Graphical User Interfaces

notes Chap 7

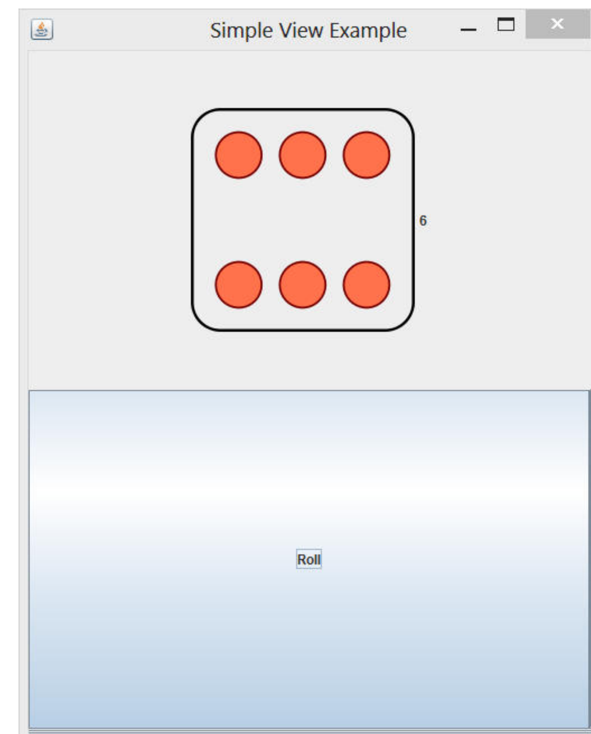
Java Swing

- Swing is a Java toolkit for building graphical user interfaces (GUIs)
  - http://docs.oracle.com/javase/tutorial/uiswing/TOC.html

- old version of the Java tutorial had a visual guide of Swing components
  - http://da2i.univ-lille1.fr/doc/tutorial-java/ui/features/components.html

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

Model-View-Controller

- 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

Model—View—Controller

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

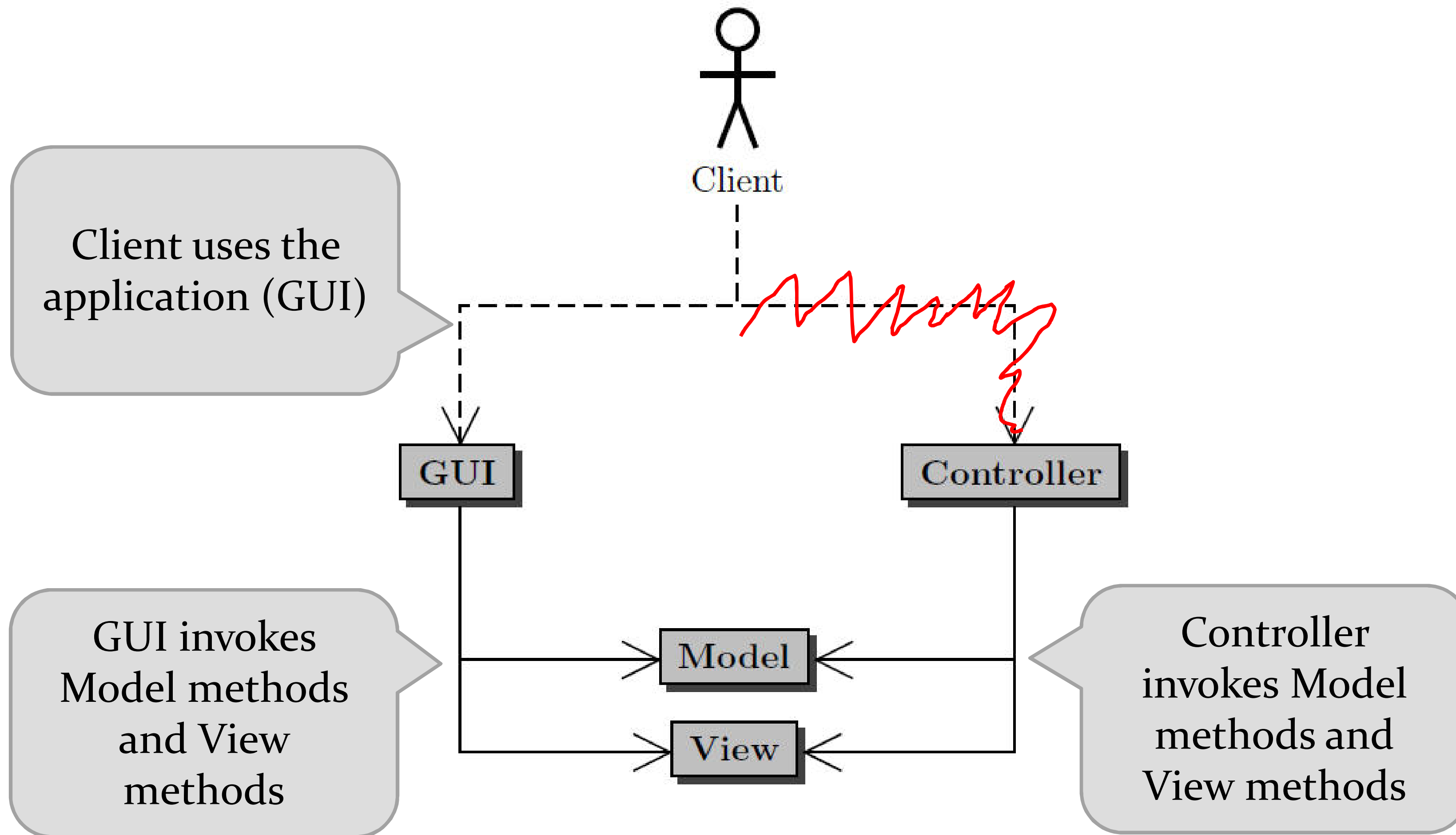
Model

View

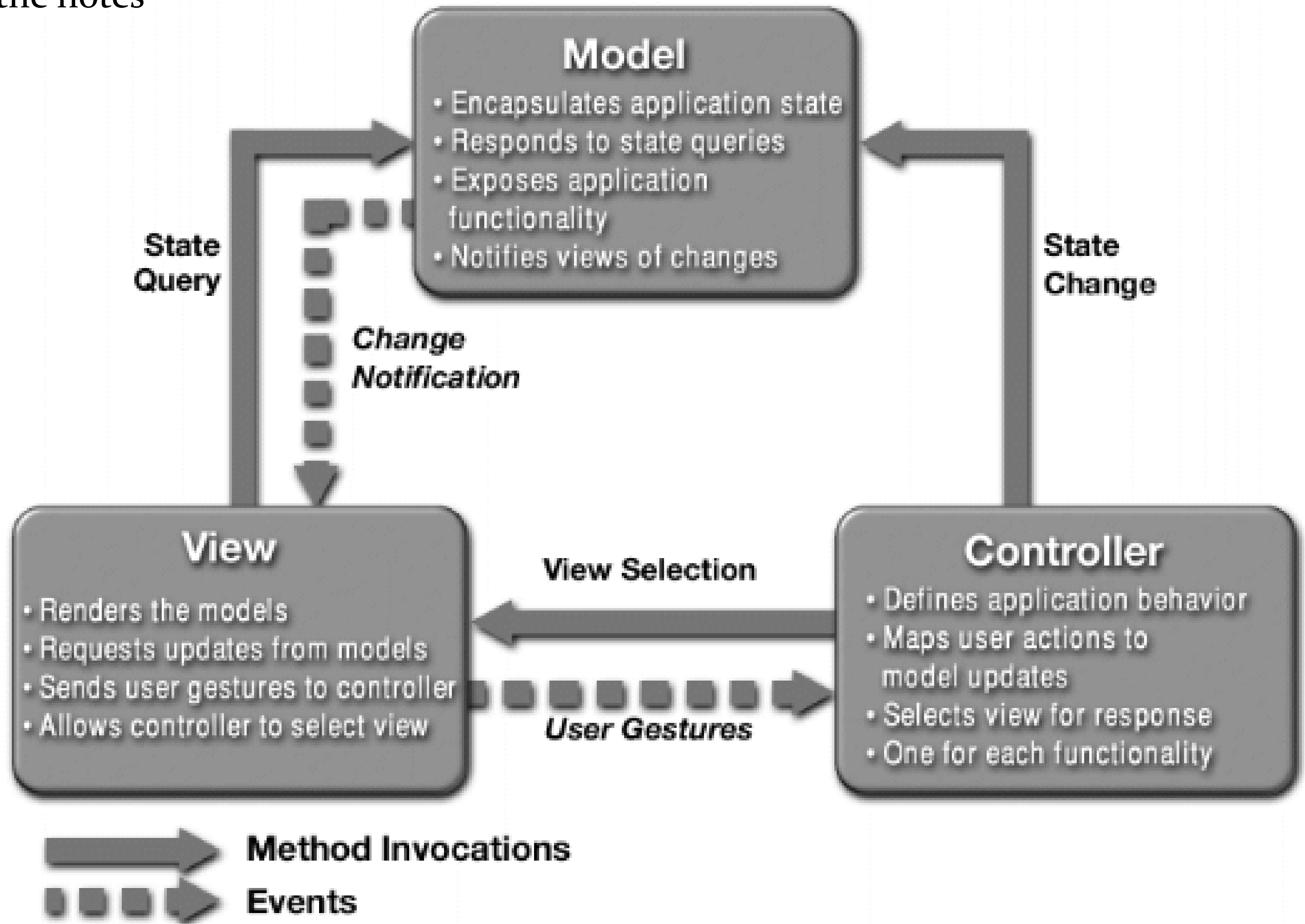
RemoteControl
+ togglePower() : void + channelUp() : void + volumeUp() : void

Controller

Model-View-Controller



a different MVC structure
than in the notes



http://java.sun.com/developer/technicalArticles/javase/mvc/

App to Roll a Die: MVC

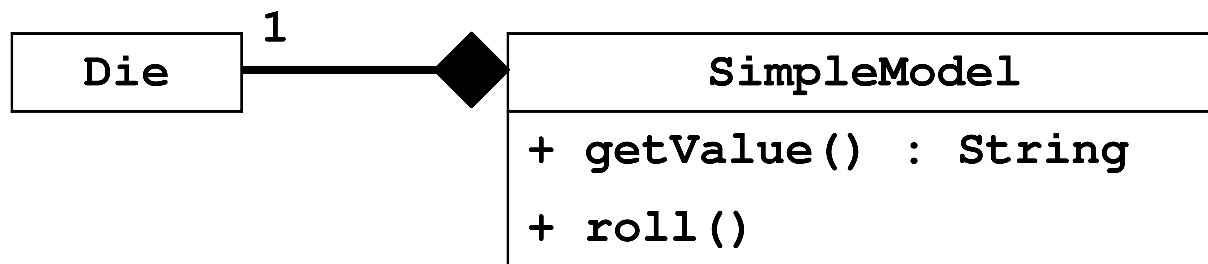
- we can also write the application using the model-view-controller pattern

App to Roll a Die: Model

- model
  - the data
  - methods that get the data (accessors)
  - methods that modify the data (mutators)

- the data
  - a 6-sided die
- accessors
  - get the current face value
- mutators
  - roll the die

App to Roll a Die: Model

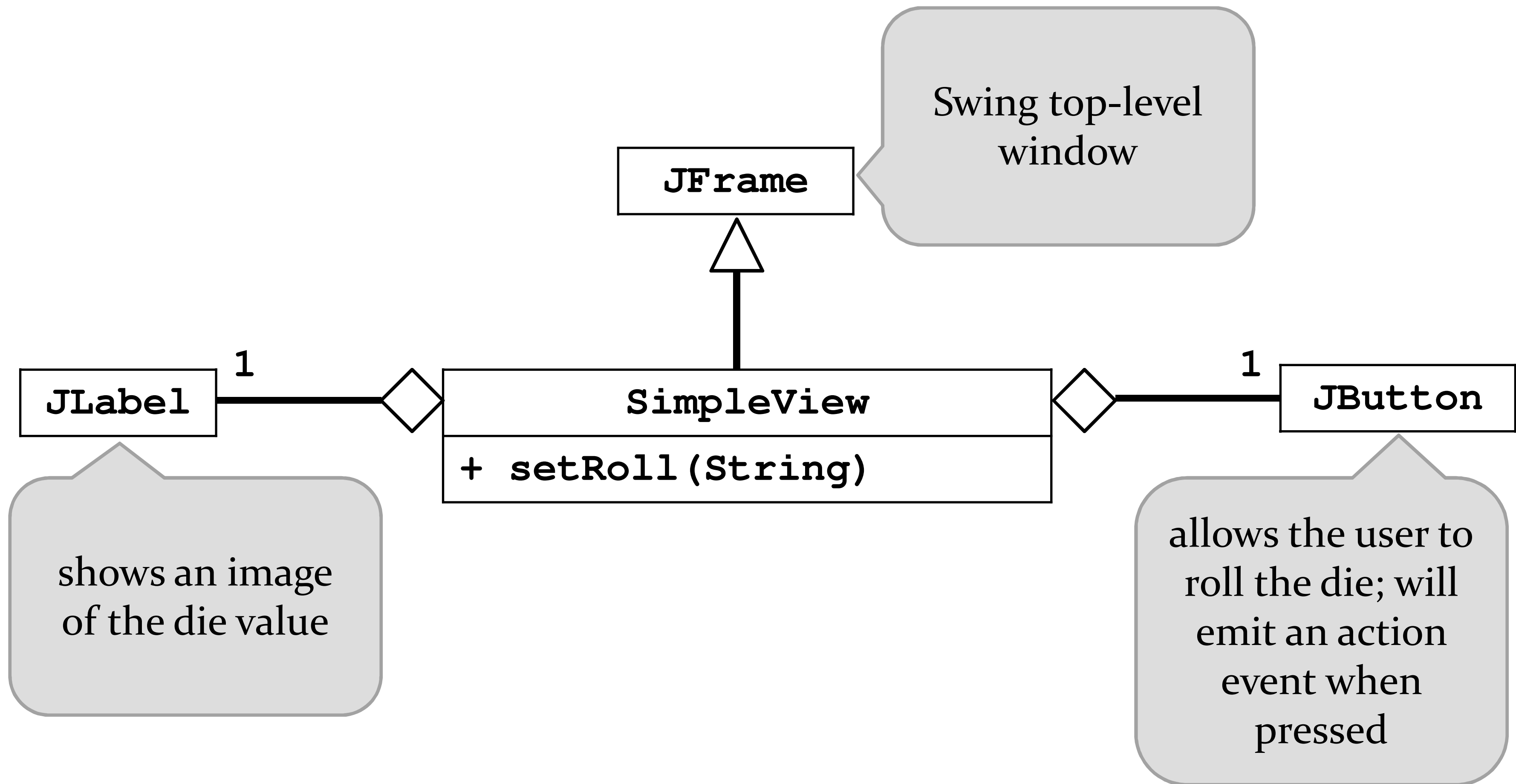


App to Roll a Die: View

- view
  - a visual (or other) display of the model
  - a user interface that allows a user to interact with the view
  - methods that get information from the view (accessors)
  - methods that modify the view (mutators)

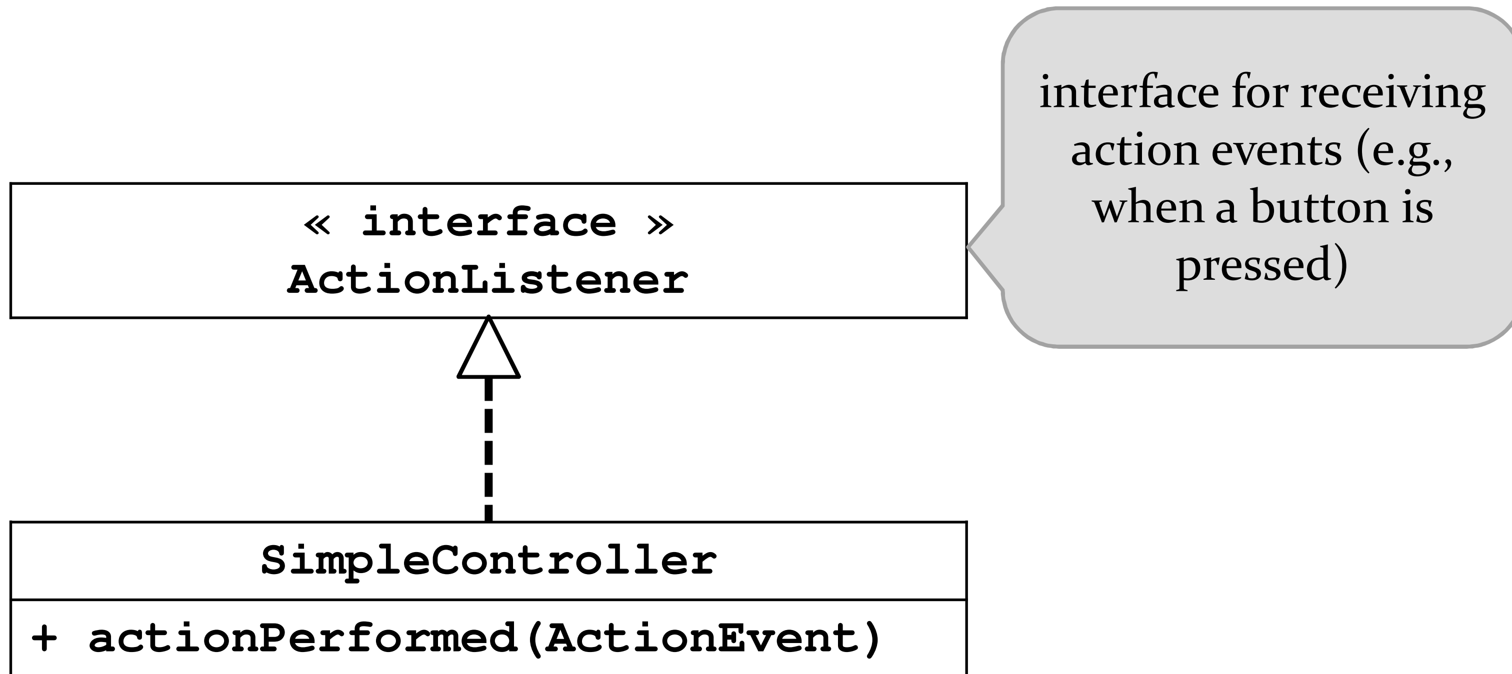
- a visual (or other) display of the model
  - an image of the current face of the die
- a user interface that allows a user to interact with the view
  - roll button

App to Roll a Die: View

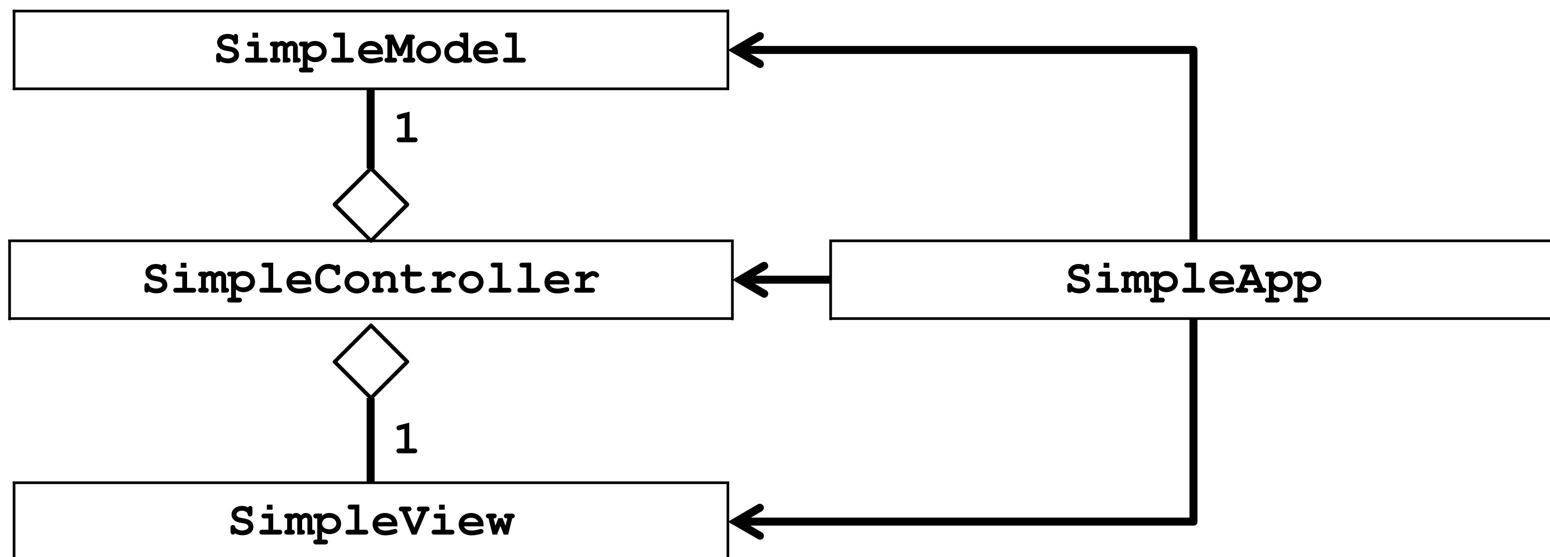


App to Roll a Die: Controller

- controller
  - methods that map user interactions to model updates



App to Roll a Die: MVC



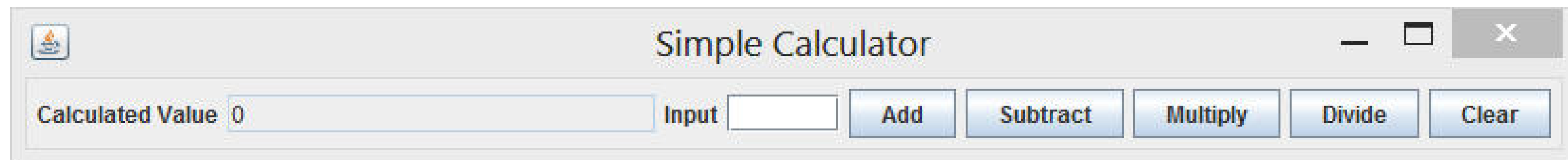
App to Roll a Die

- 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

Application Appearance



Creating the Application

- the calculator application is launched by the user
  - the notes 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

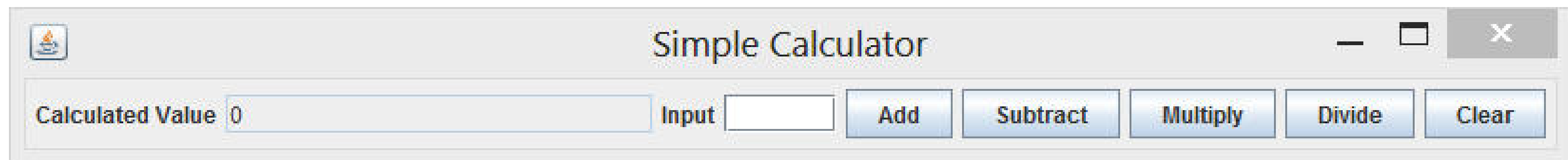
CalcMVC Application

```
public class CalcMVC
{
    public static void main(String[] args)
    {
        CalcController controller = new CalcController();
        CalcModel model = new CalcModel();
        CalcView  view  = new CalcView(model, controller);
        controller.setModel(model);
        controller.setView(view);

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

Model

- features:
  - sum, subtract, multiply, divide
  - clear



CalcModel
- calcValue : int
+ getCalcValue() : int + getLastUserValue() : int + sum(int) : void + subtract(int) : void + multiply(int) : void + divide(int) : void + clear() : void

CalcModel: Attributes and Ctor

```
public class CalcModel
{
  private int calcValue;

  /**
   * Creates a model with a calculated value of zero.
   */
  public CalcModel() {
    this.calcValue = 0;
  }
```

CalcModel: clear

```
/**
 * Clears the user values and the calculated value.
 */
public void clear() {
  this.calcValue = 0;
}
```

CalcModel: getCalcValue

```
/**
 * Get the current calculated value.
 *
 * @return The current calculated value.
 */
public int getCalcValue() {
  return this.calcValue;
}
```

CalcModel: sum

```
/**
 * Adds the calculated value by a user value.
 *
 * @param userValue
 *          The value to add to the current calculated
 *          value by.
 */
public void sum(int userValue) {
  this.calcValue += userValue;
}
```

CalcModel: subtract and multiply

```
public void subtract(int userValue) {
  this.calcValue -= userValue;
}


public void multiply(int userValue) {
  this.calcValue *= userValue;
}
```

CalcModel: divide

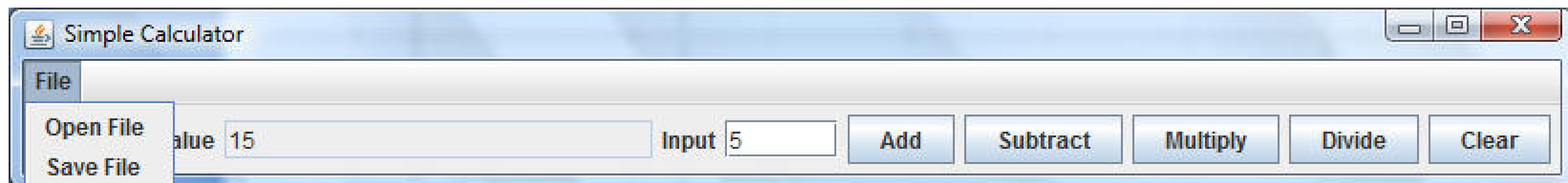
```
/**
  * Divides the calculated value by a user value.
  *
  * @param userValue
  *            The value to multiply the current calculated
  *            value by.
  * @pre. userValue is not equivalent to zero.
  */
 public void divide(int userValue) {
   this.calcValue /= userValue;
 }
```

Other model examples

- consider the Boggle app from CSE1030 last term
  - http://www.eecs.yorku.ca/course_archive/2013-14/F/1030/labs/07/lab7.html
- consider Eclipse
- pick your favourite game and design a model for the game

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)

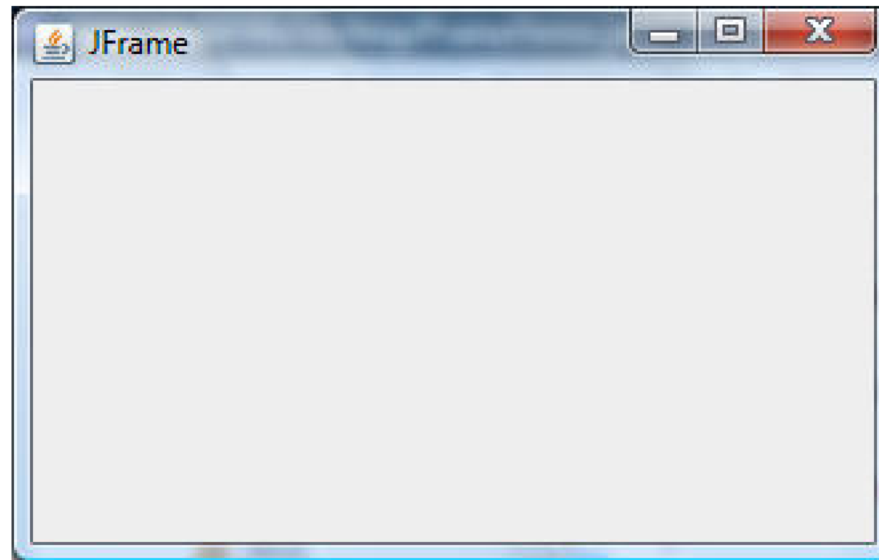


Simple Applications

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

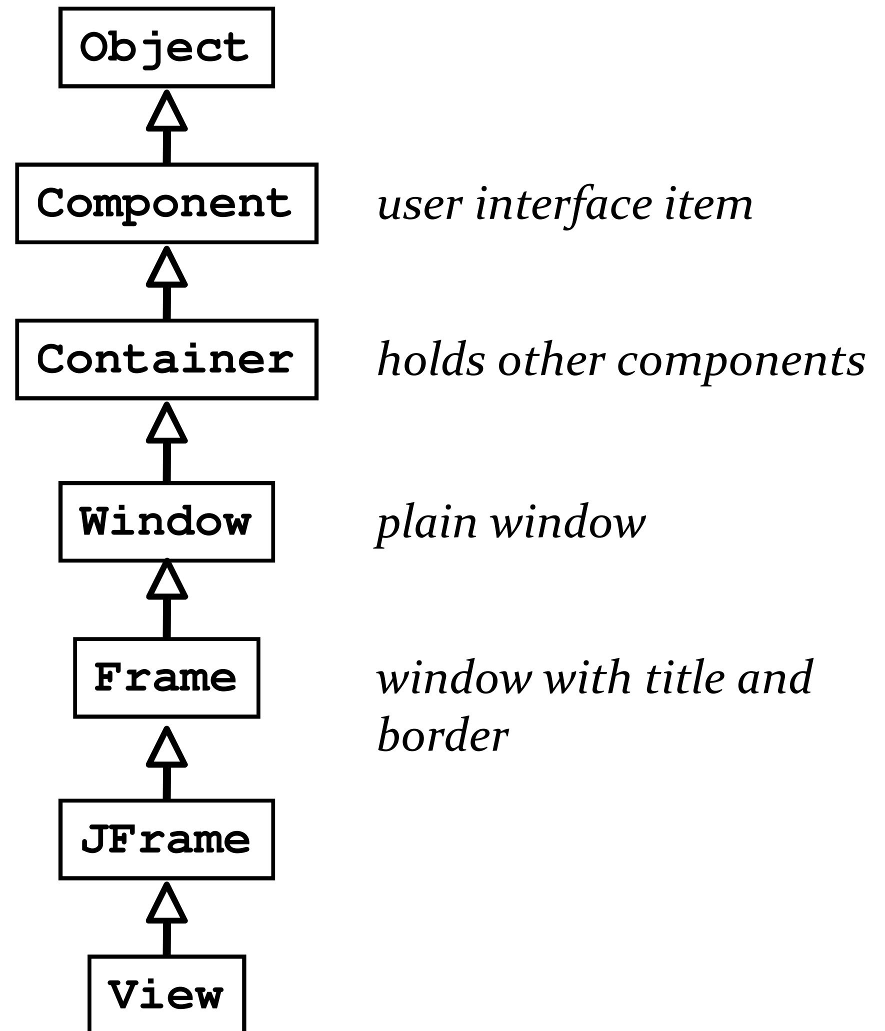
JFrame

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)



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
    - buttons JButton
    - labels JLabel
    - text fields JTextField
  - 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

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)

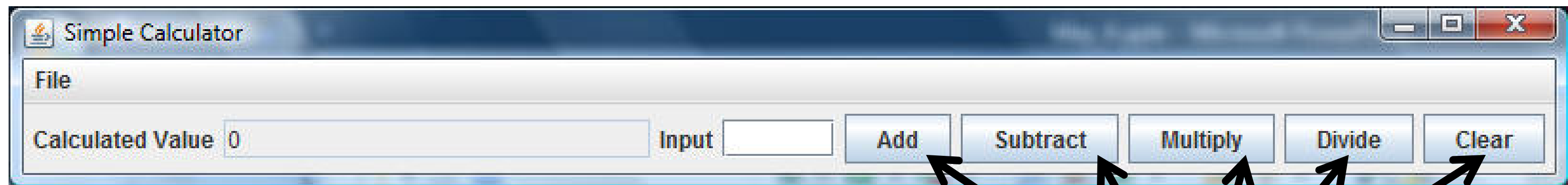
```
JTextField textField = new JTextField(20);
```

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)



button
`JButton`

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

Buttons

- to create a button

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

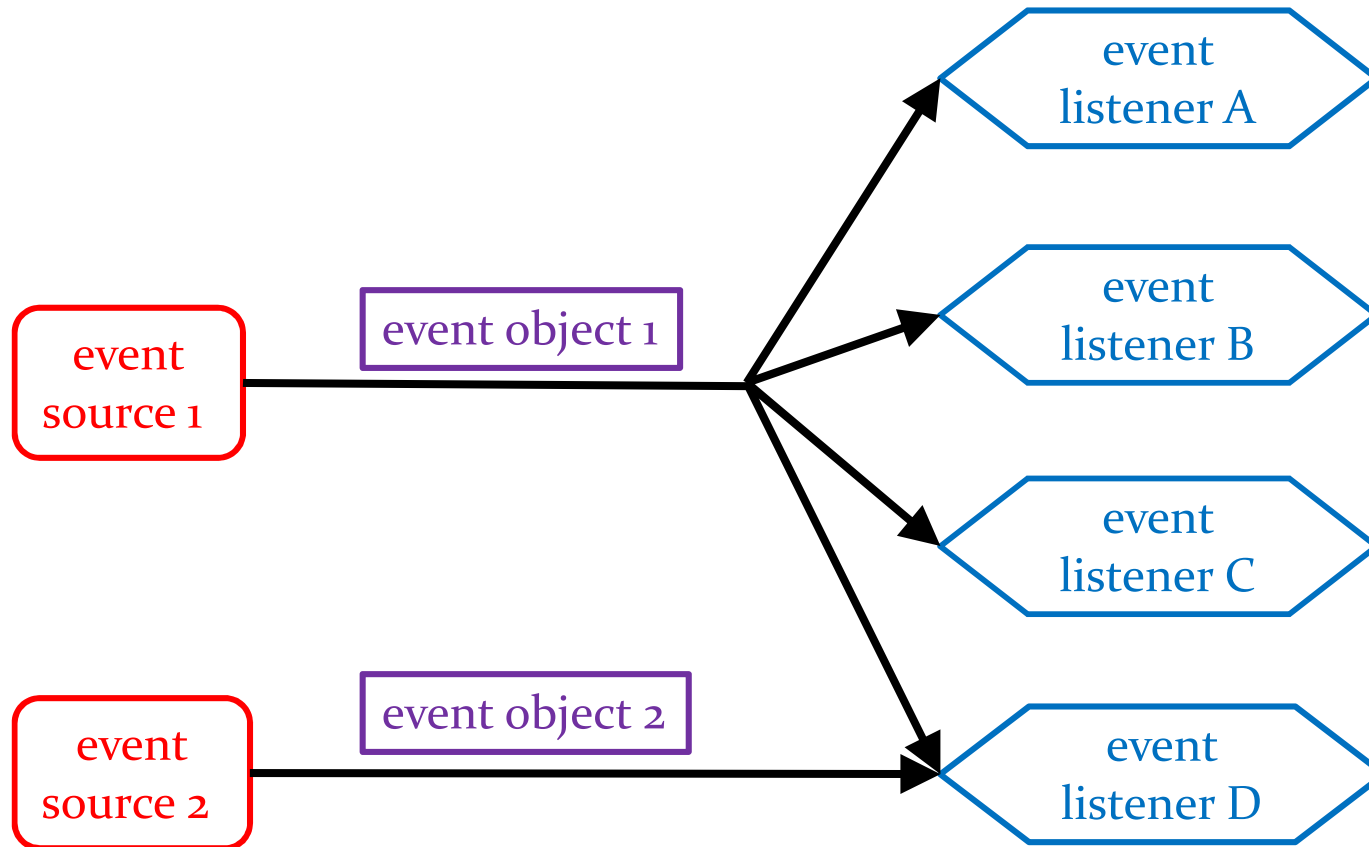
Adding the Buttons

- see CalcView constructor
  - try enabling and disabling the buttons

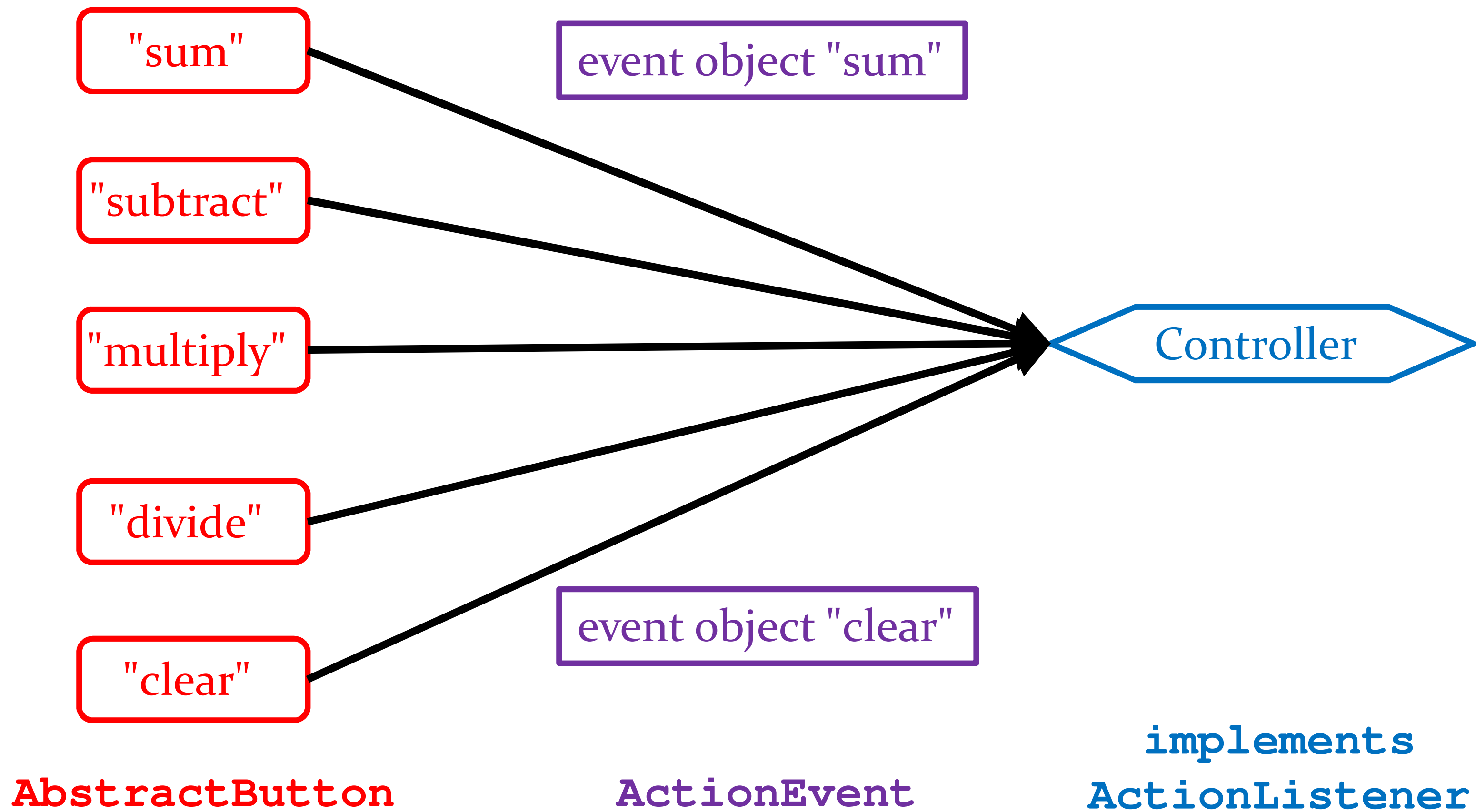
Event Driven Programming

- so far we have a View with some UI elements (buttons, text fields)
  - 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`** has two inherited methods from **`AbstractButton`**

```
public void addActionListener(ActionListener l)

public void setActionCommand(String actionCommand)
```

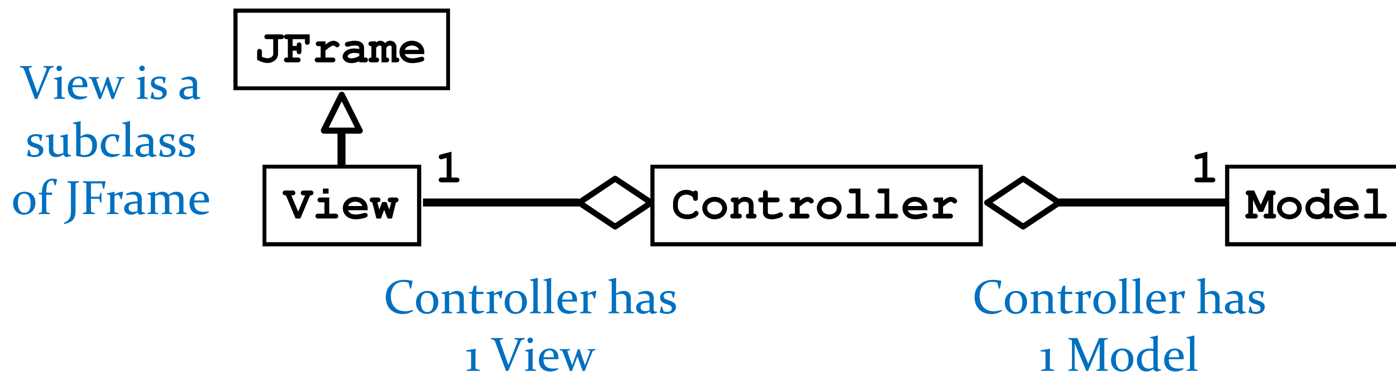
- for each **`JButton`**
  1. call **`addActionListener`** with the controller as the argument
  2. call **`setActionCommand`** with a string describing what event has occurred

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

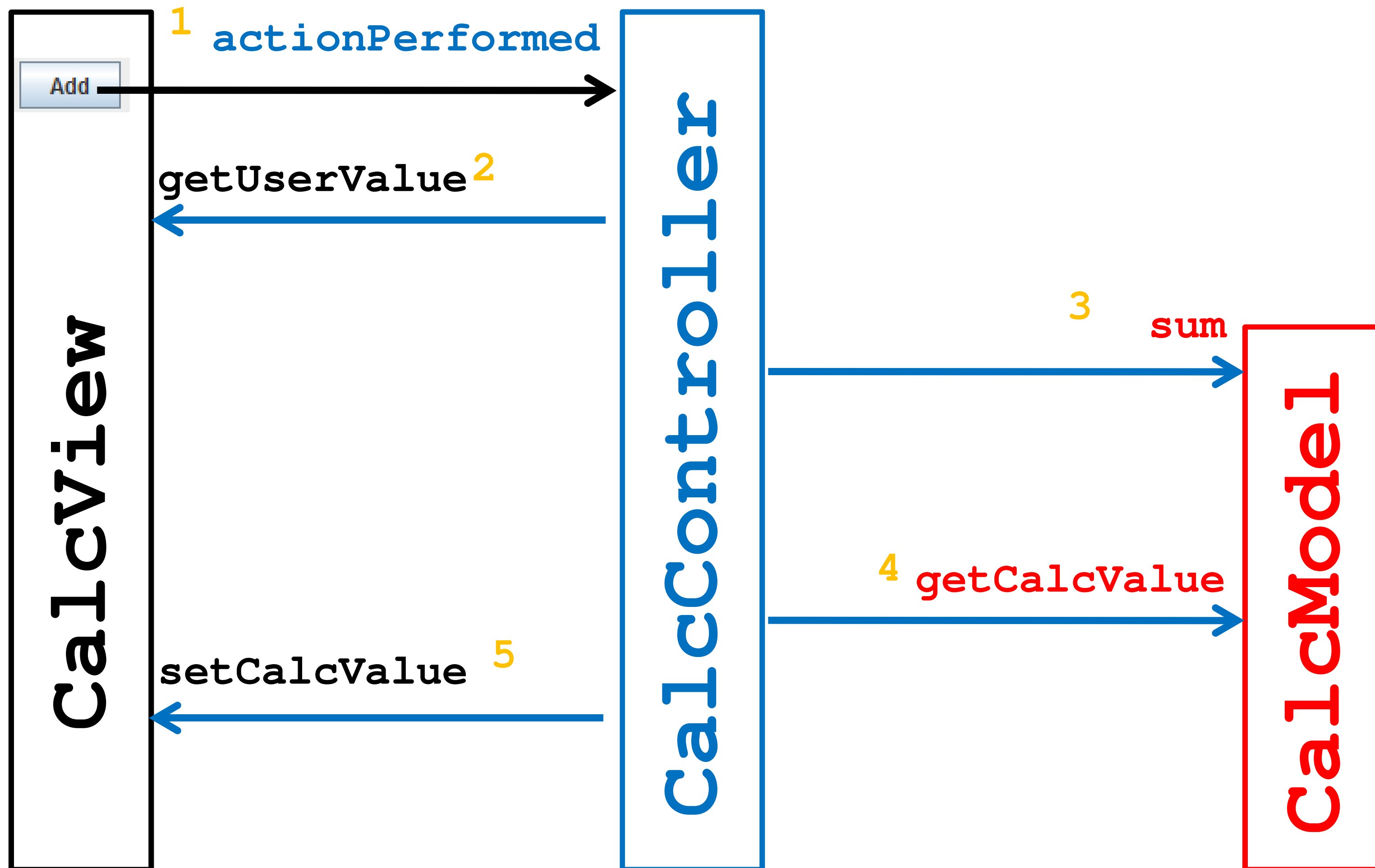
CalcController

- recall that our application only uses events that are fired by buttons (**`Jbuttons`**)
  - 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)

Sum, Subtract, Multiply, Divide



CalcController: Other Actions

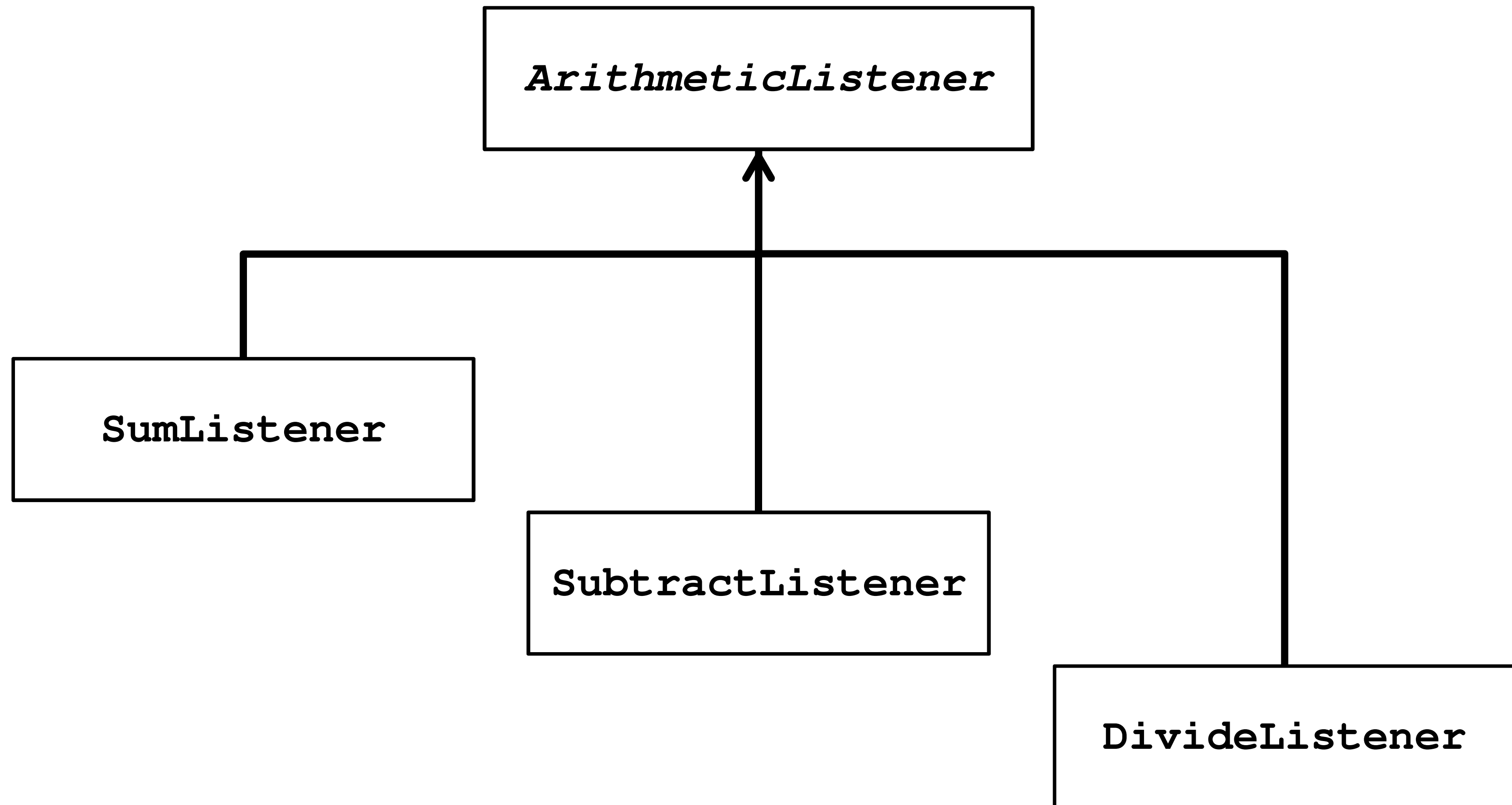
- see CalcController actionPerformed method

actionPerformed

- even with only 5 buttons 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 an int
     - **`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.   int userValue = this.getUserValue();
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(int userValue);
```

ArithmeticListener

```
private int getUserValue() {
  int userValue = 0;
  try {
    userValue = Integer.parseInt(getView().getUserValue());
  }
  catch(NumberFormatException ex)
  {}
  return userValue;
}

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

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(int userValue) {
      // ...
    }
  }
}
```

SumListener

```
private class SumListener extends ArithmeticListener {
  @Override
  protected void doOperation(int userValue) {
    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