

# Functional object-oriented programming with Common Lisp Object System (CLOS)

Dr. C. Constantinides

Department of Computer Science and Software Engineering  
Concordia University Montreal, Canada

August 14, 2013

## **Classes, objects and message passing III (ch. 30)**

# Definitions

- ▶ A *type* is a collection of values. Examples of types include the integer or the Boolean type.
- ▶ We can distinguish between *simple types* and *composite types* (or *aggregate types*) based on whether or not the values of a type can contain subparts.
- ▶ As an example, we can say that the integer type is simple, whereas a student record is composite.
- ▶ A *data item* is an instance of a type (also: a *member* of a type).

## Definitions /cont.

- ▶ A *data type* is the definition of a type together with a collection of operations that manipulate that type, e.g. the set of integers together with operations such as addition, subtraction, and multiplication.
- ▶ An *abstract data type (ADT)* is a definition for a data type solely in terms of the set of values and a set of operations on that data type.
- ▶ The behavior of each operation is determined by its inputs and outputs. This implies that an ADT is implementation-independent.
- ▶ Clients of the ADT are unaffected by any changes to the implementation as long as they conform to the interface of the ADT.

## Definitions /cont.

- ▶ A *data structure* is a specific implementation of an ADT. The implementation details are hidden from the clients of the ADT. This is referred to as *information hiding*.
- ▶ The choice of a data structure for the implementation of a particular ADT involves benefits and costs.
- ▶ Because of these trade-offs, rarely (if at all) one data structure is better than another in all situations.
- ▶ In identifying the trade-offs for a data structure to implement a particular ADT, we need to consider the following requirements:
  - ▶ The space for each data item it stores.
  - ▶ The time to perform each basic operation.
  - ▶ The programming effort involved.

## Example: Class semaphore

- ▶ We will model abstractions with CLOS (Common LISP Object System), an object-oriented extension to the LISP language.
- ▶ Consider the CLOS definition of class semaphore.

```
(defclass semaphore ()  
  ((count :accessor semaphore-count  
          :initform 0)  
   (name  :reader semaphore-name  
          :initarg :name)))
```

- ▶ All instances of a class have the same structure. This structure is in the form of *slots*.
- ▶ A slot has a name and a value. A value describes the slot's *state* at a given time. Each instance maintains its own state.
- ▶ This state information can be read and written by accessor methods.
- ▶ CLOS offers two kinds of slots: *local slots* and *shared slots*.

## Example: Class semaphore /cont.

- ▶ We can create an instance of semaphore by

```
> (setf s (make-instance 'semaphore))  
#<SEMAPHORE 200D0E93>
```

- ▶ The `:initform` slot option makes it possible to specify a default value for a slot.
- ▶ The `:initarg :name` slot option makes it possible to initialize the value of this slot when creating instances. We can, therefore, specify an alternative instantiation for class `semaphore` by providing an argument for the value of slot `name` as

```
> (setf s (make-instance 'semaphore :name 'my-resource))  
#<SEMAPHORE 200FEAEF>
```

## Example: Class semaphore /cont.

- ▶ We can encapsulate the call to `make-instance` in a constructor function to instantiate the class `semaphore` as follows:

```
(defun make-semaphore(name)
  (make-instance 'semaphore :name name))
```

- ▶ Now we can instantiate the class as

```
> (setf s (make-semaphore 'my-resource))
#<SEMAPHORE 20093193>
```



## Example: Class semaphore /cont.

- ▶ The `:accessor slot` option generates two methods: one for a reader and one for a writer. The term *accessor generic function* is an umbrella term that includes both readers and writers. We can set a new value for the slot count as

```
> (setf (semaphore-count s) 1)  
1
```

- ▶ We can read the value of count as

```
> (semaphore-count s)  
1
```

## Example: Class semaphore /cont.

- ▶ The `:reader` slot option generates a method for a *reader generic function* only.

```
> (semaphore-name s)  
MY-RESOURCE
```

## Example: Class semaphore /cont.

- ▶ We can provide methods to increment and decrement the value of slot count as follows:

```
(defmethod increment ((sem semaphore))  
  (setf (semaphore-count sem) (+ 1 (semaphore-count sem))))
```

```
(defmethod decrement ((sem semaphore))  
  (setf (semaphore-count sem) (- (semaphore-count sem) 1)))
```

```
> (increment s)
```

```
1
```

```
> (increment s)
```

```
2
```

```
> (decrement s)
```

```
1
```

- ▶ We observe that methods are not encapsulated inside classes, but they are instead defined separately from classes.

## Example: Class semaphore - Putting everything together

```
(defclass semaphore ()  
  ((count :accessor semaphore-count  
          :initform 0)  
   (name  :reader semaphore-name  
          :initarg :name)))  
  
(defun make-semaphore(name)  
  (make-instance 'semaphore :name name))  
  
(defmethod increment ((sem semaphore))  
  (setf (semaphore-count sem) (+ 1 (semaphore-count sem))))  
  
(defmethod decrement ((sem semaphore))  
  (setf (semaphore-count sem) (- (semaphore-count sem) 1)))
```

## Example: Class semaphore - Standard method combination

- ▶ Regular methods (or *primary methods*) can be augmented by *auxiliary methods* of three kinds:
- ▶ `:before` methods allow us to say “When a primary method is called, before running the code that should run, execute the code of this auxiliary method.”
- ▶ `:after` methods allow us to say “When a primary method is called, after running the code that should run, execute the code of this auxiliary method.”
- ▶ `:around` methods are called instead of the primary methods. They allow us to say “When a primary method is called, instead of running the code that should run, execute the code of this auxiliary method.” An around-method may also choose to invoke its primary method via `call-next-method`.

## Example: Subclassifying semaphore to define binary-semaphore

```
(defclass binary-semaphore (semaphore)() )

(defun make-binary-semaphore(name)
  (make-instance 'binary-semaphore :name name))

(defmethod increment :around ((binsem binary-semaphore))
  (if (= (semaphore-count binsem) 1)
      nil
      (call-next-method)))

(defmethod decrement :around ((binsem binary-semaphore))
  (if (= (semaphore-count binsem) 0)
      nil
      (call-next-method)))
```

## Example: /cont. - Using a binary-semaphore

```
> (setf bsem (make-binary-semaphore 'my-binary-resource))
#<BINARY-SEMAPHORE 200B8847>
> (increment bsem)
1
> (increment bsem)
NIL
> (semaphore-count bsem)
1
> (decrement bsem)
0
> (decrement bsem)
NIL
> (semaphore-count bsem)
0
```

## **Data structures and abstract data types II (ch. 31)**



# The Stack ADT: Protocol and primary functionality

- ▶ The *stack* ADT is a collection that stores arbitrary objects.
- ▶ Insertions and deletions follow a *last-in first-out* (LIFO) scheme.
- ▶ There are two major stack operations:
  - ▶ `push(stack element)`: Inserts element onto stack.
  - ▶ `pop()`: Removes and returns the last inserted element.

## The Stack ADT: Secondary functionality

- ▶ `top()`: Returns the last inserted element without removing it from the collection.
- ▶ `size()`: Returns the number of elements stored.
- ▶ `isempty()`: Returns a Boolean value indicating whether no elements are stored.
- ▶ `isfull()`: Returns a Boolean value indicating whether the collection has reached its capacity.

## The Stack ADT: Defining the structure

```
(defclass stack ()  
  ((elements :accessor stack-elements  
             :initarg :elements  
             :initform '()))  
  (size      :accessor stack-size  
             :initarg :size  
             :initform 0)  
  (capacity  :accessor stack-capacity  
             :initform 3  
             :allocation :class)))
```

## The Stack ADT: Defining primary operations - push

- ▶ In implementing a stack, we need to keep in mind that both major operations access the stack from the same end.
- ▶ The push operation would simply create a new list with the element to be placed on the stack and the current list and set it as the new value of the current list.
- ▶ The cons operation is well suited for this as it takes, as its arguments, an atom and a list.
- ▶ We also need to increment the size of the stack by one.

```
(defmethod push ((s stack) element)
  (setf (stack-elements s) (cons element (stack-elements s))
        (stack-size s) (+ 1 (stack-size s))))
```

## The Stack ADT: Defining primary operations - pop

- ▶ The pop operation would return the head of the list, as well as create a new list comprised by the tail of the current list, setting it as the new value for the stack.
- ▶ We also need to decrement the size of the stack by one.

```
(defmethod pop ((s stack))  
  (let ((top-element (car (stack-elements s))))  
    (setf (stack-elements s) (cdr (stack-elements s)))  
    (setf (stack-size s) (- (stack-size s) 1))  
    top-element))
```

## The Stack ADT: Defining secondary operations

```
(defmethod isempty ((s stack))  
  (equal (stack-size s) 0))
```

```
(defmethod isfull ((s stack))  
  (equal (stack-size s) (stack-capacity s)))
```

```
(defmethod top ((s stack))  
  (car (stack-elements s)))
```

## The Stack ADT: Defining auxiliary operations

```
(defmethod push :around ((s stack) element)
  (if (isfull s)
      "The stack is already full."
      (call-next-method s element)))
```

```
(defmethod pop :around ((s stack))
  (if (isempty s)
      "The stack is empty."
      (call-next-method s)))
```

```
(defmethod top :around ((s stack))
  (if (isempty s)
      "The stack is empty."
      (call-next-method s)))
```

## The Stack ADT: Instantiating stack

```
> (setq s (make-instance 'stack))
```

```
#<STACK 200934B3>
```

```
> (push s 3)
```

```
> (push s 4)
```

```
> (push s 5)
```

```
> (top s)
```

```
5
```

```
> (push s 6)
```

```
"The stack is already full."
```

```
> (pop s)
```

```
5
```

```
> (pop s)
```

```
4
```

```
> (pop s)
```

```
3
```

```
> (pop s)
```

```
"The stack is empty."
```

```
> (stack-size s)
```

```
0
```



# The Queue ADT: Structure and primary functionality

- ▶ The *queue* ADT is a collection that stores arbitrary objects.
- ▶ Insertions and deletions follow a *first-in first-out* (FIFO) scheme.
- ▶ There are two major queue operations:
- ▶ `enqueue(queue element)`: Inserts `element` at the rear of the queue.
- ▶ `dequeue()`: Removes and returns the element at the front of the queue.

## The Queue ADT: Secondary functionality

- ▶ `front()`: Returns the front element without removing it from the collection.
- ▶ `size()`: Returns the number of elements stored.
- ▶ `isEmpty()`: Returns a Boolean value indicating whether no elements are stored.
- ▶ `isFull()`: Returns a Boolean value indicating whether the collection has reached its capacity.

## The Queue ADT: Defining the structure

```
(defclass queue ()  
  ((elements :accessor queue-elements  
             :initarg :elements  
             :initform '()))  
  (size      :accessor queue-size  
             :initarg :size  
             :initform 0)  
  (capacity  :accessor queue-capacity  
             :initform 3  
             :allocation :class)))
```

# The Queue ADT: Defining primary operations

- ▶ In implementing a queue, we need to keep in mind that the two major operations access the queue from two different ends: the front and the rear.
- ▶ Recall that an ADT is an implementation-independent concept.
- ▶ This implies that it is up to us, the implementors of the ADT, to decide which end of the list we will consider as the front or the rear (as long as they are not the same end of the list).

# The Queue ADT: Defining primary operations - enqueue

- ▶ The enqueue operation adds an element to the rear and the dequeue operation removes an element from the front. We have two choices for this implementation:
- ▶ To consider the head of the list as the front of the queue. This implies that during enqueue, an element is added to the end of the list and during dequeue the head of the list is removed.
- ▶ To consider the head of the list as the rear of the queue. This implies that during enqueue an element is added to the head of the list and during dequeue the last element of the list is removed.

## The Queue ADT: Defining primary operations - enqueue /cont.

- ▶ The first choice is more convenient and we shall follow it here.
- ▶ To enqueue an element, we need to create a new list which is comprised with the current list and the element.
- ▶ How do we attach an element at the end of a current list?  
Function `cons` takes an element and a list, so that would not work.
- ▶ Function `append` can work, but it takes as arguments two lists. We can transform the element at hand into a list through the `list` function and then provide it as the second argument to `append`.
- ▶ We also need to increment the size of the queue by one.

## The Queue ADT: Defining primary operations - enqueue /cont.

```
(defmethod enqueue ((s queue) element)
  (setf (queue-elements s)
        (append (queue-elements s) (list element))))
(setf (queue-size s) (+ 1 (queue-size s))))
```

## The Queue ADT: Defining primary operations - dequeue

- ▶ To dequeue an element we simply have to return the head of the current list as well as to create a new list without the head element of the current list and set it as the new value to the queue.
- ▶ We also need to decrement the size of the queue by one.

```
(defmethod dequeue ((s queue))  
  (let ((top-element (car (queue-elements s))))  
    (setf (queue-elements s) (cdr (queue-elements s)))  
    (setf (queue-size s) (- (queue-size s) 1))  
    top-element))
```



## The Queue ADT: Defining secondary operations

```
(defmethod isempty ((s queue))  
  (equal (queue-size s) 0))
```

```
(defmethod isfull ((s queue))  
  (equal (queue-size s) (queue-capacity s)))
```

## The Queue ADT: Defining auxiliary operations

```
(defmethod enqueue :around ((s queue) element)
  (if (isfull s)
      "The queue is already full."
      (call-next-method s element)))

(defmethod dequeue :around ((s queue))
  (if (isempty s)
      "The queue is empty."
      (call-next-method s)))
```

## The Queue ADT: Instantiating queue

```
> (setq q (make-instance 'queue))
#<QUEUE 200BFCD7>
> (enqueue q 3)
> (enqueue q '(a b))
> (enqueue q 7)
> (enqueue q 11)
"The queue is already full."
> (dequeue q)
3
> (dequeue q)
(A B)
> (dequeue q)
7
> (dequeue q)
"The queue is empty."
```