```
 1: package fileSeperator;
 2:
 3: import java.io.FileNotFoundException;
 4: import java.io.FileOutputStream;
 5: import java.io.IOException;
 6: import java.util.Random;
 7:
 8: /**
 9:  * Created by bold on 9/15/16.
10:  */
11: public class FileMaker {
12:     private long sizeOfFile;
13:     private Random randomizer = new Random();
14:     private FileOutputStream fileOut;
15:
16:     //sizeOfFile in bytes
17:     public void makeFile(long sizeOfFile, String nameOfFile) {
18:         try {
19:             fileOut = new FileOutputStream(nameOfFile);
20:         } catch (FileNotFoundException e) {
21:             e.printStackTrace();
22:         }
23:         try {
24:             //randomly generates a-z characters and writes them to file.
25:             for (int i = 0; i < sizeOfFile; i++) {
26:                 char c = (char) (randomizer.nextInt(26) + 'a');
27:                 fileOut.write(c);
28:             }
29:             fileOut.close();
30:         } catch (IOException e) {
31:             e.printStackTrace();
32:         }
33:     }
34: }
```

```
    1: package fileSeperator;
    2:
    3: import javax.swing.*;
    4: import java.io.IOException;
    5: import java.util.Scanner;
    6:
    7: /**
    8:  * Created by bold on 9/15/16.
    9:  */
   10: public class FileDriver {
   11:     //these refer to bytes in a mB/kB.
   12:     private static final int mB = 1000000;
   13:     private static final int kB = 1000;
   14:
   15:     public static void main(String[] args) {
   16:         //this just makes a file.
   17:         new FileMaker().makeFile(9999999, "input");
   18:
   19:         JFileChooser fileChooser = new JFileChooser();
   20:         fileChooser.setFileSelectionMode(JFileChooser.FILES_ONLY);
   21:         Scanner console = new Scanner(System.in);
   22:         int returnVal = fileChooser.showOpenDialog(null);
   23:
   24:         if (returnVal == JFileChooser.APPROVE_OPTION) {
   25:             if (fileChooser.getSelectedFile().length() > 100 * mB || fileChooser.getSelectedFile().length() < kB) {
   26:                 System.err.println("File is out of range (1kB - 100mB).");
   27:                 System.exit(10);
   28:             }
   29:
   30:             System.out.println("You are partitioning this file: " +
   31:                     fileChooser.getSelectedFile().getName());
   32:         }
   33:         System.out.print("Please enter base name for partitioned files: ");
   34:         String baseName = console.next();
   35:
   36:         FilePartitioner filePartitioner = new FilePartitioner(fileChooser.getSelectedFile().getAbsolutePath(), baseName, 5,
 5);
   37:         try {
   38:             filePartitioner.partitionFile();
   39:         } catch (IOException ioException) {
   40:             ioException.printStackTrace();
   41:         }
   42:         console.close();
   43:     }
   44: }
```

```
 1: package fileSeperator;
 2:
 3: import java.io.FileInputStream;
 4: import java.io.FileNotFoundException;
 5: import java.io.FileOutputStream;
 6: import java.io.IOException;
 7: import java.util.Arrays;
 8:
 9: public class FilePartitioner {
10:     private String fileName;
11:     private String baseOutputName;
12:     private int numberOfFiles;
13:     private FileInputStream fileStream;
14:     private FileOutputStream[] arrOfOuts = null;
15:     private int maxByteDifference = 0;
16:
17:     /**
18:      * Constructor for the main class FileSeparator.
19:      *
20:      * @param fileName          The fileName of the file that is going to be partitioned.
21:      * @param baseOutputName    The base name of the partitoned file, they will be appended with
22:      *                          numbers 0-numFiles.
23:      * @param maxByteDifference The max number of byte difference allowed/wanted.
24:      * @param numFiles          number of files to be partitioned into.
25:      */
26:     public FilePartitioner(String fileName, String baseOutputName, int maxByteDifference, int numFiles) {
27:         this.fileName = fileName;
28:         this.baseOutputName = baseOutputName;
29:         this.maxByteDifference = maxByteDifference;
30:         this.numberOfFiles = numFiles;
31:     }
32:
33:     /**
34:      * The "driver" method of the class, this partitions the files based on what the Class
35:      * was initialized with.
36:      *
37:      * @throws IOException
38:      */
39:     public void partitionFile() throws IOException {
40:         int partitionCounter = 0;
41:         DataBuffer buffer = null;
42:
43:         initializeStreams();
44:         //assignment statement combined with while loop.
45:         while ((buffer = readFile(maxByteDifference)).hasData()) {
46:             writeFile(partitionCounter % numberOfFiles, buffer.getBytes());
47:             partitionCounter++;
48:         }
```

```
49:         closeStreams();
50:     }
51:
52:     /**
53:      * This writes to all the files.
54:      *
55:      * @param partitionNum Which number file to write to.
56:      * @param byteArr      The byte array that is going to be written to the file.
57:      * @throws IOException
58:      */
59:     private void writeFile(int partitionNum, byte[] byteArr) throws IOException {
60:         arrOfOuts[partitionNum].write(byteArr);
61:     }
62:
63:     /**
64:      * Reads the file and populates a DataBuffer object, using it's constructor with data from the
65:      * FileInputStream
66:      * and based upon the maxByteSizeDifference.
67:      *
68:      * @param sizeOfBuffer This is the max byte difference, or how many bytes are to be read in.
69:      * @return Returns an object that has been populated by calling it's constructor.
70:      * @throws IOException
71:      */
72:     private DataBuffer readFile(int sizeOfBuffer) throws IOException {
73:         return new DataBuffer(fileStream, maxByteDifference);
74:     }
75:
76:     /**
77:      * It is a private helper method, that should not be accessed outside of this class.
78:      * Initializes the InputStream and all the output streams that it keeps in an array
79:      * of FileOutputStreams.
80:      *
81:      * @throws FileNotFoundException
82:      */
83:     private void initializeStreams() throws FileNotFoundException {
84:         fileStream = new FileInputStream(fileName);
85:         arrOfOuts = new FileOutputStream[numberOfFiles];
86:
87:         for (int i = 0; i < numberOfFiles; i++) {
88:             arrOfOuts[i] = new FileOutputStream(baseOutputName + i);
89:         }
90:     }
91:
92:     /**
93:      * It is a private helper method, that should not be accessed outside of this class.
94:      * Closes the input stream and all outputStreams.
95:      *
96:      * @throws IOException
```

```
 97:      */
 98:     private void closeStreams() throws IOException {
 99:         fileStream.close();
100:         for (int i = 0; i < numberOfFiles; i++) {
101:             arrOfOuts[i].close();
102:         }
103:     }
104:
105:     /**
106:      * This class is meant specifically for FileSeparator, that's why it's an inner class.
107:      * It has two methods, getBytes() and hasData(), and they're both populated by the function
108:      * FileInputStream.read(byte[], len, off), which populates a byte[] and returns how many bytes
109:      * were read.
110:      */
111:     private class DataBuffer {
112:         private int returnCode;
113:         private byte[] readBytes;
114:
115:         /**
116:          * @param fileStream     The FileInputStream to read from, or rather the file that is
117:          *                       going to be partitoned.
118:          * @param numBytesToRead This is how many bytes will be read into the byte array.
119:          * @throws IOException
120:          **/
121:         public DataBuffer(FileInputStream fileStream, int numBytesToRead) throws IOException {
122:             this.readBytes = new byte[numBytesToRead];
123:             this.returnCode = fileStream.read(readBytes, 0, numBytesToRead);
124:         }
125:
126:         //returns what bytes were read from the file.
127:         public byte[] getBytes() {
128:             //returns a copy of the array, which may be resized to ignore null bytes when the byte[]
129:             // is not completely filled.
130:             return Arrays.copyOf(readBytes, returnCode);
131:         }
132:
133:         //if returnCode is less than 0 it has reached EOF (-1).
134:         public boolean hasData() {
135:             return this.returnCode > 0;
136:         }
137:     }
138:
139: }
```