CS193J: Programming in JAVA

January 20, 1999

Lecture #5

Announcements

- HW1
  - A few updates added to FAQ area of web site
    - Be sure to get corrected version of Convert.java
    - Keep an eye on FAQ over the next week
  - Don’t wait til last minute to start
    - Unfamiliar tools, new classes to learn, details to manage
- Section on Friday
  - Ajay & Vikash show small complete program, built-ins, on-line docs
- Reading
  - Ch 8 sections on abstract & final, Ch 9 sections on interfaces

Object re-use

- One purported advantage of OOP is code re-use
- At simplest level a useful and well-designed class can work in other programs, too
  - A class is nice clean package to work with and distribute
- Also can re-use by building classes out of other classes
  - Concept called "composition"
- Re-use by changing/extending to form new class
  - The magical process of "inheritance"

Composition

- Composition used extensively in OO programming
- Building classes out of other classes
  - Deck that contains a Vector of Card objects
  - Person with a String name, Date birthday, Vector of friends
  - Rectangle defined by two Point objects
- Can add layer upon layer
  - CardGame that contains a Deck which is a Vector of Cards
  - AddressBook that contains a Vector of Person objects
- Relationship is called "has-a" or containment
  - A Deck has a Vector of Cards
  - A Person has a birthdate

Composition syntax

- Declare instance variables of various class type

```
public class Person {
  private String name;
  private Date birthday;
  private Vector friends;
  ...
```

  Values will default to null if not otherwise assigned

- Usually allocate/assign in constructor

```
public Person(String n, Date bd) {
  name = n;
  birthday = bd;
  friends = new Vector();
}
```

- Can allocate at declaration, too

```
private Vector friends = new Vector();
```

Methods for composed objects

- Implemented by messaging the inner objects

  E.g., Person class methods would message Vector internally

```
public void addFriend(Person newFriend) {
  friends.addElement(newFriend);
}

public boolean isFriend(Person someone) {
  return friends.contains(someone);
}
```

  Why not just give an accessor to get the vector instead?

```
public Vector getFriends(){
  return friends;
}
```

  What are the advantages and disadvantages of this?

Inheritance

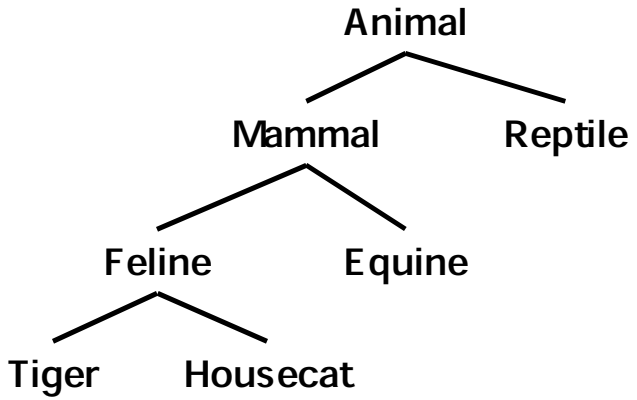
- Consider the animal kingdom

  animals reproduce

  mammals are warm-blooded

  felines have claws

  equines have hooves, etc.



- A tree is a compact way to associate properties with objects that are related to one another
- Each class described as differences from parent

  Child has all the data/features/behavior of parent plus more
- Terms subclass and superclass

  Child/parent in the hierarchy

Inheritance for code reuse

- Classes with features in common, but different, too
  - Bank accounts (checking, savings, credit line)
  - Types of resources managed by a library (books, magazines, films)
- Factor common code/behavior to superclass
  - Decomposition unit becomes the class
- Subclasses specialize parent class
  - Can extend: add entirely new data/behavior
  - Can replace: change existing behavior
  - Can refine: use existing behavior but include additions
  - Can't remove though, no good way to "take away" behavior
- Changes to parent are reflected in children
  - One place to edit, improve, fix bugs, yet all subclasses benefit

Expresses "is-a" relationship

- Not same as cooperating, composition, "has-a", "contains-a", "used-by", etc.
  - Not car & engine, hashtable & buckets, datebook & events
- Inheritance is used fairly sparingly
  - But when it's appropriate, it can be powerful
- Ask yourself: does it make sense to say “<subclass> is a <superclass>”?
  - Mammal is an Animal?
  - Housecat is a Feline? Housecat is a Mammal?
  - Zoo is an Animal? Animal is a Zoo?
- Can be subtle: should Deck of Cards be a subclass of Vector?

Example: Employee class

```
public class Employee {

  protected String name;  // note use of protected
  protected int wagePerHour, hoursPerWeek;
  protected double attitude;

  public Employee(String n, int w, int h, double a)
  {
    name = n;
    wagePerHour = w;
    hoursPerWeek = h;
    attitude = a;
  }

  // might include setters/getters for
  // the four instance variables
```

Employee methods

```
public int askSalary() {
    return wagePerHour*hoursPerWeek;
}

public void giveRaise(double percentage) {
    wagePerHour += percentage*wagePerHour;
}

public double getProductivity()  {
    return hoursPerWeek*attitude;
}

public double getGroupProductivity() {
    return getProductivity();
}
```

Example client use of Employee

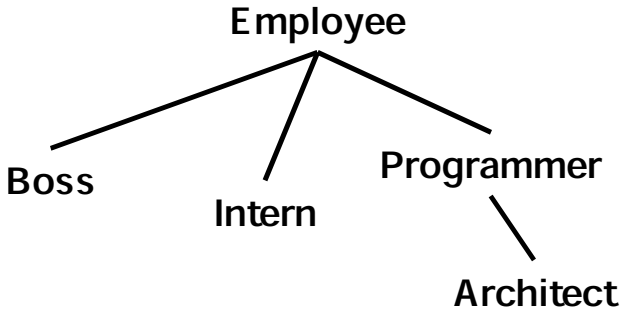
```
Employee clerk, ceo;

clerk = new Employee("Bob", 10, 40, .8);
ceo = new Employee("Sue", 90, 60, 1.1);

double prod = clerk.getProductivity();
int money = clerk.askSalary();
System.out.print("Value = " + (prod/money));
ceo.giveRaise(.10);
System.out.print("ceo earns $"+ ceo.askSalary());
```

Boss, a new type of Employee

- Boss has same basic features as employee
  - Same variables for name, wage, hours, attitude
  - Same methods about productivity, salary, raises
- But different, too
  - Wants to track an underling, or maybe a collection of them
- Consider the options
  - Add Boss behavior to Employee class?
  - Copy/paste Employee class to make similar Boss class?
  - Wasteful to duplicate the common features in two separate classes…
- Subclass Boss from Employee!

An inheritance tree

- Inheritance tree
  - General at root
  - More specific down to leaves



- Subclass has all data of superclass
  - e.g. an Programmer object has all data/methods an Employee has
- Subclass has all operations of superclass
  - e.g. can ask a Boss for productivity or give raise, just like Employee

Syntax for subclassing

```
public class Boss extends Employee {

  protected Vector underlings;

  public Boss(String nm, int wg, int hr,
              double at, Employee under) {
     super(nm, wg, hr, at);
     addUnderling(under);
  }

  public boolean isDirectReport(Employee emp) {
     return underlings.contains(emp);
  }

  public void addUnderling(Employee emp) {
     underlings.addElement(emp);
  }
```

Inheritance of instance variables

- Boss inherits data from Employee
  - Has all Employee instance variables
  - plus one additional fields, the list of underlings
- Can Boss freely access Employee ivars?
  - public yes, private no
  - protected also yes, protected is a special access for subclasses
- Access specifiers don't control inheritance, just visibility
  - Instance of subclass inherits private fields & private methods, but can't access directly
- What are tradeoffs of protected vs private?

Inheritance of methods

- Boss also inherits behavior of Employee

  Responds to all the same messages

```
Boss bob = new Boss("Peter", 40, 40, .80, null);

bob.giveRaise(.05);
int salary = bob.askSalary();
```

- Access specifiers for methods same as variables

  Within subclass implementation, can call public & protected, but not private

Subclass can extend behavior

- Add new variables

  Such as new variable to hold list of underlings

  These variables only exist for Bosses, not all Employee

- Add new methods

  Such as those to manage underlings that only apply to Boss objects

```
public void addUnderling(Employee emp)
{
    underlings.addElement(emp);
}


public boolean isDirectReport(Employee emp)
{
    return underlings.contains(emp);
}
```

Subclass can replace behavior

- Subclass can override a method implementation to provide a different version than parent
- Can completely re-implement method if desired

Boss can take credit for underling's productivity along with own

```
public double getGroupProductivity()
{
   double sum = getProductivity();
   for (int i=0; i < underlings.size(); i++) {
      Employee u = (Employee)underlings.elementAt(i);
      sum += u.getGroupProductivity();
   }
  return sum;
}
```

Boss doesn't use inherited version, just completely replaces it

Subclass can refine behavior

- Override, use inherited version, and tweak a bit

  Boss can lie about salary

```
public int askSalary() {
    return super.askSalary()*2;
}
```

  Or after giving self raise, can give one underling a little raise, too

```
public void giveRaise(double percentage) {
    super.giveRaise(percentage);
    Employee u = (Employee)underlings.elementAt(0);
    u.giveRaise(percentage/2);
}
```

- super.method()

  Sends message to this (receiving object)

  But looks for method starting in parent (or further up, if necessary)

  No way to jump to specific grandparent (unlike C++)

Client use of Employee & Boss

```
Boss v = new Boss("Viv", 20, 40, .9, null);
Employee d = new Employee("Don", 15, 20, .5);
System.out.print("Don makes " + d.askSalary());
System.out.print("Viv makes " + v.askSalary());
v.addUnderling(d);
System.out.print("Viv " + v.getProductivity());
System.out.print("Viv " + v.getGroupProductivity());
```

- Some things to note
  - Can't send setUnderling to Employee object
  - Different versions of askSalary used for different objects
  - Boss has reference to Employee, if Employee changes, Boss sees those changes

Some questions to ponder

- In Employee's getGroupProductivity() why call getProductivity(), why not just access ivars directly and compute?
- Why does Boss use super in askSalary() instead of copying code?
- Why does Boss send underling the getGroupProductivity message instead of just getProductivity?
- Why did Employee have trivial definition of getGroupProductivity()?

Invoking parent constructor

- A subclass constructor needs to indicate which parent constructor to connect to

```
public Boss(String nm, int wg, int hr,
            double at, Employee under) {
   super(nm, wg, hr, at);
   addUnderling(under);
}
```

- Passing along to superclass constructor
  - Looks strange because constructors don't really have a name
  - super(args) means to call superclass constructor
  - Has to be the very first action in the constructor
  - This syntax not valid in any method other than a constructor
- If no explicit call to a superclass constructor
  - Compiler assumes you mean to call super(), the zero-argument default constructor

Constructors and inheritance

- Constructors are not inherited

  They are tightly bound to class of definition

- Let's say Employee defines two constructors

```
public Employee(String n, int w, int h, double a)
public Employee(String n, int w)
```

- Boss defines one constructor

```
public Boss(String n, int w, int h, double a,
  Employee under)
```

- Boss does not inherit the 2 and 4-arg constructors

```
new Boss(”Georg", 40, 50, .5, null)      // Okay
new Boss(”Jin", 25);                     // Not okay
```

Constructors that chain

- If we want a 2-arg constructor, must add to Boss

```
public Boss(String nm, int wage) {
    super(nm, wage);
}
```

- Would you usually ever write a method like this?

```
public int askSalary() {
    return super.askSalary();
}
```

Not usually much point to overriding a method only to call super

Have to do it for constructors though, since they aren't inherited

Order of construction

- Superclass constructor always called

  Whether explicitly indicated or not (default used if not explicit)

- Starts at root, constructs down inheritance chain

```
public Boss(String nm, int wg){
    super(nm, wg, 40, .80); // can call diff ctor
    underlings = new Vector();
}
```

  Block of memory obtained is size of Boss object

  Object part constructed

    Default values of Object fields assigned, zero used as default

    Object constructor

  Employee part constructed

    Default values, then constructor

  Boss part constructed

Constructor summary

- Class definition lists all available constructors

  No "hidden" constructors from inheritance

- All parent class construction completed before subclass starts
- If constructor doesn't explicitly call a super constructor, it is assumed to call the no-argument
- Get used to trivial constructors that just propagate

```
public Binky(some_set_of_args) {
    super(some_set_of_args);
}
```

Overloading versus overriding

- Overloading: same name, different signature
- Overriding: same name, same signature
- Employee defines giveRaise

```
public void giveRaise(double percentage) {
```

- Boss can override giveRaise

```
public void giveRaise(double percentage){
    ...
}
```

- Boss can overload giveRaise

```
public void giveRaise(String newTitle){
    ...
}
```

- Overloads don't hide parent versions as in C++