Chapter 34

GUI Programming

In this chapter you'll learn how to develop in Scala applications that use a graphical user interface (GUI). The applications we'll develop are based on a Scala library that provides access to Java's Swing framework of GUI classes. Conceptually, the Scala library resembles the underlying Swing classes, but hides much of their complexity. You'll find out that developing GUI applications using the framework is actually quite easy.

Even with Scala's simplifications, a framework like Swing is quite rich, with many different classes and many methods in each class. To find your way in such a rich library, it helps to use an IDE such as Scala's Eclipse plugin. The advantage is that the IDE can show you interactively with its command completion which classes are available in a package and which methods are available for objects you reference. This speeds up your learning considerably when you first explore an unknown library space.

Figure 34.1 - A simple Swing application: initial (left) and resized (right).

34.1 A FIRST SWING APPLICATION

As a first Swing application, we'll start with a window containing a single button. To program with Swing, you need to import various classes from Scala's Swing API package:

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

Listing 34.1 shows the code of your first Swing application in Scala. If you compile and run that file, you should see a window as shown on the left of Figure 34.1. The window can be resized to a larger size as shown on the right of Figure 34.1.

```
import scala.swing._

object FirstSwingApp extends SimpleSwingApplication {
  def top = new MainFrame {
    title = "First Swing App"
    contents = new Button {
      text = "Click me"
    }
  }
}
```

Listing 34.1 - A simple Swing application in Scala.

If you analyze the code in Listing 34.1 line by line, you'll notice the following elements:

```
object FirstSwingApp extends SimpleSwingApplication {
```

In the first line after the import, the FirstSwingApp object inherits fromscala.swing.SimpleSwingApplication. This application differs from traditional command-line applications, which may inherit from scala.App. The SimpleSwingApplication class already defines a main method that contains some setup code for Java's Swing framework. The main method then proceeds to call the top method, which you supply:

```
def top = new MainFrame {
```

The next line implements the top method. This method contains the code that defines your top-level GUI component. This is usually some kind of Frame—*i.e.*, a window that can contain arbitrary data. In Listing 34.1, we chose a MainFrame as the top-level component. A MainFrame is like a normal Swing Frame except that closing it will also close the whole GUI application.

```
title = "First Swing App"
```

Frames have a number of attributes. Two of the most important are the frame's title, which will be written in the title bar, and its contents, which will be displayed in the window itself. In Scala's Swing API, such attributes are modeled as properties. You know from Section 18.2that properties are encoded in Scala as pairs of getter and setter methods. For instance, thetitle property of a Frame object is modeled as a getter method:

```
def title: String
```

and a setter method:

```
def title_=(s: String)
```

It is this setter method that gets invoked by the above assignment to title. The effect of the assignment is that the chosen title is shown in the header of the window. If you leave it out, the window will have an empty title.

```
contents = new Button {
```

The top frame is the root component of the Swing application. It is a Container, which means that further components can be defined in it. Every Swing container has a contents property, which allows you to get and set the components it contains. The getter contents of this property has type Seq[Component], indicating that a component can in general have several objects as its contents. Frames, however, always have just a single component as theircontents. This component is set and potentially changed using the setter contents_=. For example, in Listing 34.1 a single Button constitutes the contents of the top frame.

```
text = "Click me"
```

The button also gets a title, in this case "Click me."

34.2 PANELS AND LAYOUTS

As next step, we'll add some text as a second content element to the top frame of the application. The left part of Figure 34.2 shows what the application should look like.

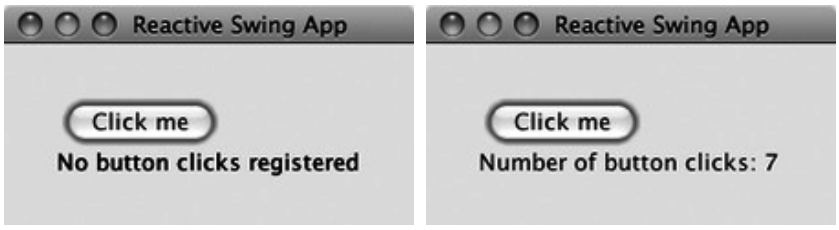

Figure 34.2 - A reactive Swing application: initial (left) after clicks (right).

```
import scala.swing._

object SecondSwingApp extends SimpleSwingApplication {
  def top = new MainFrame {
    title = "Second Swing App"
    val button = new Button {
      text = "Click me"
    }
    val label = new Label {
      text = "No button clicks registered"
    }
    contents = new BoxPanel(Orientation.Vertical) {
      contents += button
      contents += label
      border = Swing.EmptyBorder(30, 30, 10, 30)
    }
  }
}
```

Listing 34.2 - Component assembly on a panel.

You saw in the last section that a frame contains exactly one child component. Hence, to make a frame with both a button and a label, you need to create a different container component that holds both. That's what panels are used for. A Panel is a container that displays all the components it contains according to some fixed layout rules. There are a number of different possible layouts that are implemented by various subclasses of classPanel, ranging from simple to quite intricate. In fact, one of the hardest parts of a complex GUI application can be getting the layouts right—it's not easy to come up with something that displays reasonably well on all sorts of devices and for all window sizes.

Listing 34.2 shows a complete implementation. In this class, the two sub-components of thetop frame are named button and label. The button is defined as before. The label is a displayed text field that can't be edited:

```
val label = new Label {
  text = "No button clicks registered"
}
```

The code in Listing 34.2 picks a simple vertical layout where components are stacked on top of each other in a BoxPanel:

```
contents = new BoxPanel(Orientation.Vertical) {
```

The contents property of the BoxPanel is an (initially empty) buffer, to which the button and labelelements are added with the += operator:

```
contents += button
contents += label
```

We also add a border around the two objects by assigning to the border property of the panel:

```
border = Swing.EmptyBorder(30, 30, 10, 30)
```

As is the case with other GUI components, borders are represented as objects. EmptyBorder is a factory method in object Swing that takes four parameters indicating the width of the borders on the top, right, bottom, and left sides of the objects to be drawn.

Simple as it is, the example has already shown the basic way to structure a GUI application. It is built from components, which are instances of scala.swing classes such as Frame, Panel, Labelor Button. Components have properties, which can be customized by the application. Panelcomponents can contain several other components in their contents property, so that in the end a GUI application consists of a tree of components.

34.3 HANDLING EVENTS

On the other hand, the application still misses an essential property. If you run the code inListing 34.2 and click on the displayed button, nothing happens. In fact, the application is completely static; it does not react in any way to user events except for the close button of thetop frame, which terminates the application. So as a next step, we'll refine the application so that it displays together with the button a label that indicates how often the button was clicked. The right part of Figure 34.2 contains a snapshot of what the application should look like after a few button clicks.

To achieve this behavior, you need to connect a user-input event (the button was clicked) with an action (the displayed label is updated). Java and Scala have fundamentally the same "publish/subscribe" approach to event handling: Components may be publishers and/or subscribers. A publisher publishes events. A subscriber subscribes with a publisher to be notified of any published events. Publishers are also called "event sources," and subscribers are also called "event listeners". For instance a Button is an event source, which publishes an event, ButtonClicked, indicating that the button was clicked.

In Scala, subscribing to an event source source is done by the call listenTo(source). There's also a way to unsubscribe from an event source using deafTo(source). In the current example application, the first thing to do is to get the top frame to listen to its button, so that it gets notified of any events that the button issues. To do that you need to add the following call to the body of the top frame:

```
listenTo(button)
```

Being notified of events is only half the story; the other half is handling them. It is here that the Scala Swing framework is most different from (and radically simpler than) the Java Swing API's. In Java,

signaling an event means calling a "notify" method in an object that has to implement some Listener interfaces. Usually, this involves a fair amount of indirection and boiler-plate code, which makes event-handling applications hard to write and read. By contrast, in Scala, an event is a real object that gets sent to subscribing components much like messages are sent to actors. For instance, pressing a button will create an event which is an instance of the following case class:

```
case class ButtonClicked(source: Button)
```

The parameter of the case class refers to the button that was clicked. As with all other Scala Swing events, this event class is contained in a package named scala.swing.event.

To have your component react to incoming events you need to add a handler to a property called reactions. Here's the code that does this:

```
var nClicks = 0
reactions += {
  case ButtonClicked(b) =>
    nClicks += 1
    label.text = "Number of button clicks: " + nClicks
}
```

The first line above defines a variable, nClicks, which holds the number of times a button was clicked. The remaining lines add the code between braces as a handler to the reactionsproperty of the top frame. Handlers are functions defined by pattern matching on events, much like Akka actor receive methods are defined by pattern matching on messages. The handler above matches events of the form ButtonClicked(b), *i.e.*, any event which is an instance of class ButtonClicked. The pattern variable b refers to the actual button that was clicked. The action that corresponds to this event in the code above increments nClicks and updates the text of the label.

Generally, a handler is a PartialFunction that matches on events and performs some actions. It is also possible to match on more than one kind of event in a single handler by using multiple cases.

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

object ReactiveSwingApp extends SimpleSwingApplication {
  def top = new MainFrame {
    title = "Reactive Swing App"
    val button = new Button {
      text = "Click me"
    }
    val label = new Label {
      text = "No button clicks registered"
    }
    contents = new BoxPanel(Orientation.Vertical) {
      contents += button
      contents += label
      border = Swing.EmptyBorder(30, 30, 10, 30)
    }
    listenTo(button)
    var nClicks = 0
    reactions += {
```

```
        case ButtonClicked(b) =>
          nClicks += 1
          label.text = "Number of button clicks: " + nClicks
      }
    }
  }
```

Listing 34.3 - Implementing a reactive Swing application.

The reactions property implements a collection, just like the contents property does. Some components come with predefined reactions. For instance, a Frame has a predefined reaction that it will close if the user presses the close button on the upper right. If you install your own reactions by adding them with += to the reactions property, the reactions you define will be considered in addition to the standard ones. Conceptually, the handlers installed in reactionsform a stack. In the current example, if the top frame receives an event, the first handler tried will be the one that matches on ButtonClicked, because it was the last handler installed for the frame. If the received event is of type ButtonClicked, the code associated with the pattern will be invoked. After that code has completed, the system will search for further handlers in the event stack that might also be applicable. If the received event is not of type ButtonClicked, the event is immediately propagated to the rest of the installed handler stack. It's also possible to remove handlers from the reaction property, using the -= operator.

Listing 34.3 shows the completed application, including reactions. The code illustrates the essential elements of a GUI application in Scala's Swing framework: The application consists of a tree of components, starting with the top frame. The components shown in the code areFrame, BoxPanel, Button, and Label, but there are many other kinds of components defined in the Swing libraries. Each component is customized by setting attributes. Two important attributes are contents, which fixes the children of a component in the tree, and reactions, which determines how the component reacts to events.

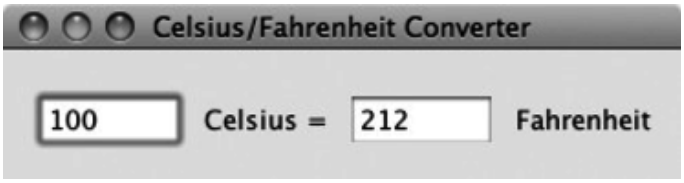

Figure 34.3 - A converter between degrees Celsius and Fahrenheit.

```
import swing._
import event._

object TempConverter extends SimpleSwingApplication {
  def top = new MainFrame {
    title = "Celsius/Fahrenheit Converter"
    object celsius extends TextField { columns = 5 }
    object fahrenheit extends TextField { columns = 5 }
    contents = new FlowPanel {
      contents += celsius
      contents += new Label(" Celsius  =  ")
      contents += fahrenheit
      contents += new Label(" Fahrenheit")
      border = Swing.EmptyBorder(15, 10, 10, 10)
```

```
      }
      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`) =>
          val c = celsius.text.toInt
          val f = c * 9 / 5 + 32
          fahrenheit.text = f.toString
      }
    }
  }
```

Listing 34.4 - An implementation of the temperature converter.

34.4 EXAMPLE: CELSIUS/FAHRENHEIT CONVERTER

As another example, we'll write a GUI program that converts between temperature degrees in Celsius and Fahrenheit. The user interface of the program is shown in Figure 34.3. It consists of two text fields (shown in white) with a label following each. One text field shows temperatures in degrees Celsius, the other in degrees Fahrenheit. Each of the two fields can be edited by the user of the application. Once the user has changed the temperature in either field, the temperature in the other field should automatically update.

Listing 34.4 shows the complete code that implements this application. The imports at the top of the code use a short-hand:

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

This is in fact equivalent to the imports used before:

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

The reason you can use the shorthand is that packages nest in Scala. Because packagescala.swing is contained in package scala, and everything in that package imported automatically, you can write just swing to refer to the package. Likewise, packagescala.swing.event, is contained as subpackage event in package scala.swing. Because you have imported everything in scala.swing in the first import, you can refer to the event package with just event thereafter.

The two components celsius and fahrenheit in TempConverter are objects of class TextField. ATextField in Swing is a component that lets you edit a single line of text. It has a default width, which is given in the columns property measured in characters (set to 5 for both fields).

The contents of TempConverter are assembled into a panel, which includes the two text fields and two labels that explain what the fields are. The panel is of class FlowPanel, which means it displays all its elements one after another, in one or more rows, depending on the width of the frame.

The reactions of TempConverter are defined by a handler that contains two cases. Each case matches an EditDone event for one of the two text fields. Such an event gets issued when a text field has been edited by the user. Note the form of the patterns, which include back ticks around the element names:

```
case EditDone(`celsius`)
```

As was explained in Section 15.2, the back ticks around celsius ensure that the pattern matches only if the source of the event was the celsius object. If you had omitted the back ticks and just written case EditDone(celsius), the pattern would have matched every event of class EditDone. The changed field would then be stored in the pattern variable celsius. Obviously, this is not what you want. Alternatively, you could have defined the two TextFieldobjects starting with upper case characters, *i.e.*, Celsius and Fahrenheit. In that case you could have matched them directly without back ticks, as in case EditDone(Celsius).

The two actions of the EditDone events convert one quantity to another. Each starts by reading out the contents of the modified field and converting it to an Int. It then applies the formula for converting one temperature degree to the other, and stores the result back as a string in the other text field.

34.5 CONCLUSION

This chapter has given you a first taste of GUI programming, using Scala's wrappers for the Swing framework. It has shown how to assemble GUI components, how to customize their properties, and how to handle events. For space reasons, we could discuss only a small number of simple components. There are many more kinds of components. You can find out about them by consulting the Scala documentation of the package scala.swing. The next section will develop an example of a more complicated Swing application.

There are also many tutorials on the original Java Swing framework, on which the Scala wrapper is based.[1] The Scala wrappers resemble the underlying Swing classes, but try to simplify concepts where possible and make them more uniform. The simplification makes extensive use of the properties of the Scala language. For instance, Scala's emulation of properties and its operator overloading allow convenient property definitions using assignments and += operations. Its "everything is an object" philosophy makes it possible to inherit the main method of a GUI application. The method can thus be hidden from user applications, including the boilerplate code for setting things up that comes with it. Finally, and most importantly, Scala's first-class functions and pattern matching make it possible to formulate event handling as the reactions component property, which greatly simplifies life for the application developer.

Footnotes for Chapter 34:

[1] See, for instance, *The Java Tutorials*. [Jav]