Abstract Data Types (ADTs), Classes, and Objects

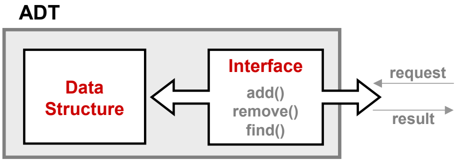
Readings: OOSC2 Chapters 6, 7, 8

EECS3311: Software Design
Fall 2017

CHEN-WEI WANG

Abstract Data Types (ADTs)

- Given a problem, you are required to filter out *irrelevant* details.
- The result is an *abstract data type (ADT)*, whose *interface* consists of a list of (unimplemented) operations.

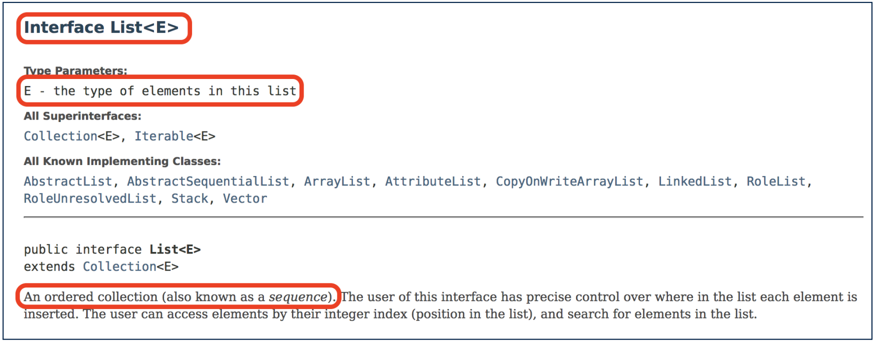


- *Supplier*'s *Obligations*:
  - Implement all operations
  - Choose the “right” data structure (DS)
- *Client*'s *Benefits*:
  - *Correct* output
  - Efficient performance
- The internal details of an *implemented ADT* should be **hidden**.

Building ADTs for Reusability

- ADTs are *reusable software components*
  e.g., Stacks, Queues, Lists, Dictionaries, Trees, Graphs
- An ADT, once thoroughly tested, can be reused by:
  - Suppliers of other ADTs
  - Clients of Applications
- As a supplier, you are obliged to:
  - *Implement* given ADTs using other ADTs (e.g., arrays, linked lists, hash tables, etc.)
  - *Design* algorithms that make use of standard ADTs
- For each ADT that you build, you ought to be clear about:
  - The list of supported operations (i.e., *interface*)
    - The interface of an ADT should be *more than* method signatures and natural language descriptions:
    - How are clients supposed to use these methods? [*preconditions*]
    - What are the services provided by suppliers? [*postconditions*]
  - Time (and sometimes space) *complexity* of each operation

Why Java Interfaces Unacceptable ADTs (1)



Interface List<E>

Type Parameters:

`E - the type of elements in this list`

All Superinterfaces:

`Collection<E>, Iterable<E>`

All Known Implementing Classes:

`AbstractList, AbstractSequentialList, ArrayList, AttributeList, CopyOnWriteArrayList, LinkedList, RoleList, RoleUnresolvedList, Stack, Vector`

```
public interface List<E>
extends Collection<E>
```

An ordered collection (also known as a *sequence*). The user of this interface has precise control over where in the list each element is inserted. The user can access elements by their integer index (position in the list), and search for elements in the list.

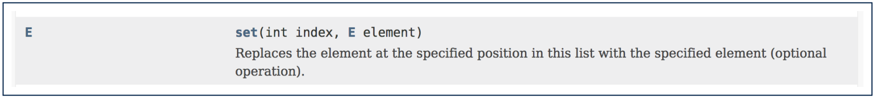
It is useful to have:

- A *generic collection class* where the ***homogeneous type*** of elements are parameterized as `E`.
- A reasonably ***intuitive overview*** of the ADT.

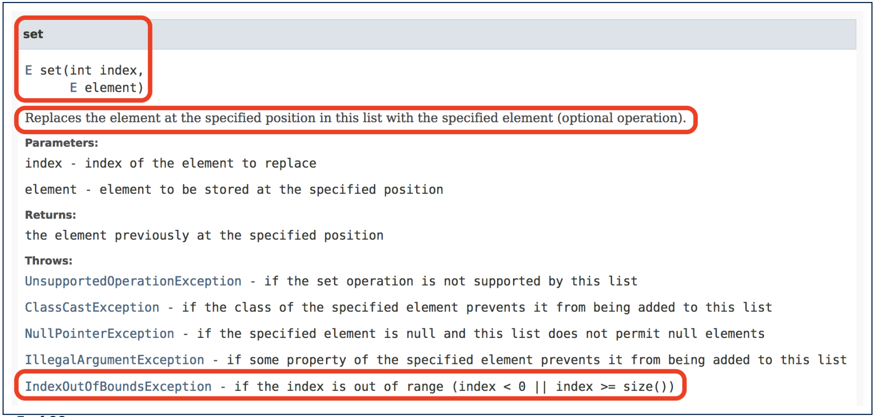
Java 8 List API

Why Java Interfaces Unacceptable ADTs (2)

Methods described in a ***natural language*** can be ***ambiguous***:



```
E          set(int index, E element)
```
Replaces the element at the specified position in this list with the specified element (optional operation).



set

```
E set(int index,
      E element)
```

Replaces the element at the specified position in this list with the specified element (optional operation).

Parameters:

`index - index of the element to replace`

`element - element to be stored at the specified position`

Returns:

`the element previously at the specified position`

Throws:

`UnsupportedOperationException - if the set operation is not supported by this list`

`ClassCastException - if the class of the specified element prevents it from being added to this list`

`NullPointerException - if the specified element is null and this list does not permit null elements`

`IllegalArgumentException - if some property of the specified element prevents it from being added to this list`

`IndexOutOfBoundsException - if the index is out of range (index < 0 || index >= size())`

Why Eiffel Contract Views are ADTs (1)

```
class interface ARRAYED_CONTAINER
feature -- Commands
  assign_at (i: INTEGER; s: STRING)
      -- Change the value at position 'i' to 's'.
    require
      valid_index: 1 <= i and i <= count
    ensure
      size_unchanged:
        imp.count = (old imp.twin).count
      item_assigned:
        imp [i] ~ s
      others_unchanged:
        across
          1 |..| imp.count as j
        all
          j.item /= i implies imp [j.item] ~ (old imp.twin) [j.item]
        end
  count: INTEGER
invariant
  consistency: imp.count = count
end -- class ARRAYED_CONTAINER
```

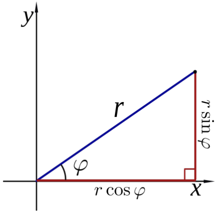
Why Eiffel Contract Views are ADTs (2)

Even better, the direct correspondence from Eiffel operators to logic allow us to present a *precise behavioural* view.

```
                ARRAYED_CONTAINER

feature -- Commands
  assign_at (i: INTEGER; s: STRING)
     -- Change the value at position 'i' to 's'.
   require
     valid_index: 1 ≤ i ≤ count
   ensure
     size_unchanged: imp.count = (old imp.twin).count
     item_assigned: imp[i] ~ s
     others_unchanged: ∀j : 1 ≤ j ≤ imp.count : j ≠ i ⇒imp[j] ~ (old imp.twin) [j]

feature -- { NONE }
  -- Implementation of an arrayed-container
  imp: ARRAY[STRING]

invariant
consistency: imp.count = count
```

Uniform Access Principle (1)

- We may implement `Point` using two representation systems:



  - The *Cartesian system* stores the *absolute* positions of `x` and `y`.
  - The *Polar system* stores the *relative* position: the angle (in radian) `phi` and distance `r` from the origin (0.0).
- How the `Point` is implemented is irrelevant to users:
  - **Imp. 1**: Store `x` and `y`. [Compute `r` and `phi` on demand]
  - **Imp. 2**: Store `r` and `phi`. [Compute `x` and `y` on demand]
- As far as users of a `Point` object `p` is concerned, having a *uniform access* by always being able to call **`p.x`** and **`p.y`** is what matters, despite **Imp. 1** or **Imp. 2** being current strategy.

Uniform Access Principle (2)

```
class
  POINT
create
  make_cartisian, make_polar
feature -- Public, Uniform Access to x- and y-coordinates
  x : REAL
  y : REAL
end
```

- A class `Point` declares how users may access a point: either get its *x* coordinate or its *y* coordinate.
- We offer two possible ways to instantiating a 2-D point:
  - `make_cartisian (nx: REAL; ny: REAL)`
  - `make_polar (nr: REAL; np: REAL)`
- Features `x` and `y`, from the client's point of view, cannot tell whether it is implemented via:
  - ***Storage*** [`x` and `y` stored as real-valued ***attributes***]
  - ***Computation*** [`x` and `y` defined as ***queries*** returning real values]

Uniform Access Principle (3)

Let's say the supplier decides to adopt strategy **Imp. 1**.

```
class POINT -- Version 1
feature -- Attributes
 x : REAL
 y : REAL
feature -- Constructors
 make_cartisian(nx: REAL; nx: REAL)
   do
     x := nx
     y := ny
   end
end
```

- Attributes `x` and `y` represent the *Cartesian system*
- A client accesses a point `p` via **`p.x`** and **`p.y`**.
  - ***No Extra Computations***: just returning current values of `x` and `y`.
- However, it's harder to implement the other constructor: the body of `make_polar (nr: REAL; np: REAL)` has to compute and store `x` and `y` according to the inputs `nr` and `np`.

Uniform Access Principle (4)

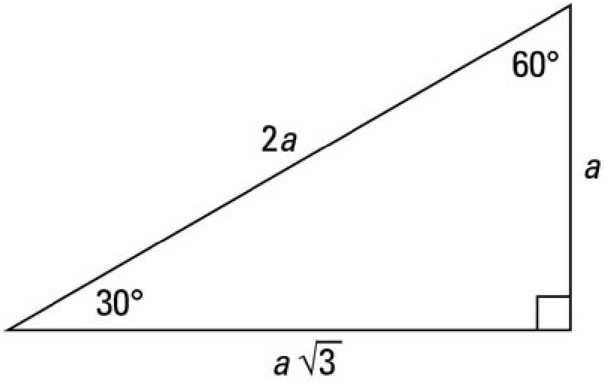
Let's say the supplier decides (*secretly*) to adopt strategy **Imp. 2**.

```
class POINT -- Version 2
feature -- Attributes
  r : REAL
  p : REAL
feature -- Constructors
  make_polar(nr: REAL; np: REAL)
    do
      r := nr
      p := np
    end
feature -- Queries
  x : REAL do Result := r × cos(p) end
  x : REAL do Result := r × sin(p) end
end
```

- Attributes `r` and `p` represent the *Polar system*
- A client **still** accesses a point `p` via **`p.x`** and **`p.y`**.
  - ***Extra Computations***: computing `x` and `y` according to the current values of `r` and `p`.

Uniform Access Principle (5.1)

Let's consider the following scenario as an example:



Note: $360^\circ = 2\pi$

Uniform Access Principle (5.2)

```
1  test_points: BOOLEAN
2    local
3      A, X, Y: REAL
4      p1, p2: POINT
5    do
6      comment("test: two systems of points")
7      A := 5; X := $A \times \sqrt{3}$; Y := A
8      create {POINT} p1.make_cartisian (X, Y)
9      create {POINT} p2.make_polar ($2 \times A$, $\frac{1}{6}\pi$)
10     Result := p1.x = p2.x and p1.y = p2.y
11   end
```

- If strategy **Imp. 1** is adopted:
  - **L8** is computationally cheaper than **L9**. [`x` and `y` attributes]
  - **L10** requires no computations to access `x` and `y`.

  If strategy **Imp. 2** is adopted:
  - **L9** is computationally cheaper than **L8**. [`r` and `p` attributes]
  - **L10** requires computations to access `x` and `y`.

Uniform Access Principle (6)

The *Uniform Access Principle* :

- Allows clients to use services (e.g., `p.x` and `p.y`) regardless of how they are implemented.
- Gives suppliers complete freedom as to how to implement the services (e.g., Cartesian vs. Polar).
  - No right or wrong implementation; it depends!
  - Choose for ***storage*** if the services are frequently accessed and their computations are expensive.
    e.g. `balance` of a bank involves a large number of accounts
    ⇒ Implement `balance` as an attribute
  - Choose for ***computation*** if the services are **not** keeping their values in sync is complicated.
    e.g., update `balance` upon a local deposit or withdrawal
    ⇒ Implement `balance` as a query
- Whether it's storage or computation, you can always change *secretly* , since the clients' access to the services is *uniform* .

Generic Collection Class: Motivation (1)

```
class STRING_STACK
feature {NONE} -- Implementation
  imp: ARRAY[ STRING ] ; i: INTEGER
feature -- Queries
  count: INTEGER do Result := i end
     -- Number of items on stack.
  top: STRING do Result := imp [i] end
     -- Return top of stack.
feature -- Commands
  push (v: STRING ) do imp[i] := v; i := i + 1 end
     -- Add 'v' to top of stack.
  pop do i := i - 1 end
     -- Remove top of stack.
end
```

- Does how we implement integer stack operations (e.g., `top`, `push`, `pop`) depends on features specific to element type `STRING` (e.g., `at`, `append`)? [***NO!***]
- How would you implement another class `ACCOUNT_STACK`?

Generic Collection Class: Motivation (2)

```
class ACCOUNT _STACK
feature {NONE} -- Implementation
  imp: ARRAY[ ACCOUNT ] ; i: INTEGER
feature -- Queries
  count: INTEGER do Result := i end
      -- Number of items on stack.
  top: ACCOUNT do Result := imp [i] end
      -- Return top of stack.
feature -- Commands
  push (v: ACCOUNT ) do imp[i] := v; i := i + 1 end
      -- Add 'v' to top of stack.
  pop do i := i - 1 end
      -- Remove top of stack.
end
```

- Does how we implement integer stack operations (e.g., `top`, `push`, `pop`) depends on features specific to element type `ACCOUNT` (e.g., `deposit`, `withdraw`)? [***NO!***]

Generic Collection Class: Supplier

- Your design ***"smells"*** if you have to create an ***almost identical*** new class (hence *code duplicates*) for every stack element type you need (e.g., `INTEGER`, `CHARACTER`, `PERSON`, etc.).
- Instead, as **supplier**, use *G* to *parameterize* element type:

```
class STACK [G]
feature {NONE} -- Implementation
  imp: ARRAY[ G ] ; i: INTEGER
feature -- Queries
  count: INTEGER do Result := i end
      -- Number of items on stack.
  top: G do Result := imp [i] end
      -- Return top of stack.
feature -- Commands
  push (v: G ) do imp[i] := v; i := i + 1 end
      -- Add 'v' to top of stack.
  pop do i := i - 1 end
      -- Remove top of stack.
end
```

Generic Collection Class: Client (1.1)

As **client**, declaring `ss: STACK[ STRING ]` instantiates every occurrence of `G` as `STRING`.

```
class STACK [~~G~~ STRING]
feature {NONE} -- Implementation
  imp: ARRAY[ ~~G~~ STRING ] ; i: INTEGER
feature -- Queries
  count: INTEGER do Result := i end
     -- Number of items on stack.
  top: ~~G~~ STRING do Result := imp [i] end
     -- Return top of stack.
feature -- Commands
  push (v: ~~G~~ STRING) do imp[i] := v; i := i + 1 end
     -- Add 'v' to top of stack.
  pop do i := i - 1 end
     -- Remove top of stack.
end
```

Generic Collection Class: Client (1.2)

As **client**, declaring `ss: STACK[ ACCOUNT ]` instantiates every occurrence of `G` as `ACCOUNT`.

```
class STACK [~~G~~ ACCOUNT]
feature {NONE} -- Implementation
  imp: ARRAY[ ~~G~~ ACCOUNT ] ; i: INTEGER
feature -- Queries
  count: INTEGER do Result := i end
      -- Number of items on stack.
  top:  ~~G~~ ACCOUNT  do Result := imp [i] end
      -- Return top of stack.
feature -- Commands
  push (v: ~~G~~ ACCOUNT ) do imp[i] := v; i := i + 1 end
      -- Add 'v' to top of stack.
  pop do i := i - 1 end
      -- Remove top of stack.
end
```

Generic Collection Class: Client (2)

As **client**, instantiate the type of `G` to be the one needed.

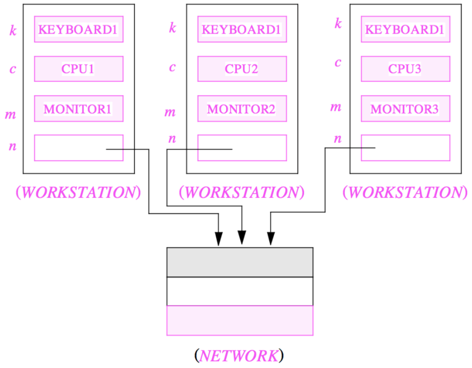
```
 1 test_stacks: BOOLEAN
 2   local
 3     ss: STACK[STRING] ; sa: STACK[ACCOUNT]
 4     s: STRING ; a: ACCOUNT
 5   do
 6     ss.push("A")
 7     ss.push(create {ACCOUNT}.make ("Mark", 200))
 8     s := ss.top
 9     a := ss.top
10     sa.push(create {ACCOUNT}.make ("Alan", 100))
11     sa.push("B")
12     a := sa.top
13     s := sa.top
14   end
```

- **L3** commits that `ss` stores `STRING` objects only.
  - **L8** and **L10** ***valid***; **L9** and **L11** ***invalid***.
- **L4** commits that `sa` stores `ACCOUNT` objects only.
  - **L12** and **L14** ***valid***; **L13** and **L15** ***invalid***.

Expanded Class: Modelling

- We may want to have objects which are:
  - Integral parts of some other objects
  - ***Not*** shared among objects

  e.g., Each workstation has its own CPU, monitor, and keyword.
  All workstations share the same network.



Expanded Class: Programming (2)

```
class KEYBOARD ... end class CPU ... end
class MONITOR ... end class NETWORK ... end
class WORKSTATION
  k: expanded KEYBOARD
  c: expanded CPU
  m: expanded MONITOR
  n: NETWORK
end
```

Alternatively:

```
expanded class KEYBOARD ... end
expanded class CPU ... end
expanded class MONITOR ... end
class NETWORK ... end
class WORKSTATION
  k: KEYBOARD
  c: CPU
  m: MONITOR
  n: NETWORK
end
```

Expanded Class: Programming (3)

```
expanded class
  B
feature
  change_i (ni: INTEGER)
    do
      i := ni
    end
feature
  i: INTEGER
end
```

```
1  test_expanded: BOOLEAN
2    local
3      eb1, eb2: B
4    do
5      Result := eb1.i = 0 and eb2.i = 0
6      check Result end
7      Result := eb1 = eb2
8      check Result end
9      eb2.change_i (15)
10     Result := eb1.i = 0 and eb2.i = 15
11     check Result end
12     Result := eb1 /= eb2
13     check Result end
14   end
```

- **L5**: object of expanded type is automatically initialized.
- **L9 & L10**: no sharing among objects of expanded type.
- **L7 & L12**: $=$ between expanded objects compare their contents.

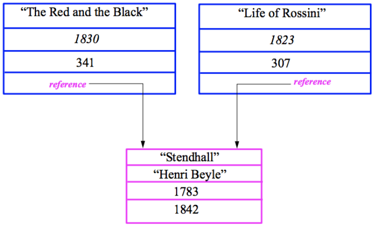
Reference vs. Expanded (1)

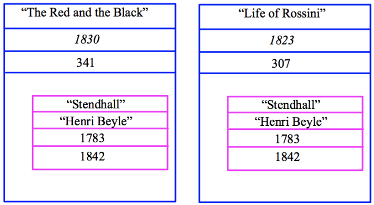
- Every entity must be declared to be of a certain type (based on a class).
- Every type is either ***referenced*** or ***expanded***.
- In ***reference*** types:
  - `y` denotes ***a reference*** to some object
  - `x := y` attaches `x` to same object as does `y`
  - `x = y` compares references
- In ***expanded*** types:
  - `y` denotes ***some object*** (of expanded type)
  - `x := y` copies contents of `y` into `x`
  - `x = y` compares contents [x ~ y]

Reference vs. Expanded (2)

Problem: Every published book has an author. Every author may publish more than one books. Should the author field of a book ***reference***-typed or ***expanded***-typed?

reference-typed author	***expanded***-typed author

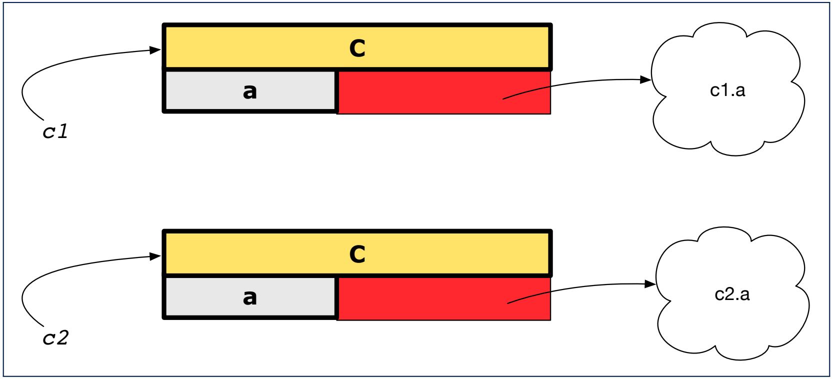




Copying Objects

Say variables `c1` and `c2` are both declared of type `C`. [*`c1, c2: C`*]

- There is only one attribute `a` declared in class `C`.
- `c1.a` and `c2.a` may be of either:
  - ***expanded*** type or
  - ***reference*** type

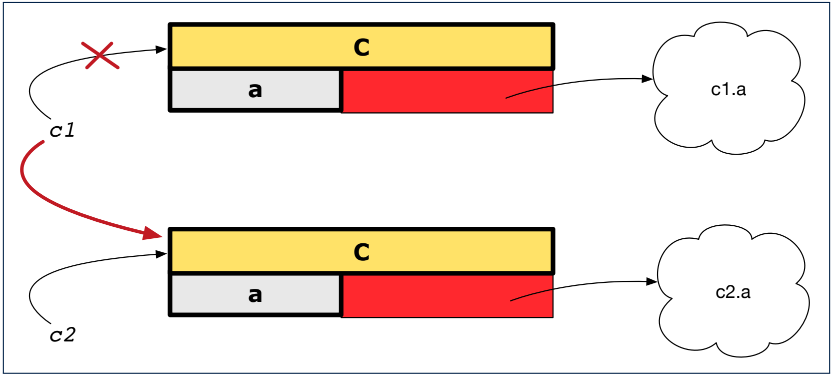


Copying Objects: Reference Copy

Reference Copy

`c1 := c2`

- Copy the address stored in variable `c2` and store it in `c1`.
  ⇒ Both `c1` and `c2` point to the same object.
  ⇒ Updates performed via `c1` also visible to `c2`. [***aliasing***]



Copying Objects: Shallow Copy

Shallow Copy

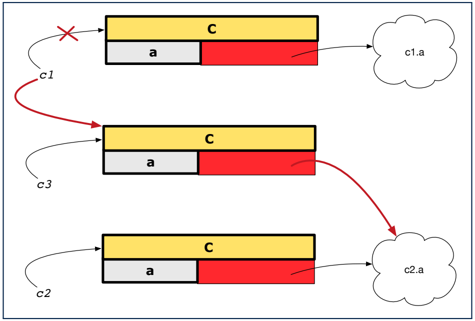
`c1 := c2.twin`

- Create a temporary, behind-the-scene object `c3` of type `C`.
- Initialize each attribute `a` of `c3` via ***reference copy***: `c3.a := c2.a`
- Make a ***reference copy*** of `c3`: `c1 := c3`

  ⇒ `c1` and `c2` ***are not*** pointing to the same object. [`c1 /= c2`]

  ⇒ `c1.a` and `c2.a` ***are*** pointing to the same object.

  ⇒ ***Aliasing*** still occurs: at 1st level (i.e., attributes of `c1` and `c2`)



Copying Objects: Deep Copy

Deep Copy

`c1 := c2.deep_twin`

- Create a temporary, behind-the-scene object `c3` of type `C`.
- *Recursively* initialize each attribute `a` of `c3` as follows:

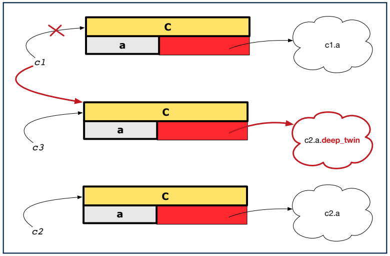
  Base Case: `a` is expanded (e.g., `INTEGER`). ⇒ `c3.a := c2.a`.

  Recursive Case: `a` is referenced. ⇒ `c3.a := c2.a.deep_twin`
- Make a ***reference copy*** of `c3`: `c1 := c3`

  ⇒ `c1` and `c2` ***are not*** pointing to the same object.

  ⇒ `c1.a` and `c2.a` ***are not*** pointing to the same object.

  ⇒ ***No aliasing*** occurs at any levels.



Copying Objects: Example

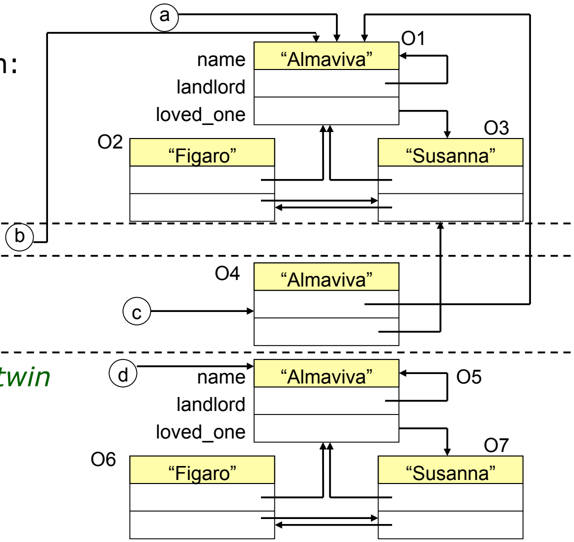
- Initial situation:

- Result of:

b := *a*

c := *a.twin*

d := *a.deep_twin*



Index (1)

Index (2)

Index (3)