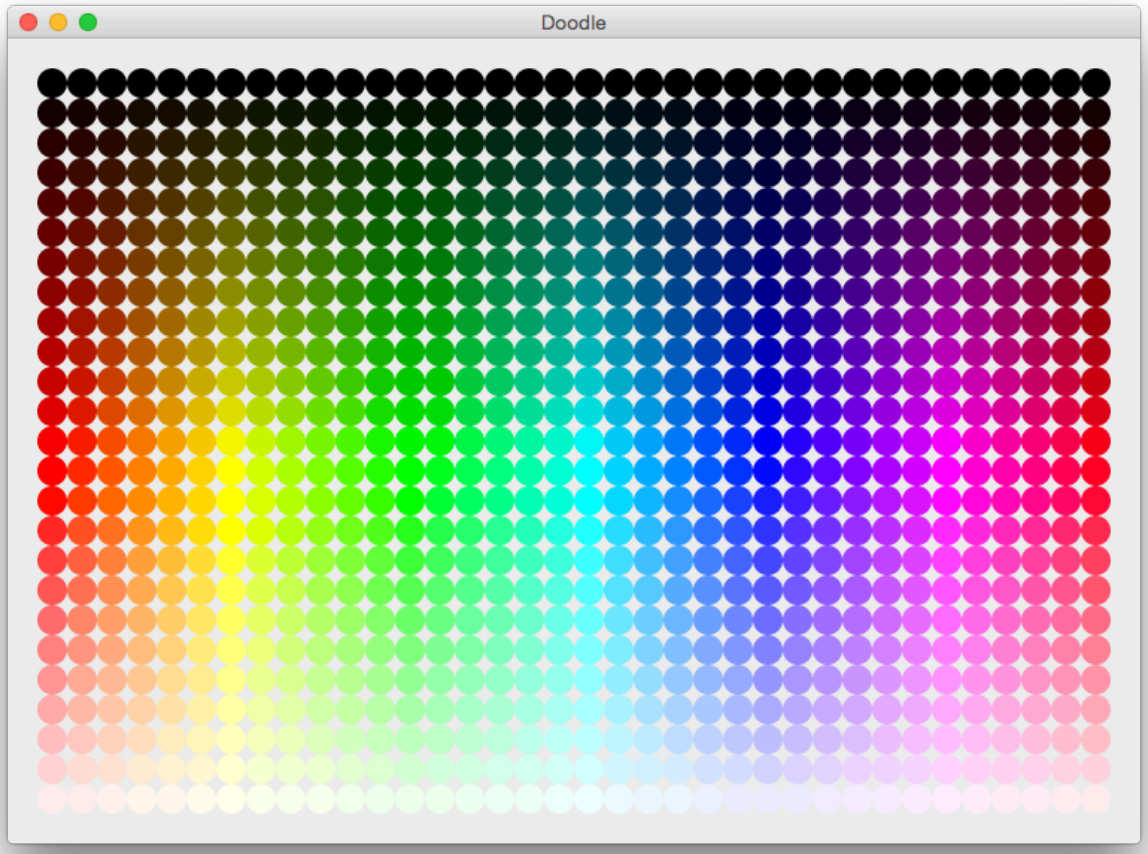

Figure 4.3: HSL Colour Palette

See the solution

There are many other methods in the Scala standard library for transforming sequences using functions as parameters. We won't cover them in this course, but here is a sample of a few really useful ones:

Start with	Method	Parameter	Result	Description
`List[A]`	`map`	`A => B`	`List[B]`	Return all elements, transformed by the function.
`List[A]`	`filter`	`A => Boolean`	`List[A]`	Return all elements for which the function returns `true`.
`List[A]`	`flatMap`	`A => List[B]`	`List[B]`	Return all elements, transformed by the function and concatenated into a single `List`.
`List[A]`	`find`	`A => Boolean`	`Option[B]`	Return the first element for which the function returns `true`.

4.3 Drawing Paths

Doodle provides another type of `Image` that makes particular use of sequences. `Paths` represent arbitrary shapes created using sequences of pen movements:

```
val image = Path(List(
  MoveTo(Vec(0, 0)),
  LineTo(Vec(100, 0)),
  LineTo(Vec(50, 100)),
  BezierCurveTo(Vec(50, 0), Vec(0, 100), Vec(0, 0))
))
// image: Path = // ...

image.draw
```

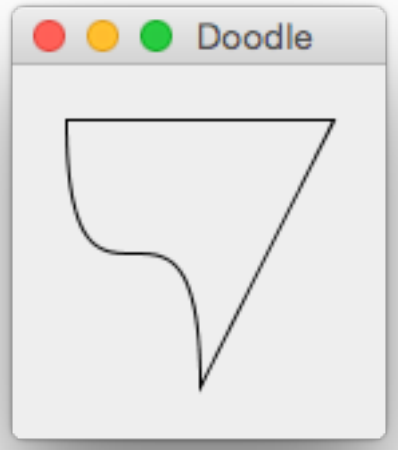

Figure 4.4: An example Path

Pen movements come in three varieties:

- `MoveTo(point)`—move the pen to `point` without drawing a line;

- `LineTo(point)`—move the pen to `point` drawing a straight line;
- `BezierCurveTo(cp1, cp2, point)`—move the pen to `point` drawing a bezier curve—`cp1` and `cp2` are "control points" determining the shape of the curve.

The arguments in each case are objects of type `Vec`, which are 2D vectors representing `x,y` points (the name `Vector` is already taken by the `scala.collection.Vector` class). There are various ways we can create and transform `Vecs`:

Code	Result	Description	Example
`Vec(num, num)`	`Vec`	Create a vector using `x,y` coordinates	`Vec(3, 4)`
`Vec.polar(angle, length)`	`Vec`	Create a vector using polar coordinates	`Vec.polar(30.degrees, 100)`
`Vec.zero`	`Vec`	A zero vector (`0,0`)	`Vec.zero`
`Vec.unitX`	`Vec`	A unit X vector (`1,0`)	`Vec.unitX`
`Vec.unitY`	`Vec`	A unit Y vector (`0,1`)	`Vec.unitY`
`vec * num`	`Vec`	Multiply `vec` by `num`	`Vec(2, 1) * 10`
`vec / num`	`Vec`	Divide `vec` by `num`	`Vec(20, 10) / 10`
`vec + vec`	`Vec`	Add vectors	`Vec(2, 1) + Vec(1, 3)`
`vec - vec`	`Vec`	Subtract vectors	`Vec(5, 5) - Vec(2, 1)`
`vec rotate angle`	`Vec`	Rotate anticlockwise by `angle`	`Vec(5, 5) rotate 45.degrees`
`vec.x`	`Double`	Get the X component of `vec`	`Vec(3, 4).x`
`vec.x`	`Double`	Get the Y component of `vec`	`Vec(3, 4).y`
`vec.length`	`Double`	Get the length of 'vec	`Vec(3, 4).length`

We can use these operations to create paths quickly by adding vectors. Notice how we start the shape with a `MoveTo` element (all paths implicitly start at the origin). This is a very common pattern.

```
val elements = (0 to 360 by 36).map { angle =>
  val point = (Vec.unitX * 100) rotate angle.degrees
  val element =
    if(angle == 0)
      MoveTo(point)
    else
      LineTo(point)
  element
}
// elements: scala.collection.immutable.IndexedSeq[doodle.core.PathElement] = // ...

val decagon = Path(elements)
// decagon: doodle.core.Path = // ...

decagon.draw
```

4.3.1 Exercise: My God, It's Full of Stars!

Let's use this pattern to draw some stars. For the purpose of this exercise let's assume that a star is a polygon with p points. However, instead of connecting each point to its neighbours, we'll connect them to the `nth` point around the circumference.

For example, the diagram below shows stars with p=11 and `n=1 to 5`. n=1 produces a regular polygon while values of n from 2 upwards produce stars with increasingly sharp points:

Write code to draw the diagram above. Start by writing a method to draw a `star` given p and n:

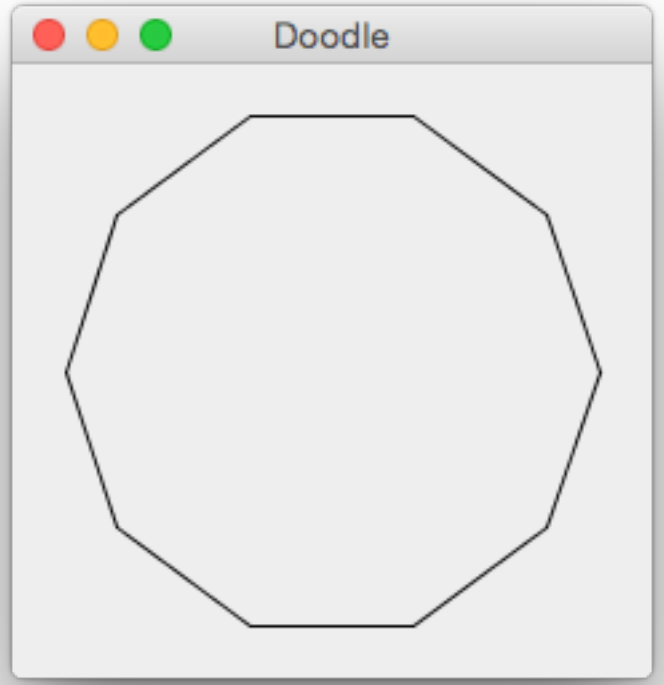

Figure 4.5: A Decagon

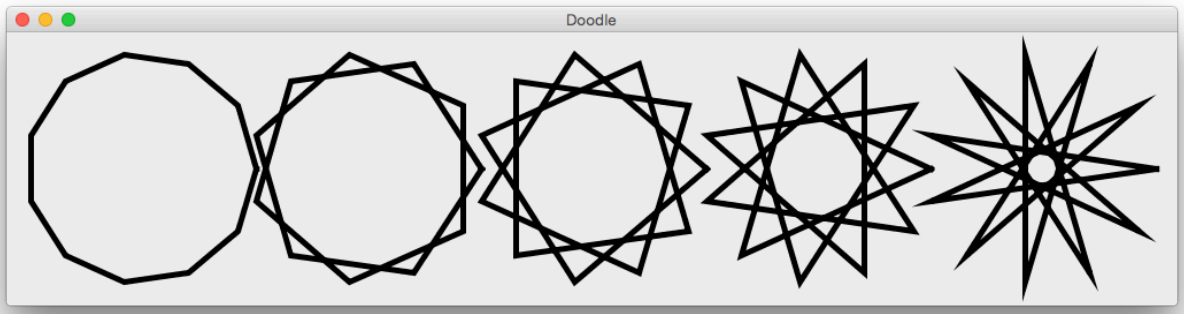

Figure 4.6: Stars with p=11 and n=1 `to` 5

```
def star(p: Int, n: Int, radius: Double): Image =
  ???
```

Create the points for your star using ranges and `Vec.polar`:

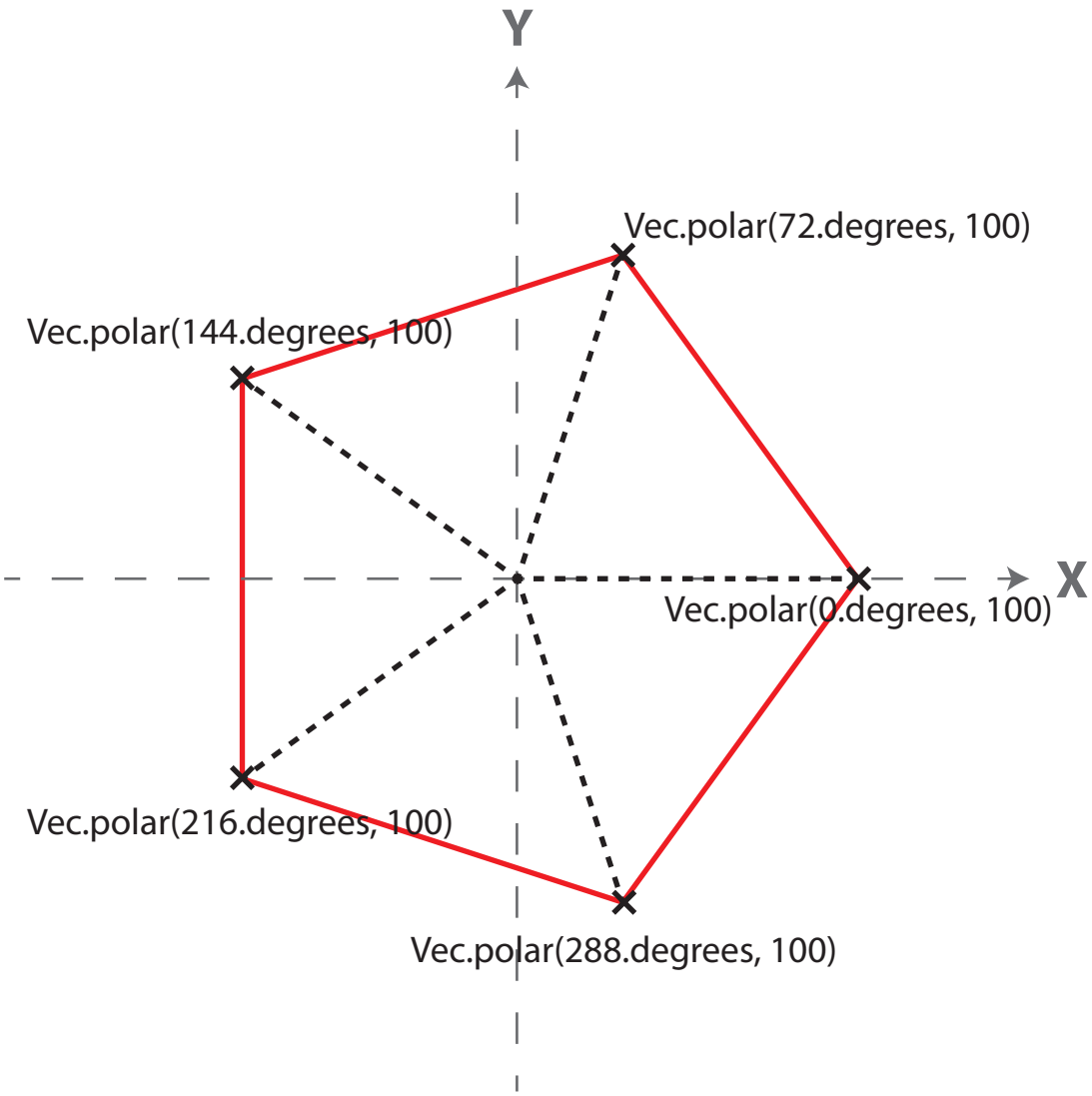

Figure 4.7: Polar coordinates on a 5 pointed polygon

Use your choice of recursion and `beside` or iteration and `allBeside` to create the row of stars.

See the solution

When you've finished your row of stars, try constructing a larger image from different values of p and n. Here's an example:

See the solution

4.4 Take Home Points

In this chapter we looked at our first Scala collections: `Lists` and `Ranges`. These are two types of *sequence* with different use cases:

- `Lists` are convenient for storing finite collections of arbitrary values of a similar type;

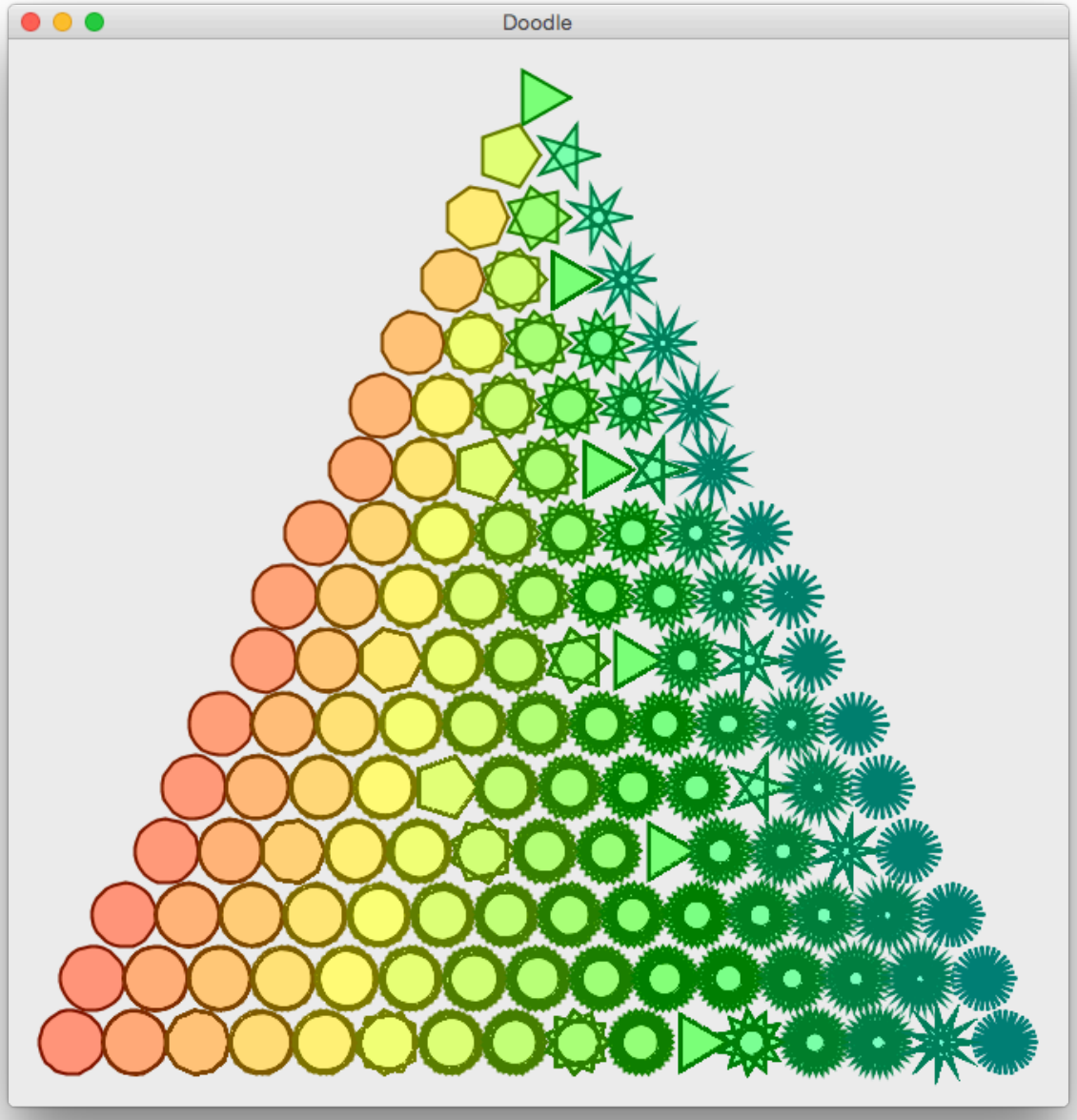

Figure 4.8: Stars with p=3 `to` 33 `by` 2 and n=1 `to` p/2

- `Ranges` provide a more compact syntax for regular sequences of numbers.

We didn't cover this in the chapter but `List` and `Range` have a common supertype, `Seq`. This explains a few things:

- `List` and `Range` both have a `map` method, the behaviour of which is the same in each case.
- Doodle's n-ary combinators—`allAbove`, `allBeside`, and so on—work equally well with sequences produced by transforming `Ranges`, as they do sequences produced by transforming `Lists`.

Scala has a large collections library containing many different useful types: sequences, maps, and sets with mutable, immutable, lazy, and parallel variations. These types form a single inheritance hierarchy that provides consistent implementations of methods such as `map`, `flatMap`, `filter`, `find`, and so on.

We looked at one kind of transform operation in this chapter: the `map` method. `map` applies a user-specified function to every item in a sequence, returning a new sequence of the results:

```
List(1, 2, 3).map(x => x * 2)
// res0: List[Int] = List(2, 4, 6)
```

The key point about `map` is that it only makes sense in a world where we have functions that are also first class values. The same can be said for many of the other cohort of combinators such as `filter`, `find`, and `flatMap`. First class functions allow us to pass operations to these methods as parameters, separating user code from the implementational detail of allocating temporary buffers.

Because many transformation methods return sequences, we can chain calls to perform complex transformations in a series of simple steps. Here is an example that demonstrates the power of thie approach:

```
// Print all even numbers from 1 to 100 that are also divisible by 3:
(1 to 50).toList.
  map(x => x * 2).
  filter(x => x % 3 == 0).
  foreach(println)
// 6
// 12
// 18
// etc...
```

The structure of this computation looks similar to the structure of our Doodle programs: we build an intermediate representation by creating, combining, and transforming primitives, and perform side-effects at the end of the program. Obviously we've put less thought into this program than into Doodle, but it's interesting to note that even ad hoc functional programs can follow the same basic structure.

Chapter 5

Summary

In this text we have covered a handful of the essential functional programming tools available in Scala.

5.1 Representations and Interpreters

We started by writing expressions to create and compose images. Each program we wrote went through two distinct phases:

1. Build an `Image`
2. Call the `draw` method to display the image

This process demonstrates two important functional programming patterns: *building intermediate representations* of the result we want, and *interpreting the representations* to produce output.

5.2 Abstraction

Building an intermediate representation allows us to only model the aspects of the result that we consider important and *abstract* irrelevant details.

For example, Doodle directly represents the primitive shapes and geometric relationships in our drawings, without worrying about implementation details such as screen coordinates. This keeps our code clear and maintainable, and limits the number of "magic numbers" we need to write. For example, it is a lot easier to determine that this Doodle program produces a house:

```
def myImage: Image =
  Triangle(50, 50) above Rectangle(50, 50)
// myImage: Image = // ...
```

than this implementation in Java2D:

```
def drawImage(g: Graphics2D): Unit = {
  g.setStroke(new BasicStroke(1.0f))
  g.setPaint(new Color(0, 0, 0))
  val path = new Path2D.Double()
  path.moveTo(25, 0)
  path.lineTo(50, 50)
  path.lineTo(0, 50)
  path.lineTo(25, 0)
  path.closePath()
```

```
  g.draw(path)
  f.drawRect(50, 50, 50, 50)
}
```

It's important to realise that all of the imperative Java2D code is still present in Doodle. The difference is we have hidden it away into the `draw` method. `draw` acts as *interpreter* for our `Images`, filling in all of the details about coordinates, paths, and graphics contexts that we don't want to think about in our code.

Separating the immediate value and the interpreter also allows us to change how interpretation is performed. Doodle already comes with two interpreters, one of which draws in the Java2D framework while the other draws in the HTML canvas. You can image yet more interpreters to, for example, achieve artistic effects such as drawing images in a hand-drawn style.

5.3 Composition

In addition to making our programs clearer, the functional approach employed by Doodle allows us to *compose* images from other images. For example, we can re-use our house to draw a street:

```
val house = Triangle(50, 50) above Rectangle(50, 50)
// house: Image = // ...

val street = house beside house beside house
// street: Image = // ...
```

The `Image` and `Color` values we create are immutable so we can easily re-use a single `house` three times within the same image.

This approach allows us to break down a complex image into simpler parts that we then combine together to create the desired result.

Reusing immutable data, a technique called *structure sharing*, is the basis of many fast, memory efficient immutable data structures. The quintissential example in Doodle is the Sierpinski triangle where we re-used a single `Triangle` object to represent an image containing nearly 20,000 distinct coloured triangles.

5.4 Expression-Oriented Programming

Scala provides convenient syntax to simplify creating data structures in a functional manner. Constructs such as conditionals, loops, and blocks are *expressions*, allowing us to write short method bodies without declaring lots of intermediate variables. We quickly adopt a pattern of writing short methods whose main purpose is to return a value, so omitting the `return` keyword is also a useful shorthand.

5.5 Types are a Safety Net

Scala's type system helps us by checking our code. Every expression has a type that is checked at compile time to see if it matches up with its surroundings. We can even define our own types with the explicit purpose of stopping ourselves from making mistakes.

A simple example of this is Doodle's `Angle` type, which prevents us confusing numbers and angles, and degrees and radians:

```
90
// res0: Int = 90

90.degrees
// res1: doodle.core.Angle = Angle(1.5707963267948966)

90.radians
// res2: doodle.core.Angle = Angle(2.0354056994857643)

90.degrees + 90.radians
// res3: doodle.core.Angle = Angle(3.606202026280661)

90 + 90.degrees
// <console>:20: error: overloaded method value + with alternatives:
//   (x: Double)Double <and>
//   (x: Float)Float <and>
//   (x: Long)Long <and>
//   (x: Int)Int <and>
//   (x: Char)Int <and>
//   (x: Short)Int <and>
//   (x: Byte)Int <and>
//   (x: String)String
// cannot be applied to (doodle.core.Angle)
//              90 + 90.degrees
//                   ^
```

5.6 Functions as Values

We spent a lot of time writing methods to produce values. Methods let us abstract over parameters. For example, the method below abstracts over colours to produce different coloured dots:

```
def dot(color: Color): Image =
  Circle(10) lineWidth 0 fillColor color
// dot: Color => Image = // ...
```

Coming from object oriented languages, methods are nothing special. More interesting is Scala's ability to turn methods into *functions* that can be passed around as values:

```
def spectrum(shape: Color => Image): Image =
  shape(Color.red) beside shape(Color.blue) beside shape(Color.green)
// spectrum: (Color => Image) => Image = // ...

spectrum(dot)
// res0: Image = // ...
```

We wrote a number of programs that used functions as values, but the quintissential example was the `map` method of `List`. In the Collections chapter we saw how `map` lets us transform sequences without allocating and pushing values onto intermediate buffers:

```
List(1, 2, 3).map(x => x * 2)
// res0: List[Int] = List(2, 4, 6)
```

Functions, and their first class status as values, are hugely important for writing simple, boilerplate-free code.

5.7 Final Words

The intention of this book has been to introduce you to the functional parts of Scala. These are what differentiate Scala from older commercial languages such as Java and C. However, this is only part of Scala's story. Many modern languages support functional programming, including Ruby, Python, Javascript, and Clojure. How does Scala relate to these languages, and why would you want to choose it over the other available options?

Perhaps the most significant draw to Scala is its type system. This distinguishes Scala from popular languages such as Ruby, Python, Javascript, and Clojure, which are dynamically typed. Having static types in a language is undeniably a trade-off—writing code is slower because we have to satisfy the compiler at every stage. However, once our code compiles we gain confidence about its quality.

Another major draw is Scala's blending of object-oriented and functional programming paradigms. We saw a little of this in the first chapter—every value is an object with methods, fields, and a class (its type). However, we haven't created any of our own data types in this book. Creating types is synonymous with declaring classes, and Scala supports a full gamut of features such as classes, traits, interitance, and generics.

Finally, a major benefit of Scala is its compatibility with Java. In many ways Scala can be seen as a superset of Java, and interoperation between the two languages is quite straightforward. This opens up a world of Java libraries to our Scala applications, and allows flexibility when translating Java applications to Scala.

5.8 Next Steps

We hope you enjoyed Creative Scala and drawing diagrams with Doodle. If you would like to learn more about Scala, we recommend that you pick one of the many great books available on the language.

Our own book, Essential Scala, is available from our web site and continues Creative Scala's approach of teaching Scala by discussing and demonstrating core design patterns and the benefits they offer.

If you want to challenge yourself, try drawing something more complex with Doodle and sharing it with us via Gitter. There are lots of things you can try—check the `examples` directory in the Doodle codebase for some suggestions:

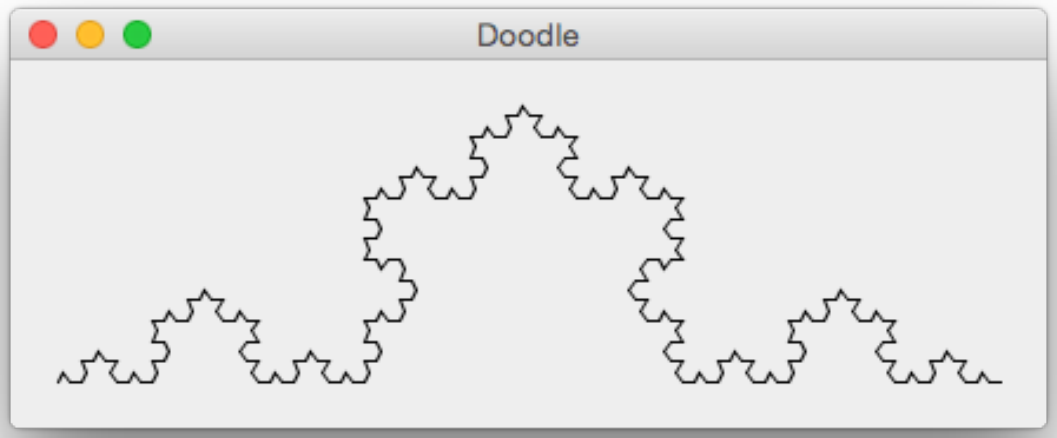

Figure 5.1: Koch Triangle (Koch.scala)

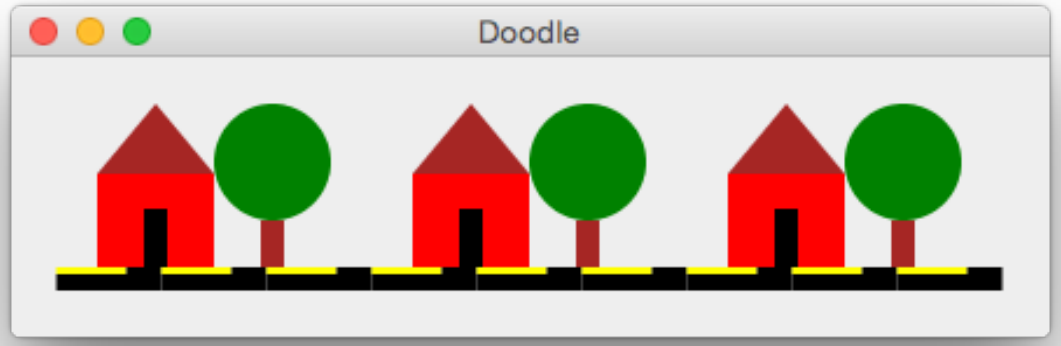

Figure 5.2: Suburban Scene (Street.scala)

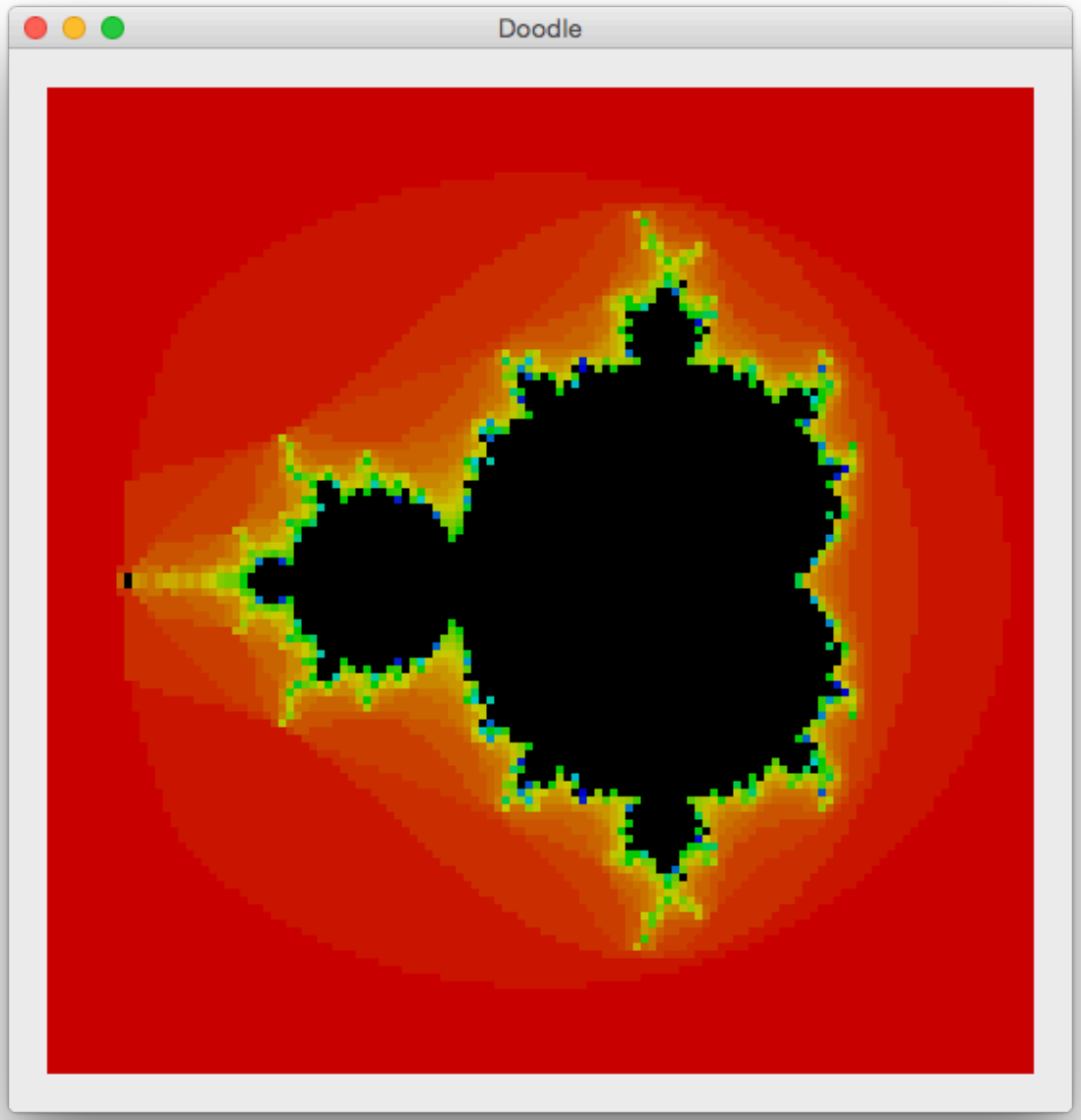

Figure 5.3: Mandelbrot Fractal by Mat Moore (Mandelbrot.scala)

Appendix A

Syntax Quick Reference

A.1 Literals and Expressions

```
// Literals:
123      // Int
123.0    // Double
"Hello!" // String
true     // Boolean

// Math:
10 + 2   // Int + Int    = Int
10 + 2.0 // Int + Double = Double
10 / 2   // Int / Int    = Double

// Boolean logic:
true && false // logical AND
true || false // logical OR
!true         // logical NOT

// String concatenation:
"abc" + "def" // String
"abc" + 123   // auto-conversion from Int to String

// Method calls and infix operators:
1.+(2)    // method call style
1 + 2     // infix operator style
1 + 2 + 3 // equivalent to 1.+(2).+(3)

// Conditionals:
if(booleanExpression) expressionA else expressionB

// Blocks:
{
  sideEffectExpression1
  sideEffectExpression2
  resultExpression
}
```

A.2 Value and Method Declarations

```
// Value declaration syntax:
val valueName: SomeType = resultExpression // declaration with explicit type
val valueName = resultExpression           // declaration with inferred type

// Method with parameter list and explicit return type:
def methodName(argName: ArgType, argName: ArgType): ReturnType =
  resultExpression

// Method with parameter list and inferred return type:
def methodName(argName: ArgType, argName: ArgType) =
  resultExpression

// Multi-expression method (using a block):
def methodName(argName: ArgType, argName: ArgType): ReturnType = {
  sideEffectExpression1
  sideEffectExpression2
  resultExpression
}

// Method with no parameter list:
def methodName: ReturnType =
  resultExpression

// Calling a method that has a parameter list:
methodName(arg, arg)

// Calling a method that has no parameter list:
methodName
```

A.3 Functions as Values

Function values are written `(argName: ArgType, ...) => resultExpression`:

```
val double = (num: Int) => num * 2
// double: Int => Int = <function1>

val sum = (a: Int, b: Int) => a + b
sum: (Int, Int) => Int = <function2>
```

Multi-line functions are written using block expressions:

```
val printAndDouble = (num: Int) => {
  println("The number was " + num)
  num * 2
}
// printAndDouble: Int => Int = <function1>

scala> printAndDouble(10)
// The number was 10
// res0: Int = 20
```

We have to write function types when declaring parameters and return types. The syntax is `ArgType => ResultType` or `(ArgType, ...) => ResultType`:

```
def doTwice(value: Int, func: Int => Int): Int =
  func(func(value))
// doTwice: (value: Int, func: Int => Int)Int

doTwice(1, double)
// res0: Int = 4
```

Function values can be written inline as normal expressions:

```
doTwice(1, (num: Int) => num * 10)
// res1: Int = 100
```

We can sometimes omit the argument types, assuming the compiler can figure things out for us:

```
doTwice(1, num => num * 10)
// res2: Int = 100
```

A.4 Doodle Reference Guide

A.4.1 Imports

```
// These imports get you everything you need:
import doodle.core._
import doodle.syntax._
```

A.4.2 Creating Images

```
// Primitive images (black outline, no fill):
val i: Image = Circle(radius)
val i: Image = Rectangle(width, height)
val i: Image = Triangle(width, height)

// Compound images written using operator syntax:
val i: Image = imageA beside imageB // horizontally adjacent
val i: Image = imageA above  imageB // vertically adjacent
val i: Image = imageA below  imageB // vertically adjacent
val i: Image = imageA on     imageB // superimposed
val i: Image = imageA under  imageB // superimposed

// Compound images written using method call syntax:
val i: Image = imageA.beside(imageB)
// etc...
```

A.4.3 Styling Images

```
// Styling images written using operator syntax:
val i: Image = image fillColor color   // new fill color (doesn't change line)
val i: Image = image lineColor color   // new line color (doesn't change fill)
val i: Image = image lineWidth integer // new line width (doesn't change fill)
val i: Image = image fillColor color lineColor otherColor // new fill and line
```

```
// Styling images using method call syntax:
val i: Image = imageA.fillColor(color)
val i: Image = imageA.fillColor(color).lineColor(otherColor)
// etc...
```

A.4.4 Colours

```
// Basic colors:
val c: Color = Color.red                       // predefined colors
val c: Color = Color.rgb(255.uByte, 127.uByte, 0.uByte)          // RGB color
val c: Color = Color.rgba(255.uByte, 127.uByte, 0.uByte, 0.5.normalized)    // RGBA color
val c: Color = Color.hsl(15.degrees, 0.25.normalized, 0.5.normalized)       // HSL color
val c: Color = Color.hsla(15.degrees, 0.25.normalized, 0.5.normalized, 0.5.normalized) // HSLA color

// Transforming/mixing colors using operator syntax:
val c: Color = someColor spin       10.degrees     // change hue
val c: Color = someColor lighten    0.1.normalized // change brightness
val c: Color = someColor darken     0.1.normalized // change brightness
val c: Color = someColor saturate   0.1.normalized // change saturation
val c: Color = someColor desaturate 0.1.normalized // change saturation
val c: Color = someColor fadeIn     0.1.normalized // change opacity
val c: Color = someColor fadeOut    0.1.normalized // change opacity

// Transforming/mixing colors using method call syntax:
val c: Color = someColor.spin(10.degrees)
val c: Color = someColor.lighten(0.1.normalized)
// etc...
```

A.4.5 Paths

```
// Create path from list of PathElements:
val i: Image = Path(List(
  MoveTo(Vec(0, 0)),
  LineTo(Vec(10, 10))
))

// Create path from other sequence of PathElements:
val i: Image = Path(
  (0 until 360 by 30) map { i =>
    LineTo(Vec.polar(i.degrees, 100))
  }
)

// Types of element:
val e1: PathElement = MoveTo(toVec)                        // no line
val e2: PathElement = LineTo(toVec)                        // straight line
val e3: PathElement = BezierCurveTo(cp1Vec, cp2Vec, toVec) // curved line

// NOTE: If the first element isn't a MoveTo,
//       it is converted to one
```

A.4.6 Angles and Vecs

```
val a: Angle = 30.degrees                // angle in degrees
val a: Angle = 1.5.radians               // angle in radians
val a: Angle = math.Pi.radians           // π radians
val a: Angle = 1.turns                   // angle in complete turns

val v: Vec = Vec.zero                    // zero vector (0,0)
val v: Vec = Vec.unitX                   // unit x vector (1,0)
val v: Vec = Vec.unitY                   // unit y vector (0,1)

val v: Vec = Vec(3, 4)                   // vector from cartesian coords
val v: Vec = Vec.polar(30.degrees, 5)    // vector from polar coords
val v: Vec = Vec(2, 1) * 10              // multiply length
val v: Vec = Vec(20, 10) / 10            // divide length
val v: Vec = Vec(2, 1) + Vec(1, 3)       // add vectors
val v: Vec = Vec(5, 5) - Vec(2, 1)       // subtract vectors
val v: Vec = Vec(5, 5) rotate 45.degrees // rotate counterclockwise

val x: Double = Vec(3, 4).x              // x coordinate
val y: Double = Vec(3, 4).y              // y coordinate
val a: Angle  = Vec(3, 4).angle          // counterclockwise from (1, 0)
val l: Double = Vec(3, 4).length         // length
```

Appendix B

Solutions to Exercises

B.1 Expressions, Values, and Types

B.1.1 Solution to: Layout

The simplest solution is to create three concentric circles using the on operator:

```
(Circle(10) on Circle(20) on Circle(30)).draw
```

For the extra credit we can create a stand using two rectangles:

```
(
  Circle(10) on
  Circle(20) on
  Circle(30) above
  Rectangle(6, 20) above
  Rectangle(20, 6)
).draw
```

Return to the exercise

B.1.2 Solution to: Colour

The trick here is using parentheses to control the order of composition. The `fillColor()`, `lineColor()`, and `lineWidth()` methods apply to a single image—we need to make sure that image comprises the correct set of shapes:

```
(
  ( Circle(10) fillColor Color.red ) on
  ( Circle(20) fillColor Color.white ) on
  ( Circle(30) fillColor Color.red lineWidth 2 ) above
  ( Rectangle(6, 20) above Rectangle(20, 6) fillColor Color.brown ) above
  ( Rectangle(80, 25) lineWidth 0 fillColor Color.green )
).draw
```

Return to the exercise

B.2 Declarations

B.2.1 Solution to: Value Declarations

An 8x8 chess board can be decomposed into four 4x4 boards, each consisting four 2x2 boards, each consisting four squares:

```
val blackSquare = Rectangle(30, 30) fillColor Color.black
val redSquare   = Rectangle(30, 30) fillColor Color.red

val twoByTwo =
  (redSquare   beside blackSquare) above
  (blackSquare beside redSquare)

val fourByFour =
  (twoByTwo beside twoByTwo) above
  (twoByTwo beside twoByTwo)

val chessBoard =
  (fourByFour beside fourByFour) above
  (fourByFour beside fourByFour)
```

This is significantly clearer and more compact than creating the whole board in one expression:

```
val b = Rectangle(30, 30) fillColor Color.black
val r = Rectangle(30, 30) fillColor Color.red

val chessBoard =
  (r beside b beside r beside b beside r beside b beside r beside b) above
  (r beside b beside r beside b beside r beside b beside r beside b) above
  (r beside b beside r beside b beside r beside b beside r beside b) above
  (r beside b beside r beside b beside r beside b beside r beside b) above
  (r beside b beside r beside b beside r beside b beside r beside b) above
  (r beside b beside r beside b beside r beside b beside r beside b) above
  (r beside b beside r beside b beside r beside b beside r beside b) above
  (r beside b beside r beside b beside r beside b beside r beside b)
```

Return to the exercise

B.2.2 Solution to: Method Declarations

The structure of `fourByFour` is identical to `twoByTwo` except that we use `twoByTwo` to construct the squares we build the board.

```
def fourByFour(color1: Color, color2: Color): Image = {
  val square = twoByTwo(color1, color2)

  (square beside square) above
  (square beside square)
}
```

Return to the exercise

B.2.3 Solution to: Complementary Colors

```
def complement(color: Color): Color =
  color.spin(180.degrees)
```

Return to the exercise

B.2.4 Solution to: Complementary Colors Part 2

We can build the method using the methods we have already created.

```
def complementaryChessBoard(color: Color) =
  fourByFour(color, complement(color))
```

Return to the exercise

B.2.5 Solution to: Analogous Colors

```
def analogous(color: Color): Color =
  color.spin(15.degrees)
```

Return to the exercise

B.2.6 Solution to: Analogous Colors Part 2

This follows the same pattern as `complementaryChessBoard`. Notice how we build big things (a colored chess board) out of smaller component parts. This idea of composing small pieces of code into larger pieces is one of the key ideas in functional programming.

```
def analogousChessBoard(color: Color) =
  fourByFour(color, analogous(color))
```

Return to the exercise

B.2.7 Solution to: Beyond Two-Color Palettes

It would be nice to have a method for creating an entire tetradic color scheme from a single color, but we don't currently have a way of wrapping up a collection of data so that we could return all four values from the methods. We'll see ways of doing this later, when we introduce classes and collections.

```
def tetradChessBoard(color: Color) = {
  val color1 = color
  val color2 = analogous(color)
  val color3 = complement(color)
  val color4 = complement(color2)

  val square1 = twoByTwo(color1, color3)
  val square2 = twoByTwo(color2, color4)

  (square1 beside square2) above
  (square2 beside square1)
}
```

Return to the exercise

B.3 (Functional) Programming

B.3.1 Solution to: Recursive Algorithms

The basic structure of our solution involves two methods: one for drawing a single circle and one for drawing n circles:

```
def singleCircle(n: Int): Image =
  ???

def concentricCircles(n: Int): Image =
  if(n == 1) {
    singleCircle(n)
  } else {
    singleCircle(n) on concentricCircles(n - 1)
  }

concentricCircles(20)
```

There is a clean division of labour here: `concentricCircles` handles the recursion through values of n and the composition of the shapes at each level, while `singleCircle` decides which actual shapes we draw at each level.

Here is the implementation of `singleCircle` we need to draw monochrome circles. We calculate an appropriate radius from the value of n provided. The `n = 1` circle has radius 55 and each successive circle is 5 pixels larger:

```
def singleCircle(n: Int): Image =
  Circle(50 + 5 * n)
```

To create multicolour circles, all we need to do is modify `singleCircle`. Here is an implementation for the extra credit example above:

```
def singleCircle(n: Int): Image =
  Circle(50 + 5 * n) lineColor (Color.red spin (n * 10).degrees)
```

Here is another implementation that fades out the further we get from `n = 1`:

```
def singleCircle(n: Int): Image =
  Circle(50 + 5 * n) fadeOut (n / 20).normalized
```

We can make countless different images by tweaking `singleCircle` without changing the definition of `concentricCircles`. In fact, `concentricCircles` doesn't care about circles at all! A more general naming system would be more suitable:

```
def singleShape(n: Int): Image =
  ???

def manyShapes(n: Int): Image =
  if(n == 1) singleShape(n) else (singleShape(n) on manyShapes(n - 1))
```

Return to the exercise

B.3.2 Solution to: Recursive Algorithms Part 2

The simple solution looks like the following:

```
def triangle: Image =
  Triangle(1, 1) lineColor Color.magenta

def sierpinski(n: Int): Image =
  if(n == 1) {
    triangle
  } else {
    val smaller = sierpinski(n - 1)
    smaller above (smaller beside smaller)
  }

sierpinski(10)
```

As we hinted above, each successive triangle in the Sierpinski pattern is twice the size of its predecessor. Even if we start with an `n = 1` triangle of side 1, we end up with an `n = 10` triangle of side 1024!

The extra credit solution involves specifying the desired size up front and dividing it by two each time we recurse:

```
def triangle(size: Double): Image =
  Triangle(size, size) lineColor Color.magenta

def sierpinski(n: Int, size: Double): Image =
  if(n == 1) {
    triangle(size)
  } else {
    val smaller = sierpinski(n - 1, size / 2)
    smaller above (smaller beside smaller)
  }

sierpinski(10, 512)
```

Finally let's look at the questions:

First, let's consider the number of pink triangles. There is one triangle in the `n = 1` base Sierpinski, and we multiply this by 3 for each successive level of recursion. There are `3^9 = 19,683` triangles in the `n = 10` Sierpinski!

That's a lot of triangles! Now let's consider the number of `Triangle` objects. This question is designed to highlight a nice property of immutable data structures called *structural sharing*. Each Sierpinski from `n = 2` upwards is created from three smaller Sierpinskis, but they *don't have to be different objects in memory*. We can re-use a single smaller Sierpinski three times to save on computation time and memory use.

The code above actually shows the optimal case. We use a temporary variable, `smaller`, to ensure we only call `sierpinski(n - 1)` once at each level of recursion. This means we only call `triangle()` once, no matter what value of `n` we start with.

We only need to create one `Triangle` object for the whole picture! Of course, the `draw` method has to process this single triangle 19,683 times to draw the picture, but the representation we build to begin with is extremely efficient.

Return to the exercise

B.3.3 Solution to: Higher Order Methods and Functions

The simplest solution is to define three `singleShapes` as follows:

```
def manyShapes(n: Int, singleShape: Int => Image): Image =
  if(n == 1) {
    singleShape(n)
  } else {
    singleShape(n) on manyShapes(n - 1, singleShape)
```

```
  }

def rainbowCircle(n: Int) = {
  val color = Color.blue desaturate 0.5.normalized spin (n * 30).degrees
  val shape = Circle(50 + n*12)
  shape lineWidth 10 lineColor color
}

def fadingTriangle(n: Int) = {
  val color = Color.blue fadeOut (1 - n / 20.0).normalized
  val shape = Triangle(100 + n*24, 100 + n*24)
  shape lineWidth 10 lineColor color
}

def rainbowSquare(n: Int) = {
  val color = Color.blue desaturate 0.5.normalized spin (n * 30).degrees
  val shape = Rectangle(100 + n*24, 100 + n*24)
  shape lineWidth 10 lineColor color
}

val answer =
  manyShapes(10, rainbowCircle) beside
  manyShapes(10, fadingTriangle) beside
  manyShapes(10, rainbowSquare)
```

However, there is some redundancy here: `rainbowCircle` and `rainbowTriangle`, in particular, use the same definition of `color`. There are also repeated calls to `lineWidth(10)` and `lineColor(color)` that can be eliminated. The extra credit solution factors these out into their own functions and combines them with the `colored` combinator:

```
def manyShapes(n: Int, singleShape: Int => Image): Image =
  if(n == 1) {
    singleShape(n)
  } else {
    singleShape(n) on manyShapes(n - 1, singleShape)
  }

def colored(shape: Int => Image, color: Int => Color): Int => Image =
  (n: Int) =>
    shape(n) lineWidth 10 lineColor color(n)

def fading(n: Int): Color =
  Color.blue fadeOut (1 - n / 20.0).normalized

def spinning(n: Int): Color =
  Color.blue desaturate 0.5.normalized spin (n * 30).degrees

def size(n: Int): Double =
  50 + 12 * n

def circle(n: Int): Image =
  Circle(size(n))

def square(n: Int): Image =
  Rectangle(2*size(n), 2*size(n))

def triangle(n: Int): Image =
  Triangle(2*size(n), 2*size(n))

val answer =
  manyShapes(10, colored(circle, spinning)) beside
```

```
  manyShapes(10, colored(triangle, fading)) beside
  manyShapes(10, colored(square, spinning))
```

Return to the exercise

B.4 Collections

B.4.1 Solution to: Transforming Sequences

First let's define a method to create a single square. We'll call the method `cell` to keep the naming shape-independent and specify `size`, hue, and `lightness` as parameters:

```
def cell(size: Int, hue: Int, lightness: Double): Image =
  Rectangle(size, size) lineWidth 0 fillColor Color.hsl(hue.degrees, 1.0.normalized, lightness.normalized)
```

Next let's create a single column of varying lightness. We start with a list of lightness values, map over the list to produce the squares, and build the column using `allAbove` or `allBelow`:

```
def column(cellSize: Int, hue: Int): Image = {
  val cells =
    (0.0 until 1.0 by 0.01).toList map { lightness =>
      cell(cellSize, hue, lightness)
    }

  allAbove(cells)
}
```

Finally let's assemble the columns into a palette. We start with a list of hues, map over it to create columns, and build the palette using `allBeside`:

```
def palette(cellSize: Int): Image = {
  val columns =
    (0 until 360 by 2).toList map { hue =>
      column(cellSize, hue)
    }

  allBeside(columns)
}
```

For the extra credit solution we add `hStep`, `lStep`, and `cell` parameters. `hStep` and `lStep` are of type `Int` and `Double` respectively, and `cell` is of type `(Int, Double) => Image`. In the example below we use a *type alias* to make the type of the `cell` parameter more explicit. Type aliases are simply a way of naming types—the compiler treats an aliased type exactly the same as an unaliased one:

```
// Type alias for cell constructor functions:

type CellFunc = (Int, Double) => Image

// Different types of cell:

def squareCell(size: Int): CellFunc =
  (hue: Int, lightness: Double) =>
    Rectangle(size, size) lineWidth 0 fillColor Color.hsl(hue.degrees, 1.0.normalized, lightness.normalized)
```

```
def circleCell(size: Int): CellFunc =
  (hue: Int, lightness: Double) =>
    Circle(size/2) lineWidth 0 fillColor Color.hsl(hue.degrees, 1.0.normalized, lightness.normalized)

// Code to construct a palette:

def column(hue: Int, lStep: Double, cell: CellFunc): Image = {
  val cells =
    (0.0 until 1.0 by lStep).toList map { lightness =>
      cell(hue, lightness)
    }

  allAbove(cells)
}

def palette(hStep: Int, lStep: Double, cell: CellFunc): Image = {
  val columns =
    (0 until 360 by hStep).toList map { hue =>
      column(hue, lStep, cell)
    }

  allBeside(columns)
}

// Example use of the palette() method:

palette(2, 0.01, circleCell(10))
```

Return to the exercise

B.4.2 Solution to: Exercise: My God, It's Full of Stars!

Here's the `star` method. We've renamed p and n to `points` and `skip` for clarity:

```
def star(sides: Int, skip: Int, radius: Double) = {
  val centerAngle = 360.degrees * skip / sides

  val elements = (0 to sides) map { index =>
    val point = Vec.polar(centerAngle * index, radius)
    if(index == 0)
      MoveTo(point)
    else
      LineTo(point)
  }

  Path(elements) lineWidth 2
}
```

We'll use `allBeside` to create the row of stars. We only need to use values of `skip` from 1 to `sides/2` rounded down:

```
(allBeside((1 to 5) map { skip =>
  star(sides, skip, 100)
})).draw
```

Return to the exercise

B.4.3 Solution to: Exercise: My God, It's Full of Stars! Part 2

To create the image above, we started by adding colours and a chunkier outline to the definition of `star`:

```
def star(sides: Int, skip: Int, radius: Double) = {
  val centerAngle = 360.degrees * skip / sides

  val elements = (0 to sides) map { index =>
    val point = Vec.polar(centerAngle * index, radius)
    if(index == 0)
      MoveTo(point)
    else
      LineTo(point)
  }

  Path(elements).
    lineWidth(2).
    lineColor(Color.hsl(centerAngle, 1.normalized, .25.normalized)).
    fillColor(Color.hsl(centerAngle, 1.normalized, .75.normalized))
}
```

The updated scene then becomes:

```
allAbove((3 to 33 by 2) map { sides =>
  allBeside((1 to sides/2) map { skip =>
    star(sides, skip, 20)
  })
})
```

Return to the exercise