1 Administrivia

- Add Date is tomorrow!
- Problem Set 2 out, due next Wednesday
- Office hours tomorrow during lab time

2 Abstract Data Types

An abstract data type (ADT) is just an abstract description of how instances of a certain type behave defined through the methods that type exposes. This includes, but is not restricted to:

- Preconditions — any conditions that need to be met before the behavior of the method is defined. This includes any arguments the method might take.
- Postconditions — any side-effects that calling this method might have. This includes returning values.

An example specification in this style might look like:

```java
static int find ( int [] a, int val )
    requires: val occurs exactly once in a
    effects: returns result such that a[result] = val
```

3 Representation Exposure

Representation exposure occurs when implementation decisions leak incorrect behavior through an abstract interface. This is bad for two major reasons: it can potentially lead to bugs or faulty behavior, and it can inadvertently couple the implementation of an interface to the client of an interface. Some common things that can lead to representation exposure:

- Inconsistent state (e.g. size variable not being updated)
- Exposing internal state (e.g. returning a mutable data structure)
- Poorly defined interfaces (e.g. Page/Weather cache)

4 Representation Invariants

A representation invariant is a formal constraint that specifies when an instance of an abstract data type is well-formed. Why do we bother with representation invariants? Because they:

- Make modular reasoning possible. Without the rep invariant documented, you might have to read all the methods to understand what’s going on before you can confidently add a new method.
- Help catch errors. By implementing the invariant as a runtime assertion, you can find bugs that are hard to track down by other means.

Representation invariants are typically expressed as a function that maps an instance of an abstract data type (i.e. an object) to a boolean value that is true if and only if the object is valid. In order to be able to ensure validity of all instances of a type, rep invariants need to be checked when an instance is:

- Created, most of the time in a constructor
- Changed, through methods that mutate its state

By guaranteeing the validity of an instance in these two places, we can essentially inductively guarantee that the instance will always be valid, as long as we don’t commit the sin of representation exposure.
Listing 1: Skeleton for IntSet and IntGen

```java
public class IntSet {
    private Vector els; // the rep, supports method remove
    private int size; // the rep

    public IntSet () {
        els = new Vector();
        // insert, remove, size methods for IntSet here
        ...
    }

    public Iterator elems () {
        return new IntGen (this);
    }

    // static inner class
    private static class IntGen implements Iterator {
        public boolean hasNext () { ... }
        public Object next () throws NoSuchElementException { ... }
        public void remove (Object o) { ... }
    }
}
```

Listing 2: A bad implementation of IntSet.elems()

```java
public class IntSet {
    public Iterator elems () {
        return els.iterator();
    }
}
```

Listing 3: Example of checkRep() using Tic Tac Toe

```java
public TicTacToe{
    private static final CHECK_REP = true;
    private final Move[][] board = new Board[3][3];

    public TicTacToe(){
        ...
        if (CHECK_REP) checkRep();
    }

    public void doMove(int r, int c, Move player) {
        ...
        if (CHECK_REP) checkRep();
    }

    public Move getVal(int r, int c) { ... }

    private void checkRep (){  
        int countX = ...; int countO = ...;
        assert Math.abs(countX - countO) <= 1 :
        "Invalid number of X and Os on board. #X - #O = " + countX - count0;
        int numWinners = ...;
        assert numWinners <= 1 : "More than one winner on board";
    }
}
```