Fondamenti della Programmazione: Metodi Evoluti

Prof. Enrico Nardelli

Lezione 13: Multiple inheritance

Combining abstractions

Given the classes

- TRAIN_CAR, RESTAURANT

how would you implement a DINER?

Examples of multiple inheritance

Combining separate abstractions:

- Restaurant, train car
- Calculator, watch
- Home, vehicle
- Taxi, bus

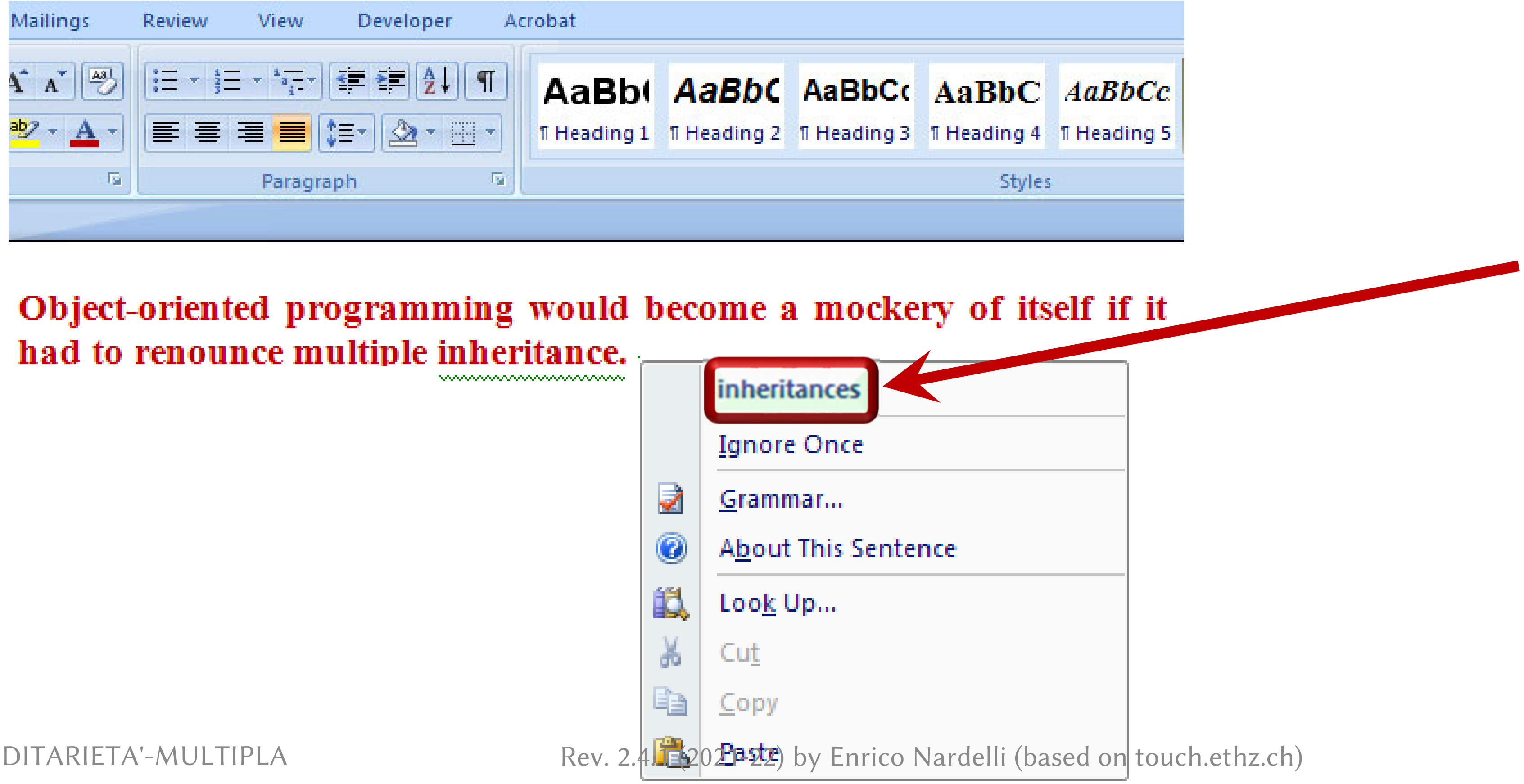
Warning

Forget all you have heard!

Multiple inheritance is **not** the works of the devil

Multiple inheritance is **not** bad for your teeth

(Even though Microsoft Word apparently did not like it:



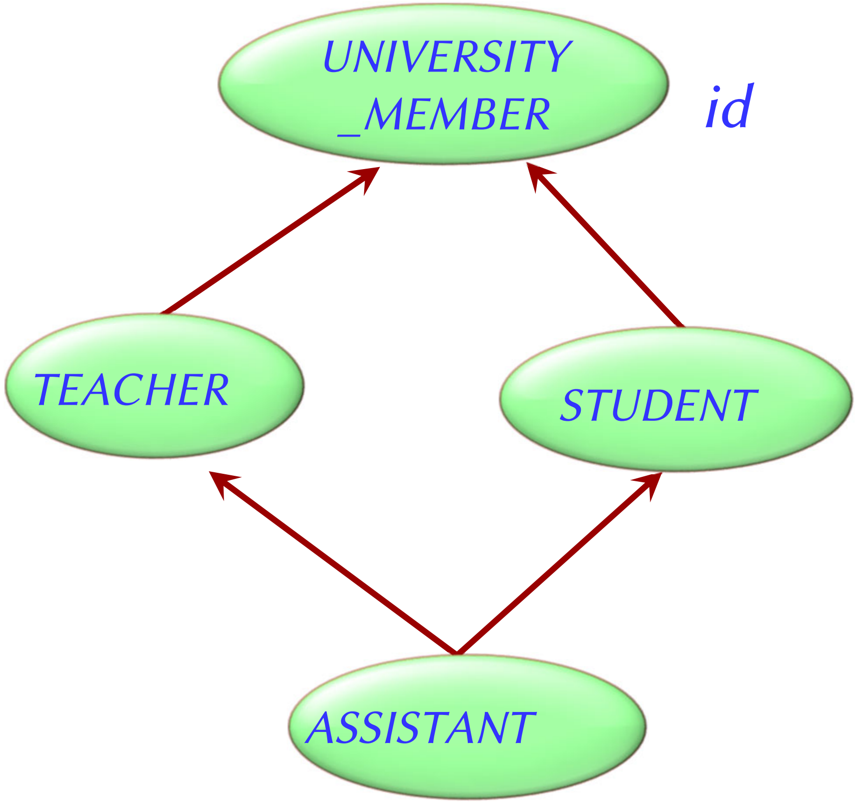
Object-oriented programming would become a mockery of itself if it had to renounce multiple inheritance.

)

An example of **repeated** inheritance

A class with two or more parents sharing a same grand-parent.

Examples that come to mind: *ASSISTANT* inherits from *TEACHER* and *STUDENT*.



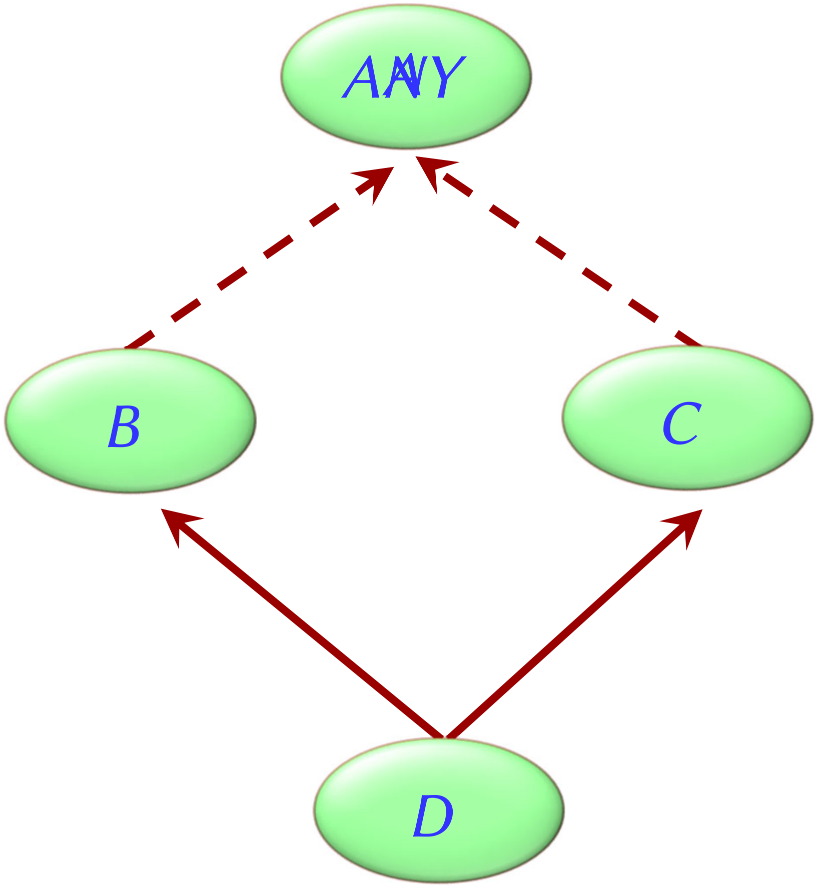
This is a case of repeated inheritance

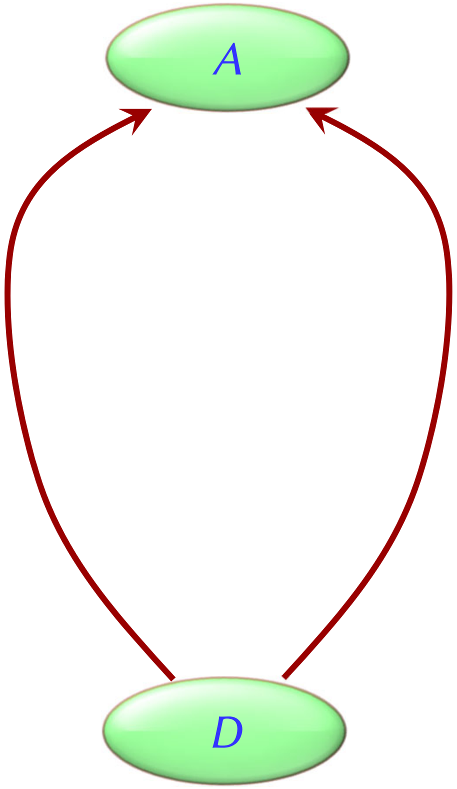
repeated and multiple inheritance

Multiple inheritance from B and C

Repeated inheritance from A

(In Eiffel is found often; why?)





This form of repeated inheritance cannot happen in Eiffel

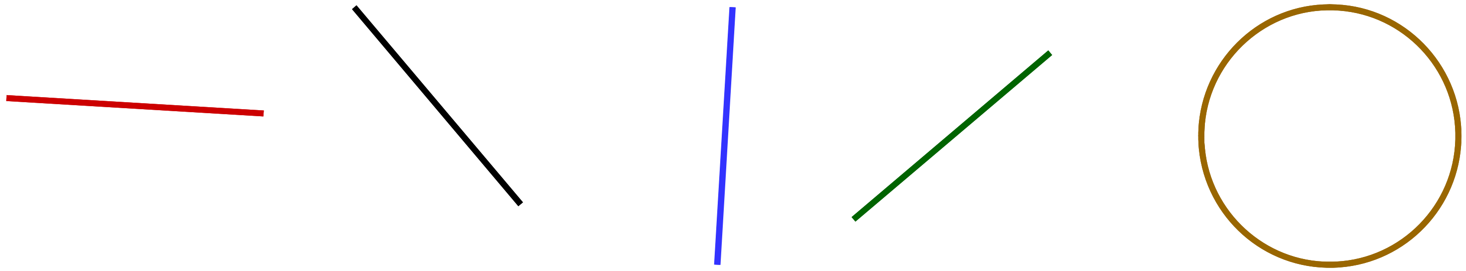
Another warning

The language part of this lecture are Eiffel-oriented

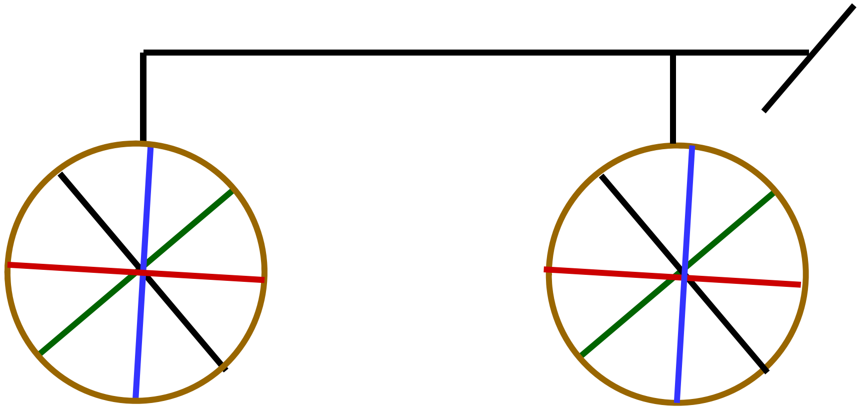
Java and C# mechanisms (single inheritance from classes, multiple inheritance from interfaces) will also be discussed

C++ also has multiple inheritance, but it will not be described

Multiple inheritance: Composite figures

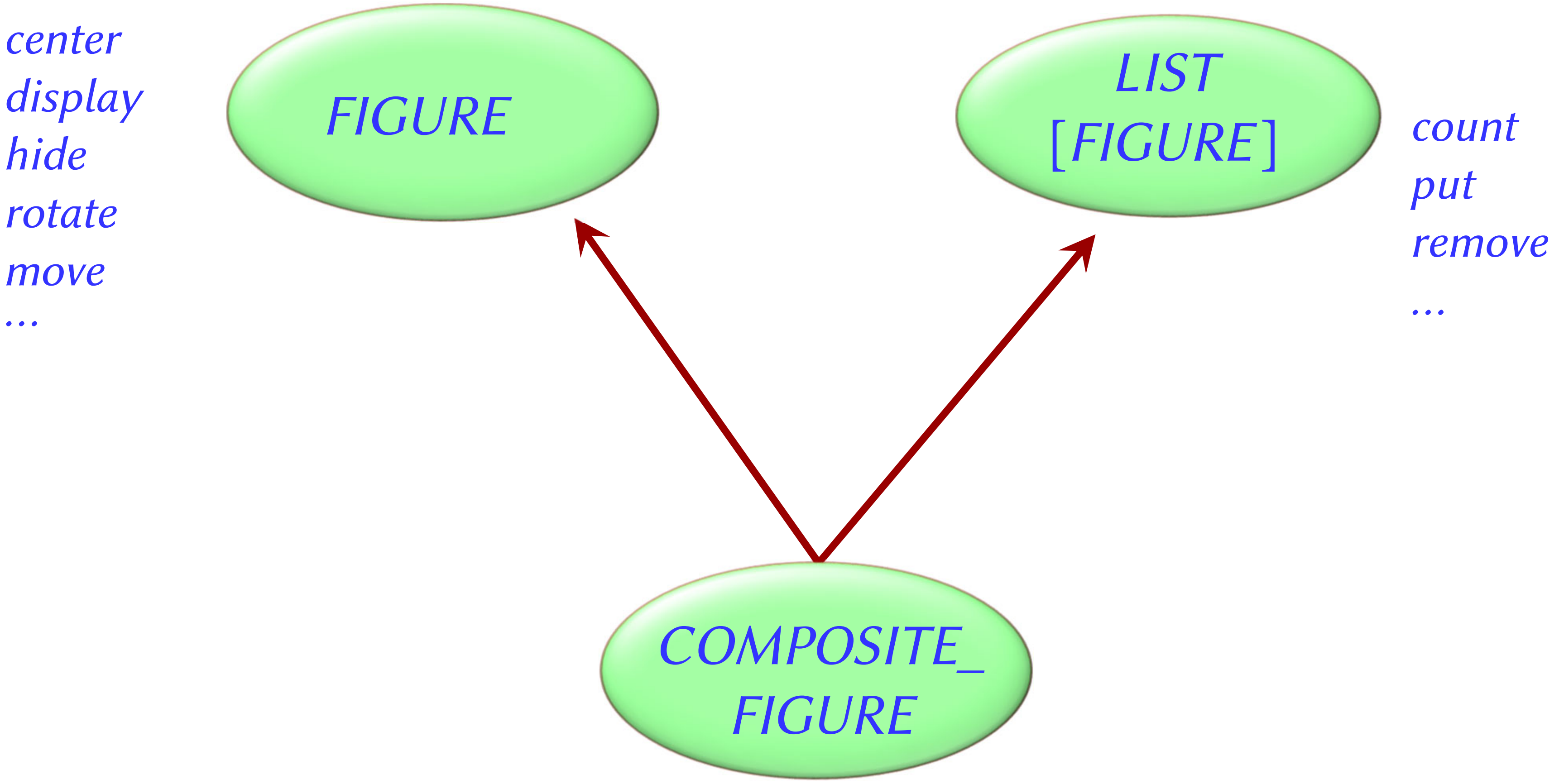


Simple figures



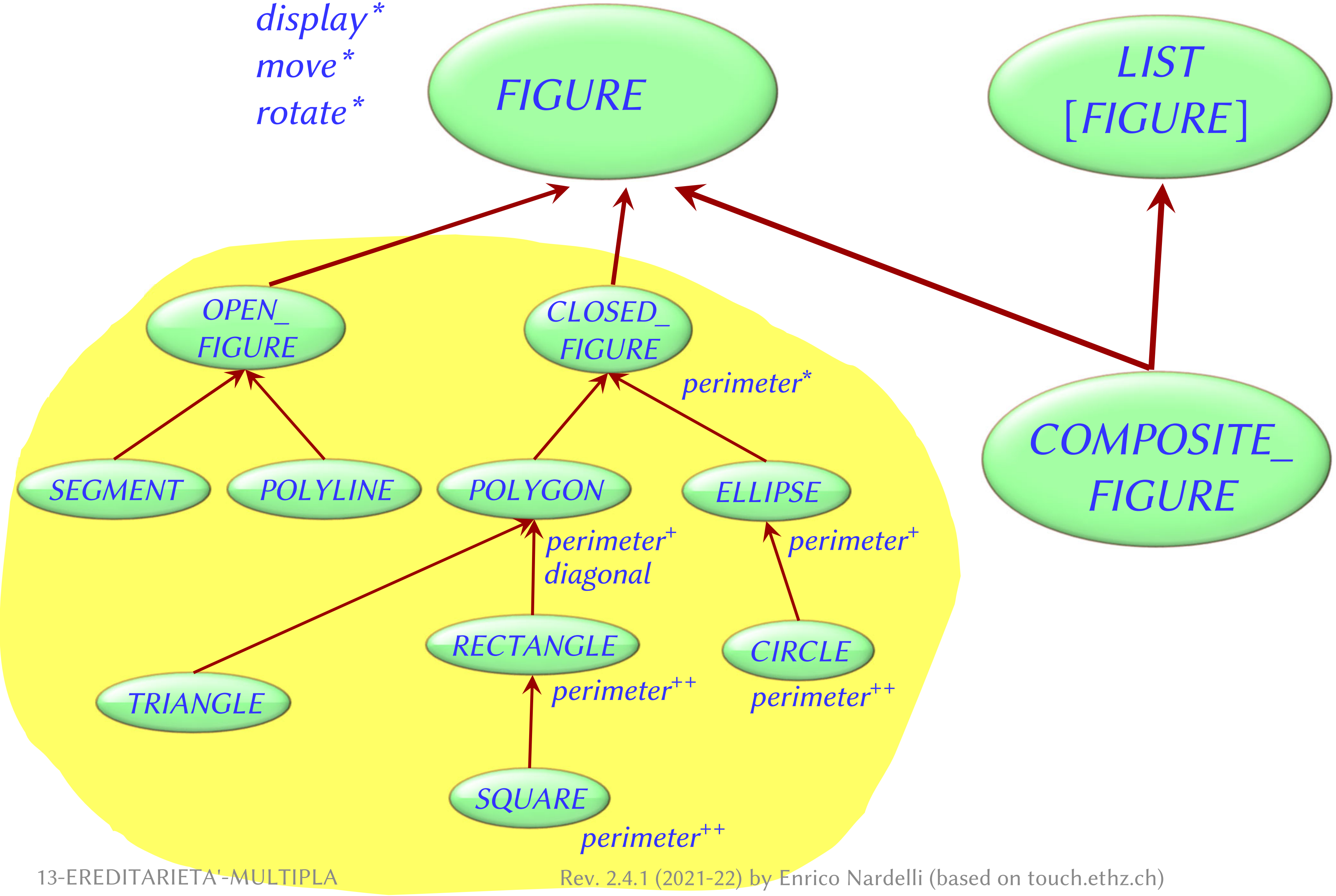
A composite figure

Defining the notion of composite figure



COMPOSITE_FIGURE inherits different features from more than one parent: this is multiple inheritance

In the overall structure

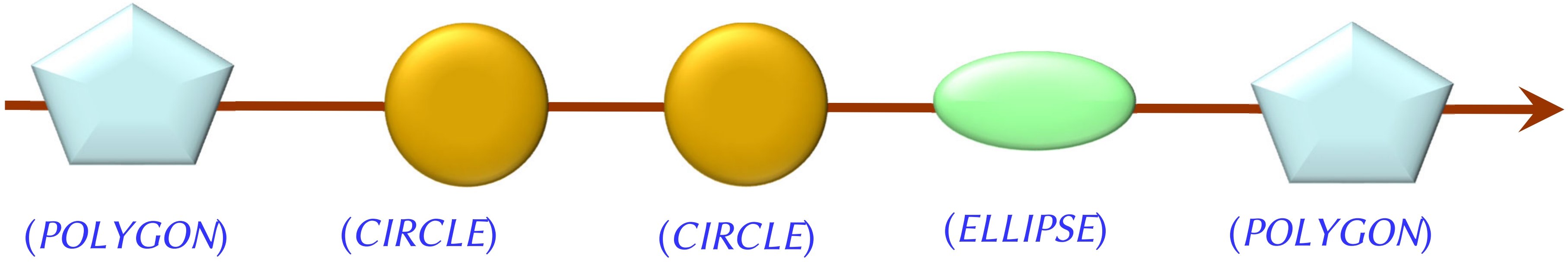


Working with polymorphic data structures

(from 10-EREDITARIETA')

```
figs: LIST [FIGURE]

…

from figs.start until figs.after loop
        figs.item.display
        figs.forth
end
```

Dynamic binding

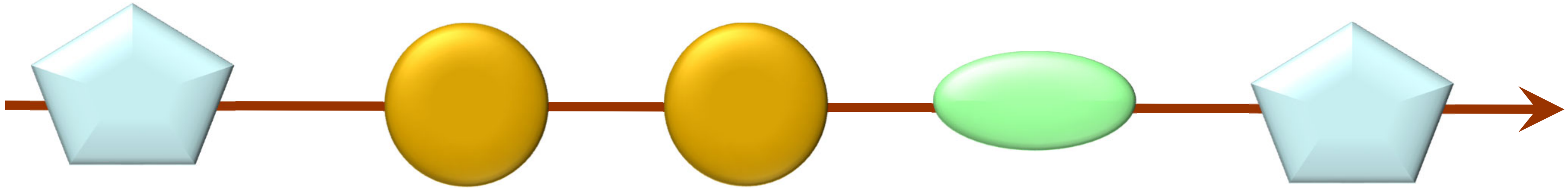


Working with polymorphic data structures

(from 10-EREDITARIETA')

```
figs: LIST [FIGURE]

...

across figs as c loop

        c.item.display

end
```

Dynamic binding



Definition (Polymorphism, adapted)

(from 10-EREDITARIETA')

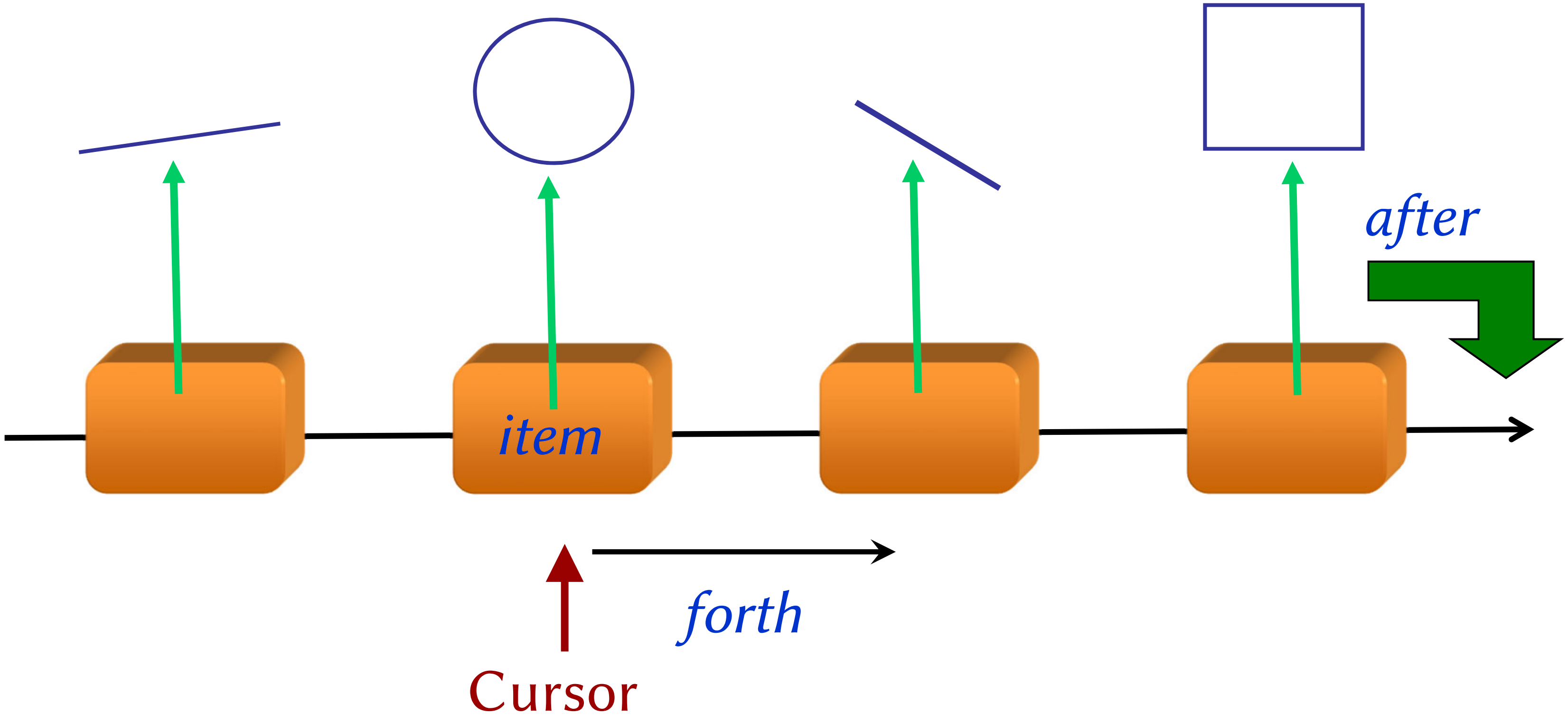
An **attachment** (assignment or argument passing) is **polymorphic** if its target entity and source expression have different types.

An **entity** or **expression** is **polymorphic** if – as a result of polymorphic attachments – it may at runtime become attached to objects of different types.

A **container data structure** is **polymorphic** if it may contain references to objects of different types.

Polymorphism is the existence of these possibilities.

A composite figure as a list

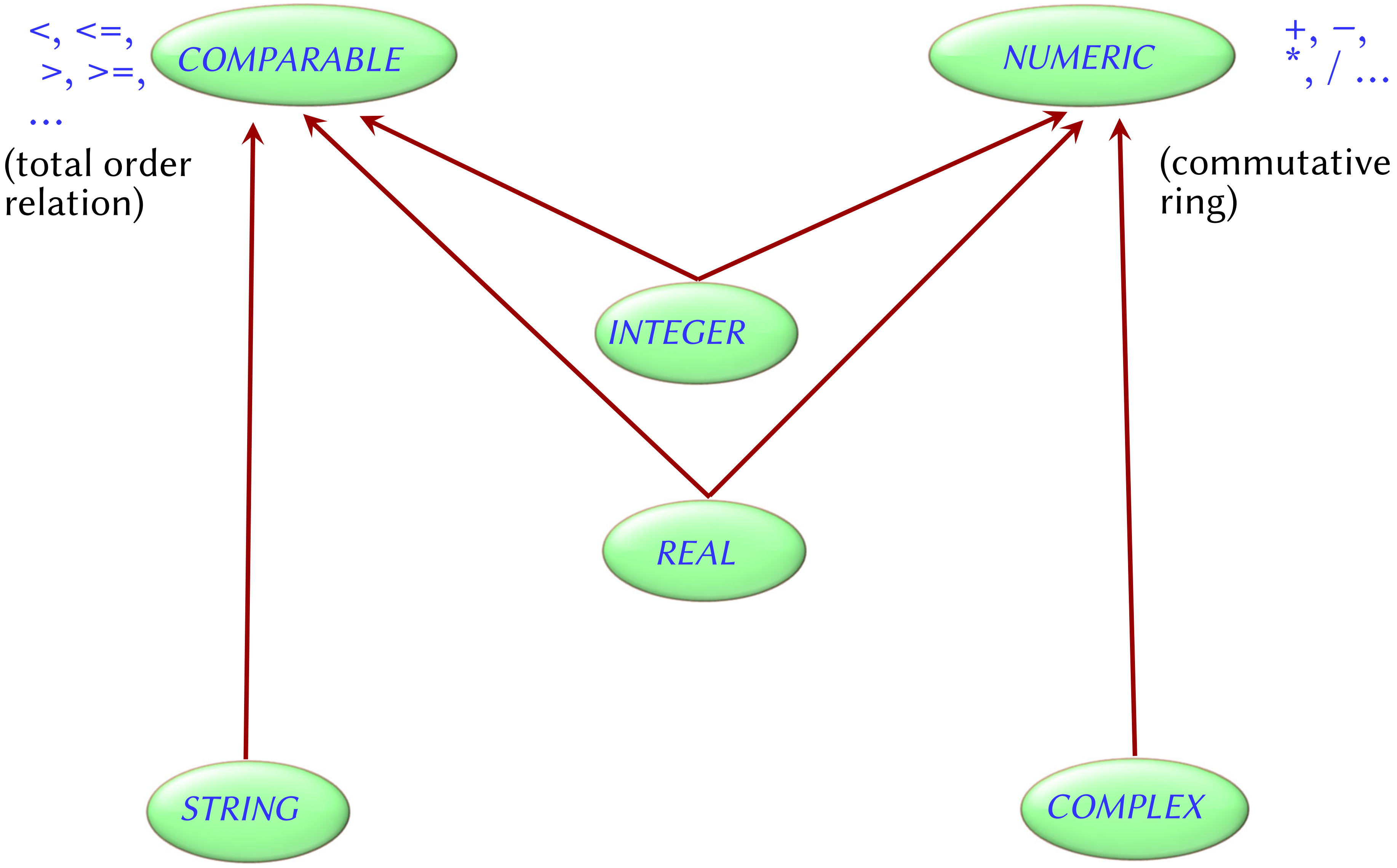


Composite figures

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

Requires dynamic binding

Multiple inheritance: Combining abstractions



How do we write *COMPARABLE*?

```
deferred class COMPARABLE [G] feature

        less alias "<" (x: COMPARABLE [G]): BOOLEAN
            deferred
            end

        less_equal alias "<=" (x: COMPARABLE [G]): BOOLEAN
            do
                Result := (Current < x or (Current = x))
            end

        greater alias ">" (x: COMPARABLE [G]): BOOLEAN
            do Result := (x  < Current) end

        greater_equal alias ">=" (x : COMPARABLE [G ]): BOOLEAN
            do Result := (x  <= Current) end

end
```

Java and .NET and C# solution

Single inheritance only for classes

Multiple inheritance from **interfaces**

An interface is like a fully deferred class, with no implementations (**do** clauses), no attributes (and also no contracts): it's only specification

A class may inherit from:

- At most one class
- Any number of interfaces

Deferred classes vs Java interfaces (1)

- Java interfaces are "entirely deferred"
  - Only method (routine) definitions
  - No method implementations
  - No attributes
  - No contracts

- Eiffel deferred classes can include effective features, possibly relying on deferred ones, as in the *COMPARABLE* example
  - Flexible mechanism to implement abstractions progressively

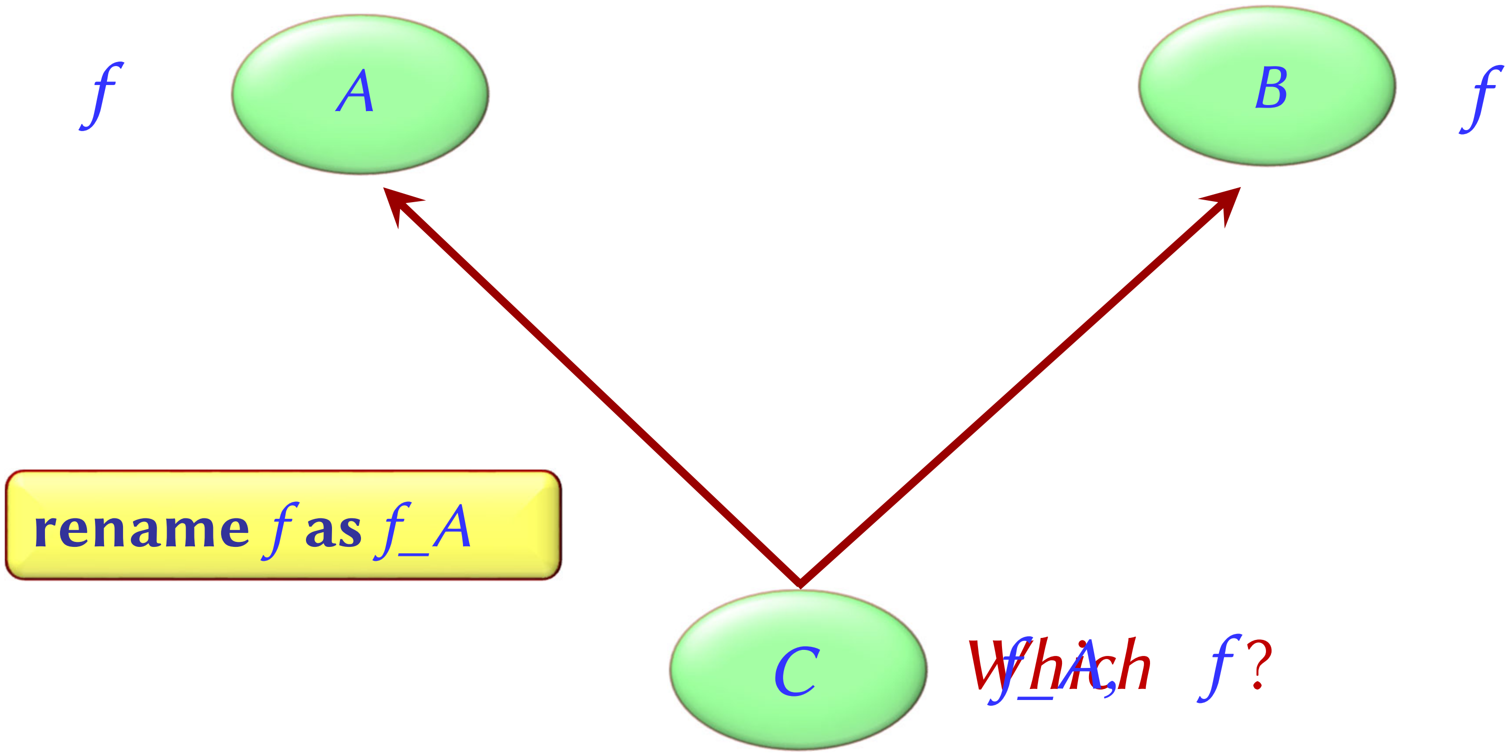
Deferred classes vs Java interfaces (2)

Java requires that every descendant of an interface must provide implementations of *all* interface’s features.

To be able to flexibly model reality we need the full spectrum from fully abstract (i.e., fully deferred) to fully implemented classes provided by Eiffel

Multiple inheritance is here to help us combine abstractions

Resolving name clashes



```
class C inherit
        A rename f as f_A end
        B
...
```

Consequences of renaming (1)

In class *C*

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

…

OK *c1.f* Version from *B*

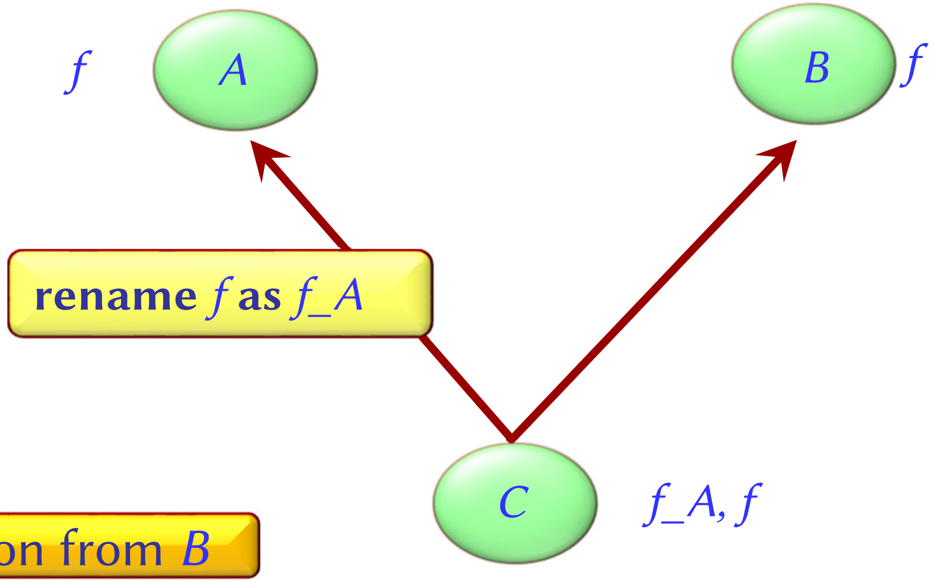
OK *c1.f_A* Version from *A*

OK *a1.f* Version from *A*

Invalid! *a1.f_A*

OK *b1.f* Version from *B*

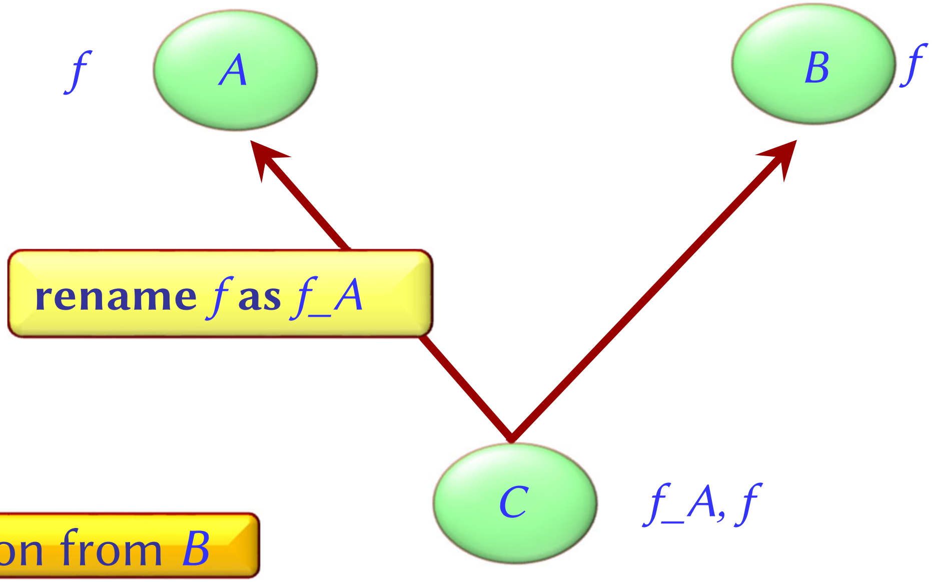
Invalid! *b1.f_A*



Consequences of renaming (2)

In class *C*

a1: *A*
b1: *B*
c1: *C*
…
a1*:= *c1



OK *c1.f* — Version from *B*

OK *c1.f_A* — Version from *A*

OK *a1.f* — Version from *A*, not from B !

Invalid! *a1.f_A*

OK *b1.f* — Version from *B*

Invalid! *b1.f_A*

Instances of ***C*** do not inherit name ***f*** from ***A***

Renaming and redefinition

Renaming keeps the feature behavior and changes its name

Redefinition changes the feature behavior and keeps its name

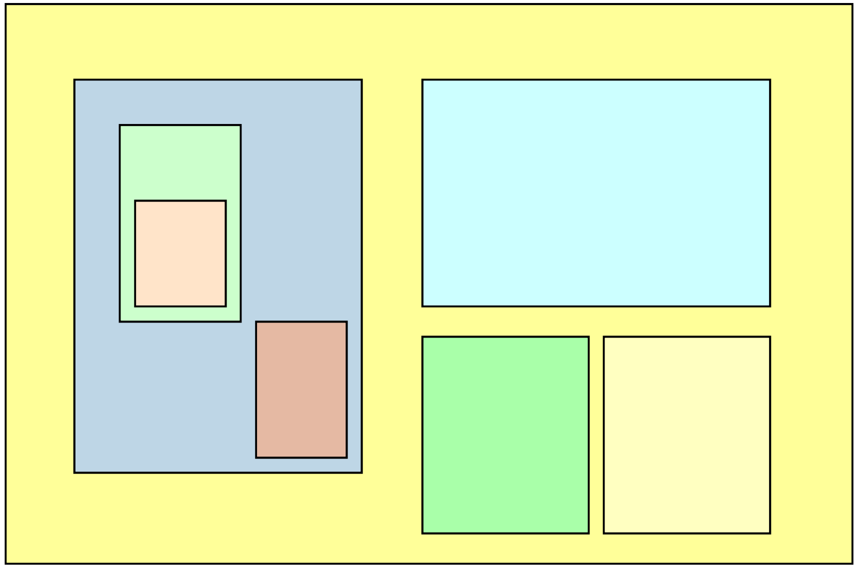
It is possible to combine both:

```
class B
    inherit
        A
            rename f as f_A
            redefine f_A
            end
        …
```

An application of renaming

Provide locally better adapted terminology.

Example: *child* (*TREE*); *subwindow* (*WINDOW*)

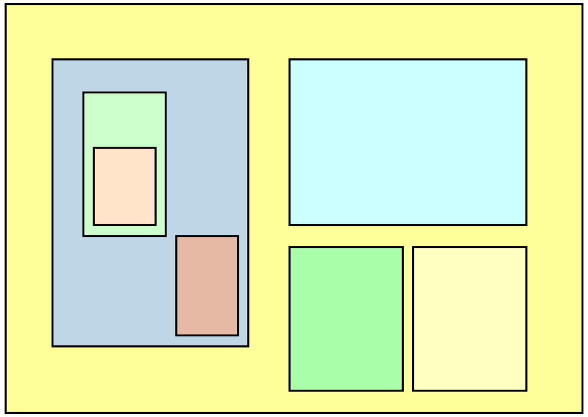


Renaming to improve feature terminology

"Graphical" features: *height, width, change_height, change_width, xpos, ypos, move...*

"Hierarchical" features: *superwindow, subwindows, change_subwindow, add_subwindow...*

```
class WINDOW  inherit
        RECTANGLE
        TREE [WINDOW]
            rename
                    parent as superwindow,
                    children as subwindows,
                    add_child as add_subwindow
                    …
            end
feature
        …
end
```



BUT: see style rules about uniformity of feature names

Are all name clashes bad?

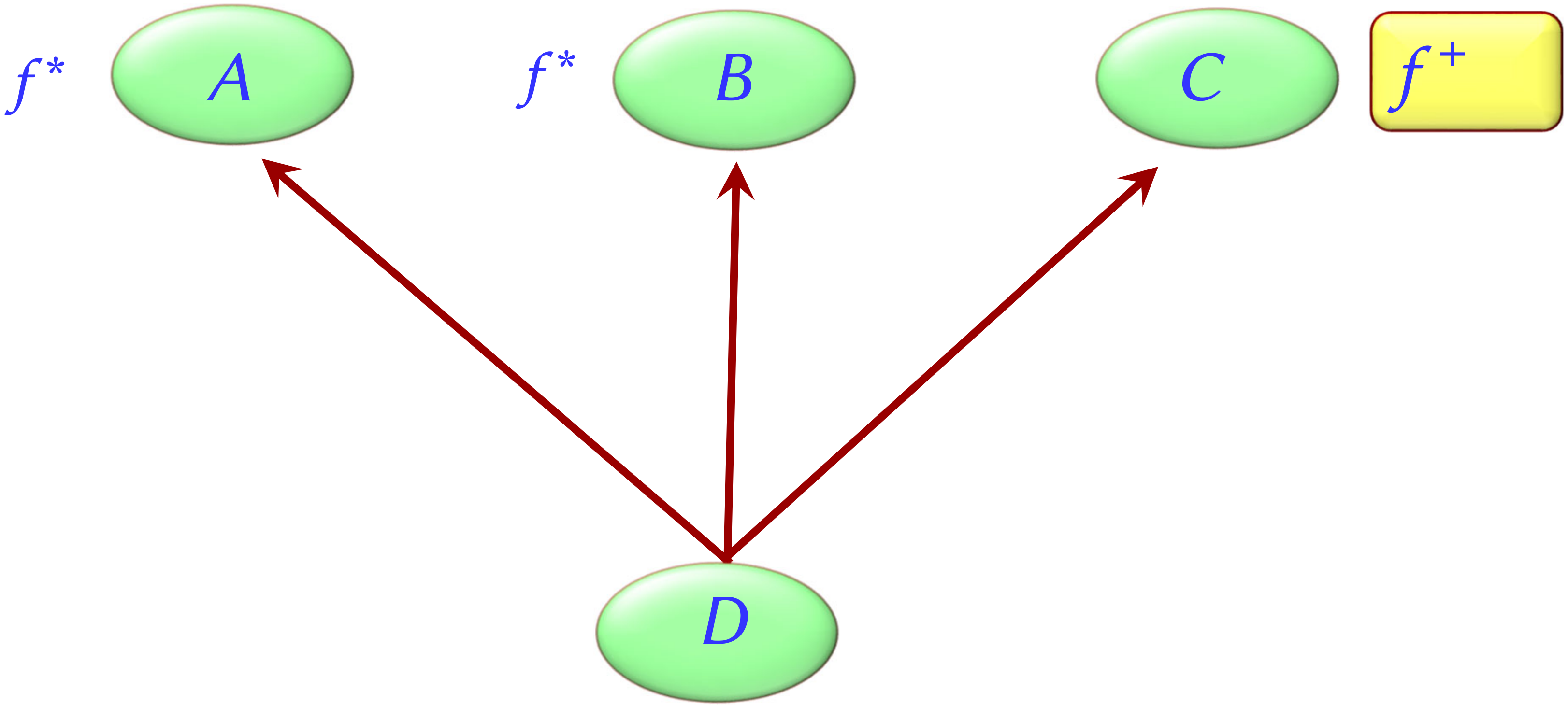
A name clash must be removed unless it is:

- Under repeated inheritance (i.e. not a real clash), OR
- All inherited features with the same name are such that
  - They all have compatible signatures
  - At most one of them is effective

Semantics of the latter case:

- All features are merged into a single one
- If there is an effective feature, its implementation is the one which is used

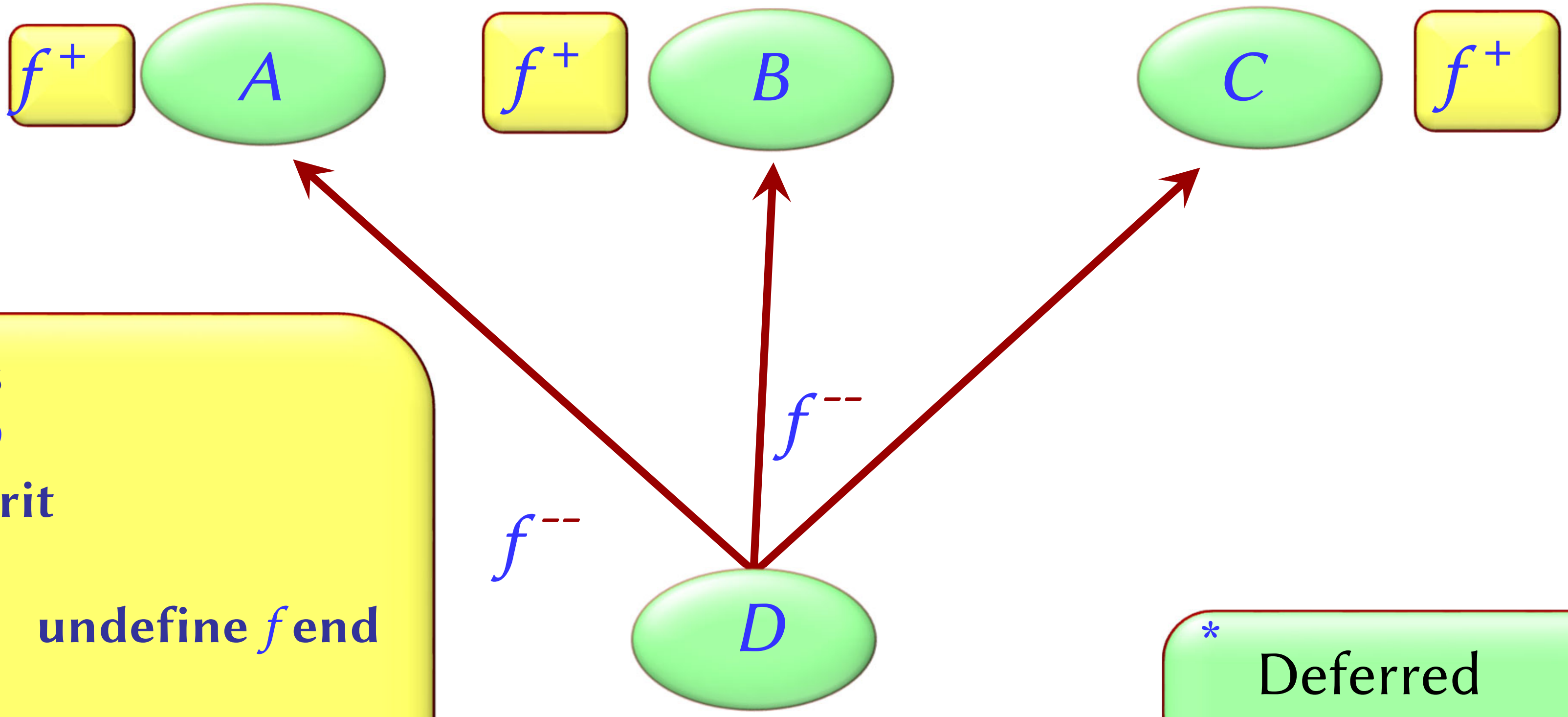
Feature merging



* Deferred

$^+$ Effective

Feature merging: case of effective features



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

feature
        ...
end
```

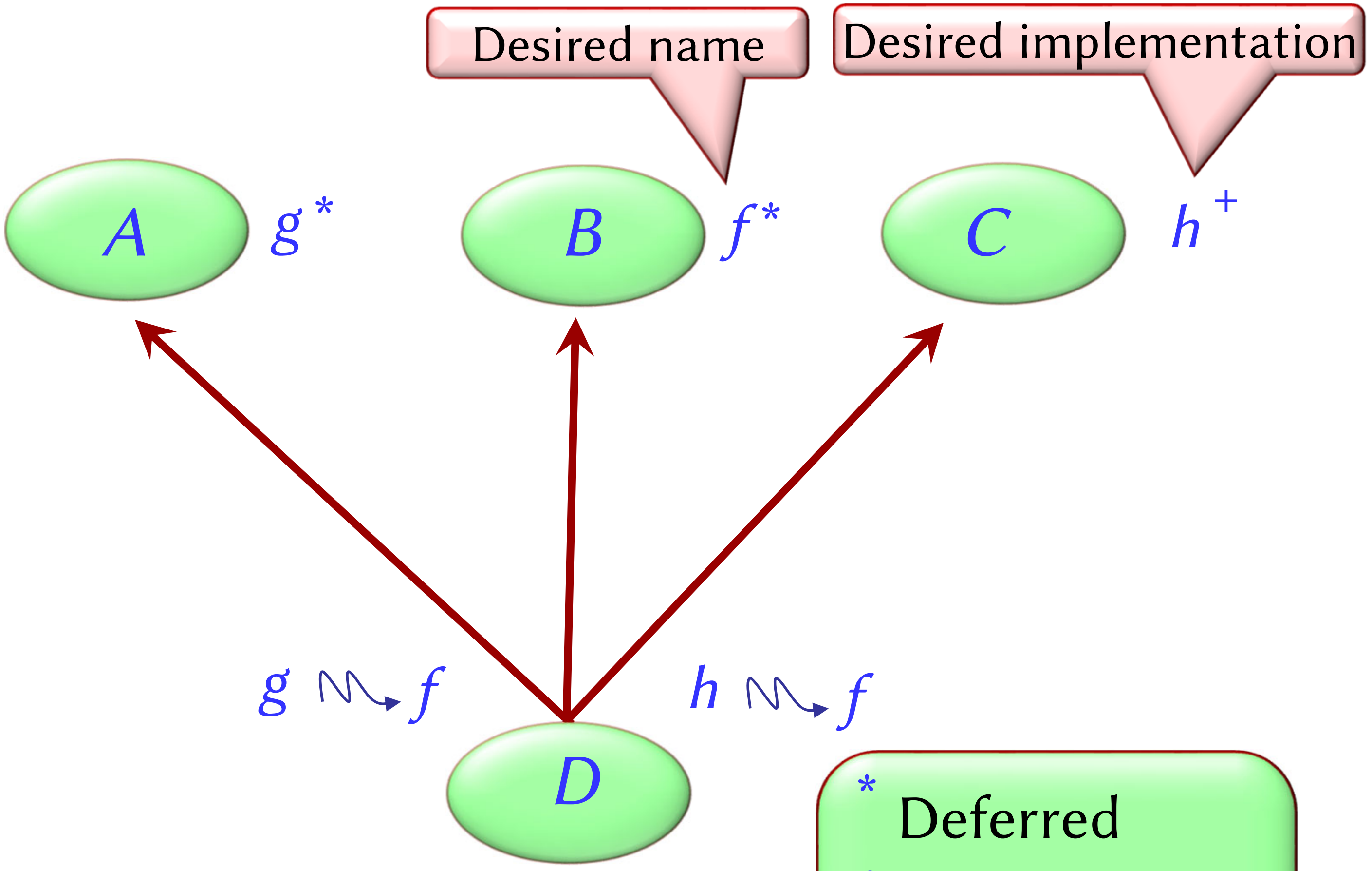
* Deferred

+ Effective

-- Undefine

Feature merging: case of different names (1)

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



Desired name

Desired implementation

* Deferred

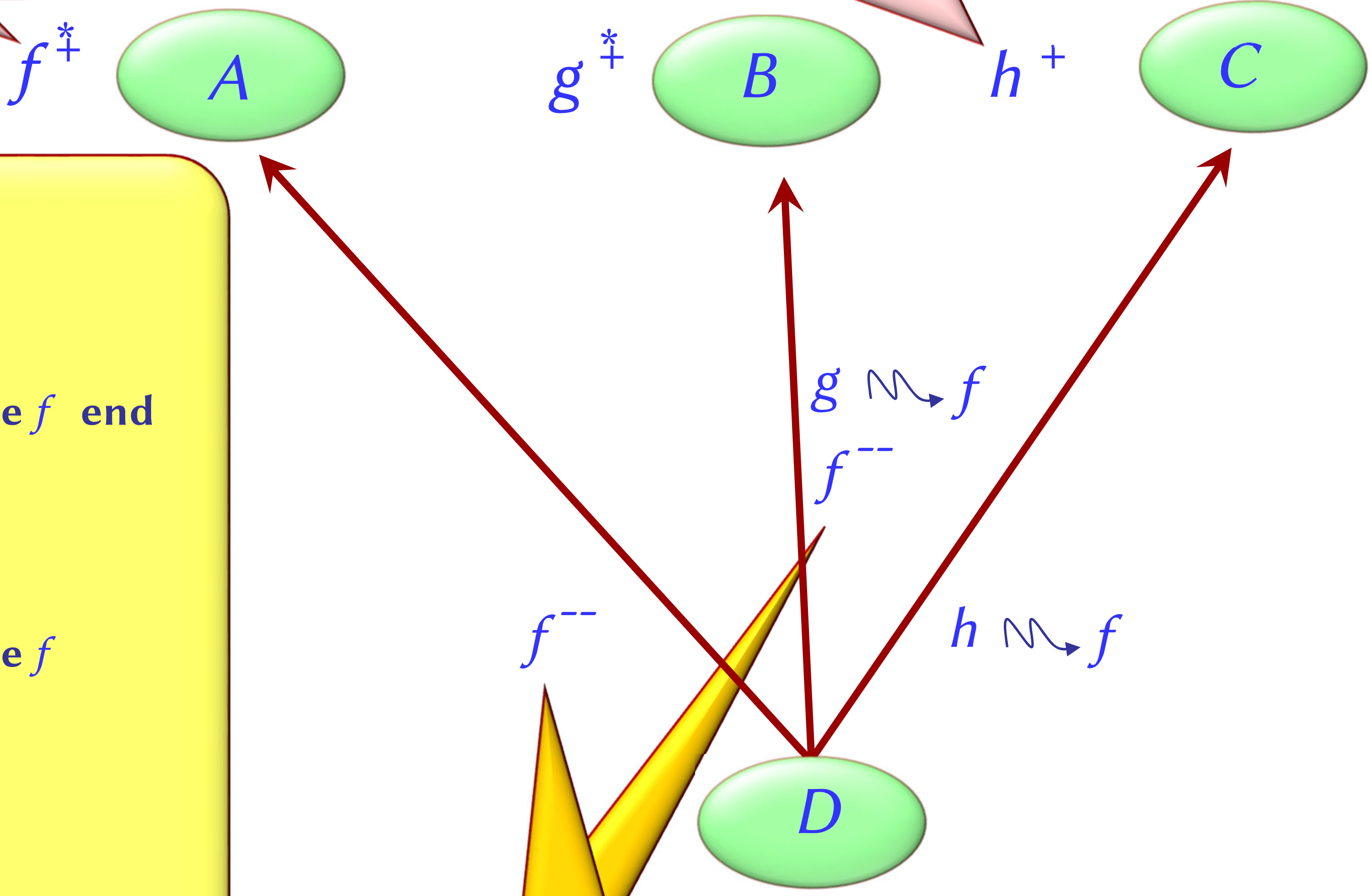
$^+$ Effective

-- Undefine

⇝ Renaming

Feature merging: case of different names (2)

```
class
    D
inherit
    A
        undefine f   end

    B
        rename
            g as f
        undefine f
        end

    C
        rename
            h as f
        end
feature    ...   end
```

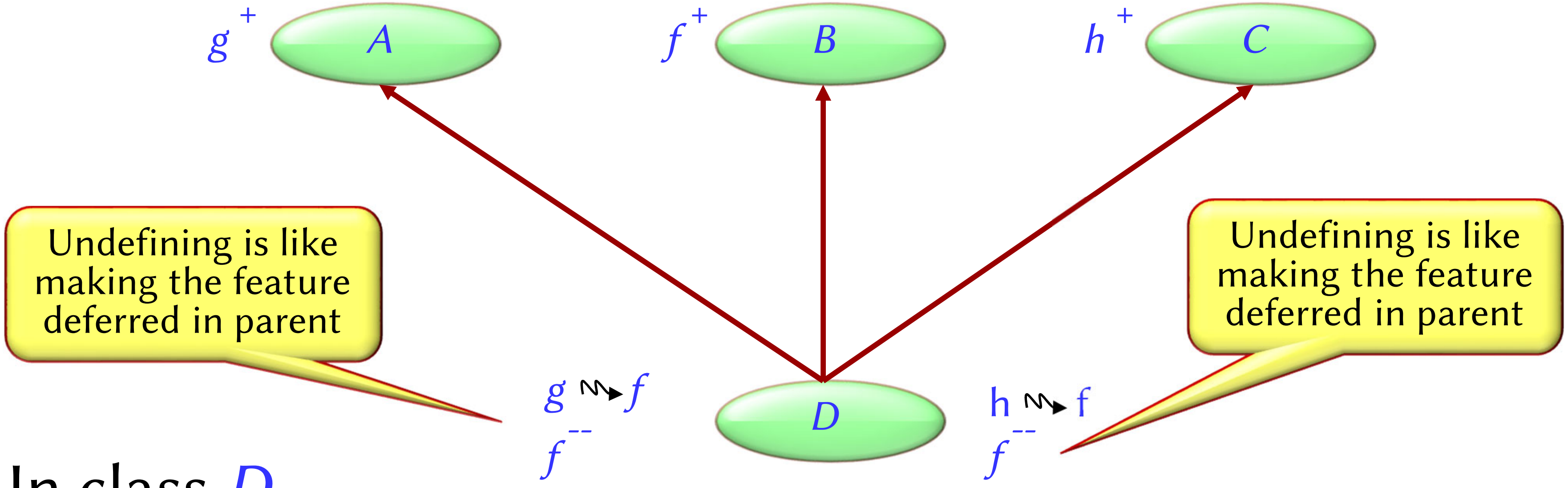


Desired name

Desired implementation

As if *f* were deferred
in the parent

Feature call after merging



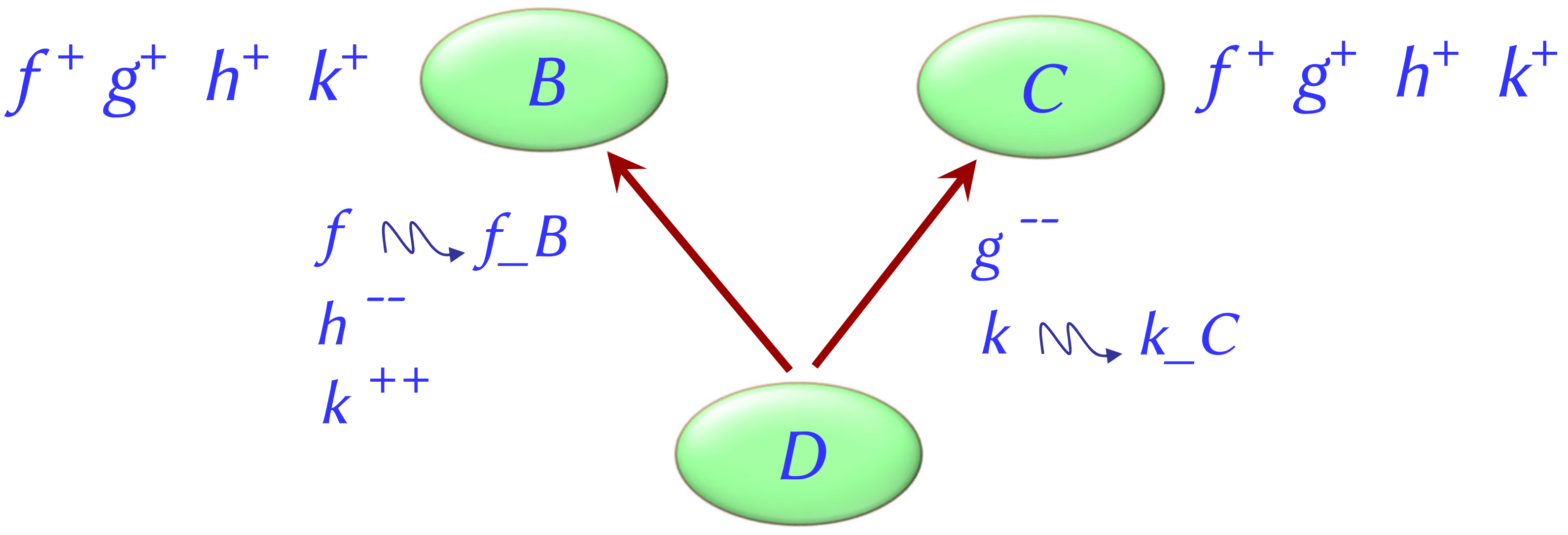
Undefining is like making the feature deferred in parent

Undefining is like making the feature deferred in parent

In class *D*

a1: A		*b1: B*		*c1: C*		*d1: D*	
a1.g	OK	*b1.f*	OK	*c1.h*	OK	*d1.f*	OK
						d1.g	Invalid!
						d1.h	Invalid!

Feature merging: case of equal names (1)



f^+ g^+ h^+ k^+ *B*

C f^+ g^+ h^+ k^+

$f \leadsto f\_B$

h^{--}

k^{++}

g^{--}

$k \leadsto k\_C$

D

f (from C) *f_B* *g* (from B) *h* (from C) *k* (from D) *k_C*

In the root class *b1: B* *d1: D*

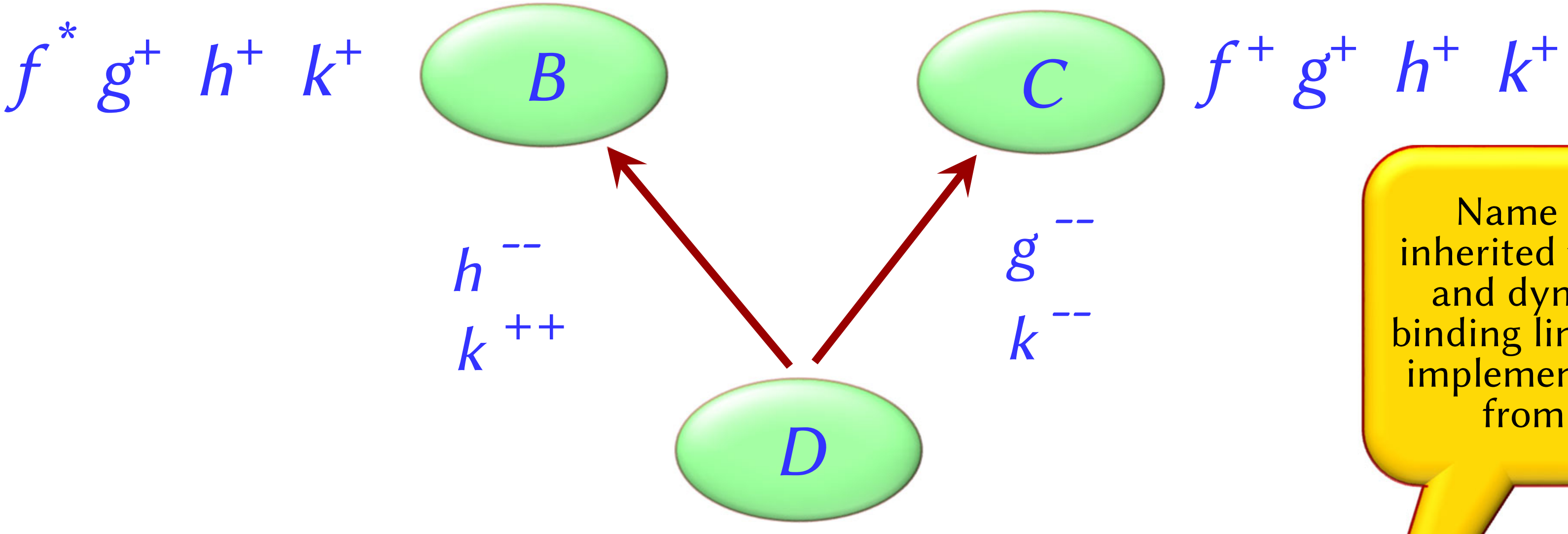
d1.f	C
d1.g	B
d1.h	C
d1.k	D

Dynamic binding cannot be applied since name ***f*** has been removed in inheritance toward ***D***

Then *b1 := d1*

b1.f	B
b1.g	B
b1.h	C
b1.k	D

Feature merging: case of equal names (2)


f^{*} g^{+} h^{+} k^{+}
B
C
f^{+} g^{+} h^{+} k^{+}
h^{--}
k^{++}
g^{--}
k^{--}
D

f (from C) *g* (from B) *h* (from C) *k* (from D)

Name ***f*** is inherited from ***B*** and dynamic binding links it to implementation from ***C***

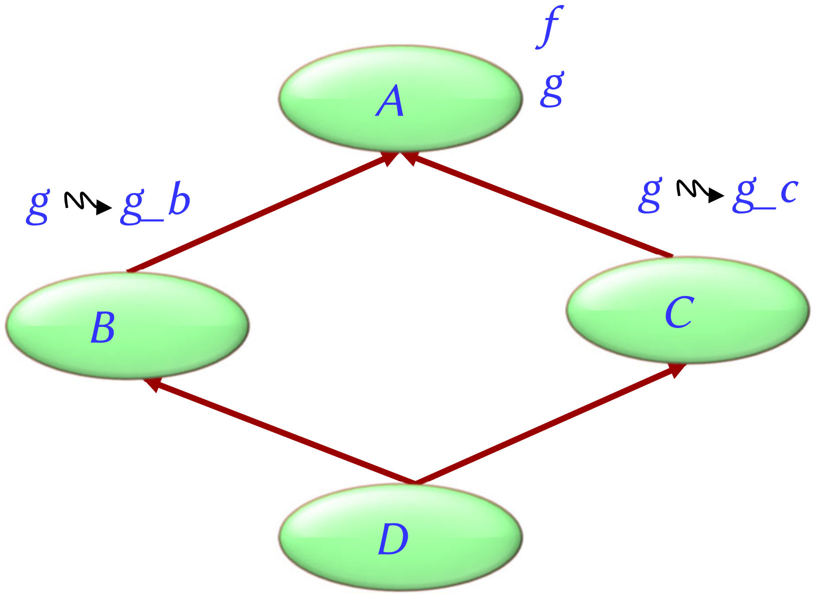
In the root class *b1: B* *d1: D*

b1.f	Invalid!	*d1.f*	C
b1.g	B	*d1.g*	B
b1.h	B	*d1.h*	C
b1.k	B	*d1.k*	D

Then *b1 := d1*

b1.f	C
b1.g	B
b1.h	C
b1.k	D

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**: there are two names to execute the same action

The need for select

A potential ambiguity arises because of polymorphism and dynamic binding:

a1 : ANY; t1 : LIST; d1 : D

…

a1.copy (…) ***ANY*** version

t1.copy (…) ***LIST*** version

d1.copy (…) ***LIST*** version

a1 := t1

a1.copy (…) ***LIST*** version

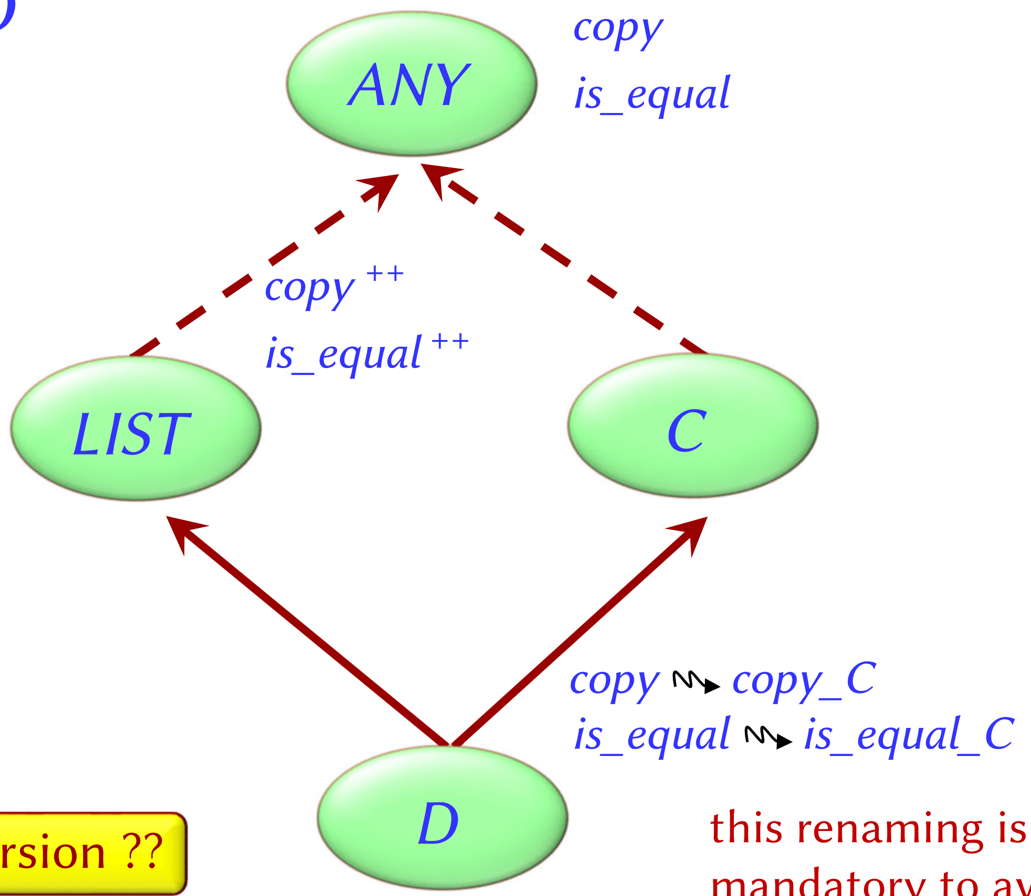
t1 := d1

t1.copy (…) ***LIST*** version

a1 := d1

a1.copy (…) ***LIST*** or ***ANY*** version ??

The run-time cannot decide !



this renaming is mandatory to avoid name clash

When the need arises?

- This happens whenever, through the combination of renaming (and possibly redefinition) in different inheritance paths, in a class ***X*** there is more than one version of an inherited feature ***f*** (**repeatedly inherited feature**)

- These versions will have different names (due to renaming) and might have different behaviours (due to redefinition)

- If a variable of the ancestor class which has provided the original version of the feature get assigned a variable of class ***X*** neither the compiler nor the runtime can decide which version of feature ***f*** should be used

Removing the ambiguity

```
class
        D
inherit
        LIST [T]
                select
                        copy,
                        is_equal
                end

        C
                rename
                        copy as copy_C,
                        is_equal as is_equal_C,
                                        ...
                end
```

The version from ***LIST*** is used under dynamic binding in the case of a polymorphic target with a possible ambiguity

Order for redeclaration clauses (standard specif.)

Prescribed in ECMA, not yet implemented!

(checked May 2021)

```
class
    AN_HEIR

inherit
    A_PARENT
        undefine
            feature_A, feature_B, …

        redefine
            feature_C, feature_D, …

        rename
            feature_C, feature_D, …

        export
            {class_X, class_Y, …} feature_A, feature_B, …
            {class_W, class_Z, …} feature_C, feature_D, …

        select
            feature_C, feature_D, …
        end

end
```

make deferred

change implementation

give a new name

change the visibility status

selection for dynamic binding

Order for redeclaration clauses (actual)

The one actually implemented in Eiffel

(checked May 2021)

```
class
    AN_HEIR

inherit
    A_PARENT
        rename
            feature_C, feature_D, ...

        export
            {class_X, class_Y, ...} feature_A, feature_B, ...
            {class_W, class_Z, ...} feature_C, feature_D, ...

        undefine
            feature_A, feature_B, ...

        redefine
            feature_C, feature_D, ...

        select
            feature_C, feature_D, ...
        end

end
```

rename: give a new name

export: change the visibility status

undefine: make deferred

redefine: change implementation

select: selection for dynamic binding

What we have seen

A number of games one can play with inheritance:

- Multiple inheritance
- Feature merging
- Repeated inheritance

ATTENZIONE

BISOGNA STUDIARE E PRESENTARE LA CAT-CALL

IL TUTORIAL NON DICE MOLTO

http://docs.eiffel.com/book/method/et-inheritance

VEDERE DISCUSSIONE NEL LIBRO DI MEYER OBJECT-ORIENTED SOFTWARE CONSTRUCTION 2ED DAL PARAGRAFO 17.3 (p.) IN AVANTI, IN PARTICOLARE 17.5 E 17.9

IL PROBLEMA È CHE QUANDO NELL'AMBIENTE SI TESTA PER CATCALL MARCA TUTTO COME CATCALL. HO FATTO (mag-21) UN NUOVO PROGETTO catcall-nuovo PER VEDERE LA SITUAZIONE ED HO RISTUDIATO UN PO'

CATcalls = Changed Availability or Type calls

Changed **A**vailability or **T**ype **calls**

Flexibility of inheritance might cause problems sometimes, when features are changed in descendants

- Changed Availability: a descendant has changed the export status of a feature
- Changed Type: a descendant has changed the type of an argument of a feature

... and polymorphic attachment causes a violation in the access or the type

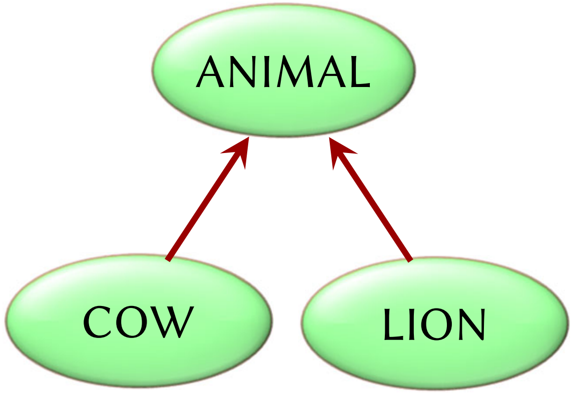
Let's see an example

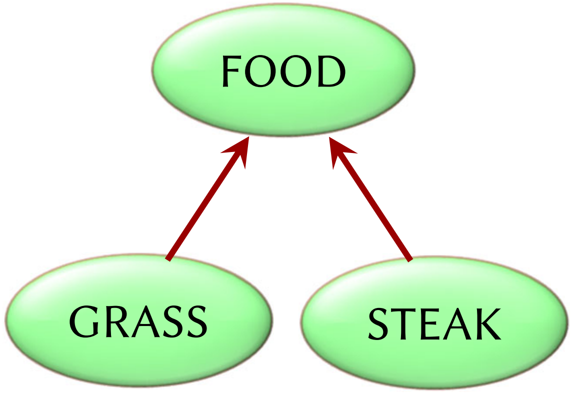
CATcall example

```
class ANIMAL
feature
    eat (a_food: FOOD)
        deferred
        end

class COW inherit ANIMAL redefine eat end
feature
    eat (a_food: GRASS)
        ...
        end

class LION inherit ANIMAL redefine eat end
feature
    eat (a_food: STEAK)
        ...
        end

my_animal: ANIMAL
my_food: FOOD

        ...
my_animal.eat (my_food)
```

A correct polymorphic feature call which could cause runtime problems:

if *my_animal* is a *LION* and *my_food* is a *GRASS*





Non-conforming inheritance

class

ARRAYED_LIST [*G*]

inherit

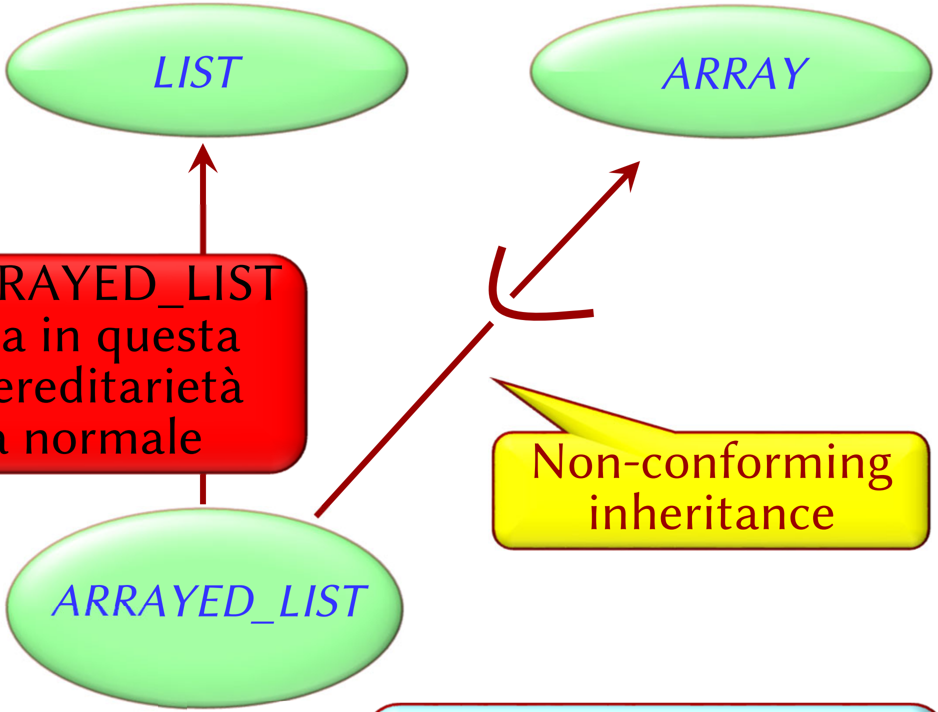
LIST [*G*]

inherit {*NONE* }

ARRAY [*G*]

In Eiffel ARRAYED_LIST non è fatta in questa modo ha ereditarietà multipla normale

Non-conforming inheritance



Non-conforming inheritance

Allow to use a different implementation for a feature, maybe more efficient

Instances of *ARRAYED_LIST* can use all *ARRAY* features but do **NOT** conform to *ARRAY*

Semantics of non-conforming inheritance

my_arrayed_list : ARRAYED_LIST [STRING]

my_list : LIST [STRING]

my_array : ARRAY [STRING]

…

my_list := my_arrayed_list OK

…

my_array := my_arrayed_list Invalid!

See EiffelStudio tutorial

http://docs.eiffel.com/book/method/et-inheritance

A common Eiffel library idiom

```
class ARRAYED_LIST [G] inherit
        LIST [G]
        ARRAY [G]
feature
        ... Implement LIST features using ARRAY features ...
end
```

For example:

```
    i_th (i: INTEGER): G
            -- Element of index `i'.
        do
            Result := item (i)
        end
```

Feature of *ARRAY*

Could use **delegation** instead

```
class ARRAYED_LIST [G] inherit
        LIST [G]
feature
        representant: ARRAY [G]
        ... Implement LIST features using ARRAY features
          applied to representant...

end
```

For example:

```
    i_th (i: INTEGER): G
            -- Element of index `i'.
        do
            Result := representant.item (i)
        end
```

Composite figures