Einführung in die Programmierung
Introduction to Programming

Prof. Dr. Bertrand Meyer

Lecture 3: Dealing with Objects II
Programming languages

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

Our programming language is Eiffel.

There are many programming languages, some "general", some "specialized".

Programming languages are artificial notations, designed for a specific purpose (programming).
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 also called “procedural”.
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, e.g. C++, Java, C# if you need to

Course series (3rd year and up): “Languages in Depth”. Currently Java, C# and Eiffel.
Basic operations are called *instructions*

Our first example had *five* instructions:

- `Paris.display`
- `Louvre.spotlight`
- `Line8.highlight`
- `Route1.animate`
- `Console.show(Route1.origin)`
Successive instructions

You may write them one after the other without semicolons:

\[ Paris \cdot display \]
\[ Louvre \cdot spotlight \]
\[ Line8 \cdot highlight \]
\[ Route1 \cdot animate \]
\[ Console \cdot show (Route1 \cdot origin) \]

You may use semicolons to separate them:

\[ Paris \cdot display ; Louvre \cdot spotlight ; Line8 \cdot highlight ; Route1 \cdot animate ; Console \cdot show (Route1 \cdot origin) \]
Write one instruction per line
Omit semicolons

If you ever feel it clearer to have more than one instruction on a line (e.g. in a paper report) use semicolons:

\[ f(x) ; g(y) \]
Expressions

An expression is a program element denoting possible run-time values

Examples:

\texttt{Console\.show(Route1\.origin)}

Also, standard mathematical expressions:

\[ a + b \]
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

An expression, e.g. \texttt{Route1\textunderscore origin} is not a value but \textit{denotes} future run-time values.

An instruction, e.g. \texttt{Paris\textunderscore display} \textit{denotes} an operation to be executed at run time.
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.
# Syntax and semantics

<table>
<thead>
<tr>
<th>Syntax</th>
<th>Semantics</th>
</tr>
</thead>
<tbody>
<tr>
<td>Instruction</td>
<td>Command</td>
</tr>
<tr>
<td>Expression</td>
<td>Query</td>
</tr>
</tbody>
</table>
class PREVIEW inherit TOURISM

feature explore

-- Show city info.
do
Paris display
Louvre spotlight

end
Programming vs natural languages: similarities

- Overall form of texts: succession of words
- Distinction between syntax and semantics
- Some words predefined, others user-defined
Programming vs natural languages: differences

- **Power of expression** much higher with natural languages
- **Precision** much higher in programming languages

Programming languages are extensions of mathematical notation

*Comments* are bits of natural language appearing in programs
Style rule

Use words from natural language (e.g. English, German) for the names you define

Examples:

- Paris, Route 1
- Feature names: show, origin

Eiffel keywords are English words: inherit, do, end...

All single words except elseif
Syntax structure of a class

class PREVIEW inherit TOURISM

feature explore

-- Show city info.
do

Paris . display

Louvre . spotlight

end

end
Specimens

Specimen: a syntactic element; for example:

- A class name, e.g. *PREVIEW*
- An instruction, e.g. *Paris.display*
- Any of the boxes on the previous page
- The whole class text!

Specimens may be nested (or embedded)

Delimiters, such as keywords (do, end, ...), semicolons, periods • etc. are not specimens
Specimens and constructs

A **construct** is a certain type of syntactic element

Every syntactic element is a **specimen** of a certain construct

For example:

- *display* is a specimen of the construct *Feature_name*

- The class text as a whole is a specimen of the construct *Class*
Syntax structure of a class

```
class PREVIEW inherit TOURISM
feature explore
    -- Show city info.
    do
        Paris display
        Louvre spotlight
    end
end
```
Other representation: abstract syntax tree

- Class declaration
  - Class name
  - Inheritance
  - Features of the class
    - Feature name
    - Header comment
      - explore
      - -- Show city...
    - Feature body
      - Instruction (feature call)
        - Target
          - Paris
        - Feature
          - display
        - Target
          - Louvre
        - Feature
          - spotlight
Abstract syntax tree

Shows the syntax structure

Specimens only: no keywords or other delimiters (that's why it's abstract)

Uses the notion of tree as in organizational charts of companies.
Trees that grow down...

Organigramm der ETH Zürich: Schulleitung, Zentrale Organe sowie Lehr- und Forschungseinrichtungen ausserhalb der Departemente

Stand 1. Januar 2008
Trees in computer science

- Represent hierarchical or nested structures
- Similar to e.g. organizational charts (previous page)
- Pictured top-down or left-to-right
Tree properties

Tree rules:

- Every branch connects two nodes
- Every node can have any number (including none) of outgoing branches
- Every node has at most one incoming branch

Types of node:

- Root: node with no incoming branch
- Leaf: node with no outgoing branches
- Internal node: neither root nor leaf

A tree has exactly one root
(Otherwise it would be a forest)
Other representation: abstract syntax tree

- **Class declaration**
- **Class name**
- **Inheritance**
- **Features of the class**
- **Feature declaration**
- **Feature name**
- **Header comment**
- **Feature body**

**PREVIEW**

**TOURISM**

- **explore**
- -- Show city...

**Root**
- **Internal node** (Nonterminal)
- **Leaf** (Terminal)
Abstract syntax tree

- **Root**: represents overall specimen (outermost rectangle)
- **Internal nodes (nonterminals)**: represent substructures containing specimens themselves
- **Leaves (terminals)**: represent specimens with no further nesting

- The syntax of a programming language is defined by a set of constructs and the structure of these constructs.
Syntax structure of a class

class PREVIEW inherit TOURISM

feature explore -- Show city info.
do
  Paris display
  Louvre spotlight
end

end
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
Three levels of description

Semantic rules define the effect of programming satisfying the syntax rules

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

Lexical rules define how to make up tokens out of characters
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.
Style rules

- Always choose identifiers that clearly identify the intended role

- For features, use full names, not abbreviations

- For multi-word identifiers, use underscores:

  \textit{bus\_station}

- Use all upper case for classes:

  \textit{PREVIEW}
What we have seen in this lecture

- Programming language concepts
- Eiffel basics
- Syntax (including lexical level) vs semantics
- Trees
- Tree terminology: root, leaf, node...
- Abstract Syntax Trees (AST)
- Basic lexical elements
- Basic style rules
What to do for next week

Read chapters 1 to 5 of *Touch of Class*

Make sure you know all the terminology introduced so far