Types: Reference vs. Expanded
Copies: Reference vs. Shallow vs. Deep
Writing Complete Postconditions
Expanded Class: Modelling

- We may want to have objects which are:
  - Integral parts of some other objects
  - *Not* shared among objects

  e.g., Each workstation has its own CPU, monitor, and keyboard. All workstations share the same network.
Expanded Class: Programming (2)

class KEYBOARD ... end class CPU ... end
class MONITOR ... end class NETWORK ... end
class WORKSTATION
  k: expanded KEYBOARD
  c: expanded CPU
  m: expanded MONITOR
  n: NETWORK
end

Alternatively:

expanded class KEYBOARD ... end
expanded class CPU ... end
expanded class MONITOR ... end
class NETWORK ... end
class WORKSTATION
  k: KEYBOARD
  c: CPU
  m: MONITOR
  n: NETWORK
end
Expanded Class: Programming (3)

```
expanded class
  B
feature
  change_i (ni: INTEGER)
    do
      i := ni
    end
feature
  i: INTEGER
end

test_expanded: BOOLEAN
  local
    eb1, eb2: B
  do
    Result := eb1.i = 0 and eb2.i = 0
    check Result end
    Result := eb1 = eb2
    check Result end
    eb2.change_i (15)
    Result := eb1.i = 0 and eb2.i = 15
    check Result end
    Result := eb1 /= eb2
    check Result end
  end
```

- L5: object of expanded type is automatically initialized.
- L9 & L10: no sharing among objects of expanded type.
- L7 & L12: = between expanded objects compare their contents.
Reference vs. Expanded (1)

- Every entity must be declared to be of a certain type (based on a class).
- Every type is either referenced or expanded.
- In reference types:
  - $y$ denotes a reference to some object
  - $x := y$ attaches $x$ to same object as does $y$
  - $x = y$ compares references
- In expanded types:
  - $y$ denotes some object (of expanded type)
  - $x := y$ copies contents of $y$ into $x$
  - $x = y$ compares contents

\[x \sim y\]
Problem: Every published book has an author. Every author may publish more than one books. Should the author field of a book reference-typed or expanded-typed?

**reference**-typed author

```
<table>
<thead>
<tr>
<th>“The Red and the Black”</th>
</tr>
</thead>
<tbody>
<tr>
<td>1830</td>
</tr>
<tr>
<td>341</td>
</tr>
<tr>
<td>reference</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>“Life of Rossini”</th>
</tr>
</thead>
<tbody>
<tr>
<td>1823</td>
</tr>
<tr>
<td>307</td>
</tr>
<tr>
<td>reference</td>
</tr>
</tbody>
</table>
```

```
<table>
<thead>
<tr>
<th>“Stendhal”</th>
</tr>
</thead>
<tbody>
<tr>
<td>“Henri Beyle”</td>
</tr>
<tr>
<td>1783</td>
</tr>
<tr>
<td>1842</td>
</tr>
</tbody>
</table>
```

**expanded**-typed author

```
<table>
<thead>
<tr>
<th>“The Red and the Black”</th>
</tr>
</thead>
<tbody>
<tr>
<td>1830</td>
</tr>
<tr>
<td>341</td>
</tr>
<tr>
<td>reference</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>“Life of Rossini”</th>
</tr>
</thead>
<tbody>
<tr>
<td>1823</td>
</tr>
<tr>
<td>307</td>
</tr>
</tbody>
</table>
```

```
<table>
<thead>
<tr>
<th>“Stendhal”</th>
</tr>
</thead>
<tbody>
<tr>
<td>“Henri Beyle”</td>
</tr>
<tr>
<td>1783</td>
</tr>
<tr>
<td>1842</td>
</tr>
</tbody>
</table>
```

```
<table>
<thead>
<tr>
<th>“Stendhal”</th>
</tr>
</thead>
<tbody>
<tr>
<td>“Henri Beyle”</td>
</tr>
<tr>
<td>1783</td>
</tr>
<tr>
<td>1842</td>
</tr>
</tbody>
</table>
```
Copying Objects

Say variables $c_1$ and $c_2$ are both declared of type $C$. $[c_1, c_2 : C]$

- There is only one attribute $a$ declared in class $C$.
- $c_1.a$ and $c_2.a$ may be of either:
  - *expanded* type or
  - *reference* type
Reference Copy

- 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 \( c2 \).

\[
c1 := c2
\]
Copying Objects: Shallow Copy

**Shallow Copy**

- 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` and `c2` *are not* pointing to the same object.  
  - `c1.a` and `c2.a` *are* pointing to the same object.
  - *Aliasing* still occurs: at 1st level (i.e., attributes of `c1` and `c2`)

```
c1 := c2.twin
```

![Diagram showing shallow copy process]
Copying Objects: Deep Copy

**Deep Copy**

- Create a temporary, behind-the-scene object \( c_3 \) of type \( C \).
- **Recursively** initialize each attribute \( a \) of \( c_3 \) as follows:
  - **Base Case**: \( a \) is expanded (e.g., INTEGER).
    \[
    \Rightarrow c_3.a := c_2.a.
    \]
  - **Recursive Case**: \( a \) is referenced.
    \[
    \Rightarrow c_3.a := c_2.a.deep_twin
    \]
- Make a **reference copy** of \( c_3 \):
  - \( c_1 \) and \( c_2 \) are not pointing to the same object.
  - \( c_1.a \) and \( c_2.a \) are not pointing to the same object.
  - No aliasing occurs at any levels.

\[
\begin{align*}
& c_1 := c_2.deep_twin \\
& c_1 := c_3
\end{align*}
\]
Shallow and deep cloning

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:

```plaintext
...  
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)
...`
```
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.
old_imp := imp

Result := old_imp = imp  -- Result = true

imp[2] := "Jim"

Result :=
across 1 |..| imp.count as j
all imp[j.item] ~ old_imp[j.item]
end  -- Result = true

Before Executing L3

After Executing L3
Shallow Copy of Collection Object (1)

1. `old_imp := imp.twin`
2. `Result := old_imp = imp -- Result = false`
4. `Result :=`
5. `across 1 |..| imp.count as j`
6. `all imp [j.item] ~ old_imp [j.item]`
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 as j`
   6. `all imp [j.item] ~ old_imp [j.item]`
   7. `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 as j

all imp [j.item] ~ old_imp [j.item] end -- Result = false

Before Executing L3

After Executing L3
Deep Copy of Collection Object (2)

```plaintext
1. old_imp := imp.deep_twin
2. Result := old_imp = imp -- Result = false
3. imp[2].append("***")
4. Result :=
5.   across 1 |..| imp.count as j
6.   all imp [j.item] ~ old_imp [j.item] 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 `:=`, of `old expressions` in the **post-condition**.
    - e.g., `old balance = balance - a`  
      [ `old_balance := balance - a` ]
    - 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 Current).accounts[i].id`
      [ `old_current := Current` ]
- 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 ]
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
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 as acc some acc.item.owner ~ n end
            -- not (across accounts as acc all acc.item.owner /~ n end)
        do ...
        ensure Result.owner ~ n
    end
    add (n: STRING)
        require -- the input name does not exist
            non_existing: across accounts as acc all acc.item.owner /~ n end
            -- not (across accounts as acc some acc.item.owner ~ n end)
        local new_account: ACCOUNT
        do
            create new_account.make (n)
            accounts.force (new_account, accounts.upper + 1)
        end
Roadmap of Illustrations

We examine 5 different versions of a command

\[ \text{deposit\_on (n : STRING; a : INTEGER)} \]

<table>
<thead>
<tr>
<th>Version</th>
<th>Implementation</th>
<th>Contracts</th>
<th>Satisfactory?</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Correct</td>
<td>Incomplete</td>
<td>No</td>
</tr>
<tr>
<td>2</td>
<td>Wrong</td>
<td>Incomplete</td>
<td>No</td>
</tr>
<tr>
<td>3</td>
<td>Wrong</td>
<td>Complete (reference copy)</td>
<td>No</td>
</tr>
<tr>
<td>4</td>
<td>Wrong</td>
<td>Complete (shallow copy)</td>
<td>No</td>
</tr>
<tr>
<td>5</td>
<td>Wrong</td>
<td>Complete (deep copy)</td>
<td>Yes</td>
</tr>
</tbody>
</table>
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 as acc some acc.item.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:
            account_of (n).balance = old account_of (n).balance + a
      end
end
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
APPLICATION

Note: * indicates a violation test case

<table>
<thead>
<tr>
<th>Case Type</th>
<th>Passed</th>
<th>Total</th>
</tr>
</thead>
<tbody>
<tr>
<td>Violation</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>Boolean</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>All Cases</td>
<td>1</td>
<td>1</td>
</tr>
</tbody>
</table>

PASSED (1 out of 1)

<table>
<thead>
<tr>
<th>State</th>
<th>Contract Violation</th>
<th>Test Name</th>
</tr>
</thead>
<tbody>
<tr>
<td>Test1</td>
<td></td>
<td>TEST_BANK</td>
</tr>
<tr>
<td>PASSED</td>
<td>NONE</td>
<td>t1: test deposit_on with correct imp and incomplete contract</td>
</tr>
</tbody>
</table>
Version 2:
Incomplete Contracts, Wrong Implementation

```java
class BANK
    deposit_on_v2 (n: STRING; a: INTEGER)
        require across accounts as acc some acc.item.owner ~ n end
        local i: INTEGER
            do
                -- same loop as in version 1

                -- wrong implementation: also deposit in the first account
                accounts[accounts.lower].deposit(a)
            end
        ensure
            num_of_accounts_unchanged:
                accounts.count = old accounts.count
            balance_of_n_increased:
                account_of (n).balance = old account_of (n).balance + a
        end
end
```

Current postconditions lack a check that accounts other than \( n \) are unchanged.
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

<table>
<thead>
<tr>
<th>Case Type</th>
<th>Passed</th>
<th>Total</th>
</tr>
</thead>
<tbody>
<tr>
<td>Violation</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>Boolean</td>
<td>1</td>
<td>2</td>
</tr>
<tr>
<td>All Cases</td>
<td>1</td>
<td>2</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>State</th>
<th>Contract Violation</th>
<th>Test Name</th>
</tr>
</thead>
<tbody>
<tr>
<td>Test1</td>
<td></td>
<td>TEST_BANK</td>
</tr>
<tr>
<td>PASSED</td>
<td>NONE</td>
<td>t1: test deposit_on with correct imp and incomplete contract</td>
</tr>
<tr>
<td>FAILED</td>
<td>Check assertion violated.</td>
<td>t2: test deposit_on with wrong imp but incomplete contract</td>
</tr>
</tbody>
</table>
Version 3:
Complete Contracts with Reference Copy

class BANK
    deposit_on_v3 (n: STRING; a: INTEGER)
        require across accounts as acc some acc.item.owner ~ n end
        local i: INTEGER
        do
            -- same loop as in version 1
            -- wrong implementation: also deposit in the first account
                accounts[accounts.lower].deposit(a)
        end
    ensure
        num_of_accounts_unchanged: accounts.count = old accounts.count
        balance_of_n_increased:
            account_of(n).balance = old account_of(n).balance + a
        others_unchanged:
            across old accounts as cursor
            all cursor.item.owner /~ n implies
                cursor.item ~ account_of (cursor.item.owner)
        end
end
end
31 of 43
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
**APPLICATION**

Note: * indicates a violation test case

<table>
<thead>
<tr>
<th>Case Type</th>
<th>Passed</th>
<th>Total</th>
</tr>
</thead>
<tbody>
<tr>
<td>Violation</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>Boolean</td>
<td>1</td>
<td>3</td>
</tr>
<tr>
<td>All Cases</td>
<td>1</td>
<td>3</td>
</tr>
</tbody>
</table>

- FAILED (2 failed & 1 passed out of 3)

<table>
<thead>
<tr>
<th>State</th>
<th>Contract Violation</th>
<th>Test Name</th>
</tr>
</thead>
<tbody>
<tr>
<td>Test1</td>
<td>TEST_BANK</td>
<td></td>
</tr>
<tr>
<td>PASSED</td>
<td>NONE</td>
<td>t1: test deposit_on with correct imp and incomplete contract</td>
</tr>
<tr>
<td>FAILED</td>
<td>Check assertion violated.</td>
<td>t2: test deposit_on with wrong imp but incomplete contract</td>
</tr>
<tr>
<td>FAILED</td>
<td>Check assertion violated.</td>
<td>t3: test deposit_on with wrong imp, complete contract with reference copy</td>
</tr>
</tbody>
</table>
class BANK
  deposit_on_v4 (n: STRING; a: INTEGER)
    require across accounts as acc some acc.item.owner ~ n end
    local i: INTEGER
    do
      -- same loop as in version 1
      -- wrong implementation: also deposit in the first account
      accounts[accounts.lower].deposit(a)
    end
  ensure
    num_of_accounts_unchanged: accounts.count = old accounts.count
    balance_of_n_increased:
      account_of (n).balance = old account_of (n).balance + a
    others_unchanged:
      across old accounts.twin as cursor
      all cursor.item.owner /~ n implies
      cursor.item ~ account_of (cursor.item.owner)
  end
end
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
end

Test of Version 4: Result

APPLICATION

Note: * indicates a violation test case

<table>
<thead>
<tr>
<th>Case Type</th>
<th>Passed</th>
<th>Total</th>
</tr>
</thead>
<tbody>
<tr>
<td>Violation</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>Boolean</td>
<td>1</td>
<td>4</td>
</tr>
<tr>
<td>All Cases</td>
<td>1</td>
<td>4</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>State</th>
<th>Contract Violation</th>
<th>Test Name</th>
</tr>
</thead>
<tbody>
<tr>
<td>Test1</td>
<td>TEST_BANK</td>
<td></td>
</tr>
<tr>
<td>PASSED</td>
<td>NONE</td>
<td>t1: test deposit_on with correct imp and incomplete contract</td>
</tr>
<tr>
<td>FAILED</td>
<td>Check assertion violated.</td>
<td>t2: test deposit_on with wrong imp but incomplete contract</td>
</tr>
<tr>
<td>FAILED</td>
<td>Check assertion violated.</td>
<td>t3: test deposit_on with wrong imp, complete contract with reference copy</td>
</tr>
<tr>
<td>FAILED</td>
<td>Check assertion violated.</td>
<td>t4: test deposit_on with wrong imp, complete contract with shallow object copy</td>
</tr>
</tbody>
</table>
Version 5:
Complete Contracts with Deep Object Copy

class BANK
  deposit_on_v5 (n: STRING; a: INTEGER)
    require across accounts as acc some acc.item.OWNER ~ n end
    local i: INTEGER
    do
      -- same loop as in version 1
      -- wrong implementation: also deposit in the first account
      accounts[accounts.lower].deposit(a)
    end
  ensure
    num_of_accounts_unchanged: accounts.count = old accounts.count
    balance_of_n_increased:
      account_of (n).balance = old account_of (n).balance + a
    others_unchanged:
      across old accounts.deep.twin as cursor
        all cursor.item.OWNER /~ n implies
          cursor.item ~ account_of (cursor.item.OWNER)
    end
end
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
end
# Application Test of Version 5: Result

Note: * indicates a violation test case

<table>
<thead>
<tr>
<th>Case Type</th>
<th>Passed</th>
<th>Total</th>
</tr>
</thead>
<tbody>
<tr>
<td>Violation</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>Boolean</td>
<td>1</td>
<td>5</td>
</tr>
<tr>
<td>All Cases</td>
<td>1</td>
<td>5</td>
</tr>
</tbody>
</table>

### FAILED (4 failed & 1 passed out of 5)

<table>
<thead>
<tr>
<th>State</th>
<th>Contract Violation</th>
<th>Test Name</th>
</tr>
</thead>
<tbody>
<tr>
<td>Passed</td>
<td>NONE</td>
<td></td>
</tr>
<tr>
<td>FAILED</td>
<td>Check assertion violated</td>
<td>t1: test deposit_on with correct imp and incomplete contract</td>
</tr>
<tr>
<td>FAILED</td>
<td>Check assertion violated</td>
<td>t2: test deposit_on with wrong imp but incomplete contract</td>
</tr>
<tr>
<td>FAILED</td>
<td>Check assertion violated</td>
<td>t3: test deposit_on with wrong imp, complete contract with reference copy</td>
</tr>
<tr>
<td>FAILED</td>
<td>Check assertion violated</td>
<td>t4: test deposit_on with wrong imp, complete contract with shallow object copy</td>
</tr>
<tr>
<td>FAILED</td>
<td>Postcondition violated</td>
<td>t5: test deposit_on with wrong imp, complete contract with deep object copy</td>
</tr>
</tbody>
</table>
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)

Expanded Class: Modelling
Expanded Class: Programming (2)
Expanded Class: Programming (3)
Reference vs. Expanded (1)
Reference vs. Expanded (2)

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)
Index (2)

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?
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
Index (3)

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
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