cc BY NC SA

Fondamenti della Programmazione: Metodi Evoluti

Prof. Enrico Nardelli

Lezione 3: Features

Definitions: class, instance, generating class

A **class** is the description of a set of possible run-time objects to which the same features are applicable

If an object *O* is one of the objects described by a class *C*:

- *O* is an **instance** of *C*
- *C* is the **generating class** of *O*

A class represents a category of things

An object represents one of these things

Objects and classes

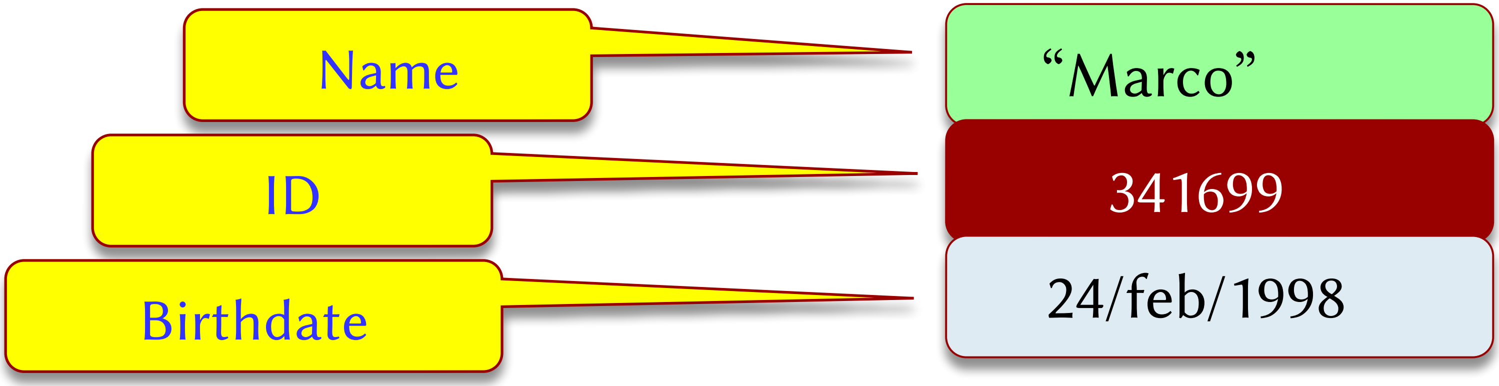
An object is a software machine storing a collection of data and allowing to access and modify them

Query

Command

- Objects may represent:
  - A city
  - A tram line
  - A route through the city
  - An element of the GUI such as a button
- Each object belongs to a certain class, defining the applicable operations, or features
- Examples:
  - The class of all cities
  - The class of all students
  - etc.

Two views of objects



Name

"Marco"

ID

341699

Birthdate

24/feb/1998

Two viewpoints:

- 1. An object has data, stored in memory.
- 2. An object is a machine offering operations (**features**)

The connection:

- The operations (2) allow other objects to access and modify the object's data (1)

Objects vs. classes

Classes exist only in the software text:

- Defined by class text
- Describe properties of associated instances

Objects exist only during execution:

- Visible in program text through names **denoting** run-time objects

Example: *Student_5*

Expressions and their types

At run time, every object has a type: its generating class. Examples:

- *STUDENT* for the object denoted by *Student_5*
- *INTEGER* for the object denoted by *Student_5.ID*

In the program text, every expression has a type. Examples:

- *STUDENT* for *Student_5*
- *INTEGER* for *Student_5.ID*

An object is a machine

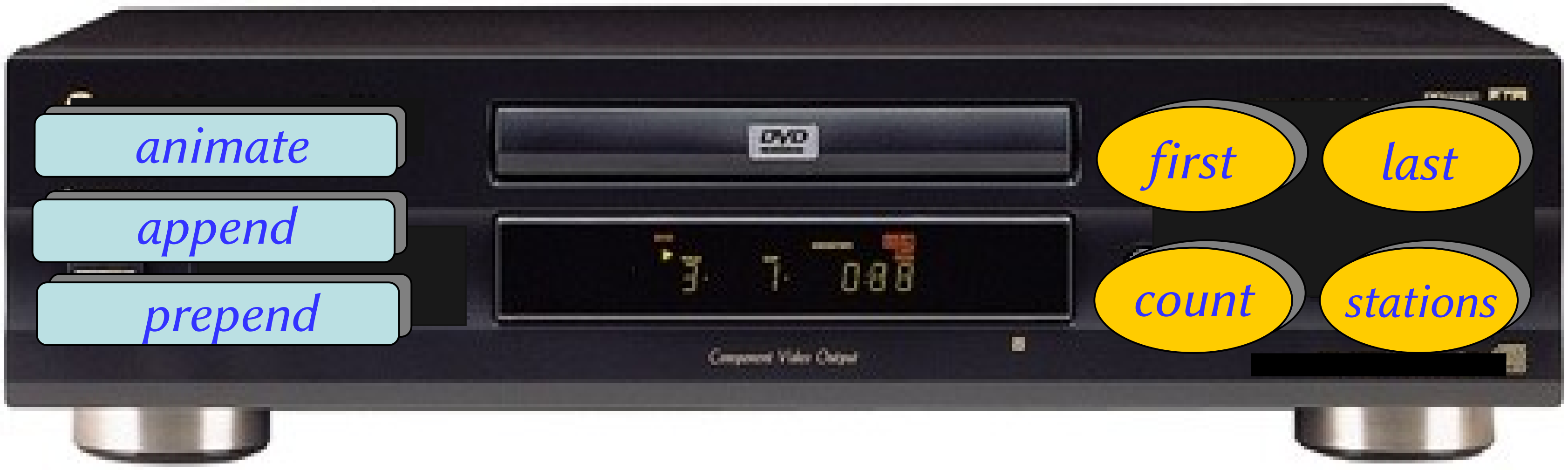
An executing program is a machine, made of smaller machines: objects

During execution there may be many objects (e.g. millions)

An object is a machine

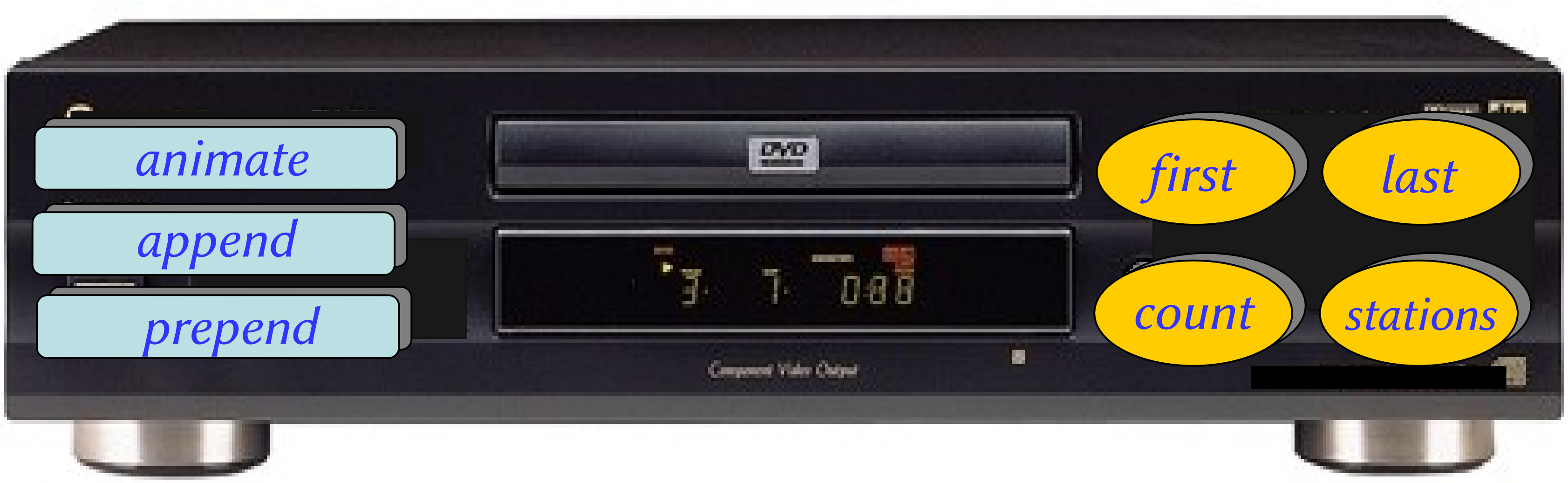
cc BY NC SA

A machine, hardware or software, is characterized by the operations ("features") users may apply



An object has an **interface**

cc BY NC SA



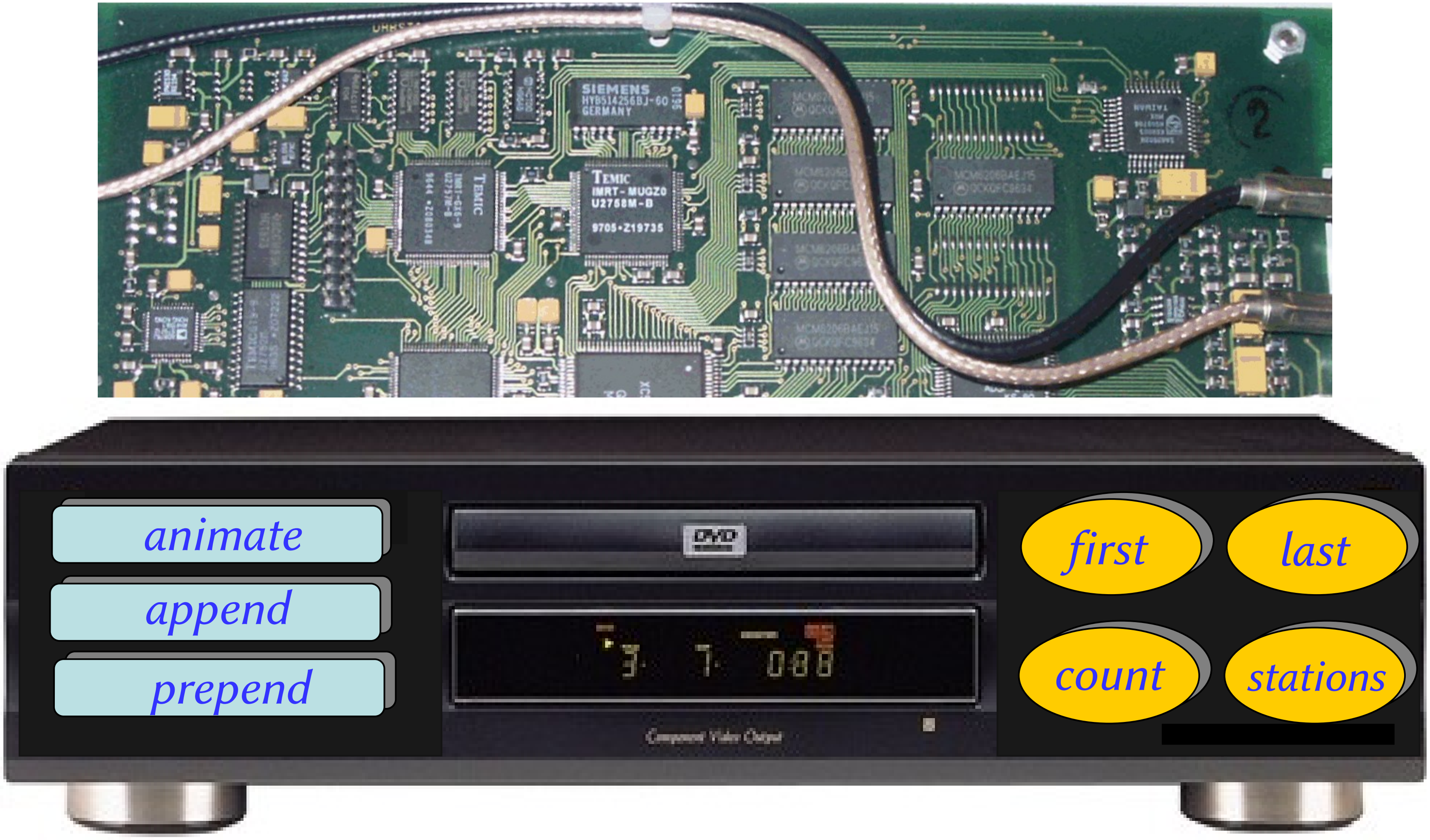
Interface: definition

An interface of a "software module" is the set of mechanisms enabling its "users" to use it.

"users" are usually called "clients"

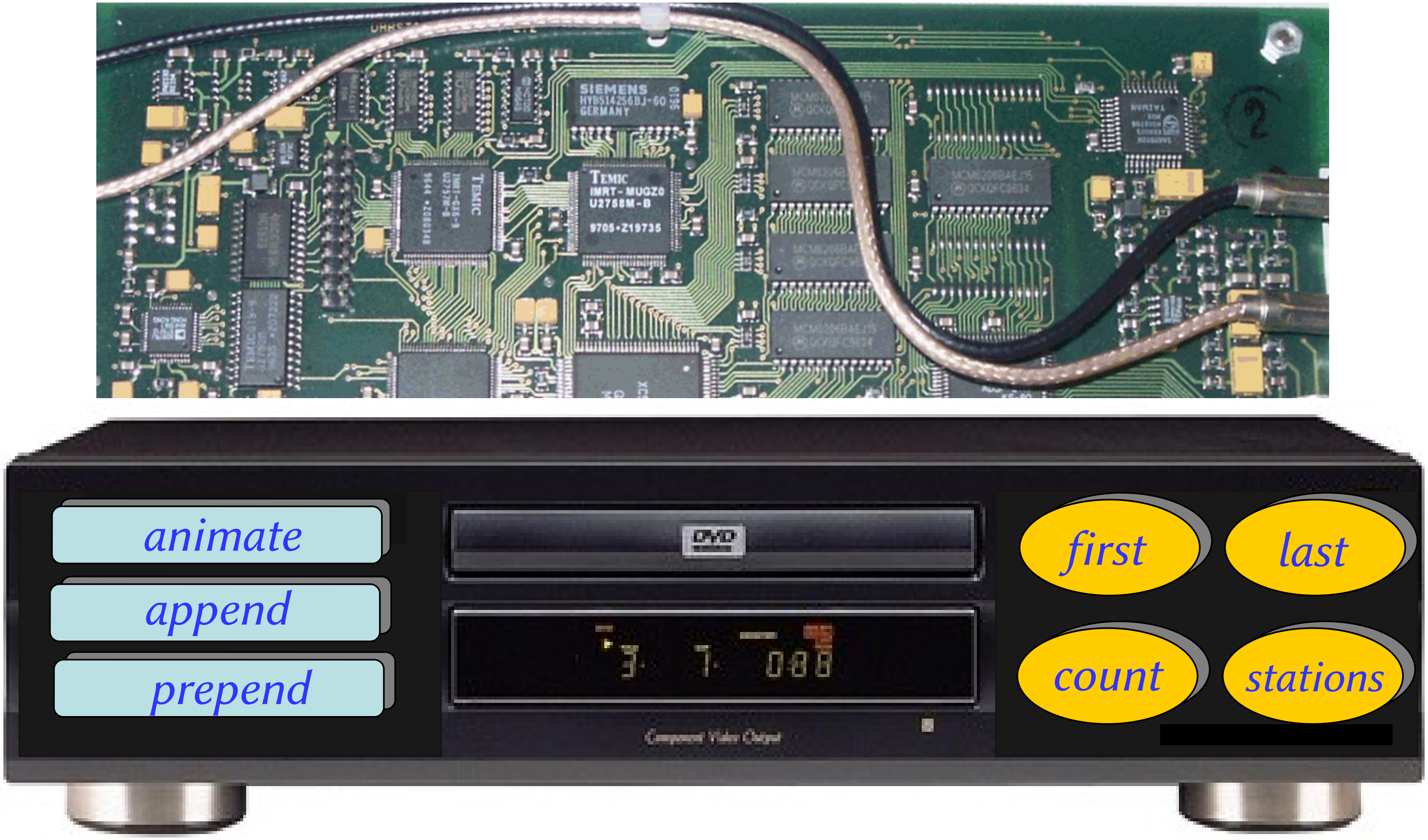
cc BY NC SA

An object has an implementation



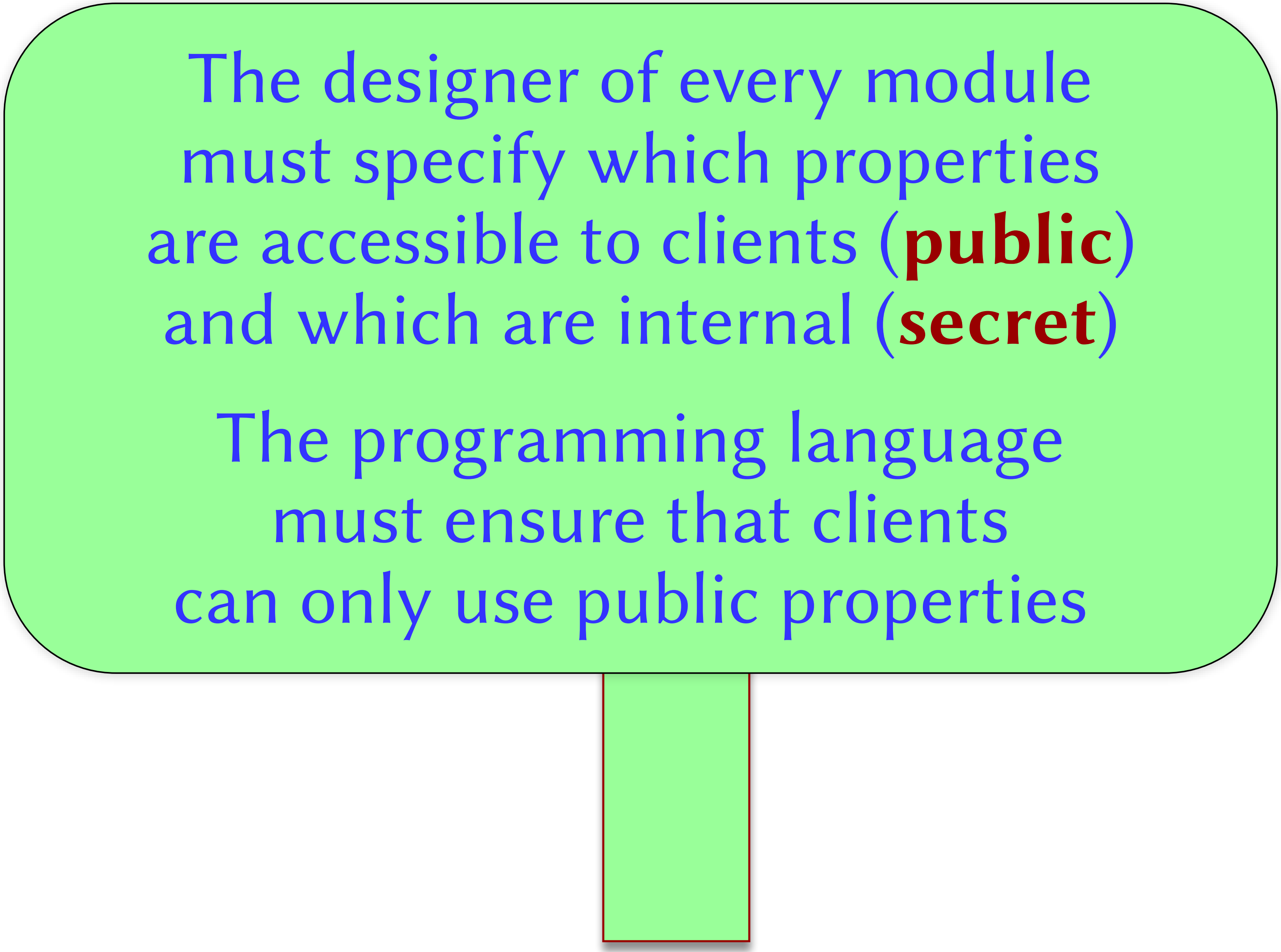
cc BY NC SA

Information hiding



BY NC SA

The information hiding principle



The designer of every module must specify which properties are accessible to clients (**public**) and which are internal (**secret**)

The programming language must ensure that clients can only use public properties

Client, supplier

Definitions

A client is a system of any kind — such as a software element, a non-software system, or a human user — that uses a software "module".

For its clients, the "module" is a supplier.

Features with arguments

your_object.your_feature (*some_argument*)

some_argument is a value that *your_feature* needs

Example: feature *show* must know what to show.

Same concept as function arguments in maths:

cos (*x*)

Features may have several arguments:

x.f (*a*, *b*, *c*, *d*) -- Separated by commas

In well written O-O software, most have 0 or 1 argument

Feature declaration vs. feature call

- You **declare** a feature when you write it into a class.

```
set_name (a_name: STRING)
                -- Set `name' to `a_name'.
        do
                name := a_name
        end
```

Within comments, use ` and ' to quote names of arguments and features. In such a way they will be taken into account by the automatic refactoring tools.

- You **call** a feature when you apply it to an object. The object is called the **target** of this feature call.

a_person.*set_name ("Peter")*

Features: commands and queries

Feature: an operation available on a certain class of objects

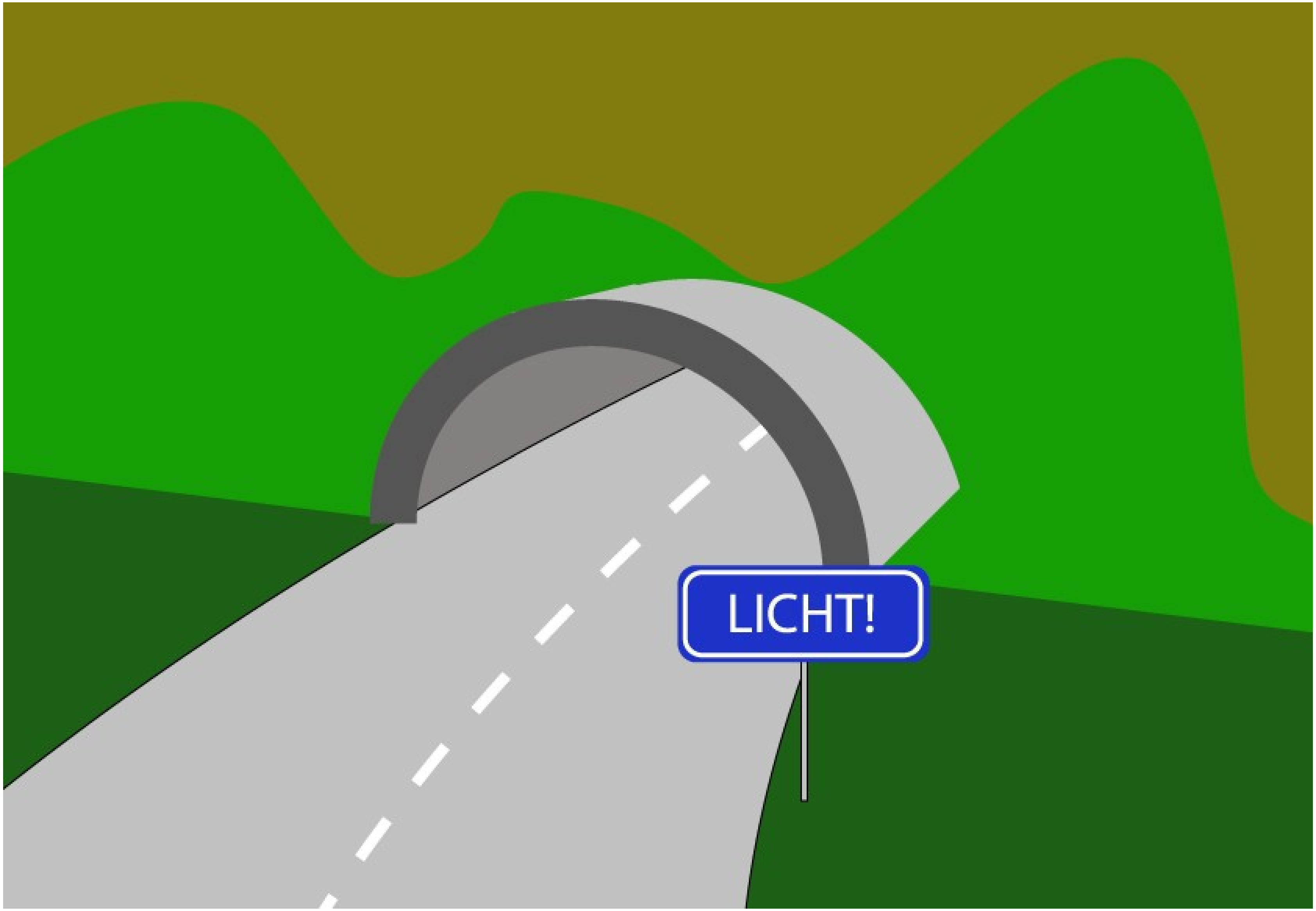
Three kinds:

- Command - a feature that may *modify* an object
- Query - a feature that *accesses* an object

- Creation procedure (seen later)

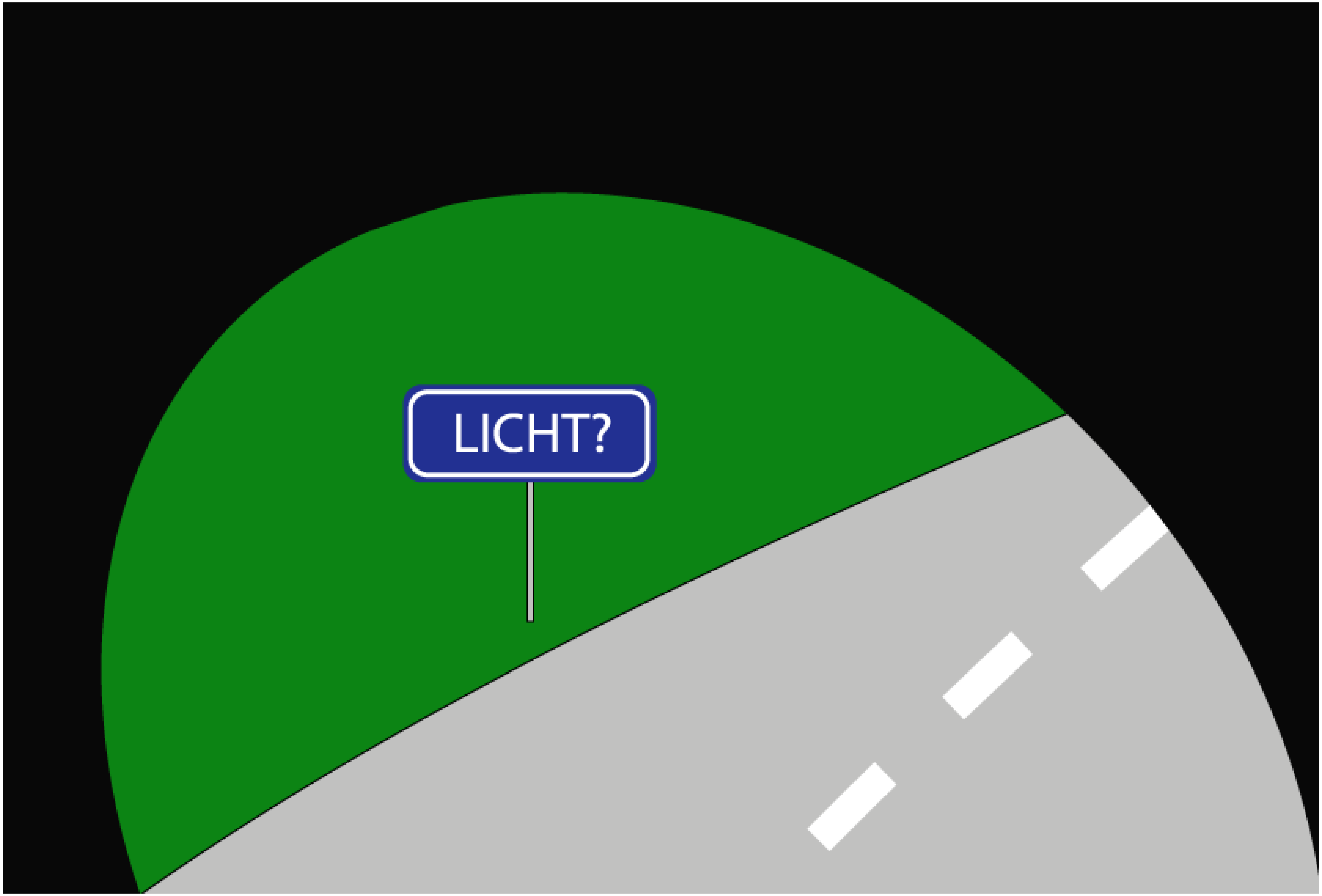
A command

CC BY NC SA



A query

BY NC SA



Commands

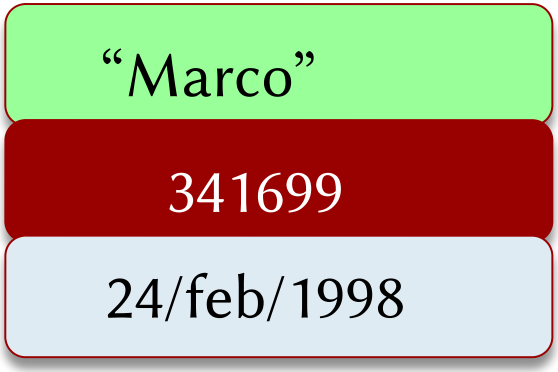
Goal: produce a change on an object, or several objects

Examples, for "**Student**" objects:

- Register an exam
- Add a course
- Modify the name

BY NC SA

Queries


“Marco”

341699

24/feb/1998

Goal: obtain properties of objects

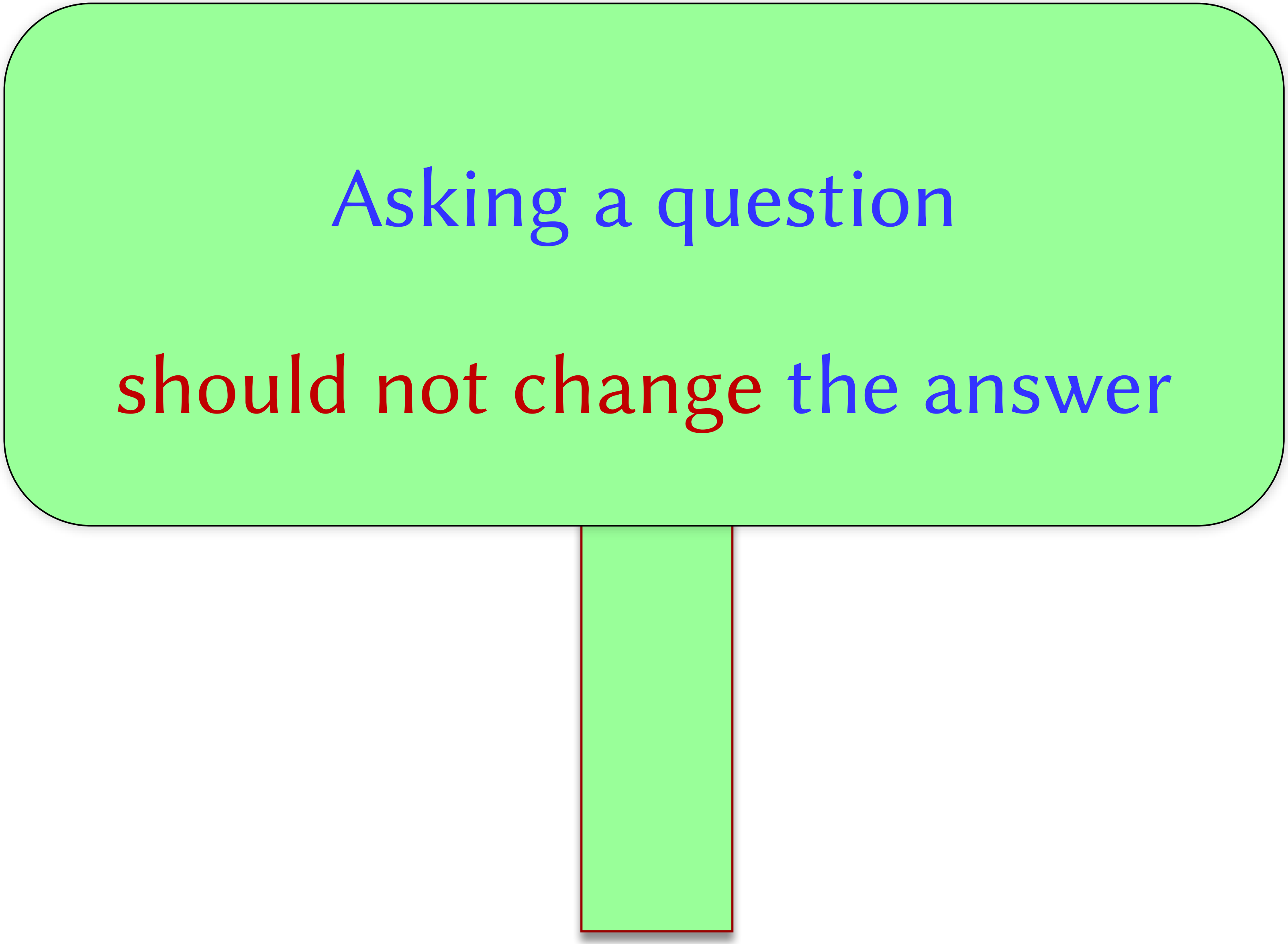
Should not modify the object,
or any other object

Examples, for a “**Student**” object :

- What is the name?
- What is the ID ?
- How many exams has she taken?
- Which courses is she following?

You may work with the return values of queries

The command-query separation principle



Asking a question

should not change the answer

Kinds of features: commands and queries

- Commands
  - Modify the state of objects
  - Do not have a return value
  - May or may not have arguments
  - Examples: register a student to a course, assign an id to a student, record the grade a student got in an exam
  - … other examples?
- Queries
  - Should not modify the state of objects
  - Do have a return value
  - May or may not have arguments
  - Examples: what is the age of a student? What is the id of a student? Is a student registered for a particular course?
  - … other examples?

Query or command?

Hands-On

BY NC SA

```
class DEMO

feature
```

command

```
    procedure_name (a1: T1; a2, a3: T2)
                        -- Comment
            do
                        ...
            end
```

- ➢ no result
- ➢ body

query

```
    function_name (a1: T1; a2, a3: T2): T3
                        -- Comment
            do
                        Result := ...
            end
```

- ➢ result
- ➢ body

Predefined variable denoting the result

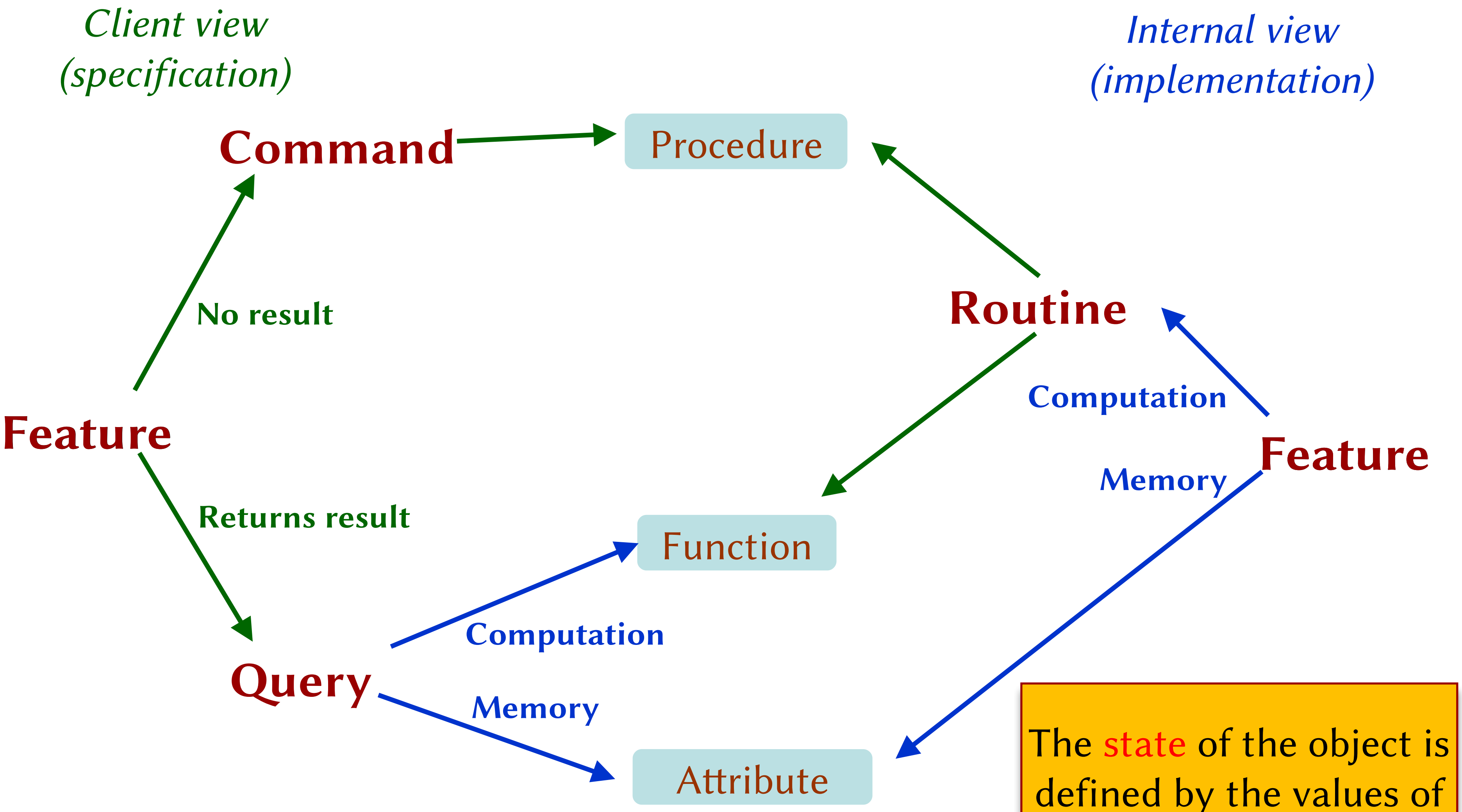
query

```
    attribute_name: T3
                        -- Comment
end
```

- ➢ result
- ➢ no body

cc BY NC SA

Features: the full story



Client view (specification)

Internal view (implementation)

Feature

No result

Command

Returns result

Query

Procedure

Function

Attribute

Computation

Memory

Routine

Computation

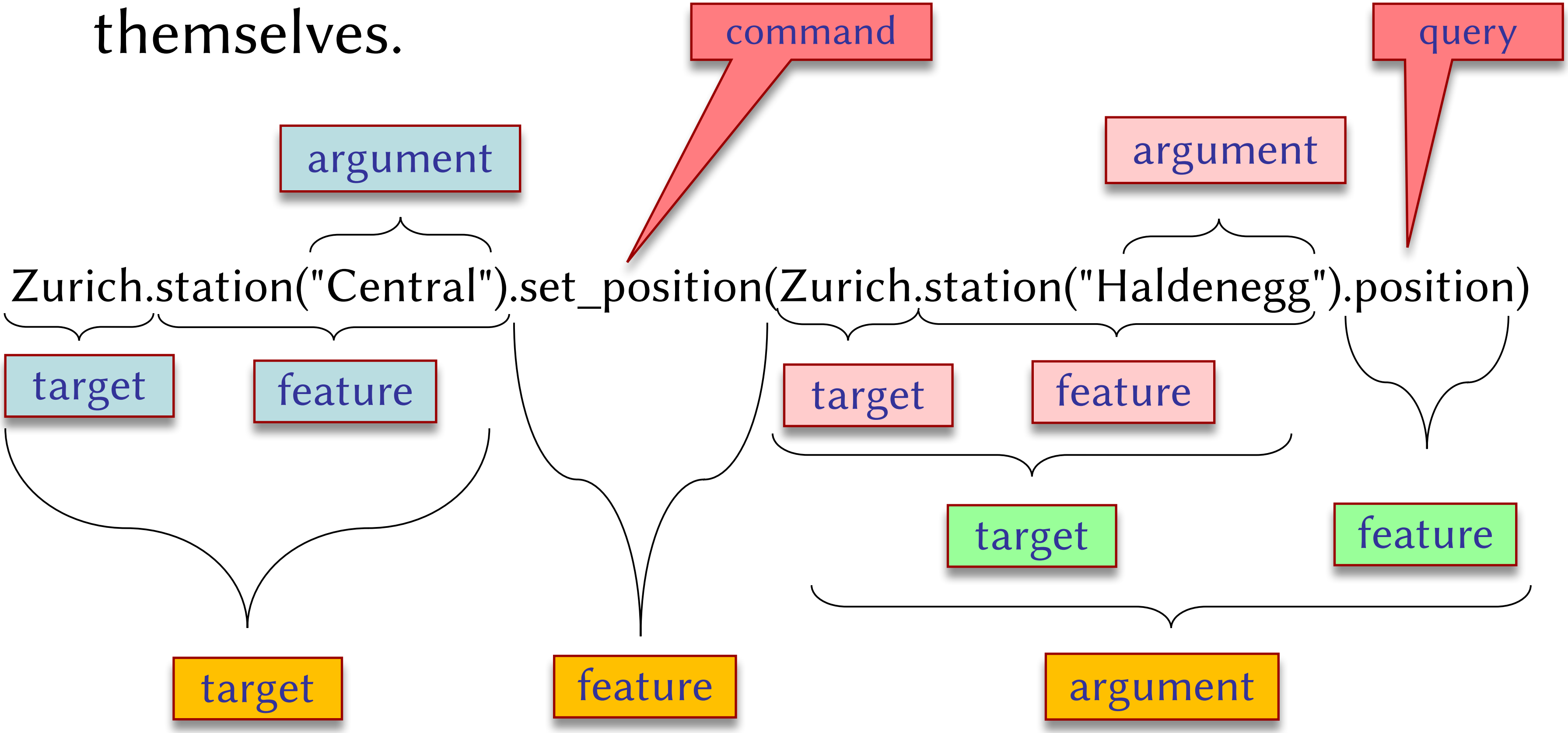
Memory

Feature

The state of the object is defined by the values of its attributes

General form of feature call instructions

- Targets and arguments can be feature calls themselves.



- Feature calls are interpreted left to right

Let’s do some work!

ESERCITAZIONE 1: parte prima (fino a p.8)

- convenzioni sui nomi

- compilazione

Current

- In object-oriented computation each feature call is performed on a certain object

  a_person.set_name ("Peter")

- The feature is executed by the called object
- What if during the execution of this feature you need to call another feature of the same object?
- How can you refer to the object which is executing one of its feature?
- We can access it using the predefined entity **Current**

  ***Current**.set_name ("Peter")*

- If one finds **Current** in a feature which is its type?
- It is the class where the feature is

Unqualified vs. qualified feature calls (1)

- All features have to be called on some **target** (always an **object**)

- A **qualified** feature call has an explicit target.
  a_person.set_name ("Peter")

- It is possible to omit writing the target in a feature call. Such a call is **unqualified**.
  set_name ("Peter")

- The implicit target is the current object, as if one had written
  ***Current**.set_name ("Peter")*

- However, if one writes **Current**, the call becomes qualified

cc BY NC SA

Qualified or unqualified?

Hands-On

Identify in the following whether feature calls are qualified or unqualified and their targets

1)	$\boxed{x}.y$	qualified
2)	x	unqualified
3)	$f\,(x.a)$	unqualified
4)	$\boxed{x.y}.z$	qualified
5)	$x\,(y.f\,(a.b))$	unqualified
6)	$\boxed{f\,(x.a)}.y\,(b)$	qualified
7)	$\boxed{\textbf{Current}}\ .x$	qualified

Result

- Inside every function you can use the predefined local variable **Result** (you needn’t and shouldn’t declare it)
- The return value of a function is whatever value the **Result** variable has at the end of the function execution
- **Result** (as well as regular local variables) is initialized, at the beginning of routine’s body, with the default value of its type
- Every regular local variable is declared with some type; and what is the type of **Result**?
- It's the function return type!

Compilation error?

Hands-On

```
class PERSON
feature
        …
        exchange_names (other: PERSON)
                do
                        Result := other.name
                        other.set_name (name)
                        set_name (Result.name)
                end

        name_with_semicolon: STRING
                do
                        create Result.make_from_string (name)
                        Result.append(‘;’)
                        print (Result)
                end
end
```

Error: Result can not be used in a procedure

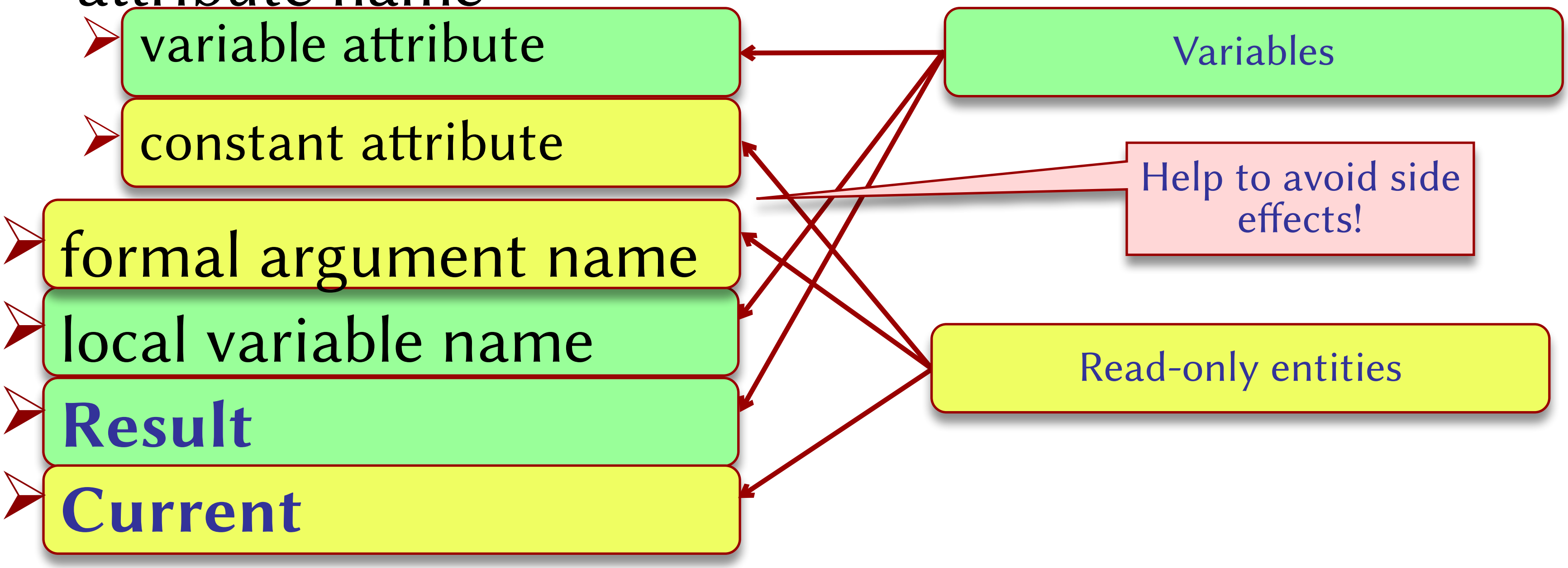
This is the mechanism to bring object to life (to be seen later).

BY NC SA

Entity: the final definition

An entity in program text is a “name” that *directly* denotes an object. More precisely: it is one of

- attribute name
  - variable attribute
  - constant attribute
- formal argument name
- local variable name
- **Result**
- **Current**



Variables

Help to avoid side effects!

Read-only entities

Only a variable can be used in a creation instruction and in the left part of an assignment

Static view

- A program consists of a set of classes.

- Features are declared in classes. They define operations on objects created from classes.
  - Queries answer questions. The answer is provided in a variable called Result.
  - Commands execute actions. They do not provide any result, so there is no variable called Result that we can use.

- Another name for a class is type.
- Class and Type are not exactly the same, but they are close enough for now, and we will learn the difference later on.

Declaring the type of an object

- The type of any object you use in your program must be declared somewhere.
- Where can such declarations appear in a program?
  - in feature declarations
    - formal argument types
    - return type for queries
  - in the **local** clauses of routines

This is where you declare any objects that only the routine needs and knows.

Declaring the type of an object

```
class DEMO
feature
    procedure_name (a1: T1; a2, a3: T2)
                    -- Comment
            local
                    l1: T3
            do
                    …
            end

    function_name (a1: T1; a2, a3: T2): T3
                    -- Comment
            do
                    …
            end

    attribute_name: T3
                    -- Comment
end
```

formal argument type

formal argument type

local variable type

formal argument type

return type

formal argument type

return type

Dynamic view

- When the program is being executed (at “runtime”) we have a set of objects (instances) created from the classes (types).

- The creation of an object implies that a piece of memory is allocated in the computer to represent the object itself.

- Objects interact with each other by calling features on each other.

Static view vs. dynamic view

- Queries (attributes and functions) have a result type. When **executing** the query, you get an object of that type.

- Routines have **formal arguments** of certain types. During the **execution** you pass objects of the same (or compatible) type as **actual arguments** to a routine call.

- Local variables are declared in their own section, associating names with types. Invoking a local returns the current object of that type referred to by that variable.

Let’s do some work!

ESERCITAZIONE 1: parte seconda (da p.9)

- facciamo il compilatore

The scope of names

Attributes:

- are declared anywhere inside a **feature** clause, but not inside a feature declaration
- are visible anywhere inside the class

Formal arguments:

- are declared after the feature name
- are only visible inside the feature body and its contracts

Local variables:

- are declared in a **local** clause inside the feature declaration
- are only visible inside the feature body (are **not** visible in its contracts!)

Is everything an object oriented call?

some_target.some_feature (some_arguments)

For example:

Paris.display
Line6.extend (Station_Parade_Platz)

x := a + b ???????

BY NC SA

Operator aliases for features

```
expanded class INTEGER  feature

        plus alias "+" (other : INTEGER): INTEGER
                        -- Sum with other
                external "built_in" end

        minus alias "-" (other : INTEGER): INTEGER
                        -- Decrement by other
                external "built_in" end

        times alias "*" (other : INTEGER): INTEGER
                        -- Product by other
                external "built_in" end

        opposite alias "-" : INTEGER
                        -- Unary minus
                external "built_in" end
        …
end
```

Features with one argument allow **alias** for infix notation

Features with zero arguments allow **alias** for prefix notation

Same string can be **alias** for different features ONLY if they have different number of arguments

Calls such as *i.plus (j)* can now be written ***i + j***
and calls such as *i.opposite* as ***-i***