

Copying Objects

Writing Complete Postconditions



EECS3311 A & E: Software Design
Fall 2020

CHEN-WEI WANG

Learning Objectives



Upon completing this lecture, you are expected to understand:

- 3 Levels of **Copying Objects**:
Reference vs. Shallow vs. Deep
- Use of the **old keyword** in Postconditions
- Writing **Complete Postconditions** using logical quantifications:
Universal (\forall) vs. Existential (\exists)

Part 1



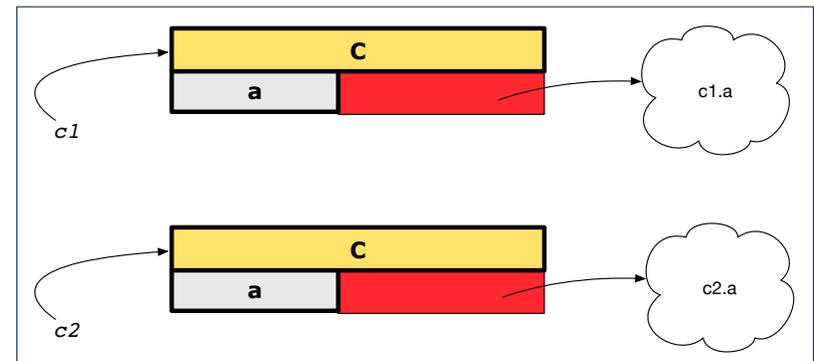
Copying Objects

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.



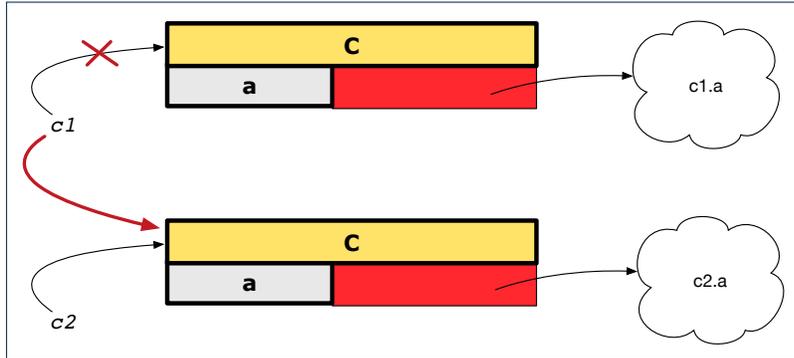
Copying Objects: Reference Copy



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*]



5 of 41

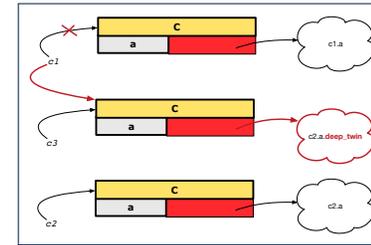
Copying Objects: Deep Copy



Deep Copy

`c1 := c2.deep_twin`

- Create a temporary, behind-the-scenes 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` 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.



7 of 41

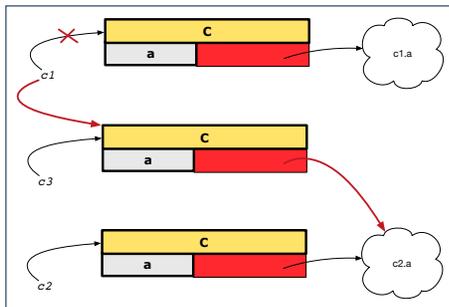
Copying Objects: Shallow Copy



Shallow Copy

`c1 := c2.twin`

- Create a temporary, behind-the-scenes 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`)

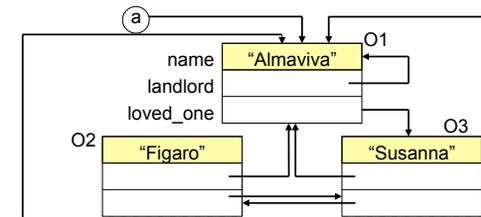


6 of 41

Copying Objects



- Initial situation:

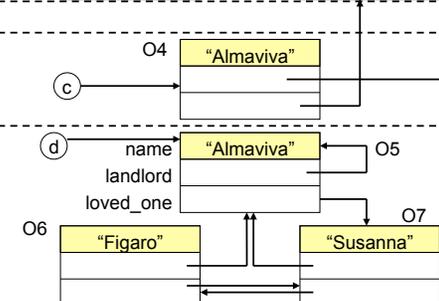


- Result of:

`b := a`

`c := a.twin`

`d := a.deep_twin`



8 of 41

Example: Collection Objects (1)

- In any OOP, 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`?

Reference Copy of Collection Object

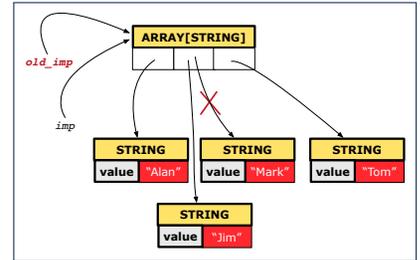
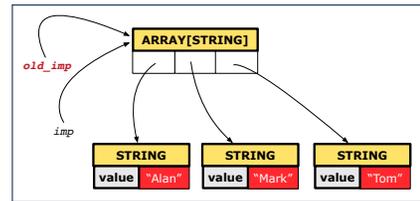
```

1  old_imp := imp
2  Result := old_imp = imp -- Result = true
3  imp[2] := "Jim"
4  Result :=
5  across 1 |..| imp.count is j
6  all imp [j] ~ old_imp [j]
7  end -- Result = true

```

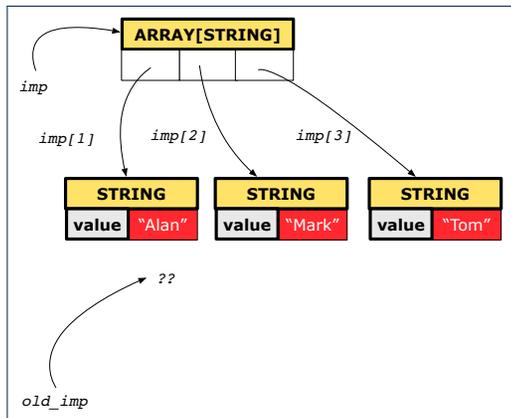
Before Executing L3

After Executing L3



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.



Shallow Copy of Collection Object (1)

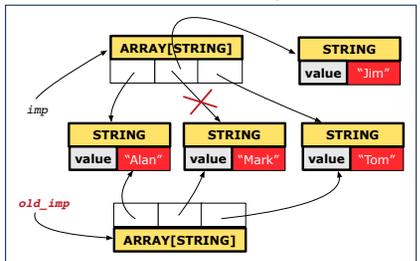
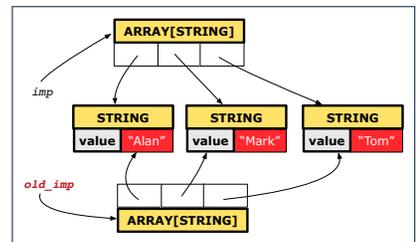
```

1  old_imp := imp.twin
2  Result := old_imp = imp -- Result = false
3  imp[2] := "Jim"
4  Result :=
5  across 1 |..| imp.count is j
6  all imp [j] ~ old_imp [j]
7  end -- Result = false

```

Before Executing L3

After Executing L3



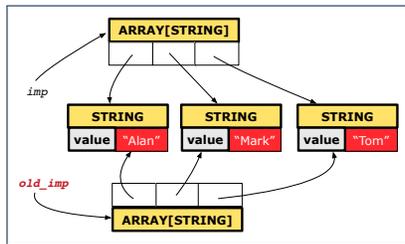
Shallow Copy of Collection Object (2)



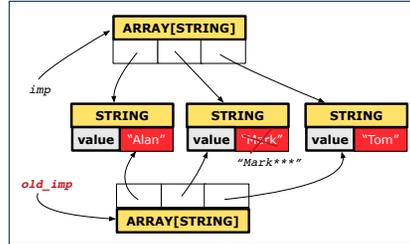
```

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

Before Executing L3



After Executing L3



13 of 41

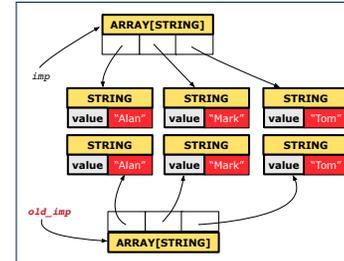
Deep Copy of Collection Object (2)



```

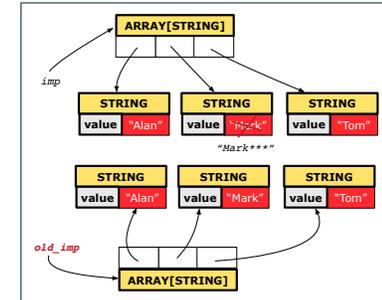
1  old_imp := imp.deep.twin
2  Result := old_imp = imp -- Result = false
3  imp[2].append ("***")
4  Result :=
5  across 1 |..| imp.count is j
6  all imp [j] ~ old_imp [j] end -- Result = false
    
```

Before Executing L3



15 of 41

After Executing L3



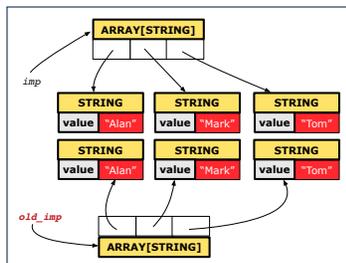
Deep Copy of Collection Object (1)



```

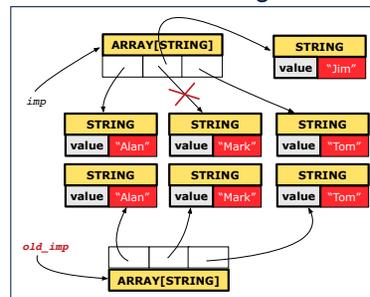
1  old_imp := imp.deep.twin
2  Result := old_imp = imp -- Result = false
3  imp[2] := "Jim"
4  Result :=
5  across 1 |..| imp.count is j
6  all imp [j] ~ old_imp [j] end -- Result = false
    
```

Before Executing L3



14 of 41

After Executing L3



Experiment: Copying Objects



- Download the Eiffel project archive (a zip file) here:

https://www.eecs.yorku.ca/~jackie/teaching/lectures/2020/F/EECS3311/codes/copying_objects.zip

- Unzip and compile the project in Eiffel Studio.
- Reproduce the illustrations explained in lectures.

16 of 41

Writing Complete Postconditions

17 of 41

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:
 - The intended change is present; **and**
 - 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*]

19 of 41

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 `:=`, 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 **post-condition** using both **current values** and **"cached" values** of attributes and queries.
 - Evaluate **invariant** using **current values** of attributes and queries.

18 of 41

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
```

20 of 41

Bank



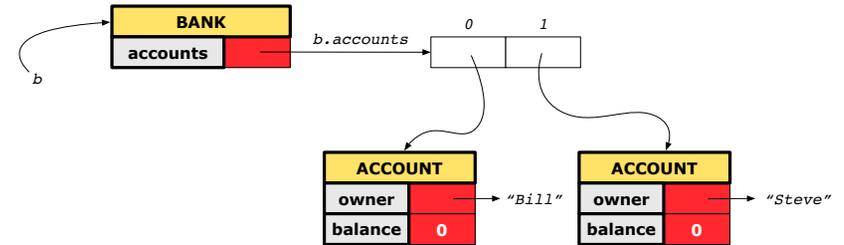
```

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
    
```

Object Structure for Illustration



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



Roadmap of Illustrations



We examine 5 different versions of a command

deposit_on (n: STRING; a: 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

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
  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
    
```

25 of 41

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.

27 of 41

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

26 of 41

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
    
```

28 of 41

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

29 of 41

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
    
```

31 of 41

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
end
    
```

30 of 41

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

32 of 41

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
end

```

33 of 41

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

35 of 41

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

```

34 of 41

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
end
end

```

36 of 41

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
  
```

37 of 41

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

38 of 41

Experiment: Complete Postconditions



- Download the Eiffel project archive (a zip file) here:

<https://www.eecs.yorku.ca/~jackie/teaching/lectures/2020/F/>

[EECS3311/codes/array_math_contract.zip](#)

- Unzip and compile the project in Eiffel Studio.
- Reproduce the illustrations explained in lectures.

39 of 41

Beyond this lecture



- 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?

40 of 41

Index (1)



Learning Objectives

Part 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)

41 of 41

Index (2)



Shallow Copy of Collection Object (2)

Deep Copy of Collection Object (1)

Deep Copy of Collection Object (2)

Experiment: Copying Objects

Part 2

How are contracts checked at runtime?

When are contracts complete?

Account

Bank

Roadmap of Illustrations

Object Structure for Illustration

42 of 41

Index (3)



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

Test of Version 3: Result

43 of 41

Index (4)



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

Experiment: Complete Postconditions

Beyond this lecture

44 of 41