Technische Hochschule Rosenheim
Technical University of Applied Sciences

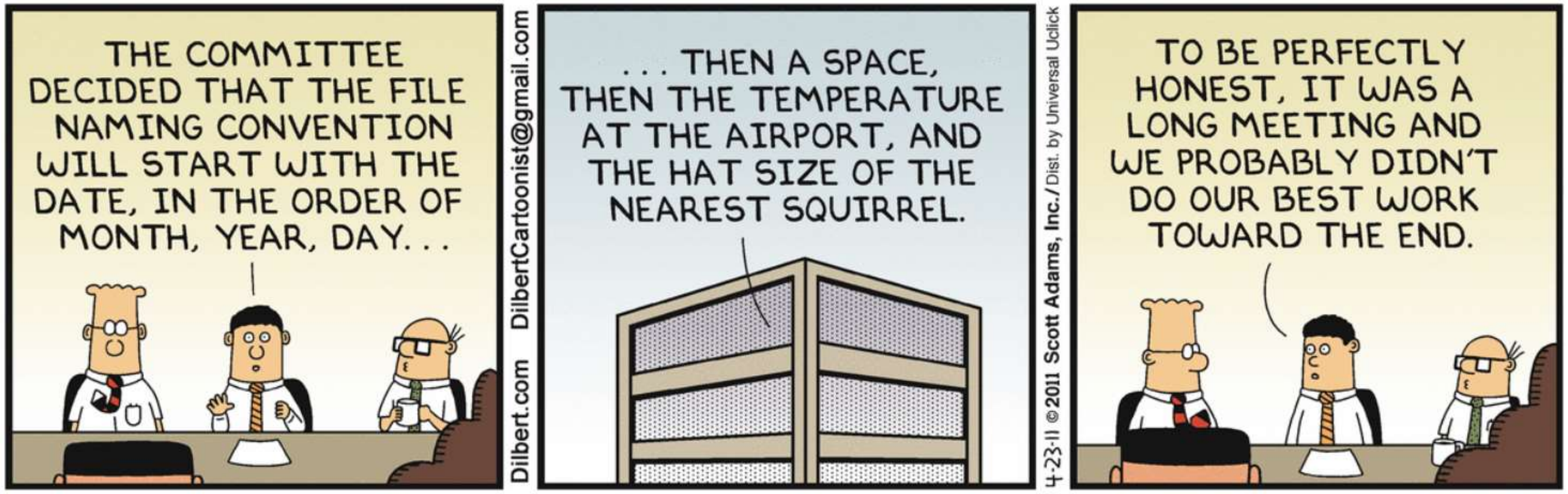
Programming Basics – WiSe21/22

I/O: Files and Data Streams

Prof. Dr. Silke Lechner-Greite

Table of contents – planned topics

Files


THE COMMITTEE DECIDED THAT THE FILE NAMING CONVENTION WILL START WITH THE DATE, IN THE ORDER OF MONTH, YEAR, DAY. . .
Dilbert.com DilbertCartoonist@gmail.com
. . . THEN A SPACE, THEN THE TEMPERATURE AT THE AIRPORT, AND THE HAT SIZE OF THE NEAREST SQUIRREL.
4-23-11 ©2011 Scott Adams, Inc./Dist. by Universal Uclick
TO BE PERFECTLY HONEST, IT WAS A LONG MEETING AND WE PROBABLY DIDN'T DO OUR BEST WORK TOWARD THE END.

Source: [2]

- How do you store data persistently?
  - **Files**
  - Databases
- How do you read or write in files?

Core classes: file input and output

- `java.io` (since Java 1.0)
  - Still widely used
  - `File` class used for everything!
    - Create, delete, copy files; structure in directories

- `java.nio.file` (since Java 7)
  - Aim: stronger decoupling of platform, abstraction of file system
  - New classes replace `File` completely.
  - Core classes
    - `FileSystem`: abstract top-level class, uniform interface independent of implementation (hard disk, RAM, floppy disk, …).
    - `FileSystems`: factory methods to get to a specific `FileSystem`.
    - Represents file/directory path, path-related methods.
    - Generated from string or URI path object.
    - Create, delete, copy, manipulate files

`java.nio`: example

- **`Path`** object
  - Represents a path.

- **`Files`**: manipulation of files and directories

```
// generate absolute Path object using a relative path
Path filePath = Paths.get("lecture10/src/test.txt").toAbsolutePath();
System.out.println(filePath.toString());
// print root directory
System.out.println("Root directory is: " + filePath.getRoot().toString());
// size of file
try {
   // size of file
   long fileSize = Files.size(filePath);
   // copy a file
   Files.copy(filePath, Paths.get("vorlesung10/src/test-kopie.txt"),
         StandardCopyOption.REPLACE_EXISTING);
   // create a new directory
   Files.createDirectory(Paths.get("vorlesung10/src/newDir"));
} catch (IOException e) {
   e.printStackTrace();
}
```

Overview: access to file content

- ***Random vs. sequential access***
  - *Random access*: jump back and forth as desired within a file.
  - : data is read or written in a fixed order.
- In Java, file content is usually accessed via ***streams***!
  - Definition: continuous sequences of data of one type.
  - Areas of application: files, sockets, ….
  - ***Files*** class has methods for creating streams
    - `Files.newXXXStream()`
- ***Granularity of access for streams***
  - *Byte-based:* each read/write accesses the next byte in the stream.
  - *Character-based:* each read/write accesses the next character in the stream.
    - Some characters have different byte lengths.
    - Example: UTF-8 encoding: http://www.utf8-zeichentabelle.de/
  - [line by line]

Streams / data streams

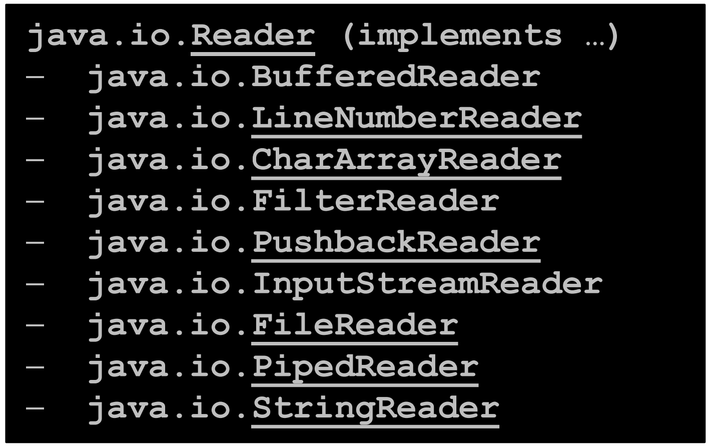
- **Abstraction of a stream** (= typical methods)
  - **`open()`** : Open stream from/to file, device, etc.
  - **`read()`** : Read the next item from stream
  - **`write()`** : Write the next item in stream
  - **`close()`** : Close stream

- There are many different streams for all kinds of different tasks
  - Inheritance hierarchy!

- **Abstract base classes of the streams**
  - **Character streams:** Usually UTF-8 encoded
    - Input: class
    - Output: class
  - **Byte streams:** 8 bits
    - Input: class **`InputStream`**
    - Output: class **`OutputStream`**

Character streams: input

- **Basis:** abstract class **`java.io.Reader`**
  - Parameterless constructor
  - Series of `read()` methods
  - `ready(), close(), mark(), reset(), skip()`
- Classes derived from `Reader` (selection of input device)
  - `InputStreamReader, FileReader, StringReader, CharArrayReader`
- Classes derived from `Reader` (nesting of input streams)
  - `FilterReader, PushbackReader, BufferedReader, LineNumberReader, PipedReader`



```
java.io.Reader (implements …)
–   java.io.BufferedReader
–   java.io.LineNumberReader
–   java.io.CharArrayReader
–   java.io.FilterReader
–   java.io.PushbackReader
–   java.io.InputStreamReader
–   java.io.FileReader
–   java.io.PipedReader
–   java.io.StringReader
```

Example: character-based reading

- **`BufferedReader`** is a subclass of Reader
  - Additional methods, e.g. **`readLine()`**
  - Temporary storage of data in internal buffers
  - Efficient reading of files!

```
// generate absolute Path object using a relative path
Path filePath = Paths.get("vorlesung10/src/test.txt").toAbsolutePath();
try {
   BufferedReader bf = Files.newBufferedReader(filePath);
   String line;
   while ((line = bf.readLine()) != null) {
      System.out.println(line);
   }
}catch (IOException e) {
   e.printStackTrace();
}
```

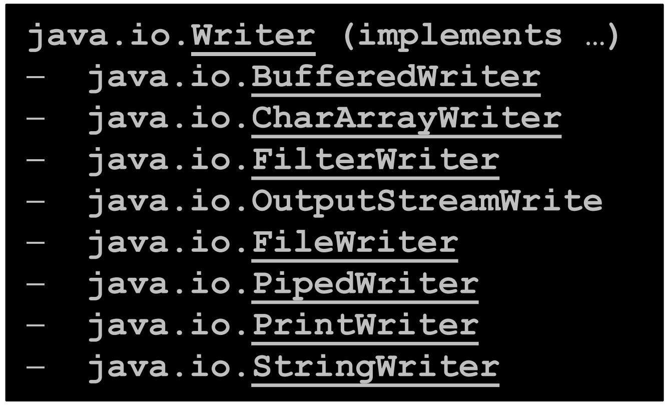
Variant 1:
Create
a
buffered
file stream
using
java.nio

Character streams: output

- Basis: abstract class **`java.io.Writer`**
  - Parameterless constructor (open the output stream and prepare for subsequent `write` call)
  - `close()`: Close the output stream
  - `flush()`: Empty the buffers & pass on the data within to the output device
  - Multiple `write` methods

- Classes derived from `Writer` (selection of output device)
  - `OutputStreamWriter, FileWriter, StringWriter, CharArrayWriter, PipedWriter`

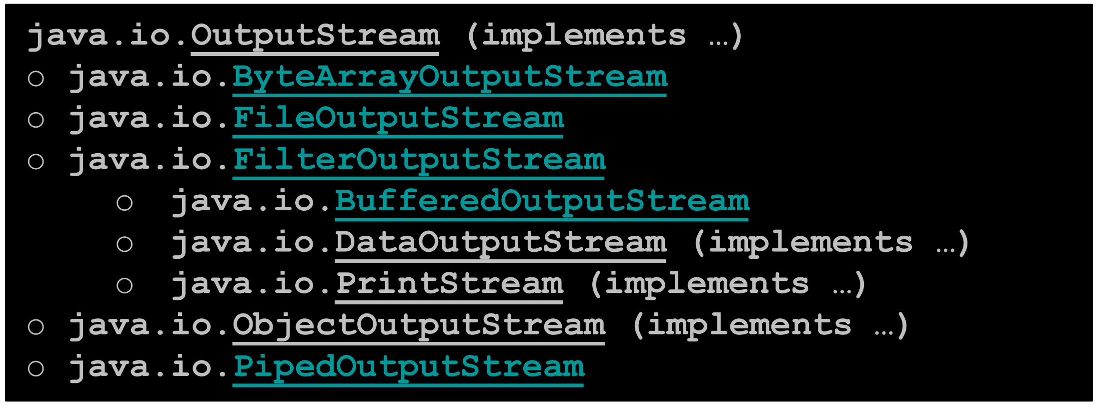
- Classes derived from `Writer` (nesting of output streams)
  - `BufferedWriter, PrintWriter, FilterWriter`



```
java.io.Writer (implements …)
–   java.io.BufferedWriter
–   java.io.CharArrayWriter
–   java.io.FilterWriter
–   java.io.OutputStreamWrite
–   java.io.FileWriter
–   java.io.PipedWriter
–   java.io.PrintWriter
–   java.io.StringWriter
```

Byte streams: example output

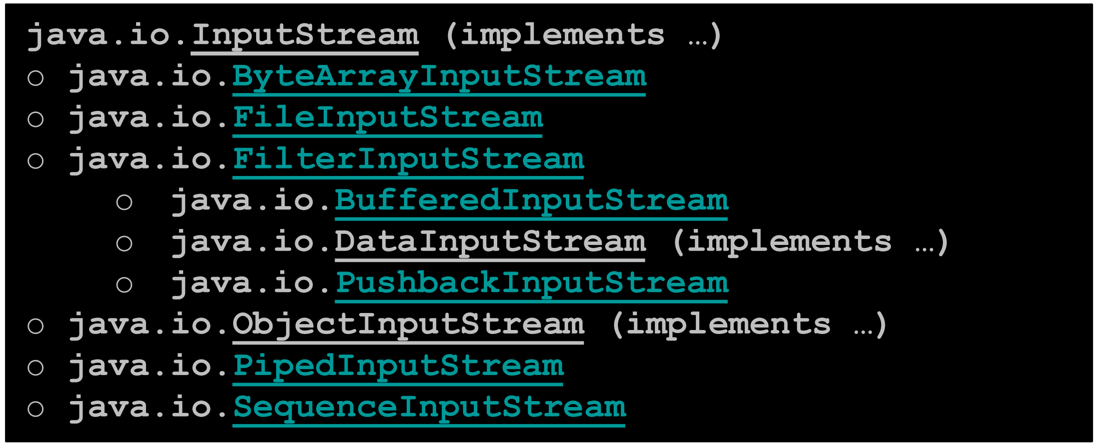
- Basis: abstract class **`java.io.OutputStream`**
  - Parameterless constructor
  - `close(), flush()`
  - `write` methods



```
java.io.OutputStream (implements …)
○ java.io.ByteArrayOutputStream
○ java.io.FileOutputStream
○ java.io.FilterOutputStream
    ○ java.io.BufferedOutputStream
    ○ java.io.DataOutputStream (implements …)
    ○ java.io.PrintStream (implements …)
○ java.io.ObjectOutputStream (implements …)
○ java.io.PipedOutputStream
```

Byte streams: input

- Basis: abstract class **`java.io.InputStream`**
  - Parameterless constructor
  - `skip(), available(), close(), markSupported(), mark(), reset()`
  - `read` methods



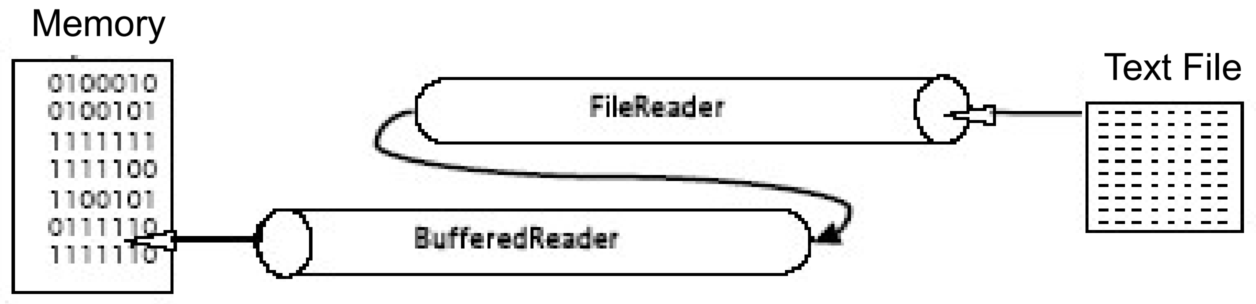
```
java.io.InputStream (implements …)
○ java.io.ByteArrayInputStream
○ java.io.FileInputStream
○ java.io.FilterInputStream
     ○ java.io.BufferedInputStream
     ○ java.io.DataInputStream (implements …)
     ○ java.io.PushbackInputStream
○ java.io.ObjectInputStream (implements …)
○ java.io.PipedInputStream
○ java.io.SequenceInputStream
```

Chaining (concatenation) of streams

- Streams can be chained to each other!
  - Creation of the 1st stream as normal through a constructor.
  - Handover in constructor!

- **Example 1:** concatenation of `FileReader` and `BufferedReader`
  - 1st stream: reads character-based data from file.
  - 2nd stream: buffers data before it is passed on.

```
FileReader fReader = new FileReader(fileName);
BufferedReader bReader = new BufferedReader(fReader);
```



Variant 2:
Create
a
buffered
file stream
using chaining
(concatenation)
of streams,
see slide 7

Chaining (concatenation) of streams

- **Applications**
  - Saving Java objects in a file or database
  - Transferring Java objects over a network
  -
    - Conversion of an object into a byte stream.
    - Standard serialisation of Java, but rarely used.

- **Outlook: prerequisite for serialisation**
  - Object must be serialisable.
  - The object must implement the `Serializable` interface
    - Marker interface, does not prescribe any methods!
  - All attributes must be serialisable, otherwise overwrite the following method of the attribute's class:
    - `writeObject(java.io.ObjectOutputStream)`
    - `readObject(java.io.ObjectInputStream)`

Outlook: Java serialisation

```java
public class Account implements Serializable {
   // ...
}

String pathName = "test.out";
Account salaryAccount = new Account(1000); // ...

// Write object to file
ObjectOutputStream oos = new ObjectOutputStream(new FileOutputStream(pathName));
oos.writeObject(salaryAccount);
// ...

// Read object from file
ObjectInputStream ois = new ObjectInputStream(new FileInputStream(pathName));
Account sa = (Account) ois.readObject();
// ...
```

Chaining :

concatenation

Rarely used in practice!