

# *Java*<sup>TM</sup> *Methods*

An Introduction  
to Object-Oriented Programming

Maria Litvin

Phillips Academy, Andover, Massachusetts

Gary Litvin

Skylight Software, Inc.

Skylight Publishing  
Andover, Massachusetts

Skylight Publishing  
9 Bartlet Street, Suite 70  
Andover, MA 01810

web: <http://www.skylit.com>  
e-mail: [sales@skylit.com](mailto:sales@skylit.com)  
[support@skylit.com](mailto:support@skylit.com)

**Copyright © 2001-2003 by Maria Litvin, Gary Litvin, and  
Skylight Publishing**

All rights reserved.

**You are allowed to print one copy of this document for each  
copy of *Java Methods* that you own.**

Otherwise, no part of this publication may be reproduced, stored in a retrieval system, or transmitted, in any form or by any means, electronic, mechanical, photocopying, recording, or otherwise, without the prior written permission of the authors and Skylight Publishing.

Library of Congress Control Number: 2001126002

ISBN 0-9654853-7-4

The names of commercially available software and products mentioned in this book are used for identification purposes only and may be trademarks or registered trademarks owned by corporations and other commercial entities. Skylight Publishing and the authors have no affiliation with and disclaim any sponsorship or endorsement by any of these products' manufacturers or trademarks' owners.

Sun, Sun Microsystems, Java, and Java logos are trademarks or registered trademarks of Sun Microsystems, Inc. in the United States and other countries.

1 2 3 4 5 6 7 8 9      07 06 05 04 03

Printed in the United States of America

# Chapter 12½



## **ArrayList**

1.1	Prologue	2
1.2	Interfaces	3
1.3	ArrayList's Constructors and Methods	7
1.4	<i>Lab</i> : Creating an Index For a Document	12
1.5	Summary	15
	Exercises	16

## 12½.1 Prologue

Java arrays are convenient and safe. However, they have one limitation: once an array is created, its size cannot change. If an array is full and you want to add an element, you have to create a new array of a bigger size, copy all the values from the old array into the new array, then reassign the old name to the new array. For example:

```
Object[] arr = new Object[someSize];  
  
...  
  
Object[] temp = new Object[2 * arr.length]; // double the size  
  
for (int i = 0; i < arr.length; i++)  
    temp[i] = arr[i];  
  
arr = temp;
```

The old array is eventually recycled by the garbage collector.

If you know in advance the maximum number of values that will be stored in an array, you can declare an array of that size from the start. But then you have to keep track of the actual number of values stored in the array and make sure you do not refer to an element that is beyond the last value currently stored in the array.

The library class `ArrayList` from the `java.util` package automates the handling of such situations. An `ArrayList` keeps track of the *capacity* of the array (the currently allocated space) and the *size* of the array (the length of the portion currently in use). When the size reaches `ArrayList`'s capacity and you need to add a value, the `ArrayList` automatically increases its capacity by executing code similar to the fragment shown above.

The `ArrayList` class provides the `get` and `set` methods, which access and set the  $i$ -th element; these methods check that  $0 \leq i < \text{size}$ . In addition, an `ArrayList` has convenient methods to insert and remove a value.

From a more abstract point of view, an `ArrayList` represents a “list” of objects. A “list” holds a number of values arranged in sequence. The `ArrayList` class falls into the Java *collection* framework and implements the `java.util.List` interface. This is explained in more detail in Section 12½.2. (Another library class, `java.util.LinkedList`, also implements `java.util.List` and has the same methods, but the list is implemented differently.)

**An `ArrayList` can only hold objects. If you need to store values of a primitive data type in a list, use a standard array or convert the values into objects of the corresponding wrapper class (`Integer`, `Double`, etc.).**

In the rest of this chapter we will briefly review Java interfaces, including the `Comparable` interface, discuss a subset of `ArrayList`’s constructors and methods, and do a lab — create an index of all the words in a text document.

## 12½ . 2 Interfaces

**A Java interface specifies one or several methods, without any code. A class that implements an interface must supply these methods.**

`interface` and `implements` are Java reserved words.

The syntax for an interface definition is similar to a class definition, only instead of class you write `interface`, and methods have no code, only signatures. For example:

```
public interface Edible
{
    String getFoodGroup();
    int getCaloriesPerServing();
    void setRecommDailyServings(int qty);
}
```

The `Edible` interface describes three methods. The keyword `public` is omitted in method signatures because all methods in an interface are assumed to be `public`.

An interface is an abstraction: it tells us what a certain type of object should be able to do, but nothing about how it will do it. A class that implements the interface supplies the code for all the methods listed in that interface.

The signatures of the corresponding methods in the class must be exactly the same as in the interface definition (except for the names of the parameters, of course). For example:

```
public class Apple implements Edible
{
    private int servingsPerDay;

    < constructors not shown >

    public String getFoodGroup() { return "Fruits"; }
    public int getCaloriesPerServing() { return 70; }
    public void setRecommDailyServings(int num)
    { servingsPerDay = num; }

    public double getFallingTime(int height)
    {...}

    < other methods not shown >
}
```

Names of interfaces often sound like adjectives because an interface supplies an additional property to the object whose class implements that interface. In the above example, it sounds reasonable to say, “An Apple is Edible.”

Why do we need interfaces? For two reasons. First, when we state that a class implements an interface, this gives the compiler a chance to verify that the class indeed has all the methods from the interface with the correct signatures. Second, an interface supplies a secondary data type to the objects of any class that implements that interface. For example, we can declare a variable of the `Edible` data type and assign to it a reference to an `Apple` object:

```
Edible x = new Apple(...);
```

This represents the IS-A relationship of inheritance between the interface’s data type and an object of the class that implements that interface: an `Apple` IS-A(n) `Edible` object. In this sense, a situation where a class implements an interface is similar to the situation where a class extends another class: the class inherits the data type of the interface, but not the code, of course, because an interface’s methods have no code.

We can also pass an `Apple` type of object to any method that expects an `Edible` type of parameter. This allows us to write more general, reusable code. For example:

```
public class Breakfast
{
    private int myTotalCalories = 0;
    ...
    public void eat(Edible obj, int servings)
    {
        myTotalCalories += obj.getCaloriesPerServing() * servings;
    }
    ...
}
```

The eat method above works whether obj is an Apple, a Pancake, or any other object of a class that implements Edible. The getCaloriesPerServing method will be called automatically for an obj of a particular type due to polymorphism.



One of the commonly used library interfaces is Comparable. This interface specifies one method:

```
int compareTo(Object other);
```

The compareTo method provides a way to compare this object to another object, usually of the same class. The compareTo method returns a positive integer if this object is “greater than” other, zero if they are “equal,” and a negative integer if this object is “less than” other. What is “greater than,” “equal,” or “less than” depends on the implementation; it is precisely the compareTo method that defines the order in a class. This order is referred to as the *natural order* for the objects of a class.

String, Integer, Double, and a few other Java library classes implement Comparable. Strings are compared lexicographically, while Integer and Double objects are compared based on their numeric value.

The compareTo method takes an argument of the type Object, so it is applicable to all the various classes of objects that implement Comparable. When this method is implemented in a particular class, other is usually expected to be an object of the same class and is cast back into its class when necessary. In the Integer class, for example, compareTo might be implemented as follows:

```
public int compareTo(Object other)
{
    return intValue() - ((Integer)other).intValue();
}
```

Note that the cast operator has a lower rank than the “dot,” so you need parentheses around `(Integer)other`. If you forget either the cast or the parentheses in the above code, you will get a “cannot resolve symbol” syntax error for the `intValue` method in the `Object` class because `Object` does not have an `intValue` method.



A class can extend one superclass and implement one or several interfaces all at the same time. For example:

```
public class Pizza extends Circle
    implements Edible, Comparable
{
    private int servingsPerDay;
    private boolean extraCheese;

    < constructors and other methods not shown >

    public int getSize()
    {
        return getRadius(); // getRadius is inherited from Circle
    }

    public String getFoodGroup() { return "Breads and Cereals"; }

    public int getCaloriesPerServing()
    {
        int r = getSize();
        int caloriesPerSqInch = 18;
        if (extraCheese) caloriesPerSqInch += 8;
        return (int) (caloriesPerSqInch * Math.PI * r * r / 6);
    }

    public void setRecommDailyServings(int num)
    { servingsPerDay = num; }

    public int compareTo(Object other)
    {
        return getSize() - ((Pizza)other).getSize();
    }
}
```

With this definition, a `Pizza` object can be treated as a `Pizza`, as a `Circle`, as an `Edible`, or as a `Comparable` object, depending on the situation.



### 12½.3 ArrayList's Constructors and Methods

As we mentioned in the prologue, the `java.util.ArrayList` class implements the `java.util.List` interface. We can import these names into our source code —

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

or

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

— and use the short versions of their names in our code.

---

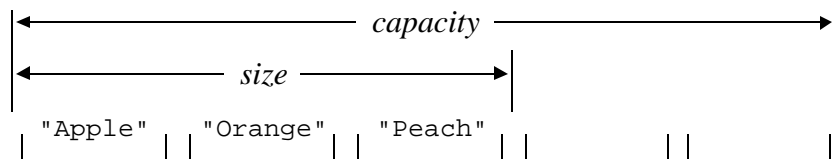
```
int size()                // Returns the number of values
                           //   currently stored in the list
boolean isEmpty()         // Returns true if the list is empty,
                           //   otherwise returns false
boolean add(Object obj)   // Appends obj at the end of the list;
                           //   returns true
void add(int i, Object obj) // Inserts obj as the i-th value;
                           //   increments the indices of the
                           //   subsequent values by 1
Object set(int i, Object obj) // Replaces the value of 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 value from the
                           //   list and returns the removed
                           //   value; decrements the indices
                           //   of the subsequent values by 1
boolean contains(Object obj) // Returns true if this list contains
                           //   an object equal to obj
                           //   (the equals method is used for
                           //   comparison)
int indexOf(Object obj)    // Returns the index of the first
                           //   occurrence of obj in this list,
                           //   or -1 if obj is not found
                           //   (the equals method is used for
                           //   comparison)
```

---

**Figure 12½ - 1. Commonly used List and ArrayList methods**

The `List` interface has over two dozen methods, but we will discuss only a subset of more commonly used methods. These are shown in Figure 12½ - 1.

As the name implies, `ArrayList` implements `List` as an array. An `ArrayList` has two instance variables that hold the capacity of the list and the size of the list, respectively (Figure 12½ - 2). The capacity of the list is the length of the currently allocated array that holds the list values. The size of the list is the number of values currently stored in the list; the `size` method returns that number.



**Figure 12½ - 2. The size and capacity of an `ArrayList`**

`ArrayList`'s no-args constructor creates an empty list (`size() == 0`) of some default capacity (e.g. 10). For example:

```
ArrayList list = new ArrayList();
```

Another constructor, `ArrayList(int capacity)`, creates an empty list with a given initial capacity. If you know in advance the maximum number of values that will be stored in your list, it is better to create an `ArrayList` of that capacity from the outset to avoid later reallocation and copying of the list.

`ArrayList` implements `List`, and `ArrayList`'s methods are `List`'s methods, shown in (Figure 12½ - 1).

The values in an `ArrayList` are numbered by the indices from 0 to `list.size() - 1`. The `get(i)` method returns the value of the *i*-th element in the list. The `set(i, obj)` method sets the value of the *i*-th element to `obj` and returns the old value. The parameter *i* must be from 0 to `list.size() - 1`, otherwise these methods report a run-time error, `IndexOutOfBoundsException`.

The `add(int i, Object obj)` method inserts `obj` into the list before the *i*-th value, so that `obj` becomes the *i*-th value in the list. This method shifts the old *i*-th value and all the subsequent values (if any) to the right and increments their indices by one. It also increments the size of the list by one. This method checks that  $0 \leq i \leq list.size()$  and throws `IndexOutOfBoundsException` if it isn't. The

overloaded version of `add` with one parameter, `add(Object obj)`, appends `obj` at the end of the list and increments the size of the list. The `remove(i)` method removes the *i*-th value from the list, shifts all the subsequent values (if any) to the left by one, and decrements their indices. It also decrements the size of the list. `remove` returns the removed value.



**Since any Java class extends `Object`, an `ArrayList` can hold objects of any type. However, when you retrieve an object from a list, you may need to cast it from `Object` back into its actual type to be able to call its methods.**

For example:

```
List listOfNumbers = new ArrayList();
int sum = 0;

listOfNumbers.add(new Integer(1));
listOfNumbers.add(new Integer(2));
...

for (int i = 0; i < numbers.size(); i++)
    sum += ((Integer)listOfNumbers.get(i)).intValue();

return sum;
```

Recall that the cast operator has a lower rank than the “dot,” so you need parentheses around `(Integer)listOfNumbers.get(i)`.



**You have to be rather careful with the `add` and `remove` methods: keep in mind that they change the indices of the subsequent values and the size of the list.**

The following innocent-looking code, for example, intends to remove all occurrences of “like” from an `ArrayList` of strings:

```
ArrayList words = new ArrayList();
...
int n = words.size();

for (int i = 0; i < n; i++)
    if ("like".equals(words.get(i)))
        words.remove(i);
```

However, after the first "like" is found and removed, the size of the list `words` is decremented and becomes smaller than `n`. Once `i` goes past the current list size, the program will be aborted with an `IndexOutOfBoundsException`.

And that is not all. Even if we fix this bug by getting rid of `n` —

```
ArrayList words = new ArrayList();
...

for (int i = 0; i < words.size(); i++)
    if ("like".equals(words.get(i)))
        words.remove(i);
```

— another bug still remains. When an occurrence of "like" is removed, the subsequent words are shifted to the left. Then `i` is incremented in the `for` loop. As a result, we will skip the next word. If there are two "like" words in a row, the second will not be removed. The correct code should increment `i` only if the word is not removed. For example:

```
int i = 0;
while (i < words.size())
{
    if ("like".equals(words.get(i)))
        words.remove(i);
    else
        i++;
}
```



**An `ArrayList` holds references to objects. It can hold duplicate values — not only `obj1.equals(obj2)`, but also several references to the same object (i.e., `obj1 == obj2`).**

**An object can change after it has been added to a list (unless that object is immutable).**

**The same object can belong to several lists.**

A list can also hold `nulls` after calls to `add(null)` or `add(i, null)`.

Consider, for example, the following two versions of the method `makeGuestList` that builds a list of people (objects of the class `Person`) from a given array of names:

Version 1:

```
public List makeGuestList(String[] names)
{
    List list = new ArrayList();
    for (int i = 0; i < names.length; i++)
        list.add(new Person(names[i]));
    return list;
}
```

Version 2:

```
public List makeGuestList(String[] names)
{
    List list = new ArrayList();
    Person p = new Person();
    for (int i = 0; i < names.length; i++)
    {
        p.setName(names[i]);
        list.add(p);
    }
    return list;
}
```

(assuming `Person` has a no-args constructor, a `Person(String name)` constructor, and a `setName(String name)` method). If

```
String names = {"Alice", "Bob", "Claire"};
List guests = makeGuestList(names);
```

and the list returned by Version 1 is displayed —

```
for (int i = 0; i < list.size(); i++)
    System.out.print(list.get(i) + " ");
```

— the result is, as expected:

```
Alice Bob Claire
```

Version 2, however, results in

```
Claire Claire Claire
```

because the list contains three references to the same object. Adding this object to the list does not shield it from being modified by the subsequent `setName` calls.



ArrayList implements the `get` and `set` methods very efficiently, because the values are stored in an array, which gives direct access to the element with a given index. Inserting or removing a value at the beginning or somewhere in the middle of an ArrayList is less efficient because it requires shifting the subsequent values. Adding a value may occasionally require reallocation and copying of the array, which may be time-consuming for a long list.

## 12½ . 4 Lab: Creating an Index For a Document

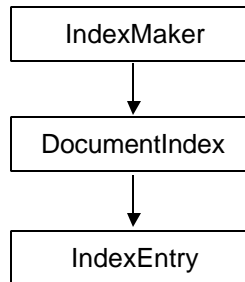


In this lab we will write a program that reads a text file and generates an index for it. All the words that occur in the text should be listed in the index in upper case in alphabetical order. Each word should be followed by the list of all line numbers that contain that word. Figure 12½ - 3 shows an example.


fish.txt	fishIndex.txt
One fish two fish Red fish Blue fish.  Black fish Blue fish Old fish New fish.  This one has a little star.  This one has a little car. Say! What a lot of fish there are.	A 12, 14, 15 ARE 16 BLACK 6 BLUE 4, 7 CAR 14 FISH 1, 2, 3, 4, 6, 7, 8, 9, 16 HAS 11, 14 LITTLE 12, 14 LOT 15 NEW 9 OF 16 OLD 8 ONE 1, 11, 14 RED 3 SAY 15 STAR 12 THERE 16 THIS 11, 14 TWO 2 WHAT 15

Figure 12½ - 3. A sample text file and its index

The *Index Maker* program consists of three classes, shown in Figure 12½ - 4.



**Figure 12½ - 4. IndexMaker classes**

The `IndexMaker` class is the main class. We have provided this class for you in the `Ch12½\IndexMaker` folder  on your student disk. Its `main` method prompts the user for the names of the input and output files (or obtains them from the command-line arguments, if supplied), opens the input file, creates an output file, reads and processes all the lines from the input file, then saves the resulting index in the output file. (`EasyReader` is used to open and read the input file, and `EasyWriter` helps create and write to the output file.)

Writing the other two classes is left to you, possibly in a team with another programmer.



An `IndexEntry` object represents one index entry. It has two fields: a word (a `String`) and a `List` of `Integer` objects that represent positions of this word in the document. Note that the `IndexEntry` class is quite general and reusable: the positions can represent line numbers, page numbers, or any other references, depending on the application. Provide a constructor for this class that takes a given word, converts it into the upper case (by calling `toUpperCase`), and saves it in a private field. The constructor should initialize the list of numbers to an empty `ArrayList`.

The `IndexEntry` class should implement `Comparable`, and the entries should be compared based on the comparison of their words using their natural order. `IndexEntry` should also have an `equals` method that compares this entry to another for equality, again based on their respective words.

Provide the `add(int num)` method that converts `num` into an `Integer` and appends it to this `IndexEntry`'s `ArrayList` of numbers, if it is not already in the list.

Finally, provide a `toString` method that generates an output string for this `IndexEntry`, in the format shown in Figure 12½ - 3.



A `DocumentIndex` object represents an index for a document, a list of index entries. Make the `DocumentIndex` class extend `ArrayList`. Provide the following two constructors that create a list of the default capacity and a list of a given capacity, respectively:

```
public DocumentIndex()  
{  
    super();  
}  
  
public DocumentIndex(int initialCapacity)  
{  
    super(initialCapacity);  
}
```

These constructors simply call the respective constructors of the superclass (`ArrayList`).

Add the following methods:

```
/**  
 * If word is in this DocumentIndex and num is in its list,  
 * does nothing; if word is in this DocumentIndex  
 * and num is not in its list, adds num to this word's  
 * IndexEntry; otherwise creates a new entry with word and  
 * num and inserts it into this index in order  
 */  
public void addWord(String word, int num)
```

and

```
/**  
 * For each word in str, adds num to this word's  
 * index entry (creating new entries when necessary)  
 */  
public void addAllWords(String str, int num)
```

The `addAllWords` method should extract all the words from a given string and add them to this `DocumentIndex`. Use the `StringTokenizer` class from the `java.util` package (see Chapter 9) to extract the words, or parse the string yourself, extracting contiguous blocks of alphanumeric characters. If you use a `StringTokenizer`, you can initialize it to skip punctuation, as follows:

```
StringTokenizer tokens = new StringTokenizer(str, " .,-;?!");
```



Also supply and use a private method

```
private void insertInOrder(IndexEntry entry)
```

that inserts a given entry into this `DocumentIndex` in alphabetical order.



Test your program thoroughly on different text data files, including an empty file, a file with blank lines, a file with multiple occurrences of a word on the same line, and a file with the same word on different lines.

## 12½.5 Summary

A Java interface specifies one or several methods, without any code. A class that implements an interface must supply all these methods. One of the commonly used library interfaces is `Comparable`. This interface specifies one method, `compareTo`, that provides a way to compare this object to another object, usually of the same class. The `compareTo` method returns a positive integer if this object is “greater than” the other, zero if they are “equal,” and a negative integer if this object is “less than” the other. What is “greater than,” “equal,” or “less than” depends on the implementation and is in effect determined by the `compareTo` method.

The same class can extend one superclass and also implement one or several interfaces.

The `java.util.ArrayList` class implements the `java.util.List` interface. An `ArrayList` automatically keeps track of the capacity of the list (the length of the allocated array) and the size of the list (the number of values in the list). The no-args constructor creates an empty list of some small default capacity; another constructor creates an empty list of a specified capacity.

The most commonly used `ArrayList` methods are shown in Figure 12½-1. The `add(obj)` method appends a value at the end of the list. The `get(i)`, `set(i, obj)`, `add(i, obj)`, and `remove(i)` methods check that `i` is from 0 to `size() - 1` (from 0 to `size()` in case of `add`) and report a run-time error, `IndexOutOfBoundsException`, if an index is out of the valid range. The `add` and `remove` methods adjust the indices of the subsequent values and the size of the list. The `contains` and `indexOf` methods help to find a value in the list.

When you get an object from an `ArrayList`, you might need to cast it into the appropriate type before you can call its methods. It is the programmer's responsibility to remember what types of objects are stored in the list.

`ArrayList`'s `add(obj)` and `add(i, obj)` methods store in the list a reference to a given object. An object can be changed after it is added to the list (unless the object is immutable). A list is allowed to hold `null` references and multiple references to the same object.

`ArrayList`'s `get(i)` and `set(i, obj)` methods are efficient because an array provides random access to its elements. The `add(i, obj)` and `remove(i)` methods may be less efficient because they require shifting of the values that follow the *i*-th value. `add` may need to reallocate and copy the whole list when the list's capacity is exceeded.

## Exercises

1. Can one interface extend another interface? Test this hypothesis. ✓
2. Consider the following interface:

```
public interface Place
{
    int distance(Object other);
}
```

Write a program that tests the following method: ✓

```
// Returns true if one of the three places is equidistant
// from the other two, false otherwise
public boolean hasMiddle(Place p1, Place p2, Place p3)
{
    return p1.distance(p2) == p1.distance(p3) ||
           p2.distance(p1) == p2.distance(p3) ||
           p3.distance(p1) == p3.distance(p2);
}
```

3. Suppose a class `Person` implements `Comparable`. A `Person` has the `getFirstName()` and `getLastName()` methods; each of them returns a `String`. Write a `compareTo` method for this class. It should compare this person to another by comparing their last names; if they are equal, it should compare their first names. The resulting order should be alphabetical order, as in a telephone directory.

4. What will happen if you try to compile and run the following code:

```
Integer i = new Integer(1);
Double d = new Double(2.0);
int diff = d.compareTo(i);
```

Will it compile? If so, what will be the result?

5. Write a method that takes an `ArrayList` and returns a new `ArrayList` in which the values are stored in reverse order. The original list should remain intact. ✓
6. Suppose an `ArrayList` holds `Comparable` objects. Write a method that removes the smallest value from such a list.
7. Implement Selection Sort for an `ArrayList` of `Comparable` objects:

```
public void selectionSort(ArrayList list)
```

Use `ArrayList`'s `get` and `set` methods; do not use the `add(i, obj)` method.

8. Implement Insertion Sort for an `ArrayList` of `Comparable` objects:

```
public ArrayList insertionSort(ArrayList list)
```

Your method should use and return a spare list, leaving the original list unchanged. Use `ArrayList`'s `get` and `add(i, obj)` methods.

9. Write and test a method

```
public void filter(List list1, List list2)
```

that removes from `list1` all objects that are also in `list2`. Your method should compare the objects using the `==` operator, not `equals`. ⚠ Hint: the `contains` and `indexOf` methods cannot be used. ⚠

**10.■** Write and test a method

```
public void removeDuplicates(List list)
```

that removes duplicate values (i.e., for which `obj1.equals(obj2)`) from the list. ≤ Hint: start with the last element and use the `indexOf` and `remove` methods. ≥

**11.■** Write a class that extends `ArrayList` and implements `Comparable`; its `compareTo` method compares two `ArrayList`s. This class assumes that the `ArrayList` holds `Comparable` objects. Two `ArrayList`s should be compared in “lexicographical” order, as follows:

- the lists are scanned in unison;
- if two different objects are found in corresponding positions, (i.e. `compareTo` for these objects is not 0), `compareTo` for the lists returns the result of `compareTo` for these objects;
- if one of the lists runs out of values, the shorter list is deemed “smaller,” and `compareTo` returns the difference between the sizes of the lists;
- if the lists have the same size and all the corresponding values match, the lists are deemed “equal.”

**12.** Can an `ArrayList` be its own element? Test this hypothesis.**13.** (a)■ Write and test a class `Polynomial` that represents a polynomial  $P(x) = a_0 + a_1x + \dots + a_nx^n$ . Use an `ArrayList` of `Doubles` of size  $n+1$  to hold the coefficients. Provide a constructor

```
public Polynomial(double[] coefficients)
```

that sets the coefficients by copying them from a given array. Make sure `Polynomial` objects are immutable. Provide one method to return the degree of the polynomial,  $n$ , and another to calculate this polynomial’s value for a given  $x$ , using the formula from exercise 14 (b) on Page 316.

- (b)♦ Write a method that multiplies this polynomial by another polynomial and returns their product.

- 14.♦** A partition of a positive integer  $n$  is its representation as a sum of positive integers  $n = p_1 + p_2 + \dots + p_k$ . Write a method that prints out all possible partitions of a given positive integer  $n$ . Consider only partitions where  $p_1 \leq p_2 \leq \dots \leq p_k$ . Hint: use an `ArrayList` of `Integers` of capacity  $n$  to represent a partition; write a recursive helper method `displayPartitions(int n, List list)`. If the sum of the values in `list` is  $n$ , just print these values. If the sum is less than  $n$ , try appending to the list another integer, in the range from the last value in the list (or 1, if the list was empty) to  $n - \text{sum}$ , and call `displayPartitions(n, list)` again. Don't forget to remove the last added value before trying the next one. ▸