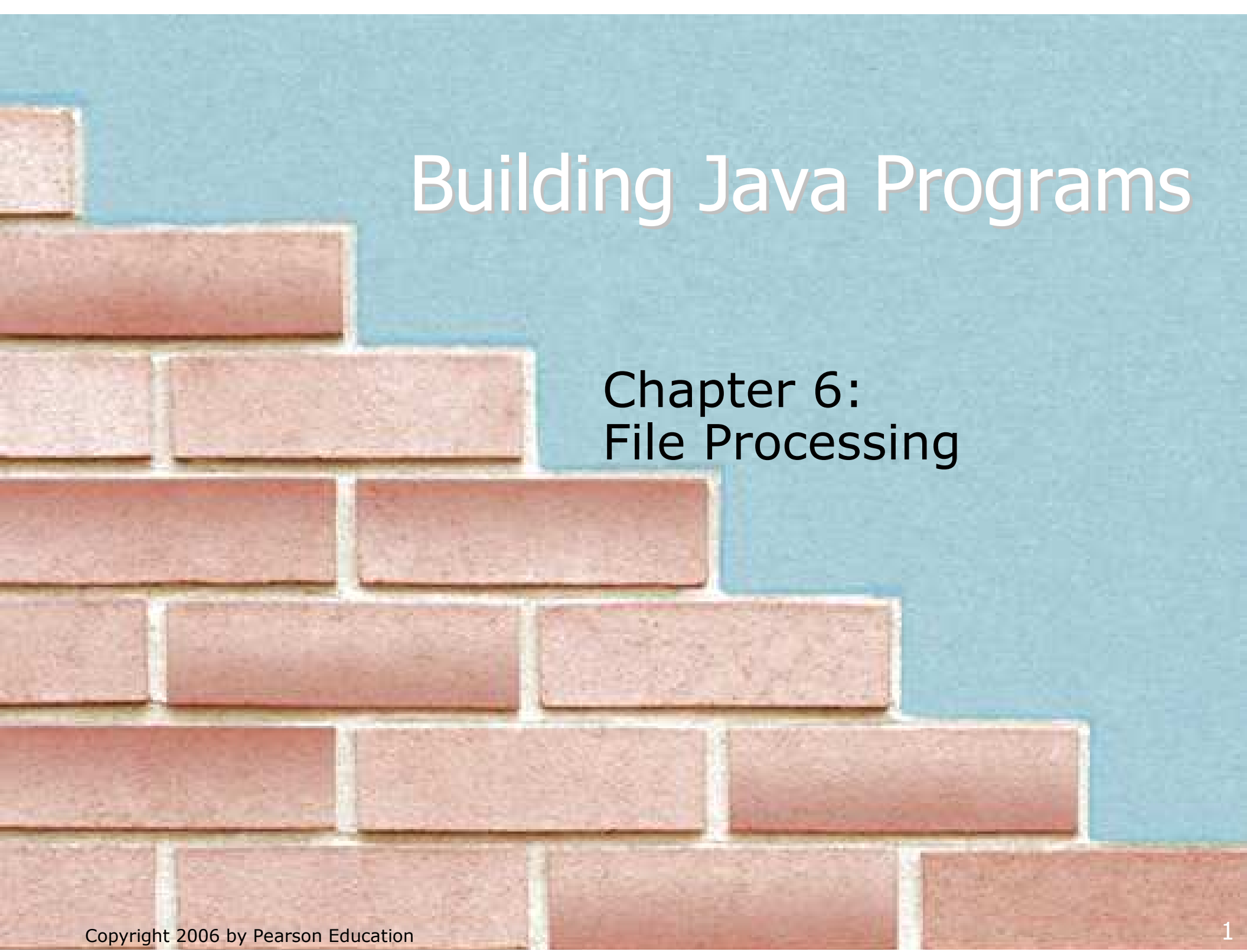


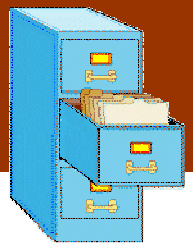
# Chapter outline

- file input using `Scanner`
  - `File` objects
  - exceptions
  - file names and folder paths
  - token-based file processing
- line-based file processing
  - processing a file line by line
  - searching for a particular line record in a file
- advanced I/O
  - prompting for a file name
  - file output using `PrintStream`

# File input using Scanner

reading: 6.1 - 6.2, 5.3

# File objects



- Programmers refer to input/output as "I/O".
- The `File` class in the `java.io` package represents files.
  - `import java.io.*;`
  - Create a `File` object to get information about a file on the disk. (Creating a `File` object doesn't create a new file on your disk.)

```
File f = new File("example.txt");
if (f.exists() && f.length() > 1000) {
    f.delete();
}
```

| Method name | Description |
|---|---|
| `canRead()` | returns whether file is able to be read |
| `delete()` | removes file from disk |
| `exists()` | whether this file exists on disk |
| `getName()` | returns file's name |
| `length()` | returns number of characters in file |
| `renameTo(`*file*`)` | changes name of file |

# Reading data from files

- To read a file, pass a `File` object as a parameter when constructing a `Scanner`.

- `Scanner` for a file, general syntax:

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

Example:

```
Scanner input = new Scanner(new File("numbers.txt"));
```

or:

```
File f = new File("numbers.txt");
Scanner input = new Scanner(f);
```

# File names and paths

- **relative path**: does not specify any top-level folder
  - `"names.dat"`
  - `"input/kinglear.txt"`
- **absolute path**: specifies drive letter or top `"/"` folder
  - `"C:/Documents/smith/hw6/input/data.csv"`
  - Windows systems can also use backslashes to separate folders.
- When you construct a `File` object with a relative path, Java assumes it is relative to the *current directory*.
  - `Scanner input = new Scanner(new File("data/readme.txt"));`
  - If our program is in `H:/hw6`, `Scanner` will look for `H:/hw6/data/readme.txt`.

# Compiler error with files

- The following program does not compile:

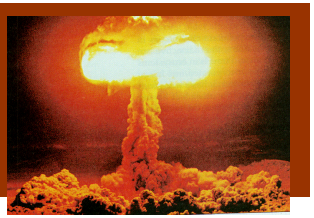
```
import java.io.*;     // for File
import java.util.*;   // for Scanner

public class ReadFile {
    public static void main(String[] args) {
        Scanner input = new Scanner(new File("data.txt"));
        String text = input.next();
        System.out.println(text);
    }
}
```

- The following compiler error is produced:

```
ReadFile.java:6: unreported exception
java.io.FileNotFoundException; must be caught or declared
to be thrown
        Scanner input = new Scanner(new File("data.txt"));
                                    ^
```

# Exceptions



- **exception**: An object that represents a program error.
  - Programs with invalid logic will cause exceptions.
  - Examples:
    - dividing by 0
    - calling `charAt` on a `String` and passing too large an index
    - trying to read a file that does not exist
  - We say that a logical error *throws* an exception.
  - It is also possible to *catch* (handle or fix) an exception.

# Checked exceptions

- **checked exception**: An error that must be handled by our program (otherwise it will not compile).
  - We must specify what our program will do to handle any potential file I/O failures.
  - We must either:
    - declare that our program will handle ("*catch*") the exception, or
    - state that we choose not to handle the exception
      (and we accept that the program will crash if an exception occurs)

# Throwing exception syntax

- **`throws` clause**: Keywords placed on a method's header to state that it may generate an exception.
  - It's 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> {
```

  - When doing file I/O, we use `FileNotFoundException`.

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

# Fixed compiler error

- The following corrected program *does* compile:

```
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"));
        String text = input.next();
        System.out.println(text);
    }
}
```

# Files and input cursor

- Consider a file `numbers.txt` 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:
  - `308.2\n   14.9 7.4  2.8\n\n\n3.9 4.7 -15.4\n2.8\n`
    `^`

- **input cursor**: Current position of the Scanner in the input.

# Input tokens

- **token**: A unit of user input, separated by whitespace.
  - When you call methods such as `nextInt`, the `Scanner` splits the input into tokens.

- Example: If an input file contains the following:

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

  - The `Scanner` can interpret the tokens as the following types:

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

# Consuming tokens

- **consuming input**: Reading input and advancing the cursor.
  - Each call to `next`, `nextInt`, etc. advances the cursor to the end of the current token, skipping over any whitespace.

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

*input.nextDouble()*

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

*input.nextDouble()*

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

# File input question

- Consider the following input file `numbers.txt`:

```
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.

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

# File input answer

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

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

public class Echo {
    public static void main(String[] args)
            throws FileNotFoundException {
        Scanner input = new Scanner(new File("numbers.txt"));
        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);
    }
}
```

# Testing for valid input (Ch5.3)

- A `Scanner` has methods to see what the next token will be:

| Method | Description |
|---|---|
| `hasNext()` | returns `true` if there are any more tokens of input to read *(always true for console input)* |
| `hasNextInt()` | returns `true` if there is a next token and it can be read as an `int` |
| `hasNextDouble()` | returns `true` if there is a next token and it can be read as a `double` |
| `hasNextLine()` | returns `true` if there are any more lines of input to read *(always true for console input)* |

- These methods do not actually consume input, just give information about what input is waiting.

# Scanner condition examples

- The `hasNext` methods are useful to avoid exceptions.

```
Scanner console = new Scanner(System.in);
System.out.print("How old are you? ");
if (console.hasNextInt()) {
    int age = console.nextInt();   // will not crash!
    System.out.println("Wow, " + age + " is old!");
} else {
    System.out.println("You didn't type an integer.");
}
```

- The `hasNext` methods are also useful with file scanners.

```
Scanner input = new Scanner(new File("example.txt"));
while (input.hasNext()) {
    String token = input.next();   // will not crash!
    System.out.println("token: " + token);
}
```

# File input question 2

- The preceding Echo program is impractical; it only processes 5 values from the input file.

- Modify the program to process the entire file:

```
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
```

# File input answer 2

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

import java.io.*;     // for File
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);
    }
}
```

# File input question 3

- Modify the program again to handle files that also contain non-numeric tokens.
  - The program should skip any such tokens.

- For example, it 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  @#*($&
```

# File input answer 3

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

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

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 the bad token
            }
        }
        System.out.println("Sum = " + sum);
    }
}
```

# File processing question

- 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
```

# File processing answer

```
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;
        }
    }
}
```

# Common Scanner errors

- `NoSuchElementException`
  - You read past the end of the input.

- `InputMismatchException`
  - You read the wrong type of token (e.g. read `"hi"` as `int`).

- Finding and fixing these exceptions:
  - Carefully read the exception text for line numbers in your code (the first line that mentions your file; often near the bottom):

```
Exception in thread "main" java.util.NoSuchElementException
        at java.util.Scanner.throwFor(Scanner.java:838)
        at java.util.Scanner.next(Scanner.java:1347)
        at CountTokens.sillyMethod(CountTokens.java:19)
        at CountTokens.main(CountTokens.java:6)
```

# Line-based file processing

reading: 6.3

# Line-by-line processing

- The `Scanner`'s `nextLine` method reads a line of input.
  - It consumes the characters from the input cursor's current position to the next `\n` character.

- 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...>;
}
```

# Line-based input example

- Given the following input data:

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

- The Scanner can read the following input:

```
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
                                          ^
```

  - The \n character is consumed but not returned.

# File processing question

- Write a program that reads a text file and "quotes" it by putting a > in front of each line.
  - Example input file, `message.txt` :

```
Please let the students know that
I'll be curving the grades downward!

Love, Prof. Meanie
```

  - Example output:

```
> Please let the students know that
> I'll be curving the grades downward!
>
> Love, Prof. Meanie
```

# File processing answer

```
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);
        }
    }
}
```

# Processing tokens of one line

- Given a file with the following 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 hours worked by one person:

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

- Observations:
  - Neither line-based nor token-based processing is quite right.
  - The better solution is a hybrid approach:
    - Break the overall input into lines.
    - Break each line into tokens.

# Scanners on Strings

- A `Scanner` can tokenize a `String`, such as a line of a file.

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

  - Example:

```
String text = "1.4 3.2 hello 9 27.5";
Scanner scan = new Scanner(text);
System.out.println(scan.next());     // 1.4
System.out.println(scan.next());     // 3.2
System.out.println(scan.next());     // hello
```

# Tokenizing lines

- We can use string Scanners to tokenize each line of a file.

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

# Line processing example

- Example: Count the tokens on each line of a file.

```
Scanner input = new Scanner(new File("input.txt"));
while (input.hasNextLine()) {
    String line = input.nextLine();
    Scanner lineScan = new Scanner(line);
    int count = 0;
    while (lineScan.hasNext()) {
        String token = lineScan.next();
        count++;
    }
    System.out.println("Line has " + count + " tokens");
}
```

| Input file input.txt: | Output to console: |
|---|---|
| 23 3.14 John Smith "Hello world"<br>45.2 19 | Line has 6 tokens<br>Line has 2 tokens |

# Complex input question

- Write a program that computes the hours worked and average hours per day for a particular person.
  - Input file `hours.txt` :

```
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
```

  - Example log of execution:

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

  - Example log of execution:

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

  - Hint: It may be easier to begin by printing all employee's hours.

# Complex input answer

```
// 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 {
        Scanner console = new Scanner(System.in);
        System.out.print("Enter a name: ");
        String searchName = console.nextLine();   // e.g. "BRAD"
        boolean found = false;                    // a boolean flag

        Scanner input = new Scanner(new File("hours.txt"));
        while (input.hasNextLine()) {
            String line = input.nextLine();
            Scanner lineScan = new Scanner(line);
            int id = lineScan.nextInt();          // e.g. 456
            String name = lineScan.next();        // e.g. "Brad"
            if (name.equalsIgnoreCase(searchName)) {
                processLine(lineScan, name, id);
                found = true;                     // we found them!
            }
        }

        if (!found) {    // found will be true if we ever found the person
            System.out.println(searchName + " was not found");
        }
    }
```

# Complex input answer 2

```
    ...

    // totals the hours worked by one person and outputs their info
    public static void processLine(Scanner lineScan,
            String name, int id) {

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

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

# IMDB movie ratings problem

- Consider the following Internet Movie Database (IMDB) Top-250 data from a text file in the following format:

```
 1  196376   9.1  Shawshank Redemption, The (1994)
 2   93064   8.9  Godfather: Part II, The (1974)
 3   81507   8.8  Casablanca (1942)
```

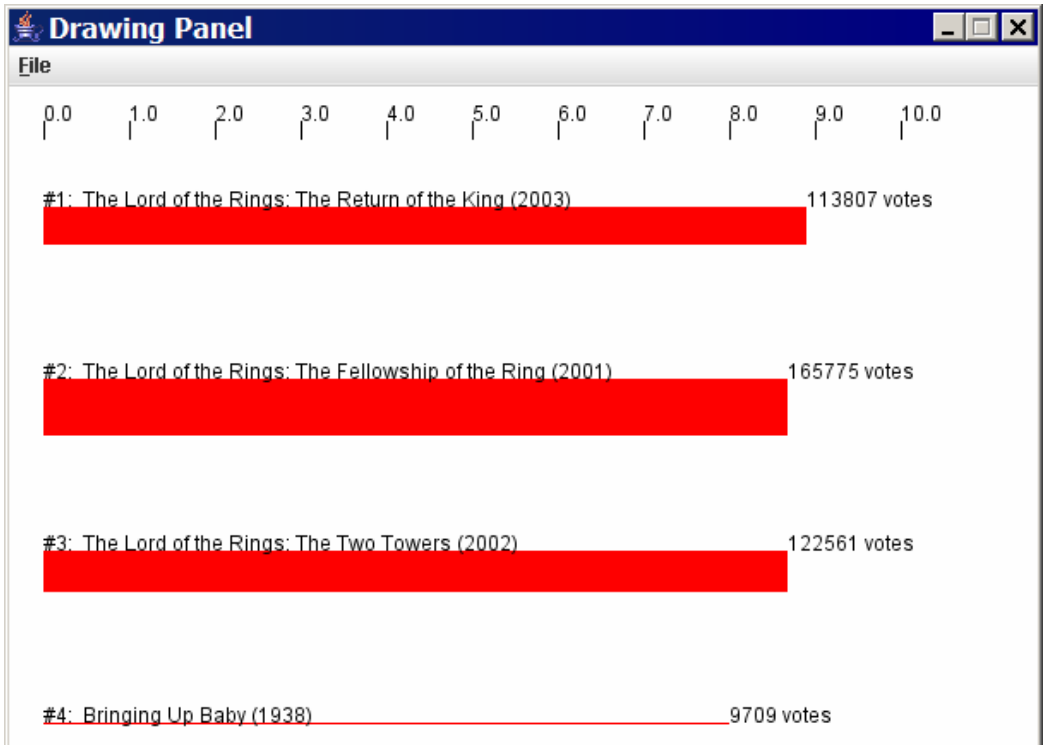
- Write a program that prompts the user for a search phrase and displays any movies that contain that phrase.

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

search word? kill
Rank    Votes    Rating   Title
40      37815    8.5      To Kill a Mockingbird (1962)
88      89063    8.3      Kill Bill: Vol. 1 (2003)
112     64613    8.2      Kill Bill: Vol. 2 (2004)
128     9149     8.2      Killing, the (1956)
4 matches.
```

# Graphical IMDB problem

- Consider making this a graphical program.
  Expected appearance:
  - top-left tick mark at (20, 20)
  - ticks 10px tall, 50px apart
  - first red bar t/l corner at (20, 70)
  - 100px apart vertically (max of 5)
  - 1px tall per 5000 votes
  - 50px wide per rating point



# Mixing graphical, text output

- When solving complex file I/O problems with a mix of text and graphical output, attack the problem in pieces.

  Do the text input/output and file I/O first:

  - Display any welcome message and initial console input.
  - Open the input file and print some file data.
    (Perhaps print the first token of each line, or every token, etc.)
  - Search the input file for the proper line record.

  Next, begin the graphical output:

  - Draw any fixed items that do not depend on the file results.
  - Draw the graphical output that depends on the search result.

# Complex input answer

```
// Displays IMDB's Top 250 movies that match the user's search string.
//
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();
        phrase = phrase.toLowerCase();
        System.out.println();
        return phrase;
    }

    ...
```

# Complex input answer 2

```
...

// Breaks apart each line, looking for lines that match the search phrase.
public static void search(Scanner input, String phrase) {
    System.out.println("Rank\tVotes\tRating\tTitle");
    int matches = 0;
    Graphics g = createWindow();

    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

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

    System.out.println();
    System.out.println(matches + " matches.");
}

...
```

# Complex input answer 3

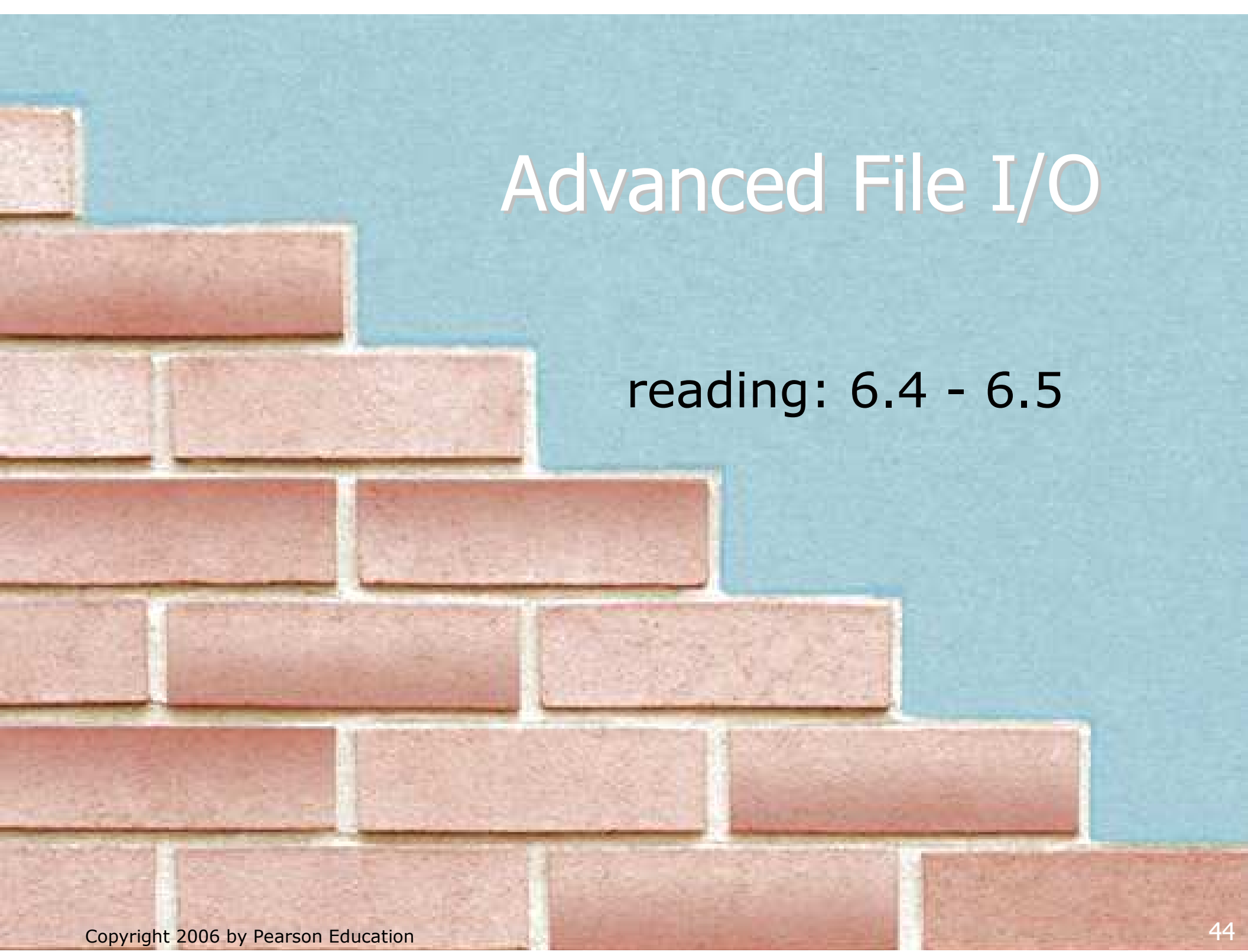
```
...
// Creates a drawing panel and draws all fixed graphics.
public static Graphics createWindow() {
    DrawingPanel panel = new DrawingPanel(600, 500);
    Graphics g = panel.getGraphics();

    for (int i = 0; i <= 10; i++) {      // draw tick marks
        int x = 20 + i * 50;
        g.drawLine(x, 20, x, 30);
        g.drawString(i + ".0", x, 20);
    }

    return g;
}

// Draws one red bar representing a movie's votes and ranking.
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();  // the rest of the line
    int y = 70 + 100 * (matches - 1);
    int w = (int) (rating * 50);
    int h = votes / 5000;

    g.setColor(Color.RED);    // draw the red bar for that movie
    g.fillRect(20, y, w, h);
    g.setColor(Color.BLACK);
    g.drawString("#" + rank + ": " + title, 20, y);
    g.drawString(votes + " votes", 20 + w, y);
}
}
```



# Mixing line-based with tokens

- Don't use both `nextLine` and the token-based methods on the same `Scanner`; confusing results occur.

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


input.nextInt()                                    // 23
23\t3.14\nJoe\t"Hello world"\n\t\t45.2  19\n
  ^

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

input.nextLine()                                   // "" (empty!)
23\t3.14\nJoe\t"Hello world"\n\t\t45.2  19\n
          ^

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

# Line-and-token example

- Another example of the confusing behavior:

```
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.");
```

Log of execution (user input underlined):

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

- Why?
  - User's overall input: `12\nMarty Stepp`
  - After `nextInt()`: `12\nMarty Stepp`
    `^`
  - After `nextLine()`: `12\nMarty Stepp`
    `^`

# Prompting for a file name

- We can ask the user to tell us the file to read.
  - We should use the `nextLine` method on the console `Scanner`, because the file name might have spaces in it.

```
// prompt for the file name
Scanner console = new Scanner(System.in);
System.out.print("Type a file name to use: ");
String filename = console.nextLine();

Scanner input = new Scanner(new File(filename));
```

- What if the user types a file name that does not exist?

# Fixing file-not-found issues

- `File` objects have an `exists` method we can use:

```
Scanner console = new Scanner(System.in);
System.out.print("Type a file name to use: ");
String filename = console.nextLine();
File file = new File(filename);

while (!file.exists()) {
    System.out.print("File not found! Try again: ");
    String filename = console.nextLine();
    file = new File(filename);
}
Scanner input = new Scanner(file);  // open the file
```

Output:

```
Type a file name to use: hourz.text
File not found! Try again: h0urz.txt
File not found! Try again: hours.txt
```

# Output to files

- **`PrintStream`**: An object in the `java.io` package that lets you print output to a destination such as a file.
  - `System.out` is also a `PrintStream`.
  - Any methods you have used on `System.out` (such as `print`, `println`) will work on every `PrintStream`.

- Printing into an output file, general syntax:

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

  - If the given file does not exist, it is created.
  - If the given file already exists, it is overwritten.

# Printing to files, example

- Example:

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

  - You can use similar ideas about prompting for file names here.

- Do not open a file for reading (`Scanner`) and writing (`PrintStream`) at the same time.
  - The result can be an empty file (size 0 bytes).
  - You could overwrite your input file by accident!