Fondamenti della Programmazione: Metodi Evoluti

Prof. Enrico Nardelli

Lezione 2: Oggetti

Programming languages

The programming language is the notation that defines the syntax and semantics of programs

There are many programming languages, some “general”, some “specialized”

Programming languages are artificial notations, designed for a specific purpose (programming).

Our programming language is Eiffel, an object-oriented language

Object technology

We work with objects

Our style of programming:

Object-Oriented programming

Abbreviation: O-O

More generally, “Object Technology”: includes O-O *databases*, O-O *analysis*, O-O *design*...

Software execution is made of operations on objects — feature calls: every operation (feature) applies to an object (the target of the call)

your_object•*your_feature*

Object technology

Source: Simula 67 language, Oslo, mid-sixties

Spread *very* slowly in seventies

Smalltalk (Xerox PARC, 1970s) made O-O hip by combining it with visual technologies

First OOPSLA in 1986 revealed O-O to the masses

Spread quickly in 1990s through

- O-O languages: Objective C, C++, Eiffel, Java, C#...
- O-O tools, O-O databases, O-O analysis...

Largely accepted today

Non O-O approaches are:

"procedural", "functional", "logic".

About Eiffel

First version 1985, constantly refined and improved since

Focus: software quality, especially reliability, extendibility, reusability. Emphasizes simplicity

Based on concepts of “Design by Contract”

Used for mission-critical projects in industry

Several implementations, including EiffelStudio from Eiffel Software (the one we use), available open-source

International standard: ECMA and ISO (International Standards Organization), 2006

Some Eiffel-based projects

Axa Rosenberg
Investment management: from $2 billion to >$100 billion
2 million lines

Chicago Board of Trade
Price reporting system
Eiffel + CORBA + Solaris + Windows + …

Xontech (for Boeing)
Large-scale simulations of missile defense

Swedish social security: accident reporting & management

etc.

So, why use Eiffel?

- Simple, clean O-O model
- Enables you to focus on concepts, not language
- Little language "baggage"
- Development environment (EiffelStudio)
- Portability: Windows / Linux / VMS & others
- Realism: not an "academic" language

Prepares you to learn other O-O languages if you need to, e.g. C++, Java, C#

Simplicity

1st Java program

```
class First {
        public static void main(String args[])
        {
                System.out.println("Hello World!");
        }
}
```

1st Eiffel program

```
class
        APPLICATION
create
        make
feature
        make
                do
                        Io.put(“Hello Eiffel world!”)
                end
end
```

Classes and objects

- The main concept in Object-Oriented programming is the concept of Class.

- Classes are pieces of software code meant to model concepts, e.g. “student”, “course”, “university”.

- Several classes make up a program in source code form.

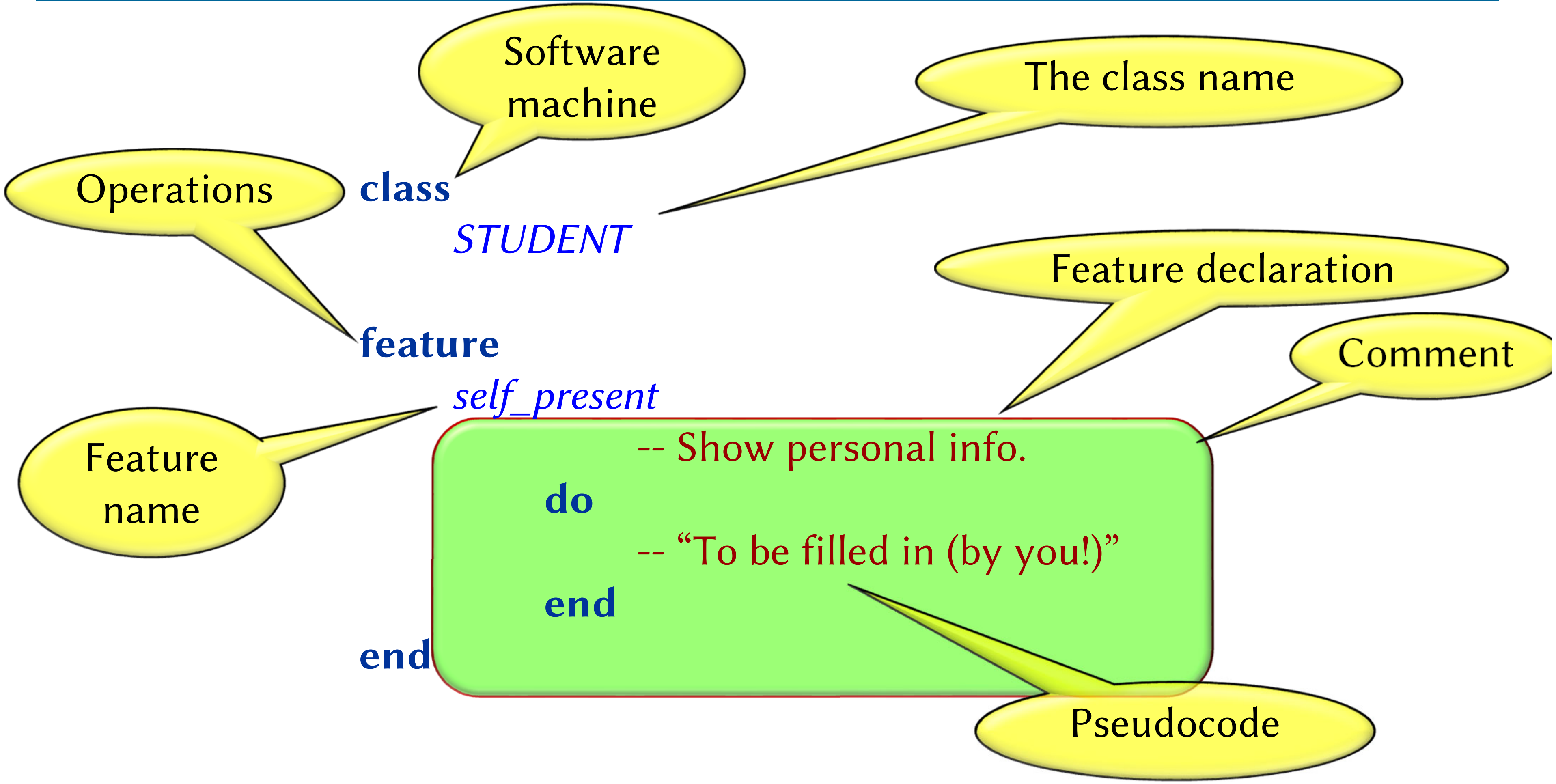
- **Objects** are particular occurrences (“instances”) of concepts (classes), e.g. “student Bill” or “student Lisa”.

- A class *STUDENT* may have zero or more instances.

Classes and objects (continued)

- Classes are like templates (or molds) defining status and operations applicable to their instances.
- A sample class *STUDENT* can define:
  - A student's status: id, name and birthday
  - Operations ("**features**") applicable to all students: subscribe to a course, register for an exam.
- Each instance (object) of class *STUDENT* will store a student's name, id and birthday and will be able to execute operations such as subscribe to a course and register for an exam.
- Only the operations defined in a class can be applied to its instances.

A class text

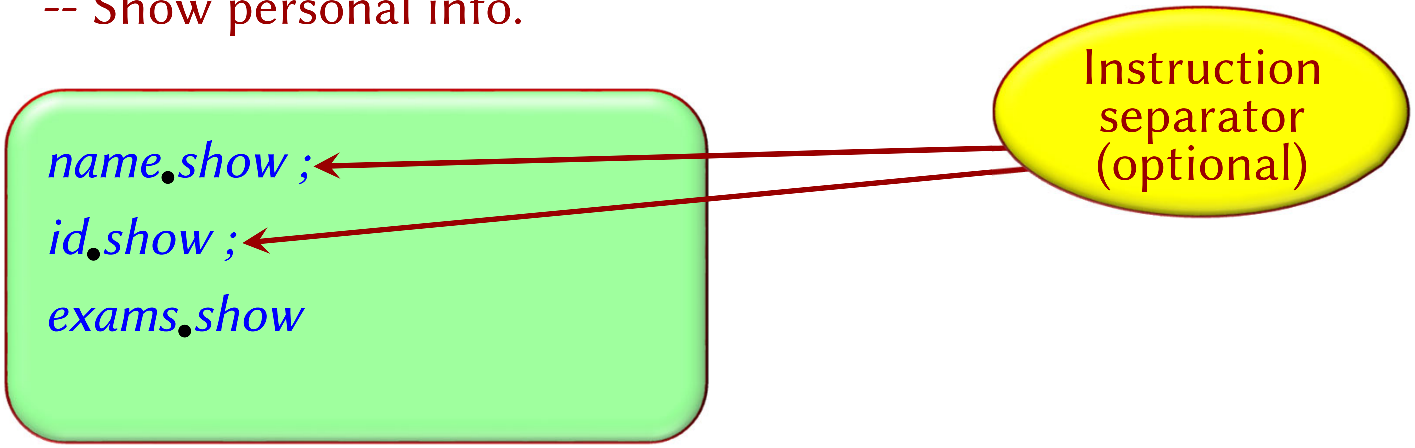


```
class
        STUDENT

feature
        self_present
                        -- Show personal info.
                do
                        -- "To be filled in (by you!)"
                end
end
```

Keywords have a special role: **class**, **inherit**, **feature**, **do**, **end**.

Filling in the feature body

```
--
class
    STUDENT


feature
    self_present
            -- Show personal info.
        do
            name.show ;
            id.show ;
            exams.show

        end
end
```

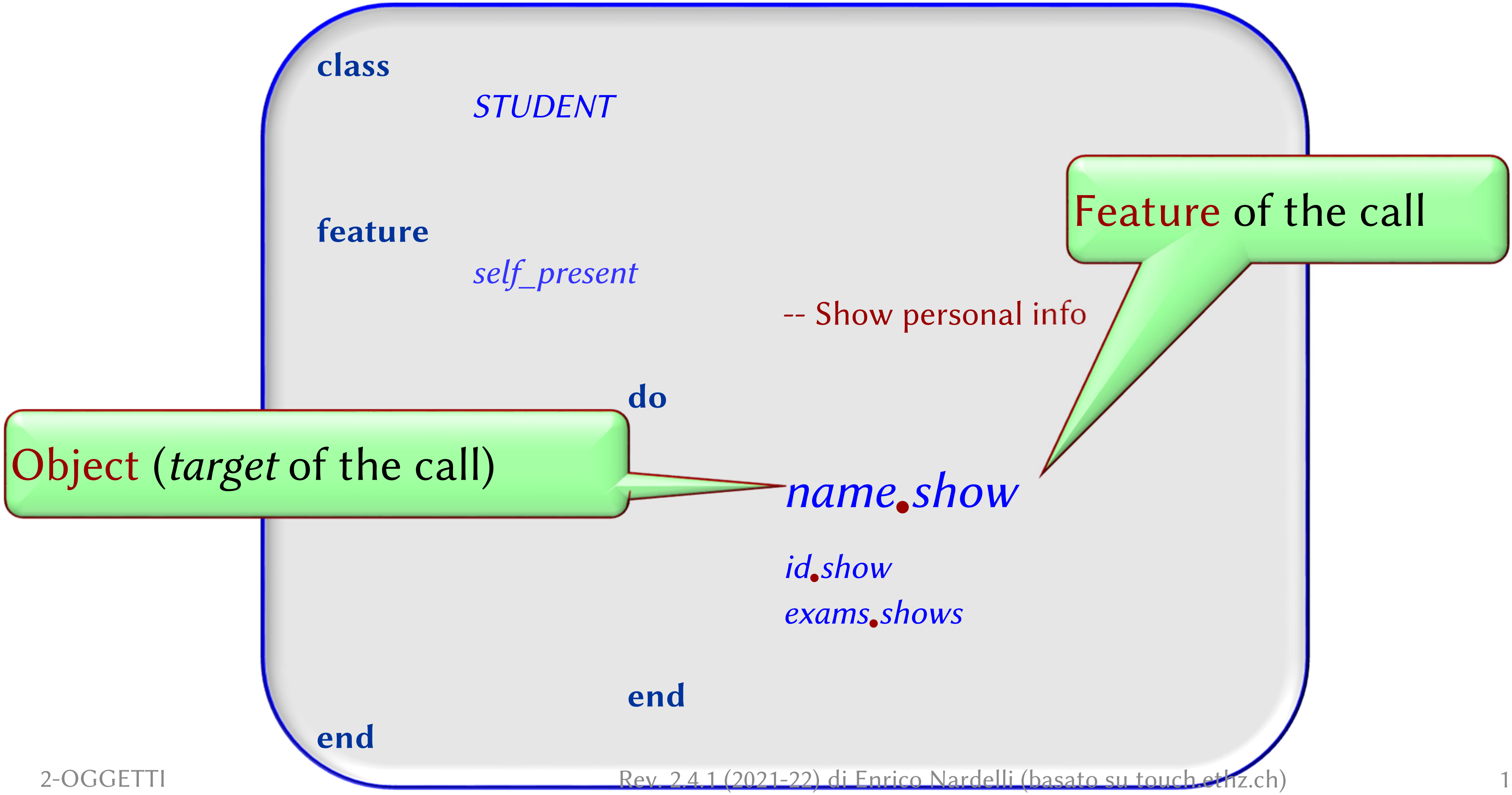


Instruction separator (optional)

Feature call

The fundamental mechanism of program execution: apply a “feature” to an “object”

Basic form: *your_object.your_feature*



```
class
        STUDENT

feature
        self_present
                        -- Show personal info
                do
                        name.show
                        id.show
                        exams.shows
                end
end
```

Object (*target* of the call)

Feature of the call

Program formatting and style rules

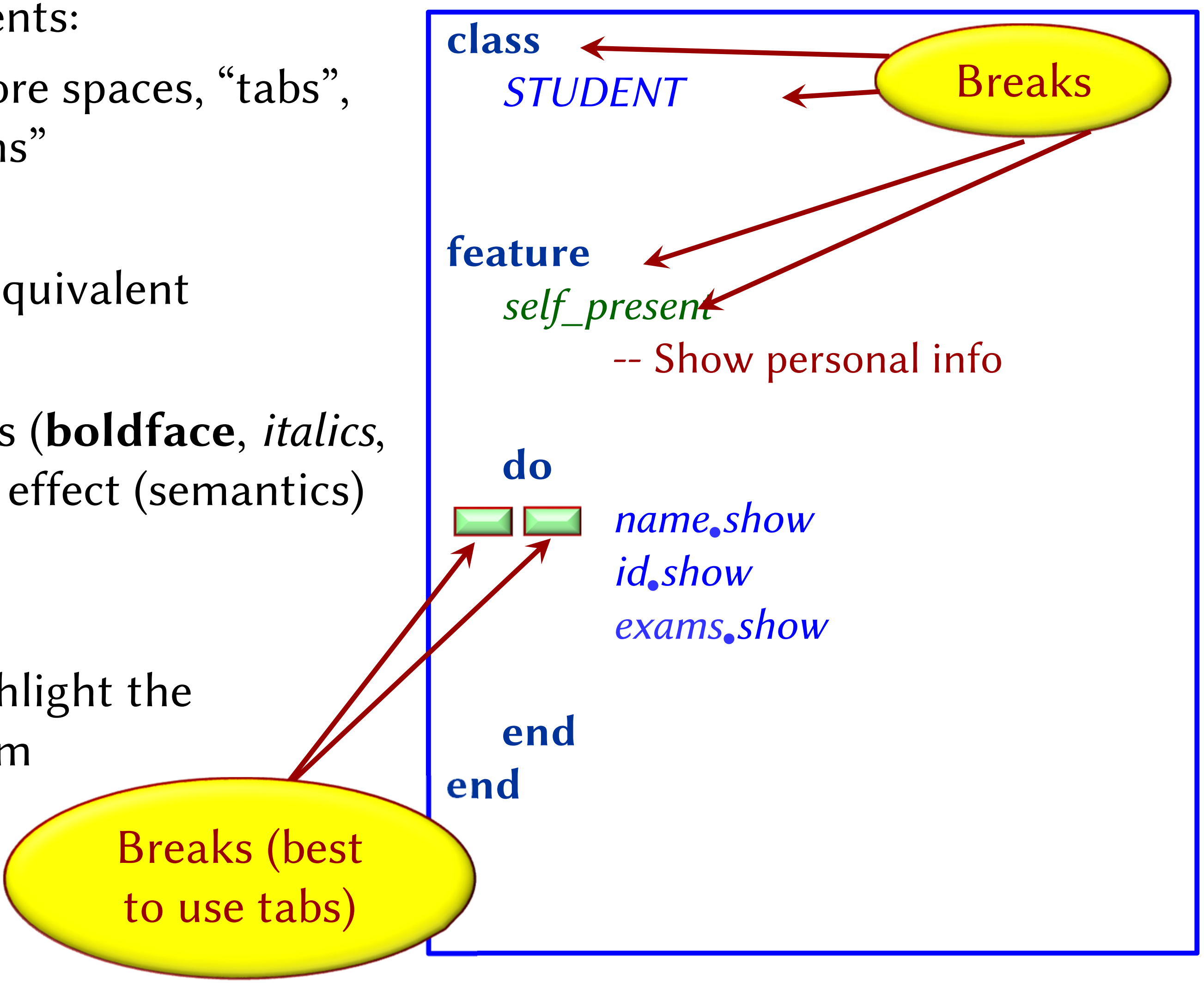
Between adjacent elements:

> **break**: one or more spaces, “tabs”, “carriage returns”

All kinds of break are equivalent

Typographical variations (**boldface**, *italics*, colors) do not affect the effect (semantics) of programs

Use **indentation** to highlight the **structure** of the program



```
class
    STUDENT


feature
    self_present
            -- Show personal info

    do
        name.show
        id.show
        exams.show


    end
end
```

Another convention

For long names, use underscores "_"

WORKING_STUDENTS
self_present

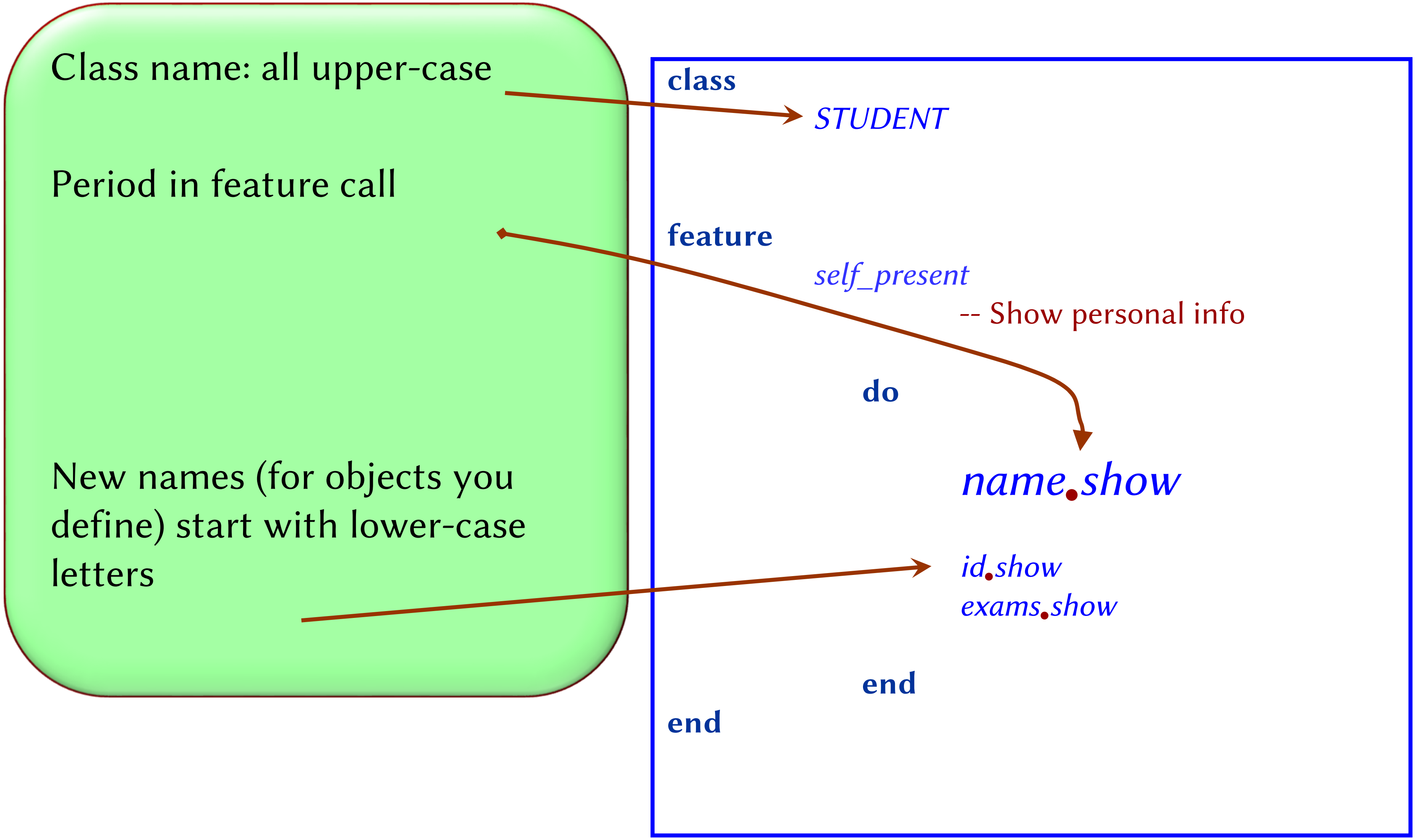
We do not use "CamelCase":

AShortButHardToDeCipherName

but underscores (sometimes called "Pascal_case"):

A_significantly_longer_but_still_perfectly_clear_name

More style rules



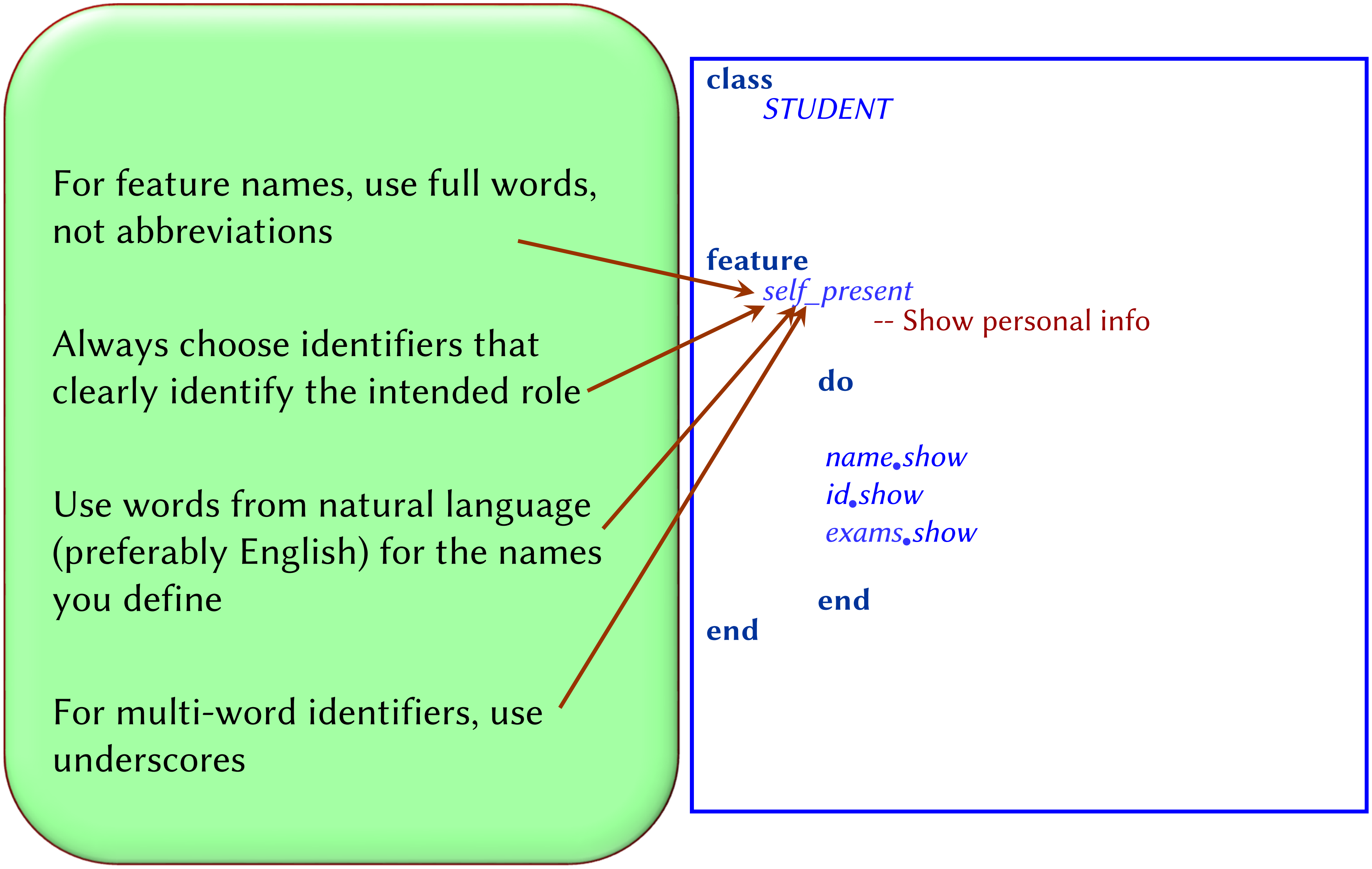
Class name: all upper-case

Period in feature call

New names (for objects you define) start with lower-case letters

```
class
        STUDENT

feature
        self_present
                -- Show personal info

            do
                name.show

                id.show
                exams.show

            end
end
```

Even more style rules



For feature names, use full words, not abbreviations

Always choose identifiers that clearly identify the intended role

Use words from natural language (preferably English) for the names you define

For multi-word identifiers, use underscores

```
class
    STUDENT


feature
    self_present
            -- Show personal info

        do

         name.show
         id.show
         exams.show

        end
end
```

A final style rule

Write one instruction per line

Omit semicolons

You may write more than one instruction on the same line

If you think it is needed (e.g. in a paper report) then use a semicolon

$$f(x)\ \boxed{;}\ (y)$$

Entities

An entity is a name in the program that denotes possible run-time values. There are two kinds of them:

Some are **constant**

Others are **variable**:

- Attributes ("general" visibility)
- Local variables (limited visibility)

- The technical term for visibility is "scope"

Constants

A **constant** entity is specified by providing its value (called “manifest value”) together with its type (name’s first letter is capitalized)

```
First_id: INTEGER = 1000
Map_title: STRING = “Plan of the metro"
Inches_to_centimeters: REAL = 2.54
```

Local entities

A **local** variable is specified inside a feature declaration before its body (the **do … end** part)

```
feature
    swap (a, b : ITEM)
        -- Swap objects referred by `a' and `b'
        local
            temp : ITEM
        do
            temp := a
            a := b
            b := temp
        end
```

A **local** variable cannot use a feature name of the same class or a formal parameter name of the same feature

Lexical rule for entity identifiers

Identifiers

An identifier starts with a letter, followed by zero or more characters, each of which may be:

- A letter.
- A digit (0 to 9).
- An underscore character "_".

You may choose your own identifiers as you please, excluding keywords

Three basic distinctions

Syntax / Semantics

Instruction / Expression

Command / Query

Syntax and semantics

The **syntax** of a program is the structure and form of its text

The **semantics** of a program is the set of properties of its potential executions

Syntax is the way you write a program: characters grouped into words grouped into bigger structures

Semantics is the effect you expect from this program

Instructions and expressions

An expression, e.g. *first_student.name*, is not a value but denotes future run-time values

An instruction, e.g. *first_student.show*, denotes an operation to be executed at run time

Definitions

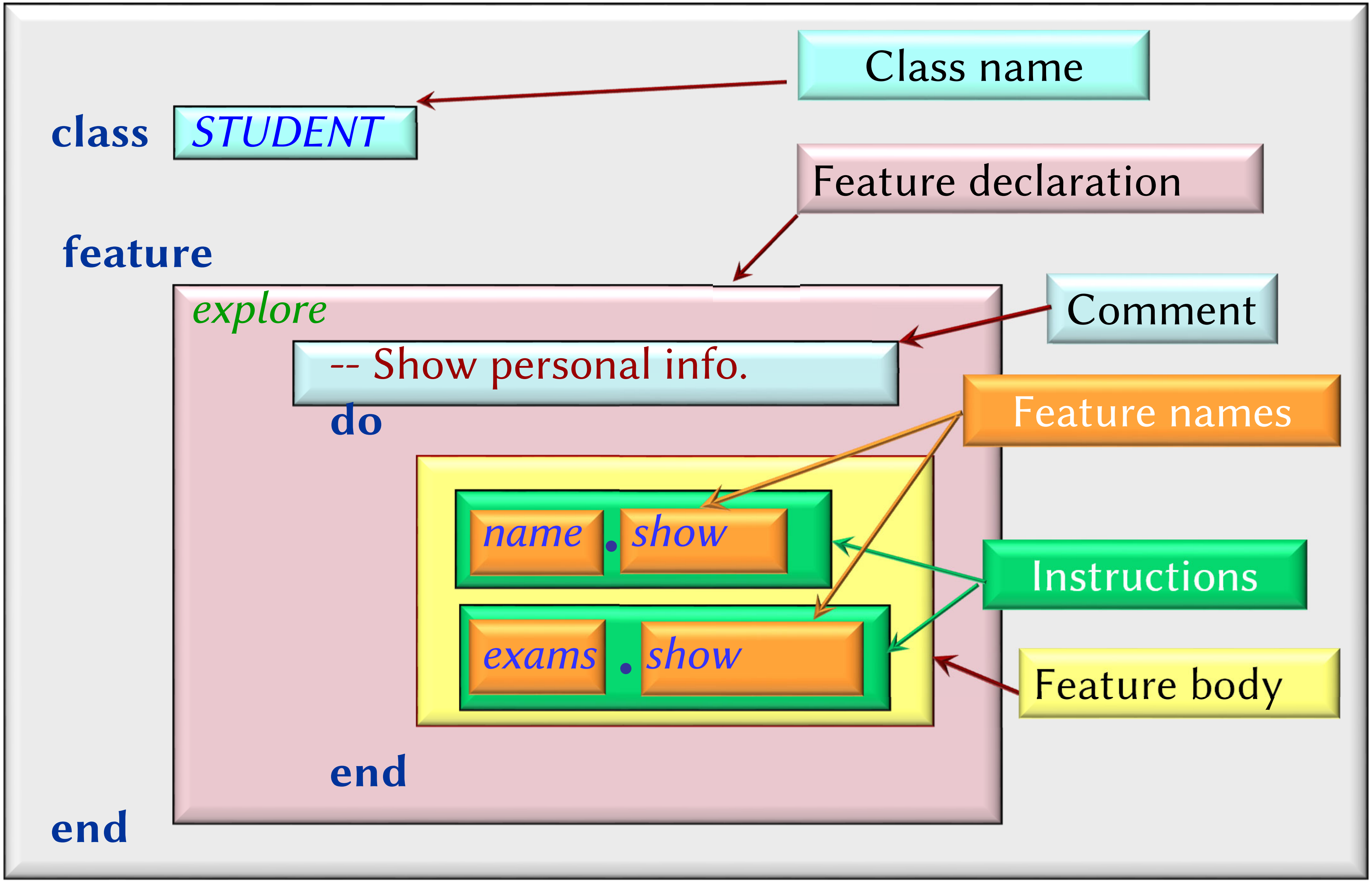
In program texts:

- An **instruction** denotes a basic operation to be performed during the program's execution.
- An **expression** denotes a value used by an instruction for its execution.

Syntax and semantics

	Syntax	Semantics
Prescriptive	Instruction	Command
Descriptive	Expression	Query Value

Syntax structure of a class



```
class STUDENT

 feature
      explore
           -- Show personal info.
           do
                name.show
                exams.show
           end
end
```

Class name

Feature declaration

Comment

Feature names

Instructions

Feature body

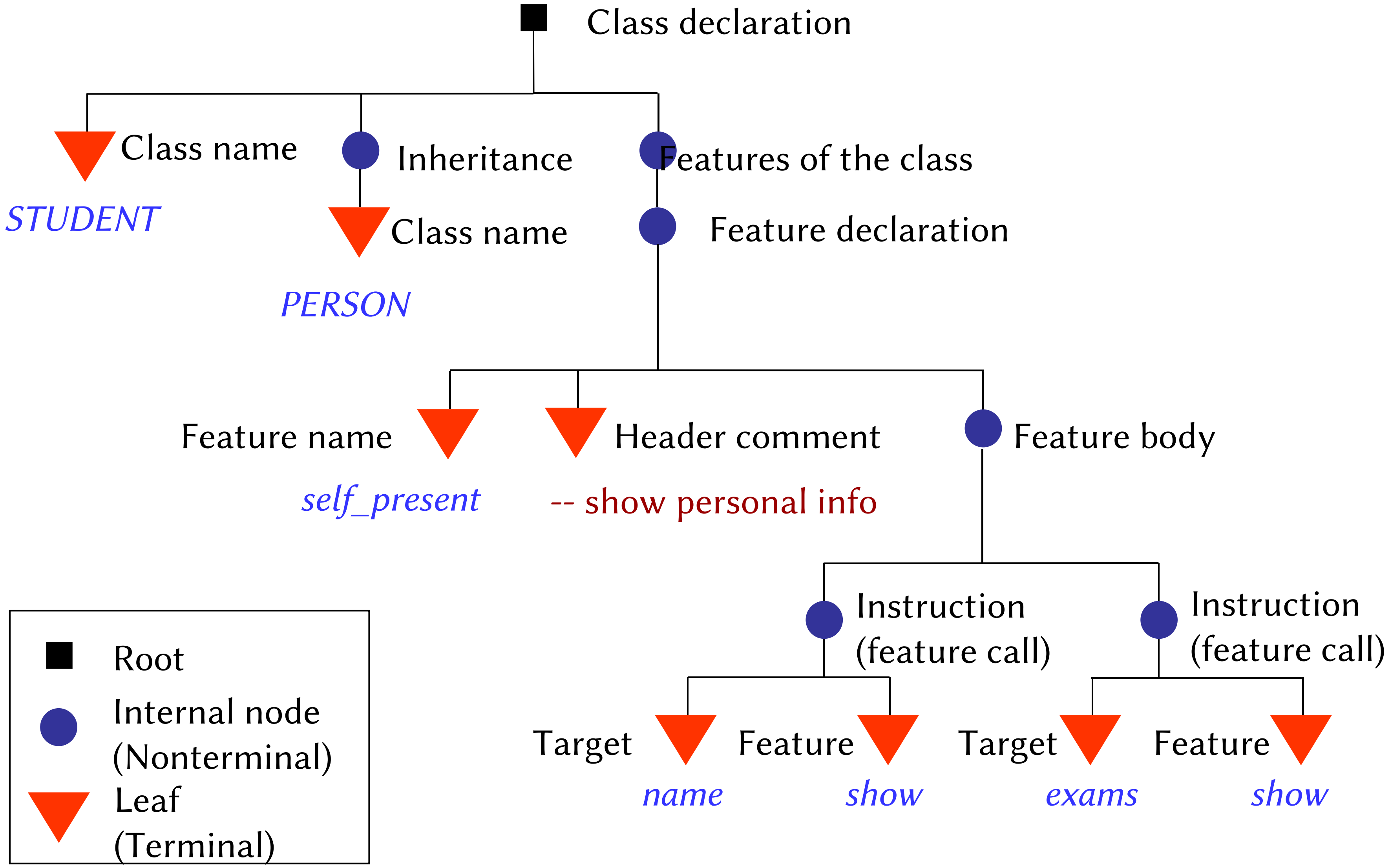
The lower level: lexical structure

The basic elements of a program text are tokens:

- Terminals
  - Identifiers: names chosen by the programmer, e.g. *Paris* or *display*
  - Constants: self-explanatory values, e.g 34
- Keywords, e.g. **class**
- Special symbols: colon (:), "**.**" of feature calls

Tokens define the lexical structure of the language

Other representation: abstract syntax tree

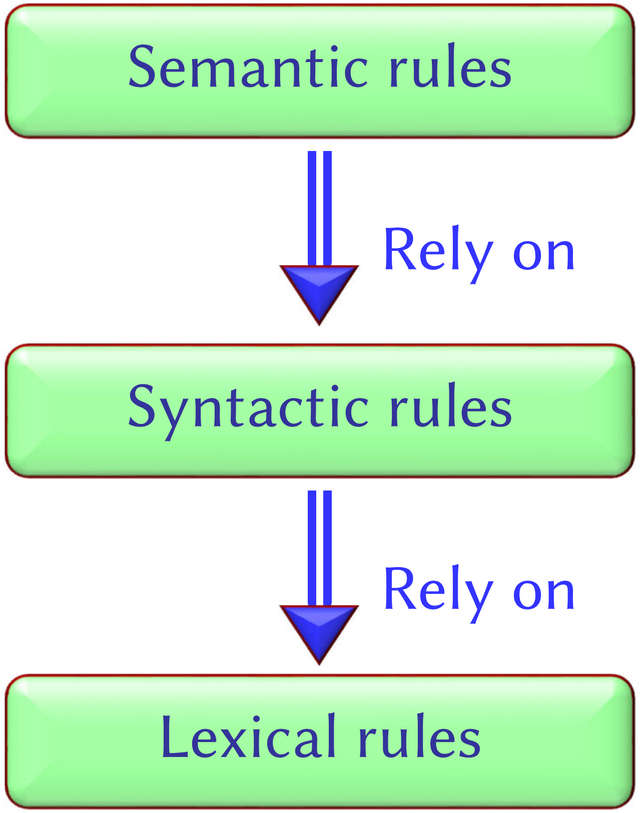


Three levels of description

Lexical rules define how to make up tokens out of characters

Syntax rules define how to make up specimens out of tokens satisfying the lexical rules

Semantic rules define the effect of programming satisfying the syntax rules


Semantic rules
Rely on
Syntactic rules
Rely on
Lexical rules