Object-Oriented Programming Languages: Application and Interpretation

Éric Tanter

June 6, 2017

last updated: Tuesday, June 6th, 2017

This booklet exposes fundamental concepts of object-oriented programming languages in a constructive and progressive manner. It follows the general approach of the PLAI book by Shriram Krishnamurthi (or at least I'd like to think it does). The document assumes familiarity with the following Parts of PLAI: I-V (especially first-class functions, lexical scoping, recursion, and state), as well as XII (macros).

OOPLAI is also available in PDF version. Note however that OOPLAI is subject to change at any time, so accessing it through the web is the best guarantee to be viewing the latest version.

I warmly welcome comments, suggestions and fixes; just send me a mail!

Acknowledgments: I am thankful to the members of the PLEIAD lab, and the students of the Programming Languages course at University of Chile, in particular Ricardo Honorato, for detecting bugs and suggesting enhancements.

1 From Functions to Simple Objects

This exploration of object-oriented programming languages starts from what we know already from PLAI, as well as our intuition about what objects are.

1.1 Stateful Functions and the Object Pattern

An object is meant to encapsulate in a coherent whole a piece of state (possibly, but not necessarily, mutable) together with some behavior that relies on that state. The state is usually called *fields* (or *instance variables*), and the behavior is provided as a set of *methods*. Calling a method is often considered as *message passing*: we send a message to an object, and if it understands it, it executes the associated method.

In higher-order procedural languages like Scheme, we have seen similar creatures:

```
(define add
  (λ (n)
    (λ (m)
      (+ m n))))

> (define add2 (add 2))
> (add2 5)
7
```

The function `add2` encapsulates some hidden state (`n = 2`) and its behavior effectively depends on that state. So, in a sense, a closure is an object whose fields are its *free variables*. What about behavior? well, it has a unique behavior, triggered when the function is *applied* (from a message passing viewpoint, `apply` is the only message understood by a function).

If our language supports mutation (`set!`), we can effectively have a stateful function with changing state:

```
(define counter
  (let ([count 0])
    (λ ()
      (begin
        (set! count (add1 count))
        count))))
```

We can now effectively observe that the state of `counter` changes:

```
> (counter)
1
```

```
> (counter)
2
```

Now, what if we want a bi-directional counter? The function must be able to do either +1 or -1 on its state depending on... well, an argument!

```
(define counter
  (let ([count 0])
    (λ (cmd)
      (case cmd
        [(dec) (begin
                 (set! count (sub1 count))
                 count)]
        [(inc) (begin
                 (set! count (add1 count))
                 count)]))))
```

Note how `counter` uses `cmd` to discriminate what action to perform.

```
> (counter 'inc)
1
> (counter 'dec)
0
```

This looks quite like an object with two methods and one instance variable, doesn't it? Let's look at another example, a stack.

```
(define stack
  (let ([vals '()])
    (define (pop)
      (if (empty? vals)
          (error "cannot pop from an empty stack")
          (let ([val (car vals)])
            (set! vals (cdr vals))
            val)))

    (define (push val)
      (set! vals (cons val vals)))

    (define (peek)
      (if (empty? vals)
          (error "cannot peek from an empty stack")
          (car vals)))

    (λ (cmd . args)
```

```
(case cmd
  [(pop) (pop)]
  [(push) (push (car args))]
  [(peek) (peek)]
  [else (error "invalid command")]))))
```

Here, instead of writing each method body in place in the lambda, we use internal `define`s. Also note that we use the dot notation for the arguments of the lambda: this enables the function to receive one argument (the `cmd`) as well as zero or more extra arguments (available in the body as a list bound to `args`).

Let's try that:

```
> (stack 'push 1)
> (stack 'push 2)
> (stack 'pop)
2
> (stack 'peek)
1
> (stack 'pop)
1
> (stack 'pop)
```

cannot pop from an empty stack

We can clearly see a code pattern that can be used to define object-like abstractions. In the following we abstract the pattern more clearly:

```
(define point
  (let ([x 0])
    (let ([methods (list (cons 'x? (λ () x))
                         (cons 'x! (λ (nx) (set! x nx))))])
    (λ (msg . args)
      (apply (cdr (assoc msg methods)) args)))))
```

Note how we are able to define the λ that dispatches to the correct method in a generic fashion. We first put all methods in an association list (ie. a list of pairs) associating a symbol (aka. message) to the corresponding method. When we apply `point`, we lookup (with `assoc`) the message and get the corresponding method. We then `apply` it.

```
> (point 'x! 6)
> (point 'x?)
6
```

1.2 A (First) Simple Object System in Scheme

We can now use macros to embed a simple object system that follows the pattern identified above.

Note that in this booklet, we use `defmac` to define macros. `defmac` is like `define-syntax-rule`, but it also supports the specification of keywords and captures of identifiers (using the `#:keywords` and `#:captures` optional parameters).

```
(defmac (OBJECT ([field fname init] ...)
                ([method mname args body] ...))
  #:keywords field method
  (let ([fname init] ...)
    (let ([methods (list (cons 'mname (λ args body)) ...)])
      (λ (msg . vals)
        (apply (cdr (assoc msg methods)) vals)))))
```

We can also define a specific notation for sending a message to an object, using an arrow `->`, eg. `(-> st push 3)`:

```
(defmac (-> o m arg ...)
  (o 'm arg ...))
```

We can now use our embedded object system to define a bi-dimensional point object:

```
(define p2D
  (OBJECT
   ([field x 0]
    [field y 0])
   ([method x? () x]
    [method y? () y]
    [method x! (nx) (set! x nx)]
    [method y! (ny) (set! y ny)])))
```

and use it:

```
> (-> p2D x! 15)
> (-> p2D y! 20)
> (-> p2D x?)
15
> (-> p2D y?)
20
```

1.3 Constructing Objects

Up to now, our objects have been created as unique specimen. What if we want more than one point object, possibly with different initial coordinates?

In the context of functional programming, we have already seen how to craft various similar functions in a proper way: we can use a higher-order function, parameterized accordingly, whose role is to produce the specific instances we want. For instance, from the add function defined previously, we can obtain various single-argument adder functions:

```
> (define add4 (add 4))
> (define add5 (add 5))
> (add4 1)
5
> (add5 1)
6
```

Because our simple object system is embedded in Scheme, we can simply reuse the power of higher-order functions to define *object factories*:

JavaScript, AmbientTalk

```
(define (make-point init-x init-y)
  (OBJECT
   ([field x init-x]
    [field y init-y])
   ([method x? () x]
    [method y? () y]
    [method x! (new-x) (set! x new-x)]
    [method y! (new-y) (set! y new-y)])))
```

The make-point function takes the initial coordinates as parameter and returns a freshly created object, properly initialized.

```
> (let ([p1 (make-point 5 5)]
        [p2 (make-point 10 10)])
    (-> p1 x! (-> p2 x?))
    (-> p1 x?))
10
```

1.4 Dynamic Dispatch

Our simple object system is sufficient to show the fundamental aspect of object-oriented programming: dynamic dispatch. Notice how, in the following, a node sends the sum message to each of its children without knowing whether it is a leaf or a node:

```
(define (make-node l r)
 (OBJECT
  ([field left l]
   [field right r])
  ([method sum () (+ (-> left sum) (-> right sum))])))
```

```
(define (make-leaf v)
 (OBJECT
  ([field value v])
  ([method sum () value])))

> (let ([tree (make-node
               (make-node (make-leaf 3)
                          (make-node (make-leaf 10)
                                     (make-leaf 4)))
               (make-leaf 1))])
   (-> tree sum))
18
```

As simple as it may seem, this object system is entirely enough to illustrate the fundamental abstraction mechanism that objects really are, as opposed to abstract data types. See the chapter on the benefits and limits of objects.

1.5 Error Handling

Let us see what happens if we send a message to an object that does not know how to handle it:

```
> (let ([l (make-leaf 2)])
    (-> l print))
cdr: contract violation
  expected: pair?
  given: #f
```

The error message is really not optimal—it exposes our implementation strategy to the programmer, and does not really give a clue of what the actual problem is.

We can change the definition of the OBJECT syntactic abstraction to deal with unknown messages properly:

```
(defmac (OBJECT ([field fname init] ...)
                ([method mname args body] ...))
  #:keywords field method
  (let ([fname init] ...)
    (let ([methods (list (cons 'mname (λ args body)) ...)])
      (λ (msg . vals)
        (let ([found (assoc msg methods)])
          (if found
              (apply (cdr found) vals)
```

```
(error "message not understood:" msg))))))
```

Rather than assuming that the message will have an associated method in the method table of the object, we now first lookup and get the result as `found`, which will be `#f` if no method was found. In that case, we generate an informative error message.

This is much better indeed:

```
> (let ([l (make-leaf 2)])
    (-> l print))
message not understood: print
```

In this section, we have successfully embedded a simple object system in Scheme that shows the connection between lexically-scoped first-class functions and objects. However, we are far from done, because the object system we have is still incomplete and primitive.

2 Looking for the Self

In the previous section, we have built a simple object system. Now consider that we want to define a method on point objects, called `above`, which takes another point as parameter and returns the one that is higher (wrt the y-axis):

```
(method above (other-point)
        (if (> (-> other-point y?) y)
            other-point
            self))
```

Note that we intuitively used `self` to denote the currently-executing object; in other languages it is sometimes called `this`. Clearly, our account of OOP so far does not inform us as to what `self` is.

2.1 What is Self?

Let us go back to our first definition of an object (without macros). We see that an object is a function; and so we want, from within that function, to be able to refer to itself. How do we do that? We already know the answer from our study on recursion! We just have to use a recursive binding (with `letrec`) to give a name to the function-object and then we can use it in method definitions:

```
(define point
  (letrec ([self
            (let ([x 0])
              (let ([methods (list (cons 'x?  (λ () x))
                                   (cons 'x! (λ (nx)
                                               (set! x nx)
                                               self)))])
                (λ (msg . args)
                  (apply (cdr (assoc msg methods)) args))))])
    self))
```

Note that the body of the `letrec` simply returns `self`, which is bound to the recursive procedure we have defined.

```
> ((point 'x! 10) 'x?)
10
```

Note how the fact that the setter method `x!` returns `self` allow us to chain message sends.

In Smalltalk, methods return `self` by default.

2.2 Self with Macros

Let us take the pattern above and use it in our OBJECT macro:

```
(defmac (OBJECT ([field fname init] ...)
                ([method mname args body] ...))
  #:keywords field method
  (letrec ([self
            (let ([fname init] ...)
              (let ([methods (list (cons 'mname (λ args body)) ...)])
                (λ (msg . vals)
                  (apply (cdr (assoc msg methods)) vals))))])
    self))

(defmac (-> o m arg ...)
  (o 'm arg ...))
```

Now let us try it out with some points:

```
(define (make-point init-x)
  (OBJECT
   ([field x init-x])
   ([method x? () x]
    [method x! (nx) (set! x nx)]
    [method greater (other-point)
            (if (> (-> other-point x?) x)
                other-point
                self)])))

> (let ([p1 (make-point 5)]
        [p2 (make-point 2)])
    (-> p1 greater p2))
self: undefined;
 cannot reference undefined identifier
```

What?? But we did introduce `self` with `letrec`, so why isn't it defined? The reason is... because of *hygiene*! Remember that Scheme's `syntax-rules` is hygienic, and for that reason, it transparently renames all identifiers introduced by macros such that they don't accidentally capture/get captured where the macro is expanded. It is possible to visualize this very precisely using the macro stepper of DrRacket. You will see that the identifier `self` in the `greater` method is not the same color as the same identifier in the `letrec` expression.

Luckily for us, `defmac` supports a way to specify identifiers that can be used by the macro user code even though they are introduced by the macro itself. The only thing we need to do is therefore to specify that `self` is such an identifier:

```
(defmac (OBJECT ([field fname init] ...)
                ([method mname args body] ...))
  #:keywords field method
  #:captures self
  (letrec ([self
            (let ([fname init] ...)
              (let ([methods (list (cons 'mname (λ args body)) ...)])
                (λ (msg . vals)
                  (apply (cdr (assoc msg methods)) vals))))])
    self))
```

2.3 Points with Self

We can now define various methods that return and/or use `self` in their bodies:

```
(define (make-point init-x init-y)
 (OBJECT
  ([field x init-x]
   [field y init-y])
  ([method x? () x]
   [method y? () y]
   [method x! (new-x) (set! x new-x)]
   [method y! (new-y) (set! y new-y)]
   [method above (other-point)
           (if (> (-> other-point y?) y)
               other-point
               self)]

   [method move (dx dy)
           (begin (-> self x! (+ dx (-> self x?)))
                  (-> self y! (+ dy (-> self y?)))
                  self)])))

(define p1 (make-point 5 5))
(define p2 (make-point 2 2))

> (-> (-> p1 above p2) x?)
5
> (-> (-> p1 move 1 1) x?)
6
```

2.4 Mutually-Recursive Methods

The previous section already shows that methods can use other methods by sending messages to self. This other example shows mutually-recursive methods.

Try the same definition in Java, and compare the results for "large" numbers. Yes, our small object system does enjoy the benefits of tail-call optimization!

```
(define odd-even
  (OBJECT ()
   ([method even (n)
            (case n
              [(0) #t]
              [(1) #f]
              [else (-> self odd (- n 1))])]
    [method odd (n)
            (case n
              [(0) #f]
              [(1) #t]
              [else (-> self even (- n 1))])])))
```

```
> (-> odd-even odd 15)
#t
> (-> odd-even odd 14)
#f
> (-> odd-even even 14)
#t
```

We now have an object system that supports self, including returning self, and sending messages to self. Notice how self is bound in methods at object creation time: when the methods are defined, they capture the binding of self and this binding is fixed from then on. We will see in the following chapters that this eventually does not work if we want to support delegation or if we want to support classes.

2.5 Nested Objects

Because objects and methods are compiled into lambdas in Scheme, our objects inherit interesting properties. First, as we have seen, they are first-class values (otherwise what would be the point?). Also, as we have just seen above, method invocations in tail position are treated as tail calls, and therefore space efficient. We now look at another benefit: we can use higher-order programming patterns, such as objects producing objects (usually called *factories*). That is, we can define *nested objects*, with proper lexical scoping.

Consider the following example:

```
(define factory
  (OBJECT
```

```
   ([field factor 1]
    [field price 10])
   ([method factor! (nf) (set! factor nf)]
    [method price! (np) (set! price np)]
    [method make ()
            (OBJECT ([field price price])
                    ([method val () (* price factor)]))])))

> (define o1 (-> factory make))
> (-> o1 val)
10
> (-> factory factor! 2)
> (-> o1 val)
20
> (-> factory price! 20)
> (-> o1 val)
20
> (define o2 (-> factory make))
> (-> o2 val)
40
```

Can you do the same in Java?

Convince yourself that these results make sense.

3 Benefits and Limits of Objects

In the language course, we have been programming by defining a data type and its variants and then defining all the "services" over these structures using procedures that work by case analysis. This style of programming is sometimes called "procedural paradigm" or "functional design" (note that "functional" here does not refer to "side-effect free"!).

In PLAI, we have done this with `define-type` to introduce the data type and its variants, and using `type-case` in case-analyzing procedures. This procedural approach is common in other languages like C (unions), Pascal (variants), ML and Haskell's algebraic data types, or plain Scheme's tagged data.

So, what does object-oriented programming really bring us? What are its weaknesses? As it turns out, using an object-oriented language does not mean that programs are "object oriented". Many Java programs are not, or at least sacrifice some of the fundamental benefits of objects.

This chapter is based on the article *On Understanding Data Abstraction, Revisited* by William R. Cook (2009).

The aim of this intermediary chapter is to step back from our step-by-step construction of OOP to contrast objects with the procedural approach, with the aim of clarifying the pros and cons of each approach. Interestingly, the simple object system we have built so far is entirely sufficient study the essential benefits and limits of objects—delegation, classes, inheritance, etc. are all interesting features, but are not *essential* to objects.

3.1 Abstract Data Types

Let us first look at abstract data types (ADTs). An ADT is a data type that hides its representation and only supplies operations to manipulate its values.

For instance, an *integer set* ADT can be defined as follows:

```
adt Set is
  empty : Set
  insert : Set x Int -> Set
  isEmpty? : Set -> Bool
  contains? : Set x Int -> Bool
```

There are many possible *representations* for such an integer set ADT. For instance, one could implement it with Scheme's lists:

```
(define empty '())

(define (insert set val)
  (if (not (contains? set val))
      (cons val set)
```

```
      set))

(define (isEmpty? set) (null? set))

(define (contains? set val)
  (if (null? set) #f
      (if (eq? (car set) val)
          #t
          (contains? (cdr set) val))))
```

The following client program can then use ADT values, without being aware of the underlying representation:

```
> (define x empty)
> (define y (insert x 3))
> (define z (insert y 5))
> (contains? z 2)
#f
> (contains? z 5)
#t
```

We could as well implement the set ADT with another representation, such as using PLAI's `define-type` mechanism to create a variant type to encode a set as a linked list.

```
(define-type Set
  [mtSet]
  [aSet (val number?) (next Set?)])

(define empty (mtSet))

(define (insert set val)
  (if (not (contains? set val))
      (aSet val set)
      set))

(define (isEmpty? set) (equal? set empty))

(define (contains? set val)
  (type-case Set set
    [mtSet () #f]
    [aSet (v next)
          (if (eq? v val)
              #t
              (contains? next val))]))
```

The sample client program above runs exactly the same, even though the underlying representation is now changed:

```
> (define x empty)
> (define y (insert x 3))
> (define z (insert y 5))
> (contains? z 2)
#f
> (contains? z 5)
#t
```

3.2 Procedural Representations

We can as well consider sets as being defined by their *characteristic function*: a function that, given a number, tells us whether or not this number is part of the set. In that case, a set is simply a function `Int -> Bool`. (In PLAI, we saw that in Chapter 11, when studying the *procedural representation* of environments.)

What is the characteristic function of the empty set? a function that always returns false. And the set obtained by inserting a new element?

```
(define empty (λ (n) #f))

(define (insert set val)
          (λ (n)
            (or (eq? n val)
                (contains? set n))))

(define (contains? set val)
  (set val))
```

Because a set is represented by its characteristic function, `contains?` simply applies the function to the element. Note that the client program is again undisturbed:

```
> (define x empty)
> (define y (insert x 3))
> (define z (insert y 5))
> (contains? z 2)
#f
> (contains? z 5)
#t
```

What do we gain with the procedural representation of sets? flexibility! For instance, we can define the set of all even numbers:

```
(define even
  (λ (n) (even? n)))
```

It is not possible to fully represent this set in any of the ADT representations we considered above. (Why?) We can even define non-deterministic sets:

```
(define random
  (λ (n) (> (random) 0.5)))
```

With the procedural representation, we have much more freedom to define sets, and in addition, they can interoperate with existing set operations!

```
> (define a (insert even 3))
> (define b (insert a 5))
> (contains? b 12)
#t
> (contains? b 5)
#t
```

In contrast, with the ADT representations we have seen above, different representations *cannot* interoperate. A set-as-list value cannot be used by a set-as-struct operation, and vice versa. ADTs abstract the representation, but they only allow *a single representation* at a time.

3.3 Objects

In essence, *sets as functions are objects*! Note that objects do *not* abstract type: the type of a set-as-function is very concrete: it is a function `Int -> Bool`. Of course, as we have seen in the first chapters, an object is a generalization of a function in that it can have multiple methods.

3.3.1 Object Interfaces

We can define a notion of *object interface* that gathers the signature of the methods of an object:

```
interface Set is
  contains? : Int -> Bool
  isEmpty? : Bool
```

Let us use our simple object system to implement sets as objects:

```
(define empty
  (OBJECT ()
          ([method contains? (n) #f]
           [method isEmpty? () #t])))

(define (insert s val)
  (OBJECT ()
          ([method contains? (n)
                   (or (eq? val n)
                       (-> s contains? n))]
           [method isEmpty? () #f])))
```

Note that `empty` is an object, and `insert` is a factory function that returns objects. A set object implements the `Set` interface. The `empty` object does not contain any value, and `isEmpty?` returns `#t`. `insert` returns a new object whose `contains?` method is similar to the set characteristic function we have seen before, and `isEmpty?` returns `#f`.

A client program is unchanged for the set construction part, and then has to use message sending to interact with set objects:

```
> (define x empty)
> (define y (insert x 3))
> (define z (insert y 5))
> (-> z contains? 2)
#f
> (-> z contains? 5)
#t
```

Note that object interfaces are essentially higher-order types: methods are functions, so passing objects around means passing groups of functions around. This is a generalization of higher-order functional programming. Object-oriented programs are inherently higher-order.

3.3.2 Principles of Object-Oriented Programming

Principle: *An object can only access other objects through their public interfaces*

Once we create an object, like the one bound to `z` above, the *only* thing we can do with it is to interact by sending messages. We cannot "open it". No attribute of the object is visible, only its interface. In other words:

Principle: *An object can only have detailed knowledge about itself*

This is fundamentally different from the way we program with ADT values: in a `type-case` analysis (recall the implementation of `contains?` in the ADT implementation with `define-type`), one is opening the value and gaining direct access to its attributes. ADTs provide encapsulation, but for the clients of the ADT; not for its implementation. Objects go further in this regard. Even the implementation of the methods of an object cannot access attributes of objects other than itself.

From this we can derive another fundamental principle:

Principle: *An object is the set of observations that can be made upon it*, as defined by its object interface.

This is a strong principle, that says that if two objects behave the same for a certain experiment (ie., a number of observations), then they should be undistinguishable otherwise. This means that the use of identity-related operations (like pointer equality) are violating this principle of OOP. With `==` in Java, we can distinguish two objects that *are* different even though they *behave* in the same way.

3.3.3 Extensibility

The above principles can be considered the characteristic feature of OOP. As Cook puts it: *"Any programming model that allows inspection of the representation of more than one abstraction at a time is NOT object oriented"*

The Component Object Model (COM) is one of the purest OO programming model in practice. COM enforces all these principles: there is no built-in equality, there is no way to determine if an object is an instance of a given class. COM programs are therefore highly extensible.

Note that the extensibility of objects is in fact completely independent from inheritance! (We don't even have classes in our language) It instead comes from the use of interfaces.

3.3.4 What about Java?

Java is not a pure object-oriented language, not importantly because it has primitive types, but because it supports many operations that violate the principles we have described above. Java has primitive equality `==`, `instanceof`, casts to class types, that make it possible to distinguish two objects even though they behave the same. Java makes it possible to declare a method that accepts objects based on their classes, not their interfaces (in Java, a class name is also a type). And of course, Java allows objects to access the internals of other objects (public fields, of course, but even private fields are accessible by objects of the same class!).

This means that Java also supports ADT-style programming. There is no nothing wrong with that! But it is important to understand the design tradeoffs involved, to make an informed choice. For instance, in the JDK, certain classes respect OO principles on the surface (allowing extensibility), but in their implementation using ADT techniques (not extensible, but more efficient). If you're interested, look at the `List` interface, and the `LinkedList` implementation.

Programming in "pure OO" in Java basically means not using class names as types (ie. use class names only after `new`), and never use primitive equality (`==`).

3.4 The Extensibility Problem

Object-oriented programming is often presented as the panacea in terms of extensible software. But what exactly is meant with "extensible"?

The extensibility problem is concerned with defining a data type (structure + operations) in a way that two kinds of extension are properly supported: adding new representational variants, or adding new operations.

Here, we use the term ADT in line with Cook's usage. It is important to clarify however that the discussion of the extensibility problem here actually contrasts objects with variant types (aka. algebraic data types). We are concerned with how to have an extensible *implementation*. The interface abstraction is not relevant here.

As it turns out, ADTs and objects each nicely support one dimension of extensibility, but fail in the other. Let us study this with a well-known example: an interpreter of simple expressions.

3.4.1 ADT

We first consider the ADT approach. We define a data type for expressions with three variants:

```
(define-type Expr
   [num  (n number?)]
   [bool (b boolean?)]
   [add (l Expr?) (r Expr?)])
```

Now we can define the interpreter as a function that type-cases on the abstract syntax tree:

```
(define (interp expr)
   (type-case Expr expr
      [num (n) n]
      [bool (b) b]
      [add (l r) (+ (interp l) (interp r))]))
```

This is good-old PLAI practice. With a little example:

```
> (define prog (add (num 1)
                    (add (num 2) (num 3))))
> (interp prog)
6
```

Extension: New Operation

Let us now consider that we want to add a new operation on expressions. In addition to interpret an expression, we want to typecheck an expression, that is, to determine the type of value it will produce (here, either `number` or `boolean`). This is fairly trivial in our case, but still makes it possible to detect without interpretation that a program is bound to fail because it adds things that are not both numbers:

```
(define (typeof expr)
  (type-case Expr expr
    [num (n) 'number]
    [bool (b) 'boolean]
    [add (l r) (if (and (equal? 'number (typeof l))
                        (equal? 'number (typeof r)))
                   'number
                   (error "Type error: not a number"))]))
```

We can determine the type of the program we defined previously:

```
> (typeof prog)
'number
```

And see that our typechecker reject non-sensical programs:

```
> (typeof (add (num 1) (bool #f)))
Type error: not a number
```

If we reflect on this extension case, we see that it all went smoothly. We wanted a new operation, and just had to define a new function. This extension is modular, because it is defined in a single place.

Extension: New Data

We now turn to the other dimension of extensibility: adding new data variants. Suppose we want to extend our simple language with a new expression: `ifc`. We extend the datatype definition:

```
(define-type Expr
  [num  (n number?)]
  [bool (b boolean?)]
  [add (l Expr?) (r Expr?)]
  [ifc (c Expr?) (t Expr?) (f Expr?)])
```

Changing the definition of `Expr` to add this new variant breaks all existing function definitions! `interp` and `typeof` are now invalid, because they type case on expressions, but do not include any case for `ifc`. We need to modify them all to include the behavior associated to the `ifc` case:

```
(define (interp expr)
  (type-case Expr expr
    [num (n) n]
    [bool (b) b]
    [add (l r) (+ (interp l) (interp r))]
    [ifc (c t f)
         (if (interp c)
             (interp t)
             (interp f))]))

(define (typeof expr)
  (type-case Expr expr
    [num (n) 'number]
    [bool (b) 'boolean]
    [add (l r) (if (and (equal? 'number (typeof l))
                        (equal? 'number (typeof r)))
                   'number
                   (error "Type error: not a number"))]
    [ifc (c t f)
         (if (equal? 'boolean (typeof c))
             (let ((type-t (typeof t))
                   (type-f (typeof f)))
               (if (equal? type-t type-f)
                   type-t
                   (error "Type error: both branches should have
same type")))
             (error "Type error: not a boolean"))]))
```

This works:

```
> (define prog (ifc (bool false)
                    (add (num 1)
                         (add (num 2) (num 3)))
                    (num 5)))
> (interp prog)
5
```

This extensibility scenario was much less favorable. We had to modify the datatype definition and all the functions.

To summarize, with ADTs, adding new operations (eg. `typeof`) is easy and modular, but adding new data variants (eg. `ifc`) is cumbersome and non-modular.

3.4.2 OOP

How do objects perform in these extensibility scenarios?

First, we start with the object-oriented version of our interpreter:

```
(define (bool b)
  (OBJECT () ([method interp () b])))

(define (num n)
  (OBJECT () ([method interp () n])))

(define (add l r)
  (OBJECT () ([method interp () (+ (-> l interp)
                                   (-> r interp))])))
```

Note that, in line with OO design principles, each expression objects knows how to interpret itself. There is no more a centralized interpreter that deals with all expressions. Interpreting a program is done by sending the `interp` message to the program:

```
> (define prog (add (num 1)
                    (add (num 2) (num 3))))
> (-> prog interp)
6
```

Extension: New Data

Adding a new kind of data like a conditional `ifc` object can be done by simply defining a new object factory, with the definition of how these new objects handle the `interp` message:

```
(define (ifc c t f)
  (OBJECT () ([method interp ()
                      (if (-> c interp)
                          (-> t interp)
                          (-> f interp))])))
```

We can now interpret programs with conditionals:

```
> (-> (ifc (bool #f)
           (num 1)
           (add (num 1) (num 3))) interp)
4
```

This case shows that, conversely to ADTs, adding new data variants with OOP is direct and modular: just create a new (kind of) object(s). This is a clear advantage of objects over ADTs.

Extension: New Operation

But before concluding that OOP is the panacea for extensible software, we have to consider the other extension scenario: adding an operation. Suppose we now want to typecheck our programs, just as we did before. This means that expression objects should now also understand the "typeof" message. To do that, we actually have to modify all object definitions:

```
(define (bool b)
  (OBJECT () ([method interp () b]
              [method typeof () 'boolean])))

(define (num n)
  (OBJECT () ([method interp () n]
              [method typeof () 'number])))

(define (add l r)
  (OBJECT () ([method interp () (+ (-> l interp)
                                    (-> r interp))]
              [method typeof ()
                      (if (and (equal? 'number (-> l typeof))
                               (equal? 'number (-> r typeof)))
                          'number
                          (error "Type error: not a number"))])))

(define (ifc c t f)
  (OBJECT () ([method interp ()
                      (if (-> c interp)
                          (-> t interp)
                          (-> f interp))]
              [method typeof ()
                      (if (equal? 'boolean (-> c typeof))
                          (let ((type-t (-> t typeof))
                                (type-f (-> f typeof)))
                            (if (equal? type-t type-f)
                                type-t
                                (error "Type error: both branches
should have same type")))
                          (error "Type error: not a boolean"))])))
```

We can if this works:

```
> (-> (ifc (bool #f) (num 1) (num 3)) typeof)
'number
> (-> (ifc (num 1) (bool #f) (num 3)) typeof)
Type error: not a boolean
```

This extensibility scenario forced us to modify all our code to add the new methods.

To summarize, with objects, adding new data variants (eg. `ifc`) is easy and modular, but adding new operations (eg. `typeof`) is cumbersome and non-modular.

Note that this is just the dual situation of ADTs!

3.5 Different Forms of Data Abstraction

Cook's paper goes more in depth in the comparison of these forms of data abstraction. It is definitely a must-read!

ADTs and objects are different forms of data abstraction, each with its own advantages and drawbacks.

ADTs have a private representation type that prohibits tampering and extension. This is good for reasoning (analysis) and optimization. But it only permits one representation at a time.

Objects have behavioral interfaces, which allow definition of new implementations at any time. This is good for flexibility and extensibility. But it makes it hard to analyze code, and makes certain optimizations impossible.

Both forms of abstraction also support different forms of modular extensions. It is possible to modularly add new operations on an ADT, but supporting new data variants is cumbersome. It is possible to modularly add new representations to an object-oriented system, but adding new operations implies crosscutting modifications.

There are ways to navigate this tradeoff. For instance, one can expose certain implementation details in the interface of an object. That sacrifices some extensibility, but recovers the possibility to do some optimizations. The fundamental question is therefore a design question: what do I really need?

Now you understand why many languages support both kinds of data abstraction.

4 Forwarding and Delegation

Using message sending, an object can always forward a message to another object in case it does not know how to handle it. With our small object system, we can do that explicitly as follows:

```
(define seller
 (OBJECT ()
  ([method price (prod)
           (* (case prod
                ((1) (-> self price1))
                ((2) (-> self price2)))
              (-> self unit))]
   [method price1 () 100]
   [method price2 () 200]
   [method unit () 1])))

(define broker
 (OBJECT
  ([field provider seller])
  ([method price (prod) (-> provider price prod)])))

> (-> broker price 2)
200
```

Object `broker` does not know how to compute the price of a product, but it can claim to do so by implementing a method to handle the `price` message, and simply forwarding it to `seller`, who does implement the desired behavior. Note how `broker` holds a reference to `seller` in its `provider` field. This is a typical example of object composition, with message forwarding.

Now, you can see that the problem with this approach is that this forwarding of messages has to be explicit: for each message that we anticipate might be sent to `broker`, we have to define a method that forwards the message to `seller`. For instance:

```
> (-> broker unit)
```
message not understood: unit

4.1 Message Forwarding

We can do better by allowing each object to have a special "partner" object to which it automatically forwards any message it does not understand. We can define a new syntactic abstraction, `OBJECT-FWD`, for constructing such objects:

```
(defmac (OBJECT-FWD target
                    ([field fname init] ...)
                    ([method mname args body] ...))
  #:keywords field method
  #:captures self
  (letrec ([self
            (let ([fname init] ...)
              (let ([methods (list (cons 'mname (λ args body)) ...)])
                (λ (msg . vals)
                  (let ([found (assoc msg methods)])
                    (if found
                        (apply (cdr found) vals)
                        (apply target msg vals))))))])
    self))
```

Note how the syntax is extended to specify a `target` object; this object is used in the dispatch process whenever a message is not found in the methods of the object. Of course, if all objects forward unknown message to some other object, there has to be a last object in the chain, that simply fails when sent a message:

```
(define root
  (λ (msg . args)
    (error "not understood" msg)))
```

Now, `broker` can be defined simply as follows:

```
(define broker
  (OBJECT-FWD seller () ()))
```

That is, `broker` is an empty object (no fields, no methods) that forwards all messages sent to it to `seller`:

```
> (-> broker price 2)
200
> (-> broker unit)
1
```

This kind of objects is often called a *proxy*.

4.2 Delegation

Suppose we want to use `broker` to *refine* the behavior of `seller`; say, we want to double the price of every product, by changing the unit used in the calculation of the prices. This is easy: we just have to define a method `unit` in `broker`:

```
(define broker
  (OBJECT-FWD seller ()
   ([method unit () 2])))
```

With this definition, we should make sure that asking the price of a product to `broker` is twice the price of the same product asked to `seller`:

```
> (-> broker price 1)
100
```

Hmmm... it does not work! It seems that once we forward the `price` message to `seller`, `broker` never gets the control back; in particular, the `unit` message that `seller` sends to `self` is *not* received by `broker`.

Let us consider why this is so. To which object is `self` bound in `seller`? To `seller`! Remember, we said previously (see §2 "Looking for the Self") that in our approach, `self` is *statically bound*: when an object is created, `self` is made to refer to the object/closure that is being defined, and will always remain bound to it. This is because `letrec`, like `let`, respects lexical scoping.

What we are looking for is another semantics, called *delegation*. Delegation requires `self` in an object to be *dynamically bound*: it should always refer to the object that originally received the message. In our example, this would ensure that when `seller` sends `unit` to `self`, then `self` is bound to `broker`, and thus the re-definition of `unit` in `broker` takes effect. In that case, we say that `seller` is the *parent* of `broker`, and that `broker` *delegates* to its parent.

How do we bind an identifier such that it refers to the value at the point of usage, rather than at the point of definition? In the absence of dynamically-scoped binding forms, the only way we can achieve this is by passing that value as parameter. So, we have to parameterize methods by the actual receiver. Therefore, instead of capturing the `self` identifier in their surrounding lexical scope, they are parameterized by `self`.

Concretely, this means that the method:

```
(λ (prod) .... (-> self unit) ....)
```

in `seller` must be kept in the methods list as:

```
(λ (self)
  (λ (prod)....(-> self unit)....))
```

Ever wondered why methods in Python must explicitly receive `self` as a first parameter?

This parameterization effectively allows us to pass the current receiver after we lookup the method.

Let us now define a new syntactic form, `OBJECT-DEL`, to support the delegation semantics between objects:

```
(defmac (OBJECT-DEL parent
                    ([field fname init] ...)
                    ([method mname args body] ...))
  #:keywords field method
  #:captures self
  (let ([fname init] ...)
    (let ([methods
           (list (cons 'mname
                       (λ (self) (λ args body))) ...)])
      (λ (current)
        (λ (msg . vals)
          (let ([found (assoc msg methods)])
            (if found
                (apply ((cdr found) current) vals)
                (apply (parent current) msg vals))))))))
```

Several things have changed: first, we renamed `target` to `parent`, to make it clear that we are defining a delegation semantics. Second, all methods are now parameterized by `self`, as explained above. Note that we got rid of `letrec` altogether! This is because `letrec` was used precisely to allow objects to refer to `self`, but following lexical scoping. We have seen that for delegation, lexical scoping is not what we want.

This means that when we find a method in the method dictionary, we must first give it the actual receiver as argument. How are we going to obtain that receiver? well, the only possiblity is to parameterize objects by the current receiver they have to use when applying methods. That is to say, the value returned by the object construction form is not a "`λ (msg . vals) ....`" anymore, but a "`λ (rcvr) ....`". This effectively parameterizes our objects by "the current receiver". Similarly, if a message is not understood by a given object, then it must send the current receiver along to its parent.

This leaves us with one final question to address: how do we send a message to an object in the first place? Remember that our definition of `->` is:

```
(defmac (-> o m arg ...)
  (o 'm arg ...))
```

But now we cannot apply `o` as a function that takes a symbol (the message) and a variable number of arguments. Indeed, an object now is a function of the form `(λ (rcvr) (λ (msg . args) ....))`. So before we can pass the message and the arguments, we have to specify which object is the current receiver. Well, it's easy, because at the time we are sending a message, the current receiver should be... the object we are sending the message to!

Why is the `let` binding necessary?

```
(defmac (-> o m arg ...)
  (let ([obj o])
    ((obj obj) 'm arg ...)))
```

Let us see delegation—that is, late binding of self—at work:

```
(define seller
 (OBJECT-DEL root ()
  ([method price (prod)
           (* (case prod
                [(1) (-> self price1)]
                [(2) (-> self price2)])
              (-> self unit))]
   [method price1 () 100]
   [method price2 () 200]
   [method unit () 1])))
(define broker
 (OBJECT-DEL seller ()
  ([method unit () 2])))

> (-> seller price 1)
100
> (-> broker price 1)
200
```

4.3 Programming with Prototypes

Object-based languages with a delegation mechanism like the one we have introduced in this chapter are called *prototype-based languages*. Examples are Self, JavaScript, and AmbientTalk, among many others. What are these languages good at? How to program with prototypes?

4.3.1 Singleton and Exceptional Objects

Since objects can be created *ex-nihilo* (ie. out of an object literal expression like `OBJECT-DEL`), it is natural to create one-of-a-kind objects. As opposed to class-based languages that require a specific design pattern for this (called Singleton), object-based languages are a natural fit for this case, as well as for creating "exceptional" objects (more on this below).

Let us first consider the object-oriented representation of booleans and a simple if-then-else control structure. How many booleans are there? well, two: true, and false. So we can create two standalone objects, `true` and `false` to represent them. In pure object-oriented languages like Self and Smalltalk, control structures like if-then-else, while, etc. are not primitives in the language. Rather, they are defined as methods on appropriate objects. Let us consider the if-then-else case. We can pass two thunks to a boolean, a truth thunk and a falsity thunk; if the boolean *is* true, it applies the truth thunk; if it *is* false, it applies the falsity thunk.

```
(define true
  (OBJECT-DEL root ()
    ([method ifTrueFalse (t f) (t)])))

(define false
  (OBJECT-DEL root ()
    ([method ifTrueFalse (t f) (f)])))
```

How can we use such objects? Look at the following example:

```
(define light
 (OBJECT-DEL root
   ([field on false])
   ([method turn-on () (set! on true)]
    [method turn-off () (set! on false)]
    [method on? () on])))

> (-> (-> light on?) ifTrueFalse (λ () "light is on")
                                 (λ () "light is off"))
"light is off"
> (-> light turn-on)
> (-> (-> light on?) ifTrueFalse (λ () "light is on")
                                 (λ () "light is off"))
"light is on"
```

The objects `true` and `false` are the two unique representants of boolean values. Any conditional mechanism that relies on some expression being true or false can be similarly defined as methods of these objects. This is indeed a nice example of dynamic dispatch!

Boolean values and control structures in Smalltalk are defined similarly, but because Smalltalk is a class-based language, their definitions are more complex. Try it in your favorite class-based language.

Let us look at another example where object-based languages are practical: exceptional objects. First, recall the definition of typical point objects, which can be created using a factory function `make-point`:

```
(define (make-point x-init y-init)
  (OBJECT-DEL root
    ([field x x-init]
     [field y y-init])
    ([method x? () x]
     [method y? () y])))
```

Suppose we want to introduce an exceptional point object that has the particularity of having *random* coordinates, that change each time they are accessed. We can simply define this random-point as a standalone object whose x? and y? methods perform some computation, rather than accessing stored state:

```
(define random-point
  (OBJECT-DEL root ()
    ([method x? () (* 10 (random))]
     [method y? () (-> self x?)])))
```

Note that random-point does not have any fields declared. Of course, because in OOP we rely on object interfaces, both representations of points can coexist.

4.3.2 Sharing through Delegation

The examples discussed above highlight the advantages of object-based languages. Let us now look at delegation in practice. First, delegation can be used to factor out *shared behavior* between objects. Consider the following:

```
(define (make-point x-init y-init)
  (OBJECT-DEL root
    ([field x x-init]
     [field y y-init])
    ([method x? () x]
     [method y? () y]
     [method above (p2)
             (if (> (-> p2 y?) (-> self y?))
                 p2
                 self)]
     [method add (p2)
             (make-point (+ (-> self x?)
                            (-> p2 x?))
                         (+ (-> self y?)
                            (-> p2 y?)))])))
```

All created point objects have the same methods, so this behavior could be shared by moving it to a common parent of all point objects (often called a prototype). Should all behavior be moved in the prototype? well, not if we want to allow different representations of points, like the random point above (which does not have any field at all!).

Therefore, we can define a point prototype, which factors out the above and add methods, whose implementation is common to all points:

```
(define point
  (OBJECT-DEL root ()
    ([method above (p2)
```

```
              (if (> (-> p2 y?) (-> self y?))
                  p2
                  self)]
      [method add (p2)
              (make-point (+ (-> self x?)
                             (-> p2 x?))
                          (+ (-> self y?)
                             (-> p2 y?)))])))
```

Note that as a standalone object, `point` does not make sense, because it sends messages to itself that it does not understand. But it can serve as a prototype from which different points can extend. Some are typical points, created with `make-point`, which hold two fields `x` and `y`:

The required accessor methods could be declared as `abstract` methods in `point`, if our language supported such a concept. In Smalltalk, one would define the methods in `point` such that they throw an exception if invoked.

```
(define (make-point x-init y-init)
  (OBJECT-DEL point
    ([field x x-init]
     [field y y-init])
    ([method x? () x]
     [method y? () y])))
```

While some can be exceptional points:

```
(define random-point
  (OBJECT-DEL point ()
    ([method x? () (* 10 (random))]
     [method y? () (-> self x?)])))
```

As we said, these different kinds of point can cooperate, and they all understand the messages defined in the `point` prototype:

```
> (define p1 (make-point 1 2))
> (define p2 (-> random-point add p1))
> (-> (-> p2 above p1) x?)
1.190463175442149
```

We can similarly use delegation to share *state* between objects. For instance, consider a family of points that share the same x-coordinate:

```
(define 1D-point
  (OBJECT-DEL point
    ([field x 5])
    ([method x? () x]
     [method x! (nx) (set! x nx)])))
```

```
(define (make-point-shared y-init)
  (OBJECT-DEL 1D-point
    ([field y y-init])
    ([method y? () y]
     [method y! (ny) (set! y ny)])))
```

All objects created by make-point-shared share the same parent, 1D-point, which determines their x-coordinate. If a change to 1D-point is made, it is naturally reflected in all its children:

```
> (define p1 (make-point-shared 2))
> (define p2 (make-point-shared 4))
> (-> p1 x?)
5
> (-> p2 x?)
5
> (-> 1D-point x! 10)
> (-> p1 x?)
10
> (-> p2 x?)
10
```

4.4 Late Binding of Self and Modularity

See the Why of Y.

In the definition of the OBJECT-DEL syntactic abstraction, notice that we use, in the definition of message sending, the self-application pattern (obj obj). This is similar to the self application pattern we have seen to achieve recursive binding without mutation.

Think about what you know from mainstream languages like C++ and Java: what means do these two languages provide to address this tradeoff between extensibility and fragility?

This feature of OOP is also known as "open recursion": any sub-object can redefine the meaning of a method in one of its parents. Of course, this is a mechanism that favors *extensibility*, because it is possible to extend any aspect of an object without having to foresee that extension. On the other hand, open recursion also makes software more *fragile*, because it becomes extremely easy to extend an object in unforeseen, incorrect ways. Imagine scenarios where this can be problematic and think about possible alternative designs. To shed some more light on fragility, think about black-box composition of objects: taking two objects, developed independently, and then putting them in a delegation relation with each other. What issues can arise?

4.5 Lexical Scope and Delegation

As we have seen previously, we can define nested objects in our system. It is interesting to examine the relation between lexical nesting and delegation. Consider the following example:

```
(define parent
 (OBJECT-DEL root ()
   ([method foo () 1])))

(define outer
 (OBJECT-DEL root
    ([field foo (λ () 2)])
    ([method foo () 3]
     [method get ()
             (OBJECT-DEL parent ()
                ([method get-foo1 () (foo)]
                 [method get-foo2 () (-> self foo)]))])))

(define inner (-> outer get))

> (-> inner get-foo1)
2
> (-> inner get-foo2)
1
```

As you can see, a free identifier is looked up in the lexical environment (see `get-foo1`), while an unknown message is looked up in the chain of delegation (see `get-foo2`). This is important to clarify, because Java programmers are used to the fact that `this.foo ()` is the same as `foo ()`. In various languages that combine lexical nesting and some form of delegation (like inheritance), this is not the case.

Other languages have different take on the question. Check out for Newspeak and AmbientTalk.

So what happens in Java? Try it! You will see that the inheritance chain shadows the lexical chain: when using `foo ()` if a method can be found in a superclass, it is invoked; only if there is no method found, the lexical environment (i.e., the outer object) is used. Referring to the outer object is therefore very fragile. This is why Java also supports an additional form `Outer.this` to refer to the enclosing object. Of course, then, if the method is not found directly in the enclosing object's class, it is then looked up in its superclass, rather than upper in the lexical chain.

4.6 Delegation Models

The delegation model we have implemented here is but one point in the design space of prototype-based languages. Study the documentation of Self, JavaScript, and AmbientTalk to understand their designs. You can even modify our object system to support a different model, such as the JavaScript model.

4.7 Cloning

In our language, as in JavaScript, the way to create objects is *ex-nihilo*: either we create an object from scratch, or we have a function whose role is to perform these object creations for us. Historically, prototype-based languages (like Self) have provided another way to create objects: by cloning existing objects. This approach emulates the copy-paste-modify metaphor we use so often with text (including code!): start from an object that is almost as you need, clone it, and modify the clone (eg. add a method, change a field).

When cloning objects in presence of delegation, the question of course arises of whether the cloning operation should be *deep* or *shallow*. Shallow cloning returns a new object that delegates to the same parent as the original object. Deep cloning returns a new object that delegates to a clone of the original parent, and so on: the whole delegation chain is cloned.

We won't study cloning in more details in this course. You should however wonder how easy it would be to support cloning in our language. Since objects are in fact compiled into procedures (through macro expansion), the question boils down to cloning closures. Unfortunately, Scheme does not support such an operation. This is a case where the mismatch between the source and target languages shows up (recall Chapter 11 of PLAI). Nothing is perfect!

5 Classes

Let's go back to the factory function (see §1.3 "Constructing Objects"):

```
(define (make-point init-x)
  (OBJECT
   ([field x init-x])
   ([method x? () x]
    [method x! (new-x) (begin (set! x new-x) self)])))

(define p1 (make-point 0))
(define p2 (make-point 1))
```

All point objects get their own version of the methods, even though they are the same. Well, at least their signature and body are the same. Are they *completely* the same though? They are not, in fact. The only difference, in this version of the object system, is that each method closes over the self of each object: i.e., `self` in a method in `p1` refers to `p1`, while it refers to `p2` in the method of `p2`. In other words, the methods, which are functions, differ by the lexical environment they capture.

5.1 Sharing Method Definitions

Instead of duplicating all method definitions just to be able to support different selves, it makes much more sense to factor out the common part (the method bodies), and parameterize them by the variable part (the object bound to `self`).

Let us try first without macros. Recall that our definition of a plain point object without macros is as follows:

```
(define make-point
  (λ (init-x)
    (letrec ([self
              (let ([x init-x])
                (let ([methods (list (cons 'x? (λ () x))
                                     (cons 'x! (λ (nx)
                                                 (set! x nx)
                                                 self)))])
                  (λ (msg . args)
                    (apply (cdr (assoc msg methods)) args))))])
      self)))
```

If we hoist the `(let ([methods...]))` out of the `(λ (init-x) ...)`, we effectively achieve the sharing of method definitions we are looking for. But, field variables are now out of scope of method bodies. More concretely, here, it means that `x` will be unbound in both methods. This means that we need to parameterize methods by state (field values) as well, in addition to self. But, fair enough, self can "hold" the state (it can capture field bindings in its lexical environment). We just need a way to extract (and potentially assign) field values through self. For that, we are going to have objects support two specific messages `-read` and `-write`:

```
(define make-point
  (let ([methods (list (cons 'x? (λ (self)
                                  (λ () (self '-read))))
                       (cons 'x! (λ (self)
                                  (λ (nx)
                                    (self '-write nx)
                                    self))))])
    (λ (init-x)
      (letrec ([self
                (let ([x init-x])
                  (λ (msg . args)
                    (case msg
                      [(-read) x]
                      [(-write) (set! x (first args))]
                      [else
                       (apply ((cdr (assoc msg methods)) self) args)])))])
        self))))
```

See how the two methods are now parameterized by `self`, and that in order to read/assign to a field, they send a special message to `self`. Now let us examine the definition of the object itself: when sent a message, it first checks if the message is either `-read` or `-write`, in which case it either returns `x` or assigns it. Let us see if this works:

```
(define p1 (make-point 1))
(define p2 (make-point 2))

> ((p1 'x! 10) 'x?)
10
> (p2 'x?)
2
```

5.2 Accessing Fields

Of course, our definition is not very general, because it only works for the one field `x`. We need to generalize: field names must be passed as arguments to the `-read` and `-write` messages. The issue is then how to go from a field name (as a symbol) to actually reading/assigning the variable with the same name in the lexical environment of the object. A simple solution is to use a structure to hold field values. This is similar to the way we handle method definitions already: an association between method names and method definitions. However, unlike in a method table, field bindings are (at least potentially) mutable. Racket does not allow mutation in association lists, so we will use a dictionary (more precisely, hashtable), which is accessed with `dict-ref` and `dict-set!`.

```
(define make-point
  (let ([methods (list (cons 'x? (λ (self)
                                   (λ () (self '-read 'x))))
                       (cons 'x! (λ (self)
                                  (λ (nx)
                                    (self '-write 'x nx)
                                    self))))])
    (λ (init-x)
      (letrec ([self
                (let ([fields (make-hash (list (cons 'x init-
x)))])
                  (λ (msg . args)
                    (case msg
                      [(-read)  (dict-ref  fields (first args))]
                      [(-write) (dict-set! fields (first args)
                                                  (second args))]
                      [else
                       (apply ((cdr (assoc msg methods)) self) args)])))])
        self))))

> (let ((p1 (make-point 1))
        (p2 (make-point 2)))
    (+ ((p1 'x! 10) 'x?)
       (p2 'x?)))
12
```

Note how `make-point` now holds the list of methods definitions, and the created object captures a dictionary of `fields` (which is initialized prior to returning the object).

5.3 Classes

While we did achieve the sharing of method definitions we were after, our solution is still not very satisfactory. Why? Well, observe the definition of an object (the body of the `(λ (msg . args) ....)` above). The logic that is implemented there is, again, repeated in all objects we create with `make-point`: each object has its own copy of what to do when it is sent a `-read` message (lookup in the `fields` dictionary), a `-write` message (assign in the `fields` dictionary), or any other message (looking in the `methods` table and then applying the method).

So, all this logic could very well be shared amongst objects. The only free variables in the object body are `fields` and `self`. In other words, we could define an object as being just its self as well as its fields, and leave all the other logic to the `make-point` function. In that case `make-point` starts to have more than one responsability: it is no longer only in charge of creating new objects, it is also in charge of handling accesses to fields and message handling. That is, `make-point` is now evolving into what is called a *class*.

How are we going to represent a class? well, for now it is just a function that we can apply (and it creates an object—an *instance*); if we need that function to have different behaviors, we can apply the same Object Pattern we saw at the beginning of this course.

In some languages, classes are objects in their own right. The paradigmatic example in this regard is Smalltalk. Definitely worth a detour!

That is:

```
(define Point
  ....
  (λ (msg . args)
    (case msg
      [(create) create instance]
      [(read) read field]
      [(write) write field]
      [(invoke) invoke method])))
```

This pattern makes clear what the role of a class is: it produces objects, and invokes methods, reads and writes fields on its instances.

What is the role of an object now? well, it is just to have an identity, know its class, and hold the values of its fields. It does not hold any behavior on its own, anymore. In other words, we can define an object as a plain data structure:

```
(define-struct obj (class values))
```

Let us see exactly how we can define the class `Point` now:

```
(define Point
  (let ([methods ....])
    (letrec
        ([class
             (λ (msg . vals)
               (case msg
                 [(create) (let ((values (make-hash '((x . 0)))))
                             (make-obj class values))]
                 [(read) (dict-ref (obj-values (first vals))
                                   (second vals))]
                 [(write) (dict-set! (obj-values (first vals))
                                     (second vals)
                                     (third vals))]
```

```
                 [(invoke)
                   (let ((found (assoc (second vals) methods)))
                     (if found
                         (apply ((cdr found) (first vals)) (cddr vals))
                         (error "message not understood"))))])))])
      class)))
```

```
> (Point 'create)
#<obj>
```

We can instantiate the class Point by sending it the create message. Now that an object is a structure, we need a different approach to sending messages to it, as well as to access its fields. To send a message to an object p, we must first retrieve its class, and then send the invoke message to the class:

```
((obj-class p) 'invoke p 'x?)
```

And similarly for reading and accessing fields.

5.4 Embedding Classes in Scheme

Let us now embed classes in Scheme using macros.

5.4.1 Macro for Classes

We define a CLASS syntactic abstraction for creating classes:

```
(defmac (CLASS ([field f init] ...)
               ([method m params body] ...))
     #:keywords field method
     #:captures self
     (let ([methods (list (cons 'm (λ (self)
                                     (λ params body))) ...)])
       (letrec
           ([class
                (λ (msg . vals)
                  (case msg
                    [(create)
                     (make-obj class
                               (make-hash (list (cons 'f init) ...)))]
                    [(read)
```

```
                     (dict-ref (obj-values (first vals)) (second vals))]
                    [(write)
                     (dict-set! (obj-values (first vals)) (second vals) (third vals))]
                    [(invoke)
                     (if (assoc (second vals) methods)
                         (apply ((cdr (assoc (second vals) methods)) (first vals)) (cddr val
                         (error "message not understood"))]))])
          class)))
```

5.4.2 Auxiliary Syntax

We need to introduce a new definition for the convenient syntax to invoke methods (`->`), and introduce similar syntax for accessing the fields of the current object (`?` and `!`).

```
(defmac (-> o m arg ...)
  (let ((obj o))
    ((obj-class obj) 'invoke obj 'm arg ...)))


(defmac (? fd) #:captures self
  ((obj-class self) 'read self 'fd))

(defmac (! fd v) #:captures self
  ((obj-class self) 'write self 'fd v))
```

We can also define an auxiliary function to create new instances:

```
(define (new c)
  (c 'create))
```

This simple function is conceptually very important: it helps to hide the fact that classes are internally implemented as functions, as well as the actual symbol used to ask a class to create an instance.

5.4.3 Example

Let us see classes at work:

```
(define Point
 (CLASS ([field x 0])
        ([method x? () (? x)]
         [method x! (new-x) (! x new-x)]
         [method move (n) (-> self x! (+ (-> self x?) n))])))
```

```
(define p1 (new Point))
(define p2 (new Point))

> (-> p1 move 10)
> (-> p1 x?)
10
> (-> p2 x?)
0
```

5.4.4 Strong Encapsulation

We have made an important design decision with respect to field accesses: field accessors `?` and `!` only apply to `self`! i.e., it is not possible in our language to access fields of another object. This is called a language with *strongly-encapsulated* objects. Smalltalk follows this discipline (accessing a field of another object is actually a message send, which can therefore be controled by the receiver object). Java does not: it is possible to access the field of any object (provided visibility allows it). Here, our *syntax* simply does not allow foreign field accesses.

Another consequence of our design choice is that field accesses should only occur *within* method bodies: because the receiver object is always `self`, `self` must be defined. For instance, look at what happen if we use the field read form `?` outside of an object:

```
> (? f)
self: undefined;
 cannot reference undefined identifier
```

It would be much better if the above could yield an error saying that `?` is undefined. In order to do this, we can simply introduce `?` and `!` as *local* syntactic forms, only defined within the confines of method bodies, instead of global ones. We do that by moving the definition of these field access forms from the top-level to a `local` scope, surrounding method definitions:

```
(defmac (CLASS ([field f init] ...)
               ([method m params body] ...))
     #:keywords field method
     #:captures self ? !
     (let ([methods
            (local [(defmac (? fd) #:captures self
                      ((obj-class self) 'read self 'fd))
                    (defmac (! fd v) #:captures self
                      ((obj-class self) 'write self 'fd v))]
              (list (cons 'm (λ (self)
                               (λ params body))) ...))])
```

```
(letrec
   ([class (λ (msg . vals) ....)]))))
```

Defining the syntactic forms `?` and `!` locally, for the scope of the definition of the list of methods only, ensures that they are available to use within method bodies, but nowhere else.

Now, field accessors are no longer defined outside of methods:

```
> (? f)
?: undefined;
 cannot reference undefined identifier
```

From now on, we will use this local approach.

5.5 Initialization

As we have seen, the way to obtain an object from a class, i.e., to instantiate it, is to send the class the `create` message. It is generally useful to be able to pass arguments to `create` in order to specify the initial values of the fields of the object. For now, our class system only supports the specification of default field values at class-declaration time. It is not possible to pass initial field values at instantiation time.

Initializer methods are a typical programming idiom in Smalltalk. In Java, these are known as constructors (That's arguably a bad name, because as we can see, they are not in charge of the construction of the object–only of its initialization after the object is actually created).

There are several ways to do this. A simple way is to require objects to implement an *initializer* method, and have the class invoke this initializer method on each newly-created object. We will adopt the following convention: if no argument is passed with the `create` message, then we do not call the initializer (and therefore use the default values). If arguments are passed, we invoke the initializer (called `initialize`) with the arguments:

```
....
(λ (msg . vals)
  (case msg
    [(create)
     (if (null? vals)
         (make-obj class
                   (make-hash (list (cons 'f init) ...)))
         (let ((object (make-obj class (make-hash))))
           (apply ((cdr (assoc 'initialize methods)) object) vals)
           object))]
    ....)) ....
```

We can refine the auxiliary function to instantiate classes such that it accepts a variable number of arguments:

```
(define (new class . init-vals)
    (apply class 'create init-vals))
```

Let us see if it works as expected:

```
(define Point
 (CLASS ([field x 0])
        ([method initialize (nx) (-> self x! nx)]
         [method x? () (? x)]
         [method x! (nx) (! x nx)]
         [method move (n) (-> self x! (+ (-> self x?) n))])))

(define p (new Point 5))

> (-> p move 10)
> (-> p x?)
15
```

5.6 Anonymous, Local and Nested Classes

We have introduced classes in our extension of Scheme, in such a way that classes are, like objects in our earlier systems, represented as first-class functions. This means therefore that classes in our language are first-class entities, which can, for instance, be passed as parameter (see the definition of the `create` function above). Other consequences are that our system also supports both anonymous and nested classes. Of course, all this is achieved while respecting the rules of lexical scoping.

```
(define (cst-class-factory cst)
 (CLASS () ([method add (n) (+ n cst)]
            [method sub (n) (- n cst)]
            [method mul (n) (* n cst)])))

(define Ops10  (cst-class-factory 10))
(define Ops100 (cst-class-factory 100))

> (-> (new Ops10) add 10)
20
> (-> (new Ops100) mul 2)
200
```

We can also introduce classes in a local scope. That is, as opposed to languages where classes are first-order entities that are globally visible, we are able to define classes locally.

```
(define doubleton
  (let ([the-class (CLASS ([field x 0])
                          ([method initialize (x) (-> self x! x)]
                           [method x? () (? x)]
```

```
                              [method x! (new-x) (! x new-x)]))])
      (let ([obj1 (new the-class 1)]
            [obj2 (new the-class 2)])
        (cons obj1 obj2))))

> (-> (cdr doubleton) x?)
2
```

In the above we introduce `the-class` only for the sake of creating two instances and returning them in a pair. After that point, the class is not accessible anymore. In other words, it is impossible to create more instances of that class. However, of course, the two instances we created still refer to their class, so the objects can be used. Interestingly, once these objects are garbage collected, their class as well can be reclaimed.

6 Inheritance

In the presence of classes, we may want a mechanism similar to §4.2 "Delegation" in order to be able to reuse and selectively refine existing classes. We therefore extend our object system with support for *class inheritance*. As we will see, many issues have to be dealt with. As usual, we will proceed in a gradual manner.

6.1 Class Hierarchy

We introduce the capability for a class to *extend* another class (called its *superclass*). We focus on *single inheritance*, where a class only extends a single class.

Multiple inheritance. C++

As a result, classes are organized in a hierarchy. The superclasses of a class (transitively) are called its *ancestors*; dually, the set of transitive *subclasses* of a class are called its *descendants*.

For example:

```
(define Point
  (CLASS extends Root
         ([field x 0])
         ([method x? () (? x)]
          [method x! (new-x) (! x new-x)]
          [method move (n) (-> self x! (+ (-> self x?) n))])))

(define ColorPoint
  (CLASS extends Point
         ([field color 'black])
         ([method color? () (? color)]
          [method color! (clr) (! color clr)])))
```

6.2 Method Lookup

When we send a message to an object, we look in its class for a method that implements the message, and then we apply it. This is reflected in our CLASS macro definition as:

```
[(invoke)
 (if (assoc (second vals) methods)
     (apply ((cdr (assoc (second vals) methods)) (first vals)) (cddr vals))
     (error "message not understood"))]
```

With inheritance, if an object is sent a message for which a method is not found in its class, we can look for a method in the superclass, and so on. We will first refine the invoke protocol and cut it in two steps: first, a `lookup` step, which includes going to lookup in superclasses if no method is found in the current class, and then the actual `invoke` step.

```
(defmac (CLASS extends superclass
               ([field f init] ...)
               ([method m params body] ...))
  #:keywords field method extends
  #:captures self ? !
  (let ([scls superclass]
        (methods
         (local [(defmac (? fd) #:captures self
                   ((obj-class self) 'read self 'fd))
                 (defmac (! fd v) #:captures self
                   ((obj-class self) 'write self 'fd v))]
           (list (cons 'm (λ (self)
                            (λ params body))) ...))))
    (letrec ([class (λ (msg . vals)
                      (case msg
                        ....
                        [(invoke)
                         (let ((method (class 'lookup (second vals))))
                           (apply (method (first vals)) (cddr vals)))]
                        [(lookup)
                         (let ([found (assoc (first vals) methods)])
                           (if found
                               (cdr found)
                               (scls 'lookup (first vals))))]))])
      class)))
```

The `CLASS` syntactic abstraction is extended with an `extends` clause (a new keyword in a class definition). Before we can try this out, we need to define a *root* class at the top of the tree, in order to put an end to the method lookup process. The `Root` class below does just that:

```
(define Root
  (λ (msg . vals)
    (case msg
      [(lookup) (error "message not understood:" (first vals))]
      [else     (error "root class: should not happen: " msg)])))
```

`Root` is implemented directly as a function, without using the `CLASS` form, so that we don't need to specify its superclass (it has none). In case it is sent a `lookup` message, it gives an error that the message is not understood. Note that in this system, it is an error to send any message that is not `lookup` to the root class.

Let us see an example of class inheritance at work, first with a very simple example:

```
(define A
  (CLASS extends Root ()
         ([method foo () "foo"]
          [method bar () "bar"])))
(define B
  (CLASS extends A ()
         ([method bar () "B bar"])))

> (define b (new B))
> (-> b foo)
"foo"
> (-> b bar)
"B bar"
```

This looks just fine: sending a message that is unknown in `B` works as expected, and sending `bar` also results in `B`'s refinement to be executed instead of `A`'s method. That is, method invocation is properly *late bound*. We say that method `bar` in `B` *overrides* the method of the same name defined in `A`.

Let us look at a slightly more complex example:

```
> (define p (new Point))
> (-> p move 10)
> (-> p x?)
10
```

and now with a `ColorPoint`:

```
> (define cp (new ColorPoint))
> (-> cp color! 'red)
> (-> cp color?)
'red
> (-> cp move 5)
hash-ref: no value found for key
  key: 'x
```

What happened? It seems that we are not able to use field `x` of a colored point. Fair enough, we haven't dealt at all about how fields have to be handled in the presence of inheritance.

6.3 Fields and Inheritance

Let us look again at how we handle object creation at the moment:

```
[(create)
 (make-obj class
           (make-hash (list (cons 'f init) ...)))]
```

That is it: we are only initializing the values dictionary for the fields that are declared in the current class! We ought to be initializing the dictionary with values for the fields of the ancestor classes as well.

6.3.1 Inheriting Fields

An object should have values for all the fields declared in any of its ancestors. Therefore, when we create a class, we should determine all the fields of its instances. To do that, we have to extend classes such that they keep a list of all their fields, and are able to provide that information to any subclass that requires it.

```
(defmac (CLASS extends superclass
               ([field f init] ...)
               ([method m params body] ...))
  #:keywords field method extends
  #:captures self ? !
  (let* ([scls superclass]
         [methods ....]
         [fields (append (scls 'all-fields)
                         (list (cons 'f init) ...))])
    (letrec
        ([class (λ (msg . vals)
                  (case msg
                    [(all-fields) fields]
                    [(create) (make-obj class
                                        (make-hash fields))]
                    ....))]))))
```

We introduce a new fields identifier in the lexical environment of the class. This identifier is bound to the complete list of fields that the class' instances should have. All fields of the superclass are obtained by sending it the all-fields message (whose implementation simply returns the list bound to fields). When creating an object, we just make a fresh dictionary with all the fields.

Because we have added a new message to the vocabulary of classes, we need to wonder what happens if this message is sent to the Root class: what are all the fields of that class? Well, it has to be the empty list, since we are using append blindly:

```
(define Root
  (λ (msg . vals)
```

```
(case msg
  [(lookup)     (error "message not understood:" (first vals))]
  [(all-fields) '()]
  [else (error "root class: should not happen: " msg)])))
```

Let us see if this works:

```
> (define cp (new ColorPoint))
> (-> cp color! 'red)
> (-> cp color?)
'red
> (-> cp move 5)
> (-> cp x?)
5
```

Good!

6.3.2 Binding of Fields

Actually, there is still one more issue that we haven't considered: what happens if a subclass defines a field with a name that already exists in one of its ancestors?

```
(define A
 (CLASS extends Root
        ([field x 1]
         [field y 0])
        ([method ax () (? x)])))
(define B
  (CLASS extends A
         ([field x 2])
         ([method bx () (? x)])))

> (define b (new B))
> (-> b ax)
2
> (-> b bx)
2
```

In both cases, we get the value bound to the `x` field of `B`. In other words, we have late binding of fields, exactly as we do for methods. Is that reasonable?

strong encapsulation

Let us see: an object is meant to encapsulate some (possibly mutable) state behind a proper procedural interface (methods). It is clear that late binding of methods is a desirable property, because methods are what makes an object's external interface. What about fields? Fields are supposed to be hidden, internal state of the object—in other words, implementation details, not public interface. Actually, notice that in our language so far, it is not even possible to access a field of another object other than `self`! So at the very least, late binding of fields is doubtful.

should private methods be late bound? are they?

Let us look at what happened with §4.2 "Delegation". How are fields handled there? Well, fields are just free variables of a function, so they are *lexically scoped*. This is a much more reasonable semantics for fields. When methods are defined in a class, they are defined in terms of the fields that are directly defined in that class, or one of its superclass. This makes sense, because all that is information known at the time one writes a class definition. Having late binding of fields means reintroducing dynamic scoping for all free variables of a method: an interesting source of errors and headaches! (Think of examples of subclasses mixing up their superclasses by accidentally introducing fields with existing names.)

6.3.3 Field Shadowing

We now see how to define the semantics known as *field shadowing*, in which a field in a class shadows a field of the same name in a superclass, but a method always accesses a field as declared in the class or one of its ancestors.

Concretely, this means that an object can hold different values for fields of the same name; which one to use depends on the class in which the executing method is defined (this is know as the *host class* of the method). Because of this multiplicity, it is not possible to use a hash table anymore. Instead, we will keep in a class the list of the field names, and in the object, a *vector* of values, accessed by position. A field access will be done in two steps: first determining the position of the field according to the list of names, and then accessing the value in the vector held by the object.

For instance, for class `A` above, the list of names is `'(x y)` and the vector of values in a new instance of `A` is `#(1 0)`. For class `B`, the list of names is `'(x y x)` and the vector of values in a new instance is `#(1 0 1)`. The advantage of keeping the fields ordered this way is that, if not shadowed, a field is always at the same position within an object.

To respect the semantics of shadowing, we have (at least) two options. We can rename shadowed fields to a mangled name, for instance `'(x0 y x)` in `B` so that methods hosted in `B` and its descendants only see the latest definition of `x`, that is, the field introduced in `B`. An other alternative is to keep the field names unchanged, but to perform lookup starting from the end: in other words, we will want to find the *last* position of a field name in the list of names. We will go for the latter.

We can update our definition of `CLASS` so as to introduce the vector and the field lookup strategy:

```
....
```

```
[(create)
 (let ([values (list->vector (map cdr fields))])
   (make-obj class values))]
[(read)
 (vector-ref (obj-values (first vals))
             (find-last (second vals) fields))]
[(write)
 (vector-set! (obj-values (first vals))
              (find-last (second vals) fields)
              (third vals))]
....
```

We obtain a vector (with the initial field values) and construct the object with it. Then, for field access, we access the vector at the appropriate position, as returned by `find-last`. However, if we try this out, we soon realize that it does not work! We still have the same broken semantics as before.

Why is it? Let us refresh memories and see how we desugar a field read `?`:

```
(defmac (? fd) #:captures self
  ((obj-class self) 'read self 'fd))
```

We see that we generate an expression that asks `self` for its class, and then send the `read` message to that class. Hmmm, but `self` is dynamically bound to the receiver, so we will always ask the same class to read the field! This is wrong. We should not send the `read` message to the class of the receiver, but rather to the *host class* of the method! How do we do that? We need a way to refer, from a method body, to its host class or, even better, to directly access the list of fields of the host class.

We could put the list of fields in the lexical environment of the method, as we do for `self`, but that would mean that programmers could accidentally interfere with the binding (in contrast, `self` is a typical keyword in object-oriented languages). The list of fields (and the name used to bind it) should rather remain local to our implementation. Since we locally define `?` and `!` in a class, we can simply declare the list of fields, `fields`, in the scope of these syntax definitions; the hygiene of macro expansion ensures that user code cannot accidentally interfere with `fields`.

```
....
(let* ([scls superclass]
       [fields (append (scls 'all-fields)
                       (list (cons 'fd val) ...))]
       [methods
        (local [(defmac (? fd) #:captures self
                  (vector-ref (obj-values self)
                              (find-last 'fd fields)))
                (defmac (! fd v) #:captures self
```

```
                    (vector-set! (obj-values self)
                                 (find-last 'fd fields)
                                 v))]
            ....)]))
```

This implementation is not so satisfactory because we call `find-last` (expensive/linear) for each and every field access. Can this be avoided? How?

Note that we are now directly accessing the `fields` list, so we do not need to send field access messages to the class anymore. And similarly for writing to a field.

Let us see if all this works as expected:

```
(define A
 (CLASS extends Root
        ([field x 1]
         [field y 0])
        ([method ax () (? x)])))
(define B
  (CLASS extends A
         ([field x 2])
         ([method bx () (? x)])))

> (define b (new B))
> (-> b ax)
1
> (-> b bx)
2
```

6.4 Cleaning up the Class Protocol

Since we first introduced §5 "Classes", we have made several changes to the protocol of classes:

- We have split the `invoke` protocol into two, by introducing `lookup` whose purpose is solely to lookup the appropriate method definition in the class hierarchy.
- We have added `all-fields` in order to be able to retrieve the fields of a class. This is used at class construction time to obtain the fields of the superclass and append them to the ones being defined.
- We got rid of the `read`/`write` protocol for field accesses, in order to properly scope field names in methods.

It is a good time to reflect upon the class protocol and see whether or not what we have here is a minimal protocol, or if we could get rid of some of it. What is a good criteria for deciding? well, since we are talking about the protocol of a class, it may better be that it actually relies on the class processing the message. For instance, since we first introduce the `read`/`write` protocol, we could have gotten rid of it. Remember:

```
....
[(read)  (dict-ref (obj-values (first vals)) (second vals))]
[(write) (dict-set! (obj-values (first vals)) (second vals)
                    (third vals))]
....
```

Is there anything here that depends upon free variables of the class function? (in other words, that depends on the state of the class object) No, the only input needed is the current object, the name of the field being accessed, and possibly the value being written to it. We could therefore have placed this code directly in the expansion of `?` and `!`, thereby effectively "compiling away" an unnecessary layer of interpretation.

What about `invoke`? Well, it's just sending out a message to itself, which we could do directly when expanding `->`, and then the application per se is independent of the class:

```
(defmac (-> o m arg ...)
  (let ([obj o])
    ((((obj-class obj) 'lookup 'm) obj) arg ...)))
```

What about the other parts of the class protocol? `all-fields`, `create`, and `lookup` do access the internal state of the class: for `all-fields` we access `fields`; for `create` we access both `fields` and `class` itself; and for `lookup` we access both `methods` and `superclass`. So, our classes only need to understand these three messages.

6.5 Super Sends

When a method overrides a method in a superclass, it is sometimes useful to be able to invoke that definition. This allows many typical refinement patterns, such as adding something to do before or after a method is executed, like additional processing on its arguments and return values, among many others. This is achieved by doing what is called a *super send*. We use `-->` as the syntax for a super send.

Let us look at a first example:

```
(define Point
 (CLASS extends Root
         ([field x 0])
         ([method x? () (? x)]
          [method x! (new-x) (! x new-x)]
          [method as-string ()
                  (string-append "Point("
```

```
                                    (number->string (? x)) ")")])))

(define ColorPoint
 (CLASS extends Point
         ([field color 'black])
         ([method color? () (? color)]
          [method color! (clr) (! color clr)]
          [method as-string ()
                  (string-append (--> as-string) "-"
                                 (symbol->string (? color)))])))

> (define cp (new ColorPoint))
> (-> cp as-string)
"Point(0)-black"
```

Note how a super send allows us to reuse and extend the definition of `as-string` in `Point` in order to define the method in `ColorPoint`. In Java, this is done by invoking a method on `super`, but what exactly is `super`? what is the semantics of a super send?

A first thing to clarify is: what is the receiver of a super send? In the example above, to which object is `as-string` sent when using `-->`? `self`! In fact, `super` only really affects method lookup. A common misunderstanding is that when doing a super send, method lookup starts with the *superclass* of the receiver, instead of its class. Let us see why this is incorrect in a small, artificial example:

```
(define A
  (CLASS extends Root ()
         ([method m () "A"])))

(define B
  (CLASS extends A ()
         ([method m () (string-append "B" (--> m) "B")])))

(define C
  (CLASS extends B () ()))

(define c (new C))
(-> c m)
```

What does this program return? Let's see. `->` expands into a send of `lookup` to `c`'s class, which is `C`. There is no methods defined for `m` in `C`, so it sends `lookup` to its superclass, `B`. `B` finds a method for `m`, and returns it. It is then applied to the current self (`c`) and then passed arguments, none in this case. The evaluation of the method implies the evaluation of the three arguments of `string-append`, the second of which is a super send. With the above definition of a super send, this means that `m` is looked up not in `C` (the actual class of the receiver) but in B (its superclass). Is there a method for m in B? Yes, and we are actually executing it.... In other words, with this understanding of super the above program does NOT terminate.

Some dynamic languages, like Ruby, allow the class inheritance relation to be changed at runtime. This is common in prototype-based languages, like Self and JavaScript.

What is the mistake? To consider that a self send implies looking up the method in the superclass of the receiver. In the example above, we should actually lookup m in A, not in B. To this end, we need to know the *superclass of the host class* of the method in which the super send is performed. Is this a value that should be bound statically in method bodies, or dynamically? Well, we've just said it: it is the superclass of the host class of the method, and that is not likely to change dynamically (at least in our language). Luckily, we already have a binding in the lexical context of methods that refers to the superclass, scls. So we just need to introduce a new local macro for -->, whose expansion asks the superclass scls to lookup the message. --> can be used by user code, so it is added to the list of #:captures identifiers:

```
(defmac (CLASS extends superclass
               ([field f init] ...)
               ([method m params body] ...))
  #:keywords field method extends
  #:captures self ? ! -->
  (let* ([scls superclass]
         [fields (append (scls 'all-fields)
                         (list (cons 'f init) ...))]
         [methods
          (local [(defmac (? fd) ....)
                  (defmac (! fd v) ....)
                  (defmac (--> md . args) #:captures self
                    (((scls 'lookup 'md) self) . args))]
            ....)]))
```

Note how lookup is now sent to the superclass of the host class of the currently-executing method, scls, instead of to the actual class of the current object.

```
> (define c (new C))
> (-> c m)
"BAB"
```

6.6 Inheritance and Initialization

We have previously seen how to address object initialization (§5.5 "Initialization"), by introducing special methods called initializers. Once an object is created, and before it is returned to the creator, its initializer is called.

In the presence of inheritance, the process is a bit more subtle, because if initializers override each other, some necessary initialization work can be missed. The work of an initializer can be quite specific, and we want to avoid subclasses to have to deal with all the details. One could simply assume the normal semantics of method dispatch, whereby `initialize` in a subclass can call the super initializer if needed. The problem with this freedom is that the initializer in the subclass may start to do things with the object when inherited fields are not yet consistently initialized. To avoid this issue, in Java, the first thing a constructor must do is to call a super constructor (it may have to compute arguments for that call, but that's all it is allowed to do). Even if the call is not in the source code, the compiler adds it. Actually, this restriction is also enforced at the VM level by the bytecode verifier: low-level bytecode surgery can therefore not be used to avoid the super constructor call.

7 A World of Possibilities

In this brief step-by-step construction of object systems in Scheme, we have only illustrated some fundamental concepts of object-oriented programming languages. As always in language design, there is a world of possibilities to explore, variations on the same ideas, or extensions of these ideas.

Here is just a (limited/arbitrary) list of features and mechanisms that you will find in some existing object-oriented programming languages, which were not covered in our tour, and that you may want to try to integrate in the object system. It is—of course—very interesting to think of other features by yourself, as well as to study existing languages and figure out how to integrate their distinctive features.

- Visibility for methods: public/private
- Declare methods that override a method in a superclass: override
- Declare methods that cannot be overriden: final
- Declare methods that are expected to be inherited: inherit
- Augmentable methods: inner
- Interfaces: sets of messages to understand
- Protocol to check if an object is instance of a class, if a class implements an interface, ...
- Proper initialization protocol with superclasses, named initialization attributes
- Multiple inheritance
- Mixins
- Traits
- Classes as objects, metaclasses, ...

There are also many optimizations, such as:

- compute fields offset for direct field access
- vtables & indices for direct method invocation

We now only briefly outline two mechanisms, interfaces and mixins, as well as their combination (ie. using interfaces in the specification of mixins).

7.1 Interfaces

Introduce a form to define interfaces (which can extend super interfaces):

```
(interface (superinterface-expr ...) id ...)
```

Introduce a new class form that expects a list of interfaces:

```
(CLASS* super-expr (interface-expr ...) decls ...)
```

Example:

```
(define positionable-interface
  (interface () get-pos set-pos move-by))

(define Figure
  (CLASS* Root (positionable-interface)
     ....))
```

Extend the protocol of a class so as to be able to check if it implements a given interface:

```
> (implements? Figure positionable-interface)
#t
```

7.2 Mixins

A mixin is a class declaration parameterized over its superclass. Mixins can be combined to create new classes whose implementation sharing does not fit into a single-inheritance hierarchy.

Mixins "come for free" by the mere fact of having classes be first-class values integrated with functions.

```
(define (foo-mixin cl)
  (CLASS cl (....) (....)))

(define (bar-mixin cl)
  (CLASS cl (....) (....)))

(define Point (CLASS () ....))

(define foobarPoint
  (foo-mixin (bar-mixin Point)))
(define fbp (foobarPoint 'create))
....
```

Combine with interfaces to check that the given base class implements a certain set of interfaces. Define a `MIXIN` form for that:

```
(MIXIN (interface-expr ...) decl ...)
```

Defines a function that takes a base class, checks that it implements all the specified interfaces, and return a new class that extends the base class with the given declarations.