CSE 331: Abstract Data Types

An **Abstract Data Type** is a type that is described by the behavior of its operations, *not* by what actual data it stores. The opposite of "abstract" is "concrete". A good example of a concrete data type in Java is a class with public fields.

You are already familiar with many examples: Lists, Sets, _____.
In all of these cases, the client can use the type without knowing how it is implemented.

Benefits of using an ADT: _____.

Downsides of using an ADT: _____.

An ADT consists of:

- A description of the **abstract state**, which is how the client should think about the type.
- An **abstract invariant**, guaranteed to always hold for valid instances.
- A set of **abstract operations**, which are methods with specifications written in terms of the abstract state.

We use "`@modifies this`" to mean that a method modifies the abstract state of the object.

```
/** 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() { ... }
}
```

An ADT **implementation** consists of:

- The **concrete state**: specific choices of how to represent the type using private fields.
- A **representation invariant** (RI) maintained by the implementation so that it always holds on valid instances whenever a method is not in the middle of running.
- An **abstraction function** (AF) describing how to map concrete states to abstract states.
  - AF can assume the representation invariant on its input and must guarantee the abstract invariant on its output.
- Implementations of all the abstract operations with concrete code. Each implementation must be correct:
  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.

```
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;
}
```