

# File I/O & Exception Handling

Brandon Krakowsky



# File I/O





# 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
- 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

- 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.)

- 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
- 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.
- 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();`



# 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();`
- 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

- 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();`
- Of course, don't forget to close the FileWriter and BufferedWriter  
`fw.close();`  
`bw.close();`





# 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
- Compilers, in general, work only with text
  - 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
- 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*
- 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
- 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



# 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 { ... }
```

- 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

- 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



# 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 a 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

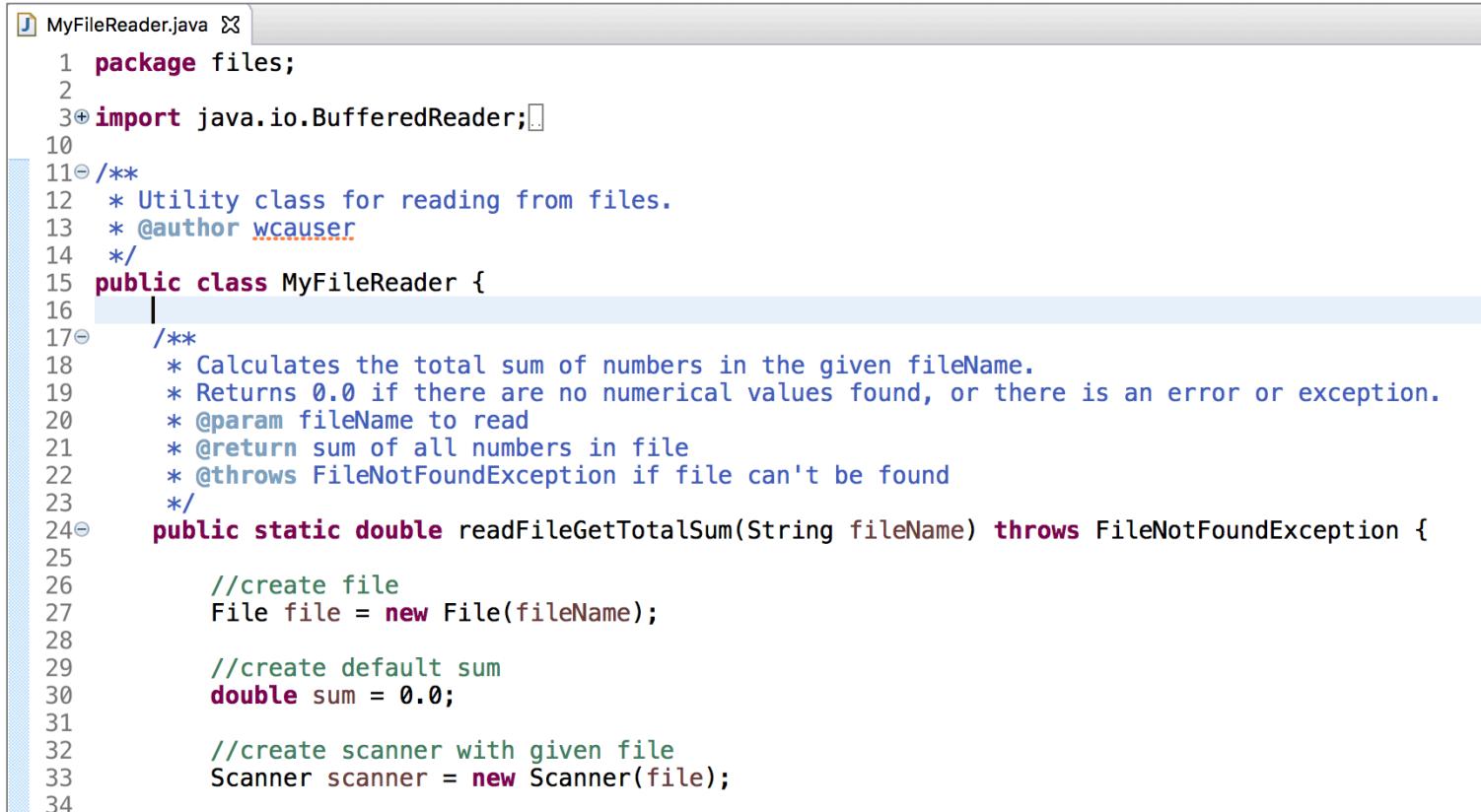
- `OutOfMemoryError`: the program has used all available memory
- `StackOverflowError`: the amount of call stack memory allocated has been exceeded

There are other predefined Error types



# File I/O Examples

# MyFileReader Class



```
1 package files;
2
3+import java.io.BufferedReader;[]
10
11 /**
12 * Utility class for reading from files.
13 * @author wcauser
14 */
15 public class MyFileReader {
16
17     /**
18      * Calculates the total sum of numbers in the given fileName.
19      * Returns 0.0 if there are no numerical values found, or there is an error or exception.
20      * @param fileName to read
21      * @return sum of all numbers in file
22      * @throws FileNotFoundException if file can't be found
23     */
24     public static double readFromFileGetTotalSum(String fileName) throws FileNotFoundException {
25
26         //create file
27         File file = new File(fileName);
28
29         //create default sum
30         double sum = 0.0;
31
32         //create scanner with given file
33         Scanner scanner = new Scanner(file);
34     }
}
```

# MyFileReader Class

```
34
35     //while scanner has another token
36     while (scanner.hasNext()) {
37         //if token is a double
38         if (scanner.hasNextDouble()) {
39             //get double value and add to sum
40             double numDouble = scanner.nextDouble();
41             sum += numDouble;
42             //if it's not a double, skip it
43         } else {
44             scanner.next();
45         }
46     }
47
48     //close scanner object
49     scanner.close();
50
51     return sum;
52 }
```

# 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

```
72
73     try {
74         fileReader = new FileReader(file);
75         bufferedReader = new BufferedReader(fileReader);
76
77         String line;
78
79         //while there is another line to read in the bufferedreader
80         while ((line = bufferedReader.readLine()) != null) {
81
82             //set default sum for line
83             double sum = 0.0;
84
85             //split the line into tokens based on whitespace, \\s+ is regular expression
86             //to indicate one or more instances of whitespace
87             String[] numStringArray = line.trim().split("\\s+");
88     }
```



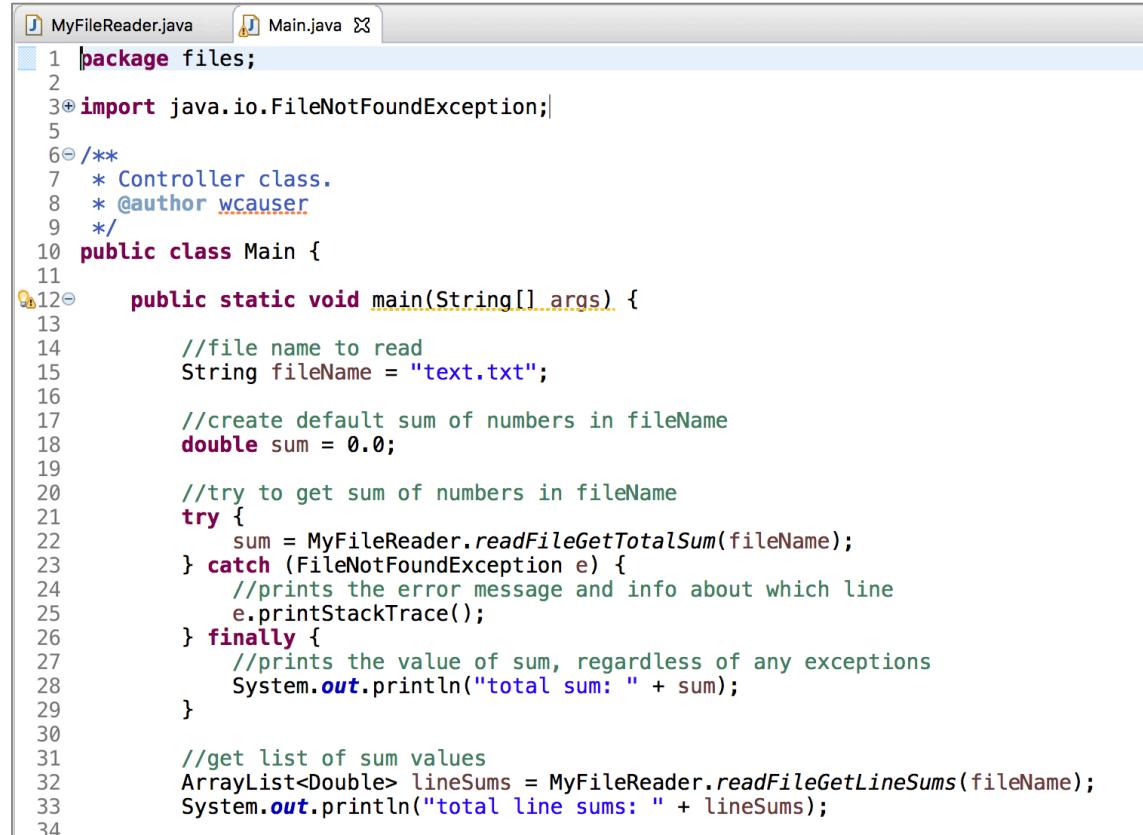
# MyFileReader Class

```
88
89     //iterate over array
90     for (int i = 0; i < numStringArray.length; i++) {
91
92         //get each value in array as String
93         String numString = numStringArray[i];
94
95         //try parsing to double
96         try {
97
98             //parse to double
99             double numDouble = Double.parseDouble(numString);
100
101            //add to sum for line
102            sum += numDouble;
103        } catch (NumberFormatException e) {
104            //gets and prints exception message
105            System.out.println("Can't parse and add value. " + e.getMessage());
106        }
107    }
108
109    //add line sum to arraylist
110    lineSums.add(sum);
111 }
```

# MyFileReader Class

```
112     } catch (FileNotFoundException e) {
113         //gets and prints filename
114         System.out.println("Sorry, " + file.getName() + " not found");
115     } catch (IOException e) {
116         //prints the error message and info about which line
117         e.printStackTrace();
118     } finally {
119
120         //regardless, close file objects
121         try {
122             fileReader.close();
123             bufferedReader.close();
124
125             } catch (IOException e) {
126                 e.printStackTrace();
127             }
128         }
129
130         return lineSums;
131
132     }
133 }
```

# Main Class



```
1 package files;
2
3 import java.io.FileNotFoundException;
4
5 /**
6  * Controller class.
7  * @author wcauser
8  */
9
10 public class Main {
11
12     public static void main(String[] args) {
13
14         //file name to read
15         String fileName = "text.txt";
16
17         //create default sum of numbers in fileName
18         double sum = 0.0;
19
20         //try to get sum of numbers in fileName
21         try {
22             sum = MyFileReader.readFileGetTotalSum(fileName);
23         } catch (FileNotFoundException e) {
24             //prints the error message and info about which line
25             e.printStackTrace();
26         } finally {
27             //prints the value of sum, regardless of any exceptions
28             System.out.println("total sum: " + sum);
29         }
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



```
1 package files;
2
3 import java.io.BufferedReader;
10
11 /**
12  * Utility class for writing to files.
13  * @author wcauser
14 */
15 public class MyFileWriter {
16
17 /**
18  * Writes the sum of numerical values in each line of the given list.
19  * @param fileName to write to
20  * @param lineSums to read from
21  * @param append to overwrite the file
22 */
23 public static void writeFileLineSums(String fileName, ArrayList<Double> lineSums, boolean append) {
24
25     //create file
26     File file = new File(fileName);
27
28     //define file writer
29     FileWriter fileWriter = null;
30
31     //define print writer
32     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();
47
```



# MyFileWriter Class

```
47      } catch (IOException e) {
48          e.printStackTrace();
49      } finally {
50
51          //regardless, close file objects
52          try {
53              fileWriter.close();
54          } catch (IOException e) {
55              // TODO Auto-generated catch block
56              e.printStackTrace();
57          }
58
59          printWriter.close();
60      }
61  }
62 }
```

# Main Class

```
34
35     //write list of sum values to new file
36     MyFileWriter.writeFileLineSums("text_line_sums.txt", lineSums, false);
37
38 }
```