

# File I/O + Exceptions

# Agenda

- `System.in` and `System.out` as streams
- Strings as streams
- File input
- File output

# Streams

Writing and reading text is an intricate process.

Fortunately, we have objects that do it for us.

# Streams as Objects

Streams are special object types that can be constructed to provide direct access for reading/writing data from/to a particular source.

Streams have certain methods that allow you to do this seamlessly.

# How Seamless Is It Really?

`System.out` is actually a `PrintStream` object.

As in, `System.out.println()`

- `System.out` is a reference to a pre-defined `PrintStream` that's set up to put text on your console.
- `println()` is the name for a bunch of methods of a `PrintStream` object that writes data to Stream this data belongs to.

`System.out.println()` ➡ *"Write some text to the Stream that shows up on your console."*

**System.out** is for writing, **System.in** is for reading.

**System.in** is actually a **InputStream** object. Specifically, it's a Stream hooked up to the keyboard.

Stream objects are used to construct **Scanner** objects that can parse data from the streams.

# Strings Can Be Used As Streams, Too

Since `Scanners` can be built from Streams, and `Strings` can be Streams, then we can scan over `Strings`!

# Practice: Parsing Data From Strings

```
public static double averageScores(String s) {  
    Scanner scnr = new Scanner(s).useDelimiter(",");  
    return 0.0;  
}
```

Example input: "8,10,12,7,13" → 10.0

Use `hasNextInt()` and `nextInt()`



# Solution: Parsing Data From Strings

```
public static double averageScores(String s) {
    Scanner scnr = new Scanner(s).useDelimiter(",");
    double sum = 0.0;
    int numInts = 0;
    while (scnr.hasNextInt()) {
        sum += scnr.nextInt();
        numInts++;
    }
    return sum / numInts;
}
```

# Strings As Streams Are Useful in Parsing Files

Files with specified delimiters are common ways of expressing data (CSVs, TSVs)



We can read every row from a CSV file as a `String`, and parse that row using a `Scanner` built on that `String`.

# Cooking Up Our Own Input Streams

We have a default input source (`System.in`) and a basic custom input source (any `String` you can get your hands on)

Solution: construct a new `FileInputStream` by specifying the name of the file that you want to read from.

```
FileInputStream fileByteStream = new FileInputStream("grades.csv");
```

 Don't forget to close your `FileInputStream`! 

# Opening Up A File Is Risky Business

```
FileInputStream fileByteStream = new FileInputStream("grades.csv");
```

What if `"grades.csv"` doesn't exist? What if it's being used by another program and can't be opened right now?

What if we try to read from the file and it's not formatted as we expect? What if I left the oven on?

(Temporary) Solution: tag the method you're writing with `throws IOException`

# Full Example:

```
import java.util.Scanner;
import java.io.FileInputStream;
import java.io.IOException;

public static void main (String[] args) throws IOException {
    FileInputStream fileByteStream = new FileInputStream("numFile.txt");
    Scanner scnr = new Scanner(fileByteStream);

    /*
     * Use the Scanner to read from the file...
     */

    fileByteStream.close();
}
```

# Worked Example: Calculate and Print Student Grades

Average each student's grades to get their total score.

Each line of `grades.txt` looks like this  (grades are doubles)

```
student_name hw1_grade hw2_grade hw3_grade
```

```
public static void printGrades(String gradeFile) throws IOException {  
    FileInputStream fileByteStream = new FileInputStream(gradeFile);  
    Scanner scnr = new Scanner(fileByteStream);  
    //...  
}
```

# Worked Example: Calculate and Print Student Grades

```
public static void printGrades(String gradeFile) throws IOException {
    FileInputStream fileByteStream = new FileInputStream(gradeFile);
    Scanner scnr = new Scanner(fileByteStream);

    while (scnr.hasNextLine()) {
        Scanner lineScanner = new Scanner(scnr.nextLine());
        String name = lineScanner.next();
        double sum = 0.0;
        while (lineScanner.hasNextDouble()) {
            sum += lineScanner.nextDouble();
        }
        System.out.printf("%-8s: %-4.1f%n", name, sum / 3);
    }
    scnr.close();
}
```

# Why Print When You Can Write?

The process for writing to a file should be very familiar.

- Set up a Stream
- Set up a Writer using that Stream
- Use the Writer like you know how--printing



# Full Example:

```
import java.util.PrintWriter;
import java.io.FileOutputStream;
import java.io.IOException;

public static void main (String[] args) throws IOException {
    FileOutputStream outputStream = new FileOutputStream("output.txt");
    PrintWriter outWriter = new PrintWriter(outputStream);

    outWriter.println("Harry was here");
    outWriter.println(1);

    outWriter.close();
}
```

# Some Extra Details on Output

- Like before, `FileOutputStream` and `PrintWriter` objects can encounter exceptions
- Creating a new `FileOutputStream` creates a file with that name in your directory--even if you don't write anything to that file!
- `PrintWriter` overwrites the previous file contents by default.

# Worked Example: Calculate and Publish Student Grades

Average each student's grades to get their total score.

Write them all on their own lines in a new file.

Each output line of `finalGrades.txt` should like this 

```
student_name : final_grade
```

```
public static void printGrades(String gradeFile) throws IOException {  
    FileInputStream fileByteStream = new FileInputStream(gradeFile);  
    Scanner scnr = new Scanner(fileByteStream);  
    //...  
}
```

```
public static void printGrades(String gradeFile) throws IOException {
    FileInputStream fileByteStream = new FileInputStream(gradeFile);
    Scanner scnr = new Scanner(fileByteStream);
    FileOutputStream outputStream = new FileOutputStream("finalGrades.txt");
    PrintWriter outWriter = new PrintWriter(outputStream);

    while (scnr.hasNextLine()) {
        Scanner lineScanner = new Scanner(scnr.nextLine());
        String name = lineScanner.next();
        double sum = 0.0;
        while (lineScanner.hasNextDouble()) {
            sum += lineScanner.nextDouble();
        }
        outWriter.printf("%-8s: %-4.1f%n", name, sum / 3);
    }
    scnr.close();
    outWriter.close();
}
```

# What is this **throws** business?

This was news to me, but it turns out that you can actually *make mistakes when you're programming!*

# Exceptions

# Exceptions

We've seen exceptions before:

- `ArrayIndexOutOfBoundsException` when you ask for an index outside of an Array
- `InputMismatchException` when you try to use a `Scanner` to read an `int` from something that doesn't represent an `int`
- `NullPointerException` when you call a method from an object that hasn't been properly initialized

# Unchecked Exceptions

"You messed up."

These are the exceptions we've seen before. It's possible to write code that compiles but doesn't behave correctly. These bugs are your fault as a programmer.



# Not everything is your fault, though

**Checked Exceptions** are those resulting from errors outside of your control.

- A file you're trying to read from just isn't there
- The operating system refuses to help you close a file
- The file ended while you were expecting there to be more to read

These have to be handled.

# Handling Exceptions

- Easy way: just don't!
  - Tag the method that could throw a checked exception with `throws <Exception-Name>`
  - Let the method that called this method worry about. *Don't say we didn't warn ya!*
- Safer way: `try/catch`
  - Specify behavior in the case that specific exceptions are thrown.

(It is actually often better to take the easier route)

# Try / Catch

Syntax for trying to run some code and providing a response in the case that an Exception is thrown.

```
try {  
    // dangerous statement(s)  
} catch (IOException e) {  
    // error handling statement(s)  
} catch (NumberFormatException e) {  
    // error handling statement(s)  
}
```

If an exception is thrown, run the code in the first `catch` block with a matching exception.

# Try / Catch / Finally

Code in `finally` will be run once the `try` block is entered, no matter what.

```
PrintWriter out = new PrintWriter("file.txt");
try {
    writingMethod(out);
} catch (NumberFormatException e) {
    System.out.println("Can't do that!");
} finally {
    out.close()
    // Make sure that out gets closed no matter what!!!
}
```

# Throwing Exceptions

Sometimes you want to throw an exception yourself!

```
throw new Exception("message describing error");
```

e.g.

```
throw new IllegalStateException("You cannot accelerate if the car is in park");  
throw new IllegalArgumentException("Can't deposit a negative value into your bank account");  
throw new ArithmeticException("This field does not have a zero element");
```

# Throwing Practice

Fix this program so that it throws an exception with a meaningful message when you try to double the value at an invalid position in the array.

```
public static void doubleElement(int i, int[] arr) {  
    arr[i] *= 2;  
}
```

# Catching Practice