Building Java Programs

Chapter 6:
File Processing

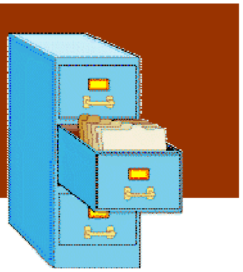
Lecture outline

- file input using `Scanner`
  - `File` objects
  - exceptions
  - file names and folder paths
  - token-based file processing

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 files, pass a `File` 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 file = new File("numbers.txt");
  Scanner input = new Scanner(file);
  ```

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 how our program will handle file I/O failures.
  - We must either:
    - Declare that our program will handle ("*catch*") the exception, or
    - State that we choose not to handle ("*throw*") 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."

- 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
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
              ^
```

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);
    }
}
```

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)
```

Testing for valid input

- 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`

- These methods do not actually consume input.
  - They just give information about what input is waiting.
  - They are useful to see what input is coming, and to avoid crashes.

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

- Modify the Echo 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, `numbers2.dat`:

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

public class Echo3 {
    public static void main(String[] args)
            throws FileNotFoundException {
        Scanner input = new Scanner(new File("numbers2.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();   // throw away 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

```
// Reads temperatures from a file and outputs the difference
// between each pair of neighboring days.

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

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;
        }
    }
}
```