

# Einführung in die Java-API „NIO“

Softwarepraktikum „Datenstrukturen“

Sommersemester 2004

## Festplattenbasierte Datenverwaltung

- Blockorientierte Speicherstrukturen
  - Festplattenzugriffe im Vergleich zu Hauptspeicherezugriffen sehr teuer
  - Ziel daher: Minimierung der Festplattenzugriffe
  - Festplatte organisiert in Blöcken fester Größe (z.B. 4 kByte)
  - Blöcke werden immer als Ganzes in Hauptspeicherseiten kopiert
- Zugriffe auf die Festplatte
  - Direkter Blockzugriff mit „normaler“ Programmiersprache nicht möglich
  - Programmiersprache kennt nur Dateien; Betriebssystem ist für Festplattenzugriffe zuständig
- Alternative (zu Übungszwecken)
  - Datei wird wie Festplatte verwaltet
  - Lesen und Schreiben von Dateiteilen der Blockgröße  $b$
  - Verwaltung von Puffervariablen (Bytestrings der Länge  $b$ )

## Blockorientierte Datenverwaltung in Java

- vor Java Ver. 1.4 problematisch
  - blockweises Einlesen von Dateiteilen zwar möglich
  - Verwaltung eines entsprechenden Hauptspeicherpuffers jedoch umständlich
  - Problem: strenge Typisierung in Java
  - Umwandlung primitiver Datentypen (int, float, ...) in (untypisierte) Bytestrings nicht direkt möglich
- seit Java Ver. 1.4: neues API „NIO“ (*new I/O*)
  - Java – Dateien bzw. –Ströme können als sog. *Channel* verwendet werden
  - Spezielle Operationen zum Transfer von Daten zwischen Channel und Hauptspeicherpuffer
  - Neuer Datentyp *ByteBuffer* gestattet direktes Ein- und Auslesen primitiver Datentypen

## Blockorientierte Datenverwaltung mit NIO

- Betriebssystemdatei als Java-RandomAccessFile *raf* deklarieren
- Channel zu dieser Datei öffnen (*raf.getChannel()*)
- ByteBuffer-Variable *bufvar* deklarieren und Speicherplatz der Blockgröße *b* allokieren
- Dateizeiger von *raf* auf Vielfache von *b* positionieren (*position*) und Dateiteile der Länge *b* in *bufvar* einlesen (*read*)
- Freie Verwaltung der Puffervariablen im Hauptspeicher
  - Lesen und Schreiben primitiver Typen mit entsprechenden *get-* und *put-*Methoden
  - Ursprünglicher Typ geht beim Schreiben in die Puffervariable verloren!
  - Blockorganisation/-struktur wird von der Anwendung festgelegt („hart verdrahtet“)
- Zurückschreiben des Pufferinhalts in die Datei (*write*)

## NIO-Channels und ByteBuffers

```
import java.io.*;
import java.nio.channels.*;
import java.nio.*;
...
// Direktzugriffsdatei anlegen und Channel öffnen
FileChannel chan = (new RandomAccessFile(„raf.txt“, „rw“)).getChannel();

// Puffervariable mit entsprechendem Speicherplatz (Blockgröße) allokalieren
ByteBuffer bufvar = ByteBuffer.allocate((int)blockSize);

// Channel auf n-ten Block positionieren
chan.position ((long)(n*blockSize));

//Block in den Puffer einlesen
int gelesen = chan.read(bufvar);
```

## ByteBuffer manipulieren

```
// An den Positionen 7, 23, 47 Daten in den Puffer eintragen
bufvar.putChar(7, 'a');
bufvar.putInt(23, 123);
bufvar.putFloat (47, 66.6)

// Daten auslesen
char myChar = bufvar.getChar(7);
int myInt = bufvar.getInt(23);
float myFloat = bufvar.getFloat(47);

// Block in den Channel zurückschreiben (direkte Positionierung)
int geschrieben = chan.write(bufvar, (long)(n*blockSize));

// Channel schließen
chan.close();
```

## Weitere Informationen zu NIO

- Weit mehr Funktionalität als hier vorgestellt
  - Typisierte Puffer (Array als Puffer)
  - Verbundene Puffer (synchronisiert)
  - Puffer-Views
  - Automatische Synchronisation von Puffer und Channel
  - Dateisperren
  - Pattern Matching mit regulären Ausdrücken
  - ...
- Literatur
  - Sun Java-Dokumentation
  - <http://www.dickbaldwin.com/>
  - Ron Hitchens: Java NIO, O'Reilly & Associates 2002.