Nebenläufige und verteilte Programme CS2301

Th. Letschert

TH Mittelhessen Gießen

University of Applied Sciences

New I/O

- NIO
- Kommunikation mit *Channels* und *Buffers*

NIO – NIO und new NIO

NIO

Klassische Java-IO

- Strom-orientiert : Daten werden aus Strömen gelesen in Ströme geschrieben
- Vorteil : ermöglicht einfache / elegante Kombination mit Filtern und Puffern
- Nachteil : Langsam, synchrone / blockierende Aktionen

NIO

- ab Java 1.4
- Packages: java.nio, java.nio.channels, java.nio.charset
- Klassen: Buffer- und Channel-Klassen
- Ziel:
  - Direkte Nutzung der IO-Funktionalität des BS
  - Effizienter: Schneller, weniger Spreicherbedarf, bessere Skalierung
- Nutzung
  - Vornehmlich für Serveranwendungen
  - Für (kleine) Client-Anwendungen weniger sinnvoll

NIO.2 (new NIO)

Mit Java 7 eingeführte überarbeitete/erweiterte Version der NIO *Packages*:

Zwei wesentliche Bestandteile:

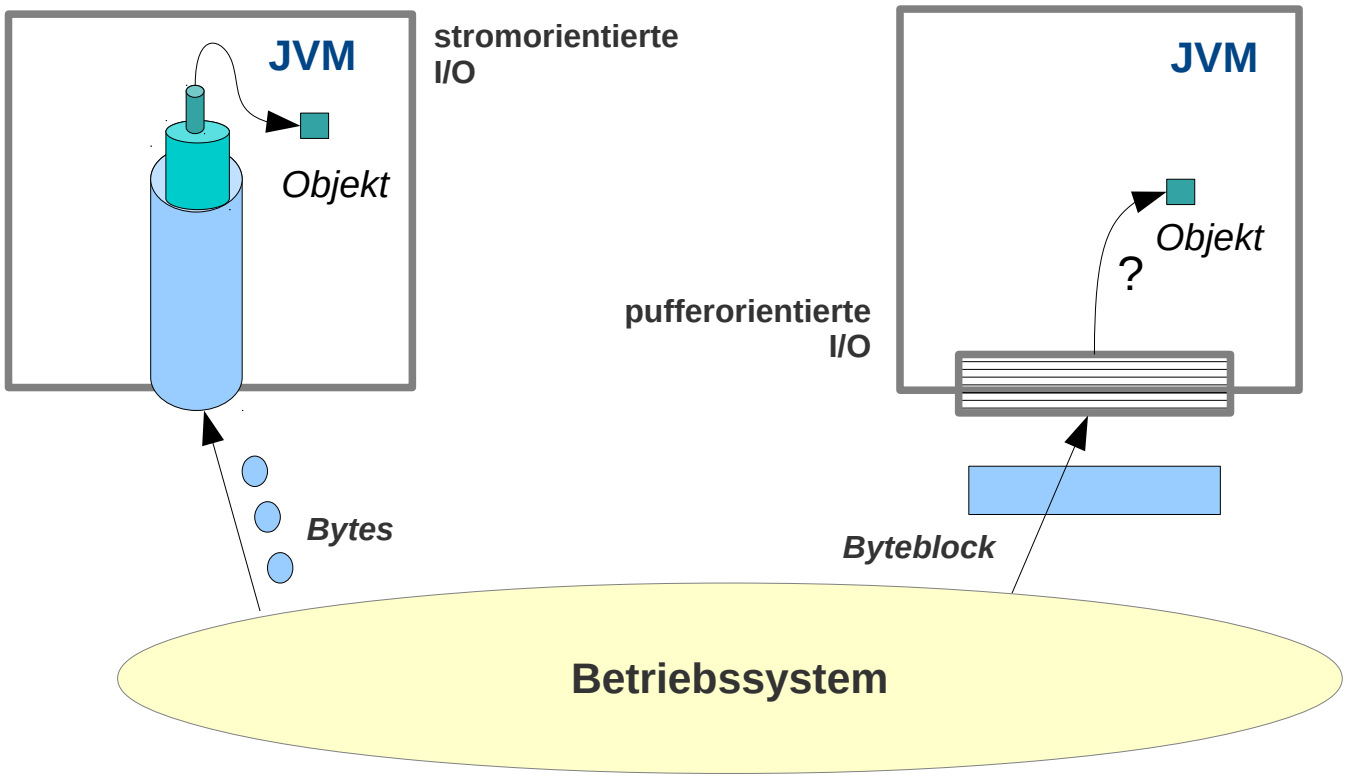
- Neue API für Dateibehandlung (Path, etc.)
- überarbeitete / neue Klassen für Kommunikationsanwendungen speziell zur Unterstützung von Multicasts und Asynchrone I/O

NIO.2 (ab Java-7) umfasst NIO (ab Java 1.4).

NIO

Block-orientierte vs Strom-orientierte IO

- Strom-orientiert: einfach, elegant, langsam
- Block- / Puffer- orientiert: komplexer, effizienter



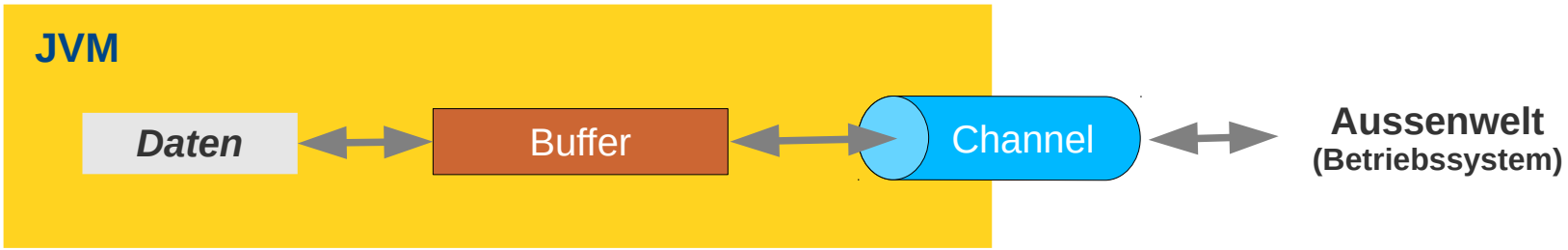
NIO

Channel

- entsprechen Streams
- Verbindung zur Außenwelt (Socket / File)

Buffer

- „Zwischenstation“ der Daten
- Senden aus einem Buffer / Empfangen in einen Buffer



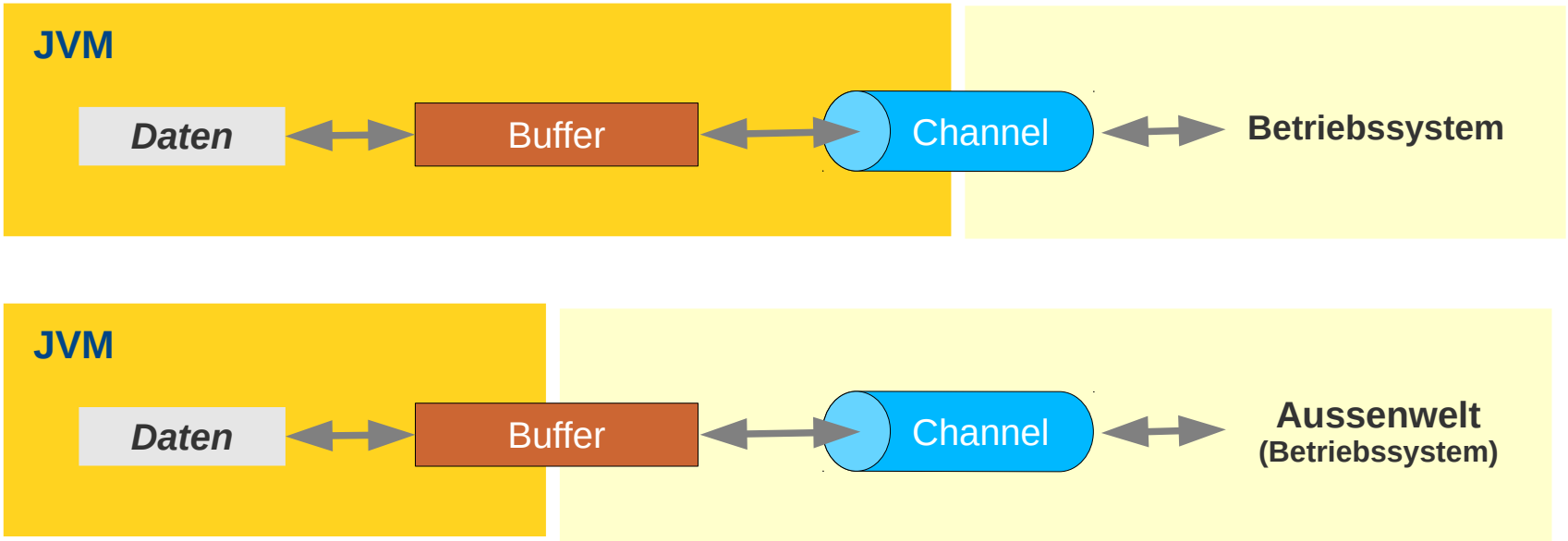
Buffer

Das Kopieren von Daten – speziell von einem Adressraum in den anderen – ist eine aufwendige Aktion

Kommunikationsanwendungen transferieren Daten zwischen dem Adressraum der Anwendung (z.B. der JVM) und dem Betriebssystem.

NIO-Buffer machen diese Datentransfers effizienter (im Vergleich zu „normaler“ I/O)

- **Effizientere Kopieroperationen**
- **Zugriff auf direkt allokierte Puffer im Adressraum des Betriebssystems**



Direkt allokierter Buffer

NIO Beispiel / Lesen von der Standard-Eingabe

```
object BasicNIO_Main extends App {
  val byteBuf: ByteBuffer  = ByteBuffer.allocate(256);                // Puffer anlegen
  val channelI: ReadableByteChannel = Channels.newChannel(System.in); // Channel aus Stdin erzeugen

  val encoding: String = System.getProperty("file.encoding");
  val charset: Charset = Charset.forName(encoding);                   // Charset bestimmen
  val decoder: CharsetDecoder = charset.newDecoder();                 // Einen passenden Decoder erzeugen

  var stopped = false
  while (true) {
    try {
      println("Waiting for STDIN");
      val n = channelI.read(byteBuf); // Puffer einlesen: Kanal => Byte-Puffer

      if (n > 0) {            // n = Zahl der gelesenen Bytes
        byteBuf.flip();       // Auslesen des Puffers ermöglichen

        // Bytes => Zeichen entsprechend der Codierung:
        val message = decoder.decode(byteBuf).toString();
        println("> " + message + " <");
        byteBuf.clear();      // Leeren: Neues einlesen ermöglichen

      } else if ( n == -1 ) {// n = -1 => Ende der Eingabe
          println("STOP !"); stopped = true
      }
    } catch {
        case e: Exception => e.printStackTrace()
    }
  }
}
```

Buffer

Behälter fester Größe mit Elementen mit primitivem Datentyp

- CharBuffer, DoubleBuffer, FloatBuffer, IntBuffer, LongBuffer, ShortBuffer
- **ByteBuffer** enthält Bytes, wichtig für I/O mit **Channels**

Erzeugen

- `ByteBuffer.allocate(n)` *Erzeugung in in der VM*
- `ByteBuffer.allocateDirect(n)` *(eventuell) außerhalb (für langlebige große Puffer)*
- `ByteBuffer bb = ByteBuffer.wrap(byteArray)` *Wrapping: Puffer operiert auf Array*

Buffer

ViewBuffer

Einen ByteBuffer als Short-/Char-/Int-/Long-/Float-/Double-Buffer „betrachten“

Beispiel:

```
import java.nio.ByteBuffer;
import java.nio.channels.Channels

object BufferViewEx_Main extends App {
  val buf         = ByteBuffer.allocate(4);
  val intViewBuf  = buf.asIntBuffer();
  val channel     = Channels.newChannel(System.in)
  channel.read(buf);          // Bytes einlesen: Kanal -> Puffer
  val i = intViewBuf.get();   // und als ints-betrachten
  println(i)
}
```

```
2
839516160
```

Big-Endian Decodierung der Bytes von „2“ als int*

** Big-Endian: Bytes mit niedriger Adresse haben den höheren Wert (stehen links im Register).*
Siehe: https://en.wikipedia.org/wiki/Endianness

Buffer

Byteordnung

Beispiel:

```
import java.nio.ByteBuffer;
import java.nio.channels.Channels
import java.nio.ByteOrder

object BufferViewEx_Main extends App {
  val buf         = ByteBuffer.allocate(4).order(ByteOrder.LITTLE_ENDIAN);
  val intViewBuf  = buf.asIntBuffer();
  val channel     = Channels.newChannel(System.in)
  channel.read(buf);          // Bytes einlesen: Kanal -> Puffer
  val i = intViewBuf.get();   // und als little.endian codierte ints-betrachten
  println(i)
}
```

```
2
2610
```

Little-Endian* Decodierung der Bytes von „2“ als int

**** Little-Endian: Bytes mit niedriger Adresse haben den niedrigeren Wert (stehen rechts im Register). Siehe: https://en.wikipedia.org/wiki/Endianness***

Buffer

Beispiel: Füllen aus Datei, Leeren nach stdout

```
import java.io.IOException;
import java.nio.ByteBuffer;
import java.nio.channels.Channels;
import java.nio.channels.ReadableByteChannel;
import java.nio.channels.WritableByteChannel;
import java.nio.file.Files;
import java.nio.file.Path;
import java.nio.file.Paths;

object BufferFlipEx_Main extends App {
  val filePath: Path = Paths.get("/some/path/to/datei.txt");

  val input: ReadableByteChannel = Files.newByteChannel(filePath);
  val output: WritableByteChannel = Channels.newChannel(System.out);

  val buffer: ByteBuffer = ByteBuffer.allocate(256);
  while (input.read(buffer) != -1) { // Buffer füllen mit Daten aus Channel
        buffer.flip();           // Flip
        output.write(buffer);    // Buffer leeren: Daten nach output transferieren
        buffer.compact();        // Compact
  }

  // Channel liefert keine Daten mehr; Buffer kann aber noch Daten enthalten
  buffer.flip();
  while (buffer.hasRemaining()) {
        output.write(buffer); // Buffer auslesen Daten nach output transferieren
  }
}
```

Vor jedem Auslesen aus dem Buffer wird flip aufgerufen.

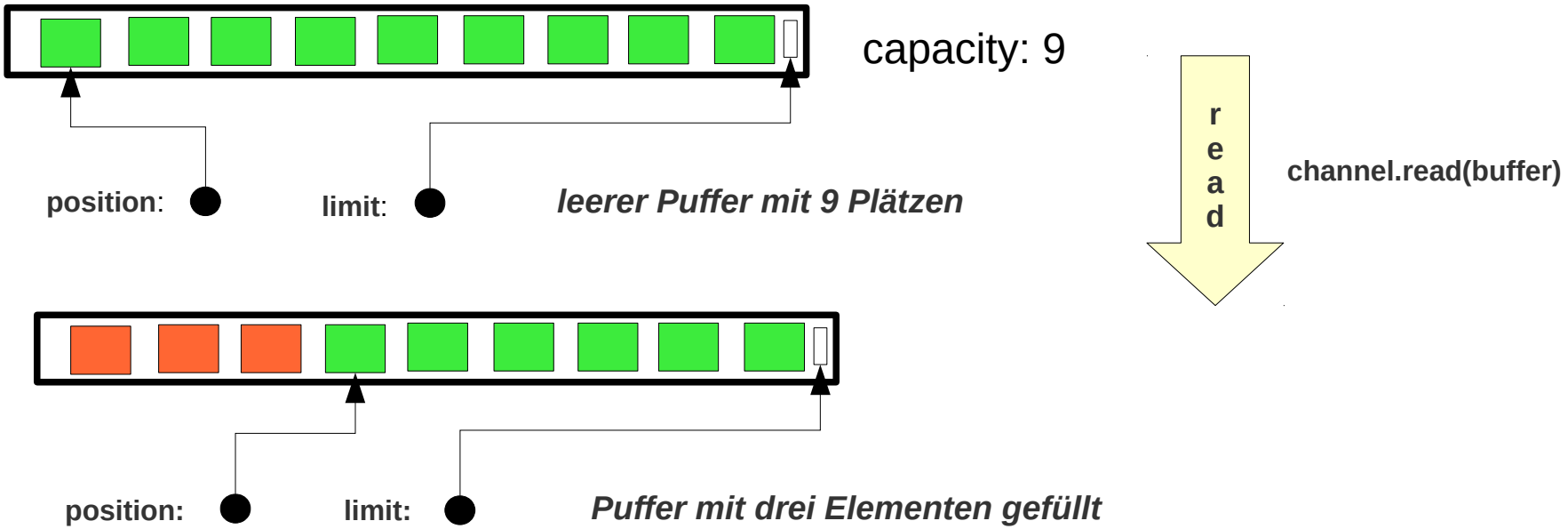
Vor jedem erneuten Befüllen wird compact aufgerufen.

Buffer

Zustand eines Buffers

- **capacity:** Kapazität / Fassungsvermögen
- **limit:** Position bis zur der Lesen/Schreiben möglich ist (ausschließlich)
  nie kleiner 0 nie größer als capacity
- **position:** Aktuelle Schreib- / bzw. Lese-Position
  wird beim Füllen und Leeren verschoben

Beispiel: Buffer (mit Lese(!)-Operation auf Channel) füllen:

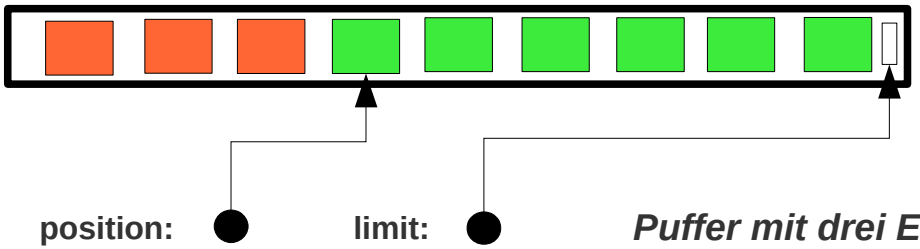


Buffer

Flip: Leeren des Puffers vorbereiten

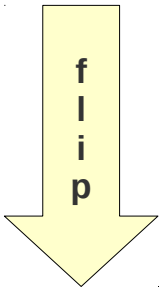
- **flip:** limit <~ position, position <~ 0
  limit wird zu position, postion wird auf 0 gesetzt,
  flip wird nach dem Füllen und vor dem Leeren des Puffers aufgerufen!

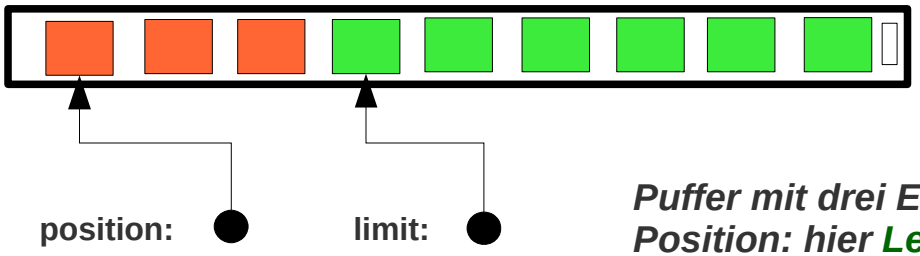
Beispiel: Buffer flip: (man lasse sich von dem seltsamen Namen der Methode nicht verwirren!)



Puffer teilweise gefüllt

Puffer mit drei Elementen gefüllt
Position: hier Lesen/Schreiben
Limit: Verbotene Position





Puffer bereit zur Leerung

Puffer mit drei Elementen gefüllt
Position: hier Lesen/Schreiben
Limit: Verbotene Position

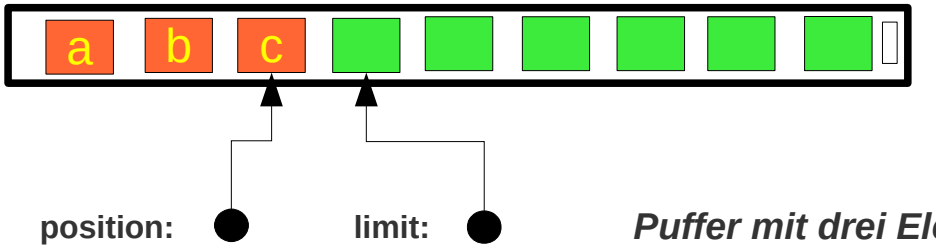
Buffer

Compact: Erneutes Befüllen des Puffers vorbereiten

(wird nicht von jedem Puffer unterstützt)

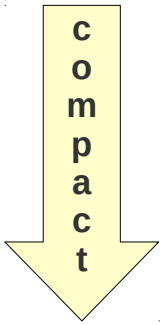
– **compact**: Gelesene Daten (Daten zwischen 0 und position) werden gelöscht.
Ungelesene Daten (Daten zwischen position und limit) werden an den Anfang des Puffers geschoben
Limit wird auf capacity gesetzt, position hinter die ungelesenen Daten

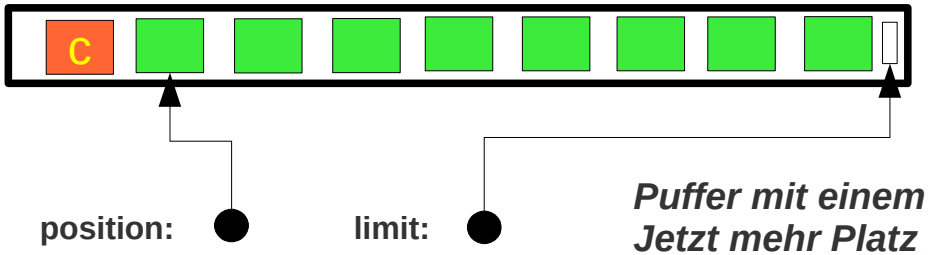
Beispiel: Buffer compact:



Puffer mit teilweise gelesenen Daten

Puffer mit drei Elementen gefüllt zwei davon wurden gelesen (a,b)





Puffer bereit zur Füllung

Puffer mit einem Element gefüllt Jetzt mehr Platz zu Füllen.

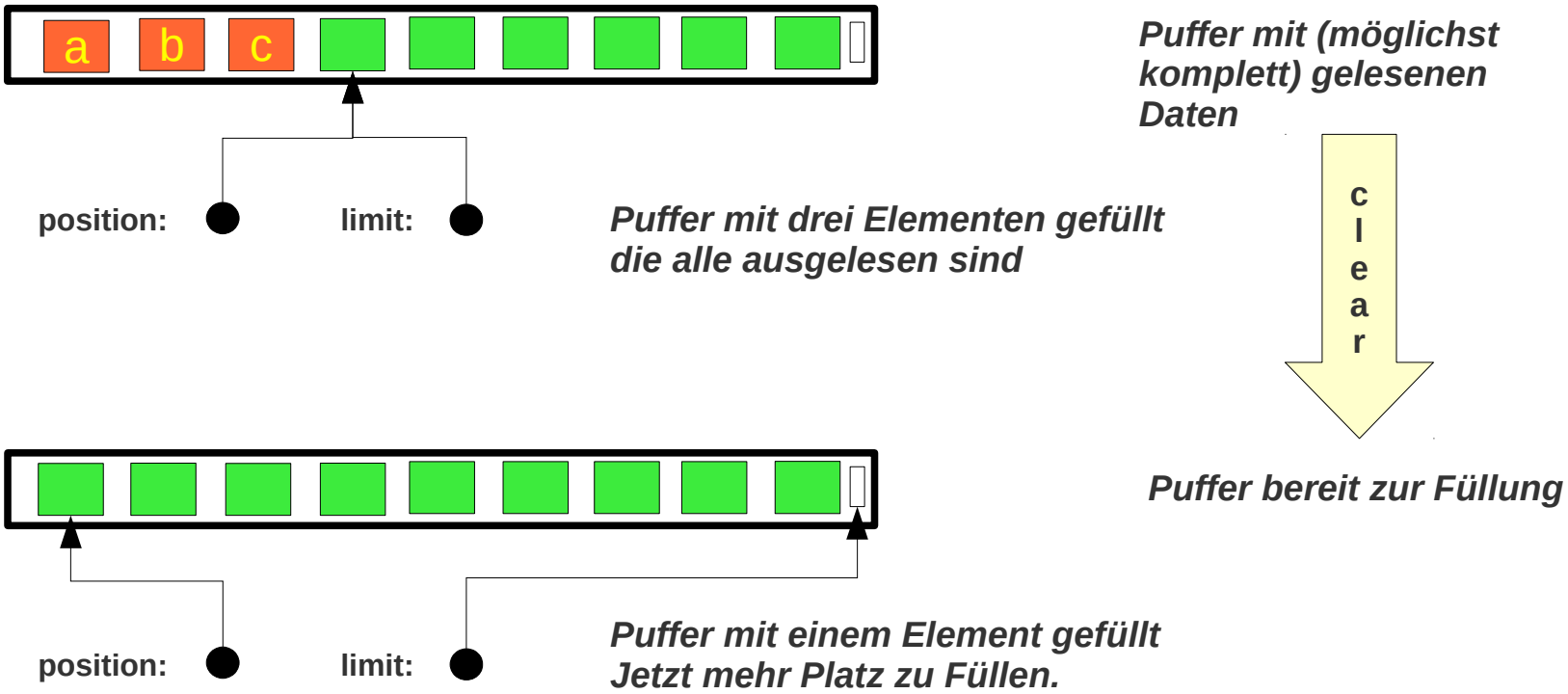
Ist der Puffer völlig geleert (alle Daten ausgelesen), dann entspricht compact einem clear.

Buffer

Clear: Puffer zurücksetzten, bereit machen zu (erneutem) Befüllen

- **clear:** Limit wird auf capacity gesetzt, position auf den Anfang

Beispiel: Buffer clear:



Buffer Verwendung

Die wichtigste Bufferklasse ist ByteBuffer

Lesen-Schreib-Operationen:

- Befüllen: buffer.put(data) channel.read(buffer)
- Leeren: buffer.get() channel.write(buffer)

Zustand

- In Befüllzustand bringen: buffer.clear() / buffer.compact()
- In Leerungszustand bringen: biffer.flip()

Buffer Verwendung Beispiel: Datei kopieren

```
while ( src.read (buffer) != -1 ) {
  // leeren vorbereiten
  buffer.flip();
  // Puffer völlig leeren:
  while (buffer.hasRemaining()) {
     dest.write (buffer);
  }
  buffer.clear();
}
```

Puffer komplett leeren

```
import java.nio.ByteBuffer;
import java.nio.channels.Channels;
import java.nio.channels.ReadableByteChannel;
import java.nio.channels.WritableByteChannel;
import java.nio.file.Files;
import java.nio.file.Paths;
import java.nio.file.StandardOpenOption;
import java.nio.file.OpenOption;

// scala set => java set
import scala.collection.JavaConversions._

object FileCopy_Main extends App {
  val options: Set[OpenOption] = Set(StandardOpenOption.APPEND, StandardOpenOption.CREATE);
  val src: ReadableByteChannel = Files.newByteChannel(Paths.get("/pfad/zu/einer/datei.txt"));
  val dest:  WritableByteChannel = Files.newByteChannel(
                   Paths.get("/pfad/zu/einer/dateiKopie.txt"),
                   options);
  val buffer: ByteBuffer = ByteBuffer.allocate(256);

  while ( src.read (buffer) != -1 ) {
    buffer.flip();
    dest.write (buffer);    // leeren (eventuell teilweise)
    buffer.compact();
  }
  buffer.flip();

  while (buffer.hasRemaining()) { // Rest entleeren
    dest.write (buffer);
  }
}
```

Puffer eventuell teilweise geleert vor „Richtungswechsel“

Channels – Kanäle

- **Interface `java.nio.Channel`**
  Ein Channel gibt Zugang zu I/O-Operationen, entweder auf Dateien, oder auf Sockets
- **Klasse `java.nio.FileChannel`**
  Datei als Channel
- **Klasse `java.nio.SeekableByteChannel`**
  Datei mit wahlfreiem Zugriff als Channel
- **Interface `java.nio.NetworkChannel`**
  Ein Channel der Zugang zu einem Socket bietet
- **Klassen die `NetworkChannel` implementieren:**
  - `java.nio.ServerSocketChannel`
  - `java.nio.SocketChannel`
  - `java.nio.DatagramChannel`
- **Interface `java.nio.MulticastChannel`**
  Ein Teil-Interface von `NetworkChannel` verkörpert die Schnittstelle eines multicast-fähigen Channels
  `DatagramChannel` ist die einzige Klasse, die dieses Interface implementiert.

Diese Übersicht ist nicht vollständig!

Beispiel TCP-Echo-Server – 1

```
import java.io.IOException;
import java.net.InetSocketAddress;
import java.nio.ByteBuffer;
import java.nio.channels.ServerSocketChannel;
import java.nio.channels.SocketChannel;

import scala.util.{ Try, Success, Failure }

object NioTcpServer_Main extends App {
  val ECHO_PORT = 4713

  val serverSocketChannelTry = Try[ServerSocketChannel] {
    ServerSocketChannel.open()
  }

  serverSocketChannelTry match {
    case Failure(e) => e.printStackTrace()
    case Success(serverSocketChannel) => TCPServer(serverSocketChannel, ECHO_PORT)
  }

}
```

Beispiel TCP-Echo-Server – 2

```
object TCPServer {

  def apply(serverSocketChannel: ServerSocketChannel, port: Int): Unit = {

    val buffer: ByteBuffer = ByteBuffer.allocateDirect(1024);

    try {
      if (serverSocketChannel.isOpen()) {
        serverSocketChannel.configureBlocking(true);             //synchrone IO: blockiere
        serverSocketChannel.bind(new InetSocketAddress(port));   // binde Adresse
        System.out.println("Server ready, wait for connection requests");

        while(true) {
            val socketChannel: SocketChannel = serverSocketChannel.accept()
            System.out.println("Incoming connection from: " +
                socketChannel.getRemoteAddress());

            while (socketChannel.read(buffer) != -1) {
              buffer.flip();
              socketChannel.write(buffer);
              if (buffer.hasRemaining()) {
                buffer.compact();
              } else {
                buffer.clear();
              }
            }
        }

      } else { System.out.println("The server socket channel cannot be opened!"); }
    } catch {
      case e: IOException => e.printStackTrace()
    }
  }
}
```

Beispiel TCP-Echo-Client Version 1 – 1

Version 1: Sendet eine Folge von Hallo-Nachrichten

```
import java.net.InetSocketAddress;
import java.nio.ByteBuffer;
import java.nio.channels.SocketChannel;
import java.nio.charset.Charset;

object NioTcpClient_V1_Main extends App {
  val ECHO_PORT = 4713
  val buffer = ByteBuffer.allocateDirect(1024);
  var i = 0;
  val helloBuffer: ByteBuffer = ByteBuffer.wrap(("Hallo Nr " + i).getBytes());
  val charset = Charset.defaultCharset();
  val decoder = charset.newDecoder();
  val socketChannel = SocketChannel.open()
  if (socketChannel.isOpen()) {
     socketChannel.configureBlocking(true);
     socketChannel.connect(new InetSocketAddress("127.0.0.1", ECHO_PORT));
     if (socketChannel.isConnected()) {
        socketChannel.write(helloBuffer);
        // receive and reply messages from server:  . . . nächste Folie . . .
        println("Client: received last msg from Server");
        socketChannel.read(buffer);
        buffer.flip();
        val charBuffer = decoder.decode(buffer);
        println("Client received from Server: " + charBuffer.toString());
        socketChannel.close();
        println("Client closed socketChannel");
     } else { println("The connection cannot be established!"); }
     println("Client finished");
  }
}
```

Beispiel TCP-Echo-Client Version 1 – 2

```
// receive and reply messages from server:

while (socketChannel.read(buffer) != -1 && i < 10) {
   println("Finished read");
   buffer.flip();
   val charBuffer = decoder.decode(buffer);
   println("Client received from Server: " + charBuffer.toString());
   if (buffer.hasRemaining()) {
        buffer.compact();
   } else {
       buffer.clear();
   }
   i = i+1
   println("Client will send Hallo Nr " + i);
   buffer.put( ("Hallo Nr " + i).getBytes());
   socketChannel.write(buffer);
   println("Client will read");
}
```

Beispiel TCP-Echo-Client Version 2 – 1

Version 2: Sendet Nachrichten, die von der Konsole eingelesen werden.

```
import java.io.BufferedReader;
import java.io.InputStreamReader;
import java.net.InetSocketAddress;
import java.net.SocketAddress;
import java.nio.ByteBuffer;
import java.nio.channels.SocketChannel;
import java.nio.charset.Charset;

object NioTcpClient_V2_Main extends App {
  val ECHO_PORT = 4713
  val socketAddr  = new InetSocketAddress("127.0.0.1", ECHO_PORT)
  val buf = ByteBuffer.allocate(256)
  val in = new BufferedReader( new InputStreamReader(System.in))
  val decoder = (Charset.forName("UTF-16")).newDecoder()

  val socketChannel = SocketChannel.open()
  if (!socketChannel.isOpen()) { System.exit(-1) }
  socketChannel.configureBlocking(true)
  socketChannel.connect(socketAddr)
  var s : String = null

  println("enter lines to be sent (stop with empty line)")
  // read user input send it to server
  // receive and print answer
  while (true) {
     . . .
  }
}
```

Beispiel TCP-Echo-Client Version 2 – 2

```
// read user input, send it to server,
// receive and print answer
while (true) {

    s = in.readLine() // read user input
    if (s == null || s.length() == 0) { System.exit(0) }
    s = s +"\n";

    val b = s.getBytes("UTF-16");          // characters -> bytes (UTF-16 Format)
      buf.put(b);                          // Bytes in buffer
      buf.flip();
      println("number sendable bytes "+buf.remaining());
      val nw = socketChannel.write(buf);  // send bytes
      println( "number of bytes sent "+nw );
      buf.clear();

      val n = socketChannel.read(buf);  // read Bytes
      println( " "+ n +" bytes hve been read" );
      if ( n > 0 ) {
        buf.flip();
        val message = (decoder.decode(buf)).toString();  // bytes -> characters
        println("Echo> " + message);
        buf.compact();
      } else if ( n == -1 ) {
        println("This the end!");
        System.exit(0)
      }
   }
```

Wichtig zu wissen

Ernsthafte Server-Implementierungen in Java nutzen NIO

NIO ist nicht einfach in der Bedienung

Anwendungsprogrammierer sollten

- **NIO kennen**
- **und i.A. nicht direkt nutzen, sondern über ein Framework oder sehr gut getestete Bibliotheksfunktionen.**

Literaturhinweis (immer noch aktuell !):
Ron Hitchens: *Java NIO*
O'Reilly 2002