Imperative Programming 3

GUIs

Peter Jeavons

Trinity Term 2019

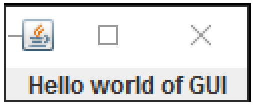
Graphical User Interfaces (GUIs)

Aim: an introduction to building simple graphics applications with scala-swing

- Layout (where things go)
- Events (listening and reacting)
- Drawing (painting shapes)
- Threads (scala-swing-style concurrency)

The `scala.swing` library

- Library gives intuitive interface into Java Swing: “Swing made easy”
- As with collections this is a big library
  - We’ll scratch the surface
  - You have code examples in a `swing` folder
- Caveat: no longer part of the Scala library
  - download `scala.swing` library separately


Hello world of GUI

Simple Swing Application

```
import scala.swing._

object FirstSwingApp extends SimpleSwingApplication {
    def top = new MainFrame {
        contents = new Label("Hello world of GUI")
    }
}
```

Note we are using an anonymous inner class to customize the MainFrame – common in GUI apps to cut down clutter

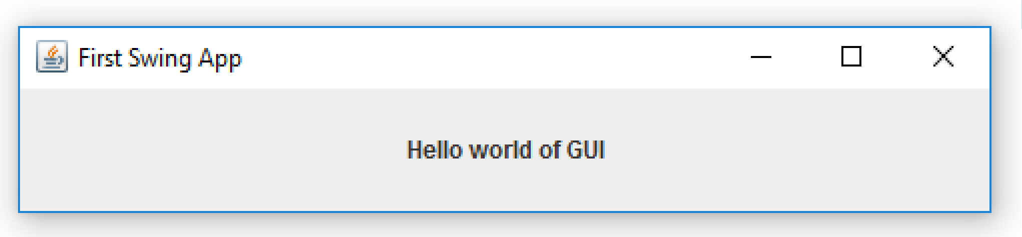
`SimpleSwingApplication` is an abstract class with one abstract method: `top`

- `top()` returns a `Frame` (window)
- `MainFrame` is a subclass of `Frame` that quits program when done

Simple Swing Application

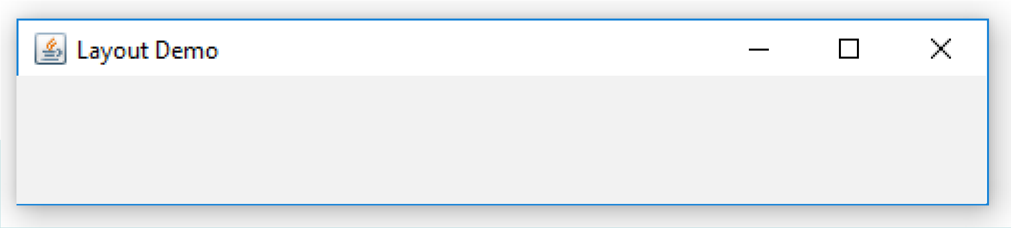
```
import scala.swing._

object FirstSwingApp extends SimpleSwingApplication {
    def top = new MainFrame {
        contents = new Label("Hello world of GUI")
        title = "First Swing App"
        location = new Point(200,400)
        size = new Dimension(500,100)
    }
}
```



Hello world of GUI

Simple Swing Application - Layout



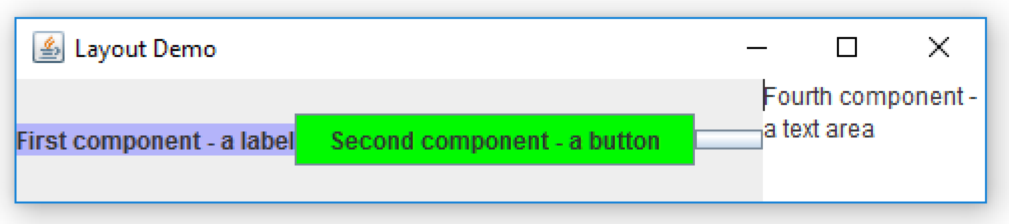
```
import scala.swing._

object LayoutDemo extends SimpleSwingApplication {
   def top = new MainFrame {
       contents = new BoxPanel(Orientation.Horizontal) {
          contents += new Label("First component - a label")
              { opaque = true; background = new Color(180,180,250) }
          contents += new Button("Second component - a button")
               { background = new Color(0,250,0) }
          contents += new ToggleButton
              { minimumSize = new Dimension(100,20) }
          contents += new TextArea("Fourth component - a text area")
               { lineWrap = true }
        }
        title = “Layout Demo"
        location = new Point(200,400)
        size = new Dimension(500,100)
    }
}
```

Other panels include:

BorderPanel, FlowPanel, GridPanel,...

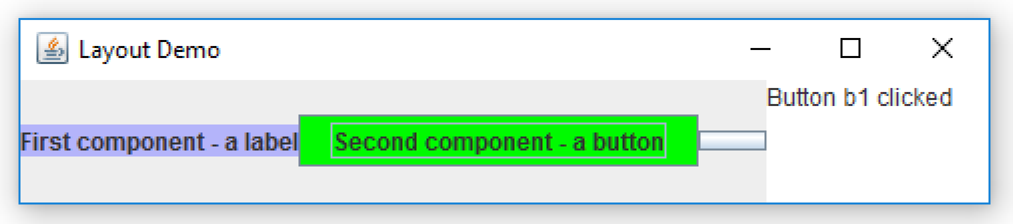
Simple Swing Application - Layout


Layout Demo
First component - a label
Second component - a button
Fourth component - a text area

```
import scala.swing._

object LayoutDemo extends SimpleSwingApplication {
   def top = new MainFrame {
       contents = new BoxPanel(Orientation.Horizontal) {
          val lab = new Label("First component - a label")
              { opaque = true; background = new Color(180,180,250) }
          val b1 = new Button("Second component - a button")
               { background = new Color(0,250,0) }
          val b2 = new ToggleButton
              { minimumSize = new Dimension(100,20) }
          val tx = new TextArea("Fourth component - a text area")
               { lineWrap = true }
          contents += (lab,b1,b2,tx)
        }
        title = “Layout Demo"
        location = new Point(200,400)
        size = new Dimension(500,100)
    }
}
```

Simple Swing Application - Events



Layout Demo

Button b1 clicked

```
import scala.swing._
import scala.swing.event._
   ...
   val b1 = new Button ...
   val b2 = new Button ...
   val tx = new TextArea("Fourth component - a text area")
               { lineWrap = true
                 listenTo(b1,b2)
                 reactions += {
                    case ButtonClicked(b) =>
                        text = "Button "+b+" clicked"
                 }
               }
```

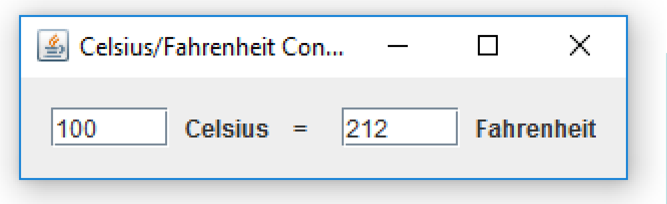
Event-driven programming

- `listenTo` allows a `Reactor` to register with a `Publisher`
- Classic Observer design pattern
- The `Reactor` trait also supplies:
  - `reactions` (partial function from events to “actions”)
  - `deafTo` (stops listening to specified publisher)
- c.f. listeners in Java: each listener independent

```
new JComponent {
  addMouseListener(new MouseAdapter {
    @Override
    def mouseClicked(e: MouseEvent) {
      System.out.println("Mouse clicked at " + e.getPoint)
    }
  })
}
```

Example: TempConverter

- Updates fields from each other


Celsius/Fahrenheit Con...

100 Celsius = 212 Fahrenheit

```
import scala.swing._
import scala.swing.event._

object TempConverter extends SimpleSwingApplication {
  def top = new MainFrame {
     val celsius     = new TextField { columns = 5 }
     val fahrenheit = new TextField { columns = 5 }
      // ...
      listenTo(celsius, fahrenheit)
      // ...
      reactions += {
        case EditDone(`fahrenheit`) =>
          val f = fahrenheit.text.toInt
          val c = (f - 32) * 5 / 9
          celsius.text = c.toString
        case EditDone(`celsius`) =>
          // ... fahrenheit.text = f.toString
}}}
```

Each TextField is a Publisher (of edit events)

Frame class is a Reactor to both

On edit to fahrenheit: calculate conversion and update celsius

Aside: Pattern Matching in Scala

```
val One = 1; val two = 2

val result = 3 match {
    case One    => "Match with One"
    case two    => "Match with two"
    case 3      => "Match with 3"
}
```

Simple names starting with lower case letters are treated as pattern variables – they match anything and take that value

- The `result` is "Match with two"
- `two` is treated as a pattern variable – it will match *anything* (because it starts lowercase)
- To force it to be taken as a constant (like `One`) write `this.two` or `` `two` ``

Aside: Pattern Matching in Scala

```
val One = 1; val two = 2

val result = 3 match {
   case One    => "Match with One"
   case `two`  => "Match with two"
   case 3      => "Match with 3"
}
```

Simple names inside back-ticks are treated as constants – they only match their current value

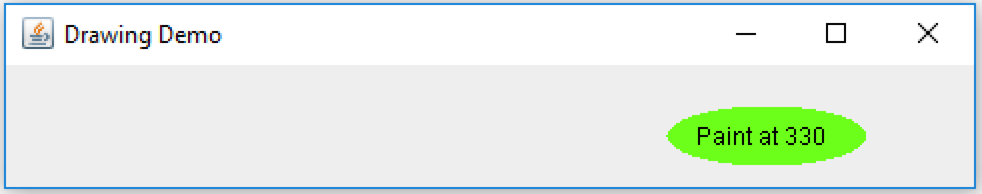
- The `result` is now “Match with 3”
- `` `two` `` is treated as a constant – it will only match the value it is currently assigned

SwingApplication abstract class

```
abstract def top: Frame
def startup(args: Array[String])
def quit() { ... }
def shutdown() { ... }
```

- `top` has to be supplied by the user and kicks off the GUI
- `startup` is main entry from command-line
- `quit` is called to gracefully shutdown
- `quit` calls `shutdown` which should clean up any resources

Simple Swing Application - Drawing



```
import scala.swing._
import scala.swing.event._
import scala.util.Random
import java.awt.Color

object DrawingDemo extends SimpleSwingApplication {
     def top = new MainFrame {
         contents = new Component {
            var x = 0
            override def paintComponent(g: Graphics2D) = {
                super.paintComponent(g)
                g.setColor(Color.getHSBColor(Random.nextFloat, 0.9f, 1.0f))
                g.fillOval(x, 20, 100, 30)
                g.setColor(new Color(0,0,0))
                g.drawString("Paint at "+x, x+15, 40)
            }

            listenTo(mouse.clicks)
            reactions += { case e: MouseClicked =>
                    x = e.point.x; repaint }
         }
     }
}
```

Any Component can be painted

Method repaint schedules paintComponent…

… can also be scheduled by window manager

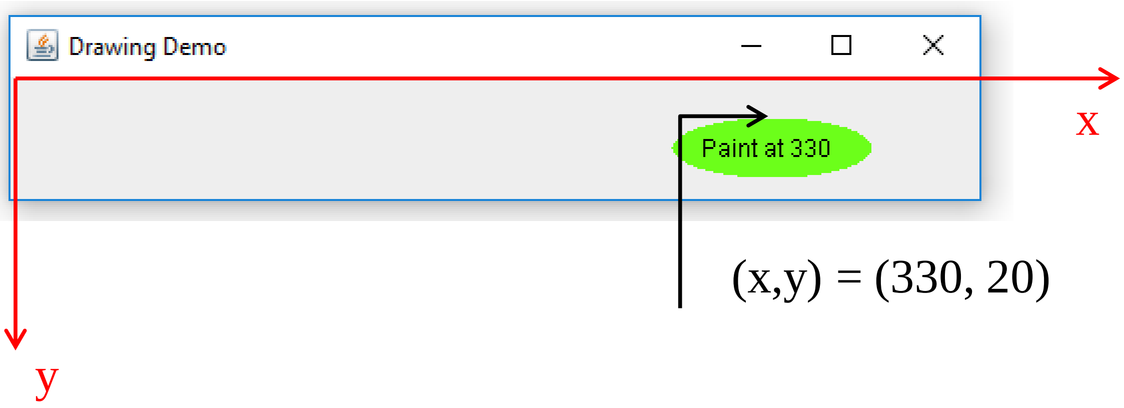
Drawing

- All drawing is done by overriding method `paintComponent(g: Graphics2D)`
  - Code should not call it directly, but use `repaint` to schedule this method to be run
  - System ensures painting is not too frequent
  - `paintComponent` is automatically called by the window manager (content is also buffered)

Drawing

- Massive functionality for shapes, lines, text, outlines etc. from the `java.awt` library
- Note standard monitor-oriented coordinates

```
...
g.fillOval(330, 20, 100, 30)
...
```

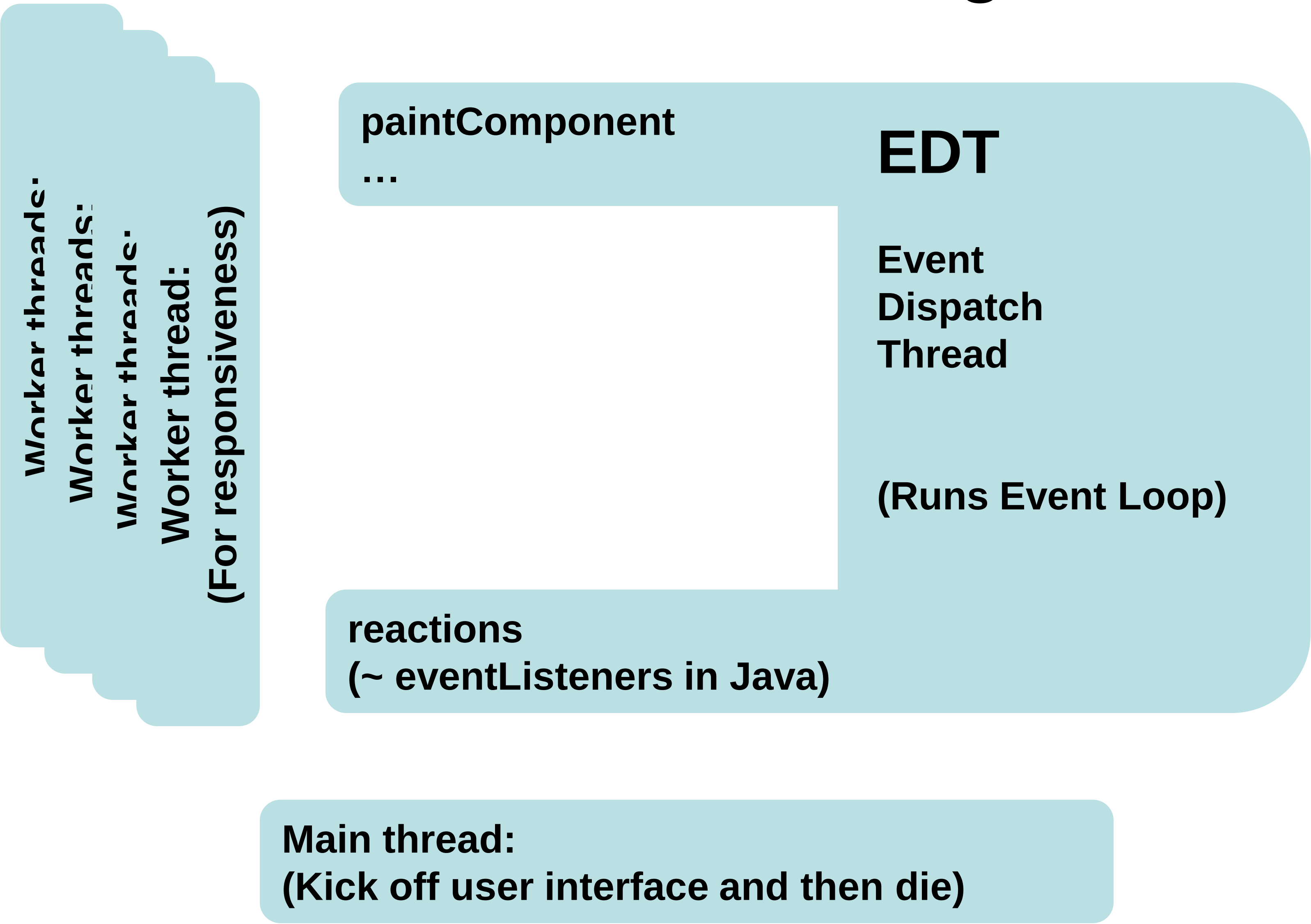


Concurrency

- An application with a GUI has to handle several different things happening at once
- Swing deals with 3 types of **thread**
  - *Initial threads* execute initial application code.
  - *Event dispatch thread* for all event-handling.

    Most code for interacting with Swing executes here
  - *Worker threads* where time-consuming background tasks are executed.
- Programmer does not create all threads: some are provided by Swing

Threads in Swing



Worker threads:

Worker threads:

Worker threads:

Worker thread:
(For responsiveness)

paintComponent
…

EDT

Event
Dispatch
Thread

(Runs Event Loop)

reactions
(~ eventListeners in Java)

Main thread:
(Kick off user interface and then die)

Swing thread rules

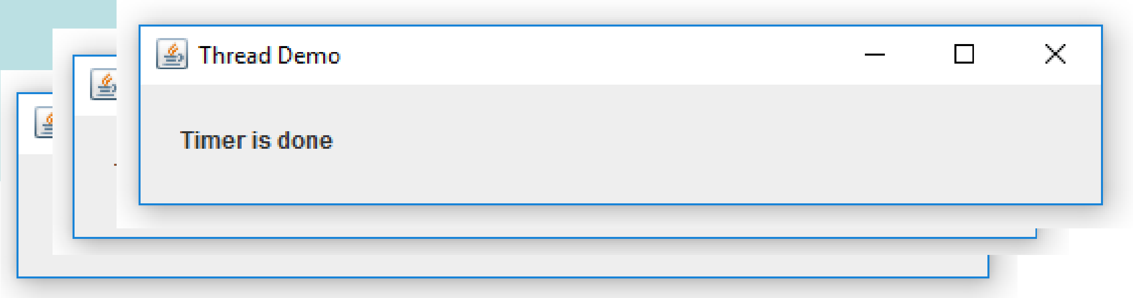
1. All creation & modification of GUI components *must be done on the event dispatch thread*
   - Swing methods are not all thread-safe (misuse may cause race-conditions or deadlock)
2. Time-consuming activities should *not* be on event dispatch thread
   - EDT must remain responsive to changes from windowing system

Simple Swing Application - Threads

```
object ThreadDemo extends SimpleSwingApplication {
    def top = new MainFrame {
       val label = new Label{text="Thread Demo initialised"}
       contents = new BoxPanel(Orientation.Vertical) {
          contents += label
          border = Swing.EmptyBorder(20, 20, 20, 20)
       }
        val timer = new Thread {
           override def run {
               Thread.sleep(2000);
               Swing.onEDT{label.text="Timer is halfway"}
               Thread.sleep(2000);
               Swing.onEDT{label.text="Timer is done"}
           }
        }
       timer.start()
    }
}
```

Invokes method on Event Dispatch Thread

Runs a new worker Thread in parallel



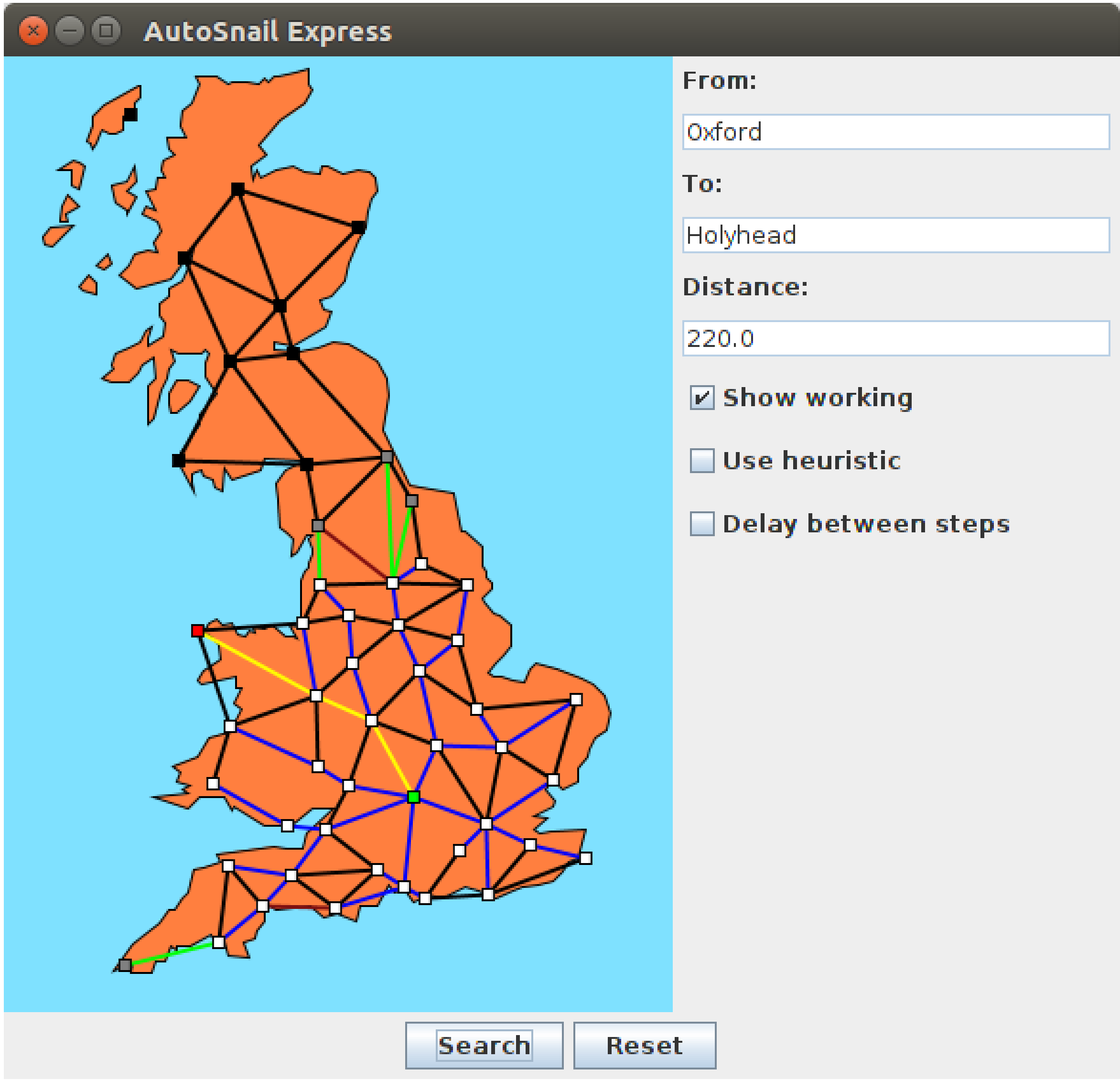
Timer is done

Summary via AutoSnail demo

Example App

- Layout
- Events
- Drawing
- Threads

See also
Programming in Scala:
Chapter 34



AutoSnail Express

From:
Oxford

To:
Holyhead

Distance:
220.0

☑ Show working

□ Use heuristic

□ Delay between steps