Timber Interface to the Cocoa Library: Tutorials

Daniel Öhrlund and Victor Danell

December 14, 2011

Tutorial 0: The Cocoa App

This tutorial will teach you how to create a Cocoa application in Timber and how to create and display a window. We will start by looking into how we can interact with the Cocoa environment to launch an application.

First of all, an object of the class *cocoa* must be instantiated. Through the interface of this object we can create a Mac OS X Cocoa application by invoking the method *startApplication*.

When you call *startApplication*, the Cocoa environment will create an application (App) for you. Once the App is ready, the method of type *(App -> Action)* passed to *startApplication* will be invoked with the App as a parameter. This App object is the only interface to your newly created Cocoa application. Here is how it looks in code:

```
module Tutorial0App where

import COCOA

root w = class
   osx = new cocoa w

   start app = action
       {-
          Inside this method you can use the app
          object to open windows, add components,
          etc.
       -}

   result action
       osx.startApplication start
```

With a Cocoa application up and running, it is now time to learn more about GUI objects. One thing that all GUI objects have in common is the notion of state. At any given time, every object is in one of these three states:

- *Inactive* - the initial state of any GUI object, when it has not yet been added to an App;
- *Active* - the state of a GUI object currently active in an App;
- *Destroyed* - the state of an object that has been destroyed and cannot be used again.

There are two categories of GUI objects: windows and components. Since a component cannot be displayed without being contained within a window, we will look at windows first. Displaying a window consists of two steps, both of which are quite straightforward: the first step is to create a window and the second step is to add it to an application.

To instantiate a Timber object representing a Cocoa window we write

```
myWindow = new mkCocoaWindow w
```

where w is a World reference. This gives us an interface through which we can modify the properties of the window. When the window is ready to be added to your application, you can use the *addWindow* method of the App object. Invoking this method will create a window in Cocoa and pair it with your Timber object, so that it can still be modified from Timber. It is also at this time of initialization that the window object will have its state set to *Active*. The final code for this tutorial looks as follows:

```
module Tutorial0App where

import COCOA

root w = class
    osx = new cocoa w
    w1  = new mkCocoaWindow w

    start app = action
        w1.setPosition ({x=100;y=100})
        w1.setSize ({width=400;height=400})
        w1.setBackgroundColor web_gray
        w1.setTitle "Tutorial"
        app.addWindow w1

    result action
        osx.startApplication start
```

Tutorial 1: Component hierarchies and event handlers

In Tutorial 0 we learned how to create a GUI object and in this tutorial, we will learn how to create multiple components and add them to a component hierarchy. We will also look at how an application can respond to events via handlers installed on components. But first, component hierarchies.

When programming a GUI, it is usually possible to add components to other components. For instance, a button might be added to a container, which in turn belongs to a window. We also support such hierarchical structures.

We have implemented two components that may contain other components, windows and containers. A CocoaWindow actually uses a *root container* to store added components, which covers the whole window area except window decorations. This is transparent to the programmer as the interface of a window allows you to add and remove components directly.

A container is a special component that can contain zero or more other components. It also has its own coordinate system, it has a size, a position and a background color. In the example below, we will show you how to create two containers: one that contains a button and one that contains a label. Both of these containers will be added to the same window where they will be placed side by side.

We will start by creating two containers and setting up some properties, after which we create a button and a label:

```
leftContainer = new mkCocoaContainer w
leftContainer.setSize ({width=200;height=200})
leftContainer.setPosition ({x=0;y=0})
leftContainer.setBackgroundColor ({r=100;b=100;g=200})

rightContainer = new mkCocoaContainer w
rightContainer.setSize ({width=200;height=200})
rightContainer.setPosition ({x=200;y=0})
rightContainer.setBackgroundColor ({r=100;b=200;g=100})

button.setTitle "Click me!"
button.setSize ({width=110;height=21})
button.setPosition ({x=40;y=100})

label.setText "This is a label"
label.setSize ({width=100;height=36})
label.setPosition ({x=40;y=100})
```

We will now add the button to the left container, a label the label to the right container and add both containers to our window. The window is then added to the App:

```
leftContainer.addComponent button
rightContainer.addComponent label
w1.addComponent leftContainer
w1.addComponent rightContainer
app.showWindow w1
```

But what if you want to handle the events that occur when someone clicks the button? We will show you how to install a click responder on the button that will change the label's text when the button is clicked.

We will start with defining a new class for the event handler:

```
buttonHandler :: Label -> Class Action
buttonHandler label = class
   clickCount := 0
   result action
       clickCount := clickCount + 1
       label.setText ("Click #" ++ show clickCount)
```

As you can see in the code above, the resulting action will on each invocation update the text of the label given as a parameter.

The next thing we need to do is to instantiate a new object of this class and install it on the button:

```
button.setClickResponder (new buttonHandler label)
```

This should be it! Try running your program and click a couple of times on the button.

Tutorial 2: Re-sizing windows (and more event responders)

In this tutorial we will introduce two new things: resizing events and text areas that supports scrolling. We will start by adding a text area to our window:

```
ta = new mkCocoaTextArea w
ta.setSize ({width=300;height=80})
ta.setPosition ({x=50;y=250})
ta.setDocumentSize ({width=400;height=800})
w1.addComponent ta
```

The next step will be to install an event responder on the window that changes the size of the text area when the window is resized, so that the ratio between the size of the window and the text area is kept constant. This is done by defining a new window event responder class:

```
windowResponder textarea = class
    onWindowResize size modifiers = request
        newWidth = floor ((fromInt size.width) * 0.8)
        newTaSize = {width=newWidth; height=80}
        newX = floor ((fromInt size.width) * 0.1)
        newTaPosition = {x=newX; y=250}
        textarea.setSize newTaSize
        textarea.setPosition newTaPosition

    onWindowCloseRequest = request

    result RespondsToWindowEvents {..}
```

As in the previous tutorial, this class takes the interface of a text area that the responder will control as a parameter. Once this class has been defined, all we have to do is install an instance of this class as a window responder:

```
window.setWindowResponder (new windowResponder ta) False
```

The second parameter *False* specifies that we do not override window close events, i.e. once the responder has executed *onWindowCloseRequest*, the window will be closed.

Try running the program and re-size the window!

Tutorial 3: Key events and the focus hierarchy

An important concept when talking about key events is that of focus. In every window, there is exactly one component which has focus, and when a window receives a key event it will be passed to the *respondToInputEvent* method of that component. Whether or not a particular component instance can be focused depends on the return value of its *getIsFocusable* method. An empty window or a window containing no components that can be in focus, will assign focus to its root container.

There are three ways to change focus in a window: by pressing the Tab key, directly calling the *setFocus* method on the window with a component as a parameter, and by clicking on a component which can have focus but currently does not. In the case of the Tab key being pressed, the focus change can be prevented if the component currently in focus consumes the event. If the event is not consumed, focus will go to the next component in the focus hierarchy, which is a list of all focusable components in a window.

Each window created with *mkCocoaWindow* has its own default responder installed, which is responsible for keeping track of and changing focus. Whenever the left mouse button is clicked or the Tab key is pressed, the responder will update the focus hierarchy and figure out which component should receive focus next. To see how this works, run Tutorial2Resizing or TestFocusHierarchy and press Tab a few times.

Although this default behavior is often what we want, overriding it when needed is easy. The default focus responder begins by asking the component currently in focus if it wishes to consume the event, and continues only if it does not. This means all we have to do to modify the response to a Tab key press, is to install a custom responder, in which the *respondToInputEvent* method returns True whenever it wishes to stay in focus.

As an example of this second scenario, we will now modify text area in Tutorial2 to consume the tab event, thereby preventing it from losing focus when Tab is pressed. All we need to do to achieve this is write a custom responder which consumes KeyEvents, and install it on the text area. The custom responder may look like this (in this case, the responder will also update a label passed to it as a parameter):

```
myTabResponder :: Label -> Class RespondsToInputEvents
myTabResponder label = class
    tabCount := 0

    respondToInputEvent (KeyEvent (KeyPressed Tab)) _ = request
       tabCount := tabCount + 1
       label.setText ("Tabs blocked #" ++ (show tabCount))
       result Consumed

   respondToInputEvent _ _ = request
       result NotConsumed

   result RespondsToInputEvents {..}
```

To install our custom responder, we call the *addResponder* method on a component of our choice. Here is how it looks when installing it on our text area:

```
tabCountLabel = new mkCocoaLabel w
tabCountLabel.setText "Tab counter"
tabCountLabel.setSize ({width=150;height=36})
tabCountLabel.setPosition ({x=40;y=70})
ta.addResponder (new myTabResponder tabCountLabel)
rightContainer.addComponent tabCountLabel
```

Tutorial 4: Multiple windows

In this tutorial we will see how multiple windows can communicate with each other once they have been activated. Continuing on the project from the previous tutorials, we will now add a second button, which opens a new window with a color palette (made of multiple tiles). Each of these tiles will, when clicked, change the text of a label in the original window to the RGB code of their respective color and change the background color of the left container. If *Shift* is held down, mouse move events will also update the label text and the background color.

We create a second window instance and add it to the App by calling the App's method *addWindow* again. The other components are created in a manner similar to previous tutorials:

```
rgbLabel       = new mkCocoaLabel w
colorButton    = new mkCocoaButton w
colorWindow    = new mkColorPicker w setColor

addColorPicker app = do
    rgbLabel.setText "R=100; G=100; B=200"
    rgbLabel.setSize ({width=150; height=36})
    rgbLabel.setPosition ({x=40; y=40})
    rightContainer.addComponent rgbLabel

    colorButton.setTitle "Open ColorPicker"
    colorButton.setSize ({width=150;height=21})
    colorButton.setPosition ({x=40; y=75})
    colorButton.setClickResponder (new mkColorToggle colorWindow colorButton)
    leftContainer.addComponent colorButton

    colorWindow.setPosition ({x=500;y=100})
    app.addWindow colorWindow
```

The only thing which stands out here is the instantiation of our *mkColorPicker*. This object will, once initialized, contain the color tiles. Its definition is found in the *Tutorial4ColorPicker* module (see Appendix). Upon creation of the color picker grid, each tile is assigned a position, which in turn determines the background color of the tile. An input responder is also added to each tile so that when it's clicked on or when the mouse is moved over it with *Shift* pressed, a callback is invoked with the new background color as a parameter.

Tutorial 5: Customizing behavior of a component

In this last tutorial we will customize the behavior of a component, namely a label. We will define a new class that modifies a Class Label, so that on each update to its text it will invoke the specified callback method:

```
mkCocoaCallbackLabel :: (Class Label) -> (String->Action) -> Class Label
mkCocoaCallbackLabel mkLabel cb = class

    Label {setText=setTextImpl;appendText=appendTextImpl;..} = new mkLabel

    setText s = request
        setTextImpl s
        send cb s

    appendText s = request
        appendTextImpl s
        send cb (<- getText)

    result Label{..}
```

As seen in the code above, we first create an ordinary Label with the given class (*mkLabel*) and then define two new methods: *setText* and *appendText*. Both these methods wrap their equivalents on the regular label but also perform the given callback.

The interface returned by this new class will also be of type Label, and will be composed of all the methods from the original Label apart from *setText* and *appendText* (named *setTextImpl* and *appendTextImpl* in the code example), which are replaced with our new implementations of *setText* and *appendText*. These are all the changes needed for the customized behavior we want!

To see this new customized component in action we will replace the label that shows the click count and the label that shows the tab block count with callback labels. After that, we will add a third label to display callbacks from the two counting labels.

```
label = new mkCocoaCallbackLabel (mkCocoaLabel w) textChangeCallback

tabCountLabel = new mkCocoaCallbackLabel (mkCocoaLabel w) textChangeCallback

textChangeCallback newText = action
    callbackLabel.setText ("CB: " ++ newText)

callbackLabel = new mkCocoaLabel

addCallbackLabel = do
    callbackLabel.setSize ({width=150; height=36})
    callbackLabel.setPosition ({x=40; y=10})
    callbackLabel.setText "Callback label"
    rightContainer.addComponent callbackLabel
```

Now, each time the text changes on tabCountLabel or label, callbackLabel will have its text set to "CB:" followed by the text sent to the callback. Try running the program and trigger some callbacks by either clicking the button or pressing the tab key while the text area is focused!

Appendix - Tutorial4ColorPicker

```
module Tutorial4ColorPicker where

import COCOA

mkColorPicker :: World -> (Color -> Action) -> Class CocoaWindow
mkColorPicker w callback = class
    CocoaWindow{initWindow=initWindowImpl;..} = new mkCocoaWindow w

    initWindow app = request
        setSize ({width=215;height=215})
        setVisible False
        setResizable False
        setWindowResponder (new class
            onWindowResize _ = request
            onWindowCloseRequest = request
            result RespondsToWindowEvents{..}) True
        initGrid
        initWindowImpl app

    initGrid = do
        tileSize = 12
        forall x <- [1..16] do
            forall y <- [1..16] do
                tile = new mkCocoaContainer w
                tile.setSize ({width=tileSize;height=tileSize})
                tile.setPosition ({x=tileSize*x;y=tileSize*y})
                tileColor = ({r=128;g=16*x;b=16*y})
                tile.setBackgroundColor tileColor
                tile.addResponder ({respondToInputEvent=invokeCallback tileColor})
                addComponent tile

    invokeCallback color (MouseEvent (MouseClicked _)) _  = request
        send callback color
        result Consumed
    invokeCallback color (MouseEvent (MouseMoved _)) modifiers = request
        if elem Shift modifiers then send callback color
        result Consumed
    invokeCallback _ _ _ = request
        result Consumed

    result CocoaWindow{..}
```