ArrayLists and Wrapper Classes

Arjun Chandrasekhar

Getting started

In this worksheet you will complete some practice problems related to your reading on ArrayLists and Wrapper classes. Here is how you should complete the worksheet:

Create a file in your text editor (either Notepad++ or BBedit). Put the following starter code into it:

```
import java.util.ArrayList;

public class ArrayLists
{
    public static void main(String[] args)
    {

    }
}
```

You must include the `import` statement at the top! Save your file as `ArrayLists.java`.

Open up the command line application (either Terminal or Command Prompt). Navigate to the location where you saved the file. Compile and run the file. You're now ready to comple the worksheet!

Re-sizing an array

An array is generally appropriate when the number of array elements is fixed in size. However, if we want to re-size an array, we essentially have to declare a new array with a new size.

Write code that creates an `int[]` called `nums` with the contents `{4, 6, 8}`. Then add the value 10 to the end of the array as follows:

1. **Create a new `int[]` called `temp` whose size is one bigger than the size of `nums`.**
2. **Copy the contents from `nums` into `temp`.**
3. **Insert the value `10` at the last place of `temp`.**
4. **Reassign `nums` to be equal to `temp`.**

Print out the contents of `nums` to confirm that your code worked.

Recall: that in order to print out the contents you need to do the following:

1. Add the line `import java.util.Arrays;` to the top of your file.
2. Use the command `System.out.println(Arrays.toString(nums));` to print out `nums`.

ArrayLists

As you can see, adding new values to an array is quite tedious. Fortunately, if we want to store several values in one place (and keep track of their order), Java has a built-in class for that exact purpose. Java's ArrayList class implements a "dynamic" array that can grow in size as needed.

To create an ArrayList of items of the same type, we use the following syntax:

```
ArrayList<TYPE> items = new ArrayList<TYPE>();
```

However you replace `TYPE` with the actual data type that you want to use. For example, to create an `ArrayList` of `String`s containing the names of characters from a movie, you could use:

```
ArrayList<String> characters = new ArrayList<String>();
```

`ArrayList`s can hold objects of any type, as long as all the objects share the same type. First we'll focus on Strings, but later we'll see how to create `ArrayList`s containing the other primitive data types that we have learned about.

Here is a table of ArrayList methods:

Method	**Description**	**Example**
`add(...)`	Inserts a new element at the *end* of the `ArrayList`	`characters.add("Rick");`
`get(i)`	Retrieves the element at the i-th position (0-based indexing)	`characters.get(4)` - retrieves the value stored at position 4 (i.e. the 5th element)
`size()`	Retrieves the size of the `ArrayList` (specifically the number of elements)	`characters.size()`

ArrayList practice

Do the following:

1. **Create an `ArrayList` of `Strings` called `aList`.**
2. **Create three `String` variables called `str1`, `str2`, and `str3` with the values `"ET"`, `"Phone"`, and `"Home"`.**
3. **Add the following Strings to the ArrayList: `"ET"`, `"Phone"`, `"Home"`.**
4. **Print out the contents of `aList` using `System.out.println(aList);`**
    a. **What does this print out?**
    b. **Is this different from what happens when you do this with an array?**

Fill in the following table for what the value of each expression is. First write down your prediction, then run code to check if your prediction was right.

Expression	Prediction	Actual Output
`aList.get(-1)`		
`aList.get(0)`		
`aList.get(1)`		
`aList.get(2)`		
`aList.get(3)`		
`aList.add('!')`		
`aList.size()`		
`aList.length`		
`aList.length()`		

Draw a picture of what data is on the stack and on the heap. Recall that for every non-primitive data value, the variable is on the stack; the variable contains a memory address which points to the actual data values on the heap.

Looping through an `ArrayList`

Examine the following code:

```
ArrayList<String> aList = new ArrayList<String>();
aList.add("Troy");
aList.add("and");
aList.add("Abed");
for(int i = 0; i < aList.size(); i++)
{
    String str = aList.get(i);
    System.out.println(str);
}
```

Trace through the code step by step.

- **Write down every single line number (and the relevant instruction) that gets executed, in the order that they get executed.**
- **Whenever a variable gets initialized or updated, state the resulting value of the variable.**
- **Trace the code to completion. Do not simply state that a certain sequence will be repeated - actually write out the repetition.**

Even if you understand this code clearly, trace through it fully. There is more room on the next page if necessary.

For-each loops

We often want to go through the items of a collection one at a time. For example, we can write the following loop:

```
int[] nums = {1, 2, 3, 4, 5};
for(int i = 0; i < nums.length; i++)
{
    int curr = nums[i];
    // do stuff to curr
}
```

However, having to manage the loop variable `i` can be very tedious. An alternative approach is to use an *enhanced for loop*, sometimes called a *for-each loop*. Here is an alternative way to write that same loop:

```
int[] nums = {1, 2, 3, 4, 5};
for(int curr : nums)
{
    // do stuff to curr
}
```

The variable `curr` is underlined, to emphasize what parts of the normal `for` loop get translated to the enhanced for loop.

Re-write each of the following pieces of code using an enhanced for loop.

```
ArrayList<String> aList = new ArrayList<String>();
// add stuff to aList
for(int i = 0; i < aList.size(); i++) {
    String curr = aList.get(i);
    // do stuff to curr
}
```

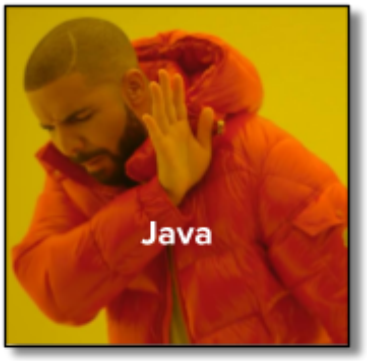
```
String[] words = {"Troy", "and", "Abed"};
for(int i = 0; i < words.length; i++)
{
    String nextWord = words[i];
    // do stuff to nextWord
}
```

Wrapper classes and auto-boxing/unboxing

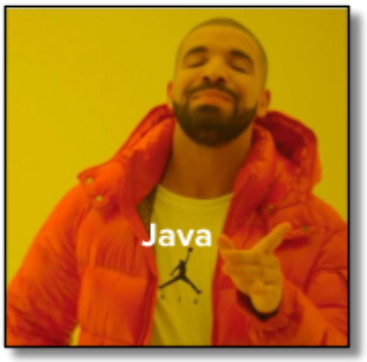
Java will not allow you to add primitive data to an `ArrayList`. This is because Java uses a design principle called generics, and this design choice requires `ArrayList`s (and every other Collection) to only be composed of objects. Objects have certain inheritance properties that primitive data does not, and inheritance is the foundation of generics. All of this will be covered in CS2 - for the time being, just remember that you cannot declare an `ArrayList` of primitive data types.

What you *can* do is declare an ArrayList of *wrapper* classes. Each primitive data type has a corresponding wrapper class that allows you to treat primitive data as if it were non-primitive. Here are all of the wrapper classes for each primitive data type.

Primitive data type	Corresponding wrapper class
`byte`, `short`, `int`, `long`	`Byte`, `Short`, `Integer`, `Long`
`float`, `double`	`Float`, `Double`
`boolean`	`Boolean`
`char`	`Character`




```
ArrayList<int> nums = new ArrayList<int>();
```




```
ArrayList<Integer> nums = new ArrayList<Integer>();
```

Auto-boxing/unboxing

Because `nums` takes `Integer` objects, you may think we need to create `Integer` objects to add into the `ArrayList`. We could add and retrieve values as follows:

```
// Create an Integer object before we add it to the ArrayList
Integer num = new Integer(42);
nums.add(num);

// Retrieve a value and store it as an Integer object
Integer myNum = nums.get(0);
```

Of course, this is a bit tedious to have to make everything into an Integer. For example, the following code is pretty tedious.

```
int num = 42;
Integer num2 = new Integer(num); // convert it from int to Integer to add
to ArrayList
nums.add(num2);
```

Fortunately, Java has a design feature called *auto-boxing* and *auto-unboxing*. Java will automatically convert between primitive data and wrapper classes as needed so that you don't have to think about it.

```
nums.add(42); // Java automatically "waps" the int 42 to an Integer
int val = nums.get(0); // Java automatically "unwraps" from Integer to an
int
```

Examine the following code

```
ArrayList<Integer> aList = new ArrayList<Integer>();
aList.add(5);
double x = aList.get(0);
System.out.println(x);
```

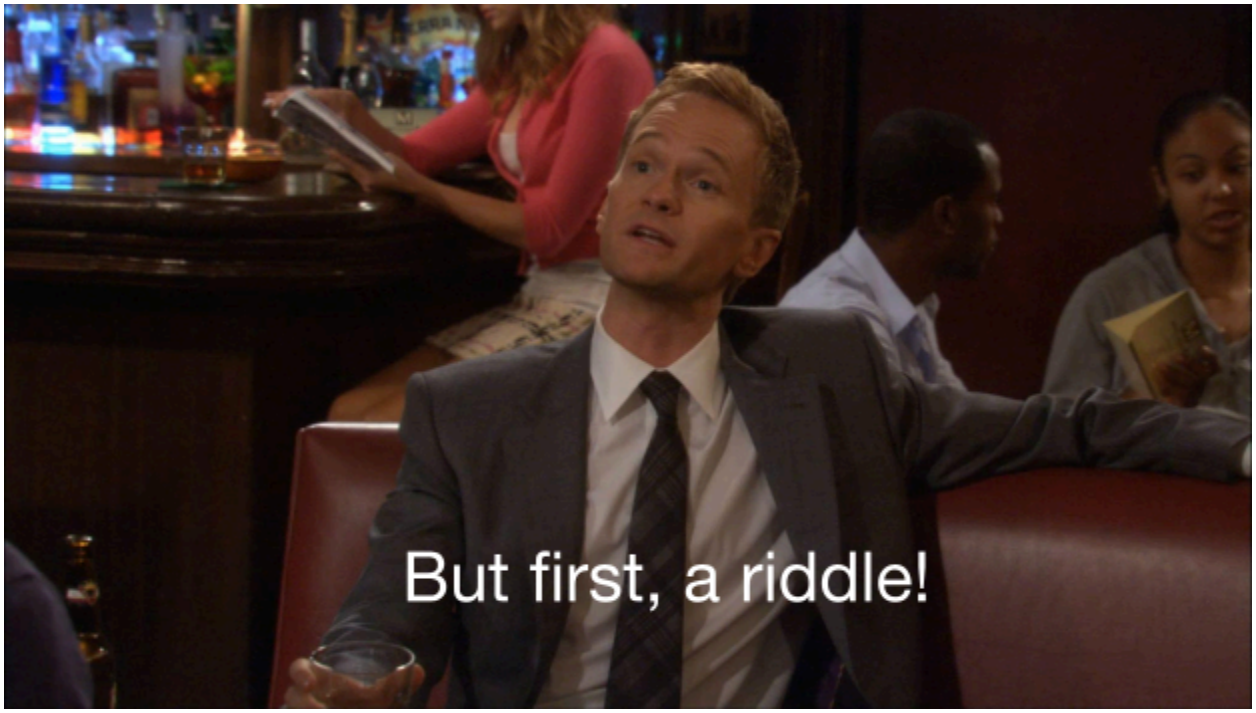
- **Predict the output of the code. Then run the code to confirm if your prediction was correct.**
- **Draw a picture of the stack and heap contents at the end of the code.**
- **State any places where Java does any sort of automatic converting between data types.**

Examine the following code

```
ArrayList<Integer> aList = new ArrayList<Integer>();
aList.add(5);
short x = aList.get(0);
System.out.println(x);
```

Predict the output of the code. Then run the code to confirm if your prediction was correct.

Duplicate objects



What color is my house?

My wife and I have the same address on our respective drivers licenses

1. I go to the address listed on my driver's licence and paint the house at that address purple
2. My wife goes to the address listed on her driver's license and paints the house house at that address orange

What color is my house?

Recall that when we create a non-primitive variable, the variable holds the object's *memory address*. The memory address points to data on the heap - but the variable's value is just a memory address.

With that in mind…

What will the following code output?

```
ArrayList<String> arr1 = new ArrayList<String>();
ArrayList<String> arr2 = arr1;
arr1.add("ABC");
arr2.add("ABC");
System.out.println(arr1.size());
```

- **Answer the question, then run the code to check if your answer was correct.**
- **Draw a picture of what is on the stack and the heap. You may want to use arrows to show which memory address on the stack point to which data values on the heap.**