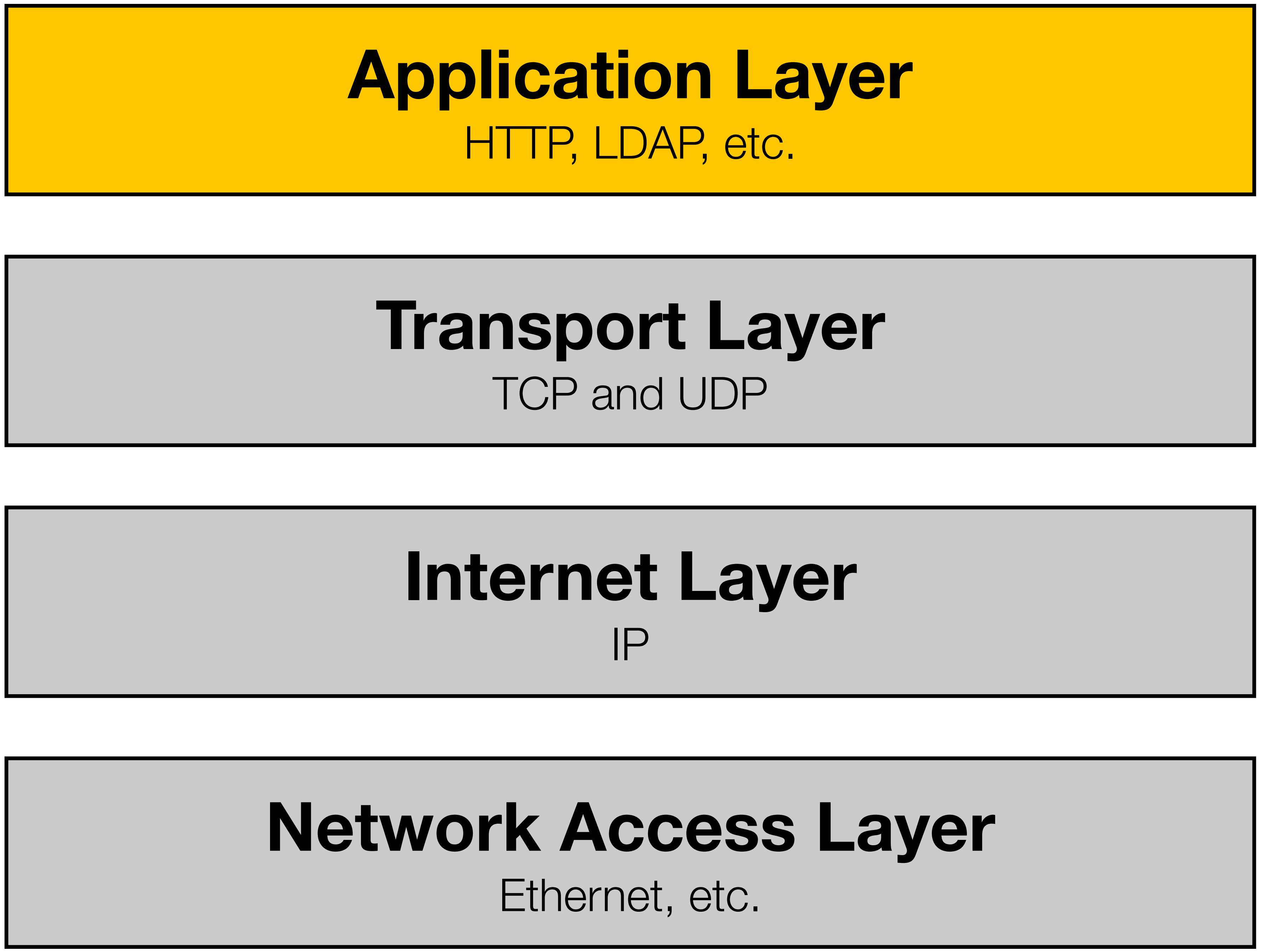
Introduction

TEchnologies Internet (TEI)

Olivier Liechti

heig-vd
Haute Ecole d'Ingénierie et de Gestion du Canton de Vaud

TEI and TCP/IP



Objectives of the Course (1)

heig-vd
Haute Ecole d'Ingénierie et de Gestion
du Canton de Vaud

- Learn how to use the **standard Java IO API** to work with streams of bytes and characters. Understand that it is possible to use the same abstractions when dealing with **files**, **network endpoints** and **memory**.
- Learn how to use the **Socket API** to write client-server applications. Learn how to use **TCP** and **UDP** from the application level.
- Study the **specification of the HTTP protocol** and be able to describe its **core concepts** (which are the basis of all RESTful systems).
- Be able to describe the **structure of HTTP requests and responses**. Be able to analyze HTTP traffic using different tools.
- Be able to implement **simplified HTTP clients and servers in Java**, starting from scratch. Be able to combine their functionality in a **HTTP proxy**.

Objectives of the Course (2)

heig-vd
Haute Ecole d'Ingénierie et de Gestion du Canton de Vaud

- Be able to **design and implement a complete infrastructure** based on open source components for delivering a HTTP service. Integrate redundancy in the infrastructure to achieve qualities of services (availability, scalability, security). Be able to describe the multiple roles of an **HTTP reverse proxy**.
- Study the concepts defined in the **LDAP specifications**. Learn how to use tools to deploy and interact with LDAP directories. Learn how to import various data into an LDAP directory.
- Get an overview of technologies related to **web services**. Understand the differences between the "**Big Web Services**" and the "**RESTful Web Services**" approaches.
- Learn **how to use an existing RESTful API**. Learn **how to design and implement a RESTful API**.

Tools

heig-vd
Haute Ecole d'Ingénierie et de Gestion
du Canton de Vaud

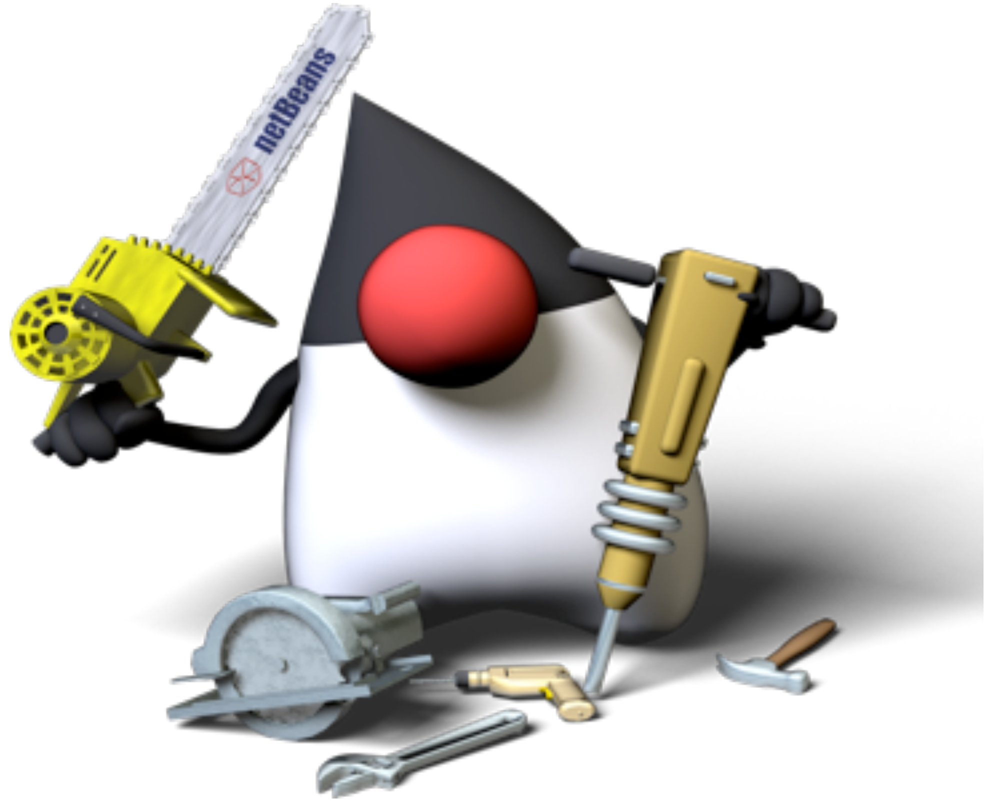
Apache
HTTP SERVER PROJECT

Web Services for the Real World

RESTful
Web Services

O'REILLY

Leonard Richardson & Sam Ruby



GlassFish

OpenDS
Open Source. Open Standards.
Open Directory Service.
Sponsored by: Sun microsystems

Schedule & Evaluation

heig-vd
Haute Ecole d'Ingénierie et de Gestion du Canton de Vaud

4w	**Foundations** Java IO Network programming with the Socket API	1 grade : programming exercise in class *(25.03.2013)*
6w	**HTTP** Protocol + Client/Server Programming Infrastructure + Performance/Load Testing	3 grades : http proxy in Java lab, infra labs 1+2, theory test (includes foundations topics)
3w	**LDAP** Data Model + Protocol + querying Tools (server, browser) + import procedure	1 grade : practical test in class *(27.05.2013)*
3w	**Web Services** Big Web Services vs RESTful Web Services Using and creating REST APIs	1 grade : REST lab

Foundations

heig-vd
Haute Ecole d'Ingénierie et de Gestion du Canton de Vaud

W1 **Introduction to Java IO**
Byte and character streams, dealing with files, the decorator pattern, custom reader/writers classes, buffered IOs

W2 **Socket API - TCP**
Client and server programming, sockets and streams, multi-threaded servers

W3 **Application-level Protocol**
How to specify your own communication protocol? Implement the client and the server.

W4 **Socket API - UDP**
Client and server programming, broadcast/multicast, service discovery protocols

This Week

heig-vd
Haute Ecole d'Ingénierie et de Gestion du Canton de Vaud

- **Monday**
  - **08:30 - 08:45** : General introduction
  - **08:45 - 09:00** : Introduction to the Java IO API
  - **09:00 - 10:00** : Exercise 1 (computing stats about text data)
    5' intro, 10' individual analysis, 15' discussion, 30' individual work
  - **10:30 - 10:50** : Java IO & the Decorator Pattern
  - **10:50 - 12:00** : Exercise 2 (custom writers & decorator pattern)
    5' intro, 10' individual analysis, 15' discussion, 40' individual work
- **Homework**
  - Read selected Java Tutorial sections (see next slide)
  - Finish exercises 1 and 2
- **Wednesday**
  - **08:30 - 08:45** : Presentation of finished exercises 1 and 2
  - **09:00 - 09:15** : Buffered IOs in Java
  - **09:15 - 10:00** : Exercise 3 (measuring the performance impact of buffered IOs in Java)

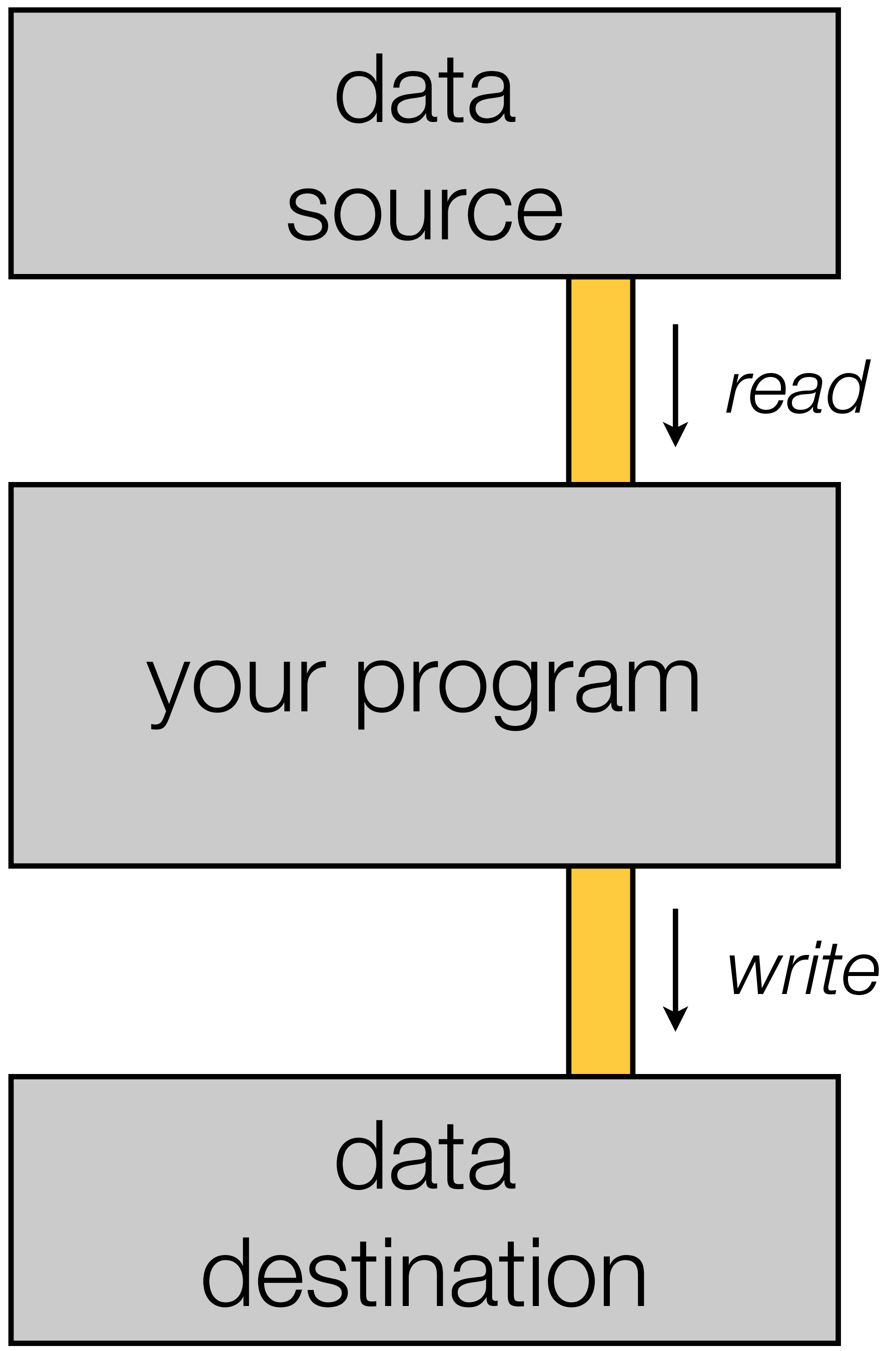
Reading for Wednesday

- http://docs.oracle.com/javase/tutorial/essential/io/streams.html
- http://docs.oracle.com/javase/tutorial/essential/io/bytestreams.html
- http://docs.oracle.com/javase/tutorial/essential/io/charstreams.html
- http://docs.oracle.com/javase/tutorial/essential/io/buffers.html

Streams

heig-vd
Haute Ecole d'Ingénierie et de Gestion du Canton de Vaud

- 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.



What can you do with IO streams?

heig-vd
Haute Ecole d'Ingénierie et de Gestion du Canton de Vaud

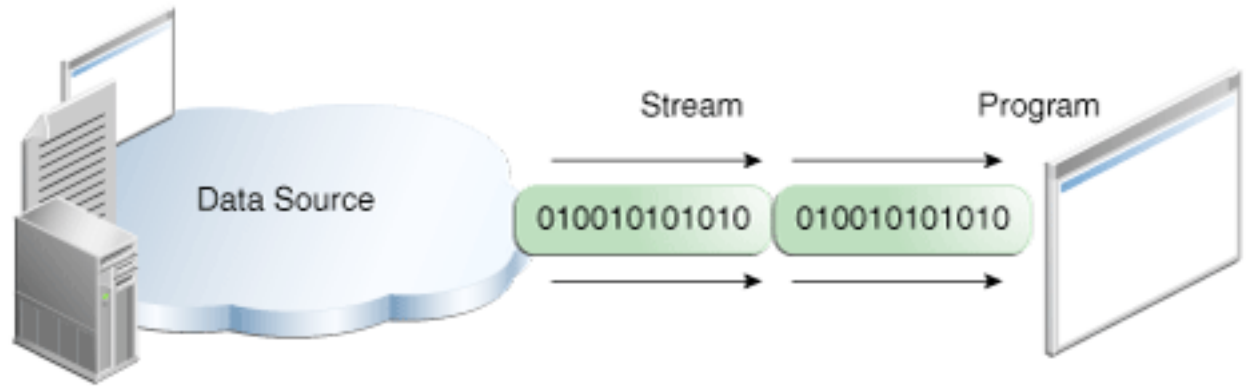
- **Read data from a file.** Think about a program that reads an XML configuration file at start-up. Think about your favorite text editor.
- **Write data to a file.** Think about a java compiler that produces .class files.
- **Receive data over the network.** Think about a web server receiving requests from clients.
- **Send data over the network.** Think about the same server sending responses to the clients.
- **Exchange data with other programs** (processes) running on the same machine.
- **Etc.**

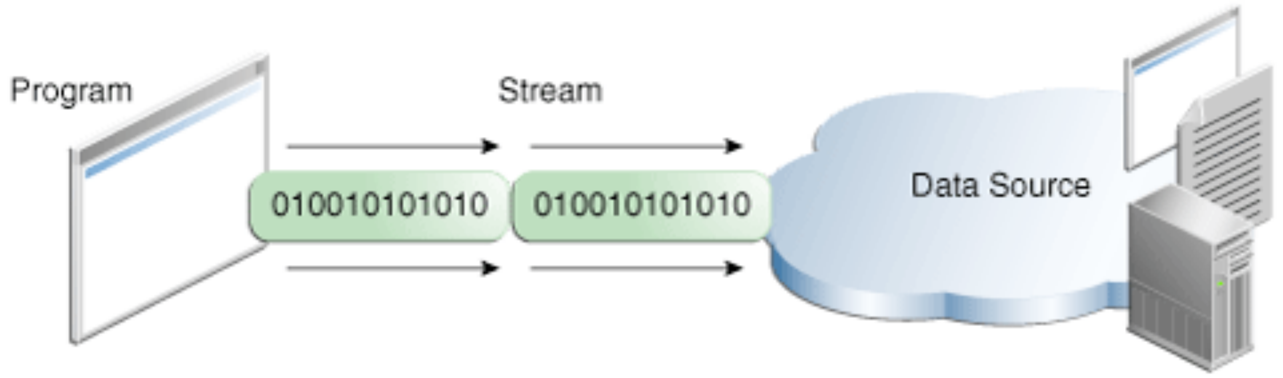
In the end, it's **always the same thing**... it's about **reading** and **writing** data!

If it's the same thing, then we want to have a **single API** for dealing with all sorts of IOs!

The IO Trail in the Java Tutorial

heig-vd
Haute Ecole d'Ingénierie et de Gestion
du Canton de Vaud



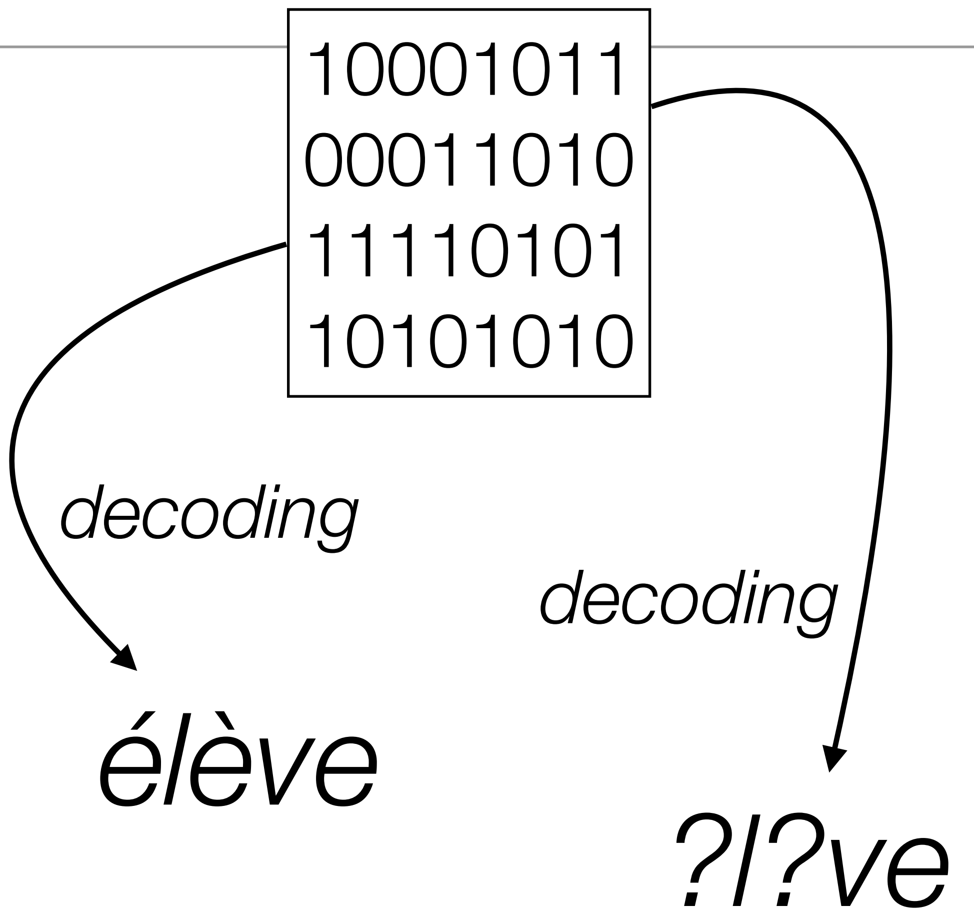


http://docs.oracle.com/javase/tutorial/essential/io/streams.html

Bytes vs. Characters

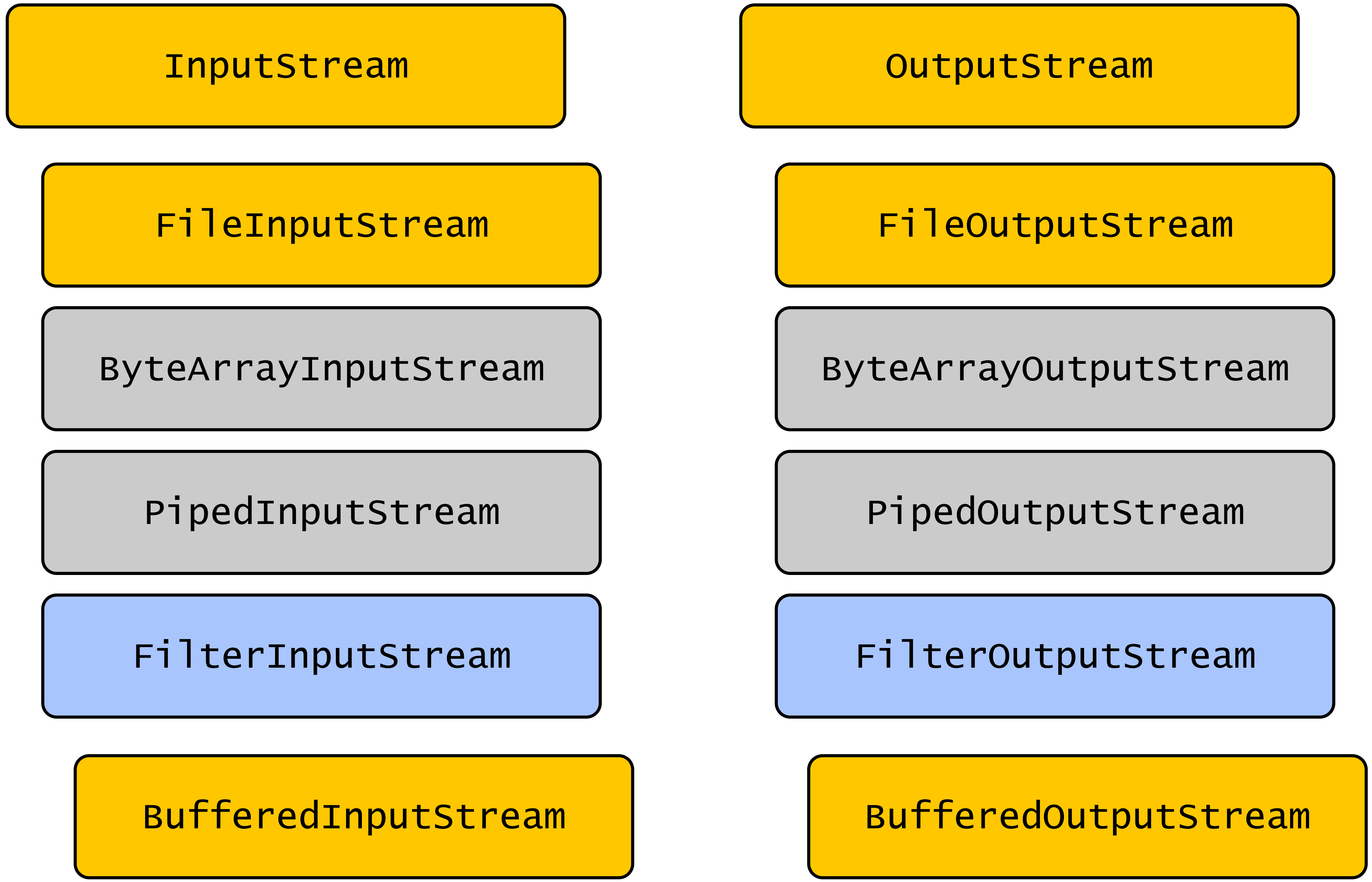
heig-vd
Haute Ecole d'Ingénierie et de Gestion du Canton de Vaud

- 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**!



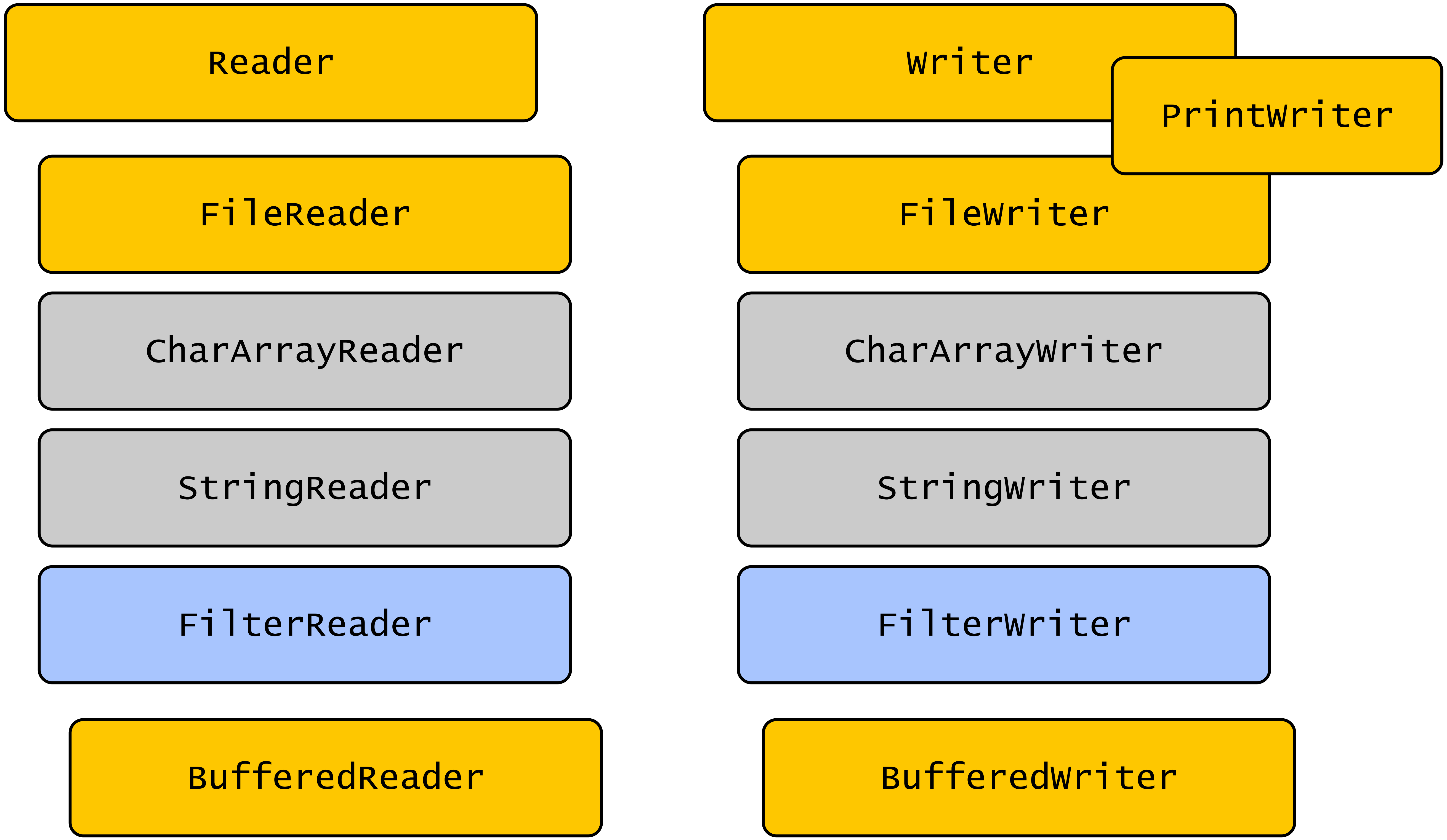
For binary data, use **inputStreams** and **outputStreams**. For text data, use **readers** and **writers**.

Key Classes for Byte Streams



Key Classes for Character Streams

heig-vd
Haute Ecole d'Ingénierie et de Gestion du Canton de Vaud



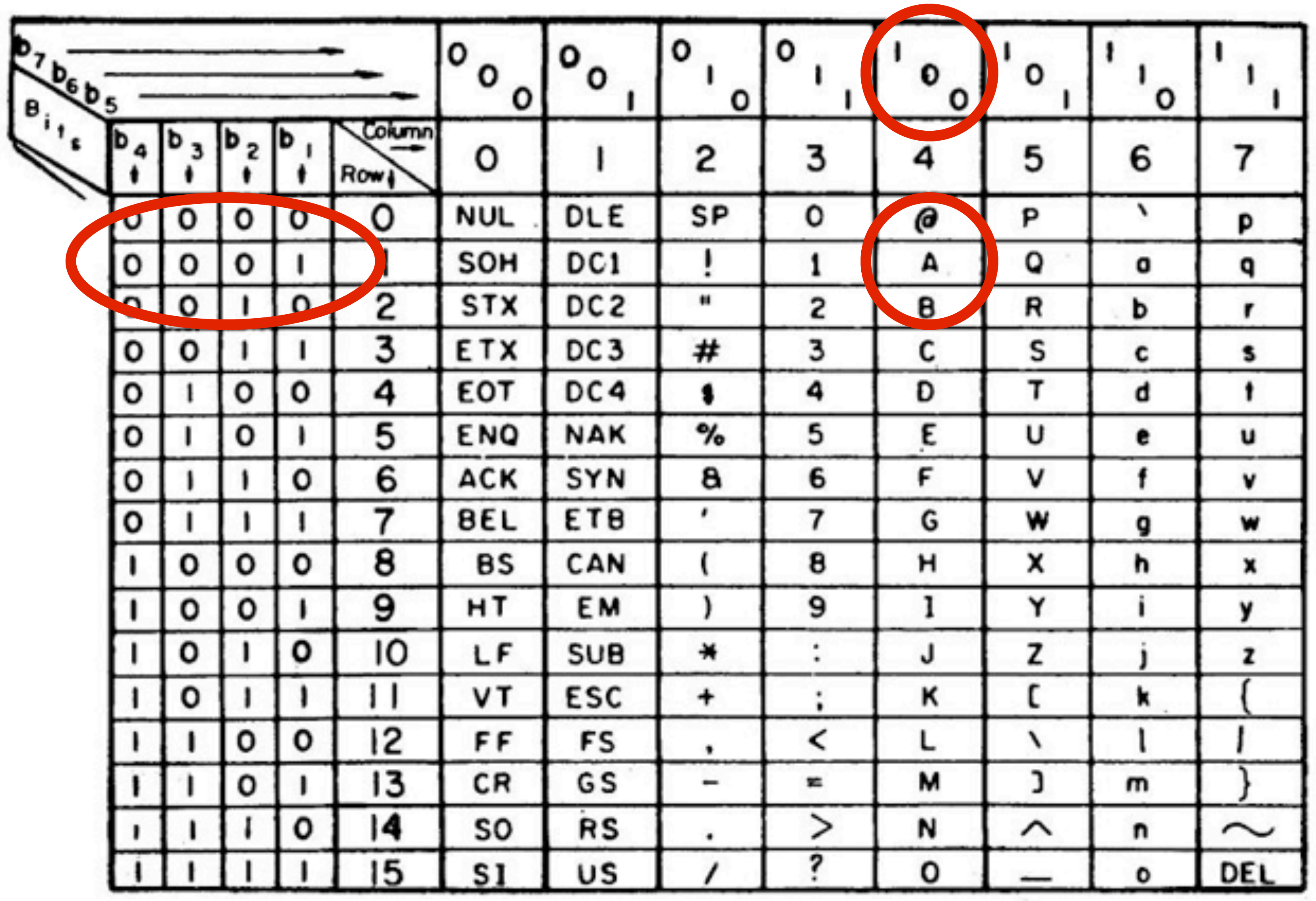
Example : the ASCII encoding

heig-vd
Haute Ecole d'Ingénierie et de Gestion du Canton de Vaud

‘ A ’ 1000001 65

64-32-16-08-04-02-01

USASCII code chart

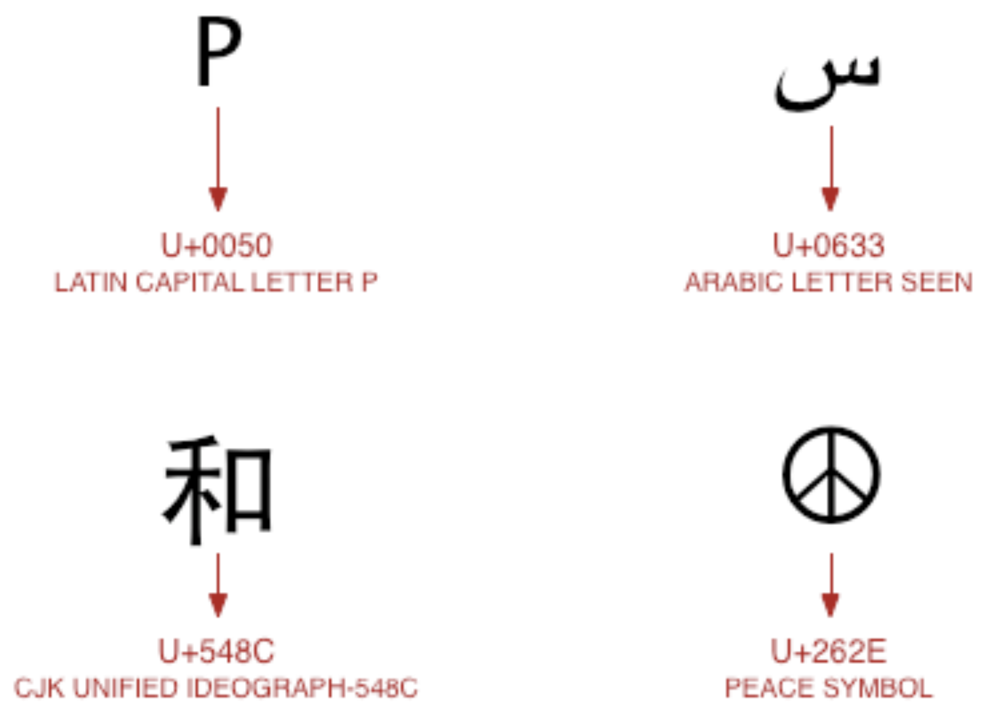


Bits b7 b6 b5 →					0 0 0	0 0 1	0 1 0	0 1 1	1 0 0	1 0 1	1 1 0	1 1 1
b4	b3	b2	b1	Column → Row ↓	0	1	2	3	4	5	6	7
0	0	0	0	0	NUL	DLE	SP	0	@	P	`	p
0	0	0	1	1	SOH	DC1	!	1	A	Q	a	q
0	0	1	0	2	STX	DC2	"	2	B	R	b	r
0	0	1	1	3	ETX	DC3	#	3	C	S	c	s
0	1	0	0	4	EOT	DC4	$	4	D	T	d	t
0	1	0	1	5	ENQ	NAK	%	5	E	U	e	u
0	1	1	0	6	ACK	SYN	&	6	F	V	f	v
0	1	1	1	7	BEL	ETB	'	7	G	W	g	w
1	0	0	0	8	BS	CAN	(	8	H	X	h	x
1	0	0	1	9	HT	EM	)	9	I	Y	i	y
1	0	1	0	10	LF	SUB	*	:	J	Z	j	z
1	0	1	1	11	VT	ESC	+	;	K	[	k	{
1	1	0	0	12	FF	FS	,	<	L	\	l	|
1	1	0	1	13	CR	GS	-	=	M	]	m	}
1	1	1	0	14	SO	RS	.	>	N	^	n	~
1	1	1	1	15	SI	US	/	?	O	_	o	DEL

Example : Unicode & UTF-8 encoding

heig-vd
Haute Ecole d'Ingénierie et de Gestion du Canton de Vaud

- **Unicode** is an industry standard for representing text in almost all world languages (e.g. japanese, hebrew, arabic, etc.).
- With Unicode, **every character is associated with a "code point"**, which is nothing else than a number. It is expressed as 'U+' followed by an **hexadecimal** value.
- For instance, 'A' has the code point **U+41** (41 is 65 in hexadecimal).
- The code points for **latin characters** have the same value as in the ASCII encoding.
- **UTF-8** is **one of the encoding systems** used to represent characters of the Unicode character set.
- UTF-8 is a **variable-length encoding system**. Some characters are encoded with 1 byte, others with 2 bytes, etc.



http://wiki.secondlife.com/wiki/Unicode_In_5_Minutes

Java & Unicode

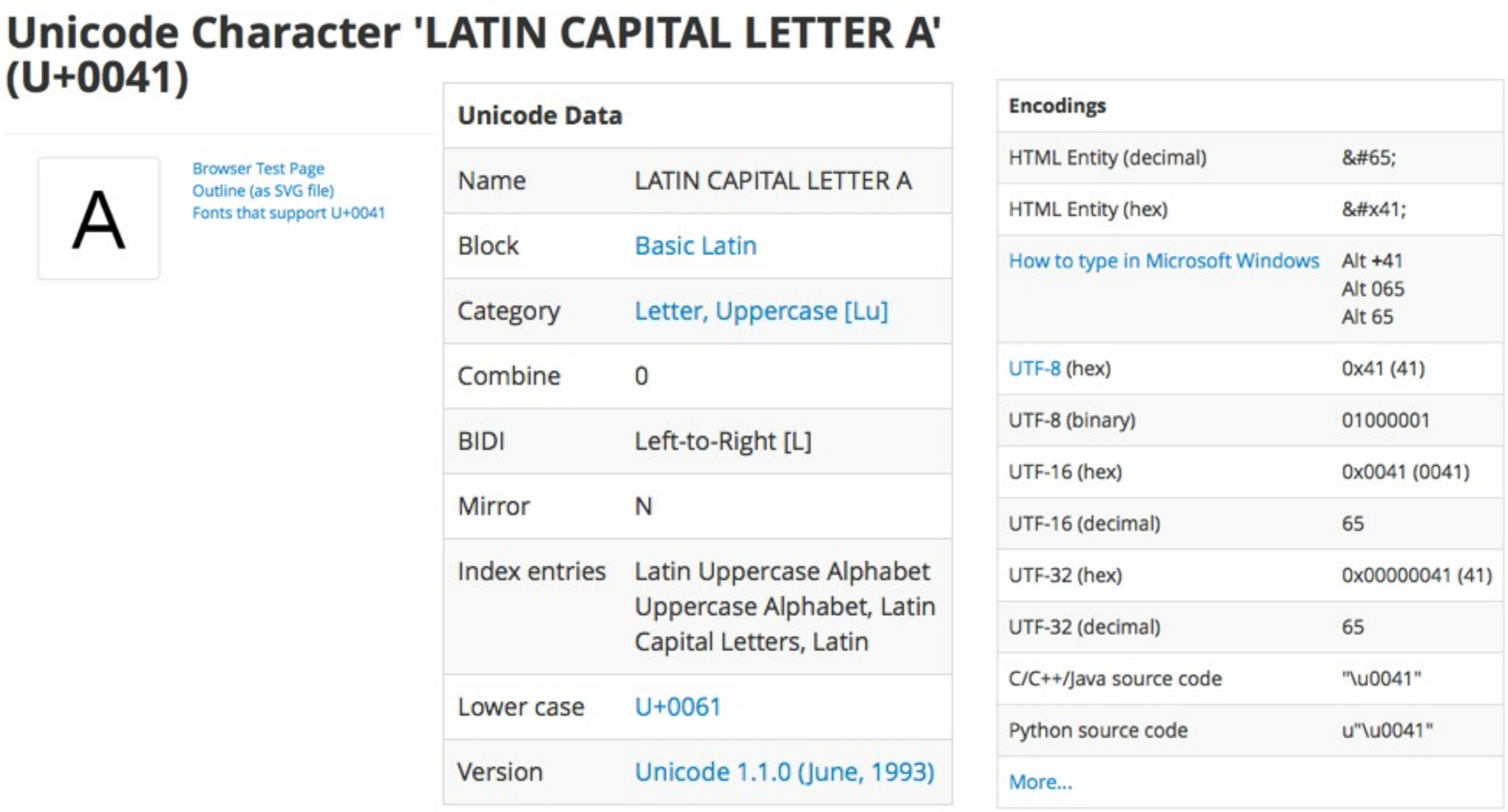
heig-vd
Haute Ecole d'Ingénierie et de Gestion du Canton de Vaud

- Java uses the **unicode character encoding system**. This means that your program can manipulate characters in different languages and alphabets.
- Every **char** variable is defined by **2 bytes**, i.e. 16 bits. The two bytes
- Think about **what happens when you read data from a source**. You will see a series of 1's and 0's. You know that these bits represent characters, but how do you know how to interpret them? The answer will depend on the source! Or more precisely, **it will depend on the encoding system used by the source**.
- **Same problem when you produce data**. You have text data in memory (char and String variables). You want to understand and control how this data is transformed in a series of bits. This is important if you want that other parties are able to read what you have produced!

Exploring the Unicode Charset

heig-vd
Haute Ecole d'Ingénierie et de Gestion
du Canton de Vaud

http://www.fileformat.info/info/unicode/char/41/index.htm



Unicode Character 'LATIN CAPITAL LETTER A' (U+0041)

A

Browser Test Page
Outline (as SVG file)
Fonts that support U+0041

Unicode Data	
Name	LATIN CAPITAL LETTER A
Block	Basic Latin
Category	Letter, Uppercase [Lu]
Combine	0
BIDI	Left-to-Right [L]
Mirror	N
Index entries	Latin Uppercase Alphabet Uppercase Alphabet, Latin Capital Letters, Latin
Lower case	U+0061
Version	Unicode 1.1.0 (June, 1993)

Encodings	
HTML Entity (decimal)	&#65;
HTML Entity (hex)	&#x41;
How to type in Microsoft Windows	Alt +41 Alt 065 Alt 65
UTF-8 (hex)	0x41 (41)
UTF-8 (binary)	01000001
UTF-16 (hex)	0x0041 (0041)
UTF-16 (decimal)	65
UTF-32 (hex)	0x00000041 (41)
UTF-32 (decimal)	65
C/C++/Java source code	"\u0041"
Python source code	u"\u0041"
More...	

Reading One Byte At The Time

```
private InputStream sourceInputStream;

public void doRead() throws IOException {
    int input = sourceInputStream.read();
    while (input != -1) {
        System.out.print((char)input);
        input = sourceInputStream.read();
    }
}
```

Let's assume that we have a stream, connected to a file, a network endpoint, etc.

We read one byte; input *has a value* ***between -1 and 255****.*

If input *equals -1, it means that we have reached the end of the stream.*

Hack! *We can* ***try*** *to convert the byte into a character. But that will break if the data is not ASCII text!* ***If we consume text, we should be using a Reader****.*

Reading Blocks of Bytes

```
InputStream sourceInputStream;

OutputStream out = new ByteArrayOutputStream();

byte[] buffer = new byte[1024];
int bytesRead;

while ((bytesRead = in.read(buffer)) != -1) {
    out.write(buffer, 0, bytesRead);
}

// We use the ByteArrayOutputStream to get the
// data at the end.
byte[] dataRead = out.toByteArray()
```

Again, we have a source input stream.

***ByteArrayOutputStream** is a special stream that writes to memory.*

*We use a memory **buffer** to read blocks of bytes.*

*We try to get bytes from the stream. It will tell us **how many were available**.*

*We can then write these bytes to the output stream. We have to be **careful**: bytesRead may not be equal to buffer.length!!*

Exercise 1

- Write a java program that **reads a text file** and produces **computes basic statistics** about its content.
- **Functional requirements**
    - It should be possible to **call the program from the command line** and to pass the **file name** as an argument.
    - When running the program, the user should see at least the following information: 1) **number of characters**, 2) **number of uppercase characters**, 3) **number of lowercase characters**, 4) **number of vowels**, 5) **number of consonants**, 6) **number of digits**.
    - When running the program, the user should also see a **table, showing every character appearing in the text and its number of occurrences**.
- **Design requirements**
    - Apply object-oriented principles to have a modular and extensible solution.

Exercise 1 - procedure

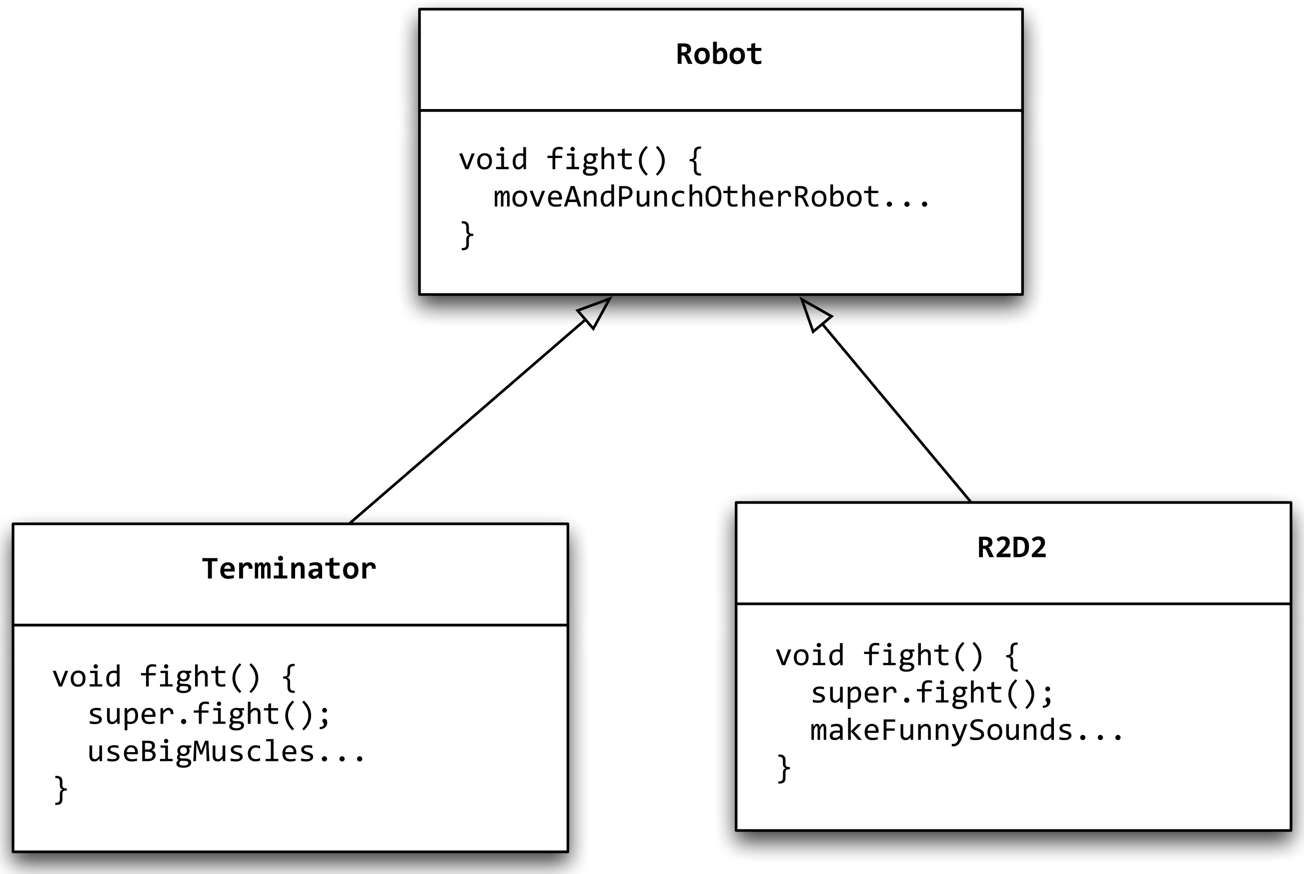
- **Step 1 : analysis & design (10’ individual work + 10’ review)**
  - How do you **decompose the problem** in sub-problems?
  - What **tasks and operations** do you need to be able to do in order to solve these sub-problems? How can **Java APIs** help you?
  - How do you **structure** the overall program (interfaces, classes & methods)?
- **Step 2 : implementation (20’ individual work + 10’ review)**

The Decorator Design Pattern

heig-vd
Haute Ecole d'Ingénierie et de Gestion du Canton de Vaud

- The **decorator design pattern** is a solution that is often used when creating object-oriented models.
- It makes it possible to **add behavior to an existing class**, without modifying the code of this class. In other words, it makes it possible to decorate an existing class with additional behaviors.
- The design pattern also makes it possible to **define a collection of decorators**, and to **combine** them in arbitrary ways at runtime. In other words, it is possible to decorate a class with several behaviors.

Example : Decorated Robots

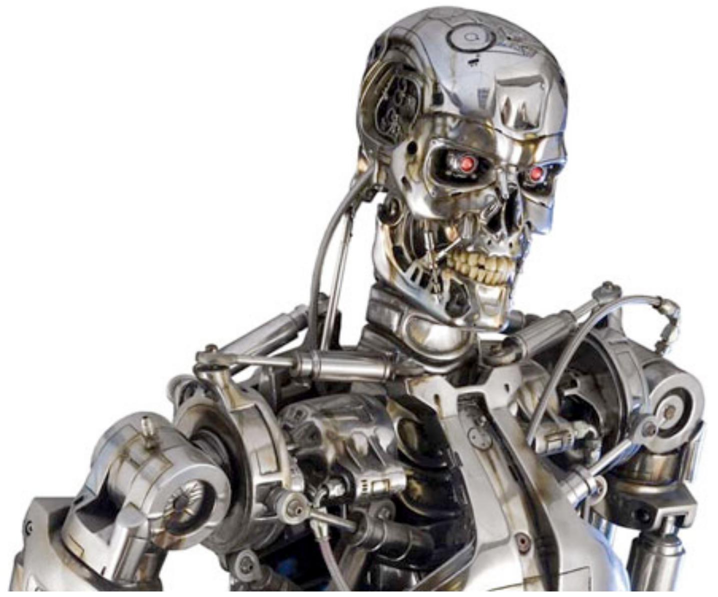


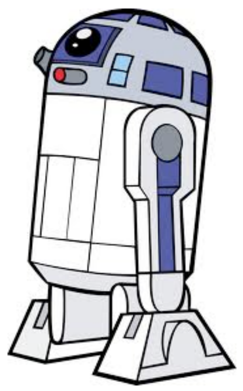
```
Robot
void fight() {
  moveAndPunchOtherRobot...
}
```

```
Terminator
void fight() {
  super.fight();
  useBigMuscles...
}
```

```
R2D2
void fight() {
  super.fight();
  makeFunnySounds...
}
```

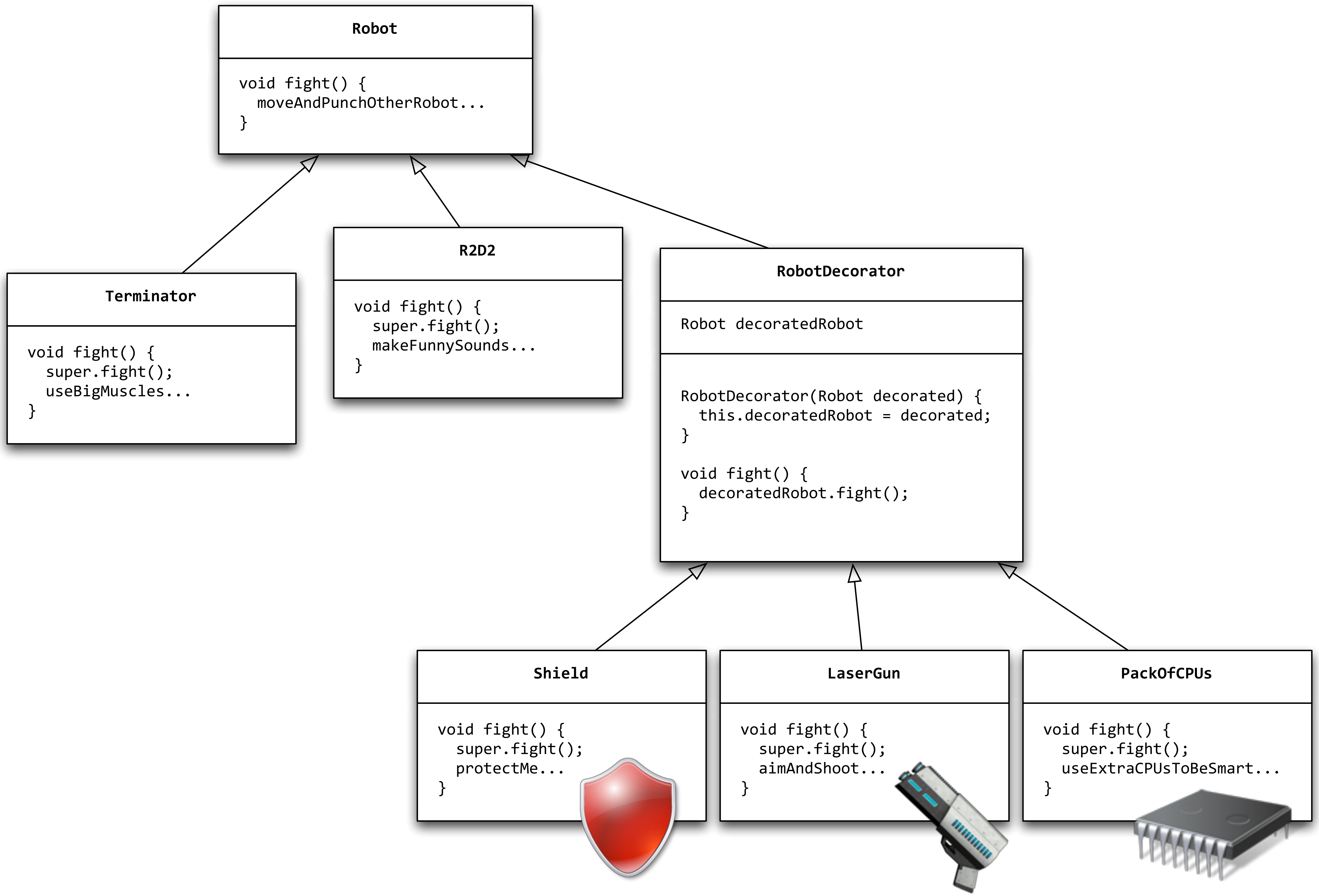
```
Robot bigOne = new Terminator();
Robot smallOne = new R2D2();

bigOne.fight();
smallOne.fight();

--
bigOne   > useBigMuscles
smallOne > makeFunnySounds

bigOne wins.
```





Example : Decorated Robots

heig-vd
Haute Ecole d'Ingénierie et de Gestion du Canton de Vaud



Robot:
```
void fight() {
  moveAndPunchOtherRobot...
}
```

Terminator:
```
void fight() {
  super.fight();
  useBigMuscles...
}
```

R2D2:
```
void fight() {
  super.fight();
  makeFunnySounds...
}
```

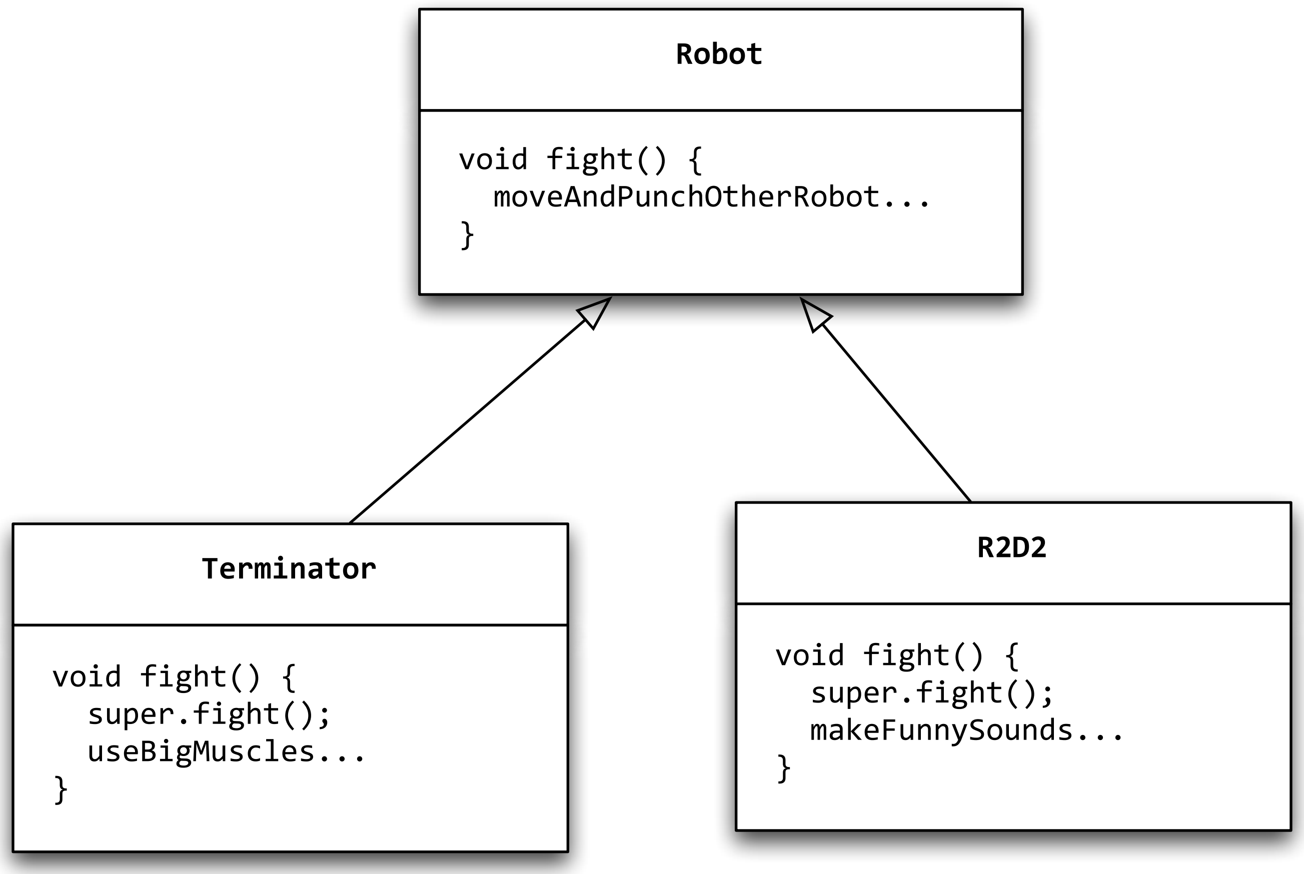
RobotDecorator:
```
Robot decoratedRobot

RobotDecorator(Robot decorated) {
  this.decoratedRobot = decorated;
}

void fight() {
  decoratedRobot.fight();
}
```

Shield:
```
void fight() {
  super.fight();
  protectMe...
}
```

LaserGun:
```
void fight() {
  super.fight();
  aimAndShoot...
}
```

PackOfCPUs:
```
void fight() {
  super.fight();
  useExtraCPUsToBeSmart...
}
```

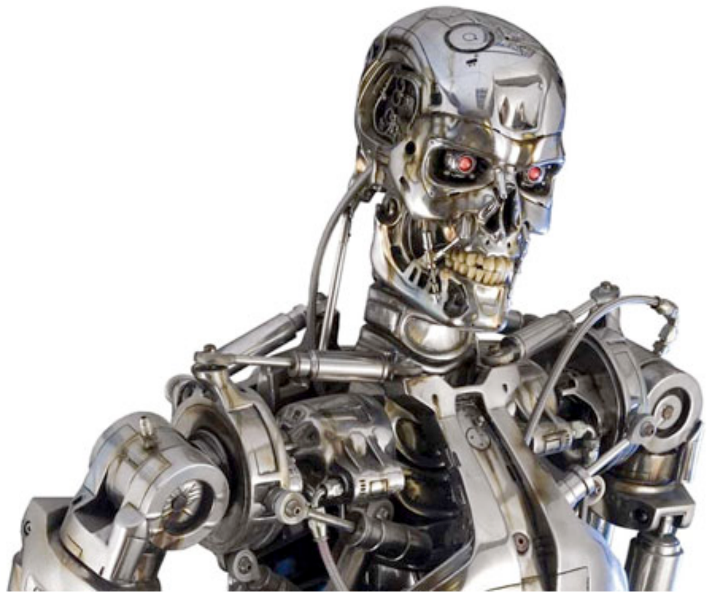
Example : Decorated Robots



```
Robot
void fight() {
  moveAndPunchOtherRobot...
}
```

```
Terminator
void fight() {
  super.fight();
  useBigMuscles...
}
```

```
R2D2
void fight() {
  super.fight();
  makeFunnySounds...
}
```



```
Robot bigOne = new Terminator();
Robot decoratedSmallOne =
  new Shield(
    new LaserGun(
      new PackOfCPU(
        new R2D2()
      )
    )
  );

bigOne.fight();
decoratedSmallOne();

--
bigOne   > useBigMuscles
smallOne > makeFunnySounds,
useExtraCPUsToBeSmart, aimAndShoot,
protectMe

decoratedSmallOne wins.
```

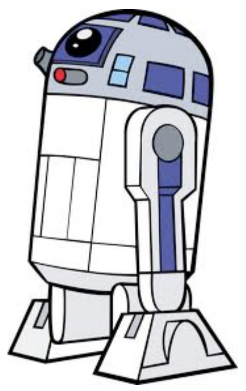
Decorators in the Java IO API

- The **standard Java classes** used to interact with **byte and character streams** follow the structure of the decorator design pattern.
- There are 4 main base classes: **InputStream**, **OutputStream**, **Reader** and **Writer**.
- There are concrete subclasses that implement the logic relative to specific data sources and destinations, for instance: **FileInputStream**, **ByteArrayOutputStream**, **StringWriter**, etc.
- There are 4 decorator base classes: **FilterInputStream, FilterOutputStream**, **FilterReader**, **FilterWriter**.
- There are several specific decorators: **BufferedInputStream**, **CipherInputStream**, **LineNumberingInputStream**, etc.
- **You can implement your own decorators by sub-classing the Filter classes.**

Example (1)

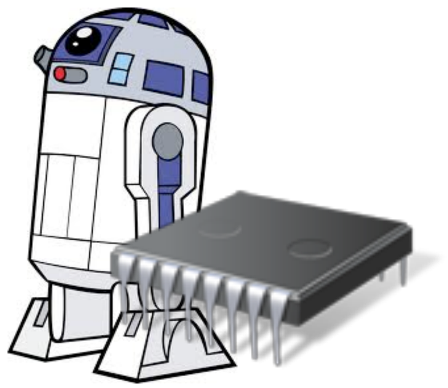
heig-vd
Haute Ecole d'Ingénierie et de Gestion
du Canton de Vaud

```
// Firstly, we create an input stream to read an image file
FileInputStream fis = new FileInputStream(“data.png”);

// We call read on file input stream, which will interact with
the file system and give some bytes
bis.read();
```



Example (2)

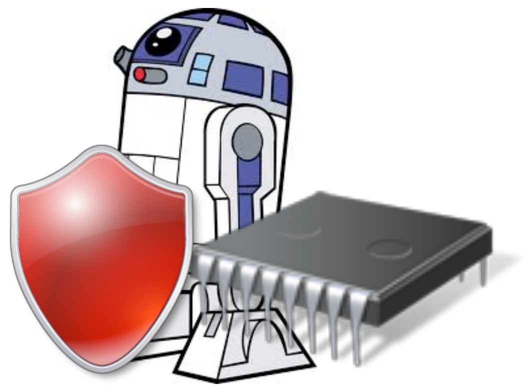
```
// Firstly, we create an input stream to read an image file
FileInputStream fis = new FileInputStream(“data.png”);

// Then, we decorate it with a class that makes reading more
efficient (by buffering data)
BufferedInputStream bis = new BufferedInputStream(fis);

// We call read on the decorator. It will implement its own
logic, but will ultimately delegate a lot of work to the
decorated file input stream.
bis.read();
```



Example (3)

heig-vd
Haute Ecole d'Ingénierie et de Gestion
du Canton de Vaud

```
// Firstly, we create an input stream to read a text file
FileInputStream fis = new FileInputStream(“japanese.txt”);

// Then, we decorate it with a class that will transform bytes
in characters, using the SJIS encoding
InputStreamReader isr = new InputStreamReader(fis, “SJIS”);

// Then, we decorate it with a class that makes reading more
efficient (by buffering data)
BufferedReader br = new BufferedReader(isr);

// We call read on the 2nd decorator. It will execute its
logic, then call the 1st decorator. The 1st decorator will
execute its logic, and then delegate work to the base class.
br.read();
```



Exercise 2

- Write a java program that **reads a text file**, and **applies various transformations** on the content and **writes the result in another text file**.
- **Functional requirements**
  - It should be possible to **call the program from the command line** and to pass the **file name** as an argument. The program will write results to a file that has the same name with the **“.out” extension** (e.g. running the program on “data.txt” will produce “data.txt.out”.
  - The **first transformation** consists of passing all characters to uppercase. The **second transformation** consists of replacing all ‘a’ and ‘A’ characters with the ‘@’ character. The **third transformation** consists of inserting the number of characters at the end of each line (e.g. **Hello -> Hello [5]**)
- **Design requirements**
  - Apply the decorator pattern, by extending the FilterReader and/or the FilterWriter classes.