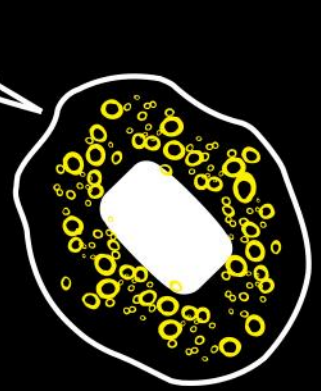
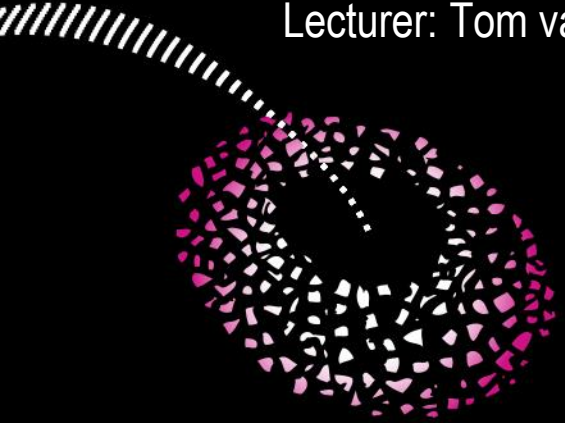
UNIVERSITY OF TWENTE.

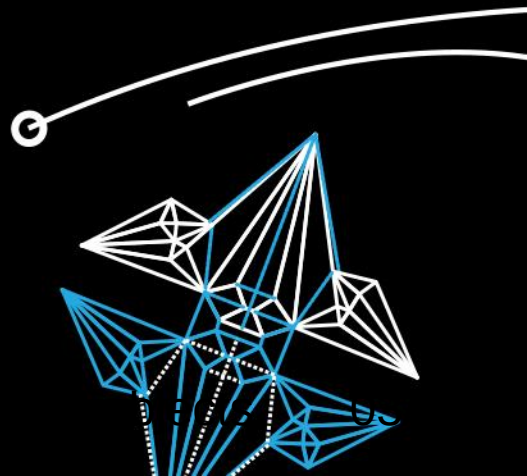


Streams, Readers, Writers

Topic of Software Systems (TCS module 2)

Lecturer: Tom van Dijk





OVERVIEW

- Streams interact with the outside world in the form of binary data (bytes)
  - Files
  - Network / the Internet
  - Other threads
- Readers/Writers wrap streams to convert between bytes and characters
  - They deal with character encodings called charsets

CLASS INPUTSTREAM

```
abstract class InputStream implements Closeable {
    // Returns next byte from stream
    /*@ensures (0 <= result && result <= 255) || result == -1; */
    abstract int read() throws IOException;

    // Closes the stream
    void close() throws IOException;
}
```

- Reads bytes (returns either a byte, or -1)
- Close stream explicitly!
- Errors? `IOException`.

CLASS OUTPUTSTREAM

```
abstract class OutputStream implements Closeable {
    // Write a byte to the stream
    /*@requires 0 <= b && b <= 255 */
    abstract void write(int b) throws IOException;

    // Closes the stream
    void close() throws IOException;

    // Force buffered output bytes to be written out
    void flush() throws IOException;
}
```

FILE STREAMS

- FileInputStream
  - `new FileInputStream(File file)`
  - `new FileInputStream(String name)`

- FileOutputStream
  - `new FileOutputStream(File file)`
  - `new FileOutputStream(File file, boolean append)`
  - `new FileOutputStream(String name)`
  - `new FileOutputStream(String name, boolean append)`

PIPED STREAMS

- PipedInputStream
  - `new PipedInputStream()`
  - `connect(PipedOutputStream source)`
  - `new PipedInputStream(PipedOutputStream source)`

- PipedOutputStream
  - `new PipedOutputStream()`
  - `connect(PipedInputStream sink)`
  - `new PipedOutputStream(PipedInputStream sink)`

BYTE ARRAY STREAMS

- ByteArrayInputStream
  - `new ByteArrayInputStream(byte[] buf)`
- ByteArrayOutputStream
  - `new ByteArrayInputStream()`
  - `byte[] toByteArray();`

STREAM WRAPPERS

Wrappers: constructor is given an existing InputStream

- BufferedInputStream: add mark and reset functionality
- DataInputStream: read primitive types: readInt(), readFloat(), etc.
- ObjectInputStream: like DataInputStream, but also readObject()

STREAM WRAPPERS

Wrappers: constructor is given an existing OutputStream

- BufferedOutputStream: add buffering (for performance)
- DataOutputStream: write primitive types: writeInt(int), writeFloat(float), etc.
- ObjectOutputStream: like DataOutputStream, but also writeObject(obj)
  - **Do not use this**
  - Better is using a format like XML or JSON

READERS AND WRITERS

- Readers and Writers are like Streams, but for characters
- FileReader and FileWriter for files
- PipedReader and PipedWriter
- CharArrayReader and CharArrayWriter and StringReader and StringWriter
- BufferedReader and BufferedWriter
  - BufferedReader can now read a Line.
- **No** DataReader/DataWriter or ObjectReader/ObjectWriter!

READERS AND WRITERS

- InputStreamReader: wrap an InputStream inside a Reader
  - `new InputStreamReader(inputStream);`
- OutputStreamWriter: wrap an OutputStream inside a Writer
  - `new OutputStreamWriter(outputStream);`

Automatically convert bytes/characters using a charset

READERS AND WRITERS

- Special convenience class: PrintWriter
  - `new PrintWriter(File file)` or `new PrintWriter(String fileName)`
  - `new PrintWriter(OutputStream os)` or `new PrintWriter(Writer w)`

- Methods like print(…), println(…), printf(…) to print stuff
- Methods do not throw exceptions; use `checkError()` to check for errors

- `System.out` is a PrintStream
- `System.in` is an InputStream

CLOSING THE STREAM

```
BufferedReader br = null;
try {
    br = new BufferedReader(new FileReader("data.txt"));
    String line;
    while ((line=br.readLine()) != null) {
        System.out.println("Read line: " + line);
    }
} catch (IOException e) {
    e.printStackTrace();
} finally {
    try {
        if (br != null) br.close();
    } catch (IOException e) {
        e.printStackTrace();
    }
}
```

CLOSING THE STREAM

```
try (var reader = new BufferedReader(new FileReader("data.txt"))) {
    String line;
    while ((line=reader.readLine()) != null) {
        System.out.println("Read line: " + line);
    }
} catch (IOException e) {
    e.printStackTrace();
}
```

This syntax is called try-with-resources

- No catch or finally required (but allowed)
- Can open multiple resources (that must implement AutoCloseable)

EXAMPLES

```
String text = "";
boolean b = true;
double salary = 0.0;
DataInputStream datIn = null;

try {
    InputStream in = new FileInputStream("data.dat");
    InputStream bufIn = new BufferedInputStream(in);
    datIn = new DataInputStream(bufIn);
    text = datIn.readUTF();
    b = datIn.readBoolean();
    salary = datIn.readDouble();
} catch (IOException e) {
    e.printStackTrace();
}
finally {
    try {
        datIn.close();
    } catch (IOException e) {
        e.printStackTrace();
    }
}
```

Define variables outside try-catch and initialise them

Define try-catch to handle I/O exceptions

Read data from file

Close the stream in finally to avoid resource leaks!

EXAMPLES

```
String text = "";
boolean b = true;
double salary = 0.0;

try (DataInputStream datIn = new DataInputStream(
        new BufferedInputStream(new FileInputStream("data.dat")))) {
    text = datIn.readUTF();
    b = datIn.readBoolean();
    salary = datIn.readDouble();
} catch (IOException e) {
    e.printStackTrace();
}
```

EXAMPLES

```
String text = "";
boolean b = true;
double salary = 0.0;

    try (BufferedReader bufIn = new BufferedReader(
            new FileReader("data.txt"))) {
        text = bufIn.readLine();
        b  = Boolean.parseBoolean(bufIn.readLine());
        salary = Double.parseDouble(bufIn.readLine());
    } catch (IOException e) {
        e.printStackTrace();
    }
```

SCANNER

- Helpful class for parsing primitive types: the Scanner

- Wraps around an InputStream or a Reader
  - `new Scanner(InputStream is)` and `new Scanner(Reader r)`
  - `new Scanner(File f)`
  - `new Scanner(String s)`

- Parses tokens separated by delimiters
- Methods like `boolean hasNextInt()`, `int nextInt()`, etc.

SCANNER EXAMPLE

```
String text = "";
boolean b = true;
double salary = 0.0;

try (Scanner scanner = new Scanner(new File("data.txt"))) {
    text = scanner.nextLine();
    b = scanner.nextBoolean();
    salary = scanner.nextDouble();
} catch (FileNotFoundException e) {
    e.printStackTrace();
} catch (NoSuchElementException e) {
    e.printStackTrace();
}
```

STREAM VS SCANNER

- Using a DataOutputStream or ObjectOutputStream
  - Data is stored in a compact binary format
  - Unreadable in a text editor

- Using a PrintWriter (and reading with a Scanner)
  - Data is stored in a readable text format
  - Can be changed in a text editor

PIPED STREAMS EXAMPLE

```
try {
    var pipeIn = new PipedInputStream();
    var pipeOut = new PipedOutputStream();
    pipeIn.connect(pipeOut);

    {
        var pw = new PrintWriter(pipeOut);
        pw.println("This is a test");
        pw.close();
    }

    {
        try (var br = new BufferedReader(new InputStreamReader(pipeIn))) {
            System.out.println(br.readLine());
        } catch (IOException e) {
            e.printStackTrace();
        }
    }
} catch (IOException e) {
    e.printStackTrace();
}
```

SUMMARY

- InputStreams and OutputStreams read/write bytes (binary)
- Readers and Writers read/write characters
- Files: FileInputStream/FileReader and FileOutputStream/FileWriter
- PrintWriter and Scanner for easy writing/reading
- Always close streams, use try-with-resources