CS 4: Lecture 14
Wednesday, March 8, 2006

OBJECT ORIENTED PROGRAMMING (OOP)

The best way to understand what object-oriented programming is is to contrast it with "procedural" programming, which existed for decades before. Object-oriented programming has everything that procedural programming has, but adds a way of organizing data and methods around "objects". Most importantly, it adds a powerful feature called "inheritance".

Procedural languages have:
- sequential execution, iteration (loops), and selection ("if", "switch"),
- procedure calls (always static--not associated with objects),
- variables,
- structures--repositories of data, like objects and classes.

(Actually, many early procedural languages, from the dark ages of computer science, don’t have procedure calls or structures.)

Object-oriented languages have:
- all the above,
- _classes_ (structure types) and _objects_ (instances of structures),
- _methods_ (procedures) associated with specific classes of objects,
- _access_controls_ like "private",
- and _inheritance_ and _polymorphism_.

Why did the originators of object orientation rename "structures" to "objects" and "procedures" to "methods"? Hubris and arrogance, in my opinion. Some people believe object orientation is a completely new "paradigm" for programming. The best known paradigms for computer languages are these.

- Procedural languages: Fortran, C, Pascal
- Functional languages: Lisp, Scheme, Haskell
- Logic languages: Prolog
- Object-oriented languages: Java, Smalltalk, C++

The first three of these really use very different ways of thinking about how to program a computer. Object-oriented languages are mostly a gloss on procedural languages.

The one part of object-orientation that really stands out, and might give it claim to being a separate "paradigm," is _polymorphism_. A few definitions:

Inheritance: A class may inherit properties from a more general class. For example, the ShoppingList class inherits from the List class the property of storing a sequence of items.
Polymorphism: The ability to have one method call work on several different classes of objects, even if those classes need different implementations of the method call. For example, one line of code might be able to call the "addItem" method on _every_ kind of List, even though adding an item to a ShoppingList is completely different from adding an item to a ShoppingCart.
Object-Oriented: Each object knows its own class and how objects in that class are manipulated. Each ShoppingList and each ShoppingCart knows which implementation of addItem applies to it.

We’ll learn more about inheritance later in the semester.

Let’s look again at the other things that separate object-oriented programming from procedural programming: classes, methods, and access controls. These ideas don’t give you any truly new abilities, but they encourage you to use good programming style and software engineering.

In a procedural language, you can place procedures anywhere in a program. In a large program, it is wise to organize your procedures--just as you wouldn’t put the chapters of a book in a random order--but the only tool for organizing them is self-discipline.

Object-oriented languages encourage you to put each method in the class that it operates on. This makes your program easier to maintain and likely to have fewer bugs. It also encourages you to use access controls to enforce information hiding.

Example: A Rational Number Class

The problem with floating-point numbers is that they get rounded off, because a computer can store only a finite number of decimal places. Sometimes, you want to be able to do math exactly, even while using division operations. People do that by expressing numbers as fractions. The set of numbers expressible as fractions is called the _rational_numbers_. We’ll define a class for fractions in a file named Fraction.java.

```
public class Fraction {
  private long numerator;
  private long denominator;
```

We’ve declared these private as part of information hiding. We don’t want other classes to be able to mess with a Fraction except through the official interface--the public methods for accessing a fraction, like this constructor.

```
  public Fraction(long n, long d) {
    if (d < 1) {
      System.out.println("Fatal error:  Non-positive denominator.");
      System.exit(0);
    }
    numerator = n;
    denominator = d;
    reduce();
  }
```

What is "reduce()"? It’s a method for reducing fractions to their simplest form. For example, if you consruct the fraction 3/6, reduce() will reduce it to 1/2. We’ll look at it later.

Thanks to method overloading, we can have more than one constructor. It’s convenient to have one for fractions representing integers.

```
  public Fraction(long n) {
    this(n, 1);
  }
```

"this" is a special keyword in Java, used to allow one constructor to call a different constructor. "this(n, 1)" calls the two-argument constructor, passing n and 1 as arguments. There’s an important difference between calling "this(n, 1)" and calling "new Fraction(n, 1)": if you did the latter, you would create _another_ Fraction. We don’t want this constructor to create two Fractions; we just want to reuse code from another constructor on the _same_ Fraction, and that’s what "this(n, 1)" does.

Why call the two-argument constructor? For good software engineering. By having this constructor call the other, we have reduced duplicate code--namely, the error-checking code and fraction reducing code in the first constructor. The program is shorter, and more importantly, if we later find a bug in the constructor, we probably only need to fix the first constructor to fix all of them. (This principle applies to all methods, not just constructors.)

Warnings:
- "this()" must be the _first_ line of a constructor. Java won’t let a constructor call "this()" after doing something else.
- You can’t use "this()" in a method that’s not a constructor.