Model-View-Controller

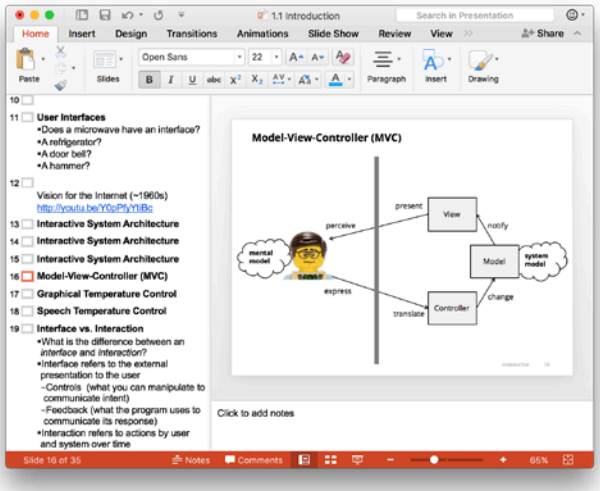
rationale

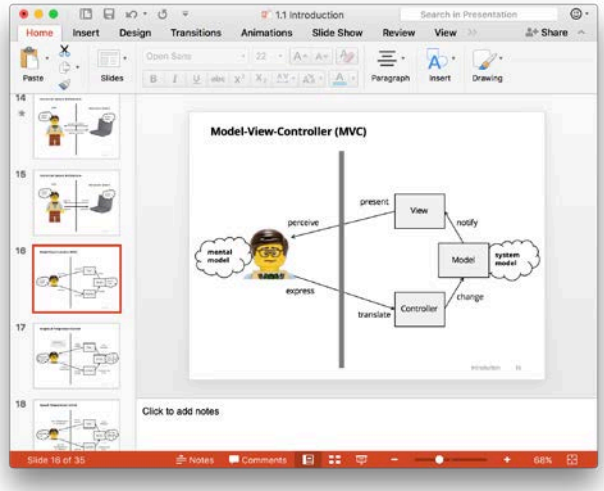
implementation

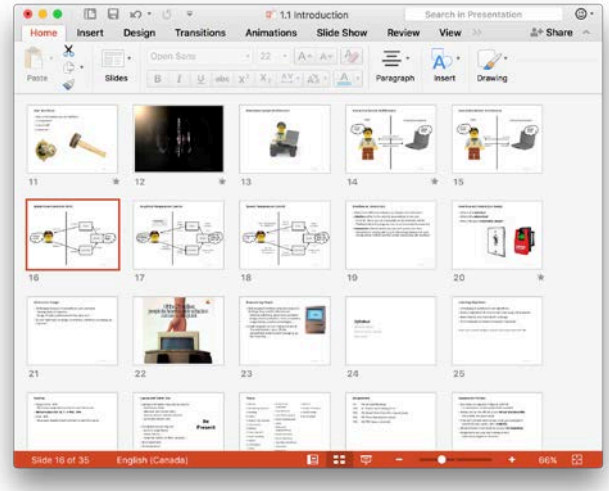
abstract model widgets

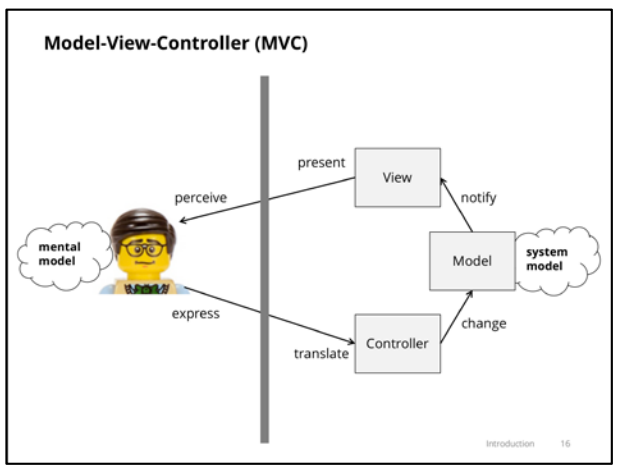
Multiple Views

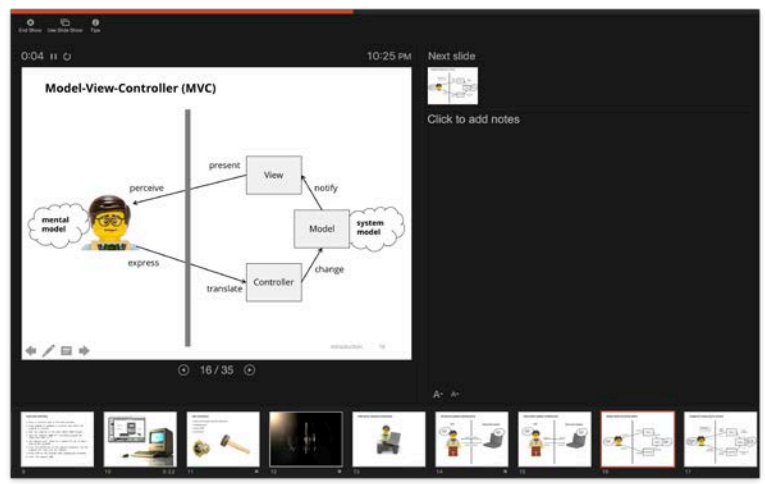
- Many applications have multiples views of one “document”









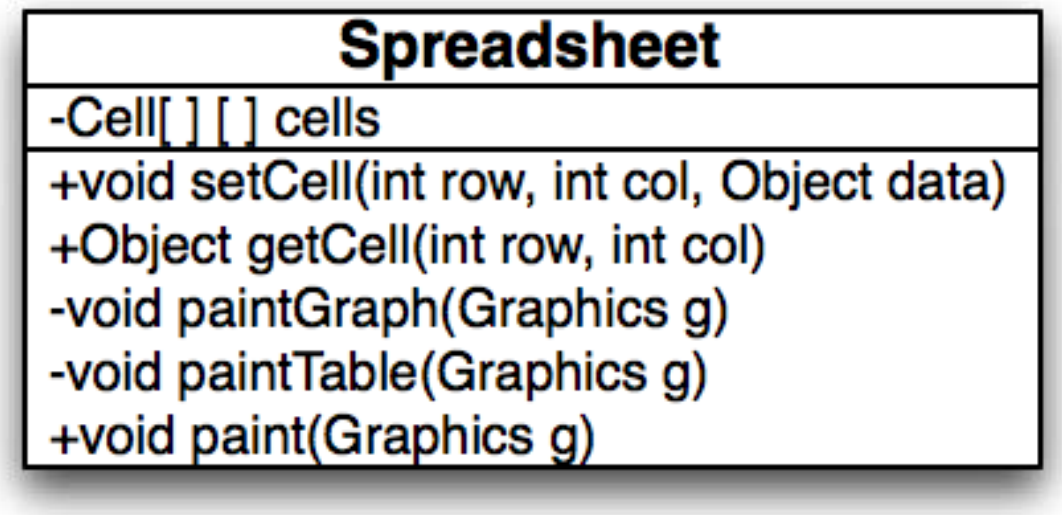


Observations

- When one view changes, the other(s) should change as well.
- UI code often changes more than the underlying application
  - (e.g. majority of recent updates to Office are in the UI)
- How do we design software to support these observations?

Possible Design

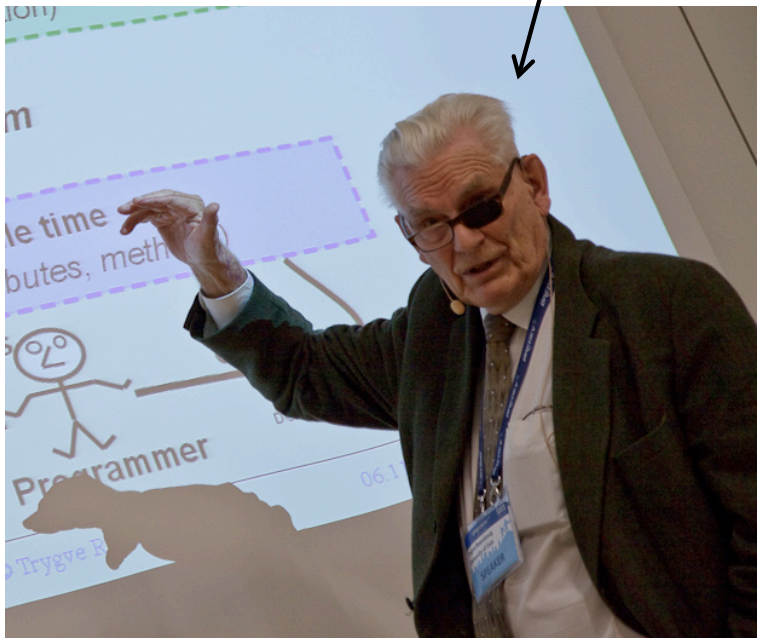
- Issues with bundling everything together:
  - What if we want to display data from a different type of source (eg: a database)?
  - What if we want to add new ways to view the data?
- Primary problem: Data and presentation are tightly coupled



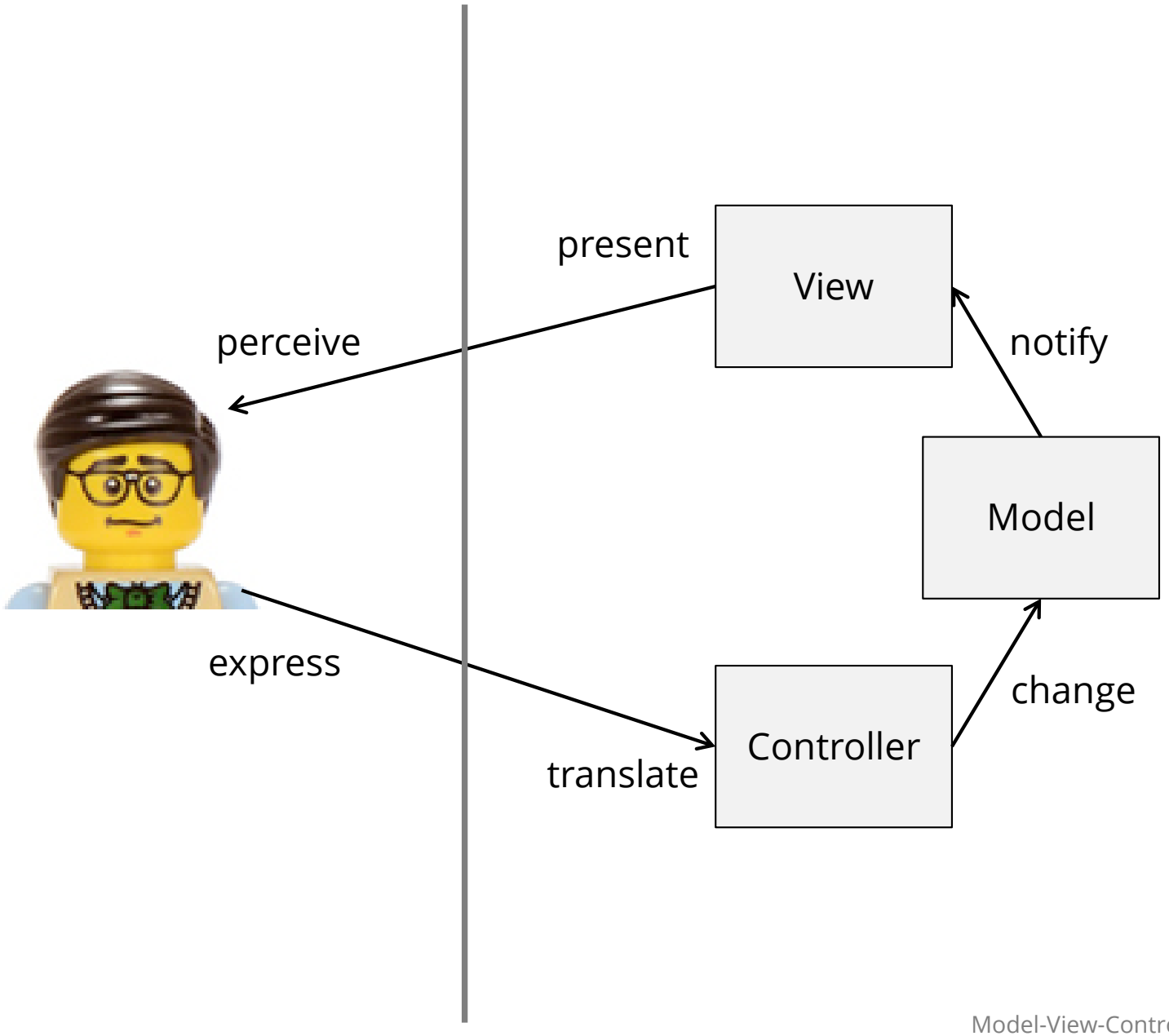
Model-View-Controller (MVC)

- MVC developed at Xerox PARC in 1979 by Trygve Reenskaug
  - for Smalltalk-80 language, the precursor to Java
- Now standard design pattern for GUIs
- Used at many levels
  - Overall application design
  - Individual components
- Many variations of MVC idea:
  - Model-View-Presenter
  - Model-View-Adapter
  - Hierarchical Model-View-Controller
- We use “standard” MVC in this course

Trygve Reenskaug

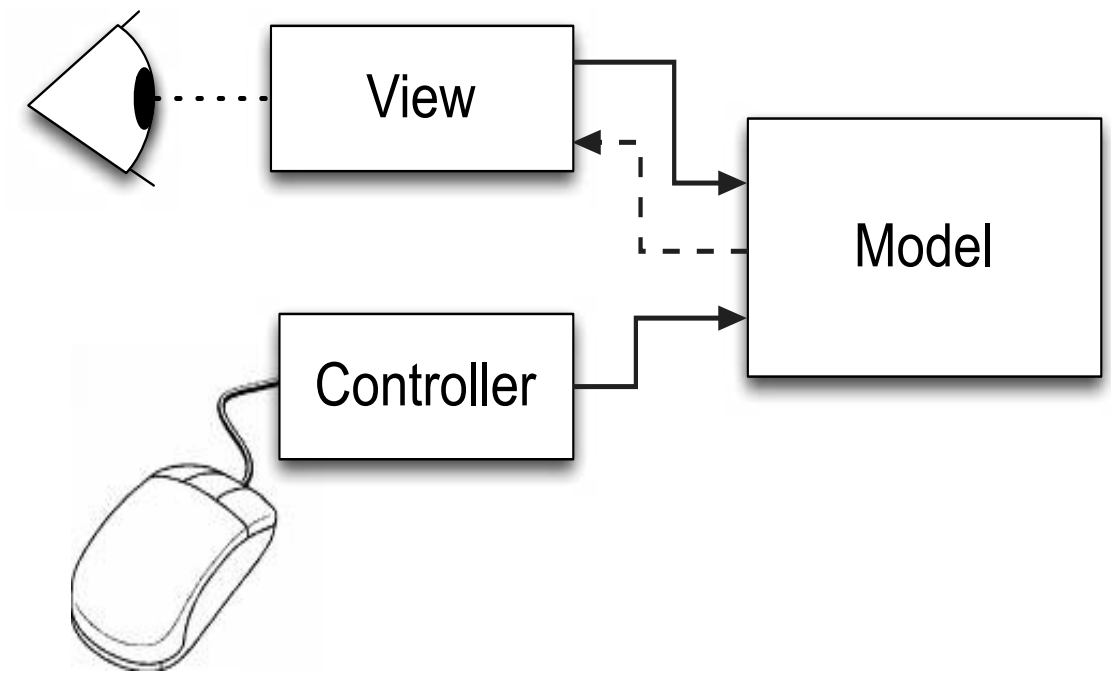


Model-View-Controller (MVC)



Model-View-Controller (MVC)

- Interface architecture decomposed into three parts:
  - **Model**: manages data and its manipulation
  - **View**: manages presentation of the data
  - **Controller**: manages user interaction

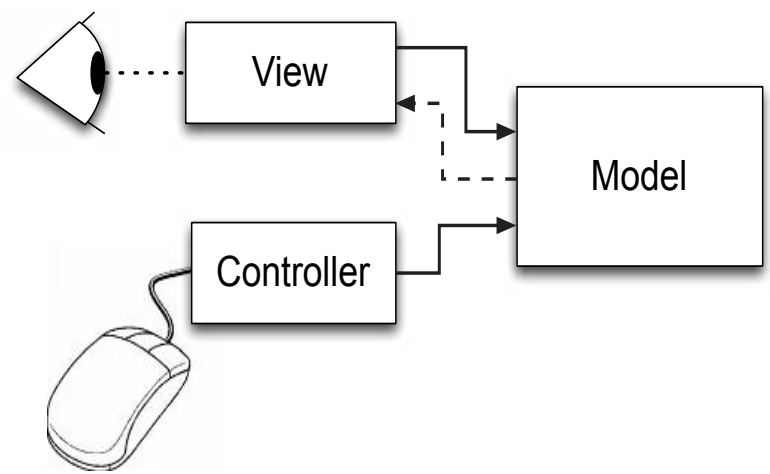


MVC Classes

- 3 classes: Model, View, Controller
- Model only knows about View interface (below)
- View and Controller know all about model
- In practice, View and Controller are often coupled …
  - View knows to send events to Controller
  - Controller knows about View …

View Interface

```
interface IView {
    public void updateView();
}
```



View Class Outline

```
class View implements IView {
   private Model model; // the model this view presents

   View(Model model, Controller controller) {

      ... create the view UI using widgets ...

      this.model = model; // set the model
      // setup the event to go to the controller
      widget1.addListener(controller);
      widget2.addListener(controller);
   }

   public void updateView() {
      // update view widgets using values from the model
      widget1.setProperty(model.getValue1());
      widget2.setProperty(model.getValue2());
      ...
   }
}
```

Controller Class Outline

```
class Controller implements Listener {
   Model model; // the model this controller changes

   Controller(Model model) {
      this.model = model; // set the model
   }

   // events from the view's widgets
   // (often separated to 1 method per widget)
   public void action1Performed(Event e){
      // note the controller does need to know about view
      if (widget1 sent event)
         model.setValue1();
      else if (widget2 sent event)
         model.setValue2();
      ...
   }
}
```

Model Class Outline

```
class Model {
   List<IView> views; // multiple views
   public void addView(IView view) {...} // add view observer

   // get model values
   public type getModelValue1() { return value1; }
   public type getModelValue2() { return value2; }
   ... more value getters ...

   // set model values
   public void setModelValue1(type value) {
      value1 = value; notifyObservers();
   }

   ... more setters, each calls notifyObservers() ...

   // notify all IView observers
   private void notifyObservers() {
      for (IView view: views) view.updateView();
   }
}
```

HelloMVC1.java

1 view

HelloMVC2.java

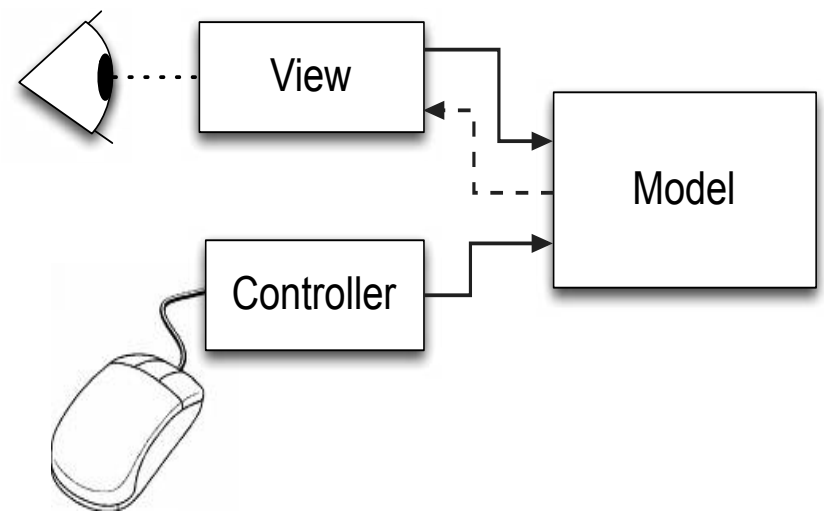
2 (or more) views

inspired by Joseph Mack: http://www.austintek.com/mvc/

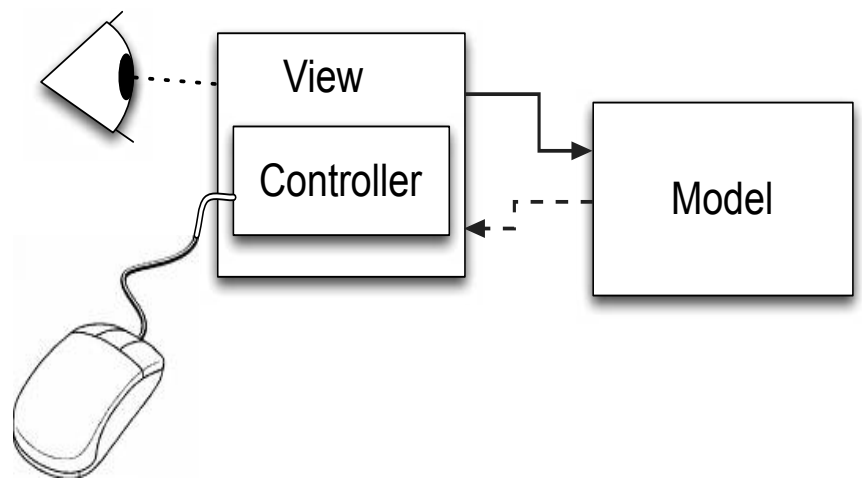
– (also a good MVC explanation, shows how to use Java Observer class)

Theory and Practice

- **MVC in Theory**
  - View and Controller both refer to Model directly
  - Model uses the observer design pattern to inform view of changes



- **MVC in Practice**
  - Model is very loosely coupled with UI using the observer pattern
  - The View and Controller are tightly coupled. <u>Why?</u>

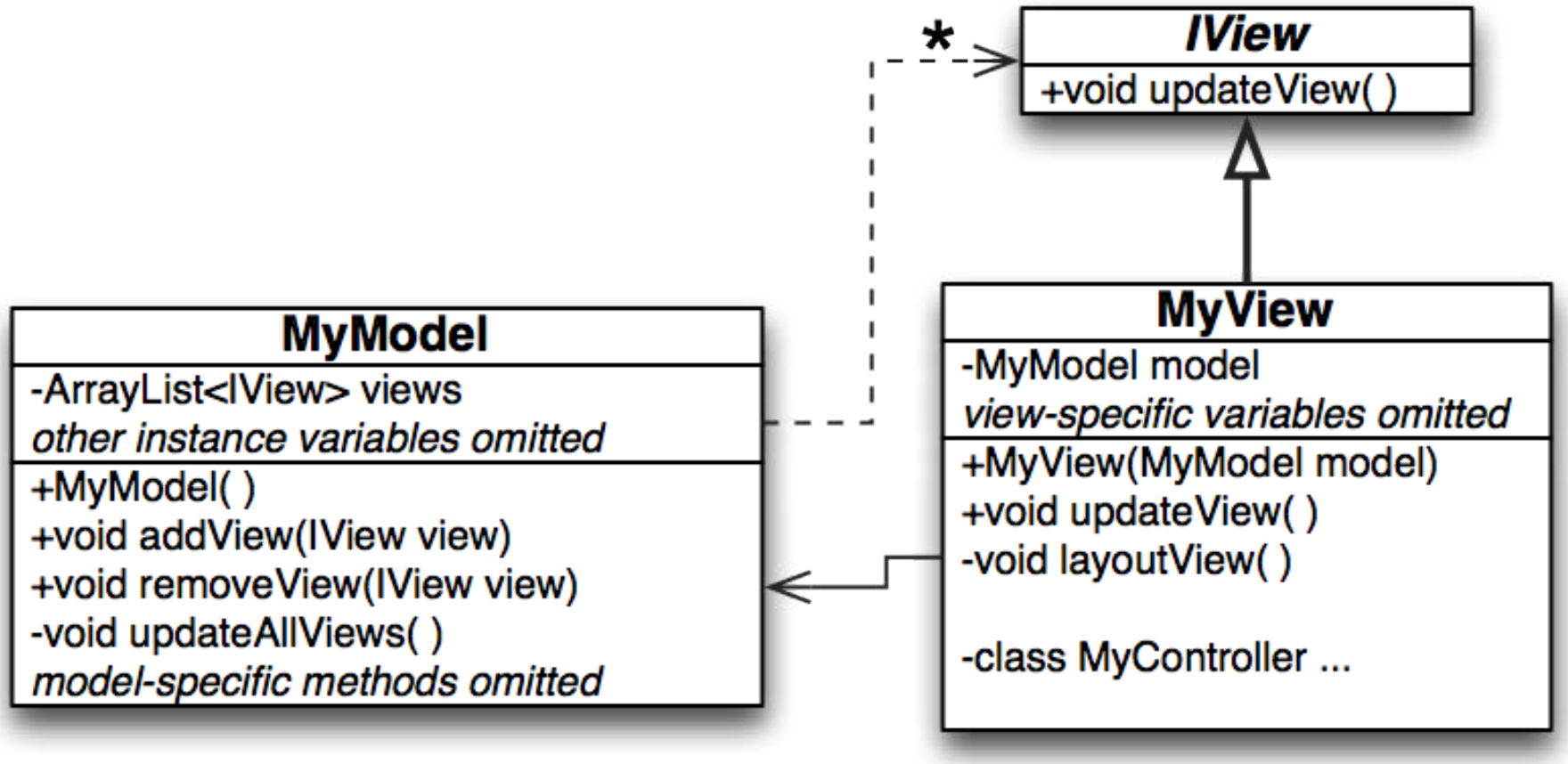


HelloMVC3.java

Controller code in View

MVC as UML

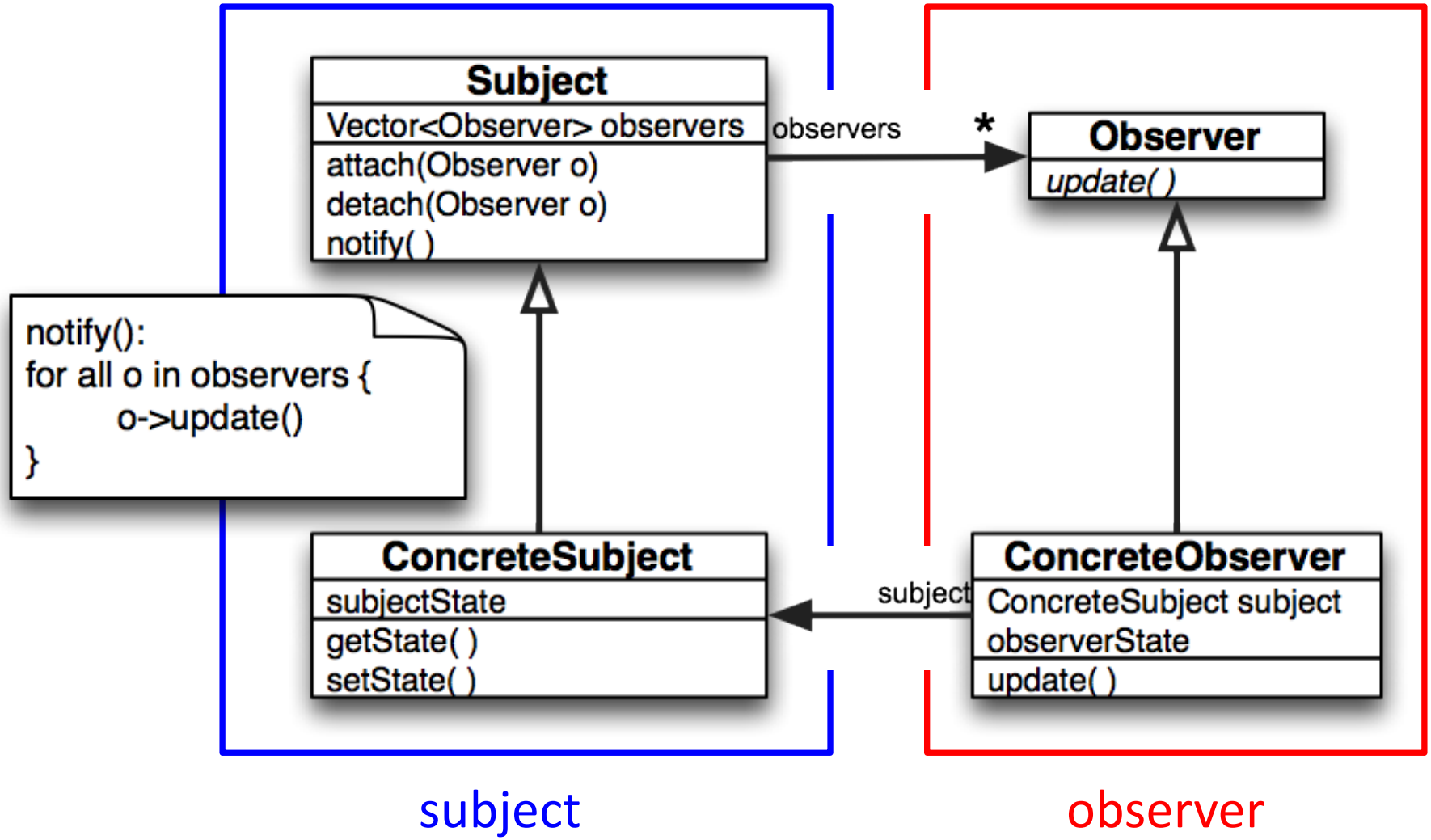
- MyView does not need to implement IView.
  - It could provide an anonymous inner class to MyModel instead.



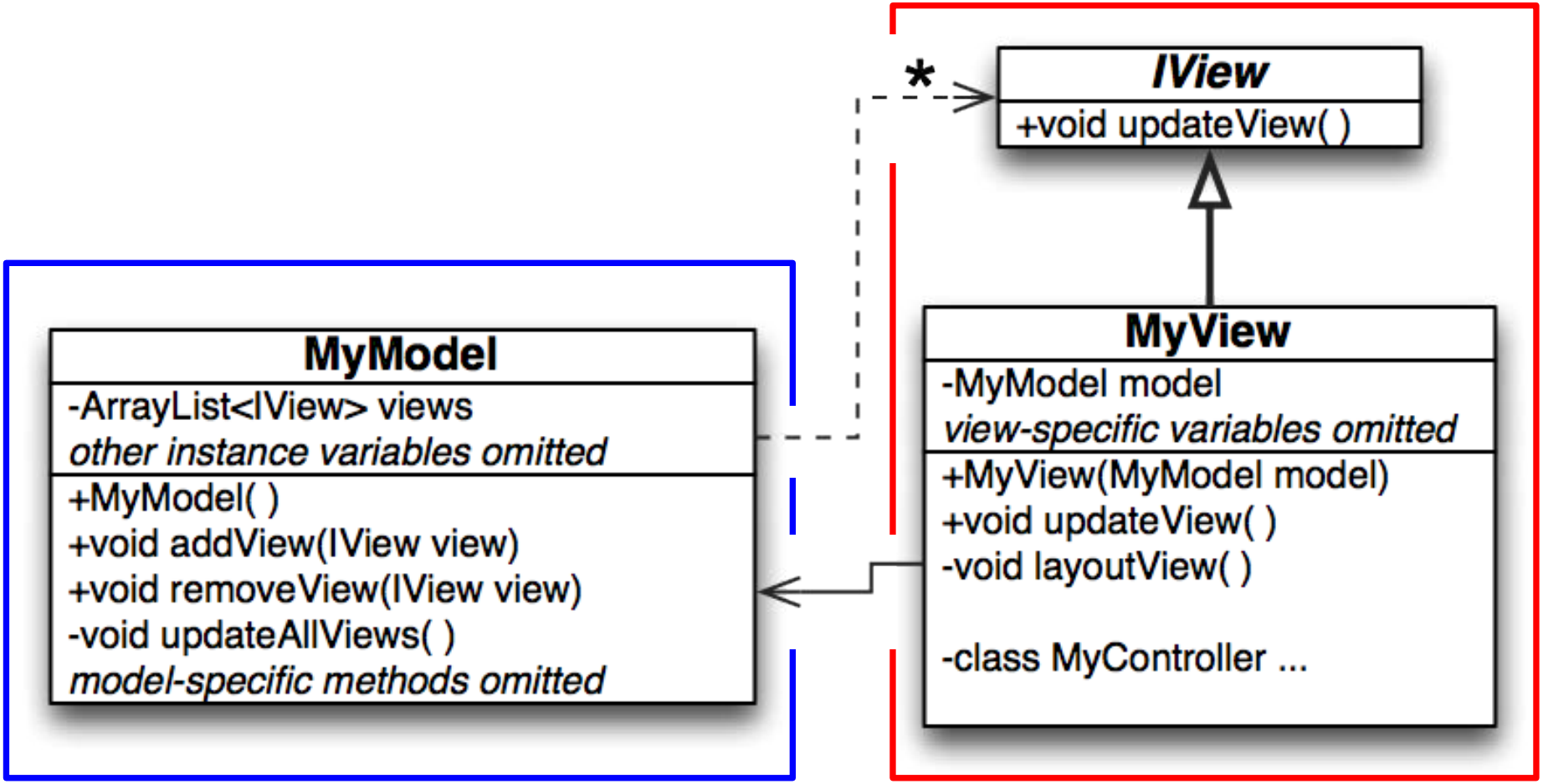
Observer Design Pattern

- MVC is an instance of the Observer design pattern
- Provides a well-defined mechanism that allows objects to communicate without knowing each others' specific types
  - Promotes loose coupling
- related to:
  - "publish-subscribe" pattern
  - "listeners"
  - delegates in C#

Observer Design Pattern



MVC as Observer Pattern



subject

observer

HelloMVC4 Code Demo

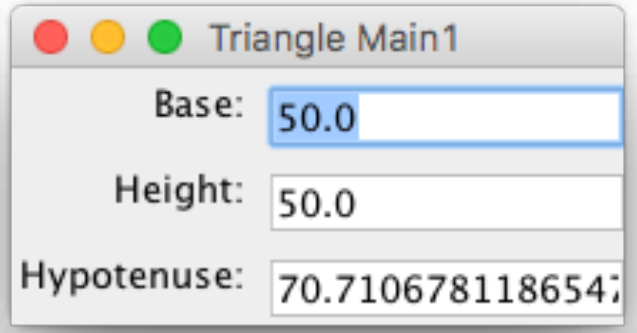
- java.util provides Observer interface and Observable class
  - Observer is like IView, i.e. the View implements Observer
  - Observable is the “Subject” being observed
    i.e. the Model extends Observable
  - base class has list of Observers and method to notify them

Triangle MVC Code Demos

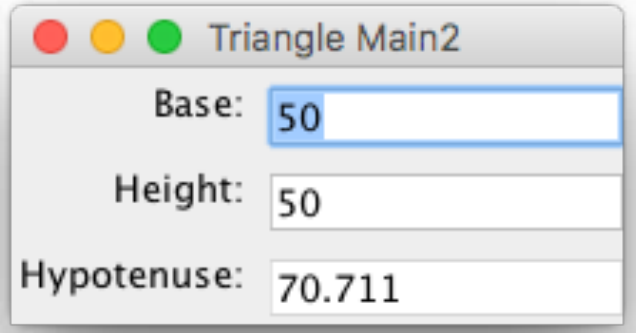
- Program requirements:
  - vary base and height of right triangle, display hypotenuse
- TriangleModel
  - stores base and height, calculates hypotenuse
  - constrains base and height values to acceptable range

Triangle: Main1.java and Main2.java

`SimpleTextView`

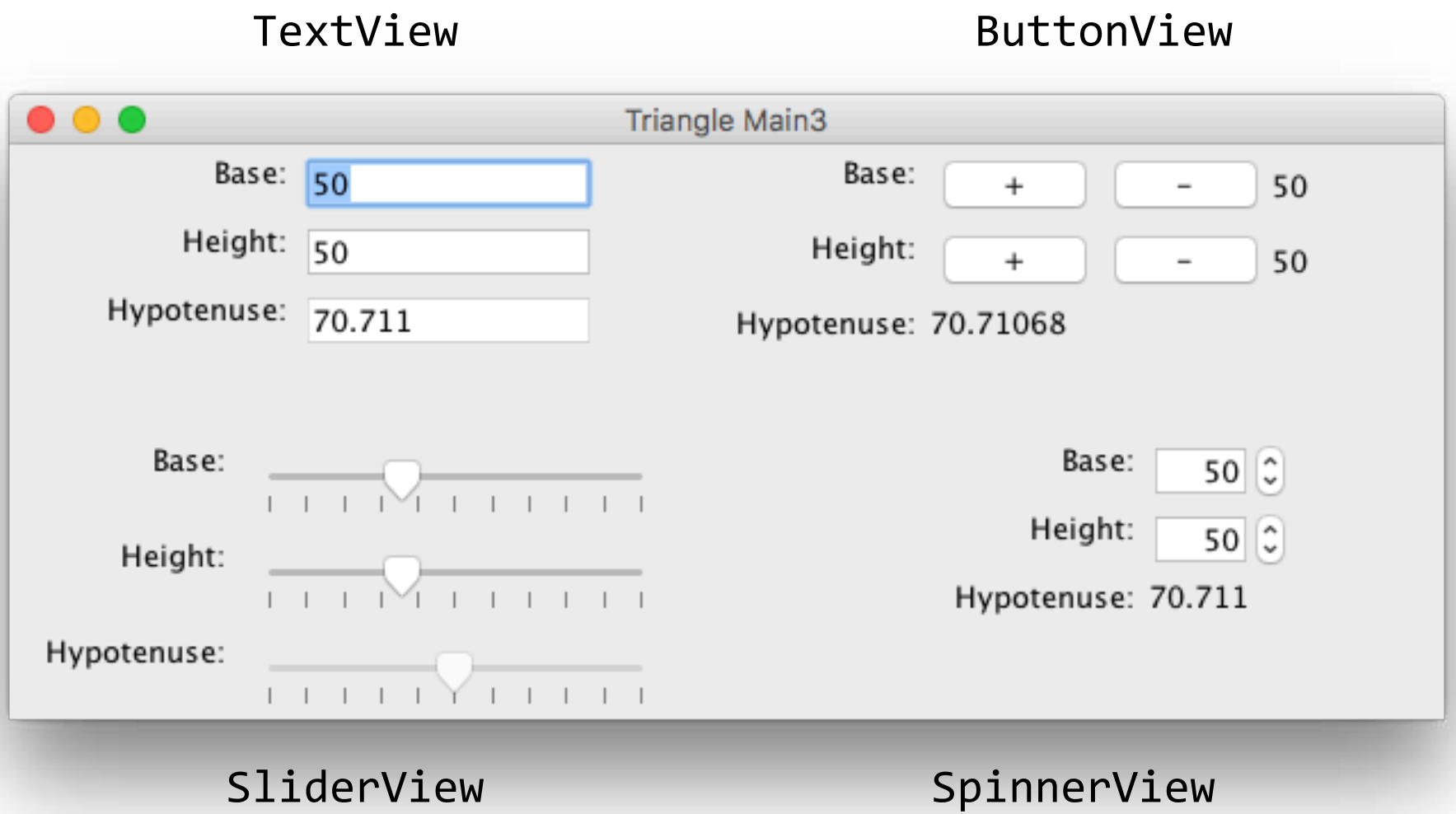


`TextView`



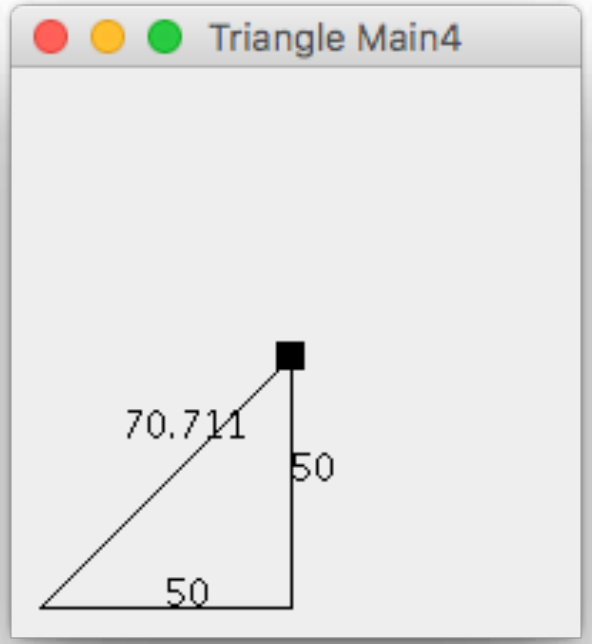
Triangle: Main3.java

Combines Multiple Views using `GridLayout`



Triangle: Main4.java

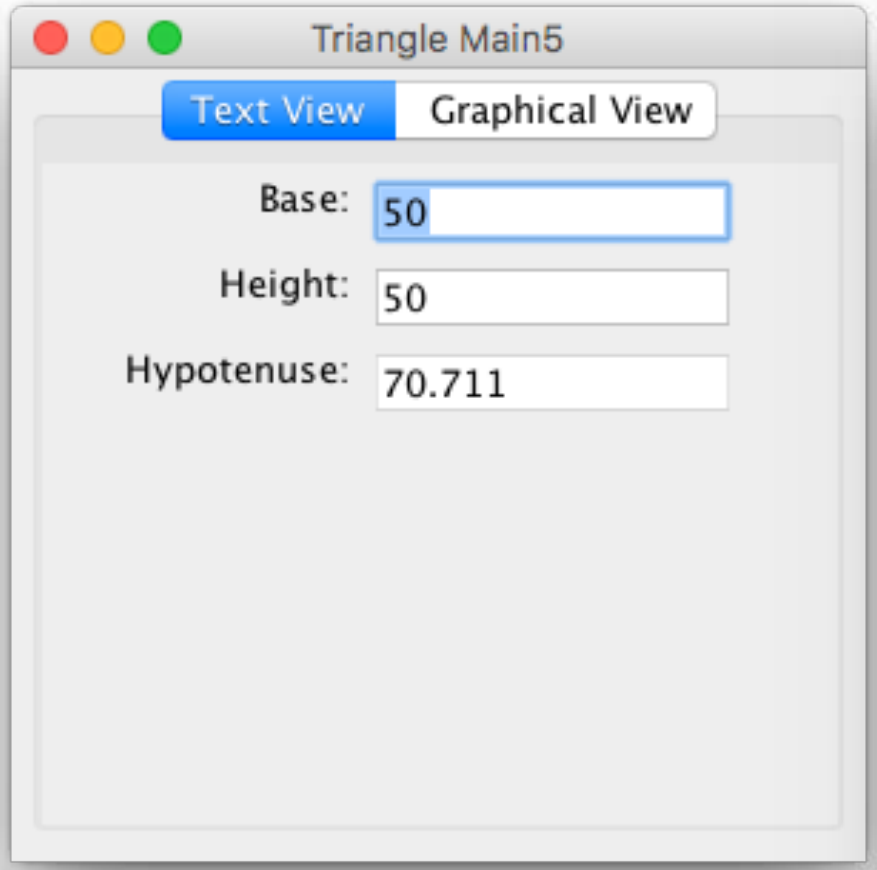
```
GraphicalView
```



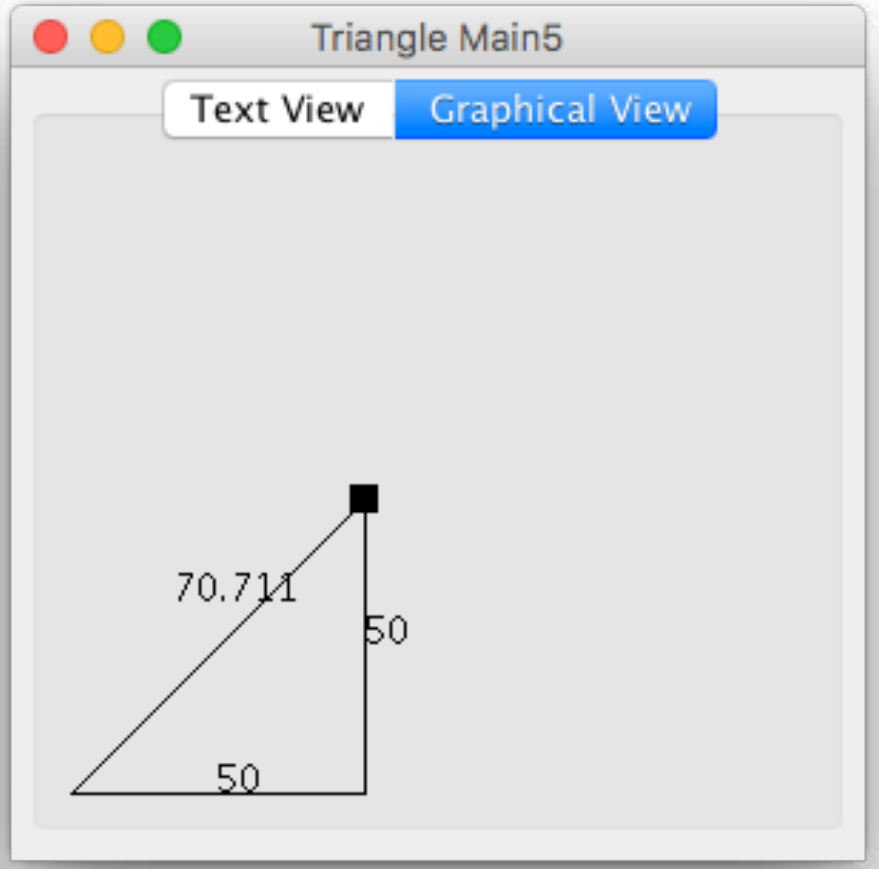
Triangle: Main5.java

Combines Multiple Views using Tab Panel

`TextView`



`GraphicalView`



MVC Implementation Process

Setup the basic code infrastructure

- the Model class
- one or more View/Controller classes (extends JComponent or JPanel)
- a class containing the main method and application JFrame

- In the main method:
  - create an instance of the model
  - create instances of the Views/Controllers, add add to them the model
  - display the View(s) in a frame

MVC Implementation Process (cont.)

Build and test the Model

- Design, implement, and test the model
  - add commands used by Controllers to *change* the model
  - add queries used by View to *update* their state
- Call `updateAllViews` just before exiting all public methods that change data

Build the Views and Controllers

- Design the UI as one or more Views. For each View:
  - Construct widgets
  - Lay the widgets out in the view
  - Write and register appropriate controllers for each widget
  - Write `updateView` to get and display info from the model
  - Register `updateView` with the model

MVC Benefit: Change the UI

- View separation enables alternative interfaces
- Controller separation enables alternative input methods
- Data and application logic in Model does not have to change
- Examples:
  - porting to new OS platforms
  - porting to different hardware platforms
  - adapting to new UI toolkits
  - taking advantage of new widgets
  - …

MVC Benefit: Multiple Views

- View separation enables multiple, simultaneous views of data
- A separate Model means views can independently use same data
  - Each view is unencumbered by the details of the other views
  - Reduces dependencies on the GUI that could change
- Examples:
  - viewing numeric data as a table, a line graph, a pie chart, …
  - displaying simultaneous “overview” and “detail” views
  - enabling “edit” and “preview” views
  - …

MVC Benefit: Code Reuse

- Separation enables programmers to more easily use same stock set of widgets to manipulate unique application data.
- Examples:
  - JTable has a “pluggable” Model to manipulate many kinds of data
  - A View-based graph widget can be re-used with different Models
  - A mouse-gesture Controller can be re-used with other Views
  - …

MVC Benefit: Testing

- Separation enables independent development of application logic and user interface elements
  - can test the Model without any UI
  - can test View/Controller without any Model
- Examples:
  - write JUnit tests using Model's API
  - use Java UI automation to test View/Controller APIs

Apple "MVC" Pattern

- Apple IOS and Cocoa emphasize the Controller as an intermediary link between the Model and View
- In my opinion, this is really a Model-View-Presenter (MVP) Pattern (though lots of debate about this ...)