CS 61B Spring 2017

List Implementations

Discussion 3: February 1, 2017

1 Encapsulation

1.1 An **API**, or application programming interface, is a set of methods and fields that define how we communicate with other software. `SLList` is our first dive into APIs which define **what an object can do** rather than **how that object does it**.

Notice that we define methods like `SLList.addFirst` and `SLList.addLast` in `SLList` but not in `IntList`. We can say that `SLList` has a view of the entire list, rather than an individual node in the list. The `size` field only makes sense in `SLList` because we think of `SLList`, the list itself, as an object.

It doesn't make sense to include a `size` field in `IntList` because `IntList` never has a complete view of the list. We can never be certain that a node in the list is really the 'front' of the list because any other node's `rest` field may refer to it.

Encapsulation is the means by which we separate the concerns of the *functionality* of a class from its *use*, or **API**. This is the central principle of how we *design* data structures.

```
class IntList {
    int first;
    IntList rest;
}

class SLList {
    static class IntNode {
        int item;
        IntNode next;
    }

    IntNode sentinel;
    int size;
}
```

2 Java Miscellany

2.1 **Access control** allows us to restrict the use of fields, methods, and classes.

- `public`: Accessible by everyone.
- `protected`: Accessible by the class itself, the package, and any subclasses.
- *default (no modifier)*: Accessible by the class itself and the package.
- `private`: Accessible only by the class itself.

2.2 **Arrays** are ordered sequences of fixed length. Arrays in Java are proper objects but you'll probably find only one field useful: `length`.

Unlike Python lists, the `length` of an array must be known when creating an array.

```
int[] a = new int[3];
int[] b = {1, 2, 3}; // shorthand for: int[] b = new int[]{1, 2, 3};
```

Uninitialized values have a default value like `0`, `false`, or `null`.

```
String[] c = new String[1];
c[0] == null;
```

Practical tip: Use `java.util.Arrays` to do cool things with arrays like sorting!

Food for thought: Why is every method in `java.util.Arrays` declared `static`?

3 Flatter Me

3.1 Write a method `flatten` that takes in a two-dimensional array `data` and returns a one-dimensional array that contains all of the arrays in `data` concatenated together.

```
public static int[] flatten(int[][] data) {
    int size = 0;
    for (int[] row : data) {
        size += row.length;
    }
    int[] result = new int[size];
    int i = 0;
    for (int[] row : data) {
        for (int value : row) {
            result[i] = value;
            i += 1;
        }
    }
    return result;
}
```

4 When Things Get Tricky

4.1 Define a **recursive** `SLList.get(int index)` method.

```
public class SLList {
    private static class IntNode {
        public int item;
        public IntNode next;
    }

    private IntNode sentinel;

    public int get(int index) {
        return SLList.get(sentinel.next, index);
    }

    private static int get(IntNode node, int index) {
        if (node == null) {
            throw new NoSuchElementException();
        } else if (index == 0) {
            return node.item;
        } else {
            return SLList.get(node.next, index - 1);
        }
    }
}
```