

# Prometheus

---

A prototype-based object system for Scheme

Jorgen Schäfer

---

Copyright © Jorgen Schäfer

This program is free software; you can redistribute it and/or modify it under the terms of the GNU General Public License as published by the Free Software Foundation; either version 2 of the License, or (at your option) any later version.

This program is distributed in the hope that it will be useful, but WITHOUT ANY WARRANTY; without even the implied warranty of MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE. See the GNU General Public License for more details.

You should have received a copy of the GNU General Public License along with this program; if not, write to the Free Software Foundation, Inc., 59 Temple Place - Suite 330, Boston, MA 02111-1307, USA.

# Table of Contents

<b>1</b>	<b>Introduction</b> .....	<b>1</b>
<b>2</b>	<b>Installation</b> .....	<b>2</b>
<b>3</b>	<b>Prometheus</b> .....	<b>3</b>
3.1	Objects .....	3
3.2	Slots .....	3
3.3	Inheritance .....	3
3.4	Root Objects .....	4
3.5	Syntactic Sugar .....	5
3.6	Private Messages .....	6
<b>4</b>	<b>Examples</b> .....	<b>7</b>
4.1	Simple Account Object .....	7
4.2	Creating Slots on Use .....	8
4.3	Diamond Inheritance .....	8
<b>5</b>	<b>Pitfalls</b> .....	<b>11</b>
5.1	Setters are Methods .....	11
<b>Appendix A</b>	<b>Hermes</b> .....	<b>12</b>

# 1 Introduction

Prometheus is a prototype-based message-passing object system for Scheme similar to the [Self language](#)

In a prototype-based object system, an object is just a set of slots. A slot has a name and a value, or a *handler procedure* which reacts on messages associated with the slot. Some slots are special, so-called *parent slots*, whose use will become apparent shortly.

Objects receive messages. A message consists of a symbol called a *selector*, and zero or more arguments. When an object receives a message, the object searches for a slot whose name is equal (eq?, actually) to the message selector. When it finds such a slot, it invokes the slot's handler, or returns the slot's value, as appropriate. When the slot is not in the object, all objects in parent slots are queried for that slot.

An object is created by *cloning* an existing object. The new object is empty except for a single parent slot, which points to the cloned object. This way, the new object behaves exactly like the old one.

In a prototype-based object system, objects are created and modified until they behave as required. Then, that object is cloned to create the real objects to work with—it forms the *prototype* for the other objects.

This manual documents version 2 of Prometheus. The canonical URL is <http://www.forcix.cx/software/prometheus.html>.

## 2 Installation

Prometheus is shipped as a package for [Scheme 48](#). The structure `prometheus` serves as the main user API. To use it, issue the following commands at the REPL:

```
> ,config ,load ../prometheus/scheme/packages.scm
> ,open prometheus
```

A simple test would be the following:

```
> (define o (*the-root-object* 'clone))
> (o 'add-value-slot! 'fnord 'set-fnord! 23)
> (o 'fnord)
23
```

The package works unmodified with [the Scheme Shell](#). Prometheus is written in R5RS Scheme with a few SRFI dependencies documented in `'packages.scm'`, and should be easy to port to other implementations of Scheme.

## 3 Prometheus

### 3.1 Objects

Prometheus objects are implemented as closures. To send a message to that object, the closure is applied to the message selector (i.e., the slot name), followed by a number of arguments. The return value(s) of the message are returned from this application.

### 3.2 Slots

Prometheus knows about three kinds of slots.

*Value slots* merely store a value which is returned when the corresponding message is received.

*Parent slots* are just like value slots, but have a special flag marking them as parents.

*Method slots* contain a procedure which is invoked for messages corresponding to this slot.

The procedure is called with at least two arguments, conventionally called *self* and *resend*. If the message received any arguments, they are also passed, after *resend*. *Self* is the object which received the messages, as opposed to the object where this message handler was found in. *Resend* is a procedure which can be used to resend the message to further parents, if the current method does not wish to handle the message. See [Section 3.3 \[Inheritance\]](#), [page 3](#), for more information about this.

A typical method handler could thus look like:

```
(lambda (self resend a b)
  (/ (+ a b)
     2))
```

Every slot, regardless of its kind, can be associated with a *setter method* when it is created. Setter methods receive a single argument, and replaces the value of the corresponding slot with this argument. Setter methods can be created automatically when a given slot is created, and are removed when the corresponding getter slot is removed (but not vice-versa). Because of this, they are sometimes not considered to be slots, even if they are. See [Section 5.1 \[Setters are Methods\]](#), [page 11](#), for an example where this distinction is important.

### 3.3 Inheritance

When a slot for a message is not found in the current object, all its parent slots are queried recursively, i.e. parent objects which don't know the slot query *their* parents, etc.

If no parent knows the slot, the original message receiving object is sent a `message-not-understood` message. If more than one parent knows the slot, the original message receiving object is sent a `ambiguous-message-send` message. See [Section 3.4 \[Root Objects\]](#), [page 4](#), for a documentation of those messages. By default, they signal an error.

Method slots can decide not to handle the message, but rather search the inheritance tree for other handlers. For this purpose, they are passed a procedure commonly called *resend*. See [Section 3.2 \[Slots\]](#), [page 3](#), for an explanation of method slots.

It is important to understand the difference between sending a message to an object, and *resending* it to the object. When a message is just sent to an object, methods will get that object as the *self* argument. When the method wants information about the object it is handling messages for, this is usually not what is intended.

Consider an account object, which inherited from the account prototype. All the methods are in the account prototype. When a message to modify the account value is sent to the actual account object, the message receiver is the account object. It does not handle this message, so it resends the message to the parent object, the account prototype. The method handler to

modify the account value should now know to modify the account object, not the prototype. Hence, the *self* argument should point to the account object, but if the message was just sent directly to the prototype, it *self* would be the prototype. Hence, resending exists. The *resend* procedure allows a method to manually request such a resending.

**resend** *whereto message args* ... [Procedure]

This procedure will try to find a different handler for the given *message*. The handler can be searched for further up the inheritance tree, or even in a totally different object and its parents.

*Whereto* can be one of the following values.

**#f** Use any parent of the object where this handler was found in.

A symbol Use the object in the parent slot with this name.

Any object  
Search for handlers in that object.

*Resend* is roughly equivalent in concept to CLOS' (*next-method*).

### 3.4 Root Objects

Since objects are created by sending a *clone* message to other objects, there has to be a kind of root object. Prometheus provides a procedure to create such root objects.

**make-prometheus-root-object** [Procedure]

This creates a new root object from which other objects can be cloned. This object is independent of any other object, and thus creates a new inheritance tree.

Prometheus also provides a single existing root object, created with the procedure above. Unless specifically wanted otherwise, using this object as the root object ensures that all prometheus objects share a common ancestor.

**\*the-root-object\*** [Variable]

This is the default root object. If not really intended otherwise, this should be used as the root of the object hierarchy in a program.

Root objects contain a number of slots by default.

**clone** [Message]

Return a clone of the message recipient. This creates a new object with a single slot, **parent**, which points to the cloned object

**add-value-slot!** *getter value* [Message]

**add-value-slot!** *getter setter value* [Message]

Add a new value slot to the recipient. The value of the slot can be retrieved with the *getter* message. If a *setter* message is given, that message can be used to change the value of the slot.

**add-method-slot!** *getter proc* [Message]

**add-method-slot!** *getter setter proc* [Message]

Add a method to the recipient. Sending the object a *getter* message now invokes *proc* with the same arguments as the message, in addition to a *self* argument pointing to the current object and a *resend* procedure available to resend the message if the method does not want to handle it directly.

The *setter* message can later be used to change the procedure.

`add-parent-slot! getter parent` [Message]

`add-parent-slot! getter setter parent` [Message]

Add a parent slot to the recipient. Parent slots are searched for slots not found directly in the object. The *setter* message, if given, can be used to later change the value of the parent slot.

`delete-slot! name` [Message]

Delete the slot named *name* from the receiving object. This also removes the setter corresponding to *name*, if any. Beware that the parents might contain the same slot, so a message send can still succeed even after a slot is deleted.

`immediate-slot-list` [Message]

This message returns a list of slots in this object. The elements of the list are lists with four elements:

- *getter-name*
- *setter-name*
- `#f`
- *type*, which can be one of the symbols `value`, `method` or `parent`.

`message-not-understood message args` [Message]

This is received when the message *message* with arguments *args* to the object was not understood. The root object just signals an error.

`ambiguous-message-send message args` [Message]

This is received when the message *message* with arguments *args* to the object would have reached multiple parents. The root object just signals an error.

### 3.5 Syntactic Sugar

Prometheus provides two forms of syntactic sugar for common operations on objects.

A very common operation is to add method slots to an object, which usually looks like this:

```
(obj 'add-method-slot!
  'average
  (lambda (self resend a b)
    (/ (+ a b)
      2)))
```

Using the special form of `define-method`, this can be shortened to:

```
(define-method (obj 'average self resend a b)
  (/ (+ a b)
    2))
```

`define-method (obj 'message self resend . args) body ...` [Syntax]

This is syntactic sugar for the often-used idiom to define a method slot, by sending a `add-method-slot!` message with a *message* name and a lambda form with *self*, *resend* and *args* formals, and a *body*.

Another common operation is to clone an object, and add a number of value and method slots:

```
(define o (*the-root-object* 'clone))
(o 'add-value-slot! 'constant 'set-constant! 5)
(o 'add-method-slot! 'add
  (lambda (self resend summand)
    (+ summand (self 'constant))))
```

This can be more succinctly written as:

```
(define-object o (*the-root-object*)
  (constant set-constant! 5)
  ((add self resend summand)
   (+ summand (self 'constant))))
```

`define-object` *name* (*parent other-parents ...*) *slots ...* [Syntax]

This is syntactic sugar for the typical actions of cloning an object from a *parent* object, and adding more slots.

*other-parents* is a list of (*name object*) lists, where each *object* is added as a parent slot named *name*.

*slots* is a list of slot specifications, either (*getter value*) or (*getter setter value*) for value slots, or ((*name self resend args ...*) *body ...*) for method slots.

### 3.6 Private Messages

Message names in Prometheus don't have any required type. They are only compared using `eq?`. Because of this, any kind of Scheme object can be used as a message name. This means that it is possible to use a private Scheme value—for example, a freshly-allocated list—as a slot name. This can be used to keep slot names private, since it is not possible to create an object which is `eq?` to such an object except by receiving a reference to that object.

## 4 Examples

### 4.1 Simple Account Object

This is from the file ‘examples/account.scm’ in the Prometheus distribution:

```

;;; This is a simple account-keeping object.

;;; It's just like a normal object
(define account (*the-root-object* 'clone))

;;; But it has a balance
(account 'add-value-slot! 'balance 'set-balance! 0)

;;; Which can be modified
(account 'add-method-slot! 'payment!
        (lambda (self resend amount)
          (self 'set-balance!
                (+ (self 'balance)
                   amount))))

;;; Some tests:
(define a1 (account 'clone))
(define a2 (account 'clone))

(a1 'payment! 100)
(a2 'payment! 200)

(a1 'balance)
;;; => 100
(a2 'balance)
;;; => 200

(a1 'payment! -20)
(a1 'balance)
;;; => 80

;;; The typing for the slot definitions above can be rather tedious.
;;; Prometheus provides syntactic sugar for those operations.

;;; A method can be added with the DEFINE-METHOD syntax. This code is
;;; equivalent to the code above which adds the PAYMENT! method:
(define-method (account 'payment! self resend amount)
  (self 'set-balance!
        (+ (self 'balance)
           amount)))

;;; And this defines the whole object with the BALANCE slot and the
;;; PAYMENT! method just as above:
(define-object account (*the-root-object*)
  (balance set-balance! 0)
  ((payment! self resend amount)

```

```
(self 'set-balance!
      (+ (self 'balance)
         amount))))
```

## 4.2 Creating Slots on Use

This is from the file 'examples/create-on-use.scm' in the Prometheus distribution:

```
;;; A simple object which creates slots as they are used. This
;;; demonstrates the use of the MESSAGE-NOT-UNDERSTOOD error message.

;;; Slots behave like value slots, and the accessors use a second
;;; argument as the "default value". If that is not given, (if #f #f)
;;; is used, which is usually not what is intended.
(define-object create-on-use-object (*the-root-object*)
  ((message-not-understood self resend slot args)
   (self 'add-method-slot! slot (lambda (self resend . default)
                                  (if (pair? args)
                                      (car args))))
   (self slot)))
```

## 4.3 Diamond Inheritance

This is from the file 'examples/diamond.scm' in the Prometheus distribution:

```
;;; This requires SRFI-23

;;; We create an amphibious vehicle which inherits from a car - which
;;; can only drive on ground - and from a ship - which can only drive
;;; on water. Roads have a type of terrain. The amphibious vehicle
;;; drives along the road, using either the drive method of the car or
;;; of the ship.

;;; First, let's build a road.
(define-object road-segment (*the-root-object*)
  (next set-next! #f)
  (type set-type! 'ground)
  ((clone self resend next type)
   (let ((o (resend #f 'clone)))
     (o 'set-next! next)
     (o 'set-type! type)
     o)))

;;; Create a road with the environment types in the ENVIRONMENTS list.
(define (make-road environments)
  (if (null? (cdr environments))
      (road-segment 'clone
                    #f
                    (car environments))
      (road-segment 'clone
                    (make-road (cdr environments))
                    (car environments))))

;;; Now, we need a vehicle - the base class.
```



```
;;; The code above works already. We can clone ships, automobiles and
;;; amphibious vehicles as much as we want, and they drive happily on
;;; roads. But we could extend this, and add gas consumption. This
;;; will even modify already existing vehicles, because they inherit
;;; from the vehicle object we extend:
(vehicle 'add-value-slot! 'gas 'set-gas! 0)
(vehicle 'add-value-slot! 'needed-gas 'set-needed-gas! 0)
(define-method (vehicle 'drive self resend)
  (let ((current-gas (self 'gas))
        (needed-gas (self 'needed-gas)))
    (if (>= current-gas needed-gas)
        (self 'set-gas! (- current-gas needed-gas))
        (error "Out of gas!"))))

;;; If you want to test the speed of the implementation:
(define (make-infinite-road)
  (let* ((ground (road-segment 'clone #f 'ground))
        (water (road-segment 'clone ground 'water)))
    (ground 'set-next! water)
    ground))

(define (test n)
  (let ((o (amphibious 'clone (make-infinite-road))))
    (do ((i 0 (+ i 1)))
        ((= i n) #t)
      (o 'drive))))
```

## 5 Pitfalls

### 5.1 Setters are Methods

Since Prometheus does not allow for ambiguous message sends, and setter methods are just messages, this can lead to a confusing situation. Consider the following code:

```
(define o1 (*the-root-object* 'clone))
(o1 'add-value-slot! 'foo 'set-foo! 1)
(define o2 (o1 'clone))
(define o3 (o2 'clone))
(o3 'add-parent-slot! 'parent2 o1)
```

This creates a diamond-shaped inheritance tree. Now it is possible to send a `set-foo!` message to `o3`, though it inherits this slot from two parents, the slot is ultimately inherited from the same object. But now witness the following:

```
> (o3 'foo)
⇒ 3
> (o2 'set-foo! 2)
> (o3 'set-foo! 3)
[error] Ambiguous message send
```

What happened here? The `set-foo!` message added the `foo` slot to `o2`, but with it, also the associated method to mutate that slot, `set-foo!`. So, sending `set-foo!` to `o3` will find the same message both in `o1` and `o2`, and cause an ambiguous message send.

Conclusion: Be extra careful with multiple inheritance.

## Appendix A Hermes

Hermes is a very small object system based solely on messages. It's the basis upon which Prometheus is built. It should be considered internal information of Prometheus, but someone might find it useful, and it is visible in every Prometheus object.

A Hermes object contains messages. A message has a name, a procedure to handle this message, and a type flag on whether the return value of this message should be treated as a parent object. A message-handling procedure is applied to the arguments of the message in addition to two arguments, probably known from Prometheus' method slots by now: *Self* and *resend*. Indeed, Prometheus' method slots are just a very thing wrapper around Hermes messages.

The `hermes` structure exports a single procedure.

`make-hermes-object` [Procedure]

Return a new Hermes object which knows the basic messages.

The basic messages known by all Hermes objects are as follows.

`add-message! name handler [parent?]` [Message]

This adds a message named *name* to the object, upon receiving which, *handler* is applied to *self*, *resend*, and the arguments to the message. If *parent?* is supplied and not false, this message returns a parent object. In this case, it must be callable with no arguments, and must not use *resend*.

`delete-message! name` [Message]

Remove the handler for the message named *name*. This causes such messages to be handled by parent objects in the future again.

`%get-handler name receiver args visited` [Message]

The message upon which inheritance in Hermes is built. This returns a procedure of no arguments which handles the receiving of the message. This is delayed so that Hermes can check for duplicate handlers, which would be an error.

*Name* is the name of the message we are looking for. *Receiver* is the original receiver of the message, to be used as the *self* argument to the handler procedure. *Args* is a list of arguments. *Visited* is a list of objects we have seen so far. This is used to detect cycles in the inheritance graph.

This message returns two values. The first one is the handler and the other one the object in which this handler was found. The handler value can also be one of two symbols to signify an error condition. If it's the symbol `message-not-understood`, then neither this object nor any parents knew how to handle this message. If it's `ambiguous-message-send`, the same message could be handled by multiple parents in the inheritance graph. The user needs to add a message which resends the ambiguous message unambiguously to the correct parent. In either case, the second return value is `#f`. The handler procedure itself accepts no arguments, and just runs the message.

It is absolutely sufficient for an object to handle only the `%get-handler` message to participate in the inheritance handling of Hermes.