Java IOs

RES, Lecture 1

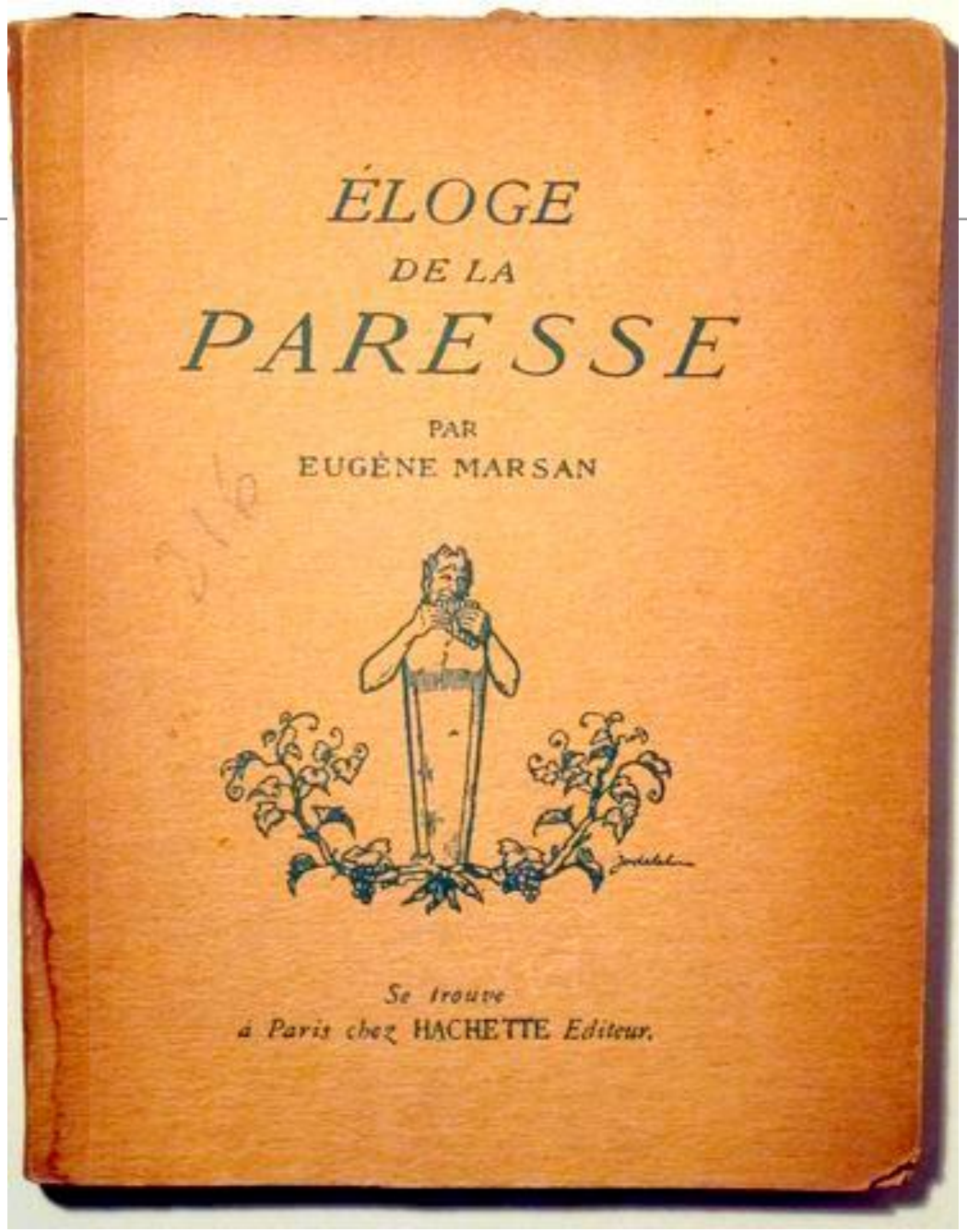
Olivier Liechti
Juergen Ehrensberger

heig-vd
HAUTE ÉCOLE
D'INGÉNIERIE ET DE GESTION
DU CANTON DE VAUD
www.heig-vd.ch

Agenda

heig-vd
HAUTE ÉCOLE
D'INGÉNIERIE ET DE GESTION
DU CANTON DE VAUD
www.heig-vd.ch

- **Week 1**
  - Universal API
  - Sources, Sinks and Streams
  - Performance and Buffering
- **Week 2**
  - The Decorator Pattern and The Mighty Filter Classes
  - Binary vs. Character-Oriented IOs
  - Shit Happens... Dealing with IO Exceptions



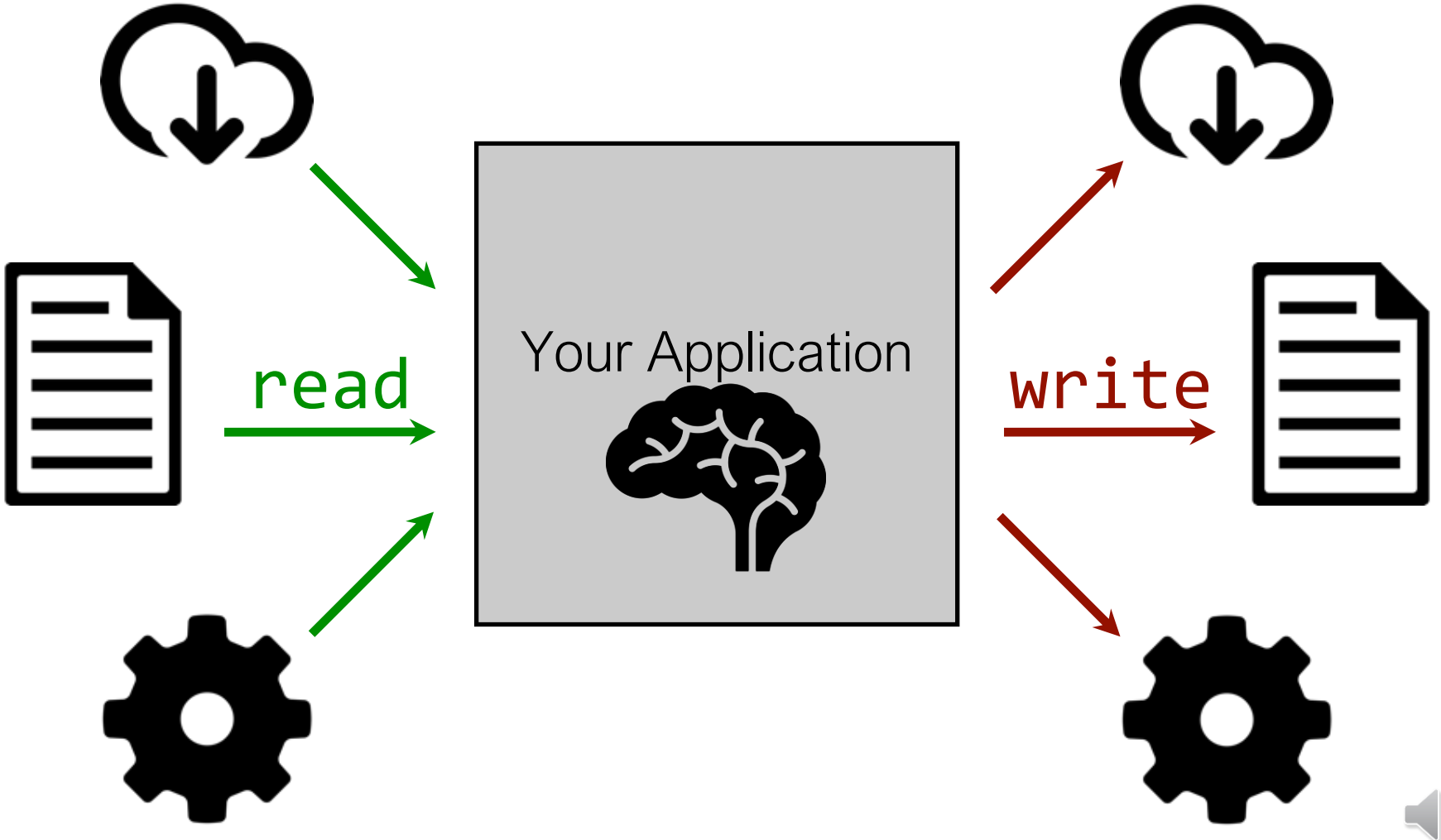
Lombok

maven

GitHub Actions

Universal API

What do we mean by IO?



A Universal API

*At the end of the day, whether you are "talking" to a **file**, to a **network endpoint** or to a **process** does not matter.*

*You are always doing the same thing: **reading** and/or **writing** bytes or characters.*

*The Java IO API is the **toolbox** that you need for that purpose.*

Sources, Sinks & Streams

System.out.println("I like IO");

out

```
public static final PrintStream out
```

The "standard" output stream. This stream is already open and ready to accept output data. Typically this stream corresponds to display output or another output destination specified by the host environment or user.

For simple stand-alone Java applications, a typical way to write a line of output data is:

```
System.out.println(data)
```

See the `println` methods in class `PrintStream`.

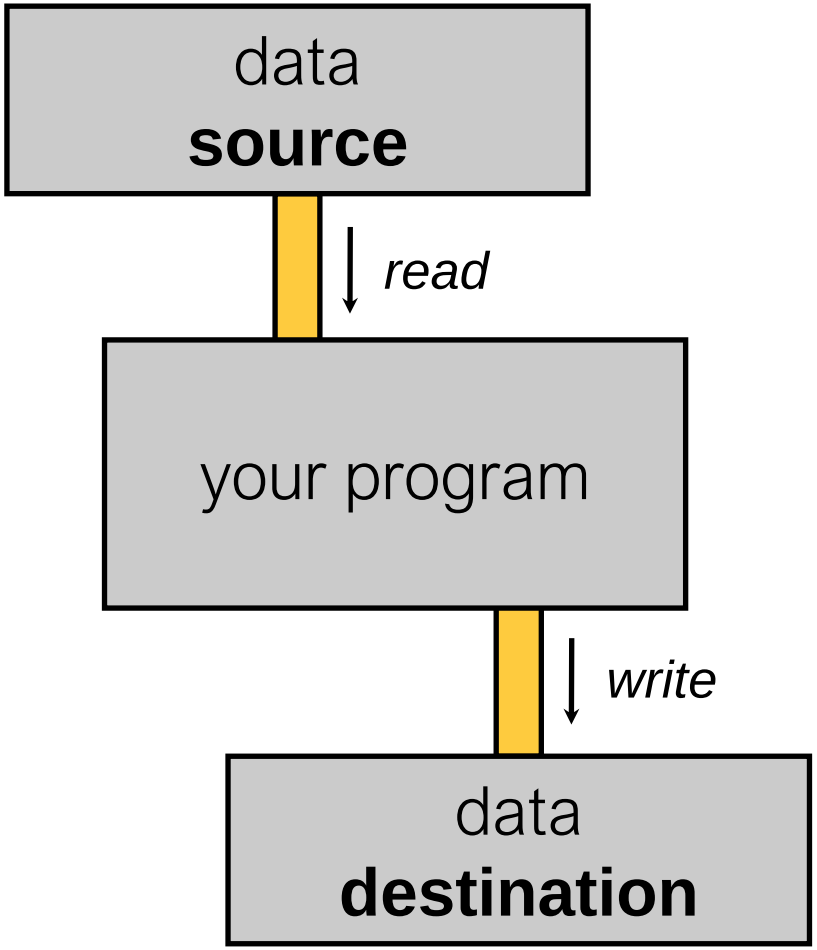
See Also:

`PrintStream.println()`, `PrintStream.println(boolean)`, `PrintStream.println(char)`, `PrintStream.println(char[])`, `PrintStream.println(double)`, `PrintStream.println(float)`, `PrintStream.println(int)`, `PrintStream.println(long)`, `PrintStream.println(java.lang.Object)`, `PrintStream.println(java.lang.String)`

Streams

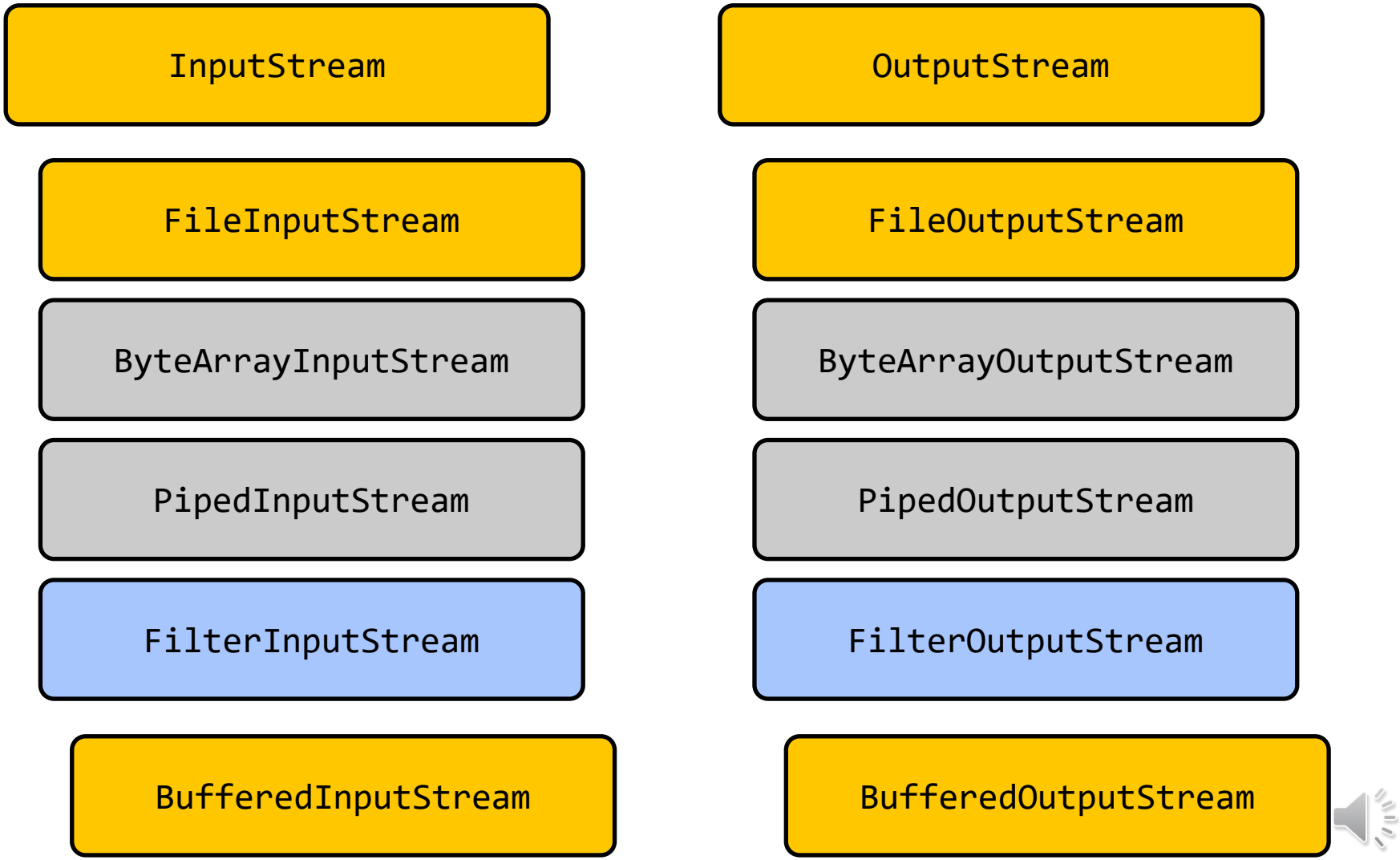
heig-vd
HAUTE ÉCOLE
D'INGÉNIERIE ET DE GESTION
DU CANTON DE VAUD
www.heig-vd.ch

- When we talk about "input/output", or IOs, we think about **producing** and **consuming streams of data**.
- There are different **sources** that contain or produce data.
- There are also different **destinations** that receive or consume data.
- Think about **files**, **network endpoints**, **memory**, **processes**, etc.
- Your **program** can **read data** from a stream. Your program can **write data** to a stream.



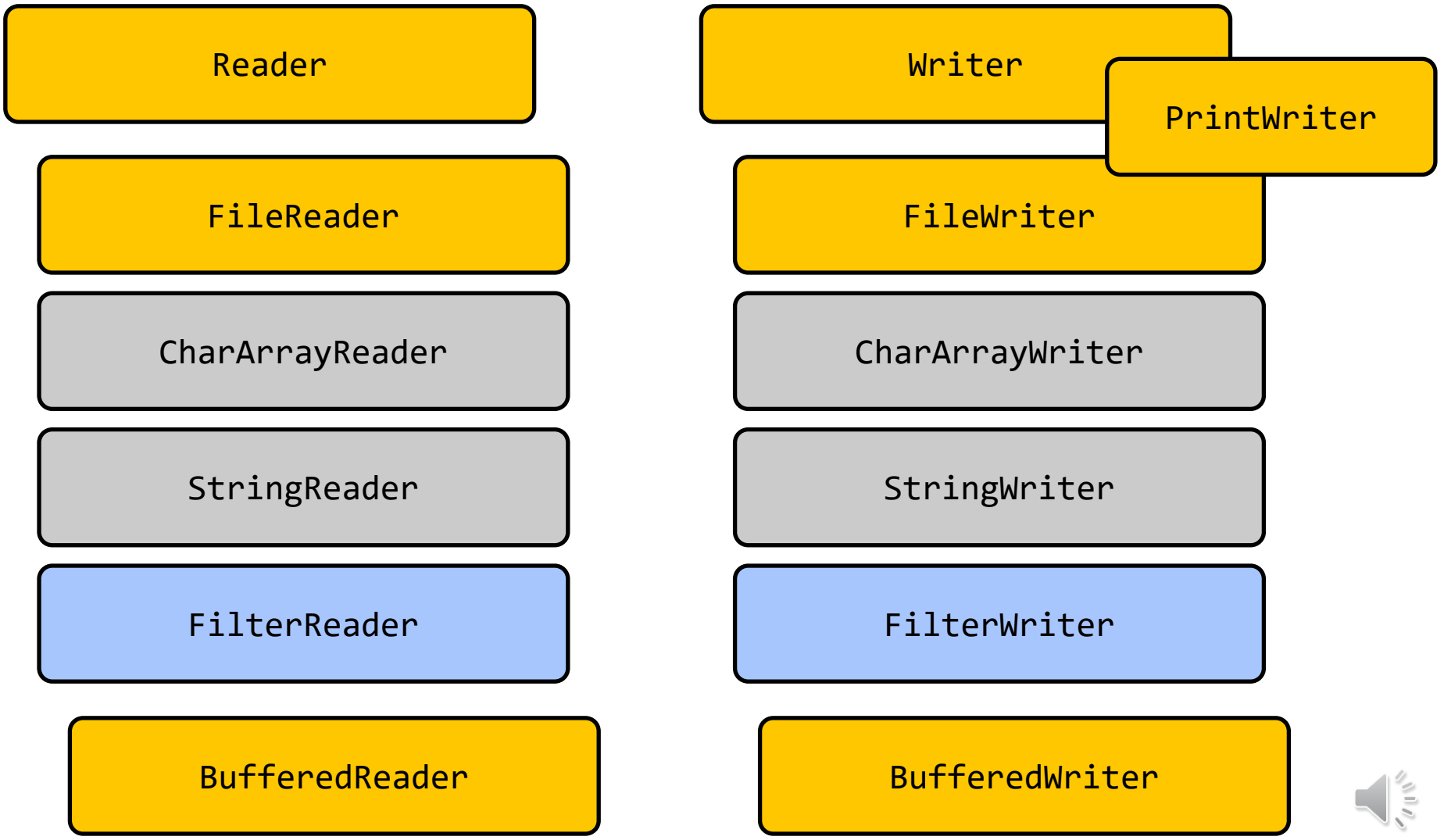
Classes in the `java.io` package (1)

heig-vd
HAUTE ÉCOLE D'INGÉNIERIE ET DE GESTION DU CANTON DE VAUD
www.heig-vd.ch



Classes in the `java.io` package (2)

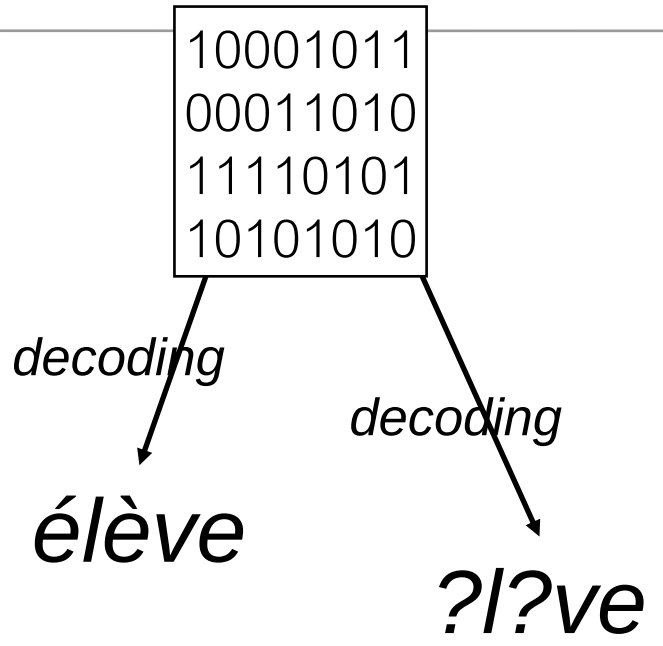
heig-vd
HAUTE ÉCOLE D'INGÉNIERIE ET DE GESTION DU CANTON DE VAUD
www.heig-vd.ch



Bytes vs. Characters

heig-vd
HAUTE ÉCOLE D'INGÉNIERIE ET DE GESTION DU CANTON DE VAUD
www.heig-vd.ch

- There are different types of data sources. Some sources produce **binary data**. Other sources produce **textual data**.
- It's always a **series of 0's and 1's**. The real question is : "how do you interpret these bits"?
- When you deal with **textual data**, the interpretation is not always the same. It depends on the "**character encoding system**"?
- When you deal with IOs, you have to **use different classes** for processing binary, respectively textual data. **Otherwise, you will corrupt data**!



For binary data, use **inputStreams** and **outputStreams**.
For text data, use **readers** and **writers**.

Reading Bytes, One at The Time

heig-vd
HAUTE ÉCOLE D'INGÉNIERIE ET DE GESTION DU CANTON DE VAUD
www.heig-vd.ch

```
int b = fis.read();
while ( b != -1 ) {

  // b has an int value between -1 and 255
  // -1 indicates that we are at the end of the stream
  // b can be casted to a byte, remember that in Java,
  // bytes are signed and have values in the range [-128..127].

  b = fis.read();
}
```

This is a blocking call

Or if you absolutely want to save 1 line...

```
int b;
while ( (b = fis.read()) != -1 ) {

}
```

Reading Bytes, in Blocks

heig-vd
HAUTE ÉCOLE D'INGÉNIERIE ET DE GESTION DU CANTON DE VAUD
www.heig-vd.ch

```
final int BUFFERSIZE = 255;
byte[] buffer = new byte[BUFFERSIZE];

int numberOfNewBytes = fis.read(buffer);

  // we know that numberOfNewBytes bytes have been read
  // we know that these bytes are available in the buffer
  // WARNING: there might be left-over junk after these bytes!

while ( numberOfNewBytes != -1 ) {

  // do something with the bytes in buffer[0..numberOfNewBytes-1]
  // ignore what is left in buffer[numberOfNewBytes.. BUFFERSIZE]

  // read the next chunk of bytes
  numberOfNewBytes = fis.read(buffer);

}
```

Writing Bytes

heig-vd
HAUTE ÉCOLE
D'INGÉNIERIE ET DE GESTION
DU CANTON DE VAUD
www.heig-vd.ch

Write 1 byte

```
OutputStream os = ...;
int b;
// The byte to be written is the eight low-order b.
// The 24 high-order bits of b are ignored. So, b can have an
// int value in the range [0..255]
os.write(b);
```

This is a blocking call

Write a block of bytes

```
OutputStream os = ...;
byte[] data = new byte[BLOCK_SIZE];

data[3] = 22; data[4] = 5; data[5] = 9; data[6] = 7;
// data[0..2] contains junk, data[7..BLOCK_SIZE-1] too

int offset = 3; // because we have started to fill at slot 3
int length = 4; // because we have filled 4 slots
os.write(data, offset, length);
```

Design **Your Code** to Be Universal

heig-vd
HAUTE ÉCOLE D'INGÉNIERIE ET DE GESTION DU CANTON DE VAUD
www.heig-vd.ch

```
/**
 * This interface will work only for data sources on the file
 * system. In the method implementation, I would need to create
 * a FileInputStream from f and read bytes from it.
 */
 public interface IPoorlyDesignedService {
  public void readAndProcessBinaryDataFromFile(File f);
}
```

data source

vs

```
/**
 * This interface is much better. The client using the service
 * has a bit more responsibility (and work). It is up to the
 * client to select a data source (which can still be a file,
 * but can be something else). The method implementation
 * will ignore where it is reading bytes from. Nice for reuse,
 * nice for testing.
 */
public interface INicelyDesignedService {
  public void readAndProcessBinaryData(InputStream is);
}
```

stream

Performance and Buffering

Buffered IOs

heig-vd
HAUTE ÉCOLE
D'INGÉNIERIE ET DE GESTION
DU CANTON DE VAUD
www.heig-vd.ch

Do you know what happens when you do a

```
int c = read();
```

?

Buffered IOs

heig-vd
HAUTE ÉCOLE
D'INGÉNIERIE ET DE GESTION
DU CANTON DE VAUD
www.heig-vd.ch

It's a bit like when you are thirsty and feel like **drinking** something...

!

Buffered IOs

heig-vd
HAUTE ÉCOLE
D'INGÉNIERIE ET DE GESTION
DU CANTON DE VAUD
www.heig-vd.ch

It takes you 56' to sip a beer

20 min 5 min 10 min 20 min 1 min

Buffered IOs

Thirsty again...

?

Buffered IOs

heig-vd
HAUTE ÉCOLE
D'INGÉNIERIE ET DE GESTION
DU CANTON DE VAUD
www.heig-vd.ch

It takes you 56' again to sip the next beer

20 min 5 min 10 min 20 min 1 min

Buffered IOs

Can we do better...

?

Buffered IOs

heig-vd
HAUTE ÉCOLE
D'INGÉNIERIE ET DE GESTION
DU CANTON DE VAUD
www.heig-vd.ch

It still takes you 56' to bring back a pack of beers...

20 min 5 min 10 min 20 min 1 min

Buffered IOs

Thirsty again...

?

Buffered IOs

heig-vd
HAUTE ÉCOLE
D'INGÉNIERIE ET DE GESTION
DU CANTON DE VAUD
www.heig-vd.ch

It now only takes 2 minutes to sip the next one!

1 min 1 min

Buffered IOs

heig-vd
HAUTE ÉCOLE
D'INGÉNIERIE ET DE GESTION
DU CANTON DE VAUD
www.heig-vd.ch

Coming back to

```
int c = read();
```

- If you don't use buffered IOs, calling read() will issue **one system call** to retrieve **one single byte**... which is not efficient.
- With buffered IOs, calling read() will **pre-fetch "several" bytes** and store it in a **temporary memory space** (i.e. in a **buffer**). "several" defines the **buffer size**.
- Subsequent calls to read() will be able to **fetch bytes directly from the buffer**, which is very fast.

Buffered IOs

What about

```
write(c);
```

?

Buffered IOs

It's the same thing! There is on gotcha:

Sometimes, you want to immediately send the content of the buffer to the output stream.

```
os.flush();
```

Buffered IOs in Java

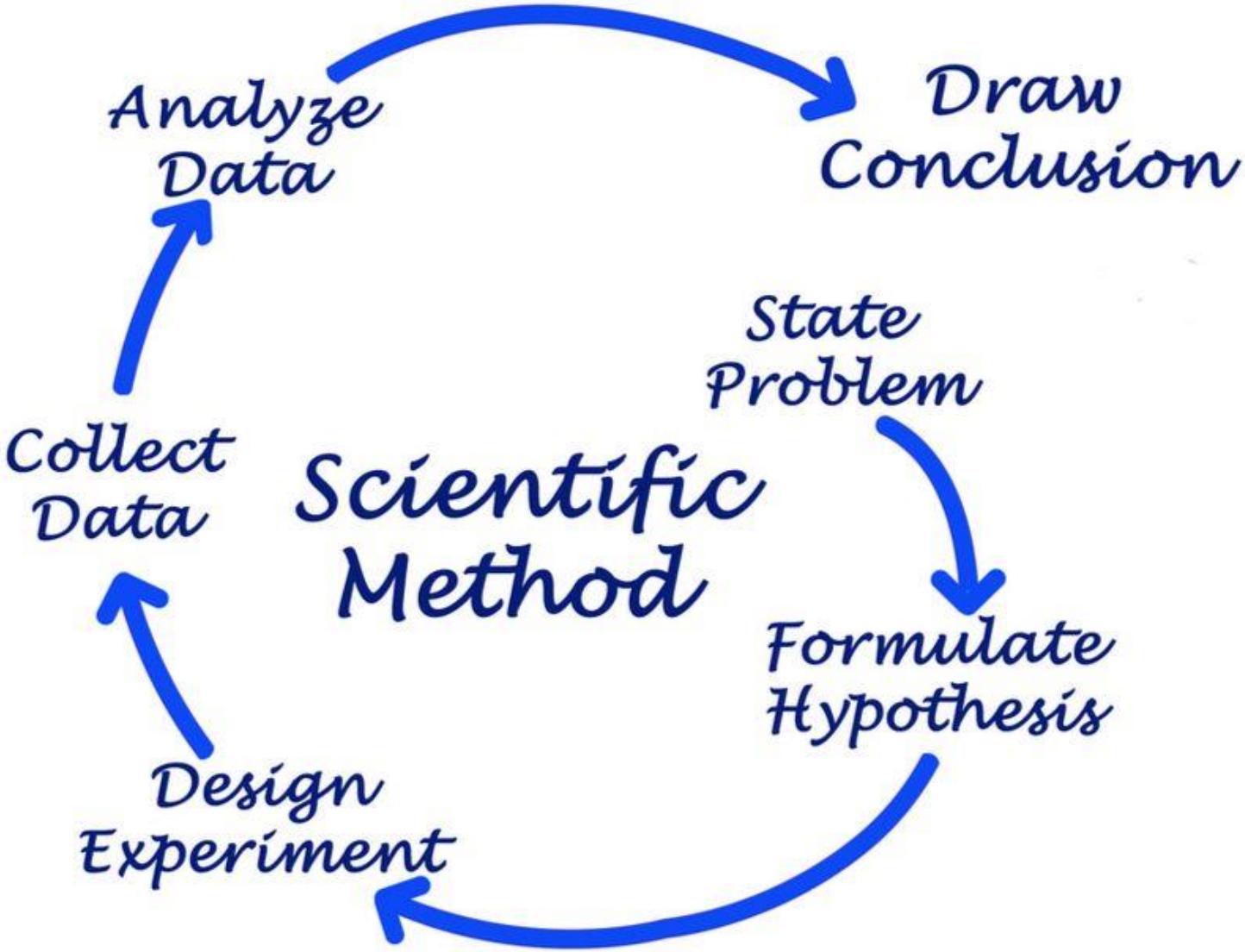
heig-vd
HAUTE ÉCOLE
D'INGÉNIERIE ET DE GESTION
DU CANTON DE VAUD
www.heig-vd.ch

- Later on, we will introduce the **Decorator** Design Pattern
- Using buffered IOs is **as simple as decorating any of your byte or character streams** (don't forget about flushing buffered output streams when required!).

```
InputStream slow;
BufferedInputStream fast = new BufferedInputStream(slow);

OutputStream slow;
BufferedOutputStream fast = new BufferedOutputStream(slow);

Reader slow;
BufferedReader fast = new BufferedReader(slow);

Writer slow;
BufferedWriter fast = new BufferedWriter(slow);
```

Example: **01-BufferedIOBenchmark**

*What is the **real** impact of buffered IOs on performance?*


Scientific Method
State Problem
Formulate Hypothesis
Design Experiment
Collect Data
Analyze Data
Draw Conclusion

BufferedIOBenchmark

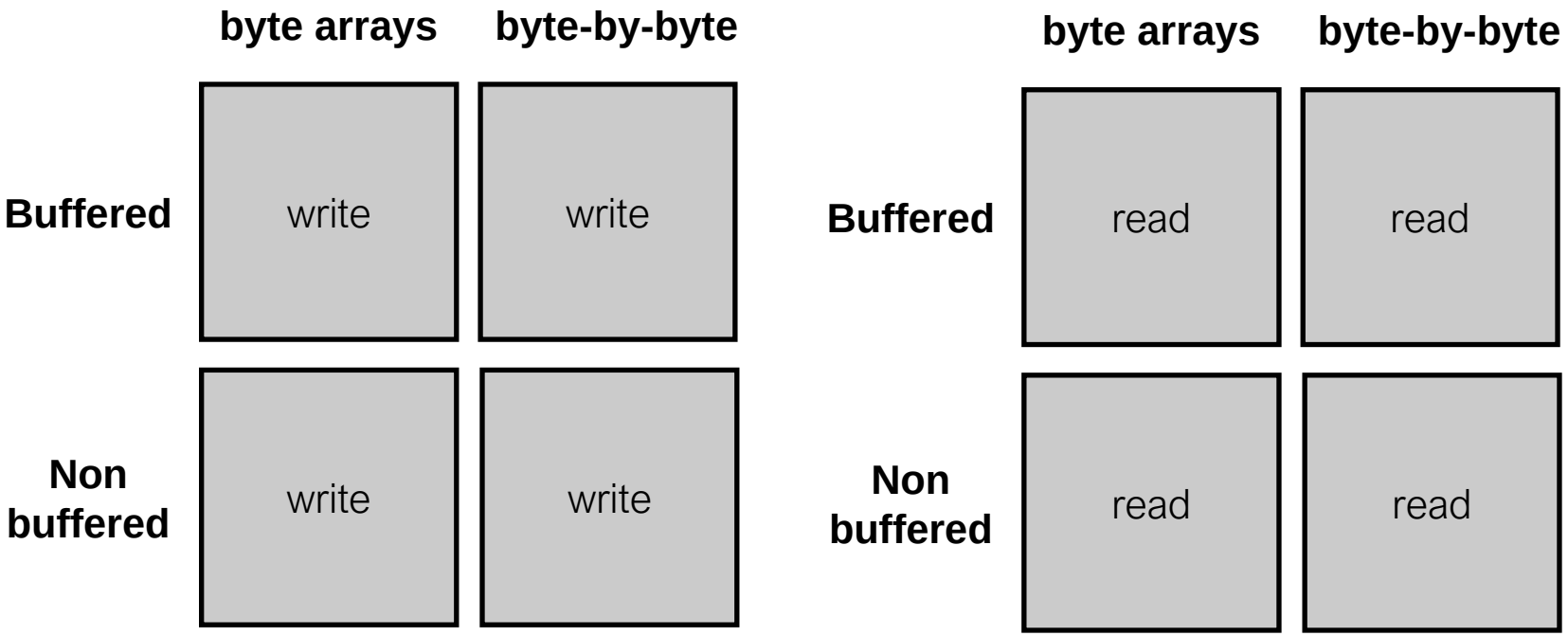
heig-vd
HAUTE ÉCOLE
D'INGÉNIERIE ET DE GESTION
DU CANTON DE VAUD
www.heig-vd.ch

- **Step 1**: code walkthrough and live demo
- **Step 2**: how can we improve the code to be able to analyze results?

Code walkthrough

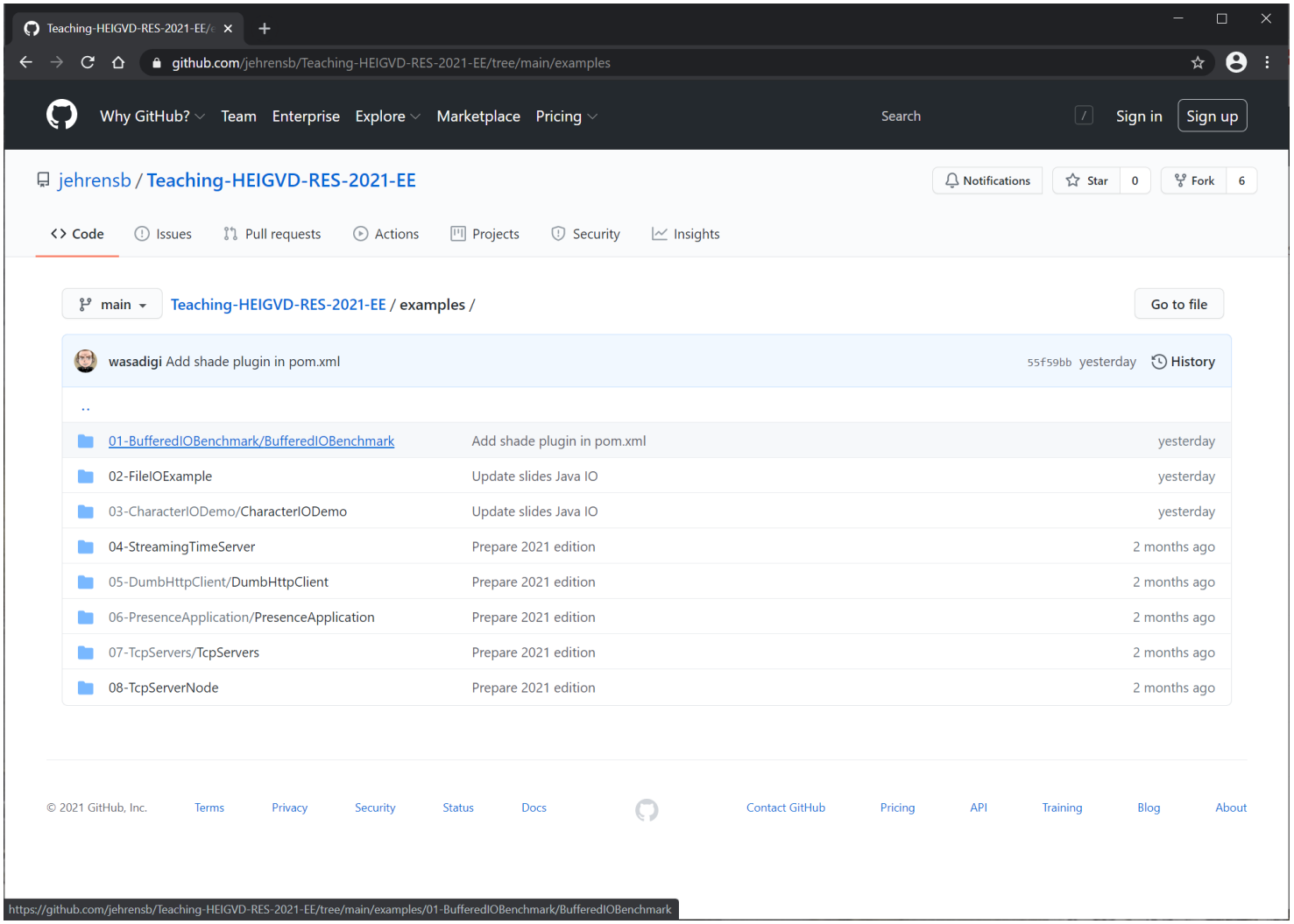
HAUTE ÉCOLE
D'INGÉNIERIE ET DE GESTION
DU CANTON DE VAUD
www.heig-vd.ch

- **Write** and **read** random data to/from **disk**.
- 4 **strategies** (with/without BufferedIOs, operations on byte arrays/single bytes)

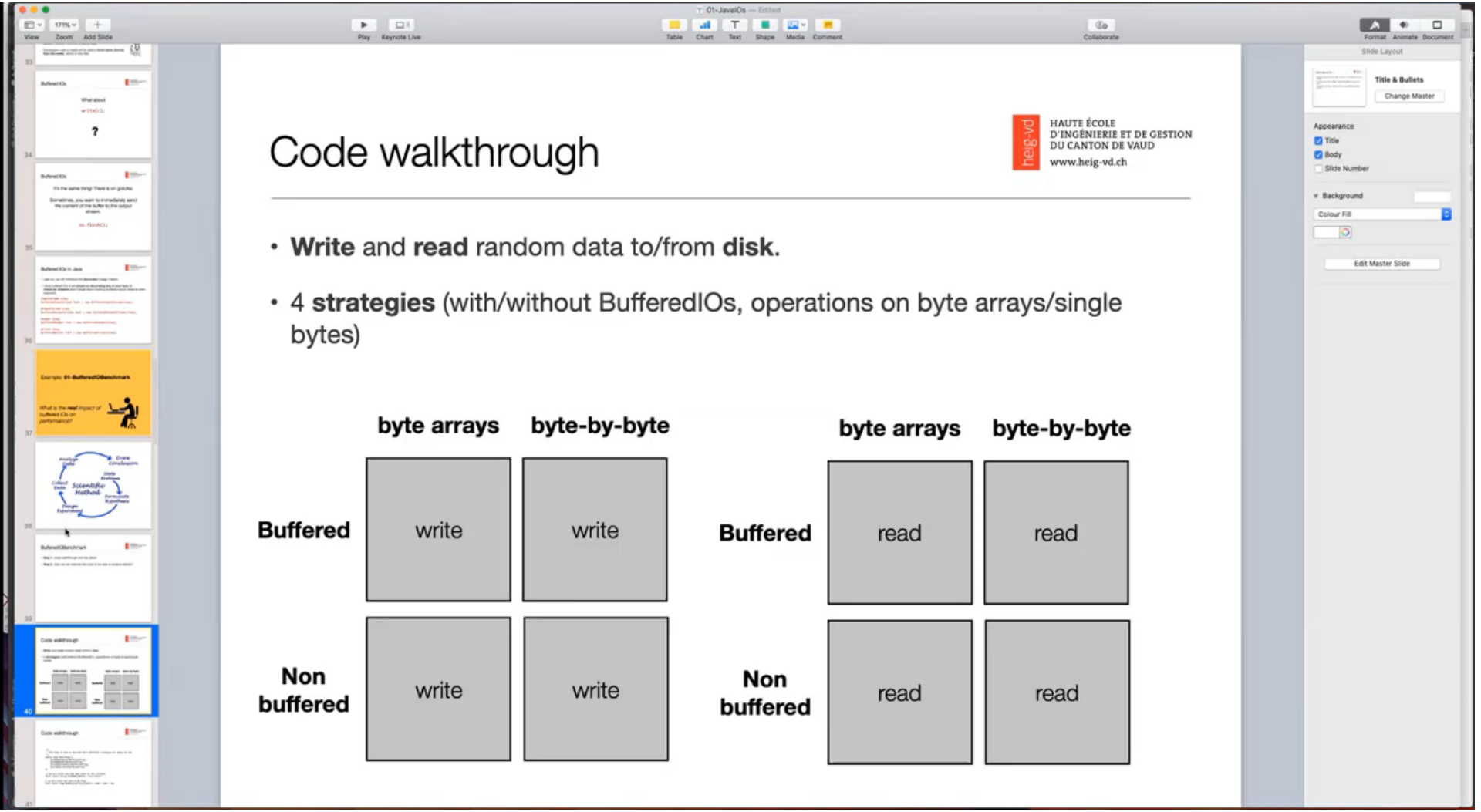


Code walkthrough

HAUTE ÉCOLE
D'INGÉNIERIE ET DE GESTION
DU CANTON DE VAUD
www.heig-vd.ch



Live demo


Code walkthrough
heig-vd
HAUTE ÉCOLE
D'INGÉNIERIE ET DE GESTION
DU CANTON DE VAUD
www.heig-vd.ch
• Write and read random data to/from disk.
• 4 strategies (with/without BufferedIOs, operations on byte arrays/single bytes)
byte arrays
byte-by-byte
Buffered
write
write
Non buffered
write
write
byte arrays
byte-by-byte
Buffered
read
read
Non buffered
read
read

Fin de la première partie