Lab 6: Loops, ArrayLists, and Arrays

Arrays

As you know from lecture, an array is a data structure that holds a fixed number of elements of a certain type. When you initialize an array, you specify both its size and the type of element it stores. Arrays can be one-dimensional or multidimensional (a 2-dimensional array is just an array of arrays).

```java
int[] numbers = new int[10]; // initializes 1D int array of size 10
HeadTA[] headTas = new HeadTA[4]; // 1D array of HeadTAs (size 4)
Square[][] chessBoard = new Square[8][8]; // 2D array of Squares (8x8)
```

To access an element stored in an array, we access the array at a particular index. Arrays are 0-indexed: the first element is stored at index 0, and the nth element is stored at index n-1.

```
<table>
<thead>
<tr>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
</tr>
</thead>
<tbody>
<tr>
<td>![dog]</td>
<td>![dog]</td>
<td>![dog]</td>
<td>![dog]</td>
<td>![dog]</td>
</tr>
</tbody>
</table>
```

To access the fourth element in the array dogs, we say: `dogs[3]`. We can use this notation to call a method on an object stored at a particular index: for example, “`dogs[3].bark();`” tells the fourth Dog in the array to bark. We could use the same notation to store a particular Dog at that index: for example, “`dogs[3] = new Husky();`”. The notation for multi-dimensional arrays is similar. We could access a particular Square in the 2D chessBoard array we created above with “`chessBoard[5][7]`”.

Confused about arrays? Check out the Oracle Java Tutorials!

Looping Through Arrays

In Java, when we initialize an array, each of its elements is initialized to the default value of the array’s element type. This means that when you initialize an array of int with “`int[] numbers = new int[10];`”, each element starts out as 0. When you initialize an array of
booleans, each element starts out as false. When you initialize an array of objects, each element starts out as null.

Loops are a useful tool for navigating arrays. One use case is populating an array with the values we want to store. Here’s a short code example that creates a 1-dimensional array of booleans and uses a for loop to set each element.

```java
boolean[] myBoolArray = new boolean[5]; // declare, initialize array
for (int i = 0; i < myBoolArray.length; i++) { // for all array’s indices:
    myBoolArray[i] = true; // set element at current index to true
}
```

The result of this code is an array that looks like: {true, true, true, true, true}. If at a later point we need to change any of the elements in the array, we can do so easily: for example, “myBoolArray[1] = false;” changes the second element of myBoolArray, making it: {true, false, true, true, true}.

Check out the lecture slides if you need a refresher on for loops before moving on!

---

### Coloring Arrays

You’re going to initialize and populate a few javafx.scene.paint.Color arrays of specified dimensions, using different colors to form patterns. Our support code will visualize these arrays for you as two-dimensional grids of colored blocks.

1. Run `cs015_install lab6` to install this lab’s stencil code. Open Eclipse and check out the contents of the `lab6` package. For the first exercise, you’ll be working within the file `ArrayBuilder.java`. Open it up now.

2. The `ArrayBuilder` class contains four stencil methods, each of which will create and return a 2D array of Colors. The first method, `buildStripeArray`, has been filled in for you. Run the program by right-clicking on `lab6`'s `App.java` class and choosing “Run as >> Java Application”. Once the window appears, click the “Stripes” button (under the “Arrays” tab) to visualize this array.

3. Fill in the remaining three stencil methods, `buildCheckerArray`, `buildTwoColorArray`, and `buildDiagonalStripeArray`.
   - In each method:
     1. Create an array of Colors of the dimensions specified in the method comments.
     2. Next, loop through the array you’ve created, figuring out the appropriate Color at each index and storing it in the array.
     3. Finally, return the array you’ve created.
ArrayLists

Arrays are handy when you know exactly how many elements you’re going to be dealing with. But what if you want to model a collection of objects whose size may change?

Luckily, the core Java library provides several implementations of “collections” of objects, whose size may change as elements are added and removed. One such implementation is the class java.util.ArrayList. Like an array, an ArrayList stores elements at “indices”, and allows you to access and modify the element stored at a particular index. However, an ArrayList provides convenient methods for adding, removing, and modifying elements, and changes size automatically as elements are added and removed. We’ll give a quick refresher on ArrayLists below—check out the lecture slides for the full scoop.

When declaring an ArrayList, we need to specify what type of object it stores. We would declare and instantiate an ArrayList of Aliens like this:

```java
ArrayList<Alien> aliens = new ArrayList<Alien>();
```

We put the type of object our ArrayList will store within the angle brackets. “ArrayList<Alien>” just means “an ArrayList of Aliens”. When we initialize an array, it takes on the size we tell it to—however, when we initialize an ArrayList, we don’t give it a size. Every ArrayList starts out empty (size 0). To add an element to the ArrayList, call
the `add` method. Let’s add a few Aliens to our `ArrayList`:

```java
aliens.add(new PurpleAlien());
aliens.add(new GreenAlien());
aliens.add(new BlueAlien());
```

To access the element at a specific index in the `ArrayList`, use the `get` method. For example, to tell the second alien in the list to do something, we would write:

```java
aliens.get(1).doSomething();
```

**Note:** This is different from the array syntax, where we would write:

```java
arrayName[1].doSomething();
```

To remove the element at a specific index in the `ArrayList`, use the `remove` method. To remove the first `Alien` in the list, we would write:

```java
aliens.remove(0);
```

Note that since we removed the purple alien, the green alien is now the first in the list, meaning it is at index 0. The blue alien is now the second in the list, meaning it is at index 1.

To replace the element at a specific index in the list with another element, use the `set` method:

```java
aliens.set(1, new PurpleAlien());
```

Looping through an `ArrayList` is pretty similar to looping through an array. For example, the following code replaces every element in the list with a `PurpleAlien`:
for (int i = 0; i < aliens.size(); i++) {
    aliens.set(i, new PurpleAlien());
}

To get the number of elements in your ArrayList, use the size method:

aliens.size();

To remove all elements from an ArrayList, use the clear method:

aliens.clear();

---

**Practice with ArrayLists**

You’re going to fill in a series of methods, each of which performs a particular operation on an ArrayList<Color> called _colors. Our support code will visualize the ArrayList for you and allow you to test the methods you fill in.

1. Open up the file ArrayListBuilder.java. The ArrayListBuilder class contains several stencil methods, each of which should perform an operation on the ArrayList<Color> instance variable called _colors.

2. Run the app and navigate to the “ArrayLists” tab. Each of the buttons at the bottom of the interface calls the corresponding method in the stencil code. Play around with the “Add Pink” and “Clear” buttons (we've filled in these methods for you--the other buttons won’t do anything until you fill in their methods)! You should see the support code’s visualization of _colors update as elements are added and removed.

3. Fill in the remaining stencil methods according to the descriptions in the method comments. You can check out the ArrayList Javadocs for a full description of the ArrayList’s methods. Run the app and play around to test your work.

---

**Check Point 2:** Call over a TA to check you off!