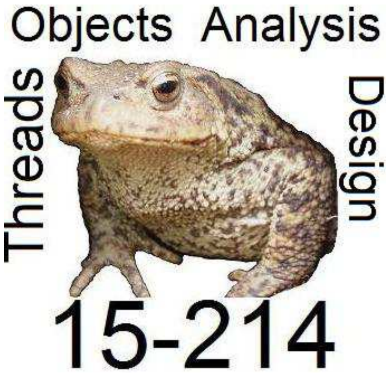
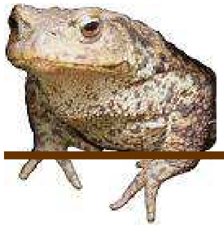
Introduction to GUIs



Principles of Software Construction: Objects, Design, and Concurrency

Jonathan Aldrich and Charlie Garrod

Fall 2013



What makes GUIs different?

- How do they compare to command-line I/O?

What makes GUIs different?

- How do they compare to command-line I/O?

Don't call us, we'll call you!

- GUI has to react to the user's actions
  - Not just a response to a prompt
  - Could involve entirely different functionality
- Requires structuring the GUI around *reacting to events*

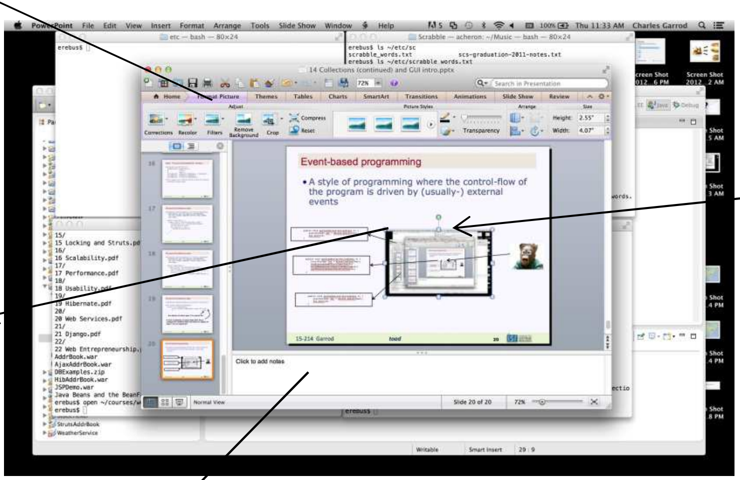
Event-based programming

- A style of programming where the control-flow of the program is driven by (usually-) external events

```
public void
performAction(ActionEvent e)
{
    printSlides()
}
```

```
public void
performAction(ActionEvent e)
{
    editFigure()
}
```

```
public void
performAction(ActionEvent e)
{
    …
}
```



```
erebus$ ls ~/etc/sc
scrabble_words.txt            scs-graduation-2011-notes.txt
erebus$ ls ~/etc/scrabble words.txt
```

```
15/
15 Locking and Struts.pdf
16/
16 Scalability.pdf
17/
17 Performance.pdf
18/
18 Usability.pdf
19/
19 Hibernate.pdf
20/
20 Web Services.pdf
21/
21 Django.pdf
22/
22 Web Entrepreneurship.
AddrBook.war
AjaxAddrBook.war
DBExamples.zip
HibAddrBook.war
JSPDemo.war
Java Beans and the BeanF
erebus$ open ~/courses/w
erebus$
```

Event-based programming

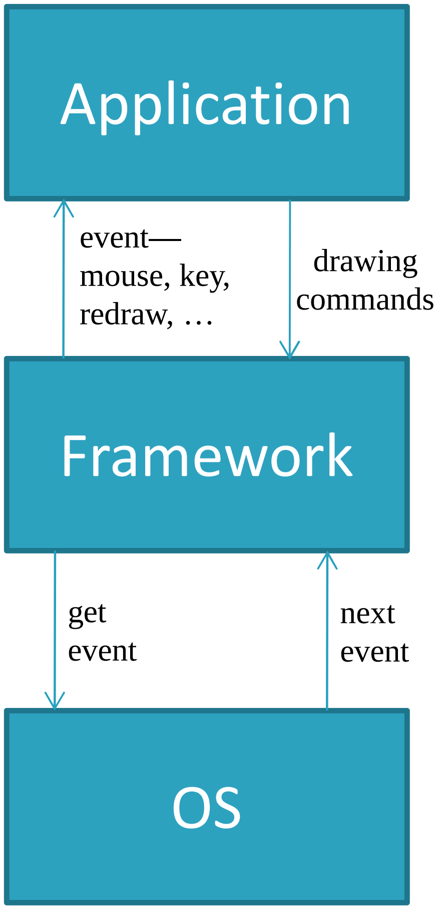
- A style of programming where the control-flow of the program is driven by (usually-) external events

15-214 Garrod   toad   20

Reacting to events - from framework

- Setup phase
  - Describe how the GUI window should look
  - Use libraries for windows, widgets, and layout
  - Define custom functionality
    - New widgets that display themselves in custom ways
    - How to react to events

- Execution phase
  - Framework gets events from OS
    - Mouse clicks, key presses, window becomes visible, etc.
  - Framework triggers application code in response
    - The customization described above



Application

event—
mouse, key,
redraw, …

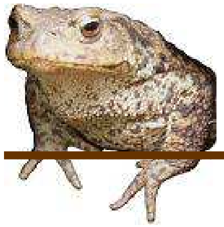
drawing
commands

Framework

get
event

next
event

OS



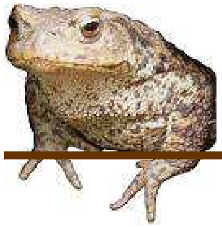
Pseudocode for GUIs

Application code

- Creates and sets up a window
- Asks framework to show the window
- main() exits
- Takes action in response to event
- May contact GUI
  - E.g. consider if event was a redraw
  - Call GUI to paint lines, text

GUI framework code

- Starts the GUI thread
- This thread loops:
  - Asks OS for event
  - Finds application window that event relates to
  - Asks application window to handle event
  - Draws lines/text on behalf of application

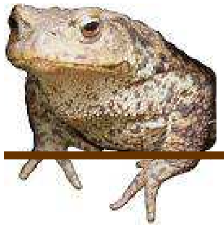


Example: RabbitWorld GUI

- …hw2.lib.ui.WorldUI.main()
  - Creates a top-level window
  - Creates a WorldUI to go in it
  - Sets some parameters
  - Makes the window (and its contents) visible

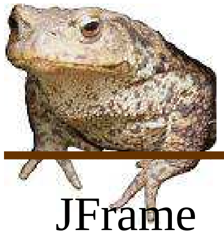
- …hw2.lib.ui.WorldPanel.paintComponent()
  - Called when the OS needs to show the WorldPanel (part of WorldUI)
    - Right after the window becomes visible
  - super.paintComponent() draws a background
  - ImageIcon.paintIcon(…) draws each item in the world

Let's look at the code…



GUI Frameworks in Java

- AWT
  - Native widgets, only basic components, dated
- Swing
  - Java rendering, rich components
- SWT + JFace
  - Mixture of native widgets and Java rendering; created for Eclipse for faster performance
- Others
  - Apache Pivot, SwingX, JavaFX, …



Swing

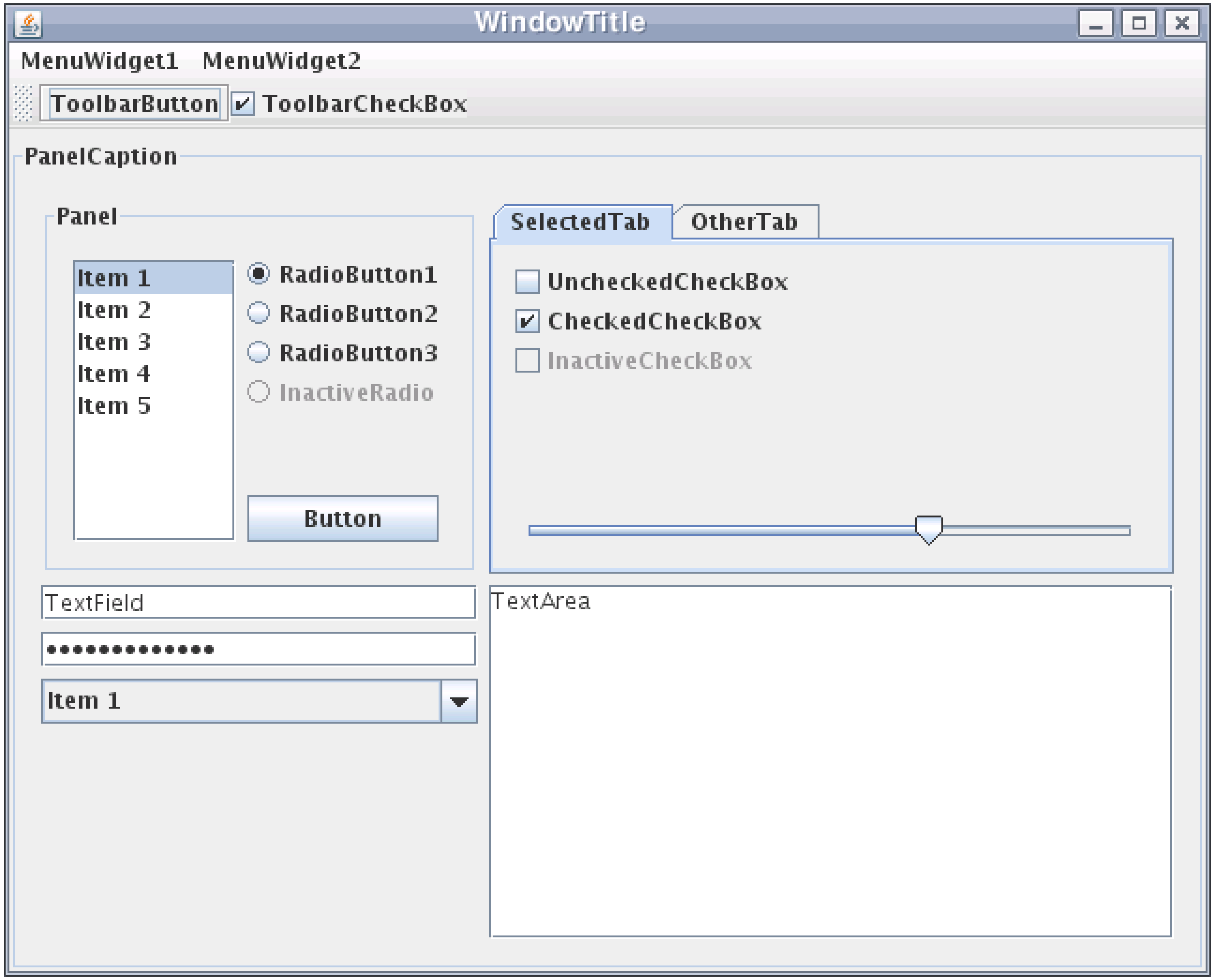
JFrame

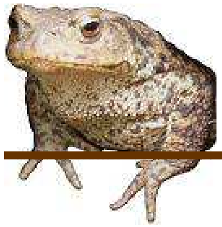
JPanel

JButton

JTextField

…





To create a simple Swing application

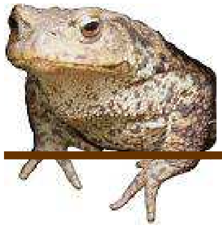
- Make a Window (a JFrame)
- Make a container (a JPanel)
  - Put it in the window
- Add components (Buttons, Boxes, etc.) to the container
  - Use layouts to control positioning
  - Set up observers (a.k.a. listeners) to respond to events
  - Optionally, write custom widgets with application-specific display logic
- Set up the window to display the container

- Then wait for events to arrive…

Components

Swing has lots of components:

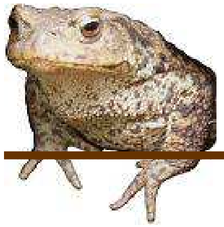
- JLabel
- JButton
- JCheckBox
- JChoice
- JRadioButton
- JTextField
- JTextArea
- JList
- JScrollBar
- … and more

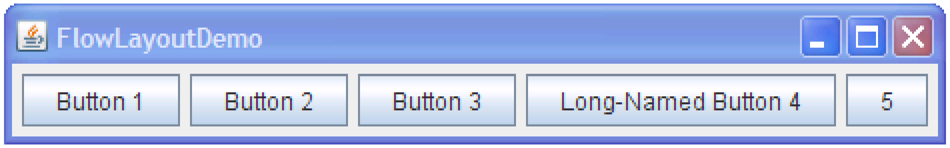


JFrame & JPanel

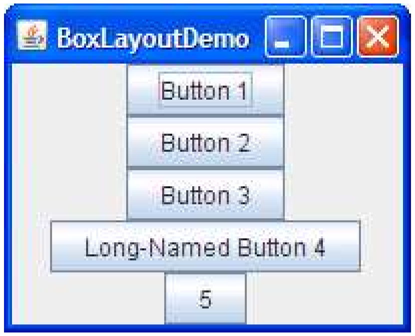
- JFrame is the Swing Window
- JPanel (aka a pane) is the container to which you add your components (or other containers)

Swing Layout Managers

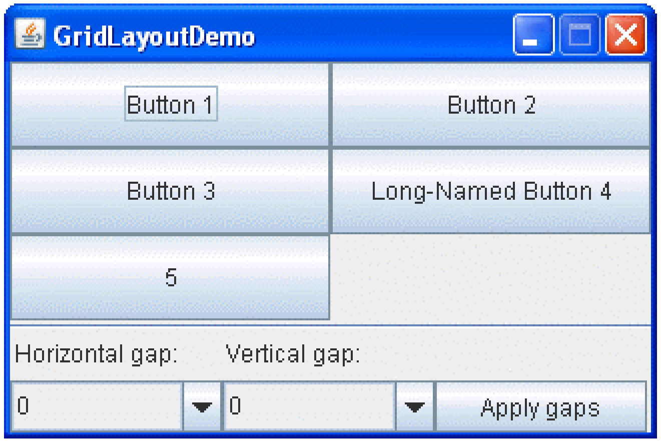




The simplest, and default, layout.
Wraps around when out of space.

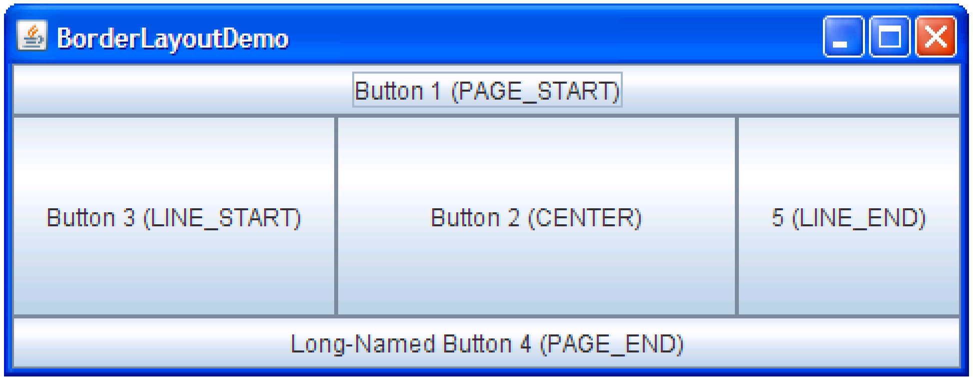


Like FlowLayout, but no wrapping



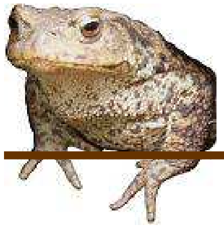
Horizontal gap: 0

Vertical gap: 0



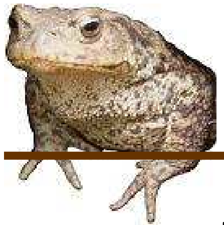
More sophisticated layout managers

see http://docs.oracle.com/javase/tutorial/uiswing/layout/visual.html



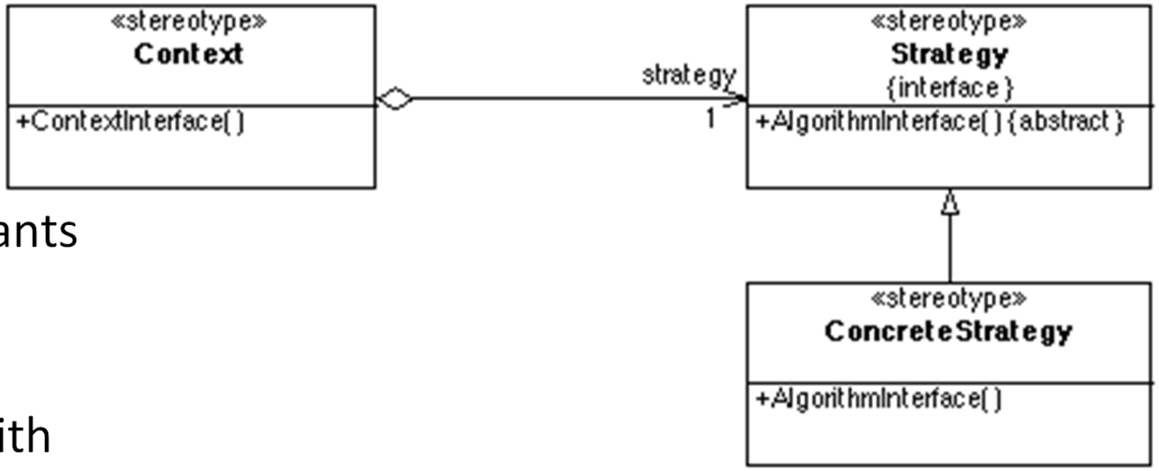
Find the pattern…

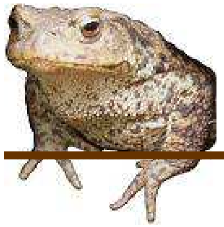
- contentPane.setLayout(new BorderLayout(0,0));
- contentPane.setBorder(new EmptyBorder(5, 5, 5, 5));



Behavioral: Strategy

- Applicability
  - Many classes differ in only their behavior
  - Client needs different variants of an algorithm
- Consequences
  - Code is more extensible with new strategies
    - Compare to conditionals
  - Separates algorithm from context
    - each can vary independently
  - Adds objects and dynamism
    - code harder to understand
  - Common strategy interface
    - may not be needed for all Strategy implementations – may be extra overhead

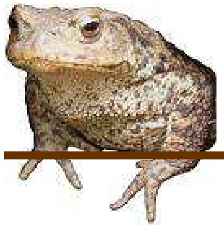




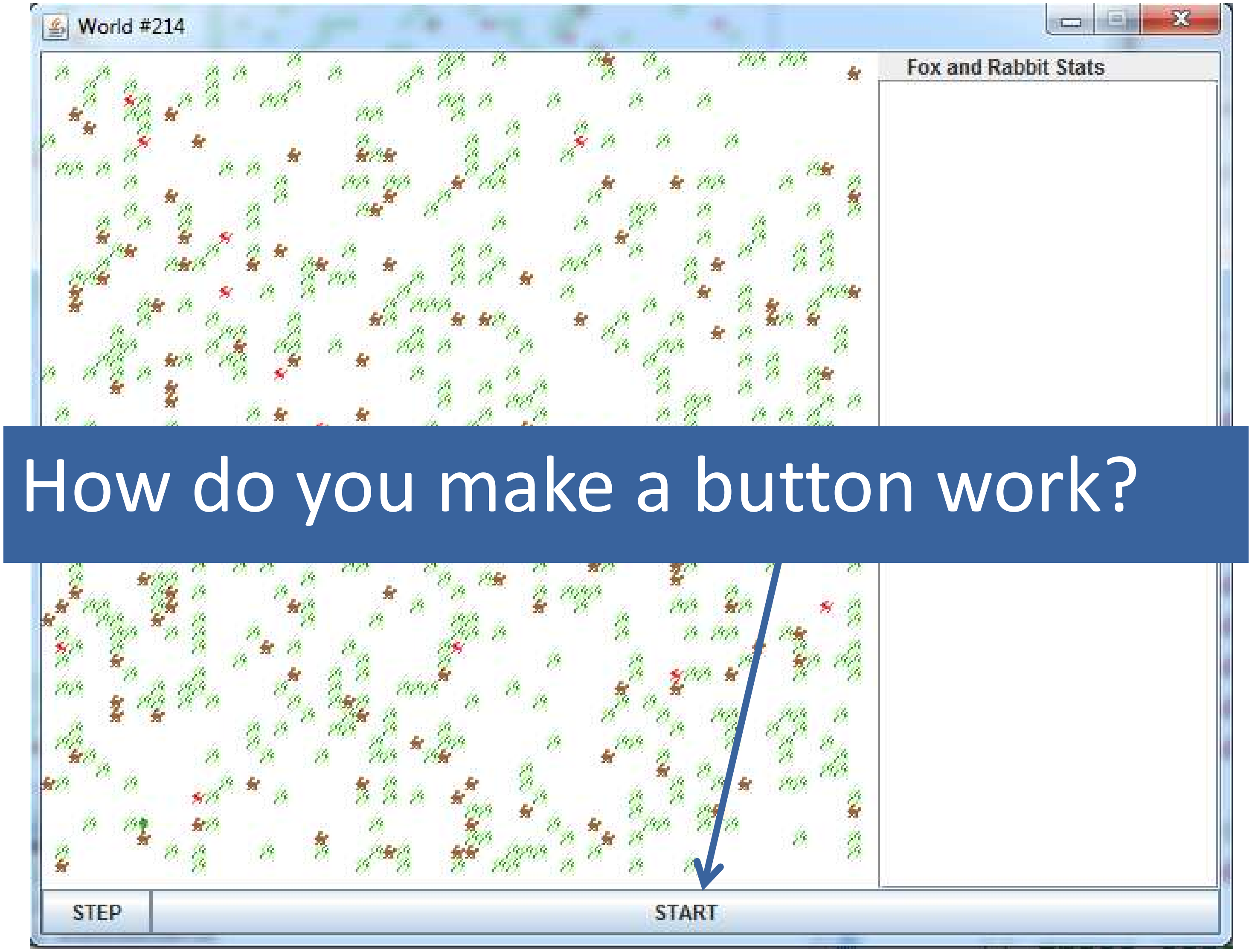
Example: RabbitWorld GUI

- …hw2.lib.ui.WorldUI.WorldUI()
  - Sets the layout to a BorderLayout
  - Adds a WorldPanel in the CENTER of the UI
  - Creates a JPanel for the buttons at the bottom
  - Adds 2 buttons to the JPanel (WEST and CENTER)
  - Puts the button JPanel at the SOUTH side of the WorldPanel

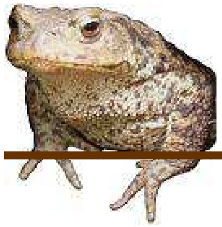
Let’s look at the code again…



Question

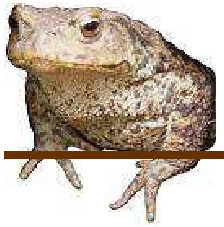


How do you make a button work?



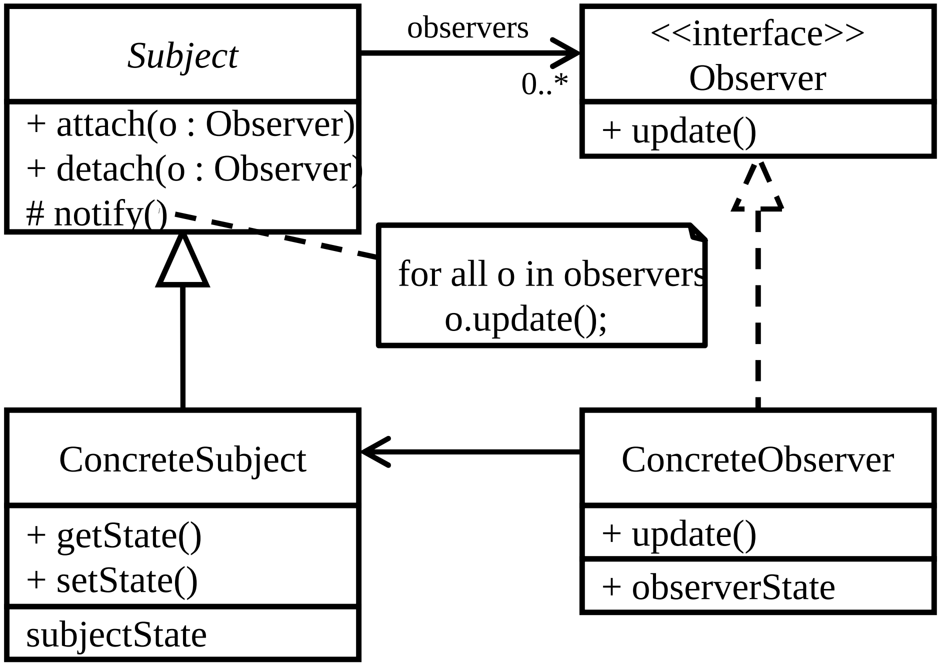
Events in Swing

- An event is when something changes
  - Button clicked, scrolling, mouse movement
- Swing (actually AWT) generates an event
- To do something you need to implement a Listener Interface and register interest



The *Observer* design pattern

- Applicability
  - When an abstraction has two aspects, one dependent on the other, and you want to reuse each
  - When change to one object requires changing others, and you don’t know how many objects need to be changed
  - When an object should be able to notify others without knowing who they are
- Consequences
  - Loose coupling between subject and observer, enhancing reuse
  - Support for broadcast communication
  - Notification can lead to further updates, causing a cascade effect



Subject
+ attach(o : Observer)
+ detach(o : Observer)
# notify()

observers
0..*

<<interface>>
Observer
+ update()

for all o in observers
o.update();

ConcreteSubject
+ getState()
+ setState()
subjectState

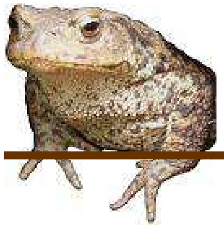
ConcreteObserver
+ update()
+ observerState

Also called **Listener**

Event Listeners

Swing has lots of event listener interfaces:

- ActionListener
- AdjustmentListener
- FocusListener
- ItemListener
- KeyListener
- MouseListener
- TreeExpansionListener
- TextListener
- WindowListener
- …and on and on…

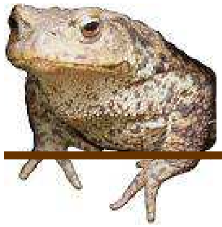


ActionListener

- Events for JButtons, JTextFields, etc
  - The things we are using
- Implement ActionListener
  - Provide actionPerformed method
- In actionPerformed method
  - Use event.getSource() to determine which button was clicked, etc.

Example: RabbitWorld GUI

- …hw2.lib.ui.WorldUI.WorldUI()
  - Sets ActionListeners for the **run** and **step** buttons
    - Anonymous inner classes used
    - A single method actionPerformed(…) is overridden
    - **step** button: just calls step() on the WorldPanel
      - Steps the world
      - Requests that the window be refreshed (so the user can see the changes)
    - **run** button
      - Starts the world continuously stepping
      - Disables the **step** button (no point!)
      - Sets a toggle flag so that pressing the button again will stop the simulation



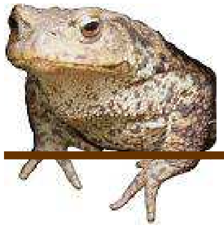
Aside: Anonymous inner classes in Java

- You can implement an interface without naming the implementing class
  - E.g.,

```
public interface Runnable {
    public void run();
}

public static void main(String[] args) {
    Runnable greeter = new Runnable() {
        public void run() {
            System.out.println("Hi mom!");
        }
    };

    greeter.run();
}
```



Scope within an anonymous inner class

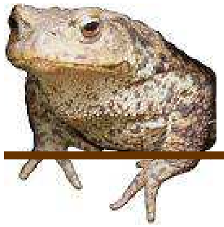
- An anonymous inner class cannot access non-final variables in the scope where it is defined

```
public interface Runnable {
    public void run();
}

public static void main(String[] args) {
    String name = "Charlie";
    Runnable greeter = new Runnable() {
        public void run() {
            System.out.println("Hi " + name);
        }
    };

    greeter.run();
}
```

compile-time error



Scope within an anonymous inner class

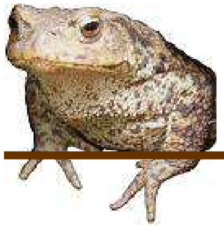
- An anonymous inner class cannot access non-final variables in the scope where it is defined

```
public interface Runnable {
    public void run();
}

public static void main(String[] args) {
    final String name = "Charlie";
    Runnable greeter = new Runnable() {
        public void run() {
            System.out.println("Hi " + name);
        }
    };

    greeter.run();
}
```

OK

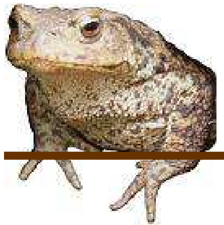


Organizational Tips

- Declare references to components you’ll be manipulating as instance variables
- Put the code that performs the actions in private “helper” methods. (Keeps things neat)

GUI design issues

- Interfaces vs. inheritance
  - Inherit from JPanel with custom drawing functionality
  - Implement the ActionListener interface, register with button
  - Why this difference?
- Models and views

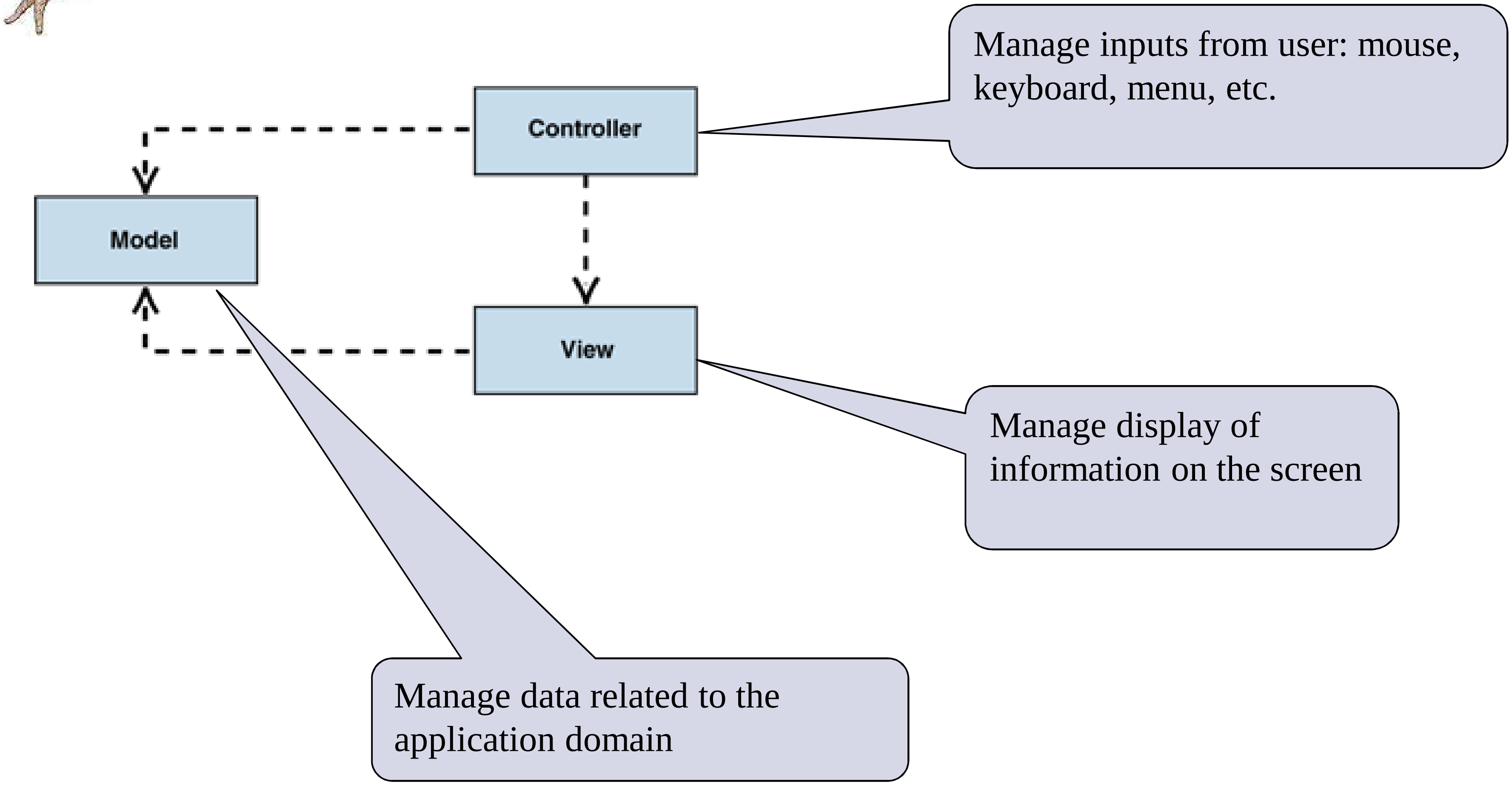


GUI design issues

- Interfaces vs. inheritance
  - Inherit from JPanel with custom drawing functionality
    - Subclass “is a” special kind of Panel
    - The subclass interacts closely with the JPanel – e.g. the subclass calls back with super()
    - The way you draw the subclass doesn’t change as the program executes
  - Implement the ActionListener interface, register with button
    - The action to perform isn’t really a special kind of button; it’s just a way of reacting to the button. So it makes sense to be a separate object.
    - The ActionListener is decoupled from the button. Once the listener is invoked, it doesn’t call anything on the Button anymore.
    - We may want to change the action performed on a button press—so once again it makes sense for it to be a separate object

- Models and views

Model-View-Controller (MVC)



Manage inputs from user: mouse, keyboard, menu, etc.

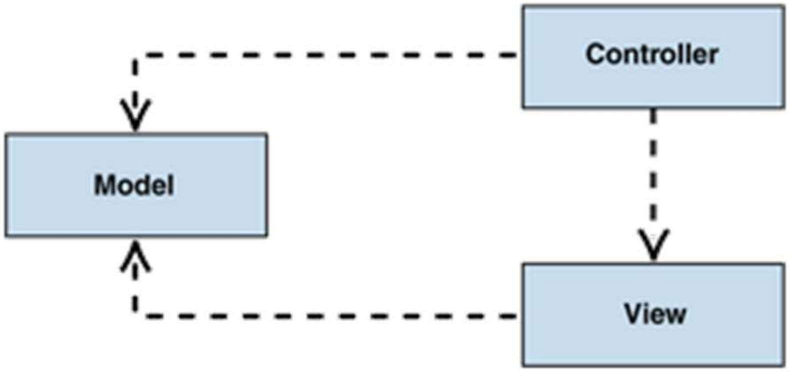
Manage display of information on the screen

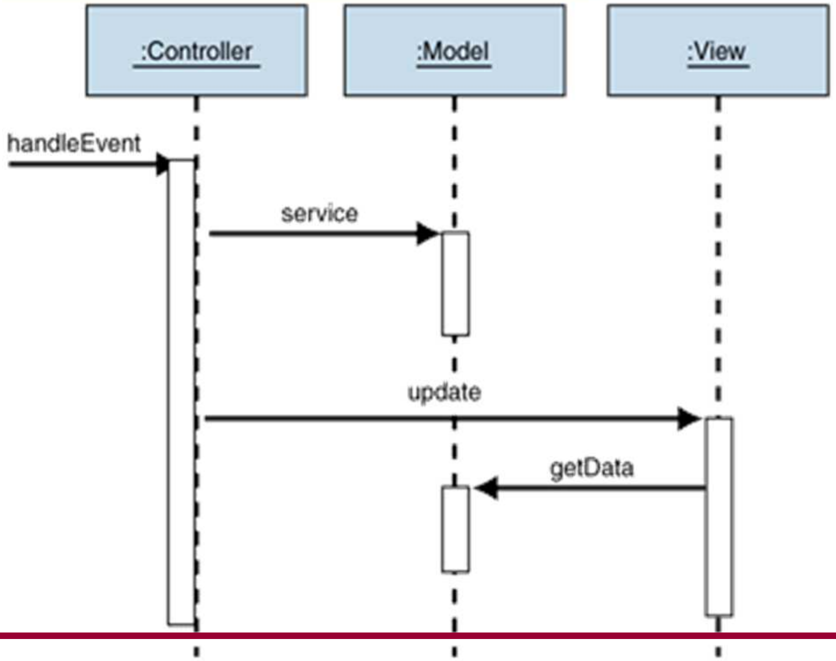
Manage data related to the application domain

http://msdn.microsoft.com/en-us/library/ff649643.aspx

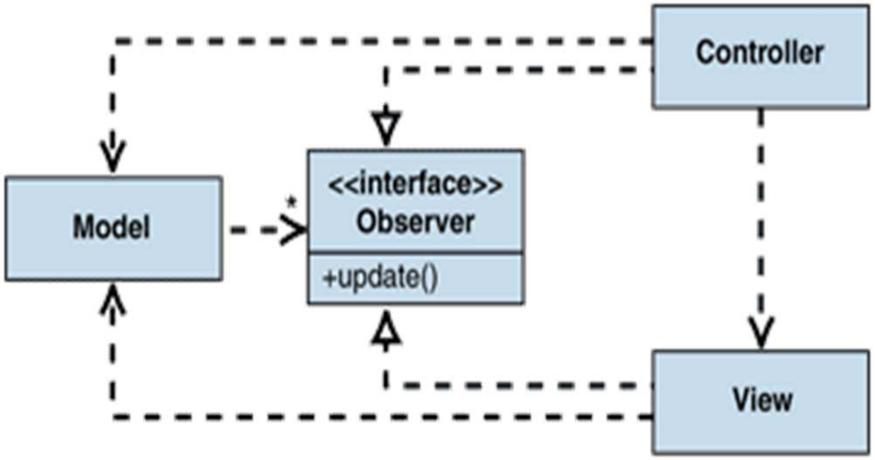
Model-View-Controller (MVC)

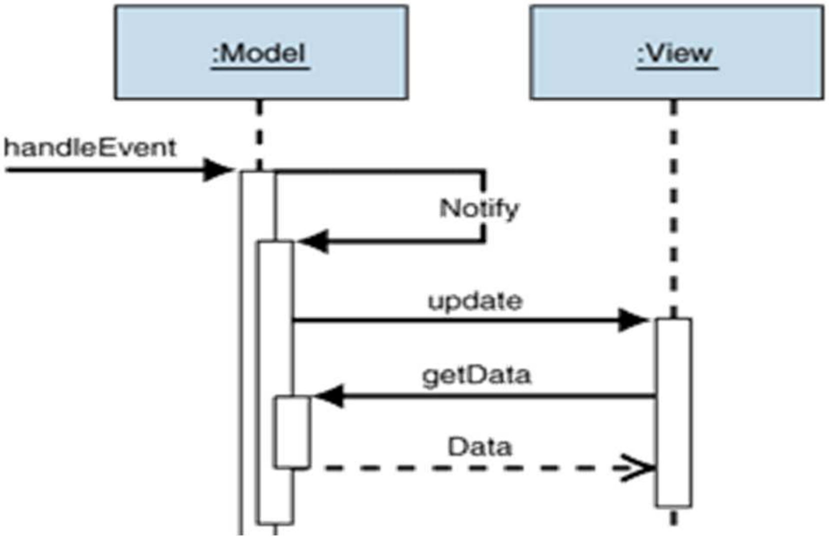
Passive model



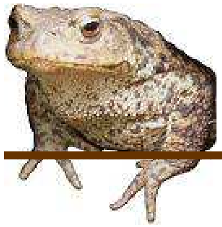


Active model



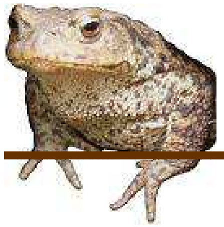


http://msdn.microsoft.com/en-us/library/ff649643.aspx



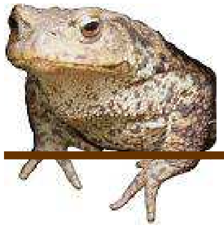
Example: RabbitWorld GUI

- …hw2.lib.ui.WorldImpl
  - The Model class
  - Model is passive: does not have a reference to the view
- …hw2.lib.ui.WorldUI
  - The Controller class
  - Listener callbacks in constructor react to events
    - Delegating to the view (is this design ideal?)
- …hw2.lib.ui.WorldPanel
  - The View class
  - Gets data from Model to find out where to draw rabbits, foxes, etc.
  - Implements stepping (in step())
    - Invokes model to update world
    - Invokes repaint() on self to update UI



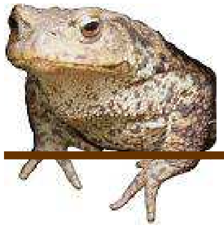
Find That Pattern!

- What pattern is BorderLayout a part of?
- What pattern is JPanel a part of?
- What pattern are the ActionListeners part of?
- There are classes representing the AI’s decision to Eat, Breed, or Move. What pattern are these representing?
- Look at the documentation for JComponent.paint(). What pattern is used?



For More Information

- Oracle’s Swing tutorials
  - http://download.oracle.com/javase/tutorial/uiswing/
- Introduction to Programming Using Java, Ch. 6
  - http://math.hws.edu/javanotes/c6/index.html



Questions?