Reading and Writing Bytes

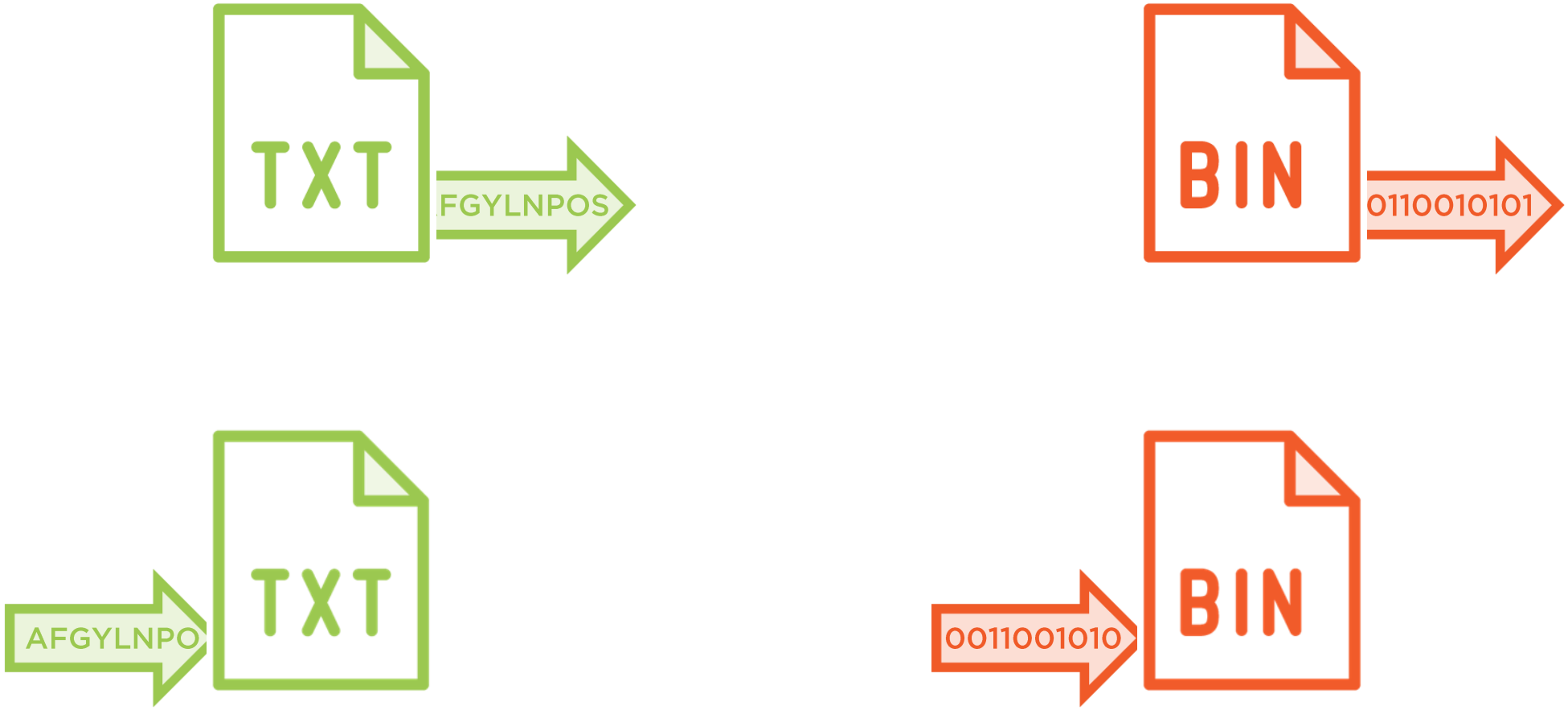
José Paumard
PHD, JAVA CHAMPION, JAVA ROCK STAR

@JosePaumard https://github.com/JosePaumard

Agenda

- Concept of binary streams
- How to write and read bytes
- On disks and in-memory
- Compressed streams

Introducing Binary Streams


TXT
FGYLNPOS
BIN
0110010101
AFGYLNPO
TXT
0011001010
BIN

First base class:

1) InputStream

Basic operations:

- Reading a single byte
- Reading an array of bytes
- Marking and resetting a given position
- Skipping bytes
- Asking for available bytes

And it can be closed

Second base class:

2) OutputStream

Basic operations:

- Writing a single byte

- Writing an array of bytes

And it can be flushed and closed

```
InputStream is = ...; // we will see how to create one later

int nextByte = is.read();

while (nextByte != -1) {
    // do something with nextByte
    nextByte = is.read();
}
```

The returned byte is converted to an `int`

```java
InputStream is = ...;
byte[] is = new byte[1024];


int number = is.read(buffer);

while (number != -1) {
    // do something with buffer
    number = is.read(buffer);
}
```

When there is no more bytes to read, the `read()` call returns -1

Be careful, `number` is the number of bytes that have been read

It can be lesser than 1024

```
OutputStream os = ...; // we will see how to create one later

os.write(0xCA); // takes an int, converted to a byte
```

The write method does not return anything

Dealing with Exceptions

All these methods declare checked exceptions, just as the readers and writers do

The patterns to deal with them are the same

Including the (preferred) try-with-resources pattern to open an input or output stream

Creating InputStream and OutputStream

Those two classes are abstract classes

Extended by 2 categories of concrete classes

1) classes for a certain type of medium

- Disk: FileInputStream / FileOutputStream

- In-memory: ByteArrayInputStream / ByteArrayOutputStream

- Network: SocketInputStream / SocketOutputStream

2) classes that add behavior

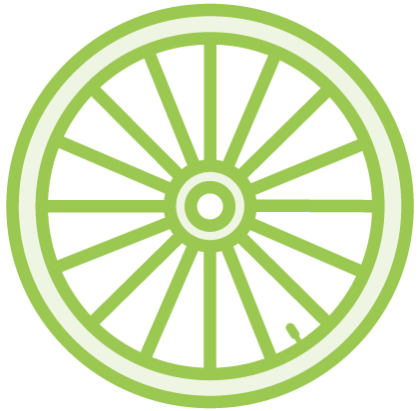
- BufferedOutputStream

- GzipOutputStream

- ZipOutputStream

And ObjectOutputStream, that we will see in the next module

Dealing with Primitive Types

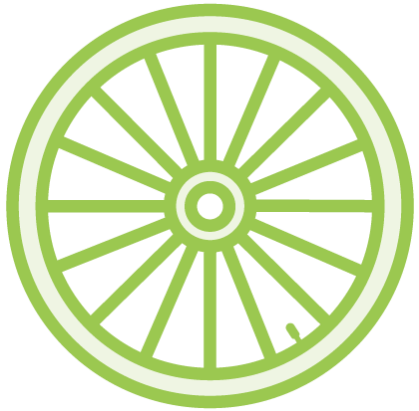


Writing and reading bytes is also about

- writing primitive types: int, float, etc...

- and objects: Serialization

There are classes for that!



DataInputStream & DataOutputStream

Those are the classes dedicated to the reading and writing of primitive types

```
OutputStream os = ...;
DataOutputStream dos = new DataOutputStream(os);

dos.writeInt(10);
dos.writeDouble(3.14d);
dos.writeChar('H');
dos.writeUTF("Hello");
```

There is a collection of `writeXXX()` methods

One for each primitive type

And a `writeUTF()` method to write strings

```
InputStream is = ...;
DataInputStream dis = new DataInputStream(is);

int i    = dis.readInt();
double d = dis.readDouble();
char c   = dis.readChar();
String s = dis.readUTF();
```

And the same goes for the `readXXX()` methods

Again, one for each primitive type

Plus a `readUTF()` method to read strings

In-Memory Streams

Two classes for reading and writing binary data in memory:

- ByteArrayInputStream

- ByteArrayOutputStream

```java
ByteArrayOutputStream bos = new ByteArrayOutputStream();
DataOutputStream dos = new DataOutputStream(bos);

dos.writeInt(10);
dos.writeDouble(3.14d);
dos.writeChar('H');

int size = dos.size();
```

The stream is connected to an array of byte

We can use them as a regular stream

And get the number of bytes written in the array

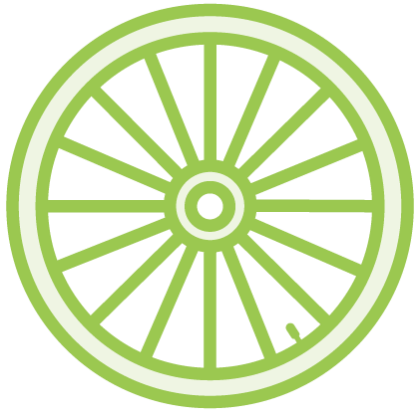
```
byte[] bytes = bos.toByteArray();
ByteArrayInputStream bis = new ByteArrayOutputStream(bytes);

DataInputStream dis = new DataInputStream(bis);

int i    = dis.readInt();
double d = dis.readDouble();
char c   = dis.readChar();
```

The corresponding input stream is built on an array of bytes

And can be used as any regular stream



These in-memory streams can be used to create complex structures in memory

Then flush them to the disk or a network

It does not give the same result as the buffered versions of the input / output streams

Compressed Streams

Two types of compressed streams:

- Gzip format: allows for one file

- Zip format: allows for several files

```
OutputStream os = Files.newOutputStream(
    Paths.get("files/file.bin"),
    StandardOpenOptions.CREATE
);


DataOutputStream dos = new DataOutputStream(os);

IntStream.range(0, 100).forEach(dos::writeInt);
```

This code writes 100 ints in a regular file

```
OutputStream os = Files.newOutputStream(
    Paths.get("files/file.bin.gz"),
    StandardOpenOptions.CREATE
);

GZIPOutputStream gzos = new GZIPOutputStream(os)
DataOutputStream dos = new DataOutputStream(gzos);

IntStream.range(0, 100).forEach(dos::writeInt);
```

This code writes 100 ints in a compressed file

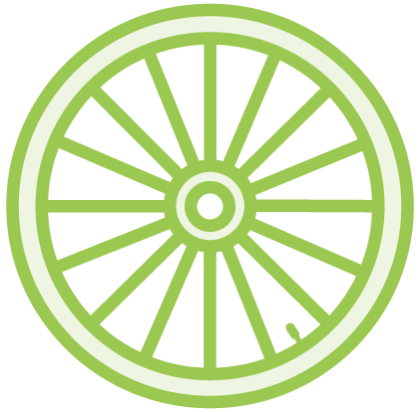
```
InputStream is = Files.newInputStream(
    Paths.get("files/file.bin.gz"),
    StandardOpenOptions.READ
);

GZIPInputStream gzis = new GZIPInputStream(is);
DataInputStream dis = new DataInputStream(gzis);

int i = dis.readInt();
```

Reading is also built on the use of the decorator pattern

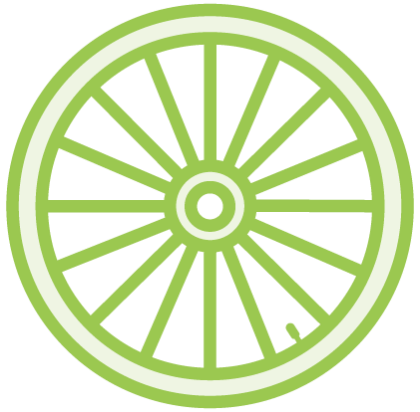
Transparent for the user



There are two limitations on the GZIP format:

- it can compress one file at a time

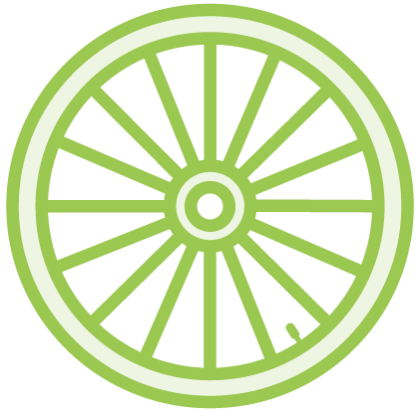
- there may be a limit on the file size



The ZIP format is more complex

- we can create directories and files

- and add content on the fly



Creating GZIP / ZIP streams in memory

And flush them to the disk or network once the compression is done

Can lead to much better performance

```
OutputStream os = Files.newOutputStream(
    Paths.get("files/archive.zip"),
    StandardOpenOptions.CREATE
);

ZipOutputStream zos = new ZipOutputStream(os);
DataOutputStream dos = new DataOutputStream(zos);
```

For the ZIP file: we need a ZIP Stream

```
ZipEntry dirEntry = new ZipEntry("files/");
zos.putNextEntry(dirEntry);

ZipEntry fileEntry = new ZipEntry("files/data.bin");
zos.putNextEntry(fileEntry);
```

For the ZIP file: we need a ZIP Stream

The we can create entries in the ZIP Stream

Entries can be directories or files

```
ZipEntry fileEntry = new ZipEntry("files/data.bin");
zos.putNextEntry(fileEntry);

dos.writeInt(1);
dos.writeInt(2);

zos.closeEntry();

zos.close()
```

Once an entry has been created, we can write data inside this entry

When we are done, we can close the entry, and create another one

And close the stream when our archive is written

```
InputStream is = Files.newInputStream(
    Paths.get("files/archive.zip"),
    StandardOpenOptions.READ
);

ZipInputStream zis = new ZipInputStream(is);
DataInputStream dis = new DataInputStream(zis);
```

Reading works in the same way

```
ZipEntry nextEntry = zis.nextEntry();
while (nextEntry != null) {

   if (nextEntry.isDirectory()) {
      // deal with directories
   }

   nextEntry = zis.nextEntry();
}
```

We need to test for the entries in the ZIP file

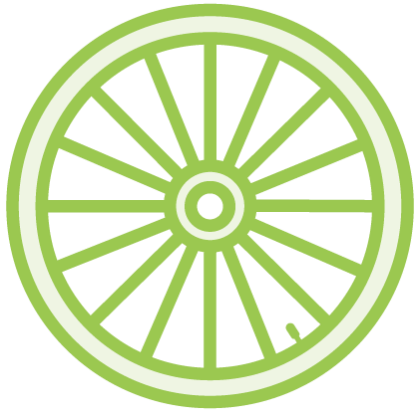
Check if the entry is a directory of a file

```
ZipEntry nextEntry = zis.nextEntry();
while (nextEntry != null) {
    ...
    else {
        int i = dis.readInt();
    }

    nextEntry = zis.nextEntry();
}
```

We need to test for the entries in the ZIP file

And read the content of the files



There are also two classes for the JAR format:

- JarInputStream

- JarOutputStream

They both extends the ZIP classes

With methods to handle the manifest file

And a Manifest class

Demo

Let us see some code!

Let us create input and output streams on examples

And see how the GZIP and ZIP formats work

Module Wrap Up

What did you learn?

Binary Streams!

How to write and read binary information

How to deal with compressed files and archive