AP Computer Science — Lists

There are two types of `Lists` in Java that are commonly used: `Arrays` and `ArrayLists`. Both types of list structures allow a user to store ordered collections of data, but there are some differences as well.

- **The `Array` type**

  The `Array` type is used to store a collection of items—primitives or objects—that are all of the same type: `int`, `double`, `boolean`, `Rectangle`, `Person`, `BankAccount`, etc. The items in the list are referred to using the name of the Array and the *index* (position) of each *element* (item in the array), enclosed in square brackets: `Person[0]` is the first Person in the array `Person`.

  Arrays are declared and instantiated in a manner similar to that of other objects:

  ```
  Person[] friends = new Person[10];
  ```

  In this case we've reserved space for ten objects of the type `Person` in the array, indexed from `0` to `9`.

  Arrays of different types are initialized with different initial values, depending on the type of the array. Arrays that hold `int` or `double` values have each element initialized to a value of `0`, while arrays of Strings and other objects are initialized to `null` values.

  You can initialize the values in an `Array` at the time of declaration by enclosing them in curly braces. In this format, you do not use the `new` operator:

  ```
  int[] fibs = {1, 1, 2, 3, 5, 8, 13, 21, 34, 55};
  ```

  You can also initialize the elements of the array after it has been created by addressing each one explicitly:

  ```
  int[] fibs = new int[10];
  fibs[0] = 1;
  fibs[1] = 1;
  ```

  etc.

  It is useful to be able to identify the length of an Array using the `length` operator—note the lack of parentheses. A common operation is traversing through an entire array using an index variable:

  ```
  for (int i = 0; i < fibs.length; i++)
  {
      System.out.println(fibs[i]);
  }
  ```

  Note that `Arrays` have a fixed size, so they can't be extended. (There are strategies for creating a new array of a larger size, however, and moving information from the current array into that larger one.) Also, an `Array` may not be completely filled with useful data. In that case, a separate `int` "companion variable" is commonly used to track the amount of useful data in the array:

  ```
  import java.util.Scanner;

  Scanner in = new Scanner(System.in);
  final int LENGTH = 50;
  String[] friends = new String[LENGTH];
  int currentSize = 0;      // companion variable
  System.out.println("Enter friends, blank line to stop");
  String entry = in.nextLine();
  while (entry != "" && currentSize < friends.length)
  {
      friends[currentSize] = entry;
      currentSize++;
      if (currentSize < friends.length)
          entry = in.next();
  }
  ```

- **The `ArrayList` type**

  The `ArrayList` type is similar to the `Array`: it is used to store a collection of items that are all of the same type, but it can only be used to store *objects*. An `ArrayList` can't be used for any type of primitive—`int` or `double` values, for example. The most important difference, however, is that `ArrayLists` are *dynamic*—they don't have a fixed size, and can shrink or grow to contain data as needed. The `ArrayList` also has methods that allow for dynamic insertion and deletion of data from the list.

  To create an `ArrayList`, the `ArrayList` package must first be imported:

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

  The ArrayList is declared (for an example array of `Person` objects) as follows:

  ```
  ArrayList<Person> people;
  ```

  and is instantiated in this way:

  ```
  people = new ArrayList<Person>();
  ```

  These two steps may be combined into a single step if desired:

  ```
  ArrayList<Person> people = new ArrayList<Person>();
  ```

  An `ArrayList` is a type of *generic class*, so we need to specify what type of data we're going to use with it by enclosing the type in angle brackets `< >`. And that type needs to be an *object*.

  To interact with an `ArrayList` object there are a number of useful methods. To get the length of an `ArrayList`, use the `.size()` method:

  ```
  for (int i = 0; i < people.size(); i++)...
  ```

  To add an element to the array, use the `.add()` method:

  ```
  people.add(friend);       # where "friend" refers to a Person object
  ```

  To refer to a specific element, use the `.get()` method:

  ```
  for (int i = 0; i < people.size(); i++)
  {
      System.out.println(people.get(i));
  }
  ```

  Other common methods include:

`.add(position, object)`	inserts a new object at the specified index
`.set(position, object)`	replaces the current object at that position with another object
`.remove(position)`	removes the object at the specified index

  In order to be able to use ArrayLists with primitives like `int` and `boolean`, Java has *wrapper classes* that "wrap" an object around a primitive class. Thus, the `Integer` class is an object that wraps around and represents an `int` value. As a result, an ArrayList that was going to be used to keep track of prime numbers might be declared this way:

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

  Values of the type `int` are *auto-boxed* ("converted") to `Integer` objects when added to an `ArrayList` of the type `Integer`.

- **Two ways of going through a List**
  Whether you're going through an `Array` or an `ArrayList`, there are two ways of looping through a list.

  - **Using an index value**
    Very often you'll need to identify the *index* (location) of a value in a List. In that case, you need to use an index variable with a `while`- or `for`-loop:

```
// Looking for the position of a searchValue
int i = 0;
while (i < values.size())
{
    if (values.get(i) == searchValue)
        return i;
}
return -1;   // searchValue not found
```

  - **Using an iterator**
    Sometimes you just need the values in the List, and you don't care about that position. In that case, you might consider using an *enhanced for-loop*:

```
// Enhanced for-loop example
double sum = 0;
for (double value : values)      // read "for each value in values..."
{
    sum += value;
}
return sum;
```

    The enhanced for-loop can be used with both Arrays and ArrayLists.

There are a wide variety of uses for these `List` data structures. Here are some of the more common ones.

4. **Finding a Value**

```
/**
 * Given a list of items, find and return the location of a specified value
 * on the list.
 * @param values an Array of int values
 * @param itemToFind an integer that may be on the list
 * @return the index of the itemToFind, or -1 if the item isn't found
 */
public int findIndex(int[] values, int itemToFind)
{
    for (int i = 0; i < values.length; i++)
    {
        if (itemToFind) == values[i])
            return i;
    }
    return -1;
}
```

5. **Finding a Maximum or Minimum Index or Value**

```
/**
 * Given a list of items, find and return the location of the largest value
 * on the list.
 * @param values an ArrayList of double values
 * @return the index of the largest value
 */
public int findIndexOfMax(ArrayList<Double> values)
{
    int maxIndex = 0;
    for (int i = 1; i < values.size(); i++)
    {
        if (values.get(i) > values.get(maxIndex))
            maxIndex = i;
    }
    return maxIndex;
}
```

6. **Other Commons List Tasks**
   You should know how to count matches (using a counter variable), sum values in a list (using a sum variable), swap two elements in a list, insert or delete items from a list, etc.

7. **Copying an Array**
   There are challenges involved with copying arrays because of how references work in Java. You need to know about *shallow copies* and *deep copies*, and cloning.
   a. The statement
   `array2 = array1;`
   doesn't make a copy of `array1`—it creates a new *reference* to `array1`, and that's rarely what you want to do.
   b. The statement
   `array2 = array1.clone();`
   creates a second array as desired, which can be manipulated independently. If it's an array of objects, however, the clone operation creates new references to *the same objects that were in the old array* (a *shallow* copy). That probably isn't what you wanted to do, either.
   c. To make a completely separate copy, you probably need ot make a new array of the same dimensions as the old one, and if there are objects in the old array, make completely new copies of those objects as well (a *deep copy*). This process is mostly beyond the scope of this course.

8. **Resizing an Array**
   If you need to resize an Array... you can't. What you have to do is make a copy of the old array into a larger array, transfer the old values into the new array, and then point the array reference to the new array. You can use arrayCopy to copy over a specific part of an array:
   `arrayCopy(Object src, int srcPos, Object dest, int destPos, int length)`
   copies a length of an array from the specified source array, beginning at the specified position, to the specified position of the destination array.

9. **Two-dimensional Arrays**
   One common use for arrays is for tracking tables of data that can be thought of as existing in rows and columns, or for maintaining an x-y style grid of "locations" that can be used to map information. This information is almost always maintained via `Arrays` (rather than `ArrayLists`).

   To declare a two-dimensional array of integers:
   `int[][] values = new int[4][7];`

This represents a "grid" of 10 rows, with each row consisting of 12 columns. It can be visualized this way:

```
grid = [  [  value0,  value1,  value2,  value3,  value4,  value5,  value6 ] ,
          [  value7,  value8,  value9, value10, value11, value12, value13 ] ,
          [ value14, value15, value16, value17, value18, value19, value20 ] ,
          [ value21, value22, value23, value24, value25, value26, value27 ]  ];
```

The information represented by `value15` is located at row 2, column 1, and we would indicate it with the reference `grid[2][1]`.

Typically, two-dimensional arrays are referenced using nested loops:

```
    for (int row = 0; row < values.length; row++)
        for (int col = 0; col < values[0].length; col++)
        {
            // Do something with grid[row][col]
        }
```

EXERCISES

1. Write a code segment that creates an integer `Array` of 100 elements and fills it with the integers 0-99.

2. Write a one-line code segment that creates a String `Array` with the elements "`Alice`", "`Bob`", "`Charlie`", and "`Dave`."

3. Write a code segment that creates an `ArrayList` of Strings. Store the elements "`Alice`", "`Bob`", "`Charlie`", and "`Dave`" in that `ArrayList`. Print out all the elements using an indexed loop. Change the element "`Bob`" to "`Betsy`", and delete the element "`Charlie`." Then print out all the elements using an enhanced `for` loop.

4. Write a code segment that takes this list of values—`{ 3, 2, 7, 9, 10, 12, -5, 42, 101, 496}`—stores them in an `Array`, and prints out their sum.

5. With the same list of values as above—stored in an `ArrayList` now—write a code segment that counts the number of even values in the list.

6. Write a code segments that creates a 10-by-12 multiplication table in a two-dimensional array. The value stored at any `[row][column]` location is the result obtained by multiplying the row by the column.

7. Write a program that takes the thirty `Integer` values (0 - 29) stored in a one-dimensional `ArrayList` called `oneD`, and transfers them into a two-dimensional, 5 × 6 `Array` called `twoD`.

8. Write a program that allows two users to play the game of tic-tac-toe. Each turn, the user enters a symbol ("X" or "O"), a row (0-2) and a column (0-2). The game should print out the board after each turn so players can see the game develop over time. The program doesn't need to check to see who wins.

9. Continue the Tic-Tac-Toe game described in Problem 8: calculate who won the game by checking rows, columns, and diagonals, and print out a message congratulating the winner.