CS193J: Programming in Java
Summer Quarter 2003

Lecture 3
Collections and More OOP

Manu Kumar
sneaker@stanford.edu
Recap

• Last time (a somewhat jumpy introduction to…)
  – OOP in Java (Student Example)
  – Explore more Java features
    • Primitives
    • Arrays
    • Multi-Dimensional Arrays
    • String Class
    • StringBuffer Class
    • Static keyword
  – OOP Design
    • Encapsulation
      – Interface vs. Implementation
    • Client Oriented Design

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

• 3 Handouts for today!
  – #8: Collections
  – #9: OOP2 - Inheritance
  – #10: OOP3 – Abstract Superclasses

• Next time
  – Complete OOP
    • Probably won’t get to all of it today
  – Start Drawing/GUI
Today

- Java Collections
  - ArrayList example
- OOP
  - Inheritance
    - Grad example
  - Abstract Superclasses
    - Account example
- Java Interfaces
- You will have all you need for HW#1
  - By the end of today’s lecture.
• Built-in support for collections
  – Similar to STL in C++
• Collection type
  – Sequence/Set
  – Example ArrayList
• Map type
  – Hashtable/dictionary
  – Example HashMap
• Collections store pointers to objects!
• Use inheritance and interfaces
• Read
  – http://java.sun.com/docs/books/tutorial/collections
• All classes implement a similar interface
  – add(), size(), iterator()…
  – Easy learning curve for using Collections
  – Possible to swap out the underlying implementation without significant code change
• Implemented as pointer to Object
  – Similar to using a void * in C
  – Require a cast back to the actual type
  – Example
    • String element = (String)arraylist.get(i)
• Java checks all casts at run-time
Collection Messages

- **Basic messages**
  - `constructor()`
    - Creates a collection with no elements
  - `size()`
    - Number of elements in the collection
  - `boolean add()`
    - Add a new pointer/element at the end of the collection
    - Returns true if the collection is modified.
  - `iterator()`
    - Returns an Iterator Object
• Utilities
  – Additional useful methods
  – boolean isEmpty()
  – boolean contains(Object o)
    • Iterative search, uses equals()
  – boolean remove(Object o)
    • Iterative remove(), uses equals()
  – Boolean addAll(Collection c)
Iterators

• Used to iterate through a collection
  – Abstracts away the underlying details of the implementation
    • Iterating through an array is the same as a binary tree
  • Responds to
    – hasNext() - Returns true if more elements
    – next() - Returns the next element
    – remove() - removes element returned by previous next() call.
Working with Iterators

• Not valid to modify a collection directly while an iterator is being used!
  – Should not call collection.add() or collection.remove()

• OK to modify the collection using the iterator itself
  – iterator.remove()

• Why?
  – Motivation for concurrency issues later in the course
ArrayList

- Most useful collection
- Replaces the “Vector” class
- Can grow over time
- Methods
  - add()
  - int size()
  - Object get(int index)
    - Index is from 0 to size() -1
    - Must cast to appropriate type when used.
  - iterator()
    - We’ll see an example!
import java.util.*;

/*
The ArrayList is replaces the old Vector class. ArrayList implements the Collection interface, and also the more powerful List interface features as well. Main methods: add(), size(), get(i), iterator()
See the "Collection" and "List" interfaces.
*/

public static void demoArrayList() {
    ArrayList strings = new ArrayList();

    ...
}
// add things...
for (int i = 0; i < 10; i++) {
    // Make a String object out of the int
    String numString = Integer.toString(i);
    strings.add(numString);  // add pointer to collection
}

// access the length
System.out.println("size:" + strings.size());
// ArrayList supports a for-loop access style...
// (the more general Collection does not support this)

for (int i=0; i<strings.size(); i++) {
    String string = (String) strings.get(i);
    // Note: cast the pointer to its actual class
    System.out.println(string);
}
ArrayList: iterating

// ArrayList also supports the "iterator" style...

Iterator it = strings.iterator();
while (it.hasNext()) {
    String string = (String) it.next(); // get and cast pointer
    System.out.println(string);
}

// Calling toString()
System.out.println("to string:" + strings.toString());
ArrayList Demo: removing

// Iterate through and remove elements
// get a new iterator (at the beginning again)

    it = strings.iterator();
    while (it.hasNext()) {
        it.next(); // get pointer to elem
        it.remove(); // remove the above elem
    }

    System.out.println("size:" + strings.size());
/* Output...
   size:10
   0
   1
   2
   3
   4
   5
   6
   7
   8
   9
   to string:[0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
   size:0
*/
• OOP so far
  – Modularity
  – Encapsulation
• Today
  – Open Pandora’s box
    • Inheritance
    • Abstract Super Classes
• Warning
  – True good uses of inheritance are rare
  – Use it only where it is really appropriate.
Hierarchy

• Classes are arranged in a tree hierarchy
  – A class’ “superclass” is the class above it in the hierarchy
  – Classes below is are “subclasses”

• Classes have all the properties of their superclasses
  – General – towards the root (top)
  – More specific – towards the leaves (down)
    • NB: In Computer Science trees grow upside down!
Example

```
Object
   /
  /  
Animal
   /
  /   
Bird
   /
  /    
Duck
```
Inheritance

• The process by which a class inherits the properties of its superclasses
  – Methods
  – Instance variables
• Message-Method resolution revisited
  – Receive message, check for method in class
  – If found, execute
  – Check for method in superclass
  – If found, execute, if not, repeat this procedure
  – Basic idea: Travel up the tree.

• Result:
  – A class automatically responds to all the messages and has all the storage of superclasses
• When an object receives a message
  – It checks its own methods first
  – Then check the superclass’ methods
• The first method in the hierarchy takes precedence
  – We can add a method with the same name as in the superclass in the class
  – The code of the superclass will not be executed
    • It is effectively “overridden” i.e. intercepted
• In C++ runtime overriding is an option invoked by the “virtual” keyword.
Polymorphism

• Do not be intimidated by the big word
  – It’s a simple concept
  – Basic idea: “it does the right thing”

• An Object always knows its true class at runtime
  – The MOST specific method found for the object is executed.
Polymorphism example

• Shape is a superclass of Rectangle
  – Shape.drawSelf() and Rectangle.drawSelf()

• Code:
  ```java
  Shape s;
  Rectangle r = new Rectangle();
  s = r;
  s.drawSelf()
  ```

• Which method will get execute?
  Shape.drawSelf() or Rectangle.drawSelf()?
- OOP Class Hierarchy
- Superclass
- Subclass
- Inheritance
- Overriding
- isA
  - the subclass is a instance of the superclass
Horse/Zebra Example

- Hierarchy of all the animals
  - Need to add in Zebra

- Options
  - Define zebra from scratch
    - Bad idea – multiple copies of code.
  - Locate the Horse class and subclass it to create the Zebra class
    - Zebra will inherit most of the characteristics of a horse
    - Override or add additional features as needed
Grad Example

• Add Grad as an extension to the Student class from previous lecture.
  – yearsOnThesis – a count of the number of years worked on the thesis
  – getStress() – overridden to be different
    • 2 * the Student stress + yearsOnThesis

• Grad is everything that a student is
  – Has additional or some different properties

• Grad is “more specific”
  – Grad (subclass) has more properties, and is more constrained that Student (superclass)
```plaintext
Student

• units
  - ctor
  - get/set Units
  - getStress

Grad

• yearsOnThesis
  - ctor
  - get/set YOT
  - getStress (override)
```
Simple Inheritance Client Code

- **Instantiation**
  - `Student s = new Student(10);`
  - `Grad g = new Grad(10, 2); // ctor takes units and yot`

- **Usage (normal)**
  - `s.getStress(); // (100) goes to Student.getStress()`

- **Usage (inheritance)**
  - `g.getUnits(); // (10) goes to Student.getUnits()`

- **Usage (overriding)**
  - `g.getStress(); // (202) goes to Grad.getStress()`
Clarifications and Reminders

• An Object never forgets it’s Class
  – The receiver always knows it’s most specific class

• Student s; in the face of inheritance
  – No: “s points to a Student object”
  – Yes: “s points to an object that responds to all the messages that Students respond to”
  – Yes: “s points to a Student, or a subclass of Student”
A subclass *isA* superclass

- A subclass object can be used when you are expecting a superclass
- The subclass has everything the superclass has and more (not less!)

Compile time error checking

- Compiler will only allow code in which the receiver respond to the given message
- Implemented as loose checking since sometimes the exact class is not known (Student or Grad both work)

Run time error checking

- More strict. Receiver knows exactly which class it is.
A pointer to a Grad object be assigned to Student pointer.

- Student s = new Student(10);
- Grad g = new Grad(10,2);
- s = g; // ok -- subclass may be used in place of superclass

The reverse is not allowed however

- Student s = new Student(10);
- Grad g = new Grad(10,2);
- g = s; // NO, does not compile
Example of method calls

Student s = new Student(10);
Grad g = new Grad(10, 2);

s = g;  // ok

s.getStress();
// (202) ok -- goes to Grad.getStress() (overriding)

s.getUnits();
// (10) ok -- goes to Student.getUnits (inheritance)

s.getYearsOnThesis();
// NO -- does not compile (s is compile time type Student)
Downcast

• Sometimes the programmer can give the compiler more information
  – Done by providing a more specific cast around a less specific object
  – ((Grad)s).getYearsOnThesis();

• Downcast
  – Makes a more specific claim

• All casts are checked at runtime
  – Will throw ClassCastException if there is a problem
  – In C, the program would crash unpredictably

• In general: Downcasting is bad style!
Student/Grad Memory Layout

- Implementation detail
  - In memory, instance variables of the subclass are layers on top of the instance variables of the subclass

- Result
  - A pointer to the base instance of the subclass can be treated as if it were a superclass object
  - A Grad object looks like a Student object
Inheritance Client Code

Student s = new Student(10);
Grad g = new Grad(15, 2);
Student x = null;

System.out.println("s " + s.getStress());
System.out.println("g " + g.getStress());

// Note how g responds to everything s responds to
// with a combination of inheritance and overriding...
g.dropClass(3);

System.out.println("g " + g.getStress());

/*
OUTPUT...
s 100
g 302
g 242
*/
// s.getYearsOnThesis(); // NO does not compile

g.getYearsOnThesis(); // ok

// Substitution rule -- subclass may play the role of superclass
x = g; // ok

// At runtime, this goes to Grad.getStress()
// Point: message/method resolution uses the RT class of the receiver,
// not the CT class in the source code.
// This is essentially the objects-know-their-class rule at work.
x.getStress();

// g = x; // NO -- does not compile,
// substitution does not work that direction
// x.getYearsOnThesis(); // NO, does not compile

((Grad)x).getYearsOnThesis(); // insert downcast
// Ok, so long as x really does point to a Grad at runtime
• **isIrate() method in the Student Class**
  – Returns true if stress > 100

• **In the Student Class:**
  ```java
  public boolean isIrate() {
    return (getStress() > 100);
  }
  ```

• **What happens with the following client code:**
  ```java
  Student s = new Student(...);
  Grad g = new Grad(...);
  s.isIrate();
  g.isIrate();
  ```
How g.isIrate() works…

• g *known* that is a Grad Object

• It looks for an isIrate method
  – Not found, so climbs up the tree to the Student class
  – Method found in Student class

• isIrate() has a call to getStress()
  – Since g knows it is a Grad Object (*and it doesn’t forget this!* it will call the Grad.getStress() 
  – Grad.getStress() in turn calls Student.getStress()!!
    • We will see this when we examine the implementation code!

• Bottom line: it does the right thing!
“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
• Continued on Lecture 4…
• Today
  – Java Collections
    • ArrayList example
  – OOP
    • Inheritance
      – Grad example

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