File Writing (Out.java)

Learning Objectives

- Describe the role of streams in reading/writing data in Java
- Be able to use the Out.java class to write text to a file
- Print nicely formatted text using the printf method

Streams

Writing and reading text is an intricate process for a computer to do

Fortunately, we have objects that do it for us called **streams**

- Special object types that can be constructed to provide direct access for reading/writing data from/to a source
- Streams have certain methods that make the reading/writing seamless (through abstraction)

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 of a method of a `PrintStream` object that writes data through the Stream

Printing, but Permanent

`System.out.print()` & `System.out.println()` are useful for debugging output

- But: The console output isn't saved anywhere by default

`Out.java` is packaged into the `cis110.jar` file already present in all your projects. Using it, you can...

- set up a Stream to a given file
- use familiar `print()` and `println()` methods to write text to a file

An Example

```java
public static void main(String[] args) {
        Out myVarName = new Out("outputFilename.txt");
        myVarName.println("Harry was here.");
        myVarName.println(1);
        myVarName.close();
}
```

Construct a new `Out` object just like you would an `In` object!

```java
Out myVarName = new Out("outputFilename.txt");
```

An Example

```
public static void main(String[] args) {
        Out myVarName = new Out("outputFilename.txt");
        myVarName.println("Harry was here.");
        myVarName.println(1);
        myVarName.close();
}
```

The filename that you use can refer to...

- a file that already exists, in which case its contents will be **overwritten**, or
- a file that doesn't exist yet, in which case it will be created

An Example

```java
public static void main(String[] args) {
        Out myVarName = new Out("outputFilename.txt");
        myVarName.println("Harry was here.");
        myVarName.println(1);
        myVarName.close();
}
```

- `println()` writes the provided data to the current end of the file, then adds a new line afterwards.
- `print()` does the same without adding a new line.

An Example

```
public static void main(String[] args) {
        Out myVarName = new Out("outputFilename.txt");
        myVarName.println("Harry was here.");
        myVarName.println(1);
        myVarName.close();
}
```

`close()` is technically something that you should have been doing to your `In` objects too, but it's very important when dealing with `Out`.

- Tells the operating system you're done writing to the file
- Guarantees that all requested "prints" will be fully completed in the file
- If you don't close the file, leads to a "resource leak"

Out.java Constructor

```java
Out myVarName = new Out("outputFilename.txt");
```

Takes in a String representing the name of a file

- That file may or may not already exist
  - If the file does exist already, then writing to it will replace its current contents (beware!)
  - If the file doesn't exist, it will be created.

Out.java Methods

There are a bunch! But mostly they are just different overloads of:

- `print(myData)` $\rightarrow$
  - For primitive types: writes exactly `myData`
  - For object types: writes exactly `myData.toString()`
- `println(myData)` $\rightarrow$ writes `myData` or `myData.toString()`, followed by a newline character (`'\n'`).

What does it mean to close a file?

`close()` tells Java that you're done writing to this file

- Java will stop letting you write to the file and do some invisible cleanup

If you don't close the `Out`…

- Your program might slow down
- Some data that you asked to write might not appear in the file
- Other programs might be locked out of the files you're writing to

CLOSE YOUR OUT OBJECTS!

Practice: `GradeCalc.java`

Given a file containing a list of student names and their grades, calculate the average grade for each student and write the average grades to a file.

Plan with a partner:

- How do we open up an input file for reading?
- How do we parse out the contents of the grade file?
- How do we calculate the average grade for each student?
- How (and at what point) do we write each student's average grade?

Solution: `GradeCalc.java`

```java
public class GradeCalc {
    public static void main(String[] args) {
        if (args.length != 1) {
            System.out.println("Usage: java GradeCalc <gradeFile>");
            return;
        }
        In rawGrades = new In(args[0]);
        Out averages = new Out("averageGrades.txt");
        while (!rawGrades.isEmpty()) {
            String name = rawGrades.readString();
            double averageGrade = 0.0;
            for (int i = 0; i < 4; i++) {
                averageGrade += rawGrades.readDouble();
            }
            averageGrade /= 4.0;
            averages.printf("%s %.1f%n", name, averageGrade);
        }
    }
}
```

What is `printf`?

In the `Out` class (and in `System.out`), there's a method **printf** that takes two inputs:

- A format String consisting of literal text and *format specifiers*
    - Format specifiers are like slots where the missing data should go, allowing you to specify how that value is displayed in the String
- The remaining arguments, *one per format specifier*, are the values that will be placed into the locations specified by the format specifiers.

Format Specifiers

The syntax for a format specifier is:

- `%[flags][width][.precision]conversion-character`

Format Specifiers

The syntax for a format specifier is:

- `%[flags][width][.precision]conversion-character`
- (yes, I know this looks incomprehensible…)

Format Specifiers

The syntax for a format specifier is:

- `%[flags][width][.precision]conversion-character`

Common Conversion Characters	Purpose
d	decimal integers (int)
f	floating point values (double)
s	String
n	A newline character (\n, a line break)

Format Specifiers

The syntax for a format specifier is:

- `%[flags][width][.precision]conversion-character`
  Flags, width, and precision modifiers can optionally be used to dictate how the arguments get displayed.

Breaking down `%4.1f`, for example…

- The width of the result is at least 4, meaning that the number will be padded with space to be at least 4 characters long
- The precision of the value is 1, meaning that the number will be rounded to one decimal place.

Formatted Printing

Previously, we'd print out data interspersed with text like this:

```java
int age = 19;
double weight = 198.3839;
String name = "John Doe";
System.out.println("Patient " + name + " is " + age + " y/o and weighs " + weight + " lbs.");
```

This is tedious to type and it's easy to make mistakes when typing:

- Forgetting spaces between words
- Missing + signs or start/end quotes

Formatted Printing

We can use `System.out.printf()` to do it more succinctly.*

```
int age = 19;
double weight = 198.3839;
String name = "John Doe";
System.out.printf("Patient %s is %d y/o and weighs %4.1f lbs.%n", name, age, weight);
```

* *OK, maybe not that much more succinctly.*