Lecture 7: References and Assignment
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.
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

---

A basic value: integer, character, “real” number...

(known as an expanded value)

A reference to another object

---

![Diagram](image)
Two kinds of type

Reference types; value of any entity is a reference.

\[ b: \text{STATION} \]

Expanded types; value of an entity is an object.

\[ d: \text{E-STATION} \]
Expanded classes

A class may be declared as

expanded class \textit{E\_STATION}
... The rest as in \textit{STATION} ...

Then any entity declared

\texttt{d: E\_STATION}

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

expanded class INTEGER ...
  (internally: INTEGER_32, INTEGER_64 etc.)
expanded class BOOLEAN ...
expanded class CHARACTER ...
expanded class REAL ...
  (internally: REAL_32, REAL_64 etc.)

n: 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
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
end

An attribute

Another attribute

Attributes are features of the class
Setting fields (in a routine of the class)

class
  POSITION

feature - Access
  x: REAL
    -- Horizontal position
  y: REAL
    -- Vertical position

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

feature - Access

\[ x : \text{REAL} \]
-- Horizontal position

\[ y : \text{REAL} \]
-- Vertical position

feature - Element change

\[ \text{set}(xval, yval : \text{REAL}) \]
-- Set coordinates to \([xval, yval]\).

require
\[ \text{x\_positive} : xval \geq 0 \]
\[ \text{y\_positive} : yval \geq 0 \]

do
\[ x := xval \]
\[ y := yval \]

ensure
\[ \text{x\_set} : x = xval \]
\[ \text{y\_set} : y = yval \]
end

Setting fields (in routines of the class)
class STATION feature

\( x, y \) : REAL
-- Coordinates of station center

\( \text{size} \) : REAL
-- Size of bounding square

\( \text{upper}_\text{left} \) : POSITION
-- Upper-left position of bounding square

\text{adjust}_\text{positions}
-- Set positions of bounding square

do
\text{upper}_\text{left}.\text{set}(x - \text{size}/2, y + \text{size}/2)
...
end

end
Feature calls

In `POSITION`, assume

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

  do
  end
```

In another class, e.g. `STATION`

```plaintext
upper_left : POSITION

adjust_positions

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

In `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
```
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 \( \text{set}(u, v) \) applies to the current object

- A qualified call such as \( x.\text{set}(u, v) \) 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 \textit{Current}
Executing a system

Root object

Root procedure

create obj1.r1

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 STATION:

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

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

Because class \textit{STATION} has a feature

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

(and calls of the form \textit{upper_left.set(\ldots)}),

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

![Diagram showing inheritance relationships]

Client

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

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

Replaces a value by another

\[ p \]

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

\[ p \text{.set}(2, 1) \]

The assignment statement is:

\[ \text{set}(x\text{val}, y\text{val}; \text{REAL}) \]

\[
\begin{array}{c}
do \\
\hline
x := x\text{val} \\
y := y\text{val} \\
end
\end{array}
\]
Setting fields (in routines of the class)

class

  \textit{POSITION}

feature - Access

  \begin{itemize}
    \item \textit{x}: REAL
      \begin{itemize}
        \item -- Horizontal position
      \end{itemize}
    \item \textit{y}: REAL
      \begin{itemize}
        \item -- Vertical position
      \end{itemize}
  \end{itemize}

feature - Element change

  \textit{set}(xval, yval: REAL)

    \begin{itemize}
      \item -- Set coordinates to [xval, yval].
    \end{itemize}

    \textbf{require}

    \begin{itemize}
      \item \textit{x\_positive}: xval \geq 0
      \item \textit{y\_positive}: yval \geq 0
    \end{itemize}

    \textbf{do}

    \begin{itemize}
      \item \textit{x} := xval
      \item \textit{y} := yval
    \end{itemize}

    \textbf{ensure}

    \begin{itemize}
      \item \textit{x\_set}: x = xval
      \item \textit{y\_set}: y = yval
    \end{itemize}
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
  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

class \textit{STATION}  
feature
    \text{location}: \text{POSITION}
    \text{name}: \text{STRING}
    \text{length}: \text{REAL}

\text{set\_all}(p: \text{POSITION}; l: \text{REAL}; n: \text{STRING})
    \text{do}
        \text{location} := p
        \text{length} := l
        \text{name} := n
    \text{end}
\text{end}
A linked list of strings: inserting at the end

- First element: Haldenegg
- Last element: Paradeplatz
- Count: 3

Diagram: A linked list with nodes labeled Haldenegg, Central, and Hauptbahnhof, connected with arrows indicating the 'item' and 'right' fields.
Inserting an item at the end

extend (v: STRING)
  -- Add v to end.
  -- Do not move cursor.
  
  local
    p: LINKABLE [STRING]
  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
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:

\[ r(...) \]

```
require
...
local
  x: REAL
  m: STATION
do
  ... Can use \( x \) and \( m \) here ...
ensure
  ...
end
```

Local variables include *Result* for a function
Reverse a list!

Exercise (uses loops)

Halden-egg -> Central -> Hauptbahnhof -> Paradeplatz

(count)

(last_element)

(first_element)

(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