Lecture 7: References and Assignments
No comment
Program for today

How do we modify objects?
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

create obj1.r1

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:

\[ b : STATION \]

**Expanded** types; value of an entity is an object.
Example:

\[ d : E\_STATION \]
Expanded classes

A class may be declared as

\[\text{expanded class } E_{-}\text{STATION} \]
\[\text{... The rest as in } \text{STATION} ...\]

Then any entity declared

\[d: E_{-} \text{STATION}\]

has the expanded semantics just described.
Basic types as expanded classes

expanded class \texttt{INTEGER} ...
   (internally: \texttt{INTEGER}_32, \texttt{INTEGER}_64 etc.)
expanded class \texttt{BOOLEAN} ...
expanded class \texttt{CHARACTER} ...
expanded class \texttt{REAL} ...
   (internally: \texttt{REAL}_32, \texttt{REAL}_64 etc.)

\( n: \texttt{INTEGER} \)
Initialization

Automatic initialization rules:

- 0 for numbers (integers, reals)
- "Null" character for characters
- False for booleans
- 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

- "Almaviva"
- "Figaro"
- "Susanna"

- landlord
- loved_one
- name

- landlord
- loved_one
- name

- landlord
- loved_one

Cycles in the diagram:
- O1
- O2
- O3
Strings are objects

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

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

  require
    x_positive: xval >= 0
    y_positive: yval >= 0

do
  x := xval
  y := yval

  ensure
    x_set: x = xval
    y_set: y = yval

end
```

Attributes are features of the class
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
         ...
      ensure
         x_set: x = xval
         y_set: 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
What you may do

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
    do
      upper_left . set (x - size/2, y + size/2)
    end

end
Feature calls

In class `POSITION` itself:

```plaintext
move (dx, dy: REAL)
    -- Move by dx horizontally, dy vertically
    require
        ...
        [Please complete!] ...
    do
        set (x + dx, y + dy)
    ensure
        ...
        [Please complete!] ...
    end
```

In another class, e.g. `METRO_STATION`

```plaintext
upper_left : POSITION
adjust_positions
    do
        upper_left.set (x - size/2, y + size/2)
    end
```

Qualified call

Unqualified call
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

Root object

Root procedure

create obj1.r1

create obj2.r2

create obj2.r2
Current object

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”
Feature calls

In *METRO_STATION*

\[\text{adjust_positions do}\]

\[\text{upper_left.set}(x - \text{size}/2, y + \text{size}/2)\]

\[\text{...}\]

\[\text{end}\]

\[\text{upper_left: POSITION}\]
The client relation

Because class \textit{METRO\_STATION} has a feature

\begin{verbatim}
    upper_left: POSITION
\end{verbatim}

(and calls of the form \texttt{upper\_left.set(...) },

\textit{METRO\_STATION} is a client of class \textit{POSITION}
Client and inheritance, graphically

Inheritance

Client

Intro. to Programming, lecture 7: References and assignments  26
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**

\[ \text{target} := \text{source} \]

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

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

Replaces a value by another

\[ p \]

\[
\begin{array}{c}
  x & 0 \\
  y & 0
\end{array}
\]

\[ p.set(2, 1) \]

\[
set(xval, yval: \text{REAL})
\]

\[
\begin{array}{c}
  x := xval \\
  y := yval
\end{array}
\]

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 (\texttt{`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
end
Do not confuse assignment with equality

\[
\begin{align*}
  x & \;:=\; y \\
  \text{if } x & \;=\; y \text{ then...} \\
  \text{if } x & \;=\; \text{Current then...}
\end{align*}
\]
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)
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
  set (n : INTEGER; r : TWO_VALUES)
    -- Reset both fields
    do
      item := n
      right := r
    end
end

t : TWO_VALUES
  ... create t
  ...
  t.set (25, Void)
Assignment

```java
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

- count: 4
- first_element
- active
- last_element
- Paradeplatz
- Haldenegg
- Central
- Hauptbahnhof

(item, right)

(LINKABLE)
Inserting an item at the end

\[ \text{extend} (v : \text{STRING}) \]

-- Add \( v \) to end.
-- Do not move cursor.

\[
\text{local} \quad p : \text{LINKABLE}[\text{STRING}]
\]

\[
\text{do}
\]

\[
\text{create} \quad p . \text{make} (v)
\]

if \( \text{is_empty} \) then

\[
\text{first_element} := p \quad \text{active} := p
\]

else

\[
\text{last_element}.\text{put_right} (p)
\]

if \( \text{after} \) then \( \text{active} := p \) end

end

\[
\text{last_element} := p
\]

\[
\text{count} := \text{count} + 1
\]

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)
        -- Put other to the right of current cell.
        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:

\[
\begin{align*}
    r(\ldots) & \quad \text{-- Header comment} \\
    \text{require} & \\
    \quad \ldots \\
    \text{local} & \\
    \quad x : \text{REAL} \\
    \quad m : \text{METRO\_STATION} \\
    \text{do} & \\
    \quad \ldots \text{ Can use } x \text{ and } m \text{ here } \ldots \\
    \text{ensure} & \\
    \quad \ldots \\
    \text{end} & \\
\end{align*}
\]

Local variables include \textit{Result} for a function
Exercise (uses loops)

Reverse a list!

(count)
(last_element)
(first_element)

Halden-egg → Central → Hauptbahnhof → Paradeplatz

(item right)
(LINKABLE)
(LINKED_LIST)
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