# Copies: Reference vs. Shallow vs. Deep
# Writing Complete Postconditions

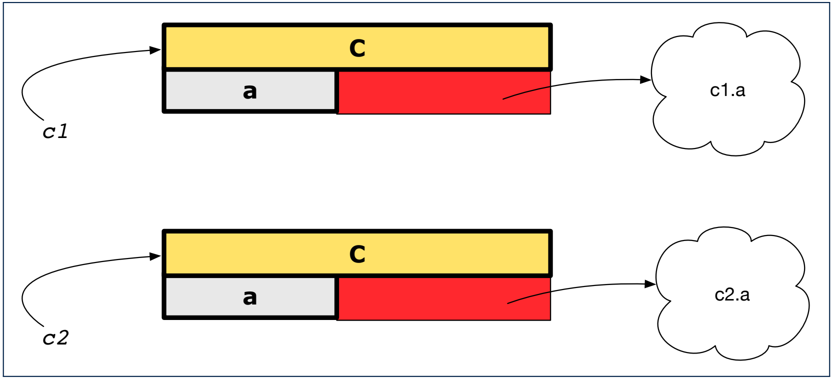
EECS3311 A: Software Design
Fall 2019

CHEN-WEI WANG

## 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` are references to objects.

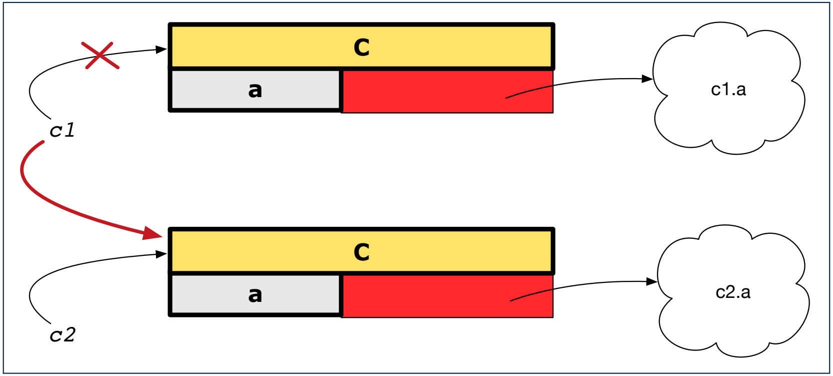


# Copying Objects: Reference Copy

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**

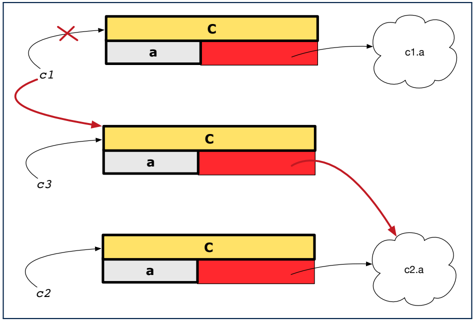
`c1 := c2.twin`

- 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:

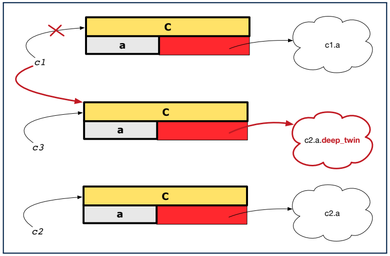
  **Base Case**: `a` is primitive (e.g., `INTEGER`). ⇒ `c3.a := c2.a`.

  **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

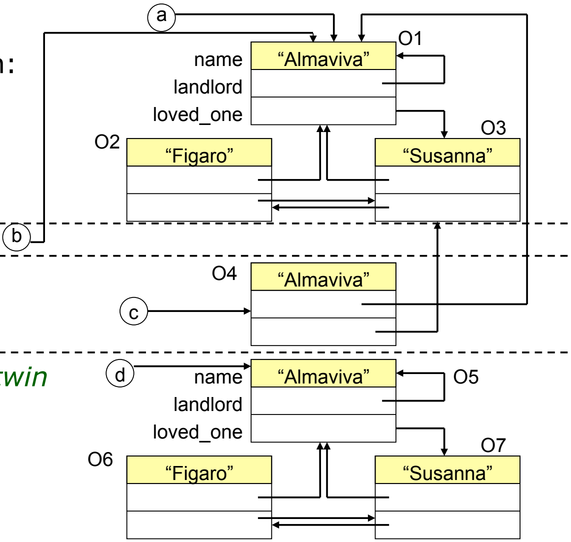
- Initial situation:

- Result of:

*b* := *a*

*c* := *a.twin*

*d* := *a.deep_twin*



## Example: Collection Objects (1)

- In any OOPL, when a variable is declared of a ***type*** that corresponds to a ***known class*** (e.g., STRING, ARRAY, LINKED_LIST, etc.):

  At *runtime*, that variable stores the *address* of an object of that type (as opposed to storing the object in its entirety).

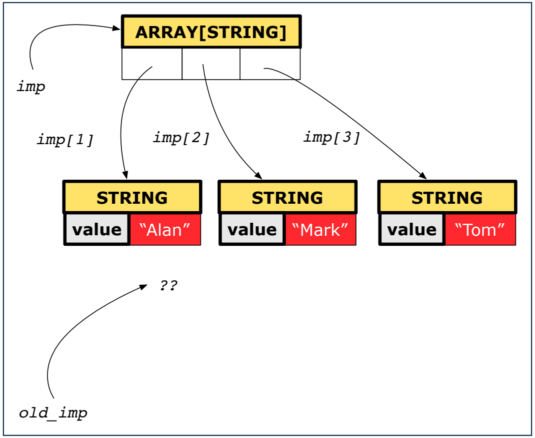
- Assume the following variables of the same type:

```
local
  imp    : ARRAY[STRING]
  old_imp: ARRAY[STRING]
do
  create {ARRAY[STRING]} imp.make_empty
  imp.force("Alan", 1)
  imp.force("Mark", 2)
  imp.force("Tom", 3)
```

  - ***Before*** we undergo a change on imp, we "*copy*" it to old_imp.
  - ***After*** the change is completed, we compare imp vs. old_imp.
  - Can a change always be *visible* between "***old***" and "***new***" imp?

## Example: Collection Objects (2)

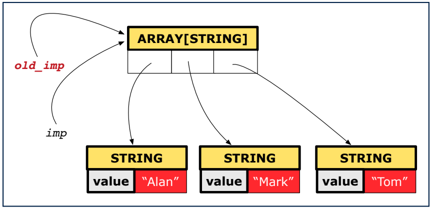
- Variables `imp` and `old_imp` store address(es) of some array(s).
- Each "slot" of these arrays stores a `STRING` object's address.



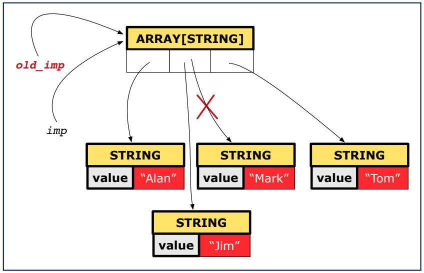
## Reference Copy of Collection Object

```
old_imp := imp
Result := old_imp = imp  -- Result = true
imp[2] := "Jim"
Result :=
  across 1 |..| imp.count is j
  all imp [j] ~ old_imp [j]
  end -- Result = true
```

Before Executing L3



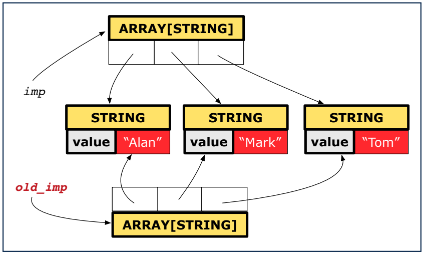
After Executing L3



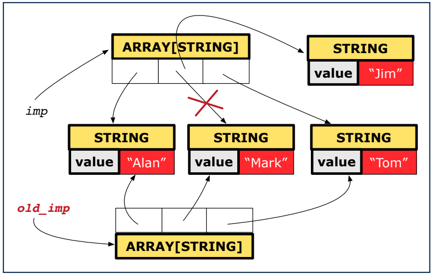
## Shallow Copy of Collection Object (1)

```
old_imp := imp.twin
Result := old_imp = imp  -- Result = false
imp[2] := "Jim"
Result :=
  across 1 |..| imp.count is j
  all imp [j] ~ old_imp [j]
  end -- Result = false
```

Before Executing L3



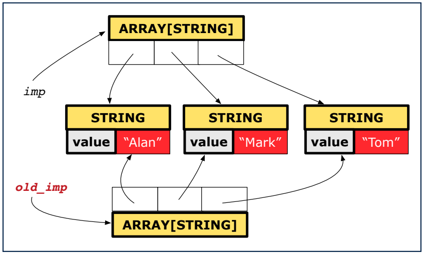
After Executing L3



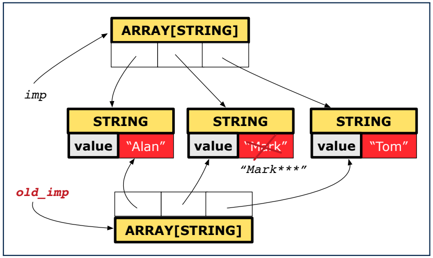
# Shallow Copy of Collection Object (2)

```
old_imp := imp.twin
Result := old_imp = imp  -- Result = false
imp[2].append ("***")
Result :=
  across 1 |..| imp.count is j
  all imp [j] ~ old_imp [j]
  end -- Result = true
```

Before Executing L3



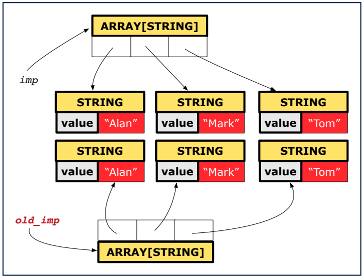
After Executing L3



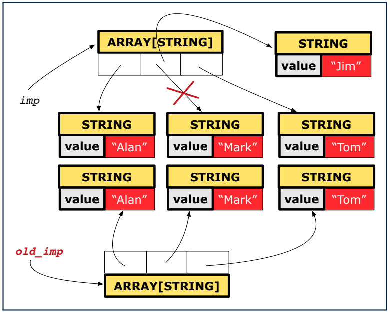
## Deep Copy of Collection Object (1)

```
old_imp := imp.deep_twin
Result := old_imp = imp   -- Result = false
imp[2] := "Jim"
Result :=
  across 1 |..| imp.count is j
  all imp [j] ~ old_imp [j] end -- Result = false
```

Before Executing L3



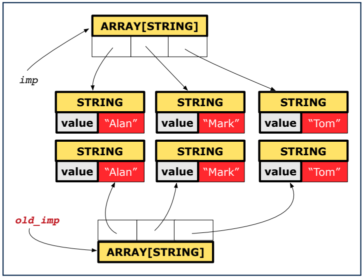
After Executing L3



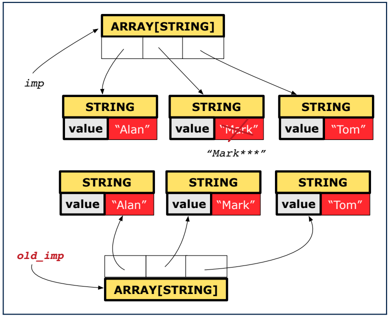
## Deep Copy of Collection Object (2)

```
old_imp := imp.deep_twin
Result := old_imp = imp  -- Result = false
imp[2].append ("***")
Result :=
  across 1 |..| imp.count is j
  all imp [j] ~ old_imp [j] end -- Result = false
```

Before Executing L3



After Executing L3



## How are contracts checked at runtime?

- All contracts are specified as Boolean expressions.
- Right **before** a feature call (e.g., `acc.withdraw(10)`):
  - The current state of `acc` is called its ***pre-state***.
  - Evaluate *pre-condition* using ***current values*** of attributes/queries.
  - Cache values, via $\boxed{:=}$, of ***old** expressions* in the *post-condition*.

| | | |
|---|---|---|
| e.g., | **old** *accounts[i].id* | [ *old_accounts_i_id* := *accounts[i].id* ] |
| e.g., | (**old** *accounts[i]*).*id* | [ *old_accounts_i* := *accounts[i]* ] |
| e.g., | (**old** *accounts[i].**twin***).*id* | [ *old_accounts_i_twin* := *accounts[i].**twin*** ] |
| e.g., | (**old** *accounts*)[*i*].*id* | [ *old_accounts* := *accounts* ] |
| e.g., | (**old** *accounts.**twin***)[*i*].*id* | [ *old_accounts_twin* := *accounts.**twin*** ] |
| e.g., | (**old** ***Current***).*accounts*[*i*].*id* | [ *old_current* := ***Current*** ] |
| e.g., | (**old** ***Current.twin***).*accounts*[*i*].*id* | [ *old_current_twin* := ***Current.twin*** ] |

- Right **after** the feature call:
  - The current state of `acc` is called its ***post-state***.
  - Evaluate *invariant* using ***current values*** of attributes and queries.
  - Evaluate *post-condition* using both ***current values*** and ***"cached" values*** of attributes and queries.

## When are contracts complete?

- In *post-condition*, for *each attribute*, specify the relationship between its ***pre-state*** value and its ***post-state*** value.
  - Eiffel supports this purpose using the **old** keyword.
- This is tricky for attributes whose structures are **composite** rather than **simple**:

  e.g., *ARRAY*, *LINKED_LIST* are composite-structured.
  e.g., *INTEGER*, *BOOLEAN* are simple-structured.
- **Rule of thumb:** For an attribute whose structure is composite, we should specify that after the update:
  1. The intended change is present; **and**
  2. *The rest of the structure is unchanged*.
- The second contract is much harder to specify:
  - Reference aliasing [ ref copy vs. shallow copy vs. deep copy ]
  - Iterable structure [ use `across` ]

## Account

```
class
 ACCOUNT

inherit
 ANY
  redefine is_equal end

create
 make

feature -- Attributes
 owner: STRING
 balance: INTEGER

feature -- Commands
 make (n: STRING)
  do
   owner := n
   balance := 0
  end
```

```
 deposit(a: INTEGER)
  do
   balance := balance + a
  ensure
   balance = old balance + a
  end

 is_equal(other: ACCOUNT): BOOLEAN
  do
   Result :=
         owner ~ other.owner
     and balance = other.balance
  end
end
```

## Bank

```
class BANK
create make
feature
 accounts: ARRAY[ACCOUNT]
 make do create accounts.make_empty end
 account_of (n: STRING): ACCOUNT
  require -- the input name exists
   existing: across accounts is acc some acc.owner ~ n end
    -- not (across accounts is acc all acc.owner /~ n end)
  do ... ensure Result.owner ~ n end
 add (n: STRING)
  require -- the input name does not exist
   non_existing: across accounts is acc all acc.owner /~ n end
    -- not (across accounts is acc some acc.owner ~ n end)
  local new_account: ACCOUNT
  do
   create new_account.make (n)
   accounts.force (new_account, accounts.upper + 1)
  end
end
```

## Roadmap of Illustrations

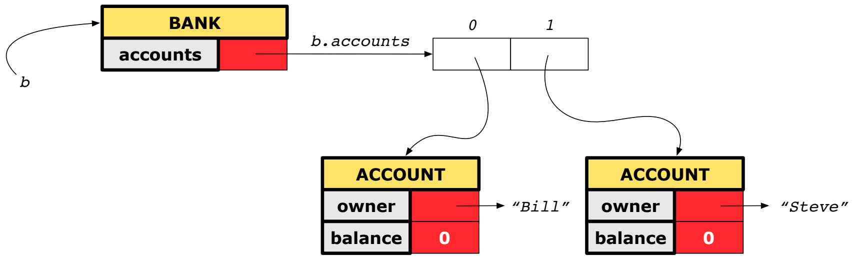
We examine 5 different versions of a command

$$\textit{deposit\_on}\ (n : \textit{STRING};\ a : \textit{INTEGER})$$

| VERSION | IMPLEMENTATION | CONTRACTS | SATISFACTORY? |
|---|---|---|---|
| 1 | *Correct* | *Incomplete* | *No* |
| 2 | *Wrong* | *Incomplete* | *No* |
| 3 | *Wrong* | *Complete* (reference copy) | *No* |
| 4 | *Wrong* | *Complete* (shallow copy) | *No* |
| 5 | *Wrong* | *Complete* (deep copy) | *Yes* |

# Object Structure for Illustration

We will test each version by starting with the same runtime object structure:



## Version 1: Incomplete Contracts, Correct Implementation

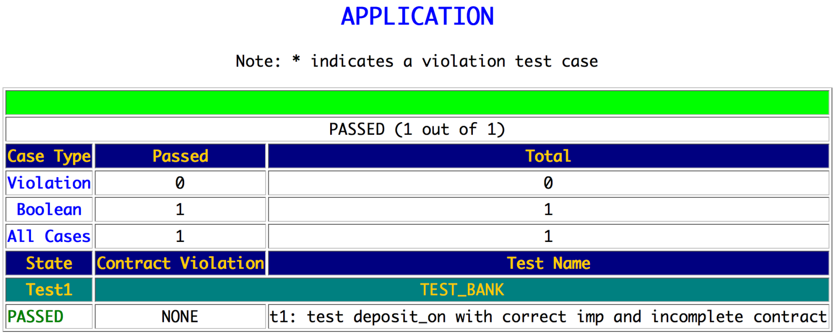
```
class BANK
  deposit_on_v1 (n: STRING; a: INTEGER)
    require across accounts is acc some acc.owner ~ n end
    local i: INTEGER
    do
      from i := accounts.lower
      until i > accounts.upper
      loop
        if accounts[i].owner ~ n then accounts[i].deposit(a) end
        i := i + 1
      end
    ensure
      num_of_accounts_unchanged:
        accounts.count = old accounts.count
      balance_of_n_increased:
        Current.account_of(n).balance =
          old Current.account_of(n).balance + a
    end
end
```

## Test of Version 1

```
class TEST_BANK
 test_bank_deposit_correct_imp_incomplete_contract: BOOLEAN
   local
     b: BANK
   do
     comment("t1: correct imp and incomplete contract")
     create b.make
     b.add ("Bill")
     b.add ("Steve")

     -- deposit 100 dollars to Steve's account
     b.deposit_on_v1 ("Steve", 100)
     Result :=
          b.account_of("Bill").balance = 0
      and b.account_of("Steve").balance = 100
     check Result end
 end
end
```

## Test of Version 1: Result



APPLICATION

Note: * indicates a violation test case

| | | |
|---|---|---|
| 🟩 | | |
| PASSED (1 out of 1) | | |
| Case Type | Passed | Total |
| Violation | 0 | 0 |
| Boolean | 1 | 1 |
| All Cases | 1 | 1 |
| State | Contract Violation | Test Name |
| Test1 | TEST_BANK | |
| PASSED | NONE | t1: test deposit_on with correct imp and incomplete contract |

## Version 2:
## Incomplete Contracts, Wrong Implementation

```
class BANK
 deposit_on_v2 (n: STRING; a: INTEGER)
  require across accounts is acc some acc.owner ~ n end
  local i: INTEGER
  do ...
   -- imp. of version 1, followed by a deposit into 1st account
   accounts[accounts.lower].deposit(a)
  ensure
   num_of_accounts_unchanged:
    accounts.count = old accounts.count
   balance_of_n_increased:
    Current.account_of(n).balance =
     old Current.account_of(n).balance + a
  end
end
```

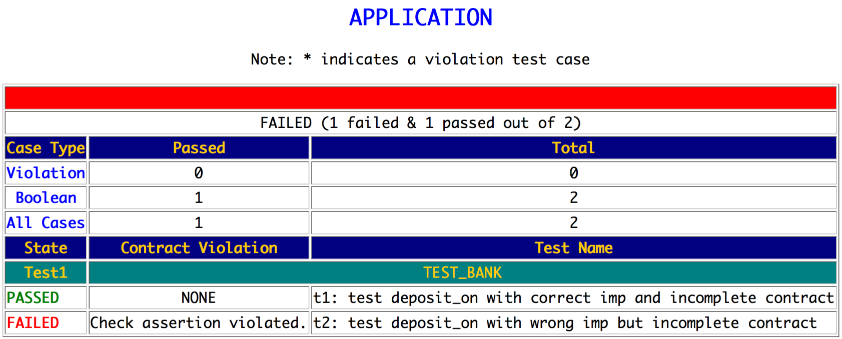
Current postconditions lack a check that accounts other than `n` are unchanged.

## Test of Version 2

```
class TEST_BANK
test_bank_deposit_wrong_imp_incomplete_contract: BOOLEAN
  local
    b: BANK
  do
    comment("t2: wrong imp and incomplete contract")
    create b.make
    b.add ("Bill")
    b.add ("Steve")

    -- deposit 100 dollars to Steve's account
    b.deposit_on_v2 ("Steve", 100)
    Result :=
          b.account_of("Bill").balance = 0
      and b.account_of("Steve").balance = 100
    check Result end
  end
end
```

## Test of Version 2: Result



APPLICATION

Note: * indicates a violation test case

| | | |
|---|---|---|
| 🟥 | | |
| FAILED (1 failed & 1 passed out of 2) | | |
| Case Type | Passed | Total |
| Violation | 0 | 0 |
| Boolean | 1 | 2 |
| All Cases | 1 | 2 |
| State | Contract Violation | Test Name |
| Test1 | TEST_BANK | |
| PASSED | NONE | t1: test deposit_on with correct imp and incomplete contract |
| FAILED | Check assertion violated. | t2: test deposit_on with wrong imp but incomplete contract |

## Version 3: Complete Contracts with Reference Copy

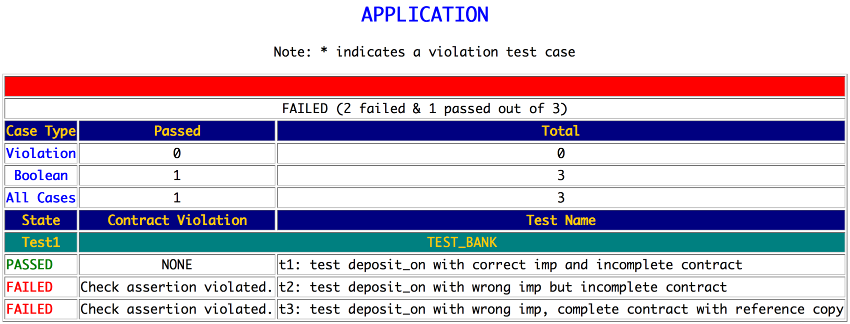
```
class BANK
 deposit_on_v3 (n: STRING; a: INTEGER)
   require across accounts is acc some acc.owner ~ n end
   local i: INTEGER
   do ...
     -- imp. of version 1, followed by a deposit into 1st account
     accounts[accounts.lower].deposit(a)
   ensure
     num_of_accounts_unchanged: accounts.count = old accounts.count
     balance_of_n_increased:
       Current.account_of(n).balance =
         old Current.account_of(n).balance + a
     others_unchanged :
       across old accounts is acc
       all
         acc.owner /~ n implies acc ~ Current.account_of(acc.owner)
       end
   end
end
```

## Test of Version 3

```
class TEST_BANK
 test_bank_deposit_wrong_imp_complete_contract_ref_copy: BOOLEAN
   local
     b: BANK
   do
     comment("t3: wrong imp and complete contract with ref copy")
     create b.make
     b.add ("Bill")
     b.add ("Steve")

     -- deposit 100 dollars to Steve's account
     b.deposit_on_v3 ("Steve", 100)
     Result :=
          b.account_of("Bill").balance = 0
      and b.account_of("Steve").balance = 100
     check Result end
   end
end
```

# Test of Version 3: Result



APPLICATION

Note: * indicates a violation test case

| | | |
|---|---|---|
| 🟥 | | |
| FAILED (2 failed & 1 passed out of 3) | | |
| Case Type | Passed | Total |
| Violation | 0 | 0 |
| Boolean | 1 | 3 |
| All Cases | 1 | 3 |
| State | Contract Violation | Test Name |
| Test1 | TEST_BANK | |
| PASSED | NONE | t1: test deposit_on with correct imp and incomplete contract |
| FAILED | Check assertion violated. | t2: test deposit_on with wrong imp but incomplete contract |
| FAILED | Check assertion violated. | t3: test deposit_on with wrong imp, complete contract with reference copy |

## Version 4:
## Complete Contracts with Shallow Object Copy

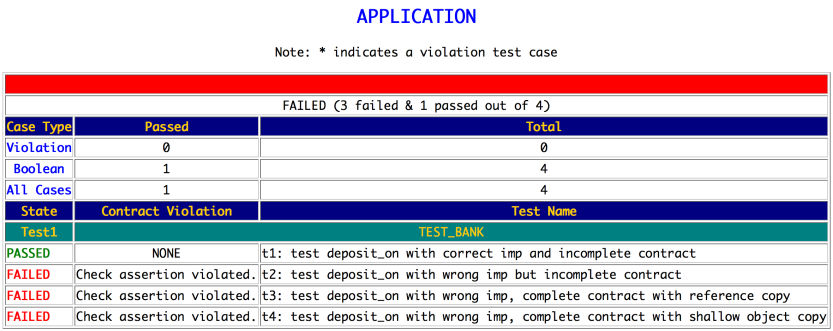
```
class BANK
  deposit_on_v4 (n: STRING; a: INTEGER)
    require across accounts is acc some acc.owner ~ n end
    local i: INTEGER
    do ...
      -- imp. of version 1, followed by a deposit into 1st account
      accounts[accounts.lower].deposit(a)
    ensure
      num_of_accounts_unchanged: accounts.count = old accounts.count
      balance_of_n_increased:
        Current.account_of(n).balance =
          old Current.account_of(n).balance + a
      others_unchanged :
        across old accounts.twin is acc
        all
          acc.owner /~ n implies acc ~ Current.account_of(acc.owner)
        end
    end
end
```

## Test of Version 4

```
class TEST_BANK
 test_bank_deposit_wrong_imp_complete_contract_shallow_copy: BOOLEAN
   local
     b: BANK
   do
     comment("t4: wrong imp and complete contract with shallow copy")
     create b.make
     b.add ("Bill")
     b.add ("Steve")

     -- deposit 100 dollars to Steve's account
     b.deposit_on_v4 ("Steve", 100)
     Result :=
          b.account_of("Bill").balance = 0
      and b.account_of("Steve").balance = 100
     check Result end
   end
end
```

## Test of Version 4: Result



APPLICATION

Note: * indicates a violation test case

FAILED (3 failed & 1 passed out of 4)

| Case Type | Passed | Total |
|---|---|---|
| Violation | 0 | 0 |
| Boolean | 1 | 4 |
| All Cases | 1 | 4 |
| **State** | **Contract Violation** | **Test Name** |
| Test1 | TEST_BANK | |
| PASSED | NONE | t1: test deposit_on with correct imp and incomplete contract |
| FAILED | Check assertion violated. | t2: test deposit_on with wrong imp but incomplete contract |
| FAILED | Check assertion violated. | t3: test deposit_on with wrong imp, complete contract with reference copy |
| FAILED | Check assertion violated. | t4: test deposit_on with wrong imp, complete contract with shallow object copy |

# Version 5:
# Complete Contracts with Deep Object Copy

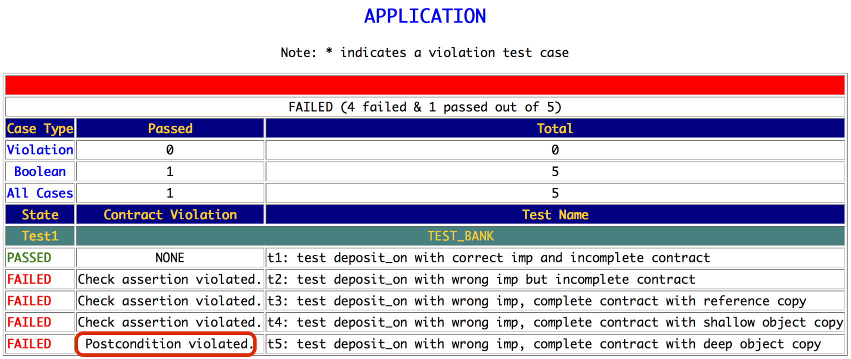
```
class BANK
 deposit_on_v5 (n: STRING; a: INTEGER)
   require across accounts is acc some acc.owner ~ n end
     local i: INTEGER
   do ...
     -- imp. of version 1, followed by a deposit into 1st account
     accounts[accounts.lower].deposit(a)
   ensure
     num_of_accounts_unchanged: accounts.count = old accounts.count
     balance_of_n_increased:
       Current.account_of(n).balance =
         old Current.account_of(n).balance + a
     others_unchanged :
       across old accounts.deep_twin is acc
       all
         acc.owner /~ n implies acc ~ Current.account_of(acc.owner)
       end
   end
end
```

## Test of Version 5

```
class TEST_BANK
 test_bank_deposit_wrong_imp_complete_contract_deep_copy: BOOLEAN
   local
     b: BANK
   do
     comment("t5: wrong imp and complete contract with deep copy")
     create b.make
     b.add ("Bill")
     b.add ("Steve")

     -- deposit 100 dollars to Steve's account
     b.deposit_on_v5 ("Steve", 100)
     Result :=
          b.account_of("Bill").balance = 0
      and b.account_of("Steve").balance = 100
     check Result end
   end
end
```

# Test of Version 5: Result



APPLICATION

Note: * indicates a violation test case

| FAILED (4 failed & 1 passed out of 5) | | |
|---|---|---|
| Case Type | Passed | Total |
| Violation | 0 | 0 |
| Boolean | 1 | 5 |
| All Cases | 1 | 5 |
| State | Contract Violation | Test Name |
| Test1 | TEST_BANK | |
| PASSED | NONE | t1: test deposit_on with correct imp and incomplete contract |
| FAILED | Check assertion violated. | t2: test deposit_on with wrong imp but incomplete contract |
| FAILED | Check assertion violated. | t3: test deposit_on with wrong imp, complete contract with reference copy |
| FAILED | Check assertion violated. | t4: test deposit_on with wrong imp, complete contract with shallow object copy |
| FAILED | Postcondition violated. | t5: test deposit_on with wrong imp, complete contract with deep object copy |

## Exercise

- Consider the query *account_of (n: STRING)* of *BANK*.
- How do we specify (part of) its postcondition to assert that the state of the bank remains unchanged:
  - `accounts = old accounts` [ × ]
  - `accounts = old accounts.twin` [ × ]
  - `accounts = old accounts.deep_twin` [ × ]
  - `accounts ~ old accounts` [ × ]
  - `accounts ~ old accounts.twin` [ × ]
  - `accounts ~ old accounts.deep_twin` [ ✓ ]
- Which equality of the above is appropriate for the postcondition?
- Why is each one of the other equalities not appropriate?

## Index (1)

Copying Objects
Copying Objects: Reference Copy
Copying Objects: Shallow Copy
Copying Objects: Deep Copy
Example: Copying Objects
Example: Collection Objects (1)
Example: Collection Objects (2)
Reference Copy of Collection Object
Shallow Copy of Collection Object (1)
Shallow Copy of Collection Object (2)
Deep Copy of Collection Object (1)
Deep Copy of Collection Object (2)
How are contracts checked at runtime?
When are contracts complete?

## Index (2)

Account

Bank

Roadmap of Illustrations

Object Structure for Illustration

Version 1:
Incomplete Contracts, Correct Implementation

Test of Version 1

Test of Version 1: Result

Version 2:
Incomplete Contracts, Wrong Implementation

Test of Version 2

Test of Version 2: Result

Version 3:
Complete Contracts with Reference Copy

Test of Version 3

## Index (3)

Test of Version 3: Result

Version 4:
Complete Contracts with Shallow Object Copy

Test of Version 4

Test of Version 4: Result

Version 5:
Complete Contracts with Deep Object Copy

Test of Version 5

Test of Version 5: Result

Exercise