

5. Rich Graphical User Interface DSL

To evaluate the ability of Scala and Groovy to be DSL hosts, we created a small JavaFX like DSL for Rich Graphical User Interface (GUI) creation. The DSL provides features that facilitate GUI development such as dynamic updates of properties based on other properties, thread utilities and an operator that aids in creating animations.

5.1. DSL in Scala

The DSL created in Scala was accomplished using the Implicit conversion (Views) feature of Scala.

5.1.1. Incremental dependency-based Evaluation (Bind)

The bind feature of the DSL is similar to JavaFX bind discussed in section 2.1.1

5.1.1.1. Bound Property

We created a new type of property called *BoundProp*. This is a generic property and can be declared with any type. It provides the ability to update a variable automatically based on the value of another variable. To understand how bound properties can be used let us first consider a trivial example.

```
import boundUtilities.BoundProp;

class Person(str: String){ //class declaration and constructor
    var name = new BoundProp[String];
    name := str;
}

object BindTester {
    def main(args: Array[String]) : Unit = {

        val user = new Person("Voltaire");
        var name = new BoundProp[String];

        //bi-directional binding between user.name and name
        name bind user.name;
        println(name); //prints "Voltaire"

        user.name := "Homer";
        println(name); //prints "Homer"
    }
}
```

Listing 5-1: Bi-directional Binding of two properties

In the above example variable *name* is bi-directionally bound to *user.name* and one gets automatically updated whenever the value of the other changes.

Given below is an excerpt from the *BoundProp* class that shows the implementation of bind. We needed to fire a property change event when the value of a *BoundProp* is

updated in order to notify other properties bound to it. For this reason we created the `:=` operator that is used to assign a value to the `BoundProp`. This operator sets the new value of the `BoundProp` and also fires the property change notifications. The above example uses the `:=` operator to assign the new value to `user.name`.

```
class BoundProp[T] extends PropertyChangeSupport with
PropertyChangeListener{

    var value: T = _;

    //bi-directional binding done between "this" and "other"
    def bind(other: BoundProp[T]){
        this.addPropertyChangeListener(other);
        other.addPropertyChangeListener(this);
        if(other.value != null){
            this := other.value;
        }
    }

    //update "this" on change notification from a property bound to it
    def propertyChange(evt: PropertyChangeEvent){
        //using "this :=" rather than "this =" so that its property
        //change event is fired notifying other properties bound to it
        if(!(evt.getNewValue().asInstanceOf[T]).equals(value)){
            this := evt.getNewValue().asInstanceOf[T];
        }
    }

    //overloaded operator for assignment with RHS value of type T
    def :=(newValue : T): BoundProp[T] = {
        val oldValue = value;
        value = newValue;
        firePropertyChange("value", oldValue, newValue);
        this;
    }

    //overloaded assignment operator with RHS of type BoundProp[T]
    def :=(newProperty : BoundProp[T]): BoundProp[T] = {
        val oldValue = value;
        value = newProperty.value;
        firePropertyChange("value", oldValue, value);
        this;
    }
    ...
}
```

Listing 5-2: Bind implementation in BoundProp

5.1.1.3. Bound Swing Components

In the last section we considered a trivial case (Listing 5-1) to show the usage and working of the bind operator of properties. The real value of bind operator can be seen in the context of Model-View-Controller pattern. Here the bind operator is used to bind the *view* (user interface) of an application to the *model* (data) of the application. Consider the following example where a model's *author* attribute is bound to a swing component.

```

import boundUtilities.BoundJComboBox._;

class Author(n: String) {
  var name = n;
  var books: String = _;
}

class Model{
  var author = new BoundProp[String];
}

object View {
  def main(args: Array[String]) : Unit = {
    ...
    val model = new Model;
    val author1 = new Author("Jane Austen");
    val author2 = new Author("Charlotte Bronte");
    val authors: Array[Object] = Array(author1.name, author2.name);

    val authorList = new JComboBox(authors);

    //Implicitly converting model.author from BoundProp[String]
    //to BoundProp[Object]
    authorList.selectedItem bind model.author;
    model.author := "Charlotte Bronte";
    ...
  }
}

```

Listing 5-3: Binding of model's attribute to View's component

In our example, when the model's data changes the view is automatically updated. The **selectedItem** of the **authorList** is bound to the **author** attribute of the Model class using the **bind** operator. This will create a bi-directional binding between the two and change in one will be reflected immediately in the other. This eliminates the boilerplate of writing listeners for notification of updates in swing components and the corresponding code to update the attributes of the model. Notice the special syntax of import statement at the top, this is the statement that makes available our DSL's bind feature to the user. More detail of how bind is implemented in swing components and how this special import statement works is given at the end of this section.

Our DSL also allows binding between two swing components. We can take the above example and extend it to include binding between the **authorList** combo box and a list.

```

import boundUtilities.BoundJComboBox._;
import boundUtilities.BoundJList._;

class Author(n: String) {
  var name = n;
  var books: String = _;
}

class Model{
  var author = new BoundProp[String];
}

```

```

}
object BindTester {
  def main(args: Array[String]) : Unit = {
    ...
    val model = new Model;

    val author1 = new Author("Jane Austen");
    author1.books = "Pride and Prejudice, Emma, Persuasion";

    val author2 = new Author("Charlotte Bronte");
    author2.books = "Jane Eyre, Shirley, Vilette, The Professor";

    val authors: Array[Object] = Array(author1.name, author2.name);
    val books: Array[Object] = Array(author1.books, author2.books);

    val authorList = new JComboBox(authors);
    val booksList = new JList(books);

    authorList.selectedItem bind model.author;
    authorList.selectedIndex bind booksList.selectedIndex;

    model.author := "Charlotte Bronte";
    ...
  }
}

```

Listing 5-4: Binding of two swing components

When the *author* is selected from the combo box, the selected item in the list updates to show the *booksList* by that author. This binding is also bi-directional so changing list selection will update the *author* in the combo box as well. Below is an screenshot of the demo application.

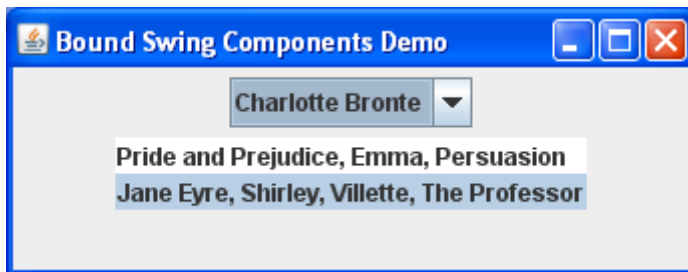


Image 1: Screenshot of Swing component binding demo

The bind feature is added to existing swing components by creating bound components for each existing swing component that contain the added functionality. The code for BoundJTextField, an extension class to JTextField is given below.

```

package boundUtilities;

class BoundJTextField(textField: JTextField){
  var text = internalBind(); //text internally bound to textField's
                             //text property

  //bi-directionally bind textField's text to local variable text
}

```

```

private def internalBind(): BoundProp[String] = {
  var temp = new BoundProp[String];

  //Update text whenever the textField's text property changes
  textField.addActionListener(new ActionListener{
    def actionPerformed(e:ActionEvent): Unit = {
      temp := textField.getText();
    }
  });

  //Update textField's text property when local text changes
  temp.addPropertyChangeListener(new PropertyChangeListener{
    def propertyChange(evt: PropertyChangeEvent){
      if(!textField.getText().equals(evt.getNewValue())){
        textField.setText(evt.getNewValue().asInstanceOf[String]);
      }
    }
  });
  return temp;
}
//companion object
object BoundJTextField {
  implicit def JTextField2BoundJTextField(tf: JTextField) =
    new BoundJTextField(tf)
}

```

Listing 5-5: Bind implementation for JTextField's companion class

The class *BoundJTextField* has an attribute *text* of type *BoundProp[String]*. This is the attribute that user's bind to, when they call bind on a *JTextField's* text property. Internally, it takes care of bi-directional updates between the *JTextField's* text property and itself. To use the Views feature of Scala we created a companion object *BoundJTextField* with the same name as our class, as shown above, and declared a method for implicit conversion from *JTextField* to our *BoundJTextField*. Now we can use the special import syntax with the “_” to allow implicit conversion of *JTextField* to *BoundJTextField*. The user can now use the bind feature as if it was a part of *JTextField*, as shown below.

```

import boundUtilities.BoundJTextField._;
{
  val field1 = new JTextField;
  val field2 = new JTextField;

  //implicitly converting field1 and field2 to BoundJTextField
  field1.text bind field2.text;
}

```

For details on Views and companion objects see section 3.1.7

5.1.1.2. Bound Bean Property

5.1.1.4. Bind with Expressions