ITE 221: Programming I

Ehsan Ali
Stamford International University (Thailand)
ehssan.aali@gmail.com

Semester 3/2020

Contents

Chapter 1

Week 9 - Parameters - Math Class - ArrayList

1.1 Parameters

In week 5 lecture note, we got introduced to parameters briefly as we could not write meaningful and practical programs without them. This week we will investigate parameters in greater detail.

We learned that we can define a task in a *method* and call it whenever we need that task to be done. We also learned that we can declare a *parameterized method*.

For example:

```
// The task here is writing spaces 'number' of times.
// 'number' is a parameter.
public static void writeSpaces(int number) {
  for (int i = 1; i <= number; i++) {
    System.out.print(" ");
  }
}
...
// We can call writeSpaces and set number to whatever
// value that we need have for white spaces
// For example: prints 5 spaces
writeSpaces (5);
// For example: prints 80 spaces
writeSpaces (80);
```

The parameter number is a local variable, but it gets its initial value from the call. Calling this method with the value 5 is equivalent to including the following declaration at the beginning of the writeSpaces method:

```
int number = 5;
```

You can even use an integer expression for the call:

```
// Java evaluates the expression to get the value 7 and then calls
// writeSpaces, initializing number to 7.
writeSpaces(3 * 4 - 5);
```

1.1.1 Formal Parameter versus Actual Parameter

Computer scientists use the word *parameter* broadly to mean both what appears in the method header (the *formal parameter*) and what appears in the method call (the *actual parameter*).

Formal Parameter is a variable that appears inside parentheses in the header of a method that is used to generalize the method's behavior. **Actual Parameter** is a specific value or expression that appears inside parentheses in a method call. The word "argument" is often used as a synonym for "parameter".

1.1.2 The Mechanics of Parameters

When Java executes a call on a method, it initializes the method's parameters. For each parameter, it first evaluates the expression passed as the actual parameter and then uses the result to initialize the local variable whose name is given by the formal parameter.

1.1.3 Limitations of Parameters

We've seen that a parameter can be used to provide input to a method. But while you can use a parameter to send a value into a method, you can't use a parameter to get a value out of a method.

When a parameter is set up, a local variable is created and is initialized to the value being passed as the actual parameter. The net effect is that the local variable is a copy of the value coming from the outside. Since it is a local variable, it can't influence any variables outside the method.

```
public class ParameterApp {
  public static void main(String[] args) {
    int x = 17;
    doubleNumber(x);
    System.out.println("x = " + x);
    System.out.println();

    int number = 42;
    doubleNumber(number);
    System.out.println("number = " + number);
  }

  public static void doubleNumber(int number) {
    System.out.println("Initial value = " + number);
    number = number * 2;
    System.out.println("Final value = " + number);
  }
}
```

The above code outputs:

```
Initial value = 17
Final value = 34
x = 17

Initial value = 42
Final value = 84
number = 42
```

The local manipulations of the parameter do not change these variables outside the method. The fact that variables are copied is an important aspect of parameters. On the positive side, we know that the variables are protected from change because the parameters are copies of the originals. On the negative side, it means that although parameters will allow us to send values into a method, they will not allow us to get values back out of a method.

1.1.4 Multiple Parameters

The syntax we use to declare static methods with parameters:

```
public static void <name>(<type> <name>, ..., <type> <name>) {
  <statement>;
  <statement>;
  ...
  <statement>;
}
```

This template indicates that we can declare as many parameters as we want inside the parentheses that appear after the name of a method in its header. We use commas to separate different parameters.

As an example of a method with multiple parameters, let's consider a variation of writeSpaces:

```
public static void writeChars(char ch, int number) {
  for (int i = 1; i <= number; i++) {
    System.out.print(ch);
  }
}
```

1.1.5 Overloading of Methods

You'll often want to create slight variations of the same method, passing different parameters. For example, you could have a drawBox method that allows you to specify a particular height and width, but you might also want to have a version that draws a box of default size. In other words, sometimes you want to specify these values, as in

```
drawBox(10, 50);
```

and other times you want to just draw a box with the standard height and width:

```
drawBox();
```

The drawBox is defined as below:

```
public static void writeChars(char ch, int number) {
  for (int i = 1; i <= number; i++) {
    System.out.print(ch);
  }
}

public static void drawBox(int height, int width) {
  // draw top of box
  writeChars('*', width);
  System.out.println();
```

```
  // draw middle lines
  for (int i = 1; i <= height - 2; i++) {
    System.out.print('*');
    writeChars(' ', width - 2);
    System.out.println("*");
  }
  // draw bottom of box
  writeChars('*', width);
  System.out.println();
}
```

Some programming languages require you to come up with different names for these versions, such as drawBox and drawDefaultBox. As you can imagine, coming up with new names for each variation becomes tedious. Fortunately, Java allows you to have more than one method with the same name, as long as they have different parameters. This is called *overloading*. The primary requirement for overloading is that the different methods that you define must have different *method signatures*.

Method Overloading is the ability to define two or more different methods with the same name but different method signatures. **Method Signature** is the name of a method, along with its number and type of parameters.

```
public static void drawBox(int height, int width) {
  // draw top of box
  writeChars('*', width);
  System.out.println();
  // draw middle lines
  for (int i = 1; i <= height - 2; i++) {
    System.out.print('*');
    writeChars(' ', width - 2);
    System.out.println("*");
  }
  // draw bottom of box
  writeChars('*', width);
  System.out.println();
}

// drawBox overloaded (default)
public static void drawBox() {
  // draw top of box
  int width = 80;
  int height = 25;
  writeChars('*', width);
  System.out.println();
  // draw middle lines
  for (int i = 1; i <= height - 2; i++) {
    System.out.print('*');
    writeChars(' ', width - 2);
    System.out.println("*");
  }
  // draw bottom of box
```

```
  writeChars('*', width);
31  System.out.println();
}
```

The situation gets more complicated when overloading involves the same number of parameters, but this turns out to be one of the most useful applications of overloading. For example, the *println* method is actually a series of overloaded methods. We can call *println* passing it a *String*, an *int*, a *double*, and so on. This flexibility is implemented as a series of different methods, all of which take one parameter: One version takes a *String*, another version takes an *int*, another version takes a *double*, and so on.

1.1.6 Methods That Return Values

The last few methods we've looked at have been action-oriented methods that perform some specific task. You can think of them as being like commands that you could give someone, as in "Draw a box" or "Draw a triangle." Parameters allow these commands to be more flexible, as in "Draw a box that is 10 by 20."

You will also want to be able to write methods that compute values. These methods are more like these questions: "What is the square root of 9?", "What is 3 to the power 2?".

In this case we have to declare a **return value** for our methods. It is to send a value out as the result of a method that can be used in an expression in your program. *Void* methods do not return any value.

A method can return any legal type: an int, a double, or any other type.

```
// double is the return type
2 public static double pow(double n, int p) {
  if (p == 0)
4   return 1.0;
  else {
6   double result = n;
    for (int i = 1; i < p; i++) {
8     result *= n;
    }
10  return result;
  }
12 }
```

Implementation of power function is trivial, but things become complex if we want to write the sqrt() method that returns the square root of a double number by ourselves.

Fortunately, you don't actually need to write a method for computing the square root of a number, because Java has one that is built in. The method is included in a class known as *Math* that includes many useful computing methods.

1.2 The Math Class

A great deal of predefined code, collectively known as the Java class libraries, has been written for Java. One of the most useful classes is *Math*. It includes predefined mathematical constants and a large number of common mathematical functions. The *Math* class should be available on any machine on which Java is properly installed.

1.2.1 dot notation

The *Math* class has a method called *sqrt* that computes the square root of a number. Unfortunately, you can't just call this method directly by referring to it as sqrt because it is in another class. Whenever you want to refer to something declared in another class, you use **dot notation**:

```
<class name>.<element>
```

Example:

```
public static void main(String[] args) {
  for (int i = 1; i <= 20; i++) {
    double root = Math.sqrt(i);
    System.out.println("sqrt(" + i + ") = " + root);
  }
}
```

Outputs:

```
sqrt(1) = 1.0
sqrt(2) = 1.4142135623730951
sqrt(3) = 1.7320508075688772
sqrt(4) = 2.0
sqrt(5) = 2.23606797749979
sqrt(6) = 2.449489742783178
sqrt(7) = 2.6457513110645907
sqrt(8) = 2.8284271247461903
sqrt(9) = 3.0
sqrt(10) = 3.1622776601683795
sqrt(11) = 3.3166247903554
sqrt(12) = 3.4641016151377544
sqrt(13) = 3.605551275463989
sqrt(14) = 3.7416573867739413
sqrt(15) = 3.872983346207417
sqrt(16) = 4.0
sqrt(17) = 4.123105625617661
sqrt(18) = 4.242640687119285
sqrt(19) = 4.358898943540674
sqrt(20) = 4.47213595499958
```

1.2.2 Math Constants

Use s Math.E and Math.PI for natural logarithms (2.71828...) and ratio of circumference of a circle to its diameter (3.14159...).

Table 1.1 lists some of the most useful static methods from the Math class. You can see a complete list of methods defined in the Math class by checking out the API documentation for your version of Java `https://docs.oracle.com/javase/8/docs/api/java/lang/Math.html`.

Method	Description	Example
abs	absolute value	Math.abs(–308) returns 308
ceil	ceiling (rounds upward)	Math.ceil(2.13) returns 3.0
cos	cosine (radians)	Math.cos(Math.PI) returns –1.0
exp	exponent base e	Math.exp(1) returns 2.7182818284590455
floor	floor (rounds downward)	Math.floor(2.93) returns 2.0
log	logarithm base e	Math.log(Math.E) returns 1.0
log10	logarithm base 10	Math.log10(1000) returns 3.0
max	maximum of two values	Math.max(45, 207) returns 207
min	minimum of two values	Math.min(3.8, 2.75) returns 2.75
pow	power (general exponentiation)	Math.pow(3, 4) returns 81.0
random	random value	Math.random() returns a random double value k such that $0.0 \leq k < 1.0$
round	round real number to nearest integer	Math.round(2.718) returns 3
sin	sine (radians)	Math.sin(0) returns 0.0
sqrt	square root	Math.sqrt(2) returns 1.4142135623730951
toDegrees	converts from radians to degrees	Math.toDegrees(Math.PI) returns 180.0
toRadians	converts from degrees to radians	Math.toRadians(270.0) returns 4.71238898038469

Table 1.1: Math class static methods

If you do look into the *Math* API, you'll notice that the *Math* class has several overloaded methods. For example, there is a version of the absolute value method (Math.abs) for integers and another for doubles.

1.2.3 Common Programming Error

Ignoring the Returned Value

Example:

```
sum(1000); // doesn't do anything
int answer = sum(1000); // better
```

Statement after Return

It's illegal to place other statements immediately after a return statement, because those statements can never be reached or executed.

Example:

```
public static double sum(double a, double b) {
  double s = a + b;
  return s;
  System.out.println(s); // this doesn't work
}
```

1.3 ArrayLists

One of the most fundamental data structures you will encounter in programming is a list. You'll want to store lists of words, lists of numbers, lists of names, and so on. We already have used *ArrayList* in order to create a list of lines in messageBox in LCD library. In this section we will examine *ArrayList* in greater detail.

Last week we studied Java arrays to store sequences of values, but arrays are fixed-size structures that require you to declare in advance exactly how many elements you want to store. In this chapter we'll explore a new structure, known as an *ArrayList*, that provides more flexibility than an array. An *ArrayList* is a dynamic structure with a variable length, so it can grow and shrink as the program executes.

The *ArrayList* structure uses *generic structure* which can be used to store values of different types. As a result, we will need to explore some issues related to generic structures in this section. We will also look at how to use primitive data with such structures, using what are known as the *wrapper classes*. Finally, we will demonstrate how to use the *Comparable* interface to put values of a particular type into sorted order and how to write classes that implement the *Comparable* interface.

In our daily lives, we often manipulate lists of one kind or another. For example, on social networking sites like Facebook.com, people list the bands they like. Suppose someone listed the following bands:

```
Tool, U2, Phish, Pink Floyd, Radiohead
```

To store the preceding list, you could declare an array of Strings of length 5. But what happens if you want to change the list later (say, to remove Tool and U2 from the list)?

You would have to shift values over, and you'd be left with empty array slots at the end. And what if you wanted to add to the list, so that it ended up with more than five names? You wouldn't have room to store more than five values in the original array, so you would have to construct a new array with a larger size to store the list.

Most of us think of lists as being more flexible than that. We don't want to have to worry about the kind of low-level details that come up in the course of manipulating an array. We want to be able to just say, "Add something to the list" or "Remove this value from the list," and we want the lists to grow and shrink over time as we add or remove values. Computer scientists would say that we have in mind a list abstraction that enables us to specify certain operations to be performed (add, remove) without having to worry about the details of how those operations are performed (shifting, constructing new arrays).

Java provides this functionality in a class called *ArrayList*. Internally, each *ArrayList* object uses an array to store its values. As a result, an *ArrayList* provides the same fast random access as an array. But unlike with an array, with an *ArrayList* you can make simple requests to add or remove values, and the *ArrayList* takes care of all of the details for you: If you add values to the list it makes the array bigger, and if you remove values it handles any shifting that needs to be done.

Remember that you can declare arrays of different types. If you want an array of *int* values, you declare a variable of type *int*[]. For an array of *String* values, you use the type *String*[]. This is a special syntax that works just for arrays, but the *ArrayList* class has almost the same flexibility. If you read the API documentation for *ArrayList*, you'll see that it is actually listed as *ArrayList<E>*. This is an example of a *generic class* in Java.

1.3.1 Generic Class (Generic)

A class such as *ArrayList<E>* that takes a type parameter to indicate what kind of values it will use is a generic class.

The "*E*" in *ArrayList<E>* is short for "Element," and it indicates the type of elements that will be included in the *ArrayList*. Generic classes are similar to parameterized methods.

You would use *ArrayList<String>* to store a list of *String*s, *ArrayList<Point>* to store a list of *Point*s, *ArrayList<Color>* to store a list of *Color*s, and so on. Notice that you would never actually declare something to be of type *ArrayList<E>*. As with any parameter, you have to replace the E with a specific value to make it clear which of the many possible *ArrayList* types you are using.

1.3.2 Basic ArrayList Operations

For example, you would construct an *ArrayList* of Strings as follows:

```
1 ArrayList<String> list = new ArrayList<String>();
```

This code constructs an empty *ArrayList<String>*. This syntax is complicated, but it will be easier to remember if you keep in mind that the <String> notation is actually part of the type: This isn't simply an *ArrayList*, it is an *ArrayList<String>* (often read as "an *ArrayList* of *String*"). Notice how the type appears when you declare the variable and when you call the constructor:

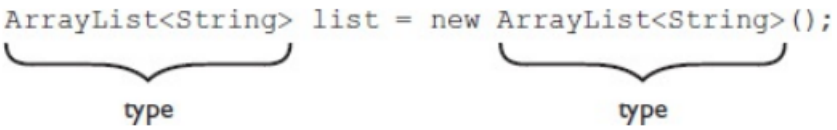

Figure 1.1:

If you think in terms of the type being ArrayList¡String¿, you'll see that this line of code isn't all that different from the code used to construct an object like a LCD:

```
1   LCD lcd = new LCD();
```

It can be cumbersome to list the element type *<String>* twice, so Java version 7 has introduced a new shorter syntax for declaring collections called the "diamond operator" where by the element type may be omitted on the right side of the statement and replaced by <>:

```
1 ArrayList<String> list = new ArrayList<>();
```

Once you've constructed an *ArrayList*, you can add values to it by calling the add method:

```
1 ArrayList<String> list = new ArrayList<String>();
  list.add("Tool");
3 list.add("Phish");
  list.add("Pink Floyd");
```

Unlike with simple arrays, printing an *ArrayList* is straightforward because the ArrayList class overrides Java's *toString* method:

```
  ArrayList<String> list = new ArrayList<String>();
2 list.add("Tool");
  list.add("Phish");
4 list.add("Pink Floyd");
  System.out.println("list = " + list);
```

outputs:

```
list = [Tool, Phish, Pink Floyd]
```

The *ArrayList* class also provides an overloaded version of the add method for adding a value at a particular index in the list. It preserves the order of the other list elements, shifting values to the right to make room for the new value. This version of add takes two parameters: an index and a value to insert. *ArrayList*s use zero-based indexing, just as arrays and Strings do:

```
System.out.println("before list = " + list);
list.add(1, "U2");
System.out.println("after list = " + list);
```

outputs:

```
before list = [Tool, Phish, Pink Floyd]
after list = [Tool, U2, Phish, Pink Floyd]
```

ArrayList also has a method for removing a value at a particular index. The remove method also preserves the order of the list by shifting values to the left to fill in any gap:

```
System.out.println("before remove list = " + list);
list.remove(0);
list.remove(1);
System.out.println("after remove list = " + list)
```

outputs:

```
before remove list = [Tool, U2, Phish, Pink Floyd]
after remove list = [U2, Pink Floyd]
```

If you want to find out the number of elements in an *ArrayList*, you can call its *size* method. If you want to obtain an individual item from the list, you can call its *get* method, passing it a specific index:

```
int sum = 0;
for (int i = 0; i < list.size(); i++) {
  String s = list.get(i);
  sum += s.length();
}
System.out.println("Total of lengths = " + sum);
```

Calling add and remove can be expensive in terms of time because the computer has to shift the values around. If all you want to do is to replace a value, you can use a method called set, which takes an index and a value and replaces the value at the given index with the given value, without doing any shifting:

```
list.set(0, "BlackCats");
```

To remove all of the values from the list you can call the *clear* method of the *ArrayList*.

1.3.3 *ArrayList* Searching Methods

Once you have built up an *ArrayList*, you might be interested in searching for a specific value in the list. The *ArrayList* class provides several mechanisms for doing so.

If you just want to know whether or not something is in the list, you can call the *contains* method, which returns a Boolean value:

name.txt File sample:

```
Maria Derek Erica
Livia Jack Anita
Kendall Maria Livia Derek
Jamie Jack
Erica
```

```
// removes duplicates from a list
Scanner input = new Scanner(new File("names.txt"));
ArrayList<String> list = new ArrayList<String>();
while (input.hasNext()) {
  String name = input.next();
  if (!list.contains(name)) {
    list.add(name);
  }
}
System.out.println("list = " + list);
```

outputs:

```
list = [Maria, Derek, Erica, Livia, Jack, Anita, Kendall, Jamie]
```

Sometimes it is not enough to know that a value appears in the list. You may want to know exactly where it occurs.

You can find out the location of a value in the list by calling the *indexOf* method. The *indexOf* method takes a particular value and returns the index of the first occurrence of the value in the list. If it doesn't find the value, it returns –1:

```
public static void replace(ArrayList<String> list,
    String target, String replacement) {
  int index = list.indexOf(target);
  if (index >= 0) {
    list.set(index, replacement);
  }
}
```

There is also a variation of *indexOf* known as *lastIndexOf*. As its name implies, this method returns the index of the last occurrence of a value. There are many situations where you might be more interested in the last occurrence rather than the first occurrence. For example, if a bank finds a broken automated teller machine, it might want to find out the name and account number of the last customer to use that machine.

Table 1.2 summarizes the *ArrayList* search methods.

Method	**Description**	***ArrayList<String>* Example**
.contains(value)	returns true if the given value appears in the list	list.contains(”hello”)
.indexOf(value)	returns the index of the first occurrence of the given value in the list (-1 if not found)	list.indexOf(”world”)
.lastIndexOf(value)	returns the index of the last occurrence of the given value in the list (-1 if not found)	list.lastIndexOf(”hello”)

Table 1.2: Math class static methods

The Java API documentation can be found here: `https://docs.oracle.com/javase/8/docs/api/java/util/ArrayList.html`.

Task 1: Search engines like Google ignore the *stop words* in users' queries. The idea is that certain words like “a” and “the” appear so often that they aren't worth indexing. Google won't disclose the exact list of words it uses, although a few examples are listed on the web site and people have speculated about what they think is on the list. Google's full list of stop words is believed to have at least 35 entries, but we'll settle for 15 of the most obvious choices. Write a Java program that reads a file called proverbs.txt that contains the following text:

```
The most famous Chinese proverbs:
0. A team of horses will struggle to chase down a spoken word.
1. If you don't do stupid things you won't end up in tragedy.
2. If small holes aren't fixed, then big holes will bring hardship.
3. It's better to walk thousands of miles than to read thousands of
   ↪ books.
4. A good name is better than a good face.
5. A bird in the hand is worth than two in the bush.
6. No man is a perfect man; no gold is sufficiently bare.
7. Man cannot be judged by looks; seas cannot be measured by cup.
8. Failure is the mother of success.
9. It just needs hard work to grind an iron rod into a needle.
```

It then constructs a list of stop words: a, be, by, how, in, is, it, of, on, or, that, the, this, to, why and then reads the file word by word, printing every word that is not a stop word. To avoid issues of case, the stop words are all in lowercase and the call on contains is passed a lowercase version of each word from the input file.

```
// This program constructs a list of stop words and echoes
// Hamlet's famous speech with the stop words removed.

import java.util.*;
import java.io.*;

public class StopWords {
  public static void main(String[] args)
    throws FileNotFoundException {
    // build the list of stop words
    ArrayList<String> stopWords = new ArrayList<String>();
    stopWords.add("a");
    stopWords.add("be");
```

```
    stopWords.add("by");
    stopWords.add("how");
    stopWords.add("in");
    stopWords.add("is");
    stopWords.add("it");
    stopWords.add("of");
    stopWords.add("on");
    stopWords.add("or");
    stopWords.add("that");
    stopWords.add("the");
    stopWords.add("this");
    stopWords.add("to");
    stopWords.add("why");

    // process the file, printing all but stop words
    Scanner input = new Scanner(new File("proverbs.txt"));
    while (input.hasNext()) {
      String next = input.next();
      if (!stopWords.contains(next.toLowerCase())) {
        System.out.print(next + " ");
      }
    }
  }
}
```

1.3.4 Using the For-Each Loop with *ArrayLists*

We discussed in previous lecture not that we can use a *for-each* loop to iterate over the elements of an array. You can do the same with an *ArrayList*:

```
ArrayList<String> list = new ArrayList<String>();
list.add("apple");
list.add("orange");
list.add("kiwi");
int sum = 0;
for (String s : list) {
  sum += s.length();
}
System.out.println("Total of lengths = " + sum);
```

Because the for-each loop has such a simple syntax, you should use it whenever you want to process each value stored in a list sequentially. You will find, however, that the for-each loop is not appropriate for more complex list problems. For example, there is no simple way to skip around in a list using a *for-each* loop. You must process the values in sequence from first to last. Also, you cannot modify the list while you are iterating over it.

```
// this doesn't work
for (String s : words) {
  System.out.println(s);
```

```
4   words.remove(0);
}
```

This code prints a String from the list and then attempts to remove the value at the front of the list. When you execute this code, the program halts with a *ConcurrentModificationException*. Java is letting you know that you are not allowed to iterate over the list and to modify the list at the same time (concurrently).

1.3.5 Wrapper Classes

So far, all of the *ArrayList* examples we have studied have involved *ArrayList*s of *String* objects. What if you wanted to form a list of integers? Given that *ArrayList<E>* is a generic class, you'd think that Java would allow you to define an *ArrayList<int>*, but that is not the case. The E in *ArrayList<E>* can be filled in with any object or reference type (i.e., the name of a class). The primitive types (e.g., *int*, *double*, *char*, and *boolean*), cannot be used as type parameters for an *ArrayList*. Instead, Java defines a series of wrapper classes that allow you to store primitive data as objects.

Wrapper Class is a class that "wraps" (stores) primitive data as an object.

Primitive types are not objects, so we can't use values of type *int* in an object context. To allow such use, we must wrap up each int into an object of type *Integer*. *Integer* objects are very simple. They have just one field: an *int* value. When we construct an *Integer*, we pass an *int* value to be wrapped; when we want to get the *int* back, we call a method called *intValue* that returns the *int*.

```
1 int x = 38;
  Integer y = new Integer(38);
```

This code leads to the following situation in memory:

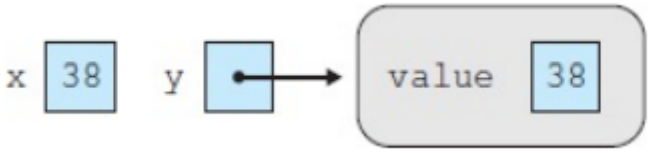

Figure 1.2:

Primitive data is stored directly, so the variable x stores the actual value 38. Objects, by contrast, are stored as references, so the variable y stores a reference to an object that contains 38.

If we later want to get the 38 out of the object (to unwrap it), we call the method *intValue*:

```
1 int number = y.intValue();
```

The wrapper classes are of particular interest in this section because when you use an*ArrayList<E>*, the *E* needs to be a reference type. You can't form an *ArrayList<int>*, but you can form an *ArrayList<Integer>*.

1.3.6 Boxing

Boxing is an automatic conversion from primitive data to a wrapped object of the appropriate type (e.g., an *int* boxed to form an *Integer*).

Similarly, you don't have to do anything special to unwrap an *Integer* to get the *int* inside. You could write code like the following:

```
int product = list.get(0) * list.get(1);
```

This code multiplies two values from the *ArrayList<Integer>* and stores the result in a variable of type *int*. The calls on get will return an *Integer* object, so normally these values would be incompatible. However, because Java understands the relationship between *int* and *Integer* it will unwrap the *Integer* objects for you and give you the *int* values stored inside. This process is known as *unboxing*.

1.3.7 Unboxing

Unboxing is an automatic conversion from a wrapped object to its corresponding primitive data (e.g., an *Integer* unboxed to yield an *int*).

Common wrapper classes:

- int → Integer
- double → Double
- char → Character
- boolean → Boolean

1.3.8 The Comparable Interface

Task 2: The method *Collections.sort* can be used to sort an *ArrayList*. It is part of the java.util package. Write a Java program that uses *Collections.sort* in order to sort the following list of *String*s four, score, and, seven, years, ago:

```
1  // Constructs an ArrayList of Strings and sorts it.
2
3  import java.util.*;
4
5  public class SortExample {
6    public static void main(String[] args) {
7      ArrayList<String> words = new ArrayList<String>();
8      words.add("four");
9      words.add("score");
10     words.add("and");
11     words.add("seven");
12     words.add("years");
13     words.add("ago");
14
15     // show list before and after sorting
16     System.out.println("before sort, words = " + words);
17     Collections.sort(words);
18     System.out.println("after sort, words = " + words);
19   }
20 }
```

If you try to make a similar call to an *ArrayList<LCD>*, or *ArrayList<Point>* you will find that the program does not compile. Why is it possible to sort a list of *String* objects but not a list of *LCD* or *Point*

objects? The answer is that the String class implements the Comparable interface, while the Point class does not.

1.3.9 Natural Ordering and *compareTo*

In order to sort data we need to have a well-defined *comparison function* that indicates the relationship between any pair of values.

Comparison function is a well-defined procedure for deciding, given a pair of values, the relative order of the two values (less than, equal to, or greater than).

Natural ordering is the order imposed on a type by its comparison function. Not all types have natural orderings because not all types have comparison functions.

For example, in this chapter we have been exploring how to construct a variety of *ArrayList* objects. How would you compare two *ArrayList* objects to determine whether one is less than another? What would it mean for one *ArrayList* to be less than another? You might decide to use the lengths of the lists to determine which one is less, but what would you do with two *ArrayList* objects of equal length that store different values? You wouldn't want to describe them as "equal." There is no agreed upon way of ordering *ArrayList*, and therefore there is no comparison function for this type. As a result, we say that the *ArrayList* type does not have a natural ordering.

Java has a convention for indicating the natural ordering of a type. Any type that has such an ordering should implement the Comparable interface:

```
public interface Comparable<T> {
  public int compareTo(T other);
}
```

This interface provides a second example of a generic type in Java. In the case of *ArrayList*, Java uses the letter "E," which is short for "Element." In the case of *Comparable*, Java uses the letter "T," which is short for "Type."

The *compareTo* method is the comparison function for the type. The convention for *compareTo* is that an object should return one of the following results:

- A negative number to indicate a less-than relationship
- 0 to indicate equality
- A positive number to indicate a greater-than relationship

We know how to compare int values to determine their relative order, so it is not surprising that the Integer class implements the Comparable interface:

```
Integer x = 7;
Integer y = 42;
Integer z = 7;
System.out.println(x.compareTo(y));
System.out.println(x.compareTo(z));
System.out.println(y.compareTo(x));
```

Outputs:

```
-1
0
1
```

The values -1, 0, and 1 are the standard values returned, but the *compareTo* method is not required to return these specific values:

```
String x = "hello";
String y = "world";
String z = "hello";
System.out.println(x.compareTo(y));
System.out.println(x.compareTo(z));
System.out.println(y.compareTo(x));
```

outputs:

```
-15
0
15
```

Instead of -1 and 1, we get -15 and 15. You don't really need to know where these numbers come from the only important fact is whether they are negative or positive but for those of you who are curious, the -15 and 15 represent the distance between the positions of the characters 'h' and 'w' in type char. 'w' appears 15 positions later than 'h'.

The important thing to remember is that "less-than" relationships are indicated by a negative number and "greater-than" relationships are indicated by a positive number.

Table 1.3 shows the compare value summary for both primitive data and objects.

Relationship	**Primitive data (int, double, etc.)**	**Objects (Integer, String, etc.)**
less than	if (x < y) {...}	if (x.compareTo(y) < 0) {...}
less than or equal to	if (x <= y) {...}	if (x.compareTo(y) <= 0) {...}
equal to	if (x == y) {...}	if (x.compareTo(y) == 0) {...}
not equal to	if (x ! = y) {...}	if (x.compareTo(y) ! = 0) {...}
greater than	if (x > y) {...}	if (x.compareTo(y) > 0) {...}
greater than or equal to	if (x >= y) {...}	if (x.compareTo(y) >= 0) {...}

Table 1.3: Comparing Values Summary

1.3.10 Implementing the Comparable Interface

Many of the standard Java classes, such as *String*, implement the *Comparable* interface. You can have your own classes implement the interface as well. Implementing the *Comparable* interface will open up a wealth of off-the-shelf programming solutions that are included in the Java class libraries. For example, there are built-in methods for sorting lists and for speeding up searches.

Task 3: Add *Comparable* Interface to LCD class so we can compare LCDs based on their dimension.

Below is the original LCD class declaration:

Listing 1.1: LCD.java

```
public class LCD {
  int width;
  int height;
  ...

  // Default constructor
```

```
  public LCD () {
    this.width = 80;
    this.height = 25;
  }

  public LCD (int width, int height) {
    this.width = width;
    this.height = height;
  }
  ...
}
```

We add *Comparable* Interface:

Listing 1.2: LCD.java

```
public class LCD implements Comparable<LCD> {
  int width;
  int height;
  ...

  public interface Comparable<T> {
    public int compareTo(T other_lcd);
  }

  public int compareTo(LCD other_lcd) {
    int dimension = width * height;
    int other_dimension = other_lcd.width * other_lcd.height;
    if (dimension < other_dimension) {
      return -1;
    } else if (dimension == other_dimension) {
      return 0;
    } else { // (dimension > other_dimension)
      return 1;
    }
  }

  // Default constructor
  public LCD () {
    this.width = 80;
    this.height = 25;
  }

  public LCD (int width, int height) {
    this.width = width;
    this.height = height;
  }
  ...
}
```

Now we are able to compare LCDs, for example:

Listing 1.3: IoTApp.java

```
public class IoTApp {
  public static void main(String[] args) {
    LCD lcd1 = new LCD(80, 25);
    LCD lcd2 = new LCD(90, 25);
    LCD lcd3 = new LCD(40, 25);
    LCD lcd4 = new LCD(100, 10);

    System.out.println(lcd1.compareTo(lcd2));
    System.out.println(lcd2.compareTo(lcd3));
    System.out.println(lcd3.compareTo(lcd4));
}
```

Outputs:

```
-1
1
0
```

Task 4: Use ArrayLists to develop a program that will read two different text files and compare their vocabulary. Determine the set of words used in each file and compute the overlap between them. Researchers in the humanities often perform such comparisons of vocabulary in selections of text to answer questions like, "Did an author of book A actually write book B?"

The above task is complex so develop the program in the following stages:

1. The first version will read the two files and report the unique words in each.
2. The second version will also compute the overlap between the two files (i.e., the set of words that appear in both files).
3. The third version will read from large text files and will perform some analysis of the results.

Hints and Some Efficiency Considerations:

Many of the early programs in this book involved fairly simple computations and fairly small data sets. As you start writing more complex programs, you'll find that you have to worry about programs running slowly because they are performing complex computations or handling large amounts of data as it is the case of task4.

For example, in the program you are about to write, you have to read in a file and come up with a list of the words from the file that doesn't have any duplicates. One approach would be to test each word as we read it in to see if it is in the list, as described in the following pseudocode:

```
list = new empty list.
while (more words to process) {
  word = next word from file.
  if (list does not contain word) {
    add word to list.
  }
}
```

The problem with this approach is that it would require us to call the *ArrayList* method called *contains* each time the program executes the loop. It turns out that the contains method can be fairly expensive to call in terms of time. To find out whether a particular value is in the list, the method has to go through each different value in the list. So as the list becomes larger and larger, it becomes more and more expensive to search through it to see if it contains a particular word.

We will run into a similar problem when we get to the second version of the program and have to compute the overlap between the two lists. The simplest way to compute the overlap would be to write a method like this:

```
overlap = new empty list.
for (each word in list1) {
  if (word is in list2) {
    add word to overlap.
  }
}
```

This approach will again require calling the contains method for a list that could potentially be very large. If both lists are large, then the approach will run particularly slowly.

Both of these potential bottlenecks can be addressed by dealing with sorted lists. In a sorted list of words, all of the duplicates are grouped together, which makes them easier to spot. And looking for the overlap between two sorted lists is easier than looking for overlap in two lists that are not ordered.Of course, sorting isn't cheap either. It takes a nontrivial amount of time to sort a list. But if we can manage to sort the list just once, it will turn out to be cheaper than making all of those calls on the *contains* method.

Instead of trying to eliminate the duplicates as we read the words, we can just read all of the words directly into the list. That way we won't make any expensive calls on the contains method. After we have read everything in, we can put the list into sorted order. When we do that, all of the duplicates will appear right next to one another, so we can fairly easily get rid of them.

Reading all of the words into the list and then eliminating duplicates will require more memory than eliminating the duplicates as we go, but it will end up running faster because the only expensive operation we will have is the sorting step.**This is a classic tradeoff between running time and memory that comes up often in computer science.** We can make programs run faster if we're willing to use more memory or we can limit memory if we don't mind having the program take longer to run.

The task of eliminating duplicates also brings up an efficiency consideration. One obvious approach would be the following:

```
for (each word in list) {
  if (word is a duplicate) {
    remove word from list.
  }
}
```

It turns out that remove is another expensive operation. A better approach is to simply build up a new list that doesn't have duplicates:

```
result = new empty list.
for (each word in list) {
  if (word is not a duplicate) {
    add word to result.
  }
}
```

This code runs faster because the method that adds a word at the end of the list runs very fast compared with the method that removes a word from the middle of the list.

```
// This program reads two text files and compares the
// vocabulary used in each.

import java.util.*;
import java.io.*;

public class Vocabulary3 {
  public static void main(String[] args)
    throws FileNotFoundException {
    Scanner console = new Scanner(System.in);
     giveIntro();

    System.out.print("file #1 name? ");
    Scanner in1 = new Scanner(new File(console.nextLine()));
    System.out.print("file #2 name? ");
    Scanner in2 = new Scanner(new File(console.nextLine()));
    System.out.println();

    ArrayList<String> list1 = getWords(in1);
    ArrayList<String> list2 = getWords(in2);
    ArrayList<String> common = getOverlap(list1, list2);

    reportResults(list1, list2, common);
    }
  // post: reads words from the Scanner, converts them to
  // lowercase, returns a sorted list of unique words
  public static ArrayList<String> getWords(Scanner input) {
    // ignore all but alphabetic characters and apostrophes
    input.useDelimiter("[a-zA-Z']+");
    // read all words and sort
    ArrayList<String> words = new ArrayList<String>();
    while (input.hasNext()) {
      String next = input.next().toLowerCase();
      words.add(next);
    }
    Collections.sort(words);
    // add unique words to new list and return
    ArrayList<String> result = new ArrayList<String>();
    if (words.size() > 0) {
    result.add(words.get(0));
    for (int i = 1; i < words.size(); i++) {
      if (!words.get(i).equals(words.get(i - 1))) {
        result.add(words.get(i));
        }
      }
    }
```

```
    return result;
  }
  // pre : list1 and list2 are sorted and have no duplicates
  // post: constructs and returns an ArrayList containing
  // the words in common between list1 and list2
  public static ArrayList<String> getOverlap(ArrayList<String> list1,
      ArrayList<String> list2) {
    ArrayList<String> result = new ArrayList<String>();
    int i1 = 0;
    int i2 = 0;
    while (i1 < list1.size() && i2 < list2.size()) {
      int num = list1.get(i1).compareTo(list2.get(i2));
      if (num == 0) {
        result.add(list1.get(i1));
        i1++;
        i2++;
      } else if (num < 0) {
        i1++;
      } else { // num > 0
        i2++;
      }
    }
    return result;
  }
  // post: explains program to user
  public static void giveIntro() {
   System.out.println("This program compares two text files");
   System.out.println("and reports the number of words in");
   System.out.println("common and the percent overlap.");
   System.out.println();
  }
  // pre : common contains overlap between list1 and list2
  // post: reports statistics about lists and their overlap
  public static void reportResults(ArrayList<String> list1,
      ArrayList<String> list2, ArrayList<String> common) {
    System.out.println("file #1 words = " + list1.size());
    System.out.println("file #2 words = " + list2.size());
    System.out.println("common words = " + common.size());
    double pct1 = 100.0 * common.size() / list1.size();
    double pct2 = 100.0 * common.size() / list2.size();
    System.out.println("% of file 1 in overlap = " + pct1);
    System.out.println("% of file 2 in overlap = " + pct2);
  }
}
```

Outputs:

```
This program compares two text files
and reports the number of words in
common and the percent overlap.
file #1 name? bookA.txt
file #2 name? bookB.txt
file #1 words = 4874
file #2 words = 4281
common words = 2108
% of file 1 in overlap = 43.24989741485433
% of file 2 in overlap = 49.24083158140621
```