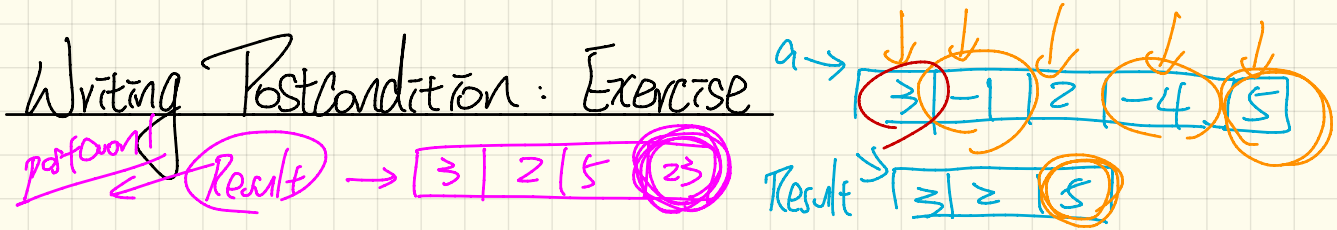
Monday January 28

Lecture 7

Writing Postcondition: Exercise



all_positive_values (a: ARRAY[INTEGER]): ARRAY[INTEGER]

require
a contains no duplicate.

ensure

post_con_1:
across Result as x
all
x.item > 0 and
a.has(x.item)
end

post_con_2:
across a as x
all
x.item > 0 implies Result.has(x.item)
end
a.has(x.item)
occurrences

Result → 3 2 5 3

Result → 1

Result → 3

Result → 4 1 67 5 9

Result → 1

S

$\{x \mid x \in a \cdot x > 0\}$ = $\{y \mid y \in \text{Result}\}$

T

all elements in a

pos.

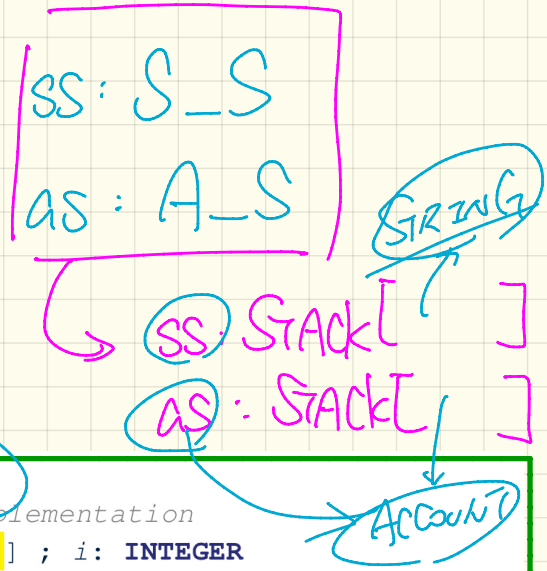
$T \subseteq S$

$S \subseteq T$

Stack of Strings vs. Stack of Accounts

```
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
```

```
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
```



A Generic Stack

Supplier

```
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
```

Client

```
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
```

```
class MY_COLLECTION [G]

    imp : ARRAY [G]

end
```

```
s.push ("A")
s.push (2)
s.push (create {BST[...]})
```

↓ 100 kinds of elements in stack

s : MY_COLLECTION [ANY]

~~s.pop~~. deposit

if s.top **instanceof** Account

elseif s.top **type of** STRING

Information Hiding Principle

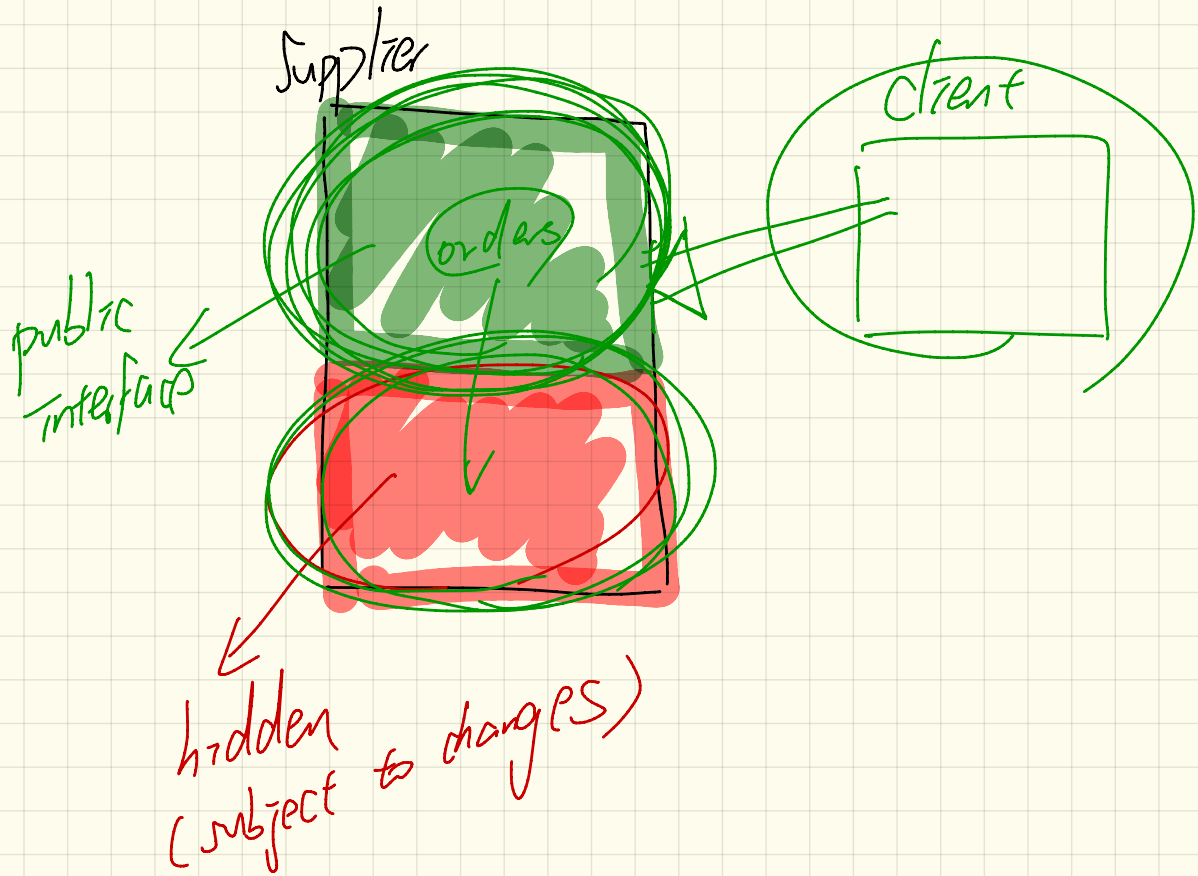
Client:

Supplier:

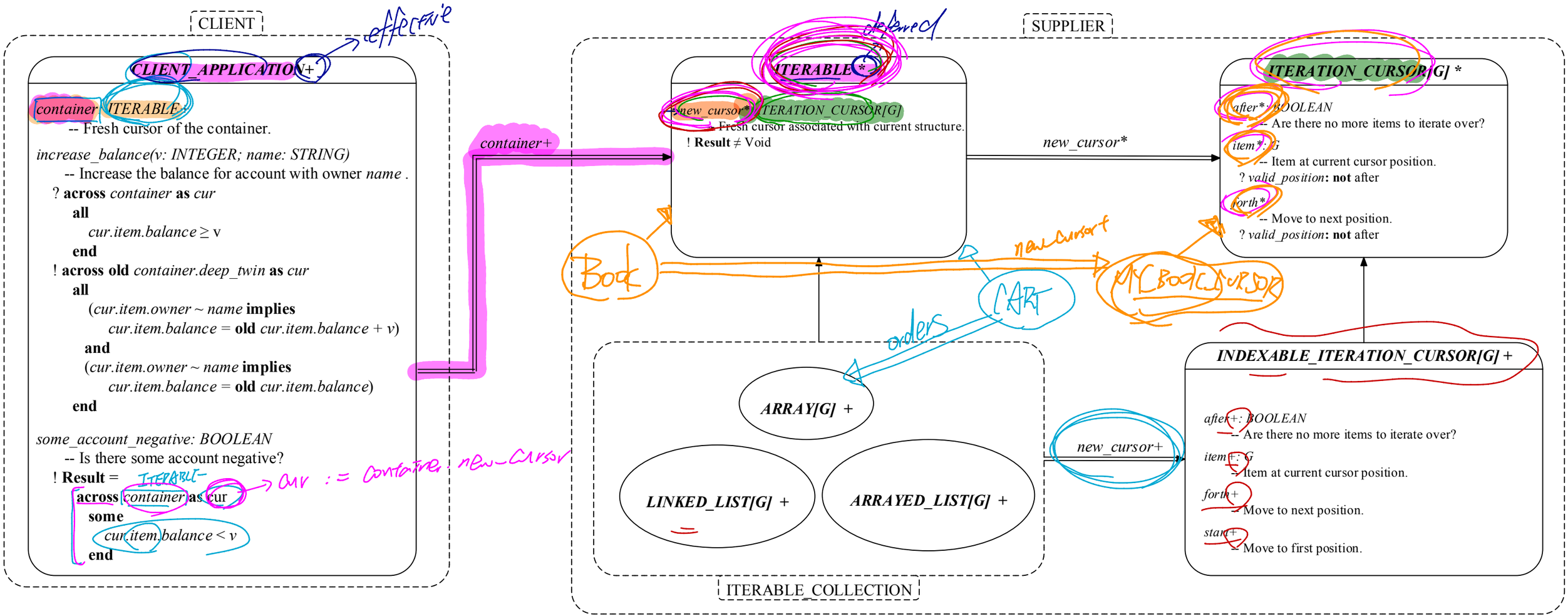
```
class
  CART
feature
  orders: ARRAY [ORDER]
end

class
  ORDER
feature
  price: INTEGER
  quantity: INTEGER
end
```

```
class
  SHOP
feature
  cart: CART
  checkout: INTEGER
    do
      from
        i := cart.orders.lower
      until
        i > cart.orders.upper
      do
        Result := Result +
          cart.orders[i].price
          *
          cart.orders[i].quantity
        i := i + 1
      end
    end
end
```

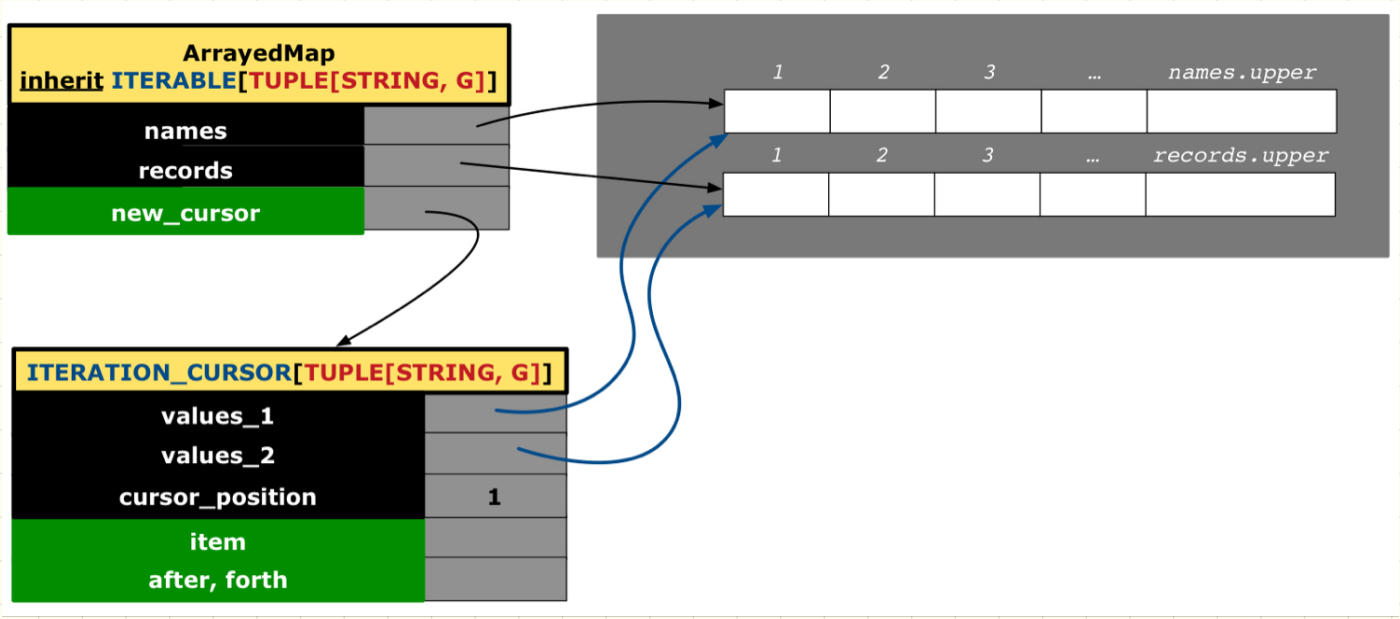
Problems?


Supplier
orders
client
public
interface
hidden
(subject to changes)

Iterator Design Pattern



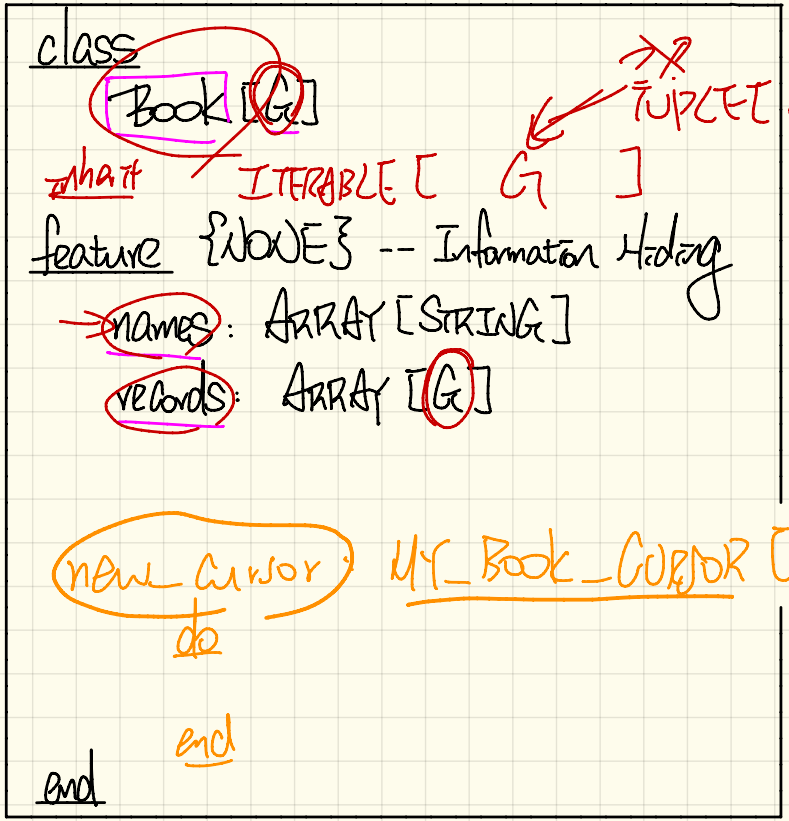
Iterator Pattern at Runtime



Implementing the ITERATOR Pattern: Easy Case

```
class
    CART
inherit
    ITERABLE [ORDER]
feature {NONE} -- Information Hiding
    orders : ARRAY [ORDER]

new_cursor : I_C [ORDER]
    do
        Result := orders.new_cursor
    end
end
```

Implementing the ITERATOR Pattern: Hard Case



Static vs. Dynamic Types

```
local
    oa : A
do
    create {?} oa.make
```

S.T static

dynamic type

```
A oa = new ?();
```

SORTED_MAP_ADT

```
deferred class
    SORTED_MAP_ADT [K -> COMPARABLE, V -> ANY]
inherit
    ITERABLE [TUPLE [K, V]]
feature -- model
    model: FUN [K, V]
        deferred
        end
feature {NONE} -- attributes
    instance: like Current
        deferred
        end
feature -- commands
    put (val: V; key: K)
        deferred
        ensure
            inserted: model ~ ((old model.deep_twin) @<+ [key, val])
        end
    sub_map (lower, upper: K): like Current xclusive
        -- may return nothing if no elements between `lower' and `upper'
        require
            lower_less_than_upper: lower < upper
        do
            Result := instance.deep_twin
            across
                Current as cursor
            loop
                if lower <= cursor.item.key and then cursor.item.k
                    Result.extend (cursor.item.key, Current [cursc
                end
            end
```

template

Smm. sub_map.

SORTED_MODEL_MAP

```
c
    SORTED_MODEL_MAP [K -> COMPARABLE, V -> ANY]
inherit
    SORTED_MAP_ADT[K,V]
create
    make_empty, make_from_array, make_from_sorted_map
feature -- model
    model: FUN [K, V]
            -- abstraction function
        do
            Result := implementation
        end
feature{NONE} -- attributes
    implementation: FUN[K,V]
            -- inefficient but abstract implementation of sorted map
        attribute
            create Result.make_empty
        end
    instance: like Current
        attribute
            create Result.make_empty
        end
feature -- commands
    put (val: V; key: K) --(key: K; val: V)
            -- puts an element of `key' and `value' into map
            -- behaves like `extend' if `key' does not exist
            -- otherwise behaves like `update'
            -- NOTE: This method follows the convention of `val'/`key'
        do
            implementation.override_by ([key, val])
        end
```