CS1004: Intro to CS in Java, Spring 2005

Lecture #24: OO Design, cont’d.

Janak J Parekh
janak@cs.columbia.edu

Administrivia

- 4 classes left (including today)... yikes!
- HW#5 due now
  - We’ll start putting up HW#4 and HW#5 solutions by the end of the week
- HW#6 already out
  - Not more implementation work, but more design and thinking than HW#5
- Final scheduling
  - If you can’t make the time due to a scheduling conflict, find me ASAP

Scoping, revisited

- Before I remotivate this again, let’s be clear on Java’s scoping rules
- A code block is usually delineated with { and }
  - Includes a class definition, a method definition, an if/else/do/for/while/switch clause
  - If you don’t use { and } in if/else/do/for/while, there is a implicit code block, but it’s exactly one statement long; switch requires {};
  - Variables declared in a for clause exist only within the for statement and corresponding code block
- Can also have an arbitrary code block inside a method
- In general, an entity is directly visible within a code block and any nested code blocks, but not in other blocks outside of the block
Scoping and variables

- There are three kinds of variables: static (class) variables, instance variables, and local variables.
- Variables exist as long as their code block does.
- Instance/static variables
  - A variable can be declared exactly once at the class level.
  - You cannot have both a static variable and an instance variable with the same name.
  - Instance variables garbage collected when the object is garbage collected (and if there are no other references to it).
  - Static variables are never garbage collected.

Scoping and local variables

- Local variables
  - Defined inside a method.
  - Can have the same name as a static/instance variable; shadows (hides) that static/instance variable by default.
  - Cannot be redefined within the same or nested code block, but can be redefined in another code block.
  - Formal parameters are in the appropriate method code block.
  - Garbage collected as soon as the method ends (and if there are no other references to it).

The this Reference

- The this reference allows a line of code to refer to the object that it’s in.
- That is, the this reference, used inside a method, refers to the object through which the method is being executed.
- Only applicable in a “non-static context.”
- Useful for two applications:
  - Disambiguating local and instance variables of the same name.
  - Handing a reference to an object to another entity within the object itself.
- We’ll see the second case later.
Disambiguation with this

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

```java
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.

```java
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.

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

```java
public class CanDo implements Doable {
    // implements is a reserved word
    public void doThis () {
        // whatever
    }
    public void doThat () {
        // whatever
    }
    // etc.
}
```

Interfaces

- 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

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.
**Method Design**

- L/L chapter 6 talks about algorithm design/decomposition
- Mostly overlap with what we’ve gone over, but there are some Java-specific aspects
- Pig Latin example: “read-only”

---

**Method Decomposition**

- A method should be relatively small, so that it can be understood as a single entity
- A potentially large method should be decomposed into several smaller methods as needed for clarity
- A public service method of an object may call one or more private support methods to help it accomplish its goal
- Support methods might call other support methods if appropriate

---

**Parameter Passing**

- Another important issue related to method design involves parameter passing
- Parameters in a Java method are *passed by value*
- A *copy* of the actual parameter (the value passed in) is stored into the formal parameter (in the method header)
- Therefore passing parameters is similar to an assignment statement
- A quick example…
Passing Objects to Methods

- When an object is passed to a method, the actual parameter and the formal parameter become aliases of each other, because a copy of the reference is made.
- What a method does with a parameter may or may not have a permanent effect (outside the method).
- Note the difference between changing the internal state of an object versus changing which object a reference points to.

Method Overloading

- Method overloading is the process of giving a single method name multiple definitions.
- If a method is overloaded, the method name is not sufficient to determine which method is being called.
- The signature of each overloaded method must be unique.
- The signature includes the number, type, and order of the parameters.

Method Overloading

- The compiler determines which method is being invoked by analyzing the parameters.

```java
float tryMe(int x){
    return x + .375;
}

float tryMe(int x, float y){
    return x*y;
}  
```

Invocation

```
result = tryMe(25, 4.32)
```
Method Overloading

- The `println` method is overloaded:
  - `println(String s)`
  - `println(int i)`
  - `println(double d)`
  - and so on...
- The following lines invoke different versions of the `println` method:
  ```java
  System.out.println("The total is:");
  System.out.println(total);
  ```

Overloading Methods

- The return type of the method is **not** part of the signature
- That is, overloaded methods cannot differ only by their return type
- Constructors can also be overloaded
- Overloaded constructors provide multiple ways to initialize a new object

The ArrayList Class

- Arrays not the only way to store data
- The `ArrayList` class is part of the `java.util` package
- Nifty feature: it's auto-resizing
- Like an array, it can store a list of values and reference each one using a numeric index
- However, you cannot use the bracket syntax with an `ArrayList`; it's an object
- Furthermore, an `ArrayList` object grows and shrinks as needed, adjusting its capacity as necessary
The ArrayList Class

- Elements can be inserted or removed with a single method invocation
- When an element is inserted, the other elements "move aside" to make room
- Likewise, when an element is removed, the list "collapses" to close the gap
- The indexes of the elements adjust accordingly
- By default, an ArrayList stores references to the Object class, which allows it to store any kind of object, but is a pain to use

ArrayLists of one type

- We can define an ArrayList object to accept only a particular type of object, like an Array
- The following declaration creates an ArrayList object that only stores Family objects
  ```java
  ArrayList<Family> reunion = new ArrayList<Family>();
  ```
- Example of generics; general discussion out of the scope of this class
- If you want to store ints, create an ArrayList of Integers, as we saw earlier, Java 1.5 is smart enough to auto-convert the two

ArrayList Efficiency

- The ArrayList class is implemented using an underlying array
- The array is manipulated so that indexes remain continuous as elements are added or removed
- The size() method returns the number of actual objects in the ArrayList, and the code prevents you from accessing empty cells
- If elements are added to and removed from the end of the list, this processing is fairly efficient
- If elements are inserted and removed from the front or middle of the list, the remaining elements are shifted
Next time

- Finish Java!
- Finish some theory topics