OOP Inheritance 2

Handout by Nick Parlante

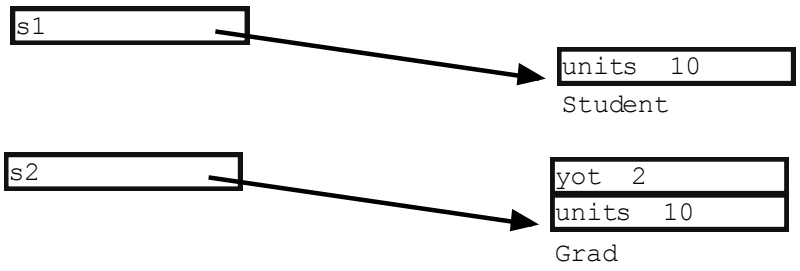
Here we look at a variety of more advanced inheritance issues.

Is-a vs. Has-a

- Specifying a superclass-subclass relationship establishes a very strong constraint between the two classes. The subclass is constrained to be an "isa" specialization of the superclass. The subclass is merely a version of the superclass category. This relationship is so constraining... that's one reason why subclassing opportunities are rare.
- e.g. Boat isa Vehicle, Chicken isa Bird isa Animal
- Contrast to the much more common "has-a" relationship where one object has a pointer to another
- e.g. Boat has-a passenger, Game has-a Tetris piece, University has-a Student has-a Advisor

Student/Grad Memory Layout

- Implementation detail where each class has a single superclass (such as in Java): in memory, the ivars of the subclass are layered on top of the ivars of the superclass (that's the simplest scheme -- the Java runtime may arrange things differently for performance reasons, but your Java code will not be able to know that.)
- Result: if you have a pointer to the base address of an instance of the subclass (Grad), you can treat it as if it were a superclass object (Student) and it just works since the objects look the same from the bottom up. Again: you can treat an instance of the subclass as if it were the superclass, and it works ok. A Grad object looks like a Student object.
- Here we have pointers Student s1, s2; The pointer s1 points to a Student while s2 points to a Grad. Note how literally the same getUnits() code can execute against both memory structures, since it just looks at the first 4 bytes for both objects -- that part of both the Student and Grad object memory look the same and have the same layout. There is just one copy of the code for getUnits() up in the Student class, and that one copy is used for Student and Grad objects. Also, you can see how it is that it does not make any sense to run getYOT() on a Student object.



Abstract Super Class Strategy

- Suppose we have three bank-account type classes, Fee Nickle and Gambler that are similar, but differ in their end-month-charge policy.
- Factor their common features up into an abstract Account class and make them subclasses
- Give Account an abstract method "abstract void endMonthCharge();". "Abstract" because it has a prototype but no code. A class with an abstract method is itself abstract. An abstract class cannot be instantiated with "new", as it is incomplete (missing a method). Subclasses must provide code for the abstract method to avoid being abstract.
- In its endMonth() method, the Account class calls the endMonthCharge() method -- knowing that this will pop-down to do the endMonthCharge() provided by the subclass.

- The abstract superclass can set out logical methods like endMonthCharge() so that subclasses can override them.

```
public abstract class Account  {
    /*
     Applies the end-of-month charge to the account.
     This is "abstract" so subclasses must override
     and provide a definition. At run time, this will
     "pop down" to the subclass definition.
    */
    protected abstract void endMonthCharge();


    public void endMonth() {
        // Pop down to the subclass for their
        // specific charge policy (abstract method)
        endMonthCharge();

        transactions = 0;
    }
...


// Fee.java
public class Fee extends Account {
    public void endMonthCharge() {
        withdraw(5.00);
    }
    ...
```

Abstract Bird Analogy

- Like the word "bird" in English -- you cannot instantiate something which is exactly a bird, but you can for subclasses like mockingbird or chicken. The word "bird" is like an abstract superclass.

Method Override -- Cannot "Narrow" Params

- When overriding a method, the parameters must be exactly the same in the subclass method as in the superclass method.
- Suppose we have a Meal class, and it supports a `same(Food food)` method (this is analogous to the equals() method in Java).

```
class Food {
  boolean same(Food food) { ...
```

- Suppose we create a Candy subclass of Food.
- In Candy, suppose we change the same() argument to take Candy instead of Food
- That looks reasonable, but it runs into logic problems.

```
class Candy extends Food {
  private int sugar;

  // incorrect override .. Candy arg more narrow than Food
  boolean same(Candy candy) {
    // Treat candy like a Candy object in here
    return (this.sugar == candy.sugar);
  }
```

- The problem is that there could be some client code (such as eat() below) that takes a Food compile time type argument, but is passed a Candy object -- that's allowed by the substitution rule that a subclass may be used in place that calls for its superclass. In eat(), the prototype seen of same() is the one from

the Food superclass which takes a Food argument, so the code can call same(broccoli). However, it pops down to the same() in Candy which takes a Candy arg, not a Food arg.

```
void eat(Food food) {
  if (food.same(broccoli))    // broccoli is a Food, not a Candy
    System.out.println("yay broccoli!");
...
}



// Problem: pass a Candy object into eat()
Candy candy = new Candy();
eat(candy);
```

- Therefore, when overriding a method, code cannot "narrow" the parameter type from the superclass. Essentially, because clients will only see the parameter type as specified in the superclass, and that's what they will pass.
- One simple solution is to copy the prototype from the superclass, paste it into the subclass, and code from there. In Java 5, the @Override annotation will check that the method does exactly match a superclass method.
- In Java 5, this rule was very slightly loosened -- the subclass method can narrow the **return type**, just not the parameters. Returning to the caller something more specific than what they asked for (returning a String instead of an Object) does not cause any problems.

instanceof

- Java includes an instanceof operator that may be used to check the run time type of a pointer -- is it a subclass of the given class (or interface)

```
if (ptr instanceof Grad) { ...
```

- instanceof with a **null** pointer always returns false
- Using instanceof is regarded as a possible sign of poor OOP design (a "smell"). Ideally, the message/method resolution selects the right piece of code depending on the class of the receiver, so further if/switch logic with instanceof should not be necessary. However, sometimes there is not a clean message solution, and instanceof is required.

Incorrect If/Switch Logic Choosing Code

```
if (x instanceof Foo) {
  // do something
} if (x instanceof Bar) {
  // do something else
}
```

Correct Logic Choosing Code

```
x.doSomething() // let message/method pick the right code to run
```

Class getClass() -- "Introspection"

- For every java class, there is an "class object" in memory that represents that class. All Java objects respond to a getClass() method that returns a pointer to that object's class object.
- Each class object is, somewhat confusingly, of the "Class" class. This is a feature of "introspection" in java -- that the classes, methods, etc. of the code are available for inspection at run time.
- Class objects have many features, including the methods getName() and newInstance(). NewInstance() makes a new object of that class, assuming it has a public zero-arg constructor.
- As with instanceof, we should avoid doing manual if/switch logic with class objects.

Java Interface

- Method Prototypes
  - An interface defines a set of method prototypes.
  - Does not provide code for implementation -- just the prototypes.
  - Can also define final constants.
- Class implements interface
  - A class that implements an interface must implement all the methods in the interface. The compiler enforces this at compile time.
  - A Java class can only have one superclass, but it may implement any number of interfaces.
- "Responds To"
  - The interface is a "responds to" claim about a set of methods.
  - If a class implements the Foo interface, I know it responds to all the messages in the Foo interface.
  - In this sense, an interface is very similar to a superclass.
  - If an object implements the Foo interface, a pointer to the object may be stored in a Foo variable. (Just like storing a pointer to a Grad object in a Student variable.)
- Lightweight
  - Interfaces allow multiple classes to respond to a common set of messages, but without introducing much complexity into the language.
  - Interfaces are lightweight compared to superclasses.
- This is similar to subclassing, however...
  - Good news: A class can only have one superclass, however it can implement any number of interfaces. Interfaces are a simple, lightweight mechanism.
  - Bad news: An interface only gives the message **prototypes**, no implementation code. The class must implement the method from scratch.
- vs. Multiple Inheritance
  - C++ multiple inheritance is more capable -- multiple superclasses -- but it introduces a lot of compiler and language complexity, so maybe it is not worth it. Interfaces provide 80% of the benefit for 10% of the complexity.

e.g. Moodable Interface

- Suppose you are implementing some sort of simulation, and there are all sorts of different objects in the program with different superclasses -- Cats, Dogs, Students, Buildings, ...
- However, you want to add a "mood ring" feature, where we can query the current color mood out an object.
- Some classes will support mood and some won't
- We define the Moodable interface -- any class that wants to support the Mood feature, implements the Moodable interface

```
// Moodable.java
public interface Moodable {
     public Color getMood();  // interface defines getMood() prototype but no code

}
```

- If a class claims to implement the Moodable interface, the compiler will enforce that the class must respond to the getMood(); message.

Student implements Moodable

- Here is what the Student class might look like, extended to implement the Moodable interface. The class must provide code for all the messages mentioned in the interface, in this case just getMood().

- A class that implements multiple interfaces separates them with commas after the "implements" keyword.

```
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...
```

Client Side Moodable

- Moodable is like an additional superclass of Student.
- It is possible to store a pointer to a Student in a pointer of type Moodable.
- The type system essentially wants to enforce the "responds to" rules. It's ok to store a pointer to a Student in a Moodable, since Student responds to getMood().
- So could say...

```
Student s = new Student(10);

Moodable m = s;        // Moodable can point to a Student
m.getMood();           // this works
m.getStress();     // NO does not compile
```

Polymorphism -- Array example

- You could have a Moodable[] array, storing pointers to all sorts of Moodable objectds. You could iterate over the array, calling getMood() on all the objects ... not worrying about their specific types.
- This feature -- that you can call getMood() with confidence that it will pop-down and do the right code depending on the class of the receiver -- is know officially as "polymorphism".

"Comparable" Interface Example

- Example of Java interface -- the Comparable interface
- Objects that work with Java's built in-sorting machinery implement the Comparable interface, which defines the one method compareTo()...

```
public interface Comparable {
  public int compareTo (Object other);
}
```

- compareTo(Object other) compares the receiver to other object, returning...
  - negative if receiver is "less" (this same convention is used in C and other languages)
  - 0 if same
  - positive if receiver is "more"
- Trick -- use subtraction: rcvr - other

Java 5 Comparable<Type>

- In Java 5, Comparable has been made generic, so a class Foo can implement Comparable<Foo>
- Then, the prototype is compareTo(Foo) instead of compareTo(Object)
- (See Ingredient example below)

Collections.sort(List)

- Collections.sort() -- works on a List (e.g. ArrayList) where the elements implement the Comparable interface. Throws a runtime exception if an element does not implement Comparable during the sort.

"Comparator" Interface

- For a custom sort not based on the built-in Comparable feature of the elements, any object can implement the **Comparator** interface which compares two objects.
- int compare(Object a, Object b) -- returns int, like Comparable
- Collections.sort(List, Comparator) -- variant that takes comparator as 2nd argument
- More flexible than Comparable, since not necessarily implemented by the objects being sorted -- can be implemented by anybody, and sort in any way.
- See Ingredient example below

Object Class

- Universal superclass in Java -- every object is, at some distance, a subclass of Object
- Object methods:
  - boolean equals(Object other); // deep comparison
  - int hashCode(); // int hash summary of object
  - String toString(); // String form of object
  - Class getClass(); // Ask an object what its class is

String toString()

- Default definition in Object, prints object's class and address
- Provide a custom toString() that produces a String summary of an object … its ivars
- Println() and String "+" know to call toString() automatically
  - System.out.println(x) will call x.toString()
  - ("hello" + x) -- calls x.toString()
- A custom toString() can be handy for debugging
  - Can just sprinkle System.out.println("about to do foo:" + x) calls around to print the state of an object over time.

boolean equals(Object other)

- Returns true if the receiver object is "deep" the same as the passed in argument object
- Much of the built-in collection machinery (ArrayList, Set, …) uses equals() to test pairs of objects.
- Test for ==this case -- fast true
- Test instanceof other object
  - if the object is not the right class, it certainly is not equal
  - instanceof test on null always yields false, what we want in this case
- Cast other to your class, do a deep comparison, ivar by ivar
- Can access .ivar of other object, though private, since we are the same class. (so called "sibling" access)
- Probably better to go through message send anyway -- avoid unnecessary dependency
- Note that the argument is type Object, and you should not make it narrower (in accordance with the general rule that the subclass cannot change the arguments as set out in the superclass prototype).
- Warning: equals() does not do the right thing with two arrays; it does a shallow == comparison. Call the static methods Arrays.equals() or Arrays.deepEquals() for deep array comparisons.

int hashCode()

- Not talking about this in much detail today. Gives an int "hash" summary of the object

- If two objects are deeply the same (.equals()) then their hashes must be the same. Rule: if a.equals(b), then it is required that a.hashCode()==b.hashCode()
- hashCode() allows an object to be a key in a HashMap. HashCode should be fast to compute.
- If a class does not implement hashCode() then it cannot be a key in a Hashap (it can still be a value). String, Integer, … all implement hashCode(), so they can be used as keys fine.
- For production quality code, if equals() is overridden, then hashCode() should also be overridden to be consistent with equals().

Ingredient Code

```
// Ingredient.java
import java.util.*;

/**
 * The Ingredient class encapsulates the name of the ingredient
 * and its quantity in grams. Used to demonstrate standard overrides:
 * toString(), equals(), hashCode(), and compareTo().
 * Also demonstrates sorting and Javadoc.
 */

public class Ingredient implements Comparable<Ingredient> {
    private String name;
    private int grams;

    /**
     * Constructs a new Ingredient.
     *
     * @param name name of ingredient
     * @param grams quantity of ingredient in grams
     */
    public Ingredient(String name, int grams) {
        this.name = name;
        this.grams = grams;
    }

    /**
     * Gets the name of the Ingredient.
     *
     * @return name of ingredient
     */
    public String getName() {
        return name;
    }

    /**
     * Gets the grams of the Ingredient.
     *
     * @return grams value
     */
    public int getGrams() {
        return grams;
    }

    /**
     * Sets the grams of the Ingredient.
     *
     * @param grams new grams value
     */
    public void setGrams(int grams) {
        this.grams = grams;
    }


    /**
     * Returns a String form of the Ingredient.
     * Uses the format <code>"<i>name</i> (<i>grams-value</i> grams)"</code>
     *
     * @return string form of ingredient
```

```
 */
@Override
public String toString() {
    return name + " (" + grams + " grams)";
}


/**
 * Compares this ingredient to the given object (standard override).
 *
 * @return true if ingredient has the same value as the given object
 */
@Override
public boolean equals(Object obj) {
    // Note: our argument must be Object, not Ingredient,
    // to match the equals() prototype up in the Object class.

    // Standard equals() tests...
    if (this == obj) return true;
    if (!(obj instanceof Ingredient)) return false;

    // Now do deep compare
    Ingredient other = (Ingredient)obj;
    return (grams==other.getGrams() && name.equals(other.getName()));
}

/**
 * Returns an int hashcode for this ingredient (standard override).
 *
 * @return int hashcode of ingredient
 */
@Override
public int hashCode() {
    // if two objects are deeply the same, their
    // hash codes must be the same
    return (grams + name.length()*11);
    // could use name.hashCode() instead of name.length()
}

/**
 * Compares the ingredient to the given object for sort order.
 * (This is the standard sorting override, implemented here
 * since we implement the "Comparable" interface.)
 * Orders increasing by name, and for the same name, increasing by grams.
 * This is the Java 5 version, generic for Comparable<Ingredient>, so
 * the arg is type Ingredient.
 *
 * @return negative/0/positive to indicate ordering vs. given object
 */
public int compareTo(Ingredient other) {
    int strCompare = name.compareTo(other.getName());
    if (strCompare != 0) return strCompare;
    else {
        if (grams < other.getGrams()) return -1;
        else if (grams > other.getGrams()) return 1;
        else return 0;
        // trick: could just return (grams - other.getGrams())
        // could access .grams directly ("sibling" access)
    }
}

/**
 * Old, non-generic compareTo(), where arg is type Object.
 *
 * @return negative/0/positive to indicate ordering vs. given object
 */
private int compareToOld(Object obj) {
    Ingredient other = (Ingredient)obj; // cast to sibling
    int strCompare = name.compareTo(other.getName());
    if (strCompare != 0) return strCompare;
    else {
```

```
        if (grams < other.grams) return -1;
        else if (grams > other.grams) return 1;
        else return 0;
        // note: here we refer to sibling ivars just as .grams
        // since we are in the same class, although using message
        // send is a slihgtly better OOP de-coupling style.
    }
}


public static void main(String args[]) {
    Ingredient a = new Ingredient("Apple", 112);
    Ingredient b = new Ingredient("Bannana", 236);
    Ingredient b2 = new Ingredient("Bannana", 236); // deeply the same as b

    System.out.println(a); // calls toString()
    System.out.println(b);
    System.out.println("Apple eq Bannana:" + (a.equals(b)));  // false
    System.out.println("Bannana eq Bannana2:" + (b.equals(b2)));  // true

    System.out.println("Apple hash:" + a.hashCode()); // 167
    System.out.println("Bannana hash:" + b.hashCode()); // 313
    System.out.println("Bannana2 hash:" + b2.hashCode()); // 313 (!)

    collectionDemo();
}


// Comparator class, new Java 5 form.
// Comparator int compare(T, T) takes two arguments
// Returns neg/0/pos if first arg is less/eq/greater
// than second arg.
private static class SortByGrams implements Comparator<Ingredient> {
    public int compare(Ingredient a, Ingredient b) {
        // trick: form the neg/0/pos by subtraction
        return (a.getGrams() - b.getGrams());
    }
}

// OLD, non generic Comparator, takes Object args
private static class SortByGramsOld implements Comparator {
    public int compare(Object a, Object b) {
        Ingredient i1 = (Ingredient)a;
        Ingredient i2 = (Ingredient)b;
        return (i1.getGrams() - i2.getGrams());
    }
}

public static void collectionDemo() {
    Ingredient a = new Ingredient("Apple", 112);
    Ingredient b = new Ingredient("Bannana", 236);
    Ingredient b2 = new Ingredient("Bannana", 100);

    List<Ingredient> ingredients = new ArrayList<Ingredient>();
    ingredients.add(b2);
    ingredients.add(a);
    ingredients.add(b);

    System.out.println("ingredients:" + ingredients);
    // ingredients:[Bannana (100 grams), Apple (112 grams), Bannana (236 grams)]
    // note: uses the collection [ ... ] built-in toString()

    Collections.sort(ingredients);
    System.out.println("sorted:" + ingredients);
    // sorted:[Apple (112 grams), Bannana (100 grams), Bannana (236 grams)]

    Collections.sort(ingredients, new SortByGrams());
    System.out.println("sorted by grams:" + ingredients);
    // sorted by grams:[Bannana (100 grams), Apple (112 grams), Bannana (236 grams)]

    System.out.println("max:" + Collections.max(ingredients));
```

```
        // max:Bannana (236 grams)
        // Also have built-in min(), max()
    }
}
```

Javadoc

- Generate HTML documentation from markup in the code
- http://java.sun.com/j2se/javadoc/writingdoccomments/
- This is how all the standard class API docs are generated
- JavaDoc sections start with two stars /** .... */
- Explain the exposed interface of the class to a client
- Eclipse code-complete/hover uses javadoc -- a great way to expose timely info to the client.
- For good examples, see the String or HashMap Javadoc pages
- In Eclipse, use Project > Generate Javadoc, and you can just leave all the defaults and generate.
- HTML markup can be used -- <p> <code> <b>
- Javadoc provides a nice, standard solution to a simple and unglamorous but very important problem -- a standard way to document a class for clients.

Class Overview Javadoc

- At the top of the class, summarize what the class encapsulates and its operational theory for the client. Imagine that the person coming to the page does not know what the class does -- give them a 2 sentence summary.
- For a large class, include <code>...</code> sections showing typical client code.
- Many classes do not do a good job on the class overview, but it's a quality touch to add.

Method Javadoc

- First sentence should summarize the whole thing. Javadoc uses the first sentence as a summary in the table of methods.
- What does it do, what effect is there on the receiver, and what are the roles of the parameters.
- Use the "s" form verb -- "Adds...", "Computes...", "Returns...". Ok to just start with the verb, leaving out the implicit "This method…" at the start.

@param

- Mention each parameter by name
- Will be redundant with the main sentence, but the info is helpful in the HTML form for the client.
- For a method int search(String target)
  - @param target the string to search for

@return

- For methods that return something
- Will also be a bit redundant with the summary sentence.
- For a method int search(String target)
  - @return index of target string if found, or -1

Example Bad Docs -- signum()

- Here's the javadoc for the signum() method in Integer...
- "signum(int i) -- Returns the signum function of the specified int value."
- Wow, it's amazing that this "doc" uses up words, and yet conveys no information.
- A better version would be dense but meaningful: "returns the -1/0/1 sign of the given int value."

Ingredient Example

- The previous Ingredient example follows the javadoc style.
- Running the Eclipse Produce Javadoc command, produces javadoc like this…