Java Fundamentals: NIO and NIO2

CREATING CHANNELS WITH JAVA NIO

José Paumard
PHD, JAVA CHAMPION, JAVA ROCK STAR

@JosePaumard https://github.com/JosePaumard

Accessing Files & Network

Java I/O

Java 1 (1996)

Java NIO

Java 4 (2002)

Java NIO 2

Java 7 (2011)

Accessing Files & Network

Java I/O

Java 1 (1996)

Java NIO

Java 4 (2002)

Java NIO 2

Java 7 (2011)

Accessing Files & Network

Java Fundamentals:
Input / Output

Java I/O

Java 1 (1996)

Java NIO

Java 4 (2002)

Java NIO 2

Java 7 (2011)

Java 4 added Java NIO to Java I/O

And Java 7 added NIO2

- NIO stands for Non-Blocking IO

- it deals with buffers and channels

- supports asynchronous operations

- NIO may be more efficient than I/O

Then NIO2 brought some more:

- file systems and events

- directory structure exploration API

This is a Java course

Fair knowledge of the language and its main API

The Collection framework

The Java I/O API: File, Path, Exceptions

Some knowledge of what a file system is

A little bit of networking, especially for the second module

And some basic concurrency

Agenda

Java NIO:

- Using channels and buffers

- Setting selectors and asynchronous I/O operations

Java NIO2:

- Using file systems

- Visiting directory trees

- Listening to file system events

Agenda

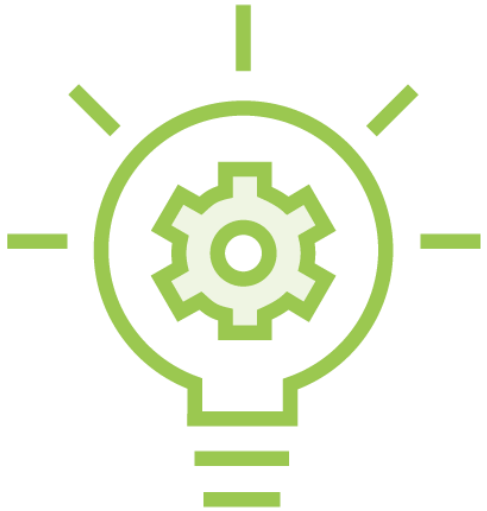
Buffers and Channels

- Channels: File, Socket, In-memory
- Buffers
- Examples: read / write operations
- Using multiple buffers
- Using charsets

Introducing Java NIO

Why a new I/O API?

- Java I/O writes one byte / char at a time
- readers are for reading only
- buffering occurs in the JVM heap memory
- handling of charsets is not great
- all the operations are synchronous



NIO provides:

- bulk access to raw bytes
- bidirectional channels
- off-heap buffering
- proper support for charsets
- support for asynchronous operations

Introducing Buffers, Channels, and Selectors

Java NIO introduces three new concepts:

- Buffer: where the data resides
- Channel: where the data comes from
- Selector for asynchronous operations

A write operation takes data from a buffer and writes it to a channel

A read operation reads data from a channel and writes it into a buffer

Understanding Channels

File Channels

Channel is an interface, implemented by:

1) FileChannel accesses to a file

- it has a cursor

- multiple reads and writes

- it is thread safe

Socket Channels

Channel is an interface, implemented by:

2) DatagramChannel accesses to a socket

- it supports multicast

- supports multiple non-concurrent reads and writes

Socket Channels

Channel is an interface, implemented by:

3) SocketChannel and ServerSocketChannel access to a socket

- it supports asynchronous operations

- supports multiple non-concurrent reads and writes

FileChannel, SocketChannel and ServerSocketChannel are abstract classes

Concrete implementations are hidden

Channels are created with factory methods

In-memory File Channels

A FileChannel can be mapped to a memory array for direct access

It is built on OS features for fast access

Can be dangerous, a single write in this array can trigger a modification of the file on the disk

In-Memory File Channels

There are 3 modes for this mapping:

- READ_ONLY: the mapped file cannot be modified

- READ_WRITE: it can...

- PRIVATE: modifications are local to this channel and will not be propagated to the disk

Understanding Buffers

Types of Buffers

Buffer is an abstract class, extended by typed buffers

- ByteBuffer,

- CharBuffer, ...

And then extended by concrete implementations

Concrete implementations are hidden

Buffers are created with factory methods

A buffer is an in-memory structure

May be stored in the off-heap space of the JVM

A buffer object has 3 properties:

- a capacity = the size of the backing array

- a current position

- a limit

A buffer keeps track of the available space

A buffer can hold a single mark

Marking a buffer does two things:

- it sets the mark to the current position

- it returns this and to chain calls

A buffer supports 4 operations

- rewind: clears the mark and sets the current position to 0

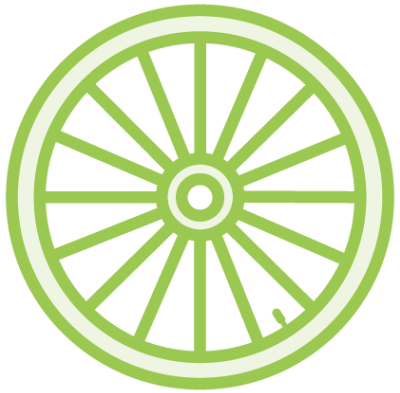
- reset: sets the current position to the previously set mark

- flip: sets the limit to the current position and rewinds the buffer

- clear: clears the buffer

These operations return this and can be chained

Writing Content to a File



We need a buffer to write our ints in

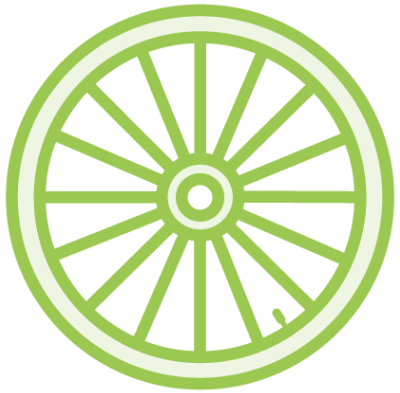
Then a channel to write to a file

A channel can only write from a byte buffer

1) create a byte buffer

2) use the putXXX() method to write data

3) have our file channel to write
the byte buffer to a file



To write Strings:

1) convert the byte buffer to a char buffer

2) use the put(String) method

```
ByteBuffer byteBuffer = ByteBuffer.allocate(1024);

byteBuffer.putInt(10);

FileChannel fileChannel = FileChannel.open(
      Paths.get("files/ints.bin"), CREATE, WRITE);
fileChannel.write(byteBuffer);
fileChannel.close();
```

First we create a byte buffer

And we write the data we need in the buffer

Then we create the channel to a file to write the content of the buffer

The write operation writes all the buffer

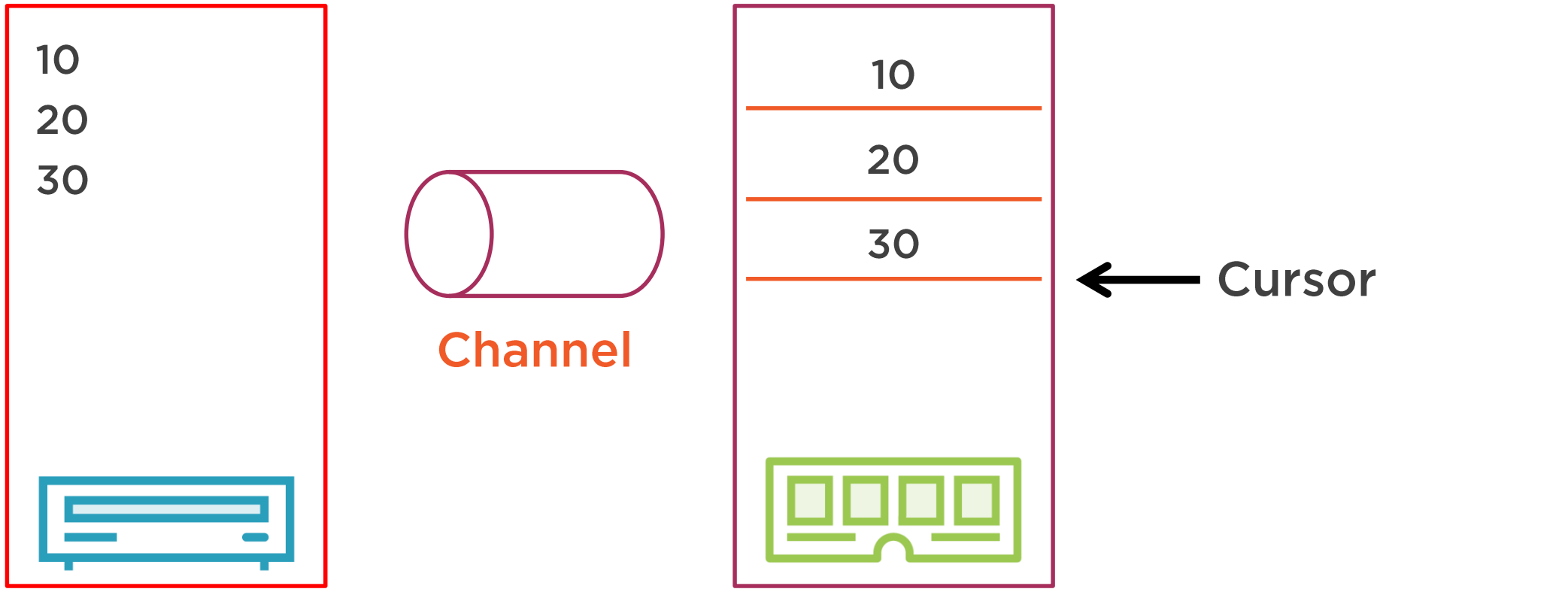
Reading from a File

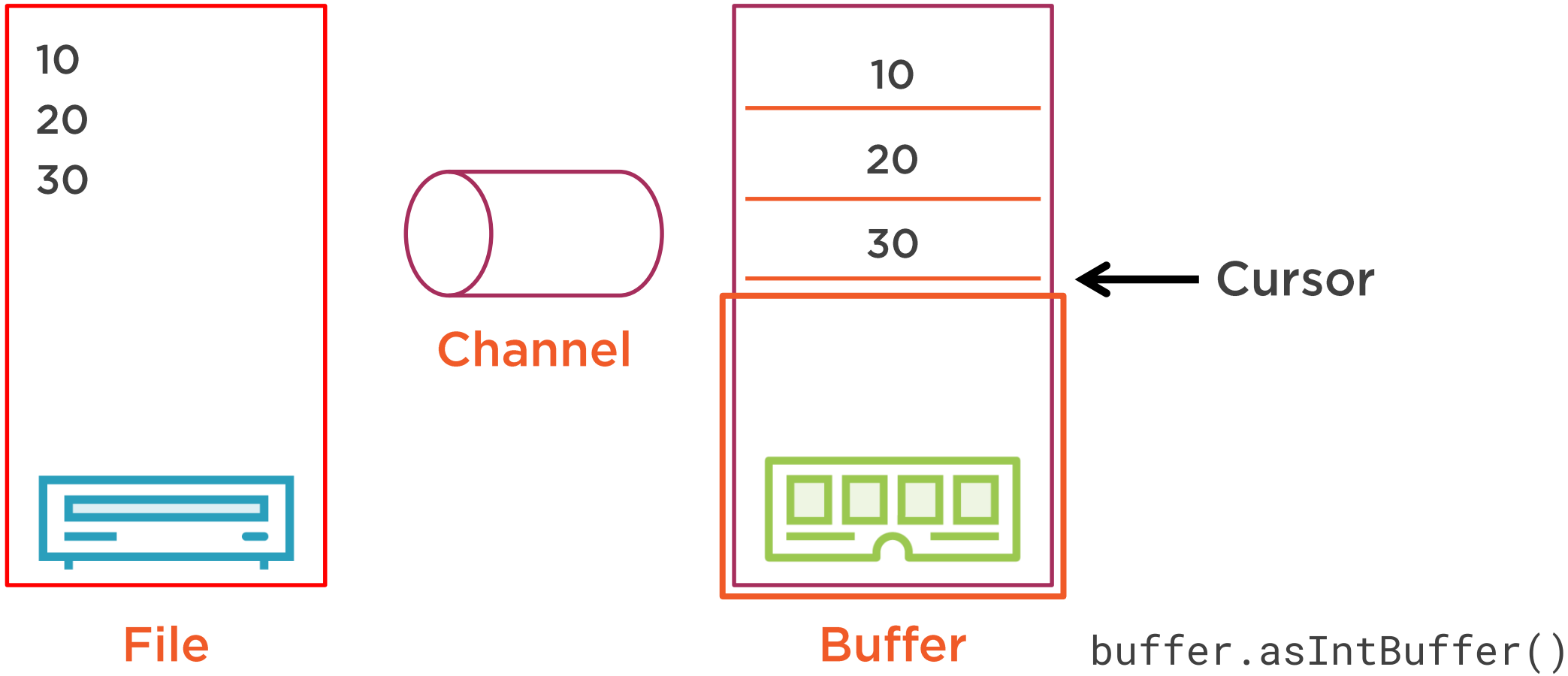
In order to properly read a file, we need to understand how buffers work

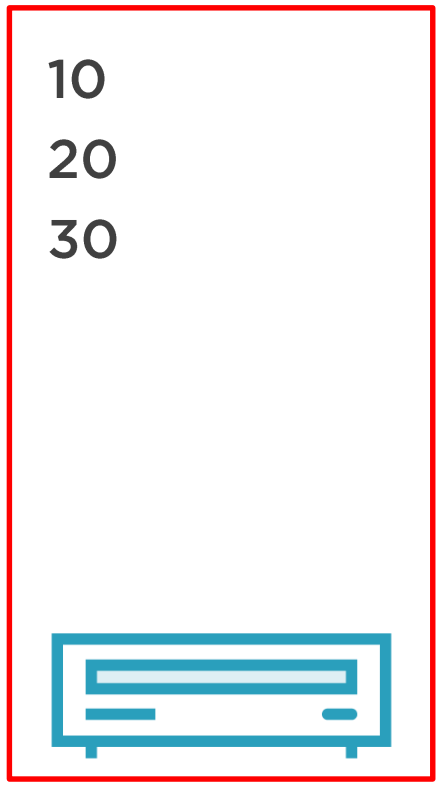
- we create a channel on the given file

- then read the content of the file in the buffer

- then read the buffer and translate its content

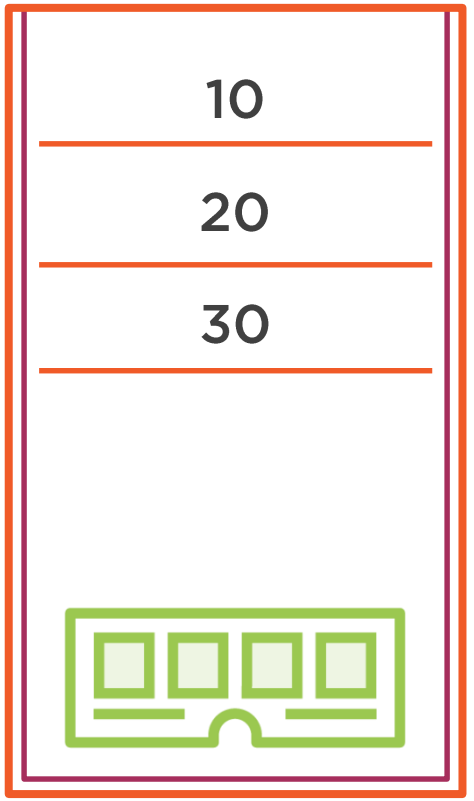

10
20
30
Channel
10
20
30
Cursor
File
Buffer
buffer.asIntBuffer()


10
20
30
Channel
10
20
30
Cursor
File
Buffer
buffer.asIntBuffer()



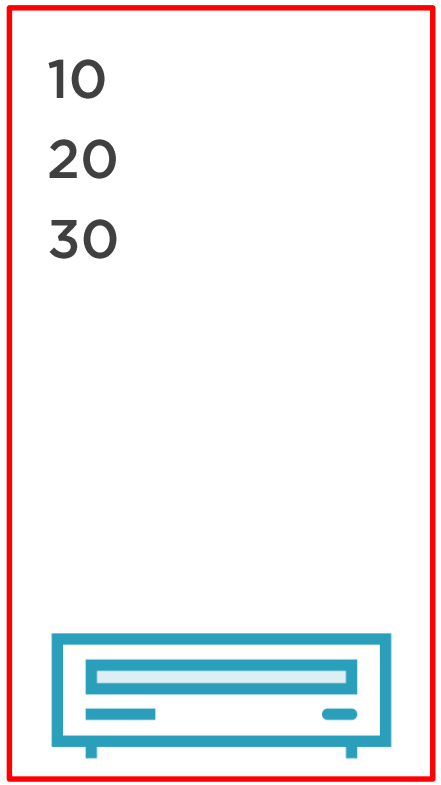
File

Channel



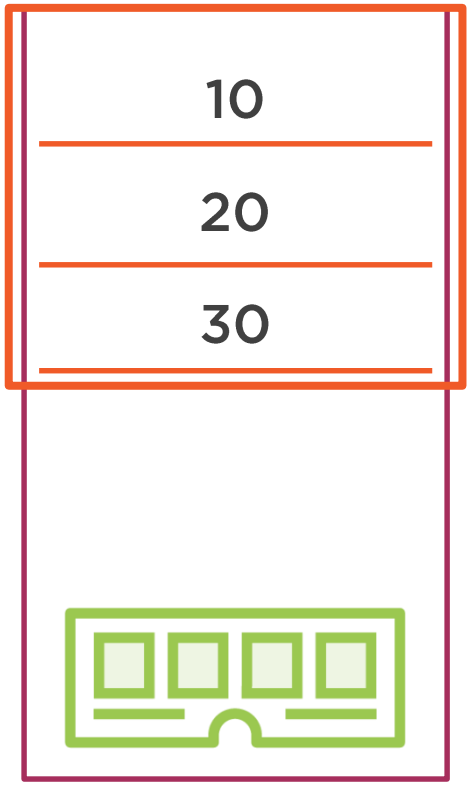
Buffer

```
buffer.rewind()
buffer.asIntBuffer()
```



File

Channel



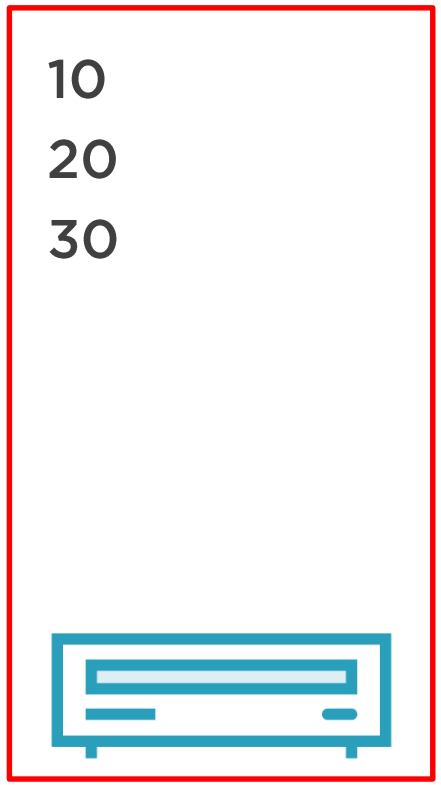
Cursor

Limit

Buffer

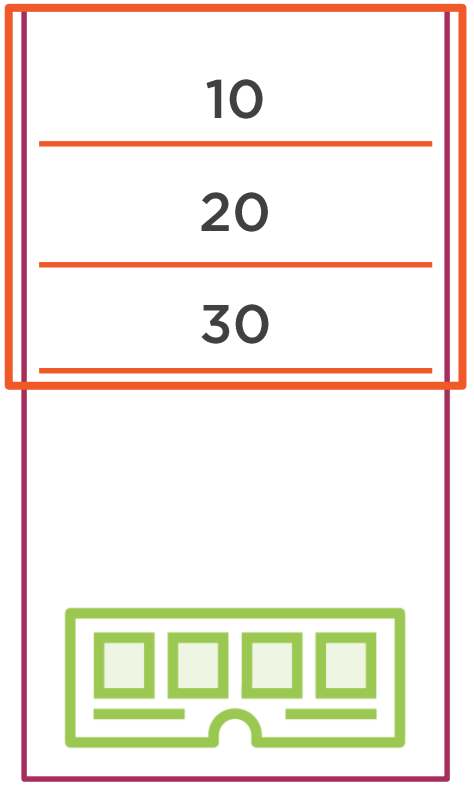
~~buffer.rewind()~~

```
buffer.asIntBuffer()
```



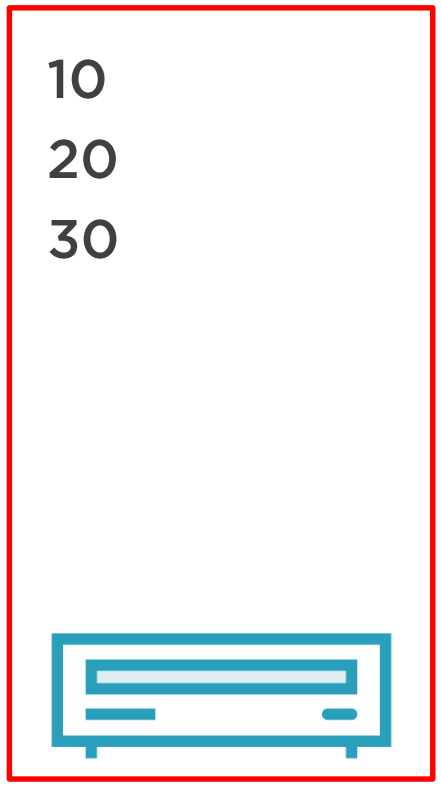
File

Channel



Buffer

```
buffer.flip()
buffer.asIntBuffer()
```



File

Channel

Buffer

```
buffer.clear()
```

```
FileChannel fileChannel = FileChannel.open(
      Paths.get("files/ints.bin"), READ);

ByteBuffer byteBuffer = ByteBuffer.allocate(1024);

fileChannel.read(byteBuffer);

byteBuffer.flip();
IntBuffer intBuffer = byteBuffer.asIntBuffer();

int i = intBuffer.get();
```

First we create a file channel and a byte buffer

And then read the content of the file to fill the buffer

Then we flip the byte buffer, and create an int buffer on it

The we can read integers from the int buffer, in a while loop

Rewind vs. Flip

Rewind resets the cursor

Flip resets the cursor and prevents readings past what has been written into the buffer

Reading and Writing to Multiple Buffers

Scattering Read Operation

The Scattering read consists in reading from a single channel to an array of buffers

It is specified by the ScatteringByteChannel interface

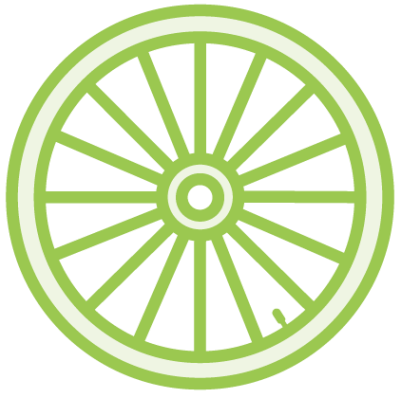
The reading process fills the first buffer and continues with the next one

Gathering Write Operation

The Gathering Write consists in writing from an array of buffers to single Channel

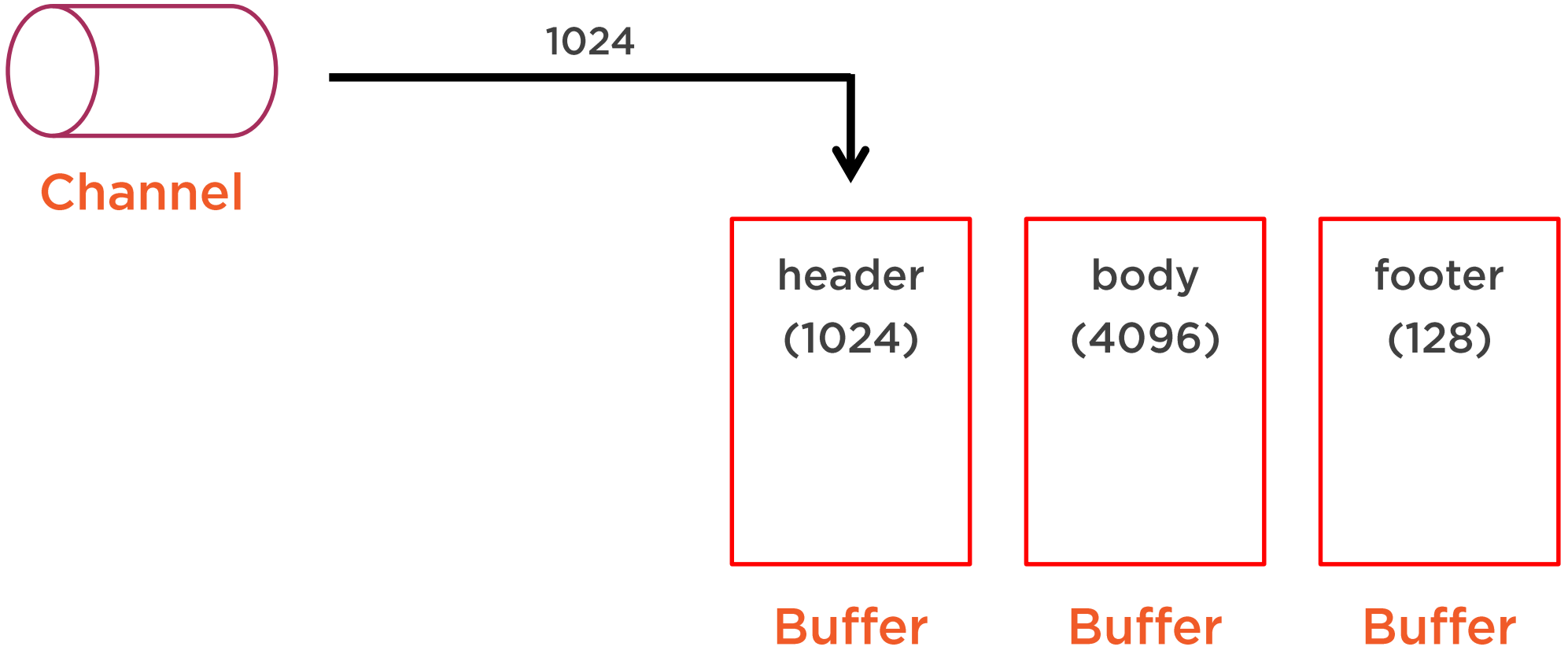
It is specified by the ScatteringByteChannel interface

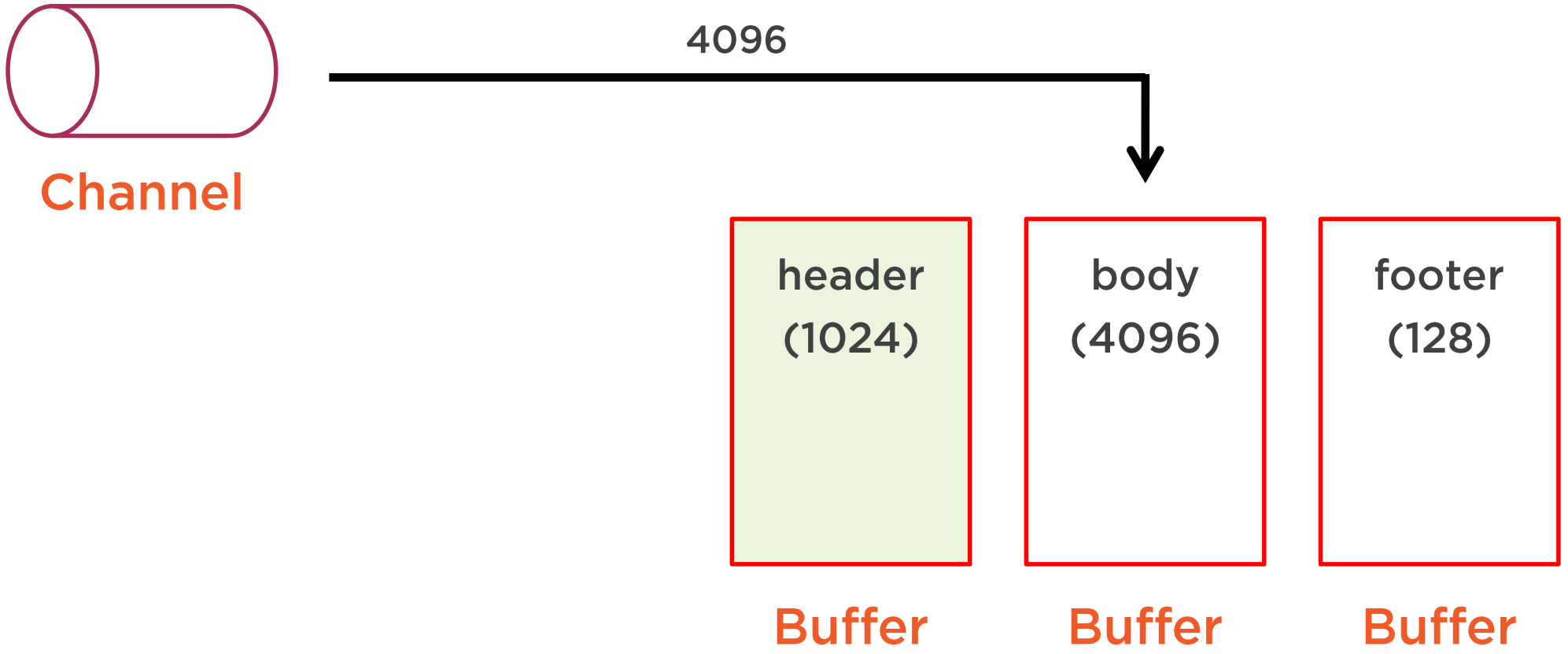
The writing operation starts with the first buffer, then the next one and so on

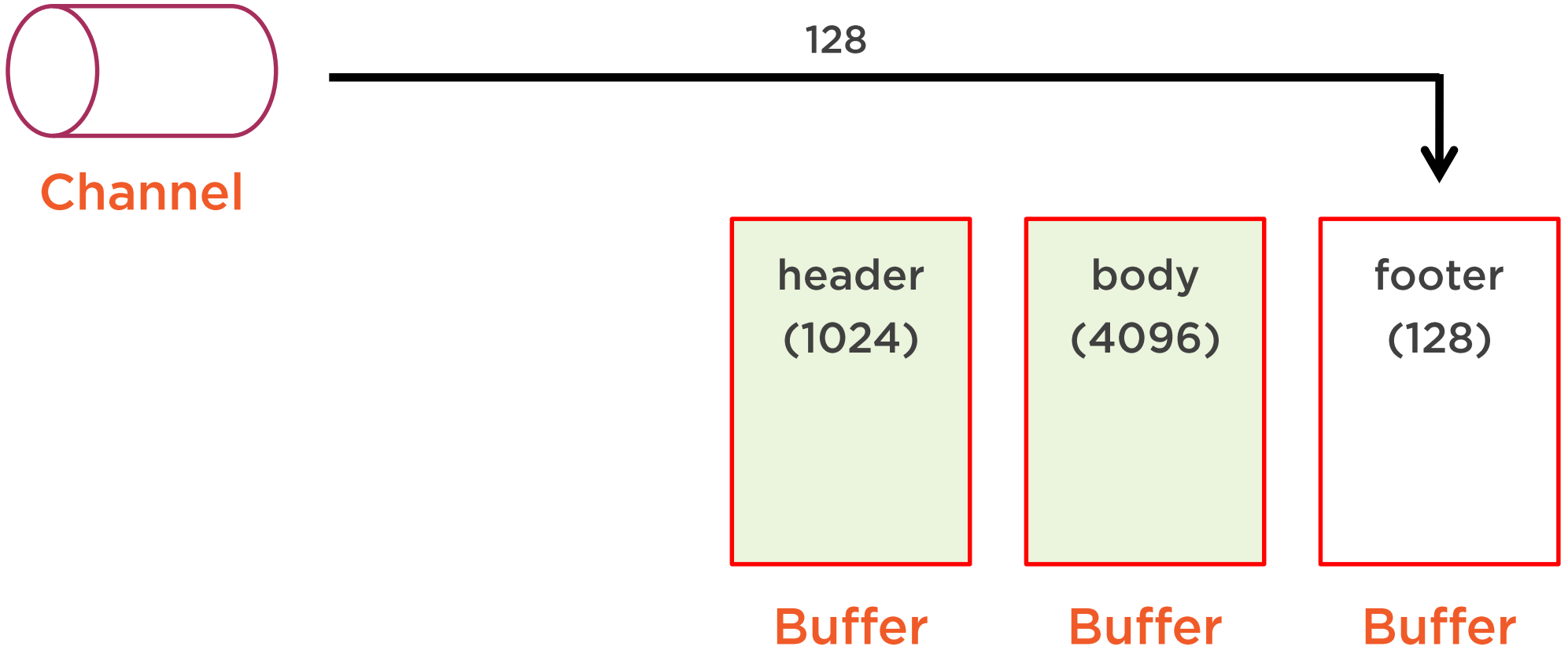


The Gather / Scatter pattern is useful when handling messages with fixed-length parts

- a 1024 bytes header
- a 4096 bytes body
- a 128 bytes footer


1024
Channel
header
(1024)
body
(4096)
footer
(128)
Buffer
Buffer
Buffer


4096
Channel
header
(1024)
body
(4096)
footer
(128)
Buffer
Buffer
Buffer


128
Channel
header
(1024)
body
(4096)
footer
(128)
Buffer
Buffer
Buffer

Channel

header
(1024)

Buffer

body
(4096)

Buffer

footer
(128)

Buffer

```
ByteBuffer header = ByteBuffer.allocate(1024);
ByteBuffer body   = ByteBuffer.allocate(4096);
ByteBuffer footer = ByteBuffer.allocate(128);

ByteBuffer[] message = {header, body, footer};

long bytesRead = channel.read(message);
```

First we create the different buffers for our message

Then we read the file channel into this array

It returns the number of bytes read in a long, since it may not fit in an integer

```
ByteBuffer header = ByteBuffer.allocate(1024);
ByteBuffer body   = ByteBuffer.allocate(4096);
ByteBuffer footer = ByteBuffer.allocate(128);

ByteBuffer[] message = {header, body, footer};

long bytesWritten = channel.write(message);
```

First we create and populate our buffers

Then we add them to a single array

And just write the array to the channel

Scattered Read / Gathering Write

A very useful pattern to handle fixed sized messages

Remember to properly rewind the buffers when using it

Using MappedByteBuffers

MappedByteBuffer is a buffer that maps a file to memory

Useful for applications that read the same file many times

Three modes:

- READ

- READ_WRITE

- PRIVATE

Can also be used for a portion of a file

```
FileChannel fileChannel = FileChannel.open(
      Paths.get("files/ints.bin"), READ);

MappedByteBuffer mappedBuffer = fileChannel.map(
      FileChannel.MapMode.READ_ONLY, 0, fileChannel.size());

CharBuffer charBuffer = StandardCharsets.UTF_8.decode(mappedBuffer);
```

First we create a file channel on a path

And then map the corresponding file to this mapped buffer

Then we can decode it to a char buffer if it is a text file

Using Charsets

Java NIO has ByteBuffer and CharBuffer

To convert one into another, we need to specify a charset

This uses decoders and encoders

Charsets

The following charsets are supported in Java:

- US_ASCII, ISO_8859_1 (aka latin1)
- UTF_8
- UTF_16LE, UTF_16BE, UTF_16

Charsets

A charset has two methods:

- encode(): takes a CharBuffer, returns a ByteBuffer

- decode(): takes a ByteBuffer, returns a CharBuffer

Using these methods is the only way to convert a CharBuffer to a ByteBuffer and vice-versa

```
FileChannel channel = FileChannel.open(
   Paths.get("files/text-latin1.txt", StandardOpenOption.READ);
ByteBuffer buffer = ByteBuffer.allocate(1024);
channel.read(buffer);

CharSet latin1 = StandardCharsets.ISO_8859_1;
CharBuffer utf8Buffer = latin1.decode(buffer);

String result = new String(utf8Buffer.array());
```

First we create a channel an read it in a buffer

Then decode method decodes a Latin1 buffer to a UTF8 char buffer

We can then put the result, for instance, in a String

```
CharSet utf8 = StandardCharsets.UTF_8;
ByteBuffer byteBuffer = utf8.encode(buffer);

anotherFileChannel.write(byteBuffer);
```

Or we can encode a UTF8 char buffer into a UTF8 ByteBuffer

Then write the result in a UTF8 file

Buffers and Charsets

Channels can only read and write byte buffers

Encoding and decoding are the only way to convert a byte buffer to a char buffer

Encoding and decoding are only available through the charsets

The Channels Factory Class

Channels

A utility class with 10 factory methods:

- to create channels from InputStream and OutputStream

- to create InputStream and OutputStream from a channel, asynchronous or not

- to create readers and writers from a channel, providing a charset

Demo

Let us see some code!

Let us play with Buffers and Channels

Use charsets

And see how we can read an write data on files with them

Module Wrapup

What did we learn?

The fundamentals of Java NIO:

- buffers and channels

- resetting, rewinding, flipping

- scattering and gathering

- charsets to encode / decode characters