CHAPTER 12: WORKING WITH INPUT AND OUTPUT

WORKING WITH INPUT & OUTPUT

Most all of what we do is interact with some form of data – either input that is keyed by a user or systematically received via files as well as displaying output to the screen or to a file for another system.

In Java we use a communication system called a **STREAM** that is implemented in the **java.io** package are enhanced by the **java.nio.file** package. We're going to look at input streams to read information and output streams to store information. In particular we will look at byte streams to handle bytes, integers and other simple data types and character streams to hand text files and the like.

INTRODUCTION TO STREAMS

In Java all data is read from and written to using a streams. A stream carries data from one place to the next – therefore it is a path that is traveled by data in a program. An input stream sends data from a source into a program and conversely an output sends from a program to a data destination.

INTRODUCTION TO STREAMS

We will look at byte and character streams. **BYTE STREAMS** can carry integers with values that range from 0 to 255 and can support formats such as numeric, executable programs, internet communications and bytecode. **CHARACTER STREAMS** are a specialized type of byte stream that handles only textual data. They are different from byte steams since they support Unicode – a standard character encoding method to allow for symbolic language support such as Greek and Chinese – technically these are double byte characters. Any kind of data that involves text should use the character stream including text files, web pages and other common data types.

USING A STREAM: INPUT STREAMS

Input Streams – First, we create an **STREAM OBJECT** associated with the data source – **FileInputStream** = a file on out computer. Then we read information from the stream by using one of the object's methods. FileInputStream includes a **read()** method that returns a byte read form the file. To finish, we call the **close()** method to close the stream and file.

USING A STREAM: OUTPUT STREAMS

Output Streams – We begin by creating a Steam Object associated with the data's destination such as **BufferedWriter** which is an efficient way to create text files. The **write()** method is the simplest way to send information to the output stream's destination. Then we close() the output when we are finished sending all output.

FILTERING A STREAM

The simplest way to use a stream is to create it then call its methods to send or receive the data. Many of the classes we work with achieve more sophisticated results when a **FILTER** is associated with a stream before reading or writing it. A filter is a type of stream that modifies how an existing stream is handled. To use a filter on a stream we should follow the procedure below:

- **Create a stream associated with a data source or data destination**
- **Associate a filter with that stream**
- **Read data from or write data to the filter rather than to the original stream**

The methods that we use to call a filter are the same as the methods that we would call a stream – read() and write(). We can also apply a filter to another filter to achieve further refinement of information before going to its destination.

HANDLING EXCEPTIONS

The java.io contains several exceptions that we may encounter while working with files and streams. The **FileNotFoundException** is the most common and occurs when we try to create a stream or file object using a file that cannot be located. Most likely, we've simply miscoded the path to the file or mistyped the name. The **EOFException**, while technically an exception will help us in determining when we reach the end of the file. However, when we are not at the end of the file, it will alert us to when the **end of file (EOF)** has been reached unexpectedly.

Both of these exceptions are subclasses of **IOException**. One way to deal with all of them is to enclose all input and output statements in a try-catch block that catches the objects as we've done previously. A call to the exception's **toString()** or **getMessage()** methods in the catch side of the block will give us more information about the problem.

BYTE STREAMS

All byte streams are subclasses of either **InputStream** or **OutputStream.** These are abstract classes so we cannot create a stream by creating objects of these classes directly but through one of their subclasses as identified below:

FileInputStream and **FileOutputStream** are byte streams that are stored in files

DataInputStream and **DataOutputStream** are filtered byte streams where we can read data such as integers and floats

It should be noted that InputStream is the superclass of all input streams.

FILE STREAMS

File Streams will most likely be the stream that we’ll work with the most since they gather data from a storage device that we identify by calling the path and the file name. As noted above, we send data to a file using an output file stream and read it in using an input file stream.

FILE INPUT STREAMS

We create an input file stream with the FileInputStream(String) constructor. The String argument should be the file name along with a path reference which enable the file to be in a different folder from the class loading it. The following snippet shows how we would create the input file stream for the file scores.dat. We include path to ensure that the system is picking up the correct file. Note the use of the forward slash /. Since Java uses the backslash \ as an escape code to control printing and such, we must either include the \\ (double backslash) in place all single backslashes to let Java know that the string is a path name or we can reverse all of the backslashes to forward slashes as shown below. Note that this is the syntax for a Linux based system but works equally well on a Windows system.

FileInputStream fis = new FileInputStream(“C:/my documents/scores.dat”);

FILE INPUT STREAMS

Instead of coding the paths as you see above, it is common practice to use a class variable **separator** which is in the File class then to concatenate it to its file name as shown here:

char sep = File.separator;

FileInputStream f2 = new FileInputStream(**sep** + “data” + **sep** + “calendar.txt”);

Output = /data/calendar.txt

Once we have the file input stream created, we can read the bytes from the stream by calling the read(). This method returns an integer containing the next byte in the stream. If it returns a -1, the EOF has been reached.

FILE INPUT STREAMS

To read more than one byte of data from the stream, call read(byte[], int, int) where the argument represent the following:

- A byte array where the data will be stored

- The element inside the array where the data’s first byte should be   stored

- The number of bytes to read

FILE INPUT STREAMS

Unlike the read() we've used previously, this one does not return data from the stream. Rather, it returns either an integer that represents the number of bytes read or a -1 if no bytes were read before the EOF was reached. In the following snippet we can see how this works using a while loop.

```
int newByte = 0;
while (newByte != -1) {
        newByte = diskfile.read();
        System.out.print(newByte + " ");}
```

What we are doing here is referencing the entire file called diskfile one byte at a time while it is not at the EOF. As long as we return a byte, we display the byte followed by space. Since we are using a while loop where it has to do one extra read to pick up the -1, the -1 is displayed as well. To navigate around this condition, we could simply use an if conditional to avoid it.

Exercise 1: ByteReader.java, P. 268

In this exercise we gain practice using an input file stream. The streams **close()** method is used to close the stream (file) after the last byte is read. **ALWAYS** close the stream when you no longer need them to free up resources.

Using Luna code the following module. Be sure to test all paths including the exception processing.

Note: C:/ … / should be changed to the directory where your files for this exercise reside.

<u>Output:</u>

```
71 73 70 56 57 97 24 0 24 0 162 255 0 255 255 255 204 204 204 153 153 153 102 102 102 0 0 0
192 192 192 0 0 0 0 0 0 33 249 4 1 0 0 5 0 44 0 0 0 0 24 0 24 0 64 3 124 88 186 220 222 4 4 17
234 184 163 134 33 67 100 209 164 141 99 247 45 33 101 97 26 87 157 74 42 204 116 61 191 0
68 236 124 239 195 168 159 144 231 72 181 88 22 83 14 213 1 56 159 208 142 103 25 147 168
72 37 28 200 138 37 41 31 224 176 120 252 136 0 4 79 140 154 227 36 64 184 171 75 235 203 20
197 51 73 109 253 186 198 208 171 93 129 83 91 19 81 134 127 5 41 135 81 122 128 104 139 79
141 137 86 144 145 131 13 149 134 100 155 11 9 0 33 254 79 67 111 112 121 114 105 103 104
116 32 50 48 48 48 32 98 121 32 83 117 110 32 77 105 99 114 111 115 121 115 116 101 109 115
44 32 73 110 99 46 32 65 108 108 32 82 105 103 104 116 115 32 82 101 115 101 114 118 101
100 46 13 10 74 76 70 32 71 82 32 86 101 114 32 49 46 48 13 10 0 59 -1

Bytes read: 266
```

BYTEREADER.JAVA

```
import java.io.*;
public class ByteReader {
    public static void main(String[] arguments) {
        try (
            FileInputStream file = new
FileInputStream("C:/Users/rita/Desktop/MyJavaFiles/save.
gif")
            ) {
            boolean eof = false;
            int count = 0;
while (!eof) {
                int input = file.read();
                System.out.print(input + " ");
                if (input == -1)
                    eof = true;
                else
                    count++;
            }
file.close();
            System.out.println("\nBytes read: " + count);
        } catch (IOException e) {
            System.out.println("Error -- " + e.toString());
        }
    }
}
```

LET'S EXAMINE THE CODE

This application reads the byte data from the save.gif file that we used when we were learning interfaces with Swing.

FILE OUTPUT STREAMS

We create the file output stream with the **FileOutputStream(String)** constructor which has basically the same usage as the FileInputStream() and we should specify the path and name of the file in the same manor.

A word of caution: If we specify the path/filename of an existing file, the original is deleted and recreated with our file. We do not get a message telling us that it is existing with a request of what to do. It just does it. We would have to code an if conditional to check if the file exists and if so, give the user an interface to decide what to do.

FILE OUTPUT STREAMS

We can also **APPEND** (add to) the end of a file using the **FileOutputStream(String, boolean)** constructor where the String specifies the file and the Boolean argument is set to true indicating that we want to append the file rather than overwrite.

The file output stream's **write(int) i**s used to write bytes to the stream. After the last byte is written, the steams **close()** is called – <u>ALWAYS close streams</u>, both input and output, to free resources.

Like input, to write more than one byte, we can use the **write(byte[], int, int**) that works much like the read() we saw earlier. The arguments are the byte array containing the byte output, the starting point and then number of bytes.

Exercise 1: ByteWriter.java, P. 270

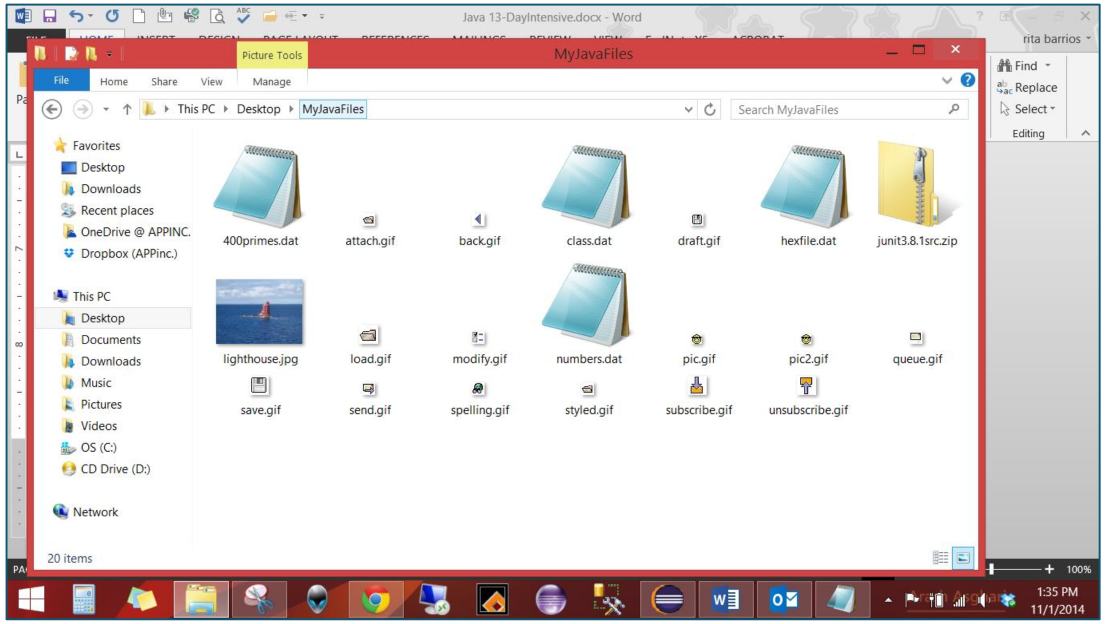
In this application we write an integer array to a file output stream.

Using Luna code the following module and test all paths.

Note: The path “C:/ … /” should be changed to the directory where your Java files reside.

<u>**Output**</u>

Console should be empty. New file should be in the directory you specified.



BYTEWRITER.JAVA

```
import java.io.*;
public class ByteWriter {
    public static void main(String[] arguments) {
        int[] data = { 71, 73, 70, 56, 57, 97, 13,
0, 12, 0, 145, 0,
            0, 255, 255, 255, 255, 255, 0, 0, 0, 0,
0, 0, 0, 44, 0,
            0, 0, 0, 13, 0, 12, 0, 0, 2, 38, 132, 45,
121, 11, 25,
            175, 150, 120, 20, 162, 132, 51, 110,
106, 239, 22, 8,
            160, 56, 137, 96, 72, 77, 33, 130, 86,
37, 219, 182, 230,
            137, 89, 82, 181, 50, 220, 103, 20, 0,
59 };
try (FileOutputStream file = new

FileOutputStream("C:/Users/rita/Desktop/
MyJavaFiles/pic2.gif")) {

            for (int i = 0; i < data.length; i++)
{
                file.write(data[i]);
            }
            file.close();
        } catch (IOException e) {
            System.out.println("Error -- " +
e.toString());
        }
    }
}
```

LET'S EXAMINE THE CODE

Lines 5 – 10: Create an integer array called data and fill it with elements

Lines 11 & 12: Create a file output stream with the file name pic2.gif (don't want to overwrite the original) – we are going to put this in our Java files folder

Lines 14 – 16: Use a for loop to cycle through the data array and write each element to the file stream

Lines 17: Close the file stream

Since we want always want to make sure that our resources are released, we code the FileOutputStream object inside the parentheses of the try statement which ensure that the resources are freed when the block finishes its execution – even in the case of an error. This is a new feature in the Java 7 implementation.

FILTERING A STREAM

FILTERED STREAMS are streams that modify the information sent through an existing stream to refine the information our programs receive. They are created with the subclasses **FilterInputStream** and **FilterOutputStream**. These classes do not handle any filtering operations themselves but rather they have subclasses such as **BufferInputStream** and **DataOutputStream** to handle specific types of filtering. We will look at these.

BYTE FILTERS

Obviously, data comes to our programs more quickly if we can send it in larger chunks as opposed to one byte at a time. Unfortunately, the often come to us faster than our programs can handle them.

A buffer is a storage place in the JVM where we can keep data until we are ready to use it. The data can be input or output. We get a huge performance benefit by doing this since we don’t have to go across the wire as often to get data. This is extremely important with large datasets.

BUFFERED STREAMS

A **BUFFERED INPUT STREAM** fills a buffer with data that has not been processed. When a program needs a data, it will go to the buffer first to get this data before going to the stream source. A buffer byte steam uses the **BufferInputStream** and **BufferedOutputStream** classes using one of the following constructors:

BufferedInputStream(InputStream) – Creates a buffered input stream for the specified InputStream object

BufferedInputStream(InputStream, int) – Creates the specified InputStream with a specified buffer size

BUFFERED STREAMS

We call a buffered input stream using read() with no arguments. We usually will get an integer between 0 and 255 representing the next byte in the stream. When the **EOS (end of stream)** has been achieved we will get a -1 returned. As before, we can use the read(byte[], int, int) to load into a byte array.

BUFFERED STREAMS

The **BUFFERED OUTPUT STREAM**’s write() is used to send a single byte to the stream and the write(byte[], int, int) writes multiple bytes from the specified byte array. As before the arguments are the byte array, the starting point and the number of bytes to be written.

It should be noted that when we direct data to a buffered stream, it is not considered output to its destination until the stream fills or he buffered stream’s **flush()** method is called to clear the buffer.

Exercise 1: BufferDemo.java, P. 272-4

This application, called BufferDemo.java, writes to a buffered stream that is associated with a text file. The first and last integers in the series are specified as two arguments. It creates a buffered input stream from the file and reads the bytes back in.

Using Luna, create and completely test the following module. Run with both command-line arguments and with no arguments.

Note: The path "C:/ .. /" should be replaced with the path where your data files reside.

Writing:

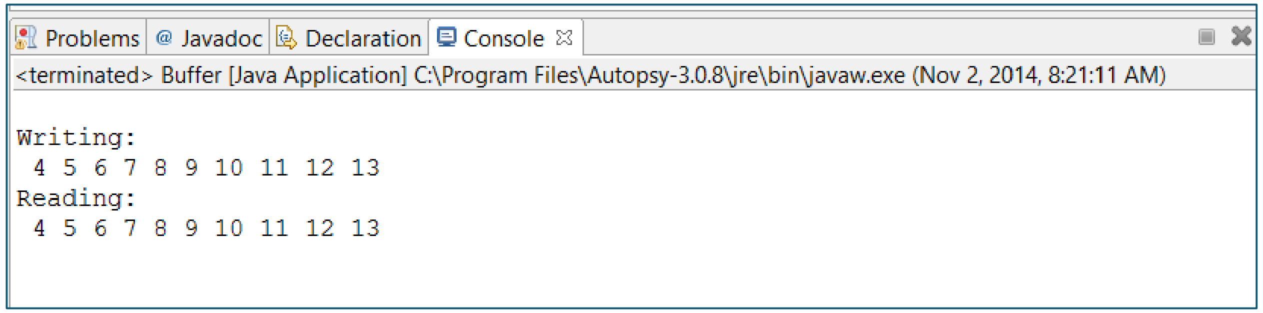
```
 0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24 25 26 27 28 29
30 31 32 33 34 35 36 37 38 39 40 41 42 43 44 45 46 47 48 49 50 51 52 53 54 55
56 57 58 59 60 61 62 63 64 65 66 67 68 69 70 71 72 73 74 75 76 77 78 79 80 81
82 83 84 85 86 87 88 89 90 91 92 93 94 95 96 97 98 99 100 101 102 103 104 105
106 107 108 109 110 111 112 113 114 115 116 117 118 119 120 121 122 123 124
125 126 127 128 129 130 131 132 133 134 135 136 137 138 139 140 141 142 143
144 145 146 147 148 149 150 151 152 153 154 155 156 157 158 159 160 161 162
163 164 165 166 167 168 169 170 171 172 173 174 175 176 177 178 179 180 181
182 183 184 185 186 187 188 189 190 191 192 193 194 195 196 197 198 199 200
201 202 203 204 205 206 207 208 209 210 211 212 213 214 215 216 217 218 219
220 221 222 223 224 225 226 227 228 229 230 231 232 233 234 235 236 237 238
239 240 241 242 243 244 245 246 247 248 249 250 251 252 253 254 255
```

EXERCISE 1: BUFFERDEMO.JAVA, P. 272-4

Output Continued: Reading:

0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24 25 26 27 28 29 30 31 32 33 34 35 36 37 38 39 40 41 42 43 44 45 46 47 48 49 50 51 52 53 54 55 56 57 58 59 60 61 62 63 64 65 66 67 68 69 70 71 72 73 74 75 76 77 78 79 80 81 82 83 84 85 86 87 88 89 90 91 92 93 94 95 96 97 98 99 100 101 102 103 104 105 106 107 108 109 110 111 112 113 114 115 116 117 118 119 120 121 122 123 124 125 126 127 128 129 130 131 132 133 134 135 136 137 138 139 140 141 142 143 144 145 146 147 148 149 150 151 152 153 154 155 156 157 158 159 160 161 162 163 164 165 166 167 168 169 170 171 172 173 174 175 176 177 178 179 180 181 182 183 184 185 186 187 188 189 190 191 192 193 194 195 196 197 198 199 200 201 202 203 204 205 206 207 208 209 210 211 212 213 214 215 216 217 218 219 220 221 222 223 224 225 226 227 228 229 230 231 232 233 234 235 236 237 238 239 240 241 242 243 244 245 246 247 248 249 250 251 252 253 254 255

If we use 4 and 13 as command-line arguments, we will achieve the shown output.



```
<terminated> Buffer [Java Application] C:\Program Files\Autopsy-3.0.8\jre\bin\javaw.exe (Nov 2, 2014, 8:21:11 AM)

Writing:
 4  5  6  7  8  9  10  11  12  13
Reading:
 4  5  6  7  8  9  10  11  12  13
```

BUFFERDEMO.JAVA

```
import java.io.*;
public class BufferDemo {
    public static void main(String[] arguments) {
        int start = 0;
        int finish = 255;
        if (arguments.length > 1) {
            start = Integer.parseInt(arguments[0]);
            finish = Integer.parseInt(arguments[1]);
        } else if (arguments.length > 0) {
            start = Integer.parseInt(arguments[0]);
        }
        ArgStream as = new ArgStream(start, finish);
        System.out.println("\nWriting: ");
        boolean success = as.writeStream();
        System.out.println("\nReading: ");
        boolean readSuccess = as.readStream();
    }
}
```

```
class ArgStream {
    int start = 0;
    int finish = 255;

    ArgStream(int st, int fin) {
        start = st;
        finish = fin;
    }
    boolean writeStream() {
        try (FileOutputStream file = new
FileOutputStream("C:/Users/rita/Desktop/MyJavaFiles/numbers.dat");
            BufferedOutputStream buff = new
                BufferedOutputStream(file)) {
```

BUFFERDEMO.JAVA

```
for (int out = start; out <= finish; out++) {
            buff.write(out);
            System.out.print(" " + out);
         }
         buff.close();
         return true;
      } catch (IOException e) {
         System.out.println("Exception: " + e.getMessage());
         return false;
      }
   }
   boolean readStream() {
      try (FileInputStream file = new
FileInputStream("C:/Users/rita/Desktop/MyJavaFiles/numbers.dat");
         BufferedInputStream buff = new
            BufferedInputStream(file)) {
         int in;
         do {
            in = buff.read();
            if (in != -1)
               System.out.print(" " + in);
         } while (in != -1);
         buff.close();
         return true;
      } catch (IOException e) {
         System.out.println("Exception: " + e.getMessage());
         return false;
      }
   }
}
```

LET'S EXAMINE THE CODE

The application has two classes: BufferDemo and the helper class ArgStream. BufferDemo receives two arguments values, if present and processes them in the ArgStream() constructor.

Line 15: Calls **writeStream()** method of ArgStream to write the series of bytes to the buffered output stream

Line 17: Calls **readStream()** to read back those bytes

As you see, both readStream() and writeStream() is basically the same and take the following order of operation:

The filename (numbers.dat) is used to create a file input or output stream

The file stream is used to create the buffered input or output stream

The buffered stream's writer() is used to send the data or the read() is used to receive the data

The buffer stream is closed

Since they can throw IOExceptions, all operations involving a stream are enclosed in a try-catch block. In addition, although we haven't used the return values of writeStream() or readStream() which is a boolean to indicate the success of the method, they should be in practice to let the caller know the result of the method.

CONSOLE INPUT STREAMS

One of the many functions our programs have to do is accept input keyed into the console. This will help us test our programs as well. We have used the System.out.print() and System.out.println() to send messages to the console but we can also use them to receive input.

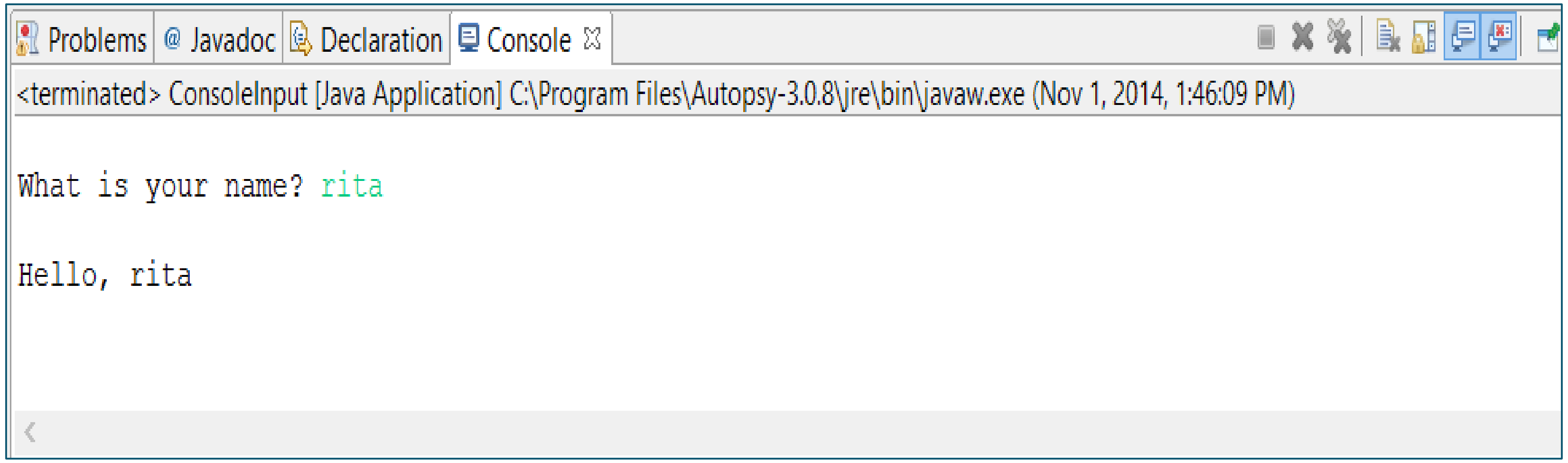
The **System** class, **java.lang** package, has a class variable called *in* that is an InputStream object. This object receives input from the keyboard through the stream and we work with as we would any other input stream. The following statement creates a new buffered input stream associated with **System.in** input stream:

BufferedInputStream command = new BufferedInputStream(System.in);

Exercise 1: ConsoleInput.java, P. 276

The ConsoleInput.java class contains a class method that we can use to receive console input of any of your Java applications. It simply takes it in then displays it back out however, we can use it as a template module.

Using Luna, please code the following and test all paths of execution.



```
<terminated> ConsoleInput [Java Application] C:\Program Files\Autopsy-3.0.8\jre\bin\javaw.exe (Nov 1, 2014, 1:46:09 PM)

What is your name? rita

Hello, rita
```

CONSOLEINPUT.JAVA

```
import java.io.*;
public class ConsoleInput {
  public static String readLine() {
    StringBuilder response = new StringBuilder();
    try (BufferedInputStream buff = new
      BufferedInputStream(System.in)) {
      int in;
      char inChar;
      do {
        in = buff.read();
        inChar = (char) in;
        if ((in != -1) & (in != '\n') & (in != '\r')) {
          response.append(inChar);
        }
} while ((in != -1) & (inChar != '\n') & (in != '\r'));
      buff.close();
      return response.toString();
    } catch (IOException e) {
      System.out.println("Exception: " + e.getMessage());
      return null;
    }
  }
  public static void main(String[] arguments) {
    System.out.print("\nWhat is your name? ");
    String input = ConsoleInput.readLine();
    System.out.println("\nHello, " + input);
  }
}
```

LET’S EXAMINE THE CODE

This application reads the user input through a buffered input stream using the stream’s read(). When the user presses the **enter key (character ‘\r’)** or a **new line (character ‘\n’)**, a -1 is returned letting the class know we are at the end of the stream.

DATA STREAMS

We often have to work with data that is not represented in bytes or characters. For this we use data input and output streams. These filters an existing byte stream so that each of the primitive types can be directly read from or written to the stream. These types include boolean, byte, double, float, int, long and short.

DATA STREAMS

DATA INPUT STREAMS are created with the **DataInputStream(InputSteam)** constructor where the argument should be an existing input stream such as a buffered input stream or a file input stream.

The **DATA OUTPUT STREAM** requires the **DataOutputStream(OutputStream)** constructor indicates the associated output stream. The following methods apply to both input and output data streams where each returns the primitive type indicated by its name. There is also the **readUnsignedByte()** and **readUnsignedShort()** that reads in an unsigned byte or short. Since Java doesn’t return these types, they are returned as an int.

DATA STREAMS

readBoolean(), writeBoolean()

readByte(), writeByte()

readDouble(), writeDouble()

readFloat(), writeFloat()

readInt(), writeInt()

readLong(), writeLong()

readShort(), writeShort()

Not all of the read methods return a value that can be used to indicate the end of the stream. As an alternative, we can wait for the EOFException to be thrown when the EOF is reached and can be handled in the catch side of the try block where we can call the close() to release the resource.

Exercise 1: PrimeWriter.java and PrimeReader.java, P. 277-80

In this exercise, there are two programs that use data streams. The PrimeWriter application writes the first 400 prime numbers to a file called 400primes2.dat. PrimeReader reads the input from the file and displays them. Please use Luna to code the and test the 2 modules, PrimeWriter.java and PrimerReader.java.

Note: The “C:/ … /” should be replaced with the path were your data files reside.

PRIMEWRITER.JAVA

```
import java.io.*;
public class PrimeWriter {
    public static void main(String[] arguments) {
        int[] primes = new int[400];
        int numPrimes = 0;
        // candidate: the number that might be prime
        int candidate = 2;
        while (numPrimes < 400) {
            if (isPrime(candidate)) {
                primes[numPrimes] = candidate;
         numPrimes++;
            }candidate++;}
        try (
            // Write output to disk
            FileOutputStream file = new
                FileOutputStream("C:/ ... /400primes2.dat");
            BufferedOutputStream buff = new
                BufferedOutputStream(file);
DataOutputStream data = new
                DataOutputStream(buff);
            ) {
for (int i = 0; i < 400; i++)
               data.writeInt(primes[i]);
            data.close();
        } catch (IOException e) {
            System.out.println("Error -- " + e.toString());
        }
    }
    public static boolean isPrime(int checkNumber) {
        double root = Math.sqrt(checkNumber);
        for (int i = 2; i <= root; i++) {
            if (checkNumber % i == 0)
                return false;
        }
        return true;
    }
}
```

PRIMERREADER.JAVA

```
import java.io.*;
public class PrimeReader {
    public static void main(String[]
arguments) {
        try (FileInputStream file = new

FileInputStream("C:/Users/rita/Desktop/
MyJavaFiles/400primes2.dat");
            BufferedInputStream buff = new
                BufferedInputStream(file);
            DataInputStream data = new
                DataInputStream(buff)) {
```

```
try {
                while (true) {
                    int in = data.readInt();
                    System.out.print(in + " ");
                }
            } catch (EOFException eof) {
                buff.close();
            }
        } catch (IOException e) {
            System.out.println("Error -- " +
e.toString());
        }
    }
}
```

LET’S EXAMINE THE CODE FOR PRIMEWRITER.JAVA

Most of **PrimeWriter.java** is taken up with logic to find the first 400 prime numbers.

Lines 17 – 33: Write the integer array to an data output stream

We use more than one filter on a stream in this application and is developed in three steps:

A file output stream associated with the file called 400primes.dat is created

A new buffered output stream is associated with the file stream

A new data output stream is associated with the buffered stream

Line 28: Use the writeInt() of the data stream to write the primes to the file

LET'S EXAMINE THE CODE FOR PRIMEWRITER.JAVA

For **PrimeReader.java**, it just reads the file using a data input stream and displays the values.

Lines 5 – 10: Use input classes (instead of output classes)

Lines 12 – 22: Use a try-catch block to handle the EOFException objects – we load the data in the try block

Lines 13 – 15: Use a while(true) loop to create an endless loop – in this case this is not an issue because an EOFException occurs automatically when the end of the stream is encountered when there is no more input.

Line 14: The readInt() reads the integers from the stream

CHARACTER STREAMS

CHARACTER STREAMS are sued to work with any text represented by the ASCII character set or Unicode – an international character set that include ASCII values as well to represent every character that can be used with a computer including the symbolic languages. We can work with plain text, HTML (Hyper Text Markup Language) documents or Java source files. The subclasses of Reader and Writer are used to read and write these streams. We should use character streams when all text files as opposed to working directly with the bytes of these files. It is much easier to use.

READING TEXT FILES

FileReader is the main class to use when reading the character streams from a file. This class inherits from **InputStreamReader** to read a byte stream and converts the bytes into integer values that equate to the associated Unicode characters.

A character input stream is associated with a file using the **FileReader(String)** constructor where the string represent the file that can include a path reference. The following statement creates a new FileReader called look and associates it with a text file called index.txt.

FileReader look = new FileReader("index.txt");

READING TEXT FILES

After we have created the FileReader, we can call the following methods to read the characters from the file:

read() – Returns the next character on the stream as an integer

read(char[], int, int) – Reads characters into the specified character array, the beginning point and the number of characters. This method works the same as byte input stream classes.

The following snippet loads a text file using the FileReader object called text and then displays the characters:

```
FileReader text = new FileReader(“readme.txt”);
int inByte;
do {inByte = text.read();
if (inByte != -1) System.out.print( (char)inByte );
} while (inByte != -1);
System.out.println(“ “);
text..close();
```

READING TEXT FILES

Since a character steam’s read() returns an integer we must cast it to a character before displaying, storing in an array or using it to form a string. As we mentioned earlier, every character has a Unicode value associated with it – this is the integer we are returning, the Unicode value. If we want to ready an entire line instead of character by character, we can use the **BufferedReader** class in conjunction with the FileReader.

In this instance, the BufferedReader class reads a character input stream and buffers it for better efficiency. In order for this be successful, we must have an existing Reader object to create a buffered version.

READING TEXT FILES

The following constructors can be used to create a BufferedReader.

– **BufferedReader(Reader)** – Creates a buffered character stream associated with the specified Reader object such as a FileReader

– **BufferedReader(Reader, int)** – Creates a buffered character stream associated with the specified file with a buffer of the specified size

The buffered character stream can be read using read() and read(char[], int, int) as described for FileReader. We read a line of text using the **readLine()**.

READING TEXT FILES

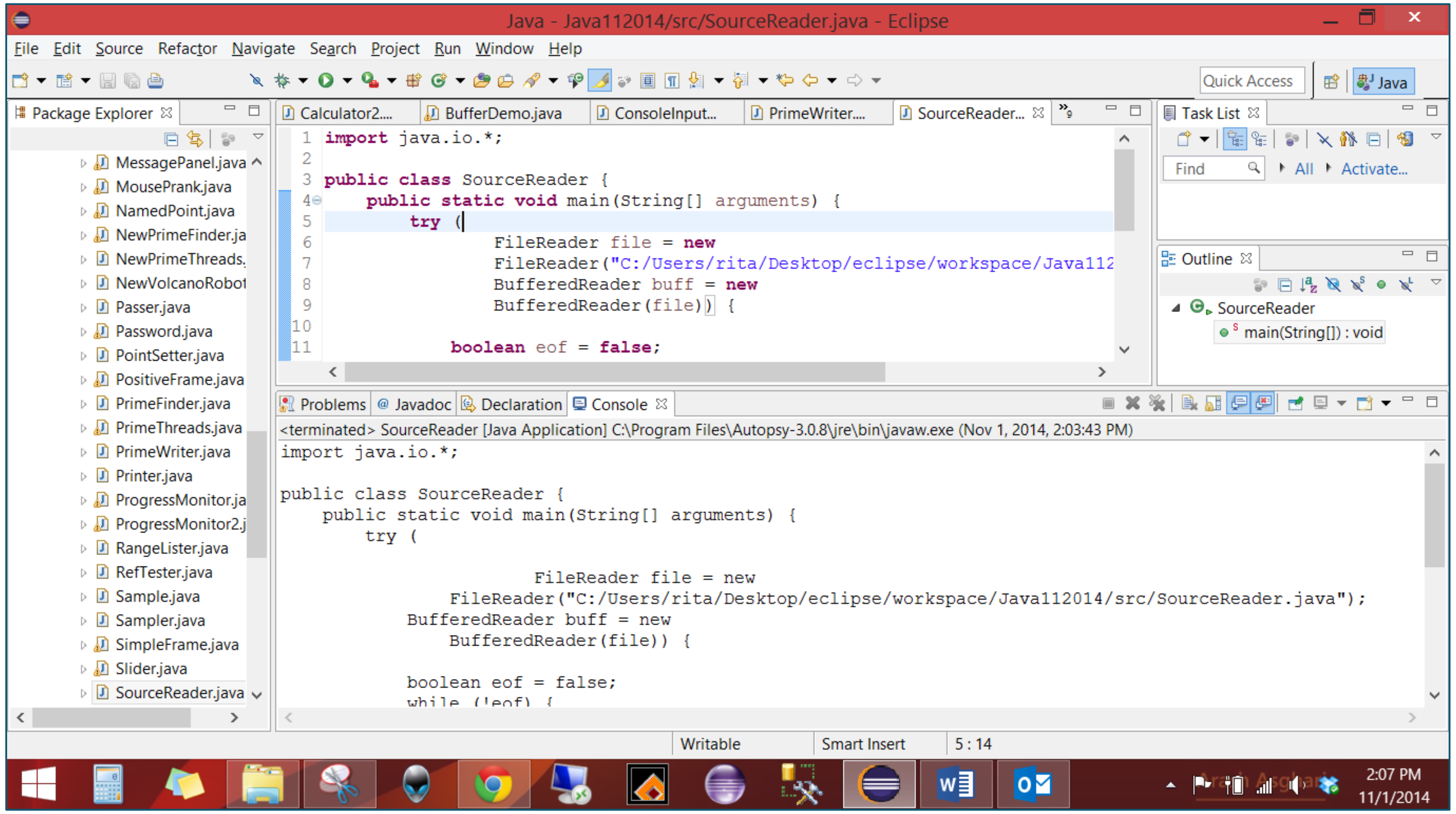
The readLine() returns a String object containing the next line of text on the stream, not including the character or characters that represent the end of the line. If the end of the stream is reached, the value of the string returned is **NULL** meaning nothing. The **end of line (EOL)** is determined by any of the following:

- New line character: **(‘\n’)**
- Carriage return (enter key): **(‘\n’)**
- Carriage return followed by new line: **(“\n\r”)**

Exercise 1: SourceReader.java, P. 282-3

In the following application, SourceReader.java, it reads its own source file through a buffered character stream and then prints it out. Use Luna to create the following code module being sure to test it completely.

Note: The path “C:/ … /” should be changed with the path where your eclipse/workspace/Java112014/src/SourceReader.java source file resides in order for it.



```
import java.io.*;

public class SourceReader {
    public static void main(String[] arguments) {
        try (
                FileReader file = new
                FileReader("C:/Users/rita/Desktop/eclipse/workspace/Java112
                BufferedReader buff = new
                BufferedReader(file)) {

            boolean eof = false;
```

```
<terminated> SourceReader [Java Application] C:\Program Files\Autopsy-3.0.8\jre\bin\javaw.exe (Nov 1, 2014, 2:03:43 PM)
import java.io.*;

public class SourceReader {
    public static void main(String[] arguments) {
        try (

                FileReader file = new
            FileReader("C:/Users/rita/Desktop/eclipse/workspace/Java112014/src/SourceReader.java");
        BufferedReader buff = new
            BufferedReader(file)) {

        boolean eof = false;
        while (!eof) {
```

SOURCEREADER.JAVA

```
public class SourceReader {
    public static void main(String[] arguments) {
        try (
            FileReader file = new FileReader("C:/Users/rita/Desktop/My JavaFiles/hexfile.dat");
            BufferedReader buff = new BufferedReader(file)) {
            boolean eof = false;
            while (!eof) {
                String line = buff.readLine();
                if (line == null) {
                    eof = true;
                } else {
                    System.out.println(line);
                }
            }
            buff.close();
        } catch (IOException e) {
            System.out.println("Error -- " + e.toString());
        }
    }
}
```

LET'S EXAMINE THE CODE

This application is much the same as what we've used earlier. The items of note are as follows:

Line 6 & 7: Create input source: FileReader object is associated with the file SourceReader.java

Line 8 & 9: Associate a buffering filter with the input source: The BufferReader Object buff

Line 11 – 19: Use readLine() inside of the while loop to read the text file one line at a time. The loop ends when the method returns the value null.

WRITING TEXT FILES

FileWriter class is used to write a character stream to a file. It is a subclass of **OutputStreamWriter** and has behavior to convert Unicode to bytes. There are two constructors: **FileWriter(String)** and **FileWriter(String, boolean)** where the String indicates the name of the file that the character stream will be directed into which can include a path. The optional boolean should be equal to true if the file is to be appended to the bottom of an existing file. Like other output streams we must be careful not to overwrite the existing data when we are appending the file. There are three methods of FileWriter to write data to the stream:

– **write(int)** – Write a character

– **write(char[], int, int)** – Write characters from the specified character array with the starting point and number of characters

– **write(String, int, int)** – Writes characters from the specified string with the starting point and the number of characters

WRITING TEXT FILES

The following snippet writes a character stream to a file using FileWriter and the write(int):

FileWriter letters = new FileWriter("alphabet.txt");

for (int i = 65; i , 91; i++)**letters.write((char)i);**

letters.close();

Like BufferedReader we have a **BufferedWriter** class as well. They are created with **BufferedWriter(Writer)** and **BufferedWriter(Writer, int)** constructors where the argument can be any of the character output stream classes such as FileWriter. The optional int argument indicates the size of the buffer to use. BufferWriter has the same three methods as FileWriter: **write(int), write(char[], int, int)** and **write(String, int,int)**. Another method that can be used is the **newLine()** which sends the selected EOL character to terminate the line. EOL markers are determined by the operating system the application is using. Of course the **close()** is used to close the stream.

FILES AND PATHS

To copy or rename files as well as other tasks, we can use a **Path** object from the **java.nio.file** package. Path represents a file or folder reference to improve the File class in the java.io package. The following statement gets a path matching the specified string:

Path source = fileSystems.getDefault().getPath("essay.txt");

This is actually a two-step process:

- First – A class method of the **FileSystems** class is called. The **getDefault()** method returns a **FileSystem** object that represents the computer's way of storing files. Both of these classes are in the **java.nio.file** package.

- Second – **getPath(String)** returns a path object matching the specified file or folder reference.

FILES AND PATHS

A **File object** can be created by Path by calling the **toFile()** as in the following statement whereas A **Path object** can be created by a File object by calling its **toPath()**:

FileWriter fw = new FileWriter(temp.toFile());

We can call several class methods of Files class in the **java.nio.file** package when working with files. The **delete(Path)** class method deletes the file.

Obviously, we must use these methods with care to avoid adverse actions on a file. These methods throw a **SecurityException** if the program does not have the appropriate security to perform the necessary file operations, a **NoSuchFileException** if the paths do not exist and a **IOException** for all other IO errors. If we try to delete a non-empty file, the NoSuchFileException will occur. As such, we need to deal with these exceptions in a try-catch block or a throws clause in the method declaration.

Exercise 1: AllCapsDemo.java, P. 285-6

This application, AllCapsDemo.java, converts all the text in a file to uppercase. The file is pulsed in using a buffered input stream where one character is read at a time. After the character is converted to uppercase, it is sent to a temporary file using a buffered output steam. File objects are used instead of strings to indicate the files involved to make it possible to rename and delete files as needed.

Using Luna code and test the following modules. Use the run procedure below to execute the tests.

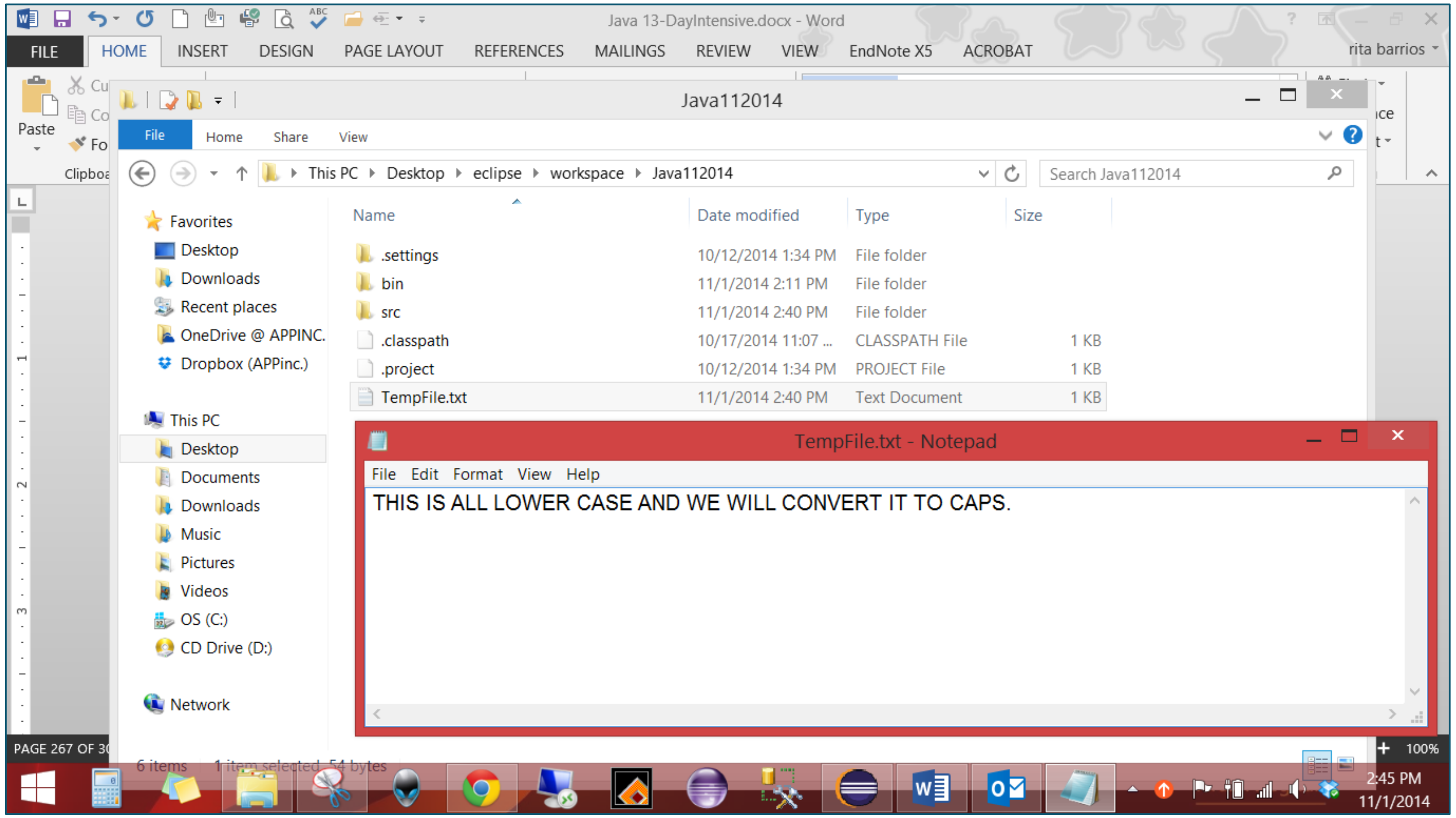
To run this application, please do the following:

- Copy TempFile.txt to the C:\ … \eclipse\workspace\Java112014\ folder
- Edit the file to contain all lower case files
- Enter the TempFile.txt as the command-line argument before you run

EXERCISE 1: ALLCAPSDEMO.JAVA, P. 285-6

Output:

1. Navigate to the project directory and double click on TempFile.txt – data should be in all caps as shown below:



Name	Date modified	Type	Size
.settings	10/12/2014 1:34 PM	File folder	
bin	11/1/2014 2:11 PM	File folder	
src	11/1/2014 2:40 PM	File folder	
.classpath	10/17/2014 11:07 ...	CLASSPATH File	1 KB
.project	10/12/2014 1:34 PM	PROJECT File	1 KB
TempFile.txt	11/1/2014 2:40 PM	Text Document	1 KB

```
THIS IS ALL LOWER CASE AND WE WILL CONVERT IT TO CAPS.
```

ALLCAPSDEMO.JAVA

```
import java.io.*;

import java.nio.file.*;

public class AllCapsDemo {

    public static void main(String[] 
arguments) {

        if (arguments.length < 1) {

            System.out.println("You must 
specify a filename as an argument");

            System.exit(-1);

        }

        AllCaps cap = new 
AllCaps(arguments[0]);

        cap.convert();
```

```
class AllCaps {

    String sourceName;

    AllCaps(String sourceArg) {

        sourceName = sourceArg;

    }

    void convert() {

        try {

            // Create file objects

            Path source = 
FileSystems.getDefault().getPath(sourceName);

            Path temp = 
FileSystems.getDefault().getPath("tmp_" + 
sourceName);
```

ALLCAPSDEMO.JAVA

```
// Create input stream
        FileReader fr = new
FileReader(source.toFile());
        BufferedReader in = new BufferedReader(fr);
        // Create output stream
        FileWriter fw = new FileWriter(temp.toFile());
        BufferedWriter out = new
            BufferedWriter(fw);
        boolean eof = false;
        int inChar;
        do {
            inChar = in.read();
            if (inChar != -1) {
                char outChar =
Character.toUpperCase( (char)inChar );
                out.write(outChar);
} else
                eof = true;
 } while (!eof);
            in.close();
            out.close();

            Files.delete(source);
            Files.move(temp, source);
        } catch (IOException|SecurityException se) {
            System.out.println("Error -- " + se.toString());
        }
    }
}
```

SUMMARY

In this chapter we will learn how to work with input and output files using streams for pulling (input) and pushing data (output). We used character streams to handle text and byte streams for other types of data. We associated Filters with the streams to alter how information was delivered through a stream or to alter the information itself.

The java.io package has other types of streams: Piped streams as useful for communications while byte arrays can connect programs to a computer’s memory. The constructors, read and write methods are basically the same across the streams

INDEPENDENT EXERCISES – 3 HOURS, P. 287

A. Modify HexReader.java, (from Chapter 7 Exceptions and Threads), that reads two-digit hexadecimal sequences from the text file hexfile.data (in your student folder) and displays their decimal equivalent.

B. Write a program that reads a file, junkfile.dat (in your student files), to determine the number of bytes it contains and then to overwrite all those bytes with 0's.

WARNING: THIS IS A DISTRUCTIVE PROGRAM – MAKE A COPY OF THE FILE JUNKFILE.DAT BEFORE YOU RUN THIS.

Solutions can be found in appendix A – Chapter 12 Independent Exercises