COMP122 Week 10

UNIVERSITY OF LIVERPOOL

Dr. Patrick Totzke
totzke@liverpool.ac.uk

https://liverpool.instructure.com/courses/59716

Java I/O

Forms of input/output in Java

- Command-line arguments: `main(String[] args)`
- `System.in` and `System.out`
- Reading and writing `File`s
- Graphical User Interfaces (GUIs): `TextField`s, mouse-clicks etc.

Streams

A *Stream* is an endless flow of data. One can read from a stream or write to a stream. A stream is connected to a data source or a data destination.

- Streams are objects
- different sources and destinations
- bits/bytes/characters/objects as basic units

Streams – Base Types

From the beginning, I/O was realized in Java using the concept of streams, implemented in an object oriented design.

Java has two basic kinds of streams:

- Bytes-based: `InputStream`/`OutputStream`
- Character-based: `Reader`/`Writer`

These `abstract` classes, together with more specific implementations and related stuff can be found in `java.io`.

java.io

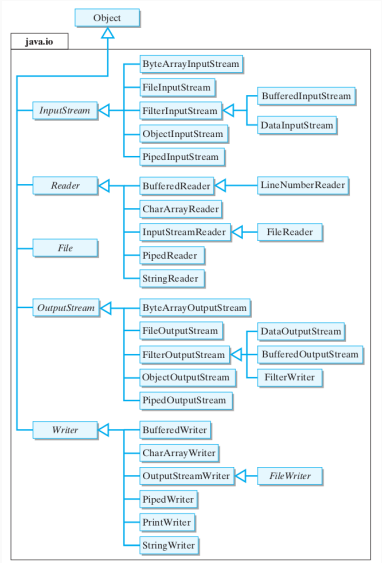

TABLE 11.1 Description of some of Java's important stream classes.

Class	Description
`InputStream`	Abstract root class of all binary input streams
`FileInputStream`	Provides methods for reading bytes from a binary file
`FilterInputStream`	Provides methods required to filter data
`BufferedInputStream`	Provides input data buffering for reading large files
`ByteArrayInputStream`	Provides methods for reading an array as if it were a stream
`DataInputStream`	Provides methods for reading Java's primitive data types
`PipedInputStream`	Provides methods for reading piped data from another thread
`OutputStream`	Abstract root class of all binary output streams
`FileOutputStream`	Provides methods for writing bytes to a binary file
`FilterOutputStream`	Provides methods required to filter data
`BufferedOutputStream`	Provides output data buffering for writing large files
`ByteArrayOutputStream`	Provides methods for writing an array as if it were a stream
`DataOutputStream`	Provides methods for writing Java's primitive data types
`PipedOutputStream`	Provides methods for writing piped data to another thread
`PrintStream`	Provides methods for writing primitive data as text
`Reader`	Abstract root class for all text input streams
`BufferedReader`	Provides buffering for character input streams
`CharArrayReader`	Provides input operations on `char` arrays
`FileReader`	Provides methods for character input on files
`FilterReader`	Provides methods to filter character input
`StringReader`	Provides input operations on `Strings`
`Writer`	Abstract root class for all text output streams
`BufferedWriter`	Provides buffering for character output streams
`CharArrayWriter`	Provides output operations to `char` arrays
`FileWriter`	Provides methods for output to text files
`FilterWriter`	Provides methods to filter character output
`PrintWriter`	Provides methods for printing binary data as characters
`StringWriter`	Provides output operations to `Strings`

(from Morelli, sec. 11)

Bits/Bytes

... are low-level and uninterpreted representations of information.

Bytes

- 8-bit
- Nomenclature: 1950's computer reads a chunk from a tape ("bite" $\rightsquigarrow$ "byte")
- 8-bit length popularized by rise of 8-bit microprocessors
- 4-bits are sometimes called a "nyble"

Java has corresponding data types `byte` (primitive) and `Byte` (wrapper).

Byte-based Streams in Java

```
1  InputStream inputstream = new FileInputStream("textfile.txt");
2
3  int data = inputstream.read();
4
5  while(data != -1) {
6    char c = (char)data;     // converts integer to character
7    System.out.print(c);     // prints character
8    data = inputstream.read();
9  }
10
11 inputstream.close();
```

- line 1 opens a stream with source `textfile.txt`
- `inputstream.read()` reads a single byte and returns an `int`
- if the file end is reached, `read()` returns `-1`.
- streams need to be closed (line 7) after use.

Byte-based Streams – Safer version

```
1  try {
2      is = new FileInputStream("textfile.txt");
3      while((i = is.read())!=-1) {
4          char c = (char)i;          // converts integer to character
5          System.out.print(c);       // prints character
6      }
7  } catch (IOException e) {
8      e.printStackTrace();           // if any I/O error occurs
9  } finally {
10      is.close();                   // releases system resources
11 }
```

This guarantees that the stream gets closed even if exceptions occur.

Byte-based Streams – Fancy versions

Java ≥ 7 has a "try-with-resource" construct which automatically closes the stream.

```
try( InputStream inputstream = new FileInputStream("textfile.txt") ) {
    while((i = inputstream.read())!=-1) {
        char c = (char)i;        // converts integer to character
        System.out.print(c);       // prints character
    }
} catch (IOException e) {
    e.printStackTrace();
}
```

This works because `InputStream` implements the `AutoCloseable` interface.

We can use `available()` before reading to check if the stream is empty.

```
while (inputstream.available() > 0)  {
   is.read()
   ...
}
```

Characters

... are higher-level representations of single symbols.

Different encodings into bits/bytes are possible and in use:

- ASCII-code[1] to translate between bytes and symbols commonly used in the west. 8-bits; `char=byte`;
- Unicode: popular encodings are UTF-8 and UTF-16

Java has corresponding data types `char` (primitive; 16 bit) and `Character` (wrapper) and uses UTF-16 internally.

[1] American Standard Code for Information Interchange

Character-based Streams

Recall that in `java.io`'s nomenclature, `Stream`s are bytes-based; `Reader`s (and `Writer`s) are character-based.

An example of reading a text file using a `FileReader` is below.

```
1  try (FileReader fr=new FileReader("textfile.txt")){
2      while((i=fr.read())!=-1){       // this reads a (16-bit!) Character
3          System.out.print((char)i);
4      }
5  } catch (IOException e) {
6      e.printStackTrace();
7  }
```

Nested Streams

One thing that makes Java's Streams powerful is that they can be nested.

You can create a type of Stream that internally uses another, lower-level stream for the "plumbing".

Examples include:

- Adding write/read Buffers to increase efficiency
- Filtering the stream contents
- Object-based streams

Nested Streams Example – The BufferedWriter

```
1  FileWriter fw = null;
2  BufferedWriter writer = null;
3
4  try (fw = new FileWriter (args[0])) {
5     try (writer = new BufferedWriter(fw)) {
6         writer.write("The quick brown fox jumped over the lazy dogs.");
7         writer.newLine();
8         writer.write("" + 200);
9         writer.newLine();
10    }
11 }
12 catch (IOException err)  {
13    err.printStackTrace();
14 }
```

Nested Streams Example – The BufferedWriter (cont.)

- The `BufferedWriter` (line 5) uses `FileWriter` (line 4) internally.
- Buffers allows to delay writing and then write in bulk. This is usually faster compared to multiple small modifications.
- The constructor for the `FileWriter` will create (or overwrite!!) the file. To append to a file, use the alternative constructor with `boolean` second parameter.

```
1  fw = new FileWriter(args[0], true);  //  append to the file
```

- I should make certain to flush the buffer (or close it) to ensure that all of the information in the buffer is written to the file.
- The `newLine()` method writes a (platform dependent) newline character

Reading and writing to the terminal

System.in, System.out and System.err

Standard input (stdin) is the source of input data for command line programs and defaults to reading from the keyboard. Standard output (stdout) correspondingly is the default data "sink", which by default prints on the terminal. A third such sink is (stderr) – which is where error messages are written to.

One of the joys of using a shell (the command interpreter) is that you can connect the output of one program to the input of others using a "pipe" (`|`).

In Java, you can access those as `System.in`, `System.out`, and `System.err`. These objects are streams and just like `FileReader` and `FileWriter`, they offer `write`, and `read` methods (among other convenience methods like `println`).

`https://docs.oracle.com/javase/7/docs/api/java/lang/System.html`

System.in, System.out and System.err

```
1  import java.io.IOException;
2
3  public class EchoChar {
4
5     /* adding "throws IOException" here is an ugly hack
6      * to demonstrate the use of reading from System.in without
7      * having introduced Exceptions yet. In your own code you
8      * should catch exceptions. */
9     public static void main (String args[]) throws IOException {
10
11       //  The System.in.read() method can throw an IOException
12       char myChar = (char) System.in.read();
13
14       System.out.write(myChar);  // one way to write a character
15
16       System.out.println(myChar);  // another way
17    }
18 }
```

Some more convenience: Scanners

Javas standard library has this neat little class that allows you to read a bunch of different data: `java.util.Scanner` offers the following methods.

- `nextBoolean()`
- `nextByte()`
- `nextDouble()`
- `nextFloat()`
- `nextInt()`
- `nextLine()`
- `nextLong()`
- `nextShort()`

`Scanner` also has a number of constructors, that allow to read directly from files, paths, or any `InputStream`.

For instance, `Scanner(System.in)` will read from standard input, whereas `Scanner("somefile.txt")` will read from a text file.

`https://docs.oracle.com/javase/7/docs/api/java/util/Scanner.html`

Some more convenience: Scanners

```java
import java.util.Scanner;  // Import the Scanner class

public class ConvenientInput {
  public static void main(String[] args) {

    // Create a Scanner object
    Scanner myScanner = new Scanner(System.in);

    // Read user input
    System.out.println("What's your name?");
    String name = myScanner.nextLine();

    System.out.println("How old are you?");
    int age = myScanner.nextInt();

    // Output user input
    System.out.println(name + " is " + age + " years old.");
  }
}
```

Files

File handles

The class `java.io.File` is used to represent files in the operating system and provides methods to interact with the file system such as deleting files, renaming files, making directories, and listing the files in a directory.

```
....
File f1 = new File("output.txt");
File f2 = new File("stuff.out");
if (f1.renameTo(f2))  {
   System.out.println("File rename successful");
}
else  {
   System.out.println("File rename failed");
}
....
```

NB: File handlers are objects!

Caution! Not all platforms behave the same...

File handling is complicated by the fact that you want to write code that will work across different operating systems.

For example, Windows uses the backslash (\) as separator between directory names, whereas other platforms use the forward slash (/) for this.

You can use the `separator` attribute of the `File` class here.

```
1  import java.io.File;
2
3  String sep = File.separator;
4  File fn = new File("data" + sep + "output.txt");
5  // fn will now point to "data/output.txt" on Unix and friends,
6  // and to "data\output.txt" on Windows
```

Example: Counting Lines

Example: Reading input from a File

Suppose we want to count the total number of lines in a text file.

The following example does this iteratively, and uses

- a file handler of type `java.io.File`
- a `java.util.Scanner` to read its content, one line of text at a time

We will accept the name of the file to read as the first command-line argument. Then the line which attempts to open it may throw a `java.io.FileNotFoundException`.

Counting lines in a text file (I)

```
1  import java.io.File;                    //  import the File object
2  import java.io.FileNotFoundException;   //  import the Exception
3  import java.util.Scanner;
4
5  /**
6   * A class that demonstrates how to check for a "file not found" error.
7   * @author Patrick Totzke
8   * @version 1
9   */
10 public class CountLinesInFile {
11
12    public static void main (String args[]) {
13
14       File fileHandler;  //  object representing the file
15       Scanner scanner;   //  object to read from a file.
16       int counter = 0;   //  counts the lines of text.
17
18       String fileName = "text-file.txt";
19       fileHandler = new File(fileName);  //  create File object
```

Counting lines in a text file (II)

```
20
21          try{
22              //  Attempt to open the file. May throw FileNotFoundException
23              scanner = new Scanner(fileHandler);
24
25              //  file was opened successfully. Iterate through it..
26              while(scanner.hasNextLine()){
27                  scanner.nextLine();
28                  counter++;
29              }
30              System.out.println(fileName + " has " + counter + " lines.");
31          }
32          catch (FileNotFoundException fnfe)  {
33              System.out.println(fnfe);  //  Oops, print out the error!
34          }
35      }  //  end of "main" method
36  }
```

Testing out CountLinesInFile

```
$> javac CountLinesInFile.java
$> ls
CountLinesInFile.class  CountLinesInFile.java  text-file.txt

$> java CountLinesInFile
text-file.txt has 33 lines.

$> rm text-file.txt
$> java CountLinesInFile
java.io.FileNotFoundException: text-file.txt (No such file or directory)
```

If we give a file name that doesn't exist then in line 22 a `FileNotFoundException` is thrown and the runtime env hands control over to the handler defined in line 31 (which prints to stdout).

Getting the file name as parameter

Here is where we can use the parameter `args` of the `main` method. This array contains all the command-line arguments that are passed to the program when it is called.

For example, if I were to execute

```
$ java CountLinesInFile text-file.txt
```

then the first (and only) element in the `args` array is the string `"text-file.txt"`.

If I were to execute

```
$ java CountLinesInFile text-file.txt  file2.txt
```

then `args[0]` is equal to `"text-file.txt"` and `args[1]` is equal to `"file2.txt"`.

Let's test a bit more..

What happens if I don't supply a filename at all?

```
$ java CountLinesInFile2
Exception in thread "main" java.lang.ArrayIndexOutOfBoundsException: 0
   at CountLinesInFile2.main(CountLinesInFile2.java:17)
```

...because I try to access the first element of a length-0 array.

Q: is `ArrayIndexOutOfBoundsException` checked or unchecked?

Exercises

- modify our program to handle this case (solution in `CountLinesInFile3.java`)
- Do the same using try-with-resource and a Reader (`CountLinesInFile4.java`)

Your Questions?

A3

Summary

We looked at...

- Java I/O
  - Streams
  - Readers/Writers
  - BufferedReader, FileInputStream, etc.
- Requirements for A3

Next Week

- Tue: Graphical User Interfaces
- Thu: Q&A
- Fri: A3 deadline