Chapter 41
GUIs in Scala Swing

41.1 Introduction

This chapter describes how to create rich client graphical displays (desktop application) using the Scala Swing windowing and graphical types.

Since Scala 2.11, the Scala wing package I is no longer considered to be a part of Scala's standard library API and thus some tools may require an additional dependency to be set up to pick it up. This is not the case with IntelliJ's IDE but is the case with the Simple Build Tool (SBT).

The reason for this is that Scala Swing library is now a community-maintained library and this considered to be unsupported by the core Scala maintainers. For more information, see https://github.com/scala/scala-swing.

In this chapter, we consider how Windows, buttons, tables, etc., are created, added to Windows, positioned and organised in Scala.

41.2 Windows as Objects

In Scala, Windows and their contents are instances of appropriate classes (such as Button or FlowPanel). Thus, when you create a window, you create an object that knows how to display itself on the computer screen. You must tell it what to display, although the framework within which the associated method (paint) is called is hidden from you. You should bear the following points in mind during your reading of this chapter; they will help you understand what you are required to do:

- You create a window by instantiating an object.
- You define what the window displays by adding a component to its contents, such as a type of panel or a button.
- You can send messages to the window to change its state, perform an operation and display a graphic object.

© Springer International Publishing AG 2018

J. Hunt, *A Beginner's Guide to Scala, Object Orientation and Functional Programming*, https://doi.org/10.1007/978-3-319-75771-1_41

- The window, or components within the window, can send messages to other objects in response to user (or program) actions.
- Everything displayed by a window is an instance and is potentially subject to all of the above.

This approach may very well contrast with your previous experience. In many other windowing systems, you must call the appropriate functions in order to obtain a window, providing default behaviour either by using pointers to functions or by associating some event with a particular function. You also determine what is displayed in the window by calling various functions on the window.

In the Object-Oriented World, you define how a subclass of the windowing classes responds to events, for example what it does in response to a request to paint itself. All the windowing functionality and window display code are encapsulated within the window itself.

41.3 Windows in Scala

Of course the above description is a little simplistic. It ignores the issue of how a window is created, initialised and displayed, and how its contents are generated. In Scala, these are handled by the concepts of a frame, a component and a container:

Frames provide the basic structure for a window: borders, a label and some basic functionality (e.g. resizing).

Components are graphical objects displayed in a frame. Some other languages refer to them as widgets. Examples of components are lines, circles, boxes and text.

Containers are special types of component that are made up of one or more components (or containers). All the components within a container (such as a panel) can be treated as a single entity.

Windows have a component hierarchy that is used (amongst other things) to determine how and when elements of the window are drawn. The component hierarchy is rooted with the frame, within which components and containers can be added. Figure 41.1 illustrates a component hierarchy for a window with a frame, two containers (subclasses of `Panel`, which is itself a direct subclass of `Container`) and a few basic components.

When the runtime environment needs to redraw the window displayed by the above hierarchy, it starts with the highest component (the frame) and asks it to redraw itself. It then works down the hierarchy to the bottom components, asking each to redraw itself. In this way, each component draws itself before any components that it contains; this process is handled by the windowing framework. The user generally only has to redefine the `paint` method to change the way in which a component is drawn.

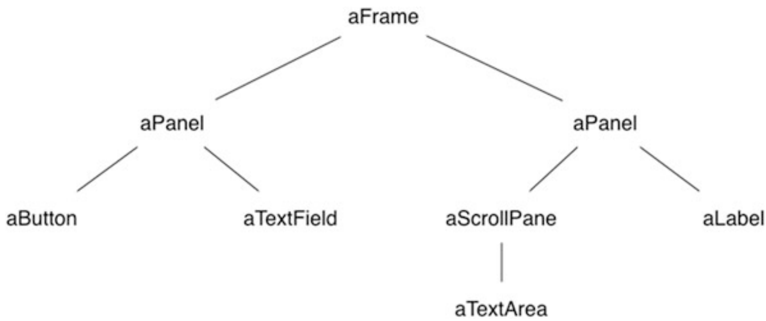

Fig. 41.1 Component hierarchy for a sample window

41.4 Scala Swing

Scala Swing is a Scala wrapping around the underlying Swing library of GUI classes and types. This of course raises the question 'What is Swing?' Swing is a generic, platform-independent, windowing system originally developed for the Java programming language. It allows you to write graphical programs that have (almost) the same look and feel, whatever the host platform. For example, the graphical applications presented within this section of the book have been written primarily on a Mac; however, they have been run on Macs, Windows machines of various flavours and Linux boxes. This is because these are the machines available to me, and depending on the time of day, I may be working on one or the other. I do not need to worry about the host environment, only about the Swing.

There are a number of concepts that are important when considering Swing. These are:

- All components in Swing are 100% implemented purely in byte codes (and are referred to as lightweight). This means that the Windows (and their contents) should look the same whatever platform they are on and should be easier to maintain across different operating systems. As all graphical components are "lightweight", their appearance is controlled purely by the runtime code (without the need for platform dependent peers—or platform dependent libraries). In Swing, a separate user interface "view" component performs the actual rendering of the component on the screen. This allows different views to be plugged in. This in turn allows the idea of installing different "look and feel". This means that it is easier to deploy the same software on different platforms, but with interfaces that match the interface manager on that particular platform. This separation of view from the actual component is based on a modified version of the Model-View-Controller architecture (described earlier in the Play framework chapter).

- Swing also provides a rich set of facilities for graphical user interfaces, including trees, icons on buttons, dockable menus bars, menus with icons, borders and improved support for fonts and colours.

However, the Scala Swing library is more than just a wrapping around the original Swing library. It significantly simplifies the Swing API and the tasks that the programmer must perform to create a user interface. In many ways, the Scala Swing library is what the original Swing library should have been; lean, efficient and simple to work with.

Whichever UI framework you choose to use, such toolkits greatly reduce the problems that software vendors often face when attempting to deliver their system on different platforms.

41.5 Scala Swing Packages

There are two key packages within the Scala Swing library.

- **scala.swing** is the package that contains the core types with which you will be working, such as Button, ComboBox, EditorPane, FileChooser, FlowPanel, MainFrame, RadioButton, ScrollPane.
- **scala.swing.event** package provides types used with the event handling mechanism that underpins how a UI can react to user inputs (such as what to do when a user clicks on a button). This package includes types such as ButtonClicked, KeyPressed, MouseMoved, WindowClosing.

However, you will primarily work with the scala.swing package as this is where the majority of the types you will need are defined. The reactor framework (discussed in the next chapter) is used to handle user input and is built on top of the lower-level event framework (inherited from Java). This framework is also defined within the scala.swing package.

The key classes within the *scala.swing* package are:

- **Component**. This is the base class for all user interface elements that can be displayed within a Scala window. Components have properties such as whether they are enabled or not, the font used with them, foreground colours and background colours, borders and tool tips (text popped up and displayed to the user if they hover the mouse over the component). Components also publish (or fire) events that allow developers to react to these events (such as the component being selected by a user). There are a large number of subtypes for Component including various types of buttons, panels, fields, menus, tables.

- **Container**. This is a base trait for user interface elements that can act as containers of other UI elements. For example, a FlowPanel can *contain* a set of other elements (such as button) organised across the panel. As well as FlowPanel, BoxPanel, BorderPanel, MenuBar, ScrollPane, SplitPane, etc., are all examples of containers.
- **MainFrame**. This class defines a *frame* that can be used as the top-level, main application window that can contain other *containers* and *components*. When the MainFrame is closed, then the UI application is terminated.
- **SwingApplication** and **SimpleSwingApplication**. These classes provide a set of utility methods to handle starting up a UI application. SwingApplication is the root class, and SimpleSwingApplication is its subclass. Most UI applications will extend the SimpleSwingApplication class (e.g. instead of mixing in the App trait). The subtype extending this class must implement the top method that must return the top-level frame (e.g. MainFrame) to be used with this application. The UI framework initialisation and initiation is done by the SimpleSwing Application. If you wish to customise the behaviour of the start-up and shutdown process, then you can override the *shutdown* method or extend the *startup* method.
- **FlowPanel**. It is a panel (i.e. container) that organises its contents horizontally, one after the other. If the contents do not fit in the current display, then the pane will introduce a break and try on the next line. It is similar in effect to a JPanel with a FlowLayout in Java Swing.
- **BorderPanel**. It is a panel (i.e. container) that organises its contents into discreet locations, such as centre, top, bottom, left and right.
- **Button** provides the basic button type display and behaviour present in most user interfaces.
- **Menu**, **MenuBar** and **MenuItem**. The components that can be used to construct a menu across the top of a UI.
- **ScrollPane** a type used to wrap up other UI elements that need to be displayed using a scrollable view. The view is controlled by a set of scroll bars.
- **TextField** and **TextArea**. Both are used to display textual information and may be editable or not. A TextField is a single line display, and a TextArea is a multi-line textual display. A TextArea should be displayed within a ScrollPane if it is too large to display within the available space.
- **Table**. It provides a tabular display comprising rows and columns and optionally a set of headings.
- **ComboBox** is used to allow a user to make a selection from a list of pre-defined items.
- **ListView** is used to present a (non-editable) list of items.

41.6 Swing Scala Worked Examples

This section presents a series of examples that explore a number of the concepts that underpin the Scala Swing framework and illustrate how some of the key classes presented above can be used.

41.6.1 Simple Hello World UI

The following listing provides a basic user interface containing a Button labelled "Click Me!" and with a window title "Hello World!". The application imports the Button, MainFrame and SimpleSwingApplication types from the scala.swing package. It then defines a class SimpleFrame that is a subclass of MainFrame. It defines the title of the main frame and creates a new instance of the Button class. The button will have the text "Click Me!" displayed. The instance of the button is then used as the contents of the main display area of the frame.

```
package com.jjh.scala.swing

import scala.swing.Button
import scala.swing.MainFrame
import scala.swing.SimpleSwingApplication

class SimpleFrame extends MainFrame {
  // set title for the window
  title = "Hello World!"
  // Create a button and set it as the
  // contents of the window
  contents = new Button {
    text = "Click Me!"
  }
}

object SwingHelloWorld extends SimpleSwingApplication {
  // define the method top to return an instance of
  // the SimpleFrame

   def top = new SimpleFrame()
 }
```

The object SwingHellWorld extends the SimpleSwingApplication type and defines the top method as returning an instance of the new SimpleFrame class. When this SwingHelloWorld application runs, the main method defined by the SimpleSwingApplication executes and sets up the windowing framework and calls the top method to display the top most element of the UI. The result of executing this application is shown in Fig. 41.2.

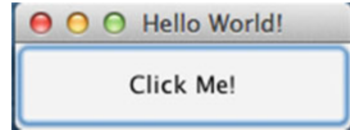

Fig. 41.2 Simple Hello World UI

41.6.2 Panels and UI Layout

The example shown in the last section works as a simple user interface example; however, the whole of the display is taken up with a single button. User interfaces are normally comprised of multiple components displayed within a single frame. This is achieved in Scala Swing using appropriate (potentially nested) containers as discussed earlier.

In this section, we will look at using a simple panel, a FlowPanel that allows multiple components and/or contains to be contained within it. The result of executing our program is shown in Fig. 41.3. The FlowPanel provides a flow-like layout with user interface components being added to the contents buffer of the FlowPanel.

In the following listing, we create two buttons (b1 and b2). These buttons are added to the contents of the FlowPanel when we create the panel. Notice that the FlowPanel instance is then set as the content for the main frame's display.

```
package com.jjh.scala.swing

import scala.swing.Button
import scala.swing.FlowPanel
import scala.swing.MainFrame
import scala.swing.SimpleSwingApplication

class ButtonPanel extends MainFrame {
  val b1 = new Button { text = "Click Me!" }
  val b2 = new Button { text = "Click Him!" }

  val panel = new FlowPanel {

   contents += b1
   contents += b2
  }

  contents = panel
}

object SwingHelloWorld2 extends SimpleSwingApplication {
  def top = new ButtonPanel()
}
```

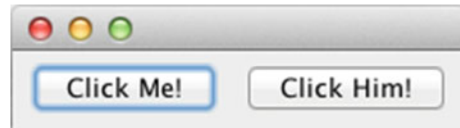

Fig. 41.3 Using a FlowPanel

Note that we could of course have created the FlowPanel and then added the buttons to the FlowPanel's contents buffer at a later date, for example

```
val panel = new FlowPanel()
panel.contents += b1
panel.contents += b2
```

You may be wondering at this point how the FlowPanel knew where to position the buttons—we did not give it absolute *X* and *Y* coordinates.

The actual positioning of the components is handled by an object used by the panel. This object is known as a layout manager. A layout manager is thus the object that works with a graphical application and the host platform to determine the best way to display the objects in the window. The programmer does not need to worry about what happens if a user resizes a window and works on a different platform or a different windowing system.

Layout managers help to produce portable, presentable user interfaces. There are a number of different layout managers that use different philosophies to handle the way in which they lay out components: `FlowLayout`, `BorderLayout`, `GridLayout`. Note that if you are familiar with Java's Swing library, then you will note that in Scala the layout managers, that handle actually determining where components are placed, are combined with the panels rather than being separate entities. Thus, we have FlowPanel, GridPanel, BorderPanel, etc.

41.6.3 Working with a BorderPanel

FlowPanel is not the only panel type available to the Scala programmer. Another type of panel is the BorderPanel. The BorderPanel possesses a layout manager that has a concept of four outer points and a central point (labelled North, East, South, West and Centre).

The panel is thus divided up as illustrated in Fig. 41.4. Of course, you do not have to place components at all available locations. If you omit one (or more) locations, the others stretch to fill up the space (except centre which depends on the size of the window in which the BorderPanel is being used). The border panel honours the height of the components in the north and south regions (but forces

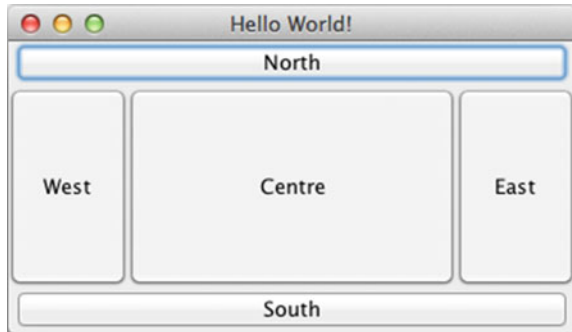

Fig. 41.4 Using the BorderPanel in a UI

them to fill the region horizontally). In turn it honours the width of the components in the East and West regions (but forces them to fill the available vertical space). The centre component is forced to fill the remaining space (thus its preferred width and height are ignored).

```
package com.jjh.scala.swing

import scala.swing.Button
import scala.swing.MainFrame
import scala.swing.SimpleSwingApplication
import scala.swing.BorderPanel

class WelcomeFrame extends MainFrame {
  title = "Hello World!"

  val b1 = new Button { text = "Centre" }
  val b2 = new Button { text = "North" }
  val b3 = new Button { text = "South" }
  val b4 = new Button { text = "East" }
  val b5 = new Button { text = "West" }
  val panel = new BorderPanel {
    add(b1, BorderPanel.Position.Center)
    add(b2, BorderPanel.Position.North)
    add(b3, BorderPanel.Position.South)
    add(b4, BorderPanel.Position.East)
    add(b5, BorderPanel.Position.West)
  }

  contents = panel

}

object SwingHelloWorld4 extends SimpleSwingApplication {
  def top = new WelcomeFrame()
}
```

In the above program, we create five buttons that are positioned within the BorderPanel using the position constraints, Centre, North, South, Each and West.

41.6.4 Working with a BoxPanel

Yet another panel is the BoxPanel; this is similar to the FlowLayout except that it can have a Horizontal or a Vertical orientation. In which case it either lays out components across the screen or down the screen depending upon the orientation. It therefore offers greater flexibility than the FlowPanel.

The following program uses the Horizontal orientation to create a display similar in style to the FlowLayout. The display contains a button and a label and is shown in Fig. 41.5.

```
package com.jeh.scala.swing

import scala.swing.BoxPanel
import scala.swing.Button
import scala.swing.Label
import scala.swing.MainFrame
import scala.swing.Orientation
import scala.swing.SimpleSwingApplication

/**
  * Orientation example
  */
object SwingHelloWorld5  extends SimpleSwingApplication {
  def top = new MainFrame {
    title = "Swing App"
    val button = new Button {
      text = "Click me"
    }
    val label = new Label {
      text = "Hello"
    }
    contents = new BoxPanel(Orientation.Horizontal) {
      //contents = new BoxPanel(Orientation.Vertical) {
      contents += button
      contents += label
    }
  }
}
```

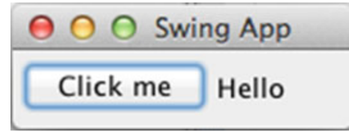

Fig. 41.5 Using a BoxPanel

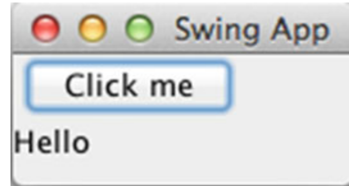

Fig. 41.6 An alternative orientation for the BoxPanel

Note that we change the orientation of the BoxPanel such that it has vertical orientation as shown below:

```
contents = new BoxPanel(Orientation.Vertical) {
  contents += button
  contents += label
}
```

This results in the display altering as shown in Fig. 41.6. Now the button is positioned above the label rather than beside it.

41.6.5 Displaying a Table

Another common user interface component is the Table. Tabes are user to present tabular information using rows and columns. A simple example table is shown in Fig. 41.7. This table has a row containing the headings (which are 'name', 'county'

Hello World!

Name	County	Town
John	Glamorgan	Cardiff
Bob	BANES	Bath
Adam	S.Glos	Bristol
Phoebe	Haverfordwest	Pembrokshire

Fig. 41.7 Using a Table in a UI

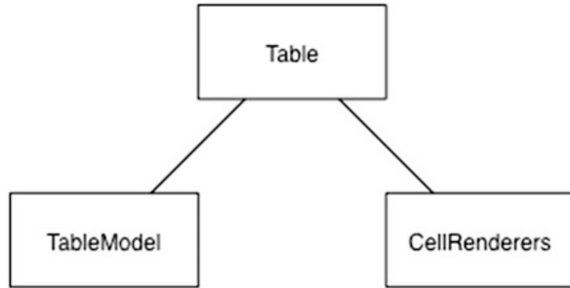

Fig. 41.8 Relationship between Table, TableModel and CellRenderers

and ‘town’) and four rows with the actual data. Such tables can respond to user selection, editing and ordering. However, in this example, it is a read only table that cannot be altered.

To create a table, we use the scala.swing.Table class. This can be constructed with an array of names and an array of arrays for the data. Strictly speaking behind the table is a table model that holds the actual data (see Fig. 41.8). There are also cell renderers that are used to determine how to display different types of data. In the following listing, we are merely using two arrays as a very simple way of initialising these models.

The following listing creates a new Table using the headers and rowData arrays. The border property of the Table is set to be a LineBorder coloured in Black. This is from the javax.swing.border package which is part of the underlying Java Swing library that is being reused here. Note that the Colour Black is from the underlying java.awt package that includes the Colour type.

Once the table is created, it added the FlowPanel which is used for the main contents of the window. Note that the table is wrapped within a ScrollPane. This provides any scrollbars for the table if they are required.

```
package com.jeh.scala.swing

import java.awt.Color

import scala.swing.FlowPanel

import scala.swing.MainFrame
import scala.swing.ScrollPane
import scala.swing.SimpleSwingApplication
import scala.swing.Table

import javax.swing.border.LineBorder

class TableFrame extends MainFrame {

  title = "Hello World!"
  val headers = Array("Name", "County", "Town")
  val rowData =
    Array(
      Array[Any]("John", "Glamorgan", "Cardiff"),
      Array[Any]("Bob", "BANES", "Bath"),
      Array[Any]("Adam", "S.Glos", "Bristol"),
      Array[Any]("Phoebe", "Haverfordwest",
                              "Pembrokshire"))

  val table = new Table(rowData, headers) {
    border = new LineBorder(Color.BLACK)
  }
  contents = new FlowPanel {
    contents += new ScrollPane(table)
  }
}

object TableSample extends SimpleSwingApplication {
  def top = new TableFrame()
}
```