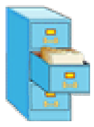
Files

Readings: 6.1 – 6.2



Reading data from files

- Creating a `Scanner` for a file, general syntax:
  `Scanner <name> = new Scanner(new File("<file name>"));`
- Example:
  `Scanner input = new Scanner(new File("numbers.txt"));`
- Instead of getting data from the keyboard via `System.in`, this `Scanner` object gets data from the file `numbers.txt` in the current folder (*directory*).

`File` class

- The `File` class is in the `java.io` package. To use it, include the import declaration:

  ```
  import java.io.*;
  ```
- `io` (or I/O) stands for input/output.

Compiler error with files

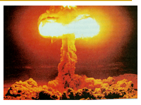
- The following program does not compile:

```
1 import java.io.*;     // for File
2 import java.util.*;   // for Scanner
3
4 public class ReadFile {
5     public static void main(String[] args) {
6         Scanner input = new Scanner(new File("data.txt"));
7         // do something
8     }
9 }
```

- The compiler reports:

```
ReadFile.java:6: unreported exception
java.io.FileNotFoundException; must be caught or
declared to be thrown
```

Exceptions



- **exception**: An object representing a program error.
  - Programs with invalid logic will cause ("*throw*") exceptions.
- Examples:
  - Trying to read a file that does not exist.
  - Dividing by 0.
  - Using `charAt(10)` on a string of length 5.

Checked exceptions

- **checked exception**: An exception that must be explicitly handled (otherwise the program will not compile).
  - We must either:
    - handle ("*catch*") the exception, or
    - explicitly state that we choose not to handle the exception (and accept that the program will crash if the exception occurs)
- Why is a `FileNotFoundException` a checked exception?

`throws` clause: How to waive your rights

- **`throws` clause**: Tells the compiler that a method may throw an exception.
  - Like a waiver of liability:
    "I hereby agree that this method might throw an exception, and I accept the consequences (crashing) if this happens."
- `throws` clause, general syntax:

```
    public static <type> <name>(<params>) throws <type> {
```

- Example:

```
public static void main(String[] args)
        throws FileNotFoundException {
```

Patched code

```
import java.io.*;     // for File, FileNotFoundException
import java.util.*;   // for Scanner

public class ReadFile {
    public static void main(String[] args)
            throws FileNotFoundException {
        Scanner input = new Scanner(new File("data.txt"));
        // do something
    }
}
```

Recap: Tokens

- The `Scanner` breaks apart the input into *tokens*. It will interpret the tokens in different ways depending on if you call `next()`, `nextInt()`, or `nextDouble()`.
- Assuming the following input file:

```
23   3.14
  "John Smith"
```

The tokens in the input can be interpreted as the given types:

Token	Type(s)
1. `23`	`int`, `double`, `String`
2. `3.14`	`double`, `String`
3. `"John`	`String`
4. `Smith"`	`String`

The input cursor

- Consider a file that contains this text:

```
308.2
   14.9 7.4  2.8


3.9 4.7    -15.4
    2.8
```

- A `Scanner` views all input as a stream of characters, which it processes with its *input cursor*:

```
308.2\n   14.9 7.4  2.8\n\n\n3.9 4.7 -15.4\n2.8\n
^
```

Consuming tokens

- Each call to `next`, `nextInt`, `nextDouble`, etc. advances the cursor to the end of the current token, skipping over any whitespace. Each call *consumes* the input.
  - `input.nextDouble();        // 308.2`

```
308.2\n   14.9 7.4  2.8\n\n\n3.9 4.7 -15.4\n2.8\n
     ^
```

  - `input.next();              // "14.9"`

```
308.2\n   14.9 7.4  2.8\n\n\n3.9 4.7 -15.4\n2.8\n
              ^
```

Exercise: Version 1

- Consider an input file named `numbers.dat`:

```
308.2
   14.9 7.4  2.8

3.9 4.7    -15.4
    2.8
```

- Write a program that reads the first 5 values from this file and prints them along with their sum.

Output:

```
number = 308.2
number = 14.9
number = 7.4
number = 2.8
number = 3.9
Sum = 337.19999999999993
```

Solution: Version 1

```
// Displays the first 5 numbers in the given file,
// and displays their sum at the end.

import java.io.*;   // for File, FileNotFoundException
import java.util.*; // for Scanner

public class Echo {
    public static void main(String[] args)
            throws FileNotFoundException {
        Scanner input = new Scanner(new File("numbers.dat"));
        double sum = 0.0;
        for (int i = 1; i <= 5; i++) {
            double next = input.nextDouble();
            System.out.println("number = " + next);
            sum += next;
        }
        System.out.println("Sum = " + sum);
    }
}
```

Version 1 deficiency

- The preceding program is impractical because it only processes exactly 5 values from the input file.
- A better program would read the entire file, regardless of how many values it contained.
- How would we accomplish that?

Look before you read (Section 5.3)

- The `Scanner` has useful methods for testing to see what the next input token will be.

Method Name	Description
`hasNext()`	whether any more tokens remain
`hasNextDouble()`	whether the next token can be interpreted as type `double`
`hasNextInt()`	whether the next token can be interpreted as type `int`

Exercise: Version 2

- Rewrite the previous program so that it reads the entire file.

Output:
```
number = 308.2
number = 14.9
number = 7.4
number = 2.8
number = 3.9
number = 4.7
number = -15.4
number = 2.8
Sum = 329.29999999999995
```

Solution: Version 2

```
// Displays each number in the given file,
// and displays their sum at the end.

import java.io.*;   // for File, FileNotFoundException
import java.util.*; // for Scanner

public class Echo2 {
    public static void main(String[] args)
            throws FileNotFoundException {
        Scanner input = new Scanner(new File("numbers.dat"));
        double sum = 0.0;
        while (input.hasNextDouble()) {
            double next = input.nextDouble();
            System.out.println("number = " + next);
            sum += next;
        }
        System.out.println("Sum = " + sum);
    }
}
```

Exercise: Version 3

- Modify the preceding program again so that it will handle files that contain non-numeric tokens.
  - The program should skip any such tokens.
- For example, the program should produce the same output as before when given this input file:

```
308.2  hello
   14.9 7.4  bad stuff 2.8

3.9 4.7  oops  -15.4
:-)    2.8  @#*($&
```

Solution: Version 3

```
// Displays each number in the given file,
// and displays their sum at the end.

import java.io.*;   // for File, FileNotFoundException
import java.util.*; // for Scanner

public class Echo3 {
    public static void main(String[] args)
            throws FileNotFoundException {
        Scanner input = new Scanner(new File("numbers.dat"));
        double sum = 0.0;
        while (input.hasNext()) {
            if (input.hasNextDouble()) {
                double next = input.nextDouble();
                System.out.println("number = " + next);
                sum += next;
            } else {
                input.next();   // consume / throw away bad token
            }
        }
        System.out.println("Sum = " + sum);
    }
}
```

19

Exercise

- Write a program that accepts an input file containing integers representing daily high temperatures.

  Example input file:
  `42 45 37 49 38 50 46 48 48 30 45 42 45 40 48`

- Your program should print the difference between each adjacent pair of temperatures, such as the following:

```
Temperature changed by 3 deg F
Temperature changed by -8 deg F
Temperature changed by 12 deg F
Temperature changed by -11 deg F
Temperature changed by 12 deg F
Temperature changed by -4 deg F
Temperature changed by 2 deg F
Temperature changed by 0 deg F
Temperature changed by -18 deg F
Temperature changed by 15 deg F
Temperature changed by -3 deg F
Temperature changed by 3 deg F
Temperature changed by -5 deg F
Temperature changed by 8 deg F
```

20

Solution

```
import java.io.*;
import java.util.*;

public class Temperatures {
    public static void main(String[] args)
            throws FileNotFoundException {
        Scanner input = new Scanner(new File("weather.dat"));
        int temp1 = input.nextInt();

        while (input.hasNextInt()) {
            int temp2 = input.nextInt();
            System.out.println("Temperature changed by " +
                               (temp2 - temp1) + " deg F");
            temp1 = temp2;
        }
    }
}
```

21

Line-based processing

Readings: 6.3

22

Line-based processing

- The `Scanner` has the following methods:

Method Name	Description
`nextLine()`	returns the entire next line of input
`hasNextLine()`	whether any more lines remain

23

Who's next in line?

- Reading a file line-by-line, general syntax:

```
Scanner input = new Scanner(new File("<file name>"));
while (input.hasNextLine()) {
    String line = input.nextLine();
    <process this line>;
}
```

- The `nextLine` method returns the characters from the input cursor's current position to the nearest `\n` character.

24

Reading between the newlines

```
23   3.14 John Smith   "Hello world"
              45.2  19

23\t3.14 John Smith\t"Hello world"\n\t\t45.2  19\n
^
```

- `input.nextLine()`

```
23\t3.14 John Smith\t"Hello world"\n\t\t45.2  19\n
                                    ^
```

- `input.nextLine()`

```
23\t3.14 John Smith\t"Hello world"\n\t\t45.2  19\n
                                                  ^
```

- NB: The `\n` character is consumed but not returned.

25

Exercise

- Write a program that reads a text file and "quotes" it by putting a > in front of each line.

Input:

```
Hey,

My students think I stink.  What deodorant should I
use?

Sincerely,
Marty Stepp
```

Output:

```
> Hey,
>
> My students think I stink.  What deodorant should I
> use?
>
> Sincerely,
> Marty Stepp
```

26

Solution

```
import java.io.*;
import java.util.*;

public class QuoteMessage {
    public static void main(String[] args)
            throws FileNotFoundException {
        Scanner input = new Scanner(new File("message.txt"));
        while (input.hasNextLine()) {
            String line = input.nextLine();
            System.out.println(">" + line);
        }
    }
}
```

27

Example

- Example file contents:

```
123 Susan 12.5 8.1 7.6 3.2
456 Brad 4.0 11.6 6.5 2.7 12
789 Jennifer 8.0 8.0 8.0 8.0 7.5
```

- Consider the task of computing the total hours worked for each person represented in the above file.

```
Susan (ID#123) worked 31.4 hours (7.85 hours/day)
Brad (ID#456) worked 36.8 hours (7.36 hours/day)
Jennifer (ID#789) worked 39.5 hours (7.9 hours/day)
```

28

Line-based or token-based?

- Neither line-based nor token-based processing works.
- The better solution is a hybrid approach
  - Break the input into lines.
  - Break each line into tokens.

29

`Scanner`s on `String`s

- A `Scanner` can be constructed to *tokenize* a particular `String` (such as one line of an input file).

```
Scanner <name> = new Scanner(<String>);
```

- Example:

```
String text = "1.4 3.2 hello 9 27.5";
Scanner scan = new Scanner(text);  // five tokens
```

30

Tokenizing lines

```
Scanner input = new Scanner(new File("<file name>"));
while (input.hasNextLine()) {
    String line = input.nextLine();
    Scanner lineScan = new Scanner(line);
    <process this line>;
}
```

Exercise

- Write a program that computes the total hours worked and average hours per day for a particular person represented in the following file:

```
123 Susan 12.5 8.1 7.6 3.2
456 Brad 4.0 11.6 6.5 2.7 12
789 Jennifer 8.0 8.0 8.0 8.0 7.5 7.0
```

Sample runs:

(run #1)

```
Enter a name: Brad
Brad (ID#456) worked 36.8 hours (7.36 hours/day)
```

(run #2)

```
Enter a name: Harvey
Harvey was not found
```

Searching for a line

- When going through the file, how do we know which line to process?
- If we are looking for a particular line, often we look for the token(s) of interest on each line.
  - If we find the right value, we process the rest of the line.
  - e.g. If the second token on the line is "Brad", process it.

Solution

```
// This program searches an input file of employees' hours worked
// for a particular employee and outputs that employee's hours data.

import java.io.*;    // for File
import java.util.*;  // for Scanner

public class HoursWorked {
    public static void main(String[] args) throws FileNotFoundException {
        String searchName = getSearchName();
        String line = getEmployeeData(searchName);
        if (line.length() > 0) {
            processLine(line);
        } else {
            System.out.println(searchName + " was not found");
        }
    }

    public static String getSearchName() {
        Scanner console = new Scanner(System.in);
        System.out.print("Enter a name: ");
        return console.next();       // e.g. "BRAD"
    }
```

Solution

```
    public static String getEmployeeData(String searchName)
        throws FileNotFoundException {
        Scanner input = new Scanner(new File("hours.txt"));
        while (input.hasNextLine()) {
            String line = input.nextLine();
            Scanner lineScan = new Scanner(line);
            lineScan.nextInt();                   // e.g. 456 (no need to save)
            String name = lineScan.next();        // e.g. "Brad"
            if (name.equalsIgnoreCase(searchName)) {
                return line;
            }
        }

        return ""; // search name not found
    }

    ...
```

Solution

```
    // totals the hours worked by one person and outputs their info
    public static void processLine(String line) {
        Scanner lineScan = new Scanner(line);
        int id = lineScan.nextInt();          // e.g. 456
        String name = lineScan.next();        // e.g. "Brad"

        double sum = 0.0;
        int count = 0;
        while (lineScan.hasNextDouble()) {
            sum += lineScan.nextDouble();
            count++;
        }

        double average = round2(sum / count);
        System.out.println(name + " (ID#" + id + ") worked " +
                round2(sum) + " hours (" + average + " hours/day)");
    }

    // returns the given double value rounded to the nearest hundredth.
    public static double round2(double number) {
        return Math.round(number * 100.0) / 100.0;
    }
}
```

Exercise

- Write a program that reads in a file containing HTML text, but with the tags missing their < and > brackets.
  - Whenever you see any all-uppercase token in the file, surround it with < and > before you print it to the console.
  - You must retain the original orientation/spacing of the tokens on each line.

Exercise: Example input

Input file:	Output to console:
`HTML`	`<HTML>`
`HEAD`	`<HEAD>`
`TITLE My web page /TITLE`	`<TITLE> My web page </TITLE>`
`/HEAD`	`</HEAD>`
`BODY`	`<BODY>`
`P There are pics of my cat here,`	`<P> There are pics of my cat here,`
`as well as my B cool /B blog,`	`as well as my <B> cool </B> blog,`
`which contains I awesome /I`	`which contains <I> awesome </I>`
`stuff about my trip to Vegas.`	`stuff about my trip to Vegas.`
`/BODY /HTML`	`</BODY> </HTML>`

Solution

```
import java.io.*;
import java.util.*;

public class WebPage {
    public static void main(String[] args) throws FileNotFoundException {
        Scanner input = new Scanner(new File("page.html"));
        while (input.hasNextLine()) {
            processLine(input.nextLine());
        }
    }

    public static void processLine(String line) {
        Scanner lineScan = new Scanner(line);
        while (lineScan.hasNext()) {
            String token = lineScan.next();
            if (token.equals(token.toUpperCase())) {
                // this is an HTML tag
                System.out.print("<" + token + "> ");
            } else {
                System.out.print(token + " ");
            }
        }
        System.out.println();
    }
}
```

Mixing line-based and token-based methods

```
23   3.14
Joe   "Hello world"
         45.2  19
```

- `console.nextInt();              // 23`
```
23\t3.14\nJoe\t"Hello world"\n\t\t45.2  19\n
  ^
```

- `console.nextDouble();           // 3.14`
```
23\t3.14\nJoe\t"Hello world"\n\t\t45.2  19\n
        ^
```

- `console.nextLine();             // empty string!!`
```
23\t3.14\nJoe\t"Hello world"\n\t\t45.2  19\n
          ^
```

- `console.nextLine();             // "Joe\t\"Hello world\"“`
```
23\t3.14\nJoe\t"Hello world"\n\t\t45.2  19\n
                              ^
```

Mixing line-based and token-based methods

```
Scanner console = new Scanner(System.in);
System.out.print("Enter your age: ");
int age = console.nextInt();
System.out.print("Now enter your name: ");
String name = console.nextLine();
System.out.println(name + " is " + age + " years old.");
```

Sample run:
```
Enter your age: 12
Now enter your name: Marty Stepp
 is 12 years old.
```

- Why?
  - Overall input: `12\nMarty Stepp\n`
  - After `nextInt()`:
```
12\nMarty Stepp\n
  ^
```
  - After `nextLine()`:
```
12\nMarty Stepp\n
    ^
```

Exercise: IMDB

- Consider the following Internet Movie Database (IMDB) Top-250 data

```
1  210,374    9.1     The Godfather (1972)
2  251,376    9.1     The Shawshank Redemption (1994)
3  119,306    8.9     The Godfather: Part II (1974)
```

- Write a program that prompts the user for a search phrase and displays any movies that contain that phrase.

```
This program will allow you to search the
imdb top 250 movies for a particular word.

Search word: part
Rank Votes   Rating  Title
3    119306  8.9     The Godfather: Part II (1974)
66   93470   8.3     The Departed (2006)
98   17710   8.2     The Apartment (1960)
179  26626   7.9     Spartacus (1960)
4 matches.
```

Solution: IMDB

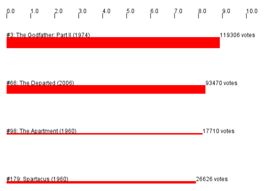
```
// This program reads a file of IMDB's Top 250 movies and
// displays information about movies that match a search
// string typed by the user.

import java.io.*;
import java.util.*;

public class Movies {
    public static void main(String[] args) throws FileNotFoundException {
        introduction();
        String phrase = getWord();
        Scanner input = new Scanner(new File("imdb.txt"));
        search(input, phrase);
    }

    // Prints introductory text to the user
    public static void introduction() {
        System.out.println("This program will allow you to search the");
        System.out.println("imdb top 250 movies for a particular word.");
        System.out.println();
    }

    // Asks the user for their search phrase and returns it.
    public static String getWord() {
        System.out.print("Search word: ");
        Scanner console = new Scanner(System.in);
        String phrase = console.next();
        return phrase.toLowerCase();
    }
```

Solution: IMDB

```
    // Breaks apart each line, looking for lines that match the search phrase.
    // Prints information about each movie that matches the phrase.
    //
    // example line: "2 251,376 9.1 The Shawshank Redemption (1994)"
    public static void search(Scanner input, String phrase) {
        System.out.println("Rank\tVotes\tRating\tTitle");

        int matches = 0;
        while (input.hasNextLine()) {
            String line = input.nextLine();
            Scanner lineScan = new Scanner(line);

            int rank = lineScan.nextInt();
            int votes = lineScan.nextInt();
            double rating = lineScan.nextDouble();
            String title = lineScan.nextLine();  // all the rest
            String lcTitle = title.toLowerCase();

            if (lcTitle.indexOf(phrase) >= 0) {
                matches++;
                System.out.println(rank + "\t" + votes + "\t" + rating + title);
            }
        }
        System.out.println(matches + " matches.");
    }
}
```

Exercise: Graphical IMDB

- Consider making this a graphical program.
- Expected appearance:
  - top-left tick mark at (20, 20)
  - ticks 10px tall, 50px apart
  - first red bar top-left at (20, 70)
  - 100px apart vertically
  - 1px tall per 5000 votes
  - 50px wide per rating point



Mixing graphical and text output

- First, tackle the text and file input/output
  1. Write code to open the input file and print some of the file's data to make sure you're reading the file properly.
  2. Process the input file and retrieve the record being searched for.
  3. Produce the complete and exact text output.
- Then, do the graphical output
  1. Draw any fixed items that do not depend on the user input or file results.
  2. Draw the graphical output that depends on the search record from the file.

Solution: Graphical IMDB

```
// This program reads a file of IMDB's Top 250 movies and
// displays information about movies that match a search
// string typed by the user.

import java.awt.*;
import java.io.*;
import java.util.*;

public class Movies2 {
    public static void main(String[] args) throws FileNotFoundException {
        introduction();
        String phrase = getWord();
        Scanner input = new Scanner(new File("imdb.txt"));
        search(input, phrase);
    }

    // Prints introductory text to the user
    public static void introduction() {
        System.out.println("This program will allow you to search the");
        System.out.println("imdb top 250 movies for a particular word.");
        System.out.println();
    }

    // Asks the user for their search phrase and returns it.
    public static String getWord() {
        System.out.print("Search word: ");
        Scanner console = new Scanner(System.in);
        String phrase = console.next();
        return phrase.toLowerCase();
    }
```

Solution: Graphical IMDB

```
    // Breaks apart each line, looking for lines that match the search phrase.
    // Prints information about each movie that matches the phrase.
    //
    // example line: "2 251,376 9.1 The Shawshank Redemption (1994)"
    public static void search(Scanner input, String phrase) {
        Graphics g = createWindow();
        System.out.println("Rank\tVotes\tRating\tTitle");

        int matches = 0;
        while (input.hasNextLine()) {
            String line = input.nextLine();
            Scanner lineScan = new Scanner(line);

            int rank = lineScan.nextInt();
            int votes = lineScan.nextInt();
            double rating = lineScan.nextDouble();
            String title = lineScan.nextLine();  // all the rest
            String lcTitle = title.toLowerCase();

            if (lcTitle.indexOf(phrase) >= 0) {
                matches++;
                System.out.println(rank + "\t" + votes + "\t" + rating + title);
                drawBar(g, line, matches);
            }
        }
        System.out.println(matches + " matches.");
    }
```

Solution: Graphical IMDB

```
// Creates a drawing panel and draws all fixed graphics
// (graphics unaffected by the file search results)
public static Graphics createWindow() {
    DrawingPanel panel = new DrawingPanel(600, 500);
    Graphics g = panel.getGraphics();

    // draw tick marks
    for (int i = 0; i <= 10; i++) {
        // first tick mark's top-left corner is at (20, 20)
        // 10px tall, 50px apart
        int x = 20 + i * 50;
        g.drawLine(x, 20, x, 30);
        g.drawString(i + ".0", x, 20);
    }

    return g;
}
```

Solution: Graphical IMDB

```
    // Draws one red bar representing a movie's votes and ranking.
    // The "matches" parameter determines the bar's y position.
    public static void drawBar(Graphics g, String line, int matches) {
        Scanner lineScan = new Scanner(line);
        int rank = lineScan.nextInt();
        int votes = lineScan.nextInt();
        double rating = lineScan.nextDouble();
        String title = lineScan.nextLine();  // all the rest

        // draw the red bar for that movie
        // first bar's top-left corner is at (20, 70)
        // 100px apart vertically
        // 1px tall for every 5000 votes earned
        // 50px wide for each ratings point
        int x = 20;
        int y = 70 + 100 * (matches - 1);
        int w = (int) (rating * 50);
        int h = votes / 5000;

        g.setColor(Color.RED);
        g.fillRect(x, y, w, h);

        g.setColor(Color.BLACK);
        g.drawString("#" + rank + ": " + title, x, y);
        g.drawString(votes + " votes", x + w, y);
    }
}
```

Multi-line records

- The following data represents students' course information.

```
Erica Kane
3 2.8 4 3.9 3 3.1
Greenlee Smythe
3 3.9 3 4.0 4 3.9
Ryan Laveree, Jr.
2 4.0 3 3.6 4 3.8 1 2.8
```

  Each student's record has the following format:
  - *Name*
  - *Credits Grade Credits Grade Credits Grade ...*

- How can we process one or all of these records?

File output

Readings: 6.4 (pg. 355 – 359)

Outputting to files

- **`PrintStream`**: A class in the `java.io` package that lets you print output to a destination such as a file.

- `System.out` is a `PrintStream` object!
  - Any methods you have used on `System.out` (such as `print`, `println`) will work on every `PrintStream` object.

Setting up the `PrintStream`

- Setting up an output file, general syntax:

  `PrintStream` ***<name>*** `=`
  `new PrintStream(new File("`***<file name>***`"));`

- Example:

```
PrintStream output = new PrintStream(new File("output.txt"));
output.println("Hello, file!");
output.println("This is a second line of output.");
```

`PrintStream` properties

- *Caution*: Do not open a file for reading (`Scanner`) and writing (`PrintStream`) at the same time.
  - You could overwrite your input file by accident!

55

Exercise

- Write a method named `copy` that takes two filenames and copies the contents from the first file into the second file.

56

Solution

```
public static void copy(String name1, String name2)
    throws FileNotFoundException {
    Scanner input = new Scanner(new File(name1));
    PrintStream output = new PrintStream(new File(name2));

    while (input.hasNextLine()) {
        output.println(input.nextLine());
    }
}
```

57