

Object Oriented Programming 2

Topic 2 – Input / Output

Prof. Annett Laube

(adapted by Andres Scheidegger, FS 2018)

Motivation

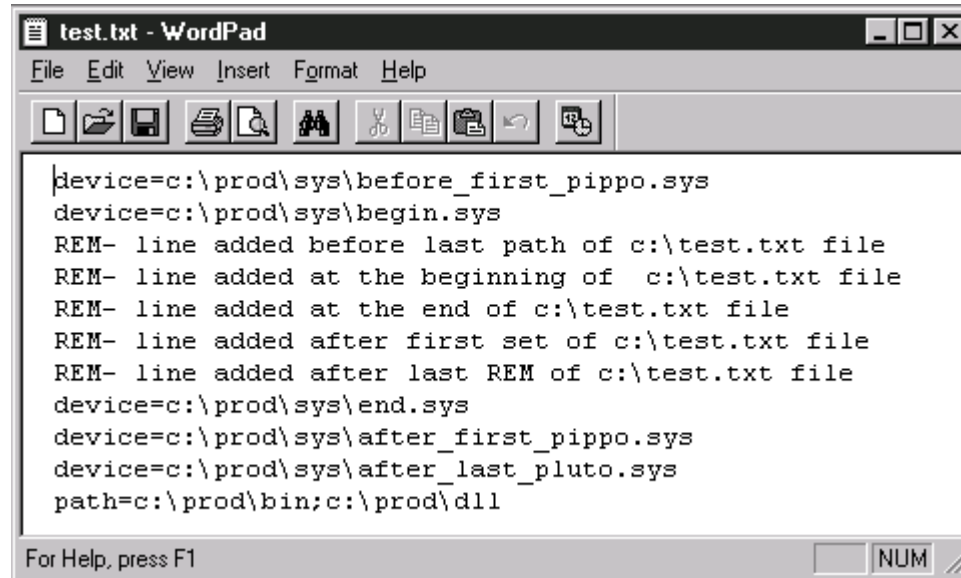
- ▶ Saving, transferring and reconstructing state information of a Java application.
 - ▶ E.g. session state information of web browser, i.e. list of currently displayed URLs.



Outline

- ▶ Text Files
- ▶ Binary Files
- ▶ Random File Access
- ▶ Serialization

Text Files

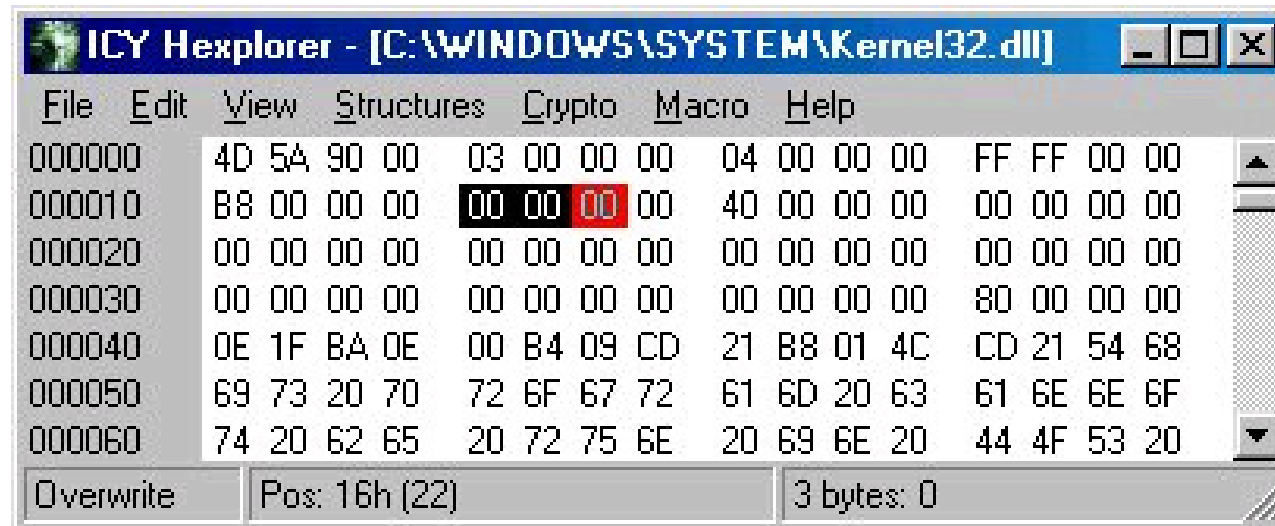


```
test.txt - WordPad
File Edit View Insert Format Help
[Icons]
device=c:\prod\sys\before_first_pippo.sys
device=c:\prod\sys\begin.sys
REM- line added before last path of c:\test.txt file
REM- line added at the beginning of c:\test.txt file
REM- line added at the end of c:\test.txt file
REM- line added after first set of c:\test.txt file
REM- line added after last REM of c:\test.txt file
device=c:\prod\sys\end.sys
device=c:\prod\sys\after_first_pippo.sys
device=c:\prod\sys\after_last_pluto.sys
path=c:\prod\bin;c:\prod\dll
For Help, press F1 NUM
```

(Plain) Text files contain a sequence of ordinary textual characters.

- Human readable
- Examples: txt , xml, csv, log,...

Binary Files



A **binary file** is a sequence of bytes that does not contain merely plain text.

- Dump utility required to read
- Examples: applications (exe, bin, dll), images, doc, pdf, ppt, ...

Outline

- ▶ Text Files
- ▶ Binary Files
- ▶ Random File Access
- ▶ Serialization

Processing I/O with Text Files

- ▶ Text file contains a sequence of characters
 - ▶ Integer 12345 stored as characters '1' '2' '3' '4' '5'
- ▶ Use **Reader** and **Writer** and their subclasses to process I/O with text files.
- ▶ To read:
`FileReader reader = new FileReader("input.txt");`
- ▶ To write:
`PrintWriter out =
 new PrintWriter("output.txt");`

Path names

- ▶ A **file name** can contain the **path** and a **drive**.
- ▶ A path instance reflects the underlying platform:
 - ▶ Linux/Mac: /home/joe/foo
 - ▶ Windows: C:\home\joe\foo

- ▶ Representation as a **String**

```
FileReader in = new  
    FileReader("c:\\homework\\input.dat");
```

- ▶ **Recall**: a single backslash inside quoted strings is an escape character that is combined with another character to form a special meaning

Path interface in Java 7/8 (1/2)



- ▶ NIO (New Input Output) API
- ▶ A *Path* is a sequence of directory names, optionally followed by a file name.
 - ▶ First component may be a *root component*, e.g. */* or *C:*
 - ▶ A path starting with a root component is *absolute*, otherwise *relative*.

```
Path absolute = Paths.get("C:", "local_data", "Prog2_FS15");  
Path relative = Paths.get("temp", "conf", "user.properties");
```

- ▶ The strings are joined with the separator of the file system (*/* for Unix/Mac, ** for Windows)

Path interface in Java 7/8 (2/2)



▶ *Paths.get()* method

- ▶ receives one or more strings
- ▶ Joins the strings with the separator of the file system (/ for unix/mac, \ for Windows)
- ▶ Validates if the paths exists and throws an **InvalidPathException**, if not

```
Path absolute = Paths.get("C:", "local_data", "Prog2_FS15");  
Path relative = Paths.get("temp", "conf", "user.properties");  
Path homedir = Paths.get("/home/luca1");
```

Reading Text Files

- ▶ Simplest way: `Scanner` class
- ▶ To read from a file, construct a `FileReader`
- ▶ Use the `FileReader` to construct a `Scanner` object

```
FileReader reader = new FileReader("input.txt");  
Scanner in = new Scanner(reader);
```

- ▶ Use the `Scanner` methods to read data from file
 - ▶ `nextLine`, `nextInt`, `nextDouble`, `next`

Writing Text Files

- ▶ To write to a file, construct a `PrintWriter` object

```
PrintWriter out = new PrintWriter("output.txt");
```

- ▶ If the file exists, it is overwritten.
- ▶ If file doesn't exist, an empty file is created.
- ▶ Use `print` and `println` to write into a `PrintWriter`:

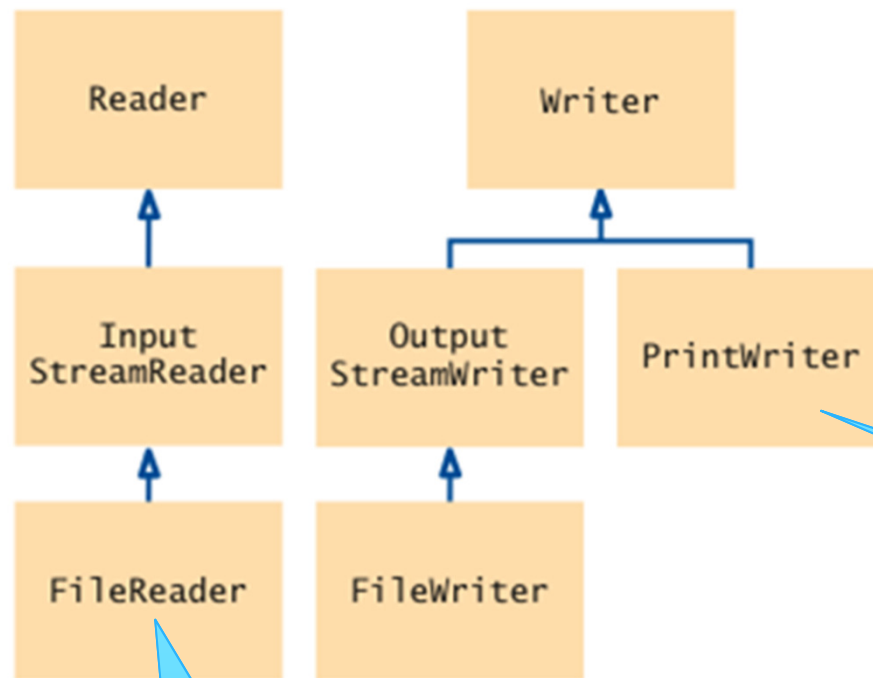
```
out.println(29.95);  
out.println(new Rectangle(5, 10, 15, 25));  
out.println("Hello, World!");
```

- ▶ Close the file after processing.

```
out.close();
```

Otherwise, output may not be completely written to the file.

Reading/writing text files



Readers and writers handle data in text form

Can be used together with **Scanner**

Works like **System.out**

NIO Text Files



- ▶ Read a file as a sequence of lines

```
List<String> lines = Files.readAllLines(path) ;
```

- ▶ Write a collection of lines into a file

```
Files.write(path, lines) ;
```

- ▶ 2 lines replace a lot of code !!!

Reading a Single Character

- ▶ Use `read` method of `Reader` class to read a single character
 - ▶ returns the next character as an integer or the integer -1 at end of file.

- ▶ Example

```
Reader reader = ...;  
int next = reader.read();  
char c;  
if (next != -1)  
    c = (char) next;
```

- ▶ Hint: class `FileReader` is a subclass of `Reader`

Processing Text Input

1. String Processing

- ▶ Use the methods of the `scanner` class
 - ▶ `nextLine()`, `nextInt()`, `nextDouble()`,...
- ▶ Use methods `Character.isDigit()` and `Character.isWhiteSpace()` to further process strings

2. Regular Expressions

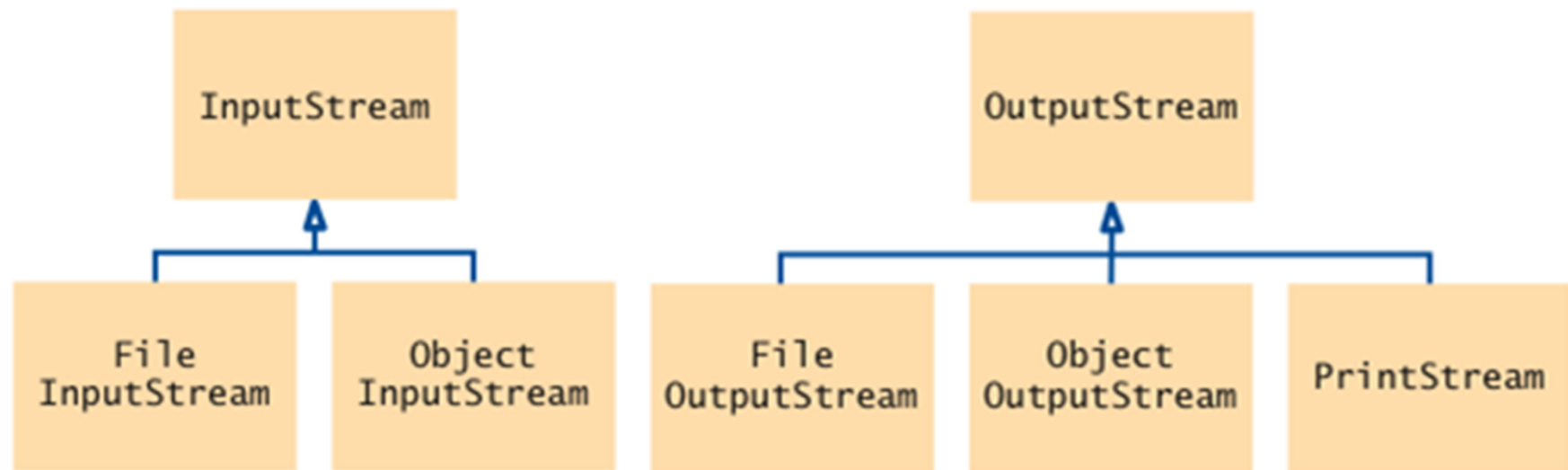
- ▶ Define a delimiter ("`\t`") and split the string there
`String[] items = line.split("\t");`

Outline

- ▶ Text Files
- ▶ Binary Files
- ▶ Random File Access
- ▶ Serialization

Processing I/O with Binary Files

- ▶ data is stored byte by byte
- ▶ Ex: Integer 12'345 stored as a byte sequence: 00 00 48 57
- ▶ Use `InputStream` and `OutputStream` and subclasses



Processing I/O with Binary Files

- ▶ To read:

```
InputStream inputStream =  
    new FileInputStream("input.bin");
```

- ▶ To write:

```
OutputStream outputStream =  
    new FileOutputStream("output.bin");
```

Reading a Single Byte

- ▶ Use `read` method of `InputStream` class to read a single byte:
 - ▶ Returns the next byte as int or the integer -1 at end of file
- ▶ Example

```
InputStream in = ...;  
int next = in.read();  
byte b;  
if (next != -1) b = (byte) next;
```

NIO Binary Files



- ▶ To read:

```
InputStream inputStream =  
    Files.newInputStream(Paths.get(inFile));
```

- ▶ To write:

```
OutputStream outputStream =  
    Files.newOutputStream(Paths.get(outFile));
```

- ▶ To read/write a whole file:

```
byte[] bytes = Files.readAllBytes(path);  
Files.write(path, bytes);
```

Outline

- ▶ Text Files
- ▶ Binary Files
- ▶ Random File Access (raf)
- ▶ Serialization

Random File Access

- **Sequential file access**
 - A file is processed line by line, byte by byte, etc.
 - Sequential processing can be very inefficient.
 - E.g. retrieving a record in a database.
- **Random file access (raf)**
 - Allows access at arbitrary locations in the file.
 - Files support random access
 - `System.in` and `System.out` do not
 - Each file has a special **file pointer position**
 - You can read or write at the position where the pointer is.

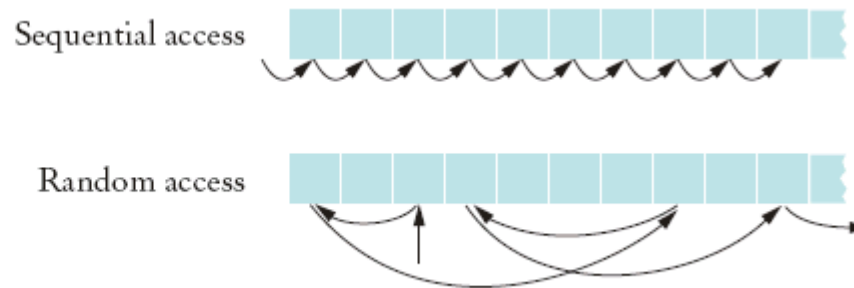



Figure 4
Sequential and Random Access

Random File Access - Records

- ▶ Structured data is written in **records**
- ▶ A **record** is a collection of *elements*, possibly of different data types, typically in fixed number , length and sequence.
- ▶ Used in databases: each line in a table is a record.

Random File Access

Java	< 7	7/8 
Interface	-	SeekableByteChannel
Class	RandomAccessFile	FileChannel
Position the pointer	<code>seek(long n)</code>	<code>position(long n)</code>
File pointer current position	<code>getFilePointer()</code>	<code>position()</code>
Get current file size (in byte)	<code>length()</code>	<code>size()</code>
read	<code>read...()</code> for all datatypes	<code>read(ByteBuffer b)</code>
write	<code>write...(...)</code> for all datatypes	<code>write(ByteBuffer b)</code>
close	<code>close()</code>	<code>close()</code>

Random File Access

- ▶ You can open a file either for

- ▶ Reading only ("r")
- ▶ Reading and writing ("rw")

```
RandomAccessFile f =  
    new RandomAccessFile("bank.dat", "rw");
```

- ▶ To move the file pointer to a specific byte

```
f.seek(n);
```

- ▶ To get the current position of the file pointer.

```
long n = f.getFilePointer();  
// of type "long" because files can be very large
```

- ▶ To find the number of bytes in a file

```
long fileLength = f.length();
```

Outline

- ▶ Text Files
- ▶ Binary Files
- ▶ Random File Access
- ▶ **Serialization**

Serialization

- ▶ Standard method to read and write objects, e.g. to a file
- ▶ Use Streams:
 - ▶ `ObjectOutputStream` to write
 - ▶ `ObjectInputStream` to read
- ▶ Objects that are written to an object stream must belong to a class that implements the `Serializable` interface.
 - ▶ `Serializable` interface has no methods.

Serialization

- ▶ Serializable classes may declare a `serialVersionUID` of type `long`
 - ▶ Verify compatibility of object and class upon deserialization
 - ▶ If missing in class, inserted automatically by java runtime
- ▶ Fields may be marked `transient` to be ignored by serialization
- ▶ `static` fields are not serialized
- ▶ Referenced objects are serialized too!
- ▶ Identical objects are serialized just once ➔ object sharing is preserved!

Example I

- ▶ Writing a `BankAccount` object to a file

```
class BankAccount implements Serializable {  
    ...  
}
```

```
BankAccount b = ...;
```

```
ObjectOutputStream out =  
    new ObjectOutputStream  
        (new FileOutputStream("bank.dat"));
```

```
out.writeObject(b);
```

- ▶ The object output stream takes care of saving the bank account including its balance

Example I

- ▶ Reading a `BankAccount` object from a file.

```
ObjectInputStream in =  
    new ObjectInputStream  
        (new FileInputStream("bank.dat"));
```

```
BankAccount b = (BankAccount) in.readObject();
```

- ▶ The `readObject` method
 - ▶ Returns an `object reference`; this reference needs to be `casted` to the original type
 - ▶ Can throw `ClassNotFoundException`
 - ▶ checked exception → catch or declare.

Example II

- ▶ Read and write an `arrayList` of bank accounts to a file.

- ▶ Write

```
ArrayList<BankAccount> a =  
    new ArrayList<BankAccount>();  
    // Now add BankAccount objects into this arrayList  
    out.writeObject(a);
```

- ▶ Read

```
ArrayList<BankAccount> a =  
    (ArrayList<BankAccount>) in.readObject();
```

- ▶ The object output stream takes care of saving the array list including **all** bank accounts and their balances.

Deep cloning using serialization

The steps for making a deep copy using serialization are:

1. Ensure that all classes in the object's graph are serializable.
2. Create input and output streams.
3. Use the input and output streams to create object input and object output streams.
4. Pass the object that you want to copy to the object output stream.
5. Read the new object from the object input stream and cast it back to the class of the object you sent.

Deep cloning using serialization

- ▶ However: Runtime speed comparison (Milliseconds to deep copy a simple class graph n times)

Iterations	1000	10000	100000
clone()	10	101	791
Serialization	1832	11346	107725

Finally a warning

- ▶ Do not use Java serialization if you have alternatives!
 - ▶ Use JSON, Protocol Buffer or XML instead
- ▶ Deserialization of untrusted streams is a big security risk!
- ▶ Read items 86 to 90 in *Effective Java, Joshua Bloch, Addison-Wesley*

