CS193J: Programming in Java
Summer Quarter 2003

Lecture 4
OOP Inheritance, Abstract classes, Interfaces

Manu Kumar
sneaker@stanford.edu
Recap

• Last time
  – Java Collections
    • Iterators
    • ArrayList example
  – OOP
    • Inheritance
      – Overriding
      – Polymorphism
      – “Pop-down” rule
    • Grad example

• To Dos
  – HW1: Pencil Me In
    • Due before midnight Wednesday July 9th, 2003
Handouts

• 1 Handouts for today!
  – #11: Drawing in Java
Today

• Continue with OOP/Inheritance
  – Pop-down rule
  – Constructors
  – instanceof
  – Grad example

• Abstract superclasses
  – Account example

• Java Interfaces
  – Moodable example

• Today or next time
  – Start Drawing/GUI
“Pop-Down” rule

- The receiver knows its class.
- The flow of control jumps around different classes.
- No matter where the code is executing, the receiver knows its class and does the message→method mapping correctly for each message.

Example
- Receiver is the subclass (Grad), executing a method in the superclass (Student).
- A message send that Grad overrides will “pop-down” to the Grad definition as in the case of getStress().
The “super” keyword is used in methods and constructors to refer to code in the superclass

- Calling `super.getStress()` in the Grad class would execute the code for `getStress()` in the Student Class
- Think of `super` as a directive to the message → method resolution process.
  - Start searching one level higher.

- Allows the subclass to not have to rewrite the code
  - Re-use the code in the superclass and add to the functionality
Subclass Constructor

- Subclass needs a constructor
  - Should take arguments for the superclass and the class itself
  - Needs to pass on the arguments for the superclass to the constructor for the superclass
    - Done by called using a special syntax: `super(…)` in the first line of the constructor

- Note:
  - If no superclass constructor is specified, the default constructor will be called

- Every class needs its own constructors with the arguments spelled out
  - In a way constructors are not inherited and must be spelled out
Multiple constructors (this())

- A class can have multiple constructors with differing parameters
  - Often used to provide a default constructor which uses default arguments
- Can re-use the code for the constructors by using this(…)
- Example:
  ```java
  public Grad() {
    this(10, 0);
  }
  public Grad(int units, int yot) {
    ...
  }
  ```
• Special operator which may be used to check the runtime type of a pointer

• Example
  – if (x instanceof Grad) {...}

• Using instanceof with a null returns false

• Note:
  – Using instanceof is generally an indication of a design flaw
  – Use sparingly, only when it is really warranted (for example in dynamic class loading)
• Complete code included in handout

• Walk through of the code…
Using Inheritance

• Most common style:
  – Have a superclass with given features
  – Need a class which has most of the features, but is more constrained or slightly different
  – Appropriate time to subclass and use inheritance/overriding to reuse code.

• Working with library code
  – Subclass off a library class
  – Inherit 90% of the standard behavior
  – Override a few key methods for the rest
• OOP
  - Encapsulation / Modularity
  - Client Oriented Design
  - Inheritance
    • Polymorphism
• Abstract Superclass
  - Factor common code up
  - Example
    • AbstractCollection class in Java libraries
    • Account example that we will be doing (coming up!)
Abstract Method

• Can apply the “abstract” keyword to any method
  – public abstract void mustImplement();
  – Note: no {} and no code!

• Abstract method
  – Defines name and arguments
  – No implementation!
  – Implementation MUST be provided in the subclass!
Abstract Class

- Can apply the “abstract” keyword to a class
  - public abstract class Account { …

- A class that has one or more abstract methods is abstract

- Abstract classes can NOT be instantiated
  - Cannot do: new Account()
  - Only subclasses can be instantiated

- Used to factor out common code!
Abstract Super Class

- A common superclass for several subclasses
- Factor up common behavior
- Define the methods all the subclasses respond to
- Methods that subclasses should implement are declared abstract
- Instances of the subclasses are created, not of the superclass
Clever Factoring Style

• Common Superclass
  – Factor common behavior up to the superclass
  – Superclass sends itself messages to invoke various parts of the behavior
    • Will rely on the “pop-down” behavior to work correctly!

• Special subclasses
  – As short as possible
  – Rely on the superclass for common behavior
  – Override key methods to customize behavior with minimal code
    • May use super.foo()
  – Rely on pop-down behavior to do the right thing!

• Example
  – JComponent in the Java Swing library
    • We will get into this later
Problem details:
- You need to store information for bank accounts
- Assume that you only need to store the current balance, and the total number of transactions for each account.
- The goal for the problem is to avoid duplicating code between the three types of account.
- An account needs to respond to the following messages:
  - `constructor(initialBalance)`
  - `deposit(amount)`
  - `withdraw(amount)`
  - `endMonth()`
- Apply the end-of-month charge, print out a summary, zero the transaction count.
Account Example

• Types of Accounts
  – Normal
    • Fixed $5.0 fee at the end of the month
  – Nickle ‘n Dime
    • $0.50 fee for each withdrawal charged at the end of the month
  – Gambler
    • With probability 0.49 there is no fee
    • With probability 0.51 the fee is twice the amount withdrawn
Design process

• Factoring
  – Put common behavior in one place
  – Subclasses are used to implement the specific deviation from the common behavior

• Abstract methods
  – Provide prototypes for Abstract Methods to be implemented by subclasses
Class Design Diagram

Account
  *balance
  *transactions
  -deposit
  -withdraw
  -endMonth
  -endMonthCharge (abstract)

Fee
  -endMonthCharge

NickleNDime
  *withdrawCount
  -withdraw
  -endMonthCharge

Gambler
  -withdraw
  -endMonthCharge
Account Code walk through

• Complete code is included in your handout

• Code walk through…
Account example: Points of note

• Gambler.withdraw() uses super.withdraw() to decrement balance
• Account.endMonth() does a popdown by sending itself the endMonthCharge() message
• Account.main() uses polymorphism
  – The right method gets called
  – Pop-down to the right implementation of withdraw depending upon the runtime type of the receiver.
Java Interfaces

• Java does not support multiple inheritance
  – This is often problematic
    • What if we want an object to be multiple things?

• Interfaces
  – A special type of class which
    • Defines a set of method prototypes
    • Does not provide the implementation for the prototypes
    • Can also define final constants
Java Interfaces

• A Class
  – Can “extend” only one class i.e. only one superclass
  – Can “implement” multiple interfaces!

• Class Server implements Pingable
  – Server is a class
  – It implement the Pingable interface
  – Server MUST provide implementations for all the method prototypes in the Pingable interface
  – The Server Object can serve as a substitute wherever we want a Pingable Object.
    • Similar to a superclass
Java interfaces

• Lightweight
  – Allow multiple classes to respond to a common set of messages but without the implementation complexity.

• Similar to Subclassing but…
  – Good news
    • Class has only one superclass
    • Can implement multiple interfaces
  – Bad news:
    • Interface only gives the method definition and not the implementation
Interface Example

• Special keyword ‘interface’
• Similar to defining a class, but instead use the keyword interface
• Methods are empty (no { and } or code)
• Example

```java
public interface Moodable {
    public Color getMood();
    // interface defines getMood() prototype
    // but no code
}
```
Implementing an Interface

• “implements” keyword
  – Similar to extend, but followed by a comma separated list

• Example
  public class Student implements Moodable {
    public Color getMood() {
      if (getStress()>100) return(Color.red);
      else return(Color.green);
    }
    // rest of Student class stuff as before...
• Moodable is like an additional superclass of Student
  – It is possible to store a pointer to a Student in a pointer of type Moodable

• Example
  
  Student s = new Student(10);
  Moodable m = s; // Moodable can point to a Student
  m.getMood(); // this works

• We will see more of this later…
• You now know
  – Basic Java language constructs
  – OOP principles
  – OOP in Java

• Next
  – Drawing in Java
    • Java Swing
    • JComponent/Drawing
    • LayoutManagers
Java GUI on Screen

- How do you put a GUI on the screen?
  - Create a window (aka Frame) object
  - Install components
    - Labels, buttons, etc
  - System manages the window and components by sending notification for user events
    - Drawing clicking typing
  - Components draw themselves
• OOP drawing vs. 106 drawing
  – 106:
    • Just start drawing when you want and the pixels show up
    • Requires re-inventing the wheel each time!
  – OOP
    • Build on a framework of GUI Classes
      – Collection of GUI elements
    • Object which correspond to visual elements
      – Anthropomorphemic – draw themselves
    • Send messages in order to have different results on the screen
• Library Class Hierarchy
  – Extensive, pre-built inheritance hierarchy of classes for common problems
    • Drawing, controls, windows, scrolling
  – Engineered to work together
    • But that also means there is a slight learning curve

• System: Event → Notifications
  – Background task ("System") manages bookkeeping and orchestration of windows and events
  – “User Events” – clicking, typing etc happen in realtime
  – System manages an “event queue”
OOP GUI Programming Tasks

• Instantiate library classes (EASY)
  – Simply requires reading the API documentation and some understanding of their design

• Subclass library classes (HARD)
  – Used to introduce custom behavior
    • Inherit, override
  – Requires deeper understanding of the superclass
  – Relies on “pop-down” feature of OOP
  – Example:
    • Subclass JComponent and override paintComponent() to provide drawing code
    • Subclass JButton so it beeps on being clicked
Java AWT

• Abstract Windowing Toolkit
  – Included in first release of Java
  – Plagued with implementation problems
  – Native peers
    • Used wrapper classes for native GUI components of the operating system
    • Advantage
      – Same look and feel as on the native platform
    • Disadvantage
      – Hard to implement reliably
      – Consistency issues across platforms
• Replacement/Enhancement for AWT
  – aka Java Foundation Classes
  – Implemented in Java
    • rt.jar contains classes for Swing
    • Same on all platforms
  – Build on AWT primitives
  – 10x more classes, depth and functionality
  – Pluggable look and feel
    • Interface can look like the native platform
    • Dynamically switchable look and feel
Java GUI Block Diagram

Operating System + its native GUI

Java VM

AWT

Swing
Java GUI Themes

• We will be using Swing
  – AWT still used in limited way

• Themes
  – Things draw themselves when sent the right messages
    • Anthropomorphic Objects
  – Layout Manager
    • Used to arrange the size and position of components on the screen
    • We will see more of this soon
Introduction to Swing classes

- **JComponent**
  - Swing analog of the Object class
  - Everything inherits from JComponent
  - Defines the basic notions of geometry

- **JLabel**
  - Built in JComponents that displays text
  - Example: `new JLabel("Hello World!");`

- **JFrame**
  - A single window
  - Has a “content pane” JComponent that can hold other components
    - `frame.getContentPane()`
  - Closing a frame simply hides it
Content Pane / Layout Manager

• Content pane is a place holder
  – An empty board where you can place components
  – Use add() to put components on the content pane

• Content pane uses a “Layout Manager”
  – Programmer provides guidelines for how the interface should look by choosing the correct layout manager
  – LayoutManager determines the size and positioning of components on the contentpane
FirstFrame example

Hello World. Another Label.
Klaatu Barada Nikto!

Ok
// FirstFrame.java
/*
   Demonstrates bringing up a frame with some labels.
*/
import java.awt.*;
import javax.swing.*;
import java.util.*;
import java.awt.event.*;
public class FirstFrame extends JFrame {
   public FirstFrame(String title) {
      super(title); // superclass ctor takes frame title
   }
   // Get content pane -- contents of the window
   JComponent content = (JComponent) getContentPane();
// Set to use the "flow" layout
// (controls the arrangement of the components in the content)
content.setLayout(new FlowLayout());

// Background color is a property of all components --
// set it to white
content.setBackground(Color.lightGray);

// Use add() to install components
content.add(new JLabel("Hello World."));
content.add(new JLabel("Another Label."));
content.add(new JLabel("Klaatu Barada Nikto!"));
content.add(new JButton("Ok"));
// Force the frame to size/layout its components
cpack();

setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
// Java 1.3 or later
setVisible(true);  // make it show up on screen

public static void main(String[] args) {
  new FirstFrame("First Frame");
}
}
Summary

• Today
  – Continue with OOP/Inheritance
    • Pop-down rule
    • Constructors
    • instanceof
    • Grad example
  – Abstract superclasses
    • Account example
  – Java Interfaces
    • Moodable example
  – Drawing in Java started (maybe)

• Assigned Work Reminder:
  – HW #1: Pencil Me In
    • Due before midnight Wednesday, July 9th, 2003