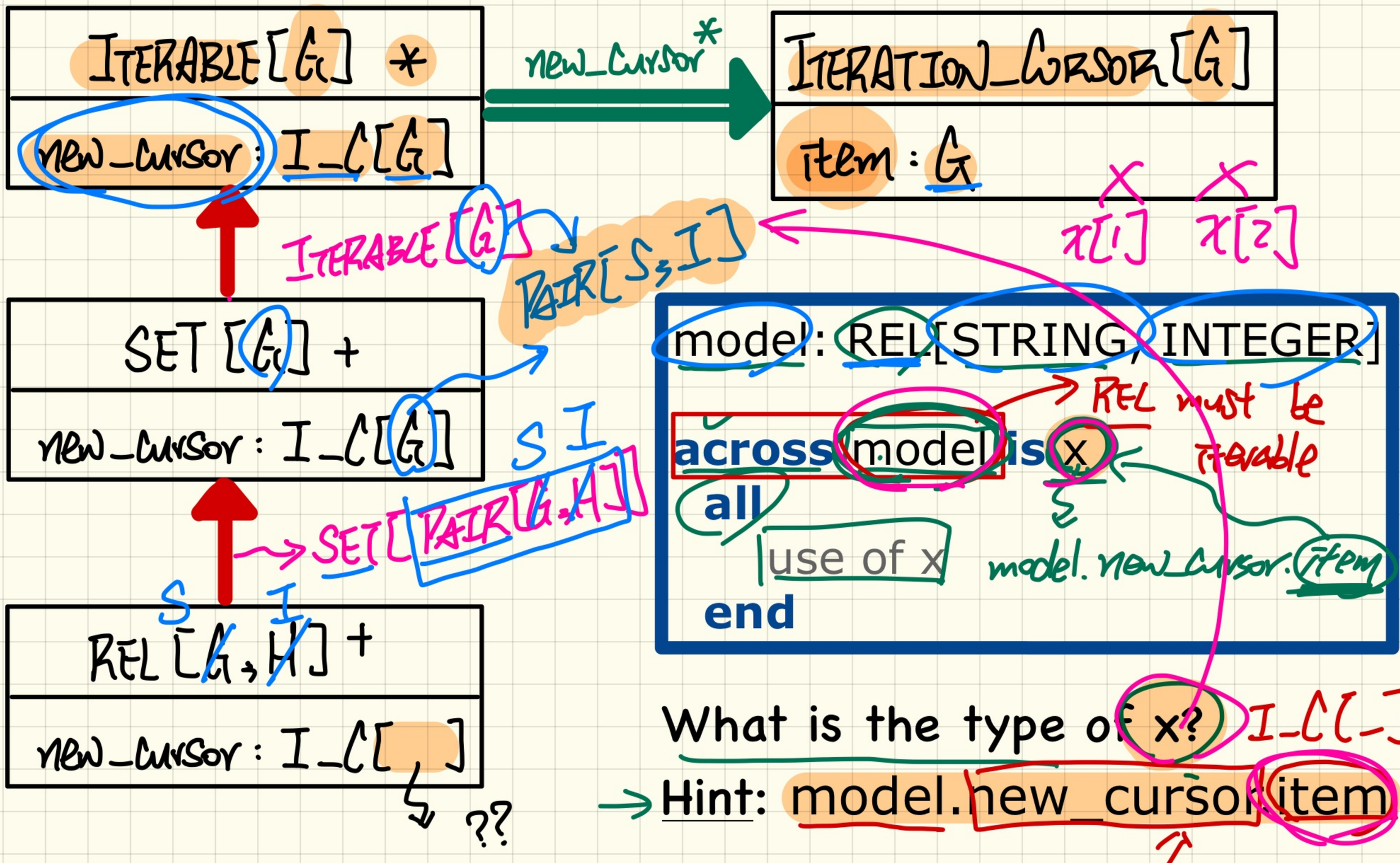
EECS3311 Software Design (Fall 2020)

Q&A - Lecture Series W4

Monday, October 5

Type of REL item in across



Exercise

```
deferred class
  ITERABLE [G]
feature -- Access
  new_cursor: ITERATION_CURSOR [G]
    deferred end
end
```

new_cursor*

```
deferred class
  ITERATION_CURSOR [G]
feature -- Cursor features
  item: G
    deferred end

  after: BOOLEAN
    deferred end

  forth
    deferred end
```

```
test_database: BOOLEAN
  local
    db: DATABASE[STRING, INTEGER]
    tuples: LINKED_LIST[TUPLE[INTEGER, STRING]]
  do
    create db.make
    create tuples.make
    across
      db is t
    loop
      tuples.extend (t)
    end
end
```

+db

```
class
  DATABASE[G, H]
inherit
  ITERABLE[          ]
feature {NONE} -- Implementation
  gs: ARRAY[G]
  hs: ARRAY[H]
feature -- Iterable
  new_cursor: ITERATION_CURSOR[          ]
    local
      db_cursor: ITEM_ITERATION_CURSOR[H, G]
    do
      create db_cursor.make (      )
      Result := db_cursor
    end
end
```

new_cursor+

```
class
  ITEM_ITERATION_CURSOR[M, N]
inherit
  ITERATION_CURSOR[          ]
create
  make
feature {NONE} -- Implementation
  ms: ARRAY[M]
  ns: ARRAY[N]
feature -- Constructor
  make (new_ns: ARRAY[N]; new_ms: ARRAY[M])
    do ... end
feature -- Cursor features
  item:      
    do ... end

  after: BOOLEAN
    do ... end

  forth
    do ... end
end
```

Exercise

```
deferred class
  ITERABLE [G]
feature -- Access
  new_cursor: ITERATION_CURSOR [G]
    deferred end
end
```

```
deferred class
  ITERATION_CURSOR [G]
feature -- Cursor features
  item: G
    deferred end

  after: BOOLEAN
    deferred end

  forth
    deferred end
end
```

new_cursor*

```
test_database: BOOLEAN
  local
    db: DATABASE[STRING, INTEGER]
    tuples: LINKED_LIST[TUPLE[INTEGER, STRING]]
  do
    create db.make
    create tuples.make
    across
      db is t
    loop
      tuples.extend (t)
    end
end
```

+db

```
class
  DATABASE [G, H]
inherit
  ITERABLE[TUPLE[H, G]]
feature {NONE} -- Implementation
  gs: ARRAY[G]
  hs: ARRAY[H]
feature -- Iterable
  new_cursor: ITERATION_CURSOR[TUPLE[H, G]]
    local
      db_cursor: ITEM_ITERATION_CURSOR[H, G]
    do
      create db_cursor.make (gs, hs)
      Result := db_cursor
    end
end
```

new_cursor+

```
class
  ITEM_ITERATION_CURSOR [M, N]
inherit
  ITERATION_CURSOR[TUPLE[M, N]]
create
  make
feature {NONE} -- Implementation
  ms: ARRAY[M]
  ns: ARRAY[N]
feature -- Constructor
  make (new_ns: ARRAY[N]; new_ms: ARRAY[M])
    do ... end
feature -- Cursor features
  item: TUPLE[M, N]
    do ... end

  after: BOOLEAN
    do ... end

  forth
    do ... end
end
```

Hints

1. TUPLE[N, M]
2. I_C[M, N]

Exercises

① Put this into Estudio.

② e.g. swap the order:

I_I_C[N, M]

↳ fixes?

↳ re-compile.

not a class

TUPLE[H, G]

↳ a type where

- 1st element H
- 2nd element G

Exercise

```
deferred class
  ITERABLE [G]
feature -- Access
  new_cursor: ITERATION_CURSOR [G]
    deferred end
end
```

new_cursor*

```
deferred class
  ITERATION_CURSOR [G]
feature -- Cursor features
  item: G
    deferred end

  after: BOOLEAN
    deferred end

  forth
    deferred end
```

```
test_database: BOOLEAN
  local
    db: DATABASE[STRING, INTEGER]
    tuples: LINKED_LIST[TUPLE[INTEGER, STRING]]
  do
    create db.make
    create tuples.make
    across
      db is t
    loop
      tuples.extend (t)
    end
end
```

+db

```
class
  DATABASE[G, H]
inherit
  ITERABLE[TUPLE[H, G]]
feature {NONE} -- Implementation
  gs: ARRAY[G]
  hs: ARRAY[H]
feature -- Iterable
  new_cursor: ITERATION_CURSOR[TUPLE[H, G]]
    local
      db_cursor: ITEM_ITERATION_CURSOR[H, G]
    do
      create db_cursor.make (gs, hs)
      Result := db_cursor
    end
end
```

new_cursor+

Solution

```
class
  ITEM_ITERATION_CURSOR[M, N]
inherit
  ITERATION_CURSOR[TUPLE[M, N]]
create
  make
feature {NONE} -- Implementation
  ms: ARRAY[M]
  ns: ARRAY[N]
feature -- Constructor
  make (new_ns: ARRAY[N]; new_ms: ARRAY[M])
    do ... end
feature -- Cursor features
  item: TUPLE[M, N]
    do ... end

  after: BOOLEAN
    do ... end

  forth
    do ... end
end
```

Writing Contract in Design Diagram

② p_key ∈ model.domain
↓ REL

TREE_NODE[K,V] ↑
bold
require
left_extend:
attached left as l

For example we have a pre-condition like we had in tree_node (attached left as l .....) then how do we show that in our design diagram?

⇒ l.is_extend

✓ left.is_external

To show Current.has(p_key) can we write exists p_key or we need to write has(p_key) which one is more appropriate with respect to tree_node class?

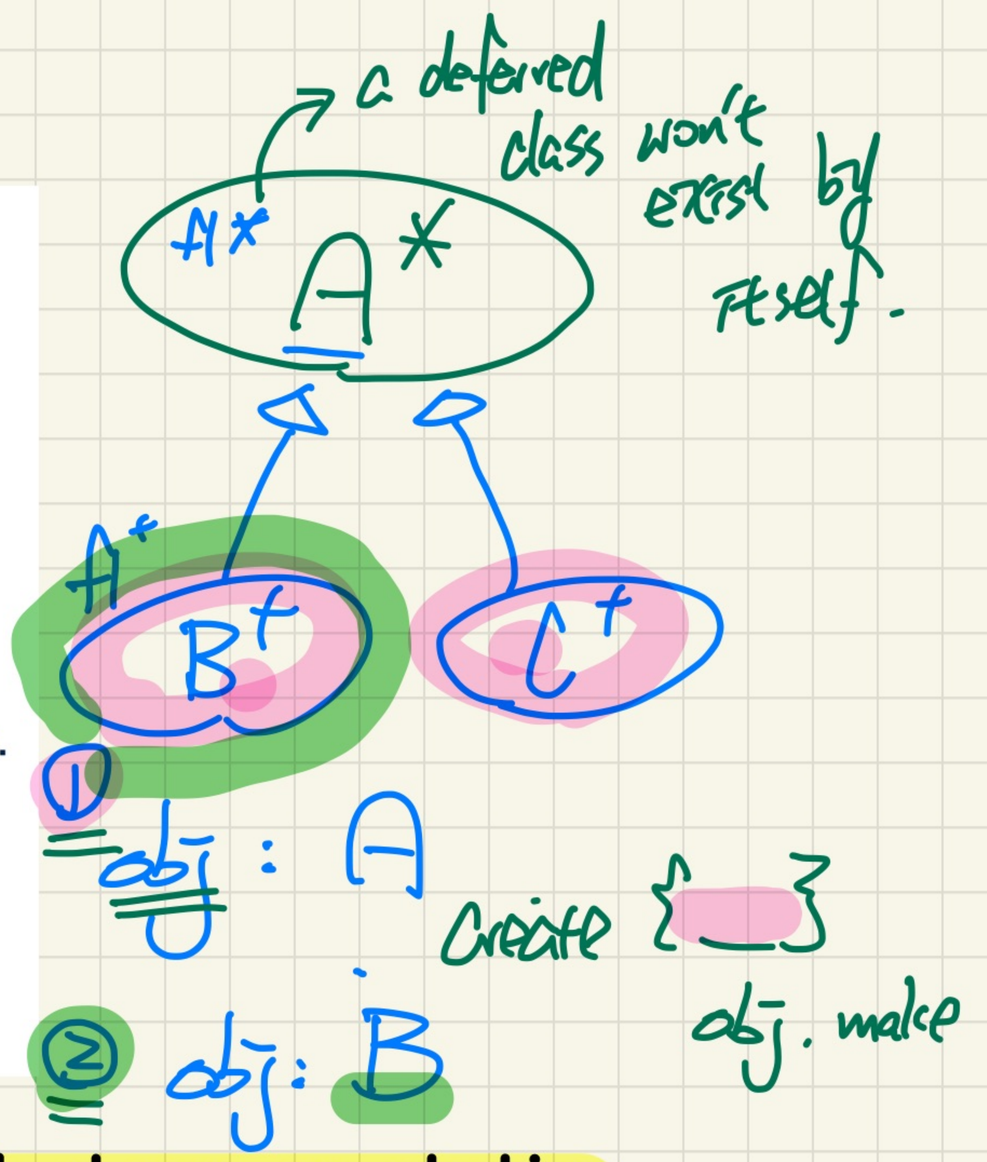
① require
Current.has(p_key)

require
left_external:
attached left as l implies l.is_external

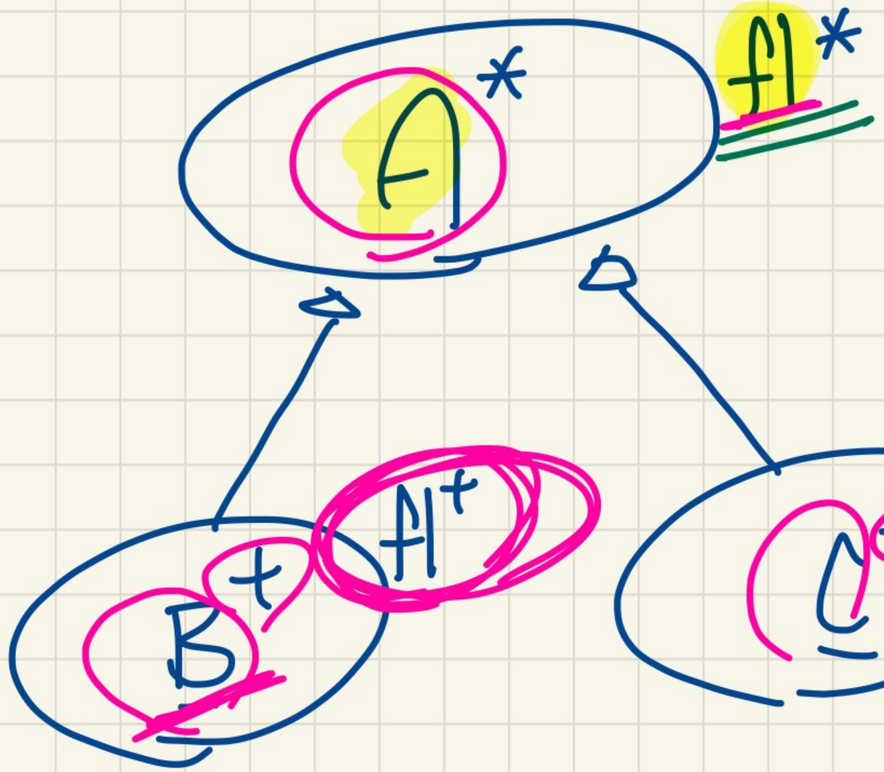
Program from the Interface

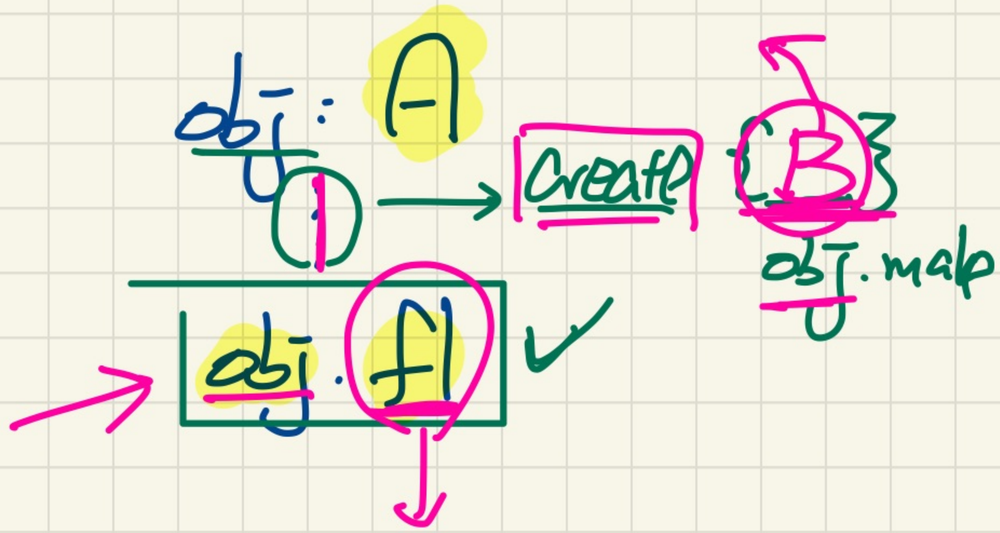
- A ***deferred** class* has **at least one** feature ***unimplemented***.
  - A *deferred class* may only be used as a ***static*** type (for declaration), but cannot be used as a ***dynamic*** type.
  - e.g., By declaring `list: LIST[INTEGER]` (where `LIST` is a ***deferred*** class), it is invalid to write:
    - **create** `list.make`
    - **create** {***LIST[INTEGER]***} `list.make`
- An ***effective** class* has **all** features ***implemented***.
  - An *effective class* may be used as both ***static*** and ***dynamic*** types.
  - e.g., By declaring `list: LIST[INTEGER]`, it is valid to write:
    - **create** {***LINKED_LIST[INTEGER]***} `list.make`
    - **create** {***ARRAYED_LIST[INTEGER]***} `list.make`

  where `LINKED_LIST` and `ARRAYED_LIST` are both ***effective*** descendants of `LIST`.



What is the purpose of using deferred class as static type? Can the static deferred feature in deferred class being declared or called?





f1 ① How many versions?

2: B or C

② Version of f1 from B will be called.

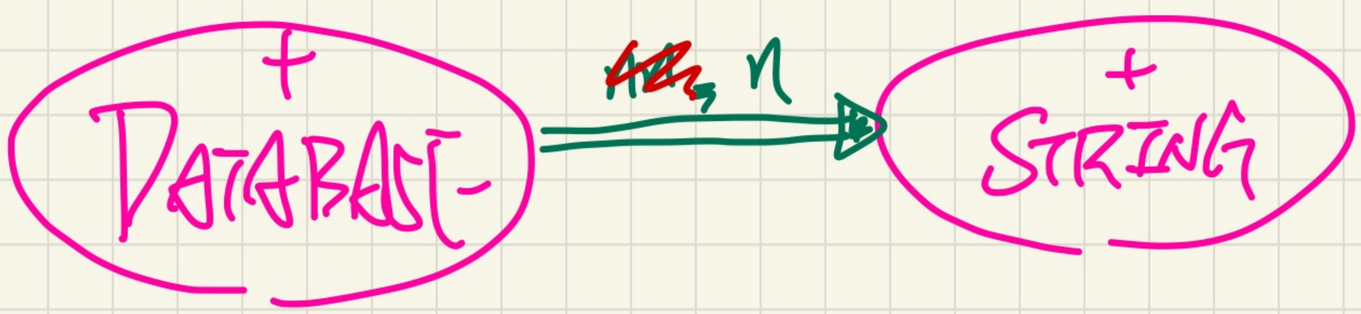
Showing Suppliers of the Same Name

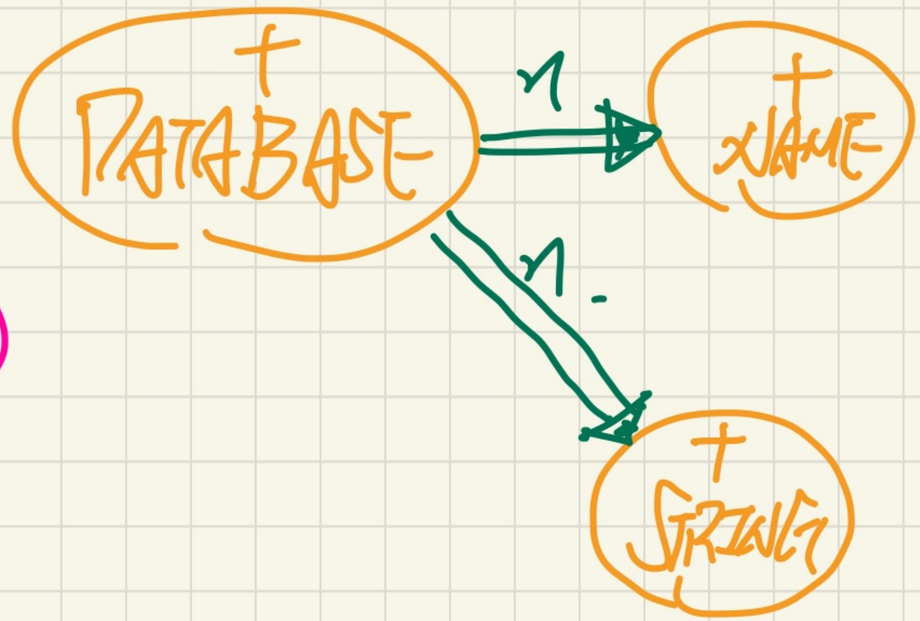
```
class DATABASE
feature {NONE} -- implementation
  data: ARRAY[STRING]
feature -- Commands
  add_name (nn: STRING)
      -- Add name `nn' to database.
    require ... do ... ensure ... end

  name_exists (n: STRING): BOOLEAN
      -- Does name `n' exist in database?
    require ...
    local
      u: UTILITIES
    do ... ensure ... end
invariant
  ...
end
```

What if in the source code, the parameter in add_name and name_exists both had the name "n"?

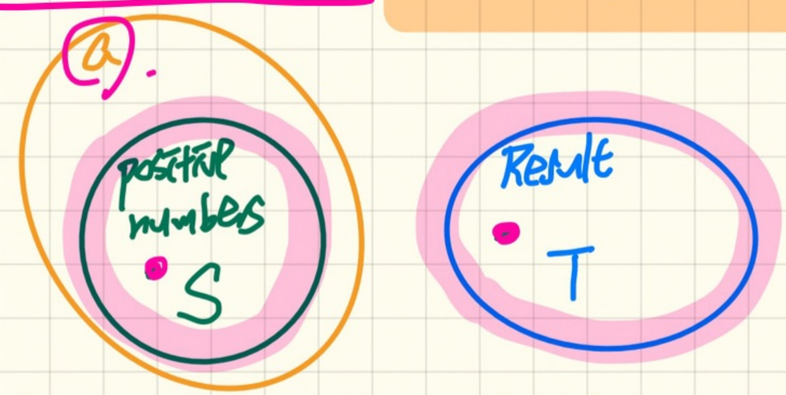
Do we have to change one of them in the design diagram or is there a way to represent them both as "n" to be consistent with the source code?





Conjunction or Implication?

```
all_positive_values (a: ARRAY[INTEGER]): ARRAY[INTEGER]
  require
    no_duplicates: ??
  ensure
    across Result is x
      all
        x > 0
      end
```



$S \subseteq T \equiv (\forall x \mid x \in S \Rightarrow x \in T)$ $S \subseteq T$

$\wedge$

$x \in T \Rightarrow x \in S$ $T \subseteq S$

all_pos_in_a_also_in_result:

```
across a is x
all x > 0 implies Result.has(x) end
```

T
T>0
F
F and

all_num_in_result_pos_and_in_a:

```
across Result is x
all x > 0 and a.has(x) end
```

↓ post cond violation.

allPos_val(

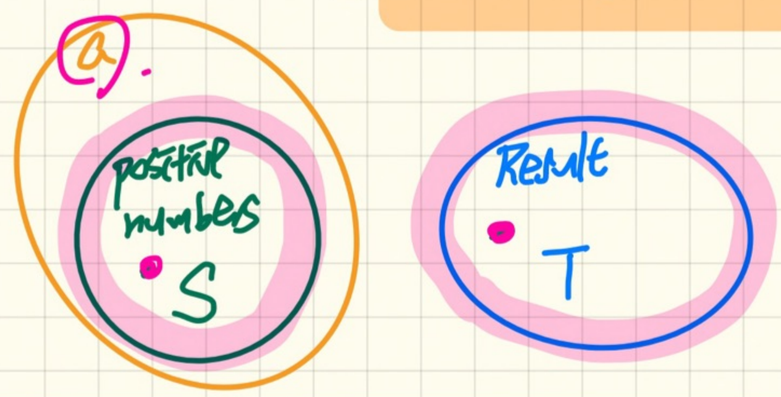
<<2, -1, 3, -2>>

↳ <<2, 3>>

<<2, 3, 4>>

Conjunction or Implication?

```
all_positive_values (a: ARRAY[INTEGER]): ARRAY[INTEGER]
    require
      no_duplicates: ??
    ensure
      across Result is x
        all
          x > 0
        end
```



$S \Rightarrow T \equiv (\forall x \mid x \in S \Rightarrow x \in T)$ $S \subseteq T$

$\wedge$

$x \in T \Rightarrow x \in S)$ $T \subseteq S$

```
all_pos_in_a_also_in_result :
        across a is x
          all x > 0 implies Result.has(x) end
```

```
all_num_in_result_pos_and_
    in_a :
    across Result is x
    all x > 0 and a.has(x) end
```

↓

implies

no postcondition violation although output is wrong.

all_pos_val(<<2, -1, 3, -2>>)

↳ <<2, 3, -1>>.

3rd.

$x > 0 \Rightarrow a.has(x)$

T

F $\Rightarrow$ —

r @<< d1

model : REL [STR, INT]

{ _ }

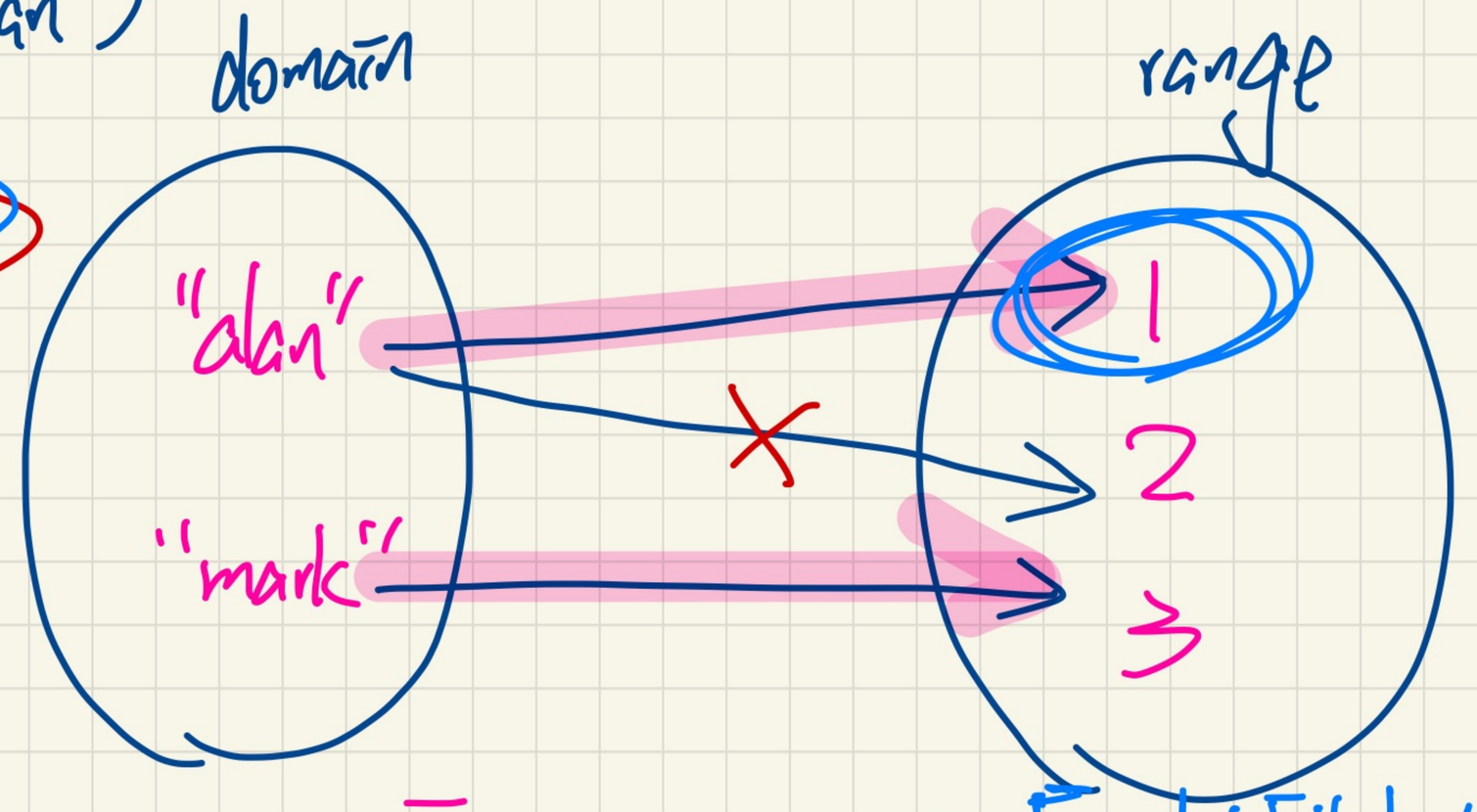
- model.image("alan")
- model["alan"]

model["alan"]

~ 1

INT

SET[INT]



not going to happen for lab2 ← not a valid function

model["alan"].count=1
and
model["alan"].has(1)