Chapter 12
Text I/O

File Processing

1. Input/Output (I/O) concept
2. Reading from a File
3. File name path
4. Input token
5. Testing for valid inputs
6. Line-based scanner
7. Scanner on strings
8. Output to test files
9. Reading input file name

Input/Output (I/O)

In order to perform I/O, you need to create objects using appropriate Java I/O classes. The objects contain the methods for reading/writing data from/to a file.

You can read/write strings and numeric values from/to a text file using the Scanner and PrintWriter classes.

Input/Output (I/O)

Typical I/O is reading from the keyboard and writing to the screen using `Scanner` objects.

```
import java.util.Scanner;
. . .
Scanner scan = new Scanner(System.in); //default I/O
. . .
System.out.print ("Enter grade: ");
grade = scan.nextInt(); //integer value
. . .
System.out.print ("Enter gallons: ");
gallons = scan.nextDouble(); //double value
. . .
System.out.print ("Enter answer: ");
answer  = scan.nextLine();   //read one line as a string
. . .
System.out.println();
System.out.println ("Average is: " + average);
```

Reading Data Using Scanner

java.util.Scanner	
+Scanner(source: File)	Creates a Scanner object to read data from the specified file.
+Scanner(source: String)	Creates a Scanner object to read data from the specified string.
+close()	Closes this scanner.
+hasNext(): boolean	Returns true if this scanner has another token in its input.
+next(): String	Returns next token as a string.
+nextByte(): byte	Returns next token as a byte.
+nextShort(): short	Returns next token as a short.
+nextInt(): int	Returns next token as an int.
+nextLong(): long	Returns next token as a long.
+nextFloat(): float	Returns next token as a float.
+nextDouble(): double	Returns next token as a double.
+useDelimiter(pattern: String): Scanner	Sets this scanner’s delimiting pattern.

Physical File vs. Logical File

Data is not always in the memory. Often it is in external files in text or binary format.

The actual data file on the storage devise is called **Physical file**.

To read input data from a file, we need to create a ***`File`*** **object** in the program (called **logical file**) to get information about the physical file located on a storage devise. Note that we don't actually create a new physical file on the hard disk, just a **logical file** in the program.

You need to import the IO package: **`import java.io.*;`**

Reading from a File

To read from a file, we need to pass a `File` object that points to the physical file on the storage device when constructing a `Scanner` object.

```
File myFile = new File("mydata.txt"); // physical file name
Scanner input = new Scanner(myFile);  // logical file name
// the scanner object reads from the file on storage device
```

Shorter syntax:

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

See example next slide.

Example

```
// Counts the number of words in text file called Hamlet.txt

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

public class CountWords {
   public static void main(String[] args)
          throws FileNotFoundException //exception handler
   {
     Scanner input = new Scanner(new File("Hamlet.txt"));
     int count = 0;
     while (input.hasNext())
     {
       String word = input.next(); //read one word at a time
       count++;  //increment the counter
     }

     System.out.println("Total # of words = " + count);
   }
}
```

File Name Paths

Absolute Path: specifies a drive or a top-level folder

```
C:/Documents/smith/CS1301/hw13/data.txt
```

(Windows can also use backslashes to separate folders)

```
Scanner input = new Scanner(new
          File("C:/Documents/smith/CS2301/hw13/data.txt"));
```

Relative Path: does not specify any top-level folder

```
names.dat
input/kinglear.txt
```

Assumed to be relative to the *current directory* where the program is saved.

```
Scanner input = new Scanner(new File("data/readme.txt"));
```

If the program is located in `H:/hw4` , object `Scanner` will look for the file in folder `H:/hw4/data/readme.txt`

Input Tokens

Token: A unit of user input, separated by a white space.

The scanner object splits the file content into tokens. This is called Token-Based Scanning.

If an input file contains the following data:

```
23   3.14   true
"John Smith"   A
```

We can read the tokens as following:

Token	**Possible Type(s)**
`23`	`int, double, string`
`3.14`	`double, string`
`True`	`boolean, string`
`"John`	`string`
`Smith"`	`string`
`A`	`character, string`

Reading Input Tokens

Consider file **`Weather.txt`** that contains this text:

```
16.2    23.5
  19.1 7.4  22.8

18.5  -1.8 14.9
```

A **`Scanner`** object views all input as a stream of characters:

```
16.2    23.5\n  19.1 7.4  22.8\n\n18.5  -1.8 14.9\n
 ^
```

Input cursor: The current position of the **`Scanner`**.

Reading Input Tokens

Consuming input is reading input and advancing the <u>input cursor</u> to the next token, skipping white spaces.

Calling methods **`nextDouble()`** or **`next()`** moves the cursor past the current token.

```
16.2    23.5\n  19.1 7.4  22.8\n\n18.5  -1.8 14.9\n
^

double d = input.nextDouble(); // 16.2
16.2    23.5\n  19.1 7.4  22.8\n\n18.5  -1.8 14.9\n
    ^

String s = input.next();       // "23.5" skips blanks
16.2    23.5\n  19.1 7.4  22.8\n\n18.5  -1.8 14.9\n
            ^

String s = input.nextDouble(); // 19.1
16.2    23.5\n  19.1 7.4  22.8\n\n18.5  -1.8 14.9\n
                    ^
```

Example

Write a program that prints the change in temperature between each pair of neighboring days.

```
16.2  to  23.5,  change = 7.3
23.5  to  19.1,  change = -4.4
19.1  to  7.4,   change = -11.7
7.4   to  22.8,  change = 15.4
22.8  to  18.5,  change = -4.3
18.5  to  -1.8,  change = -20.3
-1.8  to  14.9,  change = 16.7
```

The input file:

```
16.2    23.5
  19.1 7.4  22.8

18.5  -1.8 14.9
```

Note the input file has 8 values.

Program Code

```
// Display changes in temperature from data in an input
// file called weather.txt

import java.io.File;          // File class
import java.util.Scanner;     // Scanner class

public class Temperatures
{
   public static void main(String[] args)
              throws FileNotFoundException //exception handler
   {
    Scanner input = new Scanner(new File("weather.txt"));
    double prevTemp = input.nextDouble(); // read first value
    for (int i = 1; i <= 7; i++) // read next 7 values
     { double nextTemp = input.nextDouble();
       System.out.println(prevTemp + "\t to \t" + nextTemp +
                   ",\t change = " + (nextTemp - prevTemp));
       prevTemp = nextTemp;
     }
   }  // is this a reliable code?
}
```

Reading the Entire File

Suppose we want the program to work no matter how many numbers are in the file.

Currently, if the file has more numbers, they will not be read because of the for loop.

If the file has fewer numbers, what will happen?

A crash! Example output from a file with just 3 values:

```
16.2 to 23.5, change = 7.3
23.5 to 19.1, change = -4.4
Exception in thread "main" java.util.NoSuchElementException
   at java.util.Scanner.throwFor(Scanner.java:838)
   at java.util.Scanner.next(Scanner.java:1461)
   at java.util.Scanner.nextDouble(Scanner.java:2387)
   at Temperatures.main(Temperatures.java:15)
```

Error Messages

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

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

Finding and fixing these exceptions:

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.InputMismatchException
   at java.util.Scanner.throwFor(Scanner.java:840)
   at java.util.Scanner.next(Scanner.java:1461)
   at java.util.Scanner.nextDouble(Scanner.java:2387)
   at Temperatures.main(Temperatures.java:15)
```

Testing for Valid Input

Method	**Description**
`hasNext()`	returns `true` if there is a next token
`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 Scanner methods do not consume input; they just give information about what the next token will be.
- They are useful to see what input is coming from the input file, and to avoid program errors (crashes).
- These methods can be used with a keyboard **`Scanner`**, when called on the console, sometimes they pause waiting for input.

Testing for Valid Input

Avoid type mismatches:

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

System.out.print("How old are you? ");

if (input.hasNextInt())
   { int age = input.nextInt(); // won't crash!
    System.out.println("Wow, " + age + " years old!");
   }
else
   System.out.println("You didn't type an integer value.");
```

Testing for Valid Input

Avoid reading past the end of an input file:

```
 //file input
Scanner input = new Scanner(new File("example.txt"));
if (input.hasNext())
 {
   String token = input.next(); // won't crash!
   System.out.println("next token is " + token);
 }
```

Modified Example

Modify the temperature program to process the entire file, regardless of how many numbers it contains.

Example: If a ninth day's data is added, output might be:

```
16.2  to  23.5,  change = 7.3
23.5  to  19.1,  change = -4.4
19.1  to  7.4,   change = -11.7
7.4   to  22.8,  change = 15.4
22.8  to  18.5,  change = -4.3
18.5  to  -1.8,  change = -20.3
-1.8  to  14.9,  change = 16.7
14.9  to  16.1,  change = 1.2
```

Modified Code

```
// Display changes in temperature from data in an input
// file, called weather2.txt

import java.io.File;       // File class
import java.util.Scanner; // Scanner class

public class Temperatures2
{  public static void main(String[] args)
                throws FileNotFoundException //exception handler
   {
      Scanner input = new Scanner(new File("weather2.txt"));
      double prevTemp = input.nextDouble(); // read first value
      while (input.hasNextDouble())  // read the rest of them
      {  double nextTemp = input.nextDouble();
         System.out.println(prevTemp + "\t to \t" + nexTemp +
                    ",\t change = " + (nextTemp - prevTemp));
         prevTemp = nextTemp;
      }
    }
}
```

Mixed-Type Inputs

Now, modify program **`temperature2`** to handle files that contain non-numeric tokens (by skipping them).

For example, it should produce the same output as before when given this input file, **`weather3.txt`**:

```
16.2   23.5
Tuesday   19.1   Wed 7.4   THURS. TEMP: 22.8

18.5  -1.8  <-- Marty here is my data!  --Kim
   14.9 :-)
```

Modified Code

```
// Display changes in temperature from data in an input
// file, called weather3.txt

import java.io.File;      // File class
import java.util.Scanner; // Scanner class

public class Temperatures3
{ public static void main(String[] args)
             throws FileNotFoundException //exception handler
  {  Scanner input = new Scanner(new File("weather3.txt"));
     double prevTemp = input.nextDouble();
     while (input.hasNext())
     { if (input.hasNextDouble())
       { double next = input.nextDouble();
         System.out.println(prevTemp + "\t to \t" + nextTemp +
                    ",\t change = " + (nextTemp - prevTemp));
         prevTemp = nextTemp;
        }
        else input.next(); //skip unwanted tokens
      }
   }
}
```

Line-Based Scanners

Line-Based scanning is the idea or reading one line from a file and then breaking the line into tokens.

This is also know as the “hybrid approach” of token and line-based scanning.

Method	Description
`nextLine()`	returns next entire line of input (from cursor to `\n`)
`hasNextLine()`	returns `true` if there are any more lines of input to read (always true for console input)

Line-Based Scanners

General code:

```
Scanner input = new Scanner(new File("fileName"));

while (input.hasNextLine())
{
   String line = input.nextLine(); // get one line
   // process this line - break into tokens etc.
}
```

Line-Based Scanners

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

The Scanner reads the lines as follows:

```
23\t3.14 John Smith\t"Hello" world\n\t\t45.2  19\n
^
String line = input.nextLine();
23\t3.14 John Smith\t"Hello" world\n\t\t45.2  19\n
                                      ^
String line2 = input.nextLine();
23\t3.14 John Smith\t"Hello" world\n\t\t45.2  19\n
                                                 ^
```

Each **\n** character is read but not returned.

Scanners on Strings

A `Scanner` can tokenize the content of a `String`:

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

Example:

```
String text = "15  3.2 hello   9  27.5";
Scanner scan = new Scanner(text);

int num = scan.nextInt();
System.out.println(num);    // 15

double num2 = scan.nextDouble();
System.out.println(num2);  // 3.2

String word = scan.next();
System.out.println(word);  // hello
. . .
```

Scanners on Strings - Example

Input file **`input.txt`:**	Output to console:

`The quick brown fox jumps over`
`the lazy dog.`
	`This line has 6 words`
	`This line has 3 words`

```
// Counts the words on each line in an input file
Scanner input = new Scanner(new File("input.txt"));
while (input.hasNextLine()) {
  String line = input.nextLine(); //read one line
  Scanner lineScan = new Scanner(line);//define line scanner

  // process the line content, count words per line
  int count = 0;
  while (lineScan.hasNext())
   { String word = lineScan.next(); //get one token from line
     count++; // increment counter
   }
  System.out.println("This line has " + count + " words");
} //end outer while
```

Output to Files – PrintStream

PrintStream: An object in the **`java.io`** package that allows you to print output to a destination, such as a text file on storage media.

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

Syntax:

```
PrintStream sName = new PrintStream(new File("fileName"));
```

Example:

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

Note that java also provide class **`PrintWriter`**.

Output to Files – PrintStream

```
PrintStream sName = new PrintStream(new File("fileName"));
```

Notes:

If the given file name does not exist in current folder, it is created.

If the given file name already exists, it is overwritten. Be Careful!

The output you print appears in the file, not on the screen. You will have to open the file with an editor to see its content.

In the program, do not open the same file for both reading (Scanner) and writing (PrintStream or PrintWriter) at the same time. You will overwrite the input file with an empty file (0 bytes).

Output to Files – PrintStream

Remember, the console output object, **`System.out`**, is a **`PrintStream`**.

```
PrintStream out1 = System.out;
PrintStream out2 = new PrintStream(new File("data.txt"));
out1.println("Hello, console!"); //goes to screen
out2.println("Hello, file!");    //goes to file data.txt
```

Reading File Name

We can ask the user to tell us the file name to read from. The filename might have spaces; use `nextLine()`, not `next()`

```
// prompt for input file name
Scanner scan = new Scanner(System.in);
System.out.print("Please type a file name to use: ");
String filename = scan.nextLine(); // not next()
Scanner input = new Scanner(new File(filename));
```

Class `File` has an `exists()` method to test for file-not-found:

```
File inputFile = new File("hours.txt");
if (!inputFile.exists()) //if method returns false
{
  System.out.print("file hours.txt not found! Try again.");
  // some other code...
}
```

Reading File Name

Some useful methods in class `File`:

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 the file size (number of bytes in the file)
`renameTo(`*fileName*`)`	changes the current file name to a new name (fileName)

java.io.File	
`+File(pathname: String)`	Creates a `File` object for the specified path name. The path name may be a directory or a file.
`+File(parent: String, child: String)`	Creates a `File` object for the child under the directory parent. The child may be a file name or a subdirectory.
`+File(parent: File, child: String)`	Creates a `File` object for the child under the directory parent. The parent is a `File` object. In the preceding constructor, the parent is a string.
`+exists(): boolean`	Returns true if the file or the directory represented by the `File` object exists.
`+canRead(): boolean`	Returns true if the file represented by the `File` object exists and can be read.
`+canWrite(): boolean`	Returns true if the file represented by the `File` object exists and can be written.
`+isDirectory(): boolean`	Returns true if the `File` object represents a directory.
`+isFile(): boolean`	Returns true if the `File` object represents a file.
`+isAbsolute(): boolean`	Returns true if the `File` object is created using an absolute path name.
`+isHidden(): boolean`	Returns true if the file represented in the `File` object is hidden. The exact definition of *hidden* is system-dependent. On Windows, you can mark a file hidden in the File Properties dialog box. On Unix systems, a file is hidden if its name begins with a period(.) character.
`+getAbsolutePath(): String`	Returns the complete absolute file or directory name represented by the `File` object.
`+getCanonicalPath(): String`	Returns the same as `getAbsolutePath()` except that it removes redundant names, such as "." and "..", from the path name, resolves symbolic links (on Unix), and converts drive letters to standard uppercase (on Windows).
`+getName(): String`	Returns the last name of the complete directory and file name represented by the `File` object. For example, new `File("c:\\book\\test.dat").getName()` returns `test.dat`.
`+getPath(): String`	Returns the complete directory and file name represented by the `File` object. For example, new `File("c:\\book\\test.dat").getPath()` returns `c:\book\test.dat`.
`+getParent(): String`	Returns the complete parent directory of the current directory or the file represented by the `File` object. For example, new `File("c:\\book\\test.dat").getParent()` returns `c:\book`.
`+lastModified(): long`	Returns the time that the file was last modified.
`+length(): long`	Returns the size of the file, or 0 if it does not exist or if it is a directory.
`+listFile(): File[]`	Returns the files under the directory for a directory `File` object.
`+delete(): boolean`	Deletes the file or directory represented by this `File` object.The method returns true if the deletion succeeds.
`+renameTo(dest: File): boolean`	Renames the file or directory represented by this `File` object to the specified name represented in dest. The method returns true if the operation succeeds.
`+mkdir(): boolean`	Creates a directory represented in this `File` object. Returns true if the the directory is created successfully.
`+mkdirs(): boolean`	Same as `mkdir()` except that it creates directory along with its parent directories if the parent directories do not exist.

Writing Data Using PrintWriter

java.io.PrintWriter	
+PrintWriter(filename: String)	Creates a PrintWriter for the specified file.
+print(s: String): void	Writes a string.
+print(c: char): void	Writes a character.
+print(cArray: char[]): void	Writes an array of character.
+print(i: int): void	Writes an int value.
+print(l: long): void	Writes a long value.
+print(f: float): void	Writes a float value.
+print(d: double): void	Writes a double value.
+print(b: boolean): void	Writes a boolean value.
Also contains the overloaded println methods.	A println method acts like a print method; additionally it prints a line separator. The line separator string is defined by the system. It is \r\n on Windows and \n on Unix.
Also contains the overloaded printf methods.	The printf method was introduced in §3.6, “Formatting Console Output and Strings.”

PrintStream vs. PrintWriter

The difference is how each write data into the file.

Class PrintStream, defined in JDK 1.0, includes methods to write a **stream of bytes** in a sequential order.

Class PrintWriter, defined in JDK 1.1, includes methods to write a **stream of characters** in a sequential order.

Reading Data from the Web

Create a URL object and use method openStream(), defined in class URL, to open an input stream and use this stream to create a Scanner object as follows:

```
URL url = new URL ("www.google.com/index.html");

Scanner input = new Scanner (url.openStream());
```

See recommended textbook for example.

See recommended textbook for web crawler case study.

End of Slides