File I/O & Exception Handling

Brandon Krakowsky

Penn Engineering

File I/O

Why Java I/O is Difficult

- Java I/O (Input/Output) is very powerful, with an overwhelming number of options

Why Java I/O is Difficult

- Java I/O (Input/Output) is very powerful, with an overwhelming number of options
- Any given kind of I/O is not particularly difficult

Why Java I/O is Difficult

- Java I/O (Input/Output) is very powerful, with an overwhelming number of options
- Any given kind of I/O is not particularly difficult
- The difficulty is in finding your way through the maze of possibilities

Streams

- All modern I/O is stream-based

Streams

- All modern I/O is stream-based
- A stream is a connection to a source of data or to a destination for data (sometimes both)

Streams

- All modern I/O is stream-based
- A stream is a connection to a source of data or to a destination for data (sometimes both)
- An input stream may be associated with the *keyboard*

Streams

- All modern I/O is stream-based
- A stream is a connection to a source of data or to a destination for data (sometimes both)
- An input stream may be associated with the *keyboard*
- An input stream or output stream may be associated with a *file*

Streams

- All modern I/O is stream-based
- A stream is a connection to a source of data or to a destination for data (sometimes both)
- An input stream may be associated with the *keyboard*
- An input stream or output stream may be associated with a *file*
- Different streams have different characteristics:
  - A *file* has a definite length, and therefore an end
  - *Keyboard* input has no specific end

How to do I/O

```
import java.io.*;
```

- *Open* the stream
- *Use* the stream (read, write, or both)
- *Close* the stream

Opening & Reading a File
Method 1: FileReader & BufferedReader

- You can use a FileReader to open (or connect to) a file for input

Opening & Reading a File
Method 1: FileReader & BufferedReader

- You can use a FileReader to open (or connect to) a file for input
- First, create a File object
  ```
  File myFile = new File(pathToFile);
  ```

Opening & Reading a File
Method 1: FileReader & BufferedReader

- You can use a FileReader to open (or connect to) a file for input
- First, create a File object
  ```
  File myFile = new File(pathToFile);
  ```
- Then create a FileReader with the given File object
  ```
  FileReader fileReader = new FileReader(myFile);
  ```
  - myFile specifies a File object. If it doesn't exist, a *FileNotFoundException* is thrown. (More on this later in the lecture.)

Opening & Reading a File
Method 1: FileReader & BufferedReader

- You can use a FileReader to open (or connect to) a file for input
- First, create a File object
  ```
  File myFile = new File(pathToFile);
  ```
- Then create a FileReader with the given File object
  ```
  FileReader fileReader = new FileReader(myFile);
  ```
  - myFile specifies a File object. If it doesn't exist, a *FileNotFoundException* is thrown. (More on this later in the lecture.)
- Finally, create a BufferedReader, which takes a FileReader as an argument
  ```
  BufferedReader bufferedReader = new BufferedReader(fileReader);
  ```
  - A BufferedReader provides *buffering* of data (stored in memory) for fast and efficient reading
  - You can use a BufferedReader to read entire lines of data

Opening & Reading a File
Method 1: FileReader & BufferedReader

- To read a whole line with the BufferedReader, use the *readLine* method
  ```
  String s = bufferedReader.readLine();
  ```
  - *readLine* will return null if there is nothing more to read
  - If an I/O error occurs, an *IOException* is thrown. (More on this later in the lecture.)

Opening & Reading a File
Method 1: FileReader & BufferedReader

- To read a whole line with the BufferedReader, use the *readLine* method
```
String s = bufferedReader.readLine();
```
  - *readLine* will return null if there is nothing more to read
  - If an I/O error occurs, an *IOException* is thrown. (More on this later in the lecture.)
- You should always close the FileReader and BufferedReader
```
fileReader.close();
bufferedReader.close();
```

Opening & Reading a File
Method 2: Scanner

- You can also use a Scanner to open (or connect to) a file for input

Opening & Reading a File
Method 2: Scanner

- You can also use a Scanner to open (or connect to) a file for input
- Again, first create a File object

```
File myFile = new File(pathToFile);
```

Opening & Reading a File
Method 2: Scanner

- You can also use a Scanner to open (or connect to) a file for input
- Again, first create a File object
  ```
  File myFile = new File(pathToFile);
  ```
- Then create a Scanner with the given File object
  ```
  Scanner sc = new Scanner(myFile);
  ```
  - If the file isn't found, a *FileNotFoundException* is thrown

Opening & Reading a File
Method 2: Scanner

- You can also use a Scanner to open (or connect to) a file for input
- Again, first create a File object
  `File myFile = new File(pathToFile);`
- Then create a Scanner with the given File object
  `Scanner sc = new Scanner(myFile);`
  - If the file isn't found, a *FileNotFoundException* is thrown
- You can read and parse one "token" (value) at a time
  sc.next(), sc.nextBoolean(), sc.nextInt(), sc.nextDouble(), etc.

Opening & Reading a File
Method 2: Scanner

- Before trying to get a value, it's good to check if there actually is a value
  sc.hasNext(), sc.hasNextInt(), sc.hasNextDouble(), etc.

Opening & Reading a File
Method 2: Scanner

- Before trying to get a value, it's good to check if there actually is a value
  sc.hasNext(), sc.hasNextInt(), sc.hasNextDouble(), etc.
- Here's the logic:

```
//check for another token to read
if (sc.hasNext()) {
    //get next token
    String nextValueToRead = sc.next();
}
```

Opening & Reading a File
Method 2: Scanner

- Before trying to get a value, it's good to check if there actually is a value
  sc.hasNext(), sc.hasNextInt(), sc.hasNextDouble(), etc.
- Here's the logic:

```
//check for another token to read
if (sc.hasNext()) {
    //get next token
    String nextValueToRead = sc.next();
}
```

- And you should always close the Scanner

```
sc.close();
```

Opening & Reading a File
Method 3: Files

- For reference, java.nio.file.Files is a utility class that contains various useful file methods

Opening & Reading a File
Method 3: Files

- For reference, java.nio.file.Files is a utility class that contains various useful file methods
- The static *readAllLines* method reads all lines from a file, into a List of Strings

```
File myFile = new File(pathToFile);
List<String> allLines = Files.readAllLines(myFile.toPath());
```

Opening & Reading a File
Method 3: Files

- For reference, java.nio.file.Files is a utility class that contains various useful file methods
- The static *readAllLines* method reads all lines from a file, into a List of Strings

```
File myFile = new File(pathToFile);
List<String> allLines = Files.readAllLines(myFile.toPath());
```

- This method puts everything into memory at once
  - It’s good for small files
  - But don't use it if your file is large -- it’s not very efficient

Opening & Reading a File
Method 3: Files

- For reference, java.nio.file.Files is a utility class that contains various useful file methods
- The static *readAllLines* method reads all lines from a file, into a List of Strings

```
File myFile = new File(pathToFile);
List<String> allLines = Files.readAllLines(myFile.toPath());
```

- This method puts everything into memory at once
  - It's good for small files
  - But don't use it if your file is large -- it's not very efficient
- You can treat the List of Strings like an ArrayList

```
//iterate over arraylist and print each line
for (String line : allLines) {
    System.out.println(line);
}
```

Opening & Reading a File
Method 3: Files

- For reference, java.nio.file.Files is a utility class that contains various useful file methods
- The static *lines* method reads all lines from a file, as a Stream of Strings

```
File myFile = new File(pathToFile);
Stream<String> linesStream = Files.lines(myFile.toPath());
```

Opening & Reading a File
Method 3: Files

- For reference, java.nio.file.Files is a utility class that contains various useful file methods
- The static *lines* method reads all lines from a file, as a Stream of Strings

```
File myFile = new File(pathToFile);
Stream<String> linesStream = Files.lines(myFile.toPath());
```

- This method does not read all lines of a file at once
  - It reads lines of a file as a Stream, line by line, as needed (for operations)
  - Use this if your file is large – it's more efficient

Opening & Reading a File
Method 3: Files

- For reference, java.nio.file.Files is a utility class that contains various useful file methods
- The static *lines* method reads all lines from a file, as a Stream of Strings

```
File myFile = new File(pathToFile);
Stream<String> linesStream = Files.lines(myFile.toPath());
```

- This method does not read all lines of a file at once
  - It reads lines of a file as a Stream, line by line, as needed (for operations)
  - Use this if your file is large – it's more efficient
- You can access (and perform an action for) each element of the stream

```
linesStream.forEach(line -> {
    System.out.println(line);
});
```

File Writing
Method 1: FileWriter & PrintWriter

- Create a FileWriter object
  ```
  FileWriter fw = new FileWriter(new File(pathToFile), append);
  ```
  - If an I/O error occurs, an *IOException* is thrown
  - If append is true, you will append to the end of the file

File Writing
Method 1: FileWriter & PrintWriter

- Create a FileWriter object
  ```
  FileWriter fw = new FileWriter(new File(pathToFile), append);
  ```
  - If an I/O error occurs, an *IOException* is thrown
  - If append is true, you will append to the end of the file
- Create a PrintWriter object with the given FileWriter object
  ```
  PrintWriter pw = new PrintWriter(fw);
  ```

File Writing
Method 1: FileWriter & PrintWriter

- Create a FileWriter object
```
FileWriter fw = new FileWriter(new File(pathToFile), append);
```
  - If an I/O error occurs, an *IOException* is thrown
  - If append is true, you will append to the end of the file
- Create a PrintWriter object with the given FileWriter object
```
PrintWriter pw = new PrintWriter(fw);
```
- You can write data to the file using the *println* (or *print*) method
```
pw.println("some text on a line by itself");
pw.println("some more text on a line by itself");
pw.print("even more text");
```

File Writing
Method 1: FileWriter & PrintWriter

- After you're done writing to a file, it's good practice to force any data left in memory to be written to the file using the *flush* method

```
pw.flush();
```

File Writing
Method 1: FileWriter & PrintWriter

- After you're done writing to a file, it's good practice to force any data left in memory to be written to the file using the *flush* method
```
pw.flush();
```
- And of course, you should always close the FileWriter and PrintWriter
```
fw.close();
pw.close();
```

File Writing
Method 2: FileWriter & BufferedWriter

- Create a FileWriter object
  ```
  FileWriter fw = new FileWriter(new File(pathToFile), append);
  ```
  - If an I/O error occurs, an *IOException* is thrown
  - If append is true, you will append to the end of the file

File Writing
Method 2: FileWriter & BufferedWriter

- Create a FileWriter object
  ```
  FileWriter fw = new FileWriter(new File(pathToFile), append);
  ```
  - If an I/O error occurs, an *IOException* is thrown
  - If append is true, you will append to the end of the file
- Then create a BufferedWriter, which takes a FileWriter as an argument
  ```
  BufferedWriter bw = new BufferedWriter(fw);
  ```
  - A BufferedWriter provides *buffering* of characters (stored in memory) for efficient writing to a file

File Writing
Method 2: FileWriter & BufferedWriter

- Create a FileWriter object
  ```
  FileWriter fw = new FileWriter(new File(pathToFile), append);
  ```
  - If an I/O error occurs, an *IOException* is thrown
  - If append is true, you will append to the end of the file
- Then create a BufferedWriter, which takes a FileWriter as an argument
  ```
  BufferedWriter bw = new BufferedWriter(fw);
  ```
  - A BufferedWriter provides *buffering* of characters (stored in memory) for efficient writing to a file
- You can write data to the file using the *write* method
  ```
  bw.write("some text");
  bw.write("\n"); //write newline character to file
  ```

File Writing
Method 2: FileWriter & BufferedWriter

- Force any characters left in memory to be written to the file using the *flush* method
  `bw.flush();`

File Writing
Method 2: FileWriter & BufferedWriter

- Force any characters left in memory to be written to the file using the *flush* method
```
bw.flush();
```
- Of course, don't forget to close the FileWriter and BufferedWriter
```
fw.close();
bw.close();
```

File Writing
Method 3: Files

- Again, for reference, java.nio.file.Files is a utility class that contains various useful file methods

File Writing
Method 3: Files

- Again, for reference, java.nio.file.Files is a utility class that contains various useful file methods
- The static *write* method can be used to write data to a file

```
String text = “text to write”;
File myFile = new File(pathToFile);
Files.write(myFile.getPath(), text.getBytes());
```

File Types

- Text (.txt) files are the easiest kinds of files to work with
  - They can be used by many different programs

File Types

- Text (.txt) files are the easiest kinds of files to work with
  - They can be used by many different programs
- Formatted text files (such as .doc and .docx files) contain binary formatting information
  - Only programs that “know the secret code” can make sense of formatted text files
  - To other programs, such files are just unintelligible binary information

File Types

- Text (.txt) files are the easiest kinds of files to work with
  - They can be used by many different programs
- Formatted text files (such as .doc and .docx files) contain binary formatting information
  - Only programs that “know the secret code” can make sense of formatted text files
  - To other programs, such files are just unintelligible binary information
- Comma-separated value (.csv) files are also very easy to work with

Exceptions

Errors & Exceptions

- An error is a bug in your program
  - Dividing by zero
  - Going outside the bounds of an array
  - Trying to use a *null* reference

Errors & Exceptions

- An error is a bug in your program
  - Dividing by zero
  - Going outside the bounds of an array
  - Trying to use a *null* reference
- An exception is a problem whose cause is outside your program
  - Trying to open a file that doesn't exist
  - Running out of memory

What To Do About Errors & Exceptions

- An error is a bug in your program
  - It should be *fixed*

What To Do About Errors & Exceptions

- An error is a bug in your program
  - It should be *fixed*
- An exception is a problem that your program may encounter
  - The source of the problem is outside your program
  - An exception is not the “normal” case, but your program should be prepared to deal with it

What To Do About Errors & Exceptions

- An error is a bug in your program
  - It should be *fixed*
- An exception is a problem that your program may encounter
  - The source of the problem is outside your program
  - An exception is not the “normal” case, but your program should be prepared to deal with it
- It isn’t always clear whether a problem is an error or an exception

Dealing With Exceptions

- A lot of exceptions arise when you are handling files
  - A needed file may be missing
  - You may not have permission to write to a file
  - A file may be the wrong type

Dealing With Exceptions

- A lot of exceptions arise when you are handling files
  - A needed file may be missing
  - You may not have permission to write to a file
  - A file may be the wrong type
- Exceptions may also arise when you use someone else's classes (or they use yours)
  - You might use a class incorrectly
  - Note: Incorrect use *should* result in an exception
  - For example, using a negative number where a positive int is expected

Three Ways to Deal With Errors

- Ignore all but the most important errors
  - The code is cleaner, but the program will misbehave when it encounters an unusual error

Three Ways to Deal With Errors

- Ignore all but the most important errors
  - The code is cleaner, but the program will misbehave when it encounters an unusual error
- Do something appropriate for every error
  - The code is cluttered, but the program works better
  - You might still forget some error conditions

Three Ways to Deal With Errors

- Ignore all but the most important errors
  - The code is cleaner, but the program will misbehave when it encounters an unusual error
- Do something appropriate for every error
  - The code is cluttered, but the program works better
  - You might still forget some error conditions
- Do the normal processing in one place, handle the errors in another (this is the Java way)
  - The code is at least reasonably uncluttered
  - Java tries to ensure that you handle every error

The try-catch Statement

- Java provides a control structure, the try statement (also called the try-catch statement) to separate “normal” code from error handling:

```
try {
    //do the “normal” code, ignoring possible exceptions
} catch (some exception) {
    //handle the exception
} catch (some other exception) {
    //handle another exception
}
```

The try-catch Statement

- Java provides a control structure, the try statement (also called the try-catch statement) to separate “normal” code from error handling:

```
try {
    //do the “normal” code, ignoring possible exceptions
} catch (some exception) {
    //handle the exception
} catch (some other exception) {
    //handle another exception
}
```

- You can have as many catch blocks as you want
  - But only one per exception type
  - The first one that matches will execute

finally

- After all the catch phrases, you can have an *optional* finally block

```
try {
    //do the “normal” code, ignoring possible exceptions
} catch (some exception) {
    //handle the exception
} catch (some other exception) {
    //handle another exception
} finally { ... }
```

finally

- After all the catch phrases, you can have an *optional* finally block

```
try {
    //do the “normal” code, ignoring possible exceptions
} catch (some exception) {
    //handle the exception
} catch (some other exception) {
    //handle another exception
} finally { ... }
```

- Whatever happens in try-catch, *even if it does a return statement,* the finally code will be executed
  - If no exception occurs, the finally will be executed after the try code
  - If an exception does occur, the finally will be executed after the appropriate catch code

Two Ways to Deal With Exceptions

- You can catch exceptions with a try statement
  - When you catch an exception, you can try to repair the problem, or you can just print out information about what happened
  - For Java's exceptions, this is usually the better choice

Two Ways to Deal With Exceptions

- You can catch exceptions with a try statement
  - When you catch an exception, you can try to repair the problem, or you can just print out information about what happened
  - For Java's exceptions, this is usually the better choice
- For example:

```
void openFile(File file) {
    FileReader fileReader = null;
    BufferedReader bufferedReader = null;

    try {
        fileReader = new FileReader(file);
        bufferedReader = new BufferedReader(fileReader);
        …
    } catch (FileNotFoundException e) {
        System.out.println("Sorry, " + file.getName() + " not found.");
    } catch (IOException e) {
        //prints the error message and info about which line
        e.printStackTrace();
    }
}
```

Two Ways to Deal With Exceptions

- You can also “pass the buck” by stating that the method in which the exception occurs throws the exception
- For example:

```
void openFile(File file) throws IOException {
    FileReader fileReader = new FileReader(file);
    BufferedReader bufferedReader = new BufferedReader(fileReader);
    …
}
```

Two Ways to Deal With Exceptions

- Which of these you do depends on *whose responsibility it is* to do something about the exception
  - If the method “knows” what to do, it should do it -- catch the exception
  - If it should really be up to the user (the method caller) to decide what to do, then “pass the buck” -- throw the exception

Errors & Exceptions Are Objects

- When an *error* occurs, Java *throws* an Error object for you to use
  - You can *catch* this Error object and try to recover
  - You can *ignore* this Error object and the program will crash

Errors & Exceptions Are Objects

- When an *error* occurs, Java *throws* an Error object for you to use
  - You can *catch* this Error object and try to recover
  - You can *ignore* this Error object and the program will crash
- When an *exception* occurs, Java throws an Exception object for you to use
  - You cannot ignore an Exception -- you must catch it
  - You get a *syntax error* if you forget to take care of any possible Exception

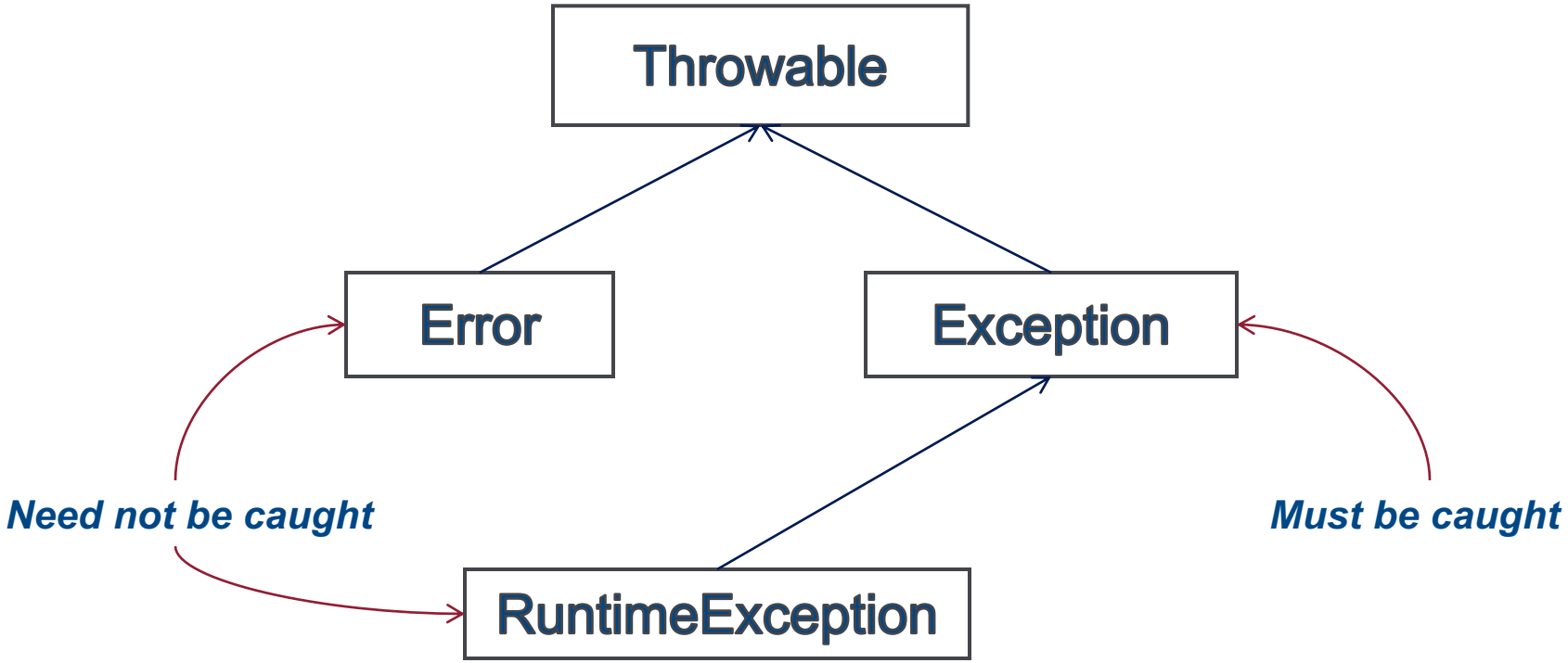
The Exception Hierarchy

Throwable: the superclass of “throwable” objects

- Error: Subclass of Throwable that *does not need to be* caught (instead, the bug that caused it can be fixed)
- Exception: Subclass of Throwable that *must be* caught
  - RuntimeException: Special subclass of Exception that *does not need to be* caught

- Hence, it is the Exceptions that are most important to us (since we have to do something about them)

The Exception Hierarchy



Different Kinds of Built-In Exceptions

Here are some important predefined exceptions in Java

- IOException: a problem doing input/output
  - FileNotFoundException: no such file
  - EOFException: tried to read past the End Of File
  - These *Exceptions* must be caught

Different Kinds of Built-In Exceptions

Here are some important predefined exceptions in Java

- IOException: a problem doing input/output
  - FileNotFoundException: no such file
  - EOFException: tried to read past the End Of File
  - These *Exceptions* must be caught
- NullPointerException: tried to use an object that was actually null
  - This is a *RuntimeException* (doesn't have to be caught)

Different Kinds of Built-In Exceptions

Here are some important predefined exceptions in Java

- IOException: a problem doing input/output
  - FileNotFoundException: no such file
  - EOFException: tried to read past the End Of File
  - These *Exceptions* must be caught
- NullPointerException: tried to use an object that was actually null
  - This is a *RuntimeException* (doesn't have to be caught)
- ArrayIndexOutOfBoundsException: tried to access an element outside of the array bounds
  - This is a *RuntimeException* (doesn't have to be caught)

Different Kinds of Built-In Exceptions

Here are some important predefined exceptions in Java

- IOException: a problem doing input/output
  - FileNotFoundException: no such file
  - EOFException: tried to read past the End Of File
  - These *Exceptions* must be caught
- NullPointerException: tried to use an object that was actually null
  - This is a *RuntimeException* (doesn't have to be caught)
- ArrayIndexOutOfBoundsException: tried to access an element outside of the array bounds
  - This is a *RuntimeException* (doesn't have to be caught)
- NumberFormatException: tried to convert a non-numeric String to a number
  - This is a *RuntimeException* (doesn't have to be caught)

Different Kinds of Built-In Errors

Here are some important predefined errors in Java

- OutOfMemoryError: the program has used all available memory
- StackOverflowError: the amount of call stack memory allocated has been exceeded

There are other predefined Error types

File I/O Examples

MyFileReader Class

MyFileReader.java

```java
package files;

import java.io.BufferedReader;

/**
 * Utility class for reading from files.
 * @author wcauser
 */
public class MyFileReader {

    /**
     * Calculates the total sum of numbers in the given fileName.
     * Returns 0.0 if there are no numerical values found, or there is an error or exception.
     * @param fileName to read
     * @return sum of all numbers in file
     * @throws FileNotFoundException if file can't be found
     */
    public static double readFileGetTotalSum(String fileName) throws FileNotFoundException {

        //create file
        File file = new File(fileName);

        //create default sum
        double sum = 0.0;

        //create scanner with given file
        Scanner scanner = new Scanner(file);
```

MyFileReader Class

```
        //while scanner has another token
        while (scanner.hasNext()) {
            //if token is a double
            if (scanner.hasNextDouble()) {
                //get double value and add to sum
                double numDouble = scanner.nextDouble();
                sum += numDouble;
            //if it's not a double, skip it
            } else {
                scanner.next();
            }
        }

        //close scanner object
        scanner.close();

        return sum;
    }
```

Main Class

MyFileReader.java | Main.java

```java
package files;

import java.io.FileNotFoundException;

/**
 * Controller class.
 * @author wcauser
 */
public class Main {

    public static void main(String[] args) {

        //file name to read
        String fileName = "text.txt";

        //create default sum of numbers in fileName
        double sum = 0.0;

        //try to get sum of numbers in fileName
        try {
            sum = MyFileReader.readFileGetTotalSum(fileName);
        } catch (FileNotFoundException e) {
            //prints the error message and info about which line
            e.printStackTrace();
        } finally {
            //prints the value of sum, regardless of any exceptions
            System.out.println("total sum: " + sum);
        }
```

MyFileReader Class

```
53
54      /**
55       * Calculates the sum of numbers in each line for the given fileName.
56       * @param fileName to read
57       * @return list of sum values
58       */
59      public static ArrayList<Double> readFileGetLineSums(String fileName) {
60
61          //create file
62          File file = new File(fileName);
63
64          //create arraylist to store sum of numbers for each line of file
65          ArrayList<Double> lineSums = new ArrayList<Double>();
66
67          //define file reader
68          FileReader fileReader = null;
69
70          //define buffered reader
71          BufferedReader bufferedReader = null;
72
```

MyFileReader Class

```
        try {
            fileReader = new FileReader(file);
            bufferedReader = new BufferedReader(fileReader);

            String line;

            //while there is another line to read in the bufferedreader
            while ((line = bufferedReader.readLine()) != null) {

                //set default sum for line
                double sum = 0.0;

                //split the line into tokens based on whitespace, \\s+ is regular expression
                //to indicate one or more instances of whitespace
                String[] numStringArray = line.trim().split("\\s+");
```

MyFileReader Class

```
            //iterate over array
            for (int i = 0; i < numStringArray.length; i++) {

                //get each value in array as String
                String numString = numStringArray[i];

                //try parsing to double
                try {

                    //parse to double
                    double numDouble = Double.parseDouble(numString);

                    //add to sum for line
                    sum += numDouble;
                } catch (NumberFormatException e) {
                    //gets and prints exception message
                    System.out.println("Can't parse and add value.  " + e.getMessage());
                }
            }

            //add line sum to arraylist
            lineSums.add(sum);
        }
```

MyFileReader Class

```
        } catch (FileNotFoundException e) {
            //gets and prints filename
            System.out.println("Sorry, " + file.getName() + " not found");
        } catch (IOException e) {
            //prints the error message and info about which line
            e.printStackTrace();
        } finally {

            //regardless, close file objects
            try {
                fileReader.close();
                bufferedReader.close();

            } catch (IOException e) {
                e.printStackTrace();
            }
        }

        return lineSums;

    }
```

Main Class

```
30
31          //get list of sum values
32          ArrayList<Double> lineSums = MyFileReader.readFileGetLineSums(fileName);
33          System.out.println("total line sums: " + lineSums);
34
```

MyFileWriter Class

MyFileReader.java | Main.java | MyFileWriter.java

```java
package files;

import java.io.BufferedWriter;

/**
 * Utility class for writing to files.
 * @author wcauser
 */
public class MyFileWriter {

    /**
     * Writes the sum of numerical values in each line of the given list.
     * @param fileName to write to
     * @param lineSums to read from
     * @param append to overwrite the file
     */
    public static void writeFileLineSums(String fileName, ArrayList<Double> lineSums, boolean append) {

        //create file
        File file = new File(fileName);

        //define file writer
        FileWriter fileWriter = null;

        //define print writer
        PrintWriter printWriter = null;
```

MyFileWriter Class

```
33
34          try {
35
36              fileWriter = new FileWriter(file, append);
37              printWriter = new PrintWriter(fileWriter);
38
39              //iterate over arraylist
40              for (double line : lineSums) {
41                  //print line (sum value) to file
42                  printWriter.println(line);
43              }
44
45              //flush memory
46              printWriter.flush();
```

MyFileWriter Class

```
        } catch (IOException e) {
            e.printStackTrace();
        } finally {

            //regardless, close file objects
            try {
                fileWriter.close();
            } catch (IOException e) {
                // TODO Auto-generated catch block
                e.printStackTrace();
            }

            printWriter.close();
        }
    }
```

Main Class

```
        //write list of sum values to new file
        MyFileWriter.writeFileLineSums("text_line_sums.txt", lineSums, false);

    }
}
```