Einführung in die Programmierung
Introduction to Programming

Prof. Dr. Bertrand Meyer

Lecture 12: More on references and the dynamic model
Reading assignment for next week

Chapters on

- Syntax (12)
- Inheritance (17)
Purpose of this lecture

Few really new concepts, but gain a better understanding of the tricky matter of references

Reminder on garbage collection and associated concepts
Assignment

Replaces a value by another

\[ x = 0 \]
\[ y = 0 \]

\( p \text{.set_coordinates}(2, 1) \)
class POSITION

feature - Access

  x: REAL
  -- Horizontal position

  y: REAL
  -- Vertical position

feature - Element change

  set_coordinates (xval, yval: REAL)
  -- Set coordinates to (xval', yval').

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

  do
    x := xval
    y := yval
  end

  ensure
    x_set: x = xval
    y_set: y = yval
end
Effect of an assignment

Reference types: reference assignment
Expanded types: value copy

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

3
6

item
right

item := n
right := r

Create t

void

25

set_fields (25, Void)
Abstraction and client privileges

If class $A$ has an attribute $x$, what may a client class $C$ do with $a1.x$ for $a1$ of type $A$?

Read access if attribute is exported

- $a1.x$ is an expression!
- An assignment $a1.x := v$ would be syntactically illegal!

(It would assign to an expression, like $a + b := v$)
An object has an **interface**
An object has an implementation

- animate
- append
- prepend
- first
- last
- count
- stations
Applying abstraction principles

Beyond read access: full or restricted write, through exported procedures.

Full write privileges: \texttt{\textit{set\_x}} procedure, e.g.

\begin{verbatim}
set_temperature (u: REAL)
  -- Set temperature value to \textit{u}.
  do
    temperature := u
  end
\end{verbatim}

Clients will use

\textit{x}.\texttt{set\_temperature} (21.5)
Abstraction and client privileges

If class \( A \) has an attribute \( x \), what may a client class \( C \) do with \( a1.x \) for \( a1 \) of type \( A \)?

The attribute may be:

- Secret
- Read-only
- Read, restricted write
- Full write

\( a1.x \) invalid

\( a1.x \) permitted in \( C \) (for access)
Taking full advantage of a setter command

\[\text{set_temperature}(u : \text{REAL})\]

\[\text{-- Set temperature value to } u.\]

**require**

\[\text{not_under_minimum: } u \geq -273\]
\[\text{not_above_maximum: } u \leq 2000\]

**do**

\[\text{temperature} := u\]

**update_database**

**ensure**

\[\text{temperature_set: temperature} = u\]

**end**
Possible client privileges

If class $A$ has an attribute $att$, what may a client class $C$ do with $a1.att$ for $a1$ of type $A$?

The attribute may be:

- Secret
- Read-only
- Read, restricted write
- Full write

Possible actions:

- Modify through $a1.some_procedure$ (for access)
- Modify through $a1.set_att(v)$

$a1.att$ invalid

$a1.att$ permitted in $C$ (for access)
C++, Java, C# convention

An attribute is either private or public

If public, it is available for both read and write!

- \( v := a1.x \)
- \( a1.x := v \)

This is almost never acceptable

Consequence: shadow every attribute with "getter function"

This can be pretty tedious!

In Eiffel: may export an attribute, but

- Only for reading
- Without the information that it is an attribute: it could be a function (Uniform Access principle)
Having it both ways

It is possible to define a query as

```
temperature: REAL assign set_temperature
```

Then the syntax

```
x.temperature := 21.5
```

is accepted as an abbreviation for

```
x.set_temperature(21.5)
```

Retains contracts and any other supplementary operations

C# has a notion of “property” which pursues the same goal.
A linked list of strings: inserting at the end

- **first_element**
- **last_element**
- **active**
- **count**: 4

Haldenegg → Central → Hauptbahnhof

(inserting at the end) (LINKABLE) (LINKABLE) (LINKABLE)
Inserting an item at the end

\[ \text{extend} (v: \text{STRING}) \]
\[
\begin{align*}
\text{-- Add } v \text{ to end.} \\
\text{-- Do not move cursor.}
\end{align*}
\]

local
\[
p: \text{LINKABLE}[\text{STRING}] \\
\]
do
\[
\begin{align*}
\text{create } p.\text{make}(v) \\
\text{if } \text{is_empty} \text{ then} \\
\text{first_element} := p \\
\text{active} := p \\
\text{else} \\
\text{last_element}.\text{put_right}(p) \\
\text{if } \text{after} \text{ then active} := p \text{ end}
\end{align*}
\]
end
\[
\begin{align*}
\text{last_element} := p \\
\text{count} := \text{count} + 1
\end{align*}
\]
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
Exercise (uses loops)

Reverse a list!

(LINKED_LIST)

3

count

last_element

first_element

Halden-egg

item right

(LINKABLE)

Central

Hauptbahnhof

Paradeplatz
Reversing a list

1 2 3 4 5

1 2 3 4 5

---

21
Reversing a list

from pivot := first_element
first_element := Void
until pivot = Void loop
  i := first_element
  first_element := pivot
  pivot := pivot.right
  first_element.put_right(i)
end
Reversing a list

from

pivot := first_element
first_element := Void
until pivot = Void loop

i := first_element
first_element := pivot
pivot := pivot.right
first_element.put_right(i)

end
Reversing a list

```plaintext
pivot := first_element
first_element := Void

until pivot = Void loop
  i := first_element
  first_element := pivot
  pivot := pivot.right
  first_element.put_right(i)
end
```
Reversing a list

pivot := first_element
first_element := Void

until pivot = Void loop
  i := first_element
  first_element := pivot
  pivot := pivot.right
  first_element.put_right(i)
end
Reversing a list

```plaintext
pivot := first_element
first_element := Void
until pivot = Void loop
  i := first_element
  first_element := pivot
  pivot := pivot.right
  first_element.put_right(i)
end
```
The loop invariant

Invariant: from first_element following right, initial items in inverse order; from pivot, rest of items in original order

pivot := first_element
first_element := Void
until pivot = Void loop

i := first_element
first_element := pivot
pivot := pivot.right
first_element.put_right(i)
end
The trouble with reference assignment

A comfortable mode of reasoning:

-- Here SOME_PROPERTY holds of \( a \)

“Apply SOME_OPERATION to \( b \)”

-- Here SOME_PROPERTY still holds of \( a \)

This applies to “expanded” values, e.g. integers

-- Here \( P(a) \) holds

\( OP(b) \)

-- Here \( P(a) \) still holds of \( a \)
Dynamic aliasing

\[ a, b: \text{LINKABLE} \ [\text{STRING}] \]

create \( a \)....

\( a.\text{put} \(" Haldenegg"\) \)

\( b := a \)

-- Here \( a.\text{item} \) has value "Haldenegg"

\( b.\text{put} \(" Paradeplatz"\) \)

-- Here \( a.\text{item} \) has value ??????
On the other hand...

-- I heard that the boss’s cousin earns less than 50,000 francs a year

“Raise Caroline’s salary by 1 franc”

-- ?????

Metaphors:
  - “The beautiful daughter of Leda”
  - “Menelas’s spouse”
  - “Paris’s lover”

= Helen of Troy
Practical advice

Reference assignment is useful

It’s also potentially tricky

As much as possible, leave it to specialized libraries of general data structures
Variants of assignment and copy

Reference assignment (a and b of reference types):
\[ b := a \]

Object duplication (shallow):
\[ c := a.twin \]

Object duplication (deep):
\[ d := a.deep_twin \]

Also: shallow field-by-field copy (no new object is created):
\[ e.copy(a) \]
Shallow and deep cloning

Initial situation:

Result of:

\[ b := a \]

\[ c := a.twin \]

\[ d := a.deep_twin \]
Where do these mechanisms come from?

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

ANY

A

D

NONE

B

C

E

Inherits from
**A related mechanism: Persistence**

```
\texttt{a.store(file)}
```

```none
....
```
\texttt{b := retrieved(file)}
```

Storage is automatic.
Persistent objects identified individually by keys.

These features come from the library class \texttt{STORABLE}.
Objects and references

States of a reference:

- Void
- Attached

Operations on references:

- create \( p \)
- \( p := q \) (where \( q \) is attached)
- \( p := \text{Void} \)
- \( p := q \) (where \( q \) is void)
- if \( p = \text{Void} \) then ...
The object-oriented form of call

\[ \text{some\_target.some\_feature(some\_arguments)} \]

For example:

\[ \text{Paris.display} \]
\[ \text{Line6.extend(Station\_Parade\_Platz)} \]
\[ x := a.\text{plus}(b) \]

????????
Infix and prefix operators

In

\[ a - b \]

the \(-\) operator is "infix"
(written between operands)

In

\[ -b \]

the \(-\) operator is "prefix"
(written before the operand)
Operator features

expanded class INTEGER feature

\texttt{plus alias "+" (other : INTEGER): INTEGER is}
\hspace{1cm} \text{-- Sum with \textit{other}}
\hspace{1cm} \texttt{do ... end}
\texttt{times alias "*" (other : INTEGER): INTEGER is}
\hspace{1cm} \text{-- Product by \textit{other}}
\hspace{1cm} \texttt{do ... end}
\texttt{minus alias "-" : INTEGER is}
\hspace{1cm} \text{-- Unary minus}
\hspace{1cm} \texttt{do ... end}
\hspace{1cm} ...
\hspace{1cm} end

Calls such as \texttt{i.plus(j)} can now be written \texttt{i + j}
What we have seen

- Playing with references:
  - list reversal
- Dynamic aliasing and the difficulties of pointers & references
- Overall inheritance structure
- Copy, clone and storage operations
- Persistence closure
- Infix & prefix operators
Reading assignment for next week

Chapters on

- Syntax (12)
- Inheritance (17)