CS11 – Advanced Java

Winter 2011-2012

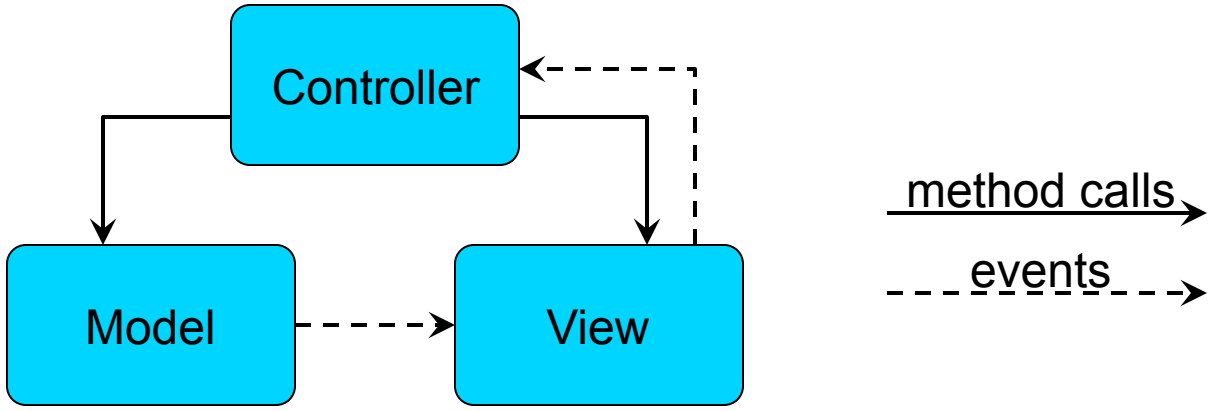
Lecture 5

User-Interface Architecture

- Model-View-Controller (MVC)
  - A very powerful design pattern for creating user interfaces
- Separate GUI applications into three components:
- Model
  - The actual data that is being displayed and manipulated via the user interface
- View
  - The visual representation, displayed in the user interface
- Controller
  - Receives user inputs from the UI, and manipulates the model and the view appropriately

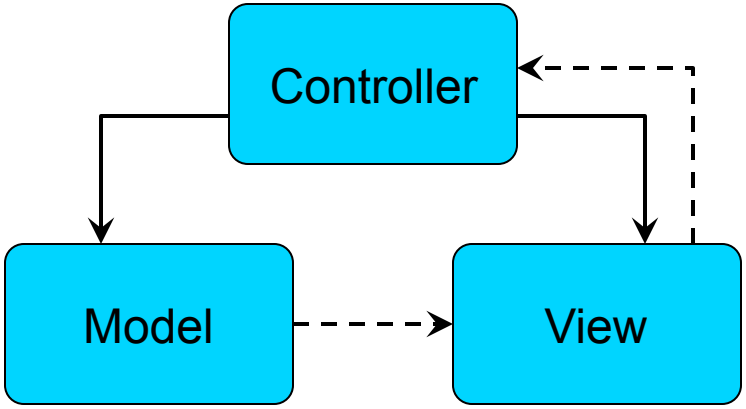
Model-View-Controller Pattern

- Frequently represented like this:



- The View "observes" the Model
  - View receives "data changed" notifications from model
  - View manages UI; updates display when model changes
  - Most efficient when model indicates exactly what changed

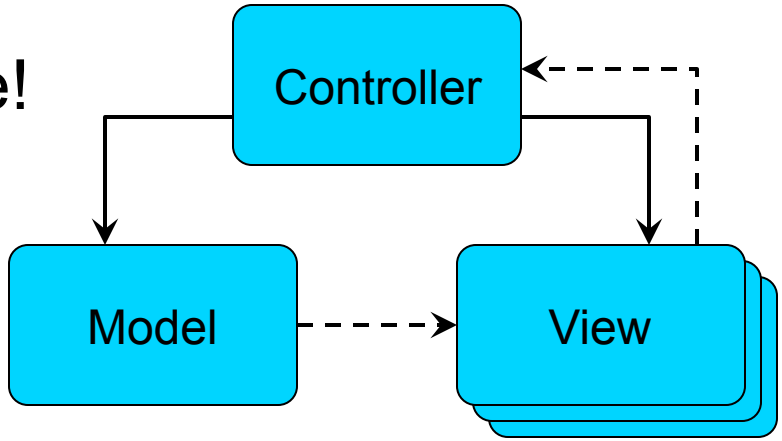
Model-View-Controller Pattern (2)



- The Controller receives input events from the View
  - e.g. “user pressed a button” or “user selected a list-item”
  - Controller then makes changes to Model, or to View, depending on user input

Benefits of MVC

- Much cleaner UI architecture!
  - Don't mix model, view, and controller code together
  - Much easier to change/add features later



- Very easy to add new views
  - Views simply register to receive "model changed" events
- Can't always use MVC approach
  - Requires extra code to make model "observable"
  - Sometimes model isn't complex enough to warrant the extra effort
  - For generic, extensible user interfaces, use MVC approach!

Swing and MVC

- Many Swing classes follow Model-View-Controller pattern
- Example: **`javax.swing.JList`**
  - **`public JList(ListModel dataModel)`**
  - **`JList`** component is a view into a list of data, exposed via the **`ListModel`** interface
  - User can interact with the view
    - View fires **`ListSelectionEvent`** objects
- You can provide the Model yourself
  - Implement the **`ListModel`** interface
- You also provide the Controller

ListModel Interface

- **`ListModel`** is a simple interface:

```
Object getElementAt(int index)
int getSize()
void addListDataListener(ListDataListener l)
void removeListDataListener(ListDataListener l)
```

- **`ListDataListener`** interface allows view to know when model's data changes

```
void intervalAdded(ListDataEvent e)
void intervalRemoved(ListDataEvent e)
void contentsChanged(ListDataEvent e)
```

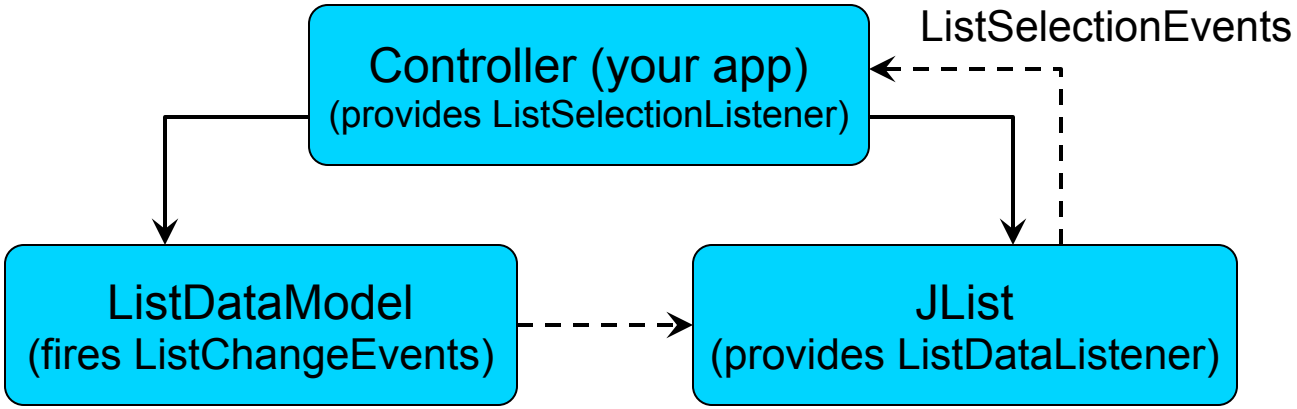
  - Model fires these list-data events
  - View updates its appearance; resyncs UI with model state

ListModel Implementations

- Most Swing apps don't need sophisticated models
- Swing has default impls. of model interfaces
  - Provide code to fire events based on model changes
  - Programmer only has to specify what is being stored
- **`javax.swing.DefaultListModel`**
  - Provides API similar to java.util.Vector or java.util.List
  - Store **`Object`** values at specific indexes in the model
  - **`toString()`** method is used to display each object's value
  - When data changes, fires events that **`JList`** receives

`JList` Diagram

- Model-View-Controller components of JList:



- JList observes ListDataModel via events
- Controller gets user input via list-selection events
- Controller manipulates both JList and ListDataModel based on user input, etc.

New Concept: Observable Objects!

- So far, only UI components fire events
  - e.g. when user does something
- Can also make data objects that fire events when their data changes
- Called the Observer pattern
  - Also known as Publish-Subscribe (or “pubsub” for short)
- Observable data object publishes change-notifications
- Interested observers subscribe to these notifications

Observer Pattern in Java

- Java provides two utility types for this pattern
- **`java.util.Observable`** base-class
  - **`addObserver(Observer o)`**
  - **`boolean hasChanged()`**
  - **`notifyObservers(Object arg)`**
  - A data object can derive from **`Observer`**
  - Argument to **`notifyObservers()`** can specify exactly what changed
- **`java.util.Observer`** interface
  - **`void update(Observable o, Object arg)`**
  - An observer can implement this interface, then register on one or more **`Observable`** objects
  - Use **`Observable`** and argument to know what happened

Problems with Java `Observable`…

- A few big limitations of `Observable` ☹
- It's a base-class, not an interface
  - If your data-object needs to derive from something else, you can't use these classes
  - No multiple-inheritance in Java
  - When you design classes like this, prefer interfaces to base-classes!
    - Effective Java, Item 16 for more details on this!
- Only have one notification method, with an `Object` argument!
  - No type constraints on argument…
  - Can't provide multiple methods that handle different kinds of data-change events (e.g. data-added, data-removed, …)

Swing and Observer Patterns

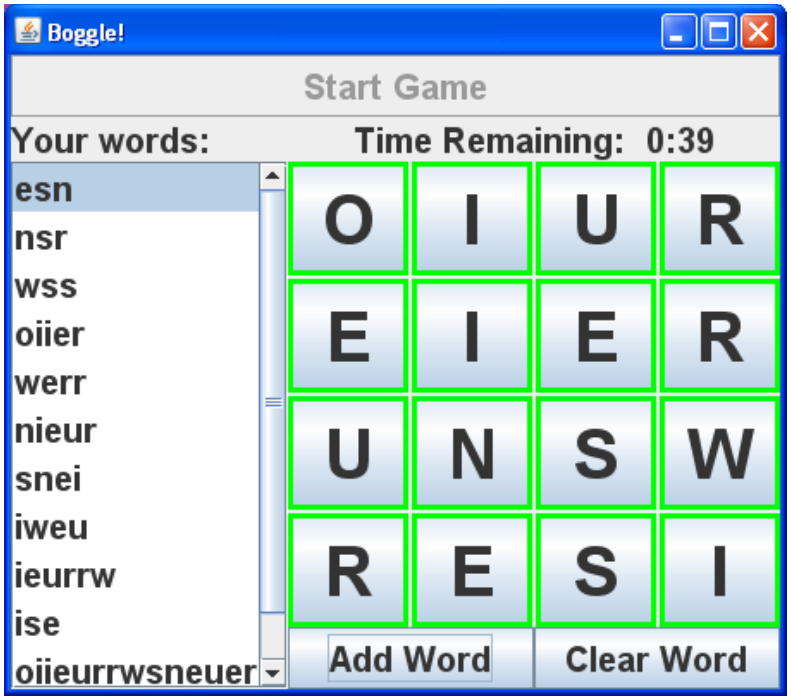
- Lists, trees, tables all use MVC pattern
  - All have observable models
  - Models and their observers are specified using interfaces
  - (None of them use **`java.util.Observable`**…)
- You can emulate this pattern too.
  - FooModel
    - The data model interface
  - FooDataEvent (a subclass of java.util.Event)
    - Describes some change in the Foo model
    - Different event-types specify different kinds of changes
  - FooDataListener (a subinterface of java.util.EventListener)
    - Observers of FooModel implement this interface
    - Provide several interface methods, for different data changes

Controllers

- Application's Controller handles events from View
  - (possibly also events from other sources…)
  - Updates Model (and possibly Views) based on user input
- Controller needs access to the Model and the Views
- For large apps, controller can be a separate top-level class
  - References to Model and Views are passed to Controller
- For small apps, controller can be an inner class that implements UI event-listener interfaces
  - Can access enclosing class' fields and methods
  - Can operate on Model and View(s) directly

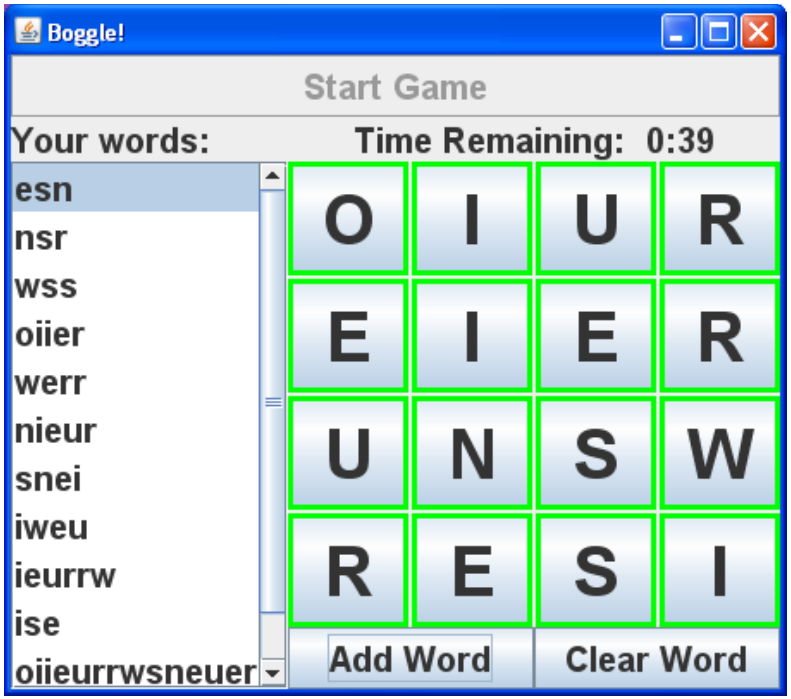
Boggle User Interface

- This week, should finish off most of Boggle client user-interface
- Too simple to apply MVC at application-level…
- Some parts will use MVC
  - List of words is a **`JList`**; definitely uses MVC
  - Boggle-board is kinda MVC, but board doesn't change
- Should have one Controller
  - Probably an inner class of Boggle app



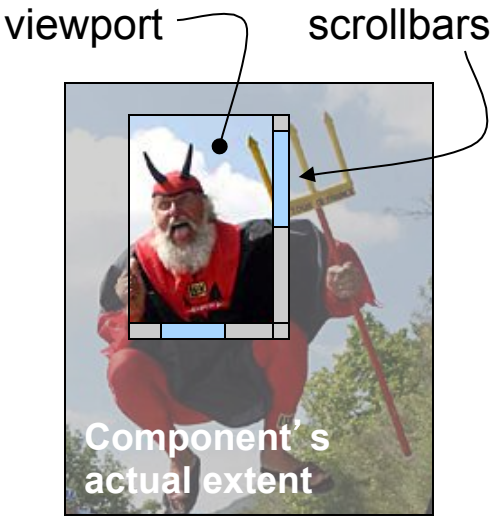
Boggle UI Controller

- Boggle Controller should be easy
  - All UI components fire ActionEvents
  - Controller is just an ActionListener handler that encodes app logic
- Apps are usually more complex, in general
  - Several different kinds of events to handle



Scrollable Lists

- Need to support scrolling in our list of words
  - Sometimes can exceed display-size of list
- Swing components don't provide scrolling themselves!
  - **`javax.swing.JScrollPane`** wraps another Swing component
  - Adds scrolling capabilities to the component
    - Called the Decorator pattern
  - Can configure scroll-pane for when scrollbars appear, etc.

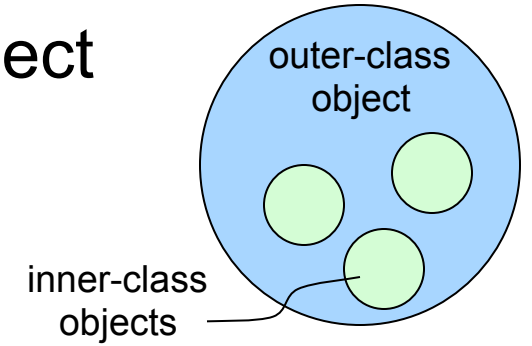


Inner Classes vs. Nested Classes

- Java has inner classes and nested classes
  - What are the differences between the two, if any?
- A class can contain class declarations
  - All such declarations are called nested classes
- Nested classes can be static or non-static
  - Non-static nested classes are called inner classes

Inner Classes

- May have used inner classes extensively
  - Particularly good for UI event-handler code
- Objects of an inner-class type can access the enclosing class' members
  - Embedded within outer-class object
  - Inner-class objects must be constructed in context of an enclosing object
  - Cannot create an inner class within a static method



Inner Class Example

- Will this work?

```
public class MyApp {
  ...
  private class ActionHandler implements ActionListener
  { ... }

  private static void initGUI() {
    JFrame f = new JFrame("My App!");
    JButton b = new JButton("Go");
    ...
    ActionHandler h = new ActionHandler();
    b.addActionListener(h);
    ...
```

- No!
  - Inner class can only be created in context of an outer object
  - e.g. can only construct inner class where **`this`** is defined
  - Static methods cannot construct inner classes

Inner Class Example (2)

- Need to change UI init code to be nonstatic:

```
public class MyApp {
  ...
  private class ActionHandler implements ActionListener
  { ... }

  private void initGUI() {
    JFrame f = new JFrame("My App!");
    JButton b = new JButton("Go");
    ...
    ActionHandler h = new ActionHandler();
    b.addActionListener(h);
    ...
```

- This can affect how some operations are performed

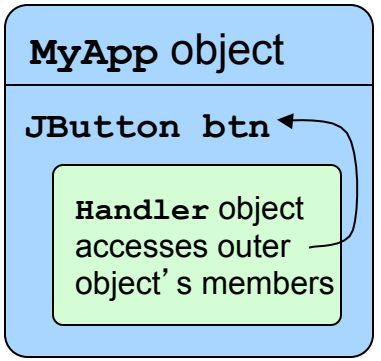
Inner Class Example (3)

- Example code:

```
public class MyApp {
  private JButton btn;

  private class Handler implements ActionListener {
    public void actionPerformed(ActionEvent e) {
      String cmd = e.getActionCommand();
      if (cmd.equals("stop"))
        btn.setEnabled(false);
    }
  }
  ...
```

  - Inner class can access enclosing object's members



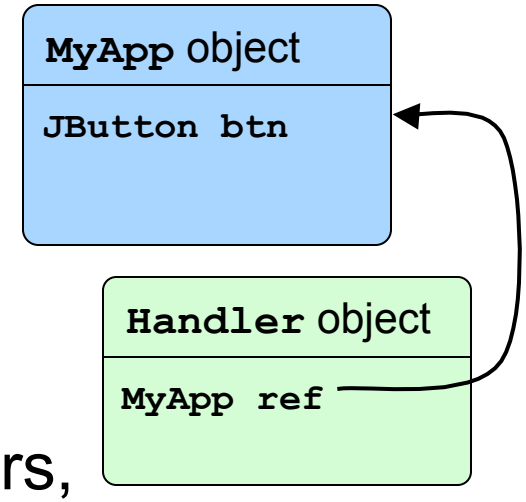
Inner Class Implementation

- When inner class is constructed, it is implicitly passed a reference from the enclosing object

```
private void initGUI() {
  btn = new JButton("Go");
  Handler h = new Handler();
  btn.addActionListener(h);
  ...
```

  - Compiler generates code like this:

```
Handler h = new Handler(this);
```



- When Handler refers to MyApp members, compiler uses reference to parent

```
if (cmd.equals(stop))
  btn.setEnabled(false);
```

  - Compiler generates code like this:

```
ref.btn.setEnabled(false);
```

More Inner-Class Details

- Can construct an inner class from outside the enclosing class!

```
class Foo {
  class Bar {
    ...
  }

  public static void main(String[] args) {
    // DOESN'T COMPILE!
    Bar b = new Bar();

    // OK:
    Foo f = new Foo();
    Bar b = f.new Bar();  // specify outer obj.
  }
}
```

Even More Inner-Class Details

- Inner class can use/return enclosing-object reference

```
class Foo {
  class Bar {
    Foo getMyFoo() {
      return Foo.this;
    }
  }

  public static void main(String[] args) {
    Foo f = new Foo();
    Bar b = f.new Bar();

    // This prints true:
    System.out.println(f == b.getMyFoo());
  }
}
```

Static Nested Classes

- Can also create static nested classes
  - Useful for grouping very closely related classes
  - (Alternative is to use packages, of course!!)
- Example:

```
public class ImageProcessor {
  /** Encapsulates image details. */
  public static class ImageInfo {
    int width, height;
    ...
  }
  ...
}
```

- Static nested classes have no enclosing object
  - Is simply a class declaration nested within another class

Static Nested Classes (2)

- Inside the outer class, can use inner class like any other class

```
public class ImageProcessor {
  /** Encapsulates image details. **/
  public static class ImageInfo { ... }

  public ImageInfo getImage(String filename) {
    ImageInfo info = new ImageInfo(...);
    ...
    return info;
}
```

- Outside outer class, must specify qualified name of inner class

```
ImageProcessor proc = ...
ImageProcessor.ImageInfo info =
  proc.getImage("image.png");
```

Static Nested Classes (3)

- Can create static nested classes in static methods
  - Static nested classes don't have an enclosing object

```
ImageProcessor.ImageInfo info =
  new ImageProcessor.ImageInfo(...);
```

Static Nested Classes and Java API

- Static nested classes used in several Java API packages
- Example: **`java.awt.geom.Point2D`**
  - An *abstract* 2D point class
  - **`Point2D`** contains two static nested classes:
    - **`Float`**
      - A concrete subclass of **`Point2D`** with **`float`** coordinates
      - Full name is **`Point2D.Float`**
    - **`Double`**
      - A concrete subclass of **`Point2D`** with **`double`** coordinates
      - Full name is **`Point2D.Double`**

This Week's Assignment

- Complete the Boggle client user interface
  - Create the UI layout
  - Create a Controller to manage everything
  - Work with JList and ListModel
- I will give you most of the timer code
  - Too much to write in one lab…