CS1004: Intro to CS in Java, Spring 2005

Lecture #23: OO Design, cont'd.

Janak J Parekh
janak@cs.columbia.edu

Administrivia

- HW#5 due Tuesday
  - And if you're cheating on (or letting others see your) HW#5… don't
  - By the way, we *do* check for cheaters across sections!

OO design aspects in Java

- We'll look at various Java constructs to help enforce OO paradigms, including:
  - Static variables and methods, revisited
  - How multiple classes can "relate" to each other
  - Interfaces: "contracts" for classes
  - Enumerated types, redux
  - Method design

Integrated Development Environments (IDEs)

- Quick detour – we've learned how to write Java code in emacs/javac, but that's not the only way
- In fact, there are specialized tools for Windows/Macs that you can also use that *integrate* the various steps of development (hence, IDE)
- Two most popular Java ones are:
  - Eclipse (http://www.eclipse.org)
  - NetBeans (http://www.netbeans.org)
  - Suggestion: for Eclipse, use the latest beta version of 3.1 (3.1M6) for full Java 5 support
- You need Java installed on your machine first
- Let's take a look

Pros and cons

- Pros
  - Nice editor, automatically shows errors
  - Easy to compile, run
  - GUI editors, integrated documentation
  - Integrated debugger
- Cons
  - *Project-based*, which is useful, but requires additional setup
  - Encourages you to design and use your own packages, which isn't required for this class
  - Requires a fair amount of computing power
  - A bazillion options and buttons; *very* confusing at first

Eclipse or NetBeans?

- No straight answer
- I primarily use Eclipse because
  - Emacs keybindings built-in
  - I like its look and feel
  - Auto-compiles
- On the other hand, I use NetBeans for GUI editing
  - Eclipse has an optional download, but not quite so robust
- You are *not required* to use either
  - If you want to try it out, be my guest, but leave some time for it
  - Post questions on the webboard, we'll try to help

Static Class Members

- Recall that a static method is one that can be invoked through its class name
- For example, the methods of the Math class are static:

```
result = Math.sqrt(25);
```

- Variables can be static as well
- Determining if a method or variable should be static is an important design decision

The static Modifier

- We declare static methods and variables using the `static` modifier
- It associates the method or variable with the class rather than with an object of that class
- Static methods are sometimes called *class methods* and static variables are sometimes called *class variables*
- Let's carefully consider the implications of each

Static Variables

- Normally, each object has its own data space, but if a variable is declared as static, only one copy of the variable exists

```
private static float price;
```

- Memory space for a static variable is created when the class is first referenced
- All objects instantiated from the class share its static variables
- Changing the value of a static variable in one object changes it for all others

Static Methods

- Example: *utility* methods are often static

```
class Helper {
   public static int cube (int num) {
      return num * num * num;
   } }
```

- Because it's static, we can execute `Helper.cube(…)` directly
- The order of the modifiers can be interchanged, but by convention visibility modifiers come first
- No benefit to creating lots of Helper objects
- On the other hand, we might create a Cube class, where "length" is an instance variable – then, we can't make `calculateArea()` static

Static Class Members

- Recall that the `main` method is static – it is invoked by the Java interpreter without creating an object
- Static methods cannot reference instance variables because instance variables don't exist until an object exists
  - Common error: instance variables in the same class as the **`main`** method
- However, a static method *can* reference static variables or local variables

main can instantiate the "same" class

- This may sound unintuitive, but if you want to access member variables in a class from within its **`main`** method:
  - First, *instantiate that class* as a variable;
  - Then, access the member through that variable declaration
- If you don't like this, feel free to put **`main`** in a different class
  - What we've been doing all along
- Quick example…

When use static variables?

- Static methods and static variables often work together
- Common paradigm: *counter* variable that keeps track of the number of objects that was instantiated
- L/L pages 294, 295
- There are indeed other ways to do this, too
  - Have a "storage" class that keeps track
  - When in doubt, avoid it

Class Relationships

- Classes in a software system can have various types of relationships to each other
- Three of the most common relationships:
  - Dependency: A *uses* B
  - Aggregation: A *has-a* B
  - Inheritance: A *is-a* B
- Inheritance is largely beyond the scope of this class; take a look at L/L chapter 8 for more info

Dependency

- A *dependency* exists when one class relies on another in some way, usually by invoking the methods of the other
- We've seen dependencies in many previous examples
- We don't want numerous or complex dependencies among classes, *nor* complex classes that don't depend on others
- A good design strikes the right balance

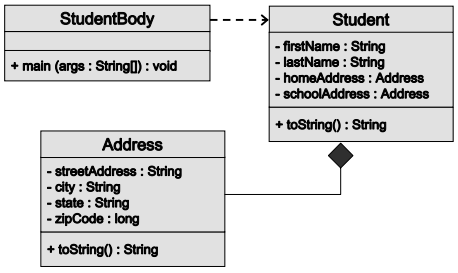
Aggregation

- An *aggregate* is an object that is made up of other objects – "has-a relationship"
  - A car *has a* chassis
- In software, an aggregate object contains references to other objects as instance data
- The aggregate object is defined in part by the objects that make it up
- This is a special kind of dependency – the aggregate usually relies on the objects that compose it

Aggregation

- In the following example, a `Student` object is composed, in part, of `Address` objects
- A student has an address (in fact each student has two addresses)
- An aggregation association is shown in a UML class diagram using an open diamond at the aggregate end
- See L/L pages 304-307 for the code

Aggregation in UML



The this Reference

- The **`this`** reference allows an object to refer to itself
- That is, the **`this`** reference, used inside a method, refers to the object through which the method is being executed
- Suppose the **`this`** reference is used in a method called **`tryMe`**, which is invoked as follows:

```
obj1.tryMe();
obj2.tryMe();
```

- In the first invocation, the this reference refers to obj1; in the second it refers to obj2

The this Reference

- The **`this`** reference can be used to distinguish the instance variables of a class from corresponding method parameters with the same names

```
public Account (String name, long acctNumber,
                double balance) {
   this.name = name;
   this.acctNumber = acctNumber;
   this.balance = balance;
}
```

Interfaces

- A Java *interface* is a collection of abstract methods and constants
- An *abstract method* is a method header without a method body
- An abstract method can be declared using the modifier `abstract`, but because all methods in an interface are abstract, usually it is left off
- An interface is used to establish a set of methods that a class will implement

Interfaces

interface is a reserved word

None of the methods in an interface are given a definition (body)

```
public interface Doable
{
   public void doThis();
   public int doThat();
   public void doThis2(float value, char ch);
   public boolean doTheOther(int num);
}
```

A semicolon immediately follows each method header

Why interfaces?

- Interfaces are commonly called a *contract* that a class agrees to by implementing the interface
- You'd do this for one of several reasons:
  - You want your buddy to implement part of the assignment, and want to tell him what to name his methods, variables, return types, etc. *(Useful in design, but not for this class!)*
  - You want to write an algorithm/program that can easily work with many different objects that all need to have some common functionality
    - For our Blackjack design yesterday, we might make a *Player interface* with two *implementations:* a user player and a computer player

Interfaces

- An interface cannot be instantiated
- Methods in an interface have public visibility by default
- A class formally implements an interface by:
  - stating so in the class header
  - providing implementations for each abstract method in the interface
- If a class asserts that it implements an interface, it must define all methods in the interface

Interfaces

```
public class CanDo implements Doable
{
   public void doThis ()
   {
      // whatever
   }

   public void doThat ()
   {
      // whatever
   }

   // etc.
}
```

implements is a reserved word

Each method listed in Doable is given a definition

Interfaces

- A class that implements an interface can implement other methods as well
- In addition to (or instead of) abstract methods, an interface can contain constants
  - (Remember that constants are declared via `public static int` or something similar)
- When a class implements an interface, it gains access to all its constants
- A class can also implement multiple interfaces; separate them with a comma

Interfaces

- The Java standard class library contains many helpful interfaces
- The `Comparable` interface contains one abstract method called `compareTo`, which is used to compare two objects
  - We discussed the `compareTo` method of the `String` class; since it's there, `String` can *implement* `Comparable`
- What's the use of implementing `Comparable`?

Easy way to sort an array in Java

- There's a static method in the `Arrays` class (in java.util) called `sort`
  - It can sort primitives: ints, doubles, floats, etc.
- For objects, it can sort them if they implement Comparable
  - In other words, it can *sort any object as long as it implements the Comparable interface*
  - Fundamental idea: Java's sort code doesn't care what your object is, as long as it knows it can compare two of them at a time
- Let's do a quick example, but note, you **can't** use this for HW5

The Iterator Interface

- Recall that an iterator is an object that provides a means of processing a collection of objects one at a time
- An iterator is created formally by implementing the `Iterator` interface, which contains three methods: hasNext, next, and remove
- By having a class implement the Iterator interface, you can use the "compact" version of the for loop
- We'll discuss this further when we talk about ArrayLists in a few weeks

Enumerated Types

- Earlier, we introduced *enumerated types*, which define a new data type and list all possible values of that type
- enums actually define a special class with those values as constants
  - You can set up special constructors and methods
- We could have used enums for Rock-Paper-Scissors

Enumerated Types

- Every enumerated type contains a static method called `values` that returns a list of all possible values for that type
- The list returned from `values` is an iterator, so a `for` loop can be used to process them easily
- A carefully designed enumerated type provides a versatile and type-safe mechanism for managing data

Next time

- Finish OO design
- Return to some theory topics