Technische Universität Berlin

Distributed Systems

Exercise 1: Sockets / Java NIO

CIT

Complex and Distributed IT Systems

Agenda

Organizational matters

Follow-up: idea behind sockets

Java New I/O

- motivation
- buffer and channels
- transformation characters <=> bytes
- asynchronous I/O using the selector approach

Notes regarding exercise sheet 1

- SMTP overview
- notes and tips

Organizational Matters

Tutors:

- Gabriel Behrendt: gabriel.behrendt@campus.tu-berlin.de
- Aurel Isaak Weinhold: a.weinhold@campus.tu-berlin.de

Material published on the ISIS website:

- lecture slides
- guidelines & tools (e.g. code templates)

Exam authorization:

- There will be 4 exercise sheets which will be evaluated

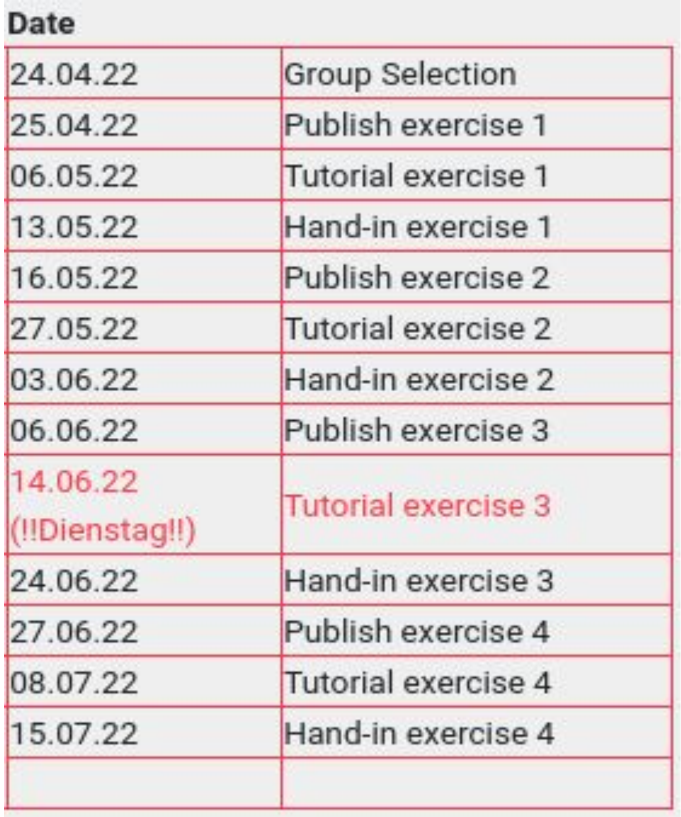
Exercise schedule published on the ISIS website

Organizational matters

Procedure

- 4 assignments throughout the semester
- Work in teams of 5 students
- One exercise sheet per assignment
- Introductory material at beginning of a assignment
- Tutorial 2 weeks later for resolving problems and asking questions

Appointments



Date	
24.04.22	Group Selection
25.04.22	Publish exercise 1
06.05.22	Tutorial exercise 1
13.05.22	Hand-in exercise 1
16.05.22	Publish exercise 2
27.05.22	Tutorial exercise 2
03.06.22	Hand-in exercise 2
06.06.22	Publish exercise 3
14.06.22 (!!Dienstag!!)	Tutorial exercise 3
24.06.22	Hand-in exercise 3
27.06.22	Publish exercise 4
08.07.22	Tutorial exercise 4
15.07.22	Hand-in exercise 4

Idea behind Sockets

Definition of distributed systems given during the lecture:

> A Distributed System is one in which hardware or software components are located at networked computers and communicate and coordinate their actions only by passing messages.
>
> © Colouris

Problem: heterogeneous hard- and software components

- How to exchange messages between them?
- The operating system has to provide a uniform interface to the applications

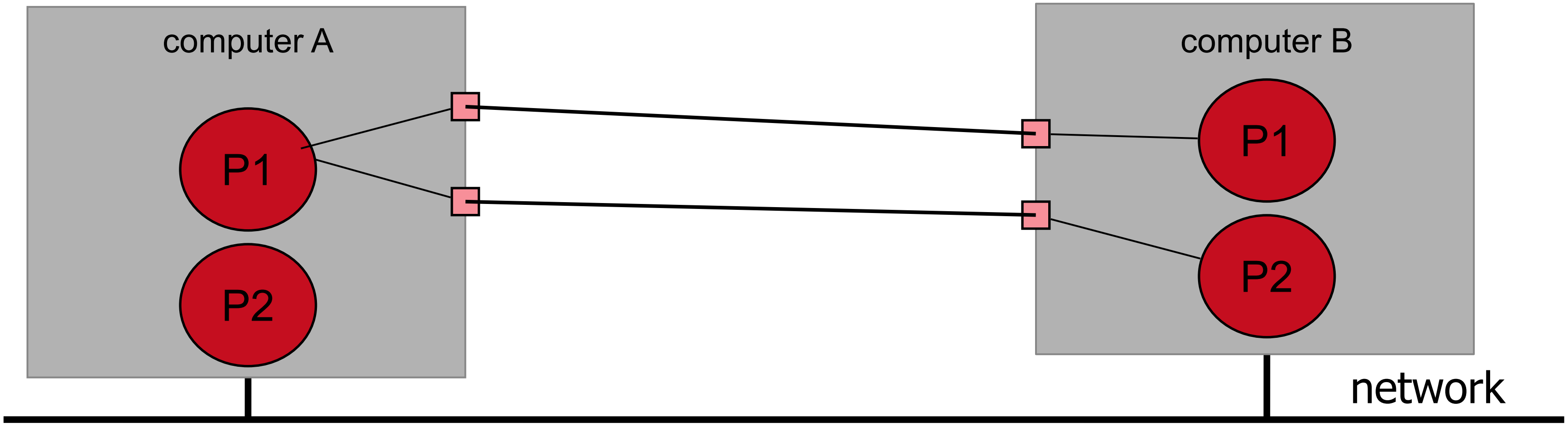
Solution: Sockets

Uniform interface used for network- and inter-process communication (IPC)

- Originally developed for Berkeley Unix (1981)
- Meanwhile supported among most of available platforms

Represent end-point of a communication channel

Each endpoint of the connection corresponds to one process

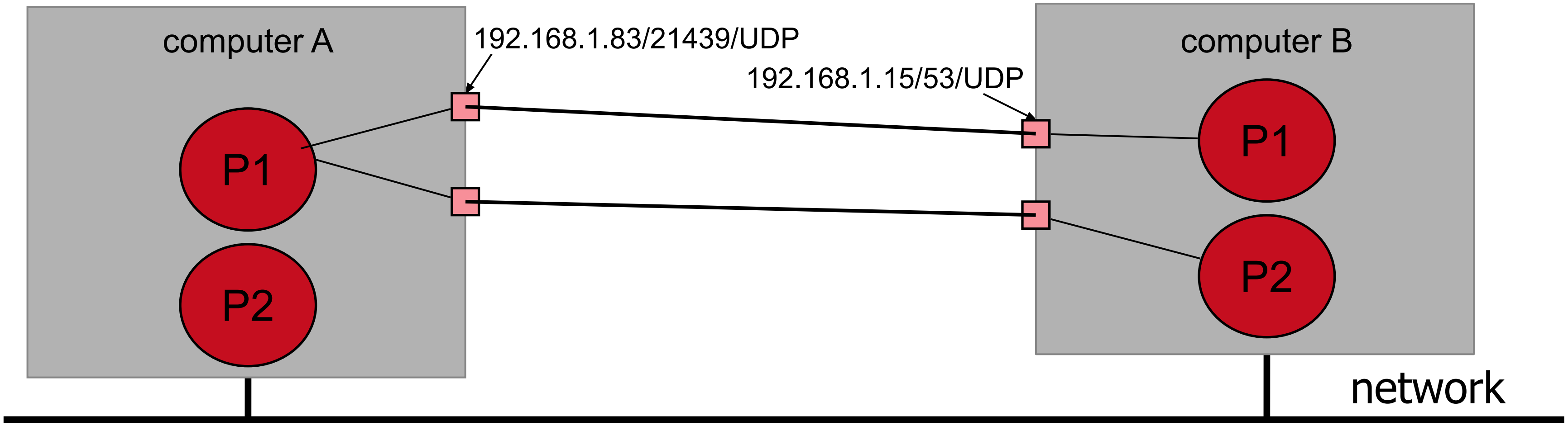


What is a socket?

Process requests a socket from the OS using:

- address and port used to access the process inside the network
- protocol to transmit data

Combination of address/port/protocol uniquely identifies a process in the network



Sockets and Protocols

Sockets can operate using different transmission protocols

Protocol selection has impact on the characteristics of the communication

Typical classification of transmission protocols:

- connectionless communication (e.g. UDP)
- connection oriented communication (e.g. TCP)
- (local communication)
- (proprietary communication protocols)

Basic Socket Operations

Basic operations when working with sockets

- Applicable for both connectionless and connection oriented protocols
  - socket(): creates a socket
  - bind(): bind process to address/port (server only)
  - send(): send data
  - recv(): receive data
  - close(): close socket
- Applicable for connection oriented protocols only
  - connect(): establish connection (client only)
  - listen(): prepare socket to accept new connections (server only)
  - accept(): wait for new connections (server only)

What is Java New I/O?

New API for Input-/Output (I/O) operations

- e.g. file-I/O, network-I/O

Benefits for I/O intensive applications (usually servers)

- faster execution of I/O operations
- higher scalability in case of many concurrent clients

Introduced 2002 along with Java 1.4

- updated with Java SE7

Why Java New I/O?

Many server applications can be characterized by:

- high requirements regarding I/O performance
  - e.g. File Server, Web Server, Mail Server, ...
- many concurrent client requests
  - several data streams need to be maintained in parallel
- heterogeneous clients
  - possibly different data formats
  - possibly different standards for character encoding
  - frequent source of error

Why Java New I/O?

former Java-abstraction for I/O is based on streams

- Advantages: intuitive, combinable

```
Socket client = server.accept();
InputStream inputStream = client.getInputStream();
while((int i = inputStream.read()) != -1) {
   /*... do something... */
}
```

- Disadvantages:
  - Byte-per-Byte-processing slow/inefficient (performance)
  - read-/write-operations blocking → many threads required (performance, scalability)

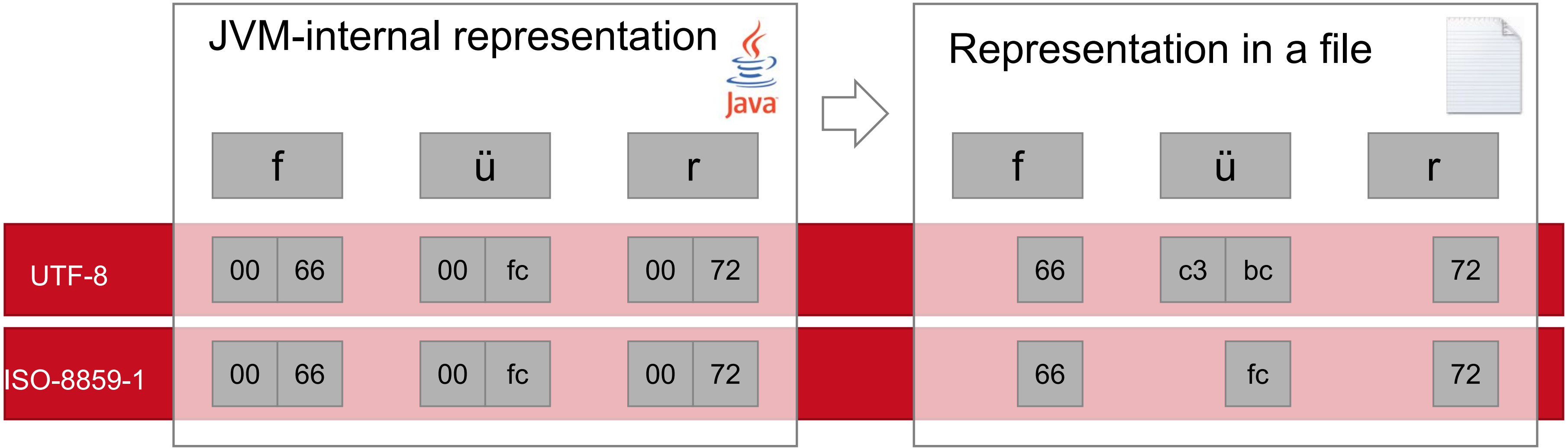
Why Java New I/O?

Stream oriented I/O classes distinguish between

- Byte streams: Byte (8 Bit)
- Character streams: UTF-16 encoded double bytes (16 Bit)

Encoding Character stream <=> Byte stream frequent source of errors

- problem: Java uses default-charset of operating system



	JVM-internal representation			Representation in a file		
	f	ü	r	f	ü	r
UTF-8	00 66	00 fc	00 72	66	c3 bc	72
ISO-8859-1	00 66	00 fc	00 72	66	fc	72

Buffers and Channels

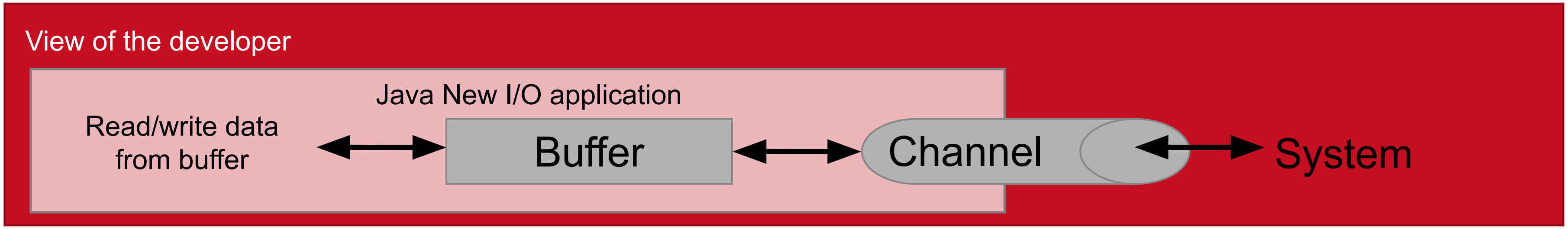
Java NIO uses buffers instead of Byte-per-Byte

- Idea: memory backing buffer is allocated in regions the operating system uses for its I/O operations

=> no CPU overhead required due to copy operations

Channels are bidirectional interfaces for I/O

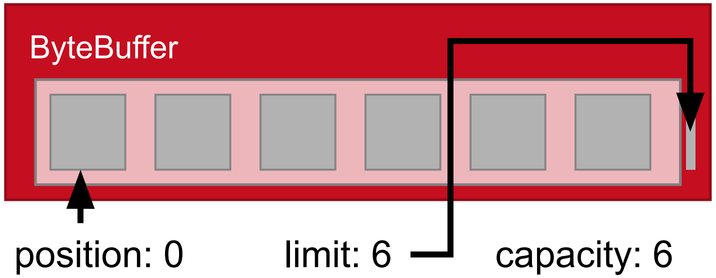
- Incoming data are written into the buffer
- Outgoing data are read from the buffer



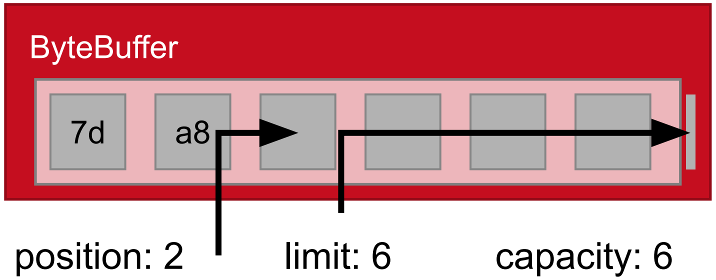
Introduction to Buffers

Buffer classes of NIO have three important properties

- position: position for next read/write operation
- limit: position until which read/write is allowed
- capacity: capacity of the buffer (static)



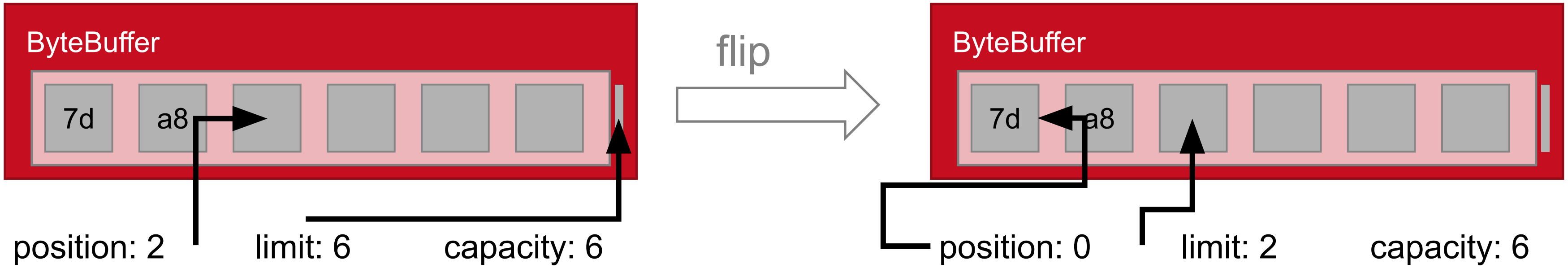
ByteBuffer of length 6,
initial state



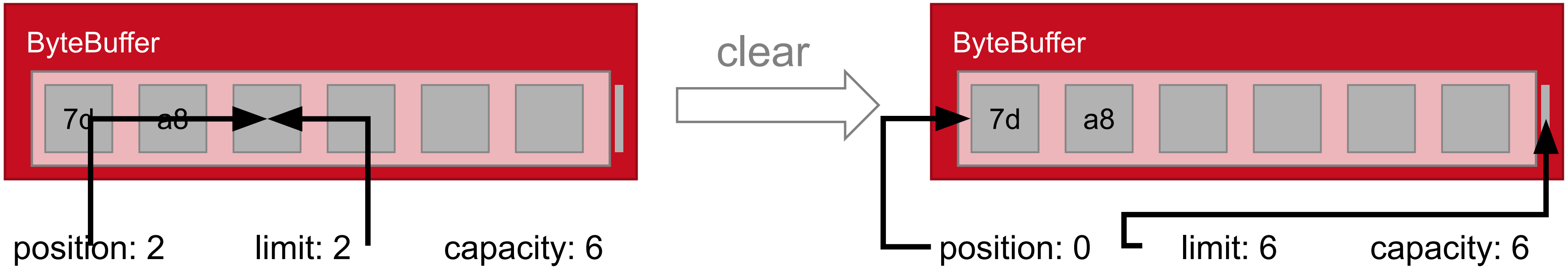
ByteBuffer of length 6,
state after inserting two bytes

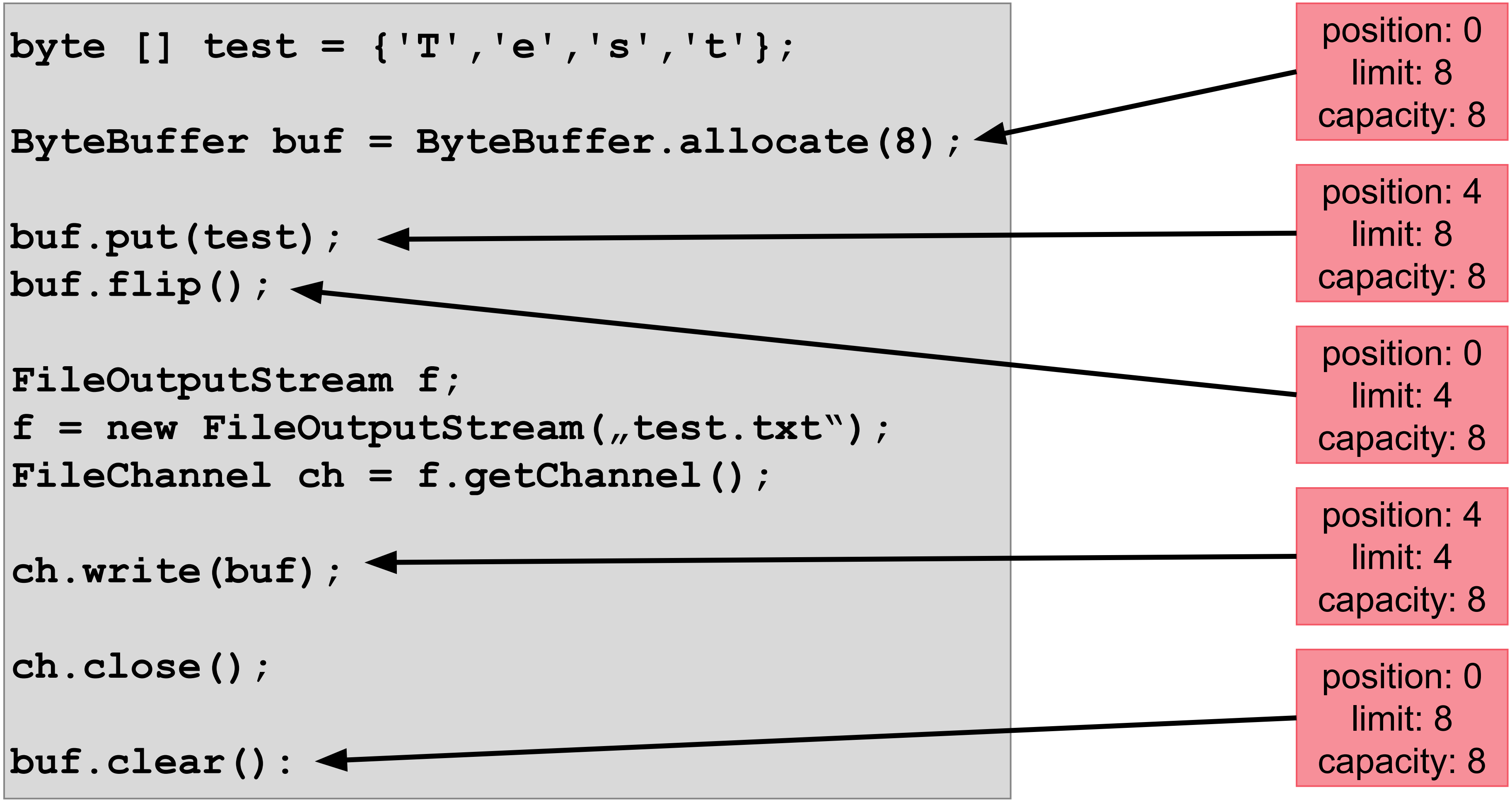
Important buffer operations

- flip-operation: limit = position, then position = 0



- clear-Operation: position = 0, then limit = capacity





```
byte [] test = {'T','e','s','t'};

ByteBuffer buf = ByteBuffer.allocate(8);

buf.put(test);
buf.flip();

FileOutputStream f;
f = new FileOutputStream(„test.txt“);
FileChannel ch = f.getChannel();

ch.write(buf);

ch.close();

buf.clear():
```

position: 0
limit: 8
capacity: 8

position: 4
limit: 8
capacity: 8

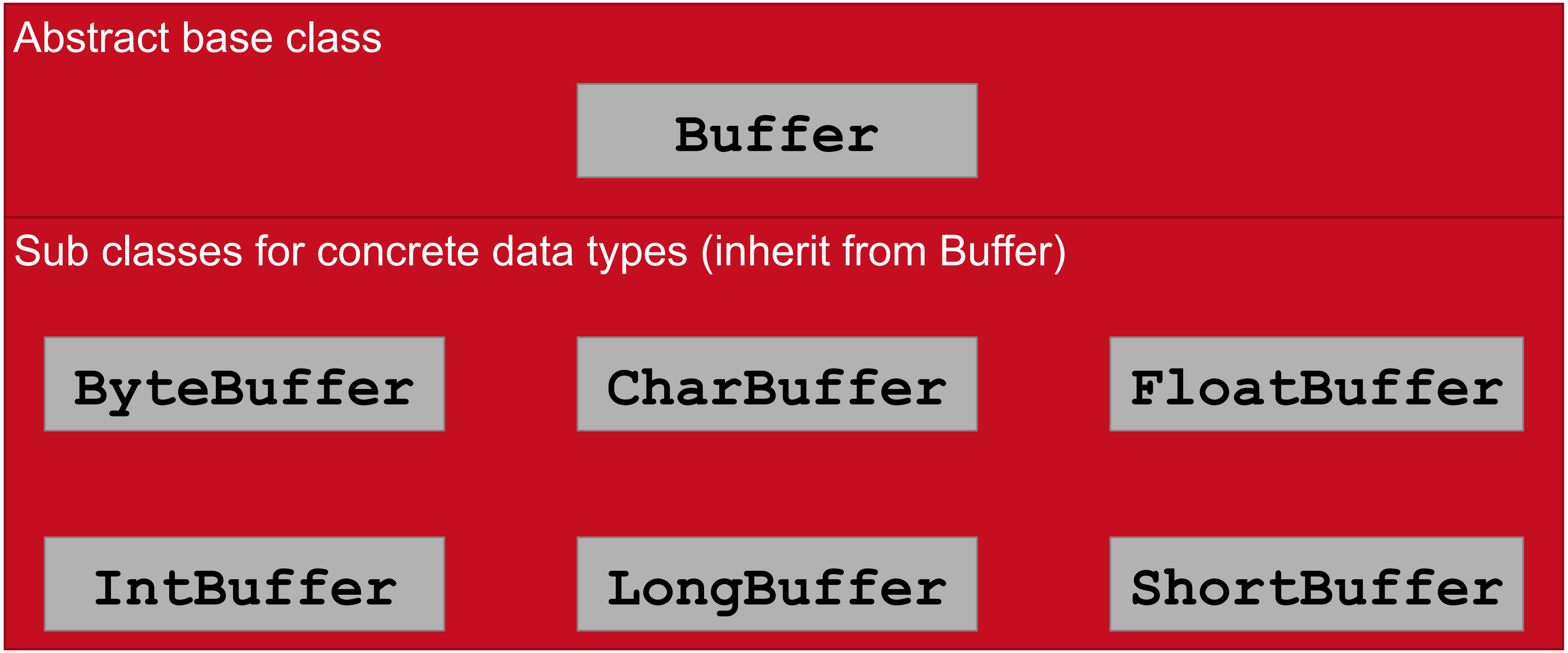
position: 0
limit: 4
capacity: 8

position: 4
limit: 4
capacity: 8

position: 0
limit: 8
capacity: 8

NIO buffer class hierarchy

Im Packet `java.nio`



Abstract base class

`Buffer`

Sub classes for concrete data types (inherit from Buffer)

`ByteBuffer` `CharBuffer` `FloatBuffer`

`IntBuffer` `LongBuffer` `ShortBuffer`

Methods for Channels

Important classes that implement the channel interface

- **`FileChannel`**
- **`SocketChannel`**
- **`ServerSocketChannel`**

Important methods used with channels

- **`write(ByteBuffer src)`**: writes **`src`** into channel
- **`read(ByteBuffer dst)`**: writes content of channel into **`dst`**
- **`close()`**: closes channel

Transformation
Characters → Bytes

Java NIO introduces explicit classes to encode/decode between strings and byte arrays

Developer can define charset

```
String s = "Verteilte Systeme";
Charset messageCharset = null;

try {
   messageCharset = Charset.forName("US-ASCII");
} catch(UnsupportedCharsetException uce) {...}

byte [] b = s.getBytes(messageCharset);
byteBuffer.put(b);
```

Transformation
Bytes → Character

```
CharsetDecoder decoder = messageCharset.newDecoder();

try {

   CharBuffer charBuf = decoder.decode(byteBuffer);

} catch (CharacterCodingException e) {...}

String s = charBuf.toString();
```

`CharBuffer` implements `CharSequence` interface

- Explicit conversion into String often unnecessary
- e.g. matching of regular expressions

Asynchronous I/O Using the Selector Approach

I/O-API before Java NIO was solely synchronous

- blocking operations
- one thread per connection
- results in loss of performance and bad scalability

Java NIO offers asynchronous I/O API if operating system support available

- application triggers I/O operation and keeps on executing
- one thread is able to maintain several connections

The Selector Class

`Selector`-class serves as a registry for channels

- Channel has a set of supported events
- Developer defines which events are of interest

Possible events:

- OP_CONNECT: connection established successfully (client)
- OP_ACCEPT: new connection established by client (server)
- OP_READ: Data are ready to read from the channel
- OP_WRITE: Channel is ready to write data

Example: Create/Register ServerSocket

```
Selector selector = Selector.open();

ServerSocketChannel servSock = ServerSocketChannel.open();
servSock.configureBlocking(false);
servSock.socket().bind(new InetSocketAddress(6332));

servSock.register(selector, SelectionKey.OP_ACCEPT);
```

`ServerSocketChannel` encapsulates `ServerSocket`

- Socket listens on port 6332
- Notification about event OP_ACCEPT requested
- Note: no exception handling shown → required for real applications!

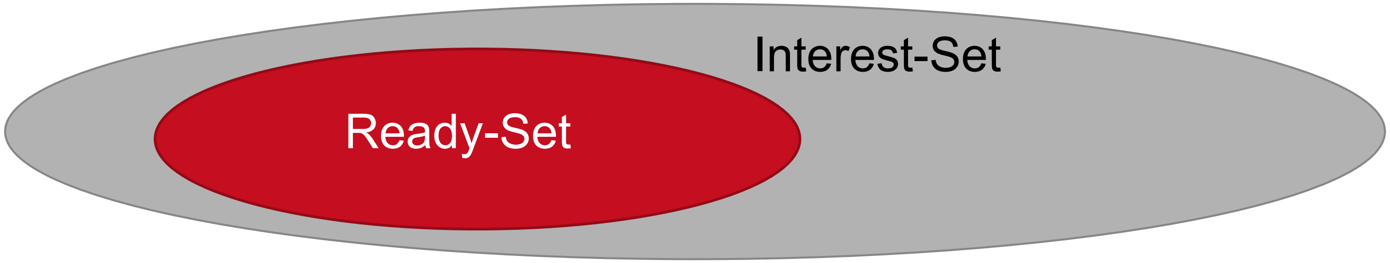
Notifications About Occurred Events

Interest-Set

- Events related to a channel we are interested in
- Define by the parameters of the **`register()`** method

Ready-Set

- Registered events that have occurred
- Subset of the interest-set of a channel



Are there channels with non-empty ready-set?

- Query using **`select()`** method of the **`Selector class`**

Example: Notification

```
...
while(true) {

    if(selector.select() == 0) /* blocking */
        continue;

    Set<SelectionKey> selectedKeys = selector.selectedKeys();
    Iterator<SelectionKey> iter = selectedKeys.iterator();

    while(iter.hasNext()) {

         SelectionKey key = iter.next();
        /* check ready set of channel */
        iter.remove();
    }
}
```

The Class SelectionKey

`SelectionKey` is the key to a channel with non-empty Ready-Set

- Important methods to check the state of a **`SelectionKey`**
  - **`isReadable()`**: Tests if data is ready for reading
  - **`isWritable()`**: Tests if channel is ready to write data
  - **`isAcceptable()`**: Tests if connection is ready to be accepted (server)
  - **`isConnectable()`**: Tests if connection is ready to be established (client)
  - **`cancel()`**: Unregister selector with channel
  - **`channel()`**: Access to the corresponding channel
  - **`attach(Object ob)`**: Attach additional data to the channel (e.g. a session object identifying the client)

Example: Event Query

```
...
Iterator<SelectionKey> iter = selectedKeys.iterator();

while(iter.hasNext()) {

    SelectionKey key = iter.next();

    if(key.isAcceptable()) {
        /*... do something ...*/
    }
    if(key.isReadable()) {
        /*... do something ...*/
    }

    iter.remove();
}
```

Example: New Connection Accepted

```
...
if(key.isAcceptable()) {

    ServerSocketChannel sock =
        (ServerSocketChanel) key.channel();

    SocketChannel client = sock.accept();
    client.configureBlocking(false);
    client.register(selector, SelectionKey.OP_READ | SelectionKey.OP_WRITE);
}
...
```

New **`SocketChannel`** for incoming connection

- configured as non-blocking
- afterwards registered with the **`Selector`**

Example: Read Data From the Channel

```
...
if(key.isReadable()) {

    ByteBuffer buf = ByteBuffer.allocate(1024);

    SocketChannel channel = key.channel();

    channel.read(buf);

    buf.flip();

    /*Further processing of data*/
}
...
```

Overview SMTP

Simple Mail Transfer Protocol used to exchange e-mails

- Simple ASCII protocol based on the Request/Reply-pattern

Well defined protocol-handshake

- HELO <Hostname>
- MAIL FROM: <e-mail address of the sender>
- RCPT TO: <e-mail address of the recipient>
- DATA
- <content of the e-mail, terminated with \r\n.\r\n>
- QUIT
- (HELP)

Notes Regarding Exercise 1

In order to test your server implementation, a test client is published on the ISIS website

- telnet-client helpful for testing first functionalities

Test client judges success of a request by inspection of the returned response code

The test client will randomly interrupt the protocol handshake with HELP commands!

Tips & Additional Material

Java API-Docs

- https://docs.oracle.com/javase/8/docs/api (first introduction of Java Streams)
- https://docs.oracle.com/javase/10/docs/api/

Source code of the SMTP-client

ISIS forums