Mehran Sahami
CS 106A

Handout #46
December 3, 2018

Section Handout #9: Objects and Data structures

Parts of this handout by Eric Roberts and Patrick Young

1. Primitive vs. Objects

Let's say a student writes the following line of code in a predicate method (i.e., a method that returns a `boolean`):

```
public boolean isNameQ() {
   String name = readLine("Enter name: ");
   return (name == "Q");
}
```

The author of this code thinks that the program will return true if the player's name is "Q". What's the problem here?

Now consider if the code were written as:

```
public boolean isNameQ() {
   String name = readLine("Enter name: ");
   return ((name.length() == 1) && (name.charAt(0) == 'Q'));
}
```

How is the code above different with respect to checking for equality with the value "Q"?

Continued on next page

2. Data structure design

So far in CS106A, we've worked a good deal with arrays and ArrayLists. While arrays have fixed sized, ArrayLists grow as more elements are added (usually, to the end). We could think of potentially an even more powerful idea: an expandable list. The idea here is that we could think of a list that is dynamically expanded to accomodate whatever index we try to access. For example, if we started with an empty expandable list, and tried to add an element at index 14, the expandable list would automatically grow large enough, so that elements 0 through 14 all existed (and we could store the given value at index 14). All the elements of the list that had not previously been given a value would have the value `null`. Then if we tried to store a value at, say, index 21, the expandable list would again grow automatically to have space for elements up to and including index 21. Note that the value at index 14 would still appear to be at index 14 in the expanded list.

Being able to expand a list dynamically is useful enough that it might be worth creating an abstraction to implement it. Such an abstraction is shown in Figure 1 (below), which shows the structure of the `ExpandableList` interface. Recall that an *interface* is simply a specification of a set of methods that must be implemented by any class claiming to implement that interface. The interface itself does not provide the actual implementation. Note that part of the point of this problem is to make an `ExpandableList` general enough to store different objects, so we make the class sufficiently general by having it be able to store any type of object (hence the use of the type `Object` for the elements).

Figure 1. The ExpandableList interface

```
public interface ExpandableList {

/**
 * Sets the element at the given index position to the specified
 * value. If the list is not large enough to contain that
 * element, the list expands to make room.
 */
   public void set(int index, Object 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);

}
```

Your job is to implement this interface in a class named `ExpandableArray`. The `ExpandableArray` class would implement the methods in the `ExpandableList` interface using an array as the underlying data structure to store the information in the list. The proposed structure for the `ExpandableArray` class that implements the `ExpandableList` interface is shown in Figure 2 on the next page.

Figure 2. Proposed structure of the ExpandableArray class

```
/**
 * 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() {
      . . . You fill in the implementation . . .
   }


/**
 * 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) {
      . . . You fill in the implementation . . .
   }


/**
 * 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) {
      . . . You fill in the implementation . . .
   }

}
```

Note that the **set** method specifies that the client can set the value at an index even if it doesn't currently exist in the expandable list. When such a situation occurs, the implementation (using arrays) provided by the **ExpandableArray** class has to allocate a new internal array that is large enough to hold the desired element and then copy all of the existing elements into the new array. To create an expandable array, and set some of its values (which in this case are Strings), you might have code such as:

```
ExpandableArray myList = new ExpandableArray();
myList.set(14, "Bob");
myList.set(21, "Sally");
```

When you wanted to retrieve the value at a particular element of the expandable array, you could do something like the following:

```
String value = (String) myList.get(14);  // Note the cast
if (value != null) {
   println("Got value: " + value);
}
```

In writing your solution, you should keep the following points in mind:

- The underlying structure in your implementation must be an array containing elements of the type **`Object`**. Although using a **`HashMap`** might well be easier, it will be less efficient than the array in terms of the time necessary to look up a specific index.
- Notice that the definition of the abstraction explicitly states that the **`get`** method should return **`null`** if the specified index value does not exist. That means that you will need to check whether the index is out of bounds before trying to select that element.
- You may assume that all of the index values supplied by the client are nonnegative and do not need to check for such values in the implementations of **`get`** and **`set`**.