Einführung in die Programmierung
Introduction to Programming

Prof. Dr. Bertrand Meyer

Exercise Session 2

Organizational

- Assignments
  - One assignment per week
  - Will be put online Friday (before 17:00)
  - Should be handed in within ten days (Monday, before 15:00)
- 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
  - feature declaration and feature call
  - commands and queries
- 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.
  - A class may have many instances.
- 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 may be applied to certain classes of objects.
- **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
    ```

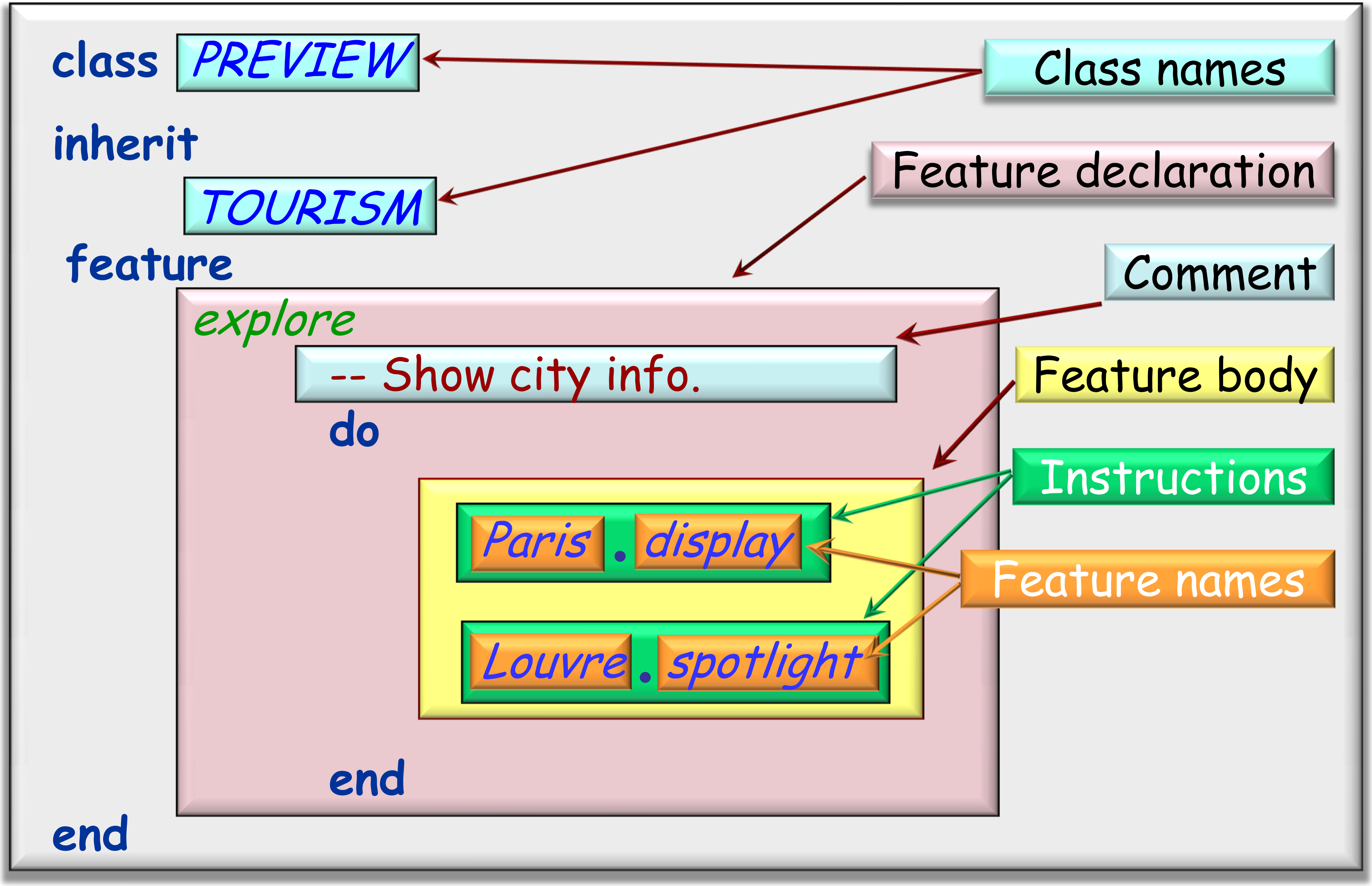
  - You call a feature when you apply it to an object.
    The object is called the **target** of this feature call.
    - *a_person.set_name ("Peter")*
  - Arguments, if any, need to be provided in feature calls.
    - *computer.shut_down*
    - *computer.shut_down_after (3)*
    - *telephone.ring_several (10, Loud)*

Features: Exercise

Hands-On

- Class *BANK_ACCOUNT* defines the following operations:
  - *deposit (a_num: INTEGER)*
  - *withdraw (a_num: 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 text



```
class PREVIEW
inherit
    TOURISM
feature
    explore
            -- Show city info.
        do
                Paris.display
                Louvre.spotlight
        end
end
```

Class names

Feature declaration

Comment

Feature body

Instructions

Feature names

Style rule

For indentation, use tabs, not spaces

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

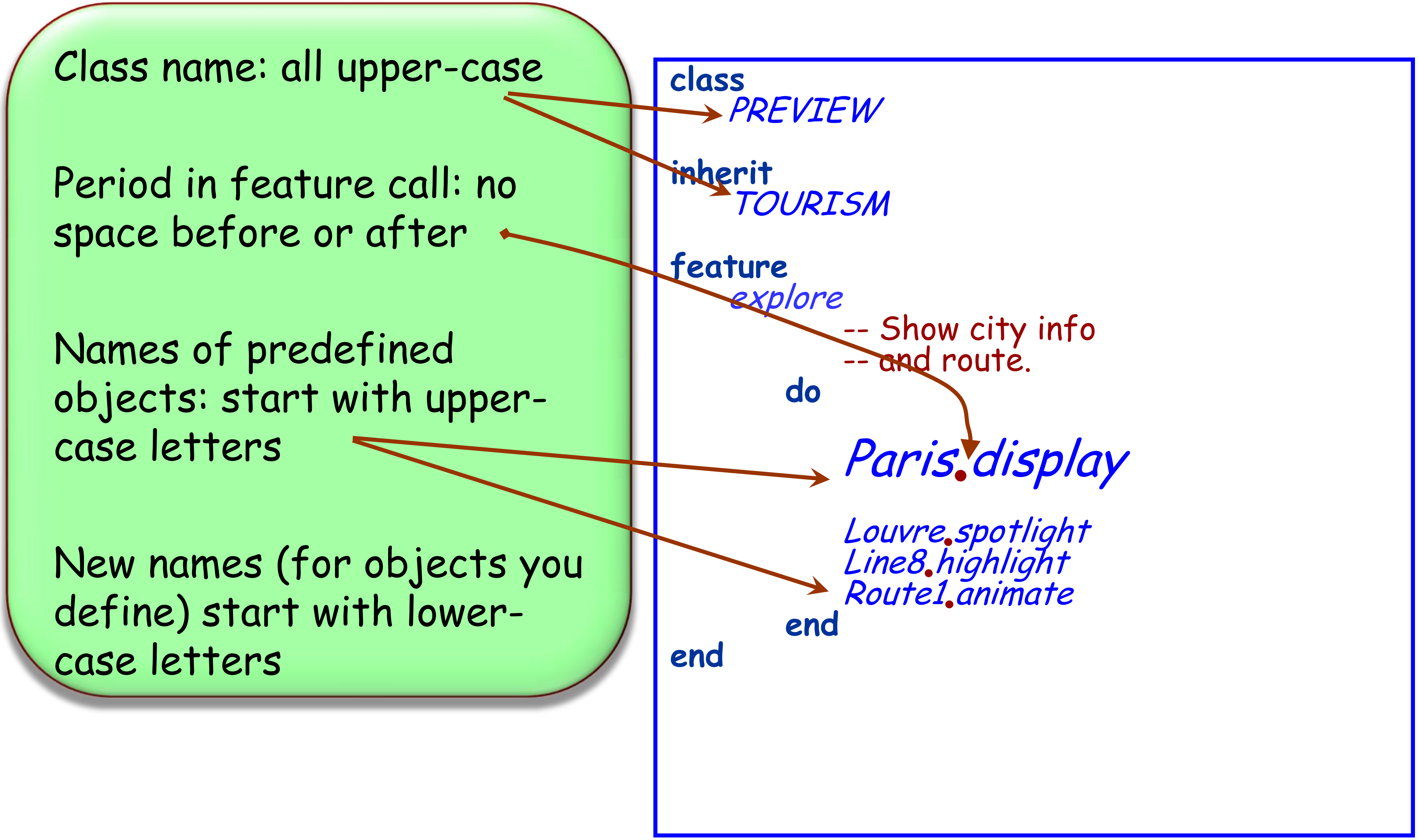
```
class
		PREVIEW

inherit
		TOURISM

feature
	explore
			-- Show city info
			-- and route.
		do
			Paris.display
			Louvre.spotlight
			Line8.highlight
			Route1.animate
		end
end
```

Tabs

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
            -- Show city info
            -- and route.
        do

            Paris.display

            Louvre.spotlight
            Line8.highlight
            Route1.animate
        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
    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

Hands-On

- Format this class:

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

Exercise: solution

```
class
    BANK_ACCOUNT

feature
    deposit (a_sum: INTEGER)
            -- Add `a_sum' to the account.
        do
            balance := balance + a_sum
        end

     balance: INTEGER
end
```

Within comments, use ` and ' to quote names of arguments and features

Commands and queries

- A feature can be:
  - a command: a feature to carry out some computation
    - Register a student to a course
    - Assign an id to a student
    - Record the grade a student got in an exam
    - … other examples?
  - a query: a feature to obtain properties of objects
    - 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?

	Query	Command
Modify object(s)?	N	Y
Return value?	Y	N

Exercise: query or command?

Hands-On

- 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

Query or command?

```
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
```

Predefined variable denoting the result

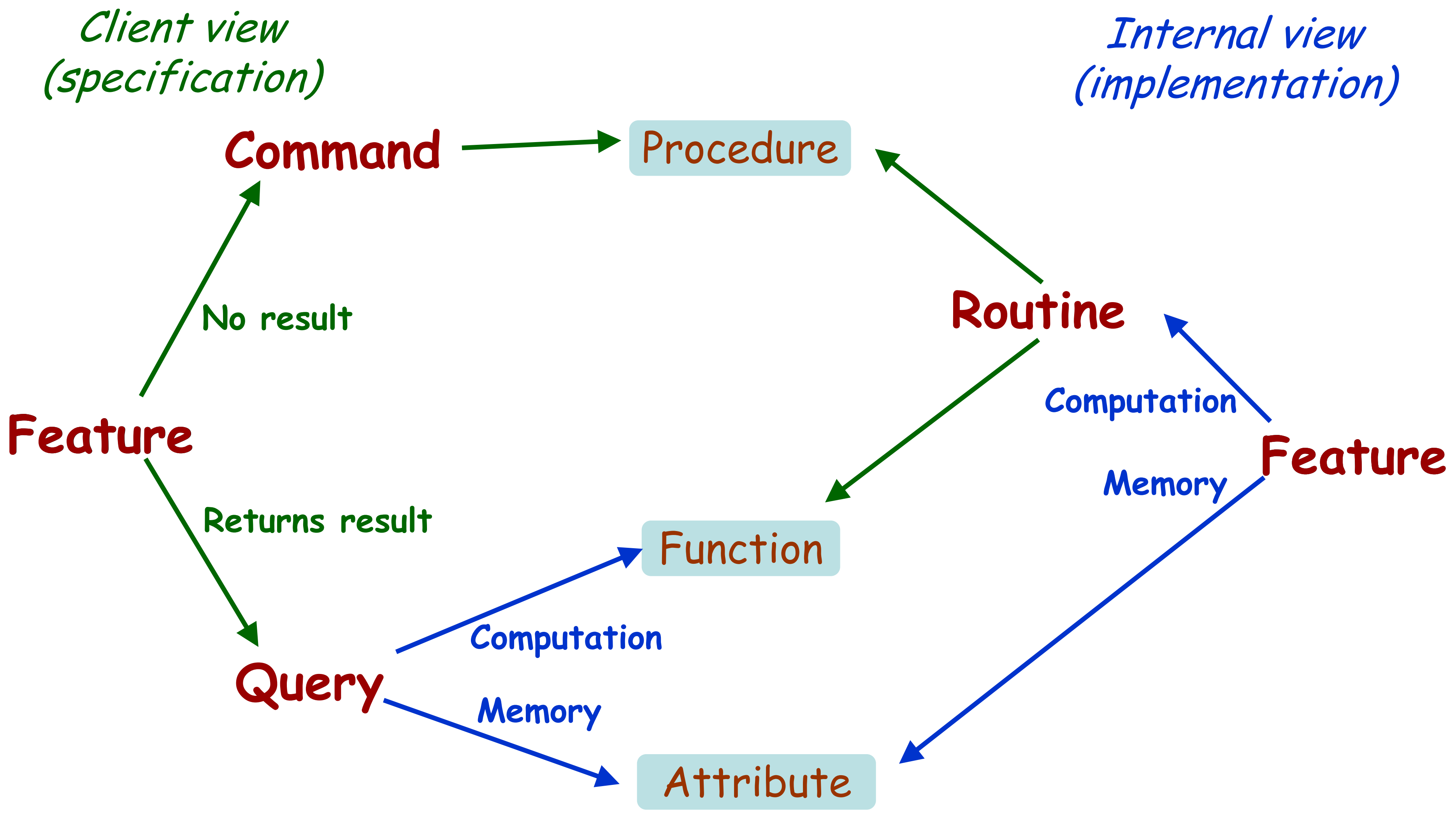
- result
- body

query

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

- result
- no body

Features: the full story



General form of feature call instructions

$$\underbrace{object1.query1.query2}_{\text{target}}.command\ (\underbrace{object2.query3.query4,\ object3)}_{\text{arguments}}$$

- Targets and arguments can be query calls themselves.

Hands-On

- Where are *query1*, *query2*, *query3* and *query4* defined?
- Where is *command* defined?

Qualified vs. unqualified feature calls

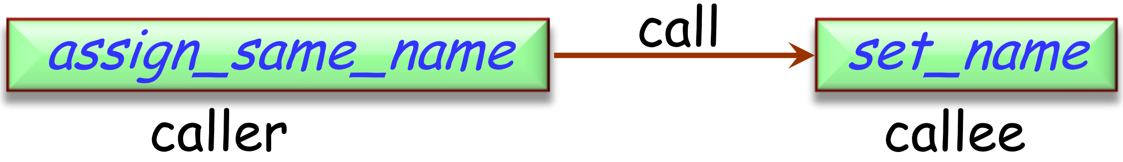
- A **qualified** feature call has an explicit target.
- An **unqualified** feature call is one whose target is left out.
  - An unqualified feature call uses the current object of its caller as the implicit target.
  - The **current object** of a feature is the object on which the feature is called. (what's the other name for this object?)

```
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
```

Qualified call

Unqualified call, same as
Current.set_name (a_name)

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



EiffelStudio

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

Integrated Development Environment

- Help & Resources
  - Online tour in the help of EiffelStudio
  - http://www.eiffel.com/
  - http://dev.eiffel.com/
  - http://docs.eiffel.com/
  - http://www.ecma-international.org/publications/files/ECMA-ST/ECMA-367.pdf

Components

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

Editor

- Syntax highlighting
- Syntax completion
- Auto-completion (CTRL+Space)
- 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.

Advanced Material

The following slides contain advanced material and are optional.

Outline

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

Eiffel vs Java: Class declaration

```
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
```

```
public 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();
       }

}
```

Eiffel vs Java: Deferred

```
deferred class
	ACCOUNT

feature
	deposit (a_num: INT)
		deferred
		end
end
```

```
abstract class Account {
	abstract void deposit(int a);
}
```

Eiffel vs Java: Frozen

```
frozen class
        ACCOUNT
inherit
        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_amount: INT)
        do end

end
```

```
public class Account {
    public Account() {}
    public Account(int a) {}
}
```

Eiffel vs Java: Overloading

```
class
    PRINTER

feature
    print_int (a_int: INTEGER)
        do end

    print_real (a_real: REAL)
        do end

    print_string (a_str: STRING)
        do end
end
```

```
public class Printer {
        public print(int i) {}
        public print(float f) {}
        public print(String s) {}
}
```

Eiffel: Exception Handling

```
class
    PRINTER
feature
    print_int (a_int: INTEGER)
        local
            l_retried: BOOLEAN
        do
            if not l_retried then
                (create {DEVELOPER_EXCEPTION}).raise
            else
                -- Do something alternate.
            end
        rescue
            l_retried := True
            retry
      end
end
```

Java: Exception Handling

```
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

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

```
public class Printer {
    public print() {
        for(int i=1;i<10;i++) {
            ...
        }
    }
}
```

Eiffel vs Java: Loop 2

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

```
public class Printer {
    public print() {
        int i=1;
        while(i<10) {
            i++;
        }
    }
}
```

Eiffel vs Java: Loop 3

```
print_1
    do
        from list.start
        until list.after
        loop
            list.item.print
            list.forth
        end
     end

print_2
    do
        -- Enable "provisional syntax" to
        -- use "across"
        across list as e loop
             e.item.print
        end
    end
```

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

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_**

Feature comments: Version 1

```
tangent_ from (a_point: POINT): LINE
        -- Return the tangent line to the current circle
        -- going through the point `a_point', if the point
        -- is outside of the current circle.
    require
        outside_circle: not has (a_point)
```

Example is from http://dev.eiffel.com/Style_Guidelines

Feature comments: Version 2

```
tangent_ from (a_point : POINT): LINE
        -- The tangent line to the current circle
        -- going through the point `a_point', if the point
        -- is outside of the current circle.
    require
        outside_circle: not has (a_point)
```

Feature comments: Version 3

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

Feature comments: Version 4

```
tangent_ from (a_point : POINT): LINE
        -- Tangent line to current circle from point `a_point’.
    require
        outside_circle: not has (a_point)
```

Feature comments: Final version

```
tangent_ from (a_point : POINT): LINE
        -- Tangent from `a_point'.
    require
        outside_circle: not has (a_point)
```

Feature comments: More information

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

Feature comments: Inherited comments

```
tangent_ from (a_point : POINT): LINE
        -- <Precursor>
    require
        outside_circle: not has (a_point)
```

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