Lecture 7: References and Assignments

Program for today

How do we modify objects?
How it all starts

Executing a system consists of creating a root object, which is an instance of a designated class from the system, called its root class, using a designated creation procedure of that class, called its root procedure.

Root creation procedure may:
- Create new objects
- Call features on them, which may create other objects
- Etc.

Executing a system

Root object

Root procedure

obj1

create obj1.r1

obj2

create obj2.r2

At any time during execution, there is a current object, on which the current feature is being executed.

Initially it is the root object.

During a "qualified" call \( x.f(a) \), the new current object is the one attached to \( x \).

At the end of such a call, the previous current object resumes its role.
Program for today

How do we modify objects?

Classes and objects

At run time: objects (software machines)

In the program text: classes

Each class describes a set of possible run-time objects.

Object structure

An object is made of fields

Each field is a value, which is either:

A basic value: integer, character, "real" number...
(known as an expanded value)

A reference to another object
Two kinds of type

Reference types; value of any entity is a reference.
Example:
\[
\text{b: \textit{STATION}}
\]

Expanded types; value of an entity is an object.
Example:
\[
\text{d: \textit{E\_STATION}}
\]

Expanded classes

A class may be declared as

\[
\text{expanded class \textit{E\_STATION}}
\]

... The rest as in \textit{STATION}...

Then any entity declared
\[
\text{d: \textit{E\_STATION}}
\]

has the expanded semantics just described.

Basic types as expanded classes

\[
\text{expanded class \textit{INTEGER} ...}
\]

... (internally: \textit{INTEGER\_32, INTEGER\_64} etc.)

\[
\text{expanded class \textit{BOOLEAN} ...}
\]

\[
\text{expanded class \textit{CHARACTER} ...}
\]

\[
\text{expanded class \textit{REAL} ...}
\]

... (internally: \textit{REAL\_32, REAL\_64} etc.)

\[
\text{n: \textit{INTEGER}}
\]
Initialization

Automatic initialization rules:
- 0 for numbers (integers, reals)
- "Null" character for characters
- False for boolean
- Void for references

These rules apply to:
- Fields (from class attributes), on object creation
- Local variables, on start of routine execution (includes Result)

References may cause cycles

Strings are objects

The `name` field is a reference field
Fields reflect attributes of the class

```plaintext
class POSITION
    feature - Access
        x: REAL  -- Horizontal position
        y: REAL  -- Vertical position
    end
end
```

Setting fields (in routines of the class)

```plaintext
class POSITION
    feature - Access
        x: REAL  -- Horizontal position
        y: REAL  -- Vertical position
    end

    feature - Element change
        set(xval, yval: REAL)
            require
                x_positive: xval >= 0
                y_positive: yval >= 0
            do
                ... end
            ensure
                x_set: x = xval
                y_set: y = yval
end
```

x := xval
y := yval
### What you may do

```plaintext
class METRO_STATION feature
  x, y: REAL  
  -- Coordinates of center of metro station
  size: REAL  
  -- Size of bounding square
  upper_left: POSITION  
  -- Upper-left position of bounding square
  adjust_positions  
  -- Set positions of bounding square
  do
    upper_left.set(x - size/2, y + size/2)
  end
end
```

### Feature calls

#### In POSITION
```
set(xval, yval: REAL)
do
end
```

#### In another class, e.g., METRO_STATION
```
upper_left: POSITION
adjust_positions  
  do
    upper_left.set(x - size/2, y + size/2)
  end
```

#### Qualified call
```
In class POSITION itself:
move(dx, dy: REAL)
require
  do
    [Please complete!]
  ensure
  end
```

#### Unqualified call
```
In another class, e.g., METRO_STATION
```

### The current object

At every moment during execution, operations are being performed on a current object.

Initially: the root object. Then:
- An unqualified call such as `set(u, v)` applies to the current object.
- A qualified call such as `x.set(u, v)` causes the object attached to `x` to become the current object. After the call, the previous current object becomes current again.

To denote the current object: use `Current`
Executing a system

Current object

Feature calls

---

At any time during execution, there is a **current object**, on which the current feature is being executed.

Initially it is the root object.

During a "qualified" call \( x.f(a) \), the new current object is the one attached to \( x \).

At the end of such a call, the previous current object resumes its role.

*General relativity*

In `METRO_STATION`

```plaintext
adjust_positions
  do
    upper_left := (x − size/2, y + size/2)
  end

upper_left: POSITION
```
The client relation

Because class `METRO_STATION` has a feature

```plaintext
upper_left: POSITION
```

(and calls of the form `upper_left . set(...)`) ,

`METRO_STATION` is a client of class `POSITION`

Client and inheritance, graphically

Entities

An entity is a name in the program that denotes possible run-time values

Some entities are constant

Others are variable:
  - Attributes
  - Local variables
Changing variable values: assignment

**source** is an expression:
- Call to a query:
  - position
  - upper_left, position
- Arithmetic or boolean expression:
  - \( a = (b + c) \)
  - \( (a < b) \) and \( (c = d) \)

**target** may be:
- An attribute
- Result in a function
- A "local variable" of a routine (not yet seen)

Assignment

Replaces a value by another

```
p
```

```
set(xval, yval: REAL)
```

```
do
  x := xval
  y := yval
end
```

Setting fields (in routines of the class)

```
class POSITION
feature - Access
  x: REAL
  -- Horizontal position
  y: REAL
  -- Vertical position
feature - Element change
  set(xval, yval: REAL): ---- Set coordinates to (xval, yval).
require
  x_positive: xval >= 0
  y_positive: yval >= 0
do
  x := xval
  y := yval
ensure
  x_set: x = xval
  y_set: y = yval
end
```
Do not confuse assignment with equality

\[
x := y
\]

if \( x = y \) then...

if \( x = \text{Current} \) then...

What to do with unreachable objects

Reference assignments may make some objects useless

Two possible approaches:
- Manual "free" (C++, Pascal)
- Automatic garbage collection (Eiffel, Oberon, Java, .NET)

The C programmer's view

Newsgroup posting by Ian Stephenson, 1993 (as cited in Object-Oriented Software Construction, 2nd edition):

I say a big NO! Leaving an unreferenced object around is BAD PROGRAMMING. Object pointers ARE like ordinary pointers — if you allocate an object you should be responsible for it, and free it when its finished with. (Didn't your mother always tell you to put your toys away when you'd finished with them?)
Arguments for automatic collection

Manual reclamation is dangerous for reliability.
- Wrong "frees" are among the most difficult bugs to detect and correct.

Manual reclamation is tedious.
Modern garbage collectors have acceptable performance overhead.
GC is tunable: disabling, activation, parameterization...

Properties of a garbage collector (GC)

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 in C-based languages.

Effect of an assignment

Reference types: reference assignment
Expanded types: value copy

class TWO_VALUES feature
  item : INTEGER
  right : TWO_VALUES
end

set(n: INTEGER; r: TWO_VALUES)
  -- Reset both fields
  do
  item := n
  end
  right := r
end

c t : TWO_VALUES
create t
  item = n
  right = r
  t.set(25, Void)
Assignment

```plaintext
class METRO_STATION feature
  location: POSITION
  name: STRING
  length: REAL

  set_all (p: POSITION; l: REAL; n: STRING)
  do
    location := p
    length := l
    name := n
  end
end
```

A linked list of strings: inserting at the end

```
<table>
<thead>
<tr>
<th>Item</th>
<th>Right</th>
<th>Item</th>
<th>Right</th>
<th>Item</th>
<th>Right</th>
</tr>
</thead>
<tbody>
<tr>
<td>Haldenegg</td>
<td>(LINKABLE)</td>
<td>Central</td>
<td>(LINKABLE)</td>
<td>Hauptbahnhof</td>
<td>(LINKABLE)</td>
</tr>
</tbody>
</table>
```

Inserting an item at the end

```
declare v: STRING
  -- Add v to end.
  -- Do not move cursor.
local p: LINKABLE
  do
    create p.make(v)
    if is_empty then
      first_element := p
      active := p
    else
      last_element.put_right(p)
      if after then active := p end
    end
    last_element := p
    count := count + 1
end
```
**LINKABLE cells**

```plaintext
class LINKABLE feature
  item: STRING  -- Value in this cell
right: LINKABLE  -- Cell, if any, to which this one is chained

put_right (other: like Current)
do
  right := other
ensure
  chained: right = other
end
end
```

**Local variables (in routines)**

A form of entity (they are also called "local entities")

Just declare them on entry to a routine:

```plaintext
r(...)
-- Header comment
require ...
local x: REAL
  m: METRO_STATION
do ...
  Can use x and m here...
ensure ...
end
```

Local variables include Result for a function.

**Exercise (uses loops)**

Reverse a list!

```
5  count
last_element
first_element

LINKED_LIST
Haldenegg  Central  Hauptbahnhof  Paradeplatz
```
Reading assignment for next 2 weeks

Control structures: chapter 7

What we have seen

• The current object
• Expanded vs reference types
• Assignment:
  ➢ For references
  ➢ For expanded values
• Linked data structures
• A glimpse of conditional instructions