G51PRG:
Introduction to Programming
Second semester
Lecture 9

Natasha Alechina
School of Computer Science & IT
nza@cs.nott.ac.uk

Previous lecture: exceptions

- what are exceptions for
- how to define your own exception
- how to get a method to throw an exception
- how to catch and handle exceptions

This lecture: I/O

- I/O in Java
- Streams
- Reading, writing, handling exceptions
- Files
- Parsing

I/O in Java

- Classes necessary to handle I/O are provided by package java.io (not the most elegant part of Java).
- All examples in this lecture assume that you add

```
import java.io.*;
```

General idea

- When you need to read data into the program or write it out of the program, you open a stream between the program and the source (or destination) of data;
- the stream does reading or writing (it has corresponding methods) ;
- when it is finished, you close the stream.

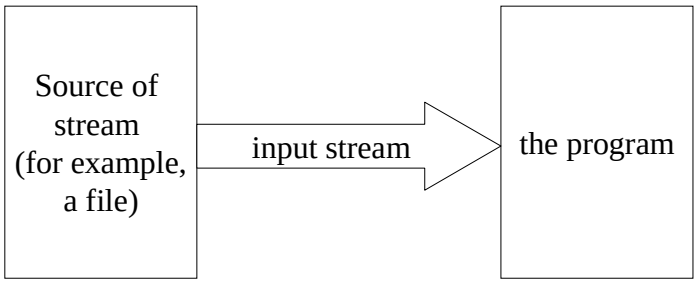
Input and output streams

- Input streams: get data from elsewhere into the program, for example:
  - from a file into the program
  - from keyboard input into the program
- Output streams: transferring data from the program to an outside source, for example:
  - writing data out to a file
  - sending output to the screen

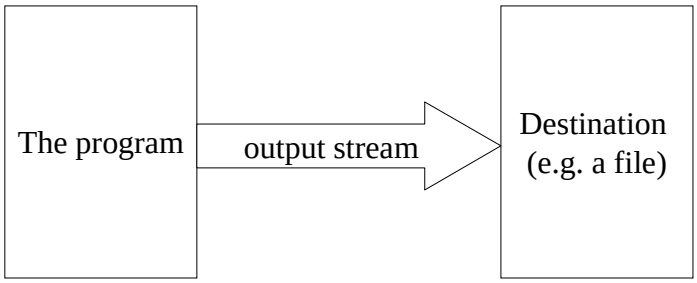
Streams

- Stream is a sequence of data.
- Byte stream carries 8 bit items of data, character streams carry 16 bit Unicode characters.
- Byte streams are called input and output streams, character streams readers and writers (just a convention).

Input stream (reader)



Output stream (writer)



InputStream class

- InputStream is an abstract class which provides methods for reading bytes from a particular source.
- Subclasses: BufferedInputStream, FilterInputStream, FileInputStream, ObjectInputStream,...
- Important method:

```
public abstract int read() throws
  IOException
```

- reads a single byte of data and returns the byte that was read, in the range 0 to 255. The value -1 is returned when the end of stream is reached. Note that the return type is int.

OutputStream class

- OutputStream class is an abstract class which provides methods for writing bytes to a destination.
- Subclasses: BufferedOutputStream, FilterOutputStream, FileOutputStream, ObjectOutputStream,...
- Important method:

```
public abstract void write(int b) throws
  IOException
```

- writes b (lowest 8 bits of b) as a byte to the destination. Note the parameter type is int.

Aside

- System.in, System.out and System.err are byte streams.
- System.in is of type BufferedInputStream (subclass of InputStream).
- System.out and System.err are objects of type PrintStream (subclass of OutputStream) which is deprecated.

Reader class

- Abstract class for input character streams
- Subclasses: BufferedReader, InputStreamReader (extended by FileReader), StringReader
- Important method:

```
public int read() throws IOException
```

- returns a character read, or -1 if the end of stream is reached.

Writer class

- Abstract class for output character streams
- Subclasses: BufferedWriter, OutputStreamWriter (extended by FileWriter), StringWriter.
- Important method:

```
public void write(int c) throws
  IOException
```

- writes a single character (lower 16 bits of c).

```
public void write(String str) throws
  IOException
```

- writes out all characters in the string str.

Reading and writing (and exception handling)

- Since InputStream, OutputStream, Reader and Writer are abstract classes, we do not create instances of those classes when we need a stream.
- We choose a suitable subclass, depending on whether we want to read from a string, from a text file, whether we want a buffered stream, etc.
- read() and write() methods throw a checked IOException, so we either need to throw this exception, too, or catch and handle it.

Example: writing to a file

- This method just throws the same exception as write():

```
public static void StringToFile(String s,
  String fileName) throws IOException {
 FileWriter fw = new FileWriter(fileName);
 fw.write(s);
 fw.close();
}
```

- FileWriter has **`write(String s)`** method which writes out a whole string. We could have written character by character

Example: writing to a file 2

- This method “handles” the exception:

```
public static void StringToFile(String s,
  String fileName) {
 try {
 FileWriter fw = new FileWriter(fileName);
 fw.write(s);
 fw.close();
 }
 catch (IOException e) {
    System.out.println(“IOException!!!”);
 }
}
```


Example:reading from a file

- Character by character method which throws an exception:

```
public static String fileToString(String
  fileName)  throws IOException {
 FileReader fr = new FileReader(fileName);
 String fileContents = new String();
 int c = fr.read();
 while(c !=-1) {
   fileContents = fileContents + (char)c;
   c = fr.read();
 }
 fr.close();
 return fileContents;
}
```

Example:reading from a file 2

- Line by line method which throws an exception:

```
public static String fileToString(String
  fileName)  throws IOException {
 BufferedReader in = new
 BufferedReader(new FileReader(filename));
 String fileContents = new String();
 String s;
 while((s = in.readLine())!= null) {
   fileContents = fileContents + s;
 }
 in.close();
 return fileContents;
}
```


Example:reading from a file 3

- Use of finally to make sure the stream is closed:

```
public static String fileToString(String
  fileName)  throws IOException {
try {
 BufferedReader in = new
 BufferedReader(new FileReader(filename));
 String fileContents = new String();
 String s;
 while((s = in.readLine())!= null)
   fileContents = fileContents + s;
 return fileContents;
} finally {
 if (in!=null) in.close();
}
```

Comment on the last example

- Finally clause is always executed so if an exception was thrown in the try clause, the stream will be closed anyway.
- The method still throws an IOException because close() method called in the finally clause throws IOException.

RandomAccessFile

- If you need a stream where you can both read and write, use random access file.

Constructors:

- **`RandomAccessFile(File file, String mode)`** - creates a random access file stream to read from, and optionally to write to, the file specified by the File argument. Mode is "rw" or "r".
- **`RandomAccessFile(String name, String mode)`** - creates a random access file stream to read from, and optionally to write to, a file with the specified **`name`**.

RandomAccessFile

Methods:

- **`int read()`** (reads a byte)
- **`void write(int b)`** (writes a byte)
- also **`readChar(), writeChar(), readInt(), writeInt(),...`**
- All throw IOException.

File class

- Nothing to do with streams!
- An abstract representation of file and directory pathnames.
- To create a File object, pass it a String pathname (which can be absolute or relative path, e.g. just the name of the file).
- For example,

```
File file1  = new File("Book.java");
File file2  = new
  File("Private/bibtex/Book.java");
```

- Understands path separators on various operating systems.

File class: some methods

Provides methods to work with files, for example

- **`boolean canRead()`** - if the program can read this file
- **`boolean canWrite()`** - if the program can write to this file
- **`boolean exists()`** - if there is such a file
- **`boolean isFile()`**- is it a file
- **`boolean isDirectory()`** - is it a directory
- **`String[] list()`** - lists directory's files and subdirectories (as strings)
- **`long length()`** - length of the file

Parsing

- When you are reading something say from a file, it is nice to be able to split it in meaningful parts/words/tokens and not just read character by character or line by line.
- For example, an obvious thing is to read word by word or sentence by sentence.
- For structured files like programs, it is good to know if you are reading an identifier name, or a reserved keyword, or a method name, and what are the method's arguments.
- In general, parsing involves splitting something in meaningful parts and understanding how they combine.

Tokenising

- Tokenising is a necessary step in parsing: splitting the text you are parsing in meaningful ***tokens***.
- Java has StreamTokenizer class which takes an input stream or a reader and split it into tokens as required (specified by flags). It understands numbers, program identifiers, comments etc.
- We'll be using a simpler class StringTokenizer which tokenizes strings into tokens separated by specified delimiters.

StringTokenizer

Constructors:

- **`public StringTokenizer(String str)`** - constructs a string tokenizer for the specified string. The tokenizer uses the default delimiter set, which is " \t\n\r\f": the space character, the tab character, the newline character, the carriage-return character, and the form-feed character.
- **`public StringTokenizer(String str, String delim)`** - constructs a string tokenizer for the specified string. The characters in the delim argument are the delimiters for separating tokens.
- **`public StringTokenizer(String str, String delim, boolean returnDelims)`**

Examples

- **`StringTokenizer st1 = new StringTokenizer("one, two, three");`**
- **`st1`** will separate **`"one, two, three"`** into **`"one,", "two,",`** and **`"three".`** They are tokens separated by white space.
- **`StringTokenizer st2 = new StringTokenizer("one$two£three", "$£");`**
- **`st2`** will separate **`"one$two£three"`** into **`"one", "two",`** and **`"three".`** They are tokens separated by $ or £.

Examples

- **`StringTokenizer st3 = new StringTokenizer("one$two£three", "$£", true);`**
- **`st3`** will separate **`"one$two£three"`** into **`"one", "$","two", "£"`** and **`"three".`** They are tokens separated by delimiters $ or £, including delimiters themselves.

StringTokenizer methods

- **`public String nextToken()`** - returns next token. Throws NoSuchElementException if there are no more tokens.
- **`int countTokens() -`** how many times can nextToken() method be called before an exception is thrown
- **`public boolean hasMoreTokens()`** - tests if there are more tokens available from this tokenizer's string.

Example

```
StringTokenizer st = new
  StringTokenizer(”one$two£three”, “$£”);
  while (st.hasMoreTokens()) {
     System.out.println(st.nextToken());
  }
```

prints the following output:

```
one
two
three
```

Summary and further reading

- I/O in Java is quite complicated.
- Most introductory textbooks hide this complexity and write their own I/O classes, like the CourseMaker's UserInput class.
- I only skimmed the surface of it. If I have time I'll talk more about saving objects in a file using ObjectStream and serialization.
- For more background, read

http://java.sun.com/docs/books/tutorial/essential/io.