Essential Scala: Doodle Case Study

Noel Welsh and Dave Gurnell

Pre-release Version, August 2015

underscore

Essential Scala: Doodle Case Study

Pre-release Version, August 2015

Copyright Noel Welsh and Dave Gurnell.

Published by Underscore Consulting LLP, Brighton, UK.

Contents

Chapter 1

Introduction

This supplement to Essential Scala brings together the book's main concepts in a sizable case study. The object is to develop a *two-dimensional vector graphics and animation* library similar to Doodle. Let us first describe the what two-dimensional graphics are, then why this makes a good case study, and finally describe the structure of the case.

What we mean by two-dimensional graphics should be obvious (though, arguably, when we add animations they become three-dimensional, in an unconventional meaning of the term). By vector graphics we mean images that are specified in terms of lines and curves, rather than in terms of pixels. Because of this, vector graphics are independent of the output format. They can be rendered to high resolution screens, such as Apple's Retina displays, as easiily as they can to screens with standard resolution. They can be arbitrarily transformed without losing information, unlike bitmaps that distort when they are rotated, for example. This structure is something our library will heavily leverage.

So, why vector graphics? There are a few reasons. Firstly, graphics are fun, and hopefully something you can enjoy studying even if you haven't worked with computer graphics before. Having a tangible output is of great benefit. It's easier to get a feel for how the library works and concepts it embodies because you simply draw the images on the screen. Finally, we'll see they are great vehicle for the concepts taught in Essential Scala, and we can wrap up most of the content in the course in this one case study.

Now let's give an overview of the structure of the case study. It is divided into three main sections:

1. Building the basic objects and methods for our library, which heavily uses algebraic data types and structural recursion.
2. Adding animations, which introduces sequencing abstractions such as `map` and `fold`.
3. Abstracting the rendering pipeline, bringing in type classes and touching briefly on some more advanced functional programming concepts.

Each part asks you to implement part of the library. Supplementing this supplement is a code repository containing support code as well as working implementations for each exercise. There are many possible implementations for each exercise, each making different design tradeoffs. Our solution is just one of these, and you shouldn't take it to be any more correct than other solutions you may come up with. A primary goal of this supplement is to understand the tradeoffs different solutions make.

To get the most from this case study *you have to do the work*. While it will be valuable if you read through and look at our solutions, it will be much more valuable if you attempt every exercise yourself before looking at our solution.

Finally, we always enjoy talking to other programmers. If you have any comments or questions about this case study, do drop us an email at `noel@underscore.io` and `dave@underscore.io`.

Chapter 2

Foundation

In this section we implement the foundation for the rest of the library: the basic objects and methods for creating and drawing pictures.

Before we get into this, let's start with some background on the problem we're trying to solve.

2.1 Background

Our goal is to make it easy to create interesting vector graphics and animations. Here is a reasonable example of an interesting image:

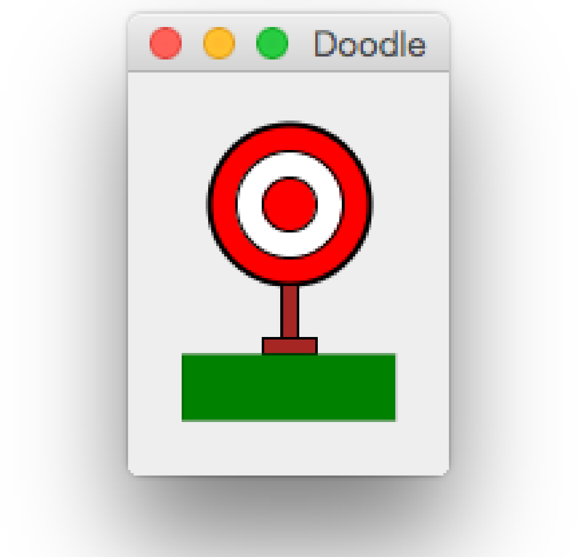

Figure 2.1: Colour archery target

Here's the code to draw it using Doodle:

```
(
  ( 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
```

Note how we build larger images from smaller ones. For example, to build the target we place three circles on top of one another.

```
  ( Circle(10) fillColor Color.red ) on
  ( Circle(20) fillColor Color.white ) on
  ( Circle(30) fillColor Color.red lineWidth 2 )
```

Note also that layout is done at a high level. We specify where images lie in relation to one-another, and Doodle works out the absolute coordinates to use when rendering to the screen.

We strongly suggest playing around with Doodle before undertaking this part of the case study. The code is online, along with many examples. You can also read Creative Scala, a free textbook that uses Doodle to introduce Scala.

The typical imperative APIs for drawing vector graphics, such as the Java 2D API or the HTML Canvas, work at a very low level of abstraction. This puts a lot of the burden on the programmer. An example of such as an API is

```
trait Canvas {
  def setStroke(stroke: Stroke): Unit
  def setFill(color: Color): Unit

  def stroke(): Unit
  def fill(): Unit

  def beginPath(): Unit
  def moveTo(x: Double, y: Double): Unit
  def lineTo(x: Double, y: Double): Unit
  def bezierCurveTo(cp1x: Double, cp1y: Double, cp2x: Double, cp2y: Double, endX: Double, endY: Double):
  def endPath(): Unit
}
```

To draw a shape we call `beginPath`, then issue a series of commands (`moveTo`, `lineTo`, or, `bezierCurveTo`), followed by an `endPath`. We can draw just the outline of the shape, in which case we next call `stroke`, or just fill it (via `fill`), or do both. The colors used for the stroke and fill and set by calls to `setStroke` and `setFill`

For example, to draw a circle we could use the following method:

```
def circle(centerX: Double, centerY: Double, radius: Double): Unit = {
  // See http://spencermortensen.com/articles/bezier-circle/ for approximation
  // of a circle with a Bezier curve.
  val c = 0.551915024494
  val cR = c * radius
  beginPath()
  moveTo(centerX, centerY + radius)
  bezierCurveTo(centerX + cR, centerY + radius,
                centerX + radius, centerY + cR,
                centerX + radius, centerY)
  bezierCurveTo(centerX + radius, centerY - cR,
                centerX + cR, centerY - radius,
                centerX, centerY - radius)
  bezierCurveTo(centerX - cR, centerY - radius,
                centerX - radius, centerY - cR,
                centerX - radius, centerY)
  bezierCurveTo(centerX - radius, centerY + cR,
                centerX - cR, centerY + radius,
                centerX, centerY + radius)
  endPath()
}
```

In my experience, there is a lot wrong with this sort of API. We have seen that it is very inconvenient compared to Doodle. Another problem is there is a lot of state involved. There is a single global stroke color, for example. This makes it difficult to abstract parts of an image into methods, as they can overwrite each other's stroke color. Similarly it's not defined what happens if we nest calls to `beginPath`, or if we call `stroke` or `fill` before calling `endPath`, and so on. All of this state makes it hard to create reusable components from which we can build up bigger pictures.

The use of a global coordinate system makes layout difficult. Imagine we have code to draw two pictures, and we now want to put these pictures side by side. First we had better have had the foresight to make the starting point of each picture a parameter to the method that draws them, or we won't be able to shift them around. Then we have to calculate our layout manually—work out how wide each picture is, and move them by an appropriate amount so they are balanced across the screen. This is a lot of donkey work, and doing donkey work is what computers are for.

We can see the imperative API is a lot more inconvenient to use. We could of course build our own abstractions, like `circle` above, to provide facilities like Doodle. Indeed that is exactly what we are going to do!

2.2 Properties of a Better System

Now we know what is wrong with the imperative approach, let's think about what kind of properties should hold in a better system. The focus here is not on the kinds of pictures we can draw—if we can use gradient fills, for example—but on the process of constructing pictures. What should that be like? Can you come up with a short description or a few terms that describe desirable properties and what they mean in the context of a drawing library? Take some time to think about this before reading our answer.

See the solution

2.3 Atoms and Operations

Now we have thought about the properties of our system, let's think about the most basic classes and operations on those classes. By now you should have had a play with Doodle and have a good idea of it's model. In Doodle the "atoms" are basic geometric shapes like circles, rectangles, and triangles. The operations that combine the atoms are layout methods like on and `beside`, and methods to manipulate stroke and fill.

Design a data type to represent the "atoms" in your model. You should probably use an algebraic data type. Hint: You might want to look at the Canvas, which is the low-level interface we'll implement our higher-level interface against.

See the solution

Now create the method signatures for the operations you'll define on your data type. You don't need to fill in the bodies yet—you can just leave then as ??? so the code compiles but won't run.

See the solution

You will also need a method draw that accepts a doodle.backend.Canvas and actually renders the images using the canvas. More on that in a moment.

2.4 Implementation

We're now ready to implement the complete system. We have provided a framework of code to build on, which you can find on Github. The project is laid as follows:

- the shared directory, which contains most of the code you will interact with;
- the jvm directory, which contains code specific to rendering graphics using the Java 2D library; and
- the js directory, which contains code for rendering using the web browser canvas.

Your code should go in the doodle.core package (in shared/src/main/scala/doodle/core). Within this package you will find utilities for handling color, angles, and other code that might be useful to you.

Within the shared directory there is also the backend package that contains the Canvas interface. When you come to actually drawing images you should assume you'll be passed as Canvas implementation.

Finally, to use the code here are some useful tips:

- you can start the Scala console by using the console command in sbt;
- within the console, you will have a Java 2D Canvas available, which you can access by calling Java2DCanvas.canvas;
- you will also have the contents of doodle.core automatically imported into the console;
- you can type Scala code into the console and use it as a simple tool to test your programs.

To quickly draw something you might try code like the following within the console:

```scala
val canvas = Java2DCanvas.canvas

// The coordinates for this (upside down) dog in the style of Picasso comes
// from a Jeremy Kun:
// http://jeremykun.com/2013/05/11/bezier-curves-and-picasso/

canvas.setSize(500, 500)
canvas.setOrigin(-250, 200)
canvas.beginPath()
canvas.moveTo(180,280)
canvas.bezierCurveTo(183,268, 186,256, 189,244) // front leg
canvas.moveTo(191,244)
canvas.bezierCurveTo(290,244, 300,230, 339,245)
canvas.moveTo(340,246)
canvas.bezierCurveTo(350,290, 360,300, 355,210)
canvas.moveTo(353,210)
canvas.bezierCurveTo(370,207, 380,196, 375,193)
canvas.moveTo(375,193)
canvas.bezierCurveTo(310,220, 190,220, 164,205) // back
canvas.moveTo(164,205)
canvas.bezierCurveTo(135,194, 135,265, 153,275) // ear start
canvas.moveTo(153,275)
canvas.bezierCurveTo(168,275, 170,180, 150,190) // ear end + head
canvas.moveTo(149,190)
canvas.bezierCurveTo(122,214, 142,204, 85,240)  // nose bridge
canvas.moveTo(86,240)
canvas.bezierCurveTo(100,247, 125,233, 140,238)   // mouth
canvas.endPath()
canvas.setStroke(Stroke(3.0, Color.black, Line.Cap.Round, Line.Join.Round))
canvas.stroke()
```

This code is in `Example.scala` in the `jvm` project.

2.4.1 Implementation Techniques

There is something that you might find a bit unexpected in the implementation of your library: an image should not been drawn until you call the `draw` method. This is necessary as we need to know the entire image before we can layout its components. Concretely, if we're rendering one image beside another, we need to know their heights so we can vertically center them. If we draw images as soon as they were created we won't know that they should be laid out in this way. The upshot is when we call, say, `image1 beside image2`, we need to represent this as a data structure somehow. When we come to do layout we also need to know the height and width of each component image. We can easily calculate this with bounding boxes—they are easy to implement and sufficient if we only allow horizontal and vertical composition.

The idea of separating the description of the computation (the image data structure) from the process that carries it out (drawing) is a classic functional programming technique, and one we will see multiple times.

2.4.2 Your Mission

Your final mission is to finish off the library:

- implement the methods for combining images; and
- implement `draw`

Along the way you will probably have to implement a bounding box abstraction.

If you're not sure where to start follow along with the rest of this section. If you think you can do it yourself, get stuck in!

When you have finished, you can compare your implementation to mine by switching to the feature/atoms-and-operations branch in your fork of the case study repository.

2.4.2.1 Drawing Something

Let's start with this representation:

```scala
import doodle.backend.Canvas

sealed trait Image {
  def on(that: Image): Image =
    ???

  def beside(that: Image): Image =
    ???

  def above(that: Image): Image =
    ???

  def draw(canvas: Canvas): Unit =
    ???
}
final case class Circle(radius: Double) extends Image
final case class Rectangle(width: Double, height:  Double) extends Image
```

Our first mission is to get some visible progress, so we'll implement `draw`. We're completely ignoring layout at this point, so you can just draw images at the origin (or anywhere else that takes your fancy).

What's the pattern we'll use in the implementation?

See the solution

Implement `draw`.

See the solution

Now we can draw stuff on the screen, let's implement the methods to combine images: `above`, `beside`, and `on`. Each method can be implemented in one line, but there is a crucial leap you need to make to implement them. Have a think about it, and read the solution when you've worked it out.

See the solution

Now we are ready to tackle the actual layout algorithm. To work out where every `Image` should be placed we need to know how much space it takes up. We can implement bounding boxes to give us this information. A bounding box is simply a rectangle that encloses an image. Bounding boxes are not precise, but they are sufficient for our choice of primitive images and layout methods.

When combining bounding boxes we will need to know the coordinate system we use to represent their coordinates. We can't use the global canvas coordinate system—the reason we're implementing this system is to work out the location of images in the global system—so we need to use a coordinate system that is local to each image. A simple choice is to say the origin is the center of the bounding box.

We can represent a bounding box as a class

```scala
final case class BoundingBox(left: Double, top: Double, right: Double, bottom: Double)
```

What methods should be have on `BoundingBox`?

See the solution

Implement these methods on `BoundingBox`.

See the solution

Now implement a `boundingBox` method (or instance variable, as you see fit) on `Image` that returns the bounding box for the image.

See the solution

Now we have enough information to do layout. Our `Image` is a tree. The top level boudning box tells us how big the entire image is. We can decide the origin of this bounding box is the origin of the global canvas coordinate system. Then we can walk down the tree (yet more structural recursion) translating the local coordinate system into the global system. When we reach a leaf node (so, a primite image), we can actually draw it. We already have the skeleton for this in `draw`—we just need to pass along the mapping from the local coordinate system to the global one. We can use a method like

```
def draw(canvas: Canvas, originX: Double, originY: Double): Unit =
  ???
```

which we call from the standard `draw` with the origin coordinates initially set to zero.

Now implement this variant of `draw`.

See the solution

Chapter 3

Animation

We are now going to add animations to our graphics library. To do this we will implement a library for processing streams of events. These systems are useful in any domain that deals with streams of data, such as financeand web analytics.

We'll start, as we did in the previous chapter, with some background to the problem and play around with a better solution before we come to implement it ourselves.

3.1 Background

To make smooth animations we must account for the characteristics of the display. Screens typically redraw sixty times a second, so we should create new frames at the same rate. We also need to provide our frames at the point in time when the screen is ready to redraw, or we might observe screen tearing.

The typical imperative solution is to setup a callback that is called every time a new frame is needed. The `Canvas` interface provides this facility, with a method `setAnimationFrameCallback`. We pass a function to `setAnimationFrameCallback`, and the `Canvas` calls this function every time the screen is ready for a new frame. We then call other methods on `Canvas` to actually create that frame.

This interface has the all problems of the imperative approach to drawing that we abandoned in the last chapter: it doesn't compose, is difficult to work with, and is difficult to reason about. What would be a better, functional, approach be?

When we look at a what an animation is, we find it quite amenable to a functional approach. Imagine we are animating a ball moving about the screen. The current position of the ball is a function of the previous postion and the current velocity.

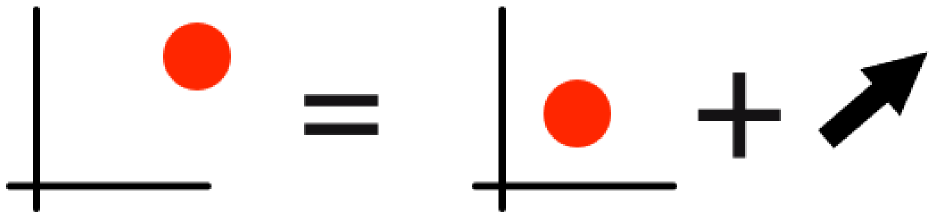

Figure 3.1: Current position is equal to the previous position plus the current velocity.

The current velocity is itself a function of the user input and the previous velocity.

Let's quickly sketch out some code to make this really concrete.

We start with a type to represent user input.

```scala
// User input is a Key
sealed trait Key
final case object Up extends Key
final case object Down extends Key
final case object Left extends Key
final case object Right extends Key
```

Now we can calculate velocity as a function of the velocity at the previous timestep and user input.

```scala
// Velocity is represented as a two dimensional Vector
def currentVelocity(previousVelocity: Vec, input: Key): Vec =
  input match {
    case Up => previousVelocity + Vec(0, 1)
    case Down => previousVelocity + Vec(0, -1)
    case Left => previousVelocity + Vec(-1, 0)
    case Right => previousVelocity + Vec(1, 0)
  }
```

Location is a function of the location at the previous time step and the velocity.

```scala
// Location is represented as a two dimensional Vector, by abuse of notation
def currentLocation(previousLocation: Vec, velocity: Vec): Vec =
  previousLocation + velocity
```

Given the current location we can draw a ball at that location. (You might not have implemented the `at` method in your version of Doodle. It places an `Image` at the given coordinates in the local coordinate system of the enclosing `Image`, or in the global coordinate system if there is no enclosing `Image`.)

```scala
// A simple image of a ball
val ball = Circle(10) fillColor Color.red

def currentBall(currentLocation: Vec): Image =
  ball at currentLocation
```

This is a good example of functional code: we've broken the problem down into small independent functions that we then compose to build the complete solution. We're still missing some parts though: how is user input obtained for example?

If we ignore interactivity for now, we can actually run the code above using `Lists` to provide the input. We `scanLeft` `currentVelocity` and `currentLocation`, and `map` `currentBall`.

You might not have seen the `scan` methods before. They are equivalent to fold but they collect the intermediate results in a list. Taking summing the elements of a `List` using a fold like so:

```scala
List(1, 2, 3, 4).foldLeft(0){ _ + _ }
// res: Int = 10
```

If we replace `foldLeft` with `scanLeft` we get a list of the partial sums.

```
List(1, 2, 3, 4).scanLeft(0){ _ + _ }
// res: List[Int] = List(0, 1, 3, 6, 10)
```

We can apply this to our Image example to get a list of intermediate image frames.

```
val input = List(Up, Up, Down, Down, Left, Right, Left, Right)

val images: List[Image] =
  input.scanLeft(Vec(0, 0)){ currentVelocity }.
    scanLeft(Vec(0, 0)){ currentLocation }.
    map(currentBall)
```

Our resulting list of images is something that we could display to make an animation.

This system is fine for rendering animations from prerecorded input, but how what about responding in real-time to user input? The values of a List are all known in advance, while user input only becomes available when keys are pressed. What we want is some kind of sequence of data where the elements are generated by external input. Imagine, for example, something like a list of keypresses where the next element springs into existence when the user presses a key. We've seen above that the basic interface of map and scanLeft allows us to express at least some animations.

We can think of a list as representing data in space. Different list indices correspond to different locations in the computer's memory. What we want is an abstraction that represents data *in time*. Indexing in this event stream corresponds to accessing events at different times. (We won't actually implement indexing as it would allow time travel, but it provides a useful conceptual model.)

The next leap is to realise that it's the interface allowing transformation (map, scanLeft, and so on) that is important, not the list-like nature. We don't want to actually store all the past inputs like we would in a list, for example. We can imagine our transformations as edges in a directed acyclic graph. User input flows into the graph and Images flow out.

3.2 A Small Example

To make this concrete let's experiment with a version of the system we'll be building. You can find this system on the feature/event branch. We're going to use it to make a little ball move around the screen in response to key presses.

We start by converting the Canvas callbacks for animation frames and key presses into event streams.

```
import doodle.backend.Key
import doodle.core._
import doodle.event._
import doodle.jvm.Java2DCanvas

val redraw = Canvas.animationFrameEventStream(canvas)
val keys = Canvas.keyDownEventStream(canvas)
```

Now we're going to convert key presses into velocity. Velocity in a vector starting at (0, 0), and we'll increment or decrement it by one as appropriate on each key press. Additionally we going to limit the x and y components of velocity to be in the range -5 to 5. This stops the ball flying around the screen to quickly.

```
val velocity = keys.scanLeft(Vec.zero)((key, prev) => {
      val velocity =
        key match {
          case Key.Up    => prev + Vec(0, 1)
          case Key.Right => prev + Vec(1, 0)
          case Key.Down  => prev + Vec(0, -1)
          case Key.Left  => prev + Vec(-1, 0)
          case _         => prev
        }
      Vec(velocity.x.min(5).max(-5), velocity.y.min(5).max(-5))
  }
)
```

Now we update the location of the ball by the velocity. Location starts at (0,0) and this time we're limiting it to be within a 600 by 600 screen.

```
val location = velocity.scanLeft(Vec.zero){ (velocity, prev) =>
    val location = prev + velocity
    Vec(location.x.min(300).max(-300), location.y.min(300).max(-300))
  }
```

Finally we create the frames and render them.

```
val ball = Circle(20) fillColor (Color.red) lineColor (Color.green)

val frames = location.map(location => ball at location)
Canvas.animate(Java2DCanvas.canvas, frames)
```

If you play with this code you'll find it has an annoying problem: it only updates the ball's position on key presses. What we really want is to the ball continually moving around the screen. We can achieve this by joining the `velocity` stream with the `redraw` stream. The resulting stream will have a value everytime there is a new value available on either `velocity` and `redraw`. Since `redraw` is updated 60 times a second (the screen refresh rate) this will give us a ball that moves around smoothly. The following redefinition of `location` is sufficient.

```
val location = redraw.join(velocity).map{ case(ts, m) => m }.
  scanLeft(Vec.zero){ (velocity, prev) =>
    val location = prev + velocity
    Vec(location.x.min(300).max(-300), location.y.min(300).max(-300))
  }
```

3.3 Interface

We can start sketching an interface. Let's call our type `EventStream[A]`, where the type variable indicates the type of elements that the event stream produces.

Write down an interface that captures the examples we've seen so far. Are there any other methods you think we should include?

See the solution

The `EventStream` interface is simpler than the `Image` interface, though we have new tools (type variables, functions) that we're using. The implementation, however, is more complex and there is much more variation in possible implementations. In the next section we'll look at one possible implementation, but I encourage you to explore your own ideas.

3.4 Implementation

We're now going to implement `EventStream`. We will start with a very simple and somewhat broken implementation, and then explore improvements of increasing complexity.

3.4.1 Mutable State

Our initial implementation will use mutable state.

Observability and equivalence. Unobservable state isn't a problem.

3.4.2 Push-driven Implementation

Our implementation approach will be push-driven. This means that when a source receives an event, we will push that event to observing nodes, and so on up the graph till we reach a node with no observers. The implication is that each node must store its observers. Let's tackle this in small steps, starting by implementing `map`.

We have the interface

```scala
sealed trait EventStream[A] {
  def map[B](f: A => B): EventStream[B]
}
```

Start by somehow implementing `map` *without worrying about observers*. Hint: use the same trick we used to implement layout for `Image`.

See the solution

Now we're going to add in the observers. These observers are a collection (a `List` will do), but of what type? An `EventStream[A]` generates events of type A but it says nothing about the type of events it accepts as input, or if it even accepts input at all. We need another type to represent this. The `Observer` type will do

```scala
sealed trait Observer[A] {
  // We'll implement this later.
  def observe(in: A): Unit =
    ???
}
```

Now we can specify that `Map` is an `Observer[A]` and an `EventStream[B]`, though we don't know how to implement `observe` yet.

```scala
final case class Map[A,B](f: A => B) extends Observer[A] with EventStream[B]
```

Now we can say every `EventStream[A]` has a collection of observers of type `Observer[A]`. It doesn't really matter what collection type we use, but we'll have to be able to update it—so it either needs to be a mutable collection or stored in a `var`. Implement this.

See the solution

When we `map` over an event stream we need to add a new observer to the `EventStream`. Implement this.

See the solution

This is the basic implementation pattern we will use for the rest of the methods. Before we implement the rest of the API let's get `observe` working. What are we going to do in `observe`? Our only concrete implementation is `Map`. In `Map` we want to transform the input using the function `f` and then push that output to all observers. We might (rightly) worry about the order in which we push the output, but for now we'll ignore that question—any order will do.

See the solution

Implement `scanLeft`. Hint: you will need to introduce mutable state to store the previous output of `scanLeft` that gets fed back into `scanLeft` when the next event arrives.

See the solution

3.4.3 Infrastructure

We still have to implement `join`, but this is trickiest part of the API. Instead of tackling it now let's work on some of the supporting infrastructure so we can get some simple animations working.

Our first step is to implement event sources, which form the beginning of an event processing graph. In our implementation an event source simply pushes its input unchanged to its observers. We need to do two things:

- a data type to represent event sources; and
- a utility function to convert a callback handler to an event source.

A callback handler is a method that allows us to set a callback that is invoked when an event arrives. The `Canvas` trait has two callback handlers, shown below.

```scala
/** Set a callback that will be called when the canvas is ready to display to a
  * new frame. This will generally occur at 60fps. There can only be a single
  * callback registered at any one time and there is no way to cancel a
  * callback once it is registered.
  *
  * The callback is passed a monotically increasing value representing the
  * current time in undefined units.
  */
def setAnimationFrameCallback(callback: Double => Unit): Unit

/** Set a callback when a key is pressed down. */
def setKeyDownCallback(callback: Key => Unit): Unit
```

To represent event sources we can simply reuse `Map` passing in the identity function. (Note that we could define another case in our algebraic data type. I'm avoiding doing so to skirt around a type inference problem that we will crash into later. I encourage you to attempt this alternative implementation to see the error, which we will learn how to solve when we discuss implementing `join`.)

We now need to implement our utility to construct a source from a callback handler. Make it so.

See the solution

With this in place you should be able to implement some simple animations.

3.4.4 Join

We are now ready to tackle `join`, the trickiest part of the system. Join takes two event streams as input, and produces a tuple of the most recent events from both streams whenever *either* stream produces an event.

All our previous nodes have had a single input, whereas join has two. Therefore we can't use the same implementation strategy as before. We also need some mutable state, so we can record the most recent event from each stream we're observing.

Let first implement a class that will hold the mutable state.

```
private [event] class MutablePair[A,B](var l: A, var r: B)
```

I've made the class private to the `event` package so we can hide this implementation detail to the outside.

Now we can implement join. Here's the start of the definition I used. See if you can implement it yourself given this start. Note that you will probably run into a compilation issue that you won't be able to solve. Read the solution for more on this.

```
final case class Join[A,B]() extends Node[(A,B),(A,B)]
```

See the solution

3.5 Easing Functions

Consider animating a UI element, such as a toolbar that slides out. We could simply linearly interpolate the position of the toolbar between the start and end points, but this simple movement lacks visual appeal. UI designers like to use movement that is more natural looking. For example, our toolbar's movement might accelarate over time, and it might bounce when it reaches it's final destination.

Functions that interpoloate position over time are known as *tweening functions* or *easing functions*. The most commonly used easing functions come from the work of Robert Penner. His easing functions include include linear functions, quadratics and higher order polynomials, and more complication equations that give oscillating behaviour. (If you've studied physics you are probably thinking of damped spring models right now. That's not a bad place to start.)

In this section we are going to build a library of easing functions. This is a good example of functional design, and it will help us make more interesting animations with our animations library.

3.5.1 The Easing Function

In our representation, an easing function will accept and return a number between 0 and 1. The input is the *time*, from 0% to 100%. The output indicates position between 0% and 100% of the final position. The output should always start at 0 and finish at 1, but can vary arbitrarily inbetween. The more mathematically inclined will recognise this as the parametric form of a path. A quadratic easing function can then be defined as *f(t) = t*ˆ2, for *t in [0, 1]*.

The obvious representation in Scala is to use a function `Double => Double`. However, not all `Double => Double` functions will be easing functions, and we'll want to add domain specific methods to our easing functions. This is makes sense to wrap our `Double => Double` functions in a type like[1]

[1]The more perfomrance oriented of your might object to the additional indirection introduced by the `Easing` wrapper. We can remove it in many cases by using a *value type*. This goes beyond Essential Scala, but it is a fairly simple thing to implement: simply extend `AnyVal`.

```scala
final case class Easing(get: Double => Double)
```

We should also implement some basic easing functions. Here are some equations to use:

- the linear function *f(t) = t*;
- the quadratic *f(t) = tˆ2*; and
- the “elastic” function *f(t) = 2ˆ(-10t)* sin((t-p/4) * 2pi / p) + 1* where *p = 0.3*

Implement these functions. You might it useful to import `scala.math`. Now implement an appropriate `apply` method on `Easing`.

3.5.2 Composing Easing Functions

Just like Bilbo, we sometimes want to go there and back again when animating objects. An *ease in* function goes from 0 to 1 ait’s input goes from 0 to 1. An *ease out* function is the reverse, going from 1 to 0 as input varies from 0 to 1. The functions we have implemented above are all ease in functions.

Given an ease in function we can construct an ease out. How? We can run it backwards and take the output away from 1. So if *f* is an ease in function, *g(t) = 1 - f(1-t)* is the corresponding ease out.

How should we represent this in code? We can easily add a method `easeOut` to `Easing`, transforming an ease in to an ease out. What would the result type of this method be? If it’s an `Easing` we could apply `easeOut` again, which yields a broken `Easing`. Hmmm…

A better solution is to rename out type `Easing` to `EaseIn` and create a new type `EaseOut`. Add a method `easeOut` to `EaseIn`, and a method `easeIn` to `EaseOut`. These methods should be inverses.

Implement this.

Given some basic functions we can compose new tweening functions from existing ones. For example, we might use the quadratic above for the first half of the element’s movement, and *f(t) = (t-1)ˆ2 + 1* for the second half (so the element deccelerates towards its final position.)

T1. Implement a few tweening functions. This shouldn’t take you very long.

Chapter 4

Type Classes and Implicits

In this section we'll explore uses of implicits. Most of our time will be spend implementing type classes, but we'll also look at how we can use implicits to provide a simpler user interface.

You should base your work off the `feature/event` branch, or your equivalent if you have implemented the previous sections.

4.1 Improving User Experience with Implicits

In this section we'll look at a few improvements we can make to the `Image` API using implicits.

4.1.1 Implicit Parameters

It's a bit inconvenient to always explicitly pass a `Canvas` to the `draw` method on `Image`. Let's add some implicit magic to make the `Canvas` optional. Make the `Canvas` argument of `draw` an implicit parameter, and makes corresponding changes to the canvas implementations to make implicit values available. Hint: look in the objects `Java2DCanvas` and `HtmlCanvas`.

4.1.2 Syntax

The interface for animating an `EventStream[Image]` is a bit clunky. We didn't add an `animate` method to `EventStream` because event streams work with generic types, though it feels like this is where such a method should live. We ended up creating a method `animate` on an object, passing it both an `EventStream[Image]` and a `Canvas`.

We can add `animate` as a method to only `EventStream[Image]` via the magic of implicit classes. In the package `doodle.syntax` (directory is `shared/src/main/scala/doodle/syntax`) add an implicit class called `EventStreamImageSyntax` following the conventions already in use in that package. Wire in your syntax to the package object in `package.scala` following the existing conventions.

When you `import doodle.syntax.eventStreamImage._` you should now have an `animate` method available on any object of type `EventStream[Image]`. This method should accept a `Canvas` as an implicit parameter, just as we have done with `draw`.

4.2 Type Classes

We're now ready to tackle the main pattern for which we use implicits: type classes.

When we designed our `EventStream` interface we drew inspiration from the existing API of `List`. It can be useful to be able to abstract over `List` and `EventStream`. If we defined such an API, we could write event processing algorithms in terms of this API, making them oblivious to the concrete implementation they run on. Then we could run our algorithms in real-time data using the `EventStream` API and over batch (offline) data using `Lists` without making an code changes.

This is a perfect application for type classes. We have two types (`EventStream` and `List`) that share a common interface but don't share a common useful supertype. In fact their are many other types that have an API much like `EventStreams` (in the standard library, `Option` and `Future` come to mind, while in Essential Scala we have implemented some of these methods for types like `Sum`). A few type classes would allow us to unify a whole load of otherwise different types, and allow us to talk in a more abstract way about operations over them.

So, what should our type classes be? We have briefly discussed functors—things that have a `map` method. What about `join` and `scanLeft`? Things that can be joined are called applicative functors, or just applicatives for short. Our scan operation has the same signature as `scanLeft` on a `List`. There is no standard type class so we'll create our own called `Scannable`. Finally we'll through `Monads` into the mix, even though `EventStream` doesn't have `flatMap` method, because they are so useful in other contexts.

We have an informal idea of the type classes. Now let's get specific. For a type `F[A]`

- A `Functor` has a method `map[B](fa: F[A])(f: A => B): F[B]`
- An `Applicative` is a `Functor` and has
- `zip(fa: F[A], fb: F[B]): F[(A, B)]`
- `point[A](a: A): F[A]`
- A `Monad` is a `Functor` and has
- `flatMap[B](fa: F[A])(f: A => F[B]): F[B]`
- `point[A](a: A): F[A]` Note you can implement `map` in terms of `flatMap` and `point`.
- A `Scannable` has a method `scanLeft[B](seed: B)(f: (B,A) => B): F[B]`

Implement these type classes, putting your code in a package `doodle.typeclasses`. Create type class instances (where you can) for `EventStream` and `List`. Put the `EventStream` instances in its companion object, and the `List` instances in `doodle.typeclasses`.

You will run into a problem doing this. Read on for the solution but *make sure you attempt the exercise before you do*.

You perhaps tried defining a type class like

```scala
trait Function[F] {
  def map[A,B](fa: F[A])(f: A => B): F[B]
}
```

and received an error like `error: F does not take type parameters`. To solve this problem we need to learn about kinds and higher-kinded types.

Kinds are like types for types. The describe the number of "holes" or parameters in a type. We distinguish between regular types like `Int` and `String`that have no holes, and *type constructors* like `List` and `EventStream` that have holes that we can fill to produce types like `List[Int]` and `EventStream[Image]`. Type constructors are analogous to functions of a single parameter, operating on the type level rather than the value level.

When we write a generic type parameter like the F in `trait Functor[F]` we must also tell Scala its kind. As you've probably guessed, no extra annotation means a regular type. To indicate a type constructor taking a single parameter we would write `F[_]`. F is the name of the type constructor, and `[_]` indicates it has a single parameter or hole. For example `trait Functor[F[_]]`

The specifying a kind on a type variable is like giving a type declaration on a regular method parameter. Just like a parameter we don't repeat the kind when we use the type variable. For example, if we write

```
trait Functor[F[_]]
```

this declares a type variable called F with kind `[_]` (so a type constructor with a single type parameter). When we use F we don't write the `[_]`. Here's an example:

```
trait Functor[F[_]] {
  def map[A,B](fa: F[A])(f: A => B): F[B]
}
```

We must enable *higher kinds* to use this feature of Scala, by importing `scala.language.higherKinds`.

Here's the complete example for `Functor`.

```
import scala.language.higherKinds

trait Functor[F[_]] {
  def map[A,B](fa: F[A])(f: A => B): F[B]
}
```

Using your new knowledge of higher kinded types, implement the rest of the type classes and the type class instances.

See the solution

For extra bonus points implement type class instances for normal types (i.e. for any type A). This is known as the identity monad / functor / applicative. Hint: types are not type constructors—they have the wrong kind! However you can get the compiler to consider types as type constructors by declaring a *type synonym* like `type Id[A] = A`.

See the solution

Why is the identity monad / functor /applicative useful?

See the solution

Go hog wild, and use your new found powers to write methods producing animations either as a `List` or an `EventStream`. This allows us to easily view individual frames (by producing a `List`) or to view the entire animations (by using an `EventStream`).

Appendix A

Solutions to Exercises

A.1 Foundation

A.1.1 Solution to: Properties of a Better System

A *compostional* library allows us to build larger pictures from smaller ones by composing them together. How could we compose images? We have already talked about layout as one possibility. We could define a new image as the composition of two images beside one another. You can imagine other layout operations to arrange images vertically, or to stack them on top of one another, and so on. We could also compose images using geometric transformations such as rotations and shearing, or using styling such as fill color.

Compostionality implies there is no global state. There are many closely related terms that all boil down to removing state: maintaining *substitution*, enabling *local reasoning*, *referential transparency*, or *purity*.

Closure is another property implied by compositionality. This means there will be operations that take two or more pictures and return an element of the same type. Closure allows us to apply operations indefinitely to build more complex pictures from simpler ones.

Our library should allow operations in terms of a *local coordinate system* to make composition easier.

Finally, we want an *expressive* library, a rather loosely defined term that we can take to mean we should write the minimal amount of code required to achieve our goal.

Return to the exercise

A.1.2 Solution to: Atoms and Operations

We can start with a very simple model like so: "An `Image` is a `Circle` or a `Rectangle`". You should be immediately be able to translate this into code.

```scala
sealed trait Image
final case class Circle(radius: Double) extends Image
final case class Rectangle(width: Double, height:  Double) extends Image
```

The low-level abstraction we are rendering to is built on paths. We could model this as, say "A `PathElement` is a `MoveTo`, `LineTo`, or `CurveTo`" and "A `Path` is a sequence of `PathElements`". This is actually the representation that Doodle uses, but we provide a higher-level interface like `Image` above for convenience. The `Path` interface can also be directly converted into code. (At this point in Essential Scala we haven't seen the `Seq` type yet. It represents a sequence of elements.)

```scala
sealed trait PathElement
final case class MoveTo(x: Double, y: Double) extends PathElement
final case class LineTo(x: Double, y: Double) extends PathElement
final case class CurveTo(cp1x: Double, cp1y: Double, cp2x: Double, cp2y: Double, endX: Double, endY: Dou

final case class Path(elements: Seq[PathElement])

final case class Image(path: Path)
```

Return to the exercise

A.1.3 Solution to: Atoms and Operations Part 2

If you're following Doodle you will have methods similar to

```scala
sealed trait Image {
  def on(that: Image): Image =
    ???

  def beside(that: Image): Image =
    ???

  def above(that: Image): Image =
    ???
}
final case class Circle(radius: Double) extends Image
final case class Rectangle(width: Double, height:  Double) extends Image
```

You might also want methods to add stroke and fill, but for the purposes of this case study we can leave them out for now.

Return to the exercise

A.1.4 Solution to: Drawing Something

`Image` is an algebraic data type, so we'll use structural recursion.

Return to the exercise

A.1.5 Solution to: Drawing Something Part 2

Save this file as `Image.scala` in `shared/src/main/scala/doodle/core`

```scala
package doodle
package core

import doodle.backend.Canvas

sealed trait Image {
  def on(that: Image): Image =
    ???

  def beside(that: Image): Image =
    ???

  def above(that: Image): Image =
    ???

  def draw(canvas: Canvas): Unit =
    this match {
      case Circle(r)      => canvas.circle(0.0, 0.0, r)
      case Rectangle(w,h) => canvas.rectangle(-w/2, h/2, w/2, -h/2)
    }
}
final case class Circle(radius: Double) extends Image
final case class Rectangle(width: Double, height:  Double) extends Image
```

With this in-place you should be able to render some simple images. From the sbt console, try

```
val canvas = Java2DCanvas.canvas
Circle(10).draw(canvas)
```

Return to the exercise

A.1.6 Solution to: Drawing Something Part 3

We need to represent the layout operations as data, which means we need to extend the `Image` algebraic data type with cases for layout. Then the method bodies just constructs the correct instance that represents the operation.

When we add these new cases to our algebraic data type we also need to add them to `draw` as well (as per the structural recursion pattern, but the compiler will complain in any case if we forget them.) Right now we're not actually doing any layout so we just recurse down the data structure and draw the leaves.

```
package doodle
package core

import doodle.backend.Canvas

sealed trait Image {
  def on(that: Image): Image =
    On(this, that)

  def beside(that: Image): Image =
    Beside(this, that)

  def above(that: Image): Image =
    Above(this, that)

  def draw(canvas: Canvas): Unit =
    this match {
      case Circle(r)      => canvas.circle(0.0, 0.0, r)
      case Rectangle(w,h) => canvas.rectangle(-w/2, h/2, w/2, -h/2)
      case Above(a, b)    => a.draw(canvas); b.draw(canvas)
      case On(o, u)       => o.draw(canvas); u.draw(canvas)
      case Beside(l, r)   => l.draw(canvas); r.draw(canvas)
    }
}
final case class Circle(radius: Double) extends Image
final case class Rectangle(width: Double, height:  Double) extends Image
final case class Above(above: Image, below: Image) extends Image
final case class On(on: Image, under: Image) extends Image
final case class Beside(left: Image, right: Image) extends Image
```

Return to the exercise

A.1.7 Solution to: Drawing Something Part 4

We will want methods to combine bounding boxes that mirror the methods to combine `Images`. So, above, `beside`, and on. We might also find it useful to store the width and height.

Return to the exercise

A.1.8 Solution to: Drawing Something Part 5

```
package doodle
package core

final case class BoundingBox(left: Double, top: Double, right: Double, bottom: Double) {
  val height: Double = top - bottom

  val width: Double = right - left

  def above(that: BoundingBox): BoundingBox =
    BoundingBox(
      this.left min that.left,
      (this.height + that.height) / 2,
      this.right max that.right,
      -(this.height + that.height) / 2
    )

  def beside(that: BoundingBox): BoundingBox =
    BoundingBox(
      -(this.width + that.width) / 2,
      this.top max that.top,
      (this.width + that.width) / 2,
      this.bottom min that.bottom
    )

  def on(that: BoundingBox): BoundingBox =
    BoundingBox(
      this.left min that.left,
      this.top max that.top,
      this.right max that.right,
      this.bottom min that.bottom
    )
}
```

Return to the exercise

A.1.9 Solution to: Drawing Something Part 6

More structural recursion! Note we can implement `boundingBox` as an instance variable as it is fixed for all time, and therefore we don't need to recalculate it.

```
package doodle
package core

import doodle.backend.Canvas

sealed trait Image {
  val boundingBox: BoundingBox =
    this match {
      case Circle(r) =>
        BoundingBox(-r, r, r, -r)
      case Rectangle(w, h) =>
        BoundingBox(-w/2, h/2, w/2, -h/2)
      case Above(a, b) =>
        a.boundingBox above b.boundingBox
      case On(o, u) =>
        o.boundingBox on u.boundingBox
      case Beside(l, r) =>
        l.boundingBox beside r.boundingBox
    }

  def on(that: Image): Image =
    On(this, that)

  def beside(that: Image): Image =
    Beside(this, that)

  def above(that: Image): Image =
    Above(this, that)

  def draw(canvas: Canvas): Unit =
    this match {
      case Circle(r)      => canvas.circle(0.0, 0.0, r)
      case Rectangle(w,h) => canvas.rectangle(-w/2, h/2, w/2, -h/2)
      case Above(a, b)    => a.draw(canvas); b.draw(canvas)
      case On(o, u)       => o.draw(canvas); u.draw(canvas)
      case Beside(l, r)   => l.draw(canvas); r.draw(canvas)
    }
}
final case class Circle(radius: Double) extends Image
```

```scala
final case class Rectangle(width: Double, height:  Double) extends Image
final case class Above(above: Image, below: Image) extends Image
final case class On(on: Image, under: Image) extends Image
final case class Beside(left: Image, right: Image) extends Image
```

Return to the exercise

A.1.10 Solution to: Drawing Something Part 7

Below is the complete code.

```scala
package doodle
package core

import doodle.backend.Canvas

sealed trait Image {
  val boundingBox: BoundingBox =
    this match {
      case Circle(r) =>
        BoundingBox(-r, r, r, -r)
      case Rectangle(w, h) =>
        BoundingBox(-w/2, h/2, w/2, -h/2)
      case Above(a, b) =>
        a.boundingBox above b.boundingBox
      case On(o, u) =>
        o.boundingBox on u.boundingBox
      case Beside(l, r) =>
        l.boundingBox beside r.boundingBox
    }

  def on(that: Image): Image =
    On(this, that)

  def beside(that: Image): Image =
    Beside(this, that)

  def above(that: Image): Image =
    Above(this, that)

  def draw(canvas: Canvas): Unit =
    draw(canvas, 0.0, 0.0)

  def draw(canvas: Canvas, originX: Double, originY: Double): Unit =
    this match {
      case Circle(r) =>
        canvas.circle(0.0, 0.0, r)
      case Rectangle(w,h) =>
        canvas.rectangle(-w/2, h/2, w/2, -h/2)
      case Above(a, b) =>
        val box  = this.boundingBox
        val aBox = a.boundingBox
        val bBox = b.boundingBox

        val aboveOriginY = originY + box.top - (aBox.height / 2)
        val belowOriginY = originY + box.bottom + (bBox.height / 2)

        a.draw(canvas, originX, aboveOriginY)
        b.draw(canvas, originX, belowOriginY)
      case On(o, u) =>
        o.draw(canvas, originX, originY)
        u.draw(canvas, originX, originY)
      case Beside(l, r) =>
        val box  = this.boundingBox
        val lBox = l.boundingBox
        val rBox = r.boundingBox

        val leftOriginX = originX + box.left  + (lBox.width / 2)
        val rightOriginX = originX + box.right - (rBox.width / 2)
        l.draw(canvas, leftOriginX, originY)
        r.draw(canvas, rightOriginX, originY)
    }
}
final case class Circle(radius: Double) extends Image
final case class Rectangle(width: Double, height:  Double) extends Image
final case class Above(above: Image, below: Image) extends Image
final case class On(on: Image, under: Image) extends Image
final case class Beside(left: Image, right: Image) extends Image
```

Return to the exercise

A.2 Animation

A.2.1 Solution to: Interface

This is a sufficient interface:

```
sealed trait EventStream[A] {
  def map[B](f: A => B): EventStream[B]

  def scanLeft[B](seed: B)(f: (B,A) => B): EventStream[B]

  def join[B](that: EventStream[B]): EventStream[(A,B)]
}
```

You might be tempted to add `flatMap`.

```
def flatMap[B](f: A => EventStream[B]): EventStream[B]
```

Let's think for a minute about what this means. The function `f` uses the input `A` to choose an `EventStream[B]` to handle further processing, thereby possibly changing the downstream event processing network on every input. There are event stream systems that allow this but it is tricky to implement and even trickier to reason about. For these reasons I've chosen to not implement `flatMap` but if you want an additional challenge you can attempt to implement it.

By the way, `scanLeft` is sometimes called `foldP`, meaning "fold over the past". This is the name you'll find in the "functional reactive programming" literature.

Return to the exercise

A.2.2 Solution to: Push-driven Implementation

Just like we did for the layout combinators (`beside`, `on`, etc.) we can represent as data the computations we don't know how to actually implement at this stage.

```scala
sealed trait EventStream[A] {
  def map[B](f: A => B): EventStream[B] =
    Map(f)
}
final case class Map[A,B](f: A => B) extends EventStream[B]
```

Note that `Map` extends `EventStream[B]`. Remember the type parameter indicates the type of events the stream produces. We need the extra parameter `A` to denote the type of events that `Map` accepts.

Return to the exercise

A.2.3 Solution to: Push-driven Implementation Part 2

I've chosen to use a mutable `ListBuffer`, but you could equally use a `List` stored in a `var`.

```scala
sealed trait Observer[A] {
  def observe(in: A): Unit =
    ???
}
sealed trait EventStream[A] {
  import scala.collection.mutable

  val observers: mutable.ListBuffer[Observer[A]] =
    new mutable.ListBuffer()

  def map[B](f: A => B): EventStream[B] =
    Map(f)
}
final case class Map[A,B](f: A => B) extends EventStream[B]
```

Return to the exercise

A.2.4 Solution to: Push-driven Implementation Part 3

```scala
sealed trait Observer[A] {
  def observe(in: A): Unit =
    ???
}
sealed trait EventStream[A] {
  import scala.collection.mutable

  val observers: mutable.ListBuffer[Observer[A]] =
    new mutable.ListBuffer()

  def map[B](f: A => B): EventStream[B] = {
    val node = Map(f)
    observers += node
    node
  }
}
final case class Map[A,B](f: A => B) extends Observer[A] with EventStream[B]
```

Return to the exercise

A.2.5 Solution to: Push-driven Implementation Part 4

We can implement `observe` using structural recursion. At this point we're not worried about the order in which we update the observers, so I've chosen left-to-right traversal. Since each call to `observe` will recursively result in another update, this choice also gives us depth-first traversal of the graph.

```scala
sealed trait Observer[A] {
  def observe(in: A): Unit =
    this match {
      case m @ Map(f) =>
        val output = f(in)
        m.observers.foreach(o => o.observe(output))
    }
}
sealed trait EventStream[A] {
  import scala.collection.mutable

  val observers: mutable.ListBuffer[Observer[A]] =
    new mutable.ListBuffer()

  def map[B](f: A => B): EventStream[B] = {
    val node = Map(f)
    observers += node
    node
  }
}
final case class Map[A,B](f: A => B) extends Observer[A] with EventStream[B]
```

You might feel some unease about this solution. A `Map` has both an input (of type `A`) and an output (of type `B`) but `Observer` only represents the input type. Nonetheless, in the `observe` method we are implicitly making use of the output type `B` when we bind `output` to the result of `f(in)` and then call `observe` on `m`'s observers.

Can we even write down a type for `output`? We can, using a feature called *existential types*. An existential type represents a specific type that we don't know. Here the existential would represent the unknown type B. Unfortunately there is a compiler bug that means the compiler actually infers `Any` in this situation.

Let's choose a different solution that doesn't introduce existential types and doesn't trigger this compiler bug. We'll introduce a type to represent a transformation that has both an input and an output—in other words `Observer[A] with EventStream[B]`—and define our structural recursion in this type where both input and output types are available.

This design also allows us to hide our mutable state (the observers) from users of the library. We haven't seen access modifiers in Scala yet, so a quick summary is in order. Scala has protected and private modifiers, like Java, but the meanings are slightly different. The good news is you can forget about the nuance. As we don't use traditional OO inheritance and overriding very often in Scala the details of access modifiers aren't very important. There is only modifier I have ever used in Scala, and that is `private[packageName]`, which makes a definition visible only within the named package. Below I've made `Node` private to the `event` package.

```scala
package doodle.event

sealed trait Observer[A] {
  def observe(in: A): Unit
}
sealed trait EventStream[A] {
  def map[B](f: A => B): EventStream[B]
}
private[event] sealed trait Node[A,B] extends Observer[A] with EventStream[B] {
  import scala.collection.mutable

  val observers: mutable.ListBuffer[Observer[B]] =
    new mutable.ListBuffer()

  def observe(in: A): Unit =
    this match {
      case m @ Map(f) =>
        val output = f(in)
        m.observers.foreach(o => o.observe(output))
    }

  def map[C](f: B => C): EventStream[C] = {
    val node = Map(f)
    observers += node
    node
  }
}
final case class Map[A,B](f: A => B) extends Node[A,B]
```

Return to the exercise

A.2.6 Solution to: Push-driven Implementation Part 5

We can use the same pattern as `map` to implement `scanLeft`.

```scala
package doodle.event

sealed trait Observer[A] {
  def observe(in: A): Unit
}
sealed trait EventStream[A] {
  def map[B](f: A => B): EventStream[B]
  def scanLeft[B](seed: B)(f: (B,A) => B): EventStream[B]
}
private[event] sealed trait Node[A,B] extends Observer[A] with EventStream[B] {
  import scala.collection.mutable

  val observers: mutable.ListBuffer[Observer[B]] =
    new mutable.ListBuffer()

  def observe(in: A): Unit =
    this match {
      case m @ Map(f) =>
        val output = f(in)
        m.observers.foreach(o => o.observe(output))
      case s @ ScanLeft(seed, f) =>
        val output = f(seed, in)
        s.seed = output
        s.observers.foreach(o => o.observe(output))
    }

  def map[C](f: B => C): EventStream[C] = {
    val node = Map(f)
    observers += node
    node
  }

  def scanLeft[C](seed: C)(f: (C,B) => C): EventStream[C] = {
    val node = ScanLeft(seed, f)
    observers += node
    node
  }
}
final case class Map[A,B](f: A => B) extends Node[A,B]
final case class ScanLeft[A,B](var seed: B, f: (B,A) => B) extends Node[A,B]
```

Return to the exercise

A.2.7 Solution to: Infrastructure

The task here is a bit underspecified, to leave it open to explore alternative designs. My design puts this method on the companion object of `EventStream`.

```scala
object EventStream {
  def fromCallbackHandler[A](handler: (A => Unit) => Unit) = {
    val stream = new Map[A,A](identity _)
    handler((evt: A) => stream.observe(evt))
    stream
  }
}
```

We can use it like so:

```
val canvas = Java2DCanvas.canvas
val source = EventStream.fromCallbackHandler(canvas.setAnimationFrameCallback _)
```

Return to the exercise

A.2.8 Solution to: Join

The trick is to realise we'll have to connect the inputs to `Join` some other way than having `Join` implement `observe` twice for both types `A` and `B`. (This won't work—consider `A` and `B` could be the same concrete type.) Then the implementation proceeds much as before. Here's my version, which doesn't compile but does follow the patterns we've used so far.

```
package doodle.event

sealed trait Observer[A] {
  def observe(in: A): Unit
}
sealed trait EventStream[A] {
  def map[B](f: A => B): EventStream[B]
  def scanLeft[B](seed: B)(f: (B,A) => B): EventStream[B]
}
object EventStream {
  def fromCallbackHandler[A](handler: (A => Unit) => Unit) = {
    val stream = new Map[A,A](identity _)
    handler((evt: A) => stream.observe(evt))
    stream
  }
}
private[event] sealed trait Node[A,B] extends Observer[A] with EventStream[B] {
  import scala.collection.mutable

  val observers: mutable.ListBuffer[Observer[B]] =
    new mutable.ListBuffer()

  def observe(in: A): Unit =
    this match {
      case m @ Map(f) =>
        val output = f(in)
        m.observers.foreach(o => o.observe(output))
      case s @ ScanLeft(seed, f) =>
        val output = f(seed, in)
        s.seed = output
        s.observers.foreach(o => o.observe(output))
      case j @ Join() =>
        j.observers.foreach(o => o.observe(in))
    }

  def map[C](f: B => C): EventStream[C] = {
    val node = Map(f)
    observers += node
    node
  }

  def scanLeft[C](seed: C)(f: (C,B) => C): EventStream[C] = {
    val node = ScanLeft(seed, f)
    observers += node
    node
  }

  def join[C](that: EventStream[C]): EventStream[(B,C)] = {
    val node = Join[B,C]()
    this.map(b => node.updateLeft(b))
    that.map(c => node.updateRight(c))
    node
  }
}
final case class Map[A,B](f: A => B) extends Node[A,B]
final case class ScanLeft[A,B](var seed: B, f: (B,A) => B) extends Node[A,B]
final case class Join[A,B]() extends Node[(A,B),(A,B)] {
  val state: MutablePair[Option[A],Option[B]] = new MutablePair(None, None)

  def updateLeft(in: A) = {
    state.l = Some(in)
    state.r.foreach { r => this.observe( (in,r) ) }
  }

  def updateRight(in: B) = {
    state.r = Some(in)
    state.l.foreach { l => this.observe( (l,in) ) }
  }
}

private [event] class MutablePair[A,B](var l: A, var r: B)
```

This gives the mysterious compilation error

```
EventStream.scala:32: error: constructor cannot be instantiated to expected type;
 found   : doodle.event.Join[A(in class Join),B(in class Join)]
 required: doodle.event.Node[A(in trait Node),B(in trait Node)]
      case j @ Join() =>
               ^
one error found
```

We have inadvertantly veered away from what Scala's type inference algorithm can handle. The source of the problem is the use of type variables in `Join` (specifically the `extends Node[(A,B),(A,B)]` part.) Here we are mixing type variables with a concrete type (a tuple), rather than simply passing them up to `Node` as we have done with `Map` and `ScanLeft`. This more complex construction is called a generalized algebraic datatype (GADT for short.)

Scala's type inference algorithm can't work out that A and B in Node are equivalent to (A,B) when Node is a Join. The solution is to abandon pattern matching, and implement our structural recursion with polymorphism. We have a bit more duplication as a result, but our code actually compiles. This seems a reasonable trade-off.

```scala
package doodle.event

sealed trait Observer[A] {
  def observe(in: A): Unit
}
sealed trait EventStream[A] {
  def map[B](f: A => B): EventStream[B]
  def scanLeft[B](seed: B)(f: (B,A) => B): EventStream[B]
}
object EventStream {
  def fromCallbackHandler[A](handler: (A => Unit) => Unit) = {
    val stream = new Map[A,A](identity _)
    handler((evt: A) => stream.observe(evt))
    stream
  }
}
private[event] sealed trait Node[A,B] extends Observer[A] with EventStream[B] {
  import scala.collection.mutable

  val observers: mutable.ListBuffer[Observer[B]] =
    new mutable.ListBuffer()

  def map[C](f: B => C): EventStream[C] = {
    val node = Map(f)
    observers += node
    node
  }

  def scanLeft[C](seed: C)(f: (C,B) => C): EventStream[C] = {
    val node = ScanLeft(seed, f)
    observers += node
    node
  }

  def join[C](that: EventStream[C]): EventStream[(B,C)] = {
    val node = Join[B,C]()
    this.map(b => node.updateLeft(b))
    that.map(c => node.updateRight(c))
    node
  }
}
final case class Map[A,B](f: A => B) extends Node[A,B] {
  def observe(in: A): Unit = {
    val output = f(in)
    observers.foreach(o => o.observe(output))
  }
}
final case class ScanLeft[A,B](var seed: B, f: (B,A) => B) extends Node[A,B] {
  def observe(in: A): Unit = {
    val output = f(seed, in)
    seed = output
    observers.foreach(o => o.observe(output))
  }
}
final case class Join[A,B]() extends Node[(A,B),(A,B)] {
  val state: MutablePair[Option[A],Option[B]] = new MutablePair(None, None)

  def observe(in: (A,B)): Unit = {
    observers.foreach(o => o.observe(in))
  }

  def updateLeft(in: A) = {
    state.l = Some(in)
    state.r.foreach { r => this.observe( (in,r) ) }
  }

  def updateRight(in: B) = {
    state.r = Some(in)
    state.l.foreach { l => this.observe( (l,in) ) }
  }
}

private [event] class MutablePair[A,B](var l: A, var r: B)
```

Return to the exercise

A.3 Type Classes and Implicits

A.3.1 Solution to: Type Classes

Once you have the hang of higher-kinded types you should find this fairly mechanical.

First the type classes themselves.

```
import scala.language.higherKinds

trait Functor[F[_]] {
  def map[A, B](fa: F[A])(f: A => B): F[B]
}

trait Monad[F[_]] extends Functor[F] {
  def flatMap[A, B](fa: F[A])(f: A => F[B]): F[B]
  def point[A](a: A): F[A]
}

trait Applicative[F[_]] extends Functor[F] {
  def zip[A, B](fa: F[A])(fb: F[B]): F[(A, B)]
  def point[A](a: A): F[A]
}

trait Scannable[F[_]] {
  def scanLeft[A,B](fa: F[A])(b: B)(f: (B,A) => B): F[B]
}
```

Now the instances

```
object ListInstances {
  implicit object list extends Functor[List] with Monad[List] with Applicative[List] with Scannable[List
    def map[A, B](fa: List[A])(f: A => B): List[B] =
      fa.map(f)
    def flatMap[A, B](fa: List[A])(f: A => List[B]): List[B] =
      fa.flatMap(f)
    def point[A](a: A): List[A] =
      List(a)
    def zip[A, B](fa: List[A])(fb: List[B]): List[(A, B)] =
      fa.zip(fb)
    def scanLeft[A,B](fa: List[A])(b: B)(f: (B,A) => B): List[B] =
      fa.scanLeft(b)(f)
  }
}

object EventStream {
  implicit object eventStream extends Functor[EventStream] with Monad[EventStream] with Applicative[Even
    def map[A, B](fa: EventStream[A])(f: A => B): EventStream[B] =
      fa.map(f)
    def point[A](a: A): EventStream[A] =
      EventStream.now(a)
    def zip[A, B](fa: EventStream[A])(fb: EventStream[B]): EventStream[(A, B)] =
      fa.zip(fb)
    def scanLeft[A,B](fa: EventStream[A])(b: B)(f: (B,A) => B): EventStream[B] =
      fa.scanLeft(b)(f)
  }
}
```

Return to the exercise

A.3.2 Solution to: Type Classes Part 2

```
object IdInstances {
  type Id[A] = A

  implicit object list extends Functor[Id] with Monad[Id] with Applicative[Id] with Scannable[Id] {
    def map[A, B](fa: Id[A])(f: A => B): Id[B] =
      f(fa)
    def flatMap[A, B](fa: Id[A])(f: A => Id[B]): Id[B] =
      f(fa)
    def point[A](a: A): Id[A] =
      a
    def zip[A, B](fa: Id[A])(fb: Id[B]): Id[(A, B)] =
      (fa, fb)
    def scanLeft[A,B](fa: Id[A])(b: B)(f: (B,A) => B): Id[B] =
      f(b,fa)
  }
}
```

Return to the exercise

A.3.3 Solution to: Type Classes Part 3

It allows us to treat normal values as if they were monads etc. and hence abstract over code that uses “real” monads / functors / applicatives and code that doesn’t. This often occurs when code is used in some contexts where it runs concurrently (e.g. in `Future`) and in other contexts where it doesn’t.

Return to the exercise