



1

# Object-Oriented Software Construction

Bertrand Meyer

## Lecture 4: Objects



## The basic structure: The class (cont'd) <sup>3</sup>

- **From the module viewpoint:**
  - Set of available services (“features”).
  - Information hiding.
  - Classes may be clients of each other.
- **From the type viewpoint:**
  - Describes a set of run-time objects (the **instances** of the class).
  - Used to declare entities ( $\approx$  variables), e.g.  
 $x: C$
  - Possible type checking.
  - Notion of subtype.



## The basic structure: The class <sup>2</sup>

2

- A class is an implementation of an ADT. It is both:
  - A module.
  - A type.
- Much of the conceptual power of the method comes from the fusion of these two notions.



## Avoid “objectspeak” <sup>4</sup>

4

- The run-time structures, some of them corresponding to “objects” of the modeled system, are **objects**.
- The software modules, each built around a *type* of objects, are **classes**.
- A system does not contain any “objects” (although its execution will create objects).

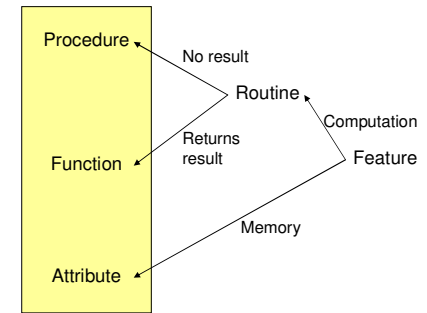
## Terminology

5

- A class is an implementation of an abstract data type.
  - **Instances** of the class may be created at run-time; they are **objects**.
- Every object is an instance of a class. (In a pure O-O language such as Eiffel and Smalltalk this is true even of basic objects such as integers etc. Not true in C++ or Java where such values have special status.)
- A class is characterized by **features**. Features comprise **attributes** (representing data fields of instances of the class) and **routines** (operations on instances).
- Routines are subdivided into **procedures** (effect on the instance, no result) and **functions** (result, normally no effect).
- Every operation (routine or attribute call) is relative to a distinguished object, the **current instance** of the class.

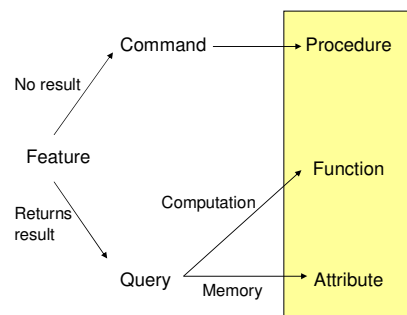
## Feature categories by implementation

7



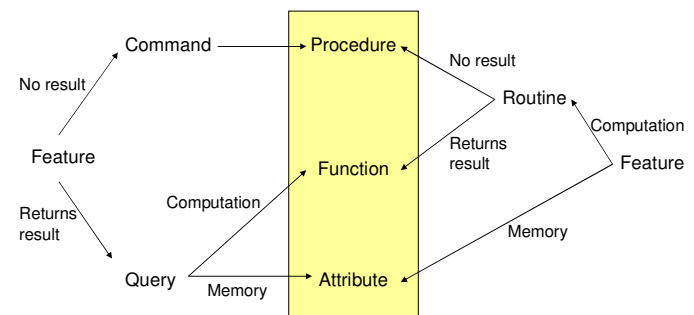
## Feature categories by role

6



## Feature categories

8



## Alternative terminology

9

- Attributes are also called **instance variables** or **data member**.
- Routines are also called **methods**, **subprograms**, or **subroutines**.
- Feature call — applying a certain feature of a class to an instance of that class — is also called **passing a message** to that object.
- The notion of **feature** is particularly important as it provides a single term to cover both attributes and routines. It is often desirable **not** to specify whether a feature is an **attribute** or a **routine** — as expressed by the **Uniform Access principle** (see next).

## The Principle of Uniform Access

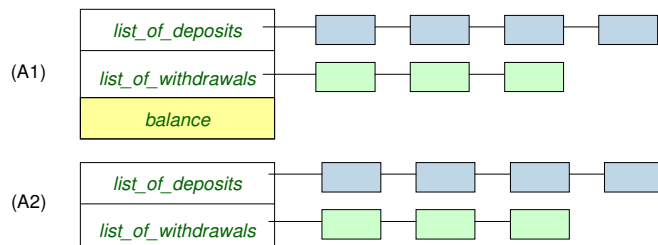
11

- **Facilities managed by a module must be accessible to clients in the same way whether implemented by computation or storage.**

## Uniform Access

10

$balance = list\_of\_deposits.total - list\_of\_withdrawals.total$



## Uniform access through feature call

12

- To access a property of a point  $p1$ , the notation is the same regardless of the representation, e.g.  
 $p1.x$   
which is applicable both in cartesian representation ( $x$  is an attribute) and in polar representation ( $x$  is a function without arguments).
- In the first case the feature call is a simple field access; in the second it causes a computation to be performed.
- There is no difference for clients (except possibly in terms of performance).

## Abstract data type POINT

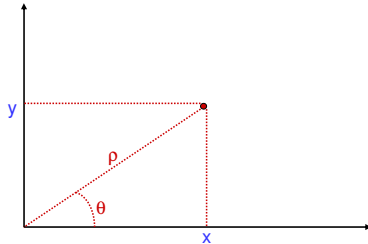
13

$x: \text{POINT} \rightarrow \text{REAL}$

$y: \text{POINT} \rightarrow \text{REAL}$

$\rho: \text{POINT} \rightarrow \text{REAL}$

$\theta: \text{POINT} \rightarrow \text{REAL}$



- Class **POINT**: Choose a representation (polar, cartesian)
- In polar representation,  $\rho$  and  $\theta$  are attributes,  $x$  and  $y$  are routines.

## Class POINT (continued)

15

```

distance (p: POINT): REAL is
  -- Distance to p
  do
    Result := sqrt ((x - p.x)^2 + (y - p.y)^2)
  end

ro: REAL is
  -- Distance to origin (0, 0)
  do
    Result := sqrt (x^2 + y^2)
  end

theta: REAL is
  -- Angle to horizontal axis
  do
    ...
  end
end

```

## Class POINT

14

class

POINT

feature

$x, y: \text{REAL}$  -- Point cartesian coordinates

**move** ( $a, b: \text{REAL}$ ) is -- Move by  $a$  horizontally and by  $b$  vertically.

```

do
  x := x + a
  y := y + b
end

```

**scale** ( $factor: \text{REAL}$ ) is -- Scale by factor.

```

do
  x := factor * x
  y := factor * y
end

```

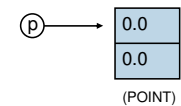
## Use of the class in a client

16

```

class GRAPHICS feature
  p, q: POINT -- Graphic points
  ...
  some_routine is -- Use p and q.
    local
      u, v: REAL
    do
      -- Creation instructions
      create p
    end
  end
end

```





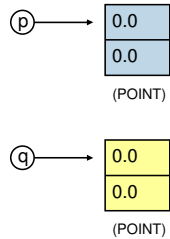
## Use of the class in a client

17

```

class GRAPHICS feature
  p, q: POINT -- Graphic points
  ...
  some_routine is -- Use p and q.
    local
      u, v: REAL
    do
      create p -- Creation instructions
      create q
    end
  end
end

```



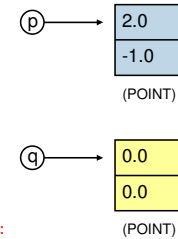
## Use of the class in a client

19

```

class GRAPHICS feature
  p, q: POINT -- Graphic points
  ...
  some_routine is -- Use p and q.
    local
      u, v: REAL
    do
      create p -- Creation instructions
      create q
      p.move (4.0, -2.0) -- Compare with Pascal, C, Ada:
                          -- move (p, 4.0, -2.0)
      p.scale (0.5)
    end
  end
end

```



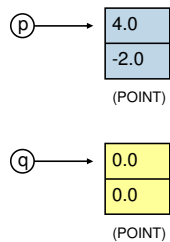
## Use of the class in a client

18

```

class GRAPHICS feature
  p, q: POINT -- Graphic points
  ...
  some_routine is -- Use p and q.
    local
      u, v: REAL
    do
      create p -- Creation instructions
      create q
      p.move (4.0, -2.0) -- Compare with Pascal, C, Ada:
                          -- move (p, 4.0, -2.0)
    end
  end
end

```



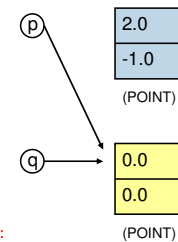
## Use of the class in a client

20

```

class GRAPHICS feature
  p, q: POINT -- Graphic points
  ...
  some_routine is -- Use p and q.
    local
      u, v: REAL
    do
      create p -- Creation instructions
      create q
      p.move (4.0, -2.0) -- Compare with Pascal, C, Ada:
                          -- move (p, 4.0, -2.0)
      p.scale (0.5)
      u := p.distance (q)
      v := p.x
      p := q
    end
  end
end

```



## Use of the class in a client

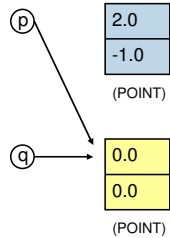
21

```

class GRAPHICS feature
  p, q: POINT -- Graphic points
  ...
  some_routine is -- Use p and q.
  local
    u, v: REAL
  do
    -- Creation instructions
    create p
    create q

    p.move(4.0, -2.0)
    -- Compare with Pascal, C, Ada:
    -- move(p, 4.0, -2.0)

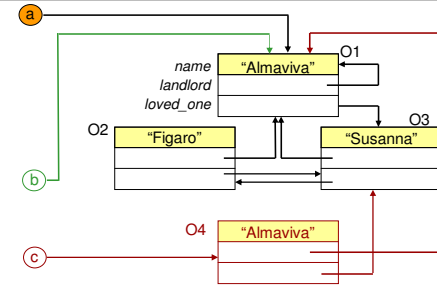
    p.scale(0.5)
    u := p.distance(q)
    v := p.x
    p := q
    p.scale(-3.0)
  end
end
  
```



## Shallow and deep cloning

23

Initial situation:

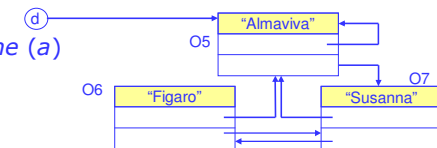


Result of:

$b := a$

$c := \text{clone}(a)$

$d := \text{deep\_clone}(a)$



## Variants of assignment and copy

22

- Reference assignment (*a* and *b* of reference types):

$b := a$

- Object duplication (shallow):

$c := \text{clone}(a)$

- Object duplication (deep):

$d := \text{deep\_clone}(a)$

- Also: shallow field-by-field copy (no new object is created):

$e.\text{copy}(a)$

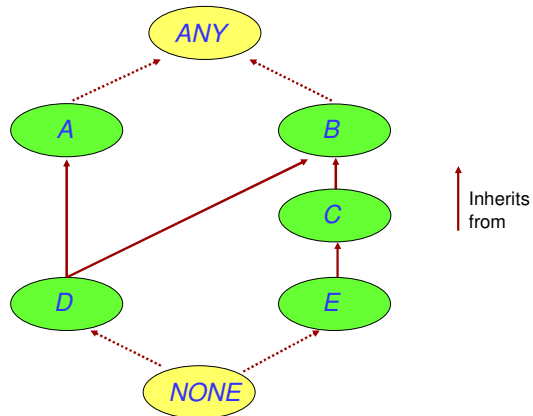
## Where do these mechanisms come from?

24

- Class *ANY* in the Eiffel "Kernel Library"
- Every class that doesn't explicitly inherit from another is considered to inherit from *ANY*
- As a result, every class is a descendant of *ANY*.

## Completing the inheritance structure

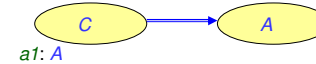
25



## Applying abstraction principles

27

- Privileges of a client  $C$  of a class  $A$  on an attribute  $attrib$ :
  - Read access if attribute is exported.



- Assuming  $a1: A$   
Then  $a1.attrib$  is an expression.
- An assignment such as  $a1.attrib := a2$  is syntactically illegal!  
(You cannot assign a value to an expression, e.g.  $x + y$ .)

## A related mechanism: Persistence

26

$a.store(file)$

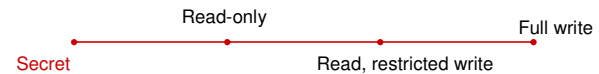
....

$b ?= retrieved(file)$

- Storage is automatic.
- Persistent objects identified individually by keys.
- These features come from the library class **STORABLE**.

## The privileges of a client

28



## Applying abstraction principles

29

- Beyond read access: full or restricted write, through exported procedures.
- Full write privileges: *set\_attribute* procedure, e.g.

```
set_temperature (u: REAL) is
  -- Set temperature value to u.
do
  temperature := u
ensure
  temperature_set: temperature = u
end
```

- Client will use e.g. *x.set\_temperature* (21.5).

## Other uses of a setter procedure

31

```
set_temperature (u: REAL) is
  -- Set temperature value to u.
require
  not_under_minimum: u >= -273
  not_above_maximum: u <= 2000
do
  temperature := u
  update_database (u, Current)
ensure
  temperature_set: temperature = u
end
```

## Setter procedures

30

- *set\_attribute* procedure, e.g.

```
set_temperature (u: REAL) is
  -- Set temperature value to u.
do
  temperature := u
ensure
  temperature_set: temperature = u
end
```

- Client will use e.g. *x.set\_temperature* (21.5).
- Client cannot directly assign to attribute

## Delphi/C# "properties"

32

- Allow  
*x.temperature* = 21.5;

if there is a "setter":

```
private int temperature_internal;
public int temperature
{
  get { return temperature_internal; }
  set {
    temperature_internal = value;
    //... Other instructions; ...
  }
}
```

## Information hiding

33

```

class
  A
feature
  f ...
  g ...
feature {NONE}
  h ...
feature {B, C}
  j ...
feature {A, B, C}
  k
end
    
```

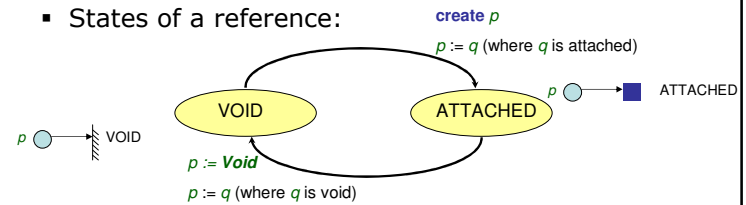
In clients, with the declaration *a1: A*, we have:

- *a1.f*, *a1.g*: valid in any client
- *a1.h*: invalid anywhere (including in *A*'s own text).
- *a1.j*: valid only in *B*, *C* and their descendants (not valid in *A*!)
- *a1.k*: valid in *B*, *C* and their descendants, as well as in *A* and its descendants

## The dynamic model

35

- States of a reference:



- Operations on references:

```

create p
p := q
p := Void
if p = Void then ...
    
```

## Information hiding

34

- Information hiding only applies to use by clients, using dot notation or infix notation, as with *a1.f* ("Qualified calls").
- Unqualified calls (within the class itself) are not subject to information hiding:

```

class
  A
feature {NONE}
  h is
    do
      -- Does something.
    end
feature
  f is
    do
      -- Use h.
    end
end
    
```

## Creating an object

36

- With the class *POINT* as given:

```

my_point: POINT
...
create my_point
    
```

- Effect of such a creation instruction:
  - Allocate new object of the type declared for *my\_point*.
  - Initialize its fields to default values (0 for numbers, false for booleans, null for characters, void for references).
  - Attach it to the instruction's target, here *my\_point*.

## Specific creation procedures

37

```
class
  POINT
create
  make_cartesian, make_polar
feature {NONE} -- Initialization
  make_cartesian (a, b: REAL) is
    -- Initialize to abscissa a, ordinate b.
    do
      x := a
      y := b
    end
  make_polar ...
feature
  ... The rest as before ...
```

## If there is no creation clause

39

- An absent creation clause, as in

```
class
  POINT
  -- No creation clause
feature
  ... The rest as before ...
end
```

is understood as one that would only list *default\_create*, as if it had been written

```
class
  POINT
create
  default_create
feature
  ... The rest as before ...
end
```

- Procedure *default\_create* is defined in *ANY* as doing nothing; any class can redefine it to provide proper default initializations.

## If there is a creation clause

38

- Creation instructions must be "creation calls", such as

```
create my_point.make_polar (1, Pi / 2)
```

## Associated convention

40

- The notation  
**create** *x*

is understood (if permitted) as an abbreviation for

```
create x.default_create
```

## To allow both forms

41

- To make both forms valid:

```
create my_point
```

as well as

```
create my_point.make_polar (1, Pi / 2)
```

it suffices to make *default\_create* (redefined or not) one of the creation procedures:

```
class
```

```
POINT
```

```
create
```

```
make_cartesian, make_polar, default_create
```

```
feature
```

```
... The rest as before ...
```

## Arguments for automatic collection

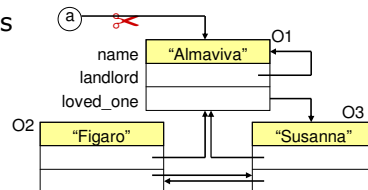
43

- Manual reclamation is dangerous. Hampers software reliability.
- In practice bugs arising from manual reclamation are among the most difficult to detect and correct. Manifestation of bug may be far from source.
- Manual reclamation is tedious: need to write "recursive dispose" procedures.
- Modern garbage collectors have acceptable overhead (a few percent) and can be made compatible with real-time requirement.
- GC is tunable: disabling, activation, parameterization....

## What to do with unreachable objects

42

- Reference assignments may make some objects useless.



- Two possible approaches:
  - Manual reclamation (e.g. C++, Delphi).
  - Automatic garbage collection (e.g. Eiffel, Smalltalk, Simula, Java, .NET)

## Properties of a garbage collector (GC)

44

- Consistency** (never reclaim a reachable object).
- Completeness** (reclaim every unreachable object - eventually).
- Consistency (also called safety) is an absolute requirement. Better no GC than an unsafe GC.
- But: safe automatic garbage collection is hard or impossible in a hybrid language environment (e.g. C++): pointers may masquerade as integers or other values.

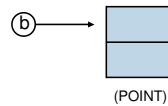
## Types

45

- Reference types; value of an entity is a reference.

Example:

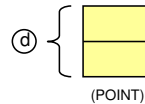
*b*: POINT



- Expanded types; value of an entity is an object.

Example:

*d*: **expanded** POINT



## Subobjects

47

- Expanded classes and entities support the notion of subobject.

```
class RECTANGLE_R
```

```
feature
```

```
corner1, corner2, corner3, corner4: POINT
```

```
...
```

```
end
```

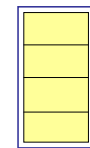
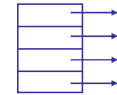
```
class RECTANGLE_E
```

```
feature
```

```
corner1, corner2, corner3, corner4:  
expanded POINT
```

```
...
```

```
end
```



## Expanded classes

46

- A class may also be declared as

**expanded class C**

... The rest as usual ...

- Then you can declare:

*a*: C

with the same effect as

*b*: **expanded** C

in the earlier syntax (still permitted, with same meaning).

## The meaning of expanded classes

48

- More than an implementation notion: a system modeling tool.
- Two forms of client relation:
  - Simple client
  - Expanded client
- What is the difference between these two statements?
  - A car has an originating factory.
  - A car has an engine.

## Basic types as expanded classes

49

**expanded class** *INTEGER* ...  
**expanded class** *BOOLEAN* ...  
**expanded class** *CHARACTER* ...  
**expanded class** *REAL* ...  
**expanded class** *DOUBLE* ...

*n*: *INTEGER*



51

End of lecture 4

## Infix and prefix operators

50

```
expanded class INTEGER feature  
  infix "+" (other: INTEGER): INTEGER is  
    -- Sum with other  
    do  
      ...  
    end  
  
  infix "*" (other: INTEGER): INTEGER is  
    -- Product by other  
    do  
      ...  
    end  
  
  prefix "-" : INTEGER is  
    -- Unary minus  
    do  
      ...  
    end  
  
  ...  
end
```

Calls are then of the form  $i + j$  rather than  $i.plus(j)$ .