
CS11 – Java

Fall 2009-2010

Lecture 4

Today's Topics

- Interfaces
 - The Swing API
 - Event Handlers
 - Inner Classes
 - Arrays
-

Sets of Behaviors

- Frequently have situations where:
 - A single, well-defined set of behaviors...
 - ...with many different possible implementations
 - Could solve this with abstract base classes
 - Base class only declares method signatures, but provides no implementation at all
 - Other classes could derive from the abstract base class and implement the methods
-

Sets of Behaviors (2)

- Abstract base class approach has weaknesses
 - Sometimes need to implement multiple sets of behaviors on one class
 - Java limits classes to only have one parent-class!
 - (No multiple-inheritance in Java!)
 - Base-class is tightly coupled to all subclasses
 - If base-class implementation is changed, may affect a large number of classes that depend on it!
 - Makes it harder to test a particular subclass, and the code that interacts with the class hierarchy!
-

Interfaces

- Classes declare and (usually) define behavior
 - Can't instantiate a class unless it completely defines its set of behaviors
 - No multiple inheritance in Java!
 - By itself, class can't provide multiple different sets of behaviors
 - Interfaces are similar to classes, but only contain method signatures with no bodies!
 - They only *declare* behavior; they don't *define* it
 - No method implementations, no instance fields
 - A class can implement multiple interfaces
-

Interfaces (2)

- Interfaces “define a protocol of communication between two objects.”
 - The interface declares a set of methods (behaviors)
 - A class implements an interface to denote that it provides that set of behaviors
 - Other objects can use the interface type to interact with the implementing object
-

Declaring Interfaces

- Interfaces are declared like classes

```
/** A generic component of a simulation. */  
public interface SimComponent {  
    /** Initialize the component. */  
    void init(SimConfig sconf);  
  
    /** Advance the simulation. */  
    void simulate(double timestep);  
  
    /** End the simulation. */  
    void shutdown();  
}
```

- Goes in `SimComponent.java`
 - No method access-modifiers! Access is public by default.
-

Interfaces and Classes

- Classes can implement interfaces
 - Allows instances to be treated as the interface type
 - A class can implement any number of interfaces
 - A simpler, cleaner version of multiple inheritance
 - Interfaces themselves cannot be instantiated
 - They must be implemented by a class, then the class is instantiated
 - Variables *can* be an interface type, just like they can be a class type
 - (or an abstract class type)
-

Implementing Interfaces

- When a class implements the interface, it must declare the methods as public.

```
public class PhysicsEngine implements SimComponent {  
    ...  
    public void init(SimConfig simConf) {  
        ... // Do some stuff  
    }  
    public void simulate(double timestep) {  
        ... // Do other stuff  
    }  
    ...  
}
```

- Anyone can call the class' implementation of interface, because it's public.
-

Using Interfaces

- Use interfaces to decouple program components
 - ...especially when a component may be implemented in multiple ways!
 - Other components interact with the general interface type, not specific implementations
- Example: storing a user's calendar of events

```
public interface CalendarStorage {  
    // Load a user's calendar of events  
    Calendar loadCalendar(String username);  
  
    // Save the user's calendar to persistent storage  
    void saveCalendar(String username, Calendar c);  
}
```

Using Interfaces (2)

- Provide multiple implementations

- Store calendars in local data files:

```
public class FileCalendarStorage
    implements CalendarStorage {
    ...
}
```

- Store calendars on a remote server:

```
public class RemoteCalendarStorage
    implements CalendarStorage {
    ...
}
```

- Write code to the interface, not implementations

```
CalendarStorage calStore = openCalendar();
Calendar cal = calStore.loadCalendar(username);
```

Using Interfaces (3)

- Can change implementation details as needed...
 - ...as long as interface definition *stays the same*.
- If interface's implementation is large and complex:
 - Other code can use a “stubbed-out” implementation of the interface, until the full version is finished

```
public class FakeCalendarStorage
    implements CalendarStorage {
    public Calendar loadCalendar(String username) {
        return Calendar(username); // Blank calendar
    }
    public void saveCalendar(String username, Calendar c) {
        // Do nothing!
    }
}
```

- Allows software development of dependent components to proceed in parallel
-

Extending Interfaces

- Can extend interfaces, just like classes:

```
/** A sim-component that runs in a network. */  
public interface DistributedSimComponent  
    extends SimComponent {  
  
    /** Establish connection to server. */  
    void connect(String hostname);  
  
    /** Disconnect from server. */  
    void disconnect();  
}
```

- This interface inherits all **SimComponent** method declarations
 - Again, they are all public access
-

Interfaces in the Java API

- *Many* parts of the Java API use interfaces
 - Threads, collections, etc.
 - AWT and Swing APIs use interfaces extensively
 - Many interfaces have “...able” naming convention
- Example: `java.lang.Runnable` and threads

```
public class MyClass implements Runnable {  
    ...  
    public void run() { // Implement the Runnable interface  
        // Do stuff in a separate thread.  
    }  
}  
...  
Runnable r = new MyClass(...);  
Thread t = new Thread(r); // Pass MyClass as a Runnable  
t.start();
```

Swing: A Quick Tour

- First GUI framework in Java was the AWT
 - Abstract Windowing Toolkit
 - Could perform basic operations
 - Not very pretty, or extensible
 - Java 1.2 introduced the Swing API
 - Built on top of some AWT functionality
 - Reimplemented many higher-level AWT classes
 - Customizable look-and-feel
 - Very extensible, feature-rich API
 - A bit slower than AWT, since it's "Pure Java"
-

Swing Classes

- Most Swing classes are in `javax.swing` package (and some sub-packages)
 - Quite a few AWT classes are used by Swing!
 - Events, event-handlers, geometry, images, drag-and-drop, etc.
 - Swing UI widgets derive from **JComponent**
 - Represents *any* UI component in Swing
 - **JComponent** derives from `java.awt.Container`
 - Custom Swing components can also use **JComponent** as their parent class
-

Heavyweight Components

- AWT UI components are “heavyweight”
 - Each component has its own native graphics resources
 - Components don’t use “pure Java” code to draw their graphics
 - Actually use operating-system calls
 - Overlapping components overwrite each other
-

Lightweight Components

- Swing UI components are “lightweight”
 - Components use only Java to draw themselves
 - Native graphics resources are shared by Swing components, as much as possible
 - Example:
 - A popup menu fully within an app’s window is drawn using that window’s resources
 - A popup menu extending outside an app’s window will get its own window
 - Swing can provide transparent regions more easily, since components share graphics resources
-

Mixing AWT and Swing

- Lightweight and heavyweight components don't mix well!
 - Heavyweight components are *always* drawn on top of lightweight components.
 - Avoid mixing Swing UI components and AWT components if possible
-

Windows and Containers

- **JWindow** represents simple windows
 - ...but no title bar, menus, min/max/close buttons!
 - **JFrame** represents application windows
 - Complete with title bar, menus, window-buttons
 - Typically use this for Java GUI applications
 - **JPanel** groups together UI components
 - A lightweight, general purpose container
 - Great for building up structure in your GUI!
 - Use **add (. . .)** method to add child-components
 - Child-components can also be containers, e.g. **JPanel**
-

Laying Out Components

- Containers position/size child-components with layout managers
 - Specified with `setLayout (LayoutManager lm)`
 - `java.awt.LayoutManager` is an interface
 - Many different layout managers
 - **FlowLayout** – arranges components line-by-line; wraps to next line when current line is full
 - **BoxLayout** – arranges components in a single row or column
 - **BorderLayout** – can place a component in one of five regions: **NORTH**, **SOUTH**, **EAST**, **WEST**, and **CENTER**
 - **GridLayout** – arranges components in a fixed-size 2D grid
 - **GridBagLayout** – very sophisticated layout manager
 - And several more! (See implementers of `LayoutManager...`)
 - ★ Default layout manager is **FlowLayout**
-

Events and Listeners

- When something happens, UI widgets fire events
 - ❑ User clicks mouse on something
 - ❑ User presses some keys
 - ❑ Window is closed or minimized
 - ❑ User moves or drags mouse
 - ❑ etc.
 - To catch events, must implement event-listeners in your program
 - ❑ Listeners are exposed as interfaces to implement
 - ❑ Contained in `java.awt.event` package
 - ❑ Typically named `...Listener`
-

ActionListener Interface

- Example: `java.awt.event.ActionListener`
 - One method to implement:

```
void actionPerformed(ActionEvent e)
```
 - `ActionEvent` contains details of what happened
 - What UI component reported the event
 - When the event occurred
 - Any modifier keys (Ctrl, Alt, Shift, etc.)
 - Other things too! (See API docs...)
 - `ActionEvent` is reported by most Swing components
-

Implementing **ActionListener**

- Swing components provide a registration method:

```
addActionListener(ActionListener l)
```

- Implement **ActionListener**:

```
public class ActionHandler implements ActionListener {  
    ...  
    public void actionPerformed(ActionEvent e) {  
        ... // Do something clever.  
    }  
}
```

- Register your listener:

```
ActionHandler handler = new ActionHandler();  
JButton button = new JButton("Start");  
button.addActionListener(handler);
```

Other AWT/Swing Listener Interfaces

- **MouseListener** – mouse enter/exit/click events
 - **MouseMotionListener** – mouse move/drag events
 - **KeyListener** – keyboard press/release events
 - **FocusListener** – component gets/loses focus
 - **ComponentListener** – component shown, hidden, resized
 - **WindowListener** – window opened, closed, maximized, minimized
-

Inner Classes and Event Listeners

- Can declare a class within a class
 - Called an “inner class”
 - Inner-class objects can access members of the outer class!
 - Inner classes are great for event-listeners
 - Listeners need to access application state
 - Don't want to clutter up the application class with many public interface-methods
 - All listeners are interfaces, and interface methods are public...
-

Event Handler, Inner Class Style

```
public class MyApp {
    /** Current state of application. */
    private boolean started;

    /** Handler for ActionEvents. */
    private class ActionHandler implements ActionListener {
        public void actionPerformed(ActionEvent e) {
            started = true;
        }
    }

    ...
    void initUI() {
        // Create a button, then use inner class to handle events.
        JButton button = new JButton("Start");
        button.addActionListener(new ActionHandler());
    }
}
```

Inner
Class!

Arrays in Java

- Arrays are also objects
 - Some different syntax though!

- Example:

```
int[] myInts = new int[10]; // Allocate the array.
for (int i = 0; i < myInts.length; i++) {
    myInts[i] = 100 * i; // Store stuff in it.
}
```

- In Java, *all* arrays are dynamically allocated
 - Elements are accessed with brackets (like C/C++)
 - Arrays expose a **length** field, indicating their size
 - **length** is read-only (of course)
-

Array Variables

- Array-types have brackets after *type*, not after variable name
 - ❑ `String[] names;` vs. `String names[];`
 - ❑ Latter form is supported, but is discouraged.
 - Can declare array-variables without assigning
 - ❑ `boolean[] flags; // Array of boolean values`
 - ❑ `float[] weights; // Array of floats`
 - Must initialize them before using
 - ❑ Can allocate new array with `new type[size];`
 - *size* can be zero! Called an “empty array.”
 - ❑ Can assign an existing array to the variable
 - (Java arrays are basically objects with additional syntax)
 - ❑ Can set to `null` too!
-

More Array Initialization

- Can also assign specific values to arrays

```
String[] colorNames = {  
    "puce", "mauve", "fuchsia", "chartreuse", "umber"  
};  
// colorNames.length == 5
```

- Syntactic sugar for the initialization operations
- Can still reassign and reinitialize such arrays
 - `colorNames` is a reference to an array of `String` objects



Arrays of Objects

- Arrays of objects initially contain `null` values
 - Array initialization does not initialize object-references
 - Must do that in a separate step

- Example:

```
// Allocate an array of 20 point-references
Point2d[] points = new Point2d[20];

// Make a new Point2d object for each elem
for (int i = 0; i < points.length; i++)
    points[i] = new Point2d();
```

Arrays of Arrays

- Arrays can contain other arrays

```
int[][] nums2d; // Array of arrays of ints.
```

- First the array-of-arrays is allocated:

```
nums2d = new int[20][];
```

- Each element of `nums2d` is of type `int[]`.

- Next, each inner array is allocated

```
for (int i = 0; i < nums2d.length; i++)  
    nums2d[i] = new int[50];
```

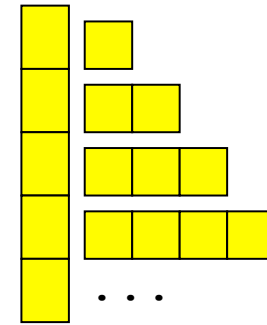
- When size is fixed, Java provides a shortcut

```
int[][] nums2d = new int[20][50]; // Same thing!
```

More Arrays of Arrays

- Inner arrays can be different sizes, if need be

```
int[][] reducedMatrix;  
reducedMatrix = new int[20][];  
for (int i = 1; i <= 20; i++)  
    reducedMatrix[i] = new int[i];
```



- Can't do this with the shortcut syntax

- Can also specify nested initial values

```
double[][] weights = {  
    {3.1, 2.6}, {1.5, 4.4, -3.6}, null, {6.2}  
};
```

Copying Arrays

- Use `System.arraycopy()` to copy one array to another efficiently
 - Can use `clone()` method to duplicate array
 - Result's type is `Object`; must cast to proper type

```
int[] nums = new int[35];  
...  
int[] numsCopy = (int[]) nums.clone();
```
 - Copy is shallow – only top-level array is copied!
 - If array of objects, the objects are not cloned
 - If array of arrays, subarrays are not cloned either
-

This Week's Homework

- First step towards implementation of Conway's Game of Life
 - Implement a Swing UI with a 10×10 grid of buttons
 - Buttons have two states
 - User can toggle button-states by clicking in the grid
 - Another button toggles state of entire button grid, all at once
 - Practice using Swing, interfaces and arrays
 - ...and maybe even inner classes! 😊
-

Next Week

- Introduction to Java Threads
 - Swing and threads
 - Useful advice for Lab 5!
-