Closed Lab #7: Pre-Conditions, Post-Conditions, and Checking Wrappers

1 Objectives

There are four objectives to this lab. First, you will reinforce your understanding of Java *interfaces*. Second, you will reinforce your understanding of *pre-conditions* and *post-conditions*. Third, you will be introduced to the concept of a *checking wrapper*. Fourth, you will gain experience using the *Decorator* pattern to implement a checking wrapper.

2 Requirements

This lab is divided into three parts. You must satisfy the requirements for each part to complete the lab successfully.

2.1 Interfaces

For the first component of the lab, you will design an interface and a corresponding implementation to represent a *bounded set of* `String`*s*. Here *"set"* refers to the mathematical abstraction and *"bounded"* indicates that there is an upper-bound on the number of `String`s that a set instance may contain. Your solution must satisfy the following requirements.

Requirements

- Your interface must be named `BoundedSetOfString`; the corresponding implementation must be named `BoundedSetOfStringImpl`.
- Your interface must provide the following methods: `getBound()`, `getSize()`, `addElement()`, `removeElement()`, `removeAny()`, and `isIn()`.
  - `getBound()` will return the upper-bound on the size of the set instance.
  - `getSize()` will return the current size of the set instance.
  - `addElement()` will accept a `String` as argument and add the `String` to the set. **Assumptions:** `addElement()` will always be called with an argument that is *not* contained within the set. `addElement()` will *not* be called if the number of elements contained in the set is equal to the set's upper-bound.
  - `removeElement()` will accept a `String` as argument and remove the `String` from the set. **Assumption:** `removeElement()` will always be called with an argument that *is* contained within the set.
  - `removeAny()` will remove and return *some* `String` from the set; the particular `String` returned doesn't matter. **Assumption:** `removeAny()` will never be invoked on an empty set.
  - `isIn()` will accept a `String` as argument and return a `boolean` value indicating whether the argument is contained within the set.
- `BoundedSetOfStringImpl` must provide a constructor that accepts the set's upper-bound as argument and assigns an *empty* value to the new set instance.

- **Hint:** Investigate the `String.equals()` function in the Java API. (An interesting exam question might ask you to explain the difference between the expression (`s1==s2`) and `s1.equals(s2)`.)
- **Hint:** A mathematical set can never contain duplicate elements.

2.2 Pre-Conditions and Post-Conditions

When implementing a method, developers often make assumptions about the argument values that may be passed and the particular state of the target object. Or phrased another way, many method implementations impose requirements on *callers* that must be satisfied for the method to work properly. When these requirements are not satisfied, the method may fail. Consider, for instance, trying to invoke `pop()` on an empty `Stack`.

Recall that these requirements are referred to as *pre-conditions*. A pre-condition is a requirement that must be satisfied by a caller *before* a method may be executed — else the method may fail. In the case of `pop()`, the pre-condition normally requires that the `Stack` be non-empty.

Similarly, when the pre-condition of a method is satisfied, the caller assumes that certain conditions will hold when the method returns. In the case of `pop()`, the caller assumes that the top element of the stack will be removed and returned and that no other modifications will be made to the stack. Recall that these conditions are referred to as *post-conditions*. A post-condition is a requirement that must be satisfied by the callee *after* a method has executed — but only if the method's pre-condition was satisfied.

Pre-conditions and post-conditions are the main elements of the *Design-by-Contract* paradigm.

Requirements

For this portion of the lab, you are required to introduce comments within the `BoundedSetOfString` interface. The comments should clearly identify the *pre-condition* of each method. In some cases, there may be no pre-condition; in this case, simply write *"none"*.

2.3 Checking Wrappers

One of the most common programming mistakes is to inadvertently violate a pre-condition. Hence, pre-conditions are often *checked*, especially for methods that will be used by many developers. (So, for example, you might check to ensure that the `Stack` is non-empty within the body of `pop()`.)
An alternative to implementing the checks within the body of the checked class is to develop a *checking wrapper*. This is simply a *Decorator* class that inserts *pre-condition* checks prior to invoking the underlying methods.

Requirements

For the final component of the lab, you must satisfy two requirements:

- You must apply the *Decorator* pattern to implement a checking wrapper that will work with *any* implementation of `BoundedSetOfString`. Your class must be named `BoundedSetOf-StringChecker` and must print a suitable error message to `stdout` whenever a pre-condition violation is detected.

- You must explain to the TA why using a checking wrapper is *better* than implementing the checks directly within the body of `BoundedSetOfStringImpl`.

An interesting exam question might ask you to develop a checking wrapper for a class that checks pre-conditions *and* post-conditions. It might be a good idea to practice with this exercise at home.

3 Grading

To receive full credit for this lab, you must demonstrate that you have satisfied the above requirements. When you think you are ready, ask one of the TAs to check your solution. When they have checked your name off of their list, you are free to leave. If you are unable to complete the lab before the end of the period, but have made a reasonable attempt, you will receive 1/2 credit for the day. If the TA feels that you have not made a reasonable attempt, you will not receive any credit for your participation.

4 Collaboration

You may work independently or with one partner. You must *not* discuss the problem or the solution with classmates outside of your group. Keep this in mind before you choose to work independently.