



---

# Einführung in die Programmierung Introduction to Programming

Prof. Dr. Bertrand Meyer

Exercise Session 10



- Multiple inheritance

# Combining abstractions

---



Given the classes

➤ TRAIN\_CAR, RESTAURANT

how would you implement a DINER?

# Inheritance is never the only way

---



Given the classes

➤ TRAIN\_CAR, RESTAURANT

how would you implement a DINER?

➤ You could have an attribute in TRAIN\_CAR

train\_service: SERVICE

➤ Then have RESTAURANT inherit from SERVICE

➤ This is flexible if the kind of service may change to a type that is unrelated to TRAIN\_CAR

➤ Changes in TRAIN\_CAR do not affect SERVICE easily

# Examples of multiple inheritance

Hands-On

Combining separate abstractions:

- Restaurant, train car
- Calculator, watch
- Other examples?



# Examples of multiple inheritance

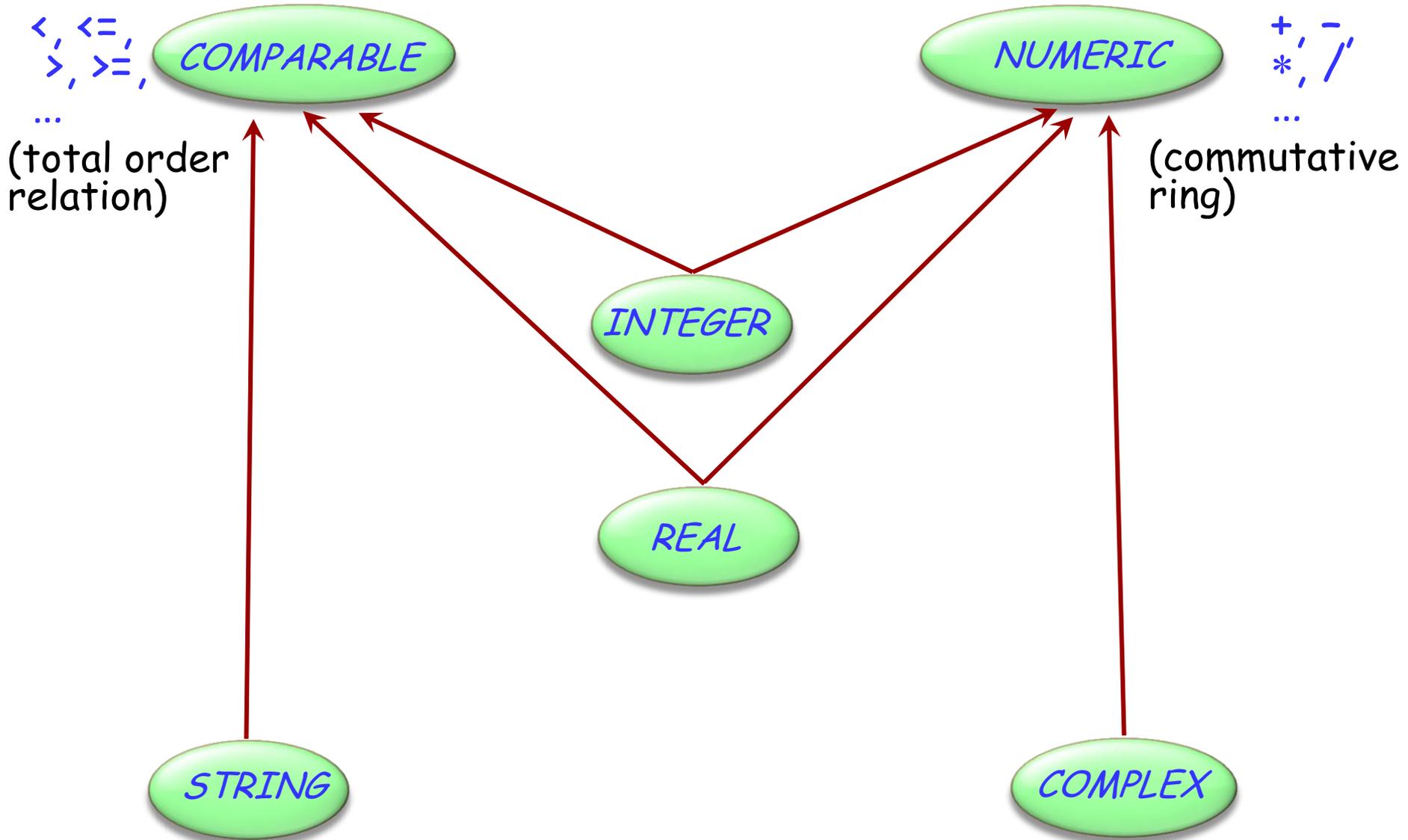
---

Hands-On

Combining separate abstractions:

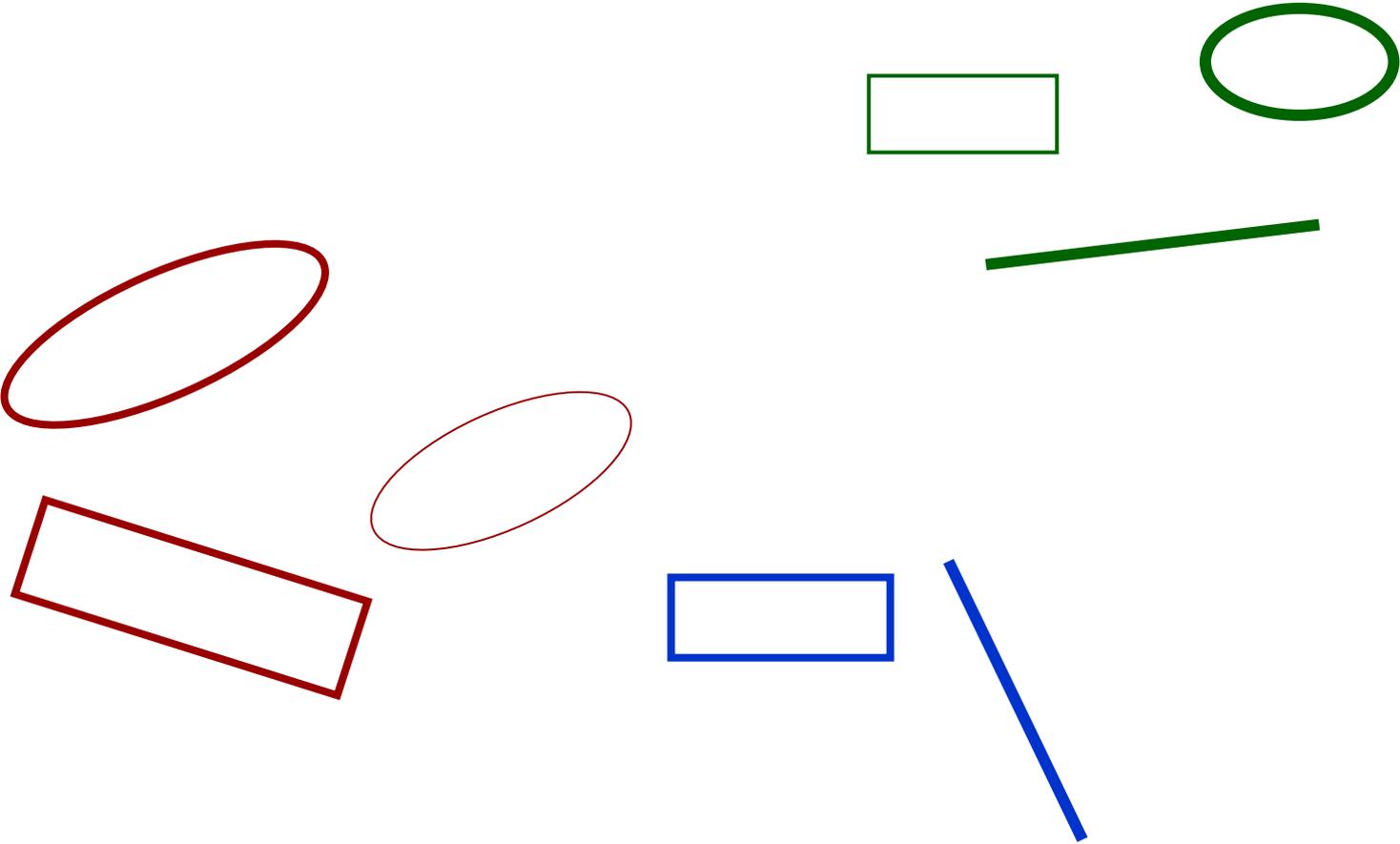
- Restaurant, train car
- Calculator, watch
  
- Other examples?
  
- Teacher, student
- Home, vehicle

# Multiple inheritance: Combining abstractions

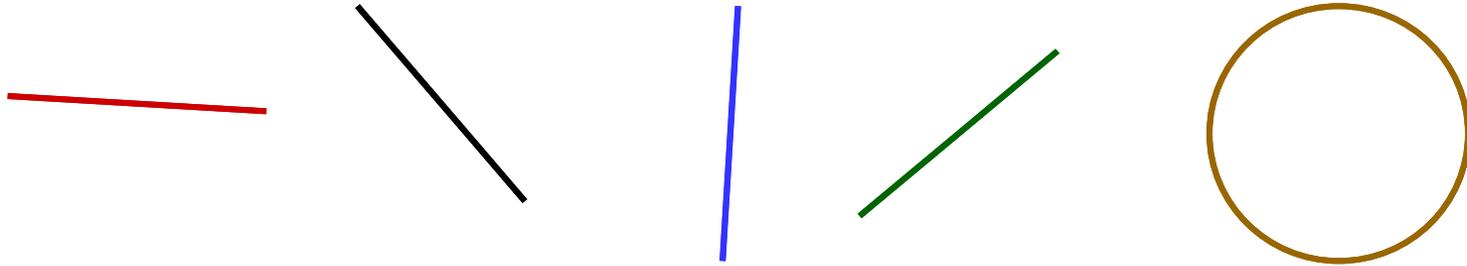


# Composite figures

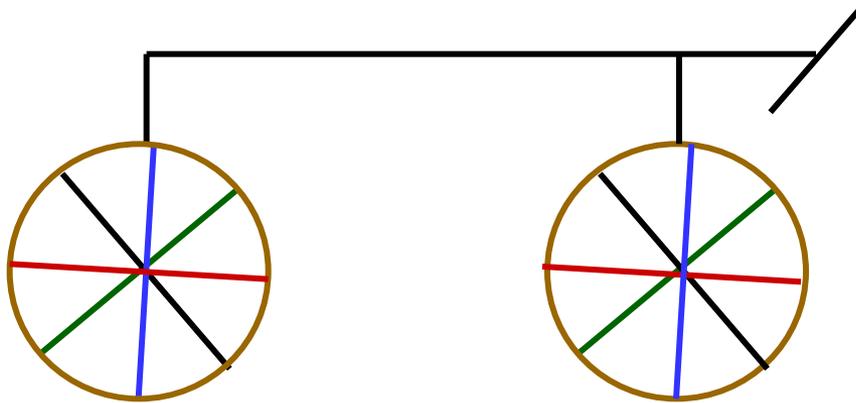
---



# Multiple inheritance: Composite figures



Simple figures



A composite figure

# Defining the notion of composite figure



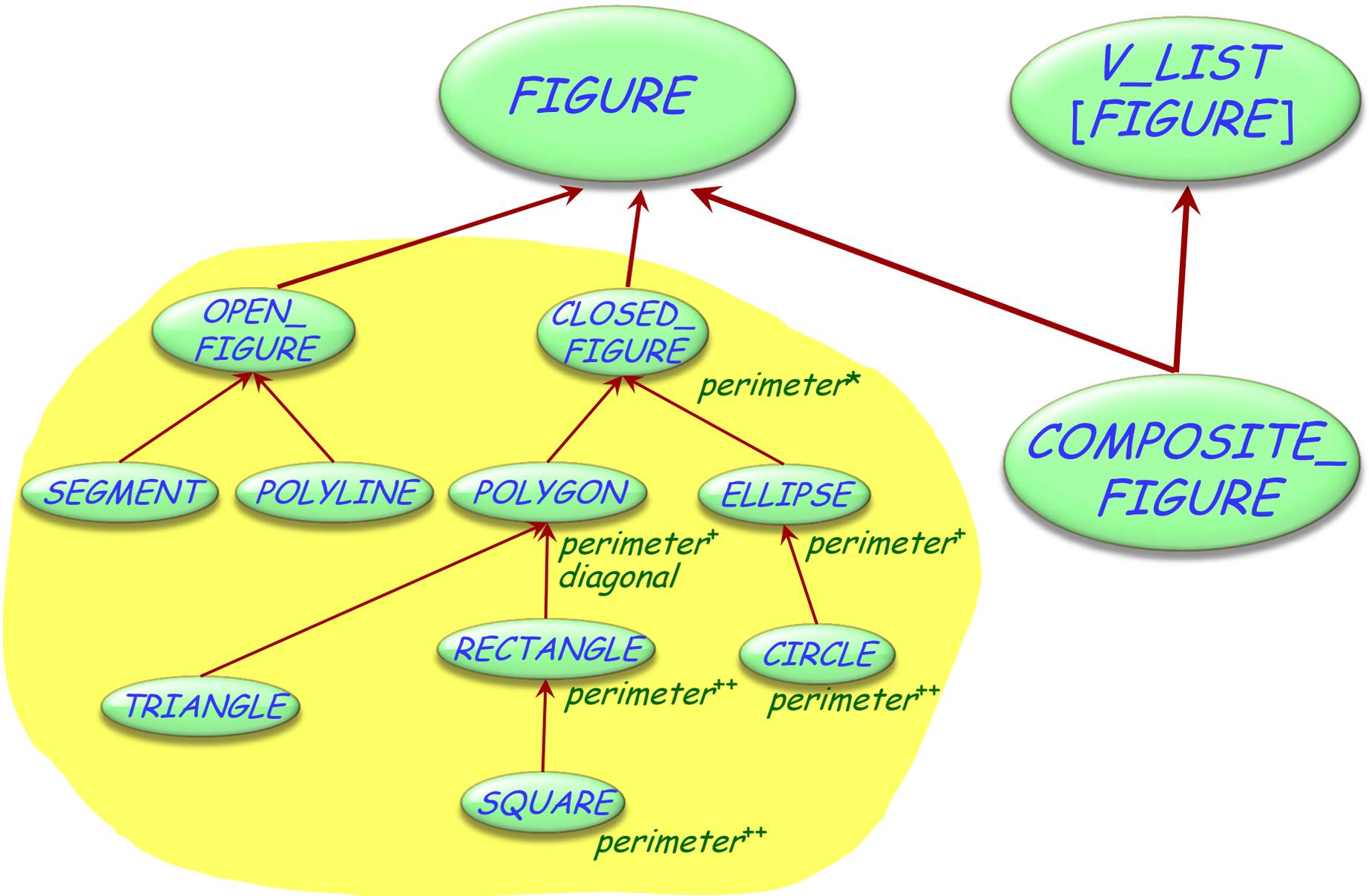
*center  
display  
hide  
rotate  
move  
...*



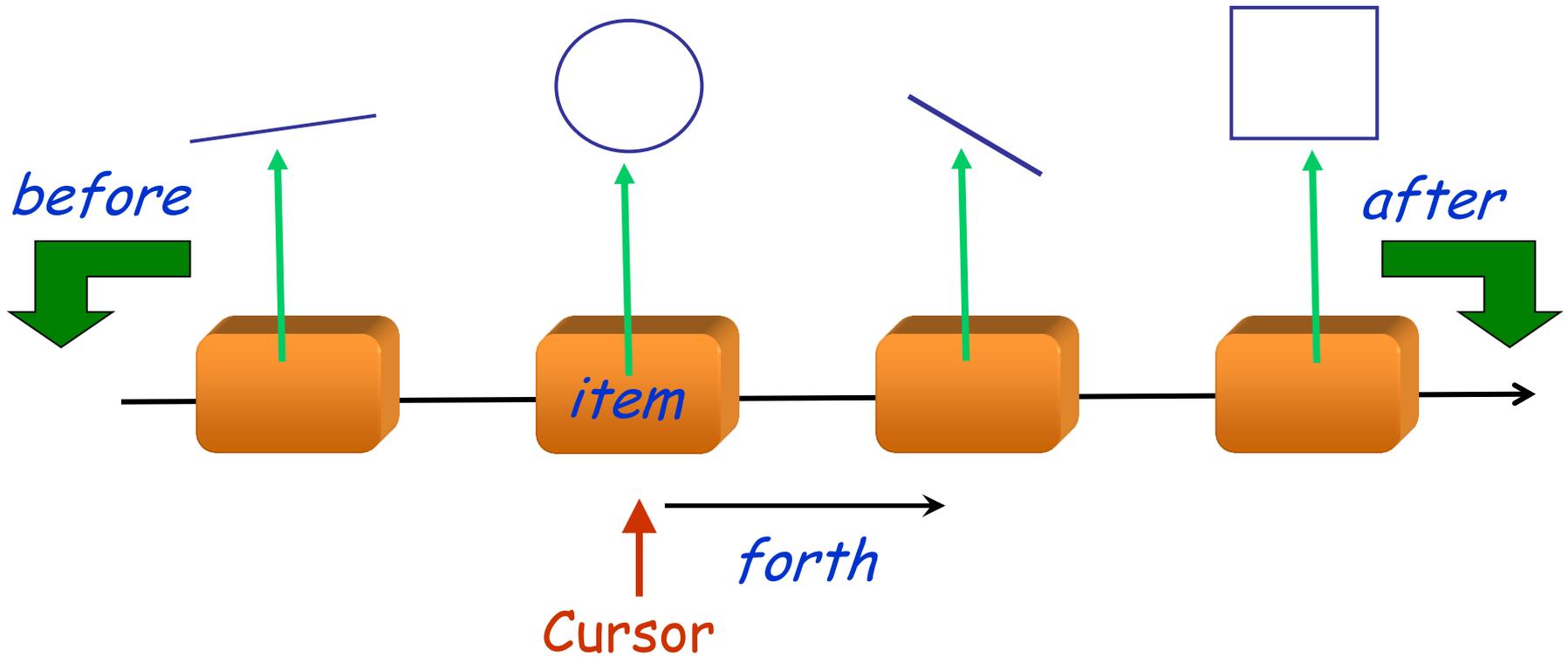
*count  
put  
remove  
...*



# In the overall structure



# A composite figure as a list



# Composite figures

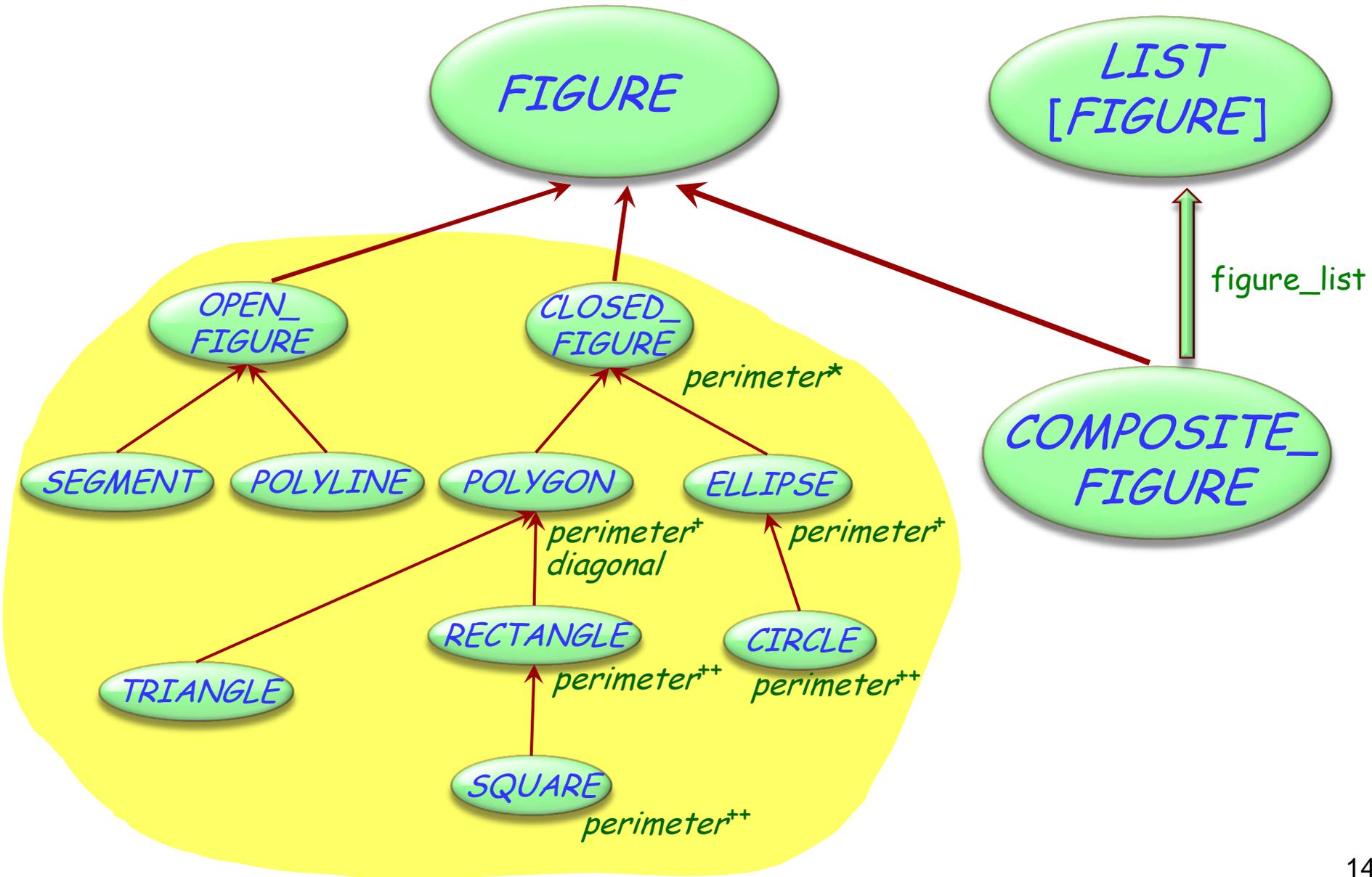


```
class COMPOSITE_FIGURE inherit
    FIGURE
    V_LIST[FIGURE]
feature
    display
do
    -- Display each constituent figure in turn.
    from start until after loop
        item.display
    forth
end
end
... Similarly for move, rotate etc. ...
end
```

*item.display*

Requires dynamic binding

# An alternative solution: the composite pattern



# Lessons from this example

---



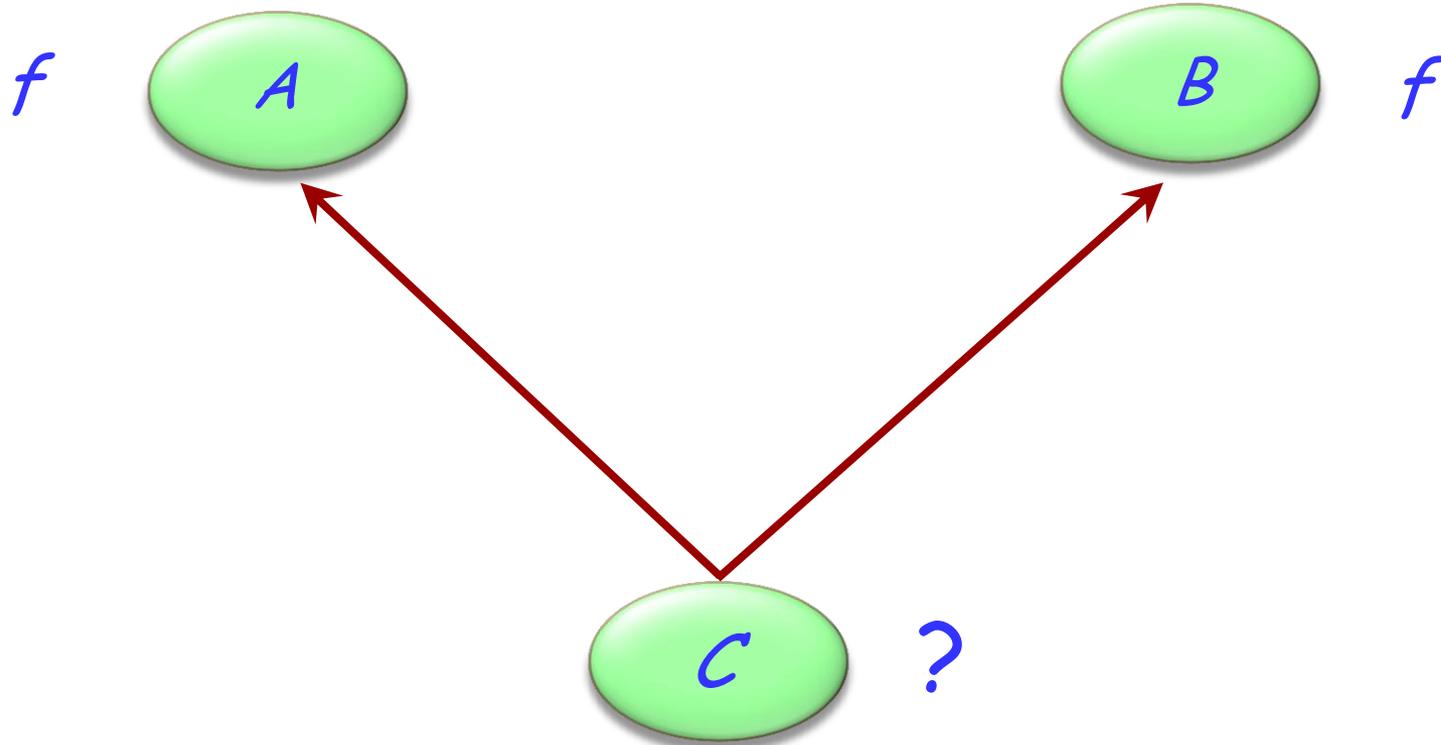
Typical example of *program with holes*

We need the full spectrum from fully abstract (fully deferred) to fully implemented classes

Multiple inheritance is there to help us combine abstractions

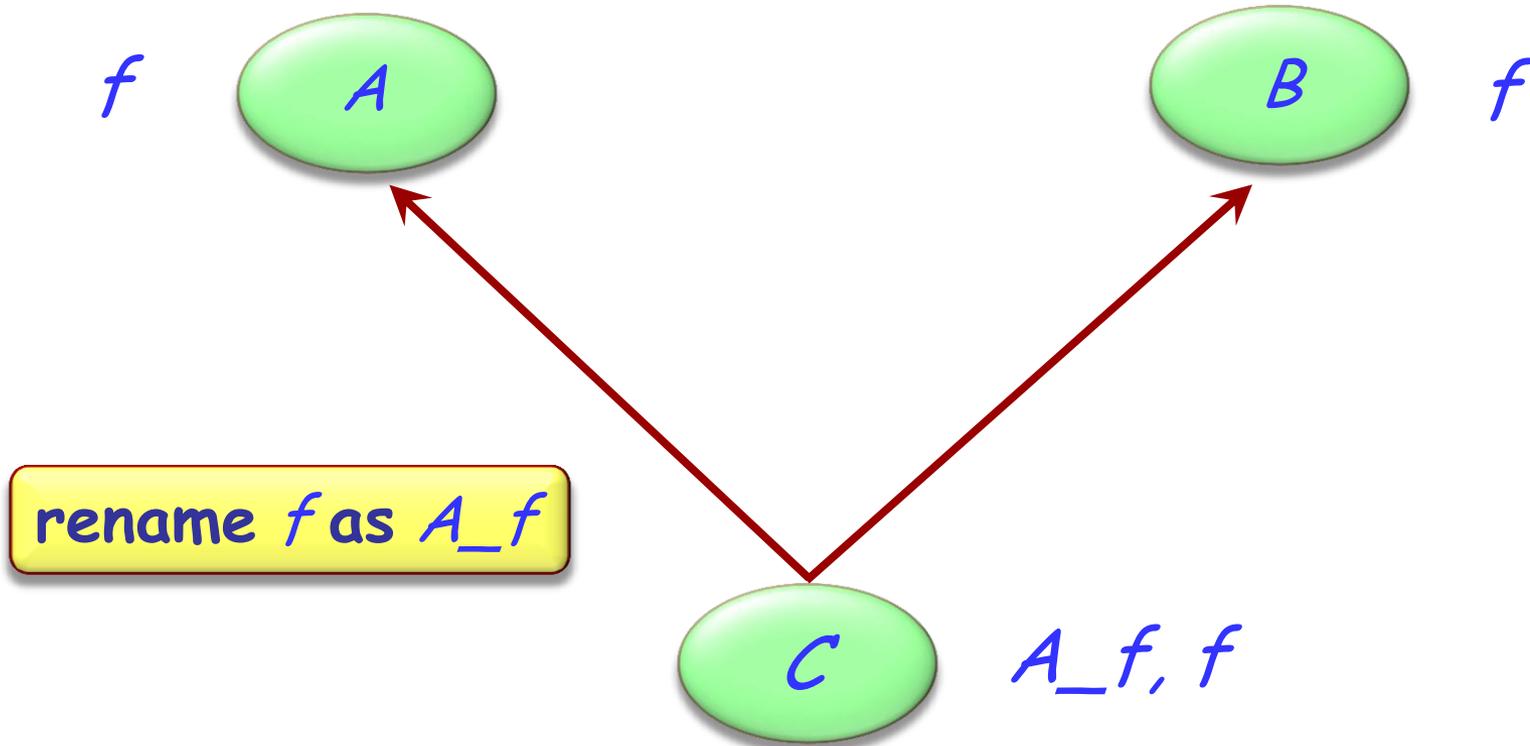
# Multiple inheritance: Name clashes

Hands-On



# Resolving name clashes

Hands-On



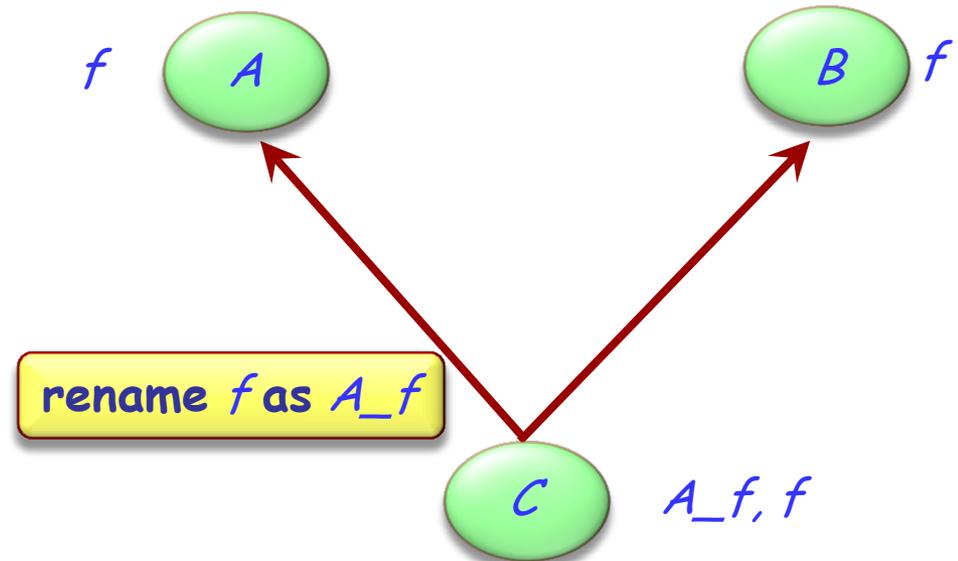
# Consequences of renaming

Hands-On

*Valid or invalid?*

*a1: A*  
*b1: B*  
*c1: C*  
...

<i>c1.f</i>	Valid
<i>a1.A_f</i>	Invalid
<i>c1.A_f</i>	Valid
<i>b1.f</i>	Valid
<i>b1.A_f</i>	Invalid



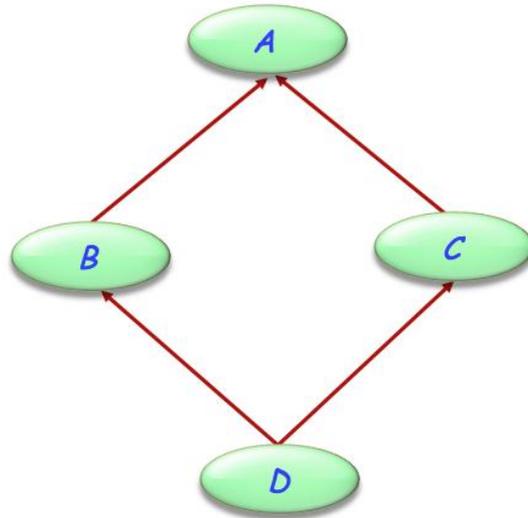
# Are all name clashes bad?



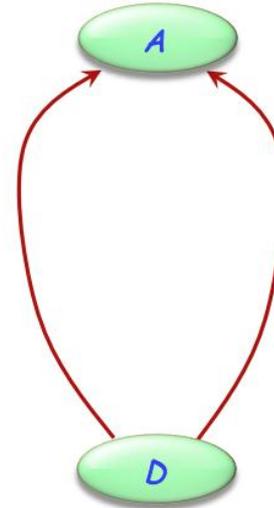
A name clash must be removed unless it is:

- Under repeated inheritance (i.e. not a real clash)

indirect repeated inheritance

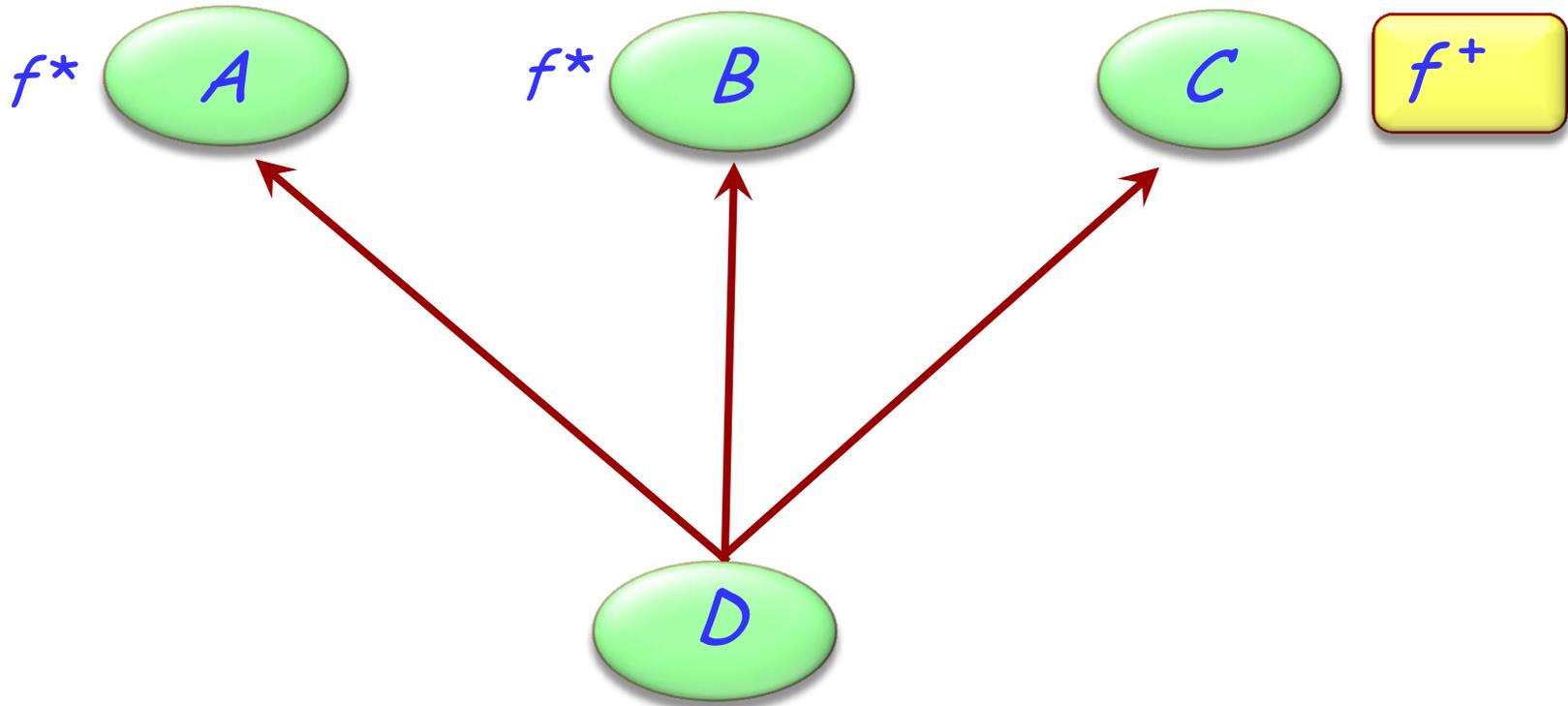


direct repeated inheritance



- Between features of which at most one is effective (i.e. others are deferred)

# Feature merging

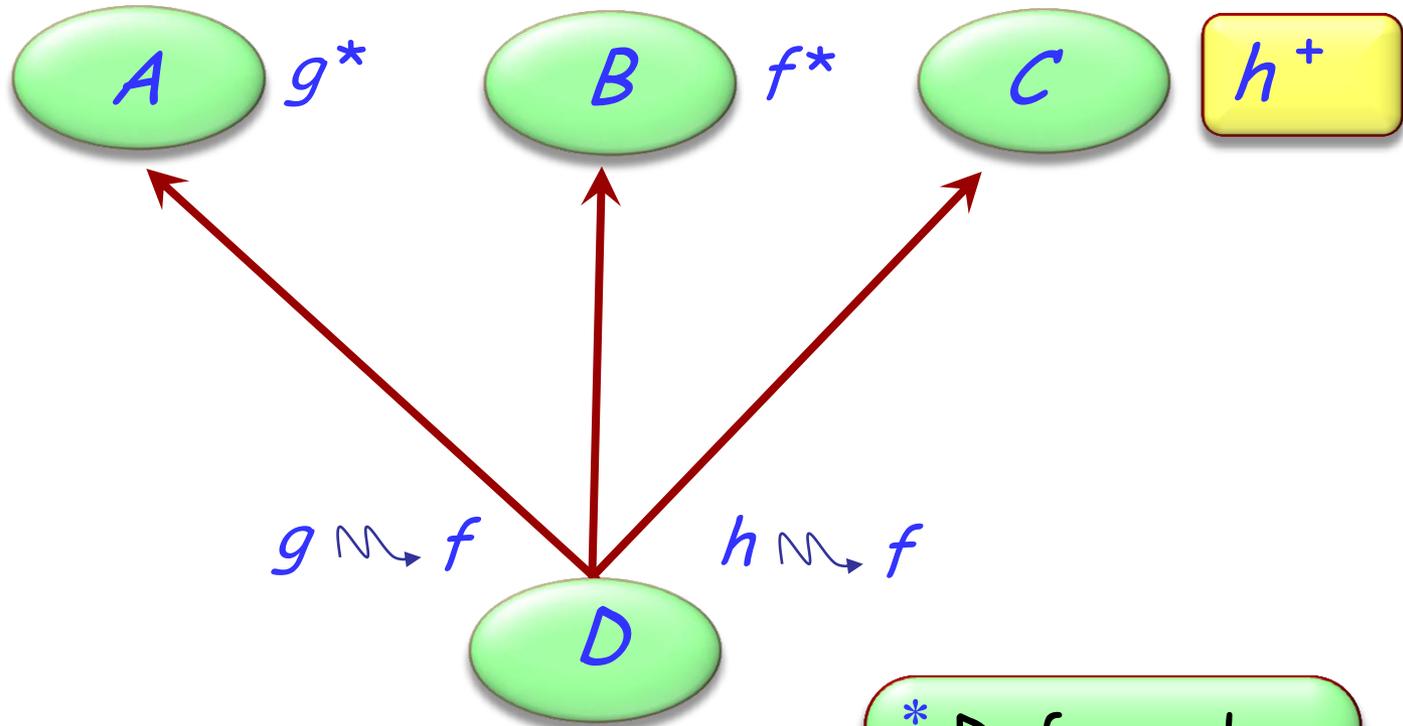


\* Deferred  
+ Effective

# Feature merging: with different names

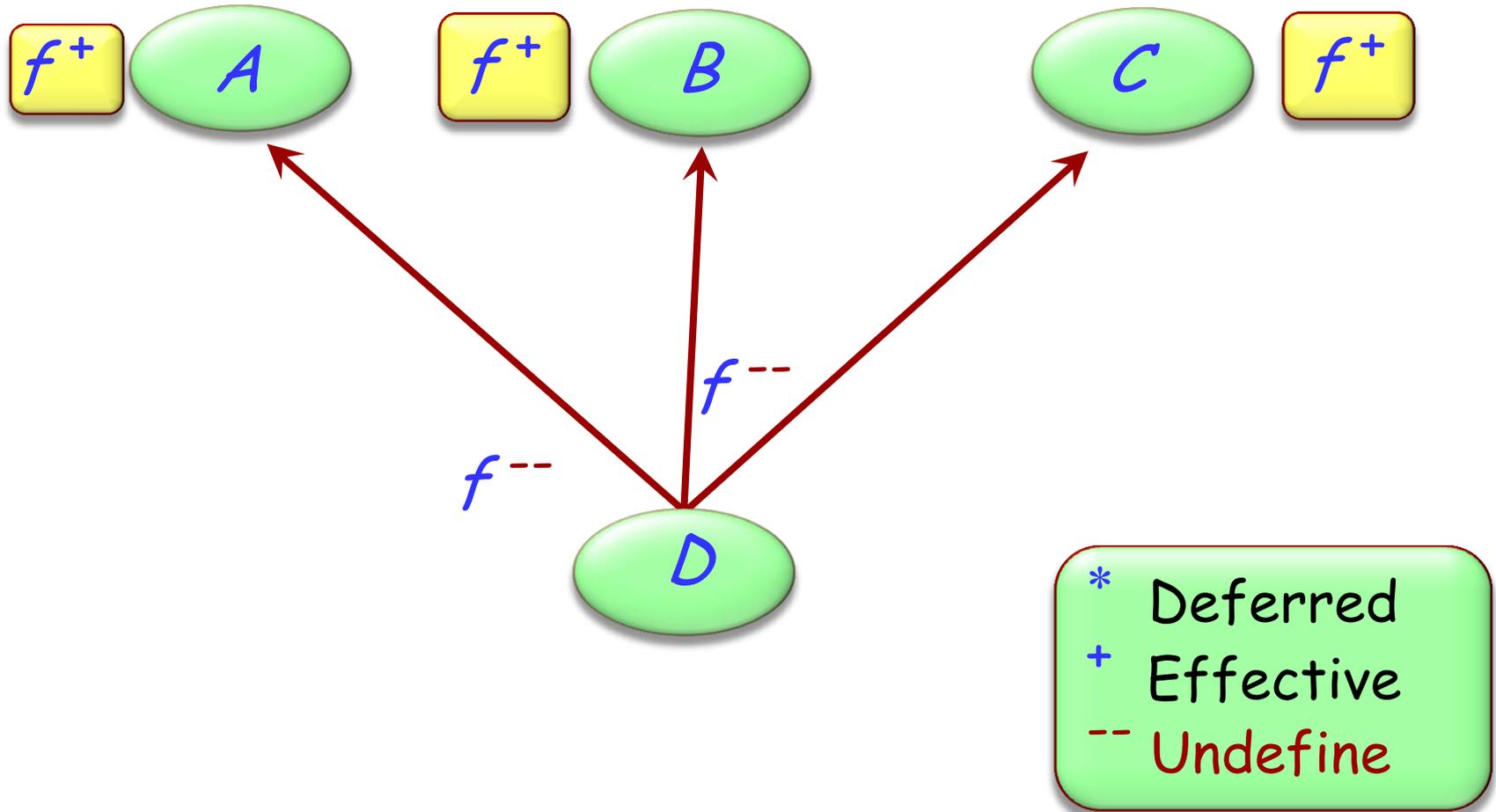


```
class
  D
inherit
  A
  rename
    g as f
  end
  B
  C
  rename
    h as f
  end
feature
  ...
end
```



\* Deferred  
+ Effective  
 $\rightsquigarrow$  Renaming

# Feature merging: effective features





deferred class

*T*

inherit

*S*

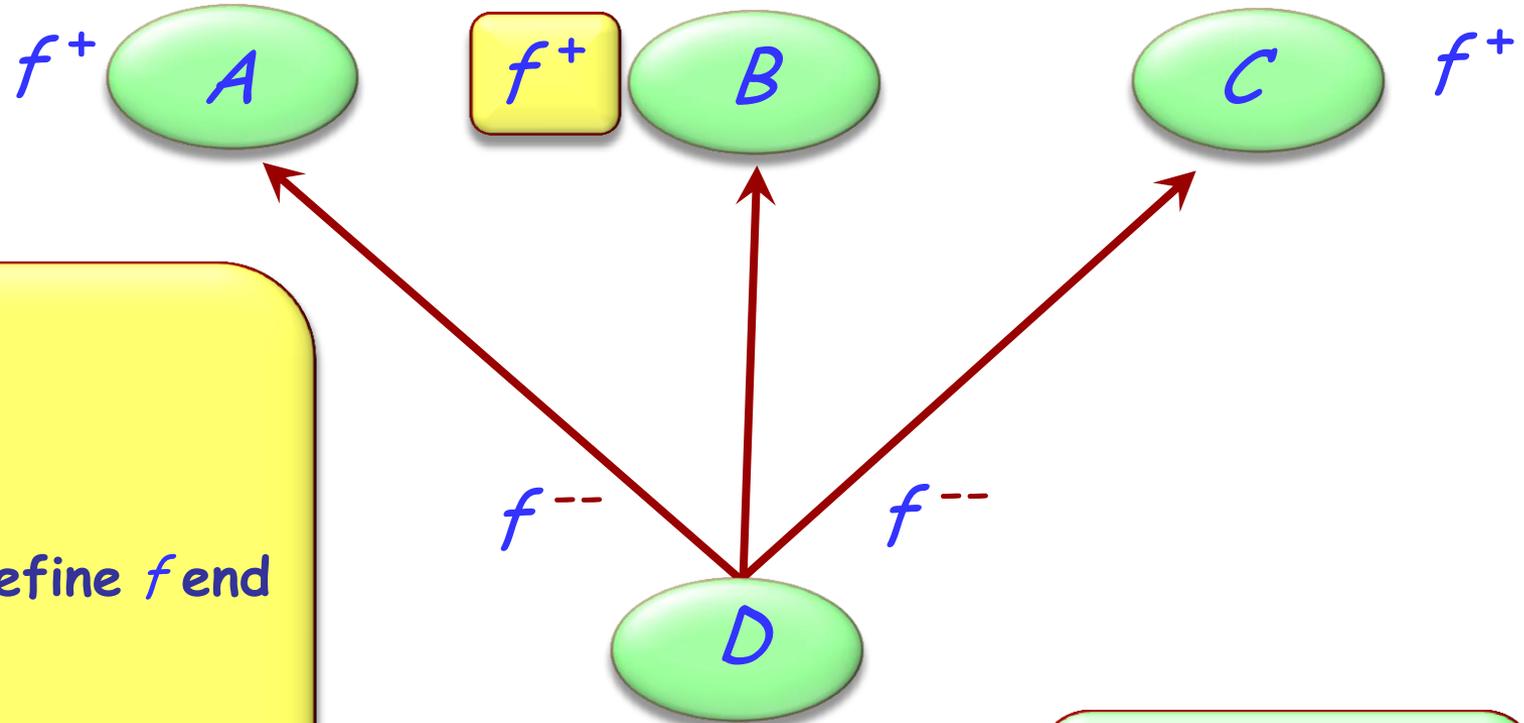
undefine *v* end

feature

...

end

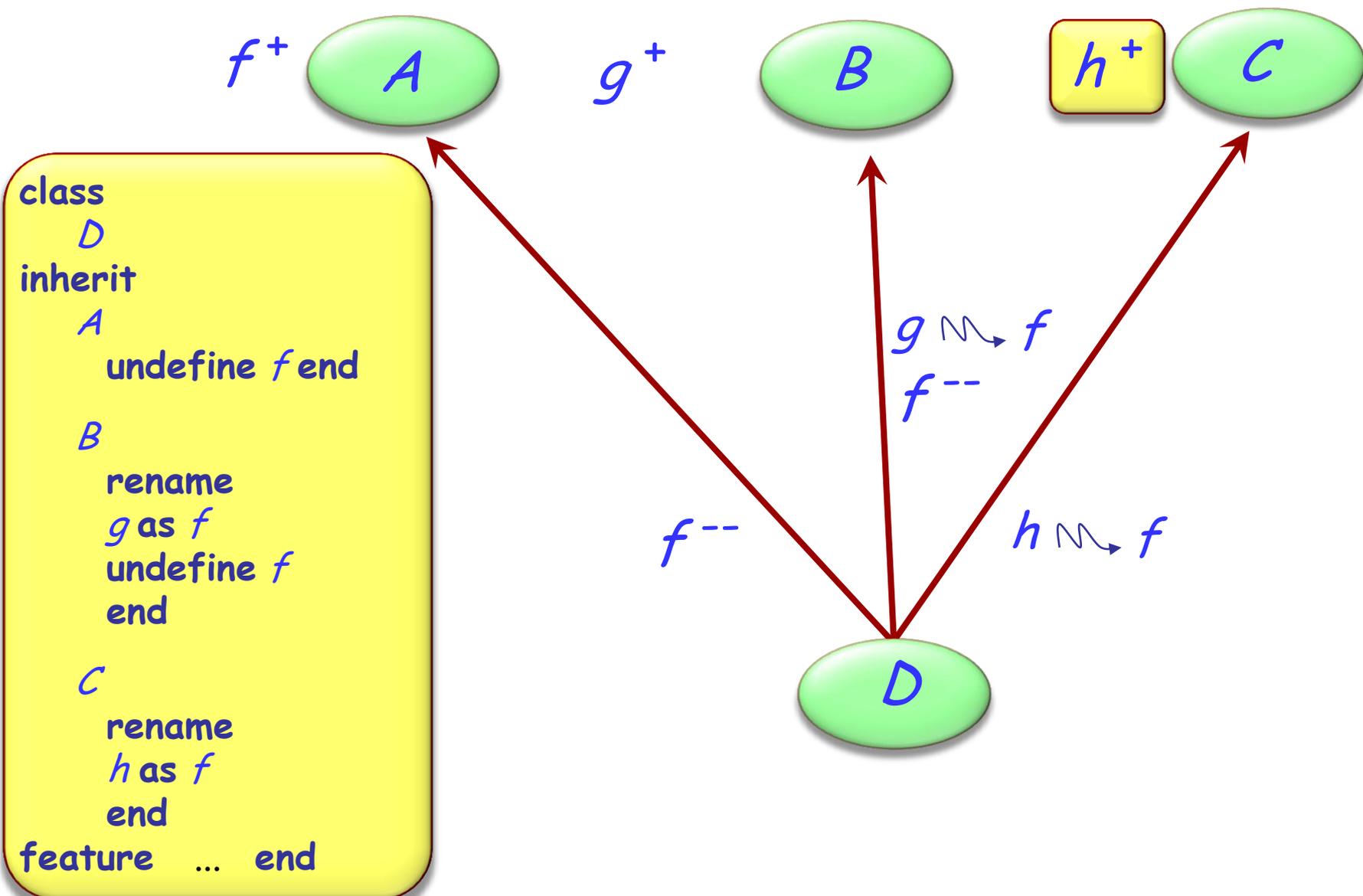
# Merging through undefinition



```
class
  D
inherit
  A
  undefine f end
  B
  C
  undefine f end
feature
  ...
end
```

\* Deferred  
+ Effective  
-- Undefine

# Merging effective features with different names





If inherited features have all the same names, there is no harmful name clash if:

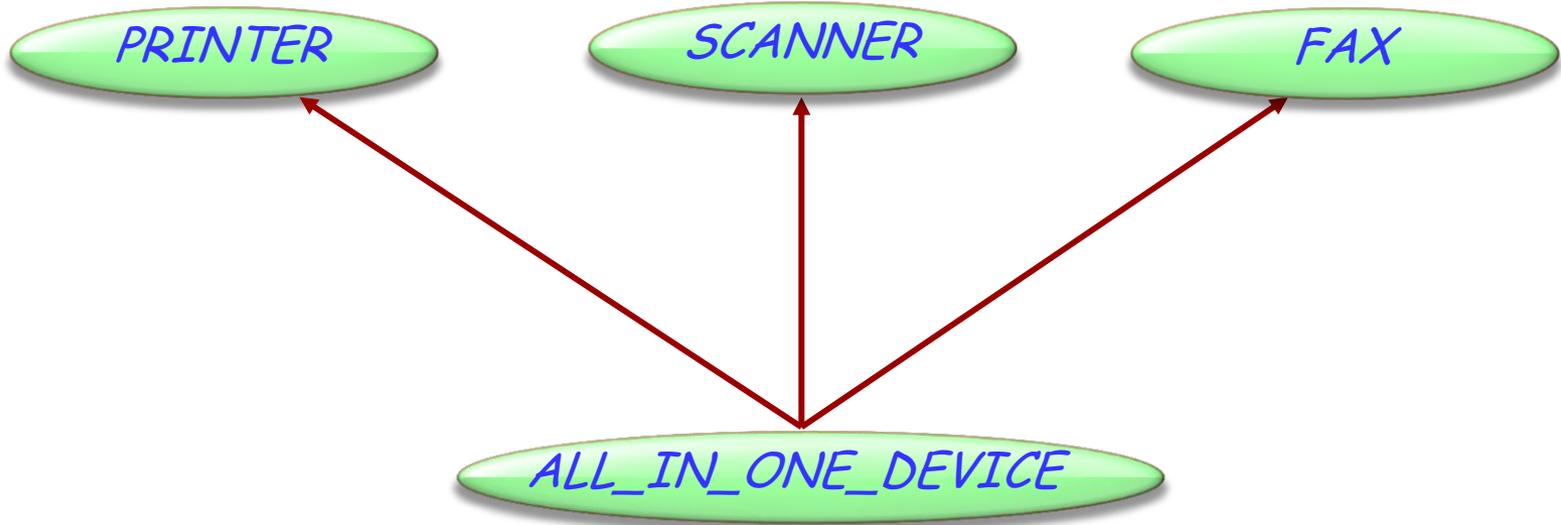
- They all have compatible signatures
- At most one of them is effective

Semantics of such a case:

- Merge all features into one
- If there is an effective feature, it imposes its implementation

# Exercise: All-in-one-device

Hands-On



# Exercise: All-in-one-device

Hands-On

```
class PRINTER
  feature
    print_page -- Print a page.
    do
      print ("Printer prints a page...")
    end

    switch_on -- Switch from 'off' to 'on'
    do
      print ("Printer switched on...")
    end

  end

class FAX
  feature
    send -- Send a page over the phone net.
    do
      print ("Fax sends a page...")
    end

    start -- Switch from 'off' to 'on'
    do
      print ("Fax switched on...")
    end

  end

end
```

```
class SCANNER
  feature
    scan_page -- Scan a page.
    do
      print ("Scanner scans a page...")
    end

    switch_on -- Switch from 'off' to 'on'
    do
      print ("Scanner switched on...")
    end

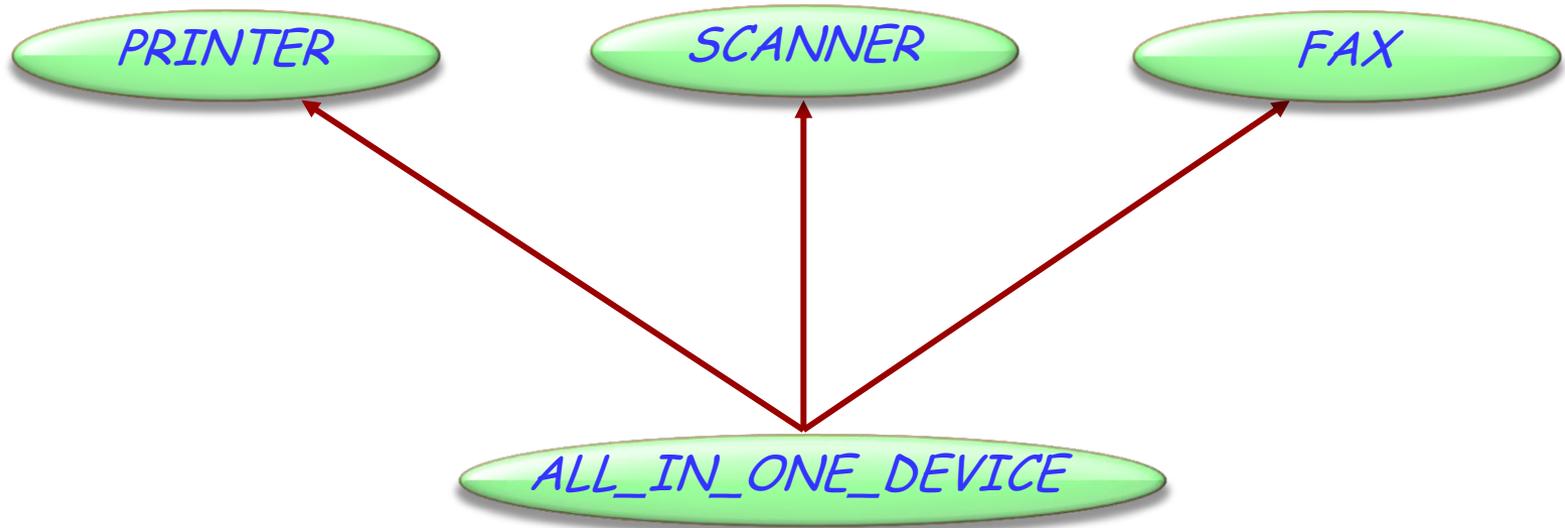
    send -- Send data to PC.
    do
      print ("Scanner sends data...")
    end

  end

end
```

# Exercise: All-in-one-device

Hands-On



```
class
  ALL_IN_ONE_DEVICE
inherit
  ...
end
```

How to resolve the name clashes?

- switch\_on
- send

# Exercise: All-in-one-device

Hands-On



```
class ALL_IN_ONE_DEVICE

inherit
  PRINTER
    rename
      switch_on as start
    undefine
      start
    end

  SCANNER
    rename
      switch_on as start,
      send as send_data
    end

  FAX
    rename
      send as send_message
    undefine
      start
    end

feature ... end
```

# Valid or invalid?

Hands-On

```
class ALL_IN_ONE_DEVICE
```

```
  inherit
```

```
    PRINTER
```

```
      rename
```

```
        switch_on as start
```

```
      undefine
```

```
        start
```

```
    end
```

```
  SCANNER
```

```
    rename
```

```
      switch_on as start,  
send as send_data
```

```
    end
```

```
  FAX
```

```
    rename
```

```
      send as send_message
```

```
    undefine
```

```
      start
```

```
  end
```

```
feature ... end
```

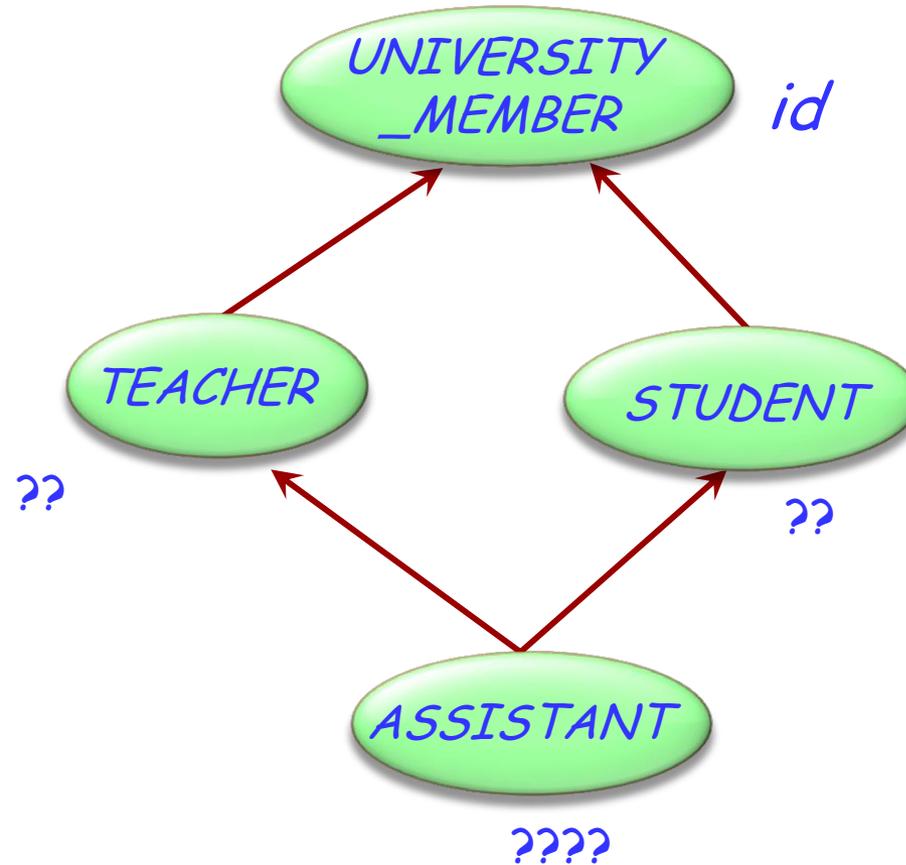
s: SCANNER

f: FAX

a: ALL\_IN\_ONE\_DEVICE

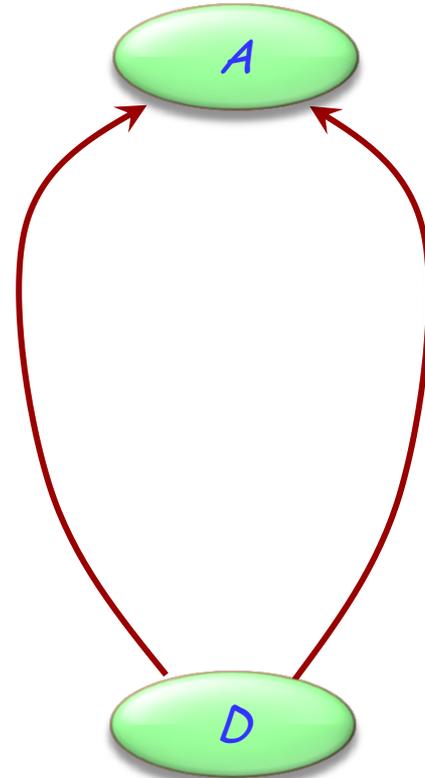
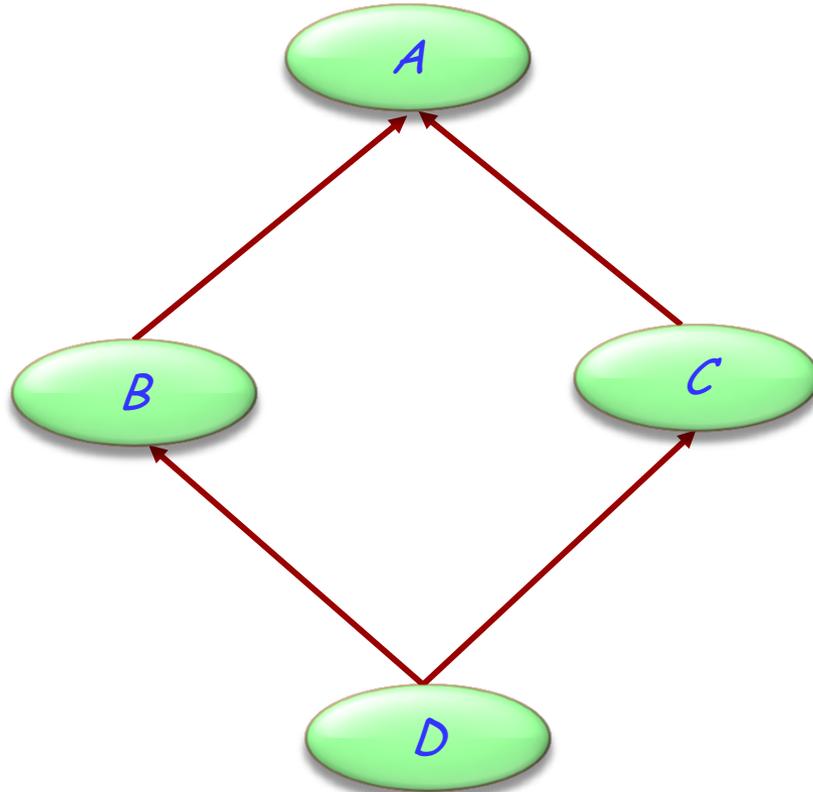
- a.switch\_on 
- a.print\_page 
- f.send\_message 
- s.switch\_on 
- f.send 
- a.send 

# A special case of multiple inheritance



This is a case of **repeated** inheritance

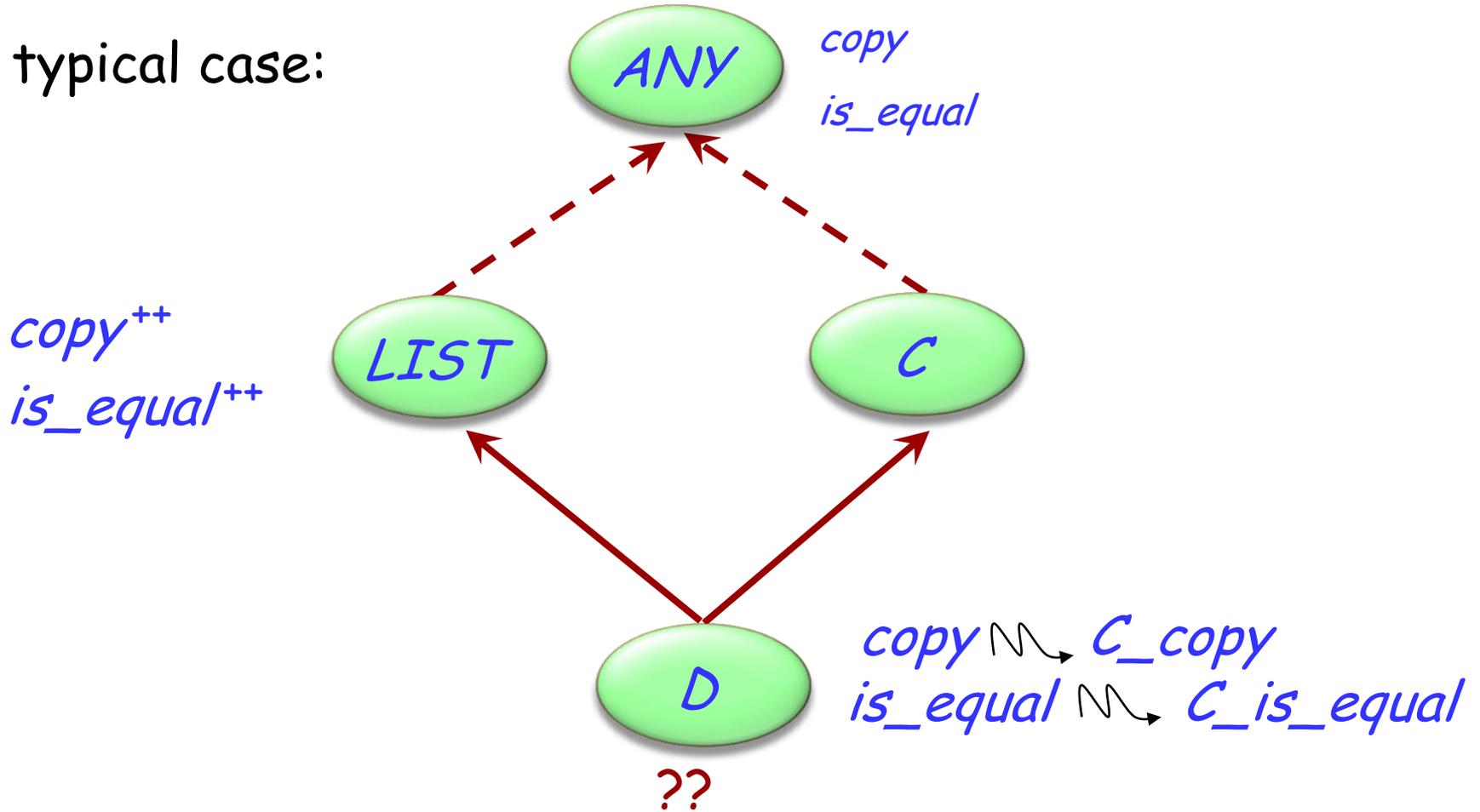
# Indirect and direct repeated inheritance



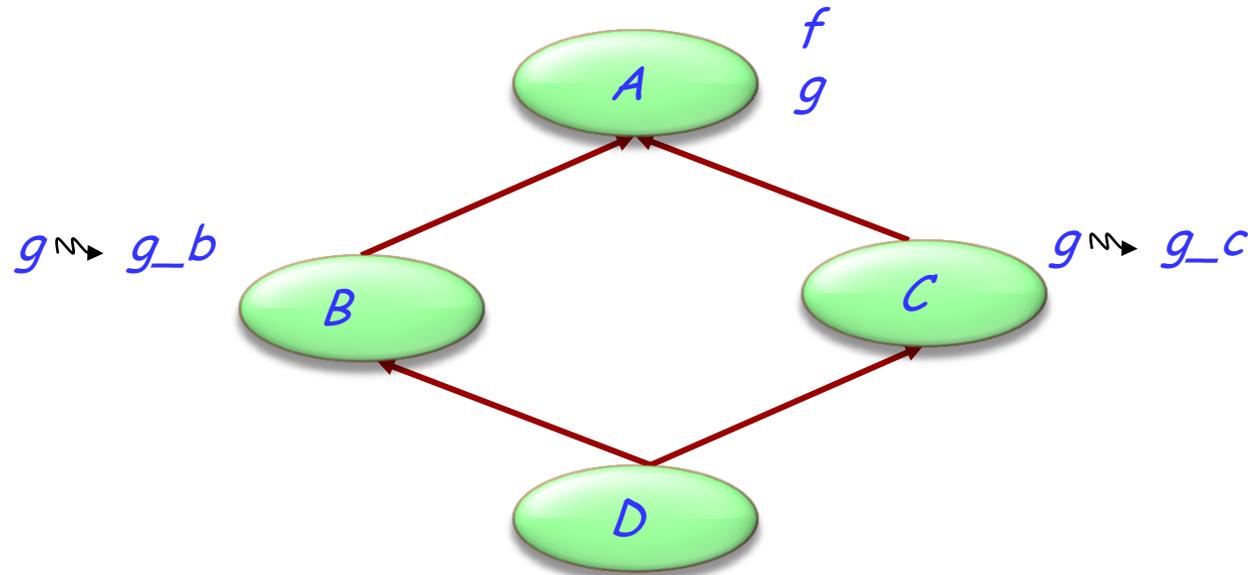
# Multiple is also repeated inheritance



A typical case:



# Sharing and replication



Features such as  $f$ , not renamed along any of the inheritance paths, will be shared.

Features such as  $g$ , inherited under different names, will be replicated.

# The need for select



A potential ambiguity arises because of polymorphism and dynamic binding:

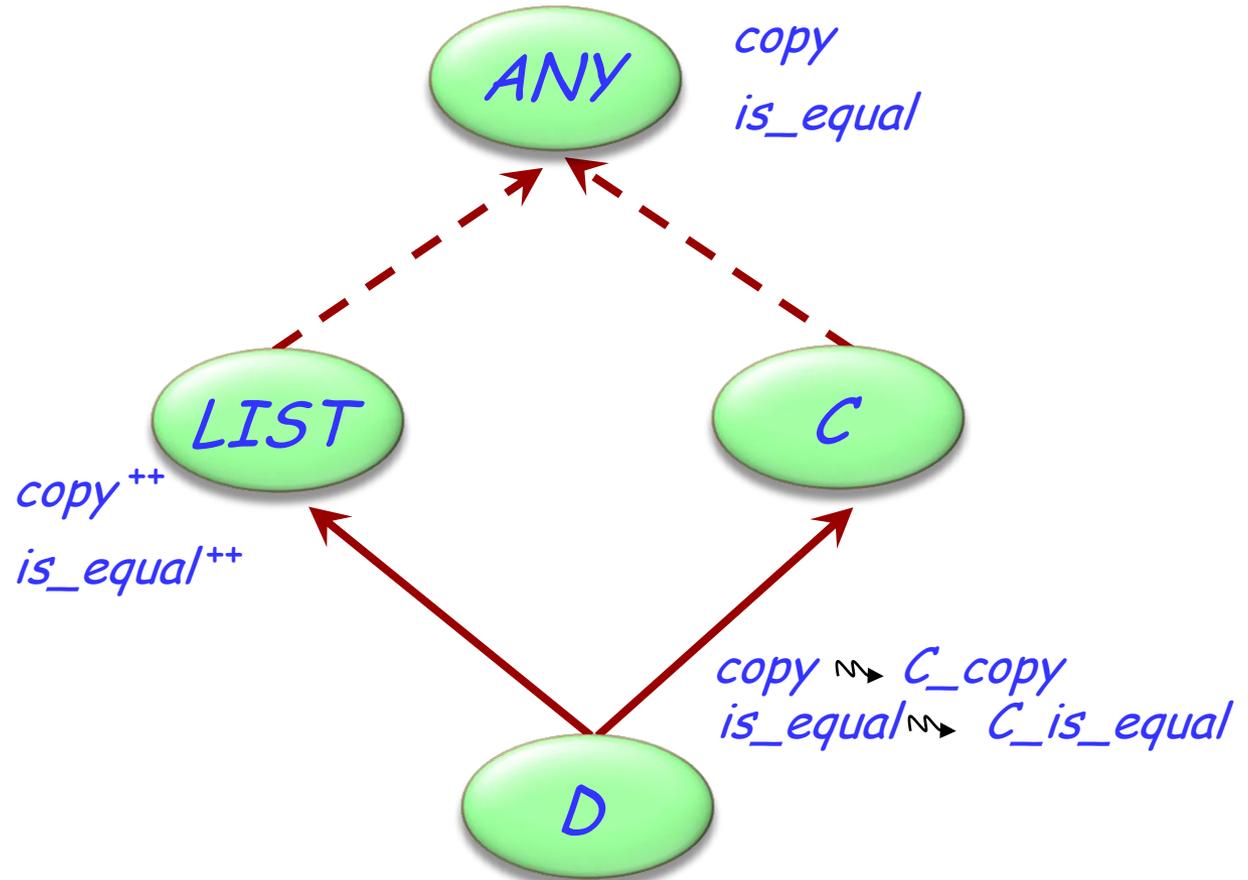
*a1: ANY*

*d1: D*

...

*a1 := d1*

*a1.copy (...)*



# Removing the ambiguity



class

*D*

inherit

*V\_LIST [ T ]*

select

*copy,*  
*is\_equal*

end

*C*

rename

*copy as C\_copy,*  
*is\_equal as C\_is\_equal,*

...

end

# When is a name clash acceptable?

---



(Between  $n$  features of a class, all with the same name, immediate or inherited.)

- They must all have compatible signatures.
- If more than one is effective, they must all come from a common ancestor feature under repeated inheritance.