CS11 – Java

Spring 2011-2012
Lecture 3
Today’s Topics

- Class inheritance
- Abstract classes
- Polymorphism
- Introduction to Swing API and event-handling
- Nested and inner classes
Class Inheritance

- A third of the “four big OOP concepts”
- A class can extend another class to build on its functionality
- Terminology:
  - Parent class, or superclass, or base class
  - Child class, or subclass, or derived class
- Child classes inherit all methods and fields within parent class
  - Can add new functionality
  - Can also override parent-class methods
Class Inheritance (2)

- Class inheritance models an “is-a” relationship
  - Example class hierarchy:
    - Vehicle
      - Wheeled Vehicle
        - Dump Truck
      - Water Vehicle
        - Sailboat
        - Barge
  - The child class is a specialization of the parent class
  - Child class also has characteristics of parent class
    - Can treat child class as if it were any parent-type
      - “A dump truck is a wheeled vehicle.”
      - “A sailboat is a vehicle.”
      - “A water vehicle is a vehicle.”
Example class hierarchy:

- Vehicle
  - Wheeled Vehicle
    - Dump Truck
  - Water Vehicle
    - Sailboat
    - Barge

Sibling types *do not* model an “is-a” relationship!

- These statements are clearly false:
  - “A dump truck is a water vehicle.”
  - “A wheeled vehicle is a barge.”

What about these statements?

- “A vehicle is a dump truck.”
- “A water vehicle is a sailboat.”

- Depends on the actual vehicle being considered!
  - Need to examine a specific vehicle to verify the statement
Example Class Hierarchy

- The number classes in Java
  ```java
  java.lang.Object
  \n  \n  java.lang.Number
  \n  \n  java.lang.Integer
  ```
  - `Integer` "is a" `Number`, "is an" `Object`
  - `Integer` extends `Number`, which extends `Object`
  - `Integer` inherits all methods that `Object` defines
    - boolean equals(Object o)
    - int hashCode()
    - String toString()
    - Class getClass()
  - `Integer` also overrides some of these methods
Overriding `Object.toString()`

- Really useful idea, especially for debugging
- Used in string concatenation
  - You type this:
    ```java
    String msg = "Point is " + pt;
    ```
  - Compiler automatically does this:
    ```java
    String msg = "Point is " + pt.toString();
    ```
- Simple to define:
  ```java
  @Override
  public String toString() {
      return "(" + xCoord + "," + yCoord + ")";
  }
  ```
Classes and Objects

- A class’ parent-class methods can be called without any special syntax.
  
  ```java
  Integer intObj = new Integer(53);
  ...
  
  Class c = intObj.getClass(); // Get type info
  ```

- Integer is also an Object – can call methods declared and/or implemented on Object.

- Child class can also provide its own methods
  
  ```java
  System.out.println("Value is " + intObj.intValue());
  ```

- Integer extends Object’s functionality
  
  - intValue() returns an int version of the Integer
Reference Types

- Every reference has a class-type associated with it
  
  ```java
  Object obj; // A reference of type Object
  Integer val; // A reference of type Integer
  
  The variable's type dictates what is accessible
  
  Example:
  ```
  ```java
  Object obj = new Integer(38);
  ...
  System.out.println(obj.intValue());  // COMPILATION ERROR
  
  Compile error, because Object doesn’t define intValue()
  ```

  intValue() is declared in Number class (parent of Integer)

  Even though obj refers to an Integer object, only the Object methods are visible
Navigating the Hierarchy

- Number hierarchy is like this:

- Moving **down** the hierarchy requires a run-time test.
  
  ```java
  Object obj = new Integer(453);
  ...
  int i = ((Integer) obj).intValue(); // Cast obj
  ```

- You could also try this:
  ```java
  float f = ((Float) obj).floatValue(); // Runtime error
  ```
- This code compiles, but it will report an error at runtime
- Java can’t assume the actual object-type at compile time!
- (Even when it’s obvious to a human…)
- So, we have a runtime type-check, and a potential error.
What Child Classes Don’t Get

- Child classes cannot access **private** members in parent classes

- **protected** access-modifier allows the child class to access parent-class’ members
  - Only available within the class, and to subclasses
  - Looser than **private**, but still not **public**!

- Child classes also don’t inherit **static** fields and methods
  - They can be accessed, but they are not inherited
public class Task {
    private String name;
    private boolean done;

    public Task(String taskName) {
        name = taskName;
        done = false;
    }

    /** Just record that the task is done. */
    public void doTask() {
        done = true;
    }

    /** Report if the task is done or not. */
    public boolean isDone() {
        return done;
    }
}
Making Useful Tasks

- Our **Task** class is *very* generic…
  - …so generic that it’s nearly useless!
- **Extend** **Task** class to provide useful tasks
  ```java
generic class FileUploadTask extends Task {
  public FileUploadTask() {
    // Call parent-class constructor
    super("upload file");
  }
}
```
  - Parent-class constructors are *not* inherited!
  - If parent class doesn’t have a default constructor, we must *explicitly* call one in the child class, using **super** keyword
Overriding Parent-Class Methods

- FileUploadTask should provide its own implementation of doTask()
  ```java
  public class FileUploadTask extends Task {
    ...
    /** Perform the file-upload operation. */
    @Override
    public void doTask() {
      ... // Open a connection, read a file, etc.
    }
  }
  ```
  - Method’s signature is same as parent-class’ method signature
  - This overrides Task’s implementation of doTask()
Polymorphism

- Now we want to upload a file:
  ```java
  Task t = new FileUploadTask();
  t.doTask();
  ```
  - Which implementation of `doTask()` does this call?

- In Java, all instance-methods are **virtual**
  - Even though `t` is a `Task` reference, the `FileUploadTask` implementation is called
  - Reason: `t` refers to an object of type `FileUploadTask`

- This is called **polymorphism**
  - The fourth “Big OOP Concept”
  - A statement’s behavior changes, depending on the type of the objects involved
Problem:
- `FileUploadTask.doTask()` doesn’t set `done` to `true`
- Also, `done` is private!

One solution:
- `FileUploadTask.doTask()` implementation can call the parent-class implementation:

  ```java
  /** Perform the file-upload operation. */
  @Override
  public void doTask() {
      ...
      // Open a connection, read file, etc.

      // All done!
      super.doTask();
  }
  ```
The **Task** Abstraction

- Actually doesn’t make sense for **Task** to have an implementation of **doTask()**
  - Change **Task** to be an **abstract** class
  - An abstract class declares a set of behaviors, but only *partially* defines it.
- Abstract classes cannot be instantiated
  - Child classes must be provided, that implement the missing functionality
  - Example: **FileUploadTask** must provide an implementation of **doTask()**, that uploads a file.
The New, Abstract Task Class

- Our abstract Task class:
  ```java
  // A class that represents a generic task
  public abstract class Task {
    private String name;
    private boolean done;

    public Task(String taskName) {
      name = taskName;
      done = false;
    }

    // Child classes implement this method.
    public abstract void doTask();

    ... // Rest of class
  }

  Abstract classes can still have fields and non-abstract methods
The New **FileUploadTask**

- **FileUploadTask** doesn’t “override” `doTask()`
  - There’s nothing to override!
  - **FileUploadTask** *implements* `doTask()`

- Again, the signatures must match up
  ```java
  /** Implement doTask() to upload a file. */
  public void doTask() {
      ... // Open a connection, read the file, etc.
  }
  ```
  - (Without the `abstract` modifier, of course!)
  - Of course, we can’t do `super.doTask()` anymore

- Child class *must* provide an implementation of every abstract parent-class method
  - If not, child class must also be declared abstract.
Completing the Abstraction

- How can a task be marked as done?
  - A simple solution: set `done` to be protected
- Another good solution:
  - `Task` can provide another protected method to do this:
    ```java
    protected void reportTaskDone() {
        if (done) {
            ... // Task was already done! Complain.
        }
        done = true;
    }
    ```
  - Now only child classes can report that the task is done
- Which solution is more extensible?
  - Might want to add other processing when a task is finished
  - Can easily add this to `reportTaskDone()` later
You can’t instantiate the abstract `Task` class

\[
\text{Task } t = \text{new Task("send e-mail");} \quad \text{COMPILE ERROR}
\]

- The implementation of `Task` is incomplete!

You can have a `Task`-reference

\[
\begin{align*}
\text{Task } t & = \text{new FileUploadTask();} \\
\text{t.doTask();} & \quad \text{// Calls FileUploadTask.doTask()} \\
\text{t} & = \text{new SendEMailTask();} \\
\text{t.doTask();} & \quad \text{// Calls SendEMailTask.doTask()}
\end{align*}
\]

- The correct implementation of `doTask()` gets called because of polymorphism

APIs are made generic by using the base-class type

\[
\text{void enqueueTask(Task t) \{} \\
\quad \text{pendingList.store(t);} \\
\}\n\]
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
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
  - Call `setLayout(LayoutManager lm)` on the container
  - `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 `[Something]Listener`
**ActionListener Interface**

- **Example:** `java.awt.event.ActionListener`
  - One method to implement:
    ```java
    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:
  ```java
  addActionListener(ActionListener l)
  ```

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

- Register your listener:
  ```java
  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
Listeners and Adapters

- Some listeners are more complicated:
  - **MouseListener** interface specifies these methods:
    - `mouseEntered()`, `mouseExited()`
    - `mousePressed()`, `mouseReleased()`
    - `mouseClicked()`
  - Frequently only want to implement one or two of these…
  - Java often provides adapters for event-listener interfaces
- Example: `java.awt.event.MouseAdapter`
  - Implements **MouseListener** interface, among others
  - All provided implementations are no-ops
  - Derive your event-handler from **MouseAdapter**, and then override just the methods you want to implement
Nested Classes in Java

- Can declare a class within a class
  - Called a *nested class*
    ```java
    class Outer {
        /* A nested class */
        class Inner {
            ...
        }
    }
    ```
  - When `Outer.java` is compiled, compiler generates two files: `Outer.class` and `Outer$Inner.class`
The nested class is a member of the outer class, and can have an access modifier
- e.g. a private nested class cannot be referred to directly from outside the outer class

The nested class can also be declared with or without the `static` keyword
- Has some dramatic impacts on how the nested class can be used, and what it can do!

class Outer {
    static class StaticNested { ... }
    class NonStaticNested { ... }
}
Static Nested Classes

- Static nested classes are simply related classes “contained within” the outer class.
- Example: `java.awt.geom.Rectangle2D`:
  - An abstract class that represents 2D rectangles.
- Contains two static nested classes:
  - `Rectangle2D.Double` derives from `Rectangle2D`, and specifies coordinates of type `double`.
  - `Rectangle2D.Float` is similar, but `float` coords.
- To use:
  - `import java.awt.geom.Rectangle2D;`
  - Refer to nested classes by `Rectangle2D.Float` or `Rectangle2D.Double`. 
Non-static Nested Classes

- Non-static nested classes are also called **inner classes**
- Like instance methods, inner classes *must be* used in the context of a containing object!
  - They actually reference their containing object
  - They can directly access the containing object’s fields and methods
- *Cannot* create inner-class objects in a static method on the outer class!
  - Can only create in instance methods
Inner Classes and Event Listeners

- Inner classes are *great* for event-listeners!
  - Listeners often need to access application state
  - Inner class can even access private members of the outer class
- Also keeps outer class’ public interface clean
  - Don’t want to have a whole bunch of public listener interface-methods exposed on outer class
- When necessary, can also create multiple inner-class objects associated with a single outer-class object
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 button, then use inner class to handle events
        JButton button = new JButton("Start");
        button.addActionListener(new ActionHandler());
    }
}