Object-Oriented Programming — Above the line view

This document should be read before Section 3.1 of the text. A second document, "Object-Oriented Programming — Below the line view," should be read after Section 3.1 and perhaps after Section 3.2; the idea is that you first learn how to use the object-oriented programming facility, then you learn how it's implemented.

Object-oriented programming is a metaphor. It expresses the idea of several independent agents inside the computer, instead of a single process manipulating various data. For example, the next programming project is an adventure game, in which several people, places, and things interact. We want to be able to say things like "Ask Fred to pick up the potstickers." (Fred is a *person* object, and the potstickers are a *thing* object.)

Programmers who use the object metaphor have a special vocabulary to describe the components of an object-oriented programming (OOP) system. In the example just above, "Fred" is called an *instance* and the general category "person" is called a *class*. Programming languages that support OOP let the programmer talk directly in this vocabulary; for example, every OOP language has a "define class" command in some form. For this course, we have provided an extension to Scheme that supports OOP in the style of other OOP languages. Later we shall see how these new features are implemented using Scheme capabilities that you already understand. OOP is not magic; it's a way of thinking and speaking about the structure of a program.

When we talk about a "metaphor," in technical terms we mean that we are providing an abstraction. The above-the-line view is the one about independent agents. Below the line there are three crucial technical ideas: message-passing (section 2.3), local state (section 3.1), and inheritance (explained below). This document will explain how these ideas look to the OOP programmer; later we shall see how they are implemented.

A simpler version of this system and of these notes came from MIT; this version was developed at Berkeley by Matt Wright.

In order to use the OOP system, you must load the file `~cs61a/lib/obj.scm` into Scheme.

Message Passing

The way to get things to happen in an object oriented system is to send messages to objects asking them to do something. You already know about message passing; we used this technique in Section 2.3 to implement generic operators using "smart" data. For example, in Section 3.1 much of the discussion will be about *bank account* objects. Each account has a *balance* (how much money is in it); you can send messages to a particular account to *deposit* or *withdraw* money. The book's version shows how these objects can be created using ordinary Scheme notation, but now we'll use OOP vocabulary to do the same thing. Let's say we have two objects `Matt-Account` and `Brian-Account` of the bank account class. (You can't actually type this into Scheme yet; the example assumes that we've already created these objects.)

```
> (ask Matt-Account 'balance)
1000
```

```
> (ask Brian-Account 'balance)
10000
> (ask Matt-Account 'deposit 100)
1100
> (ask Brian-Account 'withdraw 200)
9800
> (ask Matt-Account 'balance)
1100
> (ask Brian-Account 'withdraw 200)
9600
```

We use the procedure `ask` to send a message to an object. In the above example we assumed that bank account objects knew about three messages: `balance`, `deposit`, and `withdraw`. Notice that some messages require additional information; when we asked for the `balance`, that was enough, but when we ask an account to `withdraw` or `deposit` we needed to specify the amount also.

The metaphor is that an object "knows how" to do certain things. These things are called *methods*. Whenever you send a message to an object, the object carries out the method it associates with that message.

Local State

Notice that in the above example, we repeatedly said

```
(ask Brian-Account 'withdraw 200)
```

and got a different answer each time. It seemed perfectly natural, because that's how bank accounts work in real life. However, until now we've been using the functional programming paradigm, in which, by definition, calling the same function twice with the same arguments must give the same result.

In the OOP paradigm, the objects have *state*. That is, they have some knowledge about what has happened to them in the past. In this example, a bank account has a balance, which changes when you deposit or withdraw some money. Furthermore, each account has its own balance. In OOP jargon we say that `balance` is a *local state variable*.

You already know what a local variable is: a procedure's formal parameter is one. When you say

```
(define (square x) (* x x))
```

the variable `x` is local to the square procedure. If you had another procedure `(cube x)`, its variable `x` would be entirely separate from that of `square`. Likewise, the `balance` of `Matt-Account` is kept separate from that of `Brian-Account`.

On the other hand, every time you invoke `square`, you supply a new value for `x`; there is no memory of the value `x` had last time around. A *state* variable is one whose value survives between invocations. After you deposit some money to `Matt-Account`, the `balance` variable's new value is remembered the next time you access the account.

To create objects in this system you *instantiate* a class. For example, `Matt-Account` and

`Brian-Account` are instances of the `account` class:

```
> (define Matt-Account (instantiate account 1000))
Matt-Account
> (define Brian-Account (instantiate account 10000))
Brian-Account
```

The `instantiate` function takes a class as its first argument and returns a new object of that class. `Instantiate` may require additional arguments depending on the particular class: in this example you specify an account's initial balance when you create it.

Most of the code in an object-oriented program consists of definitions of various classes. Here is the `account` class:

```
(define-class (account balance)
  (method (deposit amount)
    (set! balance (+ amount balance))
    balance)
  (method (withdraw amount)
    (if (< balance amount)
        "Insufficient funds"
        (begin
         (set! balance (- balance amount))
         balance))) )
```

There's a lot to say about this code. First of all, there's a new special form, `define-class`. The syntax of `define-class` is analogous to that of `define`. Where you would expect to see the name of the procedure you're defining comes the name of the class you're defining. In place of the parameters to a procedure come the *initialization variables* of the class: these are local state variables whose initial values must be given as the extra arguments to `instantiate`. The body of a class consists of any number of *clauses*; in this example there is only one kind of clause, the `method` clause, but we'll learn about others later. The order in which clauses appear within a `define-class` doesn't matter.

The syntax for defining methods was also chosen to resemble that for defining procedures. The "name" of the method is actually the *message* used to access the method. The parameters to the method correspond to extra arguments to the `ask` procedure. For example, when we said

```
(ask Matt-Account 'deposit 100)
```

we associated the argument 100 with the parameter `amount`.

You're probably wondering where we defined the `balance` method. For each local state variable in a class, a corresponding method of the same name is defined automatically. These methods have no arguments, and they just return the current value of the variable with that name.

This example also introduced two new special forms that are not unique to the object system. The first is `set!`, whose job it is to change the value of a state variable. Its first argument is unevaluated; it is the name of the variable whose value you wish to change. The second argument *is* evaluated; the value of this expression becomes the new value of the variable. The return value of `set!` is undefined.

This looks a lot like the kind of `define` without parentheses around the first argument, but the meaning is different. `Define` creates a new variable, while `set!` changes the value of an existing variable.

The name `set!` has an exclamation point in its name because of a Scheme convention for procedures that modify something. (This is just a convention, like the convention about question marks in the names of predicate functions, not a firm rule.) The reason we haven't come across this convention before is that functional programming rules out the whole idea of modifying things; there is no memory of past history in a functional program.

The other Scheme primitive special form in this example is `begin`, which evaluates all of its argument expressions in order and returns the value of the last one. Until now, in every procedure we've evaluated only one expression, to provide the return value of that procedure. It's still the case that a procedure can only return one value. Now, though, we sometimes want to evaluate an expression for what it *does* instead of what it *returns*, e.g. changing the value of a variable. The call to `begin` indicates that the `(set! amount (- amount balance))` and the `balance` together form a single argument to `if`. You'll learn more about `set!` and `begin` in Chapter 3.

Inheritance

Imagine using OOP in a complicated program with many different kinds of objects. Very often, there will be a few classes that are almost the same. For example, think about a window system. There might be different kinds of windows (text windows, graphics windows, and so on) but all of them will have certain methods in common, e.g., the method to move a window to a different position on the screen. We don't want to have to reprogram the same method in several classes. Instead, we create a more general class (such as "window") that knows about these general methods; the specific classes (like "text window") *inherit* from the general class. In effect, the definition of the general class is included in that of the more specific class.

Let's say we want to create a checking account class. Checking accounts are just like regular bank accounts, except that you can write checks as well as withdrawing money in person. But you're charged ten cents every time you write a check.

```
> (define Hal-Account (instantiate checking-account 1000))
Hal-Account
> (ask Hal-Account 'balance)
1000
> (ask Hal-Account 'deposit 100)
1100
> (ask Hal-Account 'withdraw 50)
1050
> (ask Hal-Account 'write-check 30)
1019.9
```

One way to do this would be to duplicate all of the code for regular accounts in the definition of the `checking-account`. This isn't so great, though; if we want to add a new feature to the `account` class we would need to remember to add it to the `checking-account` class as well.

It is very common in object-oriented programming that one class will be a *specialization* of another: the new class will have all the methods of the old, plus some extras, just as in this bank account example. To describe this situation we use the metaphor of a *family* of object classes. The original class is the *parent* and the specialized version is the *child* class. We say that the child inherits the methods of the parent. (The names *subclass* for child and *superclass* for parent are also sometimes used.)

Here's how we create a subclass of the `account` class:

```
(define-class (checking-account init-balance)
  (parent (account init-balance))
  (method (write-check amount)
    (ask self 'withdraw (+ amount 0.10)) ))
```

This example introduces the `parent` clause in `define-class`. In this case, the parent is the `account` class. Whenever we send a message to a `checking-account` object, where does the corresponding method come from? If a method of that name is defined in the `checking-account` class, it is used; otherwise, the OOP system looks for a method in the parent `account` class. (If that class also had a parent, we might end up inheriting a method from that twice-removed class, and so on.)

Notice also that the `write-check` method refers to a variable called `self`. Each object has a local state variable `self` whose value is the object itself. (Notice that you might write a method within the definition of a class `C` thinking that `self` will always be an instance of `C`, but in fact `self` might turn out to be an instance of another class that has `C` as its parent.)

Methods defined in a certain class only have access to the local state variables defined in the same class. For example, a method defined in the `checking-account` class can't refer to the `balance` variable defined in the `account` class; likewise, a method in the `account` class can't refer to the `init-balance` variable. This rule corresponds to the usual Scheme rule about scope of variables: each variable is only available within the block in which it's defined. (Not every OOP implementation works like this, by the way.)

If a method in the `checking-account` class needs to refer to the `balance` variable defined in its parent class, the method could say

```
(ask self 'balance)
```

This invocation of `ask` sends a message to the `checking-account` object, but because there is no `balance` method defined within the `checking-account` class itself, the method that's inherited from the `account` class is used.

We used the name `init-balance` for the new class's initialization variable, rather than just `balance`, because we want that name to mean the variable belonging to the parent class. Since the OOP system automatically creates a method named after every local variable in the class, if we called this variable `balance` then we couldn't use a `balance` message to get at the parent's `balance` state variable. (It is the parent, after all, in which the account's balance is changed for each transaction.)

We have now described the three most important parts of the OOP system: message passing, local state, and inheritance. In the rest of this document we introduce some "bells and whistles"—additional features that make the notation more flexible, but don't really involve major new ideas.

Three Kinds of Local State Variables

So far the only local state variables we've seen have been *instantiation* variables, whose values are given as arguments when an object is created. Sometimes we'd like each instance to have a local state variable, but the initial value is the same for every object in the class, so we don't want to have to mention it at each instantiation. To achieve this purpose, we'll use a new kind of `define-class` clause, called `instance-vars`:

```
(define-class (checking-account init-balance)
  (parent (account init-balance))
  (instance-vars (check-fee 0.10))
  (method (write-check amount)
    (ask self 'withdraw (+ amount check-fee)))
  (method (set-fee! fee)
    (set! check-fee fee)) )
```

We've set things up so that every new checking account will have a ten-cent fee for each check. It's possible to change the fee for any given account, but we don't have to say anything if we want to stick with the ten cent value.

Instantiation variables are *also* instance variables; that is, every instance has its own private value for them. The only difference is in the notation—for instantiation variables you give a value when you call `instantiate`, but for other instance variables you give the value in the class definition.

The third kind of local state variable is a *class* variable. Unlike the case of instance variables, there is only one value for a class variable for the entire class. Every instance of the class shares this value. For example, let's say we want to have a class of workers that are all working on the same project. That is to say, whenever any of them works, the total amount of work done is increased. On the other hand, each worker gets hungry separately as he or she works. Therefore, there is a common `work-done` variable for the class, and a separate `hunger` variable for each instance.

```
(define-class (worker)
  (instance-vars (hunger 0))
  (class-vars (work-done 0))
  (method (work)
    (set! hunger (1+ hunger))
    (set! work-done (1+ work-done))
    'whistle-while-you-work ))

> (define brian (instantiate worker))
BRIAN
> (define matt (instantiate worker))
MATT
> (ask matt 'work)
WHISTLE-WHILE-YOU-WORK
> (ask matt 'work)
WHISTLE-WHILE-YOU-WORK
> (ask matt 'hunger)
2
```

```
> (ask matt 'work-done)
2
> (ask brian 'work)
WHISTLE-WHILE-YOU-WORK
> (ask brian 'hunger)
1
> (ask brian 'work-done)
3
> (ask worker 'work-done)
3
```

As you can see, asking any worker object to work increments the `work-done` variable. In contrast, each worker has its own `hunger` instance variable, so that when Brian works, Matt doesn't get hungry.

You can ask any instance the value of a class variable, or you can ask the class itself. This is an exception to the usual rule that messages must be sent to instances, not to classes.

Initialization

Sometimes we want every new instance of some class to carry out some initial activity as soon as it's created. For example, let's say we want to maintain a list of all the worker objects. We'll create a class variable called `all-workers` to hold the list, but we also have to make sure that each newly created instance adds itself to the list. We do this with an `initialize` clause:

```
(define-class (worker)
  (instance-vars (hunger 0))
  (class-vars (all-workers '())
              (work-done 0))
  (initialize (set! all-workers (cons self all-workers)))
  (method (work)
    (set! hunger (1+ hunger))
    (set! work-done (1+ work-done))
    'whistle-while-you-work ))
```

The body of the `initialize` clause is evaluated when the object is instantiated. (By the way, don't get confused about those two long words that both start with "I." *Instantiation* is the process of creating an instance (that is, a particular object) of a class. *Initialization* is some optional, class-specific activity that the newly instantiated object might perform.)

If a class and its parent class both have `initialize` clauses, the parent's clause is evaluated first. This might be important if the child's initialization refers to local state that is maintained by methods in the parent class.

Classes That Recognize Any Message

Suppose we want to create a class of objects that return the value of the previous message they received whenever you send them a new message. Obviously, each such object needs an instance variable in which it will remember the previous message. The hard part is that we want objects of this class to accept *any* message, not just a few specific messages. Here's how:

```
(define-class (echo-previous)
  (instance-vars (previous-message 'first-time))
  (default-method
    (let ((result previous-message))
      (set! previous-message message)
      result)))
```

We used a `default-method` clause; the body of a `default-method` clause gets evaluated if an object receives a message for which it has no method. (In this case, the `echo-previous` object doesn't have any regular methods, so the `default-method` code is executed for any message.)

Inside the body of the `default-method` clause, the variable `message` is bound to the message that was received and the variable `args` is bound to a list of any additional arguments to `ask`.

Using a Parent's Method Explicitly

In the example about checking accounts earlier, we said

```
(define-class (checking-account init-balance)
  (parent (account init-balance))
  (method (write-check amount)
    (ask self 'withdraw (+ amount 0.10)) ))
```

Don't forget how this works: Because the `checking-account` class has a parent, whatever messages it doesn't understand are processed in the same way that the parent (`account`) class would handle them. In particular, `account` objects have `deposit` and `withdraw` methods.

Although a `checking-account` object asks itself to `withdraw` some money, we really intend that this message be handled by a method defined within the parent `account` class. There is no problem here because the `checking-account` class itself does not have a `withdraw` method.

Imagine that we want to define a class with a method of the same name as a method in its parent class. Also, we want the child's method to invoke the parent's method of the same name. For example, we'll define a `TA` class that is a specialization of the `worker` class. The only difference is that when you ask a `TA` to work, he or she returns the sentence "Let me help you with that box and pointer diagram" after invoking the `work` method defined in the `worker` class.

We can't just say `(ask self 'work)`, because that will refer to the method defined in the child class. That is, suppose we say:

```
(define-class (TA)
  (parent (worker))
  (method (work)
    (ask self 'work)     ;; WRONG!
    '(Let me help you with that box and pointer diagram))
  (method (grade-exam) 'A+) )
```

When we ask a TA to `work`, we are hoping to get the result of asking a worker to `work` (increasing hunger, increasing work done) but return a different sentence. But what actually happens is an infinite recursion. Since `self` refers to the TA, and the TA does have its own `work` method, that's what gets used. (In the earlier example with checking accounts, `ask self` works because the checking account does *not* have its own `withdraw` method.)

Instead we need a way to access the method defined in the parent (`worker`) class. We can accomplish this with `usual`:

```
(define-class (TA)
  (parent (worker))
  (method (work)
    (usual 'work)
    '(Let me help you with that box and pointer diagram))
  (method (grade-exam) 'A+) )
```

`Usual` takes one or more arguments. The first argument is a message, and the others are whatever extra arguments are needed. Calling `usual` is just like saying `(ask self ...)` with the same arguments, except that only methods defined within an ancestor class (parent, grandparent, etc.) are eligible to be used. It is an error to invoke `usual` from a class that doesn't have a parent class.

You may be thinking that `usual` is a funny name for this function. Here's the idea behind the name: We are thinking of subclasses as specializations. That is, the parent class represents some broad category of things, and the child is a specialized version. (Think of the relationship of checking accounts to accounts in general.) The child object does almost everything the same way its parent does. The child has some special way to handle a few messages, different from the usual way (as the parent does it). But the child can explicitly decide to do something in the *usual* (parent-like) way, rather than in its own specialized way.

Multiple Superclasses

We can have object types that inherit methods from more than one type. We'll invent a `singer` class and then create `singer-TA`s and `TA-singer`s.

```
(define-class (singer)
  (parent (worker))
  (method (sing) '(tra-la-la)) )

(define-class (singer-TA)
  (parent (singer) (TA)) )

(define-class (TA-singer)
  (parent (TA) (singer)) )

> (define Matt (instantiate singer-TA))
> (define Chris (instantiate TA-singer))
> (ask Matt 'grade-exam)
A+
```

```
> (ask Matt 'sing)
(TRA-LA-LA)
> (ask Matt 'work)
WHISTLE-WHILE-YOU-WORK
> (ask Chris 'work)
(LET ME HELP YOU WITH THAT BOX AND POINTER DIAGRAM)
```

Both `Matt` and `Chris` can do anything a `TA` can do, such as grading exams, and anything a `singer` can do, such as singing. The only difference between them is how they handle messages that `TA`s and `singer`s process *differently*. `Matt` is primarily a `singer`, so he responds to the `work` message as a `singer` would. `Chris`, however, is primarily a `TA`, and uses the `work` method from the `TA` class.

In the example above, `Matt` used the `work` method from the `worker` class, inherited through two levels of parent relationships. (The `worker` class is the parent of `singer`, which is a parent of `singer-TA`.) In some situations it might be better to choose a method inherited directly from a second-choice parent (the `TA` class) over one inherited from a first-choice grandparent. Much of the complexity of contemporary object-oriented programming languages has to do with specifying ways to control the order of inheritance in situations like this.