EECS3311 Fall 2017
Lab Exercise 1
Specifying Contracts for Linear Containers

CHEN-WEI WANG

DUE DATE: 12:00 NOON, Wednesday, September 27

Check the Amendments section of this document regularly for changes, fixes, and clarifications.

1 Policies

- You are required to **work on your own** for this lab. **No** group partners are allowed.
- When you submit your lab, you claim that it is **solely** your work. Therefore, it is considered as **an violation of academic integrity** if you copy or share **any** parts of your Java code during **any** stages of your development.
- When assessing your submission, the instructor and TA will examine your code, and suspicious submissions will be reported to the department if necessary. **We do not tolerate academic dishonesty**, so please obey this policy strictly.
- You are entirely responsible for making your submission in time. Back up your work **periodically**, so as to minimize the damage should any sort of computer failures occur.
- The deadline is **strict** with no excuses: you receive **0** for not making your submission in time.
- You are free to work on this lab anywhere, but you are advised to test your code via your EECS account before the submission.

Contents

2 Learning Outcomes of this Lab Exercise

1. Implement functionalities of a linear container via two library data structures: `ARRAY` and `LINKED_LIST`.
2. Specify contracts (i.e., preconditions, postconditions, and class invariants) for the implemented functionalities.
3. Practice Test-Driven Development (TDD):
   - Write unit tests (using the ESpec library) as soon as a unit of functionality becomes executable.
   - Accumulate a suite of test classes (each of which containing test cases) for regression testing.
   - Use the debugging tool in EStudio IDE when failing tests.
   - Add more test until all the **normal** scenarios (where actual outputs of your programs match the expected outputs) and the abnormal scenarios (where contract violations are expected) are tested in your developed test suite.

3 Background Readings

- Lab Exercise 0 [see the course moodle page]
- Lecture Slides (and relevant parts of the lecture recordings):
  - DbC and TDD
  - Overview of Eiffel Syntax
- Tutorial series on DbC and TDD:

  `https://www.youtube.com/playlist?list=PL5dxAmCmjv_6r5VfzCQ5bTznoDDgh__KS`

  Pay special attention to Part 5 (how to use the debugging tool in EStudio) and Part 8 (the uniform access principle).
- For the `across` syntax, you may refer to:
  - Sample codes of using `ARRAY` and `LINKED_LIST`.
  - This short tutorial article

    Notice that there are three possible uses of the `across` keyword:

```
across
  ... as ...
all
  ...
end
```

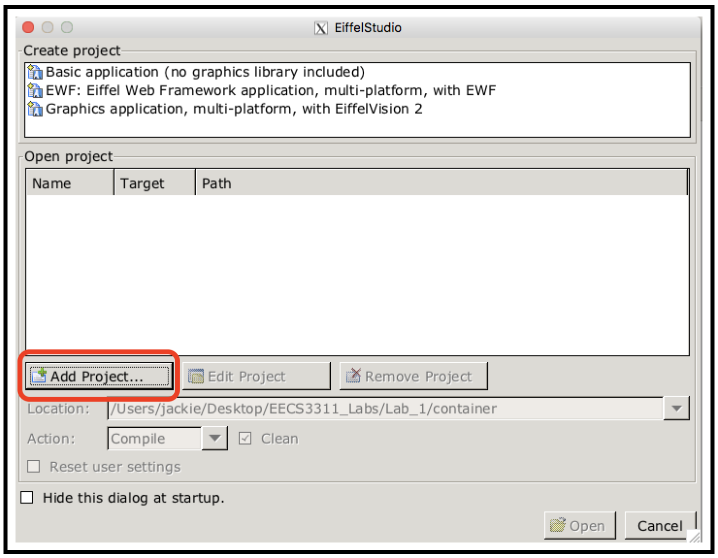
```
across
  ... as ...
some
  ...
end
```

```
across
  ... as ...
loop
  ...
end
```

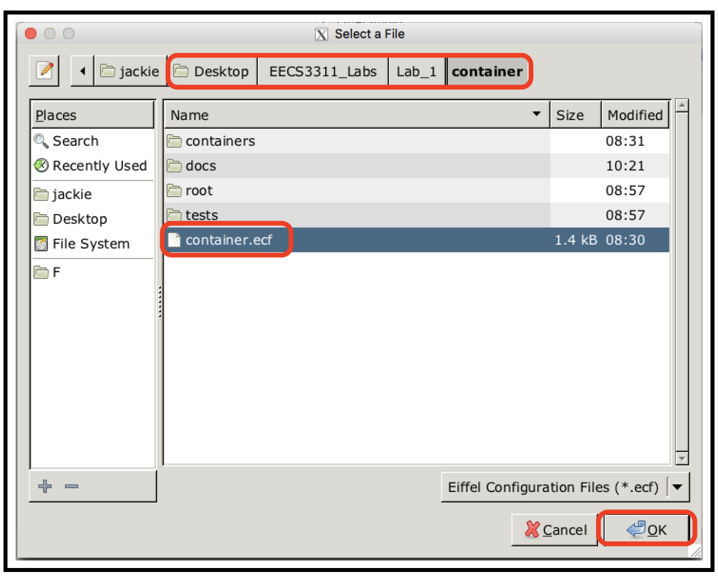
    When the `all` keyword is used, it corresponds to a universal quantification ($\forall$), and the entire expression evaluates to a Boolean value. When the `some` keyword is used, it corresponds to an existential quantification ($\exists$), which also evaluates to a Boolean value. However, it is also possible to use the `loop` keyword, in which case it is no longer an expression, but a loop instruction (just an alternative to writing loops using the `from ... until ... loop ... end` syntax). **Use all or some in the context of contracts; use loop in the context of implementation bodies**.
- You can also find an abundance of resources on DbC, TDD, ESpec tests, and Eiffel code examples from these two sites:
  - `http://eiffel.eecs.yorku.ca`
  - `https://wiki.eecs.yorku.ca/project/eiffel/`

4 Getting Started

- Go to the course moodle page. Under Lab 1, download the file `Lab_1.zip` which contains the starter project for this lab.
- Unzip the file, and you should see a directory named `Lab_1` which has a subdirectory `container`.
- Move `Lab_1` under the directory where you stored `Lab_0` (e.g., under the directory `~/Desktop/EECS3311_Labs`).
- Now it is assumed that the *project location* of the current lab is: `~/Desktop/EECS3311_Labs/Lab_1/container`.
- Launch EStudio from a terminal: `estudio17.05 &`
- Choose `Add Project` (so we can import the starter).



- Browse to the project location `~/Desktop/EECS3311_Labs/Lab_1/container` and choose the configuration file `container.ecf`.



- Click `OK` then `Open`. Wait for the compilation to complete.

5 Background Reading: Copying an Object

Let us assume the following declarations of two variables (which can be either an attribute or a local variable) of the same type (an array of strings):

```
imp : ARRAY[STRING]
old_imp: ARRAY[STRING]
```

Since Eiffel is an object-oriented programming language, when a variable is declared of a type that corresponds to a known class (e.g., `STRING`, `ARRAY`, `LINKED_LIST`, etc.), it means that at runtime, that variable stores the address of some object of the right type (as opposed to storing the object in its entirety). For example, each of the two variables `imp` and `old_imp` above stores the address of an array, and each "slot" of this array stores the address of some string object.

Remark. From now on, when a reference variable `var` stores the address of some object `obj`, we simply say that `var` points to `obj`.

Now consider the following lines of code:

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

Executing the above four lines of code, we have the runtime object structure as shown in Figure 1.

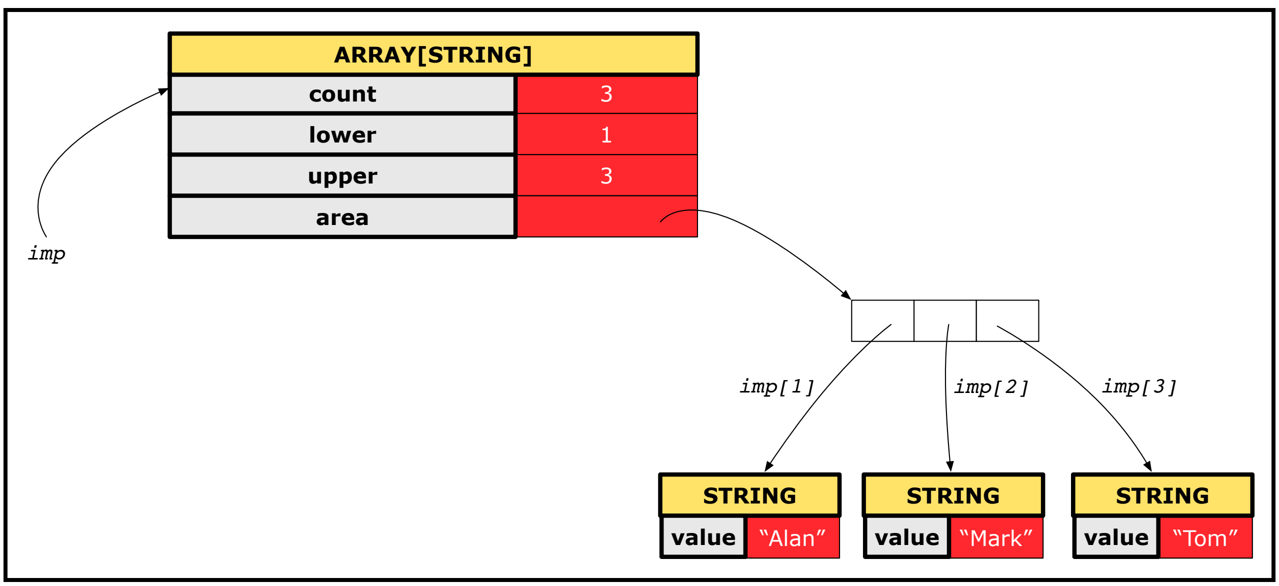


Figure 1: Visualizing an Array of Strings

In Figure 1, variable `imp` points to an array of size (or count) 3, where:

– The first "slot" `imp[1]` points a string object whose contents are "Alan".

– The second "slot" `imp[2]` points a string object whose contents are "Mark".

– The third "slot" `imp[3]` points a string object whose contents are "Tom".

Now we consider two scenarios of copying the structure of the array object pointed by `imp`.

5.1 Scenario 1: Reference Copying

Now consider this line of code:

```
old_imp := imp
```

The above line of code is valid because both variables **imp** and **old_imp** were declared of the same type (i.e., ARRAY[STRING]). Executing the assignment **old_imp := imp** does the so called **reference copying**: we make a copy of the address stored in variable **imp** and assign it to variable **old_imp**. Consequently, as shown in Figure 2 (page 7), both variables **imp** and **old_imp** store the same address, i.e., they point to the same array object. We call such an phenomenon *aliasing*, where more than one variables point to the same object.

What is so interesting about aliasing is that a change that is made via one of the pointing variables (e.g., **imp**) is visible to all other pointing variables (e.g., **old_imp**). As an example, consider the following line of code:

```
imp[2] := "Jim"
```

What the above assignment does is to store the address of a new string object into the second slot of the array object, pointed to by variable **imp**. But since we know both variables **imp** and **old_imp** point to the same array object, after this assignment, both variables still point to exactly the same object structure, as shown in Figure 3 (page 7).

Important Question: Is it possible to copy the object structure pointed to by variable imp, in such a way that, even after one of the array "slots" is re-assigned, the old copy is somehow unaffected? We next consider an alternative to referencing copying which will achieve this.

5.2 Scenario 2: Shallow Copying

Alternatively, now consider this line of code:

```
old_imp := imp.twin
```

The consequence of performing the above assignment is illustrated in Figure 4 (page 8). The use of **.twin** on the right-hand side of the above assignment does what is called **shallow copying**: a brand new, separate array object is created. Even though this copying mechanism is "deeper" than what reference copying does, it is also called **first-level copying**: it only creates a new array object, whereas each "slot" of the array object stored a *reference copy* of the corresponding "slot" in the original array. Once this shallow copy is completed on the right-hand side of the above assignment, it assigns the address of the new array object (which is different from that stored in **imp**) to variable **old_imp**.

Nonetheless, since we only intend to modify certain slot of the array, this shallow copying mechanism suffices for our purpose of this lab. As an example, consider this line of code again:

```
imp[2] := "Jim"
```

This time, due to the shallow copying that was completed, only an array "slot" that belongs to variable **imp** is modified, whereas all "slots" that belong to variable **old_imp** remain untouched. This is illustrated in Figure 5 (page 8).

5.3 Comparing Scenario 1 and Scenario 2 via Tests

Study the two test queries test_array_ref_copy and test_array_shallow_copy in class TEST_ARRAY_COPIES (under the cluster **tests/instructor**) carefully.

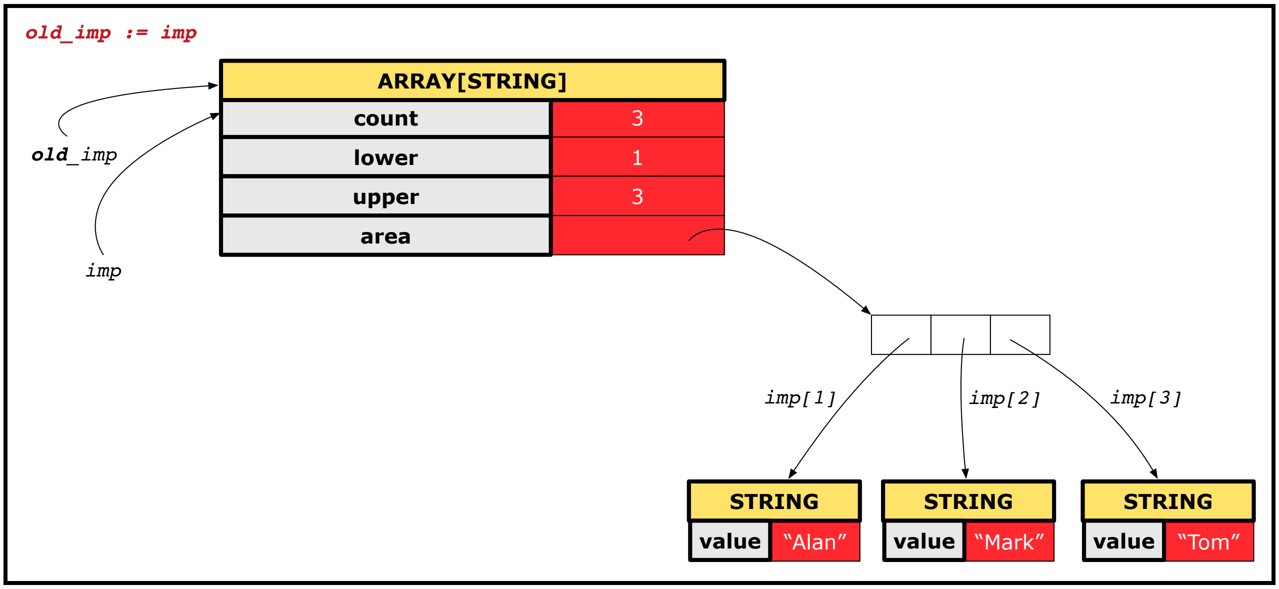


Figure 2: Making a Reference Copy via an Assignment

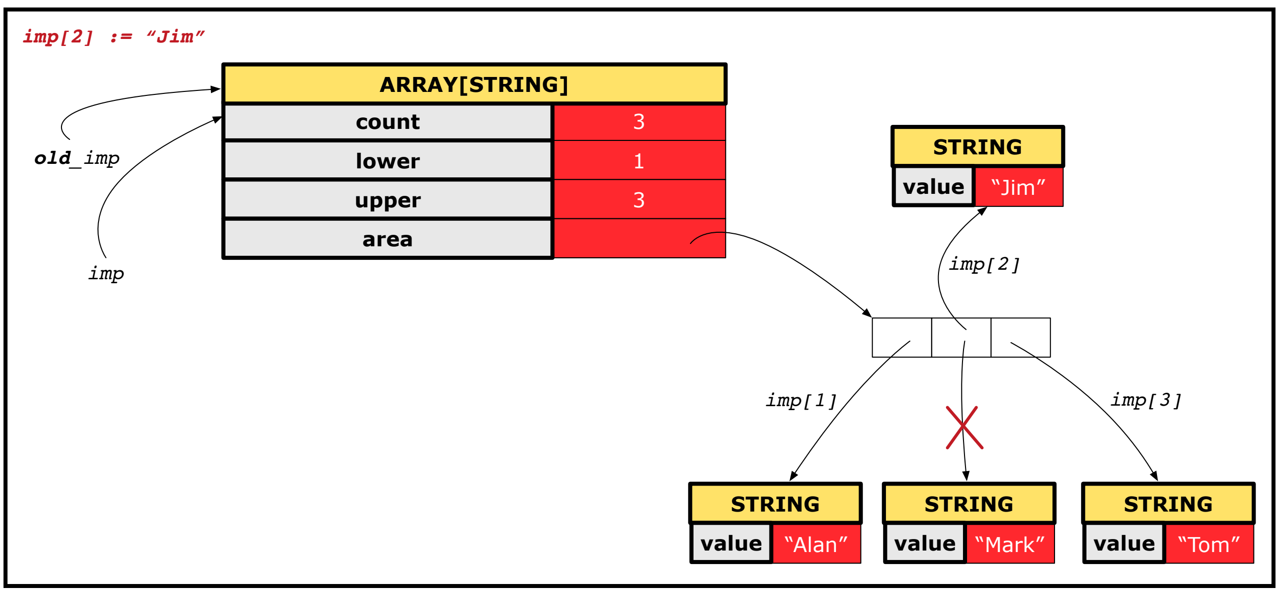


Figure 3: A Change that Affects the Reference Copy

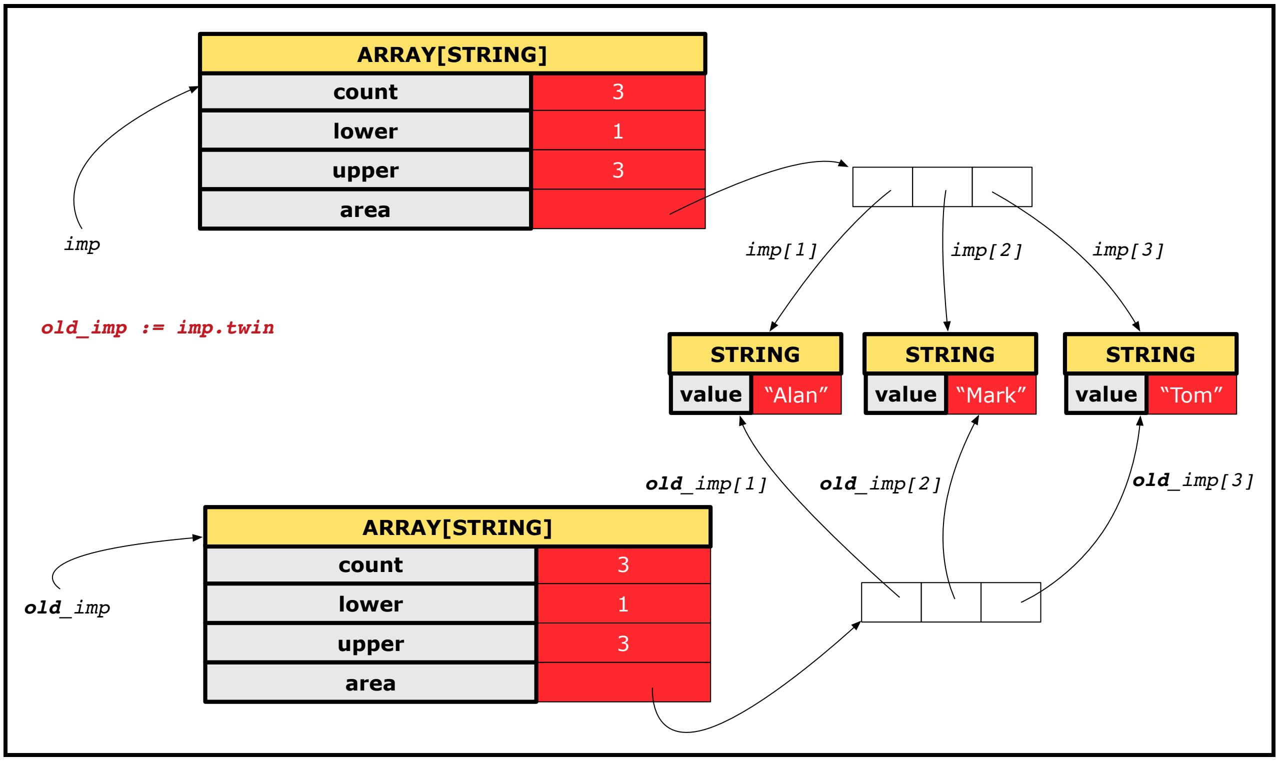


Figure 4: Making a Shallow Copy via a `twin`

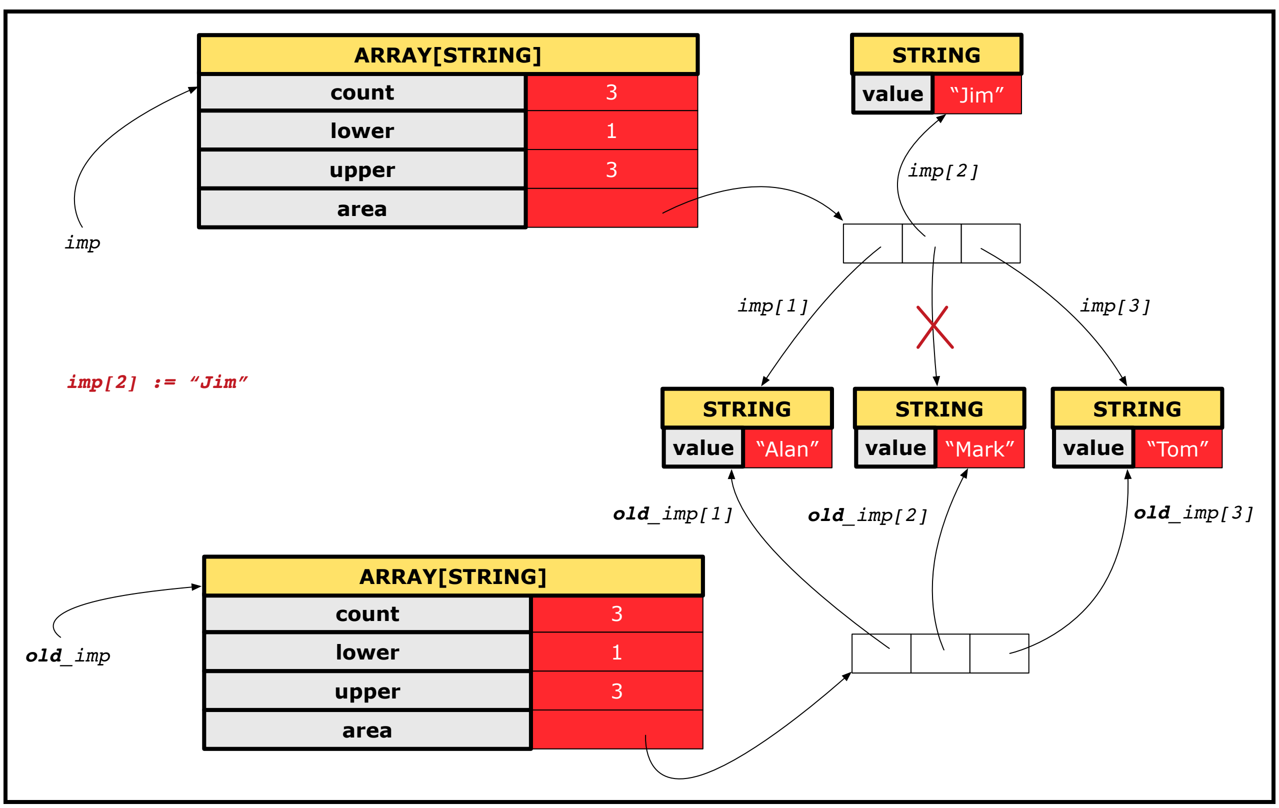


Figure 5: A Change that Does Not Affect the Shallow Copy

6 Problem

A *linear container* is an ordered collection of items with unbounded capacity. Each item or element in a container can be accessed using its absolute position in the container. A position is expressed through an integer index, starting from 1 and ending at *count*, where *count* is the number of items currently stored in the container. For the purpose of this lab, we choose to implement the following features of a linear container using an array:

- `count: INTEGER`

  The number of elements currently stored in the container. It is up to you whether to implement this feature as an attribute (storage) or as a query (computation). Review the uniform access principle in the tutorial series.

- `make`

  Initialize an empty container.

- `valid_index (i: INTEGER): BOOLEAN`

  Determine if an integer `i` represents a valid index of the current container.

  It is also expected that no elements in the container will be changed.

- `get_at (i: INTEGER): STRING`

  Given that `i` is a valid index, return the element stored at index `i`.

  It is also expected that no elements in the container will be changed.

- `assign_at (i: INTEGER; s: STRING)`

  Given that `i` is a valid index, change the value at index `i` to `s`.

  For example, given a container `<alan, mark, tom>`, calling `assign_at (2, jim)` will change it to `<alan, jim, tom>`. The expected benefits of calling this feature are: **1)** the number of elements stays unchanged; **2)** `jim` is now stored at index 2 in the updated container; and **3)** all other indices (i.e., 1 and 3) have their stored elements (i.e., `alan` and `tom`) somehow corresponding to the old container.

- `insert_at (i: INTEGER; s: STRING)`

  Given that `i` is a valid index, insert the value `s` into index `i`, while shifting all elements that used to be at index `i`, index `i + 1`, etc. to the right by one position.

  For example, given a container `<alan, mark, tom>`, calling `insert_at (2, jim)` will change it to `<alan, jim, mark, tom>`. The expected benefits of calling this feature are: **1)** the number of elements is incremented; **2)** `jim` is now stored at index 2 in the updated container; **3)** all new indices to the left of index 2 (i.e., index 1) have their stored elements (i.e., `alan`) somehow corresponding to the old container; and **4)** all new indices to the right of index 2 (i.e., index 3 and 4) have their stored elements (i.e., `mark` and `tom`) somehow corresponding to the old container.

  Hint. To specify **3)** and **4)**, think carefully the relationship between indices of the old and updated containers.

- `delete_at (i: INTEGER)`

  Given that `i` is a valid index, delete the value stored at index `i`, while shifting all elements that used to be at index `i + 1`, index `i + 2`, etc. to the left by one position.

For example, given a container <alan, mark, tom>, calling `delete_at (2)` will change it to <alan, tom>. The expected benefits of calling this feature are: **1)** the number of elements is decremented; **2)** **3)** all old indices to the left of index 2 (i.e., index 1) have their stored elements (i.e., `alan`) somehow corresponding to the new container; and **4)** all old indices to the right of index 2 (i.e., index 3) have their stored elements (i.e., `tom`) somehow corresponding to the new container.

Hint. To specify **3)** and **4)**, think carefully the relationship between indices of the old and updated containers.

– `insert_last (s: STRING)`

Insert value `s` to the last position of the container.

For example, given a container <alan, mark, tom>, calling `insert_last (jim)` will change it to <alan, mark, tom, jim>. The expected benefits of calling this feature are: **1)** the number of elements is incremented; **2)** the element stored at the last index now (i.e., 4) is `jim`; and **3)** all indices to the left of the new last index (i.e., indices 1, 2, and 3) have their stored elements (i.e., `alan`, `mark`, and `tom`) somehow corresponding to the old container.

– `remove_first`

Given that the current container is not empty, remove the element at the first position.

For example, given a container <alan, mark, tom>, calling `remove_first` will change it to <mark, tom>. The expected benefits of calling this feature are: **1)** the number of elements is decremented; and **2)** all indices to the right of the old first index (i.e., indices 2 and 3) have their stored elements (i.e., `mark` and `tom`) somehow corresponding to the new container.

Let's understand what has already been completed for you in the class `ARRAYED_CONTTAINER`:

```
1  class
2    ARRAYED_CONTAINER
3  ...
4  feature {NONE} -- Implementation of container via an array
5    imp : ARRAY[STRING]
6  ...
7  feature -- Commands
8    assign_at (i: INTEGER; s: STRING)
9        -- Change the value at position 'i' to 's'.
10     require
11       valid_index: -- Your task
12     do
13       imp [i] := s
14     ensure
15       size_unchanged: imp.count = (old imp.twin).count
16       item_assigned: imp [i] ~ s
17       others_unchanged:
18         across
19           1 |..| imp.count as j
20         all
21           j.item /= i implies imp [j.item] ~ (old imp.twin) [j.item]
22         end
23     end -- end command assign_at
24 ...
25 feature -- Queries
26   count: INTEGER
27       -- Number of items currently stored in the container.
28       -- It is up to you to either implement 'count' as an attribute,
29       -- or to implement 'count' as a query (uniform access principle).
30   valid_index (i: INTEGER): BOOLEAN
31       -- Is 'i' a valid index of the current linear container?
32       -- Indices range with 1 and the size of container.
33     do
34       Result := true
35     end
36 ...
37 end -- end class ARRAYED_CONTAINER
```

– **L4** uses the `feature` clause to declare a section, under which all features (in this case attribute `imp`) are exported to `NONE` (specified as {`NONE`}). What is written within { and } is a list of comma-separated class names, which specifies the **export status** of those features (attributes, commands, or queries) declared under this section. Think of the export status as a way of restricting these features of being accessible only by the list of classes being their *clients*. For example, in the case of **L4**, the class name `NONE` is a special one, which specifies that no other client classes may access the attribute `imp` (as it is supposed to be the **secrete** of supplier that the clients do not care), and this has the same effect of declaring an attribute as `private` in Java. Symmetrically, if you intend to make a section of features *public*, declare `feature` {`ANY`}, or simply leave out the export status (i.e., `feature`), as the default export status is that all other client classes may access the features.

Remark. This mechanism of specifying the export status of a section of feature, which can be any chosen set of classes, is more powerful than what you can do in Java (where the visibility of an attribute or a method can only be set to the class level, the package level, or the project level).

– **L5** declares an attribute `imp` of type `ARRAY`. Look up the `ARRAY` class in EStudio, and you will see that it is declared as `class ARRAY[G]`, where `G` is called a *generic parameter*, and the type of array elements is denoted using `G`, and `G` may be instantiated to any of the known classes in the context of a client's code (e.g., in `ARRAYED_CONTAINER`, we instantiate `G` as `STRING`). We will discuss generic parameters later in class. For the purpose of this lab, you will just need to know how to use a class with generic parameters (e.g., all collection data structure such as `ARRAY`, `LINKED_LIST`).

– **L8 to L23** defines a command `assign_at`, which intends to assign a string value `s` to index `i` of the arrayed-container, given that `i` is a valid index.

- **L11** specifies a tagged precondition, which currently always evaluates to `true` (Why?). You will be asked to change it.
- **L13** implements the command by a simple array assignment. Also look up and compare these two features for changing contents of an array: `put` and `force`.
- **L15 to L22** defines three tagged postconditions:
  - ⋄ **L16**: The "item_assigned" postcondition asserts that the value stored at index `i` in the new `imp`, after the implementation body (i.e., **L13**) is executed, is equal to `s`. In Eiffel, use the tilde symbol ∼ for object equality (e.g., comparing contents of string objects) and use the equal symbol = for reference equality (e.g., comparing addresses of string objects).
  - ⋄ **L17 to L22**: The "others_unchanged" postcondition asserts that values stored at all other positions of `imp` remain unaffected by the implementation body (i.e., **L13**). Convince yourself that the `across` expression here corresponds to the following mathematical predicate:

$$\forall j : \mathit{INTEGER} \mid 1 \leq j \leq \mathit{imp.count} \bullet \; (\; j \neq i \implies \mathit{imp}[j] \sim (\textbf{old} \; \mathit{imp.twin})[j] \;)$$

In order to check the above predicate in Eiffel, we need to take a snapshot of `imp` before the change at **L13** occurs. The Eiffel compiler identifies every expression that involves the **old** keyword and does a simple assignment to cache its value. For example, recall that in the command `withdraw` in the `ACCOUNT` class, the postcondition reads: `balance = (old balance) - amount`. Consequently, an **implicit** assignment `old_balance := balance` is performed in the very beginning of the command `withdraw`, so that when the reduction is performed, we can compare `balance` (already deducted) against `old_balance`.

Similarly, in the above `across` syntax, the expression that involves the **old** keyword is: `old imp.twin`. Consequently, an **implicit** assignment `old_imp := imp.twin` is performed in the very beginning of the command `assign_at`, so that when the change is made, we can compare `imp` (already changed at "slot" `i`) against `old_imp` (whose "slot" `i` is still unchanged).

As we learned from Section 5, the assignment `old_imp := imp.twin` does a shallow copying, which is deeper than a reference copying done by the assignment `old_imp := imp`. To write a postcondition that accurately specifies that "other slots are unchanged", a reference copying is not sufficient, and a shallow copying is necessary.

Important Exercises:

1. *Temporarily* make the implementation of `assign_at` incorrect:

```
imp [i] := s
if i > 1 then
  imp [1] := s
end
```

2. Quickly implement the query `get_at` and `insert_last` (without writing worrying about their contracts **just yet**; you will do that later). However, **you may need to write additional tests to make sure that these two features are properly implemented before using them in the next step.**
3. Then, write the ESpec test case below:

```
test_assign_at: BOOLEAN
  local
    ac : ARRAYED_CONTAINER
  do
    comment ("Test assign at position")
    create ac.make
    ac.insert_last ("alan")
    ac.insert_last ("mark")
    ac.insert_last ("tom")
    Result :=
          ac.get_at (1) ~ "alan"
      and ac.get_at (2) ~ "mark"
      and ac.get_at (3) ~ "tom"
    check Result end

    ac.assign_at (2, "jim")
    Result :=
          ac.get_at (1) ~ "alan"
      and ac.get_at (2) ~ "jim"
      and ac.get_at (3) ~ "tom"
end
```

4. Executing the above test, you should be reported a postcondition violation with tag "others_unchanged". This is **good**, because when there is something wrong (i.e., the implementation of `assign_at`), there is a postcondition accurately reporting about it.
5. Now let's illustrate that writing merely **(old imp)** instead of **(old imp.twin)** in the above `across` syntax is not sufficient to catch errors. Go to the tagged postcondition tagged "others_unchanged" in `assign_at`, and replace every occurrence of **(old imp.twin)** with **(old imp)**:

```
across
  1 |..| imp.count as j
all
  j.item /= i implies imp [j.item] ~ (old imp) [j.item]
end
```

   Then, rerun the above test, then you should see that there is no more postcondition violation (but the final value of `Result` is false, because the implementation was wrong). This is **bad**, because when there is something wrong (i.e., the implementation of `assign_at`), there is no contract violation reporting about it!

   Remark: For the purpose of this lab, always use (old imp.twin) to refer to the before-state value of imp.

◇ **L15**: The "size_unchanged" postcondition asserts that the size (or count) of the old `imp`, before the implementation body (i.e., **L13**) is executed, is equal to the size of the new `imp`, after **L13** is executed.

7 You Tasks

7.1 Completing the ARRAYED_CONTAINER Class

– Fill in the implementations and contracts of all listed feature in `ARRAYED_CONTAINER`.

- You must **not** change any of the feature names, parameters, or contract tags.
- The body of implementation of each command or query must be defined in terms of the private attribute `imp`. You should not need any additional attributes for `ARRAYED_CONTAINER`, but you you may declare local variables if you find it necessary.
- Create a new class `TEST_ARRAYED_CONTAINER` under the cluster `tests/student`. Then you can add a line in the `make` feature of `TEST_CONTAINERS`:

```
add_test (create {TEST_ARRAYED_CONTAINER}.make)
```

  ◇ You must add *at least 10* test features in `TEST_ARRAYED_CONTAINER`, and all of the must pass. (In fact, you should write as many as you think is necessary.)

  ◇ You will not be assessed by the quality or completeness of your tests (i.e., we will only check that you have at least 10 tests and all of them pass). However, write tests for yourself so that your software (implementation and contracts) will pass all tests that we run to assess your code. There are two categories of tests that you should write and run: **1)** test queries which test the normal scenarios (where no contract violations are expected); and **2)** test commands which test the abnormal scenarios (where some **tagged** contract violations are expected). Use `add_boolean_case` for category **1)** and use `add_violation_case_with_tag` for category **2)**.

  ◇ For each test query or test command, always start with a call to `comment(...)`.

7.2 Drawing a Design Diagram

For the purpose of this lab, you are required to draw a digram that summarizes your design: **a view from the clients** that see only contracts of features, whereas all implementation details are hidden. Your diagram must:

– Show contracts of all features and the class.

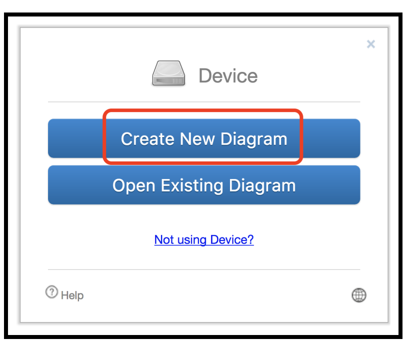
– Avoid using the **across** syntax; instead, use their logical counterparts ∀ or **exists**. That is, when there is a logical correspondence for the Eiffel operator you use (e.g., ∧ for **and**), always use the logical operator in your diagram for neatness and preciseness.

You must use the program and library template as instructed below:

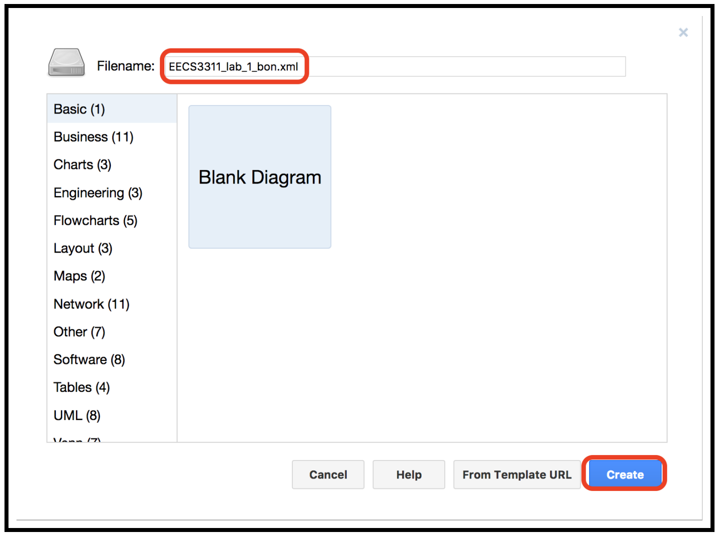
– Download a library template `EECS3311 BON Library.xml` from the course moodle page and save it to the desktop.

– Launch your web browser and go to draw.io.

– Choose `Create New Diagram`.

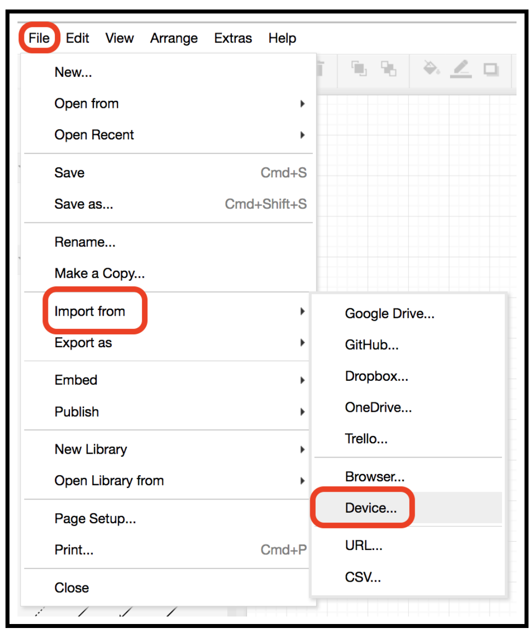


- Enter `EECS3311_lab_1_bon.xml` in the `Filename` text box, then click on `Create` to create a blank diagram.

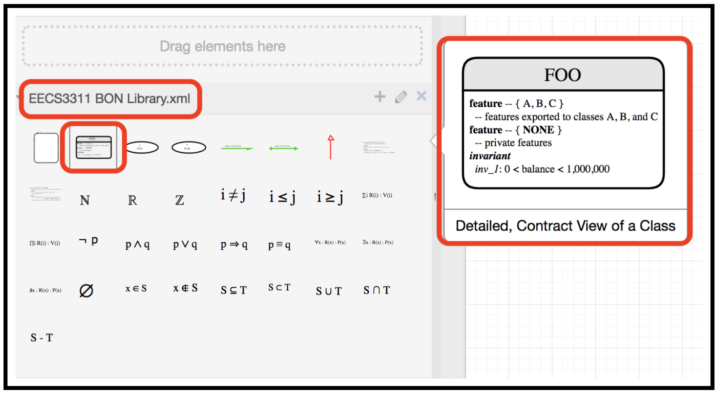


Later on, every time you make a change and need to save, you will need to browse to the same location to overwrite the existing, old version.

- Now we import the library template that you just downloaded to the desktop: `File`, then `Import from`, then `Device`, then browse to the `xml` library file on the desktop and `open`.



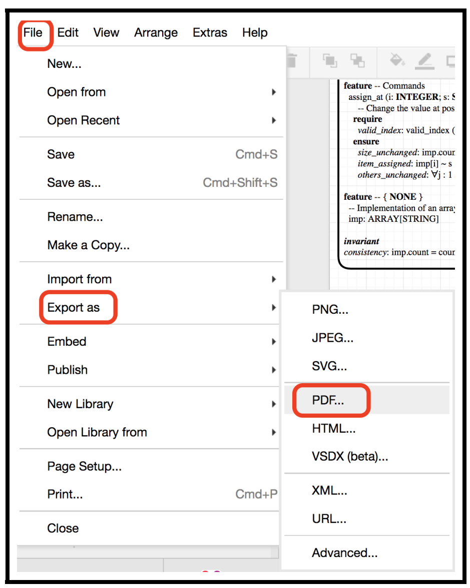
- Now you should see on the left panel a section called `EECS3311 BON Library.xml`.



Browse through the list of items that you may use. Hovering your mouse over an item in the library will pop up a description of what it represents. For this lab, you will need: **1)** the detailed, contract view of a class; and **2)** relevant math symbols.

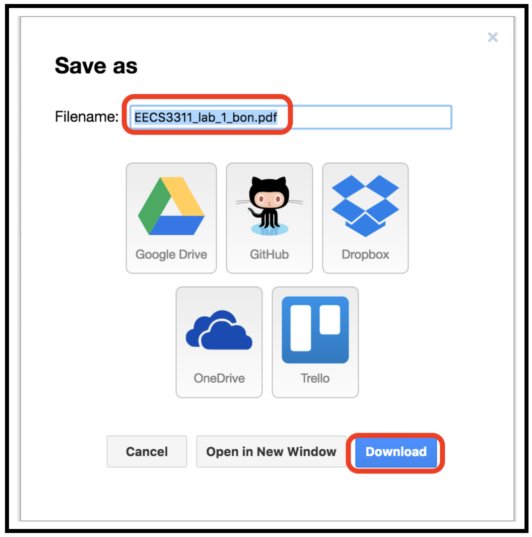
Tips: When writing a mathematical formula in your diagram, you may find it the easiest to: **1)** click on the symbol you need (then a separate text box will pop up); and **2)** cut and paste the symbol to where you need in your formula.

- Once you are happy with your diagram source file (which is an `xml` file), make sure you save it. Then, go to `File`, then `Export as`, and then `PDF`.



- Then tick the `Crop` box then `Export`.

– Enter `EECS3311_lab_1_bon.pdf` in the `Filename` text box, then click on `Download` to your desktop.



– Here is an example of showing a (partial) contract view (Take this as a standard):

ARRAYED_CONTAINER

feature -- Commands
assign_at (i: **INTEGER**; s: **STRING**)
-- Change the value at position 'i' to 's'.
require
valid_index: valid_index (i)
ensure
size_unchanged: imp.count = (**old** imp.twin).count
item_assigned: imp[i] ~ s
others_unchanged: ∀j : 1 ≤ j ≤ imp.count : j ≠ i ⇒imp[j] ~ (**old** imp.twin) [j]

feature -- { **NONE** }
-- Implementation of an arrayed-container
imp: ARRAY[STRING]

invariant
consistency: imp.count = count

Notice that:

- A tag, if any, should be included for the corresponding contract.
- For quantifications, we simply use two colons (`:`) to separate parts, rather than | and • as in math. This makes it easier for you to draw.

– Now move both

- The diagram source file `EECS3311_lab_1_bon.xml`
- Its exported PDF file `EECS3311_lab_1_bon.pdf`

to the `docs` directory of your lab project.

8 Submission

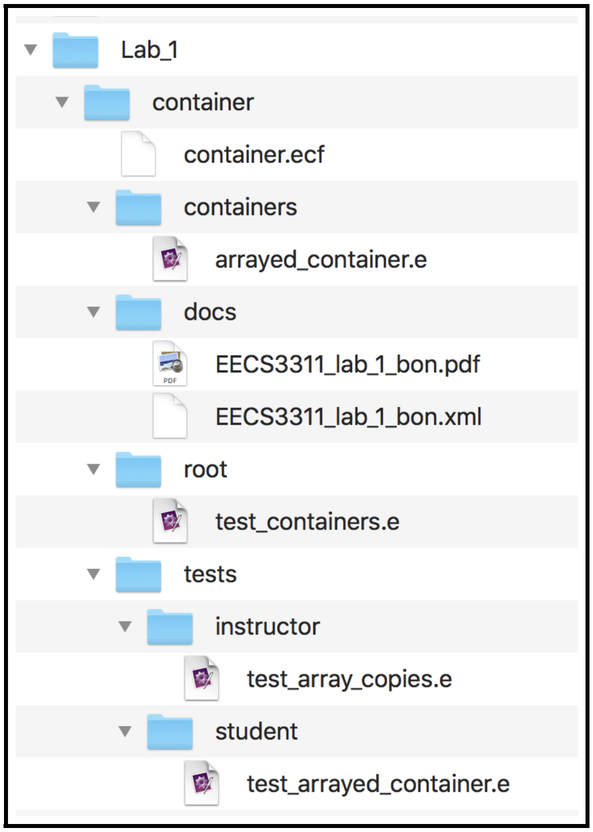
To get ready to submit:

– Close EStudio

– Type the following command (only available via your lab account) to clean up the `EIFGENs` directory:

```
cd ~/Desktop/EECS3311_Labs/Lab_1
eclean container
```

– Make sure the directory structure of your project is identical to this (with **no** `EIFGENs`):



By the due date, submit via the following command:

```
cd ~/Desktop/EECS3311_Labs/Lab_1
submit 3311 lab1 container
```

After you submit, there will be some automated program attempting to perform some basic checks on your program: if your submitted directory has the expected structure, compile your Eiffel project, and run your tests. Please be patient and wait until it finishes.

9 Amendments

List of changes, fixes, or clarifications will be added here.

–