## 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

2

`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
```

6

```
> (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.

**Reference Manual for the OOP Language**

There are only three procedures that you need to use: `define-class`, which defines a class; `instantiate`, which takes a class as its argument and returns an instance of the class; and `ask`, which asks an object to do something. Here are the explanations of the procedures:

# ASK: (`ask` *object message . args*)

`Ask` gets a method from *object* corresponding to *message*. If the object has such a method, invoke it with the given *args*; otherwise it's an error.

# INSTANTIATE: (`instantiate` *class . arguments*)

`Instantiate` creates a new instance of the given *class*, initializes it, and returns it. To initialize a class, `instantiate` runs the `initialize` clauses of all the parent classes of the object and then runs the `initialize` clause of this class.

The extra arguments to `instantiate` give the values of the new object's instantiation variables. So if you say

```
(define-class (account balance) ...)
```

then saying

```
(define my-acct (instantiate account 100))
```

will cause `my-acct`'s `balance` variable to be bound to 100.

# DEFINE-CLASS:
## (`define-class` (*class-name args...*) *clauses...*)

This defines a new class named *class-name*. The instantiation arguments for this class are *args*. (See the explanation of `instantiate` above.)

The rest of the arguments to `define-class` are various *clauses* of the following types. All clauses are optional. You can have any number of `method` clauses, in any order.

> **(METHOD (***message arguments...***) *body*)**
>
> A `method` clause gives the class a method corresponding to the *message*, with the given *arguments* and *body*. A class definition may contain any number of `method` clauses. You invoke methods with `ask`. For example, say there's an object with a
>
> ```
> (method (add x y) (+ x y))
> ```
>
> clause. Then (`ask object 'add 2 5`) returns 7.
>
> Inside a method, the variable `self` is bound to the object whose method this is. (Note that `self` might be an instance of a child class of the class in which the method is defined.) A method defined within a particular class has access to the instantiation

variables, instance variables, and class variables that are defined within the same class, but does *not* have access to variables defined in parent or child classes. (This is similar to the scope rules for variables within procedures outside of the OOP system.)

Any method that is usable within a given object can invoke any other such method by invoking (`ask self` *message*). However, if a method wants to invoke the method of the same name within a parent class, it must instead ask for that explicitly by saying

(`usual` *message*  *args...*)

where *message* is the name of the method you want and *args...* are the arguments to the method.

### (INSTANCE-VARS (*var1 value1*) (*var2 value2*) ...)

`Instance-vars` sets up local state variables `var1`, `var2`, etc. Each instance of the class will have its own private set of variables with these names. These are visible inside the bodies of the methods and the initialization code within the same class definition. The initial values of the variables are calculated when an instance is created by evaluating the expressions `value1`, `value2`, etc. There can be any number of variables. Also, a method is automatically created for each variable that returns its value. If there is no `instance-vars` clause then the instances of this class won't have any instance variables. It is an error for a class definition to contain more than one `instance-vars` clause.

### (CLASS-VARS (*var1 value1*) (*var2 value2*) ...)

`Class-vars` sets up local state variables `var1`, `var2`, etc. The class has only one set of variables with these names, shared by every instance of the class. (Compare the `instance-vars` clause described above.) These variables are visible inside the bodies of the methods and the initialization code within the same class definition. The initial values of the variables are calculated when the class is defined by evaluating the expressions `value1`, `value2`, etc. There can be any number of variables. Also, a method is automatically created for each variable that returns its value. If there is no `class-vars` clause then the class won't have any class variables. It is an error for a class definition to contain more than one `class-vars` clause.

### (PARENT (*parent1 args...*) (*parent2 args...*))

`Parent` defines the parents of a class. The *args* are the arguments used to instantiate the parent objects. For example, let's say that the `rectangle` class has two arguments: `height` and `width`:

```
(define-class (rectangle height width) ...)
```

A `square` is a kind of `rectangle`; the `height` and `width` of the `square`'s `rectangle` are both the `side-length` of the `square`:

```
(define-class (square side-length)
  (parent (rectangle side-length side-length))
  ...)
```

When an object class doesn't have an explicit method for a message it receives, it looks for methods of that name (or default methods, as explained below) in the definitions of the parent classes, in the order they appear in the `parent` clause. The method that gets invoked is from the first parent class that recognizes the message.

A method can invoke a parent's method of the same name with `usual`; see the notes on the `method` clause above.

### (DEFAULT-METHOD *body*)

A `default-method` clause specifies the code that an object should execute if it receives an unrecognized message (i.e., a message that does not name a method in this class or any of its superclasses). When the body is executed, the variable `message` is bound to the message, and the variable `args` is bound to a list of the additional arguments to `ask`.

### (INITIALIZE *body*)

The body of the `initialize` clause contains code that is executed whenever an instance of this class is created.

If the class has parents, their `initialize` code gets executed before the `initialize` clause in the class itself. If the class has two or more parents, their `initialize` code is executed in the order that they appear in the `parent` clause.

**Object-Oriented Programming — Below the line view**

This document documents the Object Oriented Programming system for CS 61A in terms of its implementation in Scheme. It assumes that you already know what the system does, i.e. that you've read "Object-Oriented Programming — Above the line view." Also, this handout will assume a knowledge of how to implement message passing and local state variables in Scheme, from chapters 2.3 and 3.1 of A&S. (Chapter 3.2 from A&S will also be helpful.)

Almost all of the work of the object system is handled by the special form `define-class`. When you type a list that begins with the symbol `define-class`, Scheme translates your class definition into Scheme code to implement that class. This translated version of your class definition is written entirely in terms of `define`, `let`, `lambda`, `set!`, and other Scheme functions that you already know about.

We will focus on the implementation of the three main technical ideas in OOP: message passing, local state, and inheritance.

**Message Passing**

The text introduces message-passing with this example from Section 2.3.3 (page 141):

```
(define (make-rectangular x y)
  (define (dispatch m)
    (cond ((eq? m 'real-part) x)
          ((eq? m 'imag-part) y)
          ((eq? m 'magnitude)
           (sqrt (+ (square x) (square y))))
          ((eq? m 'angle) (atan y x))
          (else
           (error "Unknown op -- MAKE-RECTANGULAR" m))))
  dispatch)
```

In this example, a complex number object is represented by a dispatch procedure. The procedure takes a *message* as its argument, and returns a number as its result. Later, in Section 3.1.1 (page 173), the text uses a refinement of this representation in which the dispatch procedure returns a *procedure* instead of a number. The reason they make this change is to allow for extra arguments to what we are calling the *method* that responds to a message. The user says

```
((acc 'withdraw) 100)
```

Evaluating this expression requires a two-step process: First, the dispatch procedure (named `acc`) is invoked with the message `withdraw` as its argument. The dispatch procedure returns the withdraw method procedure, and that second procedure is invoked with `100` as its argument to do the actual work. All of an object's activity comes from invoking its method procedures; the only job of the object itself is to return the right procedure when it gets sent a message.

Any OOP system that uses the message-passing model must have some below-the-line mechanism for associating methods with messages. In Scheme, with its first-class procedures, it is very natural

to use a dispatch procedure as the association mechanism. In some other language the object might instead be represented as an array of message-method pairs.

If we are treating objects as an abstract data type, programs that use objects shouldn't have to know that we happen to be representing objects as procedures. The two-step notation for invoking a method violates this abstraction barrier. To fix this we invent the `ask` procedure:

```
(define (ask object message . args)
  (let ((method (object message)))      ; Step 1: invoke dispatch procedure
    (if (method? method)
        (apply method args)             ; Step 2: invoke the method
        (error "No method" message (cadr method)))))
```

`Ask` carries out essentially the same steps as the explicit notation used in the text. First it invokes the dispatch procedure (that is, the object itself) with the message as its argument. This should return a method (another procedure). The second step is to invoke that method procedure with whatever extra arguments have been provided to `ask`.

The body of `ask` looks more complicated than the earlier version, but most of that has to do with error-checking: What if the object doesn't recognize the message we send it? These details aren't very important. `Ask` does use two features of Scheme that we haven't discussed before:

The dot notation used in the formal parameter list of `ask` means that it accepts any number of arguments. The first two are associated with the formal parameters `object` and `message`; all the remaining arguments (zero or more of them) are put in a list and associated with the formal parameter `args`.

The procedure `apply` takes a procedure and a list of arguments and applies the procedure to the arguments. The reason we need it here is that we don't know in advance how many arguments the method will be given; if we said (`method args`) we would be giving the method *one* argument, namely, a list.

In our OOP system, you generally send messages to instances, but you can also send some messages to classes, namely the ones to examine class variables. When you send a message to a class, just as when you send one to an instance, you get back a method. That's why we can use `ask` with both instances and classes. (The OOP system itself also sends the class an `instantiate` message when you ask it to create a new instance.) Therefore, both the class and each instance is represented by a dispatch procedure. The overall structure of a class definition looks something like this:

```
(define (class-dispatch-procedure class-message)
  (cond ((eq? class-message 'some-var-name) (lambda () (get-the-value)))
        (...)
        ((eq? class-message 'instantiate)
         (lambda (instantiation-var ...)
           (define (instance-dispatch-procedure instance-message)
             (cond ((eq? instance-message 'foo) (lambda ...))
                   (...)
                   (else (error "No method in instance")) ))
           instance-dispatch-procedure))
        (else (error "No method in class")) ))
```

15

(Please note that this is *not* exactly what a class really looks like. In this simplified version we have left out many details. The only crucial point here is that there are two dispatch procedures, one inside the other.) In each dispatch procedure, there is a `cond` with a clause for each allowable message. The consequent expression of each clause is a `lambda` expression that defines the corresponding method. (In the text, the examples often use named method procedures, and the consequent expressions are names rather than `lambda`s. We found it more convenient this way, but it doesn't really matter.)

**Local State**

You learned in section 3.1 that the way to give a procedure a local state variable is to define that procedure inside another procedure that establishes the variable. That outer procedure might be the implicit procedure in the `let` special form, as in this example from page 171:

```
(define new-withdraw
  (let ((balance 100))
    (lambda (amount)
      (if (>= balance amount)
          (begin (set! balance (- balance amount))
 balance)
          "Insufficient funds")))))
```

In the OOP system, there are three kinds of local state variables: class variables, instance variables, and instantiation variables. Although instantiation variables are just a special kind of instance variable above the line, they are implemented differently. Here is another simplified view of a class definition, this time leaving out all the message passing stuff and focusing on the variables:

```
(define class-dispatch-procedure
  (LET ((CLASS-VAR1 VAL1)
        (CLASS-VAR2 VAL2) ...)
    (lambda (class-message)
      (cond ((eq? class-message 'class-var1) (lambda () class-var1))
            ...
            ((eq? class-message 'instantiate)
             (lambda (INSTANTIATION-VARIABLE1 ...)
               (LET ((INSTANCE-VAR1 VAL1)
                     (INSTANCE-VAR2 VAL2) ...)
                 (define (instance-dispatch-procedure instance-message)
                   ...)
                 instance-dispatch-procedure)))))))
```

The scope of a class variable includes the class dispatch procedure, the instance dispatch procedure, and all of the methods within those. The scope of an instance variable does not include the class dispatch procedure in its methods. Each invocation of the class `instantiate` method gives rise to a new set of instance variables, just as each new bank account in the book has its own local state variables.

16

Why are class variables and instance variables implemented using `let`, but not instantiation variables? The reason is that class and instance variables are given their (initial) values by the class definition itself. That's what `let` does: It establishes the connection between a name and a value. Instantiation variables, however, don't get values until each particular instance of the class is created, so we implement these variables as the formal parameters of a `lambda` that will be invoked to create an instance.

## Inheritance and Delegation

Inheritance is the mechanism through which objects of a child class can use methods from a parent class. Ideally, all such methods would just be part of the repertoire of the child class; the parent's procedure definitions would be "copied into" the Scheme implementation of the child class.

The actual implementation in our OOP system, although it has the same purpose, uses a somewhat different technique called *delegation*. Each object's dispatch procedure contains entries only for the methods of its own class, not its parent classes. But each object has, in an instance variable, an object of its parent class. To make it easier to talk about all these objects and classes, let's take an example that we looked at before:

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

Let's create an instance of that class:

```
(define Gerry-account (instantiate checking-account 20000))
```

Then the object named `Gerry-account` will have an instance variable named `my-account` whose value is an instance of the `account` class. (The variables `my-`whatever are created automatically by `define-class`.)

What good is this parent instance? If the dispatch procedure for `Gerry-account` doesn't recognize some message, then it reaches the `else` clause of the `cond`. In an object without a parent, that clause will generate an error message. But if the object does have a parent, the `else` clause passes the message on to the parent's dispatch procedure:

```
(define (make-checking-account-instance init-balance)
  (LET ((MY-ACCOUNT (INSTANTIATE ACCOUNT INIT-BALANCE)))
    (lambda (message)
      (cond ((eq? message 'write-check) (lambda (amount) ...))
            ((eq? message 'init-balance) (lambda () init-balance))
            (ELSE (MY-ACCOUNT MESSAGE)) ))))
```

(Naturally, this is a vastly simplified picture. We've left out the class dispatch procedure, among other details. There isn't really a procedure named `make-checking-account-instance` in the implementation; this procedure is really the `instantiate` method for the class, as we explained earlier.)

17

When we send `Gerry-account` a `write-check` message, it's handled in the straightforward way we've been talking about before this section. But when we send `Gerry-account` a `deposit` message, we reach the `else` clause of the `cond` and the message is delegated to the parent `account` object. That object (that is, its dispatch procedure) returns a method, and `Gerry-account` returns the method too.

The crucial thing to understand is why the `else` clause does *not* say

```
(else (ask my-parent message))
```

The `Gerry-account` dispatch procedure takes a message as its argument, and returns *a method* as its result. `Ask`, you'll recall, carries out a two-step process in which it first gets the method and then invokes that method. Within the dispatch procedure we only want to get the method, not invoke it. (Somewhere there is an invocation of `ask` waiting for `Gerry-account`'s dispatch procedure to return a method, which `ask` will then invoke.)

There is one drawback to the delegation technique. As we mentioned in the above-the-line handout, when we ask `Gerry-account` to `deposit` some money, the `deposit` method only has access to the local state variables of the `account` class, not those of the `checking-account` class. Similarly, the `write-check` method doesn't have access to the `account` local state variables like `balance`. You can see why this limitation occurs: Each method is a procedure defined within the scope of one or the other class procedure, and Scheme's lexical scoping rules restrict each method to the variables whose scope contains it. The technical distinction between *inheritance* and *delegation* is that an inheritance-based OOP system does not have this restriction.

We can get around the limitation by using messages that ask the other class (the child asks the parent, or *vice versa*) to return (or modify) one of its variables. The (`ask self 'withdraw ...`) in the `write-check` method is an example.

**Bells and Whistles**

The simplified Scheme implementation shown above hides several complications in the actual OOP system. What we have explained so far is really the most important part of the implementation, and you shouldn't let the details that follow confuse you about the core ideas. We're giving pretty brief explanations of these things, leaving out the gory details.

One complication is multiple inheritance. Instead of delegating an unknown message to just one parent, we have to try more than one. The real `else` clauses invoke a procedure called `get-method` that accepts any number of objects (i.e., dispatch procedures) as arguments, in addition to the message. `Get-method` tries to find a method in each object in turn; only if all of the parents fail to provide a method does it give an error message. (There will be a `my`-whatever variable for each of the parent classes.)

Another complication that affects the `else` clause is the possible use of a `default-method` in the class definition. If this optional feature is used, the body of the `default-method` clause becomes part of the object's `else` clause.

When an instance is created, the `instantiate` procedure sends it an `initialize` message. Every dispatch procedure automatically has a corresponding method. If the `initialize` clause is used

in `define-class`, then the method includes that code. But even if there is no `initialize` clause, the OOP system has some initialization tasks of its own to perform.

In particular, the initialization must provide a value for the `self` variable. Every `initialize` method takes the desired value for `self` as an argument. If there are no parents or children involved, `self` is just another name for the object's own dispatch procedure. But if an instance is the `my`-whatever of some child instance, then `self` should mean that child. The solution is that the child's `initialize` method invokes the parent's `initialize` method with the child's own `self` as the argument. (Where does the child get its `self` argument? It is provided by the `instantiate` procedure.)

Finally, `usual` involves some complications. Each object has a `send-usual-to-parent` method that essentially duplicates the job of the `ask` procedure, except that it only looks for methods in the parents, as the `else` clause does. Invoking `usual` causes this method to be invoked.

## A useful feature

To aid in your understanding of the below-the-line functioning of this system, we have provided a way to look at the translated Scheme code directly, i.e., to look at the below-the-line version of a class definition. To look at the definition of the class `foo`, for example, you type

```
(show-class 'foo)
```

If you do this, you will see the complete translation of a `define-class`, including all the details we've been glossing over. But you should now understand the central issues well enough to be able to make sense of it.

We end this document with one huge example showing every feature of the object system. Here are the above-the-line class definitions:

```
(define-class (person) (method (smell-flowers) 'Mmm!))
(define-class (fruit-lover fruit) (method (favorite-food) fruit))

(define-class (banana-holder name)
  (class-vars (list-of-banana-holders '()))
  (instance-vars (bananas 0))
  (method (get-more-bananas amount)
    (set! bananas (+ bananas amount)))
  (default-method 'sorry)
  (parent (person) (fruit-lover 'banana))
  (initialize
   (set! list-of-banana-holders (cons self list-of-banana-holders))) )
```

On the next page we show the translation of the `banana-holder` class definition into ordinary Scheme. Of course this is hideously long, since we have artificially defined the class to use every possible feature at once. The translations aren't meant to be read by people, ordinarily. The comments in the translated version were added just for this handout; you won't see comments if you use `show-class` yourself.

19

```
(define banana-holder
 (let ((list-of-banana-holders '()))              ;; class vars set up
  (lambda (class-message)                          ;; class dispatch proc
    (cond
     ((eq? class-message 'list-of-banana-holders)
      (lambda () list-of-banana-holders))
     ((eq? class-message 'instantiate)
      (lambda (name)                               ;; Instantiation vars
        (let ((self '())                           ;; Instance vars
              (my-person (instantiate-parent person))
              (my-fruit-lover (instantiate-parent fruit-lover 'banana))
              (bananas 0))
          (define (dispatch message)               ;; Object dispatch proc
            (cond
             ((eq? message 'initialize)            ;; Initialize method:
              (lambda (value-for-self)             ;;  set up self variable
                (set! self value-for-self)
                (ask my-person 'initialize self)
                (ask my-fruit-lover 'initialize self)
                (set! list-of-banana-holders       ;;  user's init code
                      (cons self list-of-banana-holders))))
             ((eq? message 'send-usual-to-parent)  ;; How USUAL works
              (lambda (message . args)
                (let ((method (get-method
                                'banana-holder
                                message
                                my-person
                                my-fruit-lover)))
                  (if (method? method)
                      (apply method args)
                      (error "No USUAL method" message 'banana-holder)))))
             ((eq? message 'name) (lambda () name))
             ((eq? message 'bananas) (lambda () bananas))
             ((eq? message 'list-of-banana-holders)
              (lambda () list-of-banana-holders))
             ((eq? message 'get-more-bananas)
              (lambda (amount) (set! bananas (+ bananas amount))))
             (else                                 ;; Else clause:
              (let ((method (get-method
                              'banana-holder
                              message
                              my-person
                              my-fruit-lover)))
                (if (method? method)               ;; Try delegating...
                    method
                    (lambda args 'sorry))))))       ;; default-method

          dispatch)))                              ;; Class' instantiate
                                                   ;; proc returns object
     (else (error "Bad message to class" class-message))))))
```