Java training

I/O streams

Session overview

- **I/O streams** - what they are, when are they useful
- **Hands-on** - using I/O streams

Input / output stream

- Stream = a sequence of data
- I/O stream - an **input source** / an **output destination**
- Use-cases:
  - **Input** stream - **read data from a source**, one item at a time
  - **Output** stream - **write data to a destination**, one item at time
- Can be used as many kinds of sources and destinations:
  - Disk files
  - Devices
  - Memory arrays

Byte streams

- Use - perform I/O of 8-bit bytes
- All byte stream classes extend **`InputStream`** or **`OutputStream`**
- Examples:
  - Input
    - `FileInputStream`
    - `ByteArrayInputStream`
    - `BufferedInputStream`
  - Output
    - `FileOutputStream`
    - `ByteArrayOutputStream`
    - `BufferedOutputStream`
- ! Streams need to be closed after usage

Character streams

- Extend the **`Reader`** and **`Writer`** abstract classes
- File specialized I/O - **`FileReader`** and **`FileWriter`**
- Byte-to-character "bridge": **`InputStreamReader`** and **`OutputStreamWriter`**
- Printing output streams - **`PrintWriter`**
  - Used internally in the Servlet API and other web frameworks

Hands-on:

- Reading and displaying the content of a file
- Close the stream:
  - `try / catch / finally`
  - `try with resources` - automatic

Buffered streams

- **Buffered I/O streams** - reduce the overhead on the OS resources, by buffering the streams in memory
- Most used:
  - `BufferedInputStream` & `BufferedOutputStream` - read / write buffered *byte* streams
  - `BufferedReader` & `BufferedWriter` - read / write buffered *character* streams

Standard (system) streams

- A feature of most (/ all?) operating systems (OS's)
  - Default operation - read from keyboard (standard input) & write to the display (standard output + standard error output)
- Java - **three default streams**, accessed statically from the **`System`** class:
  - `in` - represents the 'standard input'
  - `out` - represents the 'standard output'
  - `err` - represents the 'standard error output'
- Implemented as byte streams, not as character streams
- Defined / enabled automatically; no need to be opened / closed

`Console` & `Scanner` classes

- Console class - used to read input from the keyboard
  - Returned via the `System.console()` static method
  - Must be verified if it's available - not available if the app is launched in non-interactive mode
  - Can read plain text and sensitive data (passwords)
- Simpler alternative - the `Scanner` class:

```
Scanner scanner = new Scanner(System.in);
String enteredLine = scanner.nextLine();
```

Hands-on

Reading and displaying input from the keyboard

- Using the ‘old’ `BufferedReader` way
- Using the `Console` class
  - Reading non-sensitive data
  - Reading sensitive data (passwords)

Advice - don’t remember methods usage, remember classes and domains

Exercise 1

- Create a CSV or text file with a few products, one on each line
- Read the CSV or text file from the disk, create model objects from it
  - Split each line, parse the values and create the needed objects
- After the products objects are created - display them, iteratively
- Save the products which have a certain property (choose one) in another file
- Load and display the file

Use a service class for implementing the logic, call it from a main method

Exercise 1 [extended]

- Create two CSV files with:
  - A few products, one on each line
  - A few sections, one on each line
- Read the CSV files from the disk, create model objects from them
  - Split each line, parse the values and create the needed objects
- After the products and section Java objects are created - display them, iteratively
- Save the sections which have products costing more than 100 EUR in a file
- Save the products which have a certain property (choose one) in another file
- Load and display the saved files

Use a service class for implementing the logic, call it from a main method

Exercise 2

- Write a program which reads the details of a few products from the standard input, using the `Scanner` class
- After saving the products internally (in a (`Set` | `List` | `Map`)), save them in a file (.csv or .txt format; as you wish)
- Copy that file in a new file
- Archive (Gzip) the second file (the extension is .gz)
- Use a service class for implementing the logic, call it from a main method