APCS ArrayList Notes

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.

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.

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.

However, this convenience comes at a price:

- The elements of an **`ArrayList`** are object references, not primitive data like **int** or **double**.
- Using an **`ArrayList`** is slightly slower than using an array directly. This would be important for very large data processing projects.
- 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.

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:

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

The following program is an example of the use of ArrayList:

```
1.  import java.util.*;
2.
3.  public class NameList
4.  {
5.    public static void main (String[] args)
6.    {
7.      ArrayList names = new ArrayList(10);
8.
9.      names.add("Cary");
10.     names.add("Chris");
11.     names.add("Sandy");
12.     names.add("Elaine");
13.
14.     // remove the last element from the list
15.     //   note - the remove returns an object which must "cast" to
16.     //   a String before assignment.
17.     String lastOne = (String)names.remove(names.size()-1);
18.     System.out.println("removed: " + lastOne);
19.     names.add(2, "Alyce"); // add a name at index 2
20.
21.     for (int j = 0; j < names.size(); j++)
22.       System.out.println( j + ": " + names.get(j));
23.   }
24. }
```

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);
```

Wrapper Classes

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*.

Say you want to store numeric values in an ArrayList, but an ArrayList can only hold Objects. The java.lang package provides several “wrapper” classes with which you can represent primitive data types as objects.

READ THE LESSON!!

APCS SQUARE-ROOT CALCULATOR

This activity consists of the following:

I. (20 pts) Complete the `for` loop in the `NumberDemo` class shown below so that it will generate fifteen random integers between 1,000 and 100,000 inclusive.

II. (80 pts) Write the `Number` class — to be used with the driver `NumberDemo` — to calculate the square root of 15 random numbers between 1,000 and 100,000. Note that the method(s) in the `Number` class have to be `static` because the driver does not create a `Number` object but instead it simply has the class name — in this case `Number` — followed by the method name (`Number.sqrt(num)`).

III. (20 pts Extra Credit) ➔ Write the program using a recursive method.

IMPORTANT!!!!!

You absolutely cannot use any `Math` class methods anywhere in the `Number` class (i.e., Part II).

Your program MUST calculate the square of a number to within .001 of its actual value (as determined by a scientific calculator).

Sample output:

```
  number     square root
-----------------------
  30,331       174.158
  20,866       144.451
   6,368        79.800
   5,482        74.041
   5,114        71.512
  78,503       280.184
  60,690       246.353
  67,943       260.659
  18,686       136.697
  14,295       119.562
  23,516       153.349
  37,212       192.904
  61,909       248.815
  78,938       280.959
  12,559       112.067
```

```
public class NumberDemo
{
    public static void main( String[] args )
    {
        System.out.println();
        System.out.printf("%9s  %14s", "number", "square root\n");
        System.out.print( " " );
        for(int k=0; k<24; k++) System.out.print("-");
        System.out.println();
        for( ? )
        {
            int num = ?
            System.out.printf( "%,9d %12.3f", num, Number.sqrt(num));
            System.out.println();
        }
    }
}
```