OOP!

POO! Spelled backwards ☺

Intro to OOP

- What is OOP?
  - Stands for Object Oriented Programming
  - Create different types of objects which can do multiple amounts of things
  - Super Abstraction for message passing
  - A metaphor for Multiple Independent Intelligent Agents

Intro to OOP

- Think of data being objects that are capable of doing certain tasks

- Objects are like independent agents, or ‘smart’ objects that can do certain things

OOP Terminology

- Object
  - A piece of data capable of doing certain tasks
- Instance
  - A specific piece of data that’s equipped with certain features (aka an object)
- Class
  - A collection of objects that share the same fields and methods. A general category for an object
- Method
  - A specific function an object can perform

How to use this…

- **(define-class ......**
  This does exactly what it seems like: it creates a new class. Its syntax is:
  (define-class (class-name args) clauses...)
  Where class-name is the name of the class, and args are the initial arguments that an instance of the class begins with.
- **method**
  Remember that a method is a procedure. Thus
  (method (name) call) Now when method name is called, it returns call.
- **instantiate**
  This asks our class for a "new" object of a class. i.e.
  (define carolen (instantiate person 'carolen))
  This defines carolen to be a new "person" object with the *instantiation variable* 'carolen which is the object's name.
- **ask**
  (ask obj message . arg)
  This gets a method from obj corresponding to message and applys it to args; else error.

Defining a class…

- Okay let's do a really EASY exercise.
- Let's define a class **energizer**, that makes an energizer bunny and whenever you ask it to **talk**, it says:
  '(I keep going and going …)

Defining a class answer:

```
(define-class (energizer)
    (method (talk) '(I keep going and going …)))
```

time to make a bunny…
(define bunny (instantiate energizer))

let's make the bunny talk!
(ask bunny 'talk)

Whoohoo! We created a class…okay just humor me ☺

Initialization and Local State

- In OOP we can create Local State Variables.
- Variables in OOP can have **state**, or specific knowledge of what has happened to them in the past.
- There are 3 kinds of Local State Variables:
  - Instance Variables
  - Instantiation Variables
  - Class Variables

More Terms!

- **Class variables**
  These are variables that the entire class of objects shares and can access (mutate).
- **Instance Variables**
  These are specific fields that each object in a class contains.
  (instance-vars (var1 val1) (var2 val2) ....)
  Thus in Berkeley OOP, var1 is bound to val1 and var2 to val2 ... local to the object. (This looks a lot like a let....(hint for the future)).
- **initialize**
  This simply initializes the value of a variable(s), or performs some task, on the instantiation (creation) of a new instance of the class.
  (initialize (var val))
  This initializes the instance-var var to val.

More Terms!

- **set!**
  A special non-functional form that changes the value of a variable.
  (set! x y)
  This changes the value of x to the value of y.
- **begin**
  This is a good function to know in Scheme. It executes a sequence of instructions and returns the value of the last one.
  (begin f1 f2 f3 ...)

Let's add to the bunny…

- Time to expand the energizer class.
  - When we create an object of the class, let's give it a name passed in as an argument.
  - Let's also create a variable ***energy*** initially set to 100.
  - When the energizer is created, print out "Hello I am: ***name***"
  - Let's add a counter to keep track of the number of energizers we have.
  - Methods:
    - ***talk***: (same as before)
    - ***recharge***: sets the energy back to 100
    - ***Power-My-Car*** *watts*: tries to power my car by asking for [watts] energy. If there is enough energy left, print out the new energy, otherwise print out "Need to recharge first."

Solution!

```
(define-class (energizer name)
    (instance-vars (energy 100))
    (class-vars (count 0))
    (initialize (display (se '(Hello I am) name))
                (newline)
                (set! count (+ count 1)))
    (method (talk) '(I keep going and going…))
    (method (recharge) (set! energy 100))
    (method (Power-My-Car watts)
          (if (> watts energy)
              '(Need to recharge first)
              (begin (set! energy (- energy watts))
                     energy))))
```

More method making…

- Let’s try a trickier method…
  - Write ***power-up*** that takes an argument *new-watts*, and sets the energizer’s energy to be this total.
  - However, the energizer must print out: (I added *xxx* Watts to my previous *yyy*).

A solution…

```
(method (power-up new-watts)
    (begin (set! energy new-watts)
            (se ‘(I added)
                 (- energy new-watts)
                ‘(Watts to my previous)
                energy ‘Watts)))
```

Correct Solution!

```
(method (power-up new-watts)
    (let ((old-watts energy))
       (begin (set! energy new-watts)
              (se ‘(I added )
                   (- new-watts old-watts)
                   ‘(Watts to my previous )
                   old-watts ‘Watts))))
```

Fixing up the energizer bunny…

```
(define-class (energizer name)
        (instance-vars (energy 100))
        (class-vars (count 0))
        (initialize (display (se ‘(Hello I am) name))
                (newline)
                (set! count (+ count 1)))
        (method (talk) ‘(I keep going and going…))
        (method (recharge) (set! energy 100))
        (method (Power-My-Car watts)
            (if (> watts energy)
              ‘(Need to recharge first)
              (begin (set! energy (- energy watts))
                  energy)))
        (method (power-up new-watts)
            (let ((old-watts energy))
              (begin (set! energy new-watts)
                  (se ‘(I added )
                     (- new-watts old-watts)
                     ‘(Watts to my previous )
                     old-watts ‘Watts))))
```

Wouldn’t it be better to have recharge call power-up?

How would we be able to do that? (How can we call another method (on an object) inside a method call?)

Answer…SELF!

- We can ask our self (our self being the object calling a function) to perform another task.

```
….
(method (recharge)
    (ask self ‘power-up 100))
….
```

What if we don’t know a message?

- We can define a **default-method** that will handle this case.
- So modify the energizer class to call ‘talk if we get a message we can’t handle.

Inheritance!

- What happens when we start having classes that have the same methods, fields, and initializations or more so, they are somehow related to each other?

```
(define-class (dog name)
    (instance-vars (type ‘pet))
    (method (speak) ‘Woof!))
(define-class (cat name)
    (instance-vars (type ‘pet))
    (method (speak) ‘Meow))
```

- One way to make sure you don’t use duplicate code is to use inheritance

Inheritance Terms!

- **Inheritance**
  The idea that a class can inherit (use/call) methods and fields from another class without explicitly typing them out.
- **Subclass**
  class x is a subclass of class y if x inherits from y. This is a more specific version of a parent class
- **Superclass**
  The parent class, or class that has subclasses which inherit its methods and fields. This is a general class
- **Multiple Superclasses**
  A subclass can inherit from more than one superclass. In this case, the subclass is said to have multiple superclasses.
- **(parent (super args) ...)**
  The declaration in Berkeley OOP that states this class is a subclass of class super.

Time to fix the classes…

- Let’s have our energizer class inherit from a class ***battery***. Class battery keeps a list of all the current types (classes) of batteries.
- Class battery:
  - Method: ***Price*** – returns the price of a battery (a random number between 0 and 99).
  - Keeps track of names of batteries created.
- Create class **duracell**:
  - Initially has **energy** set to 0
  - Method: ***Power-My-Car*** – returns ‘(sorry out of power)

Solution…

```
(define-class (battery name)
        (class-vars (names ‘())
        (initialize (set! names (cons name names)))
        (method (price) (random 100)))

(define-class (duracell name)
        (parent (battery name))
        (instance-vars (energy 0))
        (method (Power-My-Car watts) ‘(Sorry out of power)))
```

In energizer class:

```
(parent (battery name))
```

Superclass vs. Subclass

- What’s the difference?
- Subclasses are more specific types of the superclass.
  - So Duracell and Energizer are specific types of batteries
  - All Duracells are batteries. All Energizers are batteries
- But remember inheritance is a one way street!
  - So not all batteries are Energizers (same for Duracells).

Unuuuuuuusual

- **Usual**.
  This is the last syntactic word in OOP. Usual means "use the parents method". This is useful for adding another procedure to a function call.

  For example, lets say we wanted to add something to our price method for Energizers. Lets make our energizers price method return the sentence: (I cost xxx Dollars).

  Well we can implement this using usual. Our algorithm will be to store the value returned from our usual (parent) call to price, and then return it in a sentence.

Usual changes...

In energizer class:

```
….
  (method (price)
       (let ((cost (usual ‘price)))
           (se ‘(I cost) cost ‘dollars)))
….
```

Let’s try some more…

- Create a traffic light
  - When first created, the **color** of the light is red
  - Methods:
    - **(ask traffic-light ‘change):** changes the color to the next color, red→yellow, yellow→green, green→red
    - **(ask traffic-light ‘set-color!):** changes the color of the light.
- Create a stack.
  - Stacks will know how many things are in it
  - Methods:
    - **(ask stack ‘push):** adds something to the top of the stack
    - **(ask stack ‘pop):** takes off and returns what’s on the top of the stack. If the stack is empty, return “EMPTY!”.

Next Time:
I LOVE ED!

…Environment Diagrams of course…the fun has just begun ☺