OBJECT ORIENTED PROGRAMMING 11

GEORGE WANG
gswang.cs61a@gmail.com
Department of Electrical Engineering and Computer Sciences
University of California, Berkeley

July 12, 2010

1 Introduction

Up until now, we've been dealing with functional programming, where there's no changing of data. We've always been just concerned with the input/output pairing of a function. For the next week and a half, we'll be changing that viewpoint.

This entire week is about object oriented programming. Today and tomorrow, we'll discuss an Object Oriented framework. Later this week, we'll examine how this framework is actually implemented in Scheme.

What we'll now be thinking of, instead of a kind of mathematical view of computing, is we'll be moving to a view of having many agents acting more or less independently inside a computer. In Functional Programming, we've dealt with one large program created by composing various smaller programs together. In this next view, a view of Object Oriented Programming, the idea is that there are multiple things, each of which is designed to specialize at a certain task.

To have this happen, we need three key things:

- Message Passing - Enables cooperation and collaboration between objects.
- Local State - An object can remember things about its past history
- Inheritance - Enables specialization so that objects can fulfill niche roles without sacrificing the bigger pictures.

2 Message Passing

The first, message passing, is something we've already seen last week. As a reminder, message passing is the paradigm where instead of having functions that do things, our data is capable of handling the operations we need. Instead of operating on data, we tell the data what to do, and it is able to do it on its own.

Let's look at a program to do the same message passing things we did last week:

```
;;;;; In file cs61a/lectures/3.0/demo.scm
(define-class (square side)
  (method (area)
    (* side side))
  (method (perimeter)
    (* 4 side)) )
> (define sq (instantiate square 5))
> (ask sq 'area)
25
> (ask sq 'perimeter)[1]
20
```

Let's look at the things we just introduced. First, we should think about classes, instances, and methods.

A *class* is a blueprint for something. We create a class using DEFINE-CLASS in Scheme. It tells the computer what to do when we have a specific instance of it. However, note that we cannot manipulate a class directly, since it doesn't exist! We have to create an instance of it to do something. For example, I have the idea of a dog. But, I can't pet it, even know that I know that dogs can be petted.

An *instance* or an *object* is something that actually exists. To create an object, we use INSTANTIATE in Scheme. If a class a blueprint, then an instance is the physical building. This has specific properties and things that it does. With an instance, we can actually do things to them and on them. For example, we can pet an instance of a dog.

An INSTANTIATION VARIABLE is something that is given to a class on instantiation. My canonical example for this is your name. When you were born, you were given a name. This is what an instantiation variable is.

A METHOD is a message that the class understands. This message can take in arguments much in the same way that procedures can, which we'll see in a bit. We call a method using ASK.

Now, I've basically explained everything about this program. However, besides funny syntax, there's nothing about this program that distinguishes it from anything you did last week. There's nothing beyond message passing in this program.

3 Local State

Let's change that:

```
;;;;; In file cs61a/lectures/3.0/demo.scm
(define-class (counter)
  (instance-vars (count 0))
  (method (next)
    (set! count (+ count 1))
    count) )
> (define c1 (instantiate counter))
> (ask c1 'next)
1
> (ask c1 'next)
2
```

[1] Oops. This said something else in the draft handed out in lecture.

```
> (define c2 (instantiate counter))
> (ask c2 'next)
1
> (ask c1 'next)
3
```

Each counter has its own instance variable to remember how many times it's been sent the next message.

Don't get confused about the terms *instance variable* versus *instantiation variable*. They are similar in that each instance has its own version; the difference is that instantiation variables are given values when an instance is created, using extra arguments to instantiate, whereas the initial values of instance variables are specified in the class definition and are generally the same for every instance (although the values may change as the computation goes on.) For example, your name might be an instantiation variable, in the sense that it's assigned at birth. However, INSTANCE-VARS are things like weight and height (if we think all babies are the same weight and height).

Let's look at a method with an argument:

```
;;;;; In file cs61a/lectures/3.0/demo.scm
(define-class (doubler)
  (method (say stuff) (se stuff stuff)))
> (define dd (instantiate doubler))
> (ask dd 'say 'hello)
(hello hello)
> (ask dd 'say '(she said))
(she said she said)
```

We can also define variables that are not merely shared across instances, but shared across classes:

```
;;;;; In file cs61a/lectures/3.0/demo1.scm
(define-class (counter)
  (instance-vars (count 0))
  (class-vars (total 0))
  (method (next)
    (set! total (+ total 1))
    (set! count (+ count 1))
    (list count total)))
> (define c1 (instantiate counter))
> (ask c1 'next)
(1 1)
> (ask c1 'next)
(2 2)
> (define c2 (instantiate counter))
> (ask c2 'next)
(1 3)
> (ask c1 'next)
(3 4)
```

4 Inheritance

Now, we've pretty much covered the first two parts, let's look at the last element. To understand the idea of inheritance, we'll first define a `person` class that knows about talking in various ways, and then define a `pigger` class that's just like a person except for talking in Pig Latin.

```
;;;;; In file cs61a/lectures/3.0/demo1.scm
(define-class (person name)
  (method (say stuff) stuff)
  (method (ask stuff)
    (ask self 'say (se '(would you please) stuff)))
  (method (greet)
    (ask self 'say (se '(hello my name is) name))) )
> (define marc (instantiate person 'marc))
> (ask marc 'say '(good morning))
(good morning)
> (ask marc 'ask '(open the door))
(would you please open the door)
> (ask marc 'greet)
(hello my name is marc)
```

Notice that an object can refer to itself by the name self; this is an automatically-created instance variable in every object whose value is the object itself. (We'll see when we look below the line that there are some complications about making this work.)

```
;;;;; In file cs61a/lectures/3.0/demo1.scm
(define-class (pigger name)
  (parent (person name))
  (method (pigl wd)
    (if (member? (first wd) '(a e i o u))
        (word wd 'ay)
        (ask self 'pigl (word (bf wd) (first wd))) ))
  (method (say stuff)
    (if (atom? stuff)
        (ask self 'pigl stuff)
        (map (lambda (w) (ask self 'pigl w)) stuff))) )
> (define porky (instantiate pigger 'porky))
> (ask porky 'say '(good morning))
(oodgay orningmay)
> (ask porky 'ask '(open the door))
(ouldway ouyay easeplay openay ethay oorday)
```

The crucial point here is that the pigger class doesn't have an ask method in its definition. When we ask porky to ask something, it uses the ask method in its parent (person) class. Also, when the parent's ask method says (ask self 'say ...) it uses the say method from the pigger class, not the one from the person class. So Porky speaks Pig Latin even when asking something.

What happens when you send an object a message for which there is no method defined in its class? If the class has no parent, this is an error. If the class does have a parent, and the parent class understands the message, it works as we've seen here. But you might want to create a class that follows some rule of your own devising for unknown messages:

```
;;;;; In file cs61a/lectures/3.0/demo2.scm
(define-class (squarer)
  (default-method (* message message))
  (method (7) 'buzz) )
> (define s (instantiate squarer))
> (ask s 6)       > (ask s 7)      > (ask s 8)
36                buzz             64
```

Within the default method, the name message refers to whatever message was sent.