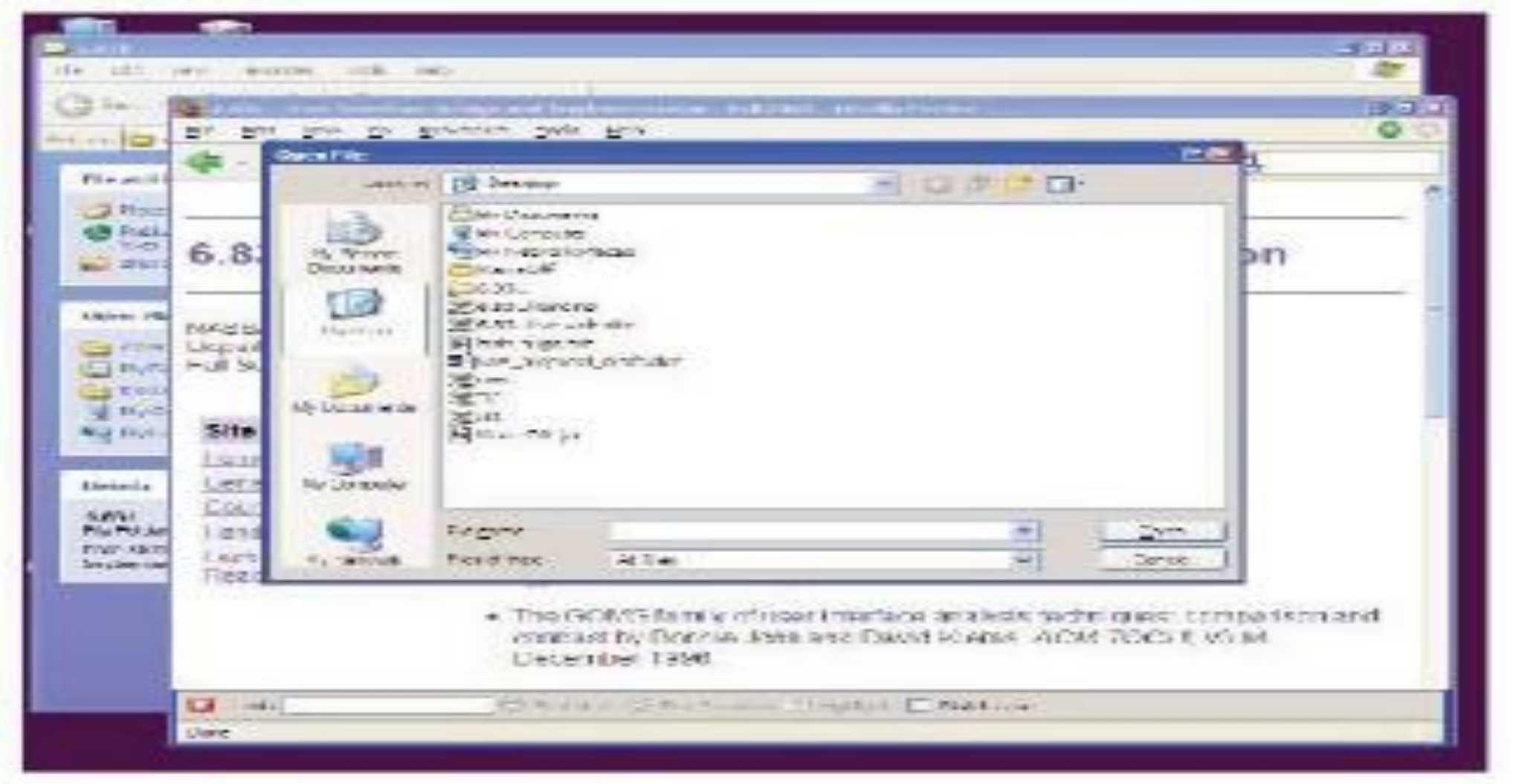
# HUMAN COMPUTER INTERACTION

Lecture 7: UI Software Architecture

# Hall of Fame or Shame?



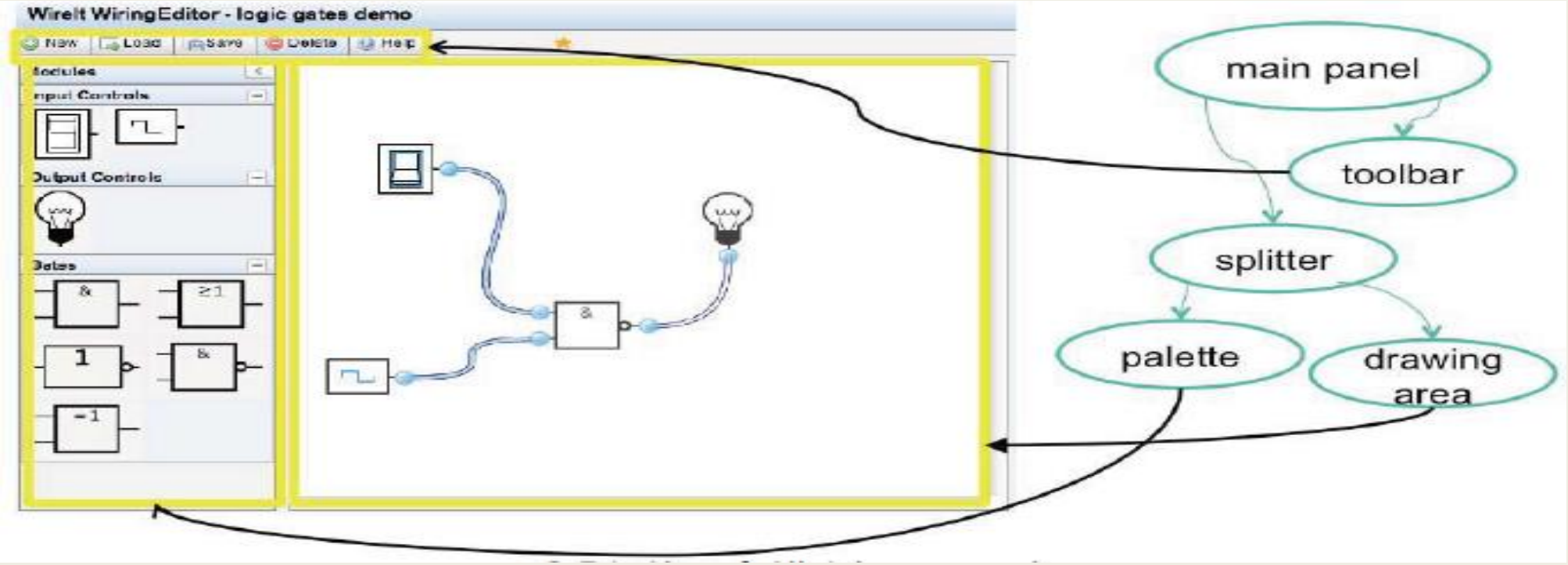
# Hall of Fame or Shame?

# Today's Topic

- Design patterns for GUI
  - *View tree*
  - *Listener*
  - *Widget*
  - *Model-view-controller*
- Approaches to GUI programming
  - *Procedural*
  - *Declarative*
  - *Direct manipulation*
- Web UI at lightning speed
  - *HTML*
  - *Javascript*
  - *JQuery*

# View Tree

- A GUI is structured as a tree of views
  - *A view is an object that displays itself on a region of the screen*

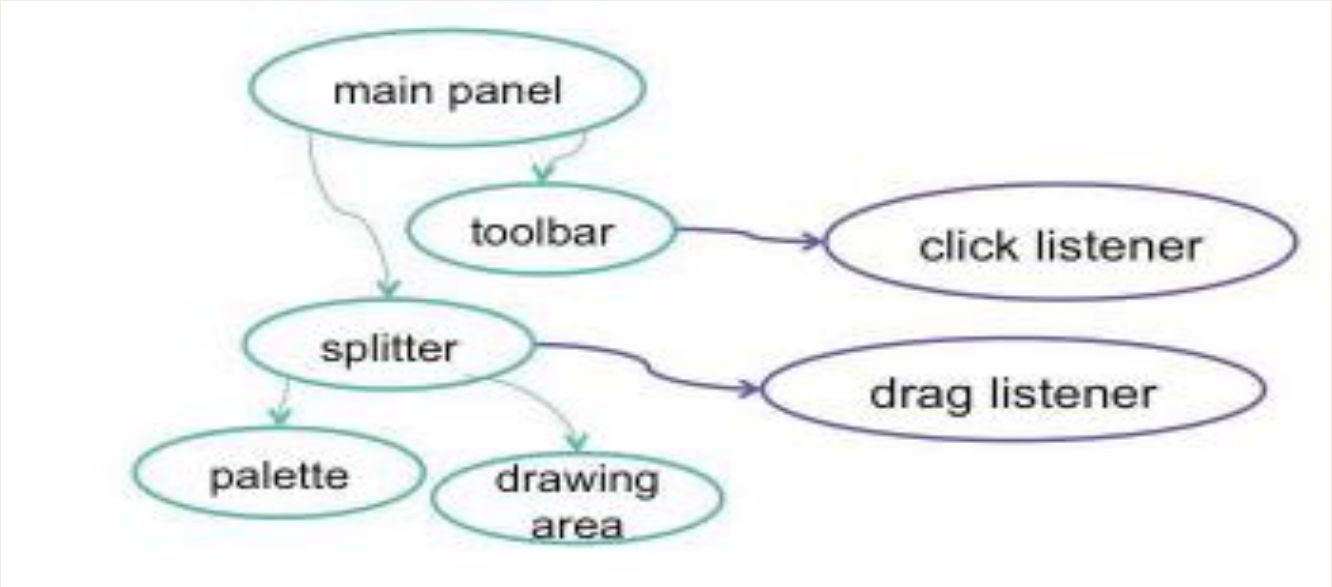


# How the View Tree Is Used

- Output
  - *GUIs change their output by mutating the view tree*
  - *A redraw algorithm automatically redraws the affected views*
- Input
  - *GUI receive keyboard and mouse input by attaching listeners to views (more on this in a bit)*
- Layout
  - *Automatic layout algorithm traverses the tree to calculate positions and sizes of views*

# Input Handling

- Input handlers are associated with views
  - *Also called listeners, event handlers, subscribers, observers*



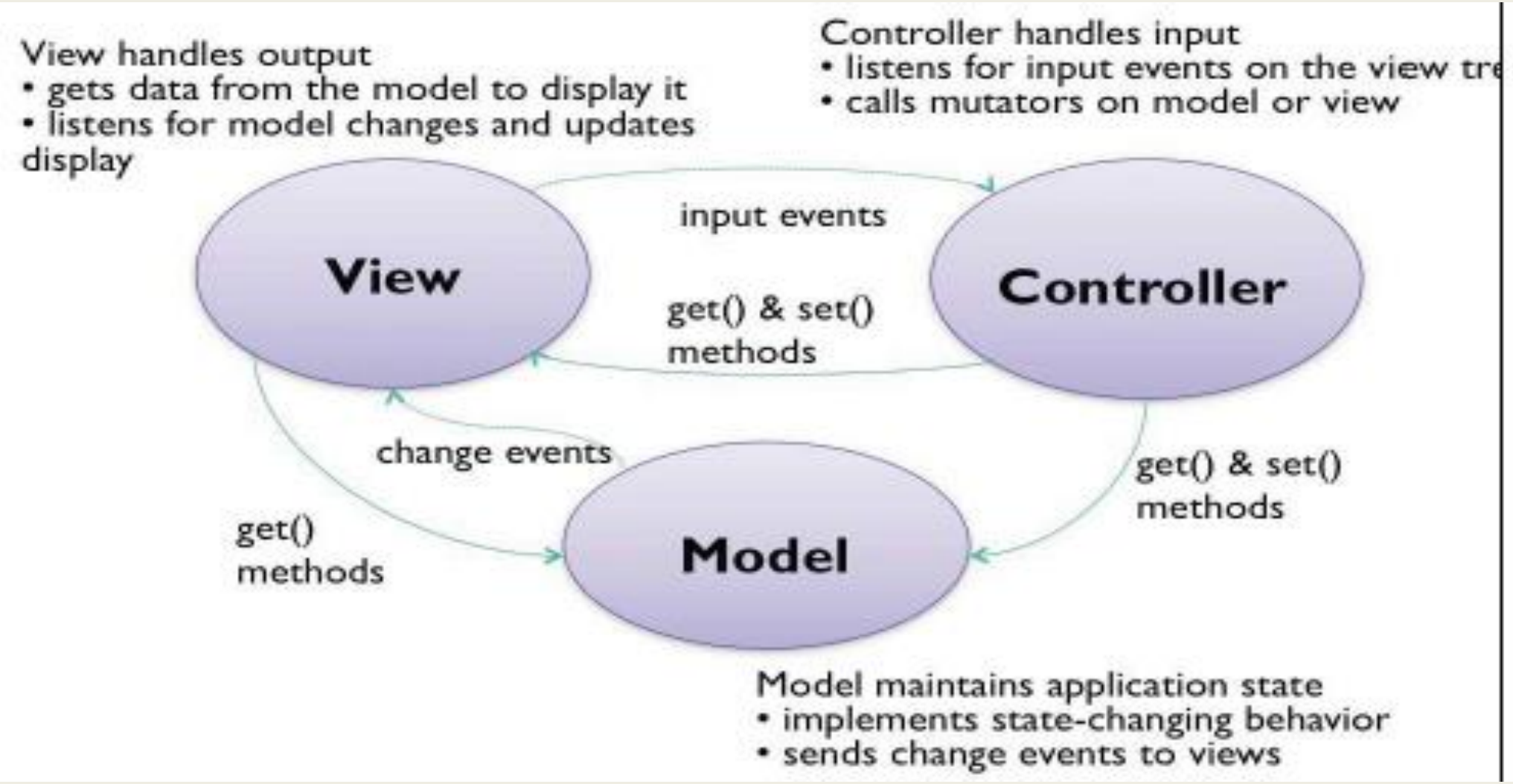
# Listener Pattern

- GUI input handling is an example of the Listener Pattern
  - *Aka Publish-Subscribe, Event, Observer*
- *An Event Source generates a stream of discrete events*
  - *e.g. mouse events*
- *Listeners register interest in the events from the source*
  - *Can Often register only for specific events e.g. only want mouse events occurring inside a view's bounds*
  - *Listeners can unsubscribe when they no longer want events*
- *When an event occurs, the event source distributes it to all interested listeners*

# Separating frontend from backend

- We’ve seen how to separate input and output in GUIs
  - *Output is represented by the view tree*
  - *Input is handled by listeners attached to views*
- Missing piece is the backend of the system
  - *Backend (aka model) represents the actual data that the user interface is showing and editing*
  - *Why do we want to separate this from the user interface?*

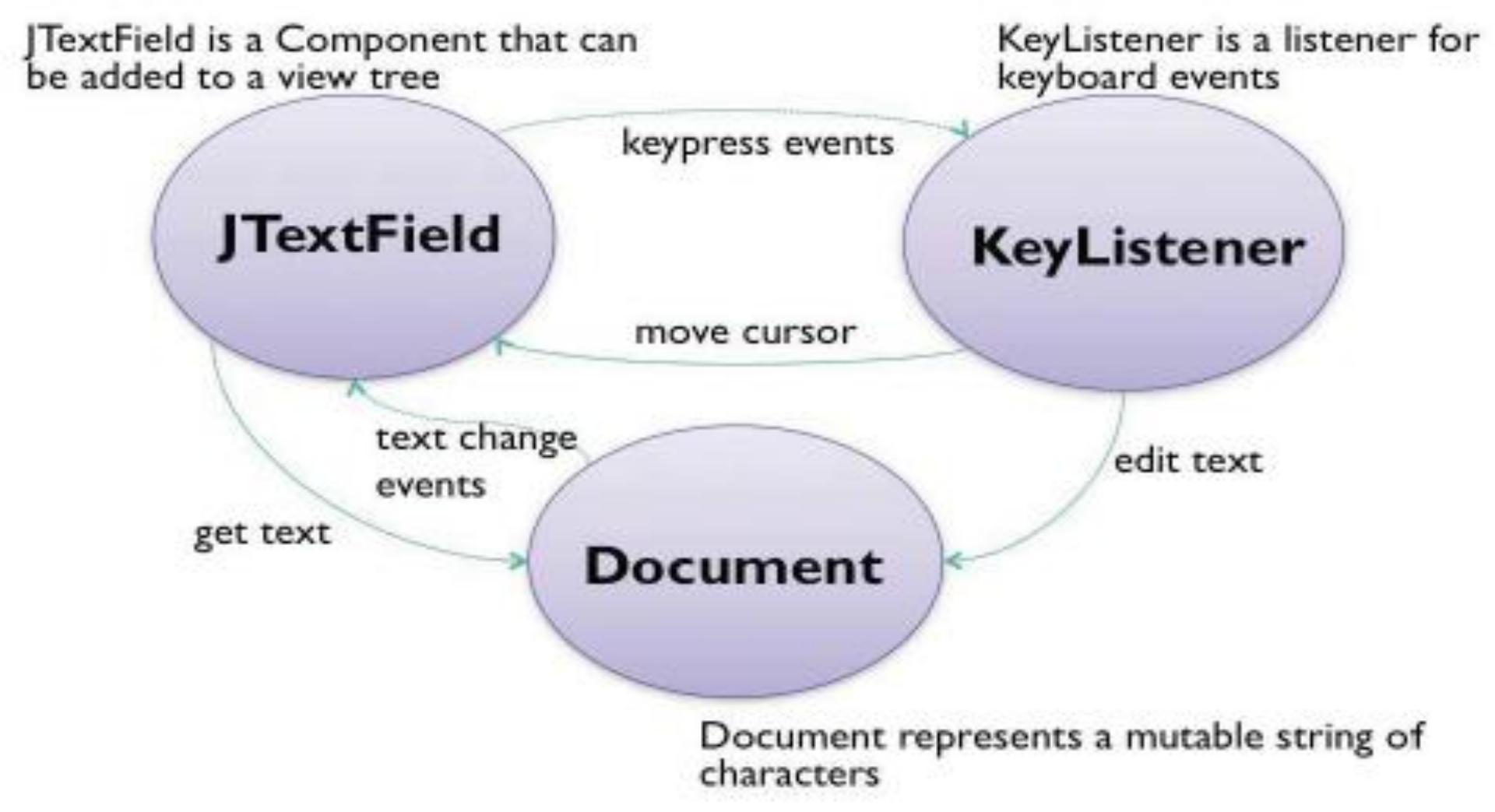
# Model-View-Controller Pattern



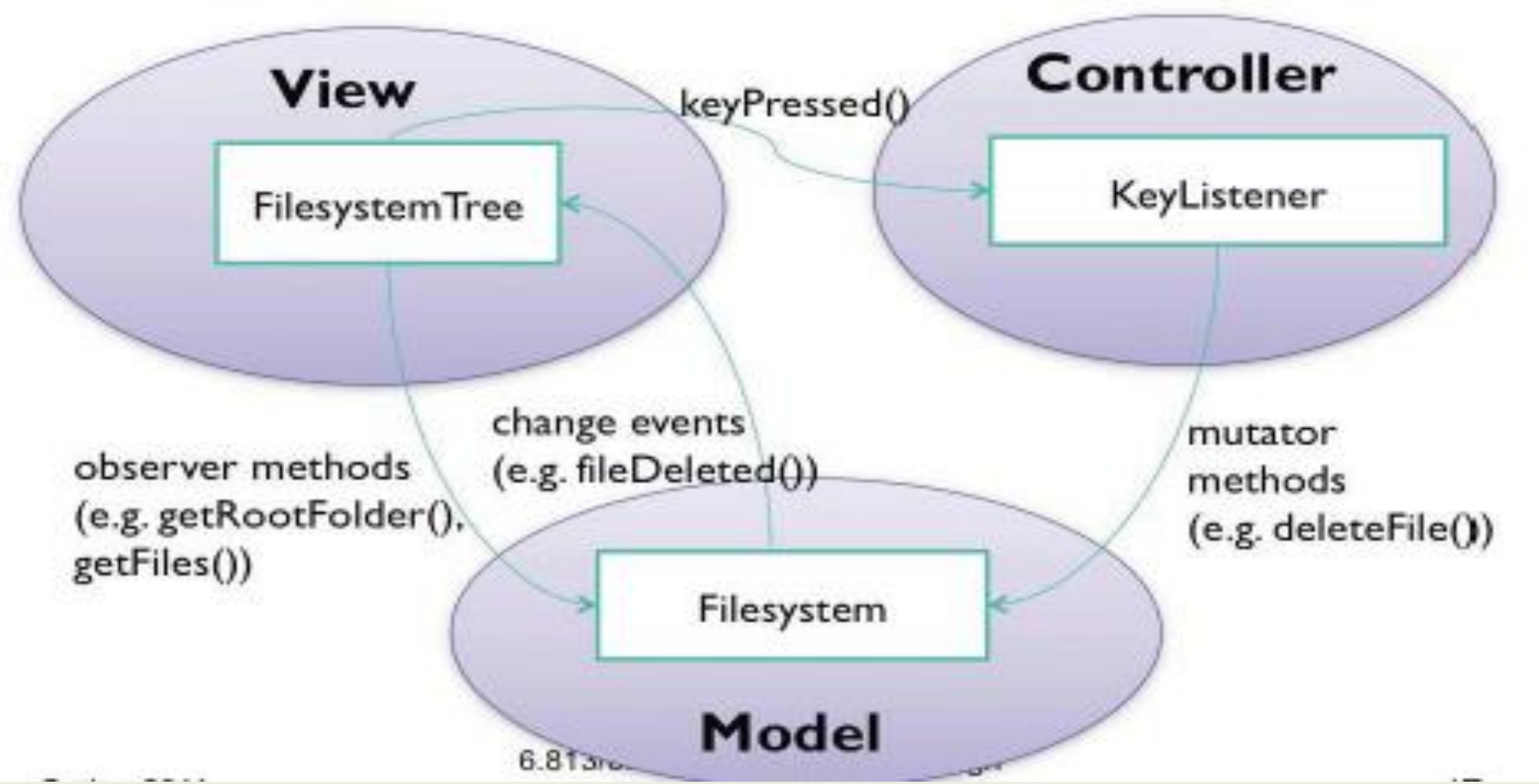
# Advantages of MVC

- Separation of responsibilities
  - *Each module is responsible for just one feature*
    - Model: Data
    - View: Output
    - Controller: Input
- Decoupling
  - *View and model are decoupled from each other, so they can be changes independently*
  - *Model can be reused with other views*
  - *Multiple views can simultaneously share the same model*
  - *Views can be reused for other models as long as the model implements an interface*

# A small MVC Example: Textbox



# A Large MVC Example



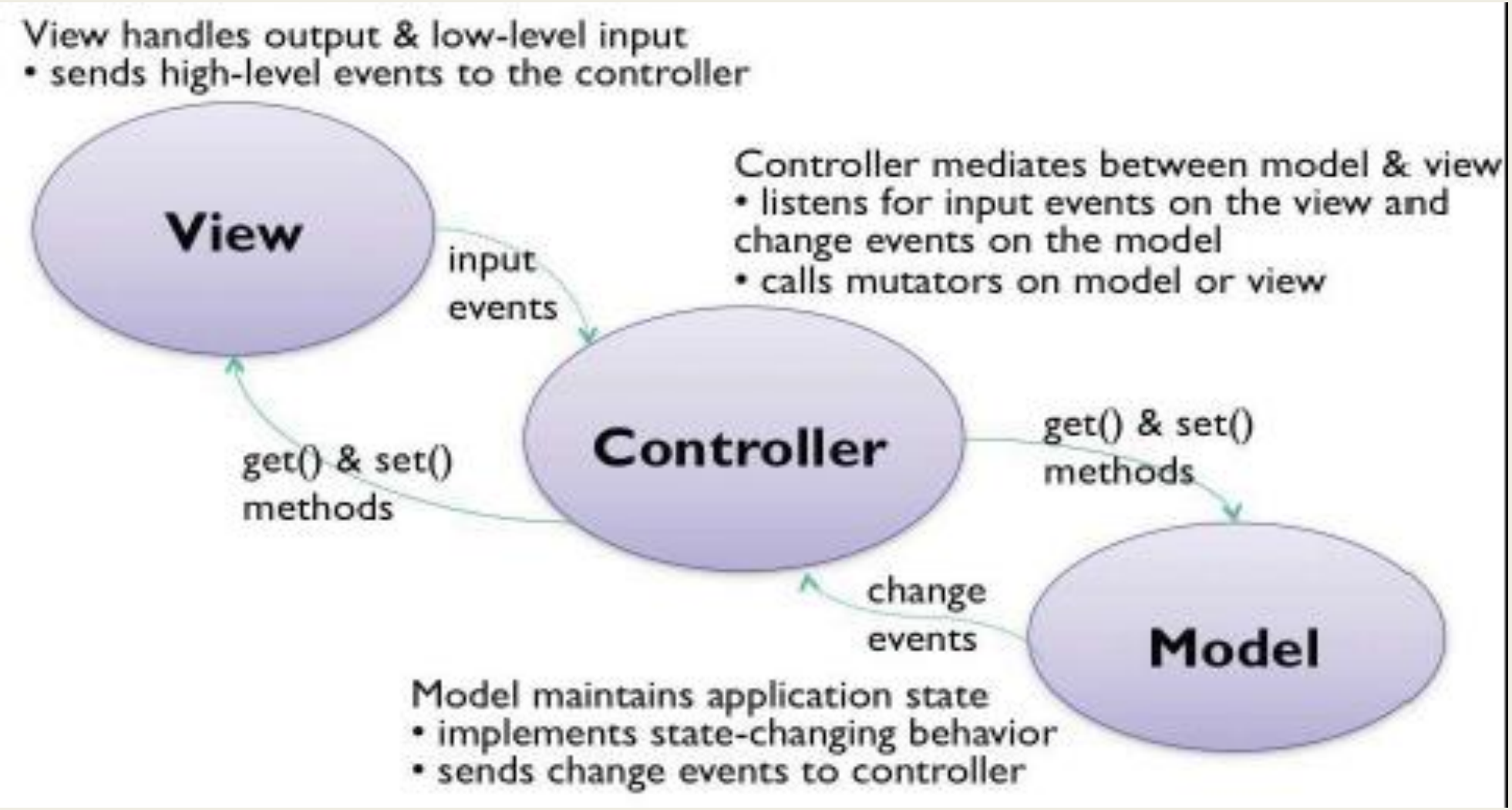
# Hard to separate controller and view

- Controller often needs output
  - *View must provide affordance for the controller (e.g. scrollbar thumb)*
  - *View must also provide feedback about controller state (e.g. depressed button)*
- State shared between Controller and view: Who manages the selection?
  - *Must be displayed by the view (as blinking text cursor or highlight)*
  - *Must be updated and used by the controller*
  - *Should selection be in model?*
    - Generally not
    - Some views need independent selections (e.g. two windows on the same document)
    - Other views need synchronized selections(e.g. table view and chart view)

# Widget: Tightly Coupled View & Controller

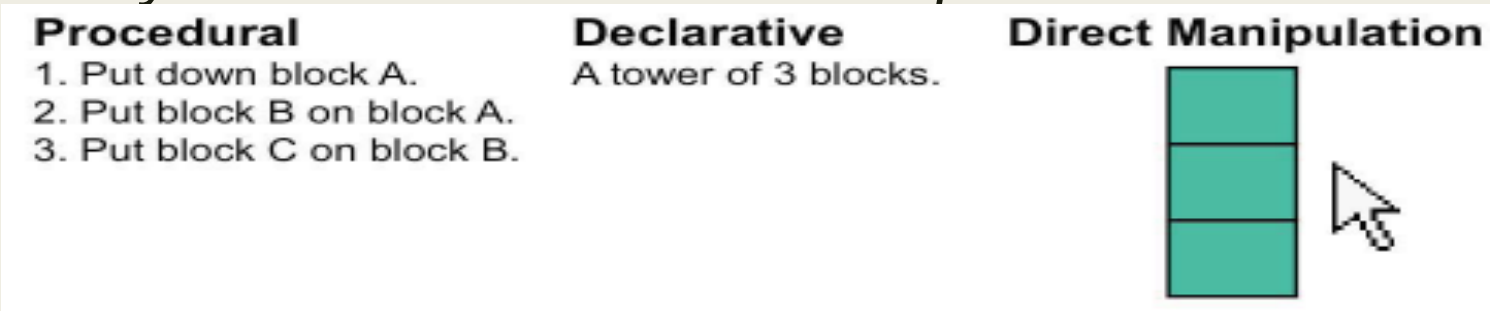
- The MVC ides has largely been superseded by an MV idea
- A widget is a reusable view object that manages both its output and its input
  - *Widgets are sometimes called components (Java, Flex) or controls (Windows)*
- Examples: Scrollbar, button, menubar

# A different perspective on MVC



# GUI Implementation Approaches

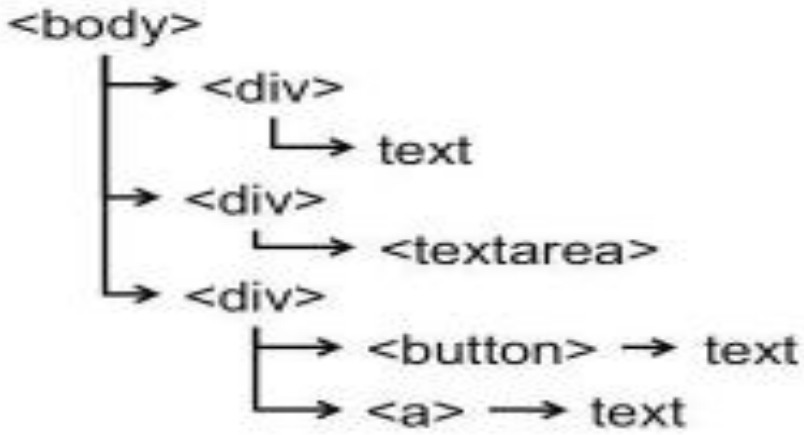
- Procedural Programming
  - *Code that says how to get what you want( flow of control)*
- Declarative Programming
  - *Code that says what you want (no explicit flow of control)*
- Direct manipulation
  - *Creating what you want in a direct manipulation interface*



# Markup Languages

- HTML declaratively specifies a view tree

```
<body>
  <div>What are you doing now?</div>
  <div><textarea></textarea></div>
  <div><button>Send</button> <a href="#">sign out</
   a></div>
</body>
```

# Important HTML Elements for UI design

- **Layout**

| | |
|---|---|
| Box |  |
| Grid | <table>, <tr>, <td> |

- **Text**

| | |
|---|---|
| Font & color |  |

- **Widgets**

| | |
|---|---|
| Hyperlink | <a> |
| Button | <button> |
| Textbox | <input type="text"> |
| Multiline text | <textarea> |
| Rich text |  |
| Drop-down | <select> <option> |
| Listbox | <select multiple="true"> |
| Checkbox | <input type="checkbox"> |
| Radiobutton | <input type="radio"> |

- **Pixel output**
  <img>
- **Stroke output**
  <canvas> (Firefox, Safari)
- **Javascript code**
  <script>
- **CSS style sheets**
  <style>

# View Tree Manipulation

- Javascript can procedurally mutate a view tree

```
<script>
var doc = document
var div1 = doc.createElement("div")
 div1.appendChild(doc.createTextNode("What are you doing now?")
...
var div3 = doc.createElement("div")
 var button = doc.createElement("button")
  button.appendChild(doc.createTextNode("Send"))
  div3.appendChild(button)
 var a = doc.createElement("a")
  a.setAttribute("href", "#")
  a.appendChild(doc.createTextNode("sign out"))
  div3.appendChild(a)
</script>
```

# Javascript in one Slide

## Like Java…

**expressions**

```
hyp = Math.sqrt(a*a + b*b)
console.log("Hello"
            + ", world");
```

**statements**

```
if (a < b) { return a }
   else { return b }
```

**comments**

```
/* this
    is a comment */
// and so is this
```

## Like Python…

**no declared types**

```
var x = 5;
for (var i = 0; i < 10; ++i) {…}
```

**objects and arrays are dynamic**

```
var obj = { x: 5, y: -1 };
obj.z = 8;
var list = ["a", "b"];
list[2] = "c";
```

**functions are first-class**

```
function square(x) { return x*x; }
var double = function(a) {
                    return 2*a; }
```

# Jquery in One Slide

Select nodes

```
$("#send")
$(".toolbar")
$("button")
```

```
<button id="send" class="toolbar">
  Send
</button>
```

Create nodes

```
$('<button class="toolbar"></button>')
```

Act on nodes

```
$("#send").text()                // returns "Send"
$("#send").text("Tweet")         // changes button label
$(".toolbar").attr("disabled", "true")
$("#send").click(function() { ... })
$("#textarea").val()
$("#mainPanel").html("<button>Press Me</button>")
```

# Mixing Declarative and Procedural Code

```
<body>
  <div>What are you doing now?</div>
  <div><textarea id="msg"></textarea></div>
  <div><button id="send">Send</button></div>
  <div id="sent" style="font-style: italic">
    <div>Sent messages appear here.</div>
  </div>
</body>

<script src="http://code.jquery.com/jquery-1.5.min.js"></script>
<script>
  $(function() {
    $("#send").click(function() {
      var msg = $("#msg").val()

    })
  })
</script>
```

What are you doing now?

Send

*Sent messages appear here.*

```
var sent = $("#sent").html()
sent += "<div>" + msg + "</div>"
$("#sent").html(sent)
```

```
var div = $("<div></div>").text(msg)
$("#sent").append(div)
```

# Pros & Cons of Declarative UI

- Usually more compact
- Programmer Only has to know how to say what, not how
  - *Automatic algos are responsible for figuring out how*
- May be harder to debug
  - *Can't set breakpoints, single-step, print in the declarative specification*
  - *Debugging may be more trial-and-error*
- Authoring tools are possible
  - *Declarative spec can be loaded and saved by a tool, procedural specs generally can't*