Mehran Sahami
CS 106A

Handout #46A
December 6, 2017

Solution to Section #9

Parts of this handout by Eric Roberts and Patrick Young

1. Primitive vs. Objects

In the first example, the student is thinking a little too literally about the expressions they've written, seeing them as what they *want* them to mean as opposed to what they in fact *do* mean. The problem lies in the comparison:

```
(name == "Q")
```

The correct English translation of this statement is: compare the *address* of the object **`name`** to the *address* of the constant string **`"Q"`**. In other words, **`name`** is a *reference* to a **`String`** object. Since **`name`** was read in from the user, this comparison will always return **`false`**, as it cannot be the same underlying object as the constant string **`"Q"`**. If we actually want to compare the *values* held in those String objects, we should write:

```
name.equals("Q")
```

For comparing values, the **==** operator should only be used with primitive types, such as **`int`**, **`double`**, **`boolean`**, and **`char`**. Variables that represent objects (like **`String`**) are always references (addresses to some location in memory).

In the second example the code actually works as intended. In the expression:

```
(name.charAt(0) == 'Q')
```

we are using the == operator to compare the primitive type **`char`**. Se we are comparing primitives (and not object references), the **==** operator is comparing actual **`char`** values rather than memory addresses. This works just as we would want it to.

Continued on next page

2. Data structure design

```
/*
 * File: ExpandableArray.java
 * -------------------------
 * This class implements the ExpandableList interface, providing
 * methods for working with an array that expands to include any
 * positive index value supplied by the caller.
 */

public class ExpandableArray implements ExpandableList {

/**
 * Creates a new expandable array with no elements.
 */
   public ExpandableArray() {
      array = new Object[0];  // Allows us to check length of array
                              // even when no elements exist
   }

/**
 * Sets the element at the given index position to the specified
 * value. If the internal array is not large enough to contain that
 * element, the implementation expands the array to make room.
 */
   public void set(int index, Object value) {
      if (index >= array.length) {

         // Create a new array that is large enough
         Object[] newArray = new Object[index + 1];

         // Copy all the existing elements into new array
         for (int i = 0; i < array.length; i++) {
            newArray[i] = array[i];
         }

         // Keep track of the new array in place of the old array
         array = newArray;
      }
      array[index] = value;
   }

/**
 * Returns the element at the specified index position, or null if
 * no such element exists.  Note that this method never throws an
 * out-of-bounds exception; if the index is outside the bounds of
 * the array, the return value is simply null.
 */
   public Object get(int index) {
      if (index >= array.length) return null;
      return array[index];
   }


/* Private instance variable */
   private Object[] array;

}
```