



Department of Computer Science

File Processing

Dr. Chia-Ling Tsai

Outline

- Basics
- Scanners for files
- Print streams
- Input differences
- Output differences
- The file-not-found exception
- Scanners and tokens
- Specifier

Basics

- We already store our programs in files
- Generally, files can be used both to *provide input* to a program and to *save output* from a program
- Almost nothing new has to be done to change the locations of input and output to files —`Scanner` works on files, as do `print`, `println` and `printf` for files attached to a `PrintStream`

Scanners For Files

- When we write


```
Scanner keyboard = new Scanner(System.in);
```

 we are making a scanner and attaching it to the `System.in` object
- The `File` class allows us to make a *file object* with a given name


```
File inFile = new File("Data.txt");
```
- For a file, we make a scanner and attach it to a file object


```
Scanner input = new Scanner(inFile);
```
- Once we make a scanner for a file, it behaves *exactly the same* as a scanner for `System.in`

Print Streams

- The object `System.out` (unlike `System.in`) can be used directly
- `System.out` is actually an object of type `PrintStream`
- We can make other print streams, based on files

```
File outFile = new File("Results.txt");
PrintStream output =
    new PrintStream(outFile);
```
- Once we make a print stream for a file, it behaves *exactly the same* as `System.out`

Input Differences

- Even though scanners for files act the same way as scanners for the keyboard, there are some differences in the way a *programmer* must deal with them
 - You must be concerned about whether the file in question is actually there
 - The keyboard usually is!
 - You won't be giving prompts to the user
 - You may have to deal with a file whose format you don't know
 - Conversely, you may have a file whose format is very specific

Output Differences

- Even though print streams for files act the same way as they do for the screen, there are some differences in the way a *programmer* must deal with them
 - You must be concerned about whether the file can actually be created
 - If there is already a file with the name you choose, it will be *replaced* by the new one
- You should always *close* an output file
`output.close();`
when you finish writing to it

The File-Not-Found Exception

- If something goes wrong while setting up a file, either for input or for output, your program will “throw” a `FileNotFoundException`
- This kind of exception is similar to those you’ve seen when programs don’t run correctly
- However, for this kind of exception, your program must “admit” it may throw it, by adding a clause to the heading of `main`

```
public static void main(String[] args)
    throws FileNotFoundException
```
- (Other ways to deal with this exist, but we won’t work with them in this course)

Sample Program

```

import java.util.Scanner;
import java.io.File;
import java.io.PrintStream;
import java.io.FileNotFoundException;

public static void main(String[] args, throws FileNotFoundException
{
    File averageFile = new File("Averages.txt");
    File gradeFile = new File("Grades.txt");

    Scanner input = new Scanner(averageFile);
    PrintStream output = new PrintStream(gradeFile);

    while (input.hasNextDouble())
    {
        double nextAverage = input.nextDouble();
        String correspondingGrade = gradeFor(nextAverage);
        output.println(correspondingGrade);
    }
    output.close();
}

```

Scanners And Tokens

- A *token* is a “chunk” of input – a word, a number, etc
- The next token is obtained by a scanner using the method `next`
 - This can be used to get a “word” from the input
- The `hasNext` method says whether there is anything at all to be read
 - This is most useful for files
- With these methods of scanners, and `printf` for output, files fit for human reading can be read and written

Sample Program

```

import java.util.Scanner;

import java.io.File;
import java.io.PrintStream;
import java.io.FileNotFoundException;

File averageFile = new File("NamesAndMarks.txt");
File reportFile = new File("GradeReport.txt");

Scanner input = new Scanner(averageFile);
PrintStream output = new PrintStream(reportFile);

output.printf("    NAME          TEST 1 MIDTERM TEST 2 FINAL  AVERAGE  GRADE\n");
System.out.println("Processing file NamesAndMarks.txt");

```

Imports

Open input and output files

Message to screen

Report heading

Sample Program (*continued*)

```

while(input.hasNext())
{
    // Read a line of input
    String name = input.next();
    int firstTest = input.nextInt();
    int midtermExam = input.nextInt();
    int secondTest = input.nextInt();
    int finalExam = input.nextInt();

    // Calculate weighted average and corresponding grade
    double weightedAverage =
        findWeightedAverage(firstTest, midtermExam, secondTest, finalExam);
    String grade = gradeFor(weightedAverage);

    // Print a line of output
    output.printf("%-15s%5d%7d%7d%9.1f%7s\n",
        name, firstTest, midtermExam, secondTest, finalExam,
        weightedAverage, grade);
}

output.close();
System.out.println("Results written to file GradeReport.txt");

```

As before, but no prompts

Same calculations

Sample Program – Files

Input file
Created in TextPad
(or by another Java
program)

Smith	66	75	87	76
Jones	91	99	87	95
Mercier	71	77	75	75
Malatesta	80	81	77	80
Gregory	100	100	99	100
DiGregorio	52	58	61	71
Manfredo	80	82	100	86
O'Connell	91	76	67	71

Output file
Created by our
Java program

NAME	TEST 1	MIDTERM	TEST 2	FINAL	AVERAGE	GRADE
Smith	66	75	87	76	75.9	C
Jones	91	99	87	95	94.4	A
Mercier	71	77	75	75	75.0	C
Malatesta	80	81	77	80	79.9	C
Gregory	100	100	99	100	99.9	A
DiGregorio	52	58	61	71	62.8	D
Manfredo	80	82	100	86	86.0	B
O'Connell	91	76	67	71	74.9	C

Specifier

- A *specifier* shows the way we want a number (or other item) to appear
- Specifiers used for formatting can be very involved – see the [documentation](http://www.java2s.com/Tutorial/Java/0120__Development/0200__printf-Method.htm).
 - http://www.java2s.com/Tutorial/Java/0120__Development/0200__printf-Method.htm
- A specifier starts with %
- After the specification string comes a list of what is to be printed
 - Items on this list are *matched in order with the specifiers*

Specifier

- First come *flags*
 - A blank means leave room for a sign in case there is one
 - A comma means group digits in threes, with commas between
- Next come the *conversions and width specifications*
 - A `d` means print as an integer ("decimal"). If you use a number in front, (e.g., `5d`) at least that many print positions are used
 - A decimal point followed by a number then `f` (e.g., `.2f`) means print as a floating point number
 - The number to the right of the point represents the number of *decimal* places to show – the printed item will be *rounded* to this number of places
 - A number to the left of the decimal point (e.g., `7` in `7.2f`) represents the minimum *total* number of positions to use
 - `\n` means continue on next line
 - A number followed by `s` means print a string using at least the given number of spaces

Other Arguments

- After the specifier string, you list one argument for each specifier, representing the *value* that will be printed using that specifier
- Each value argument may be either a constant or a variable
- The type of the argument must match the specifier
- Example:

```
int a = 12; double b = 12.987;
System.out.printf("a = %d\nb = %.2f\n", a, b);
prints
a = 12
b = 12.99
```

```

// Demonstration of the printf method
public class FormattedOutput
{
    public static void main(String[] args)
    {
        // Sample values
        int n = 12345;
        double x1 = 12345;
        double x2 = 0.5678;
        double x3 = 12345.6789;
        // Example 1
        // Using print
        System.out.println("n = " + n + " and x1 = " + x1);
        System.out.println("x2 = " + x2);
        System.out.println("x3 = " + x3);
        System.out.println();
        // Using printf
        System.out.printf(
            "n = %d and x1 = %, .0f\nx2 = %, .2f\nx3 = %, .3f\n\n",
            n, x1, x2, x3);
        // Example 2
        // Using print
        double gpa1 = 2.678;
        double gpa2 = 3;
        System.out.println("GPA1 = " + gpa1 + " and GPA2 = " + gpa2);
        System.out.println();
        // Using printf
        System.out.printf("GPA1 = %.1f and GPA2 = %.1f\n\n", gpa1, gpa2);
        // Example 3
        // Using print
        double salary1 = 123456.789;
        double salary2 = 75000;
        System.out.println("Salary 1 = $" + salary1 + " and Salary 2 = $"
            + salary2);
        System.out.println();
        // Using printf
        System.out.printf("Salary 1 = $%,.2f and Salary 2 = $%,.2f\n",
            salary1, salary2);
    }
}

```

n = 12345 and x1 = 12345.0
x2 = 0.5678
x3 = 12345.6789

n = 12,345 and x1 = 12,345
x2 = 0.57
x3 = 12,345.679

GPA1 = 2.678 and GPA2 = 3.0

GPA1 = 2.7 and GPA2 = 3.0

Salary 1 = \$123456.789 and Salary 2 = \$75000.0

Salary 1 = \$123,456.79 and Salary 2 = \$75,000.00

Amortization Table again

- What would be the specifier for *printf*?

Month	P (Principal)	I (Interest)	P + I	Total P paid	Total I paid	P balance
0	n/a	n/a	n/a	n/a	n/a	100,000.00
1	67.10	666.67	733.76	67.10	666.67	99,932.90
2	67.55	666.22	733.76	134.64	1,332.89	99,865.36
...
359	724.08	9.69	733.76	99,271.09	164,150.39	728.91
360	728.91	4.86	733.76	100,000.00	164,155.25	0.00

Summary

- Reading input from files
- Writing output to files
- Formatted output