

Graphical User Interfaces

Based on slides by Prof. Burton Ma

Model—View—Controller

TV

```
- on : boolean  
- channel : int  
- volume : int  
  
+ power(boolean) : void  
+ channel(int) : void  
+ volume(int) : void
```

Model



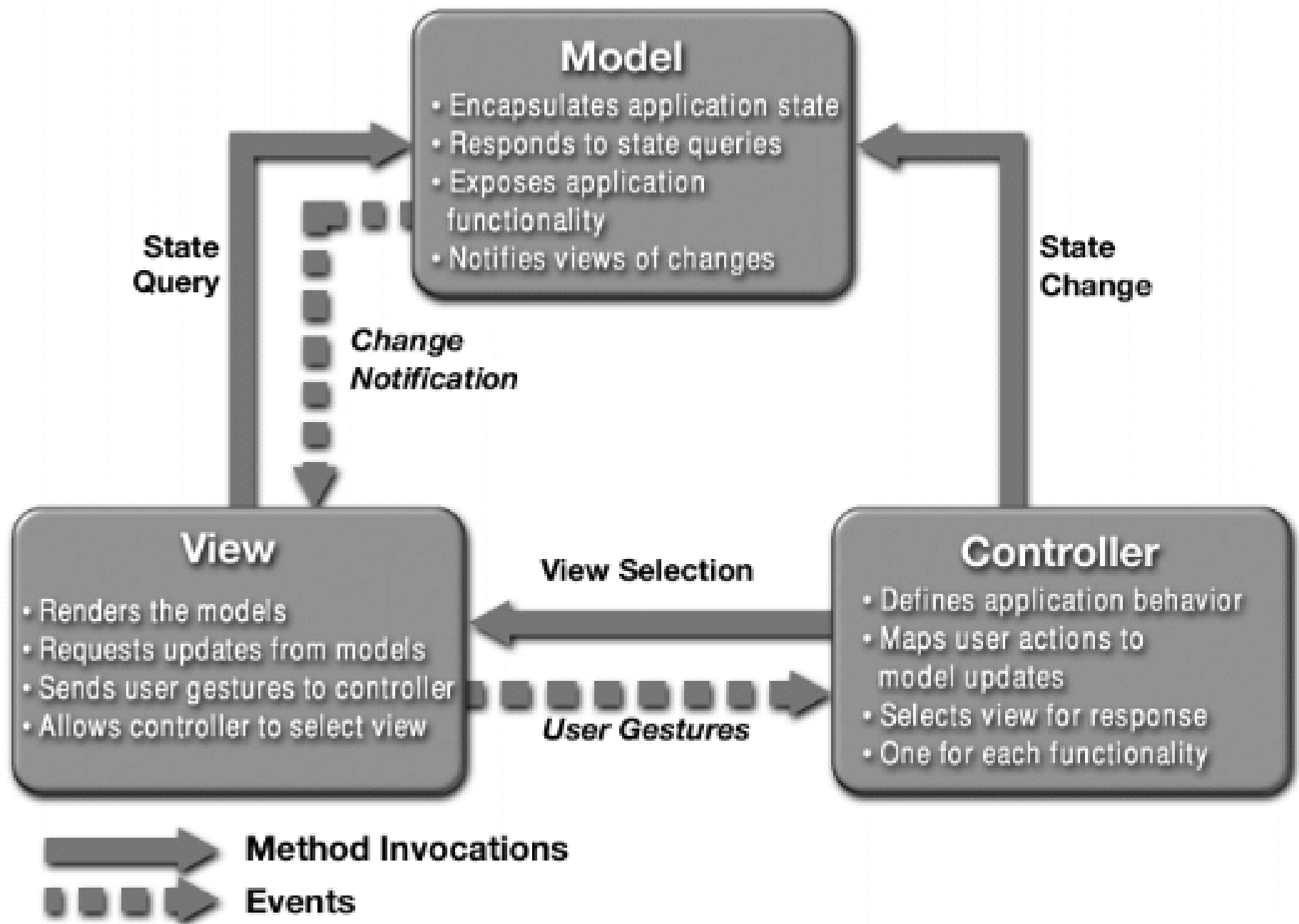
View

Controller



RemoteControl

```
+ togglePower() : void  
+ channelUp() : void  
+ volumeUp() : void
```



▶ Model

- Represents state of the application and the rules that govern access to and updates of state

▶ View

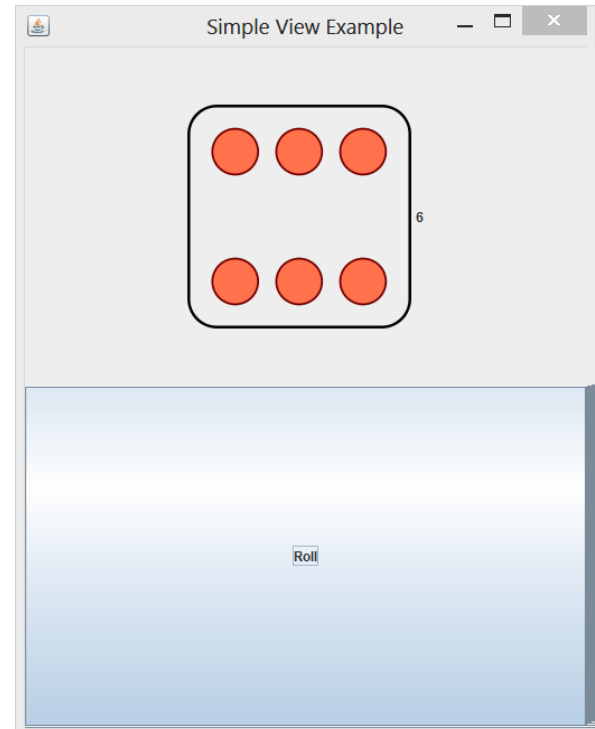
- Presents the user with a sensory (visual, audio, haptic) representation of the model state
- A user interface element (the user interface for simple applications)

▶ Controller

- Processes and responds to events (such as user actions) from the view and translates them to model method calls

App to Roll a Die

- ▶ A simple application that lets the user roll a die
 - When the user clicks the “Roll” button the die is rolled to a new random value
 - “Event driven programming”



App to Roll a Die

- ▶ This application is simple enough to write as a single class
 - SimpleRoll.java

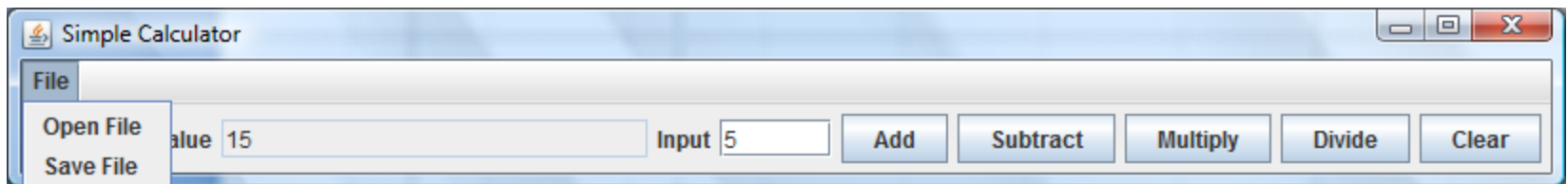
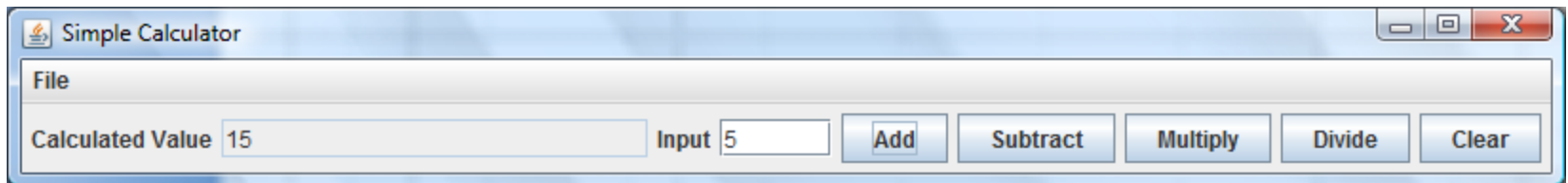
App to Roll a Die

- ▶ We can also write the application using the model–view–controller pattern
 - SimpleModel.java
 - SimpleView.java
 - SimpleController.java
 - SimpleApp.java

Simple Calculator

- ▶ Implement a simple calculator using the model–view–controller (MVC) design pattern
- ▶ Features:
 - Sum, subtract, multiply, divide
 - Clear
 - Records a log of the user actions
 - Save the log to file
 - Read the log from a file

Application Appearance



Creating the Application

- ▶ The calculator application is launched by the user
 - The text refers to the application as the GUI
- ▶ The application:
 1. Creates the model for the calculator, and then
 2. Creates the view of the calculator

CalcMVC Application

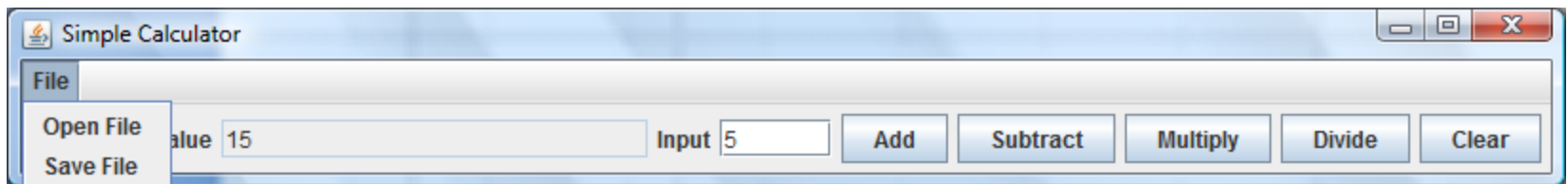
```
public class CalcMVC
{
    public static void main(String[] args)
    {

        CalcModel model = new CalcModel();
        CalcView view = new CalcView(model);

        view.setVisible(true);
    }
}
```

Model

- ▶ Features:
 - Sum, subtract, multiply, divide
 - Clear
 - Records a log of the user actions
 - Save the log to file
 - Read the log from a file



BigInteger: Immutable arbitrary-precision integers

CalcModel

```
- calcValue : BigInteger
- log : ArrayList<String>

+ getCalcValue() : BigInteger
+ getLastUserValue() : BigInteger
+ sum(BigInteger) : void
+ subtract(BigInteger) : void
+ multiply(BigInteger) : void
+ divide(BigInteger) : void
+ clear() : void
+ save(File) : void
+ open(File) : void
- updateLog(String operation, String userValue) : void
```

CalcModel: Attributes and Ctor

```
public class CalcModel
{
    private BigInteger calcValue;
    private ArrayList<String> log;

    // creates the log and initializes the attributes
    // using the clear method
    CalcModel()
    {
        this.log = new ArrayList<String>();
        this.clear();
    }
}
```

CalcModel: clear

```
// sets the calculated value to zero, clears the log,  
// and adds zero to the log  
public void clear()  
{  
    this.calcValue = BigInteger.ZERO;  
    this.log.clear();  
    this.log.add(this.calcValue.toString());  
}
```

CalcModel: getLastUserValue

```
// empty log looks like
// [0]
// non-empty log looks like:
// [0, +, 5, =, 5, -, 3, =, 2, *, 7, =, 14]
public BigInteger getLastUserValue()
{
    if(this.log.size() == 1)
    {
        return BigInteger.ZERO;
    }
    final int last = this.log.size() - 1;
    return new BigInteger(this.log.get(last - 2));
}
```


CalcModel: getCalcValue

```
// BigInteger is immutable; no privacy leak  
public BigInteger getCalcValue()  
{  
    return this.calcValue;  
}
```

CalcModel: sum

```
// sums the user value with the current calculated value
// and updates the log using updateLog
public void sum(BigInteger userValue)
{
    this.calcValue = this.calcValue.add(userValue);
    this.updateLog("+", userValue.toString());
}
```

CalcModel: subtract and multiply

```
public void subtract(BigInteger userValue)
{
    this.calcValue = this.calcValue.subtract(userValue);
    this.updateLog("-", userValue.toString());
}
```

```
public void multiply(BigInteger userValue)
{
    this.calcValue = this.calcValue.multiply(userValue);
    this.updateLog("*", userValue.toString());
}
```

CalcModel: divide

```
// cannot divide by zero; options:  
// 1. precondition  userValue != 0  
// 2. validate userValue; do nothing  
// 3. validate userValue; return false  
// 4. validate userValue; throw exception  
public void divide(BigInteger userValue)  
{  
    this.calcValue = this.calcValue.divide(userValue);  
    this.updateLog("/", userValue.toString());  
}
```

CalcModel: save

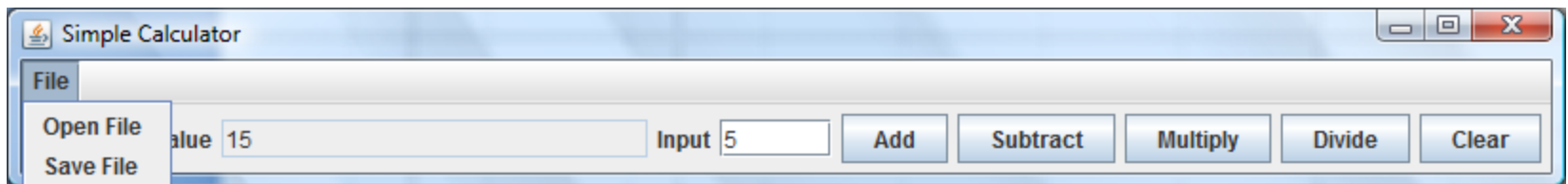
```
// relies on fact ArrayList implements Serializable
public void save(File file)
{
    FileOutputStream f = null;
    ObjectOutputStream out = null;
    try {
        f = new FileOutputStream(file); // can throw
        out = new ObjectOutputStream(f); // can throw
        out.writeObject(this.log);    // can throw
        out.close();
    }
    catch(IOException ex)
    {}
}
```

CalcModel: open

```
public void open(File file) {  
    FileInputStream f = null;  
    ObjectInputStream in = null;  
    ArrayList<String> log = null; // object to read from file  
    try {  
        f = new FileInputStream(file); // can throw  
        in = new ObjectInputStream(f); // can throw  
        log = (ArrayList<String>) in.readObject(); // can throw  
        in.close();  
        this.log = log;  
        final int last = this.log.size() - 1;  
        this.calcValue = new BigInteger(this.log.get(last));  
    }  
    catch(IOException ex) {}  
    catch(ClassNotFoundException ex) {}  
}
```

View

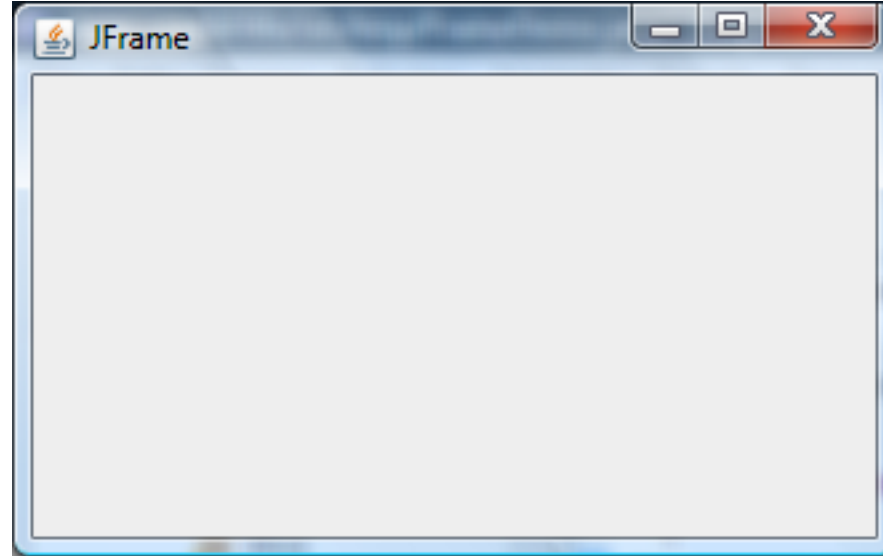
- ▶ View
 - Presents the user with a sensory (visual, audio, haptic) representation of the model state
 - A user interface element (the user interface for simple applications)



Simple Applications

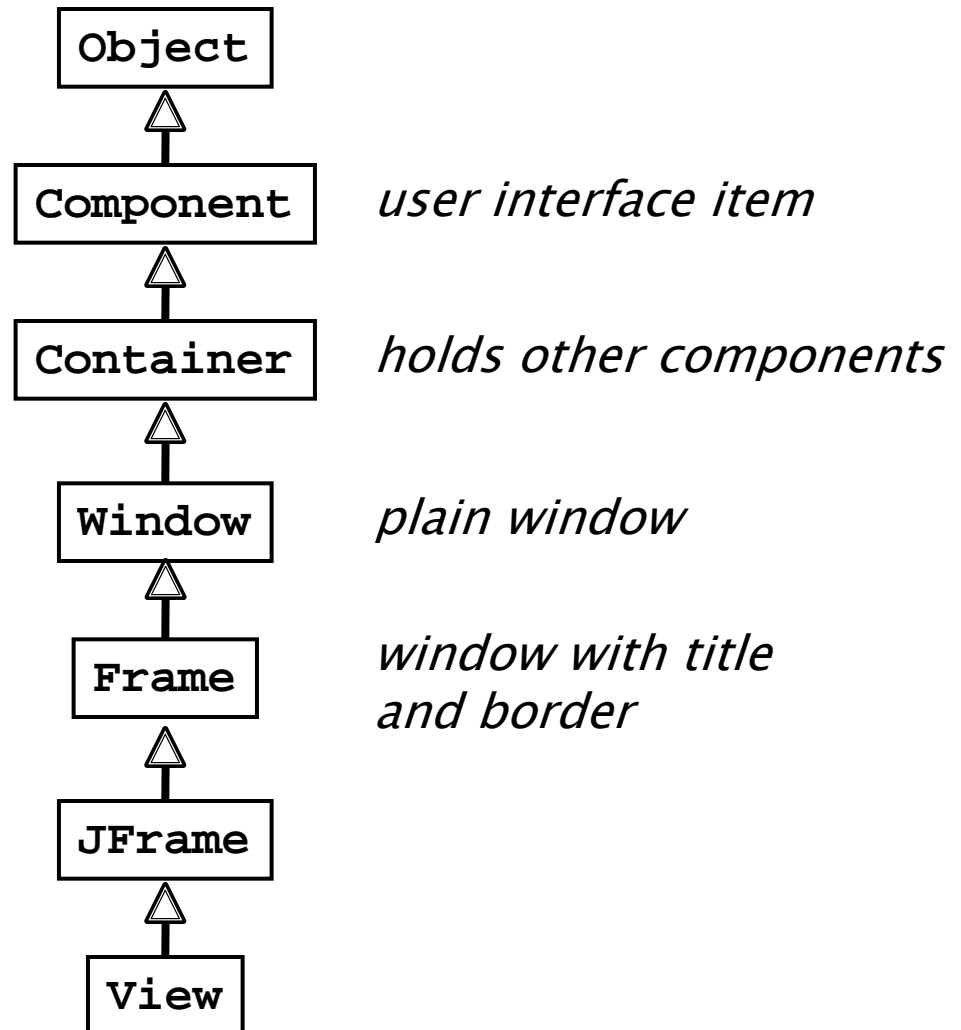
- ▶ Simple applications often consist of just a single window (containing some controls)

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)



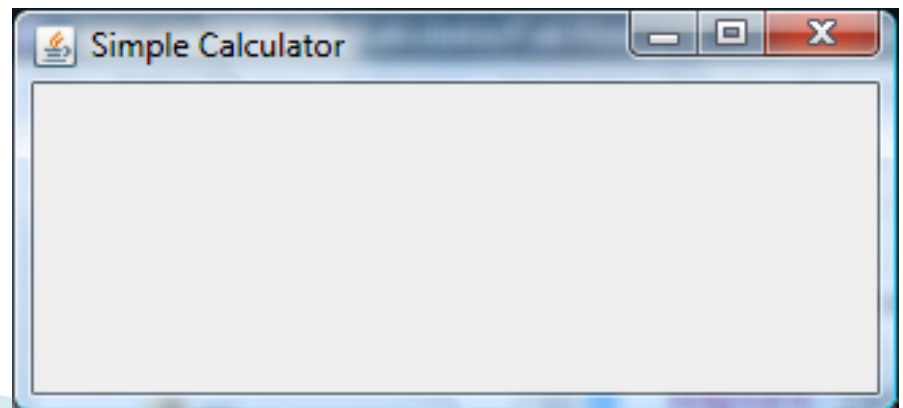
Implementing a View

- ▶ 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

CalcView: Create Controller

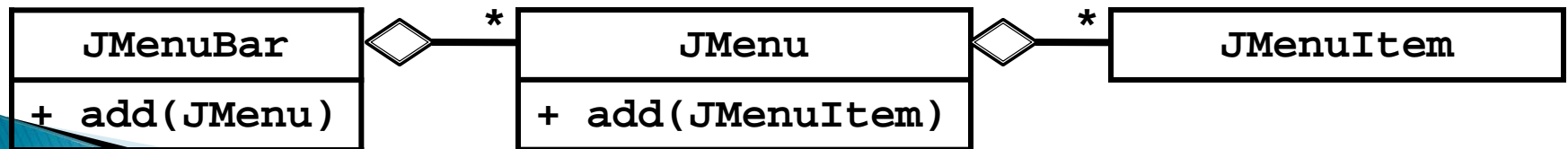
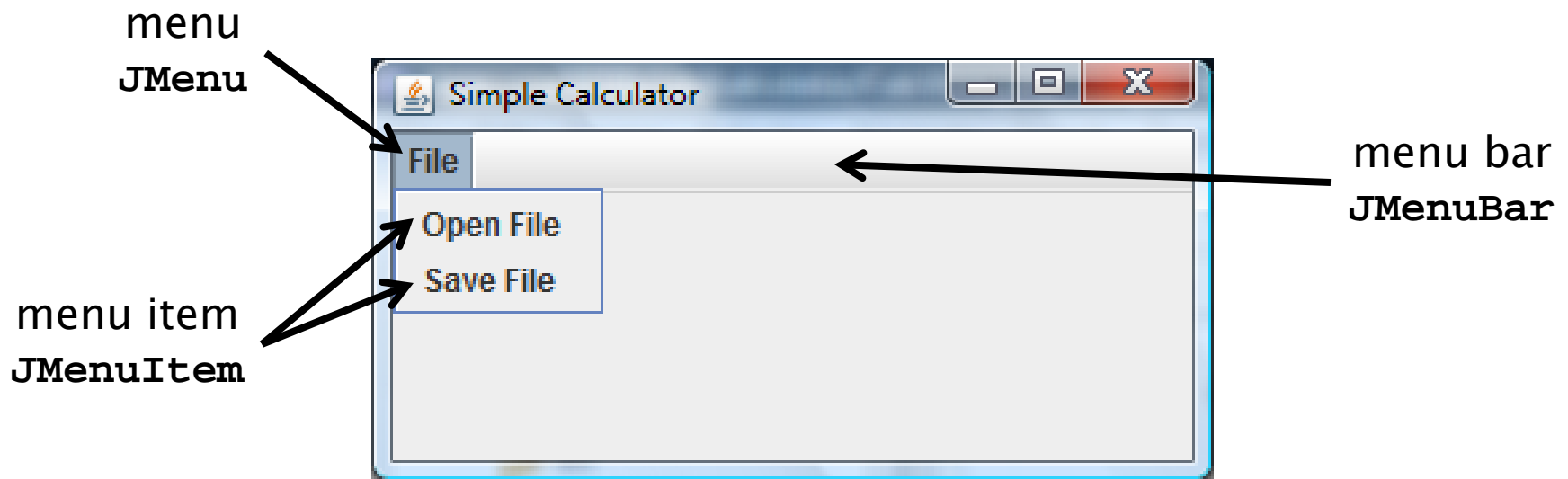
```
public class CalcView extends JFrame
{
    public CalcView(CalcModel model)
    {
        super("Simple Calculator");
        model.clear();
        CalcController controller =
            new CalcController(model, this);

        this.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
    }
}
```



Menus

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



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 objects
 - 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);
```

CalcView: Menubar, Menu, Menu Items

```
import javax.swing.JFrame;
import javax.swing.JMenu;
import javax.swing.JMenuBar;
import javax.swing.JMenuItem;

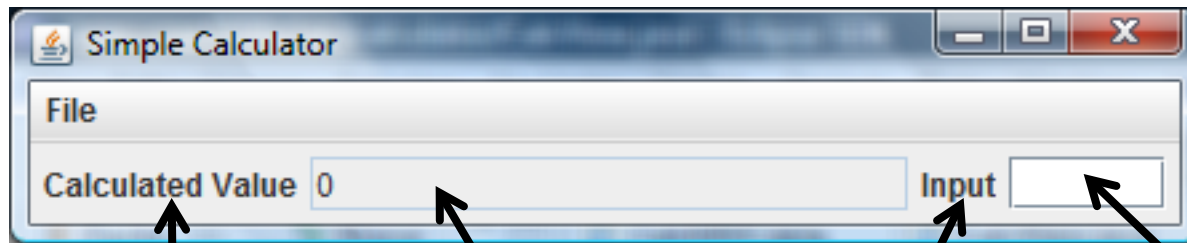
public class CalcView extends JFrame
{
    private JMenuBar menuBar;
    private JMenu fileMenu;

    public CalcView(CalcModel model)
    {
        super("Simple Calculator");
        model.clear();
        CalcController controller =
            new CalcController(model, this);

        this.menuBar = new JMenuBar();
```

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



label
JLabel

text field (edit off)
JTextField

label
JLabel

text field (edit on)
JTextField

<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);
```

CalcView: Labels and Text Fields

```
import java.awt.FlowLayout;

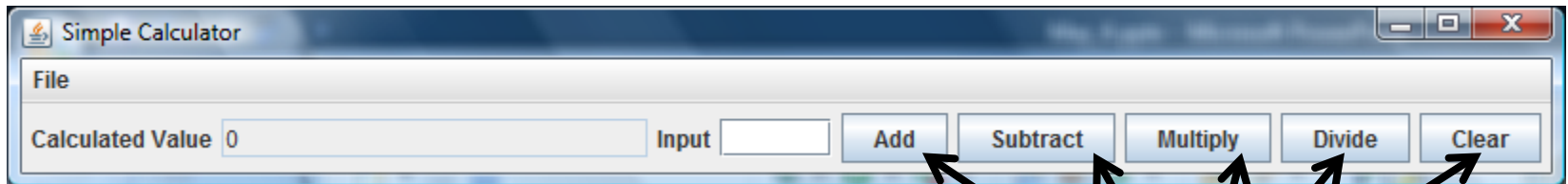
import javax.swing.JFrame;
import javax.swing.JLabel;
import javax.swing.JMenu;
import javax.swing.JMenuBar;
import javax.swing.JMenuItem;
import javax.swing.JTextField;

public class CalcView extends JFrame
{
    private JMenuBar menuBar;
    private JMenu fileMenu;
    private JTextField calcText;
    private JTextField userValueText;

    public CalcView(CalcModel model)
    {
```

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)



button
JButton

Buttons

- ▶ To create a button

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

CalcView: Buttons

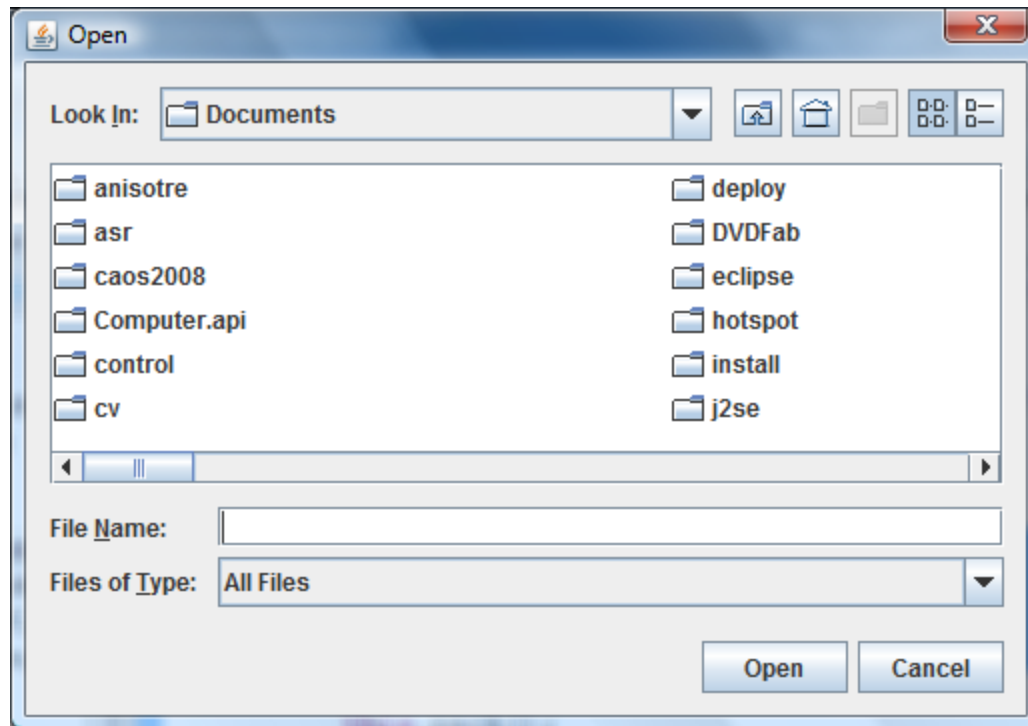
```
import java.awt.FlowLayout;

import javax.swing.JButton;
import javax.swing.JFrame;
import javax.swing.JLabel;
import javax.swing.JMenu;
import javax.swing.JMenuBar;
import javax.swing.JMenuItem;
import javax.swing.JTextField;

public class CalcView extends JFrame
{
    private JMenuBar menuBar;
    private JMenu fileMenu;
    private JTextField calcText;
    private JTextField userValueText;
    private JButton sumButton;
    private JButton subtractButton;
```

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

CalcView: File Chooser

```
import java.awt.FlowLayout;
import java.io.File;

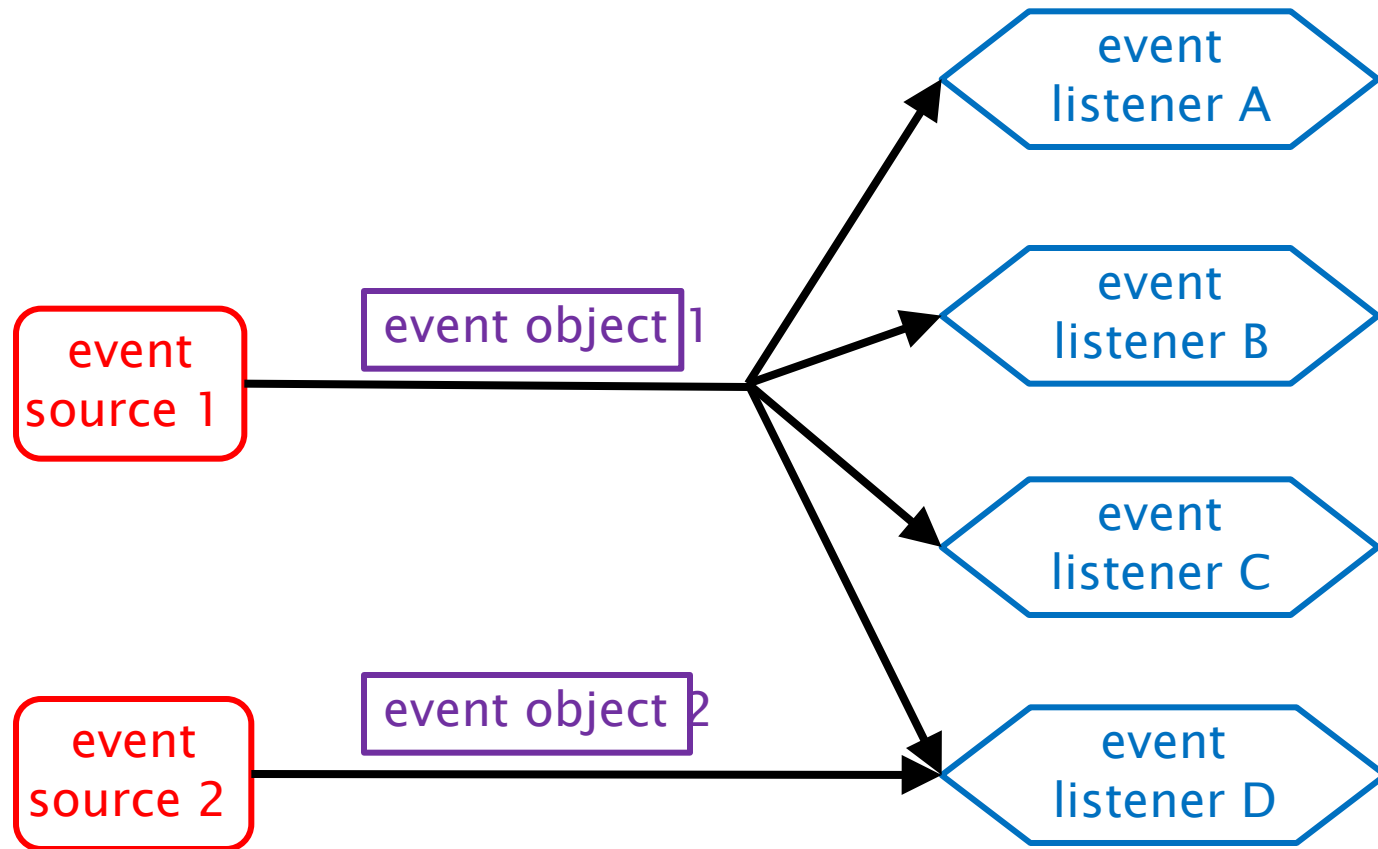
import javax.swing.JButton;
import javax.swing.JFileChooser;
import javax.swing.JFrame;
import javax.swing.JLabel;
import javax.swing.JMenu;
import javax.swing.JMenuBar;
import javax.swing.JMenuItem;
import javax.swing.JTextField;

public class CalcView extends JFrame
{
    private JMenuBar menuBar;
    private JMenu fileMenu;
    private JTextField calcText;
    private JTextField userValueText;
```

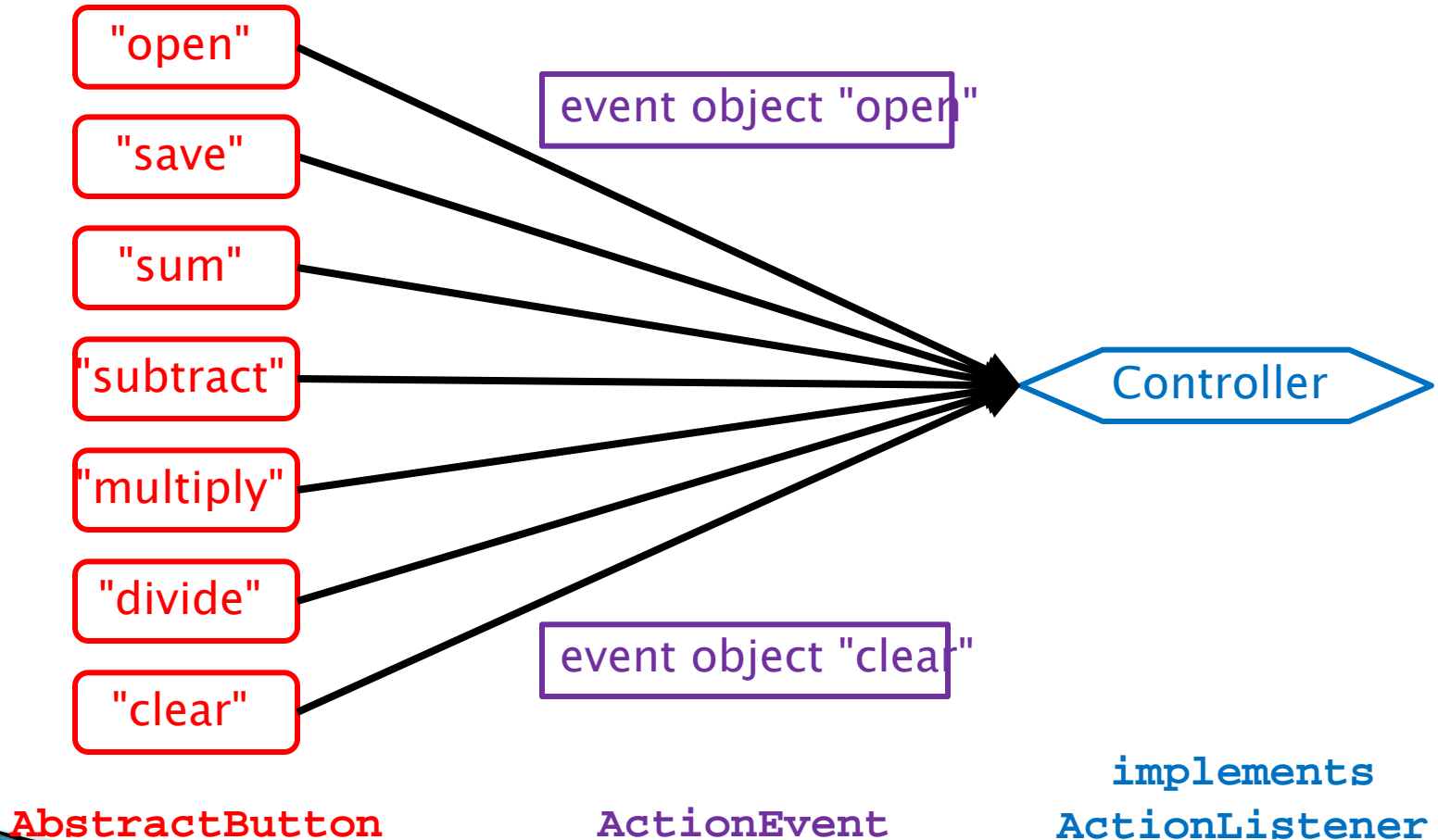
Event Driven Programming

- ▶ 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

```
import java.awt.FlowLayout;
import java.awt.event.ActionListener;
import java.io.File;

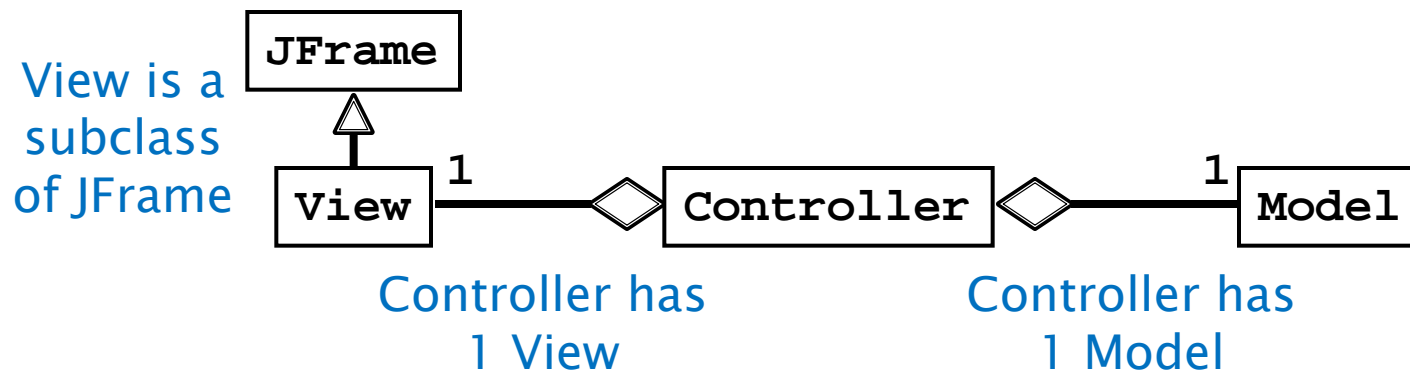
import javax.swing.JButton;
import javax.swing.JFileChooser;
import javax.swing.JFrame;
import javax.swing.JLabel;
import javax.swing.JMenu;
import javax.swing.JMenuBar;
import javax.swing.JMenuItem;
import javax.swing.JTextField;

public class CalcView extends JFrame
{
    private JMenuBar menuBar;
    private JMenu fileMenu;
    private JTextField calcText;
```

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



CalcController: Attributes & Constructor

```
import java.awt.event.ActionListener;

public class CalcController implements ActionListener
{
    private CalcModel model;
    private CalcView  view;

    public CalcController(CalcModel model, CalcView view)
    {
        this.model = model;
        this.view = view;
    }
}
```

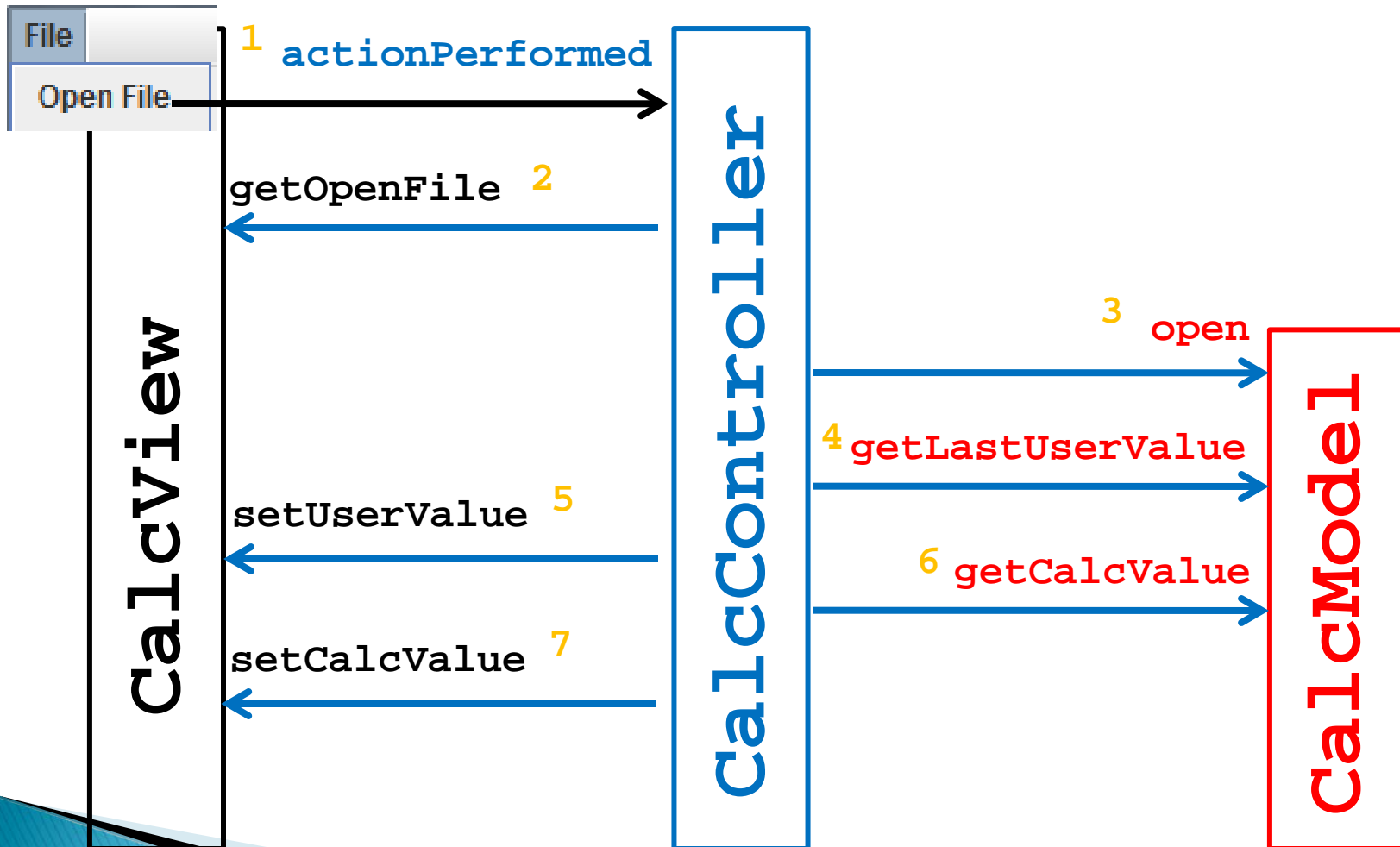
CalcController

- ▶ Recall that our application only uses events that are fired by buttons (`JButtons` and `JMenuItemS`)
 - A button fires an `ActionEvent` event whenever it is clicked
- ▶ `CalcController` listens for fired `ActionEvents`
 - How? by implementing the `ActionListener` interface

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

- ▶ `CalcController` was registered to listen for `ActionEvents` 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

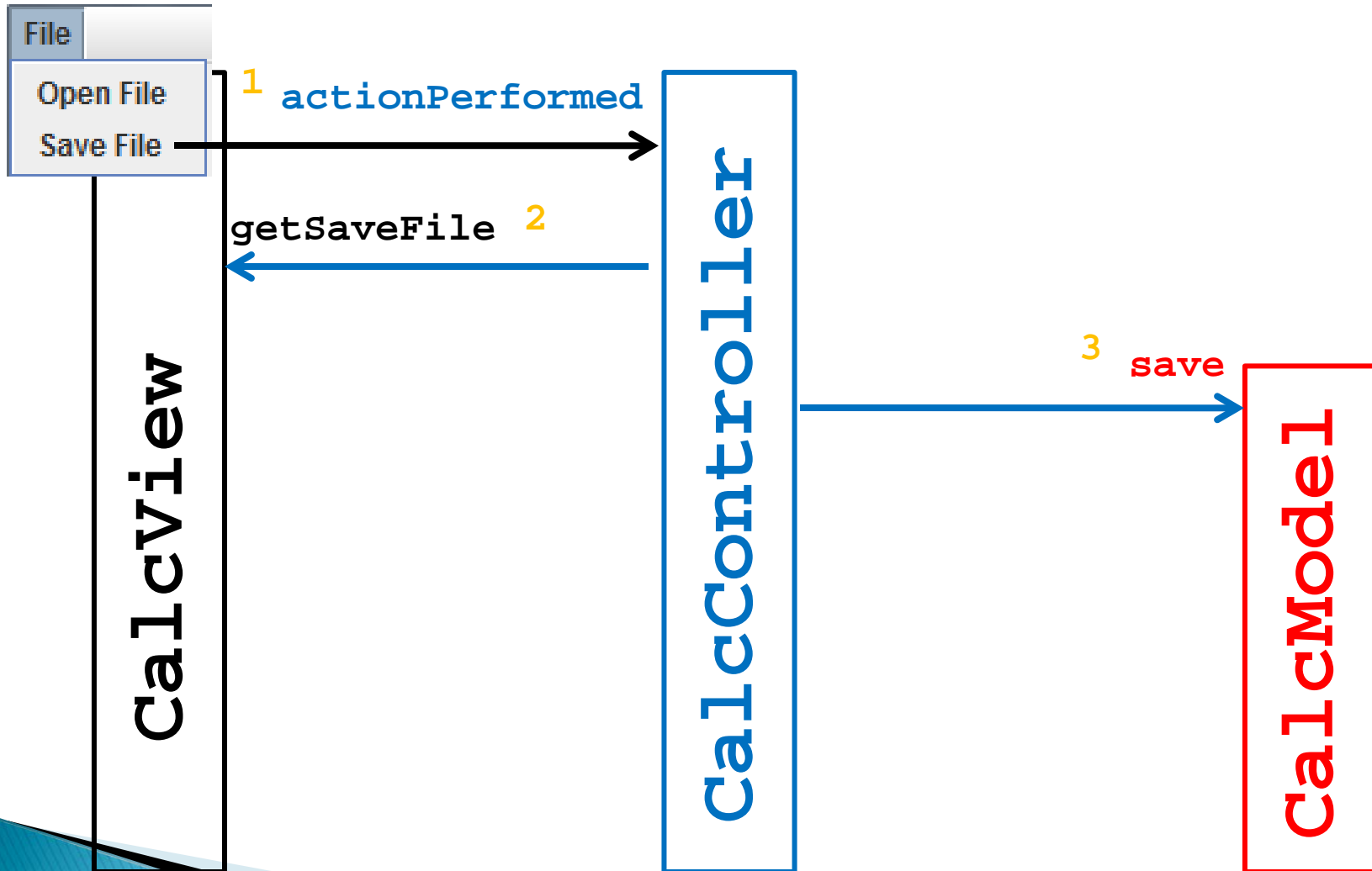
```
import java.awt.event.ActionListener;

public class CalcController implements ActionListener
{
    private CalcModel model;
    private CalcView  view;

    public CalcController(CalcModel model, CalcView view)
    {
        this.model = model;
        this.view = view;
    }

    /**
     * Invoked when an event occurs.
     *
     * @param event
     *           The event.
     */
}
```

Saving a File



CalcController: Save a File

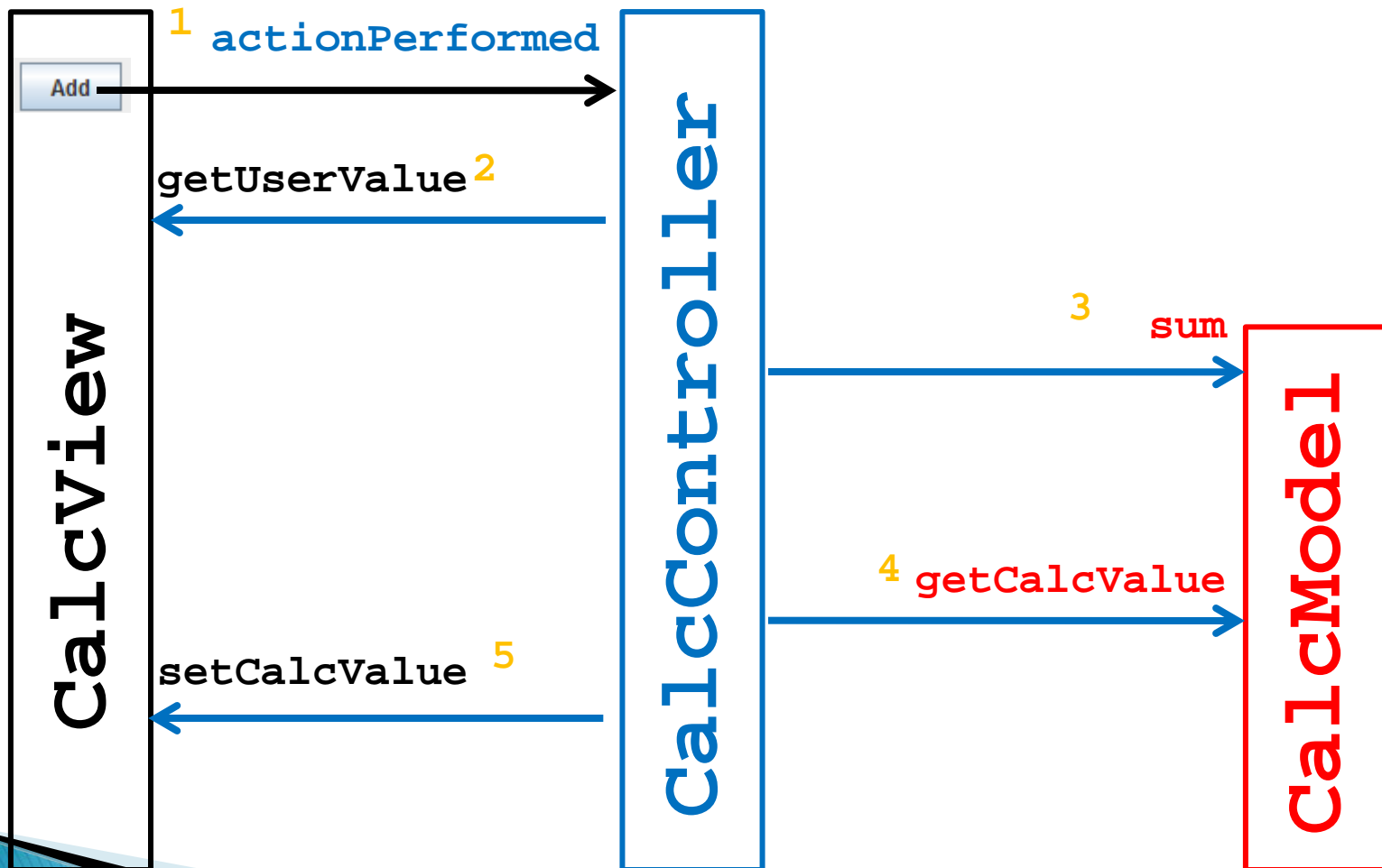
```
import java.awt.event.ActionListener;

public class CalcController implements ActionListener
{
    private CalcModel model;
    private CalcView  view;

    public CalcController(CalcModel model, CalcView view)
    {
        this.model = model;
        this.view = view;
    }

    /**
     * Invoked when an event occurs.
     *
     * @param event
     *           The event.
     */
}
```

Sum, Subtract, Multiply, Divide



CalcController: Other Actions

```
import java.awt.event.ActionListener;

public class CalcController implements ActionListener
{
    private CalcModel model;
    private CalcView  view;

    public CalcController(CalcModel model, CalcView view)
    {
        this.model = model;
        this.view = view;
    }

    /**
     * Invoked when an event occurs.
     *
     * @param event
     *           The event.
     */
}
```

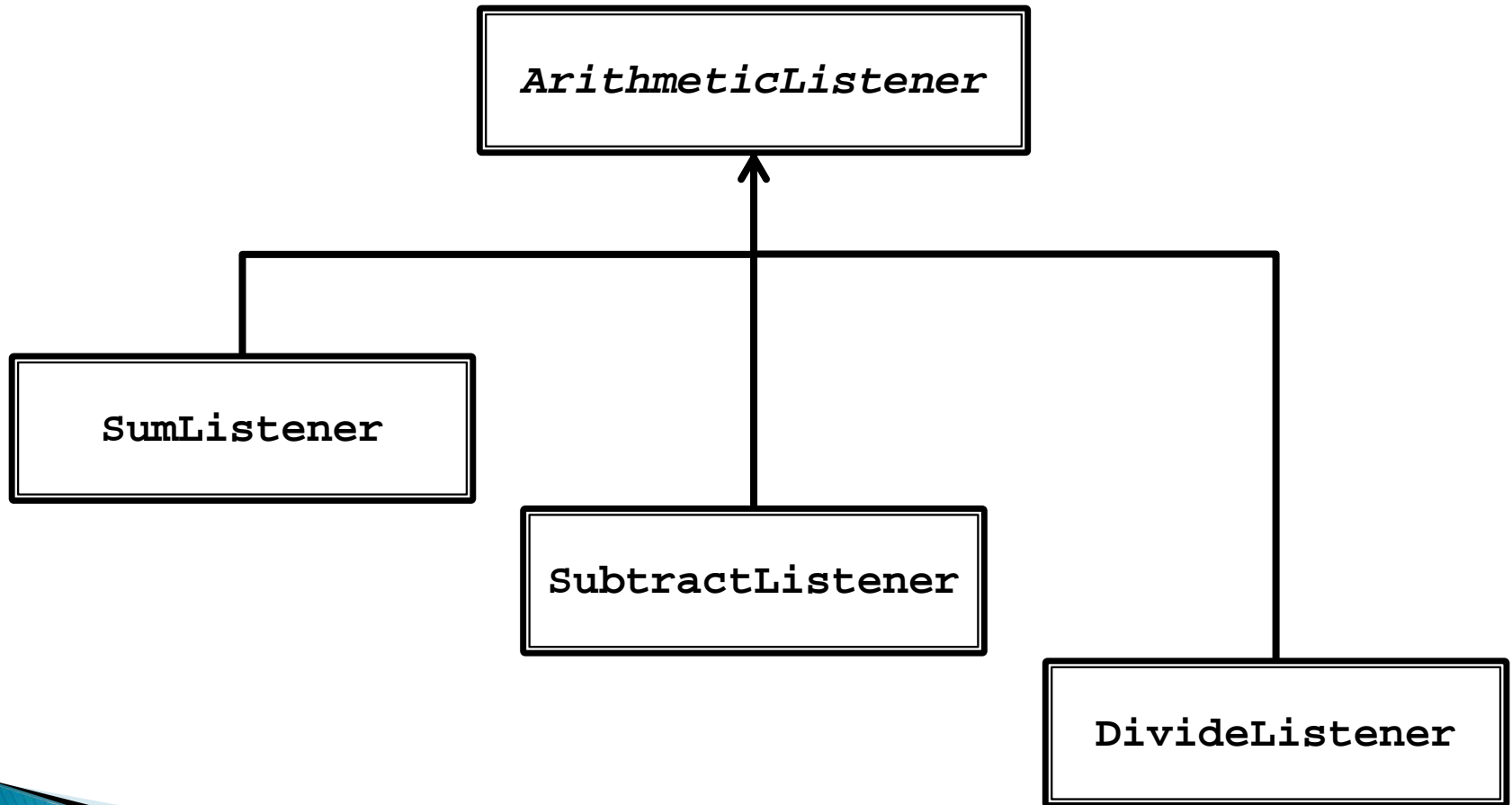
CalcMVC Source Code

- ▶ Source code for the model, view, controller, and app can be found on the course web site with the lectures

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;  
}
```

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

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

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