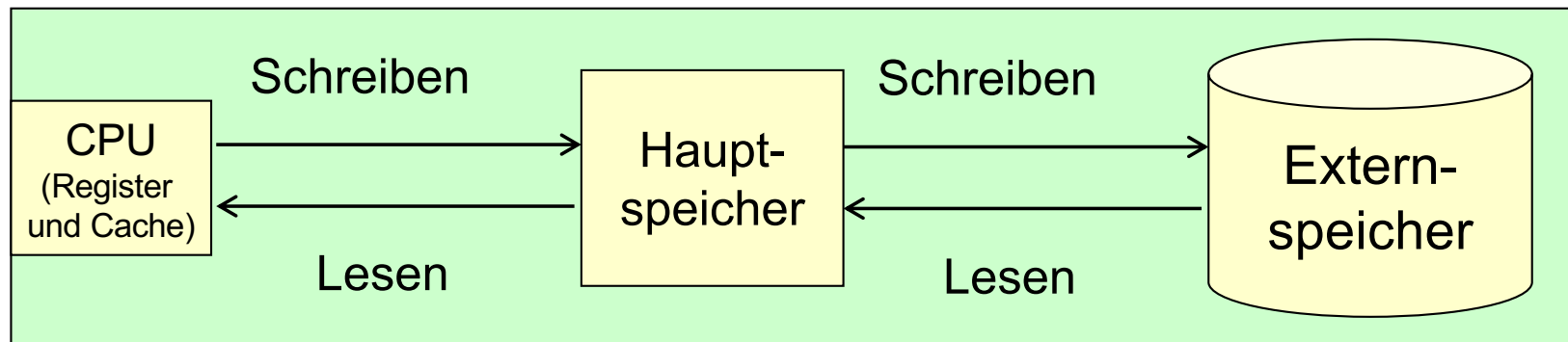


14. Ein- und Ausgabe

- Dateien
- Die Klasse `java.io.File`
- Binärdateien und die Klasse `java.io.RandomAccessFile`
- Datenströme
- Klassenhierarchie



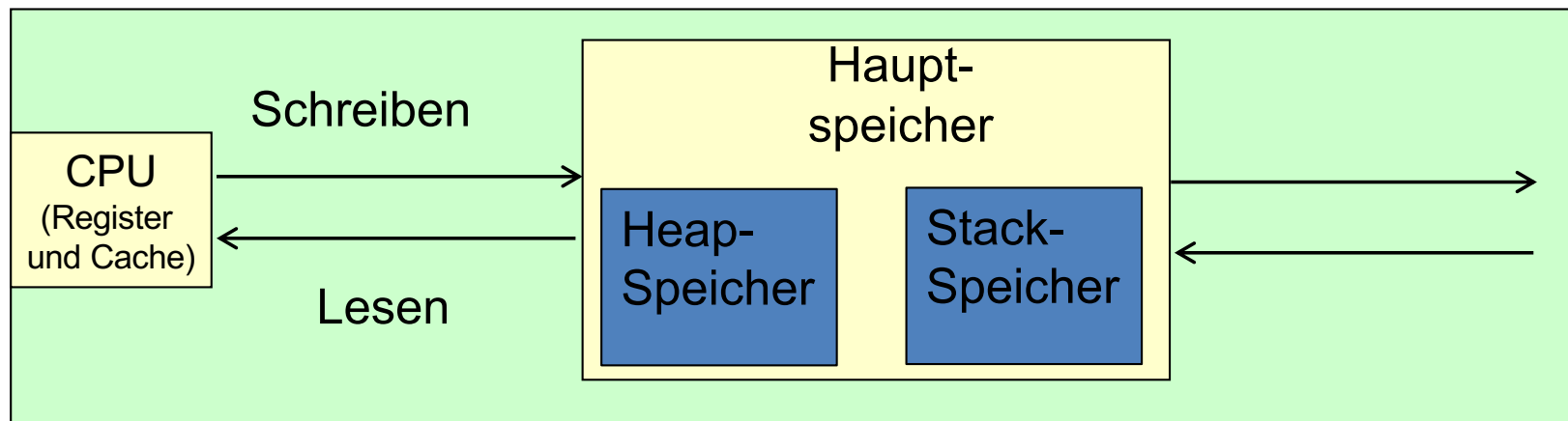
Einfaches Modell eines Computers



- CPU (central processing unit):
 - ist eine physikalische Einheit, die eine Menge von Befehlen verarbeiten kann
 - Hierfür benötigte Daten werden aus dem Hauptspeicher gelesen.
- Hauptspeicher ist ein Speichermedium mit folgenden Eigenschaften:
 - schneller Zugriff
 - relativ teuer
 - **flüchtig**: Verlust des Inhalts beim Abschalten des Computers
- Externspeicher (z. B. Festplatte, CD, Speicherstick,...)
 - langsamer Zugriff
 - relativ billig
 - **stabil**: kein Verlust des Inhalts beim Abschalten

Speicherverwaltung von Java

- Java verwaltet alle Variablen und Objekte im Hauptspeicher eines Computers



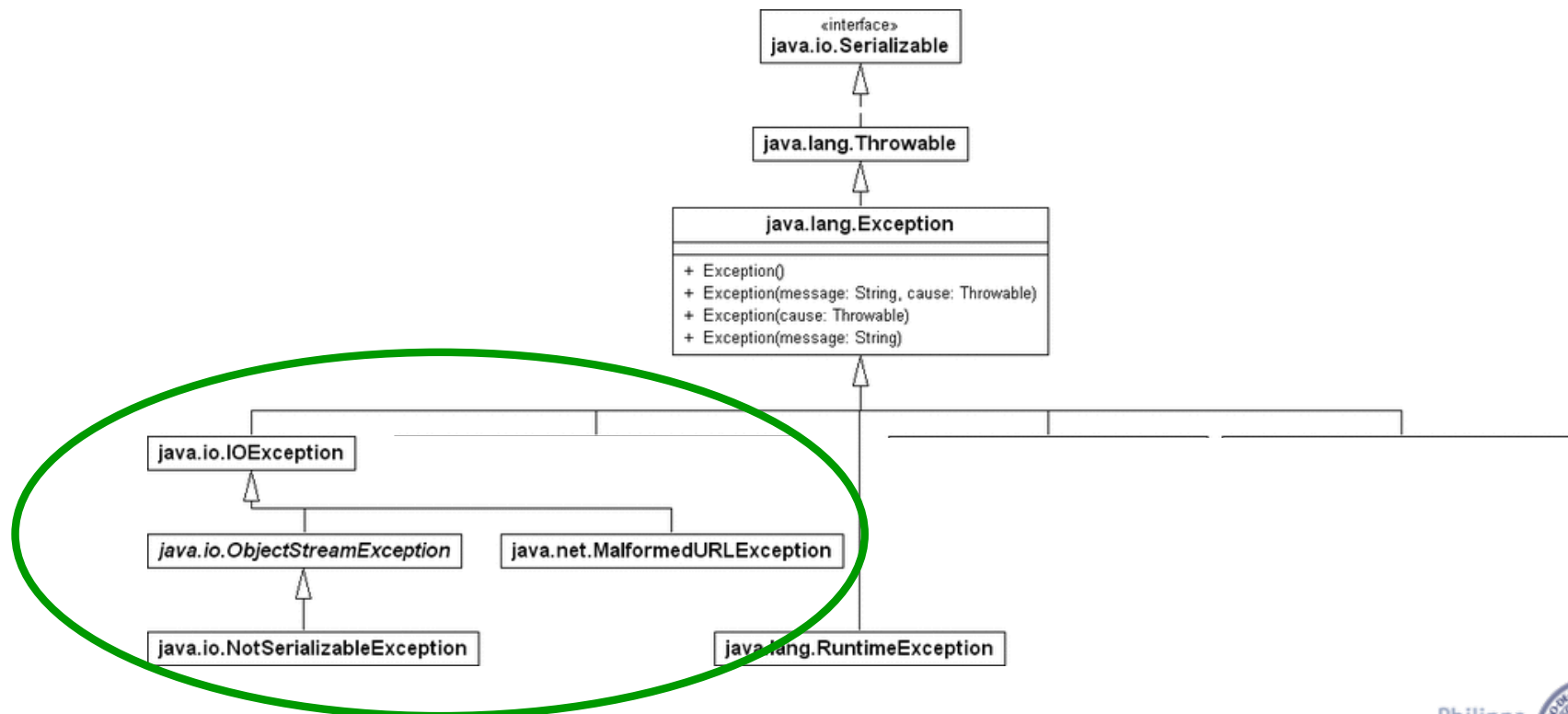
- Konsequenz
 - Beim Programmende wird der vom Programm belegte Speicherplatz im Hauptspeicher freigegeben.
- Programmende \Rightarrow Verlust der Werte aller Variablen und Objekte

Persistierung der Daten

- Wie kann man in Java es erreichen, dass
 - wichtige Daten auch noch nach dem Programmende existieren
 - und beim nächsten Programmlauf die Daten wieder im Programm vorhanden sind?
- Lösung
 - Rettung der Daten durch **Abspeichern auf dem Externspeicher**
 - Externspeicher „**überlebt**“ das Abschalten des Computers
 - Daten auf dem Externspeicher werden nicht nach dem Programmende vernichtet, sondern **bleiben** darüber hinaus **verfügbar**.
 - Einlesen der Daten beim nächsten Programmlauf
- Problem
 - Speicherverwaltung muss selbst programmiert werden.

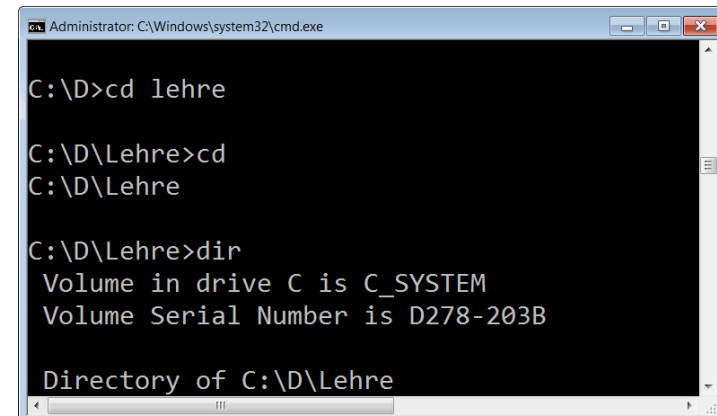
Exceptions für java.io

- Die Oberklasse IOExceptions und deren Unterklassen stellen geprüfte Ausnahmen für das Paket java.io zur Verfügung.
 - Diese Exceptions **müssen** gefangen oder weitergereicht werden!



14.1 Dateien und die Klasse File

- Daten werden auf dem Externspeicher in **Dateien** gespeichert.
- Ein **Dateisystem** verwaltet die Dateien auf dem Externspeicher.
- Es gibt verschiedene Schnittstellen, um das Dateisystem anzusprechen.
 - Die **Kommandozeile** unter Windows und Linux stellt Befehle zur Verfügung, um mit Dateien zu arbeiten.
 - Eine andere Möglichkeit ist die Benutzung eines **Datei-Explorers**.
 - Die Programmiersprache Java bietet mit der **Klasse File** ebenfalls eine Möglichkeit Befehle des Dateisystems aufzurufen.



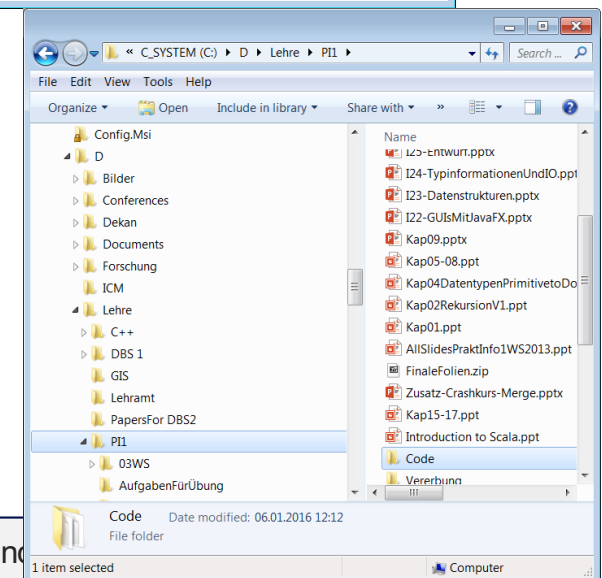
```
Administrator: C:\Windows\system32\cmd.exe

C:\D>cd lehre

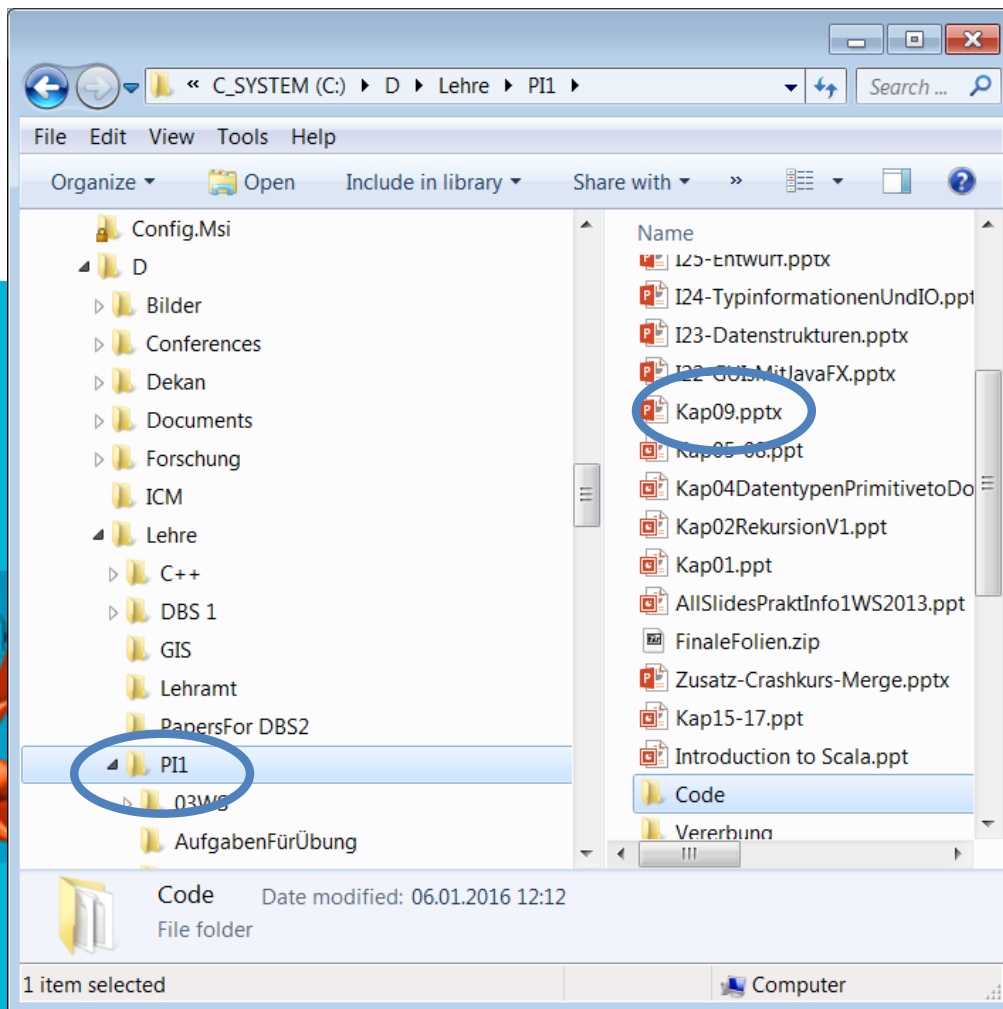
C:\D\Lehre>cd
C:\D\Lehre

C:\D\Lehre>dir
Volume in drive C is C_SYSTEM
Volume Serial Number is D278-203B

Directory of C:\D\Lehre
```



Organisation des Externspeichers



- Daten auf dem Externspeicher werden in einem Dateisystem verwaltet.
 - Daten liegen in Dateien
 - z. B. Datei Kap09.pptx
 - Verzeichnisse werden genutzt, um Daten logisch aufzuteilen.
 - Verzeichnis PI1
- Eine Datei ist ein Behälter (potentiell beliebiger Größe)
 - Vor dem Gebrauch muss dieser Behälter **geöffnet** werden.
 - Danach kann man aus dem Behälter Daten **lesen** bzw. in den Behälter Daten **schreiben**.
 - Hat man alles erledigt, sollte eine Datei wieder **geschlossen** werden.

Dateinamen

- Der vollständige Dateiname setzt sich aus zwei Teilen zusammen:
 - **lokaler** Dateiname
 - ist eindeutig innerhalb des Verzeichnisses, dem die Datei zugeordnet ist.
 - **Pfadname**
 - Verknüpfung aller Verzeichnisnamen beginnend vom Wurzelverzeichnis bis zu dem Verzeichnis, in dem die Datei liegt.
- Abhängig vom Betriebssystem wird zwischen den einzelnen Teilen eines vollständigen Dateinamens ein **Separatorzeichen** verwendet:
 - DOS: „\“
 - UNIX: „/“
- Der Name des **Wurzelverzeichnisses** („/“) besteht nur aus dem Separatorzeichen **und im Fall von Windows dem Laufwerk**.
- Beispiel:
 - lokaler Dateiname: “Brief.Key”
 - vollständiger Dateiname (Windows): “C:\Cafe\Bin\Keys\Brief.Key”

Die Klasse File im Paket java.io

- File ist eine Klasse zum Zugriff auf einen baumstrukturierten Dateikatalog
 - Die Repräsentation abstrahiert vom zugrundeliegenden Betriebssystem.
- Konstruktoren
 - **public** File(String path);
 - **public** File(String path, String name);
 - **public** File(File dir, String name);
- Konstante
 - **public final static** String separator;
- Eine Auswahl von wichtigen Objektmethoden
 - **public** String getPath(); // Resultat: Pfadname
 - **public** String getParent(); // Resultat: Namen des Elternverzeichnis
 - **public boolean** isDirectory(); // Resultat: true, falls Datei ein Verzeichnis ist.
 - **public long** length(); // Resultat: Länge der Datei (in Bytes).
 - **public boolean** exists(); // Resultat: true, falls zu diesem Dateinamen
// tatsächlich eine Datei existiert.
 - **public** String[] list(); // Liste der Dateinamen für ein Verzeichnis

Systemabhängige Konstante für
das Separatorzeichen als String.

Anwenden der Methoden

```
import java.io.File;

public class FileTest {
    public static void main(String[] args) throws Exception {
        File file;

        file = new File(args[0]);
        System.out.println("file.getPath()           : " + file.getPath());
        System.out.println("file.getCanonicalPath() : " + file.getCanonicalPath());
        System.out.println("file.getParent()       : " + file.getParent());
        System.out.println("file.isDirectory()     : " + file.isDirectory());
        System.out.println("file.length()          : " + file.length());
        System.out.println("file.exists()          : " + file.exists());
        System.out.println("file.list().length     : " + file.list().length);
        System.out.println("file.list()[0]         : " + file.list()[0]);
    }
}
```

Beispiel

- **Aufgabenstellung**

Implementieren Sie eine Methode, die zu einem vorgegebenen **Namen eines Verzeichnisses** V und einem **Suffix** S, rekursiv alle Namen der Dateien ausgibt, die in V liegen und den Suffix S besitzen.

- Vorschlag für eine Implementierung:

```
import java.io.File;

public class FSearch {
    public static void search (File dir, String suffix) { ... }

    public static void main(String[] arg) throws Exception {
        File start = new File(arg[0]);
        if (start.exists() && start.isDirectory())
            search(start, "." + arg[1]);
        else
            System.out.println("Falsche Eingabe!");
    }
}
```

Der Rumpf der Methode search

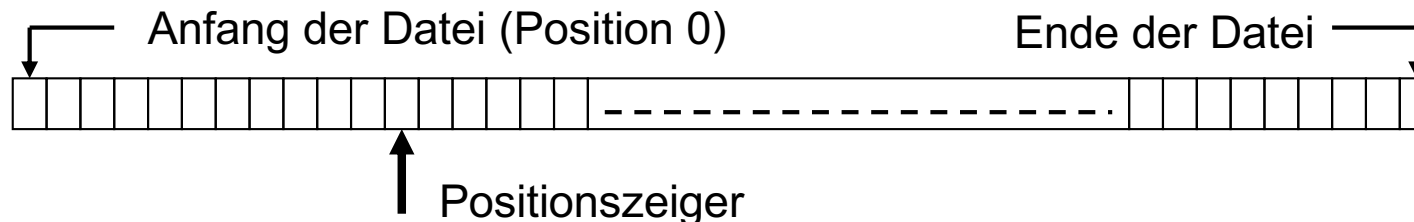
```
public static void search (File dir, String suffix) {  
    File next;  
    String[] list = dir.list();  
    int count = 0;  
  
    for (int i = 0; i < list.length; i++) {  
        if (list[i].endsWith(suffix)) {  
            // endsWith ist eine Methode der Klasse String  
            if (count == 0)  
                System.out.println("Directory: " + dir.getPath());  
            count++;  
            System.out.println(" " + list[i]);  
        }  
    }  
    for (int i = 0; i < list.length; i++) {  
        next = new File(dir, list[i]);  
        if (next.isDirectory())  
            search(next, suffix);  
    }  
}
```

Ausgabe der Dateinamen

rekursiver Aufruf

14.2 Wahlfreier Zugriff in Dateien

- Eine **Datei** ist eine Datenstruktur zur **Speicherung** von Daten auf dem **Externspeicher**.
 - Eine Datei **entspricht** dabei einer **Reihe von Werten mit dem Datentyp byte**. Zusätzlich gibt es einen **Positionszeiger**.
 - Eine Datei ermöglicht das Lesen an und Schreiben von der **Stelle**, wo der Positionszeiger gerade steht.



- Eine Datei kann erweitert werden, indem am Ende neue Daten hinzugefügt werden.

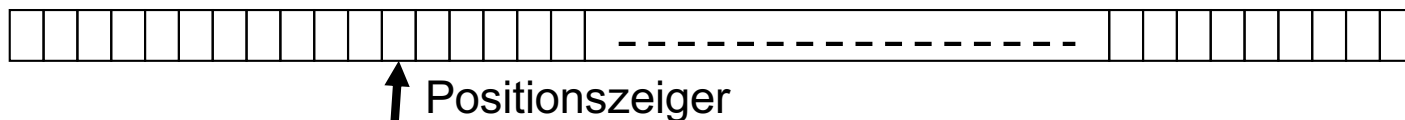
Öffnen und Schließen von Dateien

- Bevor eine Operation ausgeführt wird, muss eine Datei unter Angabe eines Namens geöffnet werden.
 - Hierbei werden Hilfsstrukturen im Hauptspeicher aufgebaut.
- Wird eine Datei nicht mehr benötigt, sollte man die Datei explizit schließen.
 - Hilfsstrukturen werden abgebaut und Speicherplatz freigegeben.
 - Danach muss die Datei wieder geöffnet werden, um wieder Operationen auf der Datei auszuführen.

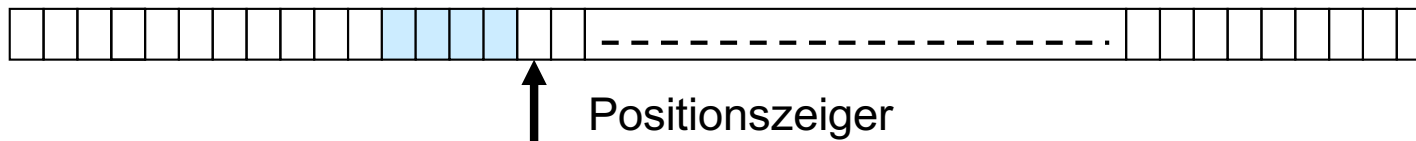
Schreiben in eine Datei

- Der **Positionszeiger** muss zunächst auf die Stelle **gesetzt** werden, wo Daten geschrieben werden sollen.

 zu schreibende Daten



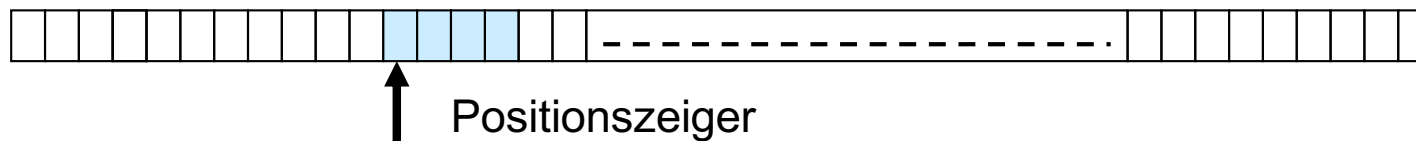
- Nach dem Schreiben zeigt der Positionszeiger auf die Stelle in der Datei, die der zuletzt geschriebenen Stelle folgt.



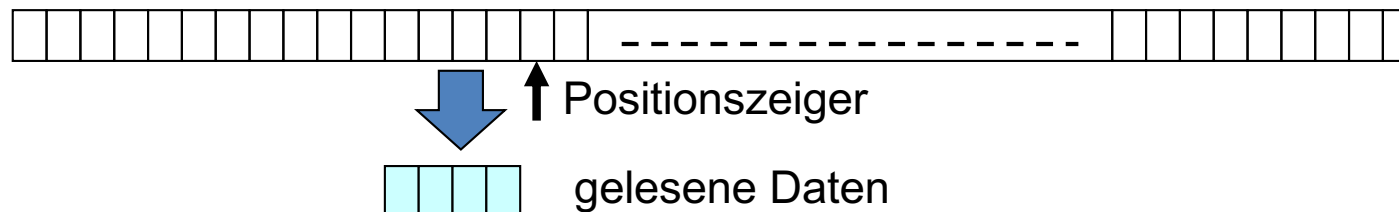
- Wird beim **Schreiben** das Ende der Datei überschritten, so wird die Datei dynamisch um eine entsprechend große Anzahl von Bytes **verlängert**.

Lesen aus einer Datei

- Der **Positionszeiger** muss zunächst auf die Stelle **gesetzt** werden, wo Daten geschrieben/gelesen werden sollen.



- Danach kann der Inhalt in den Hauptspeicher übertragen werden.
 - Der Positionszeiger steht hinter der zuletzt gelesenen Stelle.



- Beim Lesen darf das Ende der Datei nicht überschritten werden.
 - Ansonsten wird eine Ausnahme ausgelöst.

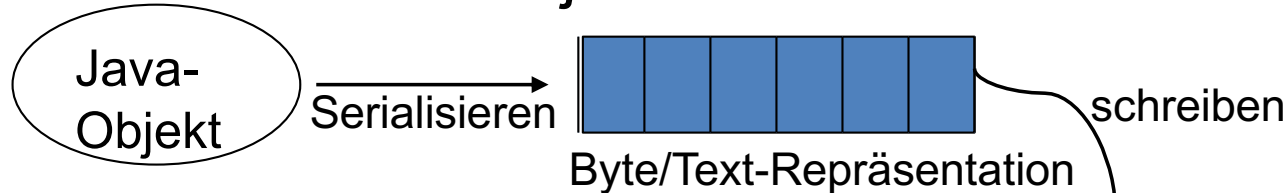
Binär- und Textdateien

- Dateizugriff wird unterschieden, in welcher Form die Daten geschrieben/gelesen werden:
 - Daten können direkt in ihrem **Binärformat** in eine Datei geschrieben werden. Solche Dateien werden auch als **Binärdateien** bezeichnet
 - Werden Daten zuerst in einen **String** transformiert und wird dann der String in eine Datei geschrieben, so spricht man von einer **Textdatei**.

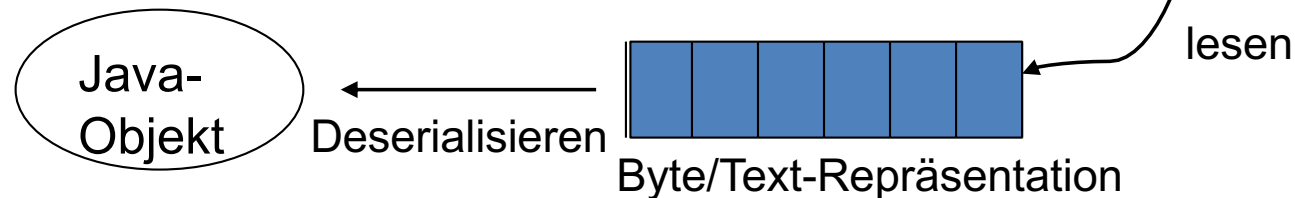


Operationen auf Dateien

- Schreiben eines Objekts in eine Datei



- Lesen aus einer Datei



- Bei einer Textdatei wird eine Serialisierung in Text und bei einer Binärdatei in Byte vorgenommen

Die Klasse RandomAccessFile

- Die Klasse stellt Dateien zur Verfügung, die den wahlfreien Zugriff auf Dateien unterstützen.
 - Zugriffsmethoden und Datenfelder zur Manipulation des Positionszeigers

Konstruktoren

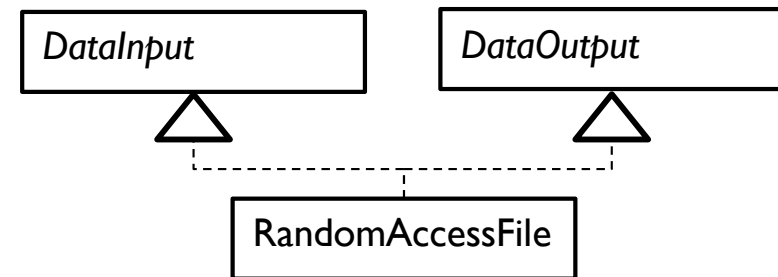
- **public** RandomAccessFile(String name, String mode);
 - Der Parameter *mode* zeigt an, ob auf die Datei nur **lesend** (*mode* == "r") oder **auch schreibend** zugegriffen werden kann (*mode* == "rw").
 - Der Parameter *name* enthält den **Dateinamen** (relativ zum Verzeichnis, wo das Programm gestartet wird oder absolut)
 - Nach einem Aufruf eines Konstruktors ist die **Datei geöffnet**, d.h. Operationen können ausgeführt werden. Der **Positionszeiger** steht **am Anfang** der Datei.
- **public** RandomAccessFile(**File** handle, String mode);
 - In diesem Konstruktor wird statt einem String eine Referenz eines Objekts der Klasse File übergeben.

Methoden zum Positionieren

- Lesen der Position des Positionszeigers
 - **public long** getFilePointer();
// Resultat: **aktueller** Wert des **Positionszeiger**
- Setzen der Position des Positionszeigers
 - **public void** seek(long pos);
// Resultat: Wert des Positionszeiger wird auf pos gesetzt.
- Abfrage nach der Länge der Datei (in Bytes)
 - **public long** length();

Gemessen in
Bytes.

Wichtige Schnittstellen



- Schnittstellen *DataOutput* und *DataInput*
 - Umwandlung von **primitiven Datentypen in eine Folge von Bytes** und umgekehrt.
 - Beispiele von Methoden
 - *DataOutput*
void writeDouble(double v) throws [IOException](#)
 - *DataInput*
double readDouble() throws [IOException](#)

Methoden zum Lesen

- Implementierung der Schnittstelle `DataInput`
- Methoden zum **Lesen** von Daten
 - Schnittstelle `DataInput`:
 - **boolean** `readBoolean()`
 - **double** `readDouble()`
 - **int** `read(byte[] b, int off, int len)`
 - Methode **liest** bis zu **len Bytes aus der Datei und schreibt diese** in das Array `b` an die Position `off`.
 - Resultat: Anzahl der tatsächlich gelesenen Bytes.
 - ...

Methoden zum Schreiben

- Implementierung der Schnittstelle `DataOutput`
- Methoden zum **Schreiben** von Daten
 - Schnittstelle `DataOutput`:
 - **void** `writeBoolean(boolean v)`
 - **void** `writeDouble(double d)`
 - **void** `write(byte[] b, int off, int len)`
 - Methode **schreibt** ab der Position `off` bis zu `len` Bytes aus dem Array `b` in die Datei.
 - ...

Beispiel

- Erweiterung der Klasse Point

```
import java.io.RandomAccessFile;

class Point {
    ...
    public void writeToFile(RandomAccessFile raf, int pointPos)
        throws IOException {
        raf.seek(pointPos*16);
        raf.writeDouble(x);
        raf.writeDouble(y);
    }

    public Point(RandomAccessFile raf, int pointPos)
        throws IOException {
        raf.seek(pointPos*16);
        x = raf.readDouble();
        y = raf.readDouble();
    }
    ...
}
```

Der wievielte „Point“
in der Datei.

seek und
writeDouble
können eine
IOException
werfen.

Ein „Point“ besteht aus
zwei doubles. Je double
werden 8 Byte benötigt.

Verwendung der Methoden

```
Point[] parr = new Point[MAX], pcopy = new Point[MAX];

// ...
File f = new File("Datei.pi1");

RandomAccessFile raf;
try {
    raf = new RandomAccessFile(f, "rw");
} catch (IOException e) {
    System.out.println("Die Datei konnte nicht geöffnet werden"); return -1;
}

// Schreiben in die Datei
try {
    for (int i = 0; i < MAX; i++)
        parr[i].writeToFile(raf, i);
} catch (IOException e) {
    System.out.println("Nicht alle Punkte konnten serialisiert werden"); return -1; }

// Einlesen der Daten aus der Datei
try {
    for (int i = 0; i < MAX; i++)
        pcopy[i] = new Point(raf, i);
} catch (IOException e) {
    System.out.println("Nicht alle Punkte konnten deserialisiert werden."); return -1; }
return 0;
```

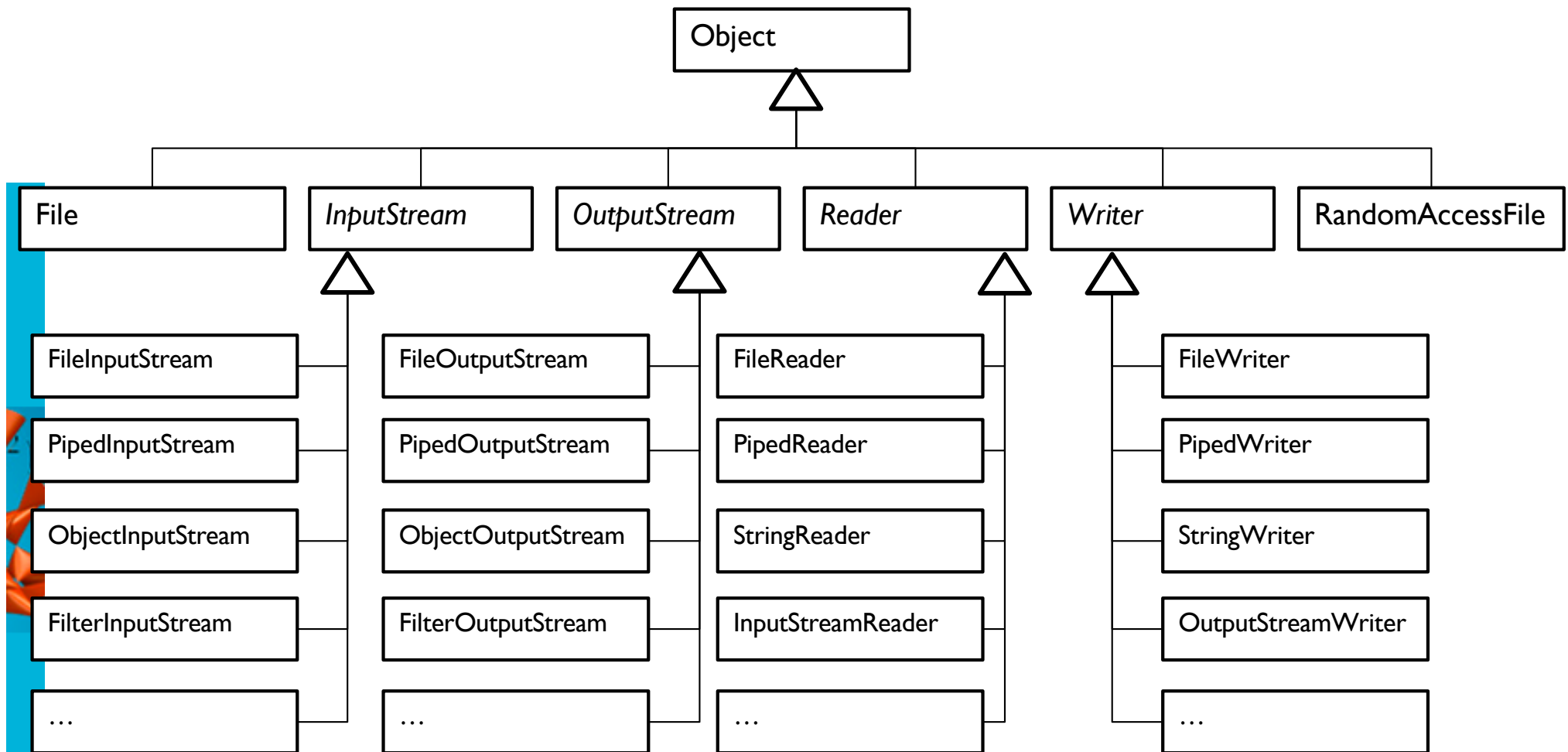
Diskussion

- Die Klasse `RandomAccessFile` ermöglicht den **wahlfreien Zugriff** auf Dateien.
 - Position in der Datei kann beliebig gesetzt werden.
 - Daten werden dann im Binärformat in die Datei geschrieben.
- Hoher Aufwand bei der Programmierung
- Hohe Fehleranfälligkeit
 - Abfangen von Exceptions
- Grundlegende Klasse zur Implementierung von
 - Datenbanksystemen
 - Indexstrukturen

14.3 Datenströme (java.io)

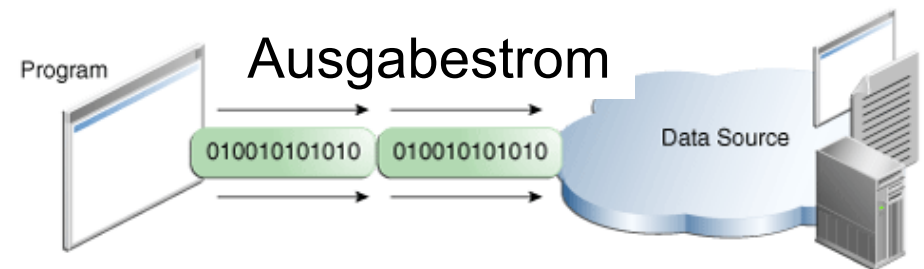
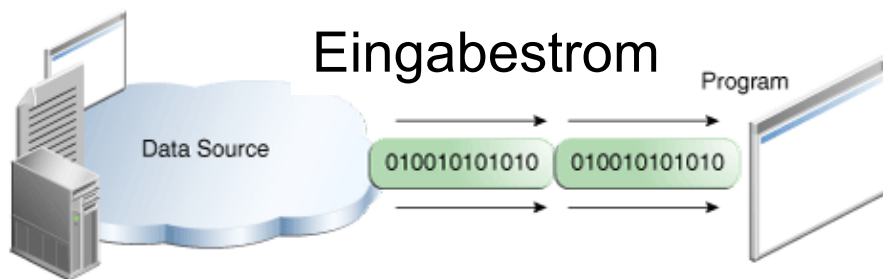
- Grundlegende Funktionalität für die **Verwaltung von Dateien** in einem Dateisystem
 - Klasse File
- Zugriff auf den **Inhalt der Dateien**: Unterscheidung zwischen strombasierten Zugriff und wahlfreien Zugriff auf Daten
 - **Strombasierter Zugriff**
 - Sequentielles Lesen aus einer Datei
 - Beginnend von vorne nach hinten
 - Sequentielles Schreiben in eine Datei
 - Typischerweise anhängen neuer Daten ans Ende der Datei
 - **Wahlfreier Zugriff**
 - Lesender und schreibender Zugriff auf beliebige Positionen in einer Datei

Übersicht zu den Klassen



Datenströme

- Das Paket `java.io` bietet mit dem Konzept des Datenstroms mehr als die reine Ein- bzw. Ausgabe (I/O) in Dateien.
- Datenströme sind grundlegend für die
 - Mensch-Maschine Kommunikation
 - Maschine-Maschine Kommunikation in einem Netzwerk
- Unterscheidung zwischen Ein- und Ausgabestrom



Datenströme (java.io)

- Datenströme bieten einen **sequentiellen Zugriff** auf Dateien an.
 - Alle Klassen zu Datenströmen befinden sich im Paket **java.io**.
- **Stream** = **abstraktes Konstrukt** mit der Fähigkeit, Zeichen auf ein imaginäres Ausgabegerät zu schreiben oder von diesem zu lesen.
 - Dieses imaginäre Gerät abstrahiert von den „echten“ Geräten (Bildschirm, Tastatur, Datei, String, Kommunikationskanal).
 - **Unterklassen binden** die Zugriffsroutinen **an** echte Ein- oder Ausgabegeräte.
- Streams können **verkettet** oder **verschachtelt** werden.
 - **Verkettung** erlaubt z.B. mehrere Dateien zusammenzufassen und als einen einzigen Stream darzustellen.
 - **Verschachteln** erlaubt die Implementierung von Zusatzfunktionen wie das Puffern von Zeichen.

Datenströme (java.io)

- Datenströme bieten einen **sequentiellen Zugriff** auf Daten.
 - Alle Klassen zu Datenströmen befinden sich im **java.io**-Paket.
- **Stream** = **abstraktes Konzept** für einen Datenfluss. Ein **Stream** ist ein imaginäres Ausgabegerät zu schreiben.
 - Dieses imaginäre Gerät kann eine Datei, ein String, ein Bildschirm, Tastatur, etc. sein.
 - **Unterstrichen** werden die Klassen für die Eingabe- oder Ausgabegeräte.
- **Streams** können **gekettelt** werden.
 - **Verknüpfung** von Streams, um mehrere Streams zusammenzufassen und als einen einzigen Stream zu betrachten.
 - **Verschachtelung** zur Implementierung von Zusatzfunktionen wie das Puffern von Zeichen.

Achtung:
nicht verwechseln mit
`java.util.stream`

Byte- und Character-Streams

- In Java wird unterschieden zwischen
 - **Byte-Streams**
 - Diese Streams benutzen ein **Byte als Einheit**.
 - **Character-Streams**.
 - Diese verwenden grundsätzlich **16 Bit lange Unicode-Zeichen** und arbeiten daher besser mit den String- und Zeichentypen von Java zusammen.
 - **Brückenklassen** erlauben eine **Überführung von Character-Streams in Byte-Streams** und umgekehrt.
- Weiterhin wird bei Streams noch unterschieden zwischen
 - Streams, auf die **geschrieben** werden kann.
 - Streams, von denen **gelesen** werden kann.

Übersicht der Klassen

	Byte-Streams	Character-Streams
<i>lesenden Zugriff</i>	<i>InputStream</i>	<i>Reader</i>
<i>schreibenden Zugriff</i>	<i>OutputStream</i>	<i>Writer</i>

- Die *Basisklassen für Byte-Streams* sind *InputStream* und *OutputStream*.
 - Diese sind jeweils Wurzel einer *Hierarchie von Klassen*.
- Entsprechend beginnt für *Charakter-Streams* die Hierarchie bei den Klassen *Reader* und *Writer*.
- Beide Klassenhierarchien befinden sich im *Paket java.io*.