Joshua Crotts

Learning Java

A Test-Driven Approach

May 14, 2024

Springer Nature

To my Viola.

Preface

A course in Java programming is multifaceted. That is, it covers several core concepts, including basic datatypes, strings, simple-to-intermediate data structures, object-oriented programming, and beyond. What is commonly omitted from these courses is the notion of proper unit testing methodologies. Real-world companies often employ testing frameworks to bullet-proof their codebases, and students who leave university without expose to such frameworks are behind their colleagues. Our textbook aims to rectify this delinquency by emphasizing testing from day one; we write tests before designing the implementation of our methods, a fundamental feature of test-driven development.

In our book, we *design* methods rather than write them, an idea stemming from Felleisen's *How to Design Programs*, wherein we determine the definition of our data, the method signature (i.e., parameter and return types), appropriate examples and test cases, and only then follow with the method implementation. Diving straight into a method implementation often results in endless hours of debugging that may have been saved by only a few minutes of preparation. Extending this idea into subsequent computer science courses is no doubt excellent.

At Indiana University, students take either a course in Python or the Beginner/Intermediate Student Languages (based on Scheme/Racket), the latter of which involves constant testing and remediation. Previous offerings of the successor course taught in Java lead students astray into a "plug and chug" mindset of re-running a program until it works. Our goal is to stop this once and for all (perhaps not truly "once and for all," but rather we aim to make it a less frequent habit) by teaching Java correctly and efficiently.

Object-oriented programming (and, more noteworthy, a second-semester computer science course) is tough for many students to grasp, but even more so if they lack the necessary prerequisite knowledge. Syntax is nothing short of a different way of spelling the same concept; we reinforce topics that students should already have exposure to: methods, variables, conditionals, recursion, loops, and simple data structures. We follow this with the Java Collections API, generics, object-oriented programming, exceptions, I/O, searching and sorting algorithms, algorithm analysis, and modern Java features such as pattern matching and concurrency.

The ordering of topics presented in a Java course is hotly-debated and has been ever since its creation and use in higher education. Such questions include the location of object-oriented programming principles: do we start off with objects or hold off until later? Depending on the style of a text, either option can work. Even though we, personally, are more of a fan of the "early objects" approach, and is how we learned Java many moons ago, we choose to place objects later in the curriculum. We do this to place greater emphasis on testing, method design, recursion, and data structures through the Collections API. Accordingly, after our midterm (roughly halfway through the semester), students should have a strong foundation of basic Java syntax sans objects and class design. The second half of the class is dedicated to

just that: object-oriented programming and clearing up confusions that coincide and introduce themselves.

We believe that this textbook can be used as any standard second-semester computer science course. At the high-school level, this book can be used as a substitute for the College Board's former AP Computer Science AB course. Certainly, this text may be well-suited for an AP Computer Science A course at the high-school level, but its material goes well beyond the scope of the AP exam and curriculum. Instructors are free to omit certain topics that may have been covered in a prerequisite (traditionally-styled) Java course. In those circumstances, it may be beneficial to dive further into the chapters on algorithm analysis and modern Java pragmatics. For students without a Java background (or instructors of said students), which we assume, we take the time to quickly yet effectively build confidence in Java's quirky syntax. Additionally, we understand that our approach to teaching loops (through recursion and a translation pipeline) may appear odd to some long-time programmers. So, an instructor may reorder these sections however they choose, but we strongly recommend retaining our chosen ordering for pedagogical purposes, particularly for those readers that are not taking a college class using this text.

At the end of Chapters 1–5, we present a slew of exercises to the readers, and we encourage them to do most, if not all, of the exercises. Some tasks in the exercises serve as simple reinforcement of topics, whereas others are long-haul marathons that take many hours to complete. Several exercises have also been used as (written) exam questions. We do not provide answers to the exercises because there are many "avenues to success." Readers should collaborate with others to solve the problems and discuss difficulties and points of confusion in aims of clarification.

The diction of our book is chosen very carefully, with hourly of scrutiny dedicated to the paragraphs, sentence structure, and their accompanying presentation. When we demonstrate an example, stop, and closely follow along. Do not rush through it. The words on the text remain in place, no matter the pace of the reader. A page, example, exercise, or chapter may require multiple passes to fully digest the content. Make notes in the margins, type out any examples and exercises, ask questions, and answer the questions posed by others. Perhaps they may answer one of yours.

Once again, by writing this book, we wish to ensure that students are better prepared for the more complex courses in a common computer science curriculum, e.g., data structures, operating systems, algorithms, programming languages, and whatever else lies ahead. A strong foundation keeps students motivated and pushes them to continue even when times are arduous, which we understand there to be plenty thereof. After all, not many moons ago were we in the shoes of our target audience!

Have a blast!
Joshua Crotts

Acknowledgements

I piloted *Learning Java* on the students in the Fall 2023 and Spring 2024 semesters of CSCI-C 212 (Introduction to Software Systems) at Indiana University. The inspiration came from those in the former, which drove me to continue writing and designing exercises. I sincerely appreciate all of the comments, suggestions, and corrections made. The following students and course staff members found mistakes: Ashley Won, Daniel Yang, Jack Liang, Shyam Makwana, and Muazzam Siddiqui.

Contents

1 Exceptions and I/O

Abstract With the foundations of Java, data structures, and classes/objects covered, we now move into more advanced topics, such as exception handling and I/O. This chapter will discuss unchecked and checked exceptions, as well as how to handle them. We will also discuss several means of working with file I/O, including more advanced topics such as serialization. Finally, the chapter ends with an explanation of more modern Java I/O techniques.

1.1 Exceptions

Exceptions, at their core, are effect handlers. We use exceptions to identify and respond to events that occur at runtime. Java uses objects to implement an exception type hierarchy, with `Throwable` being the highest class in the chain. Any subclass or instance of `Throwable` can be thrown by Java. We will discuss several different exception types by categorizing them into one of two categories: unchecked versus checked checked exceptions.

1.1.1 Unchecked Exceptions

We handle exceptions at either compile time or runtime. The exceptions themselves are thrown at runtime, but some exceptions must be explicitly handled and referenced by the program. An *unchecked exception* is a form of exception whose behavior is dictated by the runtime system, or is caught by the programmer manually. A convenience factor of unchecked exceptions is that we do not *have* to explicitly state what happens when one is thrown. We should also note that the `RuntimeException` class serves as the superclass of all unchecked exceptions.

Example 1.1. Consider what happens when a program contains code that may or may not divide a numeric value by zero. If the bad division operation occurs, Java automatically throws an `ArithmeticException` with a relevant explanation of the problem, that being a divide-by-zero. The exception halts program execution at the point thereof, but what's interesting is that we can control the behavior of an unchecked exception through a `try/catch` block. Within a `try` block, we include the code that potentially raises the exception. In the associated `catch` block, we declare a variable for the exception we aim to catch, such as `ArithmeticException`, and then manage it within that block. Let's write a method that does nothing more than divides the sum of two numbers by the third.

```
import java.lang.ArithmeticException;

class ArithmeticExceptionExample {

  double div(int a, int b, int c) {
    return (a + b) / c;
  }

  double div2(int a, int b, int c) {
    try {
      return (a + b) / c;
    } catch (ArithmeticException ex) {
      System.err.println("div2: / by zero!");
      return 0;
    }
  }
}
```

We define two versions of `div`, where the first does not perform an explicit check for the exception, and the second does. In the latter, we print a message to the standard error stream and return zero. The preferable resolution is certainly up to the programmer, but it makes more sense in this scenario to throw the exception and halt program execution, rather than propagating a zero up to the caller. Another solution might be to return an `Optional` from the method, but the `Optional` class is more about compositionality of stream methods rather than exceptions.

Example 1.2. In the preceding example, we catch the `ArithmeticException` that Java throws. Though, suppose we have a situation in which *we* want to throw the exception. Because the `div` problem arises from a bad parameter, we might wish to throw an `IllegalArgumentException`, which designates exactly what its name suggests. We insert a conditional check to test if the divisor, namely c, is zero and, if so, we throw a new `IllegalArgumentException`. Because `IllegalArgumentException` is an unchecked exception, the caller needs not to handle nor necessarily know that it may raise the exception. Should we want to signal that as a hint, the method signature may specify that the method potentially throws an `IllegalArgumentException`. As the callee that defines the location of an exception invocation, we *only* throw the exception; it is not our responsibility to control the outcome.

We can unit test a new version of `div` by determining whether it throws an exception through the `assertThrows` and `assertDoesNotThrow` assertion methods. The thing is, though, neither `assertThrows` nor `assertDoesNotThrow` are not as simple as they appear on the surface; we need to specify *what* exception the code might throw as a reference to the class definition.[1] Additionally, the argument must be passed as an executable argument. Remember though have worked with executable constructs before via lambda/anonymous functions! Simply wrap the code that might raise an exception inside a lambda, and everything works as expected.

[1] To reference a class definition as an object, we access `.class` on the class as if it were a static method.

```
import static Assertions.assertDoesNotThrow;
import static Assertions.assertThrows;

import java.lang.IllegalArgumentException;

class IllegalArgumentExceptionExampleTester {

  @Test
  void testIllegalArgumentException() {
    assertAll(
      () -> assertDoesNotThrow(div(5, 3, 1)));
      () -> assertThrows(IllegalArgumentException.class, () -> div(5, 3, 0)),
  }
}
```

```
import java.lang.IllegalArgumentException;

class IllegalArgumentExceptionExample {

  int div(int a, int b, int c) throws IllegalArgumentException {
    if (c == 0) {
      throw new IllegalArgumentException("div: / by zero");
    } else {
      return (a + b) / c;
    }
  }
}
```

What if we wanted to call `div` from a separate method and process the exception ourselves? Indeed, this is doable. Should we wish to retrieve the exception message (i.e., the message passed to the exception constructor), we can via calling the `.getMessage` method on the exception object, which is helpful for producing custom error messages/responses or redirecting the message to a different destination.

```
import java.lang.IllegalArgumentException;

class IllegalArgumentExceptionExample {

  int div(int a, int b, int c) throws IllegalArgumentException {
    if (c == 0) {
      throw new IllegalArgumentException("div: / by zero");
    } else {
      return (a + b) / c;
    }
  }

  public static void main(String[] args) {
    try {
      double res = div(2, 3, 0);
    } catch (IllegalArgumentException ex) {
      System.err.printf("main: %s\n" ex.getMessage());
    }
  }
}
```

Example 1.3. Arrays and strings both produce unchecked exceptions when incorrectly indexed via `ArrayIndexOutOfBoundsException` and `StringIndexOutOfBoundsException` respectively, both of which inherit from the `IndexOutOfBoundsException` class. We imagine that these have both been received, in great frustration, from the readers a indeterminate number of times. An index out of bounds exception stems from accessing data beyond the permissible bounds of some collection or structure.

```
import static Assertions.assertAll;
import static Assertions.assertDoesNotThrow;
import static Assertions.assertThrows;

import java.lang.StringIndexOutOfBoundsException;

class IndexOutOfBoundsExceptionExampleTester {

  @Test
  void testOobException() {
    String ex1 = "String";
    int[] ex2 = new int[]{5, 3, 1, 2, 4, 6};
    assertAll(
      () -> assertDoesNotThrow(() -> ex1.charAt(0)),
      () -> assertDoesNotThrow(() -> ex1.charAt(ex1.length() - 1)),
      () -> assertDoesNotThrow(() -> ex2[0]),
      () -> assertDoesNotThrow(() -> ex2[ex2.length - 1])
      () -> assertThrows(StringIndexOutOfBoundsException.class, () -> ex1.charAt(17)),
      () -> assertThrows(StringIndexOutOfBoundsException.class, () -> ex1.charAt(-1)),
      () -> assertThrows(ArrayIndexOutOfBoundsException.class, () -> ex2[17]),
      () -> assertThrows(ArrayIndexOutOfBoundsException.class, () -> ex2[-1]));
  }
}
```

Another uncomfortably common unchecked exception that many Java programmers encounter is the `NullPointerException`. The `NullPointerException` most often discovered when referencing an object that has yet to be instantiated, or was accidentally never instantiated at all.

Example 1.4. Casting an object of type τ_1 to an incompatible type τ_2 results in an unchecked `ClassCastException`. By "an incompatible type," we mean to say that the object is either not an instance of the τ_2 type, or τ_1 and τ_2 are not in a *discernible* superclass/subclass relationship.[2] Primitive datatypes are not subject to this exception, as they are not objects.[3] All primitive datatypes, minus booleans, can be casted into one another. For example, the statement `int x = (int) 'A';` is valid, as is `char c = (char) 65;`. On the other hand,

```
String x = (String) new Integer(5);
```

is an invalid cast. Importantly, the cast operation (or this kind) results in a compile-time error instead of a runtime exception, because `Integer` and `String` share no discernible inheritance relationship, i.e., `Integer` is not a superclass/subclass of `String`, nor vice-versa.

We can, however, treat `List<T>` as an `AbstractList<T>` by performing a cast, such as

```
AbstractList<T> x = (AbstractList<T>) ls;
```

[2] We elaborate on discernibility below.

[3] No pun intended.

where `ls` is defined as being of type `List<T>`, because the `AbstractList` class implements the `List` interface.

Example 1.5. Sometimes, a program can reach a state where continuing is impossible or illogical. In these circumstances, we can throw an `IllegalStateException`, designating that the program has reached a point that it should not under normal pretenses. An example is attempting to access a closed `Scanner` instance.

```
Scanner in = new Scanner(System.in);
in.close();
String s = in.nextLine(); // Throws IllegalStateException!
```

1.1.2 Checked Exceptions

A *checked exception* is one that the programmer must explicitly handle. Java will fail to compile a program that does not enclose a checked exception type within a `try/catch` block, or when the method signature does not specify that it throws the exception. Almost all checked exceptions arise from I/O operations, such as reading from or to a data source, so further elaboration at this point the discussion at this point is not particularly helpful. We will discuss checked exceptions in the context of I/O operations in the following (non-sub)section.

1.1.3 User-Defined Exceptions

We can define our own exceptions in terms of other exceptions. Exceptions are nothing more than class definitions, which may be extended/inherited.

Example 1.6. Consider defining the `BadStringInputException` class, which inherits from `RuntimeException`. We might define `BadStringInputException` as an exception that Java throws when, after reading the user's input, we find that the input is not a "alpha string," i.e., a string that contains only letters. Let's define a constructor that takes a string as an parameter, serving as the exception message.[4]

```
class BadStringInputException extends RuntimeException {

  BadStringInputException(String msg) {
    super(String.format("BadStringInputException: %s", msg));
  }
}
```

Then, if we write code that reads a string from the user (through standard input), we can throw a `BadStringInputException` if said input is a non-alphabetic string. The following code segment uses the `matches` method, which receives a regular expression and returns whether the string satisfies the expression. More specifically, `[a-zA-Z]+` states that there must be at least one lowercase or uppercase character in the provided string. A method that calls `readAlphaString` does not need to handle the exception, as it unchecked and will be caught by the runtime system.

[4] We note that, broadly speaking, creating new types of exceptions is rarely beneficial, because Java provides a plethora of exception definitions that cover most use cases.

```
import java.util.Scanner;

class BadStringInputExceptionExample {

  static void readAlphaString() {
    Scanner in = new Scanner(System.in);
    String s = in.nextLine();
    if (!s.matches("[a-zA-Z]+")) {
      throw new BadStringInputException(s);
    }
  }
}
```

1.2 File I/O

Presumably this section discusses the syntax and semantics of file input and output. Although this is correct, we will explain reading data from "non-plain-text" sources such as websites and network connections through sockets.

1.2.1 Primitive I/O Classes

Example 1.7. Let's write a program that reads data from a file and echos it to standard output.

```
import java.io.IOException;
import java.io.FileNotFoundException;
import java.io.FileInputStream;

class FileInputStreamExample {

  public static void main(String[] args) {
    FileInputStream fis = null;
    String inFile = "file1.in";
    try {
      fis = new FileInputStream(inFile);
      // Read in data byte-by-byte.
      int val = -1;
      while ((val = fis.read()) != -1) {
        System.out.print(val);
      }
    } catch (FileNotFoundException ex) {
      System.err.printf("main: could not find %s\n", inFile);
    } catch (IOException ex) {
      System.err.printf("main: I/O err: %s\n", ex.getMessage());
    } finally {
      fis.close();
    }
  }
}
```

Recall that in the previous section we mentioned checked exceptions, and deferred the discussion until generalized input and output. Now that we are here, we can refresh our memory and actually put them to use. A checked exception is an exception enforced at compile-time. We emphasize the word "enforced" because the exception is not handled until runtime, but we must place the line(s) that possibly throws the checked exception within a `try/catch` block. Namely, in the preceding code, the `FileInputStream` constructor is defined to potentially throw a `FileNotFoundException`, whereas its `read` method throws a generalized `IOException` if an input malfunction occurs. Since `FileNotFoundException` is a subclass/-subexception type of `IOException`, we could omit the distinct `catch` clause for this exception.

When reading from an input source that is not `System.in`, it is imperative to always close the stream resource. So, after we read the data from our file input stream object `fis`, inside the `finally` block, we invoke `.close` on the instance to release the allocated system resources and deem the file no longer available.[5] Expanding upon the `finally` block a bit more, it is a segment of code that *always* executes, no matter if the preceding code threw an exception. The `finally` block is useful for releasing resources, e.g., opened input streams, locks, that otherwise may not be released in the event of an exception or program redirection.[6] Many programmers often forget to close a resource, and then are left to wonder why a file is either corrupted, overwritten, or some other alternative. To remediate this problem, Java provides the *try-with-resources* construct, which autocloses the resource.[7]

Example 1.8. Let's use the try-with-resources block to copy the contents of one file into another. In essence, we will design a program that opens a file input stream and a file output stream, each to separate files. Upon reading one byte from the first, we write that byte to the second.

```
import java.io.*;

class FileCopyExample {

  public static void main(String[] args) {
    try (FileInputStream fis = new FileInputStream("file1.in");
         FileOutputStream fos = new FileOutputStream("file1.out")) {
      int val = -1;
      while ((val = fis.read()) != -1) { fos.write(val); }
    } catch (FileNotFoundException ex) {
      System.err.printf("main: could not find file1.in\n");
    } catch (IOException ex) {
      System.err.printf("main: I/O err: %s\n", ex.getMessage());
    }
  }
}
```

The file input/output stream classes receive/send data as raw bytes from their source/destination streams. In most instances, we probably want to read *characters* from a data source or to a data destination rather than the raw bytes themselves. To do so, we can instead opt to use the `FileReader` class, which extends `Reader` rather than the `InputStream` class. Namely, `FileReader` is for reading textual data, whereas `FileInputStream` is for reading raw byte

[5] We can check whether an input stream is available via the `.available` method.

[6] A lock is a construct used in concurrent programs, which we will exemplify in **??**.

[7] Not every resource can be autoclosed; the class of interest must explicitly implement the `AutoCloseable` interface to be wrapped inside a try-with-resources block.

content of a file. Therefore a `FileReader` can read only textual files, i.e., files without an encoding, examples include `.pdf`, `.docx`, and so forth.

Example 1.9. Using `FileReader`, we will once again write an "echo" program to read data from its file source and writes its contents to standard output. Of course, we may want to output data to a file, in which case we use the dual to `FileReader`, that being `FileWriter`. In summary, `FileWriter` provides several methods for writing strings and characters to a data destination. In the following example we will also write some data to a test file, then examine its output based on our choice of method invocations.

```
import java.io.*;

class FileReaderWriterExamples {

  public static void main(String[] args) {
    try (FileReader fr = new FileReader("file1.in")) {
      int c = -1;
      while ((c = fr.read()) != -1) { System.out.print((char) c); }
    } catch (IOException ex) {
      ex.printStackTrace();
    }

    try (FileWriter fw = new FileWriter("file2.out")) {
      fw.write("Here is a string");
      fw.write("\nHere is another string\n");
      fw.write(9);
      fw.write(71);
      fw.write(33);
    } catch (IOException ex) {
      ex.printStackTrace();
    }
  }
}
```

If we open `file2.out`, we see that it contains `"Here is a string"` on one line, followed by `"Here is another string"` on the next line. Then, we might expect it to output the numeric strings `"9"`, `"71"`, and `"33"` all on the same line. The `write` method coerces (valid) integers into their ASCII character counterparts, meaning that the file contains the tab character, an uppercase `'G'`, and the exclamation point `'!'`. As we will soon demonstrate, working directly with `FileReader` and `FileWriter` is rarely advantageous.

The problem with the file input and output stream classes, as well as the file reader and writer classes, is that they interact directly with the operating system using low-level operations. Constantly calling these low-level operations is expensive on the CPU because they read individual bytes, sequentially, which is horribly inefficient. The `BufferedReader` and `BufferedWriter` classes aim to alleviate this problem by utilizing data buffers. A better approach is to read data in chunks, rather than byte-by-byte, to reduce the number of operating system-level calls. When the allocated buffer is full, the data within is flushed to either the source or destination. By chunking the data in this manner, we reduce the number of times that the program interacts with the operating system, which consequently reduces the CPU overhead. To read from a stream using buffers, we use the `BufferedReader` class. Its constructor receives a `Reader` instance, which may be one of several classes. For example, to read from a file, we provide a `FileReader` to the `BufferedReader` constructor. Wrapping a `FileReader` inside a `BufferedReader` allows the buffered reader to interplay (using its optimization techniques) with the file reader, which in turn interacts with the operating system.

BufferedReader Methods

The `BufferedReader` class provides methods for reading from a data source using a buffered mechanism.

R `= new BufferedReader(new FileReader(`f`))` creates a new buffered reader instance that reads from the file f, where f is either a `String` or a `File` object.
`int` R`.read()` reads a single character from the input stream R. Calling `read` advances the location of the file pointer by one byte. If the stream is empty or reads an `EOF` character, returns -1.
`String` R`.readLine()` reads a line of text from the input stream R. Calling `readLine` advances the location of the file pointer to the next line. If the stream is empty or has no further lines to consume, returns `null`.
`void` R`.close()` closes the input stream R.

Fig. 1.1: Useful `BufferedReader` Methods.

BufferedWriter Methods

The `BufferedWriter` class provides methods for writing to a data source using a buffered mechanism.

W `= new BufferedWriter(new FileWriter(`f`))` creates a new buffered writer instance that writes to the file f, where f is either a `String` or a `File` object.
`void` W`.write(`s`)` writes a string s to the output stream W.
`void` W`.close()` closes the output stream W.

Fig. 1.2: Useful `BufferedWriter` Methods.

To output data using buffers, we use the analogous `BufferedWriter` class, which receives a `Writer` instance.

Example 1.10. Using `BufferedReader` and `BufferedWriter`, we will write a program that reads data from a file and outputs it to another file.

```
import java.io.*;

class BufferedReaderWriterExample {

  public static void main(String[] args) {
    try (BufferedReader br = new BufferedReader(
                              new FileReader("file1.in"));
         BufferedWriter bw = new BufferedWriter(
                              new FileWriter("file1.out"))) {
      String line = null;
      while ((line = br.readLine()) != null) { bw.write(line + "\n"); }
    } catch (IOException ex) {
      ex.printStackTrace();
    }
  }
}
```

The benefits of buffered I/O are not obvious to us as the programmers who use these classes. We can, however, directly compare the execution time of buffered I/O to non-buffered

PrintWriter Methods

The `PrintWriter` class provides utility methods for writing different types of data to a data destination.

`PrintWriter P = new PrintWriter(new FileWriter(f))` creates a new print writer instance that writes to a file f, where f is either a `String` or a `File` object.
`void P.print(x)` writes the string representation of x to the output stream P.
`void P.println(x)` writes the string representation of x to the output stream P, followed by a newline character.
`void P.printf(f, x)` writes a formatted string to the output stream P, where f is a format string and x is the value to be formatted.
`void P.close()` closes the output stream P.

Fig. 1.3: Useful `BufferedReader` and `BufferedWriter` Methods.

I/O operations. The following code shows two implementations of reading the contents of a very large file and echoing them to another. We have defined two methods: `buffered` and `nonbuffered`, which utilize the `BufferedReader/Writer` and `FileInput/OutputStream` classes respectively. Upon testing, we see that the buffered variant takes around three seconds to finish, whereas the nonbuffered version took over four minutes!

```
import java.io.*;

class PerformanceExamples {

  private static void buffered() {
    try (BufferedReader br = new BufferedReader(
                              new FileReader("huge-2m-file.txt"));
         BufferedWriter bw = new BufferedWriter(
                              new FileWriter("bigfile.out"))) {
      int c = -1;
      while ((c = br.read()) != -1) { bw.write(c); }
    } catch (IOException ex) { ex.printStackTrace(); }
  }

  private static void nonbuffered() {
    try (FileInputStream br = (new FileInputStream("huge-2m-file.txt"));
         FileOutputStream bw = (new FileOutputStream("bigfile.out"))) {
      int c = -1;
      while ((c = br.read()) != -1) { bw.write(c); }
    } catch (IOException ex) { ex.printStackTrace(); }
  }
}
```

The classes that we have explored thus far are primarily for reading/writing either binary or text data. Perhaps we want to output values that are not strictly strings or raw bytes, e.g., integers, doubles, floats, and other primitives datatypes. To do so, we can instantiate a `PrintWriter` instance, which itself receives an instance of the `Writer` class. A concern for some programmers may be that we lose the performance benefits of buffered I/O, but this is not the case; the constructor for `PrintWriter` wraps the writer object that it receives in an instantiation of a `BufferedWriter` object. Therefore, we do not forgo any performance gains from buffered writing, while gaining the ability to write non-strictly-text data.

Example 1.11. Using `PrintWriter`, let's output some arbitrary constants and formatted strings to a file.

```
import java.io.*;

class PrintWriterExample {

  public static void main(String[] args) {
    try (PrintWriter pw = new PrintWriter(new FileWriter("file4.out"))) {
      pw.println(Math.PI);
      pw.println(false);
      pw.printf("This is a %s string with %c and %d and %f and %b\n",
                "formatted", '&', 42, Math.E, true);
    } catch (IOException ex) {
      ex.printStackTrace();
    }
  }
}
```

And thus the contents of `file4.out` are, as we might expect:

```
3.141592653589793
false
This is a formatted string with & and 42 and 2.718282 and true
```

We now have methods for reading strings and raw bytes, as well as methods for outputting all primitives and formatted strings to data destinations. We still have one problem: how can we output the representation of an object? For example, take the `BigInteger` class; it has associated instance variables and fields that we also need to store. For this particular class, it might be tempting to store a stringified representation, but this is not an optimal solution because, what if a class has a field that itself is an object? We would need to recursively stringify the object, which is prone to mistakes and requires updating the "stringification" any time a field is added, removed, or altered. Instead, we can use the `ObjectOutputStream` and `ObjectInputStream` classes, which *serialize* and *deserialize* objects respectively. Serialization is the process of converting an object into a stream of (transmittable) bytes, whereas deserialization is the opposite process. In summary, when we serialize objects, we save the object itself, alongside any relevant information about the object, e.g., its fields, instance variables, and so forth. Upon deserializing said object, we can restore the object to its original state, initializing its fields.

Example 1.12. Let's use `ObjectInput/OutputStream` classes to serialize an object of type `Player`, which has a name, score, health, and array of top scores. To designate that an object can be serialized, it must implement the `Serializable` interface. `Serializable` is a *marker interface*, meaning that it has no methods to implement. Instead, it is a "flag," of sorts, that informs the compiler that the class can be serialized. The `ObjectStreamExample` class defines the private and static `Player` class as described above. Should we open the `player.out` file, we see that it contains incomprehensible data; the data within is intended to be read only by a program.

Scanner Constructor Methods

The `Scanner` class has several constructors for reading from different data sources.

`S = new Scanner(System.in)` instantiates a scanner that reads from the standard input stream.
`S = new Scanner(f)` instantiates a scanner that reads from the file f, where f is a `File` object.
`void S.close()` closes the input scanner S.

Fig. 1.4: Useful `Scanner` Constructors.

```
import java.io.Serializable;

class ObjectStreamExample {
  // ... previous code not shown.

  private static class Player implements Serializable {

    private String name;
    private int score;
    private int health;
    private double[] topScores;

    Player(String name, int score, int health, double[] topScores) {
      this.name = name;
      this.score = score;
      this.health = health;
      this.topScores = topScores;
    }

    @Override
    public String toString() {
      return String.format("Player[name=%s, score=%d, health=%d,
                           topScores=%s]", name, score, health,
                           Arrays.toString(topScores));
    }
  }
}
```

Suppose, on the contrary, that we want to store objects as strings in a file. This has two problems: first, as we said before, we would need to recursively serialize all compositional objects of the object that we are serializing. Second, we would need to write a parser to read the stringified object and reinitialize its fields. In essence, we have to reinvent worse versions of pre-existing classes.

Example 1.13. In **??**, we saw how to use the `Scanner` class to read from standard input. Indeed, standard input is a source of data input, but we can wrap any instance of `InputStream` or `File` inside a `Scanner` to take advantage of its helpful data-parsing methods. Let's design a method that reads a series of values representing `Employee` data for a company. We just saw that we can take advantage of `Serializable` to do this for us, but it is helpful to see how we can also use a `Scanner` to solve a similar problem.

We will say that an Employee contains an employee identification number, a first name, a last name, a salary, and whether or not they are full-time staff. Each row in the file contains an Employee record.

```
class Employee {

  private long employeeId;
  private String firstName;
  private String lastName;
  private double salary;
  private boolean fullTime;

  Employee(long eid, String f, String l, double s, boolean ft) {
    this.employeeId = eid;
    this.firstName = f;
    this.lastName = l;
    this.salary = s;
    this.fullTime = ft;
  }

  @Override
  public String toString() {
    return String.format("[%d] %s, %s | %.2f | FT?=%b",
                         this.employeeId, this.lastName, this.firstName,
                         this.salary, this.fullTime);
  }
}
```

Our method returns a List<Employee> that has been populated after reading the data from the file. In particular, the nextLong, nextDouble, nextBoolean, and next methods will be helpful. The next method, whose behavior is not obvious from the name, returns the next sequence of characters prior to a whitespace.

To test, we will create a file containing the following contents:

```
123 John Smith 100000.00 false
456 Jane Doe 75000.00 true
789 Bob Jones 50000.00 false
```

```
import static Assertions.assertAll;
import static Assertions.assertEquals;

import java.util.List;

class EmployeeScannerTester {

  @Test
  void testReadRecords() {
    List<Employee> emps = readRecords("employees.txt");
    assertAll(
      () -> assertEquals(emps.get(0).toString(),
                         "[123] Smith, John | 100000.00 | FT?=false"),
      () -> assertEquals(emps.get(1).toString(),
                         "[456] Doe, Jane | 75000.00 | FT?=true"),
      () -> assertEquals(emps.get(2).toString(),
                         "[789] Jones, Bob | 50000.00 | FT?=false"));
  }
}
```

Scanner Querying Methods

The `Scanner` class has several methods for determining the type of data that is next in the input stream.

`boolean` S`.hasNext()` returns `true` if the scanner has another token in its input.
`boolean` S`.hasNextInt()` returns `true` if the scanner has another integer in its input.
`boolean` S`.hasNextDouble()` returns `true` if the scanner has another double in its input.
`boolean` S`.hasNextBoolean()` returns `true` if the scanner has another boolean in its input.
`boolean` S`.hasNextLine()` returns `true` if the scanner has another line in its input.

Fig. 1.5: Useful `Scanner` Querying Methods.

```
import java.util.Scanner;
import java.util.List;
import java.util.ArrayList;
import java.io.File;
import java.io.IOException;

class EmployeeScanner {

  /**
   * Reads in a list of employee records from a given filename.
   * @param fileName - name of file.
   * @return list of Employee instances.
   */
  static List<Employee> readRecords(String fileName) {
    List<Employee> records = new ArrayList<>();

    try (Scanner f = new Scanner(new File(fileName))) {
      while (f.hasNextLine()) {
        long eid = f.nextLong();
        String fname = f.next();
        String lname = f.next();
        double s = f.nextDouble();
        boolean ft = f.nextBoolean();
        records.add(new Employee(eid, fname, lname, s, ft));
      }
    } catch (IOException ex) { ex.printStackTrace(); }

    return records;
  }
}
```

At this point, we have seen several methods and classes for reading data from different data sources. Let's now write a few more meaningful programs.

Example 1.14. Let's write a program that reads a file containing integers, then outputs (to another file) the even integers. Because our input file has only integers, we can use the `Scanner` class for reading the data and `PrintWriter` to output those even integers. To make the program a bit more interesting, we will read the input file from the terminal arguments, and output the even integers to a file whose name is the same as the input file, but instead with the `.out` extension.

Scanner Methods

The `Scanner` class has several methods for reading different types of data from its input stream.

`String` S`.next()` returns the next token from the scanner. Any leading whitespace is skipped. Generally, this method should not be used, instead opting for one of the four methods below.

`int` S`.nextInt()` returns the next integer from the scanner. If there is a newline character following the integer, it is left in the buffer. If there is no integer to be read, throws an `InputMismatchException`.

`double` S`.nextDouble()` returns the next double from the scanner. If there is a newline character following the double, it is left in the buffer. If there is no double to be read, throws an `InputMismatchException`.

`boolean` S`.nextBoolean()` returns the next boolean from the scanner. The same rules apply as for `nextInt` and `nextDouble`.

`String` S`.nextLine()` returns the next line from the scanner. The newline character is removed from the input buffer, but *not* included in the returned string.

Fig. 1.6: Useful `Scanner` Methods.

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

class EvenIntegers {

  public static void main(String[] args) {
    if (args.length != 1) {
      System.err.println("usage: java EvenIntegers <input-file>");
      System.exit(1);
    }

    String inFile = args[0];
    String outFile = inFile.substring(0, inFile.lastIndexOf('.')) + ".out";

    try (Scanner f = new Scanner(new File(inFile));
         PrintWriter pw = new PrintWriter(new FileWriter(outFile))) {
      while (f.hasNextInt()) {
        int val = f.nextInt();
        if (val % 2 == 0) {
          pw.println(val);
        }
      }
    } catch (IOException ex) {
      ex.printStackTrace();
    }
  }
}
```

Example 1.15. Let's write a program that returns an array containing the number of lines, words, and characters (including whitespaces but excluding newlines) in a given file. The array indices correspond to those quantities respectively. To simplify the program, words will be considered strings as separated by spaces. For example, if the file contains the following contents:

```
This is a test file.
It contains three lines.
Here is the last line.
```

The returned array should be `[3, 14, 46]`. We can write JUnit tests to verify that our program works as intended.

```
import static Assertions.assertAll;
import static Assertions.assertEquals;

class LineWordCharCounterTester {

  @Test
  void count() {
    int[] counts = LineWordCharCounter.count("file1.in");
    assertAll(
      () -> assertEquals(counts[0], 3),
      () -> assertEquals(counts[1], 14),
      () -> assertEquals(counts[2], 46));
  }
}
```

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

class LineWordCharCounter {

  /**
   * Counts the number of lines, words, and characters in a given file.
   * @param fileName - name of file.
   * @return array of counts.
   */
  static int[] count(String fileName) {
    int[] counts = new int[]{0, 0, 0};
    try (Scanner f = new Scanner(new File(fileName))) {
      while (f.hasNextLine()) {
        String line = f.nextLine();
        counts[0]++;
        counts[1] += line.split(" ").length;
        counts[2] += line.length();
      }
    } catch (IOException ex) {
      ex.printStackTrace();
    }
    return counts;
  }
}
```

Example 1.16. Going further with terminal arguments, let's write a program that receives multiple file names from the terminal, and outputs a file with all of the data concatenated into one. We will throw an exception if the user passes in files that do not all share the same extension. As an example, should the user input the following:

```
java ConcatenateFiles file1.txt file2.txt file3.txt output-file.txt
```

Then the program should output a file `output-file.txt` that contains the contents of `file1.txt`, `file2.txt`, and `file3.txt`, in that order.

```
import java.io.*;
import java.util.Arrays;

class ConcatenateFiles {

  /**
   * Determines whether all files have the same extension.
   * @param files - array of file names.
   * @return true if all files have same extension, false otherwise.
   */
  private static boolean sameExtension(String[] files) {
    if (files[0].lastIndexOf('.') == -1) {
      return false;
    } else {
      String extension = files[0].substring(files[0].lastIndexOf('.'));
      for (String file : files) {
        if (file.lastIndexOf(".") == -1 ||
            !file.substring(file.lastIndexOf('.')).equals(extension)) {
          return false;
        }
      }
      return true;
    }
  }

  /**
   * Concatenates the contents of a list of files into a single file.
   * @param files - array of file names.
   * @param outFile - name of output file.
   */
  private static void concatenate(String[] files, String outFile) {
    try (BufferedWriter bw = new BufferedWriter(new FileWriter(outFile))) {
      for (String file : files) {
        try (BufferedReader br = new BufferedReader(new FileReader(file))) {
          String line = null;
          while ((line = br.readLine()) != null) {
            bw.write(line + "\n");
          }
        }
      }
    } catch (IOException ex) {
      ex.printStackTrace();
    }
  }

  public static void main(String[] args) {
    if (args.length < 3) {
      System.err.println("usage: java ConcatenateFiles <i-files> <o-file>");
      System.exit(1);
    } else {
      String[] inFiles = Arrays.copyOfRange(args, 0, args.length - 1);
      String outFile = args[args.length - 1];

      if (!sameExtension(inFiles)) {
        System.err.println("ConcatenateFiles: bad extensions");
        System.exit(1);
      } else {
        concatenate(inFiles, outFile);
      }
    }
  }
}
```

1.3 Modern I/O Classes & Methods

Aside from the aforementioned classes for working with files and I/O, Java's later versions provide methods and classes that achieve the same task as those that we might otherwise need to write several lines of code.

Example 1.17. To read the lines from a given file, we might open the file using a `BufferedReader` and `FileReader` object, read the values into some collection, e.g., a list, then process those lines accordingly. Repeatedly writing these almost identical lines of code is repetitive, so it might be a good idea to write a method that does this for us, and is an exercise that we provide to the reader. Java 8 introduced two classes: `Files` and `Path` that work with files and paths respectively. Let's use a handy method from `Files`, namely `readAllLines` to, as its name implies, read the lines from an input file and store them in a `List<String>`.

```
import java.nio.file.Files;
import java.util.List;

class ReadAllLines {

  public static void main(String[] args) {
    try {
      List<String> lines = Files.readAllLines(Path.of("test.txt")));
      // Some processing with lines...
    } catch (IOException ex) {
      ex.printStackTrace();
    }
  }
}
```

We still need to catch an `IOException` because `readAllFiles` might throw one in the event of some I/O error. What may be slightly disappointing is the fact that we cannot wrap this in a try-with-resources block, because `readAllLines` opens and closes the file it receives, resulting in what might appear to be less succinct code. Moreover, the method receives a `Path`, rather than a `String`, which we believe to be an attempt made by Java to prevent the programmer from needing to mess with strings and other input resources directly.

Example 1.18. Unfortunately, `readAllLines` is extremely memory-inefficient, requiring us to store a list of every line in the file. Consider an extremely large dataset, where the input contains one billion rows. Storing this data directly into running memory is not a particularly viable option, at least at the time of writing this text. A solution is to process each line one at a time, similar to how we work with a `BufferedReader` instance. As the section title suggests, though, there is a better way that incorporates streams into the mix. The `Files` class provides the `lines` method, which returns a stream of the lines in the file. Therefore, appealing to the lazy nature of streams, if we never actually use the data from the stream, nothing happens at all. This is a meaningless exercise, so let's write a method that solves the `1BR` challenge: given a file of data points representing measurements of temperatures in differing locations, return an alphabetized string containing the location and, separated by an equals sign, the minimum, maximum, and average temperatures across all data points for that location (Morling, 2023).

To start the exercise, let's consider our options: we have one billion rows of text in the following format: `"LOCATION;TEMP"`, so storing this in direct memory is a challenge that we will not overcome. Instead, let's create a `Map` that maps location identifiers to `Measurement` objects. A `Measurement` stores a number of occurrences, its minimum, maximum, and total-

accrued temperature. Each line we read either updates an existing `Measurement` in the map or inserts a new key/value pair.

To start, let's design the skeleton for our method, which we will name `computeTemperatures`, as well as the `Measurement` private class. Moreover, when instantiating a new `Measurement` instance, its current minimum, maximum, and total are all equal to the value on the current line.

```
import java.io.IOException;
import java.nio.file.*;
import java.util.*;

class TemperatureComputer {

  /**
   * Returns a string with the locations and their
   * minimum, maximum, and average temperatures.
   * @param filename - input file with locations and
   * temperature separated by ';'.
   * @return String formatted as aforementioned.
   */
  static String computeMeasurement(String filename) {
    // TODO.
    return null;
  }

  private static class Measurement {

    private double min, max, total;
    private int noOccurrences;

    Measurement(double t) {
      this.numOccurrences = 1;
      this.min = t;
      this.max = t;
      this.total = t;
    }

    /**
     * Adds a temperature to this measurement's total.
     * We update the minimum, maximum, total, and
     * number of occurrences respectively.
     */
    void add(double t) {
      this.noOccurrences++;
      this.total += t;
      this.min = Math.min(this.min, t);
      this.max = Math.max(this.max, t);
    }
  }
}
```

As stated, using a map is the appropriate data structure, so let's instantiate a `HashMap` due to its quick lookup times. Then, we declare, inside a try-with-resources, a `Stream<String>`, returned by the `lines` method. Once either the stream is fully consumed, the stream is closed, or the program execution finishes the try block, then the file is also closed. From the stream, we could use a traditional for-each loop, but let's use stream operations instead. For every line, we want to split it on the semicolon, retrieve the location and temperature, then update the map as necessary. Because we need to update the state of an object if it exists in the

map, we will utilize the `putIfAbsent` method, which returns the associating `Measurement` if the key-to-put already exists.

Lastly, we must conjoin the sorted pairs in the map with commas, which we can do via the `sorted()` and `Collectors.joining()` methods. In addition to this, we added a `toString` method to `Measurement` that returns a formatted string containing the minimum, average, and maximum temperatures floated to one decimal.

```
import java.io.IOException;
import java.util.stream.Stream;
import java.nio.file.*;
import java.util.*;

class TemperatureComputer {

  /**
   * Returns a string with the locations and their
   * minimum, maximum, and average temperatures.
   * @param filename - input file with locations and
   * temperature separated by ';'.
   * @return String formatted as aforementioned.
   */
  static String computeMeasurement(String filename) {
    Map<String, Measurement> mMap = new HashMap<>();
    try (Stream<String> lines = Files.lines(Path.of(filename))) {
      lines.forEach(x -> {
        String[] arr = x.split(";");
        String location = arr[0];
        double temp = Double.parseDouble(arr[1]);
        Measurement ms = mMap.putIfAbsent(location, new Measurement(temp));
        if (ms != null) {
          ms.add(temp);
        }
      });
    } catch (IOException e) {
      throw new RuntimeException(e);
    }
    return mMap.keySet()
               .stream()
               .sorted()
               .map(s -> String.format("%s=%s", s, mMap.get(s)))
               .collect(Collectors.joining(", "));
  }

  // ... other class not shown.
}
```

With inputs as large as what we assume them to be, we must make reasonable considerations with our choice of data structure. We could, theoretically, use a `TreeMap` and have the program autosort the measurement map pairs, but this is a performance penalty that is greater than using the `sorted` method as provided by the stream API over the map keys. In our tests, using a `TreeMap` amounted to a forty second performance penalty.

Example 1.19. Our last example of working with File I/O is a simple Sudoku solver. *Sudoku* is a game where the objective is to fill each row, column, and sub-grid with exactly one of possible entries, generally from 1 to 9. There are nine 3×3 subgrids that form a square, which results in a 9×9 grid.

The most straightforward way to mechanically solve a Sudoku puzzle is via a backtracking algorithm. That is, we try to place a number in a cell and, if it leads to success, we continue with the solution. Otherwise, we undo the placement and try again. We will use I/O to read in a partial Sudoku puzzle and to write the solution out to another file.

Let's write the `SudokuSolver` class, whose constructor receives a file that represents a partial Sudoku puzzle. The input specification contains nine rows and nine columns, with dots to denote a missing number. From here, we will design the `boolean solve()` method, which returns whether or not a solution exists. If there is one, it is stored in an instance variable of the class. We will also design the `void output(String fileName)` method to output the solution to a file. If there is no possible solution, the program will throw an `IllegalStateException` to indicate a failure.

```
import java.io.IOException;
import java.nio.file.*;
import java.util.*;

class SudokuSolver {

  private static final int N = 9;
  private int[][] board;
  private int[][] solution;

  SudokuSolver(String filename) {
    this.board = new int[N][N];
    this.solution = new int[N][N];
    try (Stream<String> lines = Files.lines(Path.of(filename))) {
      int row = 0;
      lines.forEach(x -> {
        for (int i = 0; i < x.length(); i++) {
          this.board[row][i] = x.charAt(i) == '.' ? 0 : x.charAt(i) - '0';
          this.solution[row][i] = this.board[row][i];
        }
        row++;
      });
    } catch (IOException e) {
      throw new RuntimeException(e);
    }
  }

  boolean solve() {
    /* TODO. */
    return false;
  }

  void output(String filename) {
    /* TODO. */
  }
}
```

Our `solve` method jump-starts a backtracking algorithm that attempts to solve the puzzle using recursion. Let's design a private helper method to receive the row r and column c of the cell to fill. If r and c are both equal to N, then we have reached the end of the board and therefore have a solution. Otherwise, we need to find the next empty cell to fill. This is a three-step process:

(i) First, if the y coordinate is equal to N, then we have reached the end of the row and need to move onto the next.

(ii) If the cell is not empty, we move onto the next cell.
(iii) If the cell is empty, we try to place a number in the cell. If the number is valid, we continue with the solution. Otherwise, we undo the placement and try again.

What does it mean for a number to be valid? A number is valid in its placement if it does not already exist in the row, column, or subgrid. Let's write another private helper method that, when given a cell at row r and column c, and a number n, determines whether or not the number is valid.

```
import java.io.IOException;
import java.nio.file.*;
import java.util.*;

class SudokuSolver {
  // ... previous code not shown.

  SudokuSolver(String filename) { /* Implementation omitted. */ }

  /**
   * Returns whether or not a number is valid in a given cell.
   * @param r - row of cell.
   * @param c - column of cell.
   * @param n - number to place in cell.
   * @return true if number is valid, false otherwise.
   */
  private boolean isValid(int r, int c, int n) {
    // Check the row and column simultaneously.
    for (int i = 0; i < N; i++) {
      if (this.board[r][i] == n || this.board[i][c] == n) {
        return false;
      }
    }

    // Check the subgrid.
    int sr = (r / 3) * 3;
    int sc = (c / 3) * 3;
    for (int i = sr; i < sr + 3; i++) {
      for (int j = sc; j < sc + 3; j++) {
        if (this.board[i][j] == n) {
          return false;
        }
      }
    }
    return true;
  }
}
```

From this we can begin to work on the recursive backtracking algorithm, using the outline we provided earlier.

```
import java.io.IOException;
import java.nio.file.*;
import java.util.*;

class SudokuSolver {
  // ... previous code not shown.

  SudokuSolver(String filename) { /* Implementation omitted. */ }

  /**
   * Returns whether or not a solution exists. If a solution does not
   * exist, the variable that stores the solution is set to null.
   * @return true if a solution exists, false otherwise.
   */
  private boolean solve() {
    if (solve(0, 0, this.solution)) {
      return true;
    } else {
      this.solution = null;
      return false;
    }
  }

  /**
   * Recursive backtracking algorithm to solve the puzzle.
   * @param r - row of cell.
   * @param c - column of cell.
   * @param sol - solution array.
   * @return true if we have a solution, and false if the current
   * placement is invalid or leads to a "dead end".
   */
  private boolean solve(int r, int c, int[][] sol) {
    if (r == N) {
      return true;
    } else if (c == N) {
      return solve(r + 1, 0, sol);
    } else if (this.board[r][c] != 0) {
      return solve(r, c + 1, sol);
    } else {
      for (int i = 1; i <= N; i++) {
        if (isValid(r, c, i)) {
          this.sol[r][c] = i;
          if (solve(r, c + 1, sol)) {
            return true;
          } else {
            this.sol[r][c] = 0;
          }
        }
      }
    }
    return false;
  }
}
```

Finally, the `output` method is straightforward. We use a `PrintWriter` to write the solution to a file. If there is no solution (meaning the solution instance variable is set to `null`), then we throw an `IllegalStateException`.

```
import java.io.IOException;
import java.nio.file.*;
import java.util.*;

class SudokuSolver {
  // ... previous code not shown.

  SudokuSolver(String filename) { /* Implementation omitted. */ }

  /**
   * Outputs the solution to a file. The solution is just a 9x9 grid of
   * numbers, and does not attempt to format the output in any way.
   * @param filename - name of output file.
   */
  void output(String filename) {
    try (PrintWriter pw = new PrintWriter(new FileWriter(filename))) {
      if (this.solution == null) {
        throw new IllegalStateException("No solution exists.");
      } else {
        for (int i = 0; i < N; i++) {
          for (int j = 0; j < N; j++) {
            pw.print(this.solution[i][j]);
          }
          pw.println();
        }
      }
    } catch (IOException e) {
      throw new RuntimeException(e);
    }
  }
}
```

Chapter Exercises

Exercise 1.1. (⋆⋆)
Design the `EchoOdds` class, which reads a file of line-separated integers specified by the user (using standard input), and writes only the odd numbers out to a file of the same name, just with the `.out` extension. If there is a non-number in the file, throw an `InputMismatchException`.

Example Run. If the user types `"file1a.in"` into the running program, and `file1a.in` contains the following:

```
5
100
25
17
2
4
0
-3848
13
```

then `file1a.out` is generated containing the following:

```
5
25
17
13
```

Example Run. If the user types `"file1b.in"` into the running program, and `file1b.in` contains the following:

```
5
100
25
17
THIS_IS_NOT_AN_INTEGER!
4
0
-3848
13
```

then the program does not output a file because it throws an exception.

Exercise 1.2. (⋆⋆)
Design the `Capitalize` class, which contains one `static` method: `void capitalize(String in)`. The `capitalize` method reads a file of sentences (that are not necessarily line-separated), and outputs the capitalized versions of the sentences to a file of the same name, just with the `.out` extension (you must remove whatever extension existed previously).

You may assume that a sentence is a string that is terminated by a period and only a period, which is followed by a single space. If you use a splitting method, e.g., `.split`, you must remember to reinsert the period in the resulting string. There are many ways to solve this problem!

Example Run. If we invoke `capitalize("file2a.in")` into the running program, and `file2a.in` contains the following *(note that if you copy and paste this input data, you will need to remove the newline before the "hopefully" token):*

```
hi, it's a wonderful day. i am doing great, how are you doing. it's
hopefully fairly obvious as to what you need to do to solve this problem.
this is a sentence on another line.
this sentence should also be capitalized.
```

then `file2a.out` is generated containing the following (*again, remember to remove the newline before "hopefully".*):

```
Hi, it's a wonderful day. I am doing great, how are you doing. It's
hopefully fairly obvious as to what you need to do to solve this problem.
This is a sentence on another line.
This sentence should also be capitalized.
```

Exercise 1.3. (⋆⋆)
Design the `SpellChecker` class, which contains one `static` method: `void spellCheck(String dict, String in)`. The `spellCheck` method reads two files: a "dictionary" and a content file. The content file contains a single sentence that may or may not have misspelled words. Your job is to check each word in the file and determine whether or not they are spelled correctly, according to the dictionary of (line-separated) words. If a word is not spelled correctly, wrap it inside brackets `[]`.

Output the modified sentences to a file of the same name, just with the `.out` extension (you must remove whatever extension existed previously). You may assume that words are space-separated and that no punctuation exist. Hint: use a `Set`! Another hint: words that are different cases are not misspelled; e.g., `"Hello"` is spelled the same as `"hello"`; how can your program check this?

Example Run. Assuming `dictionary.txt` contains a list of words, if we invoke `spellChecker("dictio` `"file3a.in")`, and `file3a.in` contains the following:

```
Hi hwo are you donig I am dioing jsut fine if I say so mysefl but I
will aslo sya that I am throughlyy misssing puncutiation
```

then `file3a.out` is generated containing the following:

```
Hi [hwo] are you [donig] I am [dioing] [jsut] fine if I say so
[mysefl] but I will [aslo] [sya] that I am [throughlyy] [misssing]
[puncutiation]
```

Exercise 1.4. (⋆⋆)
Design the `OrderWebUrls` class, which contains one `static` method: `void orderWebUrls(String in)`. The `orderWebUrls` method reads in a file of line-separated web URLs. A web URL contains a protocol separated by a colon and two forward slashes, and a host name. For example, in the URL `https://www.joshuacrotts.us`, the protocol is `https` and the host name is `www.joshuacrotts.us`. The method should read in web URLs in this specific format and sort them, lexicographically, based on the hostname. If two hostnames are identical and only differ by the protocol, then the order becomes based on the protocol.

Exercise 1.5. (⋆⋆)
Recall the `Optional` class and its purpose. In this exercise you will reimplement its behavior

with the IMaybe interface with two subtypes Just and Nothing, representing the existence and absence of a value, respectively. Design the generic IMaybe interface, which contains the following three metohds: T get(), boolean isJust(), and boolean isNothing(). The constructors of these subtypes receive either an object of type T or no parameter, depending on whether it is a Just or a Nothing. Throw an UnsupportedOperationException when trying to get the value from an instance of Nothing .

Exercise 1.6. (⋆⋆)
Redo the "Maybe" exercise, only this time implement it as an abstract class/subclass hierarchy. That is, Maybe should be an abstract class containing three abstract methods: T get(), boolean isJust(), and boolean isNothing(). The Just and Nothing classes should be subclasses of Maybe and override these methods accordingly. Do not create constructors for these classes. Instead, create static factory methods Just.of(T t) and Nothing.of() that return an instance of the appropriate class.

Exercise 1.7. (⋆⋆)
A common use for file input and output is data analysis. Design a class StatisticsDescriptor that has the following methods:

(a) void read(String fileName), which reads in a list of numbers from a file into a collection. These numbers can be integers or floating-point values.
(b) double mean(), which returns the mean of the dataset.
(c) double stddev(), which returns the standard deviation of the dataset.
(d) double quantile(double q), which receives a quantile value $q \in [0, 1]$ and returns the value such that there are q, as a percentage, values below said value. As an example, if our dataset contains 3, 2, 1, 4, 5, 10, 20, and we call quantile(0.30), then we return 2.8 to indicate that 30% of the values in the dataset are below 2.8.
(e) double median(), which returns the median, or the middle value, of the dataset.
(f) double mode(), which returns the mode, or the most-frequent value, of the dataset.
(g) double range(), which returns the range, or the difference between the maximum and minimum values of the dataset.
(h) List<Double> outliers(), which returns the numbers that are outliers of the dataset. We consider a value an outlier if it is greater than three standard deviations away from the mean. Refer to the formula for z-score calculation in the exercises from Chapter **??**.
(i) void output(String fileName), which outputs all of the above statistics to the file specified by the parameter (the order is irrelevant). You should output these as a series of "key-value" pairs separated by an equals sign, e.g., mean=X. Put each pair on a separate line.

For all methods (except read), if the data has yet to be read, throw an IllegalStateException.

Exercise 1.8. (⋆⋆)
You are teaching an introductory programming course and you want to keep a seating chart for your students. A seating chart is an arrangement of numbers $1..n$, the location in the classroom of which is defined by the instructor. Numbers that are lower in the range are closer to the front of the room. Design the SeatingChart class, which has the following methods:

(a) SeatingChart() is the constructor, which initializes the seating chart to be empty. The seating chart is represented as a List<Student>, where Student is a private and static class, inside SeatingChart, that you design. Students should have a name, a seat number,

and an accommodation parameter. The seat number is an integer, and the accommodation parameter is a boolean.

(b) `void read(String fileName)` reads in a list of students from a file into the seating chart. The file contains a list of students, one per line, with their name. A student also has an optional accommodation parameter, which means they should sit in a seat closer to the front of the room. The file is comma-separated, and if the student has an accommodation, it is represented by `true` after the student's name.

```
Alice
Bob,true
Charlie
```

(c) `void scramble()` scrambles the seating chart. That is, it randomly shuffles the students in the seating chart. This also accounts for the accommodations, so that students with accommodations are closer to the front of the room.

(d) `void alphabetize()` sorts the seating chart alphabetically by the students' names. This "mode" does not account for accommodations, and is thus strictly alphabetical.

(e) `List<Student> getStudents()` returns the seating chart as a list of students.

(f) `List<Student> getAccommodationStudents()` returns the students with accommodations as a list.

(g) `void output(String fileName)` outputs the seating chart to a file specified by the parameter. The file should contain the students' names and their seat numbers, one per line, separated by a comma. The output list should be in the order of the seating chart.

Exercise 1.9. (⋆⋆⋆)
You are designing a system for looking up car information, similar to that of, say, Kelly Blue Book or Carvana.

(a) First, design the `Car` class, which stores the make, model, color, trim, and VIN (vehicle identification number) of the car as strings, the year and number of prior owners as an integer, an enumeration that contains its title status (e.g., `Clean`, `Salvaged`, or `Rebuilt`), and its MSRP (in USD) as a floating-point value.[8]

(b) Make the `Car` class serializable, like we did in the chapter. That is, implement the `Serializable` interface and override the `writeObject` and `readObject` methods respectively.

(c) Override the `public boolean equals(Object o)` and `public int hashCode()` methods. Two `Car` objects are the same if they share the same VIN. When overriding `hashCode`, return a hash code that hashes all instance variables.

(d) Finally, override the `public String toString()` method, which returns a string similar to the following (with a tab character before each line):

```
    Make: Honda
    Model: Ac-
cord
    Color: Sil-
ver
    Trim: LX
    Year: 2007
    VIN: 1G4HDSLVRLX
```

[8] As a tip: if you are ever writing real-world software that works with currency values, you should *never* store currency as floating-point numbers, e.g., `double` or `float`. This is because of the inaccuracies that come with such representations in a computer. The preferred solution is to use an object type that separately stores cents and dollars such as `BigDecimal`.

```
Number of Previous Owners: 2
Title Status: Salvaged
MSRP: $20,475.00
```

(e) Now, design the `CarDatabase` class, whose (no parameter) constructor instantiates a `List<Car>` instance variable to store the list of cars. Then, design the following methods:

(i) `void addCar(Car car)`, which adds a car with the given values to the database.
(ii) `boolean removeCar(String vin)`, which removes a car with the given VIN. If the car was in the database, the method returns `true`, and `false` otherwise.
(iii) `boolean contains(String vin)`, which returns `true` if a `Car` with the given VIN exists in the database, and `false` otherwise.
(iv) `boolean contains(Car car)`, which returns `true` if the given car exists in the database, and `false` otherwise. Note that this method should be one line long and call the other variant of `contains`.
(v) `void readFile(String in)`, which populates the database with the `Car` objects from the given file. The file should contain only serialized `Car` objects and not plain-text.
(vi) `void writeFile(String out)`, which writes all cars in the database out to a file with the given file name. The file should contain only serialized `Car` objects and not plain-text.
(vii) `void sort(Comparator<Car> cmp)`, which sorts the database of `Car` objects according to the provided `Comparator` implementation.
(viii) `void sort()`, which sorts the database of `Car` objects according to their VIN.

Exercise 1.10. (⋆⋆)
You're interested in determining the letter statistics of a file containing text. In particular, you want to design a program that reports the frequency of each alphabetic character. Design the `LetterFrequency` class, which has the following methods:

(a) `LetterFrequency(String fileName)` is the constructor, which reads a file containing text into a `long[]` instance variable with 26 elements. Convert all upper-case letters into lower-case. Index 0 of the (frequency) array corresponds to `'a'`, and index 25 corresponds to `'z'`. Before reading the contents of the file, initialize the array to contain all zeroes.
(b) `void add(char c)` adds a character c to the frequency map. If c is not alphabetic, throw an `IllegalArgumentException`.
(c) `void add(String s)` calls the other `add` method on each character in the given string s.
(d) `long get(char c)` returns the frequency of a given character, which should be converted to lowercase. If c is not a letter, throw an `IllegalArgumentException`.
(e) `char get(int i)` returns the i^{th} most frequent character. If $i \notin [0, 25)$, throw an `IllegalArgumentException`.
(f) `List<Character> getMostFrequentChars(int n)` returns the n most frequent characters. If $n \notin [0, 25)$, throw an `IllegalArgumentException`. Hint: invoke the `get` method n times for values 1 to n inclusive.

Exercise 1.11. (⋆⋆)
This exercise has two parts. A *stack-based* programming language is one that uses a stack to keep track of variables, values, and the results of expressions. These types of languages have existed for several decades, and in this exercise you will implement such a language.

Design the `StackLanguage` class, whose constructor receives no parameters. The class contains two instance methods: `void readFile(String f)` and `double interpret()`.

- The `readFile` method reads a series of "stack commands" from the file. These can be stored however you feel necessary in the class, but you should not interpret anything in this method, nor should you throw any exceptions. You may want to create a private static class for storing commands.
- The `interpret` method interprets the stored list of instructions. If no instructions have been received by a `readFile` command, throw an `IllegalStateException`. Here are the following possible instructions:

  (a) `DECL` v X declares that v is a variable with value X.
  (b) `PUSH` X pushes a number X to the stack.
  (c) `POP` v pops the top-most number off the stack and stores it in a variable v. If v has not been declared, an `IllegalArgumentException` is thrown.
  (d) `PEEK` v stores the value at the top of the stack in the variable v. If v has not been declared, an `IllegalArgumentException` is thrown.
  (e) `ADD` X adds X to the top-most number on the stack.
  (f) `SUB` X subtracts X from the top-most number on the stack.
  (g) `XCHG` v swaps the value on the top of the stack with the value stored in variable v. If v has not been declared as a variable, an `IllegalArgumentException` is thrown.
  (h) `DUP` duplicates the value at the top of the stack.

  If the command is none of these, then throw an `UnsupportedOperationException`. You may assume that all commands, otherwise, are well-formed (i.e., they contain the correct number of arguments and the types thereof are correct). After interpreting all instructions, the value that is returned is the top-most value on the stack. If there is no such value, throw a `NoSuchElementException`.
  Hint: use a `Map` to store variable identifiers to values.

Exercise 1.12. (⋆⋆⋆)
Java provides many forms of input and output stream classes, e.g., `BufferedReader`/`BufferedWriter`. Unfortunately, it does not have classes, say `BitInputStream` and `BitOutputStream`, for outputting raw bits to a file. In this exercise you will implement these classes. [9]

(a) Design the `BitOutputStream` class, which extends `OutputStream`. It should store two instance variables: an instance of `OutputStream` and an array of eight integers. Each integer index corresponds to a bit to send to the output.

  (i) Design three `BitOutputStream` constructors: one that sets the stored output stream instance to `null`, a second that receives an `OutputStream` object and assigns it to the instance variable, and a third that receives a file name, and instantiates the stored output stream as a `FileOutputStream`. All three constructors should instantiate the array of "bits."
  (ii) Override the `public void flush() throws IOException` method from `OutputStream` to output the bits, as a single byte, to the file. This does *not* mean that you should output the raw `'1'` and `'0'` characters that are stored in the buffer. Instead, convert those bits into a single `int`, and write that value to the output stream. Hint: the bitwise operations `<<` and `|` may come in handy.
  (iii) Override the `public void write(int b) throws IOException` method from `OutputStream` to assign bit b to the next-free index in the array. If you run out of bits to store in the array, call `this.flush()`.

[9] This exercise is common in Java textbooks, and in our opinion, is worth repeating.

(iv) Design the `void writeBit(int b)` method, which adds a bit to the i^{th} index of the array. If the input b is not a 0 or 1, throw an `IllegalArgumentException`, otherwise call `this.write` with b.

(b) Design the `BitInputStream` class, which extends `InputStream`. This class should also store an instance of an `InputStream` as a field, as well as an array of bit values.

(i) Design three `BitInputStream` constructors that mimic the behavior of the `BitOutputStream` class constructors.
(ii) Design the `private int readBit()` method, which reads a bit from the buffer. Your code should call `read` on the input stream once every eight bits, i.e., every byte.
(iii) Override the `public int read()` method, which returns the next bit from the buffer. If you run out of bits to return, read a byte from the input stream. If there are no more bytes to read, return -1. Hint: `read()` itself returns -1 when there are no bytes remaining.

Exercise 1.13. ($\star\star\star$)
A maze is a grid of cells, each of which is either open or blocked. We can move from one free cell to another if they are adjacent. Design the `MazeSolver` class, which has the following methods:

(a) `MazeSolver(String fileName)` is the constructor, which reads a description of a maze from a file. The file contains a grid of characters, where `'.'` represents an open cell and `'#'` represents a blocked cell. The file is formatted such that each line is the same length. Read the data into a `char[][]` instance variable. You may assume that the maze dimensions are on the first line of the file, separated by a space.
(b) `char[][] solve()` returns a `char[][]` that represents the solution to the maze. The solution should be the same as the input maze, but with the path from the start to the end marked with `'*'` characters. The start is the top-left cell, and the end is the bottom-right cell. If there is no solution, return `null`.
We can use a backtracking algorithm to solve this problem: start at a cell and mark it as visited. Then, recursively try to move to each of its neighbors, marking the path with a `'*'` character. If you reach the maze exit, then return `true`. Otherwise, backtrack and try another path. By "backtrack," we mean that you should remove the `'*'` character from the path. If you have tried all possible paths from a cell and none of them lead to the exit, then return `false`. We provide a skeleton of the class below.
(c) `void output(String fileName, char[][] soln)` outputs the given solution to the maze to a file specified by the parameter. Refer to the above description for the format of the output file and the input `char[][]` solution.

```
class MazeSolver {

  private final char[][] MAZE;

  MazeSolver(String fileName) { /* TODO read maze from file. */ }

  /**
   * Recursively solves the maze, returning a solution if it exists,
   * and null otherwise. We use a simple backtracking algorithm
   * in the helper.
   * @return a solution to the maze, or null if it does not exist.
   */
  char[][] solve() {
```

```
    char[][] soln = new char[MAZE.length][MAZE[0].length];
    return this.solveHelper(0, 0, soln) ? soln : null;
  }

  /**
   * Recursively solves the maze, returning true if we ever reach
   * the exit. We try all possible paths from the current cell, if
   * they are reachable. If a path ends up being a dead end, we
   * backtrack and try another path.
   * @param r - the row of the current cell.
   * @param c - the column of the current cell.
   * @param sol - the current solution to the maze.
   * @return true if we are at the exit, false otherwise.
   */
  private boolean solveHelper(int r, int c, char[][] sol) {
    /* TODO. */
  }
}
```

Exercise 1.14. (⋆⋆⋆)
The `cut` program is a command-line tool for extracting pieces of text from data. For this exercise, you will implement a very basic version of the program that reads data from the terminal.

(a) First, add support for the `-c` $X, Y, \ldots, Z$ flag, which outputs the characters at positions $X, Y, \ldots, Z$ in each line. If any number is less than 1, throw an `IllegalArgumentException`.
(b) Second, add support for the `-c` X-Y flag, which outputs the characters between and including positions X and Y. This option should also work with comma separators.
(c) Third, add support for the `-c` X- and `-c` -Y flags, which print the characters from X to the end of the line, and all characters up to Y.
(d) Fourth, Add support for the `-d`D and `-f`X flags. The former serves as a single character delimiter, and the latter indicates that X is either an interval or a range of fields to print. The fields are delimited by D. Note that these two flags cannot be used without the other. The format of X mirrors that of the input to the $-c$ flag. If D does not exist on a line, then the entire line is printed.

Exercise 1.15. (⋆⋆⋆)
The `sort` program is a command-line tool for sorting input from a data source. For this exercise, you will implement a very basic version of the program that reads data from the terminal.

(a) First, allow the sort command to receive either a file or a list of data. If the `-d`D flag is passed, use D as the delimiter. The default for a file is a newline, and the default for a non-file is the space.
(b) Second, add support for the `-r` flag for reversed sorting.
(c) Third, add support for the `-i` flag for case-insensitive sorting.
(d) Fourth, add support for the `-c` flag for checking to see if a file is sorted. Reports the first occurrence of out-of-order sort.
(e) Fifth, add support the the `-n` flag for sorting the values as if they are numbers. Notice the difference between sorting `9, 10, 8` with and without this flag.
(f) Sixth, add support for the `-u` flag for removing duplicate values.

(g) Seventh, add support for the `-o` flag for outputting to the file specified immediately after.

Any of these flags should be composable with another, with the exception of `-o` whose output file is the next argument, and `-c`, which outputs any disorders to standard out.

Exercise 1.16. (⋆⋆⋆)
The `awk` program is a command-line tool for text parsing and processing. For this exercise, you will implement a very basic version of the program that reads data from the terminal. Be aware that this exercise is more in line with a mini-project.

(a) Add support for the `-F` flag that, when immediately followed by a delimiter, uses that delimiter as a "field separator" when parsing input lines. For example, `-F,` uses a comma as the delimiter.
(b) Add the `-h` flag that ignores the first row in all subsequent commands. This is particularly useful when working with files that have headers, e.g., comma-separated value files.
(c) Next, add the `'print ...'` command. That is, the user should be able to type an open brace, followed by `print`, then some data, then a closing brace, all enclosed by single quotes. The `print` command receives multiple possible values, including 'column labels', i.e., N, where N is a column number. For example, `awk -F, 'print $1' input.csv` should print the first column of each row in the input file.
(d) After getting the previous command to work, add support for inlined prefix operations in the `print` command. That is, suppose we want to print the sum of the second and fourth column of each row. To do this, we might write `awk -h -F, 'print (+ $2 $4)' input.csv`. For simplicity, you may assume that there are only four operations: `'+'`, `'-'`, `'*'`, and `'/'`.
(e) After getting the previous command to work, add multiple-argument support for `print`. That is, if we want to compute the product of the first three columns, output a string saying `" multiplied is "` , followed by the product, we could write `awk -h -F, 'print $1,$2,$3 " multiplied is " (* $1 $2 $3) '`. Note that the delimiter between the column labels must match that passed by `-F`, otherwise there is no separator.
(f) After getting the previous command to work, add support for conditional expressions. That is, suppose we want to print the second column of a row only if it has a value greater than 200. We can achieve this via `"awk -h -F, 'print $2(> $2 200)'`. If you *really* want a challenge, you can add support for inlining other arithmatic expressions into a conditional. For example, if we want to print the third column only if the sum of the first two columns exceeds 1000, we might write `awk -h -F, 'print $3(> (+ $1 $2) 1000)'`.
(g) Finally, add the `-v=N:V` flag that acts as a variable map that can be used in a print command. That is, suppose we want to create a variable called `val` and assign to it `30`. We can do this via `-v=val:30`, then reference it in a `print` via `$`, e.g., `awk -F, 'print $val'`.

Exercise 1.17. (⋆⋆⋆)
A thesaurus is, in effect, a dataset of words/phrases and information about those words/phrases. For example, a thesaurus may contain a word's definition, synonyms, antonyms, part-of-speech, and more. There are hundreds of collections online that researchers use for sentiment analysis, natural language processing, and more. In this exercise you will create a mini-thesaurus parser that allows the user to lookup information about a word/phrase.

(a) First, design the skeleton for the `Thesaurus` class. It should store a `Set<Word>` S as an instance variable.

(b) Design the private and static `Word` class inside the `Thesaurus` class body. A `Word` stores a `String` s and a `Map<String, List<String>>` M as instance variables. The string is the word itself, and the map is an association of information "categories" to a list of content. For example, we can create a `Word` that represents `"happy"`, with an association of `"synonym"` to `List.of("content", "cheery", "jolly")`. Design the respective getters and setters for these two instance variables.

(c) In the `Word` class, design the `boolean updateCategory(String c, String v)` method, which receives a category c and a value v, and updates the list mapped by c to now include v. If `c` did not previously exist for that `Word`, add it to the map and return `false`. On the other hand, if c did previously exist for that `Word`, update its association and return `true`.

(d) In the `Word` class, override the `equals` and `hashCode` methods to compare two `Word` objects for equality and generate the hash code respectively. Two `Word` objects are equal if they represent the same word.

(e) In the `Thesaurus` class, design the `List<String> getInfo(String c)` and `List<String> getInfo(String w, String c, int n)` methods, where the former calls the latter with `Integer.MAX_VALUE` as `n`. The latter, on the other hand, looks up w in S, and

- If `w` $\notin S$, return `null`.
- Otherwise, return n items from the category c of w. If n is `Integer.MAX_VALUE`, return the entire list.

The list returned by both methods must be immutable and not a pointer to the reference in the map. Notice that we overload `getInfo` to perform different actions based on the number (and type) of received parameters.

References

[1] Morling, G. (2023).

Index