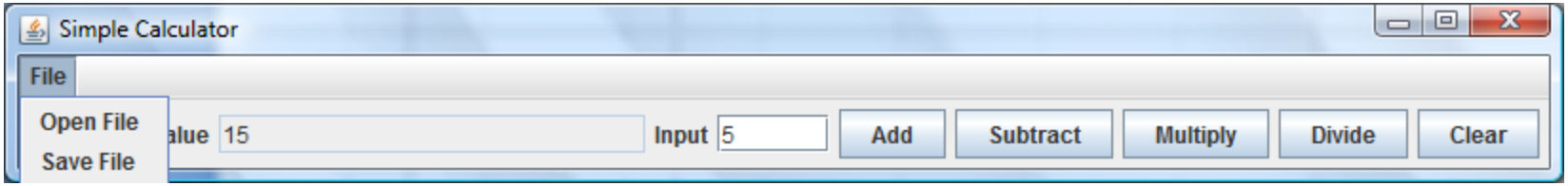
Graphical User Interfaces (Part 2)

View

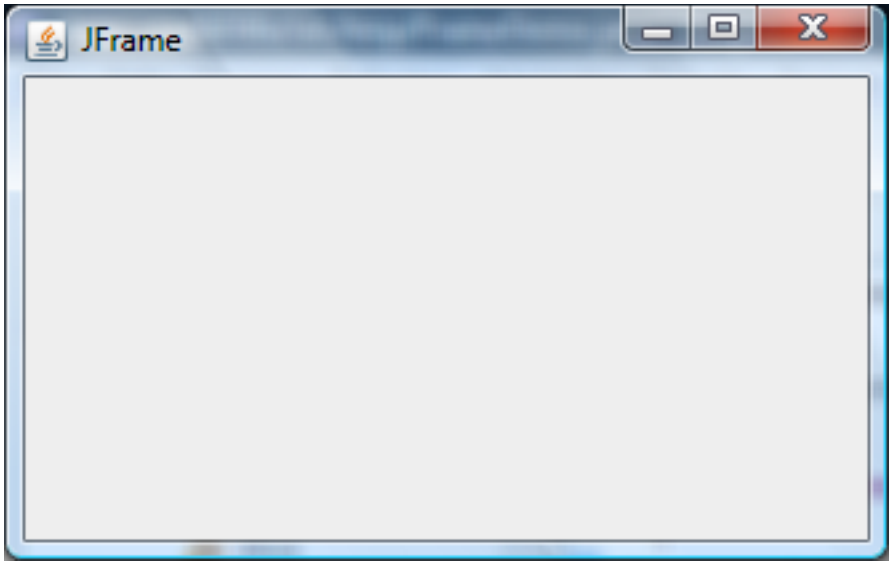
- view
  - presents the user with a sensory (visual, audio, haptic) representation of the model state
  - a user interface element (the user interface for simple applications)



Simple Applications

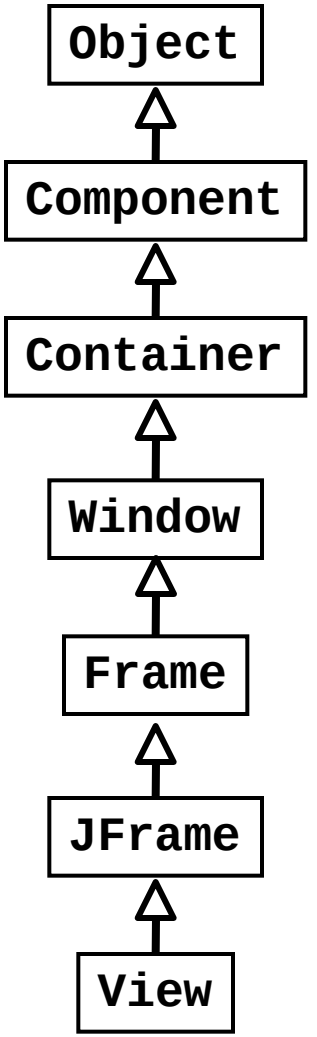
- simple applications often consist of just a single window (containing some controls)

JFrame
window with border, title, buttons



View as a Subclass of JFrame

- a View can be implemented as a subclass of a JFrame
  - hundreds of inherited methods but only a dozen or so are commonly called by the implementer (see URL below)



user interface item

holds other components

plain window

window with title and border

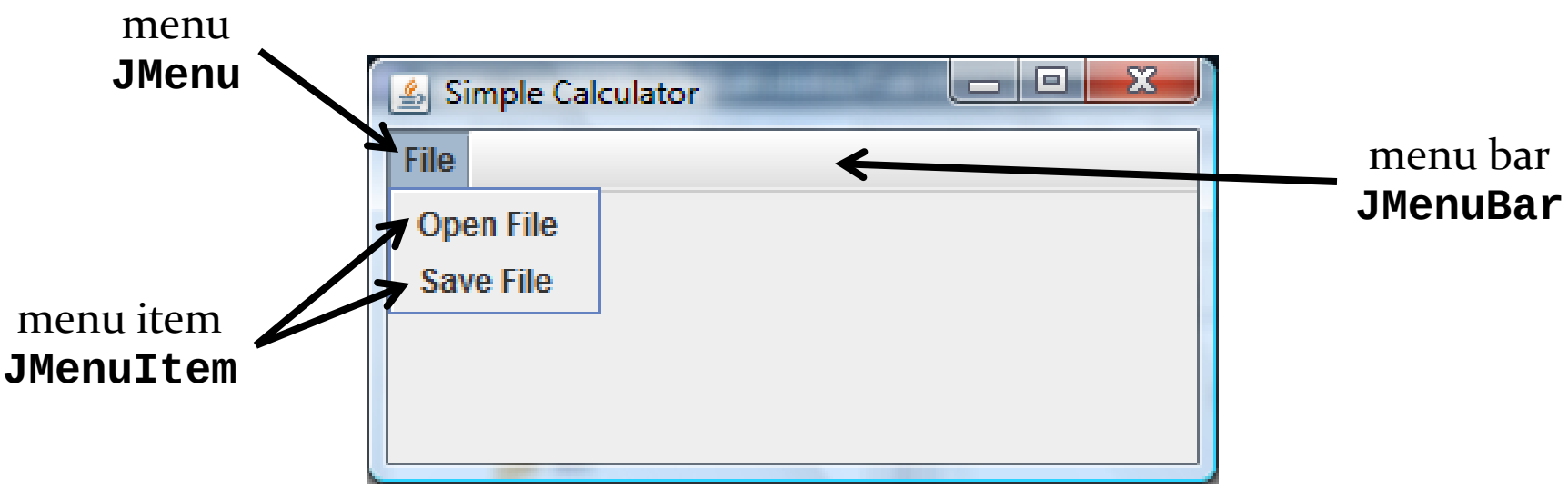
http://java.sun.com/docs/books/tutorial/uiswing/components/frame.html

Implementing a View

- the View is responsible for creating:
  - the Controller
  - all of the user interface (UI) components
    - menus — JMenuBar, JMenu, JMenuItem
    - buttons — JButton
    - labels — JLabel
    - text fields — JTextField
    - file dialog — JFileChooser
- the View is also responsible for setting up the communication of UI events to the Controller
  - each UI component needs to know what object it should send its events to

Menus

- a menu appears in a *menu bar* (or a popup menu)
- each item in the menu is a *menu item*



http://docs.oracle.com/javase/tutorial/uiswing/components/menu.html

Menus

- to create a menu
  - create a JMenuBar
  - create one or more JMenu objects
    - add the JMenu objects to the JMenuBar
  - create one or more JMenuItem objectes
    - add the JMenuItem objects to the JMenu

Menus

```
JMenuBar menuBar = new JMenuBar();

JMenu fileMenu = new JMenu("File");
menuBar.add(fileMenu);

JMenuItem printMenuItem = new JMenuItem("Print");
fileMenu.add(printMenuItem);
```

Adding the Menu

- see CalcView constructor
  - try changing the layout used by the view
  - http://docs.oracle.com/javase/tutorial/uiswing/layout/visual.html

Labels and Text Fields

- a label displays unselectable text and images
- a text field is a single line of editable text
  - the ability to edit the text can be turned on and off

http://docs.oracle.com/javase/tutorial/uiswing/components/label.html

http://docs.oracle.com/javase/tutorial/uiswing/components/textfield.html

Labels

- to create a label

```
JLabel label = new JLabel("text for the label");
```

- to create a text field (20 characters wide)

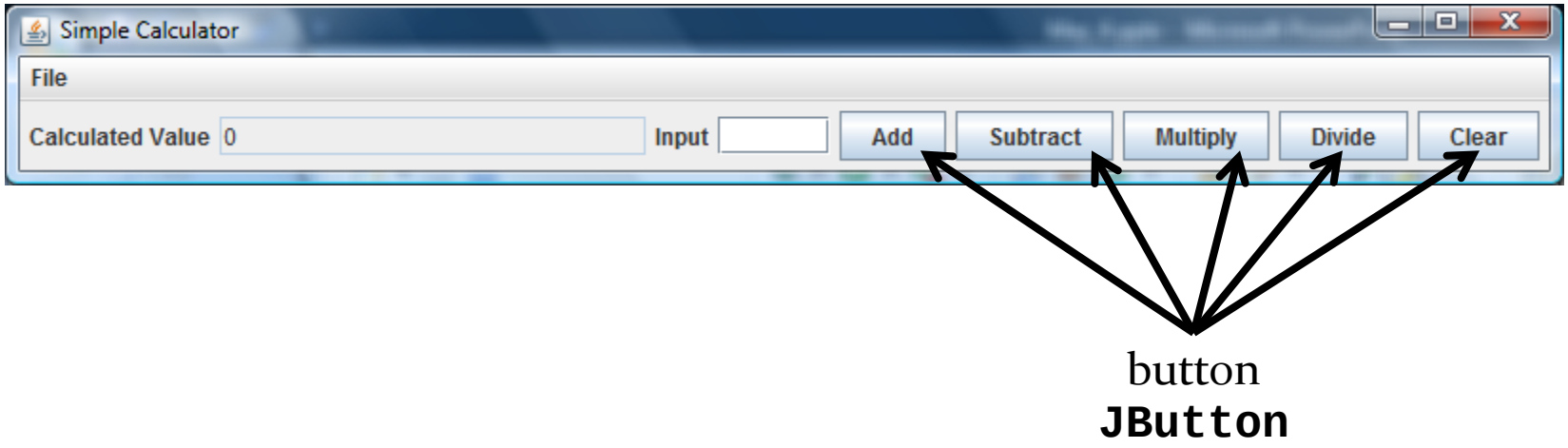
```
JTextField textField = new JTextField(20);
```

Adding the Labels and Text Fields

- see CalcView constructor
  - try making the text field editable and non-editable

Buttons

- a button responds to the user pointing and clicking the mouse on it (or the user pressing the Enter key when the button has the focus)



http://docs.oracle.com/javase/tutorial/uiswing/components/button.html

Buttons

- to create a button

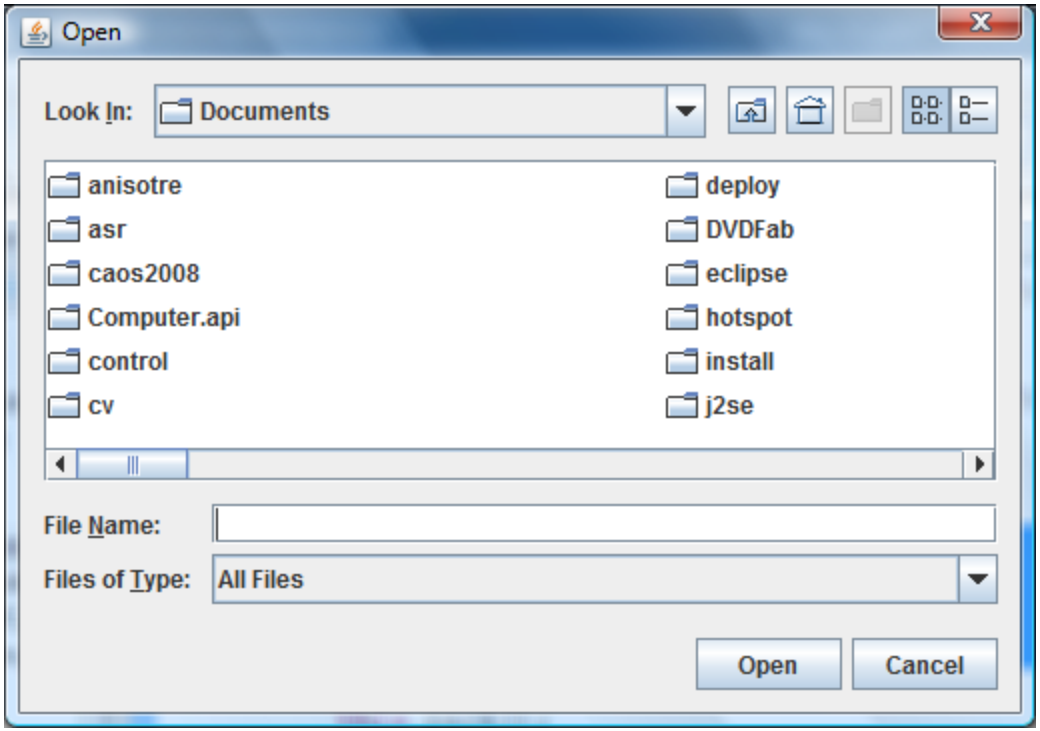
```
JButton button = new JButton("text for the button");
```

Adding the Buttons

- see CalcView constructor
  - try enabling and disabling the buttons

File Chooser

- a file chooser provides a GUI for selecting a file to open (read) or save (write)



file chooser (for choosing a file to open)
`JFileChooser`

http://docs.oracle.com/javase/tutorial/uiswing/components/filechooser.html

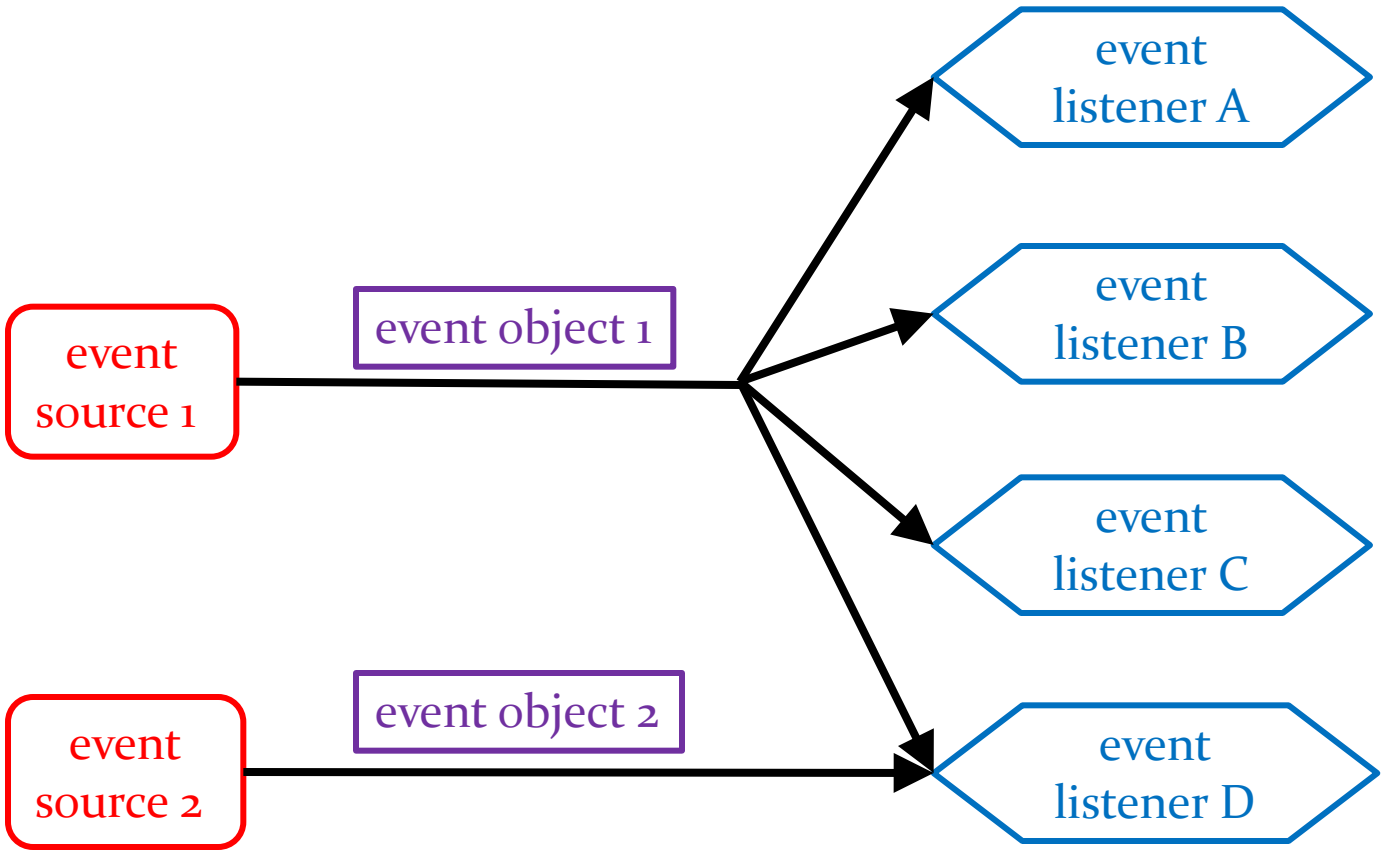
Using the File Chooser

- see CalcView getOpenFile and getSaveFile methods
  - http://docs.oracle.com/javase/tutorial/uiswing/components/filechooser.html

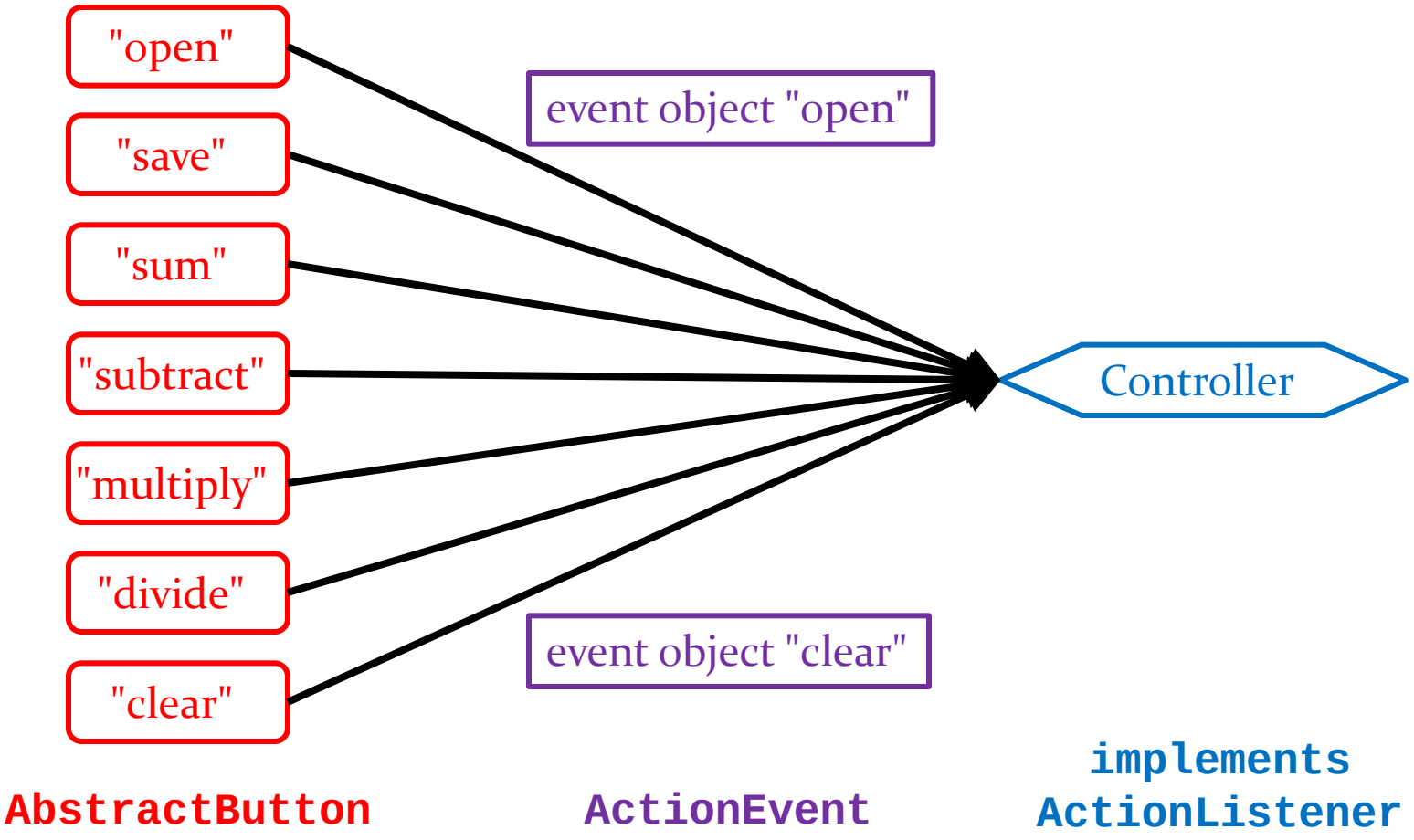
Event Driven Programming

- so far we have a View with some UI elements (buttons, text fields, menu items)
  - now we need to implement the actions
- each UI element is a source of events
  - button pressed, slider moved, text changed (text field), etc.
- when the user interacts with a UI element an event is triggered
  - this causes an event object to be sent to every object listening for that particular event
    - the event object carries information about the event
- the event listeners respond to the event

Not a UML Diagram



Not a UML Diagram



Implementation

- each **`JButton`** and **`JMenuItem`** has two inherited methods from **`AbstractButton`**

```
public void addActionListener(ActionListener l)

public void setActionCommand(String actionCommand)
```

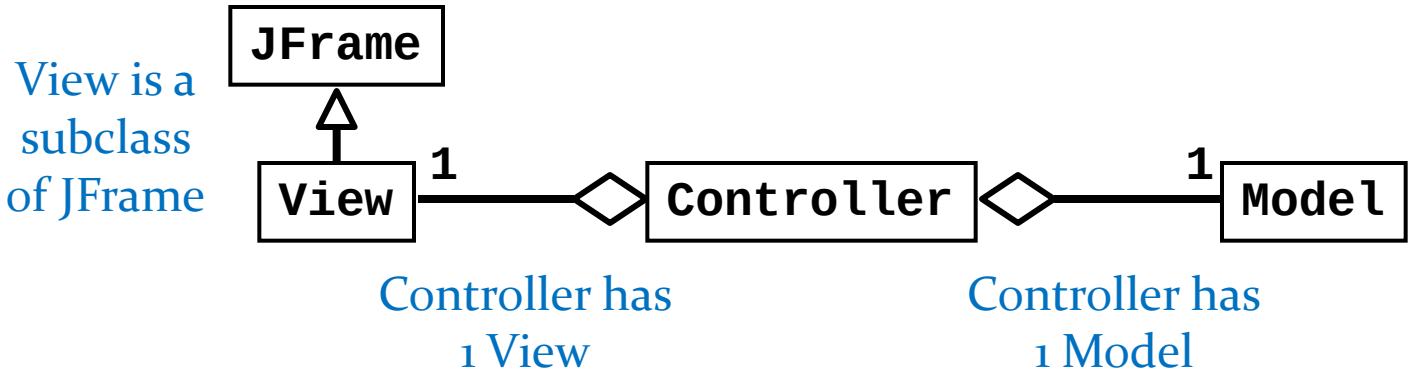
- for each **`JButton`** and **`JMenuItem`**
  1. call **`addActionListener`** with the controller as the argument
  2. call **`setActionCommand`** with a string describing what event has occurred

CalcView: Add Actions

- see CalcView setCommand method

Controller

- controller
  - processes and responds to events (such as user actions) from the view and translates them to model method calls
- needs to interact with both the view and the model but does not own the view or model
  - aggregation



Controller Fields

- see CalcController

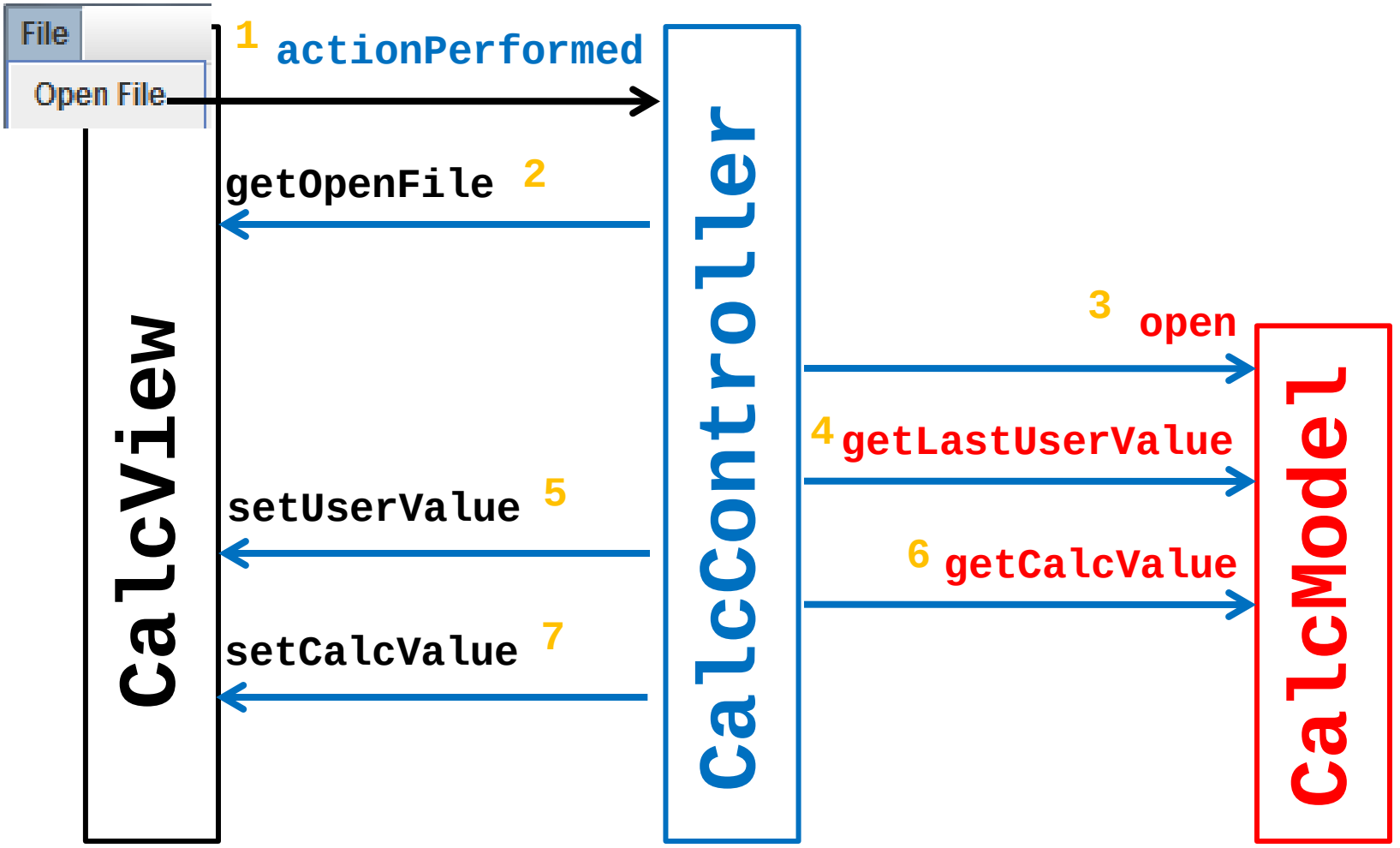
CalcController

- recall that our application only uses events that are fired by buttons (**`JButton`**s and **`JMenuItem`**s)
  - a button fires an **`ActionEvent`** event whenever it is clicked
- **`CalcController`** listens for fired **`ActionEvent`**s
  - how? by implementing the **`ActionListener`** interface

```
public interface ActionListener
{
  void actionPerformed(ActionEvent e);
}
```

- **`CalcController`** was registered to listen for **`ActionEvent`**s fired by the various buttons in **`CalcView`** (see method **`setCommand`** in **`CalcView`**)
- whenever a button fires an event, it passes an **`ActionEvent`** object to **`CalcController`** via the **`actionPerformed`** method
  - **`actionPerformed`** is responsible for dealing with the different actions (open, save, sum, etc)

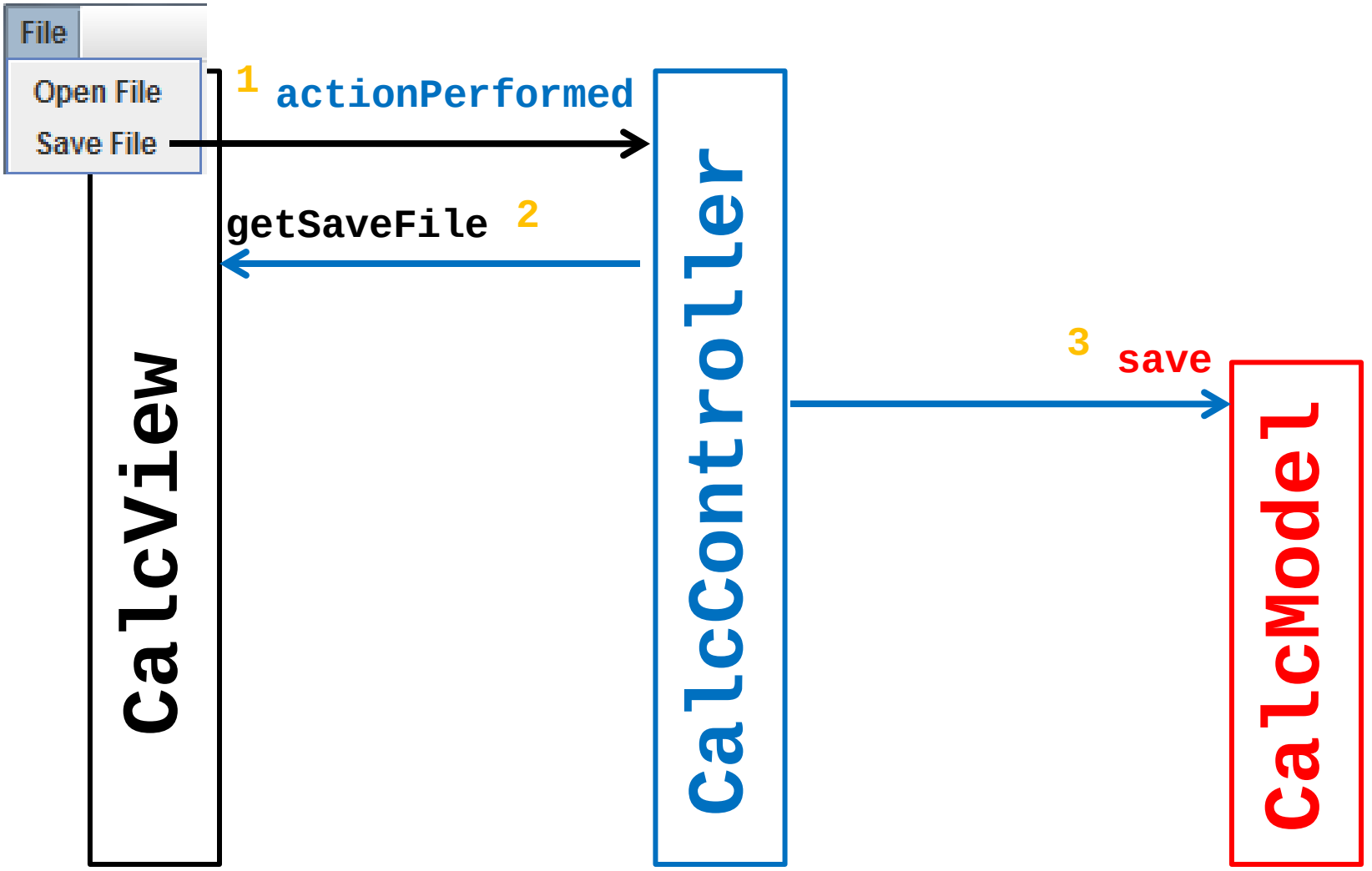
Opening a File



CalcController: Open a File

- see CalcController actionPerformed method

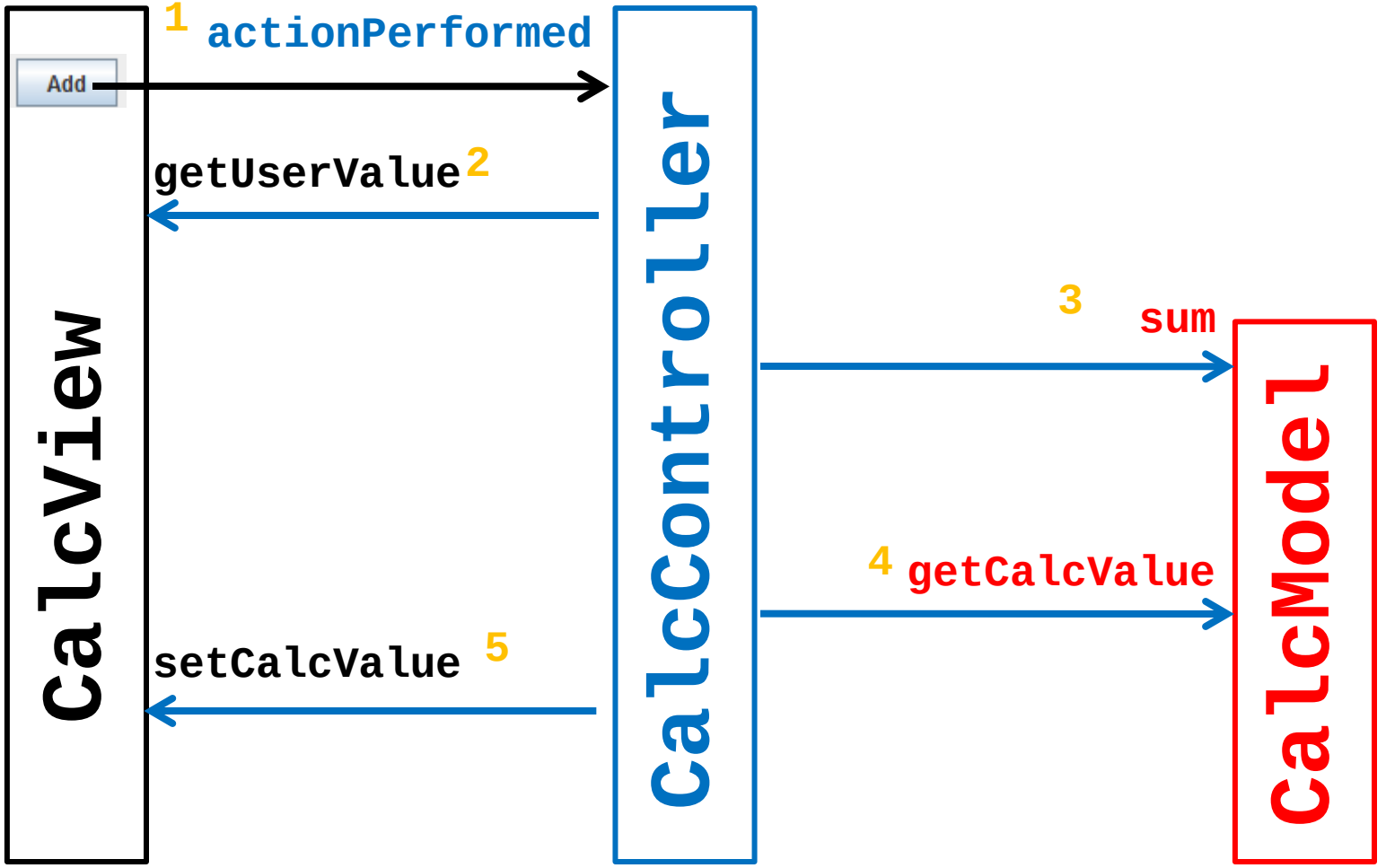
Saving a File



CalcController: Save a File

- see CalcController actionPerformed method

Sum, Subtract, Multiply, Divide



CalcController: Other Actions

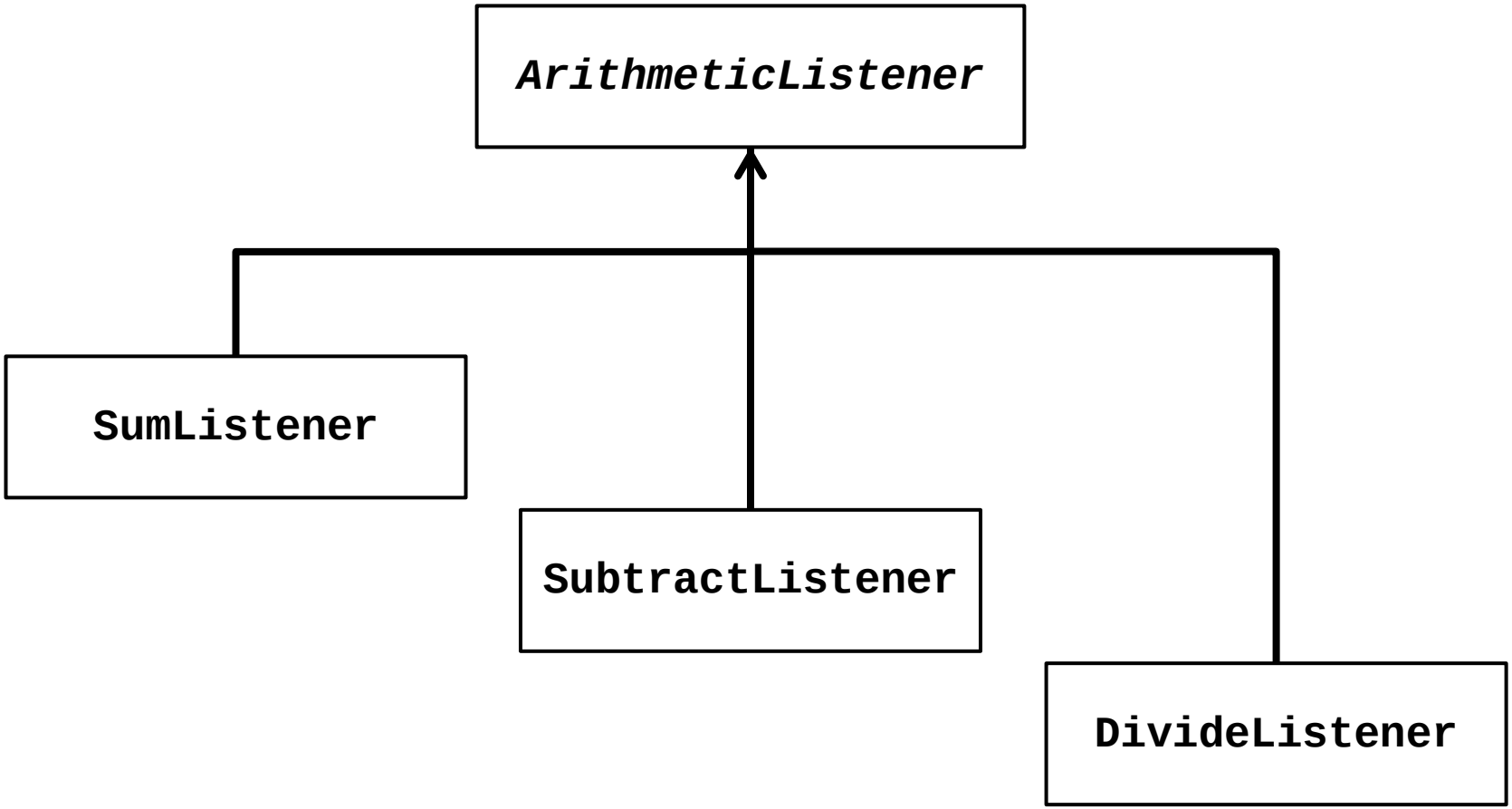
- see CalcController actionPerformed method

actionPerformed

- even with only 5 buttons and 2 menu items our **`actionPerformed`** method is unwieldy
  - imagine what would happen if you tried to implement a Controller this way for a big application

- rather than one big actionPerformed method we can register a different **`ActionListener`** for each button
  - each **`ActionListener`** will be an object that has its own version of the **`actionPerformed`** method

Calculator Listeners



Calculator Listener

- whenever a listener receives an event corresponding to an arithmetic operation it does:
  1. asks CalcView for the user value and converts it to a BigInteger
     - **`getUserValue`** method
  2. asks CalcModel to perform the arithmetic operation
     - **`doOperation`** method
  3. updates the calculated value in CalcView

ArithmeticListener

```
private abstract class ArithmeticListener implements
   ActionListener {

   @Override
   public void actionPerformed(ActionEvent action) {
1.   BigInteger userValue = this.getUserValue();
     if (userValue != null) {
2.     this.doOperation(userValue);
3.     this.setCalculatedValue();
     }
   }
```

ArithmeticListener

```
/**
 * Subclasses will override this method to add, subtract,
 * divide, multiply, etc., the userValue with the current
 * calculated value.
 */
protected abstract void doOperation(BigInteger userValue);
```

ArithmeticListener

```
private BigInteger getUserValue() {
  BigInteger userValue = null;
  try {
    userValue = new BigInteger(getView().getUserValue());
  }
  catch(NumberFormatException ex)
  {}
  return userValue;
}

private void setCalculatedValue() {
    getView().setCalcValue(getModel().getCalcValue().
                           toString());
}
```

Note: these methods need access to the view and model which are associated with the controller.

Inner Classes

- how do we give the listeners access to the view and model?
  - could use aggregation
  - alternatively, we can make the listeners be inner classes of the controller

Inner Classes

- an inner class is a (non-static) class that is defined inside of another class

```
public class Outer
{
  // Outer's attributes and methods

  private class Inner
  { // Inner's attributes and methods
  }
}
```

Inner Classes

- an inner class has access to the attributes and methods of its enclosing class, even the private ones

```
public class Outer
{
  private int outerInt;

  private class Inner
  {
    public setOuterInt(int num) { outerInt = num; }
  }
}
```

note not **`this.outerInt`**

use **`Outer.this.outerInt`**

ArithmeticListener

```
public class CalcController2 {
  // ...

  // inner class of CalcController2
  private abstract class ArithmeticListener implements
                                              ActionListener {
    // ...
  }

  // inner class of CalcController2
  private class SumListener extends ArithmeticListener {
    @Override
    protected void doOperation(BigInteger userValue) {
      // ...
    }
  }
}
```

SumListener

```
private class SumListener extends ArithmeticListener {
  @Override
  protected void doOperation(BigInteger userValue) {
    if (userValue != null) {
      getModel().sum(userValue);
    }
  }
}
```

Why Use Inner Classes

- only the controller needs to create instances of the various listeners
  - i.e., the listeners are not useful outside of the controller
  - making the listeners private inner classes ensures that only **`CalcController`** can instantiate the listeners

- the listeners need access to private methods inside of **`CalcController`** (namely **`getView`** and **`getModel`**)
  - inner classes can access private methods

Calculator using multiple listeners

- requires changes to the view to support the adding of listeners
- see CalcView2