ALPEN-ADRIA UNIVERSITÄT
KLAGENFURT | WIEN GRAZ

ESOP - Lists, Collections, Java IO

Assoc. Prof. Dr. Mathias Lux
ITEC / AAU

ALPEN-ADRIA UNIVERSITÄT
KLAGENFURT | WIEN GRAZ

Quellen / Sources

- Mössenböck (2014) Sprechen Sie Java
- The Java Tutorials:
  - Collections: http://docs.oracle.com/javase/tutorial/collections/index.html
  - IO: https://docs.oracle.com/javase/tutorial/essential/io/

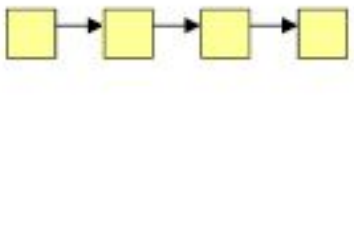
- Sierraq & Bates (2005) Head First Java
  - Collections: p.132+, p.532+, p.558+, ...
  - I/O: p. 452+

Dynamische Datenstrukturen

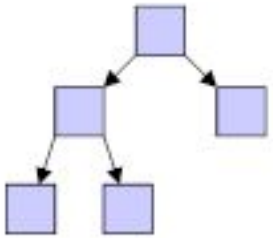
- Elemente werden zur Laufzeit angelegt
  - mit *new*, sozusagen „dynamisch“
- Datenstruktur kann dynamisch
  - wachsen: Speicher wird belegt
  - schrumpfen: Speicher wird frei

Dynamische Datenstrukturen

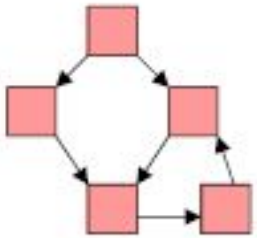
- Wichtige dynamische Datenstrukturen



Liste



Baum



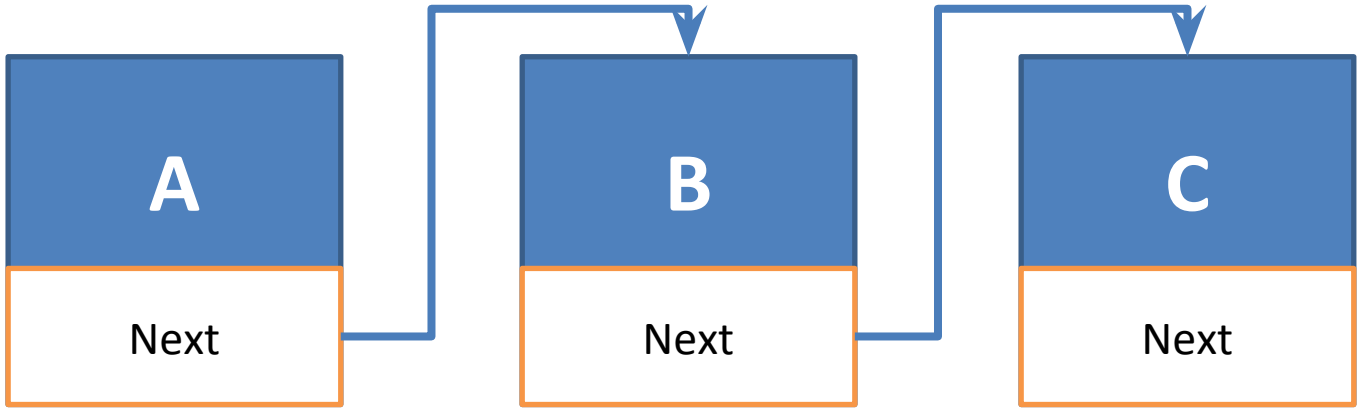
Graph

Dynamische Datenstrukturen

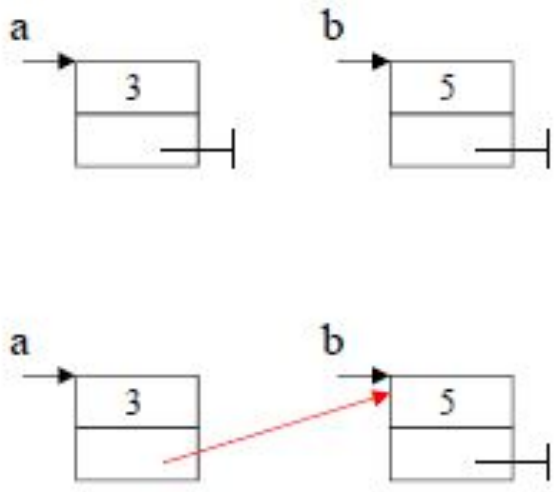
Bestehen aus Knoten die über Kanten miteinander verbunden sind.

- Knoten … Objekte
- Kanten … Zeiger / Referenzen

Verknüpfen von Knoten

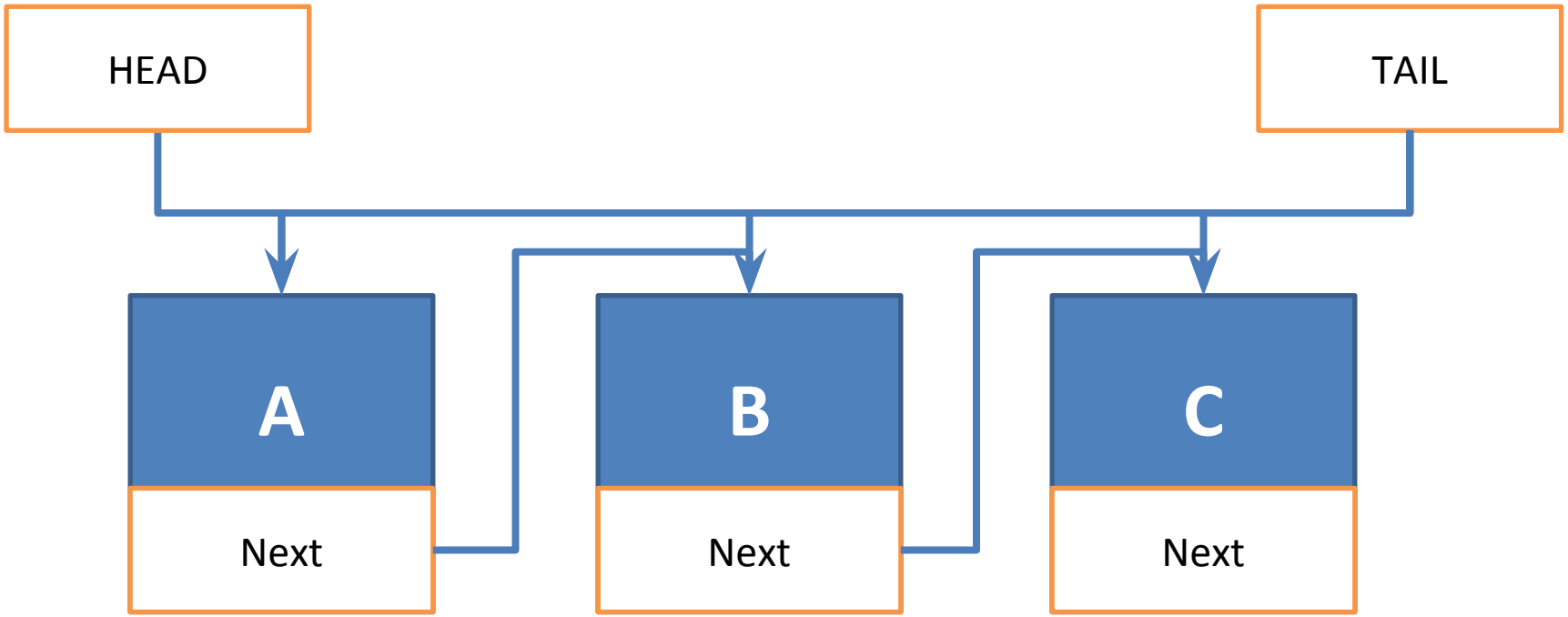


Verknüpfen von Knoten.

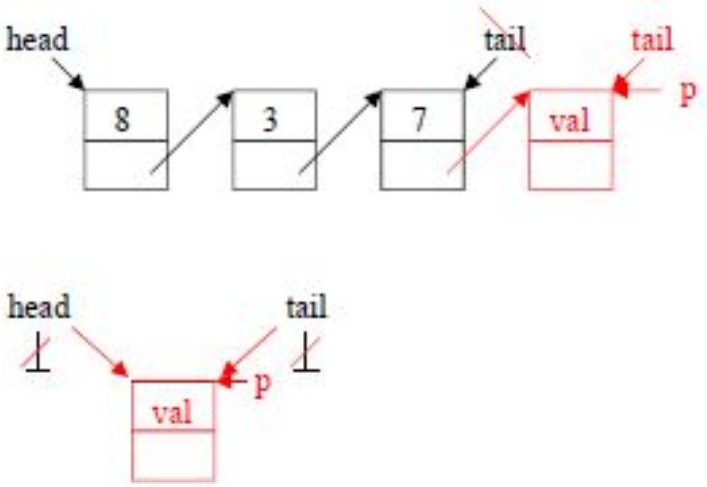



```
public class Node {
    int value;
    Node nextNode;

    public Node(int value) {
        this.value = value;
    }
}

// ...
Node a = new Node(3);
Node b = new Node(5);
a.nextNode = b;
```

Einfügen am Listenende

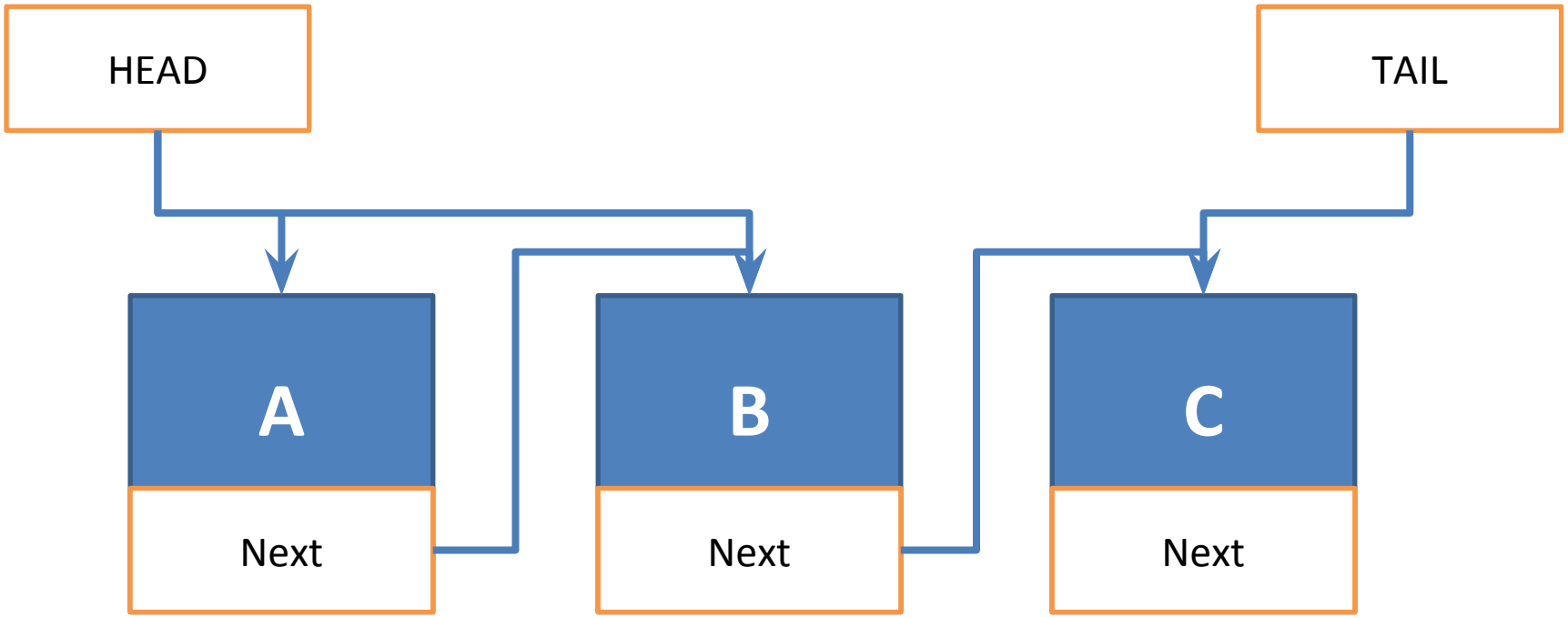


Unsortierte Liste: Einfügen am Listenende

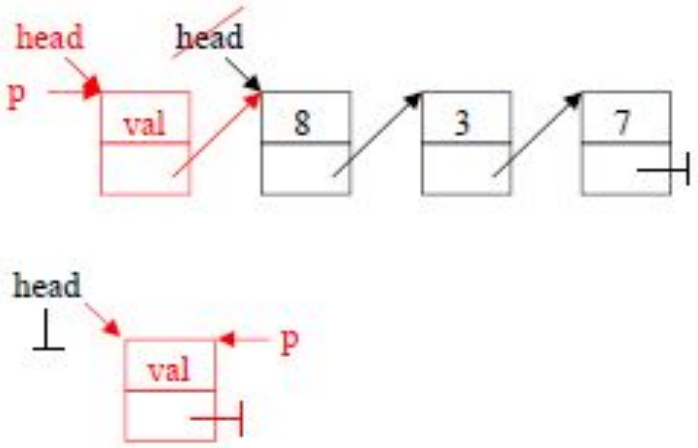



```
public class List {
    private Node head, tail;

    public void append(int val)
    {
        Node p = new Node(val);
        if (head == null)
            head = p;
        else
            tail.nextNode = p;
        tail = p;
    }
    //...
}
// ...
List l = new List();
l.append(3);
l.append(4);
```

Einfügen am Listenanfang



Unsortierte Liste: Einfügen am Listenanfang




```
public class List {
    private Node head, tail;

    public void prepend(int val) {
        Node p = new Node(val);
        p.nextNode = head;
        head = p;
    }
    // ...

// ...
List l = new List();
l.prepend(3);
l.prepend(4);
```

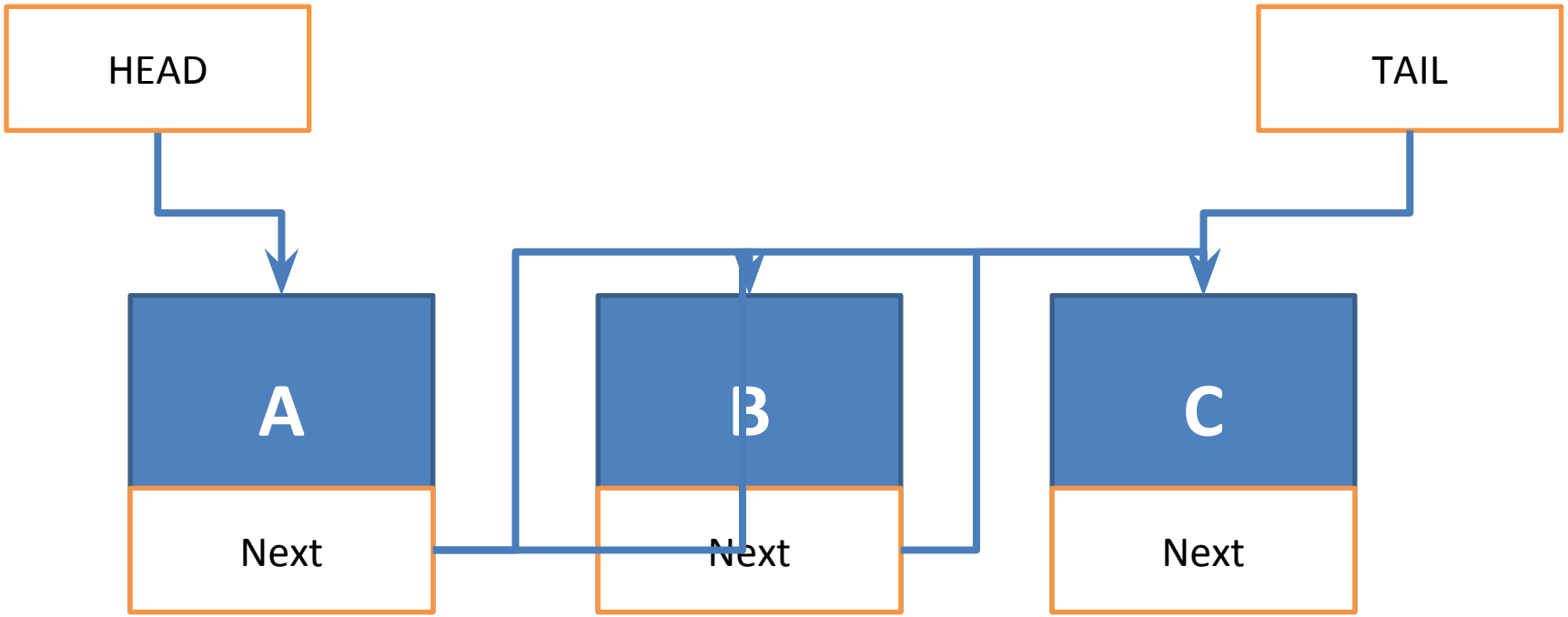
Unsortierte Liste: Eintrag suchen

```
public class List {
    private Node head, tail;

    public boolean contains(int val) {
        Node p = head;
        boolean result = false;
        while (p!=null) {
            if (p.value == val) result = true;
            p = p.nextNode;
        }
        return result;
    }
     // ...
}
// ...
List l = new List();
l.append(3);
l.append(14);
l.append(-1);
System.out.println(l.contains(3));
```

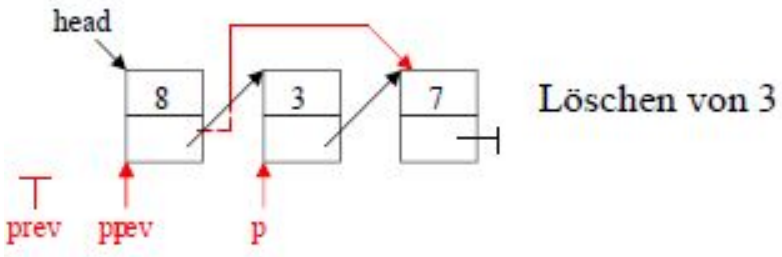
Unsortierte Liste: Eintrag suchen

```
public class List {
    private Node head, tail;

    public boolean contains(int val) {
        Node p = head;
         while (p!=null && p.value!=val) {
            p = p.nextNode;
        }
        return p != null;
    }
    // ...
}
// ...
List l = new List();
l.append(3);
l.append(14);
l.append(-1);
System.out.println(l.contains(3));
```

Unsortierte Liste: Eintrag löschen



Unsortierte Liste: Eintrag löschen




```
public class List {
    private Node head, tail;

    public void delete(int val) {
        Node p = head, prev = null;
        while (p!=null && p.value!=val) {
            prev = p;
            p = p.nextNode;
        }
        if (p != null) {
            if (p == head)
                head = p.nextNode;
            else
                prev.nextNode = p.nextNode;
            if (tail == p)
                tail = prev;
        }
    }
    // ...
}
```

Live Beispiel …

- Methode index(int val)

- Stack & Queue mit dynamischen Datenstrukturen.

Java Collections Framework

- Collection := Objekt, das andere Objekte gruppiert
- Collections werden benutzt um mit Daten umzugehen
  - speichern, suchen, ändern, verteilen, ...
- Ein Framework von
  - Interfaces
  - Implementierungen
  - Algorithmen

Collection Interfaces

- Set ... Menge
  - kann ein Element nur einmal enthalten
- List ... Liste, Sequenz
  - hat Ordnung, kann Duplikate enthalten
- Queue ... Warteschlange
  - zum Abarbeiten
- Map ... Zuordnung Name -> Wert
  - Namen sind in einer Menge (s.o.)

Interface Collection

- Hinzufügen und Löschen
  - `add(..), addAll(..), remove(..), removeAll(..)`
- Überprüfen
  - `contains(..), containsAll(..), isEmpty(..)`
- Durchgehen der enthaltenen Elmente
  - `iterator(..)`
- Und mehr ...
  - `size(), clear(), toArray(..)`

Interface Iterator

- Geht eine Collection durch indem jedes Elemement besucht wird.
- Ordnung ist nur garantiert wenn die Collection-Implmentierung die Ordnung garantiert (z.b. LinkedList, ArrayList)
- Methoden:
  - `hasNext()` - liefert `true` wenn noch ein Element existiert
  - `next()` - liefert das nächste Element

Beispiel Iterator

```
// New ArrayList with an initial size of 12 elements:
ArrayList arrayList = new ArrayList(12);
// Now add some random int values to the list:
for (int i = 0; i < 40; i++) {
    arrayList.add((int) (Math.random()*100000));
}
// Use an iterator to visit every element:
Iterator iterator = arrayList.iterator();
while (iterator.hasNext()) {
    Object next = iterator.next();
    System.out.println(next);
}
```

Iterator wird benutzt um jedes Element der Liste auszugeben

```
// New ArrayList with an initial size of 12 elements:
ArrayList arrayList = new ArrayList(12);
// Now add some random int values to the list:
for (int i = 0; i < 40; i++) {
    arrayList.add((int) (Math.random()*100000));
}
// Use an iterator to visit every element:
for (Iterator iterator = arrayList.iterator(); iterator.hasNext(); ) {
    Object next = iterator.next();
    System.out.println(next);
}
```

Gleich wie oben, nur kürzer und iterator ist nach der Schleife nicht mehr bekannt

ALPEN-ADRIA UNIVERSITÄT
KLAGENFURT | WIEN GRAZ

Interface List

- Leitet von Interface Collection ab
- Erweitert um Listenfunktionen, zB.:
  - `add(index, Element)` - Element an Stelle einfügen
  - `set(index, Element)` - Ersetzt bestehenden Eintrag an Stelle index durch Element
  - `remove(index)` - Element an Stelle index entfernen
  - `indexOf(Element)` - Sucht Stelle von Element in Liste

Ausgewählte Klassen

- LinkedList
  - Verkettete Liste, Ein Element zeigt auf das nächste, vgl. Beispiele von Listen in VO.
- ArrayList
  - Liste auf Basis von Array, Array wird vergrößert wenn zu wenig Platz.
- HashSet
  - Implementierung des Interface Set, wird typischerweise für Set verwendet.
- HashMap
  - Implementierung des Interface Map, speichert Schlüssel-Wert-Paare (key value pairs).
  - Bsp. `map.put("name", "Otto"); name.get("name");`

Wiederholung: Autoboxing

- Collections können nur mit Objekten umgehen:
  - `mySet.add(new Integer(5));`

- Basisdatentypen werden automatisch in Objekte verpackt:
  - `mySet.add(5); // gleich w.o.`

Cp. Programmiersprache C#

Collections: Set

- Beispiele
  - Set intersection – Schnittmenge
  - Set union – Vereinigungsmenge
- Cp. SetFun @ git

Java I/O

- Java input & output
  - `java.io` Paket
  - `java.nio.file` Paket
- Input & Output nutzt Streams
  - Byte Streams - Binärdaten.
  - Character Streams – char-basierte Daten.
  - Buffered Streams – Nutzung von Buffern.
  - Data Streams - Basisdatentypen & String
  - Object Streams - Binär-I/O von Objekten.

Byte Streams

- InputStream & OutputStream
  - Lesen per byte (= 8 bit)
- Alle Byte-Streams leiten davon ab.
  - FileInputStream – liest von Datei
  - ByteArrayInputStream – liest von byte[]
  - System.in – „standard“ in

InputStream

- InputStreamFun - Arbeit mit Buffern
- ByteArrayInputExample – read int vs. byte

`abstract int`	**`read()`** Reads the next byte of data from the input stream.
`int`	**`read(byte[] b)`** Reads some number of bytes from the input stream and stores them into the buffer array `b`.
`int`	**`read(byte[] b, int off, int len)`** Reads up to `len` bytes of data from the input stream into an array of bytes.

Characters & Buffer

- Character I/O mit Reader & Writer
  - Analog zu byte, aber mit char
  - char <-> x*byte
  - Beispiel: Unicode-Text
- Buffered I/O
  - Interner Buffer
  - Methode flush() beim Output

Example Reader vs. Stream

```
// String with a Unicode character:
String uString = "Victory! ✌ ä";
// two different ways to read it:
ByteArrayInputStream bis = new
    ByteArrayInputStream(uString.getBytes());
CharArrayReader car = new CharArrayReader(uString.toCharArray());
// show the difference
int read = -1;
do {
    read = bis.read();
    System.out.printf("%c (%03d) - %c\n", (char) read ,
    read,(char) car.read());
} while (read > -1);
```

java.io.File

- Klasse für den Umgang mit Dateien
  - Verzeichnisse sind auch Dateien
- Methoden erlauben
  - Existenzprüfung, Rechteprüfung
  - Auflistung der Kind-Dateien
  - Verzeichnisse anlegen
  - uvm.

java.io.FileReader & java.io.FileInputStream

- Erweitert Reader (abstrakte Klasse)
- Liest Zeichen (char) aus File
  - Vgl. FileInputStream -> byte
  - Unterschied zwischen char & byte!
- Nutzung benötigt einen Buffer
  - char[] für Reader, byte[] für InputStream

java.io.BufferedReader

- Stellt Buffer zur Verfügung
- Liefert ganze Zeilen zurück

- ACHTUNG: Ist abgeleitet von Reader -> char[]!

Schreiben in Dateien?

- FileWriter, BufferedWriter
  - Nicht auf close() bzw. flush() vergessen!

Und:

- ObjectStreamWriter, ObjectStreamReader
  - Schreibt und liest Objekte von einem InputStream
- GZipInputStream, GZipOutputStream
  - Komprimiert I/O Streams