STUDENT OUTLINE

Lesson 20 – 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)

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. 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. That's what the `ArrayList` class does.

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.

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.

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;
```

   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();
```

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);
```

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 20 – 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 thelist and returns
   // its old value; decrements the indices of the subsequent
   // elements by 1
   Object remove(int index);
   ```

   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.

   ```
   // 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.

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`).

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

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

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*.

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. 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.

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.

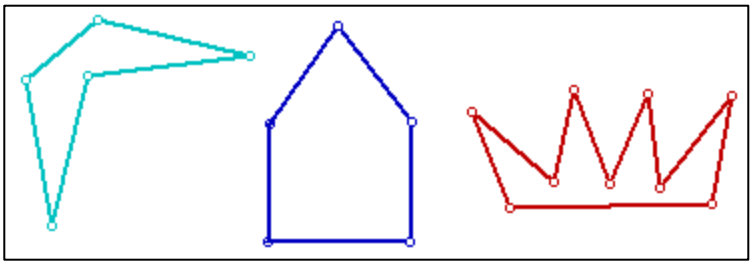
ASSIGNMENT: Lab Exercise L.A.20.1, *IrregularPolygon*
Lab Exercise L.A.20.2, *Permutations*

LAB EXERCISE

Irregular Polygon

Background:

Polygons are closed two-dimensional shapes bounded by line segments. The segments meet in pairs at corners called *vertices*. A polygon is *irregular* if not all its sides are equal in length. The figure below shows examples of irregular polygons:



(source: Intermath Dictionary, http://www.intermath-uga.gatech.edu/dictnary/descript.asp?termID=186):

Assignment:

1. Implement a class `IrregularPolygon` that contains an array list of `Point2D.Double` objects.

2. The `Point2D.Double` class defines a point specified in double precision representing a location in (x, y) coordinate space. For example, `Point2D.Double(2.5, 3.1)` constructs and initializes a point at coordinates (2.5, 3.1). Details can be found at:

   http://java.sun.com/j2se/1.4.1/docs/api/java/awt/geom/Point2D.Double.html

3. Use the following declarations as a starting point for your lab work.

```
import java.awt.geom.*;  // for Point2D.Double
import java.util.*;      // for ArrayList
import apcslib.*;        // for DrawingTool

class IregularPolygon
{
   private ArrayList myPolygon;

   // constructors
   public IregularPolygon() {   }

   // public methods
   public void add(Point2D.Double aPoint) {   }

   public void draw() {   }

   public double perimeter() {   }
```

```
    public double area() {   }
}
```

4. The program should use the Drawing Tool to draw the polygon by joining adjacent points with a line segment, and then closing it up by joining the end and start points.

5. Write methods that compute the perimeter and the area of a polygon. To compute the perimeter, compute the distance between adjacent points, and total up the distances. The area of a polygon with corners $(x_0, y_0),...,(x_{n-1}, y_{n-1})$ is the absolute value of:

$$\frac{1}{2}(x_0y_1 + x_1y_2 + ... + x_{n-1}y_0 - y_0x_1 - y_1x_2 - ... - y_{n-1}x_0)$$

   Note: add *n* products, then subtract *n* products, then divide by 2. The result will be negative or positive depending on the order in which the products are taken, i.e., which products are subtracted and which are added.

6. As a test case, the parallelogram formed by the following coordinates has a perimeter of 17.41 units and an area of 1700 square units: (20, 10), (70, 20), (50, 50), (0, 40).

LAB EXERCISE

Permutations

Assignment:

1. Write a program that produces random permutations of the numbers 1 to 10. "Permutation" is a mathematical name for an arrangement. For example, there are six permutations of the numbers 1,2,3: 123, 132, 231, 213, 312, and 321.

2. To generate a random permutation, you need to fill an `ArrayList` with the numbers 1 to 10 so that no two entries of the array have the same contents. You could do it by brute force, by calling `Random.nextInt()` until it produces a value that is not yet in the array. Instead, you should implement a smart method. Make a second `ArrayList` and fill it with the numbers 1 to 10. Then pick one of those at random, *remove it*, and append it to the permutation `ArrayList`. Repeat ten times.

3. Implement a class `PermutationGenerator` with the following method:

```
ArrayList nextPermutation
```

Instructions:

1. Turn in your source code and a printed run output.

2. The run output will consist of 10 lists of random permutations of the number 1 to 10. Example output is shown below:

```
Random Permutation List Generator

List  1:   4  6  8  1  9  7 10  5  3  2
List  2:   6  8  1  7  3  4  9 10  5  2
List  3:   2  4  9  6  8  1 10  5  7  3
List  4:   8  5  4  3  2  9  6  7  1 10
List  5:  10  3  2  6  8  9  5  7  4  1
List  6:   9 10  3  2  1  5  6  8  4  7
List  7:   3  8  5  9  4  2 10  1  6  7
List  8:   3  2  4  5  7  6  9  8 10  1
List  9:   4  1  5 10  8  3  6  2  7  9
List 10:   3  5  2  4  1  7  9  6  8 10
```

ArrayList Methods

```
int size();                     // Returns the number of elements
                                // currently stored in the list

boolean isEmpty();              // Returns true if the list is empty,
                                // otherwise returns false

boolean add(Object obj);        // Appends obj at the end of the list;
                                // returns true

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

Object set(int i, Object obj); // Replaces the i-th element with obj;
                                // returns the old value

Object get(int i);              // Returns the value of the i-th
                                // element

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

Note: this is an incomplete list: see http://java.sun.com/j2se/1.4.1/docs/api/java/util/ArrayList.html for the entire list.