Advanced Computer Programming

[Lecture 09]

Saeed Reza Kheradpisheh

kheradpisheh@ut.ac.ir

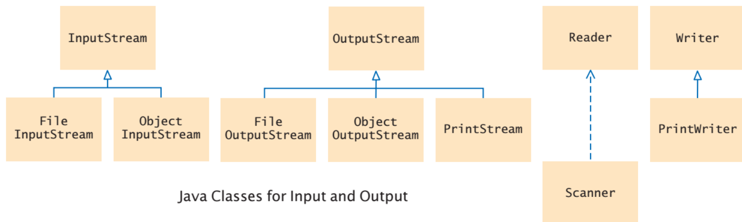
Department of Computer Science
Shahid Beheshti University
Spring 1397-98

INPUT/OUTPUT

Reading and writing files are very useful skills for processing real world data.

Input/Ouput Streams

- There are two fundamentally different ways to store data: in text format or binary format.
- In text format, data items are represented in human-readable form, as a sequence of characters. E.g., ***12,345*** is stored as ***'1' '2' '3' '4' '5'***.
- In binary form, data items are represented in bytes. E.g., ***12,345*** is stored as a sequence of four bytes: ***0 0 48 57***.
- The Java library provides two sets of classes for handling input and output. Streams handle binary data. Readers and writers handle data in text form.



Java Classes for Input and Output

Reading Text Files

- In Java, the most convenient mechanism for reading text is to use the `Scanner` class.

Reading Text Files

- In Java, the most convenient mechanism for reading text is to use the `Scanner` class.
- To read input from a disk file, the `Scanner` class relies on another class, `File`, which describes disk files and directories.
  (`import java.io.File`)
  ```
  File inputFile = new File(fileAddress);
  ```
- Connecting an `Scanner` to the file:
  ```
  Scanner in = new Scanner(inputFile);
  ```

Reading Text Files

- In Java, the most convenient mechanism for reading text is to use the `Scanner` class.
- To read input from a disk file, the `Scanner` class relies on another class, `File`, which describes disk files and directories.
  (`import java.io.File`)
  ```
  File inputFile = new File(fileAddress);
  ```
- Connecting an `Scanner` to the file:
  ```
  Scanner in = new Scanner(inputFile);
  ```
- When you are done processing a file, be sure to **close** the `Scanner` object.
  ```
  in.close();
  ```

Reading Text Files

- In Java, the most convenient mechanism for reading text is to use the `Scanner` class.
- To read input from a disk file, the `Scanner` class relies on another class, `File`, which describes disk files and directories.
  (`import java.io.File`)
  ```
  File inputFile = new File(fileAddress);
  ```
- Connecting an `Scanner` to the file:
  ```
  Scanner in = new Scanner(inputFile);
  ```
- When you are done processing a file, be sure to **close** the `Scanner` object.
  ```
  in.close();
  ```

You can read from files in a same way that you read from the console.

Writing Text Files

- To write output to a file, you construct a `PrintWriter` object with the desired file name.
  (`import java.io.PrintWriter`)
  ```
  PrintWriter out = new PrintWriter(fileAddress);
  ```
- If the output file already exists, it is emptied before the new data are written into it. If the file doesn't exist, an empty file is created.

Writing Text Files

- To write output to a file, you construct a `PrintWriter` object with the desired file name.
  (`import java.io.PrintWriter`)
  `PrintWriter out = new PrintWriter(fileAddress);`
- If the output file already exists, it is emptied before the new data are written into it. If the file doesn't exist, an empty file is created.
- You can use the familiar `print`, `println`, and `printf` methods with any `PrintWriter` object.

Writing Text Files

- To write output to a file, you construct a `PrintWriter` object with the desired file name.
  (`import java.io.PrintWriter`)
  ```
  PrintWriter out = new PrintWriter(fileAddress);
  ```
- If the output file already exists, it is emptied before the new data are written into it. If the file doesn't exist, an empty file is created.
- You can use the familiar `print`, `println`, and `printf` methods with any `PrintWriter` object.
- When you are done processing a file, be sure to **close** the `PrintWriter` object.
  ```
  out.close();
  ```

Writing Text Files

- To write output to a file, you construct a `PrintWriter` object with the desired file name.
  (`import java.io.PrintWriter`)
  ```
  PrintWriter out = new PrintWriter(fileAddress);
  ```
- If the output file already exists, it is emptied before the new data are written into it. If the file doesn't exist, an empty file is created.
- You can use the familiar `print`, `println`, and `printf` methods with any `PrintWriter` object.
- When you are done processing a file, be sure to **close** the `PrintWriter` object.
  ```
  out.close();
  ```

You can write to files in a same way that you write to the console.

We may be in Trouble

- If the file provided for a `Scanner` doesn't exist, a `FileNotFoundException` occurs when the `Scanner` object is constructed.

We may be in Trouble

- If the file provided for a `Scanner` doesn't exist, a `FileNotFoundException` occurs when the `Scanner` object is constructed.
- The `PrintWriter` constructor generates this exception if it cannot open the file for writing.

We may be in Trouble

- If the file provided for a `Scanner` doesn't exist, a `FileNotFoundException` occurs when the `Scanner` object is constructed.
- The `PrintWriter` constructor generates this exception if it cannot open the file for writing.
- The compiler insists that we specify what the program should do in this situation.
- To terminate the `main` method if the exception occurs:
  (`import java.io.FileNotFoundException`)
  ```
  public static void main(String[] args) throws
  FileNotFoundException
  ```

Common Errors

- **Backslashes in File Names**
  - When you specify a file name as a string literal, and the name contains backslash characters (as in a Windows file name), you must supply each backslash twice:

```
File inputFile = new
File("c:\\homework\\input.dat");
```

Common Errors

- **Backslashes in File Names**
  - When you specify a file name as a string literal, and the name contains backslash characters (as in a Windows file name), you must supply each backslash twice:

```
File inputFile = new
File("c:\\homework\\input.dat");
```

- **Constructing a `Scanner` with a `String`**
  - You are not allowed to write the file address directly into the `Scanner` constructor:

```
Scanner in = new Scanner("input.txt"); // Error
```

  - You should create a `File` object first and pass it to the `Scanner` constructor:

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

Exercise (`InvertFile.java`)

Write a program that reads lines from a file and prints them into another file in reverse order.

Working with Text

- Reading Words

  The `next` method reads a string that is delimited by white space.

Working with Text

- Reading Words

  The `next` method reads a string that is delimited by white space.
- Reading Characters

```
Scanner in = new Scanner(. . .);
in.useDelimiter("");
char ch = in.next().charAt(0);
```

Working with Text

- Reading Words

  The `next` method reads a string that is delimited by white space.
- Reading Characters

```
Scanner in = new Scanner(.  .  .);
in.useDelimiter("");
char ch = in.next().charAt(0);
```

- Classifying Characters

  The `Character` class has methods for classifying characters.

Method	Examples of Accepted Characters
`isDigit`	0, 1, 2
`isLetter`	A, B, C, a, b, c
`isUpperCase`	A, B, C
`isLowerCase`	a, b, c
`isWhiteSpace`	space, newline, tab

Exercise (`Count.java`)

Write a program that counts both number of digits and number of letters in a file.

Working with Text

- Reading Lines

  The `nextLine` method reads an entire line including its white-space characters (except the newline character).

  ```
  String line = in.nextLine();
  ```

Working with Text

- Reading Lines

  The `nextLine` method reads an entire line including its white-space characters (except the newline character).

  ```
  String line = in.nextLine();
  ```

- Scanning a String

  You can use a `Scanner` object to read the characters from a string:

  ```
  Scanner lineScanner = new Scanner(line);
  ```

Working with Text

- Reading Lines
  The `nextLine` method reads an entire line including its white-space characters (except the newline character).
  ```
  String line = in.nextLine();
  ```
- Scanning a String
  You can use a `Scanner` object to read the characters from a string:
  ```
  Scanner lineScanner = new Scanner(line);
  ```
- Converting Strings to Numbers
  If a string contains the digits of a number, you use the `Integer.parseInt` or `Double.parseDouble` method to obtain the number value.

Working with Text

- Avoiding Errors When Reading Numbers
  if the input is not a properly formatted number, an “input mismatch exception” occurs.
  To avoid exceptions, use the `hasNextInt` method to screen the input when reading an integer.

```
if (in.hasNextInt()) ...
```

Formatting Output

additional options of the `printf` method

A format specifier has the following structure:

- The first character is a %.
- Next, there are optional "flags" that modify the format.
- Next is the field width, the total number of characters in the field (including the spaces used for padding), followed by an optional precision for floating-point numbers.
- The format specifier ends with the format type.

Formatting Output

Table 2 Format Flags

Flag	Meaning	Example
-	Left alignment	1.23 followed by spaces
0	Show leading zeroes	001.23
+	Show a plus sign for positive numbers	+1.23
(	Enclose negative numbers in parentheses	(1.23)
,	Show decimal separators	12,300
^	Convert letters to uppercase	1.23E+1

Formatting Output

Table 3 Format Types

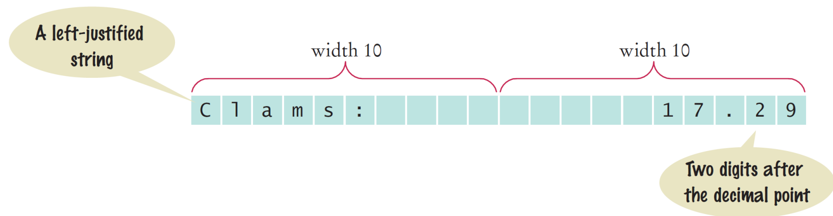
Code	Type	Example
d	Decimal integer	123
f	Fixed floating-point	12.30
e	Exponential floating-point	1.23e+1
g	General floating-point (exponential notation is used for very large or very small values)	12.3
s	String	Tax:

Formatting Output

Example:

```
System.out.printf("%-10s%10.2f", items[i] + ":", prices[i]);
```

```
Cookies:       3.20
Linguine:      2.95
Clams:        17.29
```



Command Line Arguments

Usage

When you invoke a program from the command line (typing `java` and the name of the program) you can also type in additional information that the program can use. These additional strings are called **command line arguments** (arguments for the `main` method).

Command Line Arguments

Usage

When you invoke a program from the command line (typing `java` and the name of the program) you can also type in additional information that the program can use. These additional strings are called **command line arguments** (arguments for the `main` method).

Example:

```
java ProgramClass -v input.dat
```

- The program receives two command line arguments: the strings `"-v"` and `"input"`.

Command Line Arguments

Usage

When you invoke a program from the command line (typing `java` and the name of the program) you can also type in additional information that the program can use. These additional strings are called **command line arguments** (arguments for the `main` method).

Example:

```
java ProgramClass -v input.dat
```

- The program receives two command line arguments: the strings `"-v"` and `"input"`.
- The program receives its command line arguments in the `args` parameter of the `main` method.

```
args[0]:   "-v"
args[1]:   "input.dat"
```

Exercise

Write a program that encrypts a file–that is, scrambles it so that it is unreadable except to those who know the decryption method.

Encryption works as follows:

Replacing A with a D, B with an E, and so on,

Plain text	M	e	e	t		m	e		a	t		t	h	e		
Encrypted text	P	h	h	w		p	h		d	w		w	k	h		

The program takes the following command line arguments:

- An optional `-d` flag to indicate decryption instead of encryption.
- The input file name
- The output file name

EXCEPTION HANDLING

There are two aspects to dealing with program errors: **detection** and **handling**.

Exception handling provides a flexible mechanism for passing control from the point of error detection to a handler that can deal with the error.

Throwing Exceptions

Usage

To signal an exceptional condition, use the **`throw`** statement to throw an exception object.

Syntax `throw` *exceptionObject*;

```
if (amount > balance)
{
   throw new IllegalArgumentException("Amount exceeds balance");
}
balance = balance - amount;
```

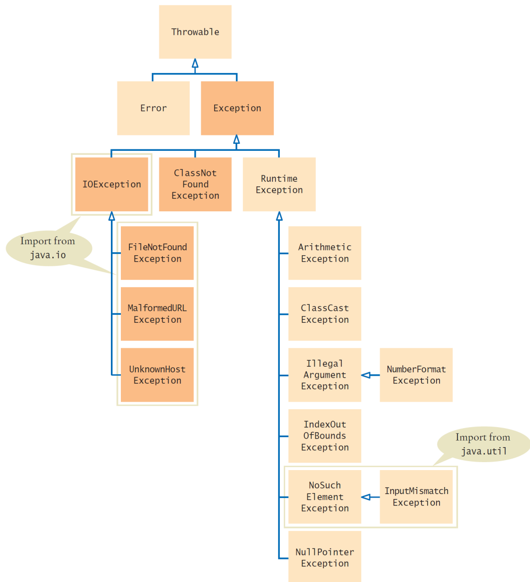
A new exception object is constructed, then thrown.

Most exception objects can be constructed with an error message.

This line is not executed when the exception is thrown.

The Java library provides many classes to signal all sorts of exceptional conditions.

EXCEPTION hierarchy



Catching Exceptions

When you throw an exception, processing continues in an exception handler.

Usage

Place the statements that can cause an exception inside a `try` block, and the handler inside a **`catch`** clause.

Catching Exceptions

Syntax

```
try
{
   statement
   statement
   . . .
}
catch (ExceptionClass exceptionObject)
{
   statement
   statement
   . . .
}
```

```
try
{
   Scanner in = new Scanner(new File("input.txt"));
   String input = in.next();
   process(input);
}
catch (IOException exception)
{
   System.out.println("Could not open input file");
}
catch (Exception except)
{
   System.out.println(except.getMessage);
}
```

This constructor can throw a FileNotFoundException.

This is the exception that was thrown.

When an IOException is thrown, execution resumes here.

A FileNotFoundException is a special case of an IOException.

Additional catch clauses can appear here. Place more specific exceptions before more general ones.

Catching Exceptions

```
try
{
   String filename = . . .;
   Scanner in = new Scanner(new File(filename));
   String input = in.next();
   int value = Integer.parseInt(input);
   . . .
}
catch (IOException exception)
{
   exception.printStackTrace();
}
catch (NumberFormatException exception)
{
   System.out.println(exception.getMessage());
}
```

Three exceptions may be thrown in this `try` block:

- The `Scanner` constructor can throw a `FileNotFoundException`.
- `Scanner.next` can throw a `NoSuchElementException`.
- `Integer.parseInt` can throw a `NumberFormatException`.

Catching Exceptions

- If a `FileNotFoundException` is thrown, then the `catch` clause for the `IOException` is executed. (If you look at Figure 2, you will note that `FileNotFoundException` is a descendant of `IOException`.) If you want to show the user a different message for a `FileNotFoundException`, you must place the `catch` clause *before* the clause for an `IOException`.
- If a `NumberFormatException` occurs, then the second `catch` clause is executed.
- A `NoSuchElementException` is *not caught* by any of the `catch` clauses. The exception remains thrown until it is caught by another `try` block.

Checked Exceptions

Definition

Checked exceptions are due to external circumstances that the programmer cannot prevent. The compiler checks that your program handles these exceptions.

Checked Exceptions

Definition

Checked exceptions are due to external circumstances that the programmer cannot prevent. The compiler checks that your program handles these exceptions.

In Java, the exceptions that you can throw and catch fall into three categories:

- Internal errors are reported by descendants of the type `Error`.
- Descendants of `RuntimeException`, such as as `IndexOutOfBoundsException` or `IllegalArgumentException` indicate errors in your code (Unchecked Exceptions).
- All other exceptions are checked exceptions. These exceptions indicate that something has gone wrong for some external reason beyond your control.

Catching Exceptions

```
try
{
   File inFile = new File(filename);
   Scanner in = new Scanner(inFile); // Throws FileNotFoundException
   . . .
}
catch (FileNotFoundException exception) // Exception caught here
{
   . . .
}
```

However, it commonly happens that the current method *cannot handle* the exception. In that case, you need to tell the compiler that you are aware of this exception and that you want your method to be terminated when it occurs. You supply a method with a `throws` clause.

```
public static String readData(String filename) throws FileNotFoundException
{
   File inFile = new File(filename);
   Scanner in = new Scanner(inFile);
   . . .
}
```

The Finally Clause

Usage

Once a `try` block is entered, the statements in a **`finally`** clause are guaranteed to be executed, whether or not an exception is thrown.

Example:

```
PrintWriter out = new PrintWriter(filename);
try
{
   writeData(out);
}
finally
{
   out.close();
}
```

The Finally Clause

Syntax

```
try
{
   statement
   statement
   . . .
}
finally
{
   statement
   statement
   . . .
}
```

```
PrintWriter out = new PrintWriter(filename);
try
{
   writeData(out);
}
finally
{
   out.close();
}
```

This variable must be declared outside the `try` block so that the `finally` clause can access it.

This code may throw exceptions.

This code is always executed, even if an exception occurs.

Exercise

Add exception handling to the previous exercise.