ENTERPRISE JAVA PROGRAMMING / PROGRAMMING II

MODULES CT545 / CT875, SEMESTER 2, ACADEMIC YEAR 2020-2021
NATIONAL UNIVERSITY OF IRELAND, GALWAY

Lecture 4b

Lecturer: Dr Patrick Mannion

TOPICS

- In the previous lecture, we've already introduced a few basic I/O *stream* operations, applied to networking
- But as you know, streams are important in other contexts too, e.g., for file operations
- Today, we look at I/O streams in more detail (including some revision)

INPUT/OUTPUT

- Input/Output (I/O) in Java means exchanging data with, e.g.,
  - a device such as a printer
  - the file system
  - some other program on the same computer
  - some other program via a network (see previous lecture)
  - a different component of the same program
- Most important packages for I/O:
  - `java.io`
  - `java.nio` (for high-speed but low-level I/O)

INPUT/OUTPUT

Streams

Revision:

- In Java, I/O operations are mainly based on the notion of *streams*.
- A stream is a *sequence of data which is made available over time*.
- Intuitively, a stream "flows" from a *source* to a *destination* (also called *sink*).
- From the viewpoint of a certain program, a stream either "flows" to its destination (*output stream*) or arrives from some source (*input stream*).

INPUT/OUTPUT
Streams

- The items in a stream can be
  - *incrementally* computed by the source (even successively *on demand*)
  - *incrementally* requested by the destination
  - ⇒ the data is not necessarily fully available when being used by the stream destination
  - ⇒ the stream length (amount of available data) might not be fixed
  - ⇒ a stream might theoretically be *infinitely* long
  - ⇒ streams are produced and consumed piecewise
  - ⇒ observe the similarity to iterators operating on collections

INPUT/OUTPUT

Streams

- Streams are natural models of many kinds of data sequences
  - Mouse or keyboard input
  - Data transfer via a network (see previous lecture)
  - Reading contents of a file

INPUT/OUTPUT
Streams

- I/O streams and concurrency/multithreading:
  - If two or more streams access the same resource concurrently (e.g., simultaneous access of the same file by two different programs or two different threads), this resource becomes a *shared resource* in the sense of multithreading
  - In that case, programs which use streams need to take care of avoiding race conditions

INPUT/OUTPUT

Streams

- Streams and concurrency/multithreading (continued):
  - Stream operations might take a relatively long time (e.g., reading the content of a large file or reading data from a remote host via the internet).

    Such operations should be performed using threads, so that the rest of the program can continue while the time-consuming stream operation runs in parallel in another thread.

INPUT/OUTPUT

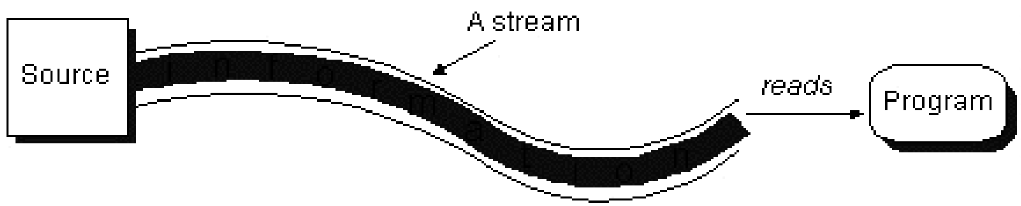
Streams

- In Java, I/O streams are represented by *stream objects* which provide methods for reading/writing data from/to the respective stream
- *Input streams* model incoming streams (for reading data arriving from some data source, such as a file on disk or a network socket)
  - `java.io.InputStream` (super class for byte input streams)
  - `java.io.Reader` (super class for character input streams)
- *Output streams* model outgoing streams (for writing data to some destination)
  - `java.io.OutputStream` (super class for byte output streams)
  - `java.io.Writer` (super class for character output streams)

INPUT/OUTPUT

Streams

- Typical lifecycle of a stream



<u>Reading from a stream</u>

Open the input stream

While more data available

read data

Close the Stream

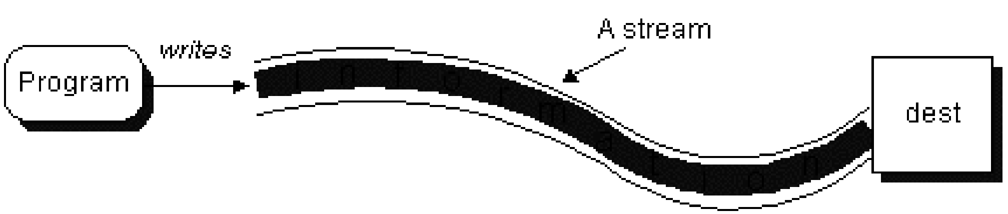
<u>Writing to a stream</u>

Open the output stream

While more data

write the data

Close the Stream



INPUT/OUTPUT

Streams

- To write or read from a stream, you first create an object of one of the stream classes (e.g., `FileOutputStream`).
- Then, you call methods on this object in order to write/read to/from this stream
- The stream objects are sometimes simply called "streams“. They provide 1) operations on streams and 2) internally, some recent chunk of data of the entire stream (e.g., what has recently been written into the stream)

INPUT/OUTPUT

Streams

- Remark (1)
  In the previous lecture, we have seen how stream objects can be used with network sockets. In this lecture, we provide other examples for stream use, such as writing to a file.

  However, most of the following stream types could also be used to transfer data between sockets, analogously to the streams shown in the previous lecture!

- Remark (2)
  Don't confuse I/O streams (today) with so-called *Java 8 Streams* (different concept that relates to Collections -> future lecture)

INPUT/OUTPUT

Streams

- *Byte streams* are the most basic kinds of I/O streams
- https://docs.oracle.com/javase/tutorial/essential/io/bytestreams.html
- They are low-level streams for the input/output of raw binary data in form of sequences of single bytes
- Examples:
  - `java.io.FileOutputStream`
  - `Java.io.FileInputStream`
  - `java.io.ByteArrayOutputStream`
  - `java.io.ByteArrayInputStream`
- Use cases, e.g., writing/reading images. Copying arbitrary files

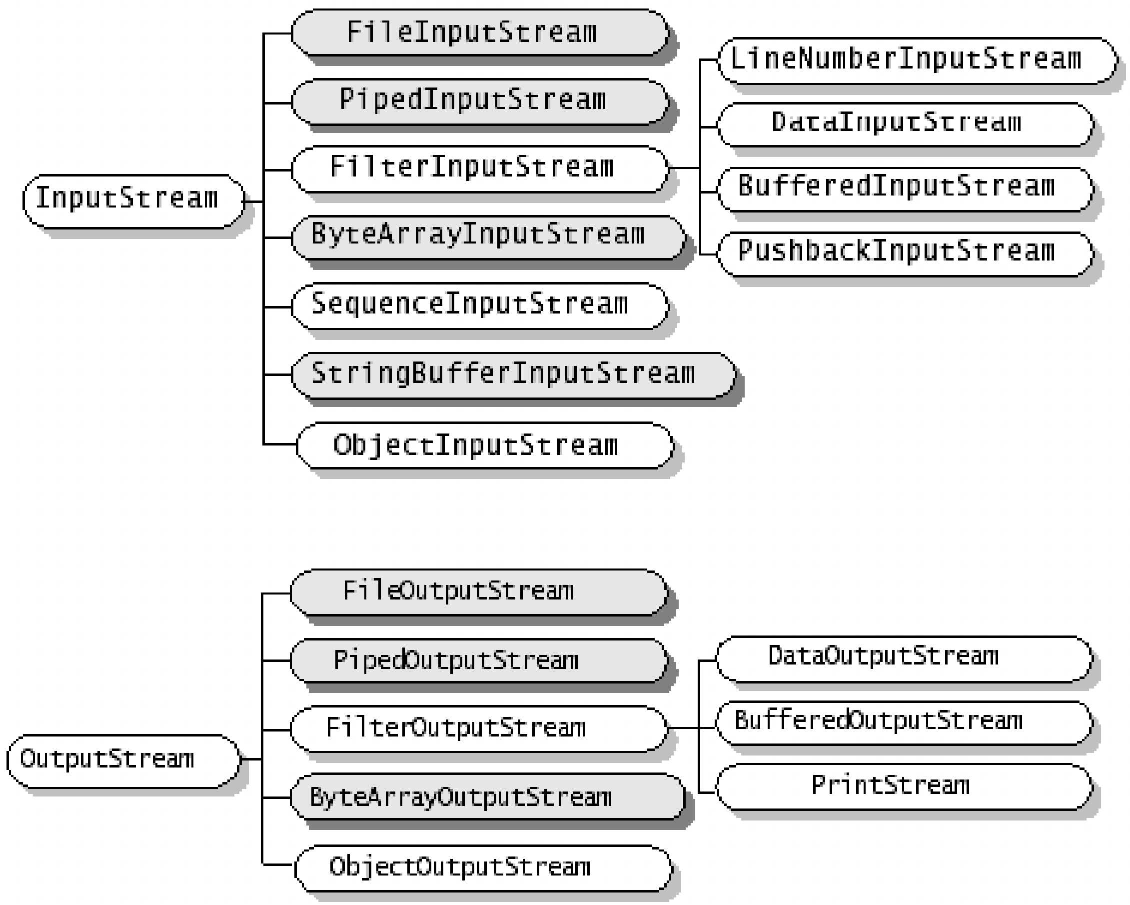
INPUT/OUTPUT

Streams

- Don't use byte streams directly if you want to write/read characters, numbers or higher-order items (entire strings or other Java objects).
- Always check whether there is a “higher” kind of stream class available for your application before using a byte stream directly
- But all “higher” streams make use of byte streams

INPUT/OUTPUT

Streams



INPUT/OUTPUT

Streams

- An example for (good) use of byte streams is the coping of *binary files* (next slide)...
- It also shows a typical loop approach to reading all data from a byte or character stream:

```
while ((c = in.read()) != -1)
        do something with byte c
```

- Here, `-1` stands for "end of file" (means: no more data available)
- Note that the while-loop above works because assignment "`c = in.read()`" not only assigns the result of `in.read()` (i.e., the most recent byte read from the stream) to variable `c`, but also returns the read byte (that is, the new value of `c`).
- https://docs.oracle.com/javase/8/docs/api/java/io/FileInputStream.html#read--

- Virtually any stream operation might throw exceptions!

INPUT/OUTPUT

Streams

```
import java.io.*;
public void copyBin() throws IOException {
      FileOutputStream out = null;
      FileInputStream in = null;
       try {
               in = new FileInputStream("sourceFile.bin");
               out = new FileOutputStream("destinationFile.bin");
               int c;  // represents one byte, not actually an int!
               while ((c = in.read()) != -1) // -1 stands of "end of file"
                       out.write(c);
        } catch (IOException e) {  // any stream operation might fail!
               System.err.println(e);  // (e.g., if file doesn't exist)
        } finally {
            if (in != null) in.close(); // always close a stream after use
            if (out != null) out.close();
        }
}
```

name of the file from which the stream takes its data

name of the file into which the out-stream "flows"

INPUT/OUTPUT

Streams

- A predefined byte input stream is provided by `java.lang.`**`System.in`**
- It represents the operating system stream for console input using the keyboard (but could be attached to other data sources too)
- System.in provides the most simple way to create a (primitive) user interface (together with the well-known `System.out` output stream)
- However, to be used for entering text, it needs to be *wrapped* into higher-order streams (later...), because it is just a byte stream
- Here's an example...

INPUT/OUTPUT

Streams

```
import java.io.BufferedReader;
import java.io.IOException;
import java.io.InputStreamReader;

public class KeyboardInputDemo {
  public static void main(String[] args) {
        System.out.println("Please enter a line of text:");
        try{
            BufferedReader bufferReader =
                new BufferedReader(new InputStreamReader(System.in));
            String s = bufferReader.readLine();
            System.out.println("This is what you have entered: " + s);

        } catch(IOException e) {  // all I/O operations can cause exceptions
            System.err.println(e);
        }
  }
}
```

INPUT/OUTPUT
Streams

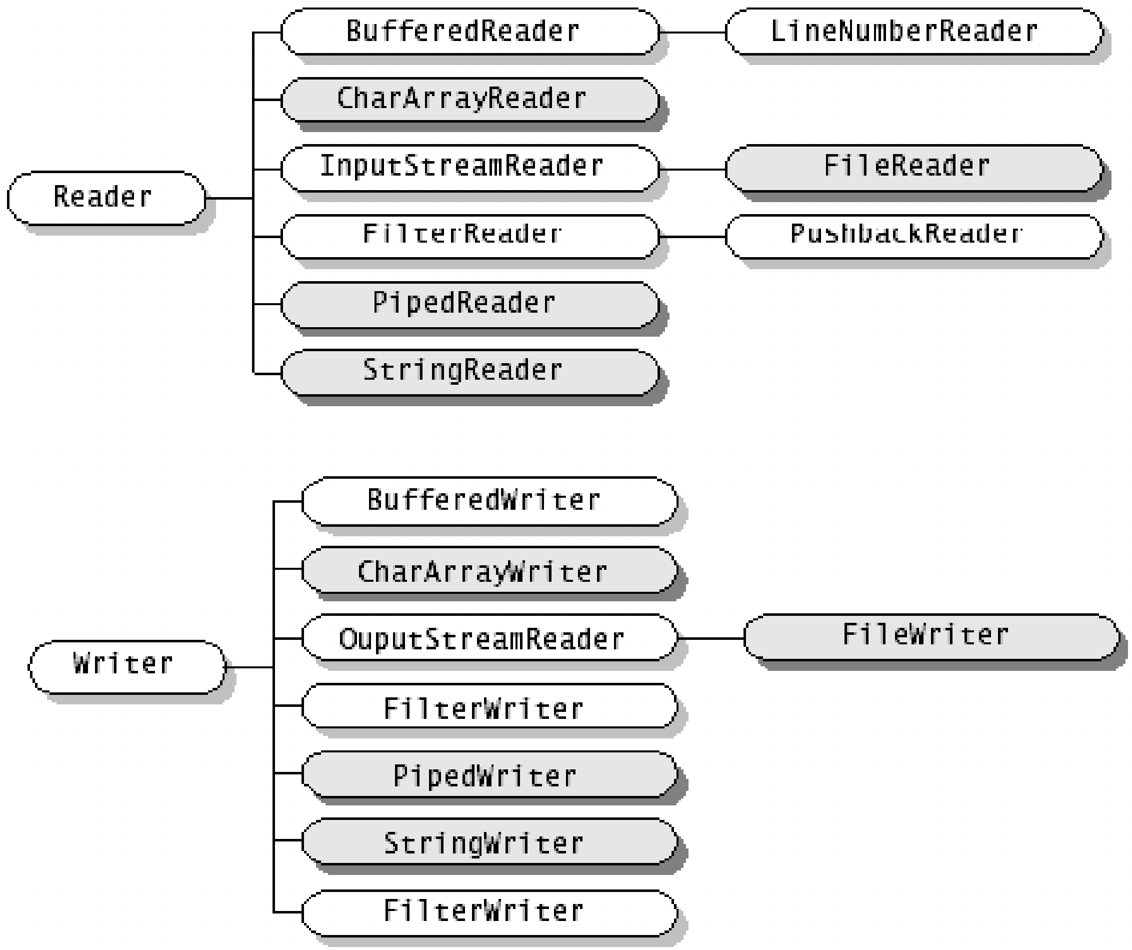
- *Character streams* are intended for the input/output of sequences of characters.
- Character streams automatically use underlying byte streams. This means that each character is automatically translated into a certain short sequence of bytes (typically one or two bytes)
- Multiple encoding (translation) schemes (so-called *charsets*) exist (e.g., ASCII, UTF-16).
  Each machine has a default charset. Most built-in charsets in Java are based on *Unicode* (UTF-8, UTF-16). Other charsets are considered to be more or less outdated

INPUT/OUTPUT

Streams

- In modern Java, character streams are subclasses (or subsub…classes) of
  - `java.io.Reader`
  - `java.io.Writer`
- These classes form <u>their own class hierarchy</u> outside the older non-writer/reader stream classes
- Examples:
  - `java.io.FileWriter`
    - Similar to `FileOutputStream`, but for files which consist of characters only
  - `java.io.FileReader`
    - Similar to `FileInputStream`, but for characters

INPUT/OUTPUT
Streams



INPUT/OUTPUT

Streams

- The following shows a variant of the file-copy example, but now using character streams instead of byte streams...
- This only works for files which consist of characters only.

INPUT/OUTPUT

Streams

```
public static void copy() throws IOException {
    FileReader inputStream = null;
    FileWriter outputStream = null;
    try {
        inputStream = new FileReader("inputTextFile.txt");
        outputStream = new FileWriter("outputTextFile.txt");
        int c; // now each int represents a single character
        while ((c = inputStream.read()) != -1)
            outputStream.write(c);
    } catch (IOException e) {
        System.err.println(e.getMessage());
    } finally {
        if (inputStream != null) inputStream.close();
        if (outputStream != null) outputStream.close();
    }
}
```

INPUT/OUTPUT

Streams

- So, `FileWriter` can be used to write character sequences into a file.
- Observe that `FileWriter` normally creates a new file. However, using the following constructor call it is instructed to *append* data to an existing file:

```
FileWriter myFileWriter = new
    FileWriter(fileName, true);
```

- (Analoguosly, use `FileOutputStream(fileName, true)` to append data to the existing contents of a binary file)

INPUT/OUTPUT

Streams

- An important operation on streams is the "wrapping" of a higher-level stream around some underlying stream.
  The higher-level stream is called a *filter stream* or *wrapper stream* (a "wrapper" for short)
- "*Wrapping*" is done by passing a stream object as constructor argument to the constructor of the wrapper stream class.
- Many of these "wrappers" are derived from one of the following classes (or subclasses of these):
  - `java.io.FilterOutputStream`
  - `java.io.FilterInputStream`
- `E.g. class CipherOutputStream extends FilterOutputStream.`
  Allows data to be encrypted before being written out to disk.

INPUT/OUTPUT

Streams

- Some types of streams need to be used as “wrappers”, they cannot be used standalone (e.g., `java.io.BufferedReader`)
- Wrapping can be nested; an example for a "double-wrapping" (see the `System.in`-example before):

```
BufferedReader bufferReader =
     new BufferedReader(
            new InputStreamReader(System.in));
```

- Byte stream `System.in` is wrapped inside an `InputStreamReader` stream (to allow reading of characters) which is wrapped inside a `BufferedReader` stream (for caching, in order to improve performance. Chunks of the file are read from the `InputStreamReader` by `BufferedReader` and are available without having to read from disk each time .read() is called.)

INPUT/OUTPUT

Streams

- Some other useful filter stream types (see Java API):
  - `javax.crypto.CipherOutputStream`
    - Encrypted output
  - `javax.crypto.CipherInputStream`
    - Decrypted input
  - `java.util.zip.ZipOutputStream`
    - Writes compressed data
  - `java.util.zip.ZipInputStream`
    - Un-compresses data
- Example:

```
ZipOutputStream out =
   new ZipOutputStream(new FileOutputStream(fileName));
```

INPUT/OUTPUT

Streams

- *Data streams* are filter streams ("wrapped" around byte streams) which allow to write/read primitive data type values (such as `int`, `boolean`, `floor`, `double`...).
- Examples
  - `java.io.DataOutputStream`
  - `java.io.DataInputStream`
- They cannot be used for writing/reading objects, however they can write/read entire strings character- or byte-wise (normally not recommended)
- Later we will see how to write and read objects, using *serialization* and *object streams*
- Data streams can be used for writing/reading strings, but this is not recommended any more. Use classes descending from classes `Reader`/`Writer` instead (e.g., `FileWriter`, `InputStreamReader`)

INPUT/OUTPUT

Streams

- The following example wraps a data stream around a file output stream, in order to write a few numbers and a boolean value into a file...
- Observe that with data output streams, values are <u>not</u> written in human-readable form into the file (in contrast to stream `PrintWriter` shown later...).

INPUT/OUTPUT

Streams

name of the file

```
...
try {
  DataOutputStream outputStream =
             new DataOutputStream(
                    new FileOutputStream("data.dat"));

  outputStream.writeInt(12); // puts number 12 into the file
  outputStream.writeDouble(534.0276);
  outputStream.writeBoolean(false);
  outputStream.close();
} catch (IOException e) {
  System.out.println(e.getMessage());
}
```

INPUT/OUTPUT

Print streams

- *Print streams* and *print writers* are string output streams which provide *formatting* capabilities ("pretty printing").
- Ready-to-use print streams for console output are provided by
  - `java.lang.System.out`
  - `java.lang.System.err`
- `System.out` and `System.err` map to the standard operating system stream used for showing text in the console (but they could be redirected to, e.g., a file).

INPUT/OUTPUT

Print streams

- Print writers are more advanced; they are especially designed for "pretty" text output. Class `java.io.PrintWriter`
- They provide methods for formatted output of data (including numbers and strings) *in text format*:
  E.g.,
  ```
  myPrintWriter.print(1234.5678);
  myPrintWriter.println("Hello!");
  ```
- These methods write out numbers in their human-readable <u>textual</u> representation(!), not the bytes which represent that data item in memory.
  E.g. number 123 is outputted as string "123". After reading this using an input stream or a reader, you would need to *parse* the string in order to get a data value of, e.g., type `int` or `double`... !
- `println()` additionally terminates the line., by appending an `'\n'` (=newline) character to the output

INPUT/OUTPUT

Print streams

- Technically, they retrieve `String.valueOf(data)` (equal to `data.toString()`) and encode it into a byte sequence (like character streams), which is then processed by some underlying byte stream.

INPUT/OUTPUT

Print streams

- Print writers have strongly enhanced formatting capabilities.
- Basic method:
  `public PrintWriter format(String f, Object arg1, Object arg2, Object arg3, …)`

  (the return value is just the print writer the method was called on, and can usually be ignored)
- `f` defines what is printed. It contains placeholders for the `arg1,…` and the specification of the formatting
- Simple example:

```
double a = 43.342;
System.out.format("The sine of %d is %f.2", a, Math.sin(a));
```

- This approach imitates the `printf()`-method known from C/C++
- For details, see Java API

INPUT/OUTPUT

Streams

- *Buffered streams* use *buffers* (areas in memory, similar to *caches*) which are used in order to speed up writing/reading. They can only be used as wrappers "around" other streams.
- Buffered output:
  - Data is not sent directly to the destination (e.g., a file), but stored in the buffer first.
  - Only when the buffer is full (or the stream is "flushed"), its content is actually sent to the destination (e.g., written into the file).
- Buffered input:
  - Data is not retrieved directly from its source (e.g., a file), but from the buffer (provided of course that the buffer is filled with data).
  - Only when the buffer is empty, it is filled again with new data from the source of the wrapped stream.

INPUT/OUTPUT

Streams

- Example:

```
BufferedReader in =
    new BufferedReader(
        new InputStreamReader(
            new FileInputStream("myfile.dat")), 5000);
```

  - `java.io.InputStreamReader` is used as a wrapper to retrieve a character stream from a primitive byte stream (consider using a `java.io.FileReader` instead…)
  - `java.io.BufferedReader` is used to add a buffer of size 5000 in order to make the reading process more efficient

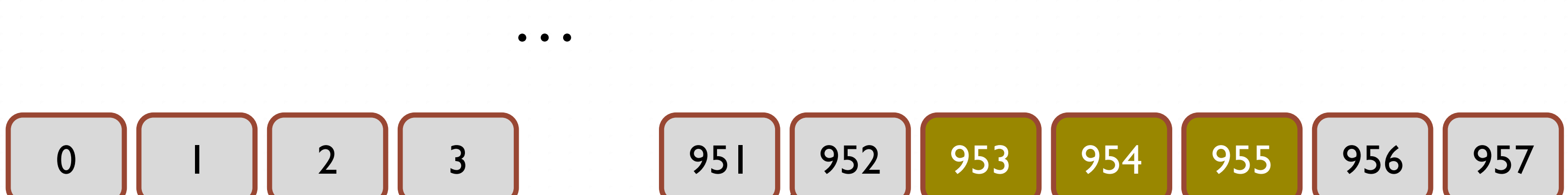
INPUT/OUTPUT
Streams

- Buffered output streams actually write data to their destination when
  - their buffer if full
  - the stream is *flushed* (call of `streamObject.flush()`)
  - the stream is closed (call of `streamObject.close();` implies flushing)
- Technically, flushing causes all buffered data written to the underlying unbuffered stream immediately
- For that purpose, buffered output streams (e.g., `BufferedWriter`) provide the method `void flush()`
- But even flushing or closing a stream <u>doesn't</u> *guarantee* that the data actually arrives correctly at its physical destination (e.g., disk, remote network host…), even if no exception is thrown!

INPUT/OUTPUT

Random access files

- A shortcoming of (ordinary) streams is that they allow only sequential access to data.
- Sometimes, its required to provide *random access* to external data, especially a file.
- Compare this concept to random access data structures (such as arrays).
- If only a portion of the file needs to be read or updated, and one knows at which position (*index*) in the file this piece of data resides, *random access files* are more efficient than streams:



- Class `java.io.RandomAccessFile`. Not a stream class!

INPUT/OUTPUT

Random access files

- Random access files are similar to arrays...
- A random access file provides a method `seek()` which allows to move a *file pointer* to <u>any</u> position within the file.
- The file pointer points at the position for the next read or write operation (the position is measured in bytes, starting with 0 = first byte in file).
- In contrast to the stream classes seen before, `RandomAccessFile` provided methods for reading or writing at the given position (similar to the methods of `DataOutputStream` and `DataInputStream`).
- However, you need to specify in the constructor call whether you need the file for reading only, or for reading <u>and</u> writing (see Java API documentation).

INPUT/OUTPUT

Random access files

"rw" specifies that we want to read <u>and</u> write from/to file test.bin

- Example:

```
RandomAccessFile f = new RandomAccessFile("test.bin", "rw");
f.seek(953);  // move file pointer to position 953 (measured
              // in bytes)
f.writeDouble(3.141592653);

f.seek(f.length());  // this trick allows to append data to
the file by writing to the position directly after the last
existing position in the file
f.writeInt(78);
```

INPUT/OUTPUT

Tokenizing streams

- Often it is required to break the data delivered by a character input stream into *tokens*.
- A *token* is an atomic element encountered during parsing of texts, e.g., a word or a number.
  To "*tokenize*" means to split a character sequence (e.g., a string or a stream) into its tokens. Tokens are separated by *delimiters* (whitespace, commas or user-defined characters).
- Which delimiter to use is decided by the programmer.

INPUT/OUTPUT

Tokenizing streams

- E.g., here's a string consisting of four tokens
  “123.45,222.99, .743, 3.141592653”
  (here the tokens are substrings which represent numbers, and the delimiter is ',')
- Another example:
  “The fox jumps over the wall”
  The tokens of this string are the words "The", "fox", "jumps", "over", "the", and "wall".
  Delimiter: whitespace, which is the default delimiter used by class `Scanner`

INPUT/OUTPUT

Tokenizing streams

- Java provides several methods for tokenization…
- Early Java had already
  - `java.io.StreamTokenizer`
  - `java.util.StringTokenizer` (not recommended any more)
- Java 1.4 introduced
  - `public String[] split(String regularExpression)`
- Since Java 5 (=1.5), there is the powerful and flexible (but slower) `java.util.Scanner`
- The main difference between the two latter approaches and the …`Tokenizer`s is the use of *regular expressions*.

INPUT/OUTPUT
Tokenizing streams

- There are several ways to use these in order to tokenize stream data:
  - read the entire stream content and store it as a string object first, then tokenize this string (e.g., using `string.split()`) (not always possible, typically not recommended…).
  - Use `new StreamTokenizer` (that's yet another input stream);
  - Use `Scanner(someInputStream)` – the most flexible approach
  - Also possible: use of `Scanner` in order to tokenize a `String` object instead of a stream: `Scanner(someString)`
- The following example prints all comma-separated tokens of a text stored in a file...

INPUT/OUTPUT
Tokenizing streams

we use Scanner like a wrapper stream, although it is actually not a stream class

```
import java.io.*;
import java.util.Scanner;

public void tokenizeFile() throws IOException {
        Scanner s = null;
        try {
            s = new Scanner(
                              new BufferedReader(new FileReader("text.txt")));
            s.useDelimiter("\\s*,\\s*"); // use commas as delimiters.
                                                  // (default delimiter is
                                                    // whitespace!)
              while (s.hasNext()) { // any more tokens in stream?
                System.out.println(s.next());  // print next token
            }
        } finally {
            if (s != null) s.close();
        }
 }
```

INPUT/OUTPUT

Serialization

Remark:

`useDelimiter("\\s*,\\s*")` uses a so-called *regular expression* to specify a string pattern (a comma with optional surrounding whitespace).

Regular expressions are not covered in this module, but they are a powerful tool so that it is recommended that you familiarize yourself with them at one point. They exist in more or less identical form in most mainstream programming languages.

https://www.ocpsoft.org/opensource/guide-to-regular-expressions-in-java-part-1/

INPUT/OUTPUT

Serialization

- What if we would like to store or read the state of an arbitrary entire object (vs. primitive data like `int`'s and characters) into/from a file, or send objects over a network...?
- So far, we could do this only in a special case (namely `String` objects, because these are just sequences of characters)
- For other kinds of Java objects, we could think of using the result of `object.toString()`...
  Problems with this idea:
  - Not for all objects an unambiguous string representation exists!
  - Requires a proprietary parser (the program part which analyses the string and converts it to a data structure). Such code might be complex and error-prone.

INPUT/OUTPUT

Serialization

- A much more convenient solution is *serialization*
- Serialization means: the state of an arbitrary Java object (of almost any class) is automatically translated into a form which can be stored outside Java (e.g., in a file), or be transmitted over a network
- The object is represented in a *serialized* form which is sufficient to be reconstructed later.
- The inverse procedure (decoding) is called *deserialization*
- (Binary) serialization writes objects as byte sequences (i.e., series of bytes, therefore the name)
- Streams which provide serialization / deserialization of objects:
  - `java.io.ObjectOutputStream`
  - `java.io.ObjectInputStream`

INPUT/OUTPUT

Serialization

- Saving an object to a file using serialization:

```
FileOutputStream out = new FileOutputStream("date.dat");
 ObjectOutputStream s = new ObjectOutputStream(out); // wrapper
 Date date = new Date();   // the object we would like to serialize
 s.writeObject(date);
 …
 out.close(); // don't forget to close all streams after use
 s.close();
```

INPUT/OUTPUT

Serialization

- Reading a serialized object (deserialization):

```
FileInputStream in = new FileInputStream("date.dat");
ObjectInputStream s = new ObjectInputStream(in);
Date date = (Date)s.readObject();
                // gives us back the original object
…
in.close();
s.close();
```

- The serialized objects must be read from the input stream in the same order in which they have been written into the output stream. (Analogously for any other type of stream.)

INPUT/OUTPUT
Serialization

- Typical uses of serialization:
  - *Persistence* of objects: objects should "survive" the end of the program, e.g., in order to be archived.
  - Networked software, such as client/server software: in order to submit Java objects to other programs, they need to be serialized before they can be sent via the network
  - Saving a "snap shot" of the current state of a program to a file.

INPUT/OUTPUT

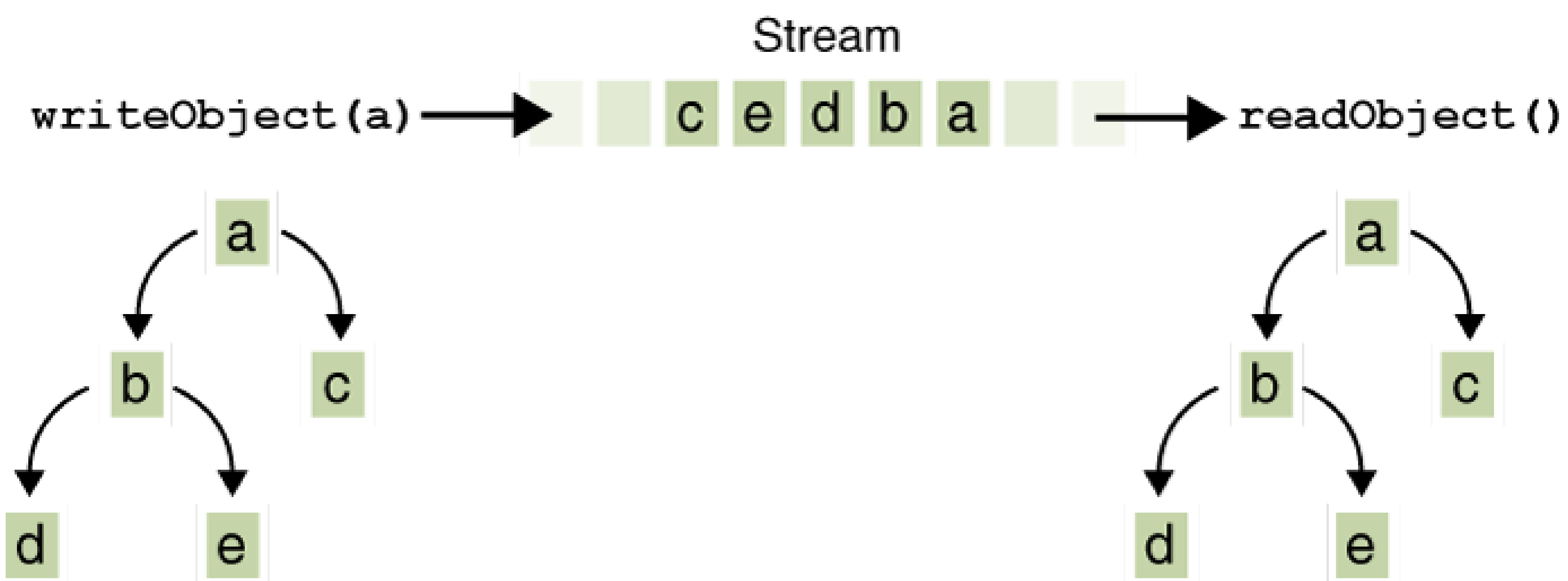
Serialization

- How does serialization work...?
- If the object consists of fields with primitive types (`int`, `Boolean`, `double`...) only, then serialization is trivial (cf. data input/output streams).
- But if an object contains references to other objects (i.e., objects stored in fields),
  serialization needs to write all referenced objects into the stream too (in a specific, reversible order).
- The referenced objects can contain references to further objects, and so on… $\rightarrow$ *recursion*
- Serialization traverses the *object reference graph* recursively.

INPUT/OUTPUT

Serialization

Serialization (simplified) of the object reference graph for some object `a` of `class A { B b; C c; }`, with `class B { D d; E e; }`



INPUT/OUTPUT

Serialization

- In case a class implements the *marker interface* `Serializable`, Java handles the serialization of objects of this class automatically.
- Most, but not all Java objects can be serialized. Some objects have <u>no state</u> which could be reasonably stored or transmitted (e.g., objects of class `Thread`).

INPUT/OUTPUT

Serialization

- If, like with the standard Java serialization, the objects are written directly as byte sequences (that is, serialization effectively copies the object as it is stored in memory), we speak about *binary serialization.*
- Binary serialization has shortcomings:
  - Transferring serialized objects between programs written in different programming languages doesn't make much sense.
  - It can not even always be guaranteed that a serialized object can be deserialized by another Java program (e.g., serialization of Swing components is JVM-dependant).

INPUT/OUTPUT

Serialization

- Another kind of serialization translate objects into JSON or XML documents *(JSON / XML serialization)*. This is slower and requires more space, but avoids the mentioned shortcomings of binary serialization.
- JSON (JavaScript Object Notation) is a common format for data which can be represented in textual and hierarchical form. Increasingly often used as a data format in Web-based APIs or as a text-based data file format.
- Many modern "Big Data" and cluster computing platforms are using the Java Virtual Machine (JVM), such as Apache Spark. They rely on fast serialization, as they need to distribute Java objects and methods to different cluster machines in the network.