CSE 331: More Abstract Data Types

Summary of ADTs:
- **abstract state**: how the client should think about the type
- **abstract invariant**: guaranteed to always hold for valid instances
- The methods of an ADT are written in terms of the abstract state

Summary of ADT implementation:
- **concrete state**: the private fields
- **representation invariant** (RI): assertion that always holds when *not* in the middle of a method
- **abstraction function** (AF): maps concrete states to abstract states

```
/** A line of people waiting to check out at the store.
  * People are represented by their name as a String.
  * Think of this as a queue with no duplicates.
public class CheckOutLine {
    /** Returns the number of people in line
    public int size() { ... }

    /** Adds the given person to the back of the line.
      * @modifies this
      * @effects this = this_0 ++ [person]
      * @throws DuplicateException if they're already in line.
    public void enqueue(String person) { ... }

    /** Removes and returns the person at the front of line.
      * @modifies this
      * @effects this = this_0[1..]
      * @throws NoSuchElementException if nobody is in line.
    public String dequeue() { ... }
}
```

```
public class CheckOutLine {
    private List<String> line;
    private Set<String> members;

    public int size() {
        return this.members.size();
    }

    public void enqueue(String person) {
        if (this.members.contains(person))
            throw new ...;
        this.line.add(person);
        this.members.add(person);
    }
}
```

```
 // RI: line and members contain exactly the same elements
 // AF(this) = this.line

 public CheckOutLine(List<String> initialOrder) {
     this.line = initialOrder;  // !
     this.members = new HashSet<>(initialOrder);
 }

 public String dequeue() {
     if (this.size() == 0) { throw new ...; }
     String front = this.line.get(0);
     this.line.remove(0);
     // !
     return front;
 }
```

The pre/postconditions of an ADT method are written in terms of the abstract state, but the implementation is written in terms of the concrete state. To check correctness, need to bridge between abstract and concrete using the AF.

In the pre/postconditions and `@modifies`/`@effects` clauses, "`this`" refers to the abstract state!

An ADT method implementation is correct if:
For any concrete state s satisfying the RI,
if AF(s) satisfies the (abstract) precondition,
then running the implementation results in a concrete state s' that
satisfies the RI and AF(s') satisfies the (abstract) postcondition.

There are three kinds of methods:
Observers/Getters: return an attribute of the object without modifying it
Mutators: change the abstract state of the object
Producers: return a new instance of the ADT with some changes applied
Constructors are a special kind of mutator that initializes the object.

Aliasing: sharing a reference to the same object.
Representation (rep) Exposure: aliasing a mutable object between a client and an ADT.
Copy In: when storing a mutable object from a client, make a copy.
Copy Out: when returning a mutable object to a client, make a copy.

Consider adding this method to the ADT:

```
public List<String> getCurrentLine() {
    return this.line;  // !
}
```

An entire ADT implementation is correct if all of its methods are correct and there is no representation exposure.