



Einführung in die Programmierung Introduction to Programming

Prof. Dr. Bertrand Meyer

Exercise Session 10



- Recursion
 - Recursion
 - Recursion
 - Recursion
 - Recursion
- Inheritance
- Genericity

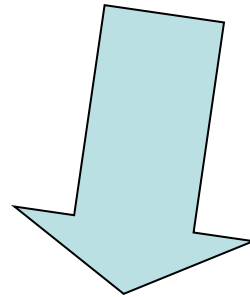


A definition for a concept is **recursive** if it involves an instance of the concept itself

- The definition may use more than one "*instance of the concept itself*"
- *Recursion* is the use of a recursive definition

„To iterate is human, to recurse - divine!“

but ... computers are built by humans 



Better use iterative approach if reasonable?



- Every recursion could be rewritten as an iteration and vice versa.
- BUT, depending on how the problem is formulated, this can be difficult or might not give you a performance improvement.

Exercise: Printing numbers

Hands-On

- If we pass $n = 4$, how many numbers will be printed and in which order?

```
print_int (n: INTEGER)  
do  
  print (n)  
  if  $n > 1$  then  
    print_int (n - 1)  
  end  
end
```

4321

```
print_int (n: INTEGER)  
do  
  if  $n > 1$  then  
    print_int (n - 1)  
  end  
  print (n)  
end
```

1234

Exercise: Reverse string

Hands-On

- Print a given string in reverse order using a recursive function.

Exercise: Solution



```
class APPLICATION

  create
    make

  feature
    make
      local
        s: STRING
      do
        create s.make_from_string ("poldomangia")
        invert(s)
      end

      invert (s: STRING)
        require
          s /= Void
        do
          if not s.is_empty then
            invert (s.substring (2, s.count))
            print (s[1])
          end
        end
      end
    end
  end
end
```

Exercise: Sequences

Hands-On

- Write a recursive and an iterative program to print the following:

111,112,113,121,122,123,131,132,133,
211,212,213,221,222,223,231,232,233,
311,312,313,321,322,323,331,332,333.

- Note that the recursive solution can use loops too.

Exercise: Recursive solution



cells: ARRAY [INTEGER]

handle_cell (n: INTEGER)

local

i: INTEGER

do

from

i := 1

until

i > 3

loop

cells [n] := i

if (n < 3) then

handle_cell (n+1)

else

print ("("+cells [1].out+", "+cells [2].out+", "+cells [3].out+")")

end

i := i + 1

end

end

Exercise: Iterative solution



```
from
  i := 1
until
  i > 3
loop
  from
    j := 1
  until
    j > 3
  loop
    from
      k := 1
    until
      k > 3
    loop
      print ("("+i.out+", "+j.out+", "+k.out+")")
      k := k + 1
    end
    j := j + 1
  end
  i := i + 1
end
```

Exercise: Magic Squares



- A magic square of size $N \times N$ is a $N \times N$ square such that:
 - Every cell contains a number between 1 and N^2 .
 - The sums in every row and column is constant.
 - The numbers are all different.

4	3	8
9	5	1
2	7	6

Exercise: Magic Squares



- Finding a magic square 3×3 is related to finding the permutations of 1 to 9.
- There exist 72 magic 3×3 squares.

123456789

123456798

123456879

123456897

123456978

123456987

...

987654321

Exercise: Magic Squares

Hands-On

- Write a program that finds all the 3x3 magic squares.
- Hints
 - Reuse the previous recursive algorithm by applying it to permutations (enforce no repetitions).
 - Use two arrays of 9 elements, one for the current permutation and one to know if a number has already been used or not.

Exercise: Solution



➤ See code in IDE.

Going backwards pays off



- Sometimes even by looking at a part of a potential solution we can say that it is wrong. In this case we can discard the whole set of potential solutions even before they have been built.
- This technique is called Backtracking.
- See code in IDE.



- You have seen several data structures
 - *ARRAY, LINKED_LIST, HASH_TABLE, ...*
- We will now look at another data structure and see how recursion can be used for traversal.

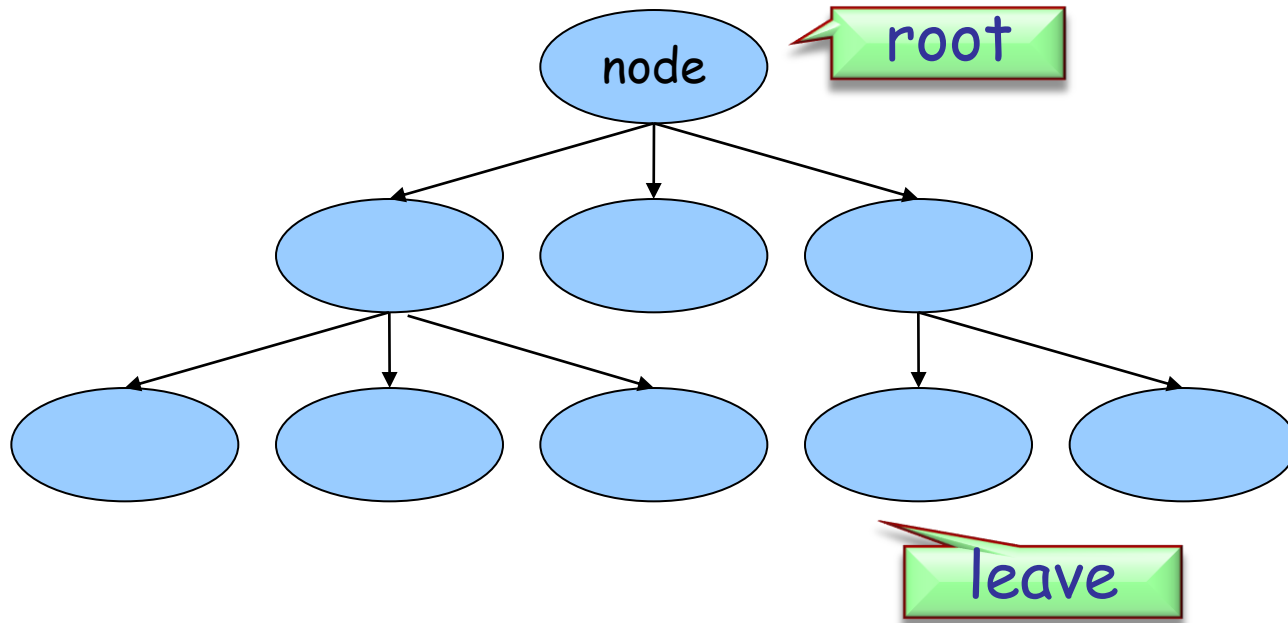
Tree



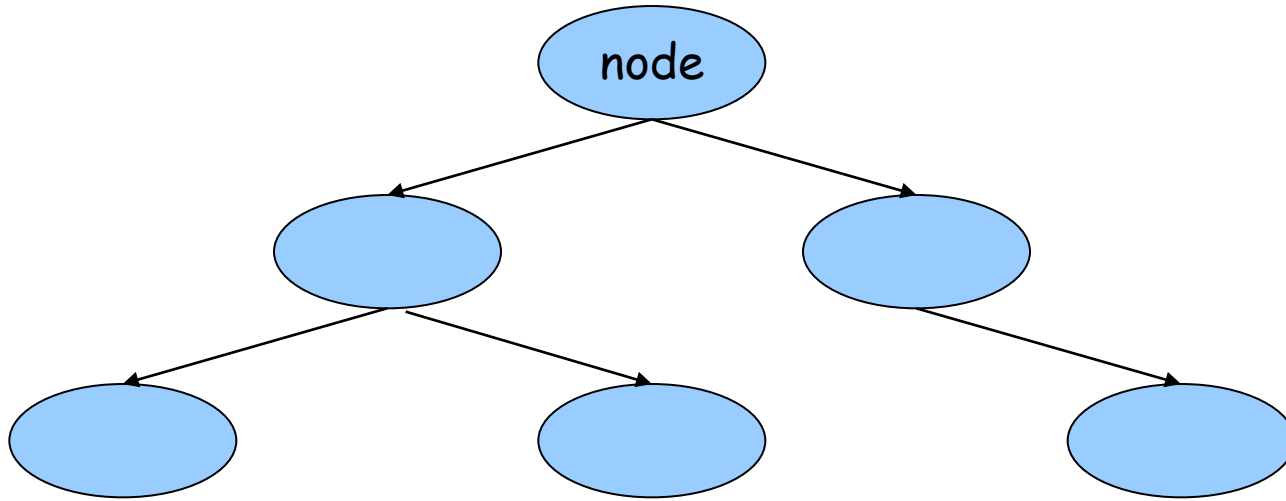
Tree



Tree: A more abstract way



- A non-empty tree has one root. An empty tree does not have a root.
- Every non-leaf node has links to its children. A leaf does not have children.
- There are no cycles.



- A binary tree is a tree.
- Each non-leaf node can have at most 2 children (possibly 0 or 1).

Exercise: Recursive traversal

Hands-On

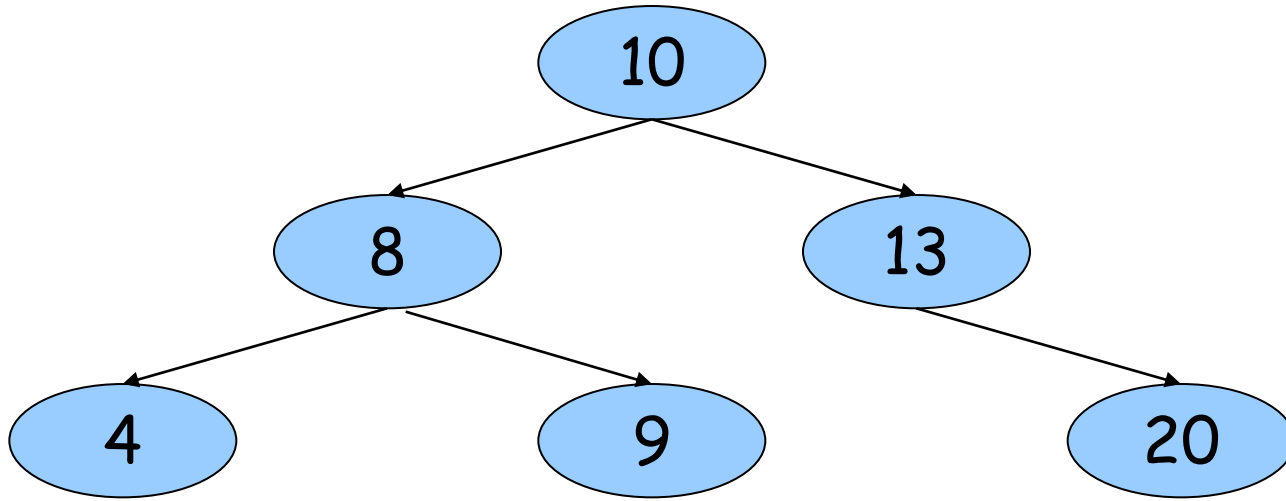
- Implement class *NODE* with an *INTEGER* attribute.
- In *NODE* implement a recursive feature that traverses the tree and prints out the *INTEGER* value of each *NODE* object.
- Test your code with a class *APPLICATION* which builds a binary tree and calls the traversal feature.

Exercise: Solution



➤ See code in IDE.

Binary search tree



- A binary search tree is a binary tree where each node has a *COMPARABLE* value.
- Left sub-tree of a node contains only values less than the node's value.
- Right sub-tree of a node contains only values greater than or equal to the node's value.

Exercise: Adding nodes

Hands-On

- Implement command *put (n: INTEGER)* in class *NODE* which creates a new *NODE* object at the correct place in the binary search tree rooted by *Current*.
- Test your code with a class *APPLICATION* which builds a binary search tree using *put* and prints out the values using the traversal feature.
- Hint: You might need to adapt the traversal feature such that the values are printed out in order.

Exercise: Solution



➤ See code in IDE.

Exercise: Searching

Hands-On

- Implement feature *has* (*n*: *INTEGER*): *BOOLEAN* in class *NODE* which returns true if and only if *n* is in the tree rooted by *Current*.
- Test your code with a class *APPLICATION* which builds a binary search tree and calls *has*.

Exercise: Solution



➤ See code in IDE.



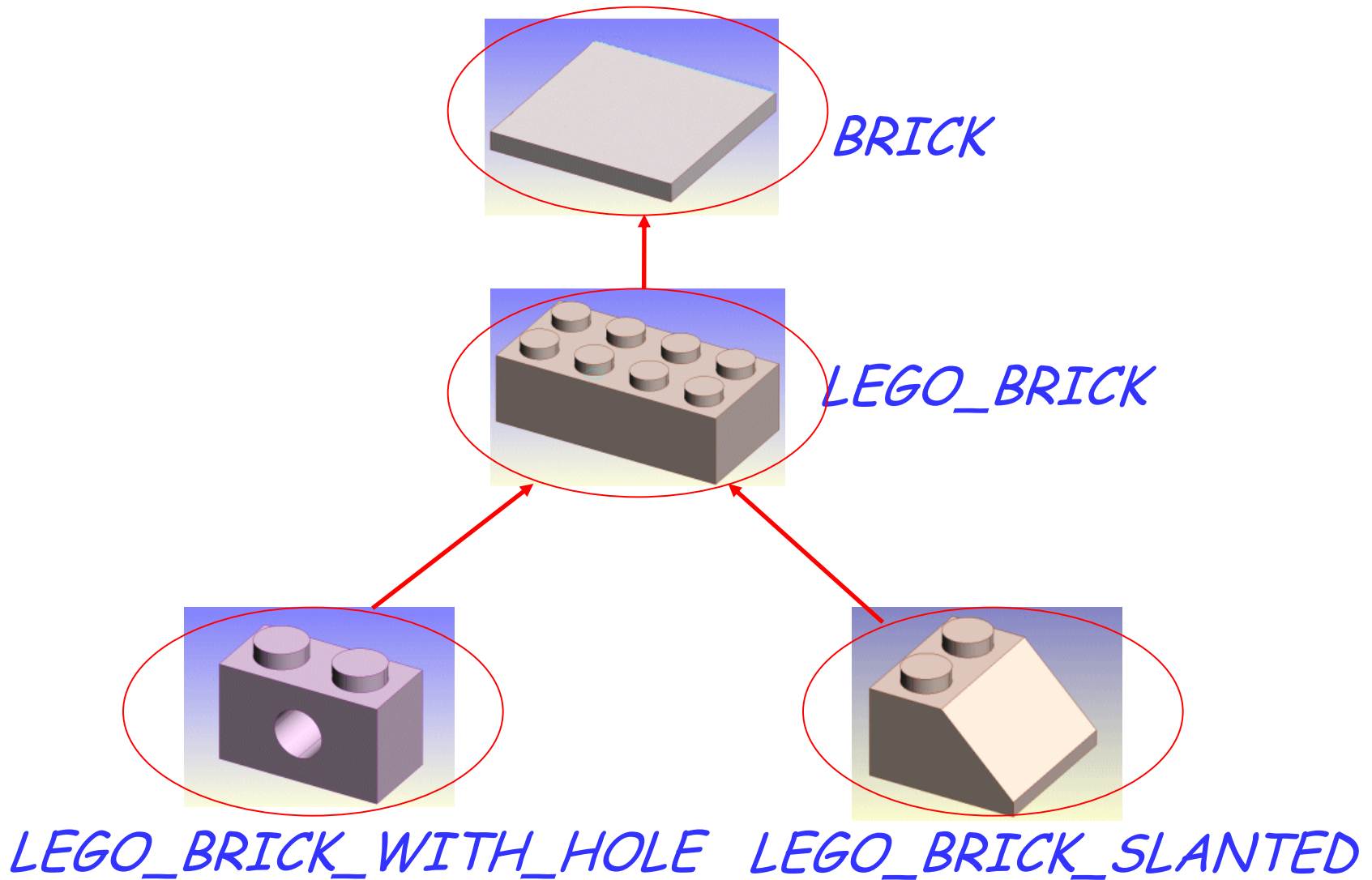
Principle:

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)

Let's play Lego!



Class *BRICK*



deferred class
BRICK

feature

width: INTEGER

depth: INTEGER

height: INTEGER

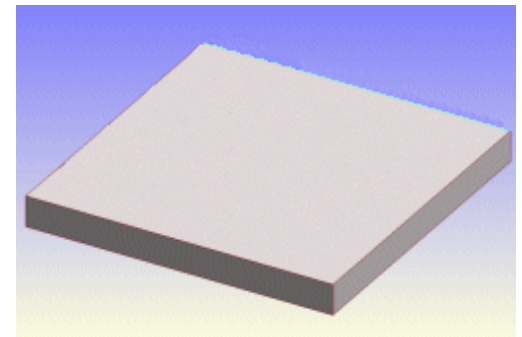
color: COLOR

volume: INTEGER

deferred

end

end



Class *LEGO_BRICK*



Inherit all features of class *BRICK*.

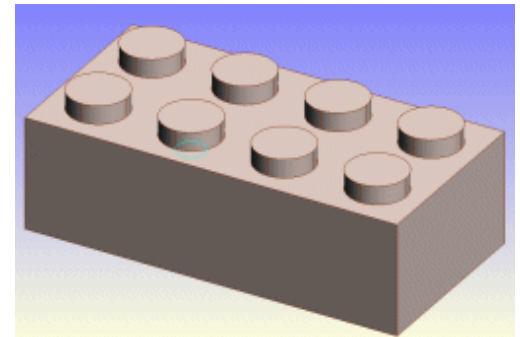
```
class  
  LEGO_BRICK  
inherit  
  BRICK
```

New feature, calculate all nubs

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

Implementation of *volume*.

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



Class *LEGO_BRICK_SLANTED*

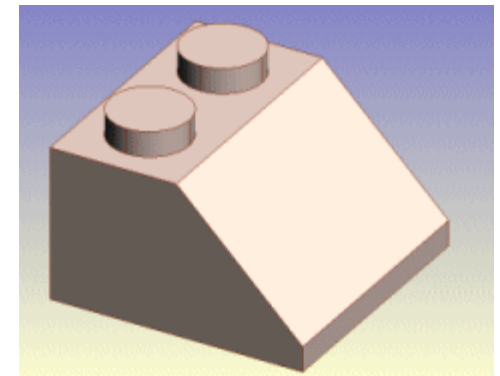


```
class  
  LEGO_BRICK_SLANTED
```

```
inherit  
  LEGO_BRICK  
  redefine  
    volume  
  end
```

The feature *volume* is going to be redefined (=changed). The feature *volume* comes from *LEGO_BRICK*

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



Class *LEGO_BRICK_WITH_HOLE*

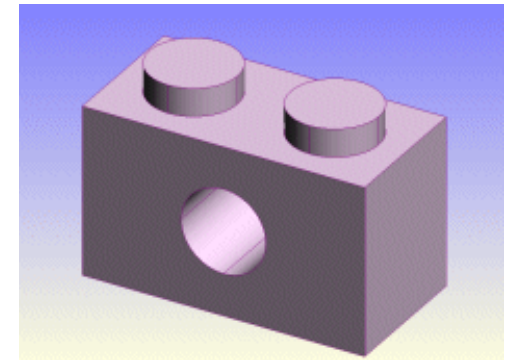


```
class  
  LEGO_BRICK_WITH_HOLE
```

```
inherit  
  LEGO_BRICK  
  redefine  
    volume  
  end
```

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

The feature *volume* is going to be redefined (=changed). The feature *volume* comes from *LEGO_BRICK*



Inheritance Notation

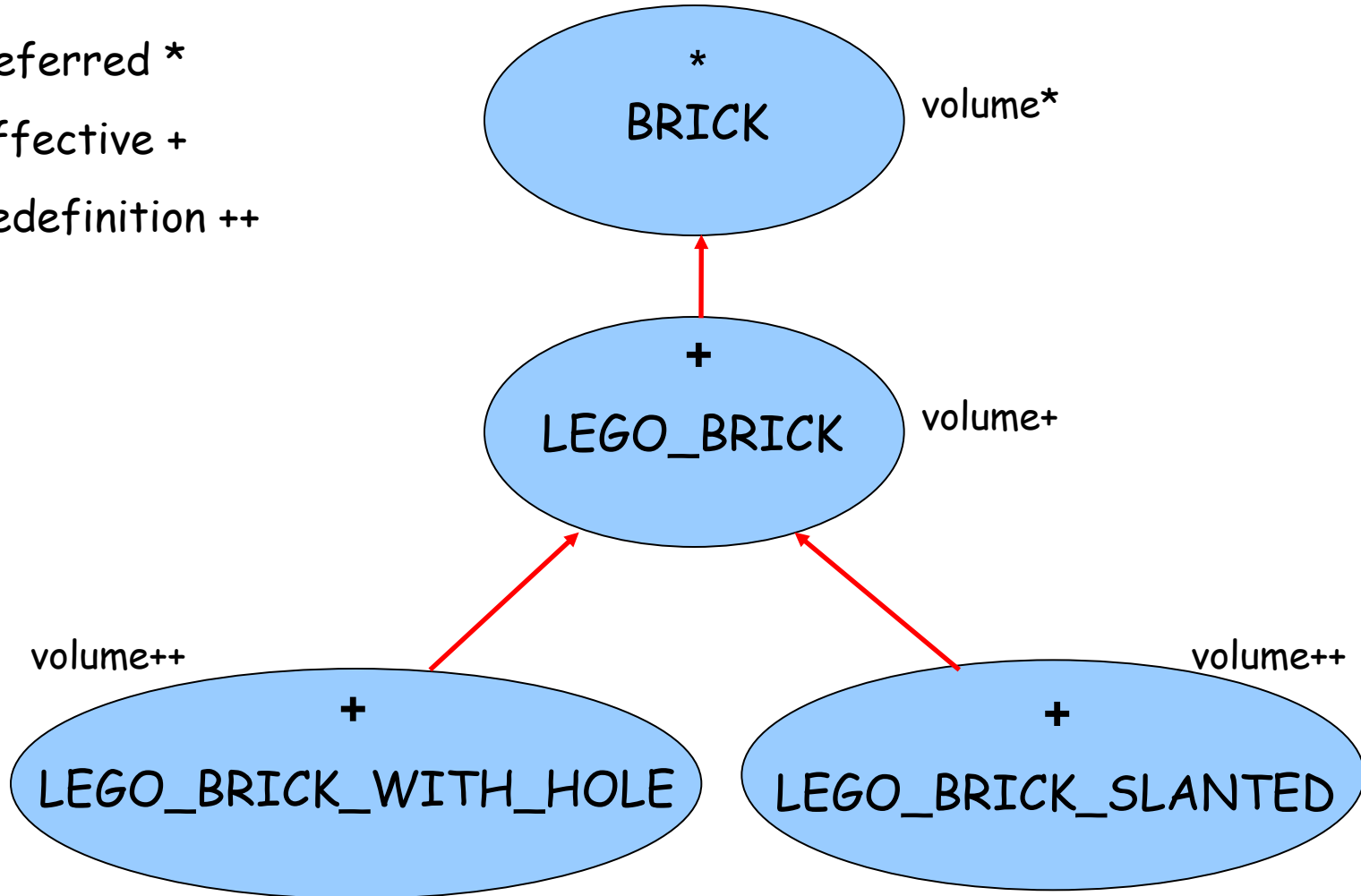


Notation:

Deferred *

Effective +

Redefinition ++



- Deferred
 - Deferred classes can have deferred features.
 - A class with at least one deferred feature must be declared as deferred.
 - A deferred feature does not have an implementation yet.
 - Deferred classes cannot be instantiated and hence cannot contain a create clause.



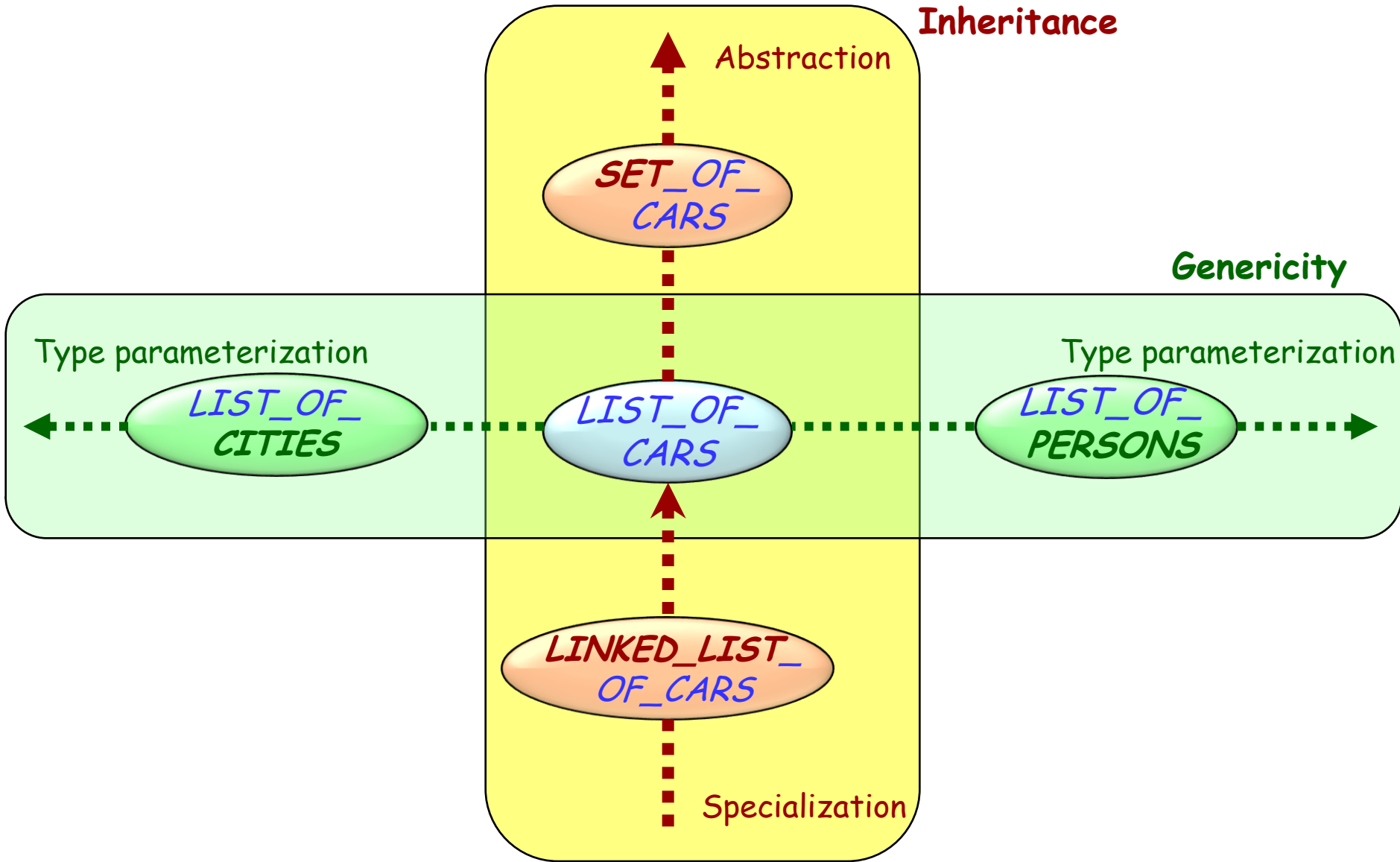
- Effective
 - Effective classes do not have deferred features (the "standard case").
 - Effective routines have an implementation of their feature body.

- If a feature was redefined, but you still wish to call the old one, use the **Precursor** keyword.

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



- Genericity lets you parameterize a class. The parameters are types. A single class text may be reused for many different types.



A generic list

Formal generic parameter

```
class LIST [G] feature  
  extend (x : G) ...  
  last : G ...  
end
```

To use the class: obtain a **generic derivation**, e.g.

Actual generic parameter

```
cities : LIST [CITY]
```



A generic list with constraints



class

STORAGE [G] > RESOURCE

inherit

LIST [G]

constrained generic parameter

feature

consume_all

do

from *start* **until** *after*

loop

item.consume

forth

end

end

end

The feature *item* is of type *G*. We cannot assume *consume*.

assume this.

Type-safe containers



- Using genericity you can provide an implementation of type safe containers. This helps avoiding object-tests.

x: ANIMAL

animal_list: LINKED_LIST[ANIMAL]

a_rock: MINERAL

animal_list.put(a_rock) -- Does this rock?