

# Chapter 42

## User Input in Scala Swing



### 42.1 Introduction

The last chapter looked at various different types of user interface component available within Scala Swing. This chapter now looks at how user input, via those components, can be handled.

### 42.2 Handling User Input

If you are familiar with the Java Swing event delegation model, then you will find the Scala approach to handling user input, a great deal simpler. It is based on the idea of using pattern matching within reactors to handle user input. Objects publish events that are listened to by a list of reactors. A `Reactor` is a `Partial Function` that is defined by the `Type Reaction` on the `Reactions` object to take an `Event` and return `Unit`.

If a reactor matches the event type and the source specified, then the associated behaviour is invoked. This framework is based on the classes that emit events mixing in the `Publisher` trait and classes that listen to publishers mixing in the `Reactor` trait.

The `Publisher` trait defines the `publish` method, the `listeners` property, and invokes the `listenTo` behaviour. The `listeners` property holds a list of reactors that will react to the events raised by the source component. An event is just an object containing information associated with the action that occurred. For example, a `ButtonEvent` indicates the source object that the user clicked on, whereas a `MouseEvent` may include the `x` and `y` coordinates of the mouse when it was clicked or moved. The `publish` method is defined as

```
def publish(e: Event) { for (l <- listeners) l(e) }
```

This sends the event 'e' to all the members of the listeners list.

The `Reactor` trait defines two methods `deafTo` and `listenTo` and the property reactions. The methods are defined as:

- `def deafTo(ps: Publisher*): Unit`
- Installed reaction will not receive events from the given publisher any longer.
- `def listenTo(ps: Publisher*): Unit`
- Listen to the given publisher as long as `deafTo` is not called for them.

For example, for an instance to listen to the event generated by a button we could write:

```
listenTo(`button`)
```

The `reactions` property is an instance of a subclass of the `Reactions` abstract class. It defines a set of methods which allow the reactions held in the reactions list to be processed. It also defines the `+=` and `-=` methods that can be used to add or remove reactors from the list of reactors:

- `def +=(r: Reaction)` Add a reaction.
- `def -=(r: Reaction)` Remove the given reaction.

An example of using the `+=` method to add some reactors is given below:

```
reactions += {
  case ButtonClicked(`b1`) => println("Hello World")
  case ButtonClicked(`b2`) => println("North")
  case ButtonClicked(`b3`) => println("South")
  case ButtonClicked(`b4`) => println("East")
  case ButtonClicked(`b5`) => println("West")
}
```

In this example, we have added five reactors to the reactions list that handle behaviour on different buttons.

Thus to listen to user events on buttons, combo boxes, tables, menus, etc., it is necessary to mix in the `Reactor` trait and to register yourself with the UI components you wish to listen to. This is what the following examples do.

The first example relies on the fact that the `MainFrame` mixes in the `Reactor` trait. This means that you can define the reactions to a button within the body of the `MainFrame` itself. Also note that the `MainFrame` uses the `listenTo` method to register itself with the button. The user interface generated by the code is shown in Fig. 42.1.

Fig. 42.1 A button in a UI



```

package com.jeh.scala.swing

import scala.swing.Button
import scala.swing.MainFrame
import scala.swing.SimpleSwingApplication
import scala.swing.event.ButtonClicked

class SimpleFrame extends MainFrame {
  title = "Hello World!"
  val button = new Button {
    text = "Click Me!"
  }
  contents = button

  listenTo(button)

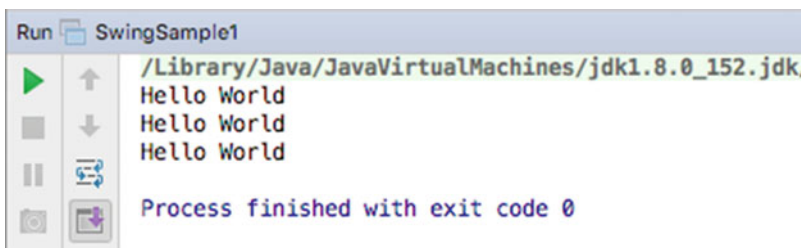
  reactions += {
    case ButtonClicked(button) =>
      println("Hello World")
  }
}

object SwingSample1 extends SimpleSwingApplication {
  def top = new SimpleFrame()
}

```

In the above code, the mainframe listens to the button for events that it is publishing. When those events are received, the reactors in the reactions list are checked in sequence to find one that will handle the event. In this case, the event is the `ButtonClicked` event. Thus when the user clicks the button the String Hello World should be printed out to the console. This is illustrated in Fig. 42.2.

One problem with this code is that the reactor is defined by the `MainFrame`. While this works for a simple application is it unlikely that this approach would work in a larger application. Therefore the following listing modifies this approach and defines a separate class `ButtonReactor` that mixes in the `Reactor` trait



```

Run SwingSample1
/Library/Java/JavaVirtualMachines/jdk1.8.0_152.jdk
Hello World
Hello World
Hello World
Process finished with exit code 0

```

Fig. 42.2 Output from the reactor when button clicked twice

and defines the reactions within itself. It is then instantiated and used by the `SimpleFrame` (`MainFrame`). Note that the reactor must be registered with the button by having the `listenTo` method called on it rather than on the `MainFrame`:

```

package com.jeh.scala.swing

import scala.swing.Button
import scala.swing.MainFrame
import scala.swing.SimpleSwingApplication
import scala.swing.event.ButtonClicked
import scala.swing.Reactor

class SimpleFrame2 extends MainFrame {
  title = "Hello World!"

  val button = new Button {
    text = "Click Me!"
  }
  val reactor = new ButtonReactor()

  contents = button

  reactor.listenTo(`button`)
}

class ButtonReactor extends Reactor {
  reactions += {
    case ButtonClicked(button) =>
      println("Hello World")
  }
}

object SwingSample2 extends SimpleSwingApplication {
  def top = new SimpleFrame2()
}

```

The actual user interface and the output remain unchanged.

### 42.2.1 *Scala Swing Actions*

An action can be used to separate the behaviour associated with a button (or menu item) from the instance of the button (or menu item) concerned. This can be useful as it is common within a user interface to have several ways to access the same operation, for example from a button bar, from a toolbar or from a menu item. Using an action, this behaviour can be defined once (in the Action) and then reused with each of the UI components that will be presented to the user. Thus an Action

separates the definition of some behaviour to be applied, from the UI component used to invoke that behaviour.

An action is defined using the `Action` type from the `scala.swing` package. The action can take a *title* and the functionality to be invoked when that action is used. Thus the following lines of code create an action with a title “Click Me” and the function to *apply* when the action is invoked (in this case to print a message to the console):

```
val myAction = Action("Click Me") {
  println("I was clicked")
}
```

This action instance can now be used with a range of UI components such as buttons and menu items. In the following listing we use this action with two buttons, `b1` and `b2`. Note that the action is used to initialise the action property of the buttons. Thus both buttons will invoke the same `println` function when clicked, wherever they are in the UI.

```
package com.jeh.scala.swing

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

class ButtonPanel extends MainFrame {

  val myAction = Action("Click Me") {
    println("I was clicked")
  }

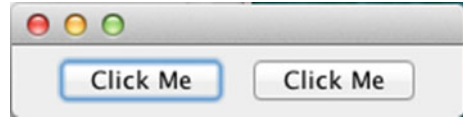
  val b1 = new Button { action = myAction }
  val b2 = new Button { action = myAction }

  val panel = new FlowPanel {
    contents += b1
    contents += b2
  }

  contents = panel
}

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

**Fig. 42.3** A UI that reuses an Action



The end result of using the `myAction` with both buttons is that there is a single definition of the action behaviour shared between the two button instances. Note that actions can also be enabled or disabled and have icons, tooltips, etc.

The display generated from the above listing is shown in Fig. 42.3. The result of clicking on either of the Click Me buttons is that the string “I was clicked” is printed to the standard output.

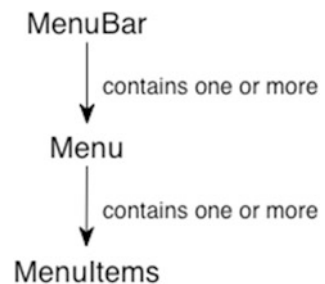
### 42.2.2 Working with Menus

Many (most) applications will have some aspect of a menu bar, menus and items on those menus. In Scala such user interface components are represented by instances of the classes `MenuBar`, `Menu` and `MenuItem`. The relationship between these components is illustrated in Fig. 42.4. These classes are all defined in the `scala.swing` package.

As can be seen a `MenuBar` references (holds) one or more `Menus`. `Menus` in turn reference one or more `MenuItems`. `MenuItems` can be either simple menu items that may be selected, or menus in their own right. This is because the `Menu` type extends the `MenuItem` type and thus we can create hierarchical menus.

The following listing illustrates how a simple `MenuBar`, with a single `Menu` (file), can be created. The `Menu` file has a single `MenuItem` (Exit) that is defined using an `Action`. Note that the `MenuBar` is used to set the `menuBar` property of the `MainFrame`. The `MenuBar` has a `contents` property, which we are adding the `Menu` to. Also note that the `Menu` has a `contents` property to which we are adding the `MenuItem` (be careful not to confuse these two).

**Fig. 42.4** Relationship between Menus, MenuBars and MenuItems



```

package com.jeh.scala.swing

import scala.swing.MainFrame
import scala.swing.SimpleSwingApplication
import scala.swing.Label
import scala.swing.MenuBar
import scala.swing.Menu
import scala.swing.MenuItem
import scala.swing.Action

class SimpleFrame extends MainFrame {
  title = "Hello World!"
  contents = new Label {
    text = "Hello"
  }
  menuBar = new MenuBar {
    contents += new Menu("File") {
      contents += new MenuItem(Action("Exit")) {
        sys.exit(0)
      }
    }
  }
}

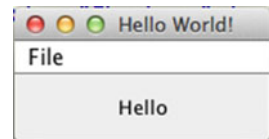
object SampleMenuUI extends SimpleSwingApplication {
  def top = new SimpleFrame()
}

```

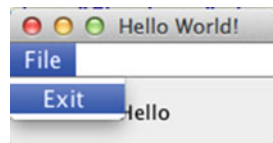
The result of executing this program is illustrated in Figs. 42.5 and 42.6. The first figure shows the basic `MenuBar` display with the File menu shown. The second figure illustrates what happens when the user moves their mouse over the File menu and selected the Exit Menu Item.

When the user selects the Exit option, as shown in Fig. 42.6, then the behaviour defined by the `Action` in the earlier listing is invoked. In this case it is to call the `sys.exit(0)` operation. This invokes the exit behaviour on the system with a return code of '0', which typically indicates that the application terminated

**Fig. 42.5** A UI with a Menu Bar



**Fig. 42.6** Selecting the 'Exit' Menu Item



normally. Explicitly invoking the `sys.exit` operation is necessary as the `main` method, which is usually used to control when an application terminates and only used to initiate the display. After that, the main method terminates and the execution of the system is handed over to a UI thread (process). We must therefore be able to terminate this process in a controlled manner; this is done using the `sys.exit` operation.

### 42.3 A Simple GUI Example

In this section we present a very simple GUI example. An instance of this class generated the window displayed in Fig. 42.7. This application performs the following functions:

- Displays the string “Hello” in a text field in response to the user clicking on the Hello button.
- Displays the string “Goodbye” in a text field in response to the user clicking on the Goodbye button.
- Exits the application in response to the user clicking on the Exit button.

It combines the layout panels and components presented in the last chapter, with the event handling mechanism described above.

The components that comprise this user interface are illustrated in Fig. 42.8. This shows that the buttons are organised (displayed by) a flow panel (which does not itself have any visible presence). The strings are displayed in a text field, and a label displays the copyright statement. These are all organised within a border panel that is the top-level contents of the `SimpleGUI MainFrame`.

The class `SimpleGui`, see the listing below, first sets the title of the frame to be “Simple GUI”. It then creates two buttons, which have a text label and a tooltip. It then creates a new `FlowPanel` to which it adds two buttons. The program then creates a non-editable field—note that the `editable` property is set to `false` after the text field is created—this allows this property to change its value over time. It finally creates a label, for the copyright string. This label uses a new font (the `Arial` font) and introduces a buffer to give a bit of spacing around the text using the `scala.swing.Swing` utility type. The button panel, text field and label are



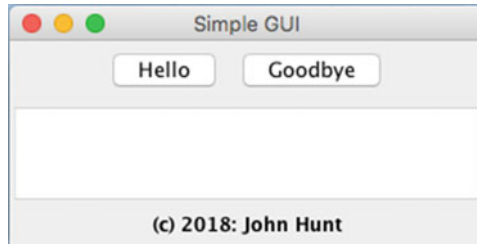


Fig. 42.7 A simple graphical application

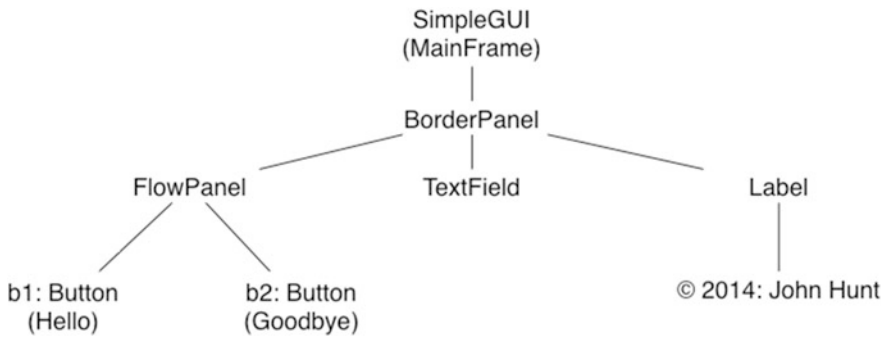


Fig. 42.8 Structure of simple GUI application

added to the border panel using the constraints `Position.North`, `Position.Center` and `Position.South`, respectively. Note that we have imported `scala.swing.BorderPanel._` so that we only need to specify `Position.North` rather than `BorderPanel.Position.North` which is easier to read. Finally the window sets a default size using a new `Dimension` instance.

```

package com.jeh.scala.sample

import java.awt.Font
import java.awt.Dimension

import scala.swing._
import scala.swing.BorderPanel._
import scala.swing.event.ButtonClicked

class SampleGui extends JFrame {
  title = "Simple GUI"
  // Set up the buttons
  val b1 = new Button {
    text = "Hello"
    tooltip = "Click to say hello"
  }
  val b2 = new Button {
    text = "Goodbye"
    tooltip = "Click to say goodbye"
  }

  // Set up panel for buttons (using a flow layout)
  val panel = new FlowPanel {
    contents += b1
    contents += b2
  }
  // Create a non-editable text field and add that
  val field = new TextField()
  field.editable = false

  // Create a label for the frame
  val label = new Label() {
    text = "(c) 2018: John Hunt"
    border = Swing.EmptyBorder(5,5,5,5)
    font = new Font("Ariel", Font.BOLD, 12)
  }

  // Setup listening to the buttons and reactions
  listenTo(b1, b2)
  reactions += {
    case ButtonClicked(`b1`) => field.text = "Hello"
    case ButtonClicked(`b2`) => field.text = "Goodbye"
  }

  contents = new BorderPanel() {
    // Add the button panel to the frame
    add(panel, Position.North)
    // Add the text field to the centre
    add(field, Position.Center)
    // Add the label to the bottom of the display
    add(label, Position.South)
  }

  // Resize the window
  size = new Dimension(300, 150)
}

object SampleGui extends SimpleSwingApplication {
  def top = new SampleGui()
}

```

Note that the two buttons are listened to and thus we define two reactors to handle what should happen when the user clicks on a button. In this case we set the `field.text` property to the appropriate string. You could have a different controller for each button. In such a situation, you do not need to test to see which button generated an event (thus eliminating the `case` pattern matching statement that selects the actual behaviour to perform).