

# Abstractions via Mathematical Models



EECS3311 A: Software Design  
Winter 2020

CHEN-WEI WANG

# Motivating Problem: Complete Contracts

- Recall what we learned in the *Complete Contracts* lecture:
  - In *post-condition*, for *each attribute*, specify the relationship between its *pre-state* value and its *post-state* value.
  - Use the **old** keyword to refer to *post-state* values of expressions.
  - For a *composite*-structured attribute (e.g., arrays, linked-lists, hash-tables, *etc.*), we should specify that after the update:
    1. The intended change is present; **and**
    2. *The rest of the structure is unchanged*.
- Let's now revisit this technique by specifying a *LIFO stack*.

# Motivating Problem: LIFO Stack (1)

- Let's consider three different implementation strategies:

Stack Feature	Array	Linked List	
	Strategy 1	Strategy 2	Strategy 3
<i>count</i>	imp.count		
<i>top</i>	imp[imp.count]	imp.first	imp.last
<i>push(g)</i>	imp.force(g, imp.count + 1)	imp.put_front(g)	imp.extend(g)
<i>pop</i>	imp.list.remove_tail (1)	list.start list.remove	imp.finish imp.remove

- Given that all strategies are meant for implementing the **same ADT**, will they have **identical** contracts?

## Motivating Problem: LIFO Stack (2.1)

```
class LIFO_STACK[G] create make
feature {NONE} -- Strategy 1: array
  imp: ARRAY[G]
feature -- Initialization
  make do create imp.make_empty ensure imp.count = 0 end
feature -- Commands
  push(g: G)
    do imp.force(g, imp.count + 1)
    ensure
      changed: imp[count] ~ g
      unchanged: across 1 |..| count - 1 as i all
                  imp[i.item] ~ (old imp.deep_twin)[i.item] end
    end
  pop
    do imp.remove_tail(1)
    ensure
      changed: count = old count - 1
      unchanged: across 1 |..| count as i all
                  imp[i.item] ~ (old imp.deep_twin)[i.item] end
    end
end
```

## Motivating Problem: LIFO Stack (2.2)

```

class LIFO_STACK[G] create make
feature {NONE} -- Strategy 2: linked-list first item as top
  imp: LINKED_LIST[G]
feature -- Initialization
  make do create imp.make ensure imp.count = 0 end
feature -- Commands
  push(g: G)
    do imp.put_front(g)
    ensure
      changed: imp.first ~ g
      unchanged: across 2 |..| count as i all
        imp[i.item] ~ (old imp.deep_twin)[i.item - 1] end
    end
  pop
    do imp.start ; imp.remove
    ensure
      changed: count = old count - 1
      unchanged: across 1 |..| count as i all
        imp[i.item] ~ (old imp.deep_twin)[i.item + 1] end
    end
end
  
```

## Motivating Problem: LIFO Stack (2.3)

```
class LIFO_STACK[G] create make
feature {NONE} -- Strategy 3: linked-list last item as top
  imp: LINKED_LIST[G]
feature -- Initialization
  make do create imp.make ensure imp.count = 0 end
feature -- Commands
  push(g: G)
    do imp.extend(g)
    ensure
      changed: imp.last ~ g
      unchanged: across 1 |..| count - 1 as i all
        imp[i.item] ~ (old imp.deep_twin)[i.item] end
    end
  pop
    do imp.finish ; imp.remove
    ensure
      changed: count = old count - 1
      unchanged: across 1 |..| count as i all
        imp[i.item] ~ (old imp.deep_twin)[i.item] end
    end
end
```

# Design Principles: Information Hiding & Single Choice

---

- **Information Hiding** (IH):
  - Hide supplier's **design decisions** that are *likely to change*.
  - Violation of IH means that your design's public API is **unstable**.
  - *Change of supplier's secrets* should not affect clients relying upon the existing API.
- **Single Choice Principle** (SCP):
  - When a **change** is needed, there should be **a single place** (or **a minimal number of places**) where you need to make that change.
  - Violation of SCP means that your design contains **redundancies**.

## Motivating Problem: LIFO Stack (3)

- *Postconditions* of all 3 versions of stack are *complete*.  
i.e., Not only the new item is *pushed/popped*, but also the remaining part of the stack is *unchanged*.
- But they violate the principle of *information hiding*:  
Changing the *secret*, internal workings of data structures should not affect any existing clients.
- How so?

The private attribute `imp` is referenced in the *postconditions*, exposing the implementation strategy not relevant to clients:

- Top of stack may be `imp[count]`, `imp.first`, or `imp.last`.
- Remaining part of stack may be `across 1 | .. | count - 1` or `across 2 | .. | count`.

⇒ *Changing the implementation strategy* from one to another will also *change the contracts for all features*.

⇒ This also violates the *Single Choice Principle*.



# Math Models: Command vs Query

- Use MATHMODELS library to create math objects (SET, REL, SEQ).
- State-changing **commands**: Implement an **Abstraction Function**

```
class LIFO_STACK[G -> attached ANY] create make
feature {NONE} -- Implementation
  imp: LINKED_LIST[G]
feature -- Abstraction function of the stack ADT
model: SEQ[G]
  do create Result.make_empty
    across imp as cursor loop Result.append(cursor.item) end
end
```

- Side-effect-free **queries**: Write Complete Contracts

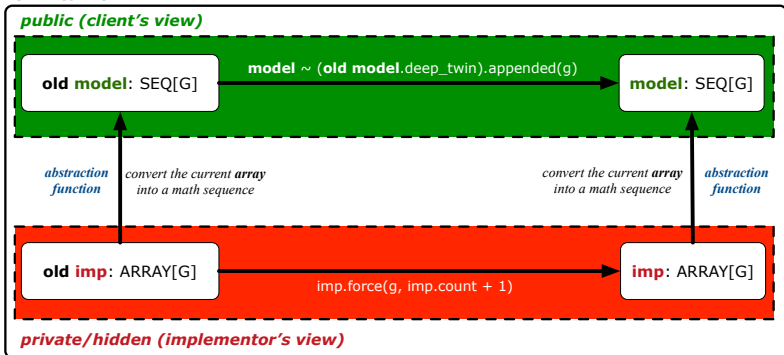
```
class LIFO_STACK[G -> attached ANY] create make
feature -- Abstraction function of the stack ADT
model: SEQ[G]
feature -- Commands
  push (g: G)
    ensure model ~ (old model.deep_twin).appended(g) end
```

# Implementing an Abstraction Function (1)

```
class LIFO_STACK[G -> attached ANY] create make
feature {NONE} -- Implementation Strategy 1
  imp: ARRAY[G]
feature -- Abstraction function of the stack ADT
  model: SEQ[G]
  do create Result.make_from_array (imp)
  ensure
    counts: imp.count = Result.count
    contents: across 1 |..| Result.count as i all
      Result[i.item] ~ imp[i.item]
  end
feature -- Commands
  make do create imp.make_empty ensure model.count = 0 end
  push (g: G) do imp.force(g, imp.count + 1)
    ensure pushed: model ~ (old model.deep_twin).appended(g) end
  pop do imp.remove_tail(1)
    ensure popped: model ~ (old model.deep_twin).front end
end
```

# Abstracting ADTs as Math Models (1)

'push(g: G)' feature of LIFO\_STACK ADT



- **Strategy 1** *Abstraction function*: Convert the *implementation array* to its corresponding *model sequence*.
- *Contract* for the `put (g: G)` feature remains the **same**:

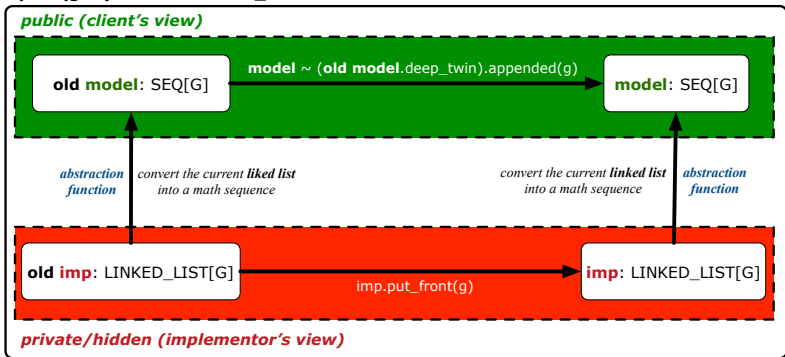
`model ~ (old model.deep_twin).appended(g)`

## Implementing an Abstraction Function (2)

```
class LIFO_STACK[G -> attached ANY] create make
feature {NONE} -- Implementation Strategy 2 (first as top)
  imp: LINKED_LIST[G]
feature -- Abstraction function of the stack ADT
  model: SEQ[G]
  do create Result.make_empty
    across imp as cursor loop Result.prepend(cursor.item) end
  ensure
    counts: imp.count = Result.count
    contents: across 1 |..| Result.count as i all
      Result[i.item] ~ imp[count - i.item + 1]
  end
feature -- Commands
  make do create imp.make ensure model.count = 0 end
  push (g: G) do imp.put_front(g)
    ensure pushed: model ~ (old model.deep_twin).appended(g) end
  pop do imp.start ; imp.remove
    ensure popped: model ~ (old model.deep_twin).front end
end
```

# Abstracting ADTs as Math Models (2)

'push(g: G)' feature of LIFO\_STACK ADT



- **Strategy 2** *Abstraction function*: Convert the *implementation list* (first item is top) to its corresponding *model sequence*.
- *Contract* for the `put (g: G)` feature remains the **same**:

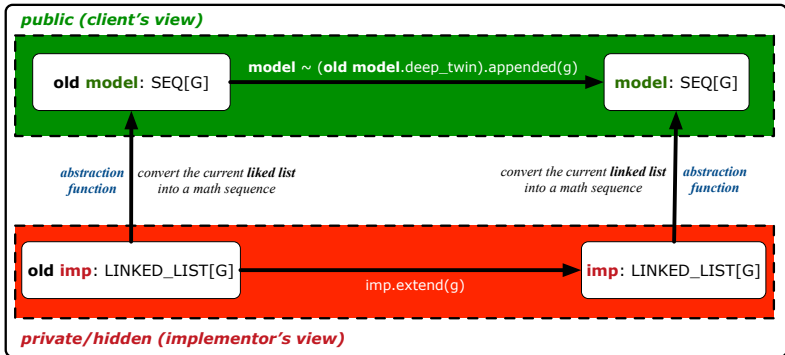
`model ~ (old model.deep_twin).appended(g)`

# Implementing an Abstraction Function (3)

```
class LIFO_STACK[G -> attached ANY] create make
feature {NONE} -- Implementation Strategy 3 (last as top)
  imp: LINKED_LIST[G]
feature -- Abstraction function of the stack ADT
  model: SEQ[G]
  do create Result.make_empty
    across imp as cursor loop Result.append(cursor.item) end
  ensure
    counts: imp.count = Result.count
    contents: across 1 |..| Result.count as i all
      Result[i.item] ~ imp[i.item]
  end
feature -- Commands
  make do create imp.make ensure model.count = 0 end
  push (g: G) do imp.extend(g)
    ensure pushed: model ~ (old model.deep_twin).appended(g) end
  pop do imp.finish ; imp.remove
    ensure popped: model ~ (old model.deep_twin).front end
end
```

# Abstracting ADTs as Math Models (3)

'push(g: G)' feature of LIFO\_STACK ADT



- **Strategy 3** *Abstraction function*: Convert the *implementation list* (last item is top) to its corresponding *model sequence*.
- *Contract* for the `put (g: G)` feature remains the **same**:

`model ~ (old model.deep_twin).appended(g)`

# Solution: Abstracting ADTs as Math Models

- Writing contracts in terms of *implementation attributes* (arrays, LL's, hash tables, *etc.*) violates **information hiding** principle.
  - Instead:
    - For each ADT, create an **abstraction** via a **mathematical model**.  
e.g., Abstract a LIFO\_STACK as a mathematical sequence.
    - For each ADT, define an **abstraction function** (i.e., a query) whose return type is a kind of **mathematical model**.  
e.g., Convert *implementation array* to *mathematical sequence*
    - Write contracts in terms of the **abstract math model**.  
e.g., When pushing an item  $g$  onto the stack, specify it as appending  $g$  into its model sequence.
    - Upon *changing the implementation*:
      - **No** change on **what** the abstraction is, hence *no change on contracts*.
      - **Only** change **how** the abstraction is constructed, hence *changes on the body of the abstraction function*.  
e.g., Convert *implementation linked-list* to *mathematical sequence*
- ⇒ The **Single Choice Principle** is obeyed.



# Math Review: Set Definitions and Membership



- A **set** is a collection of objects.
  - Objects in a set are called its *elements* or *members*.
  - *Order* in which elements are arranged does not matter.
  - An element can appear *at most once* in the set.
- We may define a set using:
  - *Set Enumeration*: Explicitly list all members in a set.  
e.g.,  $\{1, 3, 5, 7, 9\}$
  - *Set Comprehension*: Implicitly specify the condition that all members satisfy.  
e.g.,  $\{x \mid 1 \leq x \leq 10 \wedge x \text{ is an odd number}\}$
- An empty set (denoted as  $\{\}$  or  $\emptyset$ ) has no members.
- We may check if an element is a *member* of a set:
  - e.g.,  $5 \in \{1, 3, 5, 7, 9\}$  [true]
  - e.g.,  $4 \notin \{x \mid x \leq 1 \leq 10, x \text{ is an odd number}\}$  [true]
- The number of elements in a set is called its *cardinality*.  
e.g.,  $|\emptyset| = 0$ ,  $|\{x \mid x \leq 1 \leq 10, x \text{ is an odd number}\}| = 5$

# Math Review: Set Relations

Given two sets  $S_1$  and  $S_2$ :

- $S_1$  is a *subset* of  $S_2$  if every member of  $S_1$  is a member of  $S_2$ .

$$S_1 \subseteq S_2 \iff (\forall x \bullet x \in S_1 \Rightarrow x \in S_2)$$

- $S_1$  and  $S_2$  are *equal* iff they are the subset of each other.

$$S_1 = S_2 \iff S_1 \subseteq S_2 \wedge S_2 \subseteq S_1$$

- $S_1$  is a *proper subset* of  $S_2$  if it is a strictly smaller subset.

$$S_1 \subset S_2 \iff S_1 \subseteq S_2 \wedge |S_1| < |S_2|$$

# Math Review: Set Operations

Given two sets  $S_1$  and  $S_2$ :

- *Union* of  $S_1$  and  $S_2$  is a set whose members are in either.

$$S_1 \cup S_2 = \{x \mid x \in S_1 \vee x \in S_2\}$$

- *Intersection* of  $S_1$  and  $S_2$  is a set whose members are in both.

$$S_1 \cap S_2 = \{x \mid x \in S_1 \wedge x \in S_2\}$$

- *Difference* of  $S_1$  and  $S_2$  is a set whose members are in  $S_1$  but not  $S_2$ .

$$S_1 \setminus S_2 = \{x \mid x \in S_1 \wedge x \notin S_2\}$$

# Math Review: Power Sets

The **power set** of a set  $S$  is a *set* of all  $S$ ' *subsets*.

$$\mathbb{P}(S) = \{s \mid s \subseteq S\}$$

The power set contains subsets of *cardinalities*  $0, 1, 2, \dots, |S|$ .  
e.g.,  $\mathbb{P}(\{1, 2, 3\})$  is a set of sets, where each member set  $s$  has cardinality  $0, 1, 2$ , or  $3$ :

$$\left\{ \begin{array}{l} \emptyset, \\ \{1\}, \{2\}, \{3\}, \\ \{1, 2\}, \{2, 3\}, \{3, 1\}, \\ \{1, 2, 3\} \end{array} \right\}$$

# Math Review: Set of Tuples

Given  $n$  sets  $S_1, S_2, \dots, S_n$ , a **cross product** of these sets is a set of  $n$ -tuples.

Each  *$n$ -tuple*  $(e_1, e_2, \dots, e_n)$  contains  $n$  elements, each of which a member of the corresponding set.

$$S_1 \times S_2 \times \dots \times S_n = \{(e_1, e_2, \dots, e_n) \mid e_i \in S_i \wedge 1 \leq i \leq n\}$$

e.g.,  $\{a, b\} \times \{2, 4\} \times \{\$, \&\}$  is a set of triples:

$$\begin{aligned} & \{a, b\} \times \{2, 4\} \times \{\$, \&\} \\ = & \{(e_1, e_2, e_3) \mid e_1 \in \{a, b\} \wedge e_2 \in \{2, 4\} \wedge e_3 \in \{\$, \&\}\} \\ = & \{(a, 2, \$), (a, 2, \&), (a, 4, \$), (a, 4, \&), \\ & (b, 2, \$), (b, 2, \&), (b, 4, \$), (b, 4, \&)\} \end{aligned}$$

# Math Models: Relations (1)

- A **relation** is a collection of mappings, each being an *ordered pair* that maps a member of set  $S$  to a member of set  $T$ .  
e.g., Say  $S = \{1, 2, 3\}$  and  $T = \{a, b\}$ 
  - $\emptyset$  is an empty relation.
  - $S \times T$  is a relation (say  $r_1$ ) that maps from each member of  $S$  to each member in  $T$ :  $\{(1, a), (1, b), (2, a), (2, b), (3, a), (3, b)\}$
  - $\{(x, y) : S \times T \mid x \neq 1\}$  is a relation (say  $r_2$ ) that maps only some members in  $S$  to every member in  $T$ :  $\{(2, a), (2, b), (3, a), (3, b)\}$ .
- Given a relation  $r$ :
  - **Domain** of  $r$  is the set of  $S$  members that  $r$  maps from.

$$\text{dom}(r) = \{s : S \mid (\exists t \bullet (s, t) \in r)\}$$

e.g.,  $\text{dom}(r_1) = \{1, 2, 3\}$ ,  $\text{dom}(r_2) = \{2, 3\}$

- **Range** of  $r$  is the set of  $T$  members that  $r$  maps to.

$$\text{ran}(r) = \{t : T \mid (\exists s \bullet (s, t) \in r)\}$$

e.g.,  $\text{ran}(r_1) = \{a, b\} = \text{ran}(r_2)$

## Math Models: Relations (2)

- We use the power set operator to express the set of *all possible relations* on  $S$  and  $T$ :

$$\mathbb{P}(S \times T)$$

- To declare a relation variable  $r$ , we use the colon ( $:$ ) symbol to mean *set membership*:

$$r : \mathbb{P}(S \times T)$$

- Or alternatively, we write:

$$r : S \leftrightarrow T$$

where the set  $S \leftrightarrow T$  is synonymous to the set  $\mathbb{P}(S \times T)$

## Math Models: Relations (3.1)

Say  $r = \{(a, 1), (b, 2), (c, 3), (a, 4), (b, 5), (c, 6), (d, 1), (e, 2), (f, 3)\}$

- **r.domain**: set of first-elements from  $r$ 
  - $r.\mathbf{domain} = \{d \mid (d, r) \in r\}$
  - e.g.,  $r.\mathbf{domain} = \{a, b, c, d, e, f\}$
- **r.range**: set of second-elements from  $r$ 
  - $r.\mathbf{range} = \{r \mid (d, r) \in r\}$
  - e.g.,  $r.\mathbf{range} = \{1, 2, 3, 4, 5, 6\}$
- **r.inverse**: a relation like  $r$  except elements are in reverse order
  - $r.\mathbf{inverse} = \{(r, d) \mid (d, r) \in r\}$
  - e.g.,  $r.\mathbf{inverse} = \{(1, a), (2, b), (3, c), (4, a), (5, b), (6, c), (1, d), (2, e), (3, f)\}$



## Math Models: Relations (3.2)

Say  $r = \{(a, 1), (b, 2), (c, 3), (a, 4), (b, 5), (c, 6), (d, 1), (e, 2), (f, 3)\}$

- **r.domain\_restricted(ds)**: sub-relation of  $r$  with domain  $ds$ .
  - $r.\text{domain\_restricted}(ds) = \{ (d, r) \mid (d, r) \in r \wedge d \in ds \}$
  - e.g.,  $r.\text{domain\_restricted}(\{a, b\}) = \{(a, 1), (b, 2), (a, 4), (b, 5)\}$
- **r.domain\_subtracted(ds)**: sub-relation of  $r$  with domain not  $ds$ .
  - $r.\text{domain\_subtracted}(ds) = \{ (d, r) \mid (d, r) \in r \wedge d \notin ds \}$
  - e.g.,  $r.\text{domain\_subtracted}(\{a, b\}) = \{(c, 6), (d, 1), (e, 2), (f, 3)\}$
- **r.range\_restricted(rs)**: sub-relation of  $r$  with range  $rs$ .
  - $r.\text{range\_restricted}(rs) = \{ (d, r) \mid (d, r) \in r \wedge r \in rs \}$
  - e.g.,  $r.\text{range\_restricted}(\{1, 2\}) = \{(a, 1), (b, 2), (d, 1), (e, 2)\}$
- **r.range\_subtracted(ds)**: sub-relation of  $r$  with range not  $ds$ .
  - $r.\text{range\_subtracted}(rs) = \{ (d, r) \mid (d, r) \in r \wedge r \notin rs \}$
  - e.g.,  $r.\text{range\_subtracted}(\{1, 2\}) = \{(c, 3), (a, 4), (b, 5), (c, 6)\}$

## Math Models: Relations (3.3)

Say  $r = \{(a, 1), (b, 2), (c, 3), (a, 4), (b, 5), (c, 6), (d, 1), (e, 2), (f, 3)\}$

- **r.overridden(t)**: a relation which agrees on  $r$  outside domain of  $t$ .domain, and agrees on  $t$  within domain of  $t$ .domain
  - $r.\text{overridden}(t) = t \cup r.\text{domain\_subtracted}(t.\text{domain})$
  -

$$\begin{aligned} & r.\text{overridden}(\{(a, 3), (c, 4)\}) \\ = & \underbrace{\{(a, 3), (c, 4)\}}_t \cup \underbrace{\{(b, 2), (b, 5), (d, 1), (e, 2), (f, 3)\}}_{r.\text{domain\_subtracted}(t.\text{domain})} \\ & \hspace{15em} \underbrace{\hspace{10em}}_{\{a,c\}} \\ = & \{(a, 3), (c, 4), (b, 2), (b, 5), (d, 1), (e, 2), (f, 3)\} \end{aligned}$$

# Math Review: Functions (1)

A **function**  $f$  on sets  $S$  and  $T$  is a *specialized form* of relation: it is forbidden for a member of  $S$  to map to more than one members of  $T$ .

$$\forall s : S; t_1 : T; t_2 : T \bullet (s, t_1) \in f \wedge (s, t_2) \in f \Rightarrow t_1 = t_2$$

e.g., Say  $S = \{1, 2, 3\}$  and  $T = \{a, b\}$ , which of the following relations are also functions?

- $S \times T$  [No]
- $(S \times T) - \{(x, y) \mid (x, y) \in S \times T \wedge x = 1\}$  [No]
- $\{(1, a), (2, b), (3, a)\}$  [Yes]
- $\{(1, a), (2, b)\}$  [Yes]

## Math Review: Functions (2)

- We use *set comprehension* to express the set of all possible functions on  $S$  and  $T$  as those relations that satisfy the *functional property*:

$$\{r : S \leftrightarrow T \mid (\forall s : S; t_1 : T; t_2 : T \bullet (s, t_1) \in r \wedge (s, t_2) \in r \Rightarrow t_1 = t_2)\}$$

- This set (of possible functions) is a subset of the set (of possible relations):  $\mathbb{P}(S \times T)$  and  $S \leftrightarrow T$ .
- We abbreviate this set of possible functions as  $S \rightarrow T$  and use it to declare a function variable  $f$ :

$$f : S \rightarrow T$$

## Math Review: Functions (3.1)

Given a function  $f : S \rightarrow T$ :

- $f$  is *injective* (or an injection) if  $f$  does not map a member of  $S$  to more than one members of  $T$ .

$$f \text{ is injective} \iff (\forall s_1 : S; s_2 : S; t : T \bullet (s_1, t) \in r \wedge (s_2, t) \in r \Rightarrow s_1 = s_2)$$

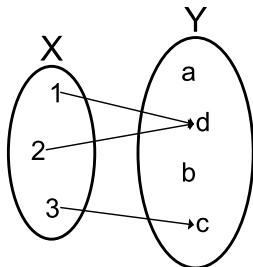
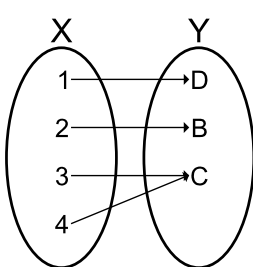
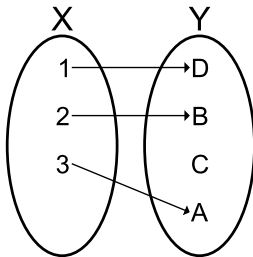
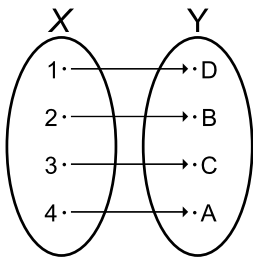
e.g., Considering an array as a function from integers to objects, being injective means that the array does not contain any duplicates.

- $f$  is *surjective* (or a surjection) if  $f$  maps to all members of  $T$ .

$$f \text{ is surjective} \iff \text{ran}(f) = T$$

- $f$  is *bijective* (or a bijection) if  $f$  is both injective and surjective.

# Math Review: Functions (3.2)



# Math Models: Command-Query Separation

<i>Command</i>	<i>Query</i>
domain_restrict	domain_restricted <b>ed</b>
domain_restrict_by	domain_restricted <b>ed</b> .by
domain_subtract	domain_subtracted <b>ed</b>
domain_subtract_by	domain_subtracted <b>ed</b> .by
range_restrict	range_restricted <b>ed</b>
range_restrict_by	range_restricted <b>ed</b> .by
range_subtract	range_subtracted <b>ed</b>
range_subtract_by	range_subtracted <b>ed</b> .by
override	overridden <b>ed</b>
override_by	overridden <b>ed</b> .by

Say  $r = \{(a, 1), (b, 2), (c, 3), (a, 4), (b, 5), (c, 6), (d, 1), (e, 2), (f, 3)\}$

- **Commands** modify the context relation objects.

`r.domain_restrict({a})` changes  $r$  to  $\{(a, 1), (a, 4)\}$

- **Queries** return new relations without modifying context objects.

`r.domain_restricted({a})` returns  $\{(a, 1), (a, 4)\}$  with  $r$  untouched

# Math Models: Example Test

```
test_rel: BOOLEAN
  local
    r, t: REL[STRING, INTEGER]
    ds: SET[STRING]
  do
    create r.make_from_tuple_array (
      <<["a", 1], ["b", 2], ["c", 3],
        ["a", 4], ["b", 5], ["c", 6],
        ["d", 1], ["e", 2], ["f", 3]>>)
    create ds.make_from_array (<<"a">>)
    -- r is not changed by the query 'domain_subtracted'
    t := r.domain_subtracted (ds)
    Result :=
      t /~ r and not t.domain.has ("a") and r.domain.has ("a")
    check Result end
    -- r is changed by the command 'domain_subtract'
    r.domain_subtract (ds)
    Result :=
      t ~ r and not t.domain.has ("a") and not r.domain.has ("a")
  end
```



## Case Study: A Birthday Book

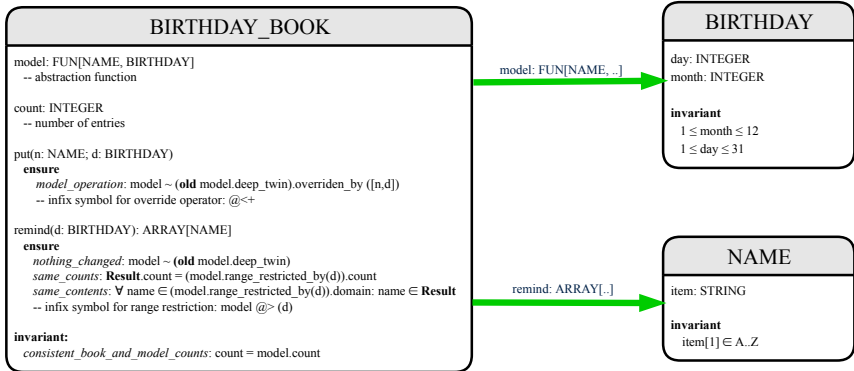
---

- A birthday book stores a collection of entries, where each entry is a pair of a person's name and their birthday.
- No two entries stored in the book are allowed to have the same name.
- Each birthday is characterized by a month and a day.
- A birthday book is first created to contain an empty collection of entries.
- Given a birthday book, we may:
  - Inquire about the number of entries currently stored in the book
  - Add a new entry by supplying its name and the associated birthday
  - Remove the entry associated with a particular person
  - Find the birthday of a particular person
  - Get a reminder list of names of people who share a given birthday

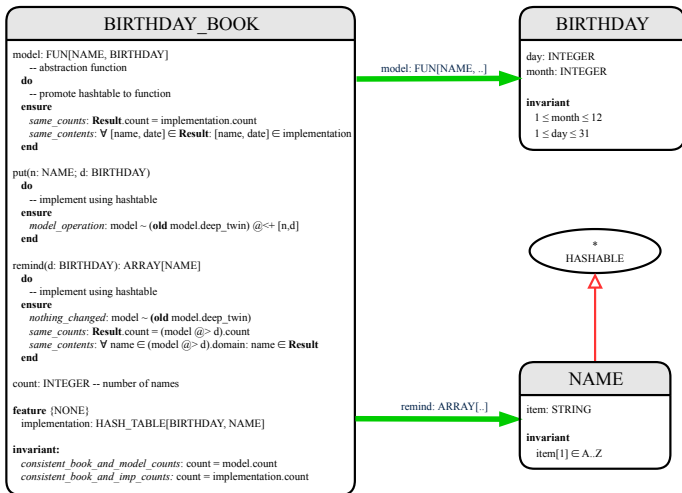
# Birthday Book: Decisions

- **Design** Decision
  - Classes
  - Client Supplier vs. Inheritance
  - Mathematical Model? [ e.g., REL or FUN ]
  - Contracts
- **Implementation** Decision
  - Two linear structures (e.g., arrays, lists) [  $O(n)$  ]
  - A balanced search tree (e.g., AVL tree) [  $O(\log \cdot n)$  ]
  - A hash table [  $O(1)$  ]
- Implement an **abstract function** that maps implementation to the math model.

# Birthday Book: Design



# Birthday Book: Implementation



## Beyond this lecture . . .

---

- Familiarize yourself with the features of classes `SEQ`, `REL`, `FUN`, and `SET` for the lab test.
- Play with the source code of the Birthday Book example:  
<https://www.eecs.yorku.ca/~jackie/teaching/lectures/2020/W/EECS3311/codes/birthday-book.zip>.
- **Exercise:**
  - Consider an alternative implementation using two linear structures (e.g., here in Java).
  - Implement the design of birthday book covered in lectures.
  - Create another `LINEAR_BIRTHDAY_BOOK` class and modify the implementation of abstraction function accordingly.  
Do all contracts still pass? What should change? What remain unchanged?

# Index (1)

---

**Motivating Problem: Complete Contracts**

**Motivating Problem: LIFO Stack (1)**

**Motivating Problem: LIFO Stack (2.1)**

**Motivating Problem: LIFO Stack (2.2)**

**Motivating Problem: LIFO Stack (2.3)**

**Design Principles:**

**Information Hiding & Single Choice**

**Motivating Problem: LIFO Stack (3)**

**Math Models: Command vs Query**

**Implementing an Abstraction Function (1)**

**Abstracting ADTs as Math Models (1)**

## Index (2)

---

**Implementing an Abstraction Function (2)**

**Abstracting ADTs as Math Models (2)**

**Implementing an Abstraction Function (3)**

**Abstracting ADTs as Math Models (3)**

**Solution: Abstracting ADTs as Math Models**

**Math Review: Set Definitions and Membership**

**Math Review: Set Relations**

**Math Review: Set Operations**

**Math Review: Power Sets**

**Math Review: Set of Tuples**

**Math Models: Relations (1)**

# Index (3)

---

**Math Models: Relations (2)**

**Math Models: Relations (3.1)**

**Math Models: Relations (3.2)**

**Math Models: Relations (3.3)**

**Math Review: Functions (1)**

**Math Review: Functions (2)**

**Math Review: Functions (3.1)**

**Math Review: Functions (3.2)**

**Math Models: Command-Query Separation**

**Math Models: Example Test**

**Case Study: A Birthday Book**



## Index (4)

---

**Birthday Book: Decisions**

**Birthday Book: Design**

**Birthday Book: Implementation**

**Beyond this lecture ...**