Fondamenti della Programmazione: Metodi Evoluti

Prof. Enrico Nardelli

Lezione 10: Ereditarietà

On the menu for today (& next time)

Two fundamental mechanisms for expressiveness and reliability:

- Inheritance (subclassing)
- Genericity (type parameterization)

with associated (just as important!) notions:

- Static typing
- Polymorphism
- Dynamic binding

Reminder: the dual nature of classes

A class is a module

A class is a type*

*Or a type template (see, later, generic classes)

As a module, a class:

- Groups a set of related services
- Enforces information hiding (not all services are visible from the outside)
- Has clients (the modules that use it) and suppliers (the modules it uses)

As a type, a class:

- Denotes possible run-time values (objects & references), the instances of the type
- Can be used for declarations of entities (representing such values)

Reminder: how the two views match

The class, viewed as a *module*, groups a set of services

(the features of the class)

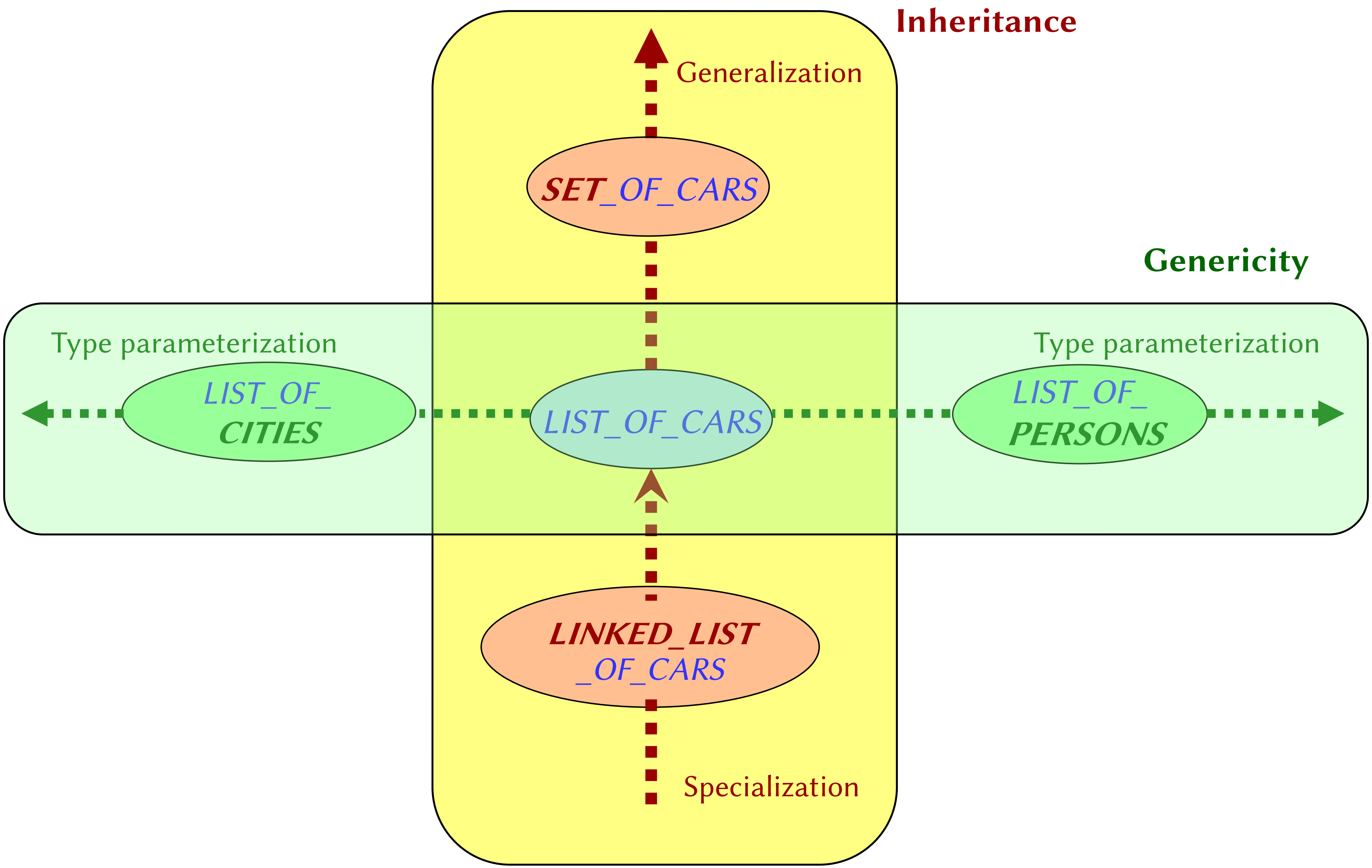
which are precisely the operations applicable to instances of the class, viewed as a *type*.

Example:

class *BUS*,

features *stop*, *move*, *speed*, *passenger_count*

Extending the basic notion of class



Basics of inheritance (subclassing)

Principle:

Describe a new class as extension or specialization of an existing class

(or several with ***multiple*** inheritance)

If *B* inherits from *A* :

- As modules: all the services of *A* are available in *B*
  (possibly with a different implementation)

- As types: whenever an instance of *A* is required, an instance of *B* will be acceptable
  ("is-a" relationship, e.g. *CAR* is a *VEHICLE*)

Terminology

If *B* inherits from *A* (by listing *A* in its **inherit** clause):

- *B* is an **heir** of *A*
- *A* is a **parent** of *B*

For a class *A*:

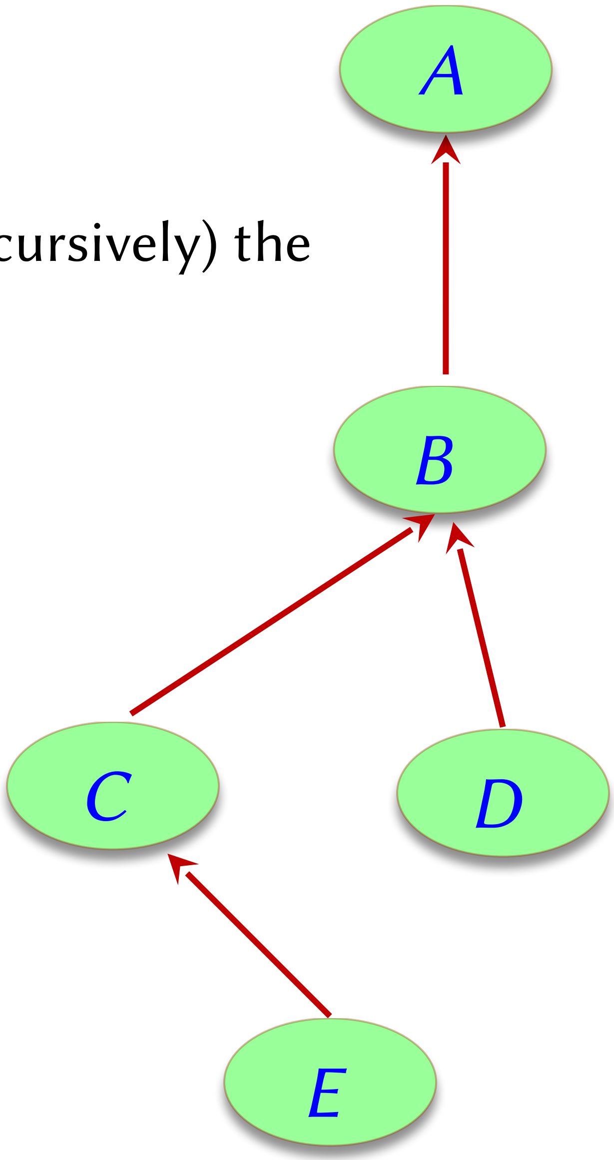
- The **descendants** of *A* are *A* itself and (recursively) the descendants of *A* 's heirs
- **Proper descendants** exclude *A* itself

Reverse notions:

- **Ancestor**
- **Proper ancestor**

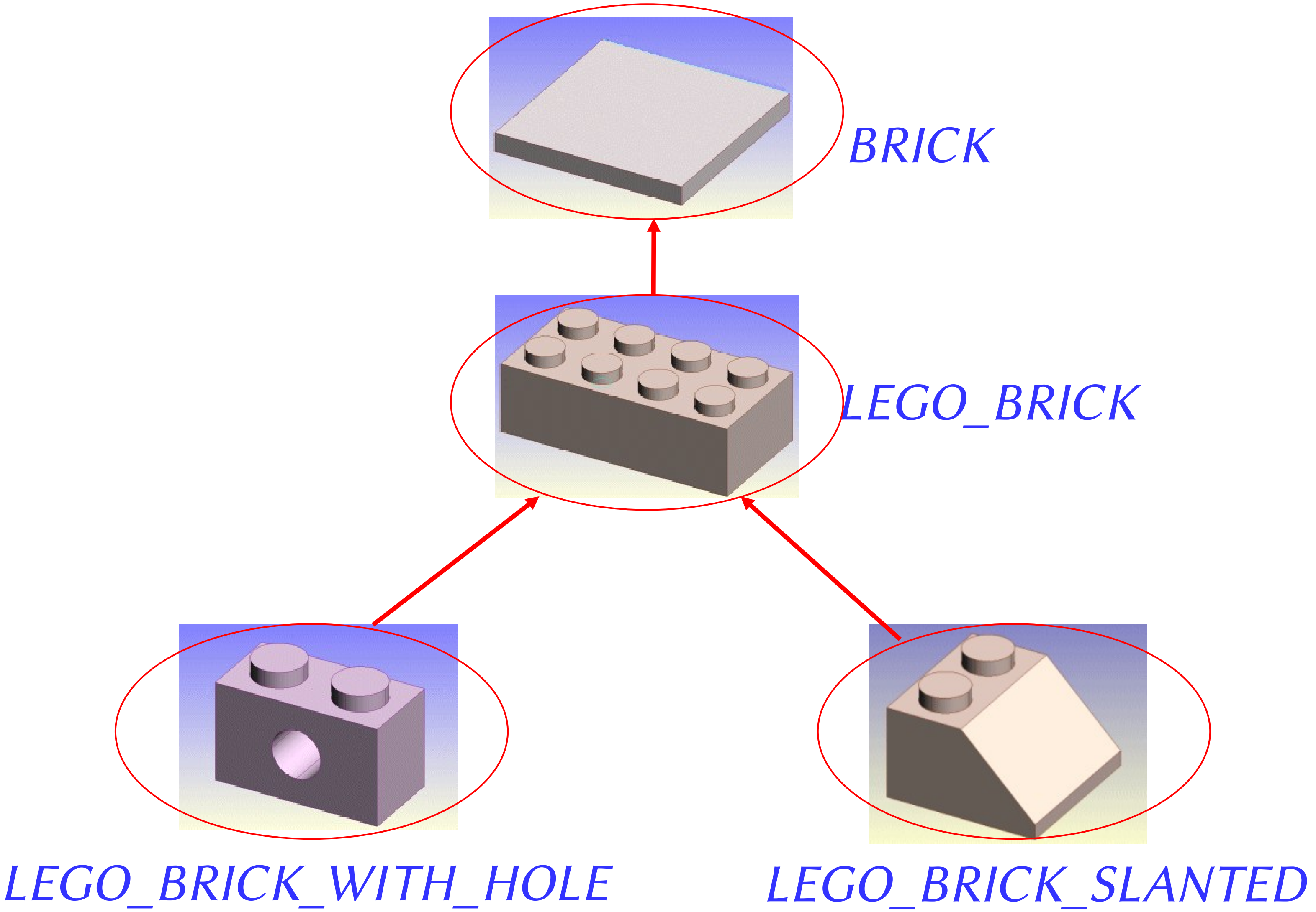
More precise notion of instance:

- **Direct instances** of *A*
- **Instances** of *A*: the direct instances of *A* and its descendants



(Other terminology: subclass, superclass, base class)

Let's play Lego!



Class *BRICK*

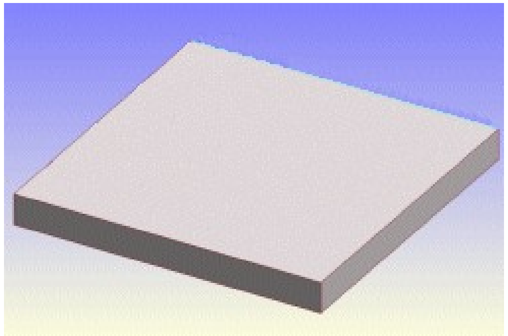

```
deferred class
        BRICK

feature
        width: INTEGER
        depth: INTEGER
        height: INTEGER
        color: COLOR

        volume: INTEGER
                deferred
                end
end
```

Explained later

Explained later



Deferred classes and features

- A **deferred class** is declared as such with the keyword *deferred*
- Deferred classes **cannot** be instantiated and hence **cannot** contain a *create* clause
- A class with *at least one deferred* feature **must** be declared as deferred, but…
  - … a class with *all effective* features **can** be defined as deferred
- A **deferred feature** does **not** provide an implementation
  - **deferred** instead of **do … … …**

Class *LEGO_BRICK*

Inherit all features of class *BRICK.*

New feature, calculate all nubs

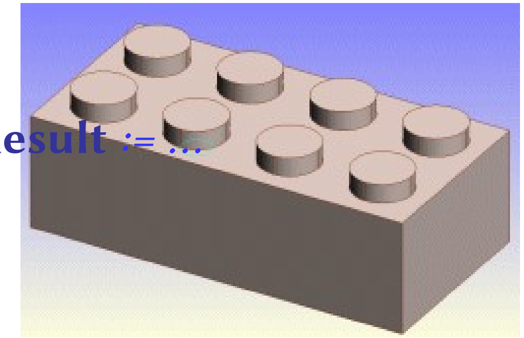
Implementation of *volume* (was deferred in class *BRICK*)

```
class
        LEGO_BRICK

inherit
        BRICK

feature
        number_of_nubs: INTEGER
                do
                        Result := ...
                end

        volume: INTEGER
                do
                        Result := ...
                end
end
```



Effective

- Effective
  - Effective classes do not have deferred features (the "standard case").
  - Effective routines have an implementation of their feature body.
  - Effective classes can be instantiated

Terminology: **Effective** = non-deferred
(i.e. fully implemented)

Deferred

- Deferred classes **cannot** be instantiated and hence **cannot** contain a *create* clause
  - hence the target type of a *create* instruction **cannot** be a deferred class, but …
  - … variables of the type of a deferred class **can** be used and refer to objects !

Remember *BRICK* is a deferred class

a_brick: BRICK
a_lego_brick: LEGO_BRICK

create *a_brick* Wrong!

create *a_lego_brick* Correct!

a_brick := a_lego_brick Correct!

Deferred features

- A deferred feature does **not** have an implementation yet
  - **deferred** instead of **do … … …**
- A call to a deferred feature **can** be written:
  - it will only be executed for an instance of an effective (sub)-class
  - there is no way of executing a deferred feature for an instance of a deferred class, since such an instance can never be created

Remember *BRICK* is a deferred class and *LEGO_BRICK* is an effective sub-class of *BRICK*

a_brick: BRICK
a_lego_brick: LEGO_BRICK

create *a_lego_brick*
a_brick := a_lego_brick
a_brick.volume

It is deferred feature for a *a_brick*, but since *a_brick* can never refer to an instance of *BRICK* , only to an instance of an **effective** sub-class, there is no problem.

Class *LEGO_BRICK_SLANTED*

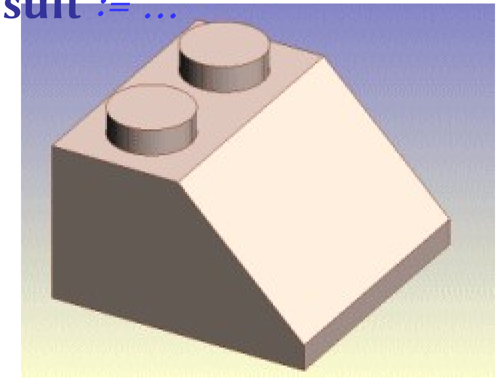

```
class
        LEGO_BRICK_SLANTED

inherit
        LEGO_BRICK
                redefine
                        volume
                end

feature
        volume: INTEGER
                do
                        Result := ...
                end
end
```

Declares previous implementation of *volume* is going to be changed.

The new implementation (substitutes the one coming from *LEGO_BRICK*)



Class *LEGO_BRICK_WITH_HOLE*

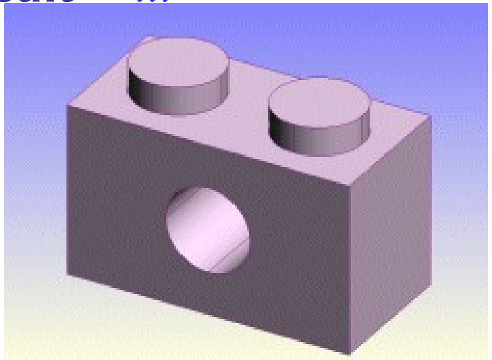

```
class
        LEGO_BRICK_WITH_HOLE

inherit
        LEGO_BRICK
                redefine
                        volume
                end

feature
        volume: INTEGER
                do
                        Result := ...
                end
end
```

Declares previous implementation of *volume* is going to be changed.

The new implementation (substitutes the one coming from *LEGO_BRICK*)



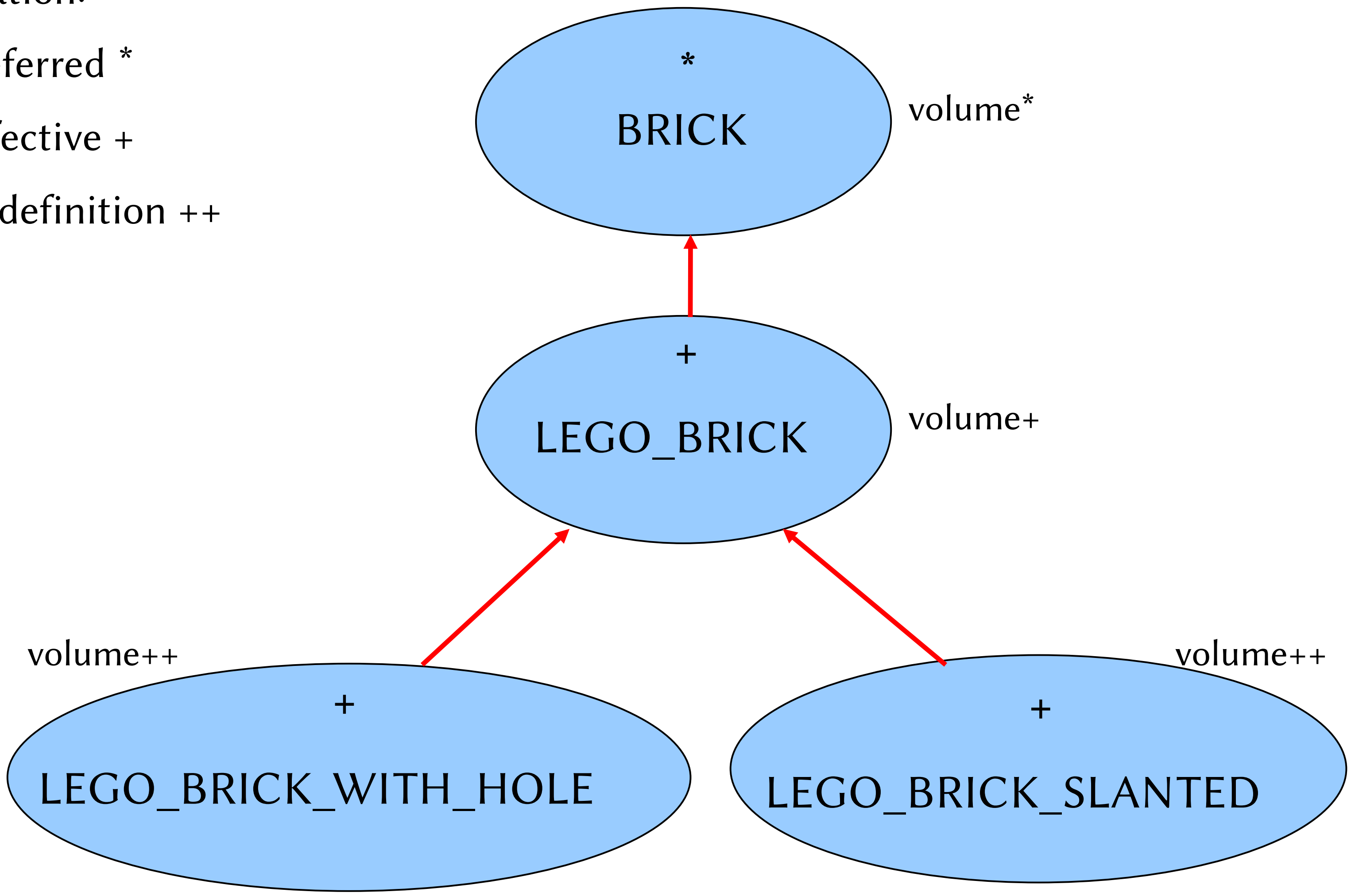
Inheritance Notation

Notation:

Deferred *

Effective +

Redefinition ++



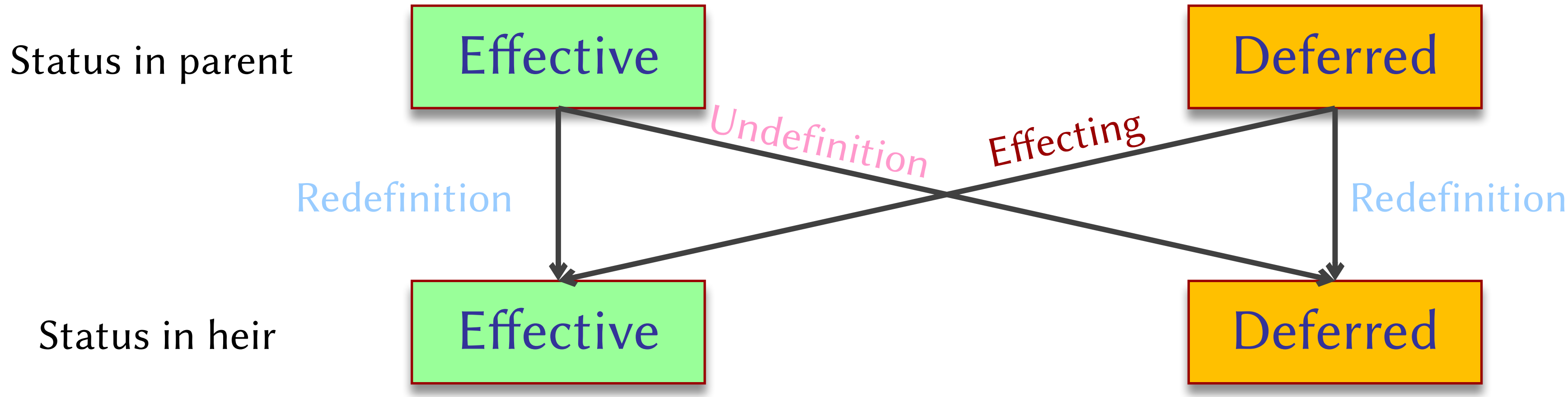
Redeclaration of features (1)

Redeclaration is the general term covering various cases:

- **Effecting**: transforming a deferred feature into an effective one
- **Undefining**: transforming an effective feature into a deferred one
- **Redefining**: changing signature, contract, implementation of a deferred or effective feature

undefine

redefine



Redeclaration of features (2)

Redefining an *effective* feature may change:

- contracts
- implementation
- signature (both arguments and result), keeping conformance

covariance rule: class and feature must **both** become more specific

Effecting a *deferred* feature may change:

- contracts
- signature (both arguments and result), keeping conformance

covariance rule: class and feature must **both** become more specific

An attribute **cannot** be redefined as a function

- for performance reasons (implies replacing a simple memory access with potentially a function call)

A function **can** be redefined as an attribute

Precursor

- If a feature was redefined, but you still wish to call the old version of the **same** feature, use the **Precursor** keyword (possibly with arguments) within the redefining body
  - It has the effect of calling the feature as inherited from the super class
  - **Cannot** be used to call the inherited version of another feature (you can call only the inherited version of the same feature)
  - It must be used as an expression or instruction depending on the kind of feature (query or command)

```
volume: INTEGER
    do
        ... Precursor ...
    end
```

Example hierarchy (from Traffic)

FOR CLASSES

↑ Inherits from

* Deferred: class does not need to have deferred features

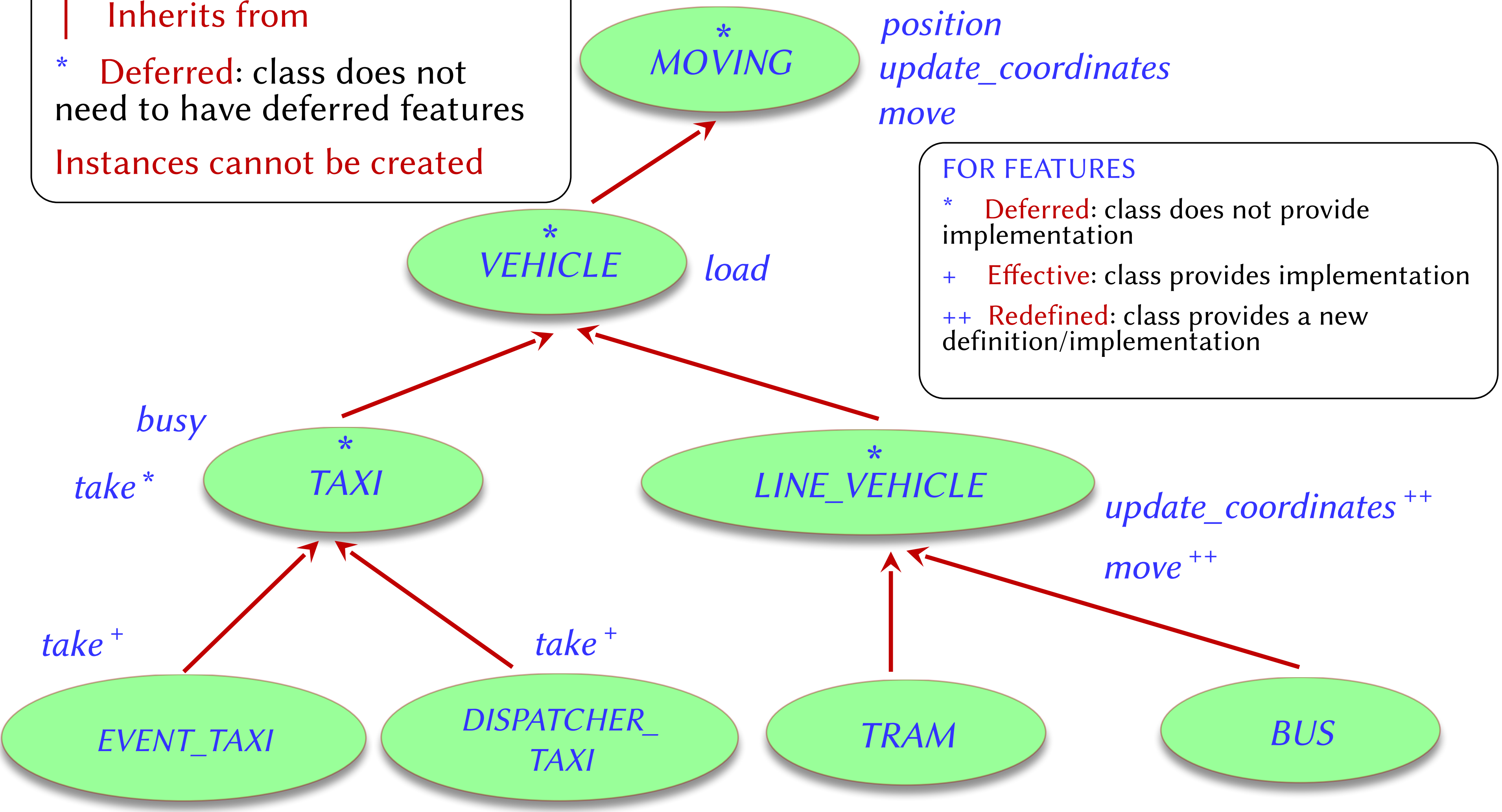
Instances cannot be created

FOR FEATURES

* Deferred: class does not provide implementation

\+ Effective: class provides implementation

++ Redefined: class provides a new definition/implementation



Features in the example

Feature	defined in class
take (from_location, to_location : COORDINATE) -- Bring passengers -- from `*from_location*' -- to `*to_location*'	*EVENT_TAXI* *DISPATCHER_TAXI*
busy : BOOLEAN -- Is taxi busy?	*TAXI*
load (q : INTEGER) -- Load `*q*' passengers.	*VEHICLE*
position : COORDINATE -- Current position on map.	MOVING

Inheriting features

```
deferred class
    VEHICLE
inherit
    MOVING
feature
    [... Rest of class ...]
end
```

All features of *MOVING* are applicable to instances of *VEHICLE*

For *v: VEHICLE* we can write *v.move*

```
deferred class
    TAXI
inherit
    VEHICLE
feature
    [... Rest of class ...]
end
```

All features of *VEHICLE* are applicable to instances of *TAXI*

For *t: TAXI* we can write *t.load*

```
class
    EVENT_TAXI
inherit
    TAXI
feature
    [... Rest of class ...]
end
```

All features of *TAXI* are applicable to instances of *EVENT_TAXI*

For *e: EVENT_TAXI* we can write *e.busy*

Definitions: kinds of feature

A "**feature of a class**" is one of:

- An **inherited** feature if it is a feature of one of the ancestors of the class.

- An **immediate** feature if it is declared in the class, and not inherited. In this case the class is said to **introduce** the feature.

Changing export status of inherited features (1)

A feature of the parent may become interesting to clients of the descendant

- To be able to use it, its status has to be changed from secret to exported

A feature of the parent may not be suitable for direct use by clients of the descendant

- Its status will change from exported to secret
- For example, feature *fly* in a class *BIRD* does not make sense in the descendant *OSTRICH*

It is possible to arbitrarily change the export status of any inherited feature

Changing export status of inherited features (2)


```
class
        AN_HEIR

inherit
        A_PARENT
                export
                        {class_X, class_Y, …}   feature_A, feature_B, …
                        {class_W, class_Z, …}   feature_C, feature_D, …
                end
…
end
```

New export status

{NONE} make the feature(s) secret

keyword **all** may be used instead of explicitly listing features…

… however, explicit listing takes precedence over implicit listing by means of **all**

Inherited features

m: *MOVING*; *v*: *VEHICLE*; *t*: *TAXI*;
e: *EVENT_TAXI*; *d*: *DISPATCHER_TAXI*

e.*load* (...)
d.*take* (...)
e.*position* -- query
d.*busy* -- query

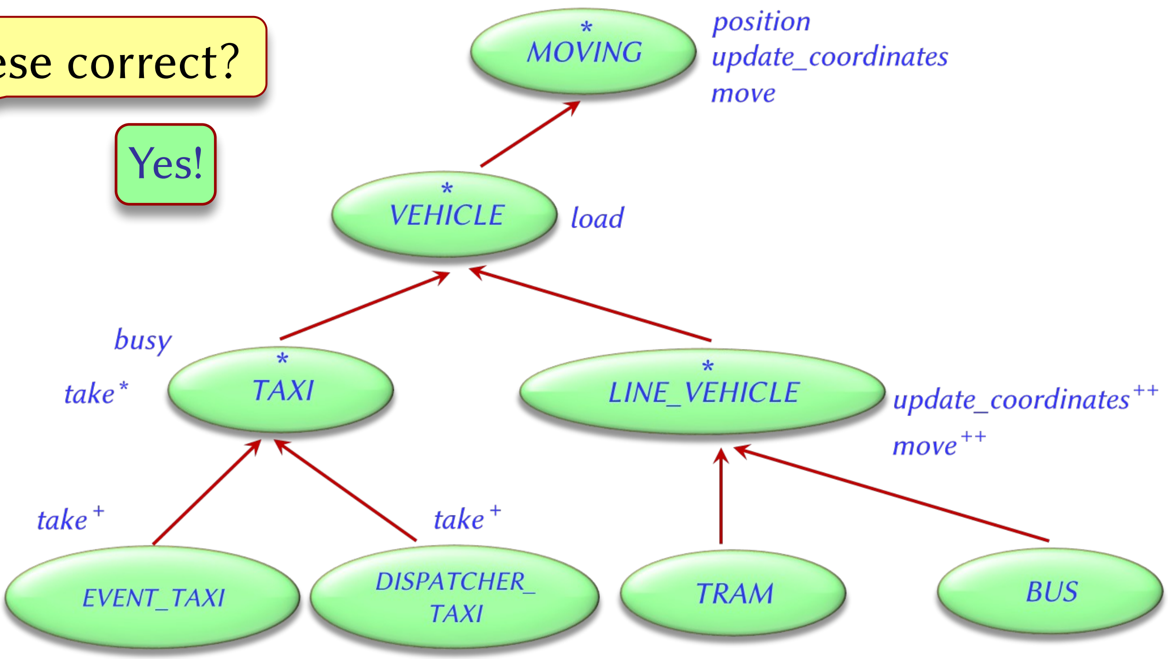
Are these correct?

Yes!

m.*load* (...)
v.*busy* -- query

Are these correct?

No!



Polymorphic assignment

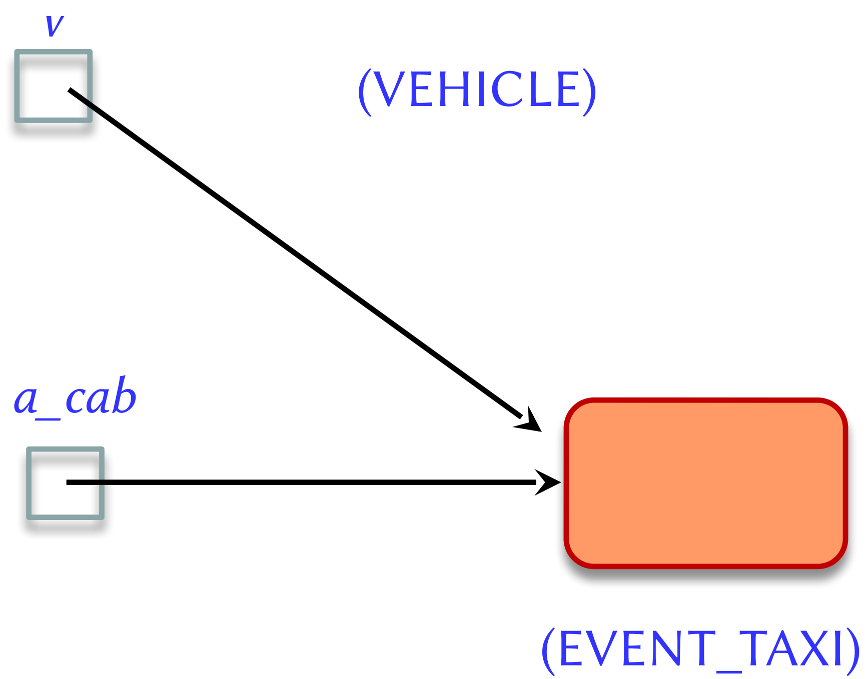
A **proper descendant** type of the original

v : *VEHICLE*

a_cab : *EVENT_TAXI*

a_tram: TRAM

v := *a_cab*



More interesting:

```
if some_condition then
        v := a_cab
else
        v := a_tram
end
…
```

Assignments

Assignment:

target **:=** *expression*

So far (no polymorphism):

expression was always of the **same type** as *target*

With polymorphism:

The type of *expression* is a **descendant** of the type of *target*

Polymorphism is also for argument passing

register_trip (v: VEHICLE)
do … **end**

A particular call:

register_trip (a_cab)

Type of actual argument is generally a **descendant** of type of formal

Definitions: Polymorphism

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

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

Polymorphism is the existence of these possibilities.

Definitions: Static and dynamic type

The **static type** of an entity is the type used in its declaration in the corresponding class text

If the value of the entity, during a particular execution, is attached to an object, the type of that object is the entity's **dynamic type** at that time

Static and dynamic type

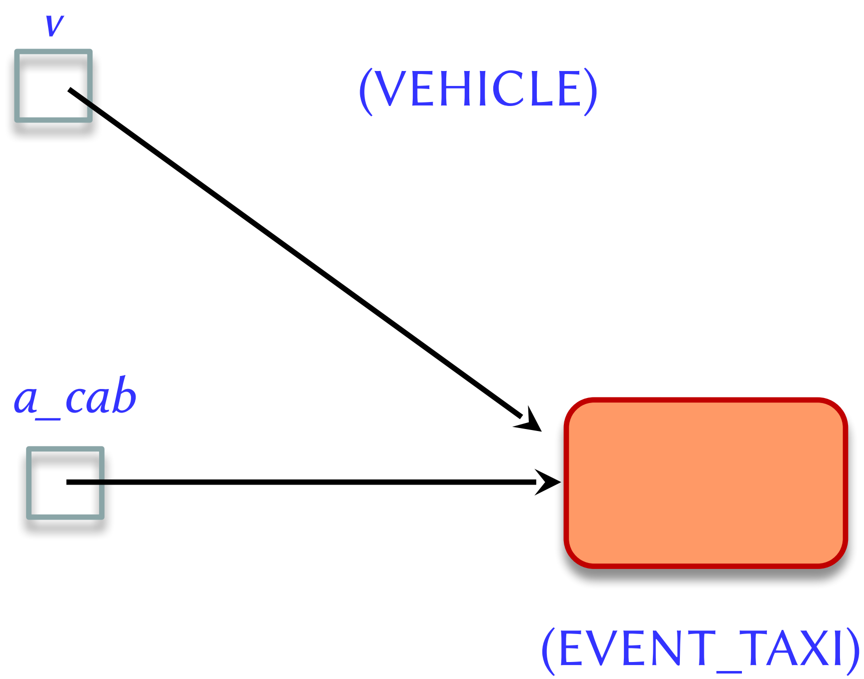
Static type of *v* : *VEHICLE*

v : *VEHICLE*

a_cab : *EVENT_TAXI*

v := *a_cab*

Dynamic type after this assignment:
EVENT_TAXI



Basic type property

Static and dynamic type

The dynamic type of an entity
must conform to its static type

(Ensured by the type system of the compiler)

Static typing

Type-safe call:

A feature call *x.f* such that any object attached to *x* during execution has a feature corresponding to *f*

[Generalizes to calls with arguments, *x.f (a, b)*]

Static type checker:

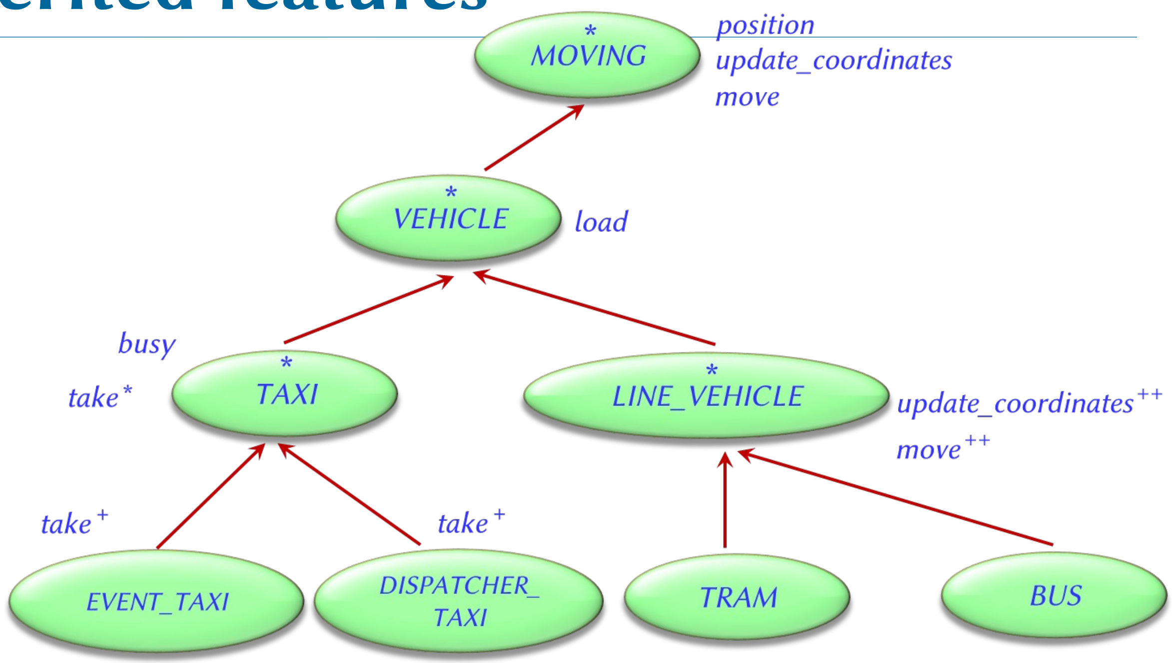
A program-processing tool (such as a compiler) that guarantees, for any program it accepts, that any call in any execution will be *type-safe*

Statically typed language:

A programming language for which it is possible to write a *static type checker*

Type safety and inherited features

m: *MOVING*
v: *VEHICLE*
t: *TAXI*;
e: *EVENT_TAXI*
d: *DISPATCHER_TAXI*



v•*load* (…)
e•*load* (…)
t•*take* (…)
d•*take* (…)
m•*move* (…)
e•*move* (…)

m•*load* (…)
m•*take* (…)

type-safe calls

type-unsafe calls

Conformance: base definition

Basic inheritance type rule

For a polymorphic attachment to be valid,
the type of the source must **conform**
to the type of the target

Conformance: base definition

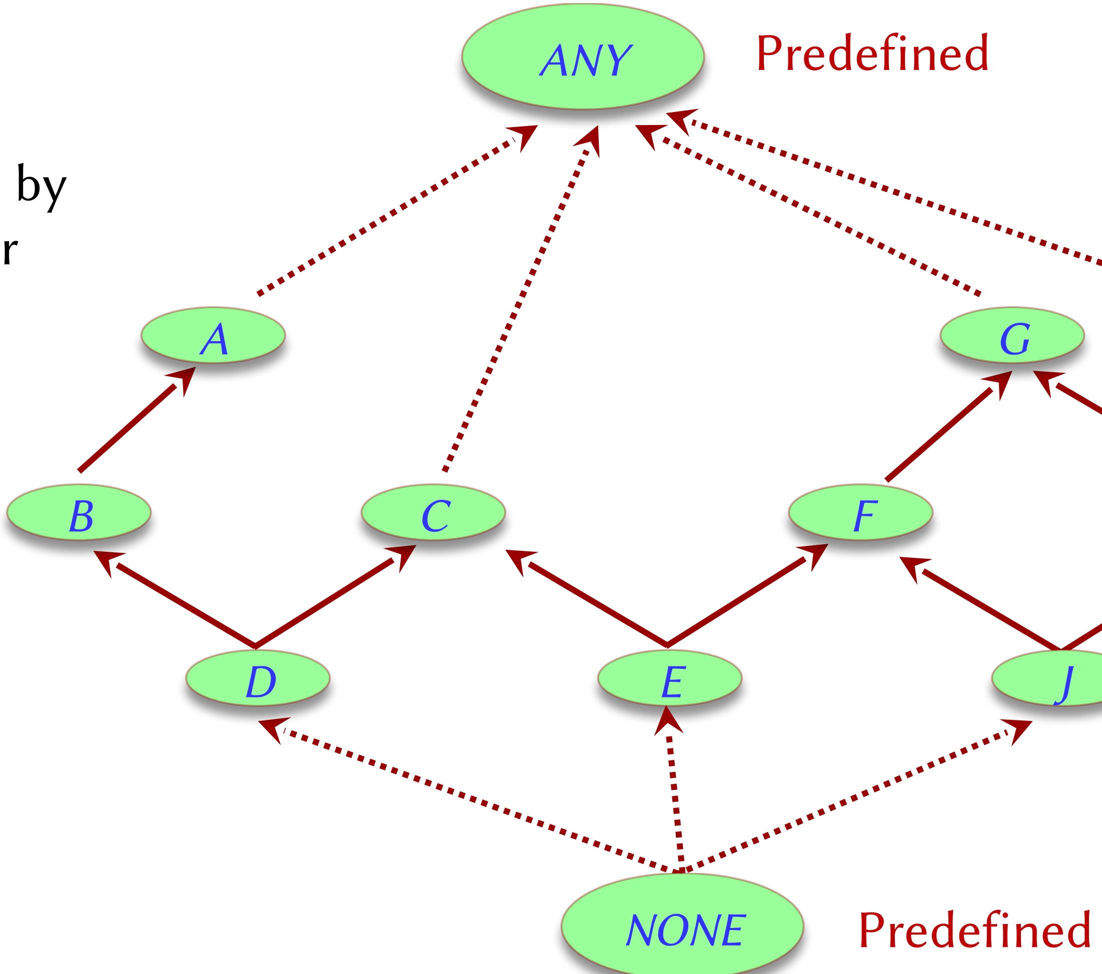
Reference types (non-generic):

U **conforms** to *T* if *U* is a descendant of *T*

An *expanded* type conforms only to itself

A fictitious inheritance hierarchy

Classes defined by
the programmer



The role of deferred classes

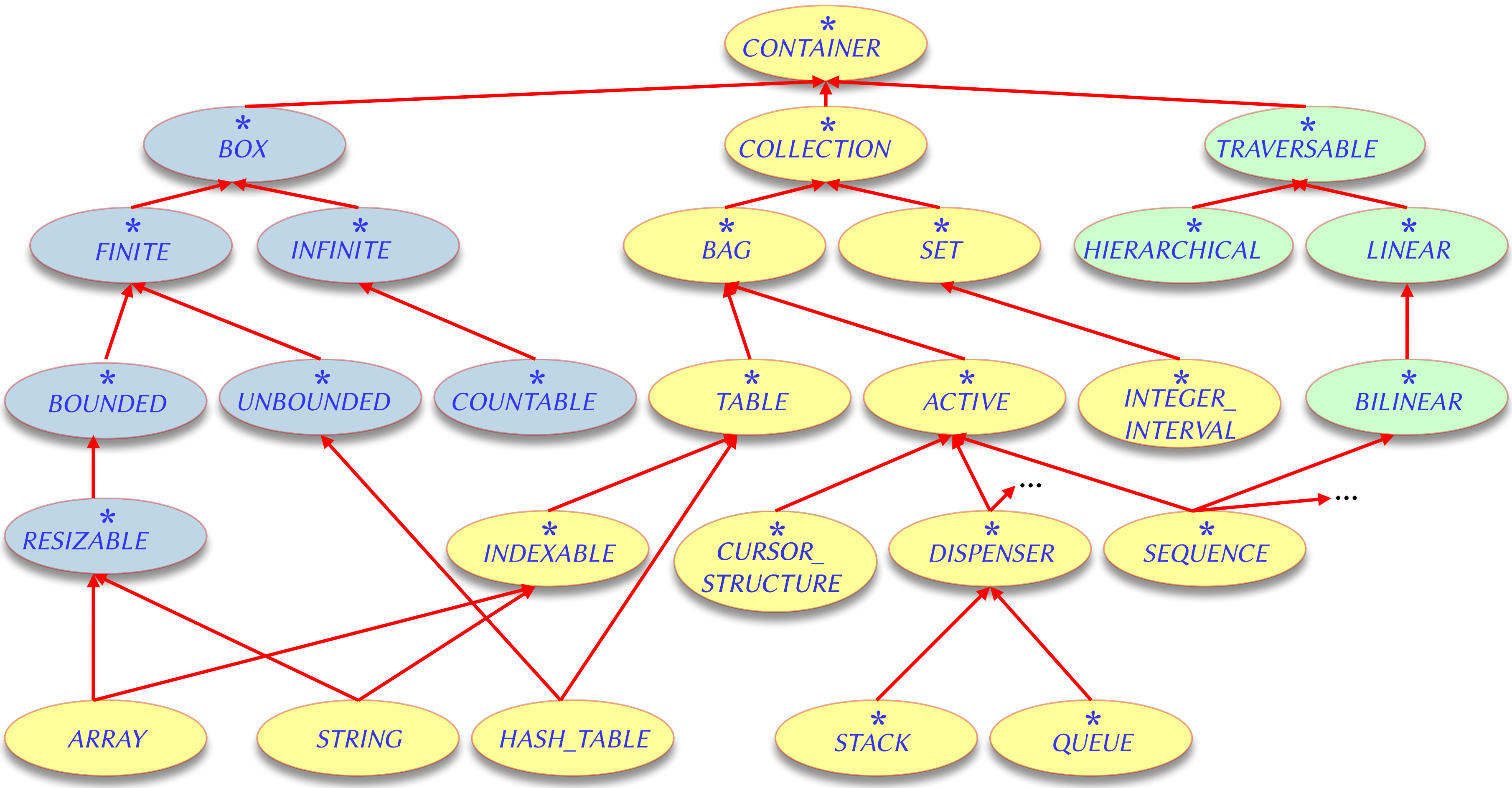
Top-down definition of software architecture without deciding too early on implementation

only hierarchies of names and contracts

Capturing high-level concepts and their taxonomy in the application domain

Representing common behaviors and their taxonomy in libraries

Deferred classes in EiffelBase



* deferred

A deferred feature

In e.g. *LIST*:

```
forth
    require
        not after
    deferred
    ensure
        index = old index + 1
    end
```

Mixing deferred and effective features

In the same class

Effective!

```
search (x: G)
        -- Move to first position after current
        -- where x appears, or after if none.
    do
        from until after or else item = x loop
            forth
        end
    end
```

Deferred!

“Programming with holes”

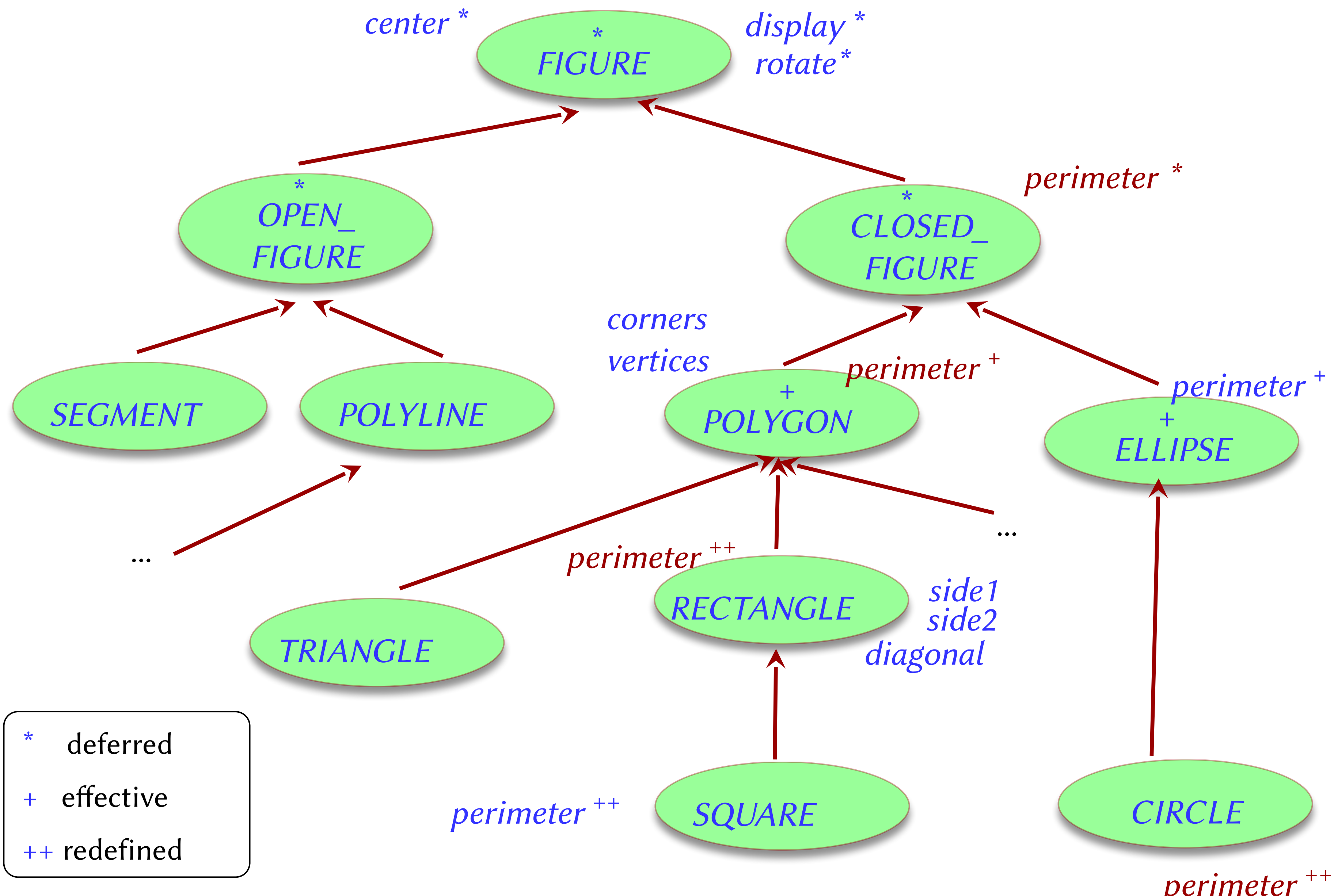
“Programming with holes”

A powerful form of reuse:

- The reusable element defines a general scheme
- Specific cases fill in the holes in that scheme

Combine reuse with adaptation

A more realistic example of inheritance hierarchy



Remember the basis of feature redefinition


```
class B
inherit
    A
            redefine
            f
            end
    …
```

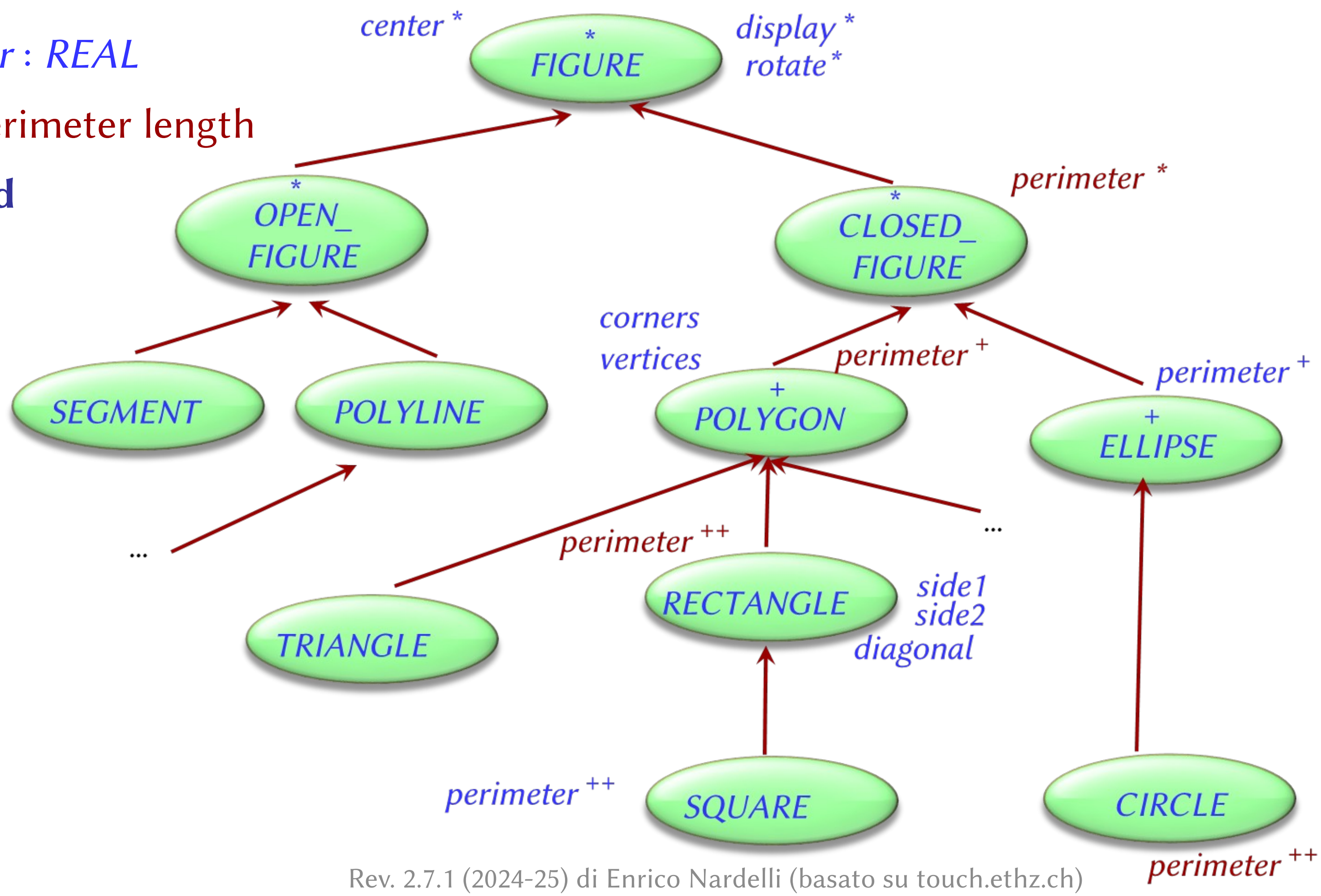
Signature (order, number and types of formal parameters, type of returned value) of redefinition of *f* in *B* must **conform** to signature of *f* in *A*

Creation procedure must be re-declared (i.e., the **create** clause in the ancestors' code is **not** inherited) but their definition is inherited. Instead, **default_create** doesn't need to be re-declared as creation procedure.

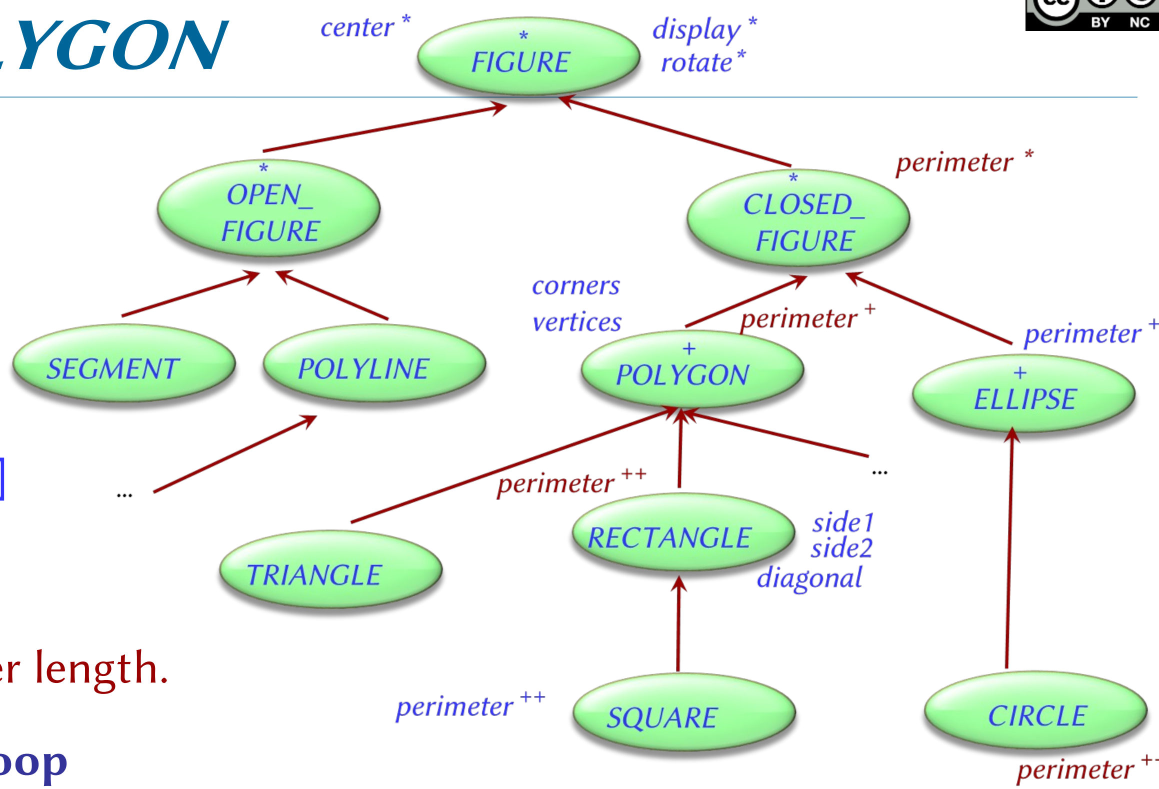
In the implementation of *f* in *B* the keyword **Precursor** (possibly with arguments) uses *A* 's version of *f*

Redefinition 1: *CLOSED_FIGURE*

```
deferred class CLOSED_FIGURE
inherit
        FIGURE
feature
        perimeter : REAL
                -- Perimeter length
        deferred
        end
end
```



Redefinition 2: *POLYGON*

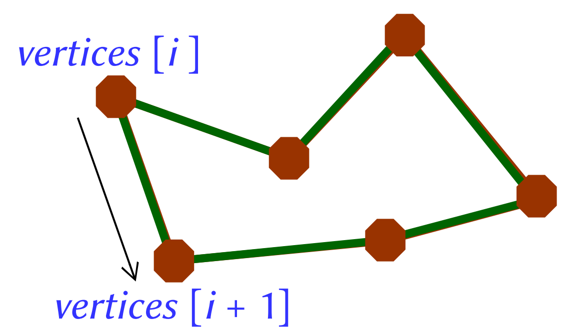



```
class POLYGON
inherit
        CLOSED_FIGURE
create
        make
feature
        vertices : ARRAY [POINT]

        corners : INTEGER

        perimeter : REAL
                        -- Perimeter length.
            do
                from ... until ... loop

                    Result := Result + vertices [i ] . distance (vertices [i + 1])
                    ...
                end
            end
        ...
invariant
        corners >= 3
        corners = vertices.count
```



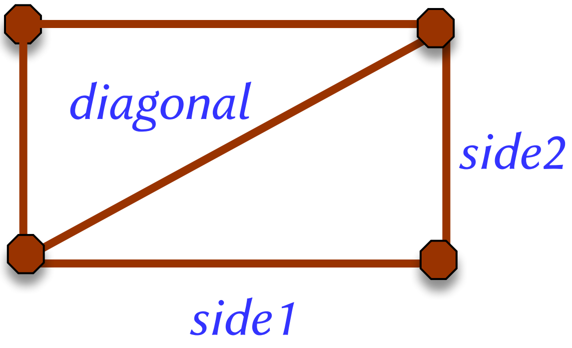
Redefinition 3: *RECTANGLE*


```
class RECTANGLE
inherit
    POLYGON
        redefine
            perimeter
        end
create
    make
feature
    diagonal, side1, side2 : REAL

    perimeter : REAL
            -- Perimeter length.
        do Result := 2 * (side1 + side2) end
invariant
    corners = 4
end
```

Must return a conforming **Result**!

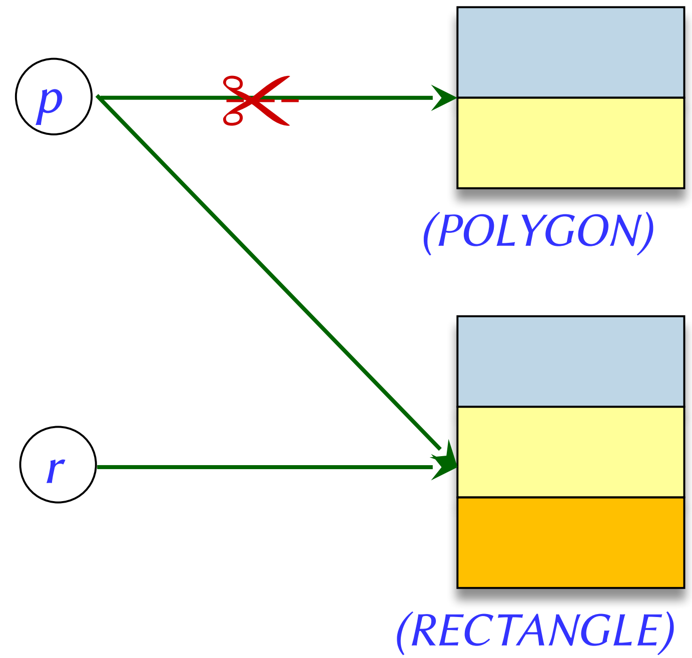
Inherited invariants still holds!



Inheritance, typing and polymorphism

Assume:

p: *POLYGON* ; *r*: *RECTANGLE* ; *t*: *TRIANGLE*
x: *REAL*



Permitted:

x := *p.perimeter*
x := *r.perimeter*
x := *r.diagonal*
p := *r*

Is it permitted (independently from what happens earlier)?

x := *p.diagonal*

Static type checker reveals an unsafe call: the target type does not know the feature

r := *p*

Source does not conform to the target!

Dynamic binding

What is the effect of the following?

```
if some_test then
    p := r
else
    p := t
end
x :=   p.perimeter
```

Redefinition: A class may change an inherited feature, as with *POLYGON* redefining *perimeter*.

Polymorphism: *p* may have different forms at run-time.

Dynamic binding: Effect of *p.perimeter* depends on the run-time form of *p*, which determines the executed version of *perimeter*

Definitions: Dynamic binding

Dynamic binding (a semantic rule):

- Any execution of a feature call will use the version of the feature best adapted to the type of the target object

Binding and typing

(For a call $x.f$)

Static typing: The guarantee that there is at least one version for f

Dynamic binding: The guarantee that every call will use the most appropriate version of f

Without dynamic binding?


```
display (f: FIGURE)
    do
        if "f is a CIRCLE" then
            ...
        elseif "f is a POLYGON" then
            ...
        end
    end
```

and similarly for all other routines!

Tedious; must be changed whenever there's a new figure type

With inheritance and associated techniques

With:

```
f : FIGURE
c : CIRCLE
p : POLYGON
```

and:

```
create c.make (...)
create p.make (...)
```

Initialize:

```
if ... then
    f := c
else
    f := p
end
```

Then just use:

```
f.move (...)
f.rotate (...)
f.display (...)
        -- and so on for every
        -- operation on f !
```

Creation and inheritance

Assume:

p : *POLYGON*

t : *TRIANGLE*

r : *RECTANGLE*

Right or wrong?:

create *t*

p := *t*

Right!

create *p*

t := *p*

Wrong!

Creation expression and instruction

With *p* : *POLYGON*

> Must be a subclass

create {*TRIANGLE*} *p*

> *p* created with type *TRIANGLE*

it's a creation **instruction**

> Must be a subclass

p := **create** {*RECTANGLE*}

> *p* created with type *RECTANGLE*

it's a creation **expression**

The latter is useful for anonymous object creation

Instead of

p := **create** {*RECTANGLE*}
target.set (*p*)

Just write

target.set (**create** {*RECTANGLE*})

> anonymous object creation

Be aware!

Assume:

p : *POLYGON*

t : *TRIANGLE*

r : *RECTANGLE*

Right or wrong?:

create {*TRIANGLE*} *p*

t := *p*

Wrong!

create *p* -- create a *POLYGON*

create *t* -- create a *TRIANGLE*

p := *t* -- source conforms to target

t := *p* -- source does not conform to target

p := **create** {*RECTANGLE*}

r := *p*

Wrong!

p := **create** {*POLYGON*}
-- create a *POLYGON*

r := **create** {*RECTANGLE*}
-- create a *RECTANGLE*

p := *r* -- source conforms to target

r := *p* -- source does not conform to target

Contracts and inheritance

Issue: what happens, under inheritance, to

- Class invariants?

- Routine preconditions and postconditions?

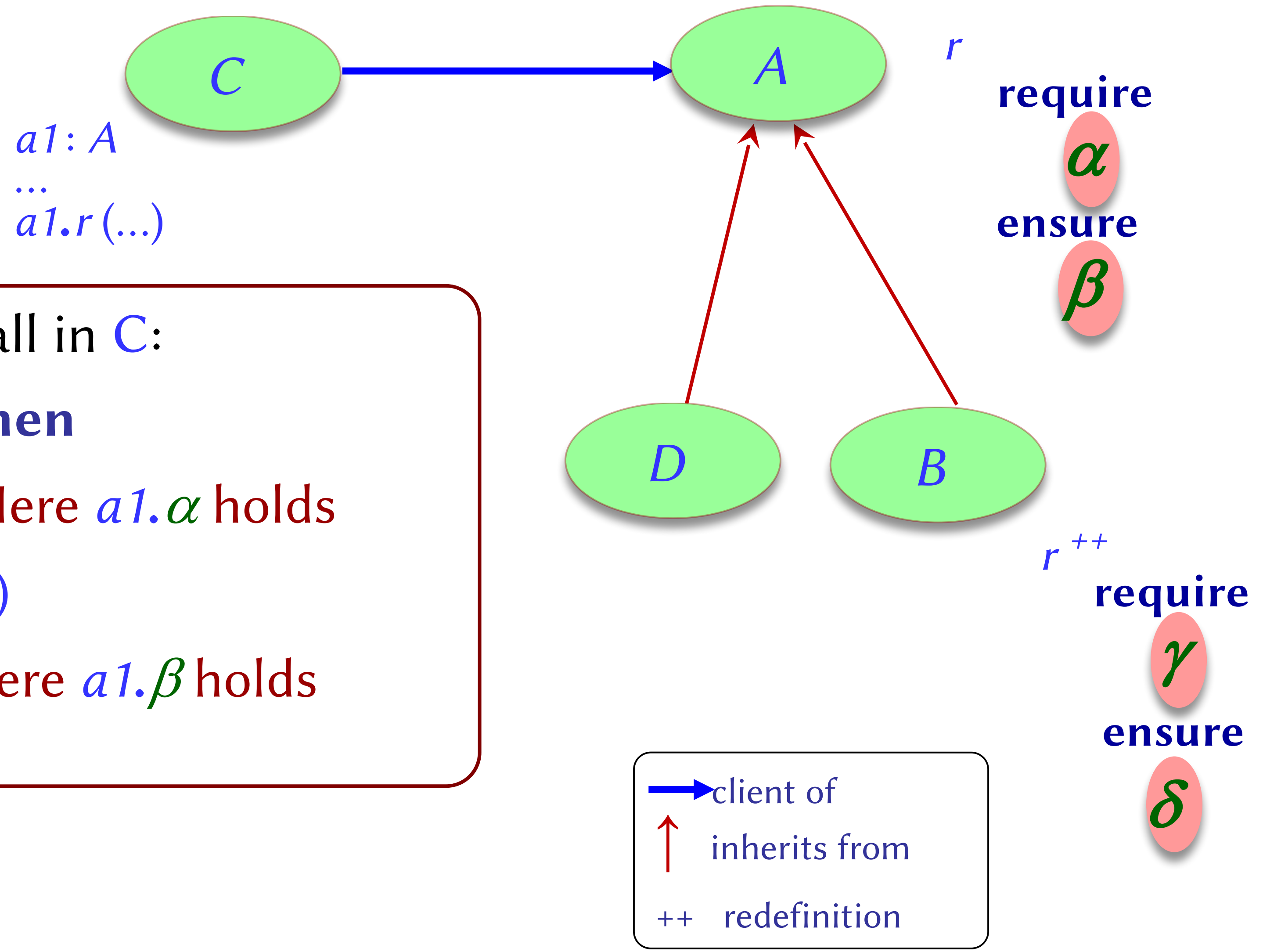
Invariants

Invariant Inheritance rule:

- The invariant of a class automatically includes the invariant clauses from all its parents
- Remember: all invariant clauses are "AND"-ed.

Accumulated result is visible in flat and interface views.

Contracts and inheritance



Standard call in C:

if *a1.α* **then**

-- Here *a1.α* holds

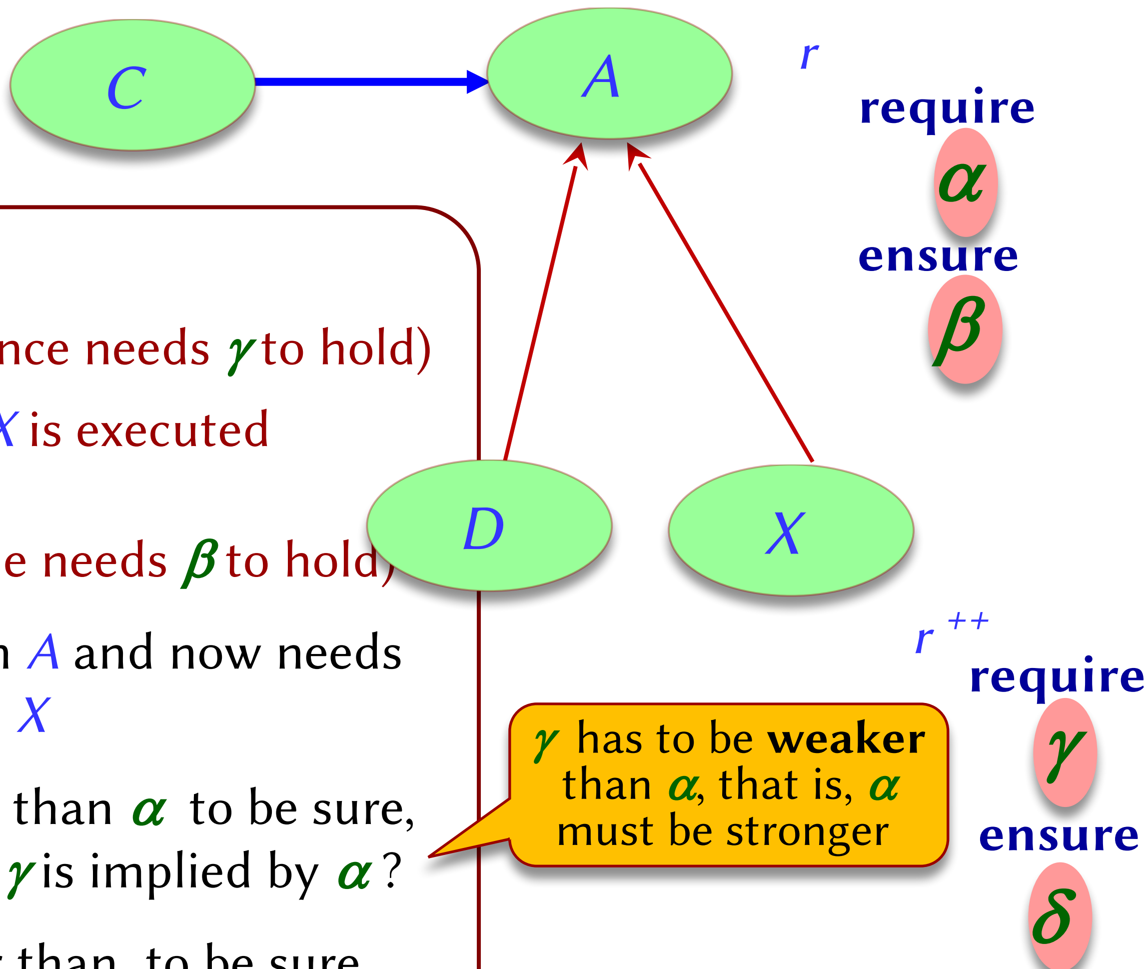
a1.r (...)

-- Here *a1.β* holds

end

Contracts and inheritance

a1 : *A*
x1 : *X*
…
a1* := *x1



if *a1*.α **then**
-- Here α holds (but the instance needs γ to hold)

a1.*r* -- The version of *r* in *X* is executed

end
-- Here δ holds (but other code needs β to hold)

Code satisfied contracts of *r* in *A* and now needs to satisfy also contracts of *r* in *X*

γ has to be stronger or weaker than α to be sure, before the execution of *r*, that γ is implied by α ?

δ has to be stronger or weaker than to be sure, after the execution of *r*, that β is implied by δ ?

γ has to be **weaker** than α, that is, α must be stronger

δ has to be **stronger** than β, that is, β must be weaker

Assertion redeclaration rule

When redeclaring a routine, we may **only**:

- Keep or **weaken** the precondition

- Keep or **strengthen** the postcondition

Assertion redeclaration rule in Eiffel

A simple language rule does the trick!

Contracts in the redefined version of feature in the subclass may say nothing (assertions kept by default), or say

require else *new_pre*
ensure then *new_post*

Resulting complete assertions in the subclass are:

- *original_precondition* **or else** *new_precondition*
- *original_postcondition* **and then** *new_postcondition*

provides one more possibility: **weaker**

provides one more constraint: **stronger**

Inheritance: summary

Type mechanism: lets you organize our data abstractions into taxonomies

Module mechanism: lets you build new classes as extensions of existing ones

Polymorphism: Flexibility *with* type safety

Dynamic binding: automatic adaptation of operation to target, for more modular software architectures

What we have seen

The basics of fundamental O-O mechanisms:

- Inheritance
- Polymorphism
- Dynamic binding

Characteristic of Eiffel implementation of O-O:

- Static typing