Set 26: ArrayLists

Skill 26.01: Explain the advantages and disadvantages of ArrayLists as compared to normal Arrays
Skill 26.02: Implement an ArrayList
Skill 26.03: Store primitive data types inside an ArrayList
Skill 26.04: Traverse an ArrayList

Skill 26.01: Explain the advantages and disadvantages of ArrayLists as compared to normal Arrays

Skill 26.01 Concepts

An ArrayList is one of several classes that implement the List interface. As its name suggests, an ArrayList also involves arrays. Basically, everything we learned concerning arrays can also be applied to ArrayList objects, however, with slightly different methods.

While ArrayLists certainly have their advantages over normal arrays, they also have their disadvantages. These are discussed below.

Advantages

Ordinarily arrays are fixed in size. When we create an array, we anticipate the largest possible size it will ever need to be, and when instantiating the array, dimension it to that size. We call this the physical size of the array and it always remains that size even though at some point in your program you may wish to only use a portion of the array. The size of that portion is called the logical size. Your own code must keep up with that size.

By contrast, an ArrayList expands and contracts to meet your needs. If you remove items from or add items to your ArrayList, the physical and logical sizes are always identical. This can be very important if you wish to be conservative of memory usage. With memory being so abundant and inexpensive today, this is no longer the advantage it once was.

Because ArrayLists can be resized at anytime, the *add* method associated with ArrayLists allows for very easy insertion of new items in the interior of the list without the nuisance of having to pre-move ore remove existing items.
A final advantage of ArrayLists is that iterator objects are provided, which allows for easy traversal of the list. Normal arrays on the other hand must be traversed using loops.

Disadvantages

The biggest disadvantage of using ArrayLists over normal arrays is that they can only store objects. If we wish to store primitives such as integers, doubles, or booleans, they must be converted into their object counterpart (more on this later). Similarly, when we retrieve things from an ArrayList, they come out as objects.

“Big deal”, you say....certainly, if we store objects in the list, then we expect to get objects back when we retrieve them from the list. Yes, but it’s worse than one might think. When retrieving an object from a list, it doesn’t come back as the same type object that was originally stored. Rather, it comes back as an Object type object. To use the retrieved element, it will be necessary to cast it down to its original object type (more on this later).

- An ArrayList is similar to an Array, but is more flexible and can be resized
- Only objects can be stored in an ArrayList
- An ArrayList implements the List interface

Skill 26.02: Implement an ArrayList

Skill 26.02 Concepts

Before using ArrayLists, you must import the required java packages. These packages are listed below.

```
import java.util.ArrayList;
import java.util.List;
```

Alternatively, you can import all the utilities at once using the following,

```
import java.util.*;
```

ArrayLists can be created in two different ways.

Method 1

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

Method 2

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

The <String> parameter indicates that objects we add to the list can only be String types. (Instead of String we could use any object type.) This insures “type safety” and would result in a compile time error if we ever tried to add some other type object to aryLst.

Type parameters also remove the burden of casting Object type objects retrieved from a list back to their original type. The ArrayList utilities provides a number of useful methods for manipulating our ArrayLists. These are summarized below. In each case we can assume an ArrayList of String elements called *aryLst* has been declared.

signature	example	description
void add(Object o)	aryLst.add("Dog");	Adds "Dog" to the end of the list
void add(int index, Object o)	aryLst.add(3, "Cat");	Inserts "Cat" at index 3 after moving the existing object at index 3 and greater up one notch
Object get(int index)	String q = aryLst.get(3);	Retrieve object at position 3
Object remove(int index)	String q = aryLst.remove(3);	Removes object at position 3, then adjusts the size of the list
Object removeLast( )	String q = aryLst.removeLast( );	Removes object at end of list and returns that object
Object set(int index, Object o)	String q = aryLst.set(3, "fish");	Replaces object at position 3 with "fish" and returns original object
boolean isEmpty( )	aryLst.isEmpty();	Returns true if there are no objects in the list
int size()	aryLst.size();	Returns the number of objects in the list
void clear()	aryLst.clear();	Removes all objects from the list

The video below illustrates how these methods can be applied to ArrayLists

https://www.youtube.com/watch?v=oyEnR-AGQx4&t=4s

Skill 26.02: Exercise 1

Skill 26.03: Store primitive data types inside an ArrayList

Skill 26.03 Concepts

For each primitive data type tyere is a corresponding Object type.

- int -> Integer
- double -> Double
- boolean -> Boolean

Java versions 5 or later support *autoboxing*, which allows us to easily convert between the primitive and object types. Consider the examples below,

```
Integer iObject = 10;
int iPrimitive = iObject;

Double dObject = 1.1;
double dPrimitive = dObject;

Boolean bObject = true;
boolean bPrimitive = bObject;
```

Notice that in the above examples we did not need to create a new Object for any of the corresponding primitive object types. Nor did we have to get the value of the object to assign it to its corresponding primitive equivalent. The complete process of creating an Integer object and assigning its value to its primitive equivalent is illustrated below. Java automatically took care of the extra steps required of creating the Integer Object and getting its value. This process is called *autoboxing*.

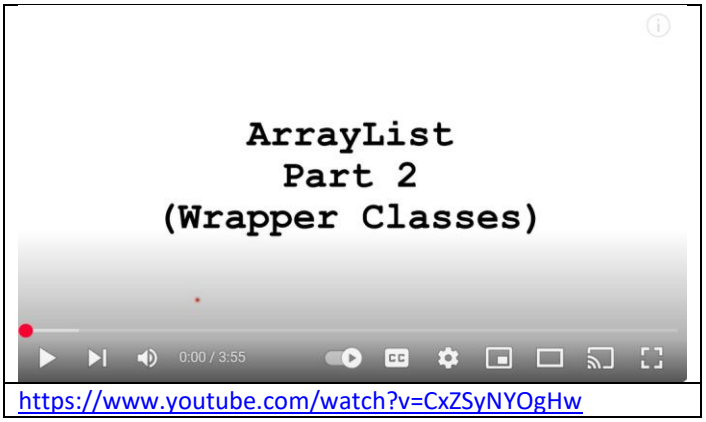
```
Integer iObject = new Integer(10) //creates an Integer object with a value of 10
int iPrimitive = Integer.valueOf(iObject); //gets the value of the iObject and assigns it to
iPrimitive
```

The concept of autoboxing is very useful as it applies to ArrayLists, because recall that ArrayLists *can only store objects*. Autoboxing therefore allows us to easily create ArrayLists and then convert the values to the required primitive equivalent.

With autoboxing the below code is valid.

```
ArrayList<Integer> i = new ArrayList<Integer>(); //creates an ArrayList of Integer objects
i.add(1); //adds 1 to the end of the list
i.add(2); //adds 2 to the end of the list
i.add(3); //adds 3 to the end of the list
int iPrimitive = i.get(0); //assigns the Integer object at position 0 to the primitive iPrimitive
```

This concept is further illustrated in the video below



https://www.youtube.com/watch?v=CxZSyNYOgHw

[Skill 26.03: Exercise 1]

Skill 26.04: Traverse an ArrayList

Skill 26.04 Concepts

ArrayLists can be traversed using the same techniques as normal arrays. The only differences are how we determine the length of the list be traversed and how we retrieve the values. Consider the following example,

Traversing an ArrayList with a for-loop

```
ArrayList i = new ArrayList(); //creates an ArrayList of whatever you want
i.add(1); //adds 1 to the end of the list
i.add(2); //adds 2 to the end of the list
i.add(3); //adds 3 to the end of the list

System.out.println(i.size());

for(int j = 0; j < i.size(); j++){ //size() is used to get the length of the array

        System.out.println(i.get(j)); //get() is used to get the value at position j

}
```

A modified for-each loop can also be applied to traverse an array. This technique is illustrated in the code below,

Traversing an ArrayList using a modified for-each loop

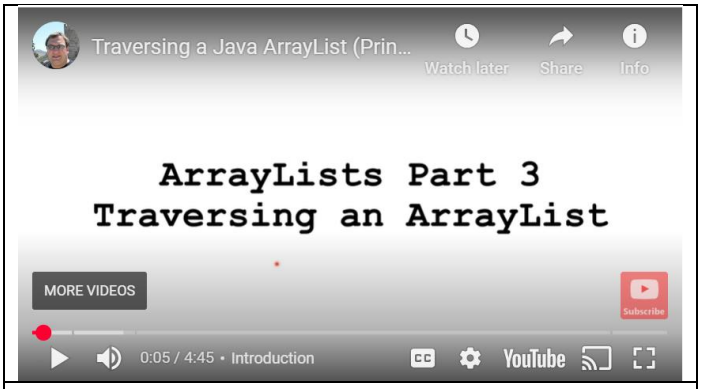
```
ArrayList<Integer> i2 = new ArrayList<Integer>(); // creates an ArrayList of Integer
objects
i2.add(1); // adds 1 to the end of the list
i2.add(2); // adds 2 to the end of the list
i2.add(3); // adds 3 to the end of the list

for (int tempValue : i2) { // gets each Integer and assigns it to tempValue

       System.out.println(tempValue); // prints the tempValue

}
```

Another important consideration is that because items can be added and removed dyanmically from an ArrayList during a traversal objects may be outputted twice or missed entirely. This concept is illustrated further in the video below.



https://www.youtube.com/watch?v=r5VavXQd3pc&t=5s

Skill 26.04: Exercise 1