Object oriented programming

- ✔ The main ideas behind object-oriented programming:
  - ✗ the same type of action may be carried out differently on different types of objects
    - cutting a cake is different from cutting cloth
    - driving a car is different from driving a bicycle
  - ✗ however, a type of action may involve similar subactions when carried out on similar objects
    - mailing a letter involves many of the same moves as mailing a postcard
    - eating at a French restaurant involves many of the same procedures as eating at a Chinese restaurant
  - ✗ a useful way of describing similarities among objects is by describing their place in an object type hierarchy
    - a Chinese restaurant is a type of Asian restaurant, which is a type of restaurant, which is a type of retail food outlet, which is a type of retail outlet, which is a type of business, ...
- ✔ So, think about procedures in relation to a hierarchy of object types

Object oriented programming

- There are two kinds of object-oriented programming perspectives

  - message-passing systems: “object oriented perspective”

    - in a message passing system, messages are sent to objects
    - what happens depends on what the message is, and what type the object is
    - each object type “knows” what to do (what method to apply) with each message

```
(send starship-1 'move-to-andromeda)
```

  - generic function systems: “data-driven perspective”

    - in a generic function system, objects are passed as arguments to generic functions
    - what happens depends on what the function is, and what type the object is
    - each generic function “knows” what to do (what method to apply) with each object type

```
(move-to-andromeda starship-1)
```

- The Common LISP Object System is a generic function system

The Common LISP Object System (CLOS)

- ✔ CLOS classes are similar to LISP structure types... but different!
- ✔ CLOS generic functions are similar to ordinary LISP functions...
- ✔ Example: defining types for some geometrical figures and a function to compute their area, using structures

```
(defstruct triangle (base 0) (altitude 0))

(defstruct rectangle (width 0) (height 0))

(defstruct circle (radius 0))


(defun area (figure)
   (cond  ((triangle-p figure)
           (* 1/2 (triangle-base figure)
                  (triangle-altitude figure)))
          ((rectangle-p figure)
           (* (rectangle-width figure)
              (rectangle-height figure)))
          ((circle-p figure)
           (* pi (expt (circle-radius figure) 2)))))
```

  - ✗ the area function is complicated
  - ✗ the area function must be edited and made more complicated if you add another type of geometrical figure

Generic functions in CLOS

- Example: using a *generic function* to compute their area
  - a generic function is a collection of *methods* with the same name
  - a method is a CLOS function that is *specialized* on a certain data type or types

```
(defmethod area ((figure triangle))
    (* 1/2 (triangle-base figure)
           (triangle-altitude figure)))

(defmethod area ((figure rectangle))
    (* (rectangle-height figure)
       (rectangle-width figure)))

(defmethod area ((figure circle))
    (* pi (expt (circle-radius figure))))
```

  - the area methods are simple
  - the area generic function can be easily extended by defining additional area methods

Generic functions in CLOS

- ✔ Define a method on a generic function by using `defmethod`

```
(defmethod function-name specialized-lambda-list
    expression-1
    ...
    expression-n)
```

- ✔ `defmethod` looks just like `defun` except the lambda list is a *specialized lambda list*

- ✔ a specialized lambda list is just like an ordinary function lambda list, except that the name of a required parameter can be replaced by a *specialized parameter*

- ✔ a specialized parameter is a list of the form

  `(symbol typename)`

  or

  `(symbol (eql expression))`

- ✔ for a method to be applicable, all required arguments must match the corresponding specialized parameter:

  - ✗ must be of the type *`typename`* if the specialized parameter is of the form `(symbol typename)`

  - ✗ must be eql to *`expression`* if the specialized parameter is of the form `(symbol (eql expression))`

Classes in CLOS

- ✔ Classes resemble structure types, with some differences:
  - ✗ both have named slots, and default values can be specified
  - ✗ structure types have automatically defined slot accessor functions
  - ✗ classes have slot accessor generic functions which must be specified
  - ✗ structures have automatically defined keyword initial value specifiers
  - ✗ classes have keyword initial value specifiers, but these must be specified
  - ✗ each structure type has an automatically defined instance creation function
  - ✗ all classes use the function `make-instance` for instance creation
  - ✗ both structure types and classes are included in the LISP type hierarchy; `typep` and `type-of` work with instances
  - ✗ a class can be a subtype of another class. This user-defined class hierarchy interacts in useful ways with methods on generic functions

Classes in CLOS

✔ Define a class in CLOS with the `defclass` macro

```
(defclass class-name list-of-direct-superclasses
    ((slot-name-1 :accessor accessor-1
                  :initform init-expression-1
                  :initarg :init-keyword-1)
       ...

     (slot-name-N :accessor accessor-N
                  :initform init-expression-N
                  :initarg :init-keyword-1))
```

Classes in CLOS

- ✔ Consider this class hierarchy

p. 188

top

winston & horn

Classes in CLOS

- ✔ We can define the Article class hierarchy in CLOS:

```
(defclass article ()
    ((title :accessor title :initarg :title)
     (author :accessor author :initarg :author))
```

- ✔ This defines a class named `article`
- ✔ `article` has no direct superclasses
- ✔ every instance of the `article` class will have two slots:
  - ✗ `title`, with accessor method `title` and initial value specifier `:title`
  - ✗ `author`, with accessor method `author` and initial value specifier `:author`
- ✔ an instance of the article class can now be created with a call to make-instance

```
USER: (make-instance 'article :title "LISP Hacking"
                     :author "I. M. Hacker")

#<ARTICLE @ #x2344a0e>
```

Classes in CLOS

- ✔ Defining the article hierarchy:

```
(defclass computer-article (article) ())

(defclass business-article (article) ())

(defclass political-article (article) ())
```

- ✔ These classes have the article class as direct superclass
- ✔ These classes inherit all slot properties from the superclass that they do not change
  - ✗ in this example, they do not change any inherited slot properties
- ✔ These classes can add new slots
  - ✗ in this example, they do not add any new slots
- ✔ These classes are added to the LISP type hierarchy

```
USER: (setf my-art (make-instance 'computer-article))
#<COMPUTER-ARTICLE @ #x123df0>

USER: (typep my-art 'computer-article)
T

USER: (typep my-art 'article)
T
```

Classes in CLOS

- ✔ Defining the friend hierarchy

```
(defclass friend ()
   ((name :accessor name :initarg :name))

(defclass hacker-friend (friend) ())

(defclass entrepreneur-friend (friend) ())

(defclass philosopher-friend (friend) ())
```

Generic functions and classes in CLOS

- Suppose we have a list of articles of various classes, and a list of friends of various classes
- We want to tell friends about articles they might be interested in
- Here is a matrix of possible interest relationships:

p. 191
bottom
winston & horn

- We will define methods that implement these relationships...

Generic functions and classes in CLOS

- Suppose there is an ordinary function `mail-notify` which takes two arguments, a friend and an article, and sends mail to the friend about the article
- We can write a collection of methods on the generic function process which will help us send mail only to people interested in particular articles...

```
;; send hacker-friend a computer-article
(defmethod process   ((friend hacker-friend)
                      (article computer-article))
   (mail-notify friend article))



;; send entrepreneur-friend a business-article
(defmethod process   ((friend entrepreneur-friend)
                      (article business-article))
   (mail-notify friend article))



;; send philosopher-friend any article
(defmethod process   ((friend philosopher-friend)
                      (article article))
   (mail-notify friend article))



;; do nothing on any other combinations
(defmethod process   ((friend t)
                      (article t))
   )
```

Generic functions and classes in CLOS

- ✔ Now if `friends` is a list of instances of `friend`, and `articles` is a list of instances of `article`,

```
(dolist (friend friends)
   (dolist (article articles)
      (process friend article))
```

will send notifications only to those people we want...