Graphical User Interfaces

[notes Chap 7] and [AJ Chap 17, Sec 13.2]

Model—View—Controller

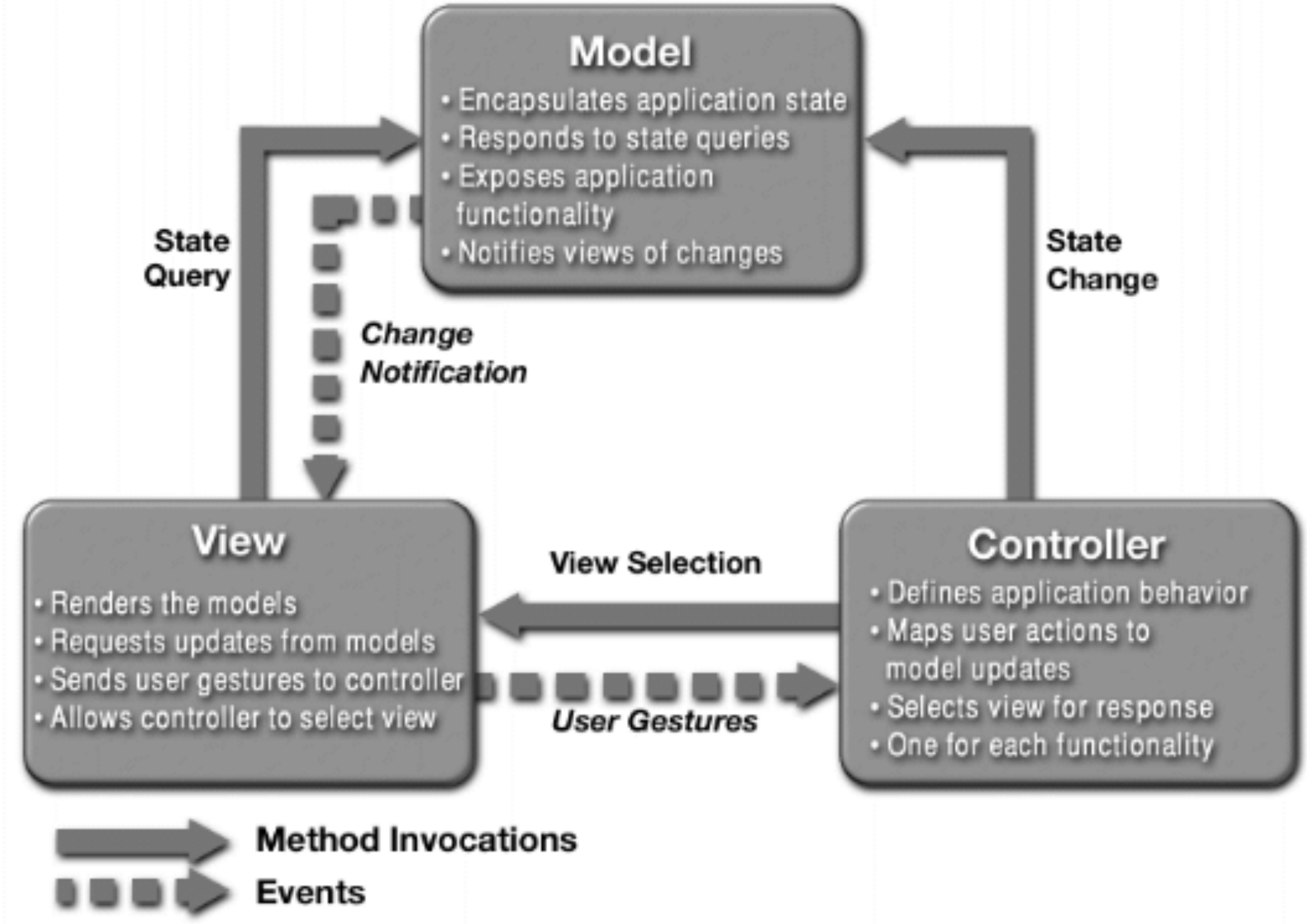
TV
- on : boolean *- channel : int* *- volume : int*
+ power(boolean) : void *+ channel(int) : void* *+ volume(int) : void*

Model

View

Controller

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



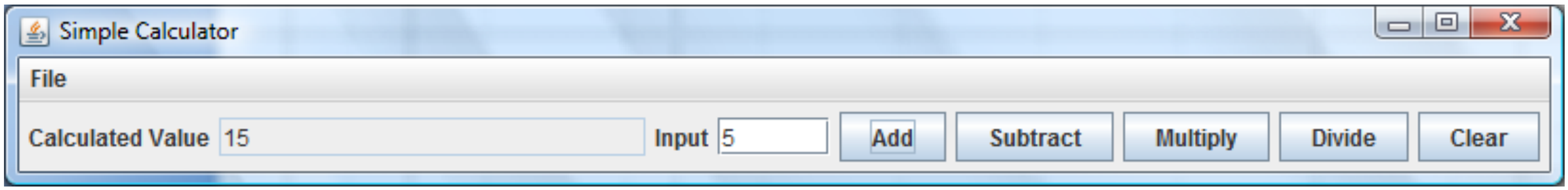
http://java.sun.com/developer/technicalArticles/javase/mvc/

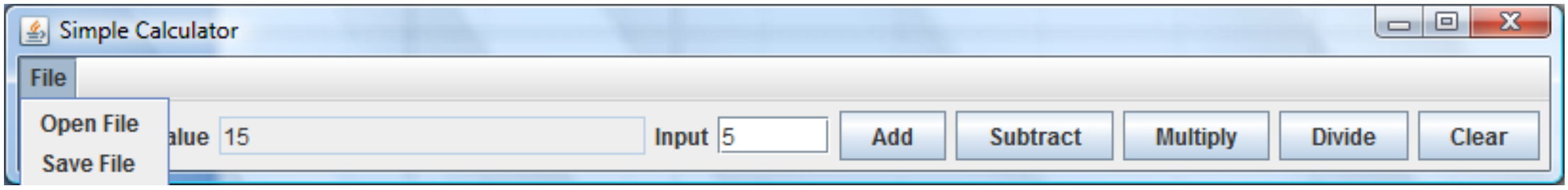
- model
  - represents state of the application and the rules that govern access to and updates of state
- view
  - presents the user with a sensory (visual, audio, haptic) representation of the model state
  - a user interface element (the user interface for simple applications)
- controller
  - processes and responds to events (such as user actions) from the view and translates them to model method calls

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



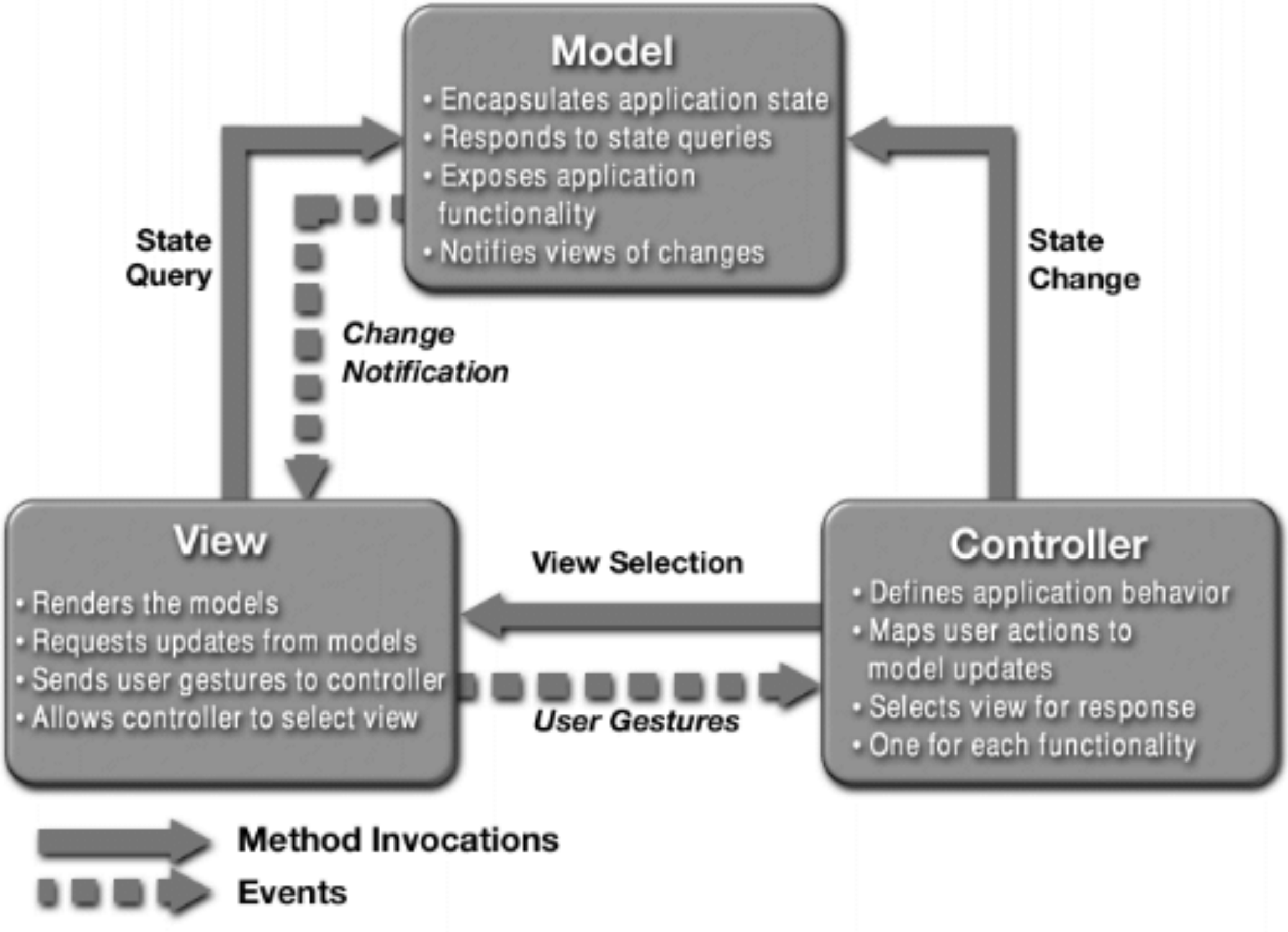


Creating the Application

- the calculator application is launched by the user
  - the 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

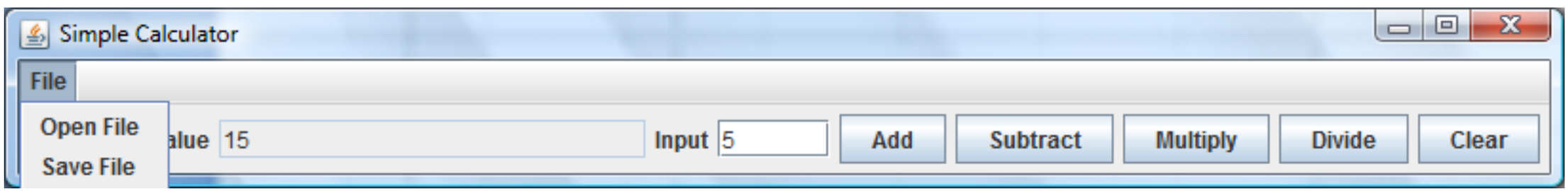
```
public class CalcMVC
{
    public static void main(String[] args)
    {

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

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



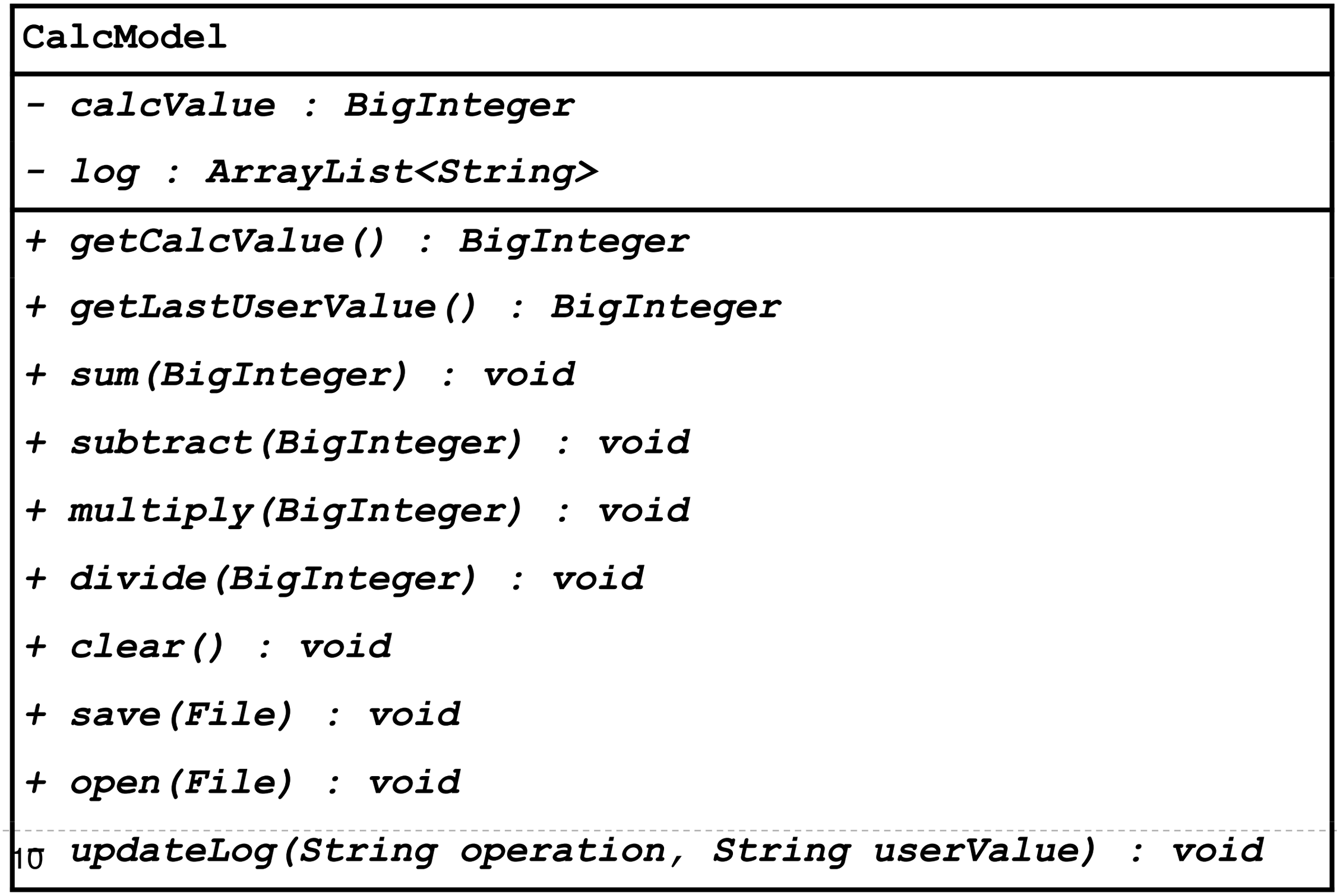
http://java.sun.com/developer/technicalArticles/javase/mvc/

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: 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

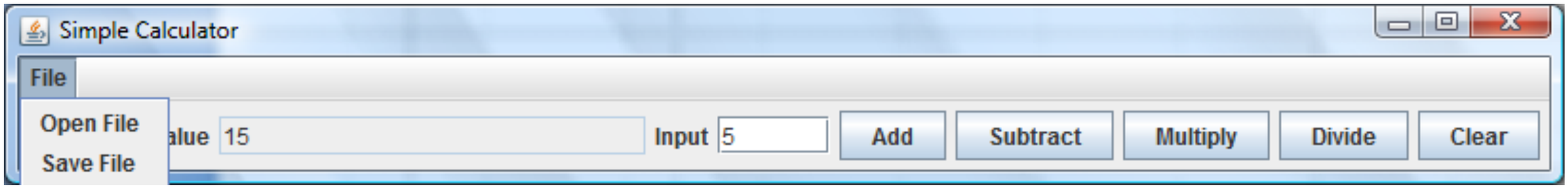
```
// relies on fact ArrayList implements Serializable
public void save(File file)
{
  FileOutputStream f = null;
  ObjectOutputStream out = null;
  try {
    f = new FileOutputStream(file);   // can throw
    out = new ObjectOutputStream(f);  // can throw
    out.writeObject(this.log);        // can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

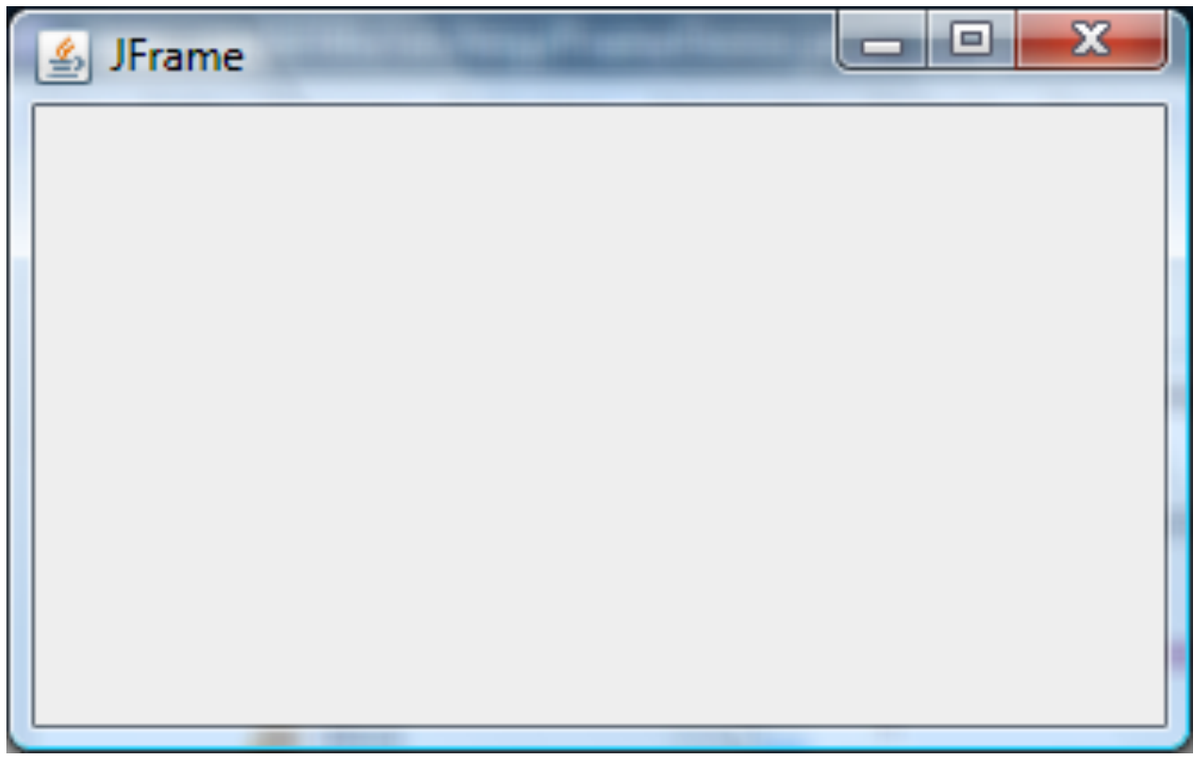


Simple Applications

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

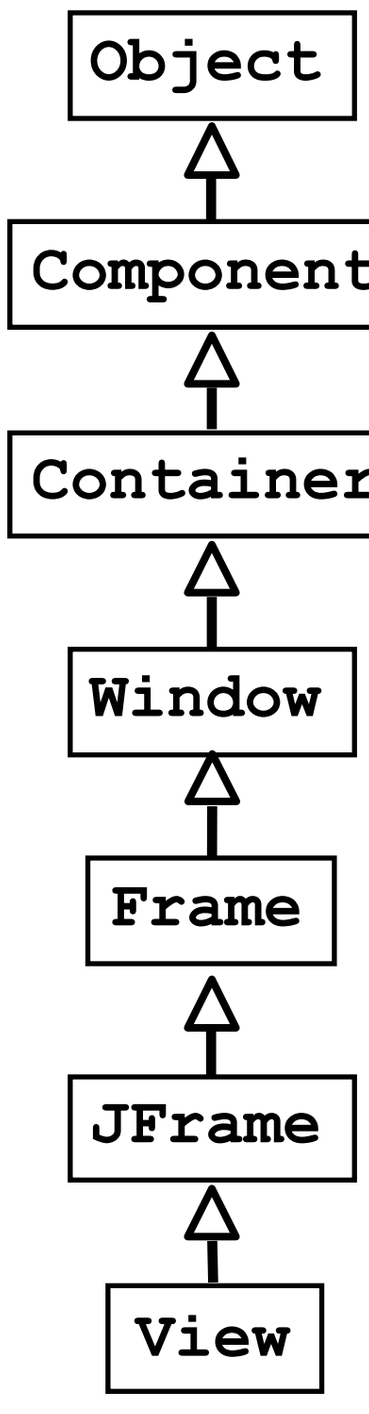
JFrame

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)



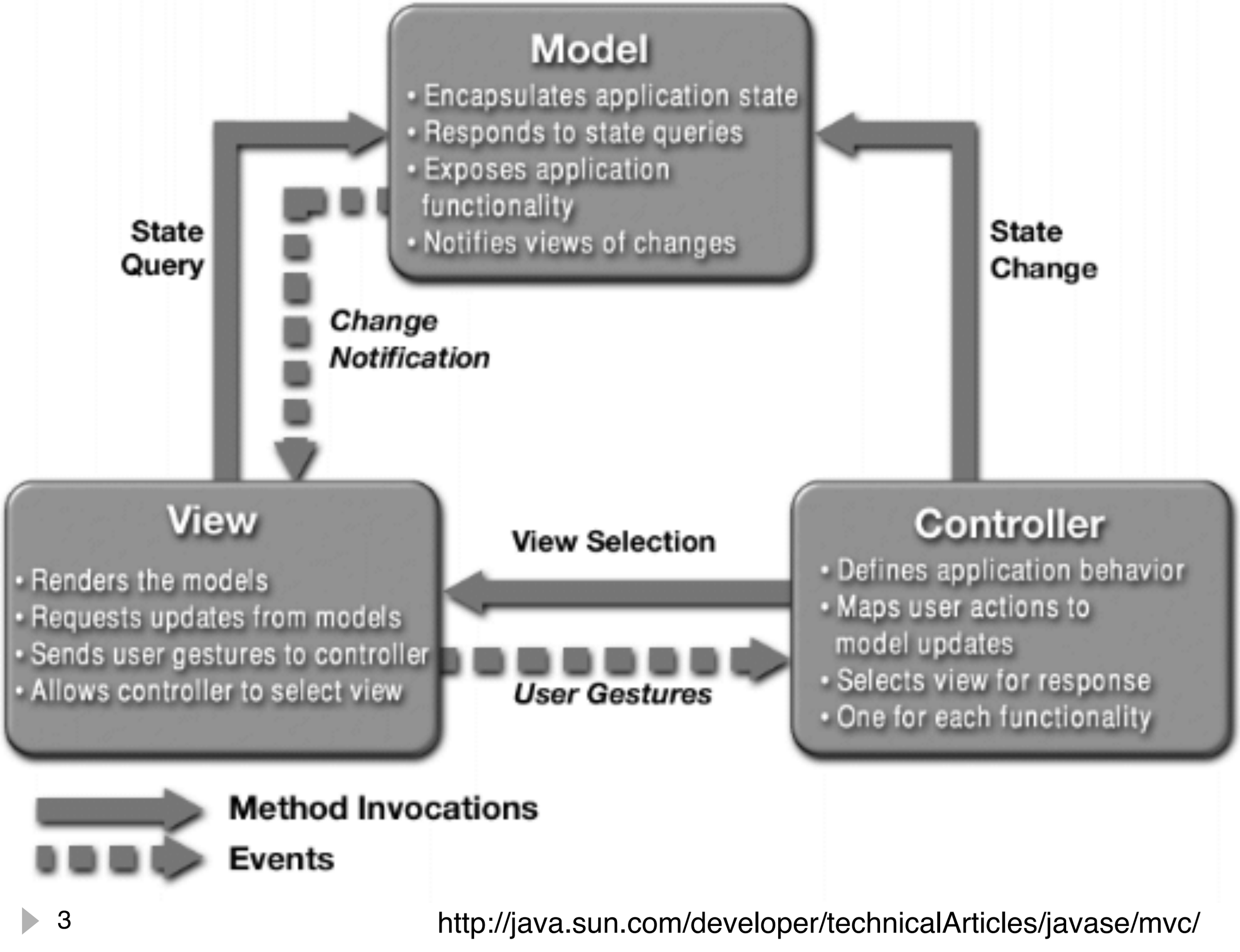
user interface item

holds other components

plain window

window with title and border

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



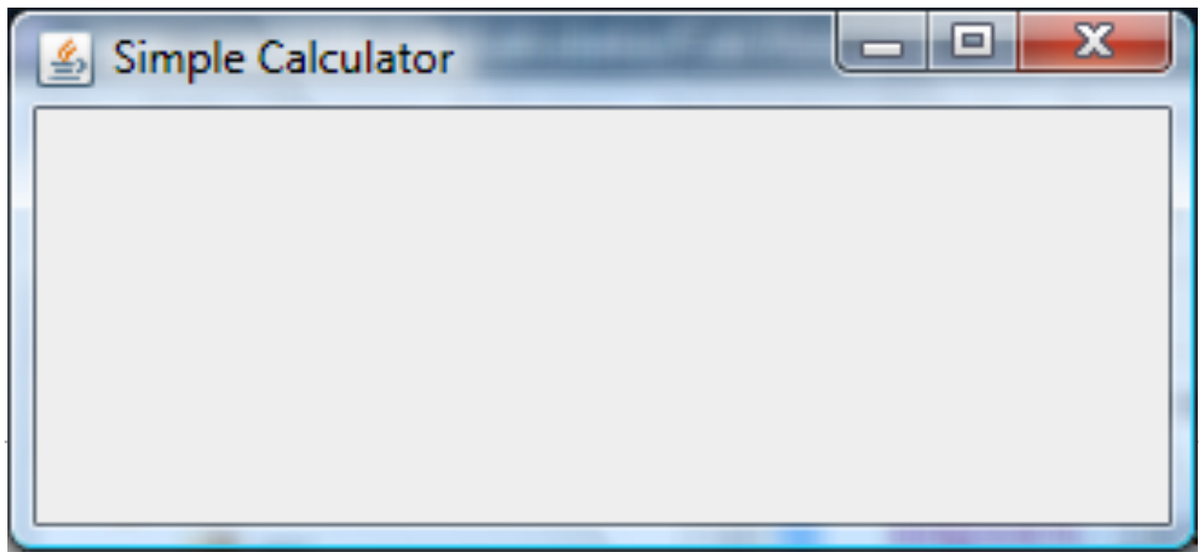
http://java.sun.com/developer/technicalArticles/javase/mvc/

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

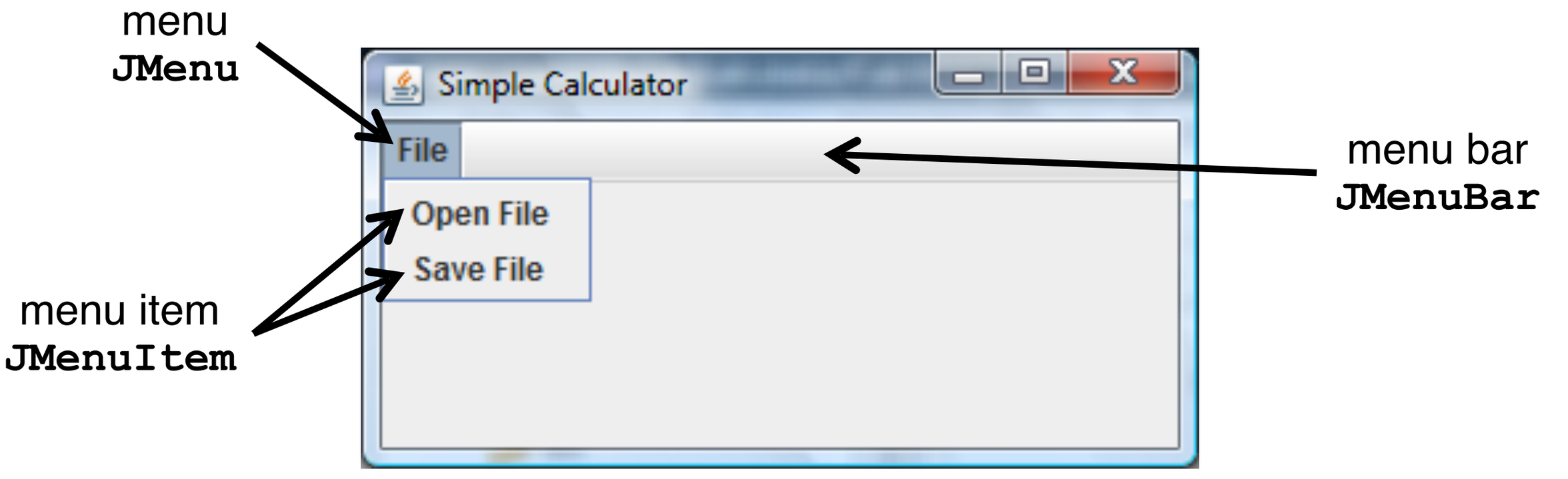
CalcView: Create Controller

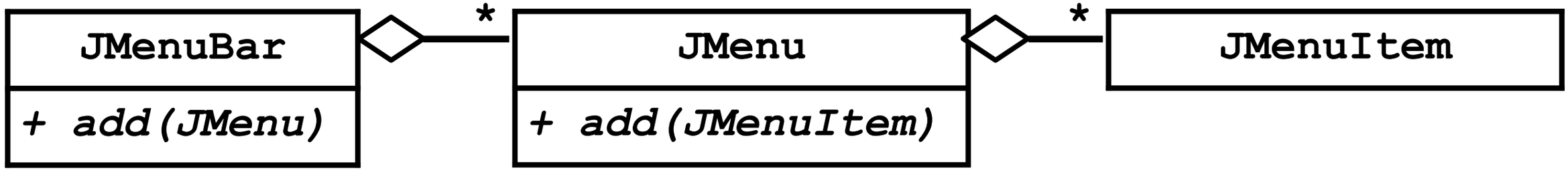
```
public class CalcView extends JFrame
{
  public CalcView(CalcModel model)
  {
    super("Simple Calculator");
    model.clear();
    CalcController controller =
                  new CalcController(model, this);

    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*





CalcView: Menubar, Menu, Menu Items

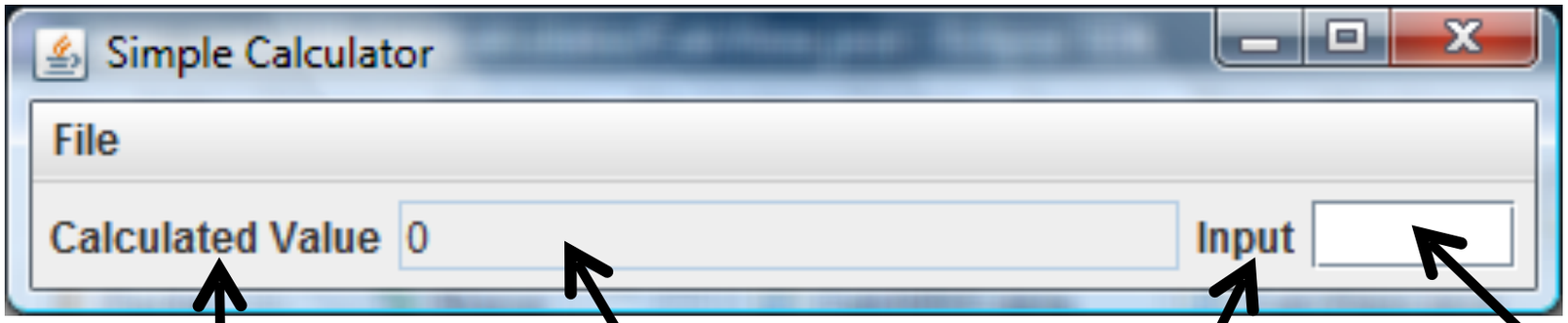
```
import javax.swing.JFrame;
import javax.swing.JMenu;
import javax.swing.JMenuBar;
import javax.swing.JMenuItem;

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();
    this.setJMenuBar(this.menuBar);
```

CalcMVC Application

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

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

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

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`

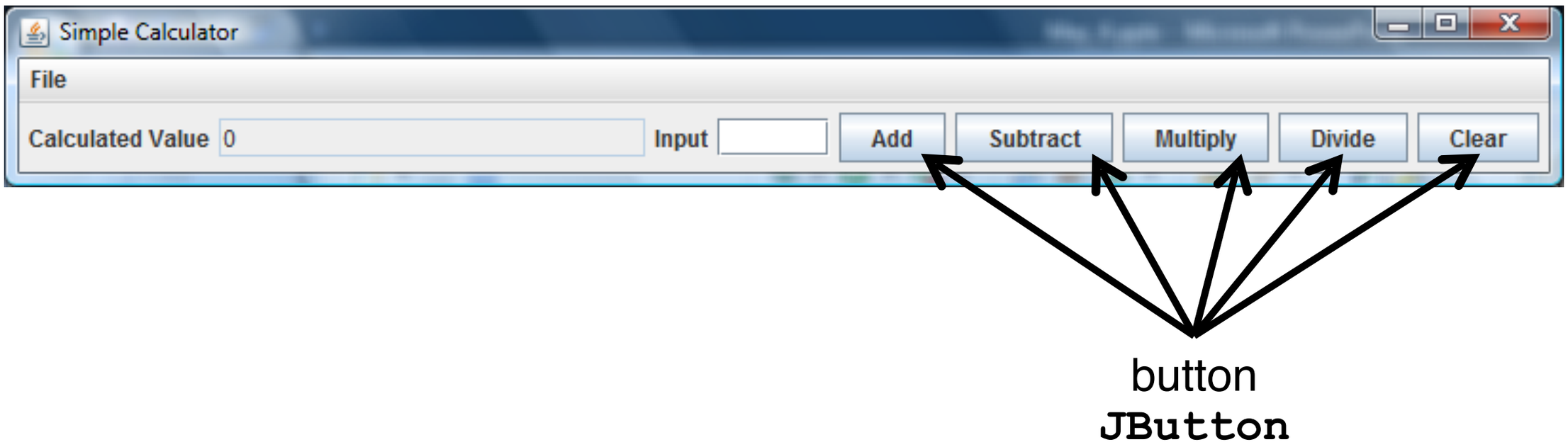
http://java.sun.com/docs/books/tutorial/uiswing/components/label.html

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)
  {
    super("Simple Calculator");
```

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)



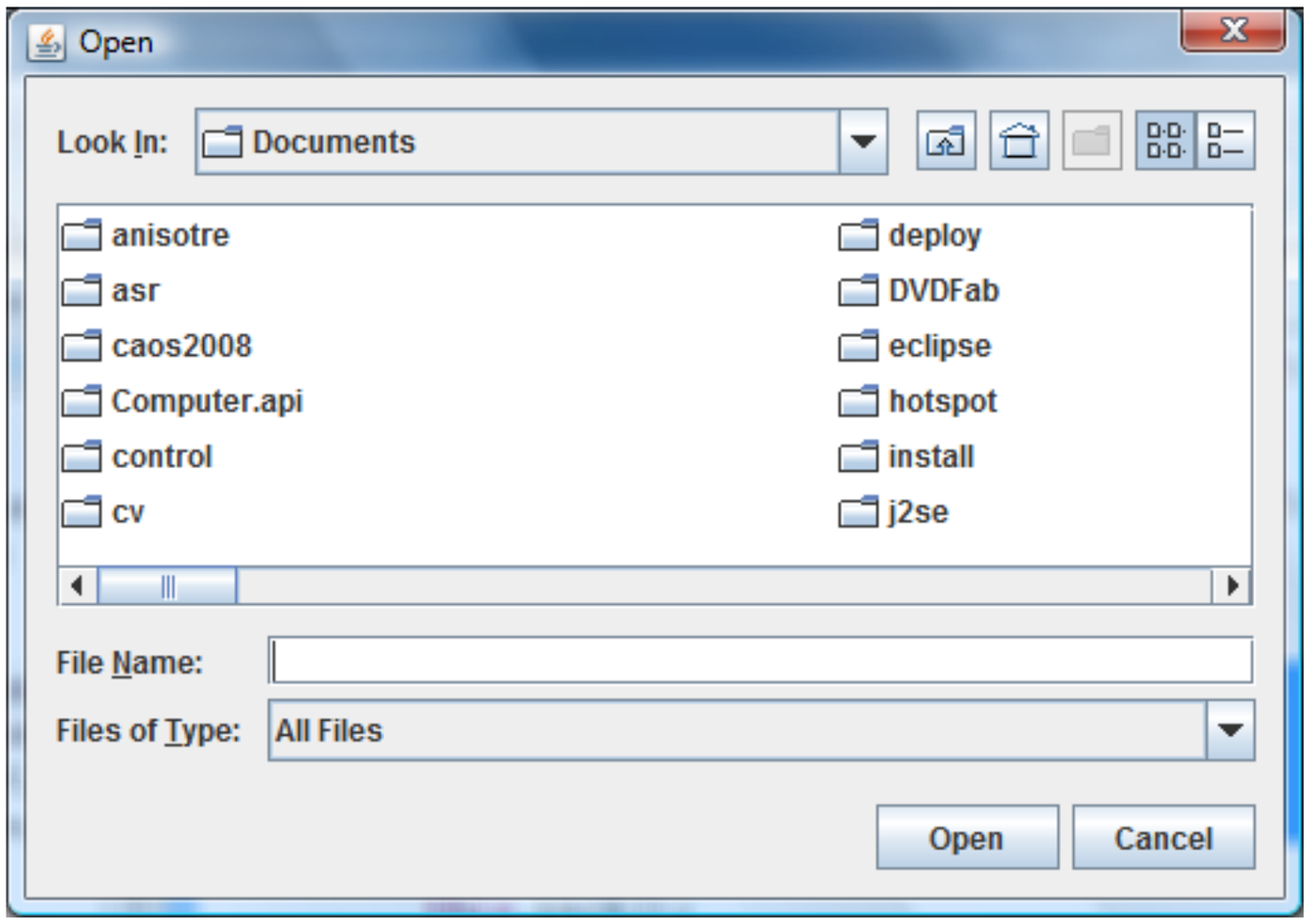
http://java.sun.com/docs/books/tutorial/uiswing/components/button.html

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;
  private JButton multiplyButton;
```

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`

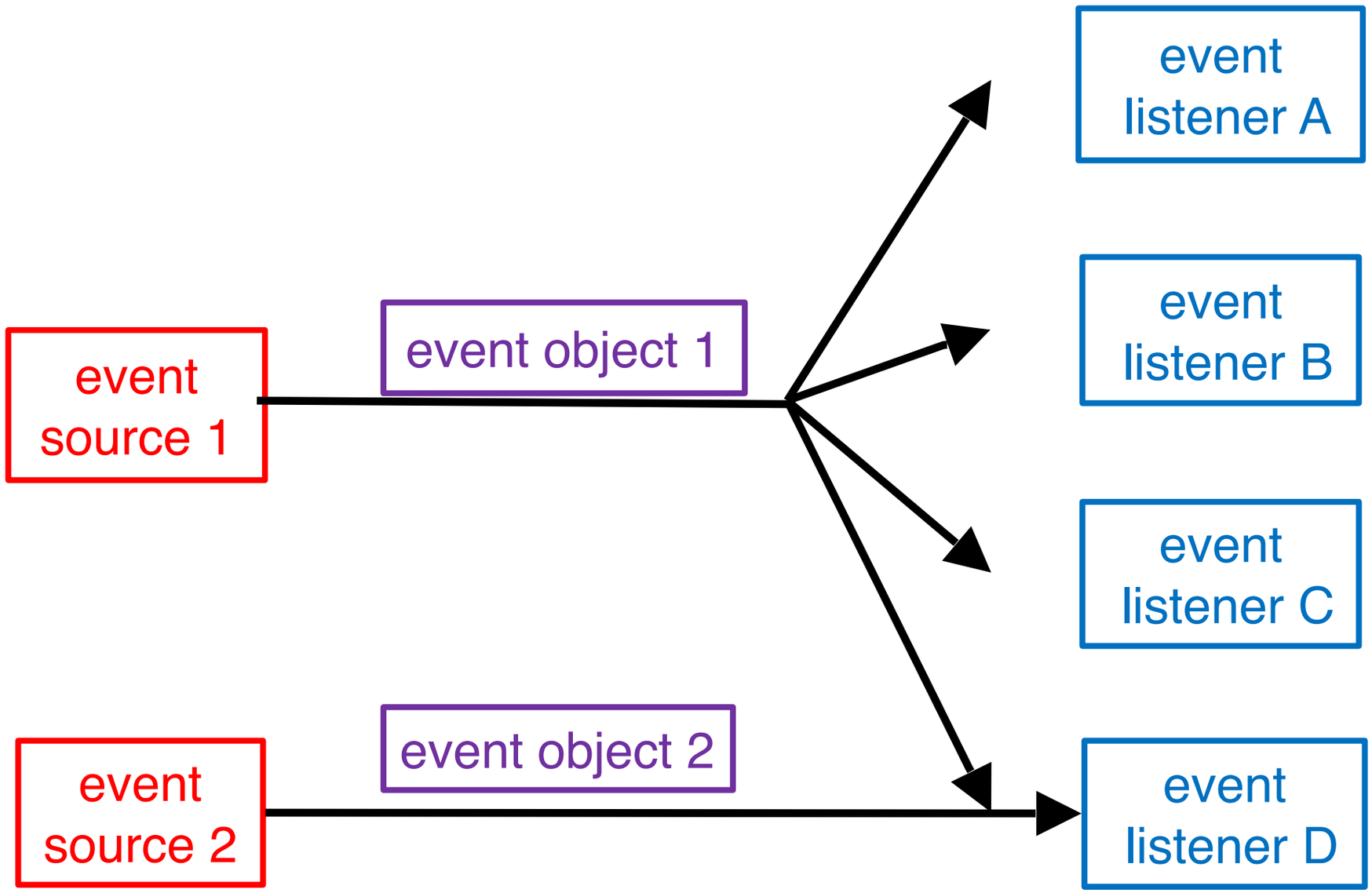
CalcView: File Chooser

```
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;
  private JButton sumButton;
```

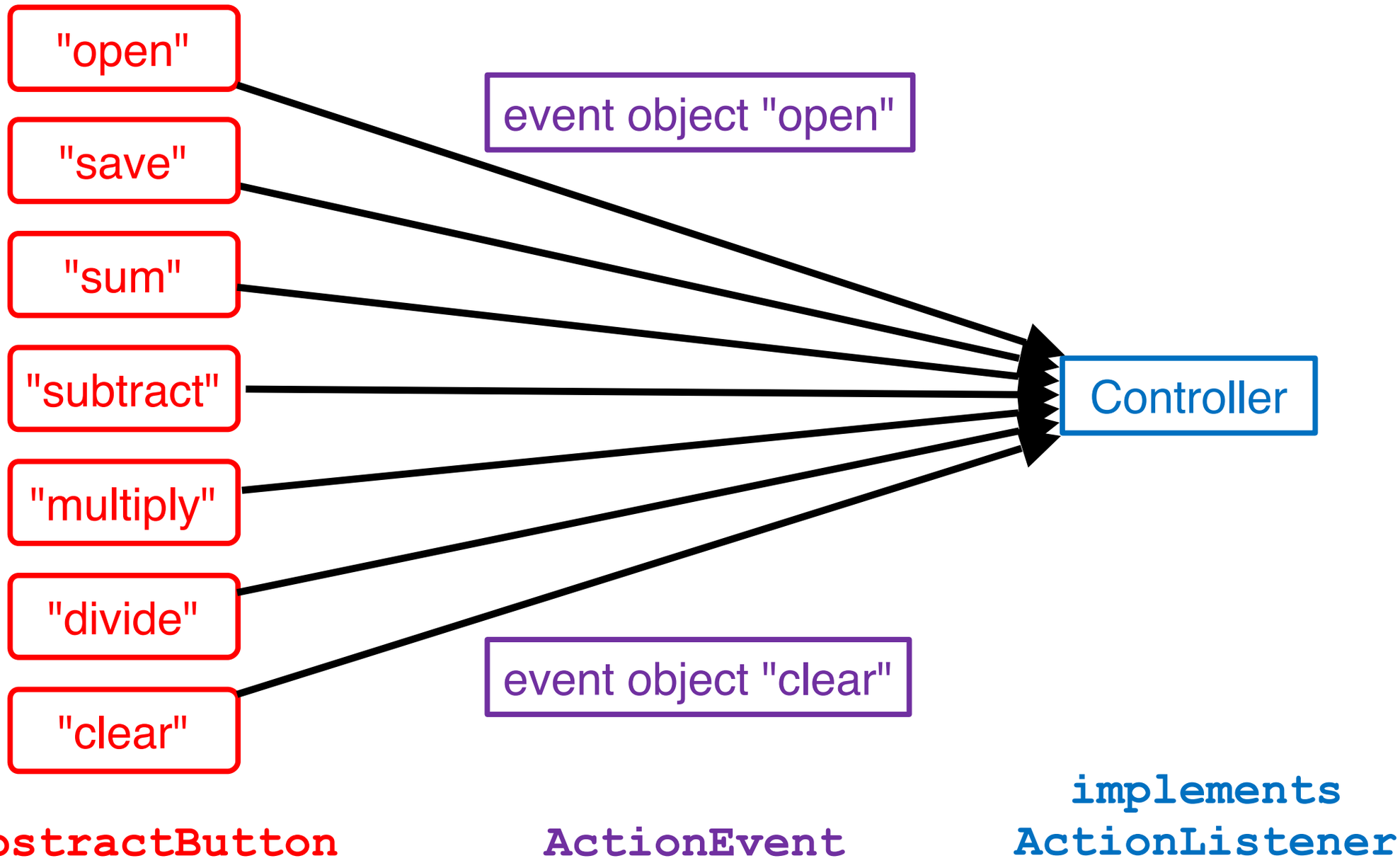
Event Driven Programming

- 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)
```

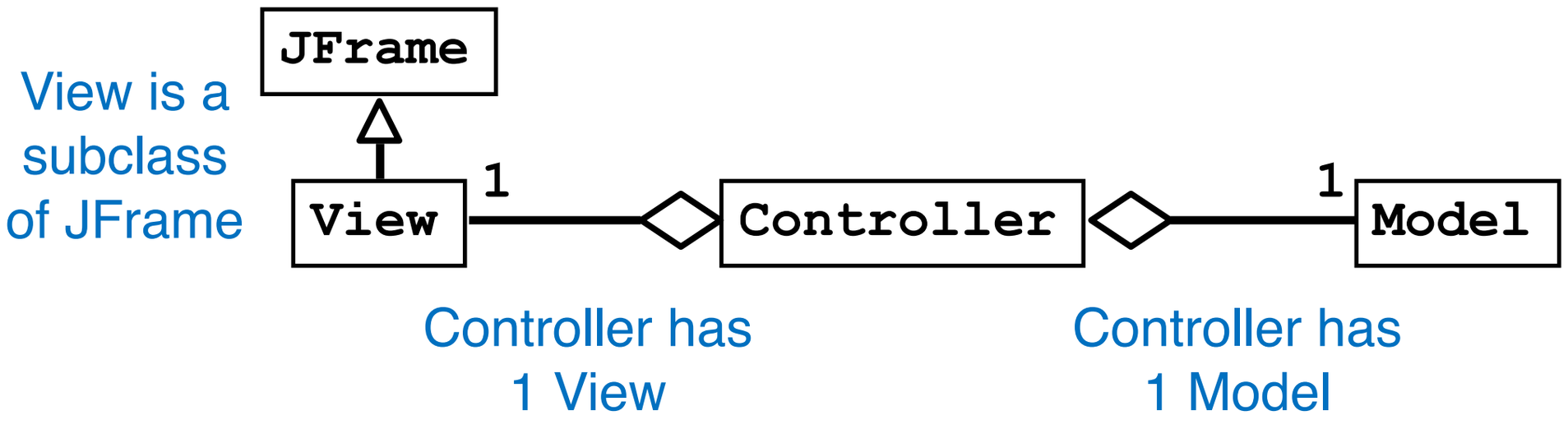
- for each **`JButton`** and **`JMenuItem`**
  1. call **`addActionListener`** with the controller as the argument
  2. call **`setActionCommand`** with a string describing what event has occurred

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;
  private JTextField userValueText;
```

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;
  }

}
```

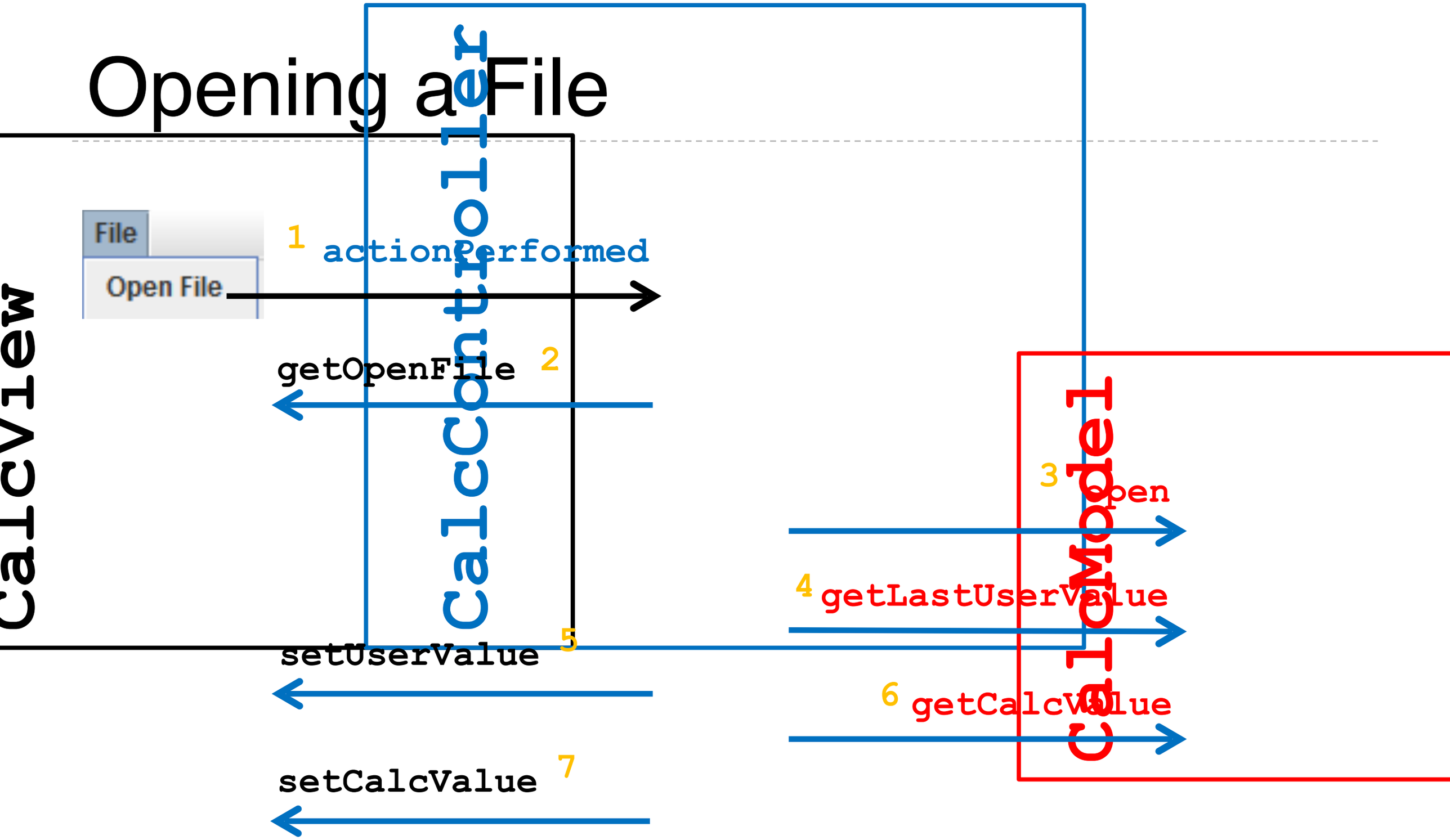
CalcController

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

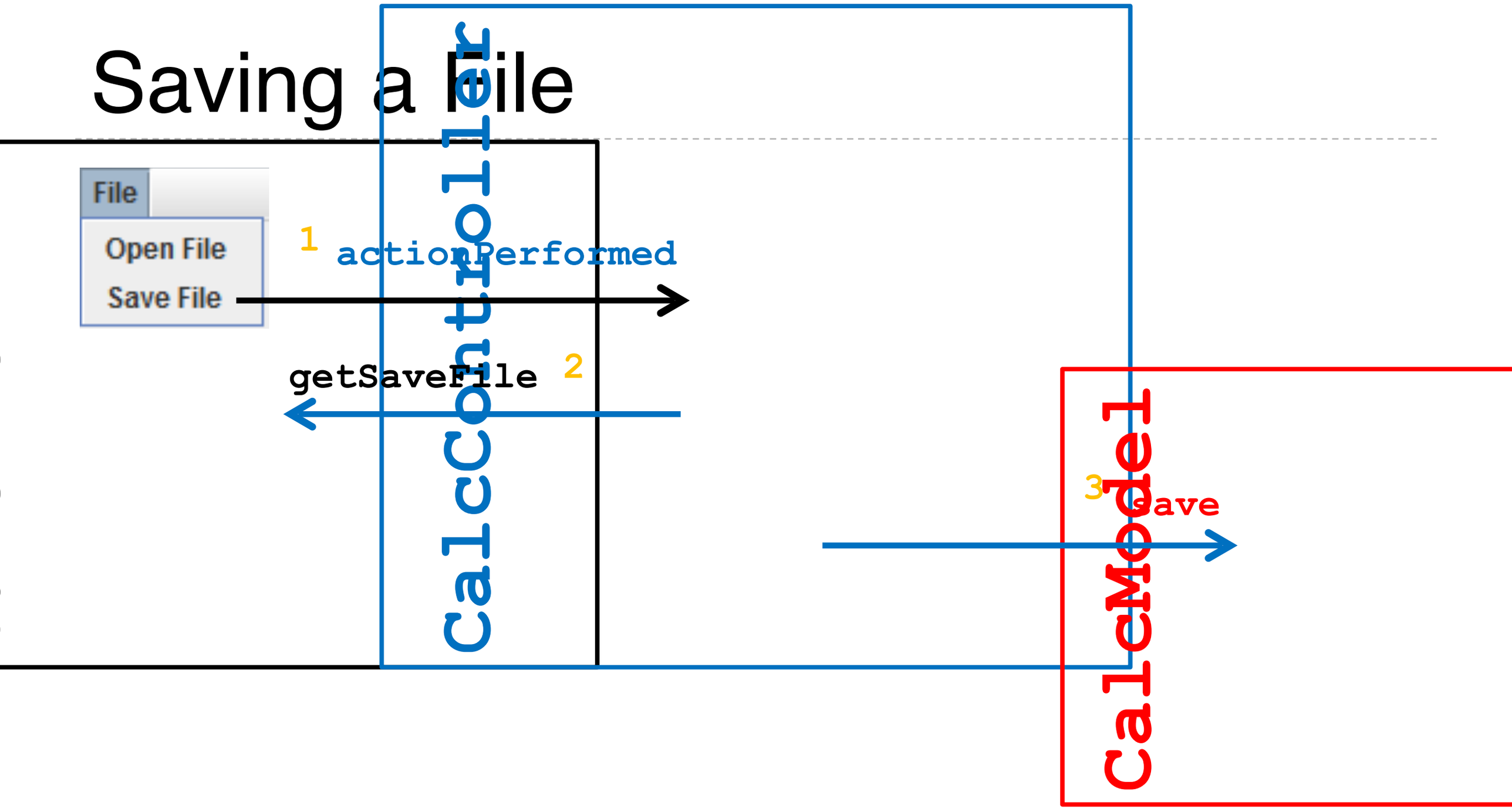
Opening a File



CalcController: Open a File

```
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.
   */
```

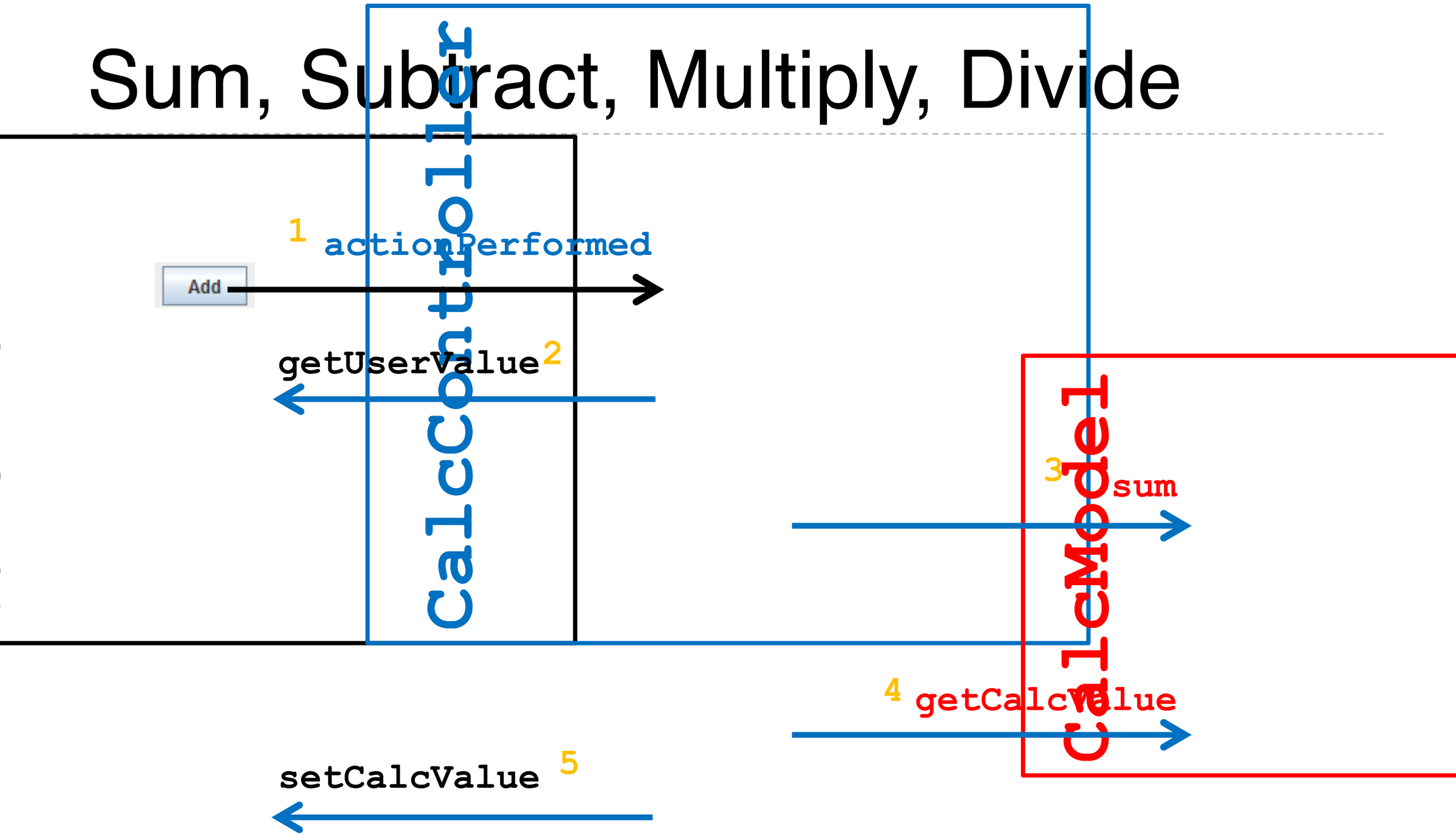
Saving a File



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

actionPerformed

- even with only 5 buttons and 2 menu items our **`CalcController 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

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
   }
}
```

- 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`

CalcController Inner Classes

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

CalcController2: ArithmeticListener

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

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

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

    this.view.addDivideListener(new DivideListener());
  }

  // methods...

  /**
   * Abstract base class for classes that listen
```

Why Use Inner Classes

- only CalcController needs to create instances of the various listeners
  - 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