EECS3311 Fall 2019
Lab Exercise 1
Implementing and Specifying a Task Scheduler

Chen-Wei Wang

Common Due Date for Sections A & E: **2pm, Friday, September 20**

- **Check the Amendments section of this document regularly for changes, fixes, and clarifications.**
- **Ask questions on the course forum on the moodle site for Sections A and E.**

1 Policies

- **Your (submitted or un-submitted) solution to this lab exercise (which is not revealed to the public) remains the property of the EECS department. Do not distribute or share your code in any public media (e.g., a non-private Github repository) in any way, shape, or form. The department reserves the right to take necessary actions upon found violations of this policy.**
  - 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 may examine your code, and suspicious submissions will be reported to the department/faculty if necessary. **We do not tolerate academic dishonesty**, so please obey this policy strictly.
- You are entirely responsible for making your submission in time.
  - You may submit **multiple times** prior to the deadline: only the last submission before the deadline will be graded.
  - Practice submitting your project early **even before it is in its final form**.
  - No excuses will be accepted for failing to submit shortly before the deadline.
  - Back up your work **periodically**, so as to minimize the damage should any sort of computer failures occur. Follow this tutorial series on setting up a **private** Github repository for your Eiffel projects.
  - The deadline is **strict** with no excuses: you receive **0** for not making your electronic submission in time.
  - Emailing your solutions to the instruction or TAs will not be acceptable.
- You are free to work on this lab on your own machine(s), but you are responsible for testing your code at a Prims lab machine before the submission.

Contents

2 Learning Outcomes of this Lab Exercise

1. Implement functionalities of an array-based heap via the `ARRAY` library class, and functionalities of a task scheduler via the `HASH_TABLE` class.
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.
   - Draw a design diagram (showing the Contract Views of `ARRAYED_HEAP` and `SCHEDULER`) using the BON notation.

3 Background Readings

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

  `https://www.youtube.com/playlist?list=PL5dxAmCmjv_6r5VfzCQ5bTznoDDgh__KS`
- 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 cursor_var
all
  ... cursor_var.item ...
end
```

```
across
  ... as cursor_var
some
  ... cursor_var.item ...
end
```

```
across
  ... as cursor_var
loop
  ... cursor_var.item ...
end
```

    Each of the above three `across ... as ...` constructs creates a local cursor variable, indirectly pointing to a member of the collection being iterated through.
  - Alternatively, you may modify the `as` keyword to the `is` keyword:

```
across
  ... is value_var
all
  ... value_var ...
end
```

```
across
  ... is value_var
some
  ... value_var ...
end
```

```
across
  ... is value_var
loop
  ... value_var ...
end
```

    Each of the above three `across ... is ...` constructs creates a local variable, directly storing a member of the collection being iterated through.

Notice the subtle difference between `across` ... `as` ... and `across` ... `is` ... as explained above.

- When the `all` keyword is used, it corresponds to a universal quantification (∀), and the entire expression evaluates to a Boolean value. When the `some` keyword is used, it corresponds to an existential quantification (∃), 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 Problem

We consider the *implementation*, and more importantly the *specification*, of an efficient task scheduler (which is a crucial component for resource management in, e.g., an operating system). Tasks added to the scheduler have *unique* integer priority values. A scheduler may be considered as a *priority queue*: the task with the largest priority value is selected next for execution.

A common strategy for implementing an efficient selection procedure is by using the *heap* data structure (which can be visualized as a balanced binary tree). Specifically, we consider a maximum heap, where the root of every subtree stores the maximum key of that subtree. Given n keys in a heap, retrieving the maximum takes $O(1)$ time, whereas building a heap from scratch (out of an unsorted array), inserting a new key, and removing an existing key all take $O(log\ n)$ time. We will use an array to represent the binary-tree structure of a heap.

In this section we briefly summarize the important characteristics of a heap. You may consider reading about the more thorough discussion on **heaps** (pages 151 to 158) and **priority queues** (pages 162 to 165) in [1] (See the **References** section). When reading the above recommended pages, you need not worry about the runtime analysis; focus on how the data structure works.

4.1 Using an Array to Represent a Balanced Binary Tree

Given an array `a` of size N, the rule of representation is defined as:

- `a[1]` stores the root of tree.
- Given integer i being the index of an *internal* node (i.e., one with one or two child nodes):
  - `a[i × 2]` denotes its left child node.
  - `a[i × 2 + 1]` denotes its right child node.

    Question. Conversely, given integer i being the index of an internal or external (i.e., leaf) non-root node, how do you calculate the index of its parent node?

  Remark. Convince yourself that:

  - Indices of the array correspond to filling up the tree from top to bottom, left to right. For example, `a[1]` denotes the root, then `a[2]` goes down one level and `a[3]` goes to the right and stays at the same level, and so on.
  - At the bottom level of the tree, external nodes are filled in from left to right.
  - `a[1]`, `a[2]`, ..., `a[`$\lfloor \frac{N}{2} \rfloor$`]` denote the set of *internal* nodes. This is an important insight which would help you implement some of the heap operations.

The above rule of representation guarantees that the binary tree is *balanced*: its height (i.e., the longest path from the root to a leaf node) is $O(log\ N)$.

Here are a few syntax reminders about Eiffel:

- Eiffel arrays by default start their indices at 1. This actually corresponds to the discussion in [1].
- To calculate $\lfloor \frac{N}{2} \rfloor$ (floor of `N`), you may consider using either `//` (for integer quotient) or `\\` (for integer remainder). The division operator `/` calculates the result with precision (with a fractional point).

4.2 Heap: a Balanced Binary Tree Satisfying the Heap Property

A balanced binary tree T satisfies the (maximum) heap property if:

> For every subtree t in T, t's root is larger than both t's left child and right child nodes[1].

A heap is **not** a balanced binary search tree. An in-order traversal of a balanced BST results in a sorted sequence. On the other hand, an in-order traversal of a heap does not result in a sorted sequence (because in this case t's right child is larger than t's root, violating the heap property).

[1] The order of t's left and right child nodes does not matter.

4.3 Operations Preserving the Heap Properties

- Inquiring the maximum is a trivial constant-time operation.
- Inserting a new key involves:
  1. Insert the new key as the right-most external node at the bottom of tree.
  2. Perform an upward operation from this new external node (possibly all the way to the root of tree) to restore the heap property if necessary: swap the node n with its parent p, if it is not the case that $p > n$.
- Removing the maximum involves:
  1. Move the right-most external node (at the bottom of the tree) to replace the root (which is the maximum).
  2. Perform a downward operation from the new root (possibly all the way to the bottom of tree) to restore the heap property if necessary: swap the node n with the larger of its left child node l and right child node r, if it is not the case that $n > l \wedge n > r$.
- **Remark.** Given that a heap is always a balanced binary tree, each "restore" operation, either upwards or downwards, should the time corresponding to the height of tree: $O(log\ N)$.

4.4 Building a Heap out of an Unsorted Array

To build a heap from an arbitrary array, representing a balanced binary tree which may not satisfy the heap property, the idea is to restore the heap proper for *all* nodes, in the reverse way as how we fill up the tree: from right to left, bottom to top. More precisely. start with the right-most (external) node at the bottom level, then the second-right-most (external) node at the bottom level, ..., then the left-most (external) node at the bottom level, then the right-most node at the second-last level, ..., then the left-most node at the second-last level, ..., then the root. For each node n that we visit in this reverse manner, we restore the heap property of the subtree rooted at n (by using the restore operation same as that for removing the maximum of heap).

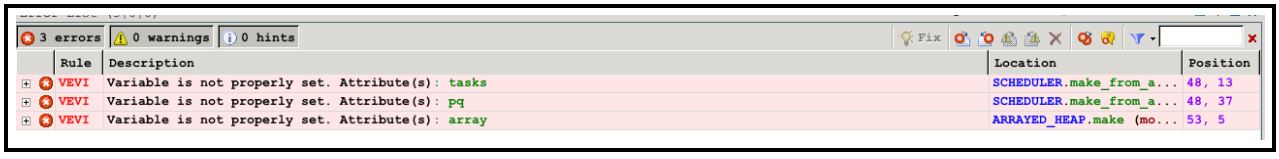
4.5 A Tutorial Video to Help You Understand Heap

The following tutorial video demonstrates the above discussion in Sections 4.1 to 4.4:

`https://www.youtube.com/watch?v=bfRNzNu3jSM&list=PL5dxAmCmjv_7MQ6OPDUhnxSmcCAExJyNa&index=2`

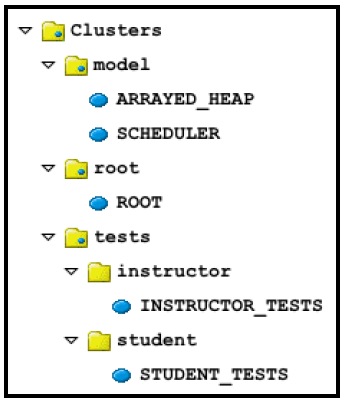
5 Getting Started

- Go to the course moodle page for Sections A and E. Under Lab 1, download the file `scheduler.zip` which contains the starter project for this lab.
- Unzip the file, and you should see a directory named `scheduler`.
- Move the `scheduler` project into where you stored the Lab0 project (e.g., `~/Desktop/eecs3311_workspace`).
- Now it is assumed that the *project location* of the current lab is: `~/Desktop/eecs3311_workspace`.
- Add the `scheduler` project to EStudio and try to compile it.
- The compilation is expected to fail with the following list of errors:



These initial errors are due to the notion of *void safety*, which we will cover soon in the lectures. Briefly speaking, what used to occur in Java at runtime as instances of `NullPointerException` have now become in Eiffel at compile time as *compilation errors*. This kind of void-safety-related compilation errors occur when, e.g., a reference-typed attribute (e.g., `tasks` and `pq` in `SCHEDULER`, `array` in `ARRAYED_HEAP`), a reference-typed return value (e.g., `Result`) is uninitialized.

– Finally, here is the list of classes for you to start:



Notice that there is also an empty `docs` folder not being shown in EStudio. You will need to place the source and PDF of your design diagram there.

6 You Tasks

6.1 Complete the ARRAYED_HEAP and SCHEDULER Classes

– You are expected to write valid implementations and contracts in the ARRAYED_HEAP and SCHEDULER classes. Each instance of "TODO:" in these two classes indicates which implementation or contract you have to complete.

- Read the in-line comments carefully in both ARRAYED_HEAP and SCHEDULER carefully.
- Study the INSTRUCOTR_TESTS class carefully: it documents how ARRAYED_HEAP and SCHEDULER are expected to work together.
- You must **not** change any of the feature names, parameters, or contract tags.
- You must **not** add any additional preconditions, postconditions, or class invariants.
- In the STUDENT_TESTS class, you are required to add as many tests as you judge necessary to test the correctness of ARRAYED_HEAP and SCHEDULER.
  - ◇ You must add ***at least 10*** test features in STUDENT_TESTS, 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.
  - ◇ For each test feature, always start with a call to `comment(...)`.

6.2 Draw 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:

– You are only allowed **one page** for your design diagram.

– Show contracts of **a selection of** critical features of the ARRAYED_HEAP and SCHEDULER classes.

– Any proper relations between classes (e.g., client-supplier, inheritance) must also be shown on your diagram.

– In your diagram, also show the two library classes ARRAY and HASH_TABLE (in the concise forms, i.e., ovals with the class names only). Indicate how these classes are used by ARRAYED_HEAP and SCHEDULER (i.e., client-supplier? inheritance?).

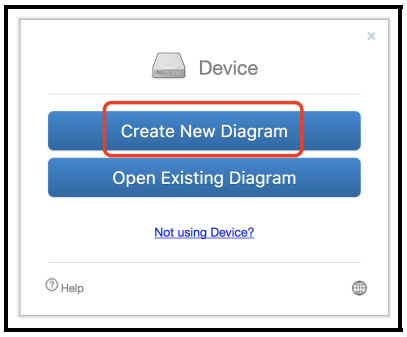
– **Do not** use the across syntax; instead, use their logical counterparts $\forall$ or $\exists$. That is, when there is a logical correspondence for the Eiffel operator you use (e.g., $\wedge$ for and), always use the logical operator in your diagram for neatness and precision.

To prepare your design diagram, you must use the program and library template as instructed below:

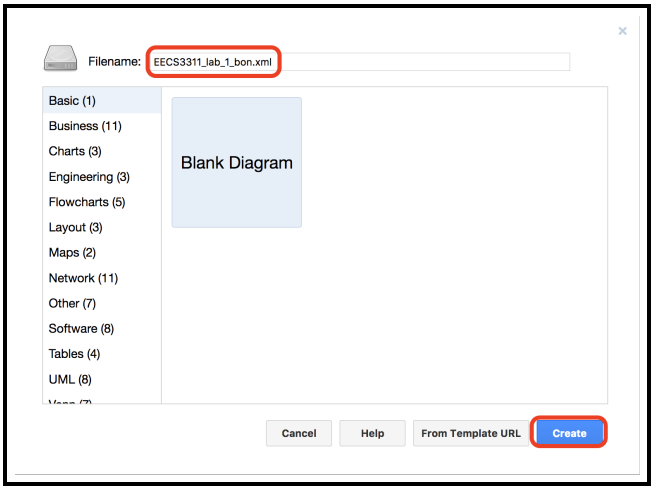
– Download a library template EECS3311 BON Library.xml from the course moodle page and save it to the desktop.

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

– Choose Create New Diagram.

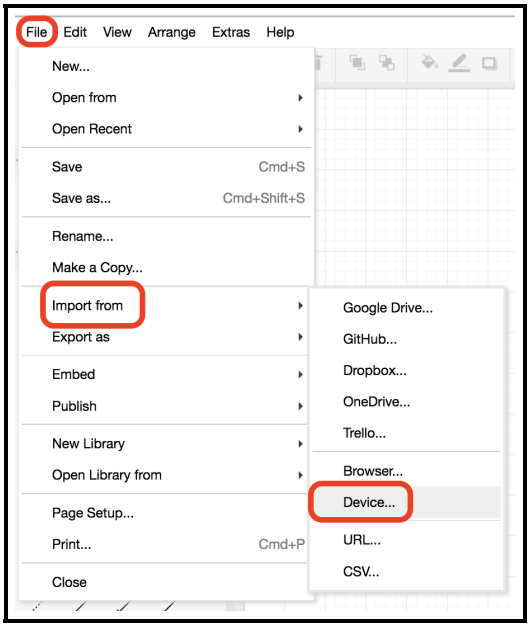


– Enter `EECS3311_lab1_bon.xml` in the `Filename` text box, then click on `Create` to create a blank diagram.

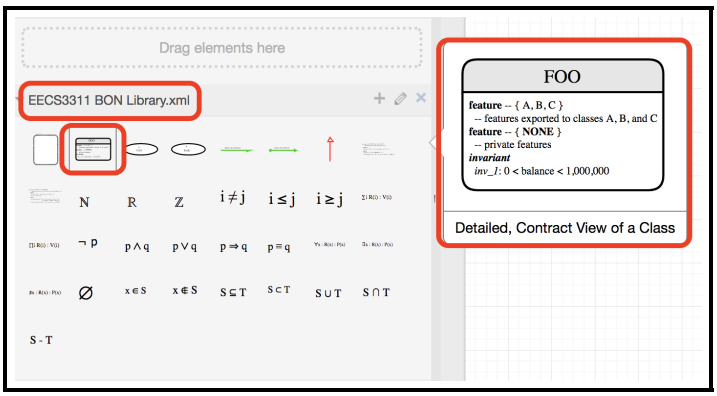


Later on, every time you make a change and need to save, you will need to browse to the same location to overwrite the existing, old version.

– 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`.



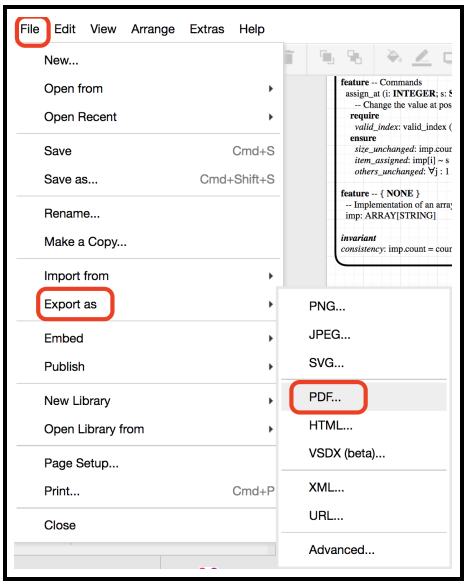
– Now you should see on the left panel a section called `EECS3311 BON Library.xml`.



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.

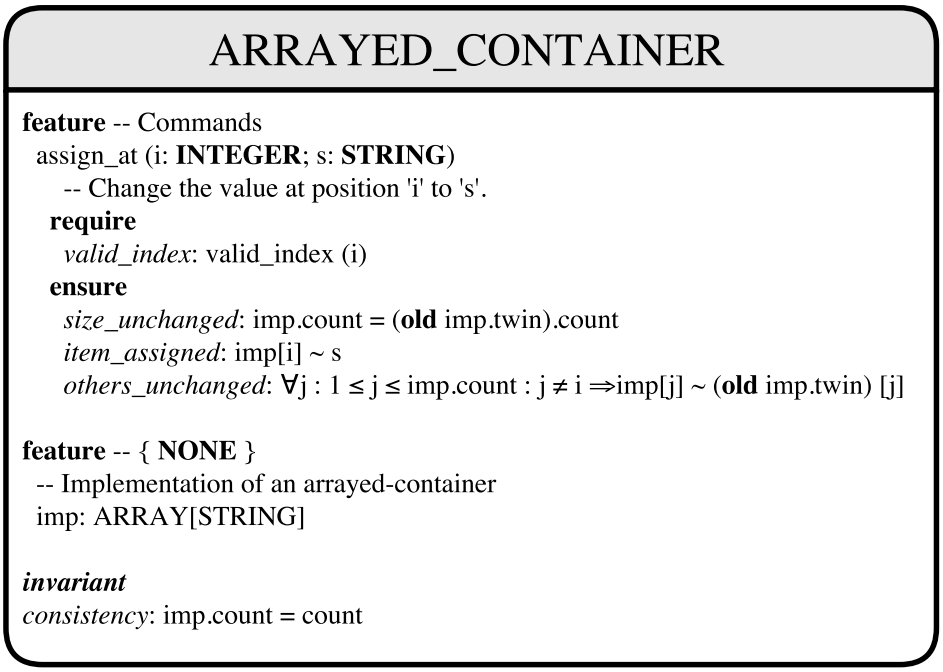
– Once you are happy with your diagram source file (which is an `xml` file), make sure you save it. Then, go to `File`, then `Export as`, and then `PDF`.



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

– Enter `EECS3311_lab1_bon.pdf` in the `Filename` text box, then click on `Download` to your desktop.

– Here is an example of showing a (partial) contract view of an array-based container (your submitted diagram will be graded using this as the standard):



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_lab1_bon.xml`
- Its exported PDF file `EECS3311_lab1_bon.pdf`

to the `docs` directory of your lab project.

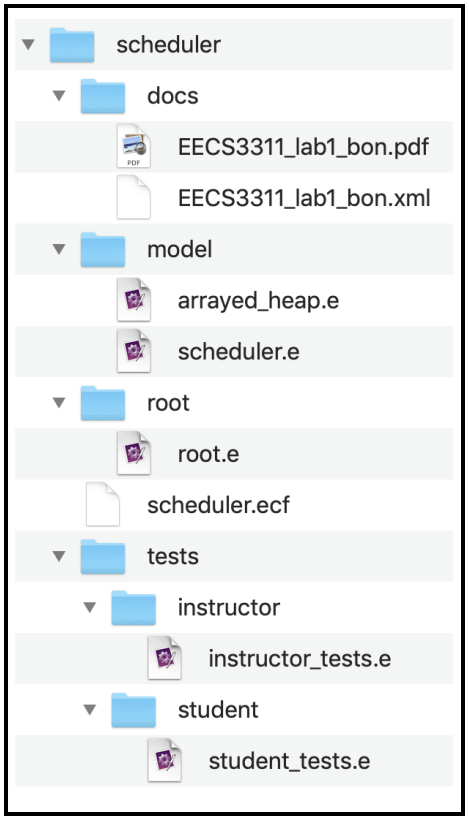
7 Submission

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_workspace
eclean scheduler
```

- 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_workspace
submit 3311 Lab1 scheduler
```

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, if Eiffel project compiles, and if your supplied tests pass. Please be patient and wait until it finishes.

References

[1] Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, and Clifford Stein. *Introduction to Algorithms, Third Edition*. The MIT Press, 3rd edition, 2009. [Cited: Page 5.]

8 Amendments

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