Generic Functions

Intro: Class-based OOP

```
class OutputStream {
   void println(Object obj) { ... }
   ...
}
```

... allows you to say:

```
out.println(pascal);
```

Intro: Class-based OOP

out.println(pascal);

...or, in Lisp syntax:

(send out 'println pascal)

The receiver is just another argument.

So let's change...

(send receiver message args ...)

...to...

(call message receiver args ...)

“Call” is redundant.

So let’s change...

(call message receiver args ...)

...to...

(message receiver args ...)

So now we have generic functions!

(send out ‘println pascal)

...is now...

(println out pascal)

Let's define methods.

```
(defmethod println ((out output-stream)
                    (p person))
  ...)
```

This is a method with multiple dispatch!

Generic functions.

- Invented when Lispers implemented OOP. Function calls appear more natural in Lisp. (LOOPS, New Flavors, CLOS)
- Generic functions were already needed. Mathematical operations are generic! They work on ints, floats, complex, etc.

Mathematical ops as generic functions.

```
(defmethod + ((x int) (y int))
  ...)
(defmethod + ((x float) (y float))
  ...)
(defmethod + ((x complex) (y complex))
  ...)
```

...but how does it work?

- Let's implement generic functions!

Further notes.

- Efficiency:
  Cache everything!
- Multiple dispatch:
  Consider all the args when selecting applicable and most specific methods!
- Advice:
  Add qualified methods that are called before, after or around everything else!

Further information.

- Pick a good Common Lisp tutorial.
  - Peter Seibel, Practical Common Lisp, http://www.gigamonkeys.com/book/
  - David Lamkins, Successful Lisp, http://www.psg.com/~dlamkins/sl/

Further information.

- Papers about CLOS:
  http://www.dreamsongs.com/CLOS.html

```
(in-package :closless)

;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;; A class has a name, a direct superclass and some direct slots.
;;;
;;; The default values are:
;;; - nil for the name.
;;;   In other words, classes are by default anonymous.
;;; - 'object for the direct superclass.
;;;   That class will be defined later.
;;; - The empty list for the direct slots.
;;;
;;; This defstruct automatically creates the following functions:
;;; - make-class for creating a class.
;;; - class-name for accessing the name.
;;; - class-direct-superclass for accessing the direct superclass.
;;; - class-direct-slots for accessing the direct slots.
;;;
;;;

(defstruct class
  (name nil)
  (direct-superclass 'object)
  (direct-slots '()))

;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;; The *class-table* contains mappings of class names to instances
;;; of the struct class defined above.
;;;
;;; The function find-class and its setter (setf find-class) are
;;; used to access that class table.
;;;
;;;

(defvar *class-table* (make-hash-table))

(defun find-class (class-name)
  (gethash class-name *class-table*))

(defun (setf find-class) (class class-name)
  (setf (class-name class) class-name)
  (setf (gethash class-name *class-table*) class))
```

```
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;; Now we can define the class 'object which is the root of the
;;; class hierarchy. The direct superclass is nil (no direct
;;; superclass), and the list of direct slots is empty (no
;;; direct slots).
;;;
;;; Here you can also see how setters are used:
;;; (find-class 'object) is the left hand side of the setf expression
;;; (make-class :direct-superclass nil) is the right hand side.
;;;
;;; This is similar to what would roughly be expressed like
;;; (find-class 'object) := (make-class :direct-superclass nil)
;;; in other languages.
;;;
;;;

(unless (find-class 'object)
  (setf (find-class 'object)
        (make-class :direct-superclass nil)))

;; All superclassses of a class is
;; the transitive closure of class-direct-superclass.

(defun class-all-superclasses (class-name)
  (loop for c = (find-class class-name) then (find-class (class-direct-superclass c))
        while c collect c))

;; class-name is the subclass of superclass-name
;; if superclass-name is an element of all superclasses of class-name.

(defun subclassp (class-name superclass-name)
  (member (find-class superclass-name)
          (class-all-superclasses class-name)))

;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;; An object is an instance of a class and has some slots.
;;;
;;; The default values are:
;;; - 'object as the class of the object.
;;; - An empty hash table for the slots of the object.
;;;
;;; This defstruct automatically creates the following functions:
;;; - make-object for creating an object.
;;; - object-class for accessing the class of an object.
;;; - object-slots for accessing the slots of an object.
;;;
;;;

(defstruct object
  (class 'object)
  (slots (make-hash-table)))

;; An object is an instance of a class
;; if the class of this object is a subclass of that class.

(defun instancep (object class-name)
  (subclassp (object-class object) class-name))

;; Slots are stored in hashtables.
;; slot-value returns the value for a slot in an object.
;; (setf slot-value) sets the value for a slot in an object.

(defun slot-value (object slot-name)
  (gethash slot-name (object-slots object)))

(defun (setf slot-value) (new-value object slot-name)
  (setf (gethash slot-name (object-slots object))
        new-value))
```

```lisp
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;; A generic function is a collection of methods.
;;;
;;; The default value is the empty list.
;;;
;;; This defstruct automatically creates the following functions:
;;; - make-generic-function for creating a generic function.
;;; - generic-function-methods for accessing the methods of a generic function.
;;;
;;;

(defstruct generic-function
  (methods '()))

;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;; A method is specialized on a class
;;; and has a function to be executed when the method is called.
;;;
;;; The default values are:
;;; - 'object as the specializer of the method.
;;; - An error for the function.
;;;   This means that a method function must be explicitly provided.
;;;
;;; This defstruct automatically creates the following functions:
;;; - make-method for creating a method.
;;; - method-specializer for accessing the specializer of a method.
;;; - method-function for accessing the function of a method.
;;;
;;;

(defstruct method
  (specializer 'object)
  (function (error "No method function provided.")))

;; We can find a method in a generic function that is specialized on a given class.

(defun find-method (gf specializer)
  (loop for method in (generic-function-methods gf)
        when (eql specializer (method-specializer method))
        return method))

;; We can remove a method from a generic function.

(defun remove-method (gf method)
  (setf (generic-function-methods gf)
        (remove method (generic-function-methods gf))))

;; We can add a method to a generic function.
;; If a method with the given specializer already exists
;; we remove that old method first.
;; (We don't want to have more than one method for a given specializer!)

(defun add-method (gf new-method)
  (let ((old-method (find-method gf (method-specializer new-method))))
    (when old-method
      (remove-method gf old-method)))
  (push new-method (generic-function-methods gf)))
```

```
;; The applicable methods for a receiver in a generic function
;; are those where the receiver is an instance of the method specializer.

(defun compute-applicable-methods (gf receiver)
  (loop for method in (generic-function-methods gf)
        when (instancep receiver (method-specializer method))
        collect method))

;; The most specific method of a set of methods
;; is the one whose specializer is a subclass of
;; all the other specializers.

(defun select-most-specific-method (methods)
  (loop with candidate = (first methods)
        for method in (rest methods)
        when (subclassp (method-specializer method) (method-specializer candidate))
        do (setf candidate method)
        finally (return candidate)))

;; The following steps are performed when a generic function is called:
;; 1. The applicable methods for the receiver object are selected.
;; 2. The most specific method in the set of applicable methods is selected.
;; 3. The function for the most specific method is called.
;;    It gets passed
;;    - the receiver,
;;    - the other arguments,
;;    - the set of applicable methods without the most specific method.
;;    The latter is needed so that the most specific method can perform supercalls.

(defun call-generic-function (gf receiver &rest args)
  (let* ((applicable-methods (compute-applicable-methods gf receiver))
         (most-specific-method (select-most-specific-method applicable-methods)))
    (funcall (method-function most-specific-method)
             receiver args
             (remove most-specific-method applicable-methods))))

;; The following steps are performed when the next method is called,
;; or in other words, when a super call is performed:
;; 1. The most specific method in the set of next methods is selected.
;; 2. The function for the next most specific method is called.
;;    It gets passed
;;    - the receiver,
;;    - the other arguments,
;;    - the set of next methods without the next most specific method.

(defun call-next-method (receiver args next-methods)
  (let ((next-specific-method
         (select-most-specific-method next-methods)))
    (funcall (method-function next-specific-method)
             receiver args (remove next-specific-method next-methods))))
```

```
(in-package :closless)

;; 'person is a subclass of 'object
;; with the slots 'name and 'address.

(setf (find-class 'person)
      (make-class :direct-superclass 'object
                  :direct-slots '(name address)))

;; 'employee is a subclass of 'person
;; with the additional slot 'employer.

(setf (find-class 'employee)
      (make-class :direct-superclass 'person
                  :direct-slots '(employer)))

;; Pascal is a 'person
;; with the name "Pascal" and the address "Brussels".

(defparameter *pascal*
  (make-object :class 'person))
(setf (slot-value *pascal* 'name) "Pascal")
(setf (slot-value *pascal* 'address) "Brussels")

;; Display is a generic function.

(defparameter <display>
  (make-generic-function))

;; The display method for 'person
;; prints the name and the address of a 'person.

(add-method
 <display>
 (make-method
  :specializer 'person
  :function (lambda (receiver args next-methods)
              (print (slot-value receiver 'name))
              (print (slot-value receiver 'address)))))

;; Let's call display on Pascal.

(call-generic-function <display> *pascal*)

;; We change the class of Pascal to 'employee
;; and set his employer to "Vrije Universiteit Brussel".

(setf (object-class *pascal*) 'employee)
(setf (slot-value *pascal* 'employer) "Vrije Universiteit Brussel")

;; The display method for 'employee
;; performs a super call and then
;; prints the employer of an 'employee.

(add-method
 <display>
 (make-method
  :specializer 'employee
  :function (lambda (receiver args next-methods)
              (call-next-method receiver args next-methods)
              (print (slot-value receiver 'employer)))))

;; Let's call display on Pascal again.

(call-generic-function <display> *pascal*)
```

CLOS

Classes

```
(defclass person (standard-object)
   ((name :accessor person-name
          :initarg :name
          :allocation :instance)
    (address :accessor person-address
             :initarg :address
             :allocation :instance))
   (:documentation
    "This is a class for person objects."))
```

Class and Superclasses

```
(defclass person (standard-object)
   ((name :accessor person-name
          :initarg :name
          :allocation :instance)
    (address :accessor person-address
             :initarg :address
             :allocation :instance))
   (:documentation
    "This is a class for person objects."))
```

Slots and Slot Options

```
(defclass person (standard-object)
   ((name :accessor person-name
          :initarg :name
          :allocation :instance)
    (address :accessor person-address
             :initarg :address
             :allocation :instance))
   (:documentation
    "This is a class for person objects."))
```

Class Options

```
(defclass person (standard-object)
    ((name :accessor person-name
           :initarg :name
           :allocation :instance)
     (address :accessor person-address
              :initarg :address
              :allocation :instance))
    (:documentation
     "This is a class for person objects."))
```

CLOS: Instances and Accessors

- (defparameter pascal
    (make-instance 'person
                   :name "Pascal"
                   :address "Brussels"))
- (person-name pascal)
  => "Pascal"

Generic Functions and Methods

- (defgeneric display (object))
- (defmethod display ((object person))
  (print (person-name object))
  (print (person-address object)))
- (display pascal)

Inheritance

- (defclass employee (person)
  ((employer :accessor person-employer)))
- (defmethod display ((object employee))
  (call-next-method)
  (print (person-employer object)))