STUDENT OUTLINE

Lesson 19 – ArrayList

INTRODUCTION: It is very common for a program to manipulate data that is kept in a list. You have already seen how this is done using arrays. Arrays are a fundamental feature of Java and of most programming languages. But because lists are so useful, the Java Development Kit includes the `ArrayList` class, which works much like an array but has additional methods and features.

The key topics for this lesson are:

A. Array Implementation of a list
B. The `ArrayList` Class
C. Object Casts
D. The Wrapper Classes

VOCABULARY:

ABSTRACT DATA TYPE	LIST
`ArrayList`	CAST
WRAPPER	

DISCUSSION:

A. Array Implementation of a list

1. A data structure combines data organization with methods of accessing and manipulating the data. For example, an array becomes a data structure for storing a list of elements when we provide methods to find, insert, and remove an element. At a very abstract level, we can think of a general "list" object: a list contains a number of elements arranged in sequence; we can find a target value in a list, add elements to the list, and remove elements from the list.

2. An abstract description of a data structure, with the emphasis on its properties, functionality, and use, rather than on a particular implementation, is referred to as an *Abstract Data Type* (ADT). An ADT defines methods for handling an abstract data organization without the details of implementation.

3. A "List" ADT, for example, may be described as follows:

   Data organization:
   - Contains a number of data elements arranged in a linear sequence

   Methods:
   - Create an empty List
   - Append an element to List
   - Remove the i-th element from List
   - Obtain the value of the i-th element
   - Traverse List (process or print out all elements in sequence, visiting each element once)

CAPACITY VS. SIZE

4. A one-dimensional Java array already provides most of the functionality of a list. When we want to use an array as a list, we create an array that can hold a certain maximum number of elements; we then keep track of the actual number of values stored in the array. The array's length becomes its maximum capacity and the number of elements currently stored in the array is the size of the list.

5. However, Java arrays are not resizable, hence they are called *static*. If we want to be able to add elements to the list without worrying about exceeding its maximum capacity, we must use a class with an `add` method, which allocates a bigger array and copies the list values into the new array when the list runs out of space, hence it is called *dynamic*. That's what the `ArrayList` class does.

CAN MODIFY THE CAPACITY

6. The `ArrayList` class builds upon the capabilities of arrays. An `ArrayList` object contains an array of object references plus many methods for managing that array. The biggest convenience of an `ArrayList` is that you can keep adding elements to it no matter what size it was originally. The size of the `ArrayList` will automatically increase and no information will be lost.

WILL GROW AS NEEDED

7. However, this convenience comes at a price:

   a. The elements of an `ArrayList` are object references, not primitive data like **`int`** or **`double`**.

   b. Using an `ArrayList` is slightly slower than using an array directly. This would be important for very large data processing projects.

   c. The elements of an `ArrayList` are references to `Object`. This means that often you will need to use type casting with the data from an `ArrayList`. "Type Casting" means to change the type of an object in order to conform to another use. See Part C, Object Casts, below.

THIS IS THE "OLD SCHOOL" WAY OF DOING THINGS. WE MUST LEARN THIS AS THE COLLEGE BOARD REQUIRES US TO BE ABLE TO DO THIS. THERE NEWER METHODOLOGIES WHICH WE'LL LEARN SOON.

B. The `ArrayList` Class

1. To declare a reference variable for an `ArrayList`, do this:

```
// myArrayList is a reference to a future ArrayList object
ArrayList myArrayList;
```

DECLARATION →

You do not say what type of object you are intending to store. An `ArrayList` is like an array of references to `Object`. This means that any object reference can be stored in an `ArrayList`. To declare a variable and to construct an `ArrayList` with an unspecified initial capacity do this:

```
// myArrayList is a reference to an ArrayList object. The
// Java system picks the initial capacity.
ArrayList myArrayList = new ArrayList();
```

INSTANTIATE → DEFAULT

This may not be very efficient. If you have an idea of what size `ArrayList` you need, start your `ArrayList` with that capacity. To declare a variable and to construct an `ArrayList` with an initial capacity of 15, do this:

```
// myVector is a reference to an ArrayList object with an
// initial capacity of 15 elements.
ArrayList myArrayList = new ArrayList (15);
```

WISER, MORE EFFICIENT →

2. The elements of an `ArrayList` are accessed using an integer index. As with arrays, the index is an integer value that starts at 0.
   - For retrieving data from an `ArrayList` the index is 0 to `size-1`.
   - For setting data in an `ArrayList` the index is 0 to `size-1`.
   - For inserting data into an `ArrayList` the index is 0 to `size`. When you insert data at index `size`, you are adding data to the end of the `ArrayList`.

3. To add an element to the end of an `ArrayList` use:

```
// add a reference to an Object to the end of the
// ArrayList, increasing its size by one
boolean add(Object obj);
```

Here is an example program. To use the `ArrayList` you must import the `java.util` package:

Program 19 – 1

```
import java.util.* ;

class NameList
{
  public static void main(String[] args)
  {
    ArrayList names = new ArrayList(10);

    names.add("Cary");
    names.add("Chris");
    names.add("Sandy");
    names.add("Elaine");

    // remove the last element from the list
    //  note - remove returns an object which must be "cast" to
    //  a String before assignment. Explained in next section
    String lastOne = (String)names.remove(names.size()-1);
    System.out.println("removed: " + lastOne);
    names.add(2, "Alyce"); // add a name at index 2

    for (int j = 0; j < names.size(); j++)
      System.out.println( j + ": " + names.get(j));
  }
}
```

Run Output:

```
removed: Elaine
0: Cary
1: Chris
2: Alyce
3: Sandy
```

4. The `add()` method adds to the end of an `ArrayList`. To set the data at a particular index use:

```
// Replaces the element at index with objectReference
Object set(int index, Object obj)
```

The `index` should be within 0 to `size-1`. The data previously at `index` is replaced with `obj`. The element previously at the specified position is returned.

5. To access the object at a particular index use:

```
// Returns the value of the element at index
Object get(int index)
```

The index should be 0 to `size-1`.

6. Removing an element from a list: The `ArrayList` class has a method that will do this without leaving a hole in place of the deleted element:

```
// Removes the element at index from the list and returns
// its old value; decrements the indices of the subsequent
// elements by 1
Object remove(int index);
```

"AUTO COMPACTING"

The element at location `index` will be eliminated. Elements at locations `index+1`, `index+2`, ..., `size()-1` will each be moved down one to fill in the gap.

7. Inserting an element into an `ArrayList` at a particular index: When an element is inserted at `index` the element previously at `index` is moved up to `index+1`, and so on until the element previously at `size()-1` is moved up to `size()`. The size of the `ArrayList` has now increased by one, and the capacity can be increased again if necessary.

OVERLOADED

```
// Inserts obj before the i-th element; increments the
// indices of the subsequent elements by 1
void add(int index, Object obj);
```

Inserting is different from setting an element. When `set(index, obj)` is used, the object reference previously at `index` is replaced by the new `obj`. No other elements are affected, and the size does not change.

C. Object Casts

1. One of the difficulties with building array lists with `Object` for the item type is that methods for returning the items of the array list return things of type `Object`, instead of the actual item type.

ERROR b/c AN Object IS NOT A KIND OF String

2. For example, consider the following:

```
ArrayList aList = new ArrayList();
aList.add("Chris");
String nameString = aList.get(0); // THIS IS A SYNTAX ERROR!
System.out.println("Name is " + nameString);
```

This code creates an `ArrayList` called `aList` and adds to the list the single `String` object `"Chris"`. The intent of the third instruction is to assign the item `"Chris"` to `nameString`. The state of program execution following the `add` is that `aList` stores the single item, `"Chris"`. Unfortunately, this code will never execute, because of a syntax error with the statement:

```
String nameString = aList.get(0); // THIS IS A SYNTAX ERROR!
```

The problem is a type conformance issue. The `get` method returns an `Object`, and an `Object` does not conform to a `String` (even though this particular item happens to be a `String`).

FIX IT IN THE "OLD SCHOOL" WAY

3. The erroneous instruction can be modified to work as expected by incorporating the `(String)` cast shown below.

```
String nameString = (String)aList.get(0);
```


YOU MUST SOLVE LABS FOR 19 IN THIS WAY. (POINTS)

D. Wrapper Classes

1. Because numbers are not objects in Java, you cannot insert them directly into array lists. To store sequences of integers, floating-point numbers, or **boolean** values in an array list, you must use *wrapper classes*.
   Recall Lab 3.1 MathFun

2. The classes `Integer`, `Double`, and `Boolean` wrap number and truth values inside objects. These wrapper objects can be stored inside array lists.

3. The `Double` class is a typical number wrapper. There is a constructor that makes a `Double` object out of a double value:

   ```
   Double r = new Double(8.2057);
   ```

   Conversely, the `doubleValue` method retrieves the double value that is stored inside the `Double` object

   ```
   double d = r.doubleValue();
   ```

4. To add a primitive data type to an array list, you must first construct a wrapper object and then add the object. For example, the following code adds a floating-point number to an `ArrayList`:

   ```
   ArrayList grades = new ArrayList();
   double testScore = 93.45;
   Double wrapper = new Double(testScore);
   grades.add(wrapper);
   ```

   To retrieve the number, you need to cast the return value of the `get` method to `Double`, and then call the `doubleValue` method:

   ```
   Double wrapper = (Double)grades.get(0);
   double testScore = wrapper.doubleValue();
   ```

5. This was an unwieldy usage in the early versions of Java. We'll discover another approach to deal with this in the next lesson.

6. The `ArrayList` class contains an `Object[]` array to hold a sequence of objects. When the array runs out of space, the `ArrayList` class allocates a larger array; it doubles in size each time a resize is required.

SUMMARY/ REVIEW: Like an array, an `ArrayList` contains elements that are accessed using an integer index. However, unlike an array, the size of an `ArrayList` will expand if needed as items are added to it. As these examples show, `ArrayList` can be very useful. The package `java.util` also includes a few other classes for working with objects. We'll look at some of them in later lessons.

ASSIGNMENT: Lab Exercise L.A.19.1, *IrregularPolygon*
Lab Exercise L.A.19.2, *Permutations*