Einführung in die Programmierung
Introduction to Programming

Prof. Dr. Bertrand Meyer

Exercise Session 2
Organizational

- Assignments
  - One assignment per week
  - Will be put online on Monday, due on Tuesday the next week

- Testat
  - You have to hand in $n - 1$ out of $n$ assignments
    - Must include the last one
    - Show serious effort
  - You have to hand in two mock exams

- Military service or illness → contact assistant

- Group mailing list
  - Is everybody subscribed?
Today

- Give you the intuition behind object-oriented (OO) programming
- Teach you about formatting your code
- Distinguishing between
  - commands and queries
  - feature declaration and feature call
- Understanding feature call chains
- Getting to know the basics of EiffelStudio
Classes and objects

- Classes are pieces of software code.
- Several classes make up a program.
- Objects are instances of classes.
- Classes define operations applicable to their instances.
  - Example: A class STUDENT can define operations applicable to all its instances, such as subscribing to a course, registering for an exam, etc. This means that all class STUDENT's instances (such as the students Bob, Mike, Steve, etc.) will be able to subscribe themselves to a course, to register for an exam, etc.
- Only operations defined in a class can be applied to its instances.
Features

- A feature is an operation that programs may apply to certain classes of objects. A feature can be called on an object. This object is called the target of the feature call.

Examples

- `next_message.send`
- `computer.shut_down`
- `telephone.ring`

- A feature call can have arguments.

Examples

- `next_message.send_to (recipient)`
- `computer.shut_down_after (3)`
- `telephone.ring_several (10, Loud)`
Features: Exercise

- Class `BANK_ACCOUNT` defines and implements the following operations:
  - `deposit (i: INTEGER)`
  - `withdraw (i: INTEGER)`
  - `close`

- If `b: BANK_ACCOUNT` (b is an instance of class `BANK_ACCOUNT`) which of the following feature calls are possible:
  - `b.deposit (10)` ✓
  - `b.deposit` ×
  - `b.close` ✓
  - `b.close (“Now”)` ×
  - `b.open` ×
  - `b.withdraw (100.50)` ×
  - `b.withdraw (0)` ✓
class TOURISM inherit PREVIEW

feature explore

-- Show city info.
do

Paris . display

Louvre . spotlight

end

end
For indentation, use tabs, not spaces

Use this property to highlight the **structure** of the program, particularly through indentation.

```ruby
class PREVIEW
  inherit TOURISM
  feature explore
    do
      Paris.display
      Louvre.spotlight
      Line8.highlight
      Route1.animate
    end
  end
end
```

---

---

---

---
More style rules

- Class name: all upper-case
- Period in feature call: no space before or after
- Names of predefined objects: start with upper-case letters
- New names (for objects you define) start with lower-case letters

```
class PREVIEW
  inherit TOURISM
  feature explore
    do
      Paris.display
        Louvre.spotlight
        Line8.highlight
      end
    end
  end
end
```
Even more style rules

For features, use full names, 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 PREVIEW
inherit TOURISM
feature explore
  -- Show city info
  -- and route.
  do
    Paris.display
    Louvre.spotlight
    Line8.highlight
    Line8.remove_all_sections
    Route1.animate
end
end
Exercise: style rules

Format this class:

class bank_account
feature deposit (sum: INTEGER)
  -- Add `sum' to the account.
  do balance := balance + sum end
balance: INTEGER end
Exercise: solution

class
  BANK_ACCOUNT

feature
  deposit (sum: INTEGER)
    -- Add `sum' to the account.
    do
      balance := balance + sum
    end
  balance: INTEGER
end
Commands and queries

- A feature can be a:
  - Command: a feature that may modify an object
  - Query: a feature that accesses an object
Queries

- Goal: obtain properties of objects
- Always return a value.
- Should not modify any objects.
- Examples
  - What is the name of a person?
  - What is the age of a person?
  - What is the id of a student?
  - Is a student registered for a particular course?
  - Are there any places left in a certain course?
  - ... other examples?
Commands

- **Goal:** produce a change on an object, or several
- **Does not return a value.**
- **May modify objects.**

**Examples**

- Register a student to a course
- Assign an id to a student
- Record the grade a student got in an exam
- ... other examples?
Exercise: query or command?

- What is the balance of a bank account?
- Withdraw some money from a bank account
- Who is the owner of a bank account?
- Who are the clients of a bank whose deposits are over 100,000 CHF?
- Change the account type of a client
- How much money can a client withdraw at a time?
- Set a minimum limit for the balance of accounts
- Is Steve Jobs a client of Credit Suisse?
Command-query separation principle

“*Asking* a question *shouldn’t change* the answer”

i.e. a query
class DEMO

feature

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

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

attribute_name: T3
  -- Comment

end

query

command

no result
body

result
body

result
no body
Features: the full story

Client view (specification)

Command

Procedure

Feature

Query

No result

Returns result

Internal view (implementation)

Routine

Function

Attribute

Computation

Memory

Computation

Memory
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
  ```

- You call a feature when you apply it to an object.
  ```
  a_person.set_name ("Peter")
  ```
General form of feature call instructions

object1.query1.query2.command (object2.query3.query4, object3)

- Targets and arguments can contain feature calls themselves.

- Where are query1, query2, query3 and query4 defined?
- Where is command defined?
It is possible to leave out the target in a feature call. Such a call is called unqualified. The implicit target will be the current object. A qualified feature call has a target.

The current object in a feature is always the instance of the surrounding class.

```plaintext
assign_same_name (a_name: STRING; a_other_person: PERSON)
  -- Set 'a_name' to this person and 'a_other_person'.
  do
    a_other_person.set_name(a_name)
    set_name(a_name)
  end

person1.assign_same_name("Hans", person2)
```

In this call the current object will be person1.
EiffelStudio

- EiffelStudio is a software tool (IDE) to develop Eiffel programs.

- Help & Resources
  - Online tour in the help of EiffelStudio
  - [http://docs.eiffel.com/](http://docs.eiffel.com/)
Components

- editor
- context tool
- clusters pane
- features pane
- compiler
- project settings
- ...
Editor

- Syntax highlighting
- Syntax completion
- Class name completion (SHIFT+CTRL+Space)
- Smart indenting
- Block indenting or unindenting (TAB and SHIFT+TAB)
- Block commenting or uncommenting (CTRL+K and SHIFT+CTRL+K)
- Infinite level of Undo/Redo (reset after a save)
- Quick search features (first CTRL+F to enter words then F3 and SHIFT+F3)
Compiler

- Uses incremental compilation
  - freezing: Generates C code from the whole system and then compiles it to machine code. This code is used during development. Initially the system is frozen.
  - melting: Generates bytecode for the changed parts of the system. This is much faster than freezing. This code is used during development.
  - finalizing: Creates an executable production version. Finalization performs extensive time and space optimizations.
Debugger: setup

- The system must be melted/frozen (finalized systems cannot be debugged).
- Set / delete breakpoints
  - An efficient way of adding breakpoints consists in dropping a feature in the context tool.
  - Click in the margin to enable/disable single breakpoints.
- Use the toolbar debug buttons to enable or disable all breakpoints globally.
Debugger: run

- Run the program by clicking on the Run button.
- Pause by clicking on the Pause button or wait for a triggered breakpoint.
- Analyze the program:
  - Use the call stack pane to browse through the call stack.
  - Use the object tool to inspect the current object, the locals and arguments.
- Run the program or step over / into the next statement.
- Stop the running program by clicking on the Stop button.
The following slides contain advanced material and are optional.
Outline

- Syntax comparison: Eiffel vs Java
- Naming in Eiffel
- Feature comments: Less is better (sometimes...)
class ACCOUNT
end

class Account {
}

Eiffel vs Java: Inheritance

class ACCOUNT
inherit ANY
end

public class Account
extends Object {

}
Eiffel vs Java: Feature redefinition

class ACCOUNT
inherit ANY
  redefine out end
feature
  out: STRING
  do
    Result := "abc"
  end
end

class Account extends Object {
  String toString() {
    return "abc";
  }
}
Eiffel vs Java: Precursor call

class ACCOUNT
inherit ANY
redefine out end

feature
  out: STRING
  do
    Result := Precursor {ANY}
  end
end

public class Account
  extends Object {
    String toString() {
      return super();
    }
  }

deferred class ACCOUNT
feature deposit (a: INT)
defered
end
end

abstract class Account {
    abstract void deposit(int a);
}
Eiffel vs Java: Frozen

frozen class ACCOUNT
inhibit ANY
end

final class Account
    extends Object {
    }


Eiffel vs Java: Expanded

expanded class ACCOUNT

end

int, float, double, char
Eiffel vs Java: Constructors

class ACCOUNT
create make
feature make do end
end

public class Account {
    public Account() {}
}

Eiffel vs Java: Constructor overloading

class ACCOUNT
create
  make, make_amount
feature
  make do end
  make_amount (a: INT)
    do end
end

public class Account {
  public Account() {}
  public Account(int a) {}
}
Eiffel vs Java: Overloading

class PRINTER
feature
  print_int (a: INTEGER) do end
  print_real (a: REAL) do end
  print_string (s: STRING) do end
end

public class Printer {
  public print(int i) {}
  public print(float f) {}
  public print(String s) {}
}
Eiffel vs Java: Exception Handling

class PRINTER

feature

print_int (a: INTEGER)
do
(create EXCEPTION).raise
rescue
retry
end
end

public class Printer {

public print(int i) {
try {
throw new Exception()
}
catch(Exception e) {
}
}
Eiffel vs Java: Conditional

class PRINTER
feature
  print do
    if True then
    else
    end
  end
end

public class Printer {
  public print() {
    if (true) {
    }
    else {
    }
  }
}
Eiffel vs Java: Loop 1

**Eiffel**

```eiffel
class Example
  print
  local
    i: INTEGER
  do
    from
      i := 1
    until
      i >= 10
    loop
      i := i + 1
    end
  end
end
```

**Java**

```java
public class Example {
  public void print() {
    for (int i = 1; i < 10; i++) {
    }
  }
}
```
Eiffel vs Java: Loop 2

```eiffel
print
local
  i: INTEGER
do
  from
    i := 1
  until
    i >= 10
loop
  i := i + 1
end
end
```

```java
public class Printer {
    public print() {
        int i=1;
        while(i<10) {
            i++;
        }
    }
}
```
Eiffel vs Java: Loop 3

```java
public class Printer {
    public void print() {
        for(Element e : list) {
            e.print();
        }
    }
}
```

ECMA committee is discussing about `foreach` for Eiffel
Eiffel Naming: Classes

- Full words, no abbreviations (with some exceptions)
- Classes have global namespace
  - Name clashes arise
- Usually, classes are prefixed with a library prefix
  - Traffic: TRAFFIC_
  - EiffelVision2: EV_
  - Base is not prefixed
Eiffel Naming: Features

- Full words, no abbreviations (with some exceptions)
- Features have namespace per class hierarchy
  - Introducing features in parent classes, can clash with features from descendants
Eiffel Naming: Locals / Arguments

- Locals and arguments share namespace with features
  - Name clashes arise when a feature is introduced, which has the same name as a local (even in parent)
- To prevent name clashes:
  - Locals are prefixed with `l_`
  - Some exceptions like „i“ exist
  - Arguments are prefixed with `a_`
tangent_ from (p: POINT): LINE
   -- Return the tangent line to the current circle
   -- going through the point p, if the point
   -- is outside of the current circle.

require
   outside_circle: not has (p)

Example is from http://dev.eiffel.com/Style_Guidelines
tangent__ from (p: POINT): LINE
  -- The tangent line to the current circle
  -- going through the point p, if the point
  -- is outside of the current circle.

require
  outside_circle: not has (p)
tangent_ from (p: POINT): LINE
-- Tangent line to current circle from point p
-- if the point is outside of the current circle.

require
outside_circle: not has (p)
Feature comments: Version 4

tangent_ from (p: POINT): LINE
   -- Tangent line to current circle from point p.

require
   outside_circle: not has (p)
tangent__ from (p: POINT): LINE
   -- Tangent from p.

require
   outside_circle: not has (p)
tangent_ from (p: POINT): LINE
    -- Tangent from p.
    --
    -- `p': The point from ...
    -- `Result': The tangent line ...
    --
    -- The tangent is calculated using the
    -- following algorithm:
    -- ...

require
outside_circle: not has (p)
tangent_from (p: POINT): LINE

-- <Precursor>

require

outside_circle: not has (p)
Ideas for future sessions

- Inheritance concepts: Single/Multiple/Non-conforming
- CAT Calls (Covariance and generics)
- Once/Multiple inheritance vs. Static
- Exception handling
- Design by contract in depth
- Void-safety
- Modeling concepts
- Best practices in Eiffel
- A look at ECMA specification of Eiffel