CS2030 Programming Methodology II
Semester 2 2022/2023

25 & 26 January 2023
Problem Set #1 Suggested Guidance
Abstraction and Encapsulation

1. Study the following classes `P` and `Q`:

```
 1:   class P {
 2:       private final int x;
 3:
 4:       P(int x) {
 5:           this.x = x;
 6:       }
 7:
 8:       P foo() {
 9:           P newP = new P(this.x + 1);
10:           return newP;
11:       }
12:
13:       P bar(P p) {
14:           p.x = p.x + 1;
15:           return p;
16:       }
17:   }
18:
19:   class Q {
20:       P baz(P p) {
21:           return new P(p.x + 1);
22:       }
23:   }
```

(a) Which line(s) above violate the `final` modifier of property `x` in class `P`?

Line 14 violates the `final` modifier of `x` due to the assignment.

(b) Which line(s) above violate the `private` modifier of property `x` in class `P`? Relate your observation to the concept of an "abstraction barrier".

Line 21 violates the `private` accessibility modifier of `x`, while line 14 does not.

The abstraction barrier sits between the client and the implementer. Here class `P` is the implementer, and `Q` is the client that makes use of the `p`, an object of `P`.

The barrier is not broken when one one object of type `P` accesses the instance variables of another type `P` object, since `P` is the sole implementer.

2. Consider the following definition of a `Vector2D` class:

```
class Vector2D {
    private final double x;
    private final double y;

    Vector2D(double x, double y) {
        this.x = x;
        this.y = y;
    }

    private double getX() {
        return this.x;
    }

    private double getY() {
        return this.y;
    }

    Vector2D add(Vector2D v) {
        Vector2D newVector = new Vector2D(
                this.getX() + v.getX(),
                this.getY() + v.getY());
        // line A
        return newVector;
    }

    public String toString() {
        return "(" + this.getX() + ", " + this.getY() + ")";
    }
}
```

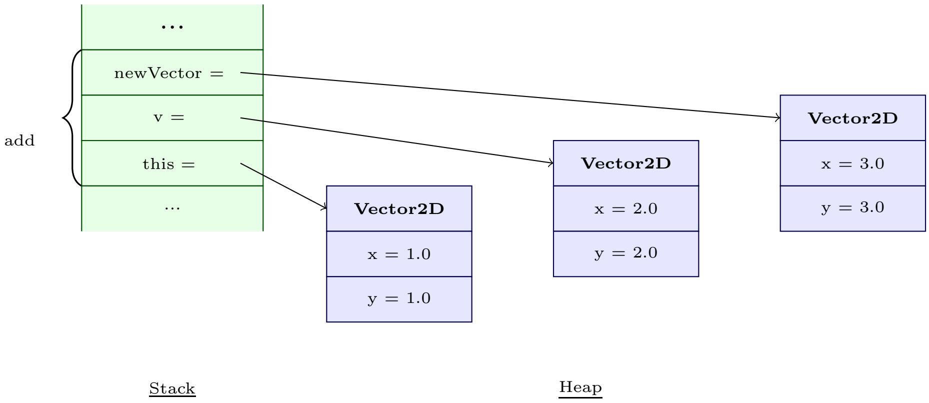
(a) Do the `getX()` and `getY()` methods violate the *Tell-Don't-Ask* principle?

The Tell-Don't-Ask principle relates to an external client asking an object for data and acting on that data, instead of the client telling the object what to do. Since both methods have `private` *access, clients are not allowed to call them, and hence the principle is not violated. The methods here serves as useful helper methods that abstract out the implementation details of retrieving the individual coordinate values from the underlying data representation.*

(b) Suppose that the following is executed in `JShell`:

```
new Vector2D(1.0, 1.0).add(new Vector2D(2.0, 2.0))
```

Show the content of the stack and the heap when the execution reaches the line labelled `A` above. Label your variables and the values they hold clearly. You can use arrows to indicate object references.



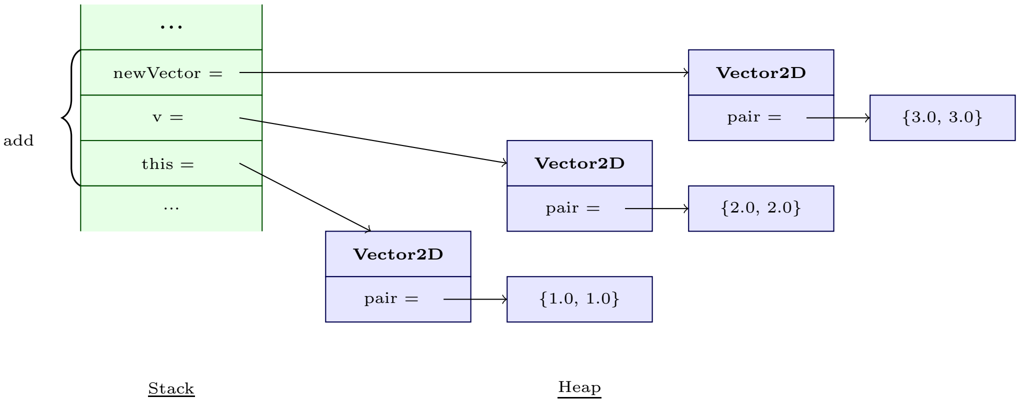
(c) Suppose that the representation of `x` and `y` have been changed to a pair of `double` values:

```
class Vector2D {
    private final Pair<Double, Double> pair;

    Vector2D(double x, double y) {
        this.pair = new Pair<Double, Double>(x, y);
    }

    ...
```

i. What changes do you need for the other parts of class `Vector2D`?
One just needs to modify the implementation of the `getX()` and `getY()` methods.

```
    private double getX() {
        return this.pair.first();
    }

    private double getY() {
        return this.pair.second();
    }
```

ii. Would the statement in 2b above be valid? Show the content of the stack and the heap when the execution reaches the line labelled `A` again.
Yes, the program fragment is still valid. The lower-level implementation of how the x amd y coordinates are stored and operated on in `Vector2D` *is encapsulated from other clients.*


...
newVector =
v =
this =
...
add
Vector2D
pair =
{3.0, 3.0}
Vector2D
pair =
{2.0, 2.0}
Vector2D
pair =
{1.0, 1.0}
Stack
Heap

3. The most effective way to get acquainted with Object Oriented Programming is to look at everyday objects around us, and try to model them. In this exercise, we shall be modeling a book.

   A book comprises of some pages of words (we ignore pictures for now). When reading a book, one can flip from one page to the next, flip a page back, or go to a specified page. Each page of a book comprises several lines of text.

   Here's an example of page 8 from the Dr. Seuss' "One fish, two fish, red fish, blue fish".

   ```
   Yes. Some are red. And some are blue.
   Some are sad.
   And some are glad.
   --- Page 8 ---
   ```

   Here are some specifications of how a `Book` object should behave.

   - `Book::nextPage() : Book`
     Advances the book by one page. Turning the next page beyond the last page remains at the last page.
   - `Book::prevPage() : Book`
     Turns the page back by one page. Turning the page back from the first page remains at the first page.
   - `Book::gotoPage(int pageNumber) : Book`
     Turns to the page specified by `pageNumber`. If `pageNumber` is not a valid page number, then remain with the current page.

   The user (or client) of a `Book` object will only make use of the above methods to read a book. Once a book is created, it's contents will not change. You are free to represent a `Page` as appropriate; just implement the `Book` constructor accordingly.

   Design and test your program, bearing in mind the following Object-Oriented design considerations.

   - Identify the **objects** in the problem statement.
   - **OO Principle #1: Abstraction** — identify the relevant **data** and **functionality** of each object.
   - **OO Principle #2: Encapsulation** — **package** data/functionality into each object and utilize **information-hiding** techniques to prevent client access.
   - Maintain an **abstraction barrier** between the client and the implementor.
   - Application of the **Tell-Don't-Ask** principle.
   - Making use of **immutable** objects for ease of testing.

   Write `JShell` tests to test your program.

Here are some considerations in the design of the classes

- Class dependencies: `Book` has-a list of `Pages`
- Creating a book with default first page using

  ```
  new Book(new ImList<Page>().add(new Page(..)).add(new Page(..))...)
  ```

- Encapsulating the data representation of pages of book:

  ```
  class Book {
      private final ImList<Page> pages;
      private final int pageNum;

      Book(ImList<Page> pages) {
          this.pages = pages;
          this.pageNum = 1;
      }
  }
  ```

- Defining the `toString` method with `private` helper methods:

  ```
  @Override
  public String toString() {
      return this.getCurrentPage() + "\n" + this.pageNum);
  }
  ```

- Deciding which method among `prevPage`, `nextPage` and `gotoPage` to implement first. Which methods depend on which other one?
  - `gotoPage` is more general, `prevPage` and `nextPage` can call `gotoPage`.

    ```
    Book gotoPage(int newPageNum) {
        return new Book(this.pages, newPageNum);
    }
    ```
  - create and return a new `Book`;
  - define appropriate overloaded constructor that takes two arguments;
  - applying the abstraction principle so that one constructor calls the other one.
  - add appropriate conditional check(s) to keep within range of pages.
- Having implemented and tested `gotoPage`, follow up with the definitions of `prevPage` and `nextPage`.
- Testing via method chaining, e.g.

  ```
  new Book(..).gotoPage(1).nextPage().prevPage()...
  ```