CS1004: Intro to CS in Java, Spring 2005

Lecture #22: Algorithms and OO

Janak J Parekh
janak@cs.columbia.edu

Administrivia

- HW#5 due next Monday
  - Please tell me you've started…
  - Written #1 is a bit of a brainteaser

Selection sort

1. Get values for *n* and the *n* list items
2. Set the marker for the unsorted section at the end of the list
3. While the sorted section of the list is not empty, do steps 4 through 6
4. Select the largest number in the unsorted section of the list
5. Exchange this number with the last number in the unsorted section of the list
6. Move the marker for the unsorted section left one position
7. Stop

- Needs the "find largest" algorithm as a "sub-algorithm"
- Let's quickly write out the Java code in our little List program
- What's the complexity of this sort?

L/L Chap 5.9-5.12

- Basically the same GUI concepts covered in chapter 4, but with loops and conditionals
- "Read-only" – take a look through in your spare time, understand the concepts
- We may have GUI programming on HW#6, but there won't be on the final

Next steps

- We finally have a good idea of algorithms and ways to tell Java to structure data for them
- How do we choose the appropriate structure?
  - Either have your instructor tell you to, or;
  - Learn it yourself
- We'll start exploring design methodologies today, but this is a lifelong learning process
- In general, designing software is a huge challenge

Program Development

- The creation of software involves four basic activities:
  - establishing the requirements
  - creating a design
  - implementing the code
  - testing the implementation
- These activities are not strictly linear – they overlap and interact
- Documentation also very important

Requirements

- *Software requirements* specify the tasks that a program must accomplish
  - what to do, not how to do it
- Often an initial set of requirements is provided, but they should be critiqued and expanded
- It is difficult to establish detailed, unambiguous, and complete requirements
- Careful attention to the requirements can save significant time and expense in the overall project
- In general, we give you the requirements

Design

- A *software design* specifies how a program will accomplish its requirements
  - How the solution can be broken down into manageable pieces
  - What each piece will do
- An *object-oriented design* determines which classes and objects are needed, and specifies how they will interact
- Low level design details include how individual methods will accomplish their tasks
- We've given you the design for the first 5 HWs; in HW6, you'll get to design various aspects of your program

Implementation

- *Implementation* is the process of translating a design into source code
- Novice programmers often think that writing code is the heart of software development, but actually it should be the least creative step
- Almost all important decisions are made during requirements and design stages
- Implementation should focus on coding details, including style guidelines and documentation

Testing

- *Testing* attempts to ensure that the program will solve the intended problem under all the constraints specified in the requirements
- A program should be thoroughly tested with the goal of finding errors
- *Debugging* is the process of determining the cause of a problem and fixing it

Identifying Classes and Objects

- The core activity of object-oriented design is determining the classes and objects that will make up the solution
- The classes may be part of a class library, reused from a previous project, or newly written
- One way to identify potential classes is to identify the objects discussed in the requirements
  - Objects are generally nouns, and the services that an object provides are generally verbs

Identifying Classes and Objects

- A partial requirements document:

The user must be allowed to specify each product by its primary characteristics, including its name and product number. If the bar code does not match the product, then an error should be generated to the message window and entered into the error log. The summary report of all transactions must be structured as specified in section 7.A.

Of course, not all nouns will correspond to a class or object in the final solution

Identifying Classes and Objects

- Remember that a class represents a group (classification) of objects with the same behaviors
- Generally, classes that represent objects should be given names that are singular nouns
  - `Coin`, `Student`, `Message`
- A class represents the concept of one such object
  - We are free to instantiate as many of each object as needed

Identifying Classes and Objects

- Sometimes it is challenging to decide whether something should be represented as a class
  - Should an employee's address be represented as a set of instance variables or as an Address object?
- The more you examine the problem and its details the more clear these issues become
- When a class becomes too complex, it often should be decomposed into multiple smaller classes to distribute the responsibilities

Identifying Classes and Objects

- We want to define classes with the proper amount of detail
- For example, it may be unnecessary to create separate classes for each type of appliance in a house
- It may be sufficient to define a more general `Appliance` class with appropriate instance data
- It all depends on the details of the problem being solved

Identifying Classes and Objects

- Part of identifying the classes we need is the process of *assigning responsibilities* to each class
- Every activity that a program must accomplish must be represented by one or more methods in one or more classes
- We generally use verbs for the names of methods
- In early stages it is not necessary to determine every method of every class – begin with primary responsibilities and evolve the design

Documenting OOD

- Some combination of
  - Psuedocode
  - Concise English descriptions
  - UML
- Key: we should avoid writing code at this stage, keep things higher-level
- Let's do a quick example

OO design aspects in Java

- This week, 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

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

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

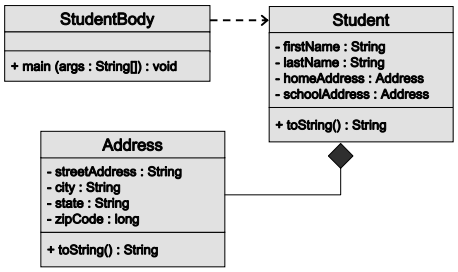
Aggregation

- 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)
- See L/L pages 304-307
- An aggregation association is shown in a UML class diagram using an open diamond at the aggregate end

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;
}
```

Next time

- Continue OO