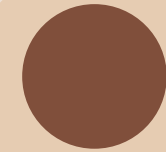
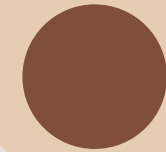


Topic 4.2: Strings & File I/O

Learning Goals (Week 4):



Uses, throw, and catch built-in exceptions.



Understand and use Finally block when handling exceptions.



Order of operations in a try/catch/finally block



Write code that can read and write text files



Write code that uses data from a file to instantiate objects



trim() & split() functions



charAt() & substring()



Additional helpful String manipulation expressions

File I/O in Java

- Java has multiple built-in classes for reading and writing to a file
- Low-level classes are simple and basic, they don't offer many methods and are not very user-friendly
- Mid or high-level classes are often “wrapped” around lower-level classes, they offer more methods and are easier to use
- You can also divide them into 2 groups
 - Those who read / write bytes
 - Those who read / write characters
- Java reads or writes through “streams” of data (either in bytes or characters)

Typical Use

- For reading:
 - Open a stream (building an object)
 - Use the object's methods to get data
 - Close the stream (calling `close()` on the object)
- For writing:
 - Open a stream (building an object)
 - Use the object's methods to write data
 - Close the stream (calling `close()` on the object, which also “flushes” the stream; emptying it in the file)

Low-Level First

- **FileReader** is an example of a low-level, basic reader
 - It's not really meant to be used directly
 - Will usually be “wrapped” inside another class (another type of reader)
 - this is how we will use it

Constructors for FileReader:

- **new FileReader(String)**
 - `FileReader fr = new FileReader("data.txt");`
 - `FileReader fr = new FileReader("C:\\Users\\Me\\Documents\\data.txt");`
- **new FileReader(File)**
 - We'll cover File objects later.
- So now we can Open the File but we can't do very much with it...We need some methods to extract the information from the file...

Mid-Level

- **BufferedReader**

- Handles the **FileReader** for us
- Provides more user-friendly methods
- Still not very fancy (won't read an int or a double) directly
- Reads line by line (Strings) which we can split/parse ourselves

- Constructor:

- `BufferedReader br = new BufferedReader(FileReader)`
- `BufferedReader br = new BufferedReader(new FileReader("data.txt"));`
- This is known as “wrapping” a **FileReader** inside a **BufferedReader**

Mid-Level

The fundamental methods:

- `String readLine()`
 - Reads the next entire line from the file
 - Returns null if there are no more lines
- `int read()`
 - Reads one character from the file
 - But returns its character code as an int, not a char.
 - Cast it to (char) if you want to, but not if...
 - It returns -1 for “end of file” (no more chars)
- `void close()`
 - Release control of the file
 - Should always be done after any file I/O of any type
 - After the stream has been closed, calling methods on the `BufferedReader` will throw an `IOException`

Example

```
try {  
    BufferedReader br = new BufferedReader(new FileReader("text.txt"));  
    String line = br.readLine();  
    while(line != null) {  
        // do whatever with line  
        line = br.readLine();  
    } catch(IOException ioe) {  
        // handle exception  
    }
```


Note on File Reading

- For any type of file I/O you need to:
 - use `import java.io.*;`
 - catch many kinds of `IOExceptions`
- `BufferedReader` will still only give you characters, or a complete line
 - to parse this properly ourselves, we can use `Scanner`!

```
try {
    BufferedReader br; //open it like before (not shown)
    Scanner lineReader = new Scanner(br.readLine());
    int i = lineReader.nextInt();
    double x = lineReader.nextDouble();
    lineReader.close(); //Clean up afterwards. Be nice
    br.close();
    // may need a loop to continue to process each line of a file
    // resetting the Scanner input each time
} catch(IOException ioe) {
    // handle exception
}
```

Advanced Scanner

- When a Scanner reads a line or String, it treats it as a sequence of “tokens” (any consecutive non-blank characters)
- Example:
- If it gets the line or String
 - 34.2 Fred false -3
- it sees it as
 - **"34.2"** then **"Fred"** then **"false"** then **"-3"**

```
boolean hasMore = scannerObject.hasNext( ) //gives a boolean answer
```

- “Is there another token?”

```
String nextToken = scannerObject.next( ) // gives a String
```

- “Give me the next token.”

Advanced Scanner 2

- If you want to read tokens as int, long, float, double, or boolean (but not char):

```
scannerObject.hasNextInt( ) //or Long or Float
```

- Returns true if the next token is OK for that type
- **BUT DOES NOT READ IT**

```
scannerObject.nextInt( ) //or Long or Float or ...
```

- Actually reads it, as that type

```
scannerObject.nextLine( ) // entire line, like BufferedReader
```

- This ignores tokens and just gives you the whole line as one big String
- It's your problem what to do with it

String Reminder

- When you get a String from `readLine()`, sometimes it can be useful to “split” the String

```
try {  
    BufferedReader br = new BufferedReader(new FileReader("data.txt"));  
    String line = br.readLine();  
    while(line != null) {  
        String[] tokens = line.split("\\s+");  
        // do things with the individual values now  
        line = br.readLine();  
    }  
} catch (IOException) ....
```

Pause & Practice

- Checkout the file **4.2-data.txt** in UMLearn
- Open the text file and review the data format
- Write a program to read in this file, line by line.
 - Each line has a command and 2 numbers on it
 - You can assume each line is correct (you do not need to error check)
 - If the first word is **ADD**, add the two numbers on the line together and print the result out
 - if the first word is **MINUS**, subtract the numbers
 - if the first word is **POW**, calculate the first number to the power of the second number
- You may use BufferedReader or Scanner methods

E.g.

ADD 2 2 // this line would result in a print of $2 + 2 = 4$

POW 3 2 // this line would result in a print of $3^2 = 9$

Writing to files - low level

- The low-level object is a **FileWriter**
- Also not really meant to be used directly
- Constructors:
 - `new FileWriter(String);` //Give it the file name (the String)
 - `new FileWriter(File);` //We'll cover "File" objects later
- These two will immediately erase all old contents of the file, and create new content (**these are for overwriting**)

Writing to files - low level

- A **PrintWriter** will control a **FileWriter** for you, and give you the usual convenient methods:
- Methods:
 - `void print(..any type..)`
 - `void println(..any type..)`
 - `void printf(String, ...others...) //we'll discuss later`
 - `void format(String, ...others...) //same as printf`
 - `void close() //should always be done for files`

Again, you must:

- Use `import java.io.*;`
- catch different types of `IOExceptions`

File Objects

- A **File** object does not do any I/O
 - Instead, it contains all of the information needed to identify a particular file
 - Its name, the "path" to find it, what disk it's on, etc.
 - Usually very system-dependent
- One constructor:
 - `new File("file name")`
- You can use one of these to create a **FileReader**, a **FileWriter**, a **PrintWriter** or even a **Scanner** directly
 - `FileReader fr = new FileReader(new File("data.txt"));`
 - `Scanner sc = new Scanner(new File("data.txt"));`
- For the Scanner example, what happens if we didn't do **new File(...)** and instead just did the ... right in the Scanner brackets?

Extra Material - JFileChooser

- A **JFileChooser** object can be used to
 - give the user an ordinary file dialog box
 - obtain a **File** object
 - which can be used to create a **FileReader**, **FileWriter**, etc.
 - which in turn can be used to create a **BufferedReader** or **PrintWriter**
- You can also add features to:
 - Default to the current directory
 - Restrict the choice to a particular type of file
 - Handle the "Cancel" button properly
 - Etc.
- This is beyond the scope of this course
- Read over the JFileChooser documentation if you need to do this

Extra Material - Binary Files (Bytes)

- We saw earlier that there are two types of file I/O:
 - char (text): human readable with your favorite text editor
 - byte (binary): not human readable
- You are going to use I/O of characters most of the time, but know that you can output in bytes too
- If you do
 - `int x=987654321; //A 4-byte number`
 - `outFile.print(x); //printing to a text file, using a PrintWriter (chars)`
 - It will not print 4 bytes (the size of an int). It will print the 9 characters:
 - `'9' '8' '7' '6' '5' '4' '3' '2' and '1'`
- But you can write the actual 4 bytes of raw memory (for x) into a file
- The file will be more compact, but not human-readable
- E.g. for input:
 - `DataStream in = new DataInputStream(new FileInputStream("rawData.xxx"));`
- E.g. for output:
 - `DataOutputStream out = new DataOutputStream(new FileOutputStream("rawData.xxx"));`
- Methods for reading and writing:
 - `readInt()`, `readDouble()`, `readLong()`, `readUTF()`, ...
 - `writeInt(x)`, `writeDouble(x)`, `writeLong(x)`, `writeUTF(x)`, ...

Extra Material - Binary Files (Bytes)

- You can also use the **ObjectOutputStream** and **ObjectInputStream** classes to write / read byte versions of objects to / from a file
- **FYI:** your objects actually need to support the **java.io.Serializable** interface in order to do that
- This is also beyond the scope of this course

Pause & Practice

- Remember that practice we did with 4.2-data.txt?
- Remember that console output you printed?
- Try sending that output to a file instead. Name the file **4.2-output.txt**

Bonus:

- At the start of your program, ask the user if you want to overwrite **or** append to an existing output file (if it exists) and then perform the appropriate action based on the user feedback