ITE 221: Programming I

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

Semester 3/2020

Contents

Chapter 1

Week 8 - File Processing - Arrays

1.1 File Processing

We discussed how to construct a *Scanner* object to read input from the console. Now we will look at how to construct *Scanner* objects to read input from files. The idea is fairly straightforward, but Java does not make it easy to read from input files. This is unfortunate because many interesting problems can be formulated as file-processing tasks. Many introductory computer science classes have abandoned file processing altogether and left the topic for the second course because it is considered too advanced for novices.

There is nothing intrinsically complex about file processing, but Java was not designed for it and the designers of Java have not been particularly eager to provide a simple solution. They did, however, introduce the *Scanner* class as a way to simplify some of the details associated with reading files. The result is that file reading is still awkward in Java, but at least the level of detail is manageable.

Before we start writing file-processing programs, we have to explore some issues related to Java exceptions. Remember that exceptions are errors that halt the execution of a program. In the case of file processing, trying to open a file that doesn't exist or trying to read beyond the end of a file generates an exception.

1.1.1 File-Reading Basics

Data

People are fascinated by data. Today, every person with an Internet connection has access to a vast array of databases containing information about every facet of our existence. Visit `https://bigdata-madesimpl.com/70-amazing-and-free-data-sources-for-data-visualization/` to see a sample of data available to us.

File

When you store data on your own computer, you store it in a **file**.

A file is a collection of information that is stored on a computer and assigned a particular name. Every file has a *name* and an *extension*. The extension is a special suffix that indicates the kind of data the file contains or the format in which it has been stored.

Table 1.1 lists some common file extensions.

Extension	Description
.txt	text file
.java	Java source code file
.class	compiled Java bytecode file
.doc	Microsoft Word file
.xls	Microsoft Excel file
.pdf	Adobe Portable Document File
.mp3	audio file
.jpg	image file
.zip	compressed archive
.html	hypertext markup language files (most often web pages)
.exe	executable file

Table 1.1: Logical Operators

Text and Binary Files

Files can be classified into **text** files and **binary** files depending on the format that is used. Text files can be edited using simple text editors. Binary files are stored using an internal format that requires special software to process. Text files are often stored using the .txt extension, but other file formats are also text formats, including .java and .html files.

To access a file from inside a Java program, you need to construct an internal object that will represent the file. The Java class libraries include a class called *File* that performs this duty.

File Class

You construct a *File* object by passing in the name of a file, as in the following line of code:

```
File f = new File("sample.txt");
```

Once you've constructed the object, you can call a number of methods to manipulate the file.

Task 1: Create an IntelliJ IDEA project named *FileApp*. Create sample.txt text file with the following content inside the project folder:

```
Well, this is a simple text file.
This is the 2nd line.
Let us have space after this line:

Ummm, some numbers: 1, 2, 3, 4, ..
Some stars: *****
Some dashes: ----
I guess this is line number 8.
Line number 9.
Bye !
```

Run the program shown in Listing 1.1. It calls a method that determines whether a file exists, whether it can be read, what its length is (i.e., how many characters are in the file), and what its absolute path is (i.e., where it is stored on the computer).

Listing 1.1: FileApp.java

```java
// Report some basic information about a file.

import java.io.File;

public class FileInfo {
  public static void main(String[] args) {
    File f = new File("sample.txt");
    System.out.println("exists returns " + f.exists());
    System.out.println("canRead returns " + f.canRead());
    System.out.println("length returns " + f.length());
    System.out.println("getAbsolutePath returns "
      + f.getAbsolutePath());
    }
}
```

Useful methods of File class is shown in Table 1.2.

Method	Description
canRead()	Whether or not this file exists and can be read
delete()	Deletes the given file
exists()	Whether or not this file exists on the system
getAbsolutePath()	The full path where this file is located
getName()	The name of this file as a String, without any path attached
isDirectory()	Whether this file represents a directory/folder on the system
isFile()	Whether this file represents a file (nonfolder) on the system
length()	The number of characters in this file
renameTo(file)	Changes this file's name to the given file's name

Table 1.2: Logical Operators

1.1.2 Reading a File with a Scanner

The Scanner class that we have been using is flexible in that Scanner objects can be attached to many different kinds of input.

A scanner can be attached to 1) Keyboard 2) File 3) Network:

```java
// Scanner object attached to keyboard (console input):
Scanner console = new Scanner(System.in);

// Scanner object attached to File object representing sample.txt
// text file
File f = new File("sample.txt");
Scanner input = new Scanner(f);


// Shortened code: (object f is not necessary)
// Not that this weill not compile!
// As we have an unhandled exception.
Scanner input = new Scanner(new File("sample.txt"));
```

The last line of the above code won't compile as there is an unhandled exception. Remember that exceptions are errors that prevent a program from continuing normal execution. In this case the compiler is worried that it might not be able to find a file called sample.txt. What is it supposed to do if that happens? It won't have a file to read from, so it won't have any way to continue executing the rest of the code. If the program is unable to locate the specified input file, it will generate an error by throwing what is known as a *FileNotFoundException*. This particular exception is known as a *checked exception*.

1.1.3 Checked Exception

A **checked exception** is an exception that must be caught or specifically declared in the header of the method that might generate it.

Because *FileNotFoundException* is a checked exception, you can't just ignore it. Java provides a construct known as the *try/catch* statement for handling such errors (will be described later), but it allows you to avoid handling this error as long as you clearly indicate the fact that you aren't handling it. All you have to do is include a *throws* clause in the header for the main method to clearly state the fact that your main method might generate this exception.

1.1.4 throws Clause

A **throws clause** is a declaration that a method will not attempt to handle a particular type of exception.

Here's how to modify the header for the main method to include a throws clause indicating that it may throw a FileNotFoundException:

```
public static void main(String[] args)
  throws FileNotFoundException {
    Scanner input = new Scanner(new File("sample.txt"));
    ...
}
```

After this modification, the program compiles.

Task 2: Write a Java program to count the number of words in sample.txt input file.

Listing 1.2: FileApp.java

```
// Task 2: Count words in a text file.

import java.io.File;
import java.io.FileNotFoundException;
import java.util.Scanner;

public class FileApp {
  public static void main(String[] args)
  throws FileNotFoundException {

    Scanner input = new Scanner(new File("sample.txt"));

    int count = 0;
    while (input.hasNext()) {
      String word = input.next();
      count++;
```

```
    }

    System.out.println("total words = " + count);
  }
}
```

1.1.5 Token-Based Processing

One way to process a file is token by token.

The primary token-reading methods for the Scanner class are:

1. *nextInt* for reading an int value
2. *nextDouble* for reading a double value
3. *next* for reading the next token as a String

Task 3: Write a Java program that reads 5 double numbers from a file and prints their sum.

Listing 1.3: FileApp.java

```
// Task 3: Read 5 numbers and prints the sum
//
/* sample.txt file content:
308.2 14.9 7.4
2.8
3.9 4.7 -15.4
2.8
*/

import java.io.File;
import java.io.FileNotFoundException;
import java.util.Scanner;

public class FileApp {
  public static void main(String[] args)
  throws FileNotFoundException {

    Scanner input = new Scanner(new File("sample.txt"));

    double sum = 0.0;
    for (int i = 1; i <= 5; i++) {
      double next = input.nextDouble();
      System.out.println("number " + i + " = " + next);
      sum += next;
    }
    System.out.println("Sum = " + sum);
  }
}
```

You can use *hasNextDouble* to test whether there is such a value to read as shown in Listing 1.4

Listing 1.4: FileApp.java

```
// Task 3: Read unlimited numbers and prints the sum
//
/* sample.txt file content:
308.2 14.9 7.4
2.8
3.9 4.7 -15.4
2.8
*/

import java.io.File;
import java.io.FileNotFoundException;
import java.util.Scanner;

public class FileApp {
  public static void main(String[] args)
  throws FileNotFoundException {

    Scanner input = new Scanner(new File("sample.txt"));

    double sum = 0.0;
    int count = 0;
    while (input.hasNextDouble()) {
      double next = input.nextDouble();
      count++;
      System.out.println("number " + count + " = " + next);
      sum += next;
    }

    System.out.println("Sum = " + sum);
  }
}
```

1.1.6 Structure of Files and Consuming Input

We think of text as being two-dimensional, like a sheet of paper, but from the computer's point of view, each file is just a one-dimensional sequence of characters. For example, consider the file numbers.dat:

```
308.2 14.9 7.4
2.8


3.9 4.7 -15.4
2.8
```

We think of this as a six-line file. However, the computer views the file differently. When you typed the text in this file, you hit the Enter key to go to a new line. This key inserts special "new line" characters

in the file. You can annotate the file with \n characters to indicate the end of each line:

```
308.2 14.9 7.4\n
2.8\n
\n
\n
3.9 4.7 -15.4\n
2.8\n
```

Therefore, computer sees this file like this:

```
308.2 14.9 7.4\n 2.8\n \n \n 3.9 4.7 -15.4\n 2.8\n
```

1.1.7 InputMismatchException

It's easy to write code that accidentally reads the wrong kind of data. If we have data file with the following content:

```
308.2 14.9 7.4
hello
2.8
3.9 4.7 -15.4
2.8
```

And we need to read the numbers as double then the first line of the file contains three numbers that the program will read properly. But when it attempts to read a fourth number, the computer finds that the next token in the file is the text "hello". This token cannot be interpreted as a double, so the program generates an exception.

1.1.8 Input Cursor

Input cursor is a pointer to the current position in an input file.

When a *Scanner* object is first constructed, the cursor points to the beginning of the file. But as you perform various *next* operations, the cursor moves forward. Figure 1.1 shows the initial input cursor.

Figure 1.1: Initial input cursor.

After the first call on nextDouble the cursor will be positioned in the middle of the first line, after the token "308.2" as shown in Figure 1.2.

Figure 1.2: Input cursor moves forward after *next* method is called.

We refer to this process as *co-nsuming input*.

1.1.9 Consuming Input

Consuming input is referred to moving the input cursor forward past some input.

The process of consuming input doesn't actually change the file, it just changes the corresponding Scanner object so that it is positioned at a different point in the file.

The first call on nextDouble consumes the text "308.2" from the input file and leaves the input cursor positioned at the first character after this token.

Notice that this leaves the input cursor positioned at a space. When the second call is made on nextDouble, the Scanner first skips past this space to get to the next token, then consumes the text "14.9" and leaves the cursor positioned at the space that follows it.

Scanner object simply skips past any leading whitespace characters (spaces, tabs, new line characters) until it finds an actual token.

If you attempted to call nextDouble again after there is no more tokens, it would throw a *NoSuchElementException* because there are no more tokens left to process.

1.1.10 Paths and Directories

Files are grouped into folders, also called directories. Directories are organized in a hierarchy, starting from a root directory at the top. For example, most Windows machines have a disk drive known as C:.

File Path

A description of a file's location on a computer, starting with a drive and including the path from the root directory to the directory where the file is stored.

Example: "D:\workspace\IDEA\FileProject\sample.txt"

When you just use "sample.txt" as an argument to File object then Java assumes a *relative* file path. It means, it looks in the *current directory* to find the file.

1.1.11 Current Directory (Working Directory)

Current directory is the directory that Java uses as the default when a program uses a simple file name. The default directory varies with the Java environment you are using. In most environments, the current directory is the one in which your program appears. In IntelliJ IDEA, go to "Run - Edit Configuration" and set the "Working Directory".

You can also use a fully qualified, or complete, file name (sometimes called an *absolute* file path). However, this approach works well only when you know exactly where your file is going to be stored on your system:

```
Scanner input = new Scanner(
  new File("D:/workspace/IDEA/FileProject/sample.txt"));
```

Task 4: Write a program that asks for a filename and the reads all numbers from it and prints their sum.

Listing 1.5: FileApp.java

```
// Task 4: Asks for a filename and then reads 5 numbers and prints
   ↪ the sum

import java.io.File;
import java.io.FileNotFoundException;
import java.util.Scanner;

public class FileApp {
  public static void main(String[] args)
  throws FileNotFoundException {
    System.out.println("This program will add a series");
    System.out.println("of numbers from a file.");
    System.out.println();

    Scanner console = new Scanner(System.in);
    System.out.print("What is the file name? ");
    String name = console.nextLine();
    Scanner input = new Scanner(new File(name));
    System.out.println();

    double sum = 0.0;
    int count = 0;
    while (input.hasNextDouble()) {
      double next = input.nextDouble();
      count++;
      System.out.println("number " + count + " = " + next);
      sum += next;
    }
    System.out.println("Sum = " + sum);

  }
}
```

Task 5: Write a Java program that read the name of employees and their wage per hour, and their working hours from a files and prints their total salary. The file format is ”Name / Baht per hour / working hours ...” and the data is shown below:

Listing 1.6: hours.dat

```
Ehsan 100 7.5 8.5 10.25 8 8.5
Maryam 90 10.5 11.5 12 11 10.75
Zahra 85 8 8 8
Fatima 90 6.5 8 9.25 8
Ali 90 2.5 3
```

Listing 1.7: FileApp.java

```
// Task 5: Read salary information from a file and prints total
   ↪ salary

import java.io.File;
import java.io.FileNotFoundException;
import java.util.Scanner;

public class FileApp {
  public static void main(String[] args)
  throws FileNotFoundException {
    Scanner input = new Scanner(new File("hours.dat"));
    process(input);
  }

  public static void process(Scanner input) {
    System.out.printf("%-15s %-15s %-12s\n",
      "Employee Name", "Total hours", "Total Salary");

    while (input.hasNext()) {
      String name = input.next();
      double wage_per_hour = input.nextDouble();
      double sum = 0.0;
      while (input.hasNextDouble()) {
        sum += input.nextDouble();
      }

      System.out.printf("%-15s %-15.2f %-12.2f THB\n",
        name, sum , (sum * wage_per_hour));
    }
  }
}
```

output:

```
Employee Name   Total hours     Total Salary
Ehsan           42.75           4275.00      THB
Maryam          55.75           5017.50      THB
Zahra           24.00           2040.00      THB
Fatima          31.75           2857.50      THB
Ali             5.50            495.00       THB

Process finished with exit code 0
```

Figure 1.3: Task 5 output.

1.1.12 Line-Based Processing

You'll often find yourself working with input files that are line-based: Each line of input represents a different case, to be handled separately from the rest. These types of files lend themselves to a second style of file processing called *line-based processing*.

Line-Based Processing is the practice of processing input line by line (i.e., reading in entire lines of input at a time).

Most file processing involves a combination of line- and token-based styles, and the Scanner class is flexible enough to allow you to write programs that include both styles of processing. For line-based processing, you'll use the *nextLine* and *hasNextLine* methods of the Scanner object.

Task 6: Write a Java program that reads a text file and outputs it in uppercase.

poem.txt content:

```
Of all the travellers on this endless road
Not one returns to tell us where it leads
There's little in this world but greed and need
Leave nothing here, for you will not return
- Omar Khayyam
```

Listing 1.8: FileApp.java

```
// Task 6: print file content in uppercase

import java.io.File;
import java.io.FileNotFoundException;
import java.util.Scanner;

public class FileApp {
  public static void main(String[] args)
  throws FileNotFoundException {
    Scanner input = new Scanner(new File("poem.txt"));

```

```
    while (input.hasNextLine()) {
      String text = input.nextLine();
      System.out.println(text.toUpperCase());
    }

  }
}
```

This combination of line-based and token-based processing is powerful. You will find that you can use this approach (and slight variations on it) to process a large variety of input files. To summarize, this approach involves a two-step process:

- Break the file into lines with a Scanner using calls on *hasNextLine* and *nextLine*.
- Break apart each line by constructing a Scanner just for that line of input and making calls on token-based methods like *hasNext* and *next*.

Example of line-based and token-based processing combo:

```
Scanner input = new Scanner(new File("sample.dat"));
while (input.hasNextLine()) {
  String text = input.nextLine();
  processLine(text);
}

public static void processLine(String text) {
  Scanner data = new Scanner(text);
  int id = data.nextInt();
  String name = data.next();
  double sum = 0.0;
  while (data.hasNextDouble()) {
    sum += data.nextDouble();
  }
  ...
}
```

1.2 Output Files with PrintStream

All of the programs we've studied so far have sent their output to the console window by calling *System.out.print* or *System.out.println*. But just as you can read input from a file instead of reading from the console, you can write output to a file instead of writing it to the console.

The *System.out* is a variable that stores a reference to an object of type *PrintStream*. The print and *println* statements you've been writing are calls on methods that are part of the *PrintStream* class. The variable *System.out* stores a reference to a special *PrintStream* object that is tied to the console window. However, you can construct other *PrintStream* objects that send their output to other places. Suppose, for example, that you want to send output to a file called results.txt. You can construct a *PrintStream* object as follows:

```
PrintStream output = new PrintStream(new File("results.txt"));
```

If no such file already exists, the program creates it. If such a file does exist, the computer overwrites the current version. Initially, the file will be empty. It will end up containing whatever output you tell it to produce through calls on *print* and *println*.

The line of code that constructs a *PrintStream* object can generate an exception (e.g *FileNotFoundException*) if Java is unable to create the file you've described. There are many reasons that this might happen: You might not have permission to write to the directory, or the file might be locked because another program is using it. Therefore, Java requires you to include the throws clause in whatever method contains this line of code.

Example:

```
// This program creates result.txt in current directory.
import java.io.File;
import java.io.FileNotFoundException;
import java.io.PrintStream;


public class FileApp {
  public static void main(String[] args)
  throws FileNotFoundException {
    PrintStream output =
    new PrintStream(new File("result.txt"));
    output.println("Hello, world.");
    output.println();
    output.println("This program produces four");
    output.println("lines of output.");

  }
}
```

Task 7: Write a Java program that gets a text file as input, and removes all extra spaces between words and outputs the result into a new file.

Input file "word.txt":

```
four   score and
seven   years ago our
fathers   brought   forth on   this     continent
a   new   nation,   conceived in   liberty
  and    dedicated   to the proposition that
all   men   are   created    equal
```

Listing 1.9: FileApp.java

```
// Task 7: This program removes excess spaces in an input file.

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

public class FileApp {
  public static void main(String[] args)
  throws FileNotFoundException {
```

```
    Scanner input = new Scanner(new File("words.txt"));

    PrintStream output =
    new PrintStream(new File("words2.txt"));
    while (input.hasNextLine()) {
      String text = input.nextLine();
      echoFixed(text, output);
      echoFixed(text, System.out);
    }
  }

  public static void echoFixed(String text, PrintStream output) {
    Scanner data = new Scanner(text);
    if (data.hasNext()) {
      output.print(data.next());
      while (data.hasNext()) {
        output.print(" " + data.next());
      }
    }
    output.println();
  }
}
```

1.2.1 Guaranteeing That Files Can Be Read

The programs we have studied so far assume that the user will provide a legal file name. But what if the user accidentally types in the name of a file that doesn't exist or that can't be read for some reason? In this section we explore how to guarantee that a file can be read.

Let's explore how you might handle the task of prompting the user for a file name in the console window. If the user does not input a legal file name, you can keep prompting until the user does enter a legal name. The File class has a method called *canRead* that you can use to test whether a file exists and can be read:

```
System.out.print("input file name? ");
File f = new File(console.nextLine());
while (!f.canRead()) {
  System.out.println("File not found. Try again.");
  System.out.print("input file name? ");
  f = new File(console.nextLine());
}
```

1.3 Boilerplate Code

Boilerplate code is a code that tends to be the same from one program to another.

To get the name of a file is so common task that we can put it into a method:

```
// Prompts the user for a legal file name; creates and
// returns a Scanner tied to the file
public static Scanner getInput(Scanner console)
  throws FileNotFoundException {
  System.out.print("input file name? ");
  File f = new File(console.nextLine());
  while (!f.canRead()) {
    System.out.println("File not found. Try again.");29 System.out.
       ↪ print("input file name? ");
    f = new File(console.nextLine());
    }
  // now we know that f is a file that can be read
  return new Scanner(f);
}
```

The *getInput* method is a good example of the kind of boilerplate code that you might use in many different file-processing programs.

```
public class CountWords {
  public static void main(String[] args)
    throws FileNotFoundException {
    Scanner console = new Scanner(System.in);
    Scanner input = getInput(console);

    // and count words
    int count = 0;
    while (input.hasNext()) {
       String word = input.next();
       count++;
    }
  System.out.println("total words = " + count);
}
```

Task 8: Write a method called *stripHtmlTags* that accepts a *Scanner* representing an input file containing an HTML web page as its parameter, then reads that file and prints the file's text with all HTML tags removed. A tag is any text between the characters < and >. For example, consider the following text:

```
<html>
<head>
<title>My web page</title>
</head>
<body>
<p>There are many pictures of my cat here,
as well as my <b>very cool</b> blog page,
which contains <font color="red">awesome
stuff about my trip to Vegas.</p>
Here's my cat now:<img src="cat.jpg">
</body>
</html>
```

If the file contained these lines, your program should output the following text:

```
My web page
There are many pictures of my cat here,
as well as my very cool blog page,
which contains awesome
stuff about my trip to Vegas.
Here's my cat now:
```

You may assume that the file is a well-formed HTML document and that there are no ¡ or ¿ characters inside tags.

Task 9: Write a method called *stripComments* that accepts a *Scanner* representing an input file containing a Java program as its parameter, reads that file, and then prints the file's text with all comments removed. A comment is any text on a line from // to the end of the line, and any text between /* and */ characters. For example, consider the following text:

```
import java.util.*;
/* My program
by Suzy Student */
public class Program {
  public static void main(String[] args) {
    System.out.println("Hello, world!"); // a println
  }
  public static /* Hello there */ void foo() {
    System.out.println("Goodbye!"); // comment here
  } /* */
}
```

If the file contained this text, your program should output the following text:

```
import java.util.*;
public class Program {
  public static void main(String[] args) {
    System.out.println("Hello, world!");
  }
  public static void foo() {
    System.out.println("Goodbye!");
  }
}
```

1.4 Arrays

An **array** is a flexible structure for storing a sequence of values that are all of the same type. An **array** is an indexed structure that holds multiple values of the same type. The values stored in an array are called *elements*. The individual elements are accessed using an integer *index*. An index is an integer indicating the position of a particular value in a data structure.

Suppose you want to store some different temperature readings. You could keep them in a series of variables:

```
double temperature1;
double temperature2;
double temperature3;
```

This isn't a bad solution if you have just 3 temperatures, but suppose you need to store 3000 temperatures. Then you would want a more flexible way to store the values. You can instead store the temperatures in an array.

When you use an array, you first need to declare a variable for it, so you have to know what type to use. The type will depend on the type of elements you want to have in your array. To indicate that you are creating an array, follow the type name with a set of square brackets: []. If you are storing temperature values, you want a sequence of values of type double, so you use the type *double*[]. Thus, you can declare a variable for storing your array as follows:

```
double[] temperature;
```

Arrays are objects, which means that they must be constructed. Simply declaring a variable isn't enough to bring the object into existence. In this case you want an array of three double values, which you can construct as follows:

```
double[] temperature = new double[3];
```

You can access the array elements by using: temperature[0], temperature[1], temperature[2].

The general syntax for declaring and constructing an array is as follows:

```
<element type>[] <name> = new <element type>[<length>];
```

You can use any type as the element type, although the left and right sides of this statement have to match. For example, any of the following lines of code would be legal ways to construct an array:

```
int[] numbers = new int[10]; // an array of 10 ints
char[] letters = new char[20]; // an array of 20 chars
boolean[] flags = new boolean[5]; // an array of 5 booleans
String[] names = new String[100]; // an array of 100 Strings
Color[] colors = new Color[50]; // an array of 50 Colors
```

1.4.1 Auto-Initialization

Auto-initialization is the initialization of variables to a default value, such as on an array's elements when it is constructed.

When Java performs auto-initialization, it always initializes to the zero-equivalent for the type:

```
int 0
double 0.0
char '\0'
boolean false
objects null
```

If the array size is large then you can perform bulk operation on the elements using a loop:

```
for (int i = 0; i < 100; i++) {
  // read from a file and put values into an array
  temperature[i] = input.nextDouble();
}
```

1.4.2 Array Traversal

Array traversal is the processing each array element sequentially from the first to the last.

This pattern is so useful that it is worth including it in a more general form:

```
for (int i = 0; i < <array>.length; i++) {
  <do something with array[i]>;
}
```

1.4.3 Accessing an Array

Example:

```
// works only for an array of length 5
System.out.println("first = " + list[0]);
System.out.println("middle = " + list[2]);
System.out.println("last = " + list[4]);

// You can manipulate elements:
list[i] = 3;
list[i]++;
list[i] *= 2;
list[i]--;


//you can go through them one by one:
for (int i = 0; i < list.length; i++) {
  list[i]++;
}
```

Task 10: Write a Java program that reads a series of high temperatures and reports the average and the number of days above average.

Listing 1.10: TemperatureApp.java

```
// Task 10: Reads a series of high temperatures and reports the
// average and the number of days above average.

import java.util.*;

public class TemperatureApp {
  public static void main(String[] args) {
    Scanner console = new Scanner(System.in);
    System.out.print("How many days' temperatures? ");
    int numDays = console.nextInt();
    int[] temps = new int[numDays];

    // record temperatures and find average
    int sum = 0;
    for (int i = 0; i < numDays; i++) {
      System.out.print("Day " + (i + 1) + "'s high temp: ");
```

```
      temps[i] = console.nextInt();
      sum += temps[i];
      }
    double average = (double) sum / numDays;

    // count days above average
    int above = 0;
    for (int i = 0; i < temps.length; i++) {
      if (temps[i] > average) {
        above++;
      }
    }

    // report results
    System.out.println();
    System.out.println("Average = " + average);
    System.out.println(above + " days above average");
  }
}
```

1.4.4 Sequential Access vs Random Access

Most of the algorithms we have seen so far have involved *sequential access* which is manipulating values in a sequential manner from first to last. A Scanner object is often all you need for a sequential algorithm, because it allows you to access data by moving forward from the first element to thelast. But as we have seen, there is no way to reset a Scanner back to the beginning. A Scanner object is often all you need for a sequential algorithm, because it allows you to access data by moving forward from the first element to thelast. But as we have seen, there is no way to reset a Scanner back to the beginning.

An array is a powerful data structure that allows a more flexible kind of access known as *random access*. Random Access is manipulating values in any order whatsoever to allow quick access to each value.

When you work with arrays, you can jump around in the array without worrying about how much time it will take. For example, suppose that you have constructed an array of temperature readings that has 10,000 elements and you find yourself wanting to print a particular subset of the readings with code like the following:

```
System.out.println("#1394 = " + temps[1394]);
System.out.println("#6793 = " + temps[6793]);
System.out.println("#72 = " + temps[72]);
```

This code will execute quickly even though you are asking for array elements that are far apart from one another.

1.4.5 Arrays and Methods

You will find that when you pass an array as a parameter to a method, the method has the ability to change the contents of the array. We'll examine in detail later in the chapter why this occurs, but for now, the important point is simply to understand that methods can alter the contents of arrays that are passed to them as parameters.

Let's explore a specific example to better understand how to use arrays as parameters and return values for a method.

The following code constructs an array of odd numbers and increments each array element:

```
int[] list = new int[5];

for (int i = 0; i < list.length; i++) {
  list[i] = 2 * i + 1;
}

for (int i = 0; i < list.length; i++) {
  list[i]++;
}
```

Let's see what happens when we move the incrementing loop into a method:

```
public static void incrementAll(int[] data) {
  for (int i = 0; i < data.length; i++) {
    data[i]++;
  }
}
```

You might think this method will have no effect whatsoever, or that we have to return the array to cause the change to be remembered. But when we use an array as a parameter, this approach actually works. We can replace the incrementing loop in the original code with a call on our method:

```
int[] list = new int[5];
for (int i = 0; i < list.length; i++) {
  list[i] = 2 * i + 1;
}
incrementAll(list);
```

The key lesson to draw from this is that when we pass an array as a parameter to a method, that method has the ability to change the contents of the array.

1.4.6 The For-Each Loop

Java has a loop construct that simplifies certain array loops. It is known as the enhanced for loop, or the *for-each loop*. You can use it whenever you want to examine each value in an array. For example, the program TemperatureApp had an array variable called temps and the following loop:

```
for (int i = 0; i < temps.length; i++) {
  if (temps[i] > average) {
    above++;
  }
}
```

We can rewrite this as a for-each loop:

```
for (int n : temps) {
  if (n > average) {
    above++;
  }
```

```
}
```

This loop is normally read as,"“For each int n in temps...”. The basic syntax of the for-each loop is:

```
for (<type> <name> : <array>) {
  <statement>;
  <statement>;
  ...
  <statement>;
}
```

1.4.7 Initializing Arrays

Java has a special syntax for initializing an array when you know exactly what you want to put into it.

```
int[] daysIn = new int[12];
daysIn[0] = 31;
daysIn[1] = 28;
daysIn[2] = 31;
daysIn[3] = 30;
daysIn[4] = 31;
daysIn[5] = 30;
daysIn[6] = 31;
daysIn[7] = 31;
daysIn[8] = 30;
daysIn[9] = 31;
daysIn[10] = 30;
daysIn[11] = 31;

String[] dayNames = new String[7];
dayNames[0] = "Mon";
dayNames[1] = "Tue";
dayNames[2] = "Wed";
dayNames[3] = "Thu";
dayNames[4] = "Fri";
dayNames[5] = "Sat";
dayNames[6] = "Sun";
```

This code works, but it’s a rather tedious way to declare these arrays. Java provides a shorthand:

```
int[] daysIn = {31, 28, 31, 30, 31, 30, 31, 31, 30, 31, 30, 31};
String[] dayNames =
  {"Mon", "Tue", "Wed", "Thu", "Fri", "Sat", "Sun"};
```

The general syntax for array initialization is as follows:

```
<element type>[] <name> = {<value>, <value>, ..., <value>};
```

1.4.8 The Arrays Class

Arrays have some important limitations that you should understand. Over the years Java has attempted to remedy these limitations by providing various utility methods in a class called Arrays. This class provides many methods that make it easier to work with arrays. The *Arrays* class is part of the *java.util* package.

The first limitation you should be aware of is that you can't change the size of an array in the middle of program execution. If you find that you need a larger array, you should construct a new array and copy the values from the old array to the new array. The method *Arrays.copyOf* provides exactly this functionality.

For example, if you have an array called data, you can create a copy that is twice as large with the following line of code:

```
1 int[] newData = Arrays.copyOf(data, 2 * data.length);
```

If you want to copy only a portion of an array, there is a similar method called *Arrays.copyOfRange* that accepts an array, a starting index, and an ending index as parameters.

The second limitation is that you can't print an array using a simple print or println statement. The *Arrays* class once again offers a solution: The method *Arrays.toString* returns a conveniently formatted version of an array. For example, the following three lines of code:

```
1 int[] primes = {2, 3, 5, 7, 11, 13, 17, 19, 23};
  System.out.println(primes);
3 System.out.println(Arrays.toString(primes));
```

It produces the following output:

```
  [I@fee4648
2 [2, 3, 5, 7, 11, 13, 17, 19, 23]
```

Notice that the first line of output is not at all helpful. The second line, however, allows us to see the list of prime numbers in the array because we called *Arrays.toString* to format the array before printing it.

Table 1.3 shows useful Methods of the *Arrays* Class.

Method	Description
copyOf(array, newSize)	returns a copy of the array with the given size
copyOfRange(array, startIndex, endIndex)	returns a copy of the given subportion of the given array from startIndex (inclusive) to endIndex (exclusive)
equals(array1, array2)	returns true if the arrays contain the same elements
fill(array, value)	sets every element of the array to be the given value
sort(array)	rearranges the elements so that they appear in sorted (nondecreasing) order
toString(array)	returns a String representation of the array, as in [3, 5, 7]

Table 1.3: Useful Methods of the Arrays Class

The third limitation is that you can't compare arrays for equality using a simple == test.

1.4.9 Printing an Array

You can then call this method with the variable list: print(list);

```
public static void print(int[] list) {
  for (int n : list) {
    System.out.println(n);
  }
}

// You can then call this method with the variable list:
print(list);
```

In some cases, the for-each loop doesn't get you quite what you want, though. For example, consider how the Arrays.toString method must be written. It produces a list of values that are separated by commas, which is a classic fencepost problem (e.g., seven values separated by six commas). To solve the fencepost problem, you'd want to use an indexing loop instead of a for-each loop so that you can print the first value before the loop:

```
System.out.print(list[0]);
for (int i = 1; i < list.length; i++) {
  System.out.print(", " + list[i]);
}
System.out.println();
```

Even this code is not correct, though, because it assumes that there is a list[0] to print. It is possible for arrays to be empty, with a length of 0, in which case this code will generate an *ArrayIndexOutOfBoundsException*.

A better version:

```
public static void print(int[] list)
  {if (list.length == 0) {
    System.out.println("[]");
  } else {
    System.out.print("[" + list[0]);
    for (int i = 1; i < list.length; i++) {
      System.out.print(", " + list[i]);
    }
    System.out.println("]");
  }
}
```

1.4.10 Testing for Equality

Below show the *equals* that can be used to check for arrays equality:

```
public static boolean equals(int[] list1, int[] list2) {
  if (list1.length != list2.length) {
    return false;
  }
  for (int i = 0; i < list1.length; i++) {
    if (list1[i] != list2[i]) {
      return false;
```

```
    }
  }
  return true;
}
```

1.4.11 Streams

Instead of specifying exactly how to traverse an array, you can instead tell Java what you want to do with the array elements and allow Java to figure out how to do the traversal. The addition of the for-each loopstarting with version 5 of Java was an initial move in this direction, but the new features go much further.

Suppose, for example, that you have an array of values defined as follows:

```
int[] numbers = {8, 3, 2, 17};
```

Let's look at the code you would write for two simple tasks: finding the sum and printing the values. Using the standard traversal loops, you would write the following code.

```
// sum an array of numbers and print them (for loop)
int sum = 0;
for (int i = 0; i < numbers.length; i++) {
  sum += numbers[i];
}
System.out.println("sum = " + sum);

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

The for-each loop simplifies this code by specifying that you want to manipulate each of the different values in the array in sequence, but it doesn't require you to include an indexing variable to say exactly how that is done.

```
// sum an array of numbers and print them (for-each loop)
int sum = 0;
for (int n : numbers) {
  sum += n;
}
System.out.println("sum = " + sum);
for (int n : numbers) {
  System.out.println(n);
}
```

With the new Java 8 features, this becomes even simpler. The task of finding the sum of a sequence of values is so common that there is a built-in method that does it for you. And the task of printing each value with a call on the *println* method of System.out can also be expressed in a very concise manner.

```
// sum an array of numbers and print them
// Arrays.stream is a static method that return an IntStream Object
// IntStream Object has an instance methd sum()
// sum() Returns the sum of elements in this stream.
```

```
int sum = Arrays.stream(numbers).sum();
System.out.println("sum = " + sum);
Arrays.stream(numbers).forEach(System.out::println);
```

This code doesn't at all describe how the traversal is to be performed. Instead, you tell Java the operations you want to have performed on the values in the array and leave it up to Java to perform the traversal.

Look at the documentation of IntStream here: `https://docs.oracle.com/javase/8/docs/api/java/util/stream/IntStream.html`

1.4.12 Reference Semantics

In Java, arrays are objects. Objects are stored in the computer's memory in a different way than primitive data are stored. For example, when we declare the integer variable:

```
int x = 8;
```

the variable stores the actual data inside a memory location.

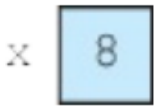

Figure 1.4: Primitive variable stored the actual data.

The situation is different for arrays and other objects. With regard to objects, the variable doesn't store the actual data. Instead, the data are stored in an object (several memory locations) and the variable stores a reference (in another memory location) to the location at which the object is stored.

```
int[] list = new int[5];
```

As the figure 1.5 indicates, two different values are stored in memory: the array itself, which appears on the right side of the diagram, and a variable called list, which stores a reference to the array (represented in this picture as an arrow). We say that list *refers* to the array.

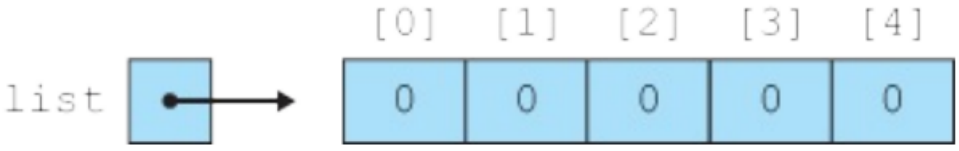

Figure 1.5: Variable stores a reference to object.

It may take some time for you to get used to the two different approaches to storing data, but these approaches are so common that computer scientists have technical terms to describe them. The system for the primitive types like int is known as **value semantics**, and those types are often referred to as

value types. The system for arrays and other objects is known as **reference semantics**, and those types are often referred to as *reference types*.

Value Semantics (Value Types) is a system in which values are stored directly and copying is achieved by creating independent copies of values. Types that use value semantics are called value types.

Reference Semantics (Reference Types) is a system in which references to values are stored and copying is achieved by copying these references. Types that use reference semantics are called reference types.

The key thing to remember is that when you are working with objects, you are always working with references to data rather than the data itself.

At this point you are probably wondering why Java has two different systems. Java was designed for object-oriented programming, so the first question to consider is why Sun decided that objects should have reference semantics. There are two primary reasons:

- **Efficiency**. Objects can be complex, which means that they can take up a lot of space in memory. If we made copies of such objects, we would quickly run out of memory. A String object that stores a large number of characters might take up a lot of space in memory. But even if the *String* object is very large, a reference to it can be fairly small
- **Sharing**. Often, having a copy of something is not good enough. Suppose that your instructor tells all of the students in the class to put their tests into a certain box. Imagine how pointless and confusing it would be if each student made a copy of the box. The obvious intent is that all of the students use the same box. Reference semantics allows you to have many references to a single object, which allows different parts of your program to share a certain object.

Without reference semantics, Java programs would be more difficult to write. Then why did Sun also decide to include primitive types that have value semantics? The reasons are primarily historical. Sun wanted to leverage the popularity of C and C++, which had similar types, and to guarantee that Java programs would run quickly, which was easier to accomplish with the more traditional primitive types. If Java's designers had a chance to redesign Java today, the company might well get rid of the primitive types and use a consistent object model with just reference semantics.

1.4.13 Multiple Objects

Consider the following code:

```
int[] list1 = new int[5];
int[] list2 = new int[5];
for (int i = 0; i < list1.length; i++) {
  list1[i] = 2 * i + 1;
  list2[i] = 2 * i + 1;
}
int[] list3 = list2;
```

which will set the memory to look like this:

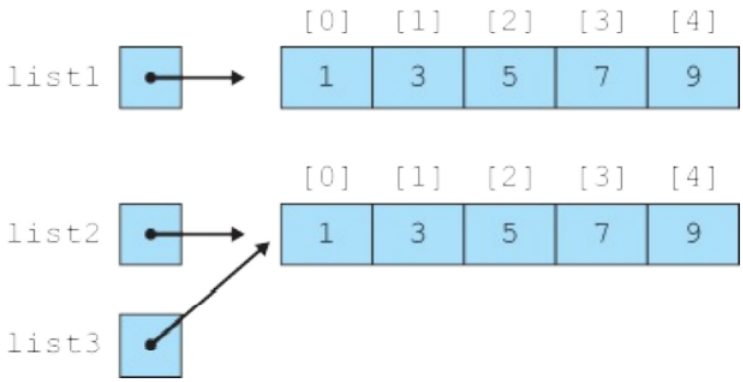

Figure 1.6: Two different objects, and three variables.

We have three variables but only two objects. The variables *list2* and *list3* both refer to the same array object.

The implication of this method is that list2 and list3 are in some sense both equally able to modify the array to which they refer. The line of code

```
list2[2]++;
```

will have exactly the same effect as the line

```
list3[2]++;
```

Since both variables refer to the same array object, you can access the array through either one.

Reference semantics help us to understand why a simple == test does not give us what we might expect. When this test is applied to objects, it determineswhether two references are the same (not whether the objects to which they refer are somehow equivalent). In other words, when we test whether two references are equal, we are testing whether they refer to exactly the same object. The variables list2 and list3 both refer to the same array object. As a result, if we ask whether list2 == list3, the answer will be yes (the expression evaluates to true). But if we ask whether list1 == list2, the answer will be no (the expression evaluates to false) even though we think of the two arrays as somehow being equivalent.

Sometimes you want to know whether two variables refer to exactly the same object, and for those situations, the simple == comparison will be appropriate. But you'll also want to know whether two objects are somehow equivalent in value, in which case you should call methods like *Arrays.equals* or the string *equals* method.

Understanding reference semantics also allows you to understand why a method is able to change the contents of an array that is passed to it as a parameter!

1.4.14 null

null is a Java keyword signifying no object.

The concept of null doesn't have any meaning for value types like int and double that store actual values. But it can make sense to set a variable that stores a reference to null. This is a way of telling the computer that you want to have the variable, but you haven't yet come up with an object to which it should refer. So you can use null for variables of any object type, such as a String or array:

```
String s = null;
int[] list = null;
```

There is a difference between setting a variable to an empty string and setting it to *null*. When you set a variable to an empty string, there is an actual object to which your variable refers (although not a very interesting object). When you set a variable to *null*, the variable doesn't yet refer to an actual object. If you try to use the variable to access the object when it has been set to null, Java will throw a *NullPointerException*.

1.4.15 Arrays of Objects

All of the arrays we have looked at so far have stored primitive values like simple int values, but you can have arrays of any Java type. Arrays of objects behave slightly differently, though, because objects are stored as references rather than as data values. Constructing an array of objects is usually a two-step process, because you normally have to construct both the array and the individual objects.

Suppose that you want to construct an array of LCD objects. Consider the following statement:

```
LCD[] lcds = new LCD[3];
```

This statement declares a variable called *lcds* that refers to an array of length 3 that stores references to LCD objects. Using the *new* keyword to construct the array doesn't construct any actual Point objects. Instead it constructs an array of length 3, each element of which can store a reference to a LCD. When Java constructs the array, it auto-initializes these array elements to the zero-equivalent for the type. The zero-equivalent for all reference types is the special value *null*, which indicates "no object".

The actual LCD objects must be constructed separately with the *new* keyword, as in the following code:

```
1 LCD[] lcds = new LCD[3];
  lcds[0] = new LCD(80, 25);
3 lcds[1] = new LCD(40, 25);
  lcds[2] = new LCD(16, 2);
```

Notice that the *new* keyword is required in four different places, because there are four objects to be constructed: the array itself and the three individual LCD objects.

1.4.16 Multidimensional Array

Multidimensional array is an array of arrays, the elements of which are accessed with multiple integer indexes.

The most common use of a multidimensional array is a two-dimensional array of a certain width and height. For example, suppose that on three separate days you took a series of five temperature readings. You can define a two-dimensional array that has three rows and five columns as follows:

```
double[][] temps = new double[3][5];
```

As with one-dimensional arrays, the values are initialized to 0.0 and the indexes start with 0 for both rows and columns. Once you've created such an array, you can refer to individual elements by providing specific row and column numbers (in that order):

```
1 temps[0][3] = 98.3; // fourth value of first row
  temps[2][0] = 99.4; // first value of third row
```

It is helpful to think of referring to individual elements in a stepwise fashion, starting with the name of the array. For example, if you want to refer to the first value of the third row, you obtain it through the following steps:

```
temps the entire grid
temps[2] the entire third row
temps[2][0] the first element of the third row
```

You can pass multidimensional arrays as parameters just as you pass one-dimensional arrays. You need to be careful about the type, though. To pass the temperature grid, you would have to use a parameter of type *double*[][] (with both sets of brackets):

```
public static void print(double[][] grid) {
  for (int i = 0; i < grid.length; i++) {
    for (int j = 0; j < grid[i].length; j++) {
      System.out.print(grid[i][j] + " ");
    }
    System.out.println();
  }
}
```

Notice that to ask for the number of rows you ask for *grid.length* and to ask for the number of columns you ask for *grid[i].length*.

To print multidimensional arrays:

```
System.out.println(Arrays.deepToString(temps));
```

1.4.17 Jagged Arrays

The previous examples have involved rectangular grids that have a fixed number of rows and columns. It is also possible to create a jagged array in which the number of columns varies from row to row.

To construct a jagged array, divide the construction into two steps: Construct the array for holding rows first, and then construct each individual row:

```
int[][] jagged = new int[3][];
jagged[0] = new int[2];
jagged[1] = new int[4];
jagged[2] = new int[3];
```

This code would construct an array that is shown in Figure 1.7.

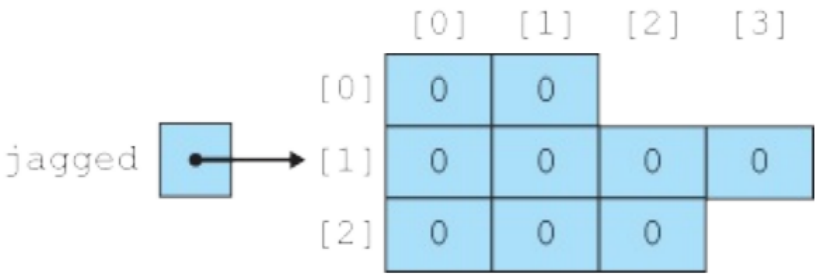

Figure 1.7: Jagged Array.

Task 11: Write a piece of code that examines an array of integers and reports the maximum value in the array. Consider putting your code into a method called max that accepts the array as a parameter and returns the maximum value. Assume that the array contains at least one element.

Task 12: Write a piece of code that prints an array of integers in reverse order, in the same format as the print method from Section 7.2. Consider putting your code into a method called printBackwards that accepts the array as a parameter.

Task 13: Write a method called swapPairs that accepts an array of integers and swaps the elements at adjacent indexes. That is, elements 0 and 1 are swapped, elements 2 and 3 are swapped, and so on. If the array has an odd length, the final element should be left unmodified. For example, the list [10, 20, 30, 40, 50] should become [20, 10, 40, 30, 50] after a call to your method.

Task 14: Write code that accepts an array of Strings as its parameter and indicates whether that array is a palindrome-that is, whether it reads the same forward as backward. For example, the array ”alpha”, ”beta”, ”gamma”, ”delta”, ”gamma”, ”beta”, ”alpha” is a palindrome.

Task 15: Write a piece of code that constructs a jagged two-dimensional array of integers with five rows and an increasing number of columns in each row, such that the first row has one column, the second row has two, the third has three, and so on. The array elements should have increasing values in top-to-bottom, left-to-right order (also called row-major order). In other words, the array's contents should be the following:

```
1
2, 3
4, 5, 6
7, 8, 9, 10
11, 12, 13, 14, 15
```

Use nested for loops to build the array.