Java, Swing, and Eclipse: The Calculator Lab.

ENGI 5895. Winter 2014

January 13, 2014

1 A very simple application

(Some images were prepared with an earlier version of Eclipse and may not look exactly as they would with the version you are using.)

1. Start Eclipse

   - If you get a 'workspace not available message,' click OK
   - Set the workspace to somewhere on the H: drive. (Or if you are using your own computer, wherever you like.)
   - Note the workspace directory – you will need to find it later.
   - Click Ok.
   - Close the 'Welcome' window if there is one.
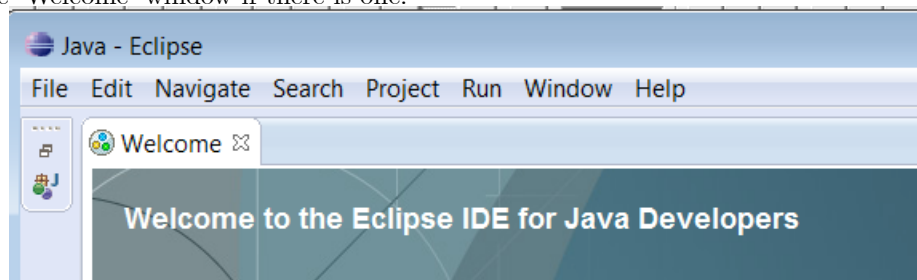
2. Create a new project (see Fig 1)

   - On the Menu select "File / New / Project ...".
   - Select "Java Project" and click "Next".
   - Set "Project Name" to "Calculator"
   - Set the execution environement to JavaSE-1.7.
   - Select "Create separate folders for source and class files"
   - Click Finish.
   - (If you get a message to the effect that EGit can't be found, just click OK. If there is a further complaint, click OK again.

- In the "Package Explorer" view (on the left of the window) you should see your project."

3. Create a class (see Fig 2)

   - Right-click on the Calculator project in the Package Explorer
   - On the context menu select "New / Class".
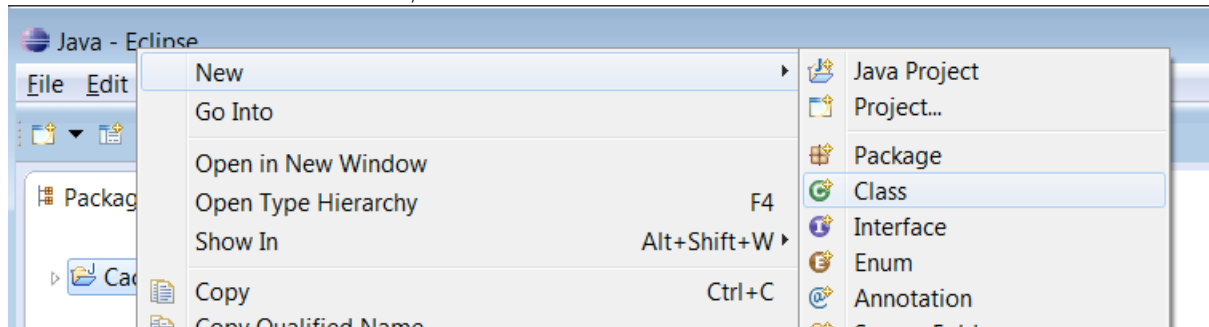
   - Set 'Source folder' to Calculator/src
   - Set the 'Package' to calculator.
   - Set the 'Name' to View
   - Set Modifiers to public
   - Set Superclass to javax.swing.JFrame (Click Browse and then type the full class name.)
   - Check checkboxes 'public static void main(String [] args) and 'Generate comments'
   - Click on Finish.

4. Edit the import declaration to be

   import javax.swing.*;
   import java.awt.*;

5. Add a 0-parameter private constructor for View:

   - In the constructor for View call setSize(300,300), setVisible(true) and setDefaultClose-Operation(JFrame.EXIT_ON_CLOSE)
   - Your constructor should look like this

     private View() {
         setSize(300,300) ;
         setVisible(**true**) ;
         setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE) ;
     }

6. main is a bit complicated (and strange). It should look like this:

   public static void main(String[] args) {

2

```
        SwingUtilities.invokeLater( new Runnable() {
            public void run() {
                new View() ;
            }} ) ;
    }
```

7. Run the program (see Fig 3)

   - Save the file with control-s (on Macs: command-s)
   - Check that there are no errors. (Errors will be indicated by red Xs in the Package Explorer.)
   - In the Package Explorer, right click on the file `View.java`.
   - On the popup context menu, select 'Run As / Java Application'
   - You should see the frame and should be able to close it.

2 Adding a Model

1. Obtain the `Model.java` file and the `Op.java` file from the course's website and save them both to the `Calculator/src/calculator` directory with the workspace directory.

2. In the Package Explorer, right click on the Calculator project and select Refresh. The two new files should appear in the Package Explorer.

3. Open the Model class in the editor. Identify its public methods.

4. Open Eclipse's Outline view by selecting on the menu Window / Show View / Outline. The public methods are indicated in the Outline view by green circles.

5. To the View class add an initialized private field

   private final Model model = **new** Model();

3 Adding some components

In the constructor for View:

1. Set the layout manager for the View by adding

   setLayout(**new** FlowLayout());

   as the first line of the constructor.

2. Add (after the call to setLayout) code to add 10 buttons to the View, labeled 0 to 9. I added each button with code

   ```
   JButton digitButton = new JButton( Integer.toString(i) ) ;
   add( digitButton ) ;
   ```

3. Also **add** a button labeled "+", a button labelled "Clear", and a button labelled "=".

4. Declare a **private final** field of type **JLabel**. Call it **valueLabel**. Initialize this field by creating a new JLabel.

5. In the constructor **add** the value label to the frame.

6. Create a new method in **View**

   void refresh() {
       valueLabel.setText(model.getResult()) ; }

7. Add a call to **refresh** as the final command in **View**'s constructor.

8. Try running your application. See Fig. 4

4   Closing the loop

Create a class **DigitListener** that implements the interface java.awt.event.ActionListener. Each **DigitListener** should know a **View** and a **Model**. (I.e. it should have pointers to a **View** and to a **Model** as its fields.) The constructor of **DigitListener** should record a pointer to a **View**, a pointer to a **Model**, and an **int**. Since **DigitListener** implements **ActionListener**, it must have a subroutine with signature

@Override **public void** actionPerformed(ActionEvent e)

This subroutine should update the model by calling **digit** and then **refresh** the view. (Note that although the parameter is not used, it must still be declared.)

Back in **View**'s constructor you need to create instances of **DigitListener** and associate them with the appropriate listener like this.

JButton digitButton = **new** JButton(Integer.toString(i)) ;
digitButton.addActionListener(**new** DigitListener(**this**, model, i)) ;
add(digitButton) ;

Try your application now. Click on the digit buttons. You should see the effect in the operand label.

Notice how the Swing framework is calling your code even though the dependence goes the other way. This is an example of "inversion of dependence".

Create a class **OperationListener** similar to **DigitListener**, but that calls method **operation** rather than **digit** in the model. Note that the **operation** method of class **Model** takes a parameter of the enumeration class **Op**. The **Op** class defines a number of constants of type **Op**. To refer to these, you simply write **Op.ADD** or **Op.CLEAR** etc.

Associate an **OperationListener** with the "+" button, the Clear button, and the "="
Figure 5 shows your application at this point as a UML class diagram.
Try your application now.

5 More to try

Try adding one button for each operation. Try changing the style of the buttons. Try adding more operations to the model. See if you can add buttons to change the precision or the base. If you prefer RPN, create an RPN calculator. Make the calculator programmable.

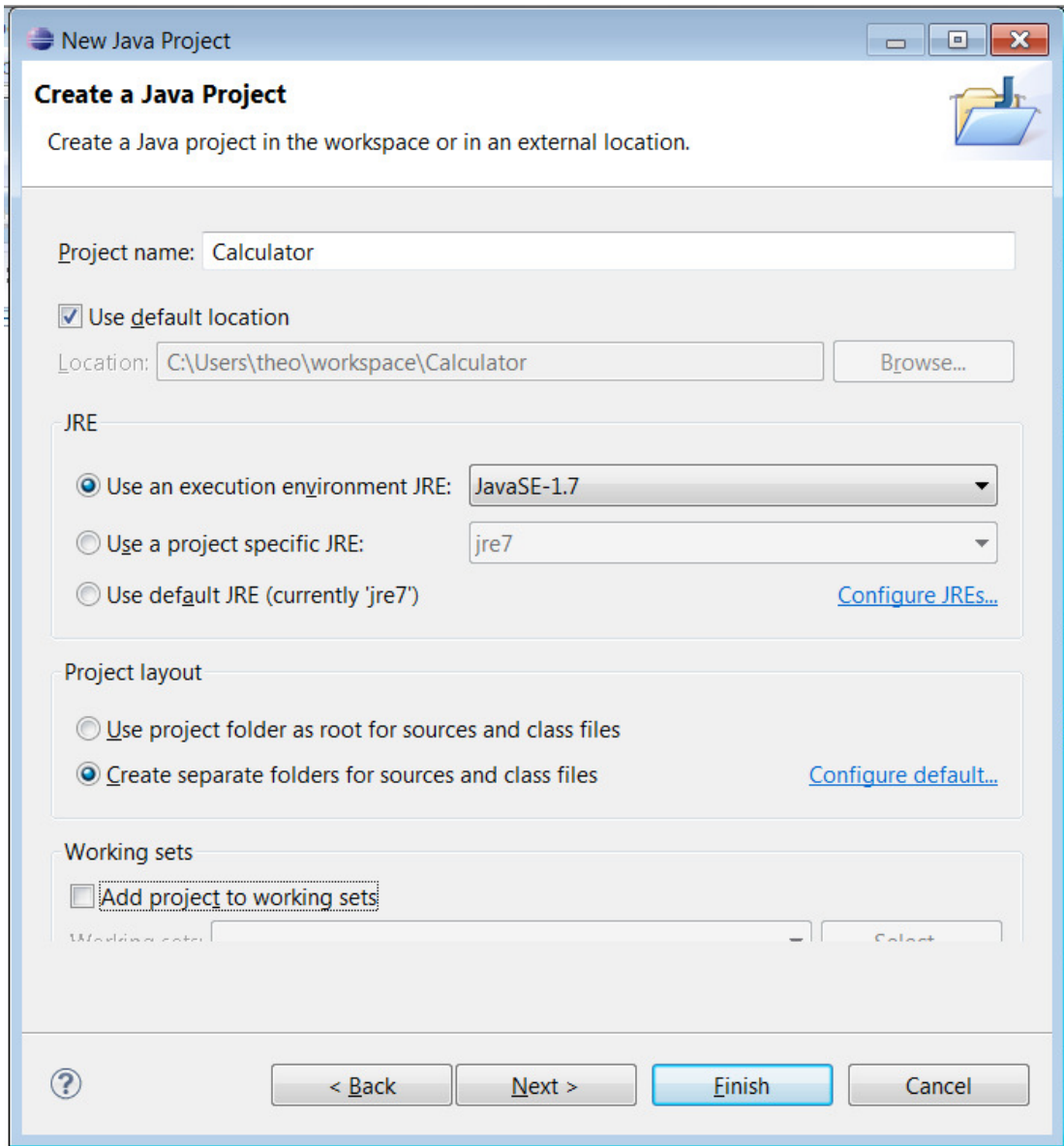Obviously there is much more to learn about layout of components within containers. Try using GridLayout or GridBagLayout and also using JPanels and borders.
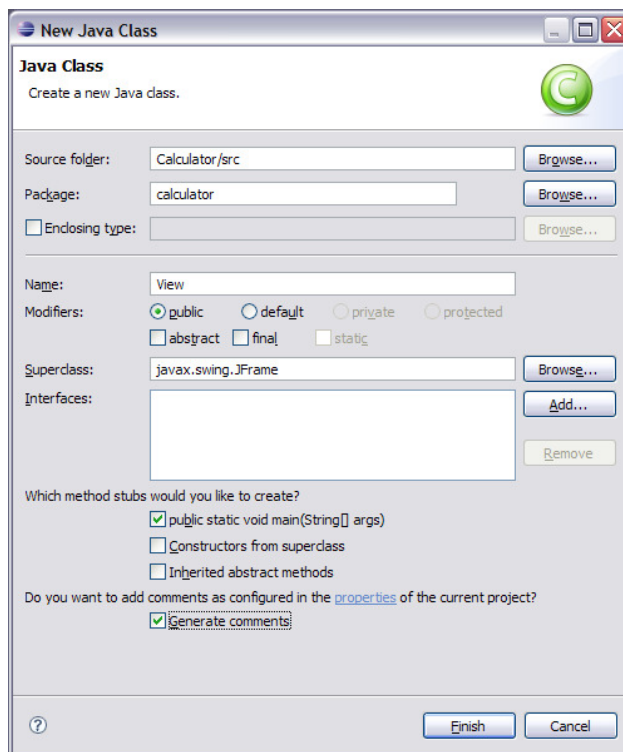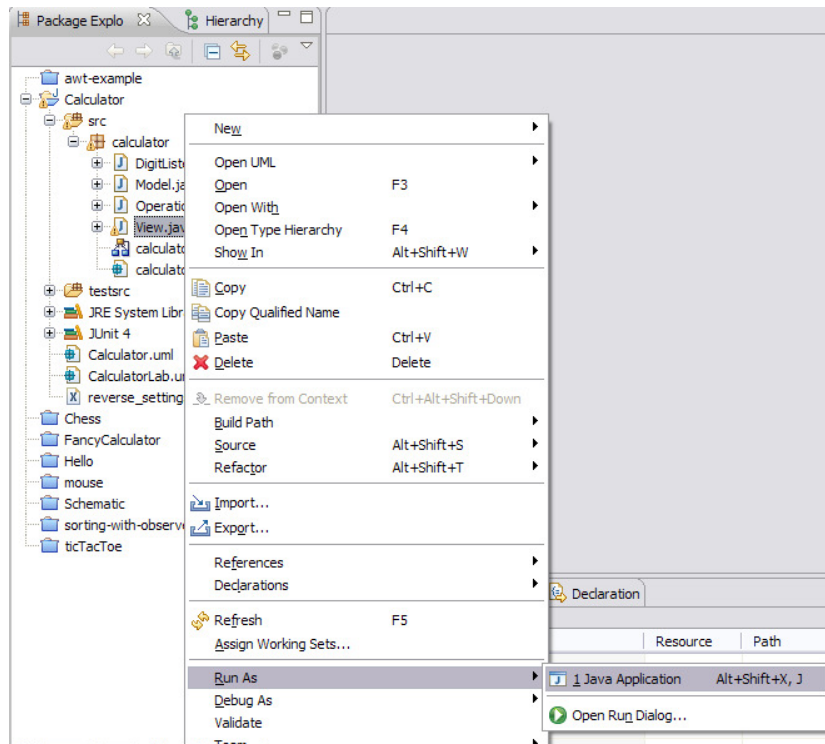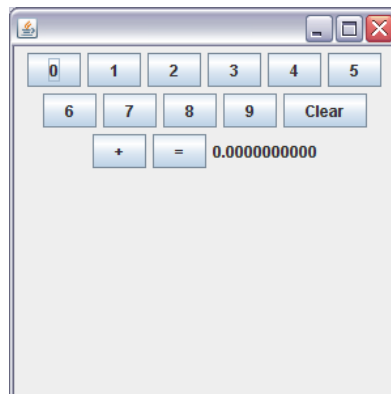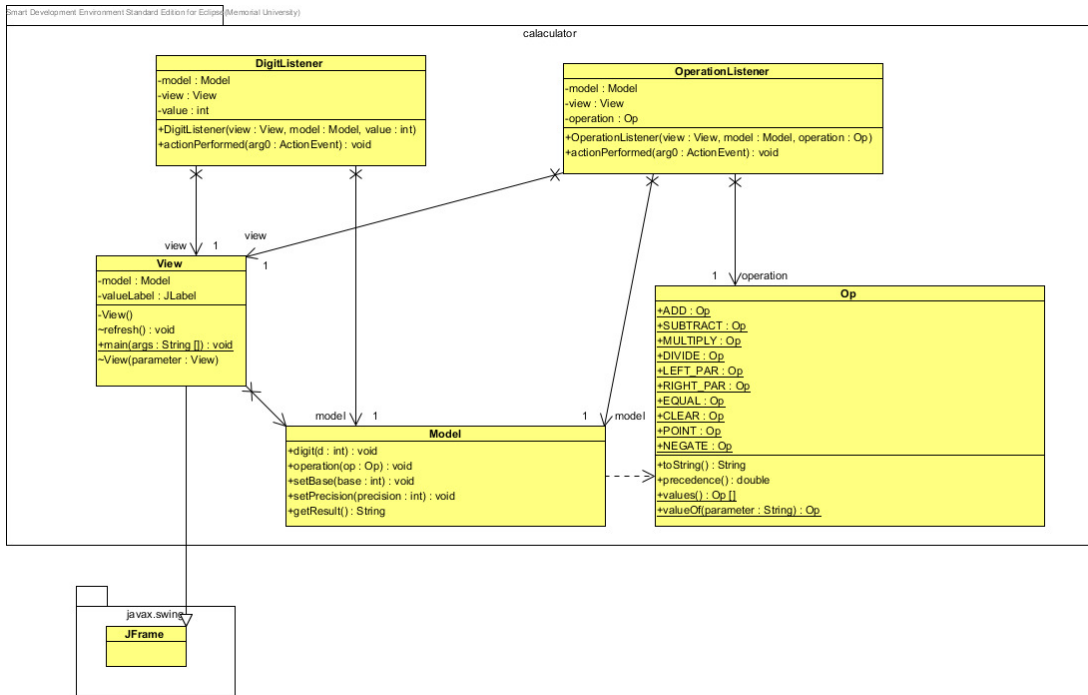
Figure 1:

Figure 2:

Figure 3:

Figure 4:

calaculator

DigitListener

-model : Model
-view : View
-value : int

+DigitListener(view : View, model : Model, value : int)
+actionPerformed(arg0 : ActionEvent) : void

OperationListener

-model : Model
-view : View
-operation : Op

+OperationListener(view : View, model : Model, operation : Op)
+actionPerformed(arg0 : ActionEvent) : void

view   view   1

View

-model : Model
-valueLabel : JLabel

-View()
~refresh() : void
+main(args : String []) : void
~View(parameter : View)

1   /operation

Op

+ADD : Op
+SUBTRACT : Op
+MULTIPLY : Op
+DIVIDE : Op
+LEFT_PAR : Op
+RIGHT_PAR : Op
+EQUAL : Op
+CLEAR : Op
+POINT : Op
+NEGATE : Op

+toString() : String
+precedence() : double
+values() : Op []
+valueOf(parameter : String) : Op

model   1   1   model

Model

+digit(d : int) : void
+operation(op : Op) : void
+setBase(base : int) : void
+setPrecision(precision : int) : void
+getResult() : String

javax.swing

JFrame

Figure 5:

9