Method inheritance in CLOS

- We have the following class hierarchy, together with "printing" and "non-printing" methods of the `process` generic function:

fig 14.2
Winston & Horn

Method inheritance in CLOS

- ✔ We can extend the class heirarchy with two new classes:

```
(defclass stocks-article (business-article) ())

(defclass new-stocks-article (stocks-article) ())
```

- ✔ Now suppose we want not to notify entrepreneur friends about stocks articles, while notifying them about new-stocks-articles and other business-articles. We can define new methods

```
(defmethod process   ((friend entrepreneur-friend)
                       (article new-stocks-article))
       (mail-notify friend article))
```

```
(defmethod process   ((friend entrepreneur-friend)
                       (article stocks-article))
       )
```

- ✔ This works, because the class precedence algorithm for method inheritance says the most specialized applicable method takes precedence

Method inheritance in CLOS

- ✔ CLOS implements rules for deciding which of a generic function's methods is actually applied in a given call to the generic function
  - ✗ Each required argument's type is checked against the corresponding specialized parameter in the method
  - ✗ If all arguments are of the types specified by the parameters' specializers, the method is *applicable*
    - an argument `arg` is considered of a type `type` if `(typep arg 'type)` is T
    - thus if an argument `arg` is of a type `type`, it is also of every type that is a supertype of `type`
  - ✗ Of all the applicable methods, the one with the most specific parameter specializers takes precedence and is actually applied
- ✔ Deciding which applicable method has the most specific parameter specializers can get complicated...

Method inheritance in CLOS

- ✔ We now have the following class hierarchy, together with "printing" and "non-printing" methods of the `process` generic function:

fig. 14-3

Winston & Horn

Method inheritance in CLOS

- ✔ We can further extend the class hierarchy with a new class:

```
(defclass retired-hacker-friend (hacker-friend) ())
```

- ✔ And we can write two new methods on the `process` generic function:

```
(defmethod process   ((friend retired-hacker-friend)
                      (article business-article))
    (mail-notify friend article))


(defmethod process   ((friend hacker-friend)
                      (article stocks-article))
    )
```

- ✔ But the first of these is more specialized on its first required parameter, and the second is more specialized on its second required parameter...

- ✔ Question: Which one takes precedence in the case

```
(process   (make-instance 'retired-hacker-friend)
           (make-instance 'stocks-article))
```

Method inheritance in CLOS

- Answer: precedence is determined by left-to-right consideration of parameter specializers in the applicable methods
  - First, the applicable methods whose first required parameter are most specific are considered; other applicable methods are eliminated
  - Next, of the remaining applicable methods, those whose second required parameter are most specific are considered; other applicable methods are eliminated
  - Next, the third required parameter is considered, etc., until only one applicable method remains; this one takes precedence
- Therefore, in this case, the retired-hacker-friend is notified about the stocks-article

Method inheritance in CLOS

- ✔ We now have the following class hierarchy, together with "printing" and "non-printing" methods of the `process` generic function:

fig 14-4

Winston & Horn

Method inheritance with multiple superclasses

- We can extend the class hierarchy by defining a class that has two direct superclasses:

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

- We already have the following method, to notify hacker-friends about computer-articles:

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

- We can define another method, attempting to explicitly not notify hacker-friends about political articles:

```
(defmethod process ((friend hacker-friend)
                    (article political-article))
  )
```

- Question: which one takes precedence in the case

```
(process  (make-instance 'hacker-friend)
          (make-instance 'computer-political-article))
```

Method inheritance with multiple superclasses

- We now have the following class hierarchy, together with "printing" and "non-printing" methods of the `process` generic function:

fig 14-5

Winston & Horn

Method inheritance in CLOS

- ✔ Which class is more specialized than another is determined by the ordering of classes in a *class precedence list*

- ✔ Three simple rules suffice to construct the class precedence list in most cases:
  - ✗ the depth first rule
  - ✗ the left-to-right rule
  - ✗ the up-to-join rule

The depth-first rule

- Until reaching a *split* or a *join*, traverse the class hierarchy upward, adding classes to the class precedence list

fig. 14-6. left

Winston & Horn

The left-to-right rule

- A class with more than one direct superclass is a *split*. At a split, traverse the hierarchies above the split in left-to-right order

fig. 14-6 . middle

Winston & Horn

The up-to-join rule

- A superclass that can be reached in more than one way is a *join*. Each time a join is encountered, except for the last time, hierarchy traversal stops, and the join is not added to the class precedence list. The last time the join is encountered, the join iis added to the class precedence list and traversal continues at the join.

fig. 14-6. right

Winston & Horn

Method inheritance with multiple superclasses

- ✔ In the example, the class precedence list is:

```
computer-political-article ;;split
computer-article     ;;leftmost superclass above split
political-article    ;;rightmost superclass above split
article              ;;join
standart-object      ;;implicit depth-first above join
T                    ;;implicit depth-first
```

Method inheritance with multiple superclasses

- ✔ Therefore, in the case

```
    (process   (make-instance 'hacker-friend)
               (make-instance 'computer-political-article))
```

the method

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

takes precedence over the method

```
(defmethod process ((friend hacker-friend)
                        (article political-article))
   )
```

Specializing on individual instances

- Suppose you don't want to send any notification about any articles to the friend instance which is the third element of your list of friends... You can do this with an `eql` specializer:

```
(defmethod process ((friend (eql (third friends)))
                    (article article))
  )
```

- An `eql` specializer is more specific than any class specializer...

- Suppose you don't want to send any notification to anyone about the second article in your list of articles... You can try to do this with an `eql` specializer:

```
(defmethod process ((friend friend)
                    (article (eql (second articles))))
  )
```

Will this work? Why or why not?