Chapter 18

Exceptions and Files

18.1 Exceptions

Consider this program:

```
import java.util.Scanner;

public class ErrorProne {

    public static void main(String[] args) {
        int[] data = {10, 66, 47, 11};

        Scanner input = new Scanner(System.in);

        System.out.print("Enter index 0-3: ");
        int index = input.nextInt();

        System.out.print("Enter number to divide by: ");
        int divisor = input.nextInt();

        int result = data[index] / divisor;
        System.out.printf("quotient of %d and %d is %d\n",
            data[index], divisor, result);
    }
}
```

If you enter a non-number the program crashes:

```
Enter index 0-3: two
Exception in thread "main" java.util.InputMismatchException
        at java.base/java.util.Scanner.throwFor(Scanner.java:939)
        at java.base/java.util.Scanner.next(Scanner.java:1594)
        at java.base/java.util.Scanner.nextInt(Scanner.java:2258)
        at java.base/java.util.Scanner.nextInt(Scanner.java:2212)
        at ErrorProne.main(ErrorProne.java:11)
```

The lines beginning with `at` are a *stack trace*. They show the chain of method calls in reverse chronological order with the file name and line number. You'll want to look for the one that is in your program. In this case, the error was in `ErrorProne.java:11`, where the `11` is the line number in the source file with the `nextInt` call.

If you enter an index outside the array bounds, the program crashes (the output has been reformatted to fit on the line length of this page):

```
Enter index 0-3: 5
Enter number to divide by: 0
Exception in thread "main"
  java.lang.ArrayIndexOutOfBoundsException:
  Index 5 out of bounds for length 4
        at ErrorProne.main(ErrorProne.java:21)
```

And if you enter a zero as the divisor, you get yet another error:

```
Enter index 0-3: 2
Enter number to divide by: 0
Exception in thread "main"
  java.lang.ArithmeticException: / by zero
    at ErrorProne.main(ErrorProne.java:21)
```

All of these errors are called *exceptions*—exceptional conditions after which the program cannot continue to run. In Java, we say that the program *throws* an exception when it fails.

You already know how to handle these problems before they crash your program: you can use an `if` statement with `Scanner`'s `hasNextInt` method to make sure that the user enters an integer. You can use `if` statements to check

that the index number is between 0 and the array's length, and that the divisor is non-zero.

In addition to using an `if` statement to avoid errors, Java has another general mechanism for catching exceptions before they stop your program: `try` and `catch`.

Let's enclose the code that could have an error in a `try` block:

```
try {
    System.out.print("Enter index 0-3: ");
    int index = input.nextInt();

    System.out.print("Enter number to divide by: ");
    int divisor = input.nextInt();

    int result = data[index] / divisor;
    System.out.printf("quotient of %d and %d is %d\n",
        data[index], divisor, result);
}
```

The `try` block is followed by a `catch` block that specifies the exception we want to handle and how to handle it. Let's start with the division by zero, which generated a `java.lang.ArithmeticException`:

```
catch (ArithmeticException ex) {
    System.out.println("Number to divide by cannot be zero.");
}
```

If you recompile and run the program and enter `2` and `0` as your numbers, you'll get the error message in the `catch` block. Notice that the `printf` statement after the division doesn't occur—when an exception is thrown, execution imediately jumps to the `catch`.

If you enter `five` or `5` for the first input, you'll still get the `NumberFormatException` or `ArrayIndexOutOfBoundsException`.

You may follow a `try` block with as many `catch` blocks as you want. Let's add two more `catch` blocks to handle these other two errors:

```
catch (NumberFormatException ex) {
    System.out.println("You must enter digits for numbers.");
}
catch (ArrayIndexOutOfBoundsException ex) {
    System.out.printf("Index must be in range 0-%d\n", data.length);
}
```

The variable in parentheses after `catch` is local to the `catch` block. This means you can use the same variable name in all the `catch` blocks, and, by convention, most programmers name it `ex`. (We will put it to use later in the chapter.)

When an exception occurs, Java goes through the `catch` blocks in the order that they appear in your program and finds the first one that applies. In the preceding example, we could have put the `catch` blocks in any order. However, the order does become important once we examine the hierarchy of exceptions.

18.2 The Hierarchy of Exceptions

All exceptions descend from the `Exception` class[1]. This list shows many of the most common exceptions you will encounter when learning Java; each category contains many other classes:

- `Exception`
  - `IOException`
    - `FileNotFoundException`
  - `RunTimeException`
    - `ArithmeticException`
    - `IllegalArgumentException`
      - `IllegalFormatException`
      - `InvalidParameterException`
      - `NumberFormatException`

[1] `Exception` is a child of the `Throwable` class. Another child of `Throwable` is `Error`, which is used for serious, system-level problems. You will very rarely encounter one of these.

    - `IndexOutOfBoundsException`
        - `ArrayIndexOutOfBoundsException`
        - `StringIndexOutOfBoundsException`
    - `NullPointerException`

If you put a `catch` for a parent class *before* a `catch` for a child class, the parent class will catch the error. Thus, in this code fragment:

```
try {
    int n = 12 / 0;
}
catch (Exception ex) {
    System.out.println("Something unexpected occurred.");
}
catch (ArithmeticException ex) {
    System.out.println("You can't divide by zero.");
}
```

You will see the "Something unexpected occurred." message. For this reason, always `catch` the more specific (child) exception classes before you `catch` the more general (parent) exception classes.

18.3 Using the Exception Variable

Let's say you `catch` the most generic `Exception` possible, or one that could have many possible causes, such as `FileNotFoundException`. How can you give the user more information than just "something unexpected occurred"? You can use the variable that you declared in the `catch` clause. Here are some methods that you can use[2]:

`getMessage()`
: Returns a detailed message string

`toString()`
: Returns a short description

[2]These methods are from the `Throwable` class, which is the parent of all Java exceptions.

`printStackTrace()`
: This `void` method prints the exception and its stack trace to the standard error stream, which is your terminal window

For example, you could `catch` *only* `Exception` and use one of these methods to tell users what went wrong:

```
catch (Exception ex) {
    System.out.println("An error occurred:");
    System.out.println(ex.toString());
}
```

You can see this in action in file *ErrorProneGeneralException.java* in the code repository. As you can see, the error messages are not as satisfactory as those you would write yourself when handling the specific exceptions.

18.4 The `finally` clause

Sometimes programs need to take an action whether the code in the `try` block succeeded or not. (For example, if you have allocated resources and want to "clean up" before exiting the program.) This is the role of the `finally` clause. It is executed whether the `try` block succeeded or an exception was caught by a `catch` block. In fact, it is executed even when there is a `return` or `break` in a block:

```
public class FinallyTest {

    public static void main(String[] args) {
        Scanner input = new Scanner(System.in);

        try {
            int result = 17 / 5;
            System.out.printf("quotient of 17 and 5 is %d\n",
               result);
            return; // exits the main() method
        }
        catch (ArithmeticException ex) {
            System.out.println("You can't divide by zero.");
        }
        finally {
            System.out.println("In the finally clause.");
        }
        System.out.println("Does not print if return succeeds");
    }
}
```

This program will print `In the finally clause.` even though the `return` prevents the last `println` from happening.

18.5 Throwing Exceptions

Before proceeding, let's ask a philosophical question: why do these exceptions even exist? Why doesn't the JVM simply do something reasonable when it encounters one of these situations? That's the whole problem—what does "reasonable" mean? For dividing by zero, some programs might want to return a zero as a default answer. Other programs might want to print an error message and end the program. Still others might want to detect the attempt to divide by zero and ask for new input.

Because the definition of "reasonable" is different for every program and every programmer, it makes sense for these exceptions that happen in many different circumstances to throw the problem back to the programmer and say, "you handle this."

If you are writing a library of Java methods for other people to use, you won't be able to anticipate all of your users' needs either. Your methods will also need to `throw` an `Exception` so that the people who use your methods can handle errors as they see fit.

For example, let's say you have a Java library with a method that calculates the average of an array of `double` values. If somebody hands you an empty array, that's an illegal argument, and you can write your code to `throw` an `IllegalArgumentException`:

```
1  public class ArrayStats {
2    public static double average(double[] data)
3     throws IllegalArgumentException {
4      int n = data.length;
5      if (n > 0) {
6        double sum = 0.0;
7        for (double value: data) {
8          sum += value;
9        }
10       return sum / n;
11     } else {
12       throw new IllegalArgumentException("Empty array");
13     }
14   }
15 }
```

In line 3, specify that this method `throws` an `IllegalArgumentException`.

In line 12, when the length of the array is zero, use the keyword `throw` and create a `new` `IllegalArgumentException`. The argument to the constructor is the value that will be returned when someone calls the exception's `getMessage` method.

Here's an example that calls the code (without `try` and `catch`) and the resulting error, formatted to fit the page:

```
public class TestArrayStats {
    public static void main(String[] args) {
        double[] items = new double[0];
        double result = ArrayStats.average(items);
        System.out.println("Average is " + result);
    }
}

Exception in thread "main"
  java.lang.IllegalArgumentException: Empty array
        at ArrayStats.average(ArrayStats.java:12)
        at TestArrayStats.main(TestArrayStats.java:5)
```

If your method can throw more than one exception, you list them separated by commas in the `throws` clause:

```
public static void example() throws IllegalArgumentException,
  ArithmeticException, NumberFormatException {
    // ...code
}
```

18.6 Checked and Unchecked Exceptions

All of the exceptions used in the preceding examples (and all exceptions that are descendants of `RuntimeException`) are called *unchecked exceptions*. Java doesn't require you to enclose operations that cause such exceptions in a `try-catch` block. This is a good thing, or you'd need `try-catch` blocks around every division, array access, and string-to-numeric conversion.

For many operations that could throw an unchecked exception, you are better off using an `if` statement to avoid the error in the first place. You can, for example, use an `if` to check if an index is within array bounds, a divisor is non-zero, or if the user has actually entered data that can be converted to integer. This also gives you greater control over the program flow and structure. See, for example, the *NormalErrorChecking.java* file in the code repository.

So, why are we talking about `try` and `catch` at all? Because there are some exceptions that can't easily be handled by an `if-else`. These exceptions are

called *checked exceptions*. The Java compiler requires you to enclose operations that might throw these exceptions in a `try` block and provide a `catch` block to handle them. You are also required to list them in a `throws` clause if you are throwing those exceptions yourself. Foremost of these checked exceptions is the `IOException`, generated by I/O (Input/Output) operations. When you are working with files in Java, you will need to check this exception. This leads us to the next major topic in this chapter:

18.7 Files

Up to this point, you've entered all the data for a program from the keyboard, using a `Scanner` with `System.in`. What if someone sends you a file of several months' worth of weather data from Munich, Germany, in a file named *klima.txt*. You can find this file in the *ch018* folder in the repository[3].

```
MESS_DATUM;TMK;TXK;TNK
20200106;0.8;6.0;-2.5
20200107;2.4;4.6;-2.1
20200108;3.6;7.2;-1.3
20200109;7.7;14.5;3.6
...
20210704;17.9;22.9;14.7
20210705;18.5;22.4;13.9
20210706;21.8;31.5;15.4
20210707;18.2;20.5;16.7
20210708;16.9;20.1;14.7
```

The columns stand for date, average daily temperature, high temperature, and low temperature (temperatures are in °C). If you want to find the maximum temperature and minimum temperature across the whole time period, you certainly don't want to have to type all the numbers again at the keyboard. Instead, you want Java to be able to read the file from your disk.

[3]The file is an edited version of the data at https://opendata.dwd.de/climate_environment/CDC/observations_germany/climate/daily/kl/recent/tageswerte_KL_03379_akt.zip

18.7.1 The `File` Object

In order to access a file, you must use its path name to create a `File` object. A path name describes how to get to a file in the file system. For this chapter, we'll presume that your data files are in the same directory as the *.class* file for your program. That way, the path name is the same as the file name. (For more details about path names, see Appendix A.)

The resulting object doesn't give you access to the file contents; rather, it gives you access to information about the file.

Here's the start of a program that lets you enter a path name and find out about that file:

```
import java.util.Scanner;
import java.io.File; // this is a new import

public class FileInfo {
    public static void main(String[] args) {
        Scanner input = new Scanner(System.in);
        System.out.print("Enter a path name: ");
        String pathName = input.nextLine();

        File f = new File(pathName);
```

The variable `f` is what, in other programming languages, is called a *file handle* or *file descriptor*. The code continues by calling some of the more useful methods in the `File` class:

```
        System.out.println("File exists: " + f.exists());
        System.out.println("File size:   " + f.length());
        System.out.println("Readble:     " + f.canRead());
        System.out.println("Writeable:   " + f.canWrite());
        System.out.println("Executable:  " + f.canExecute());
        System.out.println("Directory:   " + f.isDirectory());
        System.out.println("Normal file: " + f.isFile());
        System.out.println("Hidden file: " + f.isHidden());
    }
}
```

A couple of notes: the `length` method returns the file size in bytes. It is possible for a file to be neither a directory nor a "normal file"—the `/dev/zero` path on Linux refers to a *virtual file* that is neither.

The `File` class also has methods that let you delete files, rename them, and create directories. For details, see https://docs.oracle.com/en/java/javase/16/docs/api/java.base/java/io/File.html.

18.7.2 Reading Files

Let's write a program that opens the *klima.txt* file and finds the maximum and minimum daily temperature across the time period described in the file.

In order to read the contents of a file, you must open a `Scanner` based on the `File` object. But if you try code like this:

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

public class LineCount {

    public static void main(String[] args) {
        File f = new File("klima.txt");

        Scanner input = new Scanner(f);
    }
}
```

The compiler will complain (message reformatted to fit page width):

```
Klima.java:9: error: unreported exception
FileNotFoundException; must be caught or declared to be thrown
        Scanner input = new Scanner(f);
                        ^
1 error
```

The `FileNotFoundException` is a *checked exception*, and the compiler insists that you either catch it or throw it to the caller.

This will require you to import `java.io.FileNotFoundException` and set up a `try-catch` block.

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

public class Klima {

    public static void main(String[] args) {
        File f = new File("klima.txt");
        try {
            Scanner input = new Scanner(f);
            // code goes here
            input.close();
        }
        catch (FileNotFoundException ex) {
            System.out.println("Can't find file klima.txt");
        }
    }
}
```

Now we are set to read the file's contents. Rather than write the whole program right now, let's start by reading the file one line at a time, printing them out to the screen, and counting the number of lines. Here's the code that goes inside the `try` block. The `hasNextLine` method returns `false` when it hits the end of the file:

```
            while (input.hasNextLine()) {
                String oneLine = input.nextLine();
                System.out.println(oneLine);
                lineCount++;
            }
            System.out.println("# of lines: " + lineCount);
            input.close();
```

Now, that we know that we can sucessfully open and read the file, we can modify the code to accomplish the task we want to do: finding the minimum and maximum temperatures.

The program will read lines one at a time and then extract the data from each line. Because `Scanner`'s `next` and `nextDouble` methods use whitespace to separate items, we can't use them "as-is" here, where data items are separated by semicolons. We could solve this problem by using the `useDelimiter` method to change the separator to a semicolon, but that would deprive us of the opportunity to learn about a new `String` method and practice more with exceptions.

Let's replace the line-counting code with this code for finding the maximum and minimum temperatures:

```
double max = -1000.0;
double min = 1000.0;
String[] items = oneLine.split(";");
if (items.length == 4) {
    dayMax = Double.parseDouble(items[2]);
    dayMin = Double.parseDouble(items[3]);
    if (dayMax > max) {
        max = dayMax;
    }
    if (dayMin < min) {
        min = dayMin;
    }
}
```

The new `String` method is `split`. Given a delimiter, this method splits the given `String` into an array of strings wherever it finds the delimiter that you give it as an argument. For example, after this code executes:

```
String s = "sister-in-law";
String[] parts = s.split("-");
```

The `parts` array will be `{"sister", "in", "law"}`.

Similarly, the program uses `oneLine.split(";")` to separate the items on each line. If there aren't four items on a line, it's incomplete, and the program does nothing (effectively skipping over the line).

We now have a problem: the first line doesn't have any numbers on it, and trying to use `parseDouble` on the titles will throw a `NumberFormatException`.

One way to solve the problem is to read in the first line before entering the `while` loop and discarding the result. Another way to solve this problem (which could also occur if the file we were given had bad data in it), is to use another `try-catch`:

```
try {
    double dayMax = Double.parseDouble(items[2]);
    double dayMin = Double.parseDouble(items[3]);
}
catch (NumberFormatException ex) {
    System.out.println("Ignoring non-numeric data "
        + ex.getMessage());
}
```

18.7.3 Writing Files

There are many Java classes for reading files. In this book, we've been using `Scanner` to read input because it contains many methods to make getting input simple.

In a similar way, there are many Java classes for writing files. We'll discuss only one of them: `PrintWriter`. To use this class, you must:

```
import java.io.PrintWriter;
```

Just as you created a `Scanner` by using a `File` object as a parameter to the constructor, you can write to a disk file by creating a `File` object with the path you want and then use that object as a parameter to the `PrintWriter` constructor. And, just as the compiler required you to enclose the code in a `try-catch` block, you must do the same when opening a `PrintWriter`

```
File f = new File("output.txt");
try {
    PrintWriter output = new PrintWriter(f);
    // code to write to file goes here
}
catch (FileNotFoundException ex) {
    System.out.println("Unable to open output file.");
}
```

Just like `System.out`, the `PrintWriter` object `output` we have created has `print`, `println`, and `printf` methods. Instead of writing to your screen, they write data to the file you specified.

18.7.4 Writing Files - Two Important Notes

1. When you open a `PrintWriter` to a `File` that does not exist, it will be created for you. If you open a `PrintWriter` to a `File` that *does* exist, **the existing file will be deleted and re-created**. Any information that was in the file will be gone, even if you never write anything to the `PrintWriter`.

   This means that it is always useful to use the `exists` method to check if a file already exists and, when possible, give the user the option to overwrite the old file or exit the program.

2. When you do a `println` to a `PrintWriter`, the data is not written to disk immediately. Instead, it is kept in a *buffer*, and is written only when the buffer is full. If you exit the program with the buffer partially filled, there is a chance that it might not be written to disk. *Always* call the `close` method on your output files to make sure that the buffer is written to disk.

   If you run this program:

```
import java.io.File;
import java.io.FileNotFoundException;
import java.io.PrintWriter;

public class PartialWrite {

    public static void main(String[] args) {
        File f = new File("write_test.txt");
        try {
            PrintWriter output = new PrintWriter(f);
            output.println("Example of writing to a file.");
            // output.close(); // uncomment this line
        }
        catch (FileNotFoundException ex) {
            System.out.println("Can't open file write_test.txt");
        }
    }
}
```

without closing the file, the resulting *write_ test.txt* file will be empty. If you uncomment the `output.close();` line and run the program again, the file will contain the output.

18.8 try with Resources

Because it is important to close files, Java has a syntax for associating `Files` with input and output classes as part of the `try` syntax. When using this syntax, the Java Virtual Machine will automatically close the input and output when the `try` block exits. Here is an example that uses `try` with resources to open a `PrintWriter`:

```
File outFile = new File("output_path.txt");
try (PrintWriter outWriter = new PrintWriter(outFile)) {
    outWriter.println("Example of try with resources.");
}
catch (FileNotFoundException ex) {
    System.out.println("Error: " + ex.getMessage);
}
```

The declaration of `outWriter` is now in parentheses after `try` rather than after the block's opening brace. We no longer need to call `outWriter.close()`—the JVM will automatically do the call when it exits the `try-catch` block.

You can declare as many input and output objects as you want inside the parentheses:

```
File inFile = new File("input_path.txt");
File outFile = new File("output_path.txt");
try (
    Scanner inScan = new Scanner(inFile);
    PrintWriter outWriter = new PrintWriter(outFile);
) {
    // code...
}
catch (Exception ex) {
    // error handling...
}
```

The `File` declarations cannot go inside the parentheses; the compiler won't let you do that.

18.9 Exercises

Exercise 18.1 Write a program named *TestAverages.java* that asks the user for the name of an input file that has people's names and test scores. Each line has the person's name, a colon, and a list of scores separated by commas.

Next, the program asks the user for the name of an output file. The program will then read the input file and write a new output file with the names and their average scores, including the number of tests on which it is based.

Your program has to do the following error handling:

- If the input file does not exist or cannot be opened, print an appropriate error message.
- If the output file cannot be opened, print an appropriate error message.

- If an input line has a non-numeric entry for a score, print an error message to the screen with the bad data and skip that score.
- If an input line has no scores, print an appropriate message to the screen.

```
Juan Fulano: 88, 82, 89
Tran Thi B: 91, 87.5, 92, 89, 88.5
Jan Kovacs: 91, 93, 8r, 74.5
Joseph Schmegeggie:
Erika Mustermann: 79.5, 83.5, 90, 92
```

Here is the result of running the program several times. First, with a bad input file name:

```
Enter input file name with test scores: nosuchfile.txt
Cannot find input file nosuchfile.txt
```

With a good input file name but a bad output file name. Notice that the message tells why the output file failed:

```
Enter input file name with test scores: sampleScores.txt
Enter output file name for averages: /usr/output.txt
IO Error: /usr/output.txt (Permission denied)
```

With a good input and output file name:

```
Enter input file name with test scores: sampleScores.txt
Enter output file name for averages: output.txt
Ignoring bad number  8r for Jan Kovacs
No numbers found on line Joseph Schmegeggie:
File output.txt written successfully.
```

After the program finishes successfully, the *output.txt* file looks like this:

```
Juan Fulano: 86.33 (3 tests)
Tran Thi B: 89.60 (5 tests)
Jan Kovacs: 86.17 (3 tests)
Erika Mustermann: 86.25 (4 tests)
```

Some hints:

- Do not presume that the input file will always be named *sampleScores.txt* or the output file will always be named *output.txt*—the user gets to decide those names, not you!
- If a person has scores, but none of them are numeric, then they will have no tests to average. Your program needs to handle that situation correctly.
- Use the `split` method in the `String` class to separate the name from the scores, and use it again to separate the individual scores. You will need to convert the resulting strings to `double`. Do this by using the `Double.parseDouble` method, which throws a `NumberFormatException` if given a non-numeric string as its parameter.

Exercise 18.2 Are you tired of having to write all the getter and setter methods when you create a new `class`? In this exercise, your program will ask for the name of a file that contains Java declarations, like this:

```
public int age;
private double[] weights;
```

It will then ask for an output file name and create that file with the getters and setters:

```
public int getAge() {
    return age;
}

public void setAge(int age) {
    this.age = age;
}

public double[] getWeights() {
    return weights;
}

public void setWeights(double[] weights) {
    this.weights = weights;
}
```

You may presume that the input file is formatted as follows:

1. Each line will begin with either `public` or `private`, without leading blanks.
2. The data type does not contain any blanks.
3. There is only one blank between the items on a line, and the data type does not contain any blanks. (These three items allow you to do `string.split(" ")` to get the individual parts of the declaration.)
4. Each line always ends with a semicolon.

If the input file has blank lines, ignore them.

As in the preceding exercise, use exception handling to make sure that:

- If the input file does not exist or cannot be opened, print an appropriate error message.
- If the output file cannot be opened, print an appropriate error message.

Hint: You can use the `replaceAll` method in the `String` class to get rid of the semicolon. The `replaceAll` method takes two parameters: a pattern to look for and a replacement string. For example, to replace all occurrences of `"ab"` with the empty string `""`:

```
String original = "abracadabra";
String result = original.replaceAll("ab", "");
// result will be "racadra"
```

Exercise 18.3 In this exercise, you will consolidate data from one file to create another file in a program named *Consolidate.java*. The original data is in file *ch18/wildfires_jan_2015.txt* and was extracted from a database of 1.8 million wildfires from 1992 to 2015.[4]

Each line in the file contains:

- the year

[4]Short, Karen C. 2017. Spatial wildfire occurrence data for the United States, 1992-2015 [FPA_FOD_20170508]. 4th Edition. Fort Collins, CO: Forest Service Research Data Archive. https://doi.org/10.2737/RDS-2013-0009.4

- the day of the year (1-365) the fire was discovered
- fire perimeter (acres)
- cause of fire (1 = lightning, 7 = arson, anything else is "other")

The first few lines of the file look like this:

```
2015,1,2,2
2015,1,0,2
2015,1,1,2
2015,1,0,3
2015,1,0,3
2015,1,3,4
2015,1,0,4
```

Your program will read this file and create a new file where each line contains, separated by commas:

- The year
- The day of year
- The number of lightning-caused fires and their total acreage
- the number of arson-caused fires and their total acreage
- The number of fires with other causes and their total acreage

The first few lines of the output file look like this:

```
2015,1,0,0,2,3,57,260
2015,2,2,1,1,0,35,13
2015,3,1,0,3,1,26,123
2015,4,0,0,2,33,21,255
```

Your program will read the input and output file names from the command line, so it could be invoked as follows:

```
java Consolidate wildfires_jan_2015.txt wildfire_summary.txt
```

If the user gives too many or too few command line arguments, give an appropriate "usage" message. Your program should give an appropriate message if the input file cannot be opened or the output file cannot be created. As in the preceding exercise, use `split` to separate the items, and check that data can be converted to `double` by `catch`ing `NumberFormatException`.

Exercise 18.4 In this exercise, you will write two classes: a class that represents information about a city and a main program that processes that information. You can find some pseudocode for this exercise that may help you get the program organized and started in the repository, file *ch18/CityInfo_Pseudocode.java*.

The `City` class

First, write the `City` class that defines an object with these properties:

- `private String country`: the two-letter country code
- `private String name`: the city name
- `private int population`: the city's population

Implement these methods:

`public City(String country, String name, int population)`
: The constructor

Accessors and mutators
: Write an accessor (getter) and mutator (setter) for the population, and only a getter for the country code and city name; once the object is constructed, the country and city name never change. These must be `public`.

`public String toString()`
: Returns a string giving the country name, city name, and population, separated by semicolons.

Figure 18.1 shows a UML diagram for the `City` class.

The `CityInfo` class

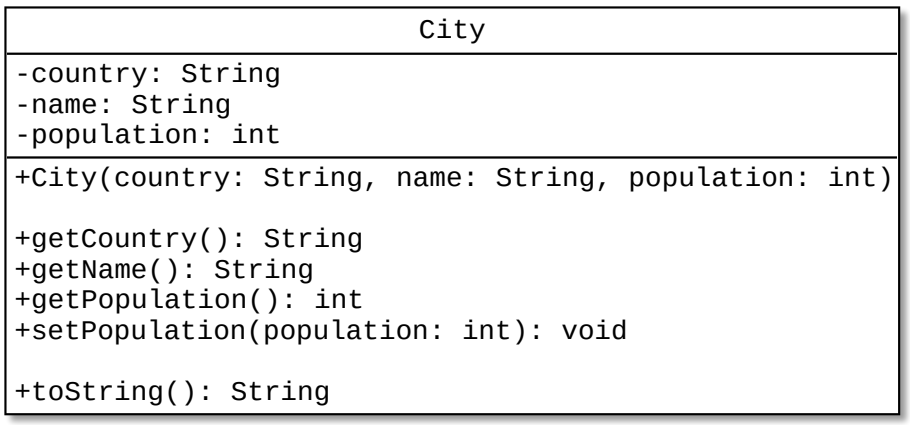

Figure 18.1: UML diagram showing attributes and methods of the City class

Then, write a class named `CityInfo` that contains the `main()` method.Use the *citylist.dat* file[5]to set up the initial list of cities. The file is available from the repository at *ch18/citylist.dat* Each line of the text file has a country code, city name, and population, separated by semicolons. Your program must first read the city data file with this method:

```
public static ArrayList<City> readCityFile(String fileName)
```

This method will open the given `fileName`, read it, and return an `ArrayList` of `City` objects corresponding to each line of the file.

If a line in the file has bad data (non-numeric population or too many or too few entries), your program will print an error that displays the bad line without entering any new data into the `ArrayList`. The `citylist.dat` file in the repository purposely contains some lines with bad data so you can test to see if your error-handling code works. Do not write your code for this specific file; another person's file might have bad data on different lines with different data on them.

If the file does not exist, the method prints an appropriate error message and returns an empty `ArrayList<City>`.

[5]The data for this exercise came from http://geonames.org/, licensed under a Creative Commons Attribution 4.0 License (see https://creativecommons.org/licenses/by/4.0/)

If the returned `ArrayList` is not empty, the program will repeatedly ask for a country code (or ENTER to quit). Once the user enters a country code, the `main()` method will call this method:

```
public static int statistics(String countryCode,
   ArrayList<City> cityList)
```

The `statistics()` method will go through the city list and

- Calculate the total number of cities in the given country.
- If there are more than zero cities:
  - Print the total number of cities
  - Print the average population for those cities
- Otherwise, prints an appropriate error message.
- Returns the total number of cities found.

The `main()` method will then use this returned value. If it is greater than zero, you will write a new file name *CC.dat*, where *CC* stands for a country code. This file will have the information for the cities in that country in the same format as `citylist.dat`. For example, if the country code is `JP`, your program will create a file named *JP.dat*. You will use a method named

```
public static void writeCountryFile(String countryCode,
    ArrayList<City> cityList)
```

to do this. If there is an `IOException` while opening the output file or writing the data, print an appropriate error message. You must use the exception's `getMessage()` method when constructing your error message. If the file is written successfully, print a message letting the user know that the file has been created. This message *must* contain the file name.

Handling Exceptions

Your program must handle these exceptions:

- Catch `FileNotFoundException`when opening a `Scanner` for the input file. You could use the `File.exists()` method, but let's use exceptions to get more practice.)

- Catch `NumberFormatException` when reading the input file and converting strings to numbers.
- You will probably need nested `try`/`catch` blocks for file input: one outside the read loop when opening the file, and one inside the read loop to skip badly formatted lines.
- Catch `IOException` when writing an output file.

Here are some things to note:

- You must accept input in either upper or lower case.
- You will need to loop through the `cityList` twice; once in `statistics()` to get the total and average (if applicable), and again in `writeCountryFile()` to create the output file. This is a design decision—I decided these are two separate tasks to be done by two methods, rather than combining both tasks into one method.

Here is what the program output might look like. Your output does not have to look exactly like this, but it must reflect the same information.

```
Reading city file...
"AB;Too few" does not have three entries.
"CD;Too many;1056382;extra" does not have three entries.
"EF;Bad number;one million" does not have a number on it.

Enter a two-letter country code, or press ENTER to quit: UA
Number of cities in UA: 5
Average population is 1,457,504.
File UA.dat written successfully.
Enter a two-letter country code, or press ENTER to quit: LU
No cities found in LU.
Enter a two-letter country code, or press ENTER to quit: MX
Number of cities in MX: 11
Average population is 2,427,622.
File MX.dat written successfully.
Enter a two-letter country code, or press ENTER to quit:
```

Here are the contents of the file *UA.dat*

```
UA;Kiev;2797553
UA;Kharkiv;1430885
UA;Dnipropetrovsk;1032822
UA;Donets'k;1024700
UA;Odessa;1001558
```

Here is an example of the output you might get for an error while writing the file (I deliberately caused this error by changing my directory to be read-only).

```
Error writing FR.dat
FR.dat (Permission denied)
```

Exercise 18.5 In this exercise, you will write two classes: a class that represents a bank account and a program that works like an ATM for bank accounts. The code repository has some pseudocode in file *ch18/ATM_Pseudocode.java* that may help you get the program organized and started.

The `Account` class

First, create an `Account` class. An `Account` object has these properties:

- `private int acctNumber`: the account number
- `private String name`: the account holder's name
- `private double balance`: the current balance in the account

Implement these methods:

`public Account(int acctNumber, String name, double balance)`
: The constructor

`public String toString()`
: Returns a string giving the account ID, name, and balance, separated by colons. Do not use `format()` on the balance; you want to keep the number as accurate as possible.

`public void deposit(double amount)`
: Adds the given amount to the current balance. If the amount is negative, the balance must not be changed; otherwise, the `balance` property is updated to reflect the deposit.

`public void withdraw(double amount)`

Withdraws the given amount from the current balance. If the amount is negative or greater than the current balance, the balance must not be changed; otherwise, the `balance` is updated to reflect the withdrawal.

You must also write a getter and setter for the balance, and only a getter for the name and account number; once the account is constructed, the name and account number never change. The getters and setters must be `public`.

Note that `deposit()` and `withdraw()` do not print error messages if they get incorrect input; they simply ignore it. It is up to the program that calls these functions to provide the error messages for the user of the program.

The `Customer` class

Next, write a program named `Customer.java` that has a `main()` method. This class provides an "ATM"-like interface to a set of bank accounts. Use the following *accounts.dat* file to set up the initial set of accounts. Each line of the text file has an account ID, name, and starting balance, separated by colons. Your program must first read the account data file and build an `ArrayList` of `Account` objects corresponding to each line. If the file does not exist, your program must output a reasonable error message and then quit.

```
15725:Christina Plaka:456.71
23981:Roz Chast:1853.22
57012:Georges Remi:3571.85
46287:Raquel Corcoles:783.00
31954:Eiichiro Oda:854.02
84373:Scott McCloud:2733.96
```

If the file is read successfully, the program will repeatedly ask for an account number (or ENTER to quit). This is equivalent to inserting an ATM card. Repeat until the account number is valid. Hint: write a method like this:

```
public static int findIndex(ArrayList<Account> accountList,
   int accountNumber)
```

This method will go through the `accountList` and return the index of the account with the given `accountNumber`, or -1 if the account number does not belong to any of the accounts in the array.

Print a message that greets the customer by name. Then repeatedly ask the customer if they want to deposit, withdraw, or finish the transactions.

- If the customer wishes to deposit, ask for the amount until you get a number greater than or equal to zero; then perform the transaction and display the balance. Print an appropriate message for invalid input. You must handle exceptions here.
- If the customer wishes to withdraw, ask for the amount until you get a number greater than or equal to zero and less than or equal to the current balance; then perform the transaction and display the balance. Print an appropriate message for invalid input. You must handle exceptions here.
- If the customer is finished, print a message to say goodbye to the customer, write the entire account array back to disk, and return to the account number prompt. (This is equivalent to giving the customer their card back).

This program must print all monetary amounts preceded by a dollar sign and with two digits after the decimal point. This is where you *do* want to use `format` to round to two decimal places.

Handling Exceptions

Your program must handle exceptions when opening the *accounts.dat* file. It must produce a reasonable error message and quit rather than crashing if the file doesn't exist.

Your program must also handle exceptions if the user enters invalid input such as"fifteen" for the account number or amount to deposit/withdraw.

The exceptions you will probably want to use the most are `java.IO.IOException` for file errors and `NumberFormatException` for invalid numbers.

Here is what you might see as output when you run your program:

```
Enter your account number: 12345
Unknown account number
Enter your account number: five
five is not a number
Enter your account number: 57012
Hello, Georges Remi!
Your current balance is $3571.85
D)eposit, W)ithdraw, or F)inish? D
Enter amount to deposit: $-20
You cannot deposit a negative amount.
D)eposit, W)ithdraw, or F)inish? D
Enter amount to deposit: $fifty
fifty is not a valid number.
D)eposit, W)ithdraw, or F)inish? D
Enter amount to deposit: $50
Your current balance is $3621.85
D)eposit, W)ithdraw, or F)inish? w
Enter amount to withdraw: $-200
You cannot withdraw a negative amount.
D)eposit, W)ithdraw, or F)inish? w
Enter amount to withdraw: $570.00
Your current balance is $3051.85
D)eposit, W)ithdraw, or F)inish? f
Goodbye, Georges Remi.

Enter your account number: 23981
Hello, Roz Chast!
Your current balance is $1853.22
D)eposit, W)ithdraw, or F)inish? w
Enter amount to withdraw: $1890
You cannot withdraw more than you have.
D)eposit, W)ithdraw, or F)inish? w
Enter amount to withdraw: $80
Your current balance is $1773.22
D)eposit, W)ithdraw, or F)inish? f
Goodbye, Roz Chast.

Enter your account number:
ATM program concludes.
```

After you finish running this program, the *accounts.dat* file will have the new balance(s):

```
15725:Christina Plaka:456.71
23981:Roz Chast:1773.22
57012:Georges Remi:3051.85
46287:Raquel Corcoles:783.0
31954:Eiichiro Oda:854.02
84373:Scott McCloud:2733.96
```