Objektorientierte und Formale Programmierung
-- Java Grundlagen --

8. Ausnahmebehandlung

8. Ausnahmebehandlung

Exceptions (Ausnahmen) signalisieren Fehler zur Laufzeit eines Programms

```
class ExceptTest {
  public static void main(String[] args) {
    int a = (int)(10*Math.random());
    int b = (int)(10*Math.random());
    System.out.println("a/b="+(a/b));
}
```

```
$ javac ExceptTest.java; java ExceptTest
Exception in thread "main" java.lang.ArithmeticException: / by zero
        at ExceptTest.main (ExceptTest.java:5)
```

8. Ausnahmebehandlung

Exceptions (Ausnahmen) signalisieren Fehler zur Laufzeit eines Programms

```
class ExceptTest {
  public static void main(String[] args) {
    int a = (int)(10*Math.random());
    int b = (int)(10*Math.random());
    try {
      System.out.println("a/b=" + (a/b));
    }
    catch (ArithmeticException e) {
      System.out.println("a/b=?  -- Es gab ein Problem: " + e.getMessage());
    }
}
```

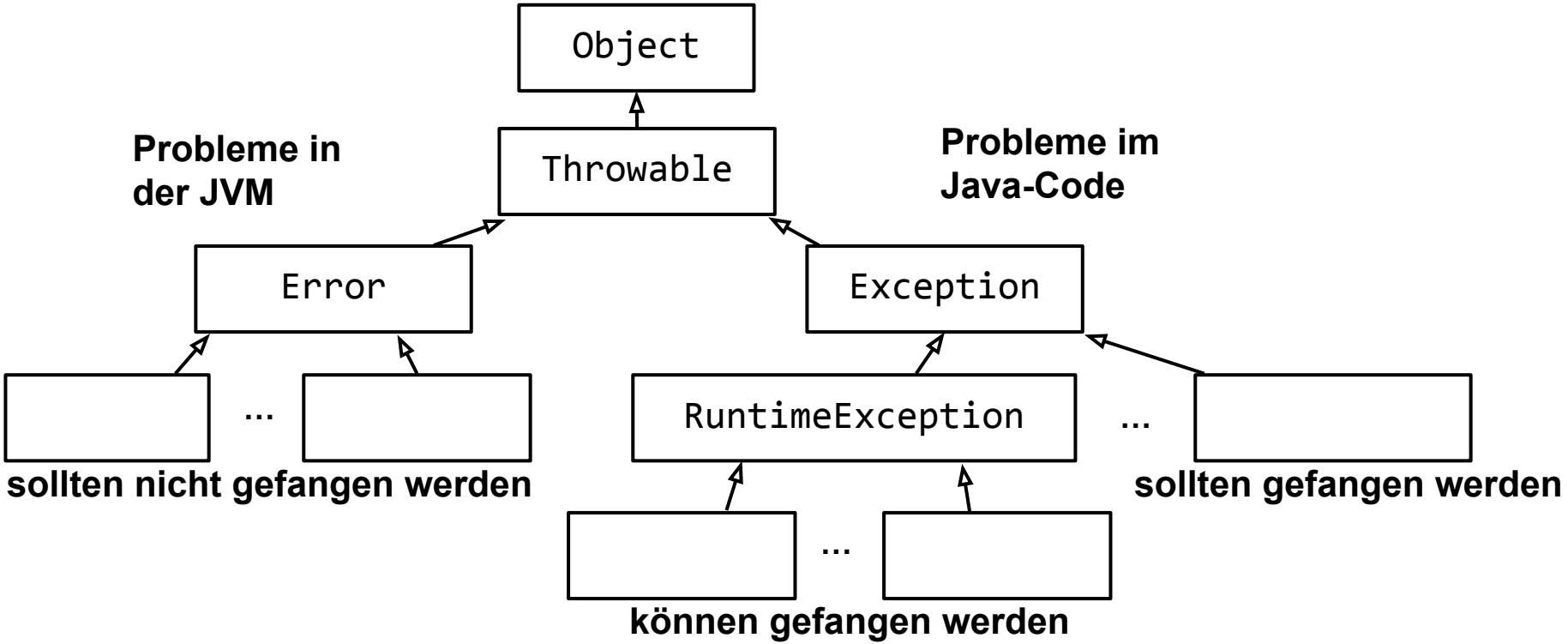
```
$ javac ExceptTest.java; java ExceptTest
a/b=?  -- Es gab ein Problem: / by zero
```

8. Ausnahmebehandlung

- Exceptions (Ausnahmen) signalisieren Fehler zur Laufzeit eines Programms
- Sie können
  - implizit durch Java-Anweisungen (z.B. `x/0`, `a[-1]`)
  - explizit durch die Anweisung `throw`

  ausgelöst (”geworfen“) werden
- Exceptions sind Ausnahmesituationen – sie sollten außerhalb des normalen Programmcodes behandelt (”gefangen“) werden
  - höhere Übersichtlichkeit des Codes
- In Java sind Exceptions Objekte, die in einer Fehlersituation dynamisch erzeugt werden
  - ihre Attribute beschreiben den Fehler genauer

Exceptions: Objekthierarchie



Unterklasse `RuntimeException`

- **`RuntimeExceptions`** werden meist durch Fehler im Programmcode verursacht, sie müssen daher nicht behandelt werden ("Unchecked" von Java Compiler)
- Beispiele (siehe auch Dokumentation zum Paket **`java.lang`**):

  `ArithmeticException:` z.B. `1/0` (ganzzahlig)

  `IndexOutOfBoundsException:` z.B. `array[-1]`

  `NegativeArraySizeException:` z.B. `new double[-5]`

  `NullPointerException:`

  z.B. `Hund meinHund = null; meinHund.print();`

  `ClassCastException:`

  z.B. `Tier t = new Hund("fido"); Katze k = (Katze)t;`

8. Ausnahmebehandlung

Definition eigener Exceptions → github

```
public class MyExcept extends Exception {

  private int elNr = -1;

  public MyExcept() {}                        // diese beiden Konstruktoren
  public MyExcept(String s) { super(s); } // werden immer definiert
  // extra Konstruktor mit Zusatzinformation zum Fehler:
  public MyExcept(String s, int elNr) { super(s); this.elNr = elNr; }

  public int getElementNr() { return elNr; }

  public String toString() {
    // getMessage() ist Methode der Oberklasse Throwable, liefert den String
    // zurueck, der dem Konstruktor uebergeben wurde:
    return "Eigener Fehler im Element " + elNr + ": " + getMessage();
  }
}
```

Definition eigener Exceptions

```
public class Vector {
  double[] vector;
  // …

  public void invert() throws MyExcept {
    for (int i = 0; i < vector.length; i++) {
      if (vector[i] == 0.0) {
        throw new MyException("Divide by 0.", i);
      }
      vector[i] = 1.0 / vector[i];
    }
  }
}
```

Werfen (**throw**) von Exceptions

- Exceptions können im Programm explizit durch die Anweisung **throw** ausgelöst werden:

```
public static double invert(double x) {
  if (x == 0.0) throw new ArithmeticException("Divide by 0.");
  return 1.0 / x;
}
```

- **throw** kann auch in einem catch-Block verwendet werden:

```
catch (Exception e) {
  // ... Lokale Fehlerbehandlung  ...
  throw e;  // Exception an Aufrufer weitergeben
}
```

Behandlung von Exceptions

- Einfachster Fall: Exception wird nicht behandelt
  - Methode bricht beim Auftreten der Exception sofort ab
  - Exception wird an Aufrufer der Methode weitergegeben
  - Wenn die Exception nicht spätestens in der main-Methode gefangen wird → Abbruch des Programms
  - Jede Methode muss deklarieren, welche Exceptions sie werfen kann (ausgenommen **`Error`** und **`RuntimeException`**):

```
void anmelden(...) throws AnmeldungsException
```

oder

```
public static void main(...) throws AnmeldungsException, OtherException
```

Behandlung von Exceptions

- **`try - catch`** - Block:

```
try {
    int z = zahlen[index];
    int kehrwert = 1 / z;
    // ...
}
catch (IndexOutOfBoundsException e) {
    System.out.println("Unzulässiger Index");
}
catch (ArithmeticException e) {
    System.out.println("Fehler: " + e.getMessage());
}
catch (Exception e) {
    System.out.println(e);
}
```

Behandlung von Exceptions

- Wenn Exception im **`try`**-Block auftritt:
  - **`try`**-Block wird verlassen
  - Erster ”passender“ **`catch`**-Block wird ausgeführt,
    Ausführung wird nach dem letzten catch-Block fortgesetzt
  - falls kein passender catch-Block vorhanden:
    Abbruch der Methode, Weitergeben der Exception an Aufrufer
- Der **`catch`**-Block ist ”passend“ wenn das erzeugte Exception-Objekt an den Parameter des catch-Blocks zugewiesen werden kann.
  Die erzeuge Exception ist identisch mit der spezifizierten Exception-Klasse oder ist eine Unterklasse davon

8. Ausnahmebehandlung

`finally`-Block

- Nach dem letzten catch-Block kann noch ein **`finally`**-Block angefügt werden
  - auch **`try - finally`** (ohne **`catch`**) ist erlaubt
- Die Anweisungen dieses Blocks werden immer nach Verlassen des **`try`**-Blocks ausgeführt, egal ob
  - der try-Block normal beendet wird
  - der Block (und die Methode) durch return verlassen wird
  - eine Exception auftritt und durch **`catch`** gefangen wird
    - hier: **`finally`** nach **`catch`** ausgeführt
  - eine Exception auftritt und an den Aufrufer weitergegeben wird
- Anwendung: ”Aufräumarbeiten“
  - z.B. Löschen temporärer Dateien, Schließen von Fenstern, ...

Beispiel -- Wordle: Laden von Wörterdatei

```
int len = 0;
try {  // load 5-letter words from a predefined file
  words = new String[WORDS_MAX];
  Scanner scanner = new Scanner(new File("words.txt"));
  while ( (scanner.hasNextLine()) && (len<WORDS_MAX) ) {
    words[len] = scanner.nextLine();
    len++;
  }
  scanner.close();
}
catch (java.io.IOException e) {  // use backup dictionary
  len = 3;
  words = new String[len];
  words[0] = "worse"; words[1] = "worth"; words[2] = "words";
}
```

Beispiel -- Wordle: Laden von Wörterdatei (🌶🌶🌶)

```
/**  WordleClass: Implementieren Sie das Wordle-Spiel mit Wörterliste:
  */
import java.util.Scanner;  // scan strings in der Konsole
import java.util.Random;   // Random Number Generator
import java.io.File;

class WordleClass {
  /*...*/
  private String dialog( String hint ) { /*...*/ }     // private Methode
  private void loadWords(String filename) { /*...*/ }  //
  public void run() { /*...*/ }  // public Methode

  public static void main( String[] str ) {  // main ist eine Klassenmethode
    WordleClass w = new WordleClass();
    w.run();
  }
}
```

Beispiel -- WordleClass Lösung (1/4) → github

```
/**  WordleClass: das komplette Wordle-Spiel mit Wörterliste
  */

import java.util.Scanner;  // scan strings in der Konsole
import java.util.Random;   // Random Number Generator
import java.io.File;

class WordleClass {
  private String loesung = "weary"; // default Loesunswort
  private String hint = "-----";    // Hinweis mit gefundenen Buchstaben
  private String[] words = null;    // Wörterarray
  private static final int MAX_WORDS = 10000; // Maximum # Wörter

  public static String dialog( String hint ) {  // Zeige Hinweis, Eingabe
    System.out.print( hint + ", what is your guess? ");
    Scanner scan = new Scanner(System.in);
    return scan.next();
  }
```

Beispiel -- WordleClass Lösung (2/4) → github

```
private void loadWords(String filename) {
  int len = 0;
  try {  // lade Wörter aus filename (z.B. "words.txt"):
    words = new String[MAX_WORDS];
    Scanner scanner = new Scanner(new File(filename));
    while ( (scanner.hasNextLine()) && (len<MAX_WORDS) ) {
      words[len] = scanner.nextLine();
      len++;
    }
    scanner.close();
  }
  catch (java.io.IOException e) {  // backup Wörter falls Datei nicht besteht
    len = 3;
    words = new String[len];
    words[0] = "worse";
    words[1] = "worth";
    words[2] = "words";
  }
}
```

Beispiel -- WordleClass Lösung (3/4) → github

```
public void run() {
  String in;            // Eingabe Benutzer
  char[] hintChar;      // Array von char fuer hint zu aendern
  loadWords("words.txt");

  do {
    in = dialog( hint );
    hintChar = hint.toCharArray();
    if ( in.length() >= loesung.length() ) {
      for (int i = 0; i < loesung.length(); i++ ) {  // Kontrolliere einzelne
        if ( in.charAt(i) == loesung.charAt(i) ) {   // Buchstaben von in
          hintChar[i] = in.charAt(i);                // und zeige sie in hintChar
        } else {                                     // wenn richtig
          boolean found = false;
          for (int j = 0; j < loesung.length(); j++ ) {  // suche Buchstaben an
             if ( in.charAt(i) == loesung.charAt(j) ) {  // andere Stellen in loesung
               found = true;
             }
          }
```

Beispiel -- WordleClass Lösung (4/4) → github

```
            hintChar[i] = (found?'?':'-');
          }
        }
      } else {
        System.out.println("word too short.");
      }
      hint = new String(hintChar);
    } while ( ! in.equals( loesung ) );
  }

  public static void main( String[] str ) {  // main ist eine Klassenmethode
    WordleClass w = new WordleClass();
    w.run();
  }
}
```

Objektorientierte und Formale Programmierung
-- Java Grundlagen --

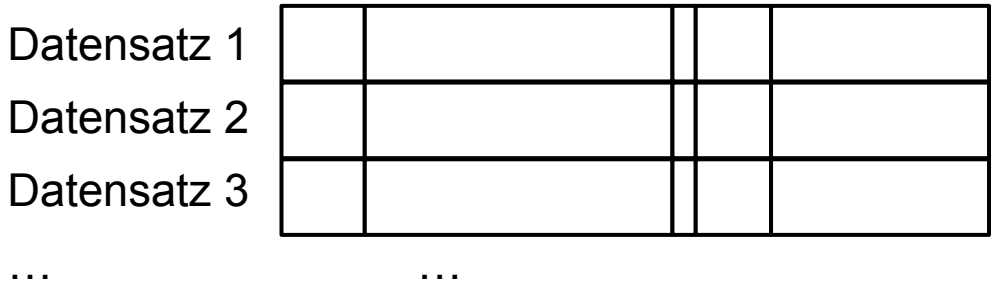
9. Dateien, Ströme und Serialisierung

9.1. Pakete der Java-Klassenbibliothek

- Die Sprache Java wird von einer (standardisierten) Klassenbibliothek ergänzt, mit tausenden Klassen in hunderten Paketen
- Häufig genutzte Pakete (importieren mit **`import`**, z.B.: `import java.awt.*;`):
  - **`java.lang`**: Klassen, die zum Kern der Sprache Java gehören (z.B.: **`String`**, **`StringBuffer`**, **`Object`**, **`System`**). Diese müssen nicht explizit importiert werden
  - **`java.io`**: Ein- und Ausgabe (Konsole und Dateien)
  - **`java.util`**: nützliche Hilfsklassen
    u.a. Java Collection Framework (Container-Klassen)
  - **`java.awt`**: Elemente für Bedienoberflächen
  - **`javax.swing`**: verbesserte Bibliothek für Bedienoberflächen
  - **`java.net`**: Netzwerkkommunikation
  - **`java.sql`**: Datenbank-Anbindung
  - **`java.beans`**: Komponentenmodell

9.2. Die Datenstruktur ”Datei“ (file)

- Eine Datei ist eine nach bestimmten Gesichtspunkten zusammengestellte Menge von Daten:

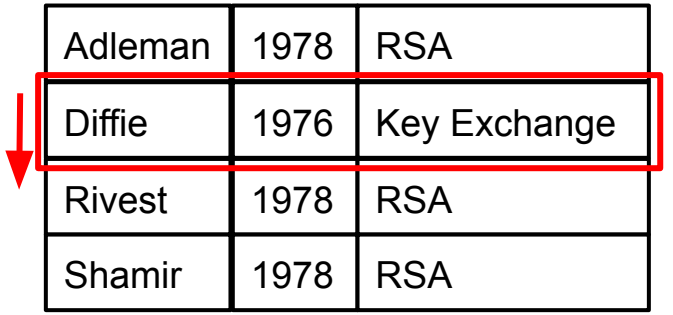


- Sie besteht aus einer Folge gleichartig aufgebauter Datensätze
  - ein Datensatz besteht aus mehreren Feldern unterschiedlichen Typs
  - die Anzahl der Datensätze muss nicht festgelegt werden
- Dateien werden i.d.R. dauerhaft auf Hintergrundspeichern gespeichert

9.2. Die Datenstruktur ”Datei“ (file)

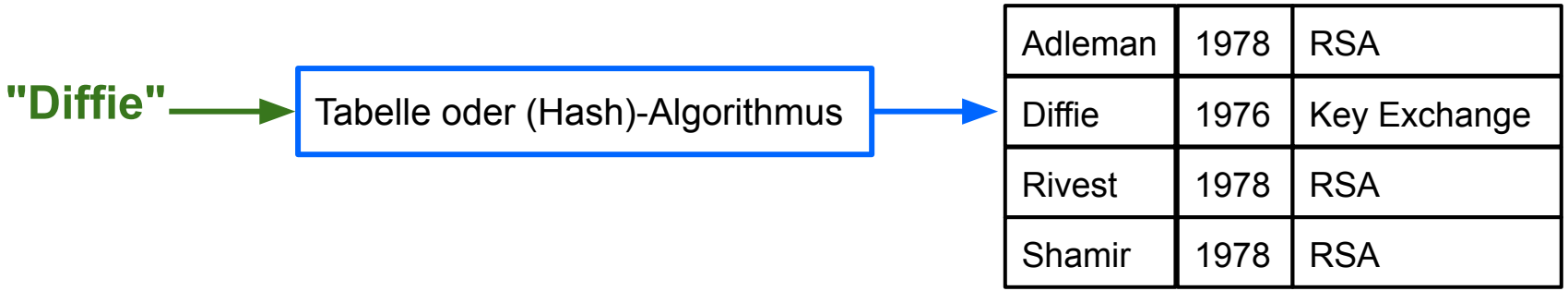
Gängige Datei-Organisationen: Sequentielle Datei

- Daten sind fortlaufend gespeichert und können nur in dieser Reihenfolge gelesen werden
- es gibt ein **Dateifenster**, das bei jedem Lesen bzw. Schreiben um eine Position (einen Datensatz) weiter rückt
- es kann nur jeweils der Datensatz im Dateifenster gelesen bzw. geschrieben werden
- das Dateifenster kann zum Teil auch direkt positioniert werden



9.2. Die Datenstruktur ”Datei“ (file)
Gängige Datei-Organisationen: Direkte Datei

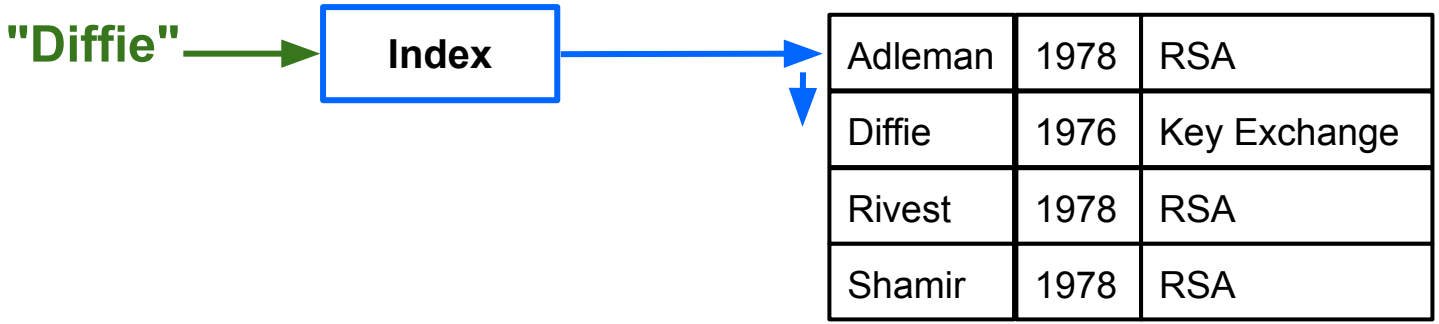
- Zugriff auf Datensätze erfolgt über einen Schlüssel, aus dem direkt die Position in der Datei bestimmt wird



9.2. Die Datenstruktur ”Datei“ (file)

Gängige Datei-Organisationen: Indexsequentielle Datei, Mischform

- Nutzung einer Tabelle (Index), die für einen Schlüssel in die Nähe des Datensatzes führt. Von dort aus sequentielle Suche

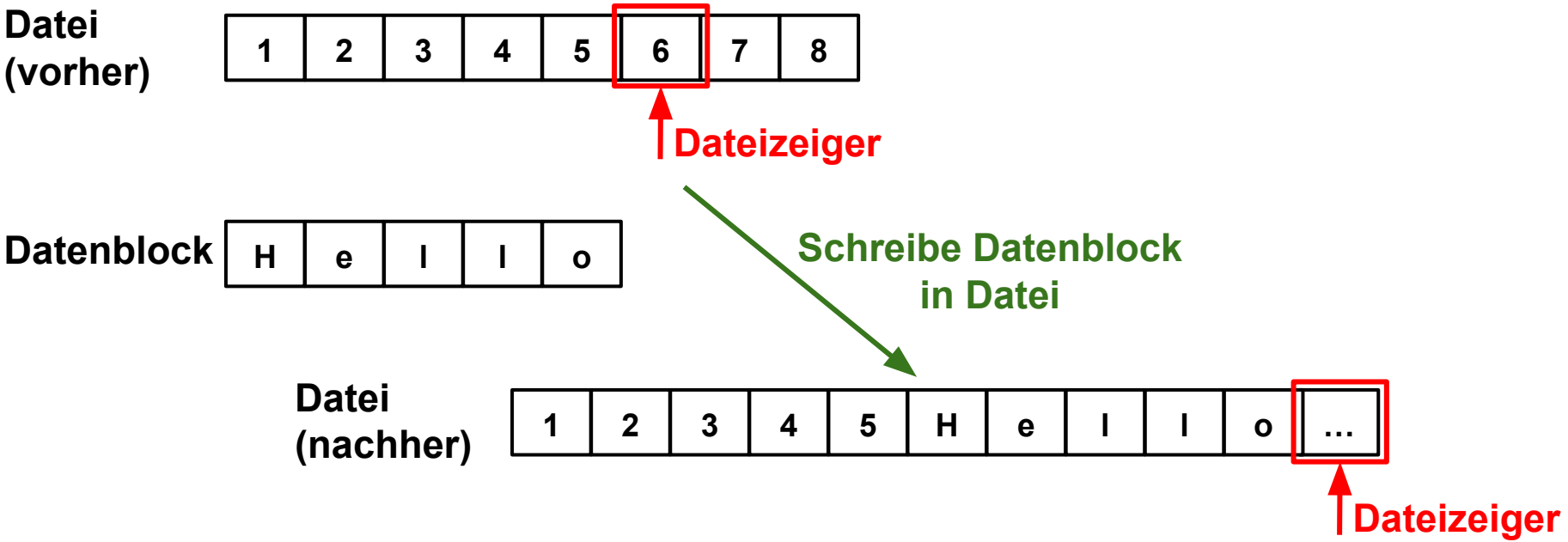


9.2. Die Datenstruktur ”Datei“ (file)
Das Dateimodell von Java

- Eine Datei in Java ist eine (unstrukturierte) Folge von Bytes. z.B. Textdatei: Folge von 8-Bit-Zeichen
- Nach dem Öffnen einer Datei verweist ein *Dateizeiger* auf das nächste zu lesende bzw. zu schreibende Byte
- Lese- und Schreiboperationen kopieren einen Datenblock aus der Datei bzw. in die Datei, der Dateizeiger wird entsprechend weitergeschoben
- Lesen über das Dateiende hinaus (End-of-file, EOF) ist nicht möglich
- Schreiben über das Dateiende führt zum *Anfügen* an die Datei
- Der Dateizeiger kann auch explizit positioniert werden

9.2. Die Datenstruktur ”Datei“ (file)

Das Dateimodell von Java, Beispiel: Schreiben in eine (Text-)Datei



9.2. Die Datenstruktur ”Datei“ (file)

Grundoperationen auf Dateien

- öffnen (**open**) einer durch ihren Namen gegebenen Datei
  - zum Lesen: Dateizeiger wird auf Anfang positioniert
  - zum Schreiben: Dateizeiger wird auf Anfang bzw. Ende positioniert (überschreiben der Datei bzw. Anfügen)
  - i.d.R. wird beim Öffnen auch ein Dateipuffer eingerichtet
    - speichert einen Teil der Datei im Hauptspeicher zwischen
    - verhindert, dass jede Datei-Operation sofort auf dem langsamen Hintergrundspeicher ausgeführt werden muss
- Schließen (**close**) einer geöffneten Datei
  - Dateien sollten nach Verwendung immer geschlossen werden
    - sonst evtl. Datenverlust: Zurückschreiben des Dateipuffers
  - nach dem Schließen sind keine Operationen mehr zulässig

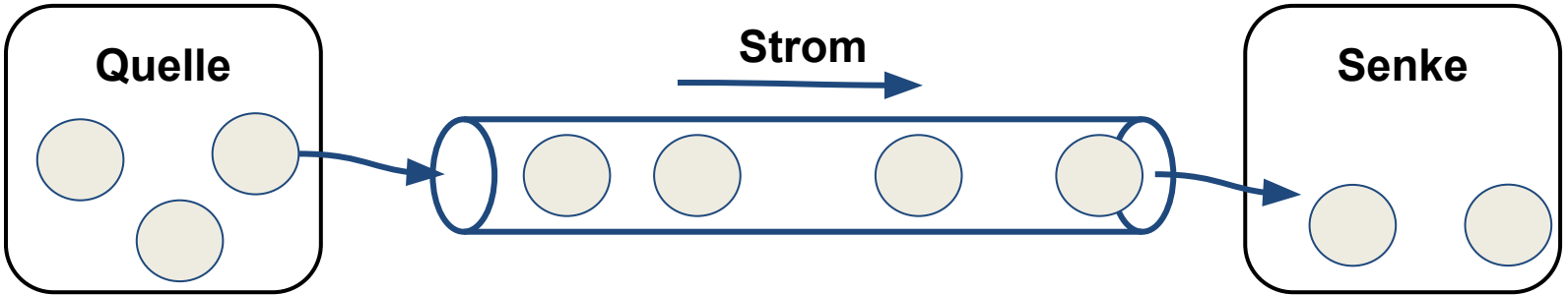
9.2. Die Datenstruktur ”Datei“ (file)

Grundoperationen auf Dateien

- lesen (**read**) eines Datenblocks
  - die Daten ab dem Dateizeiger werden in eine Variable (z.B. Byte-Array) kopiert
  - Dateizeiger wird entsprechend weiterbewegt
- schreiben (**write**) eines Datenblocks
  - Inhalt einer Variable (z.B. Byte-Array) wird ab dem Dateizeiger in die Datei kopiert (ggf. angefügt)
  - Dateizeiger wird entsprechend weiterbewegt
- **flush**: Leeren des Dateipuffers
  - Inhalt des Dateipuffers wird in die Datei zurückgeschrieben
- **seek**: explizites Positionieren des Dateizeigers
  - ermöglicht wahlfreien Zugriff auf die Datei

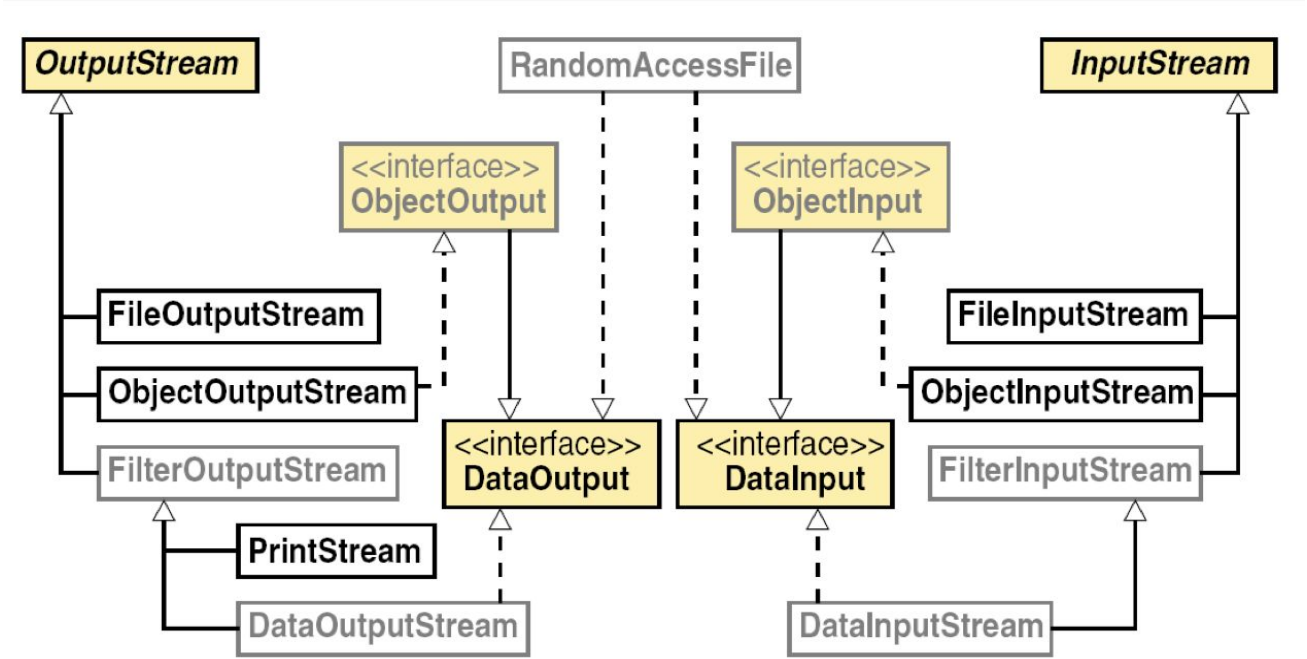
9.3. Ein- und Ausgabe mit Strömen (Streams)

- In Java erfolgt jede Ein-/Ausgabe (auch in Dateien) über Ströme.
  Sie stellen die Schnittstelle des Programms nach außen dar
- Ein Strom ist eine geordnete Folge von Daten mit einer Quelle und einer Senke
  - Ströme sind i.d.R. unidirektional (entweder Ein- oder Ausgabe)
  - ein Strom puffert die Daten so lange, bis sie von der Senke entnommen werden (Warteschlange)



9.3. Ein- und Ausgabe mit Strömen (Streams)

Wichtige Strom-Klassen / Schnittstellen im Paket **`java.io`**



9.3. Ein- und Ausgabe mit Strömen (Streams)

Wichtige Strom-Klassen / Schnittstellen im Paket **`java.io`**

- Abstrakte Basisklassen: **`InputStream`**, **`OutputStream`**
  allgemeine Ströme für Ein- bzw. Ausgabe
- Dateiströme: **`FileInputStream`**, **`FileOutputStream`**
  spezielle Ströme für die Ein-/Ausgabe auf Dateien
- Serialisierung mittels **`ObjectInputStream`**, **`ObjectOutputStream`**
- Bidirektionaler Dateistrom: **`RandomAccessFile`**
  ermöglicht zusätzlich Positionieren des Dateizeigers
- Filterströme: **`FilterInputStream`**, **`FilterOutputStream`**
  - erhalten Daten von einem anderen Strom und filtern diese bzw. geben gefilterte Daten an einen anderen Strom weiter
  - Filterung: z.B. Umwandlung von Datentypen in Byteströme

9.3. Ein- und Ausgabe mit Strömen (Streams)
Wichtige Strom-Klassen / Schnittstellen im Paket **`java.io`**

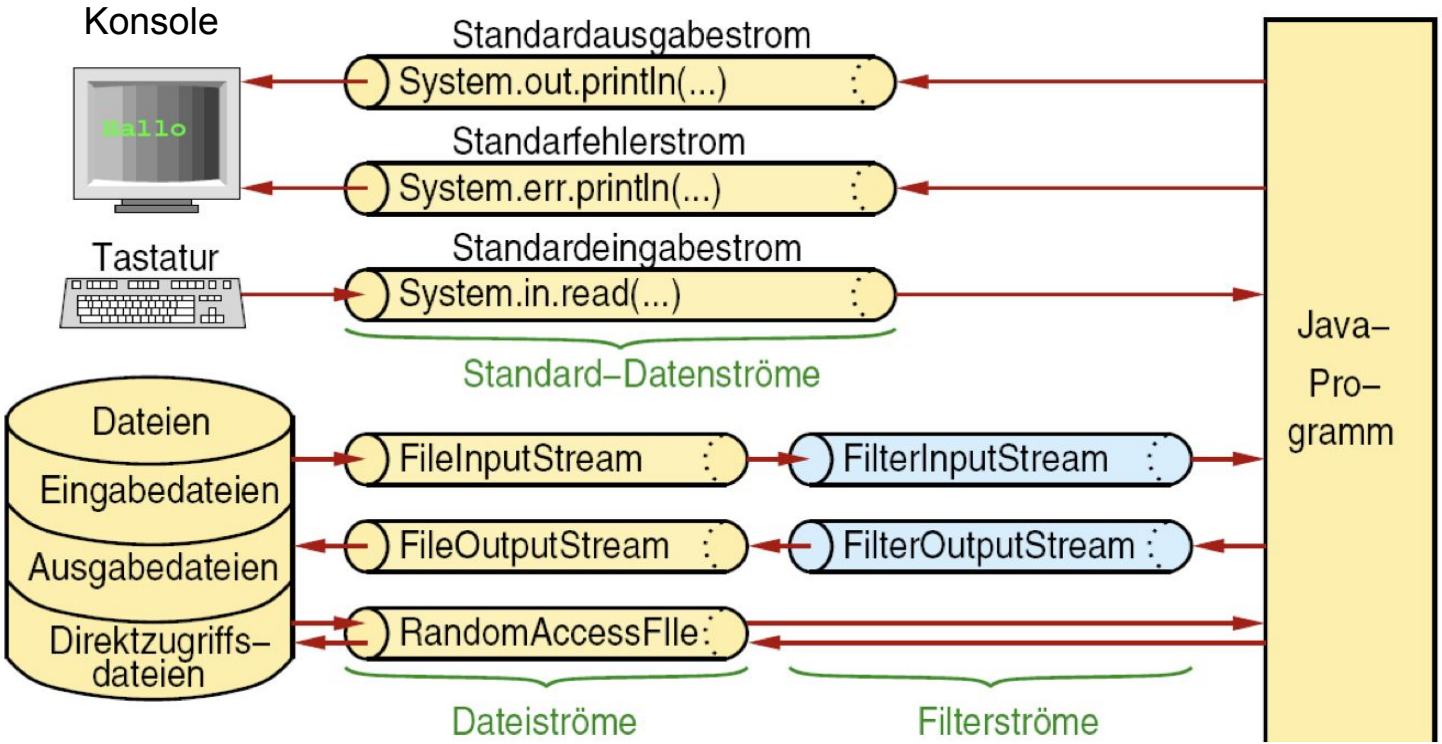
- Schnittstellen **`DataInput`**, **`DataOutput`**
  - definieren Operationen zur Ein-/Ausgabe von einfachen Datentypen (**`int`**, **`double`**, ...) und **`String`**s
  - implementiert von den Filterströmen **`DataInputStream`**, **`DataOutputStream`**
- Schnittstellen **`ObjectInput`**, **`ObjectOutput`**
  - definieren Operationen zur Ein-/Ausgabe von Objekten
  - implementiert von den Strömen **`ObjectInputStream`**, **`ObjectOutputStream`**
- Strom zur formatierten Text-Ausgabe von Daten: **`PrintStream`**

9.3. Ein- und Ausgabe mit Strömen (Streams)
Standard-Datenströme

Java definiert drei Standard-Datenströme für die Ein-/Ausgabe von / zur Konsole:

- **`InputStream System.in`** zum Einlesen von Zeichen von der Tastatur,
  z.B. `char ch = (char) System.in.read();`
- **`PrintStream System.out`** zur Ausgabe von Zeichen auf den Bildschirm,
  z.B. `System.out.println("Hallo");`
- **`PrintStream System.err`** zur Ausgabe von Zeichen auf den Bildschirm, speziell für Fehlermeldungen

9.3. Ein- und Ausgabe mit Strömen (Streams)



9.3. Ein- und Ausgabe mit Strömen (Streams)

Wichtige Operationen der Klasse **`InputStream`**

- **`abstract int read() throws IOException`**
  - liest ein Byte (0 ... 255) aus dem Strom
  - blockiert, falls keine Eingabe verfügbar ist
  - am Stromende (z.B. Dateiende) wird -1 zurückgegeben
- **`int read(byte[] buf) throws IOException`**
  - liest bis zu buf.length Bytes aus dem Strom
  - blockiert, bis eine Eingabe verfügbar ist
  - Ergebnis: Zahl der gelesenen Bytes bzw. -1 am Stromende
- **`void close() throws IOException`**
  - schließt den Strom: Freigabe belegter Ressourcen

9.3. Ein- und Ausgabe mit Strömen (Streams)

Beispiel: Bytes im Eingabestrom zählen (🌶️) → github

```
import java.io.*;

public class Count {
  // IOException wird nicht gefangen, dies muß deklariert werden:
  public static void main(String[] args) throws IOException {
    int count = 0;
    // Zeichen einlesen mit System.in.read(), bis Stromende (Strg-D):

    String msg = "  Eingabe hatte " + count + " Bytes\n";
    // Nur zur Demonstration. Ausgabe als Bytearray via write():
    System.out.write(msg.getBytes());
  }
}
```

9.3. Ein- und Ausgabe mit Strömen (Streams)

Wichtige Operationen der Klasse **`OutputStream`**

- **`abstract void write(int b) throws IOException`**
  - schreibt das Byte b (0 ... 255) in den Strom
  - (nur die unteren 8 Bit von b sind relevant)
- **`void write(byte[] buf) throws IOException`**
  - schreibt die Bytes aus buf in den Strom
- **`void flush() throws IOException`**
  - leert den Puffer des Stroms
  - alle noch im Puffer stehenden Bytes werden z.B. auf den Bildschirm oder in die Datei geschrieben
- **`void close() throws IOException`**
  - schließt den Strom: Freigabe belegter Ressourcen

9.3. Ein- und Ausgabe mit Strömen (Streams)

Dateiströme

- **`FileInputStream(String path) throws FileNotFoundException`**
  öffnet Datei mit angegebenem Namen zum Lesen
- **`FileOutputStream(String path) throws FileNotFoundException`**
  - öffnet Datei mit angegebenem Namen zum Schreiben
  - Datei wird ggf. neu erzeugt
- Operationen werden von InputStream bzw. OutputStream geerbt und teilweise mit neuen Implementierungen überschrieben

9.3. Ein- und Ausgabe mit Strömen (Streams)

Dateiströme Beispiel: Datei Kopieren (1/2) (🌶️) → github

```
class Copy {
  public static void copyFile(String from, String to) throws IOException {
    // Ein- und Ausgabedateien öffnen:
    FileInputStream in = new FileInputStream(from);
    FileOutputStream out = new FileOutputStream(to);
    // Datei byteweise kopieren mit read() und write():


    // Dateien schließen:
    in.close(); out.close();
  }
```

9.3. Ein- und Ausgabe mit Strömen (Streams)

Dateiströme Beispiel: Datei Kopieren (2/2) (🌶️) → github

```
  // Aufruf: java Copy <Eingabedatei> <Ausgabedatei>
  public static void main(String[] args) {
    if (args.length != 2) {
      System.err.println("Programm benötigt 2 Argumente: " +
                         "<Eingabedatei> <Ausgabedatei>  ");
      return;
    }
    try {
      copyFile( args[0], args[1] );
    }
    catch (IOException e) {
      System.err.println("Fehler beim Kopieren: " + e );
    }
  }
}
```

9.3. Ein- und Ausgabe mit Strömen (Streams)

Beispiel Dateien / Strömen: TextLogger.java (🌶🌶🌶) → github

```
import java.io.*;
class TextLogger {
  public static void main(String[] args) {
    // IOException soll in main gefangen werden:

      FileOutputStream out = new FileOutputStream("log.txt");  // Ausgabe
      // Zeichen einlesen bis Stromende (^D), schreibe jedes Zeichen
      // via out in die Ausgabedatei:

      // Datei schließen:

  }
}
```

9.4. Serialisierung von Objekten

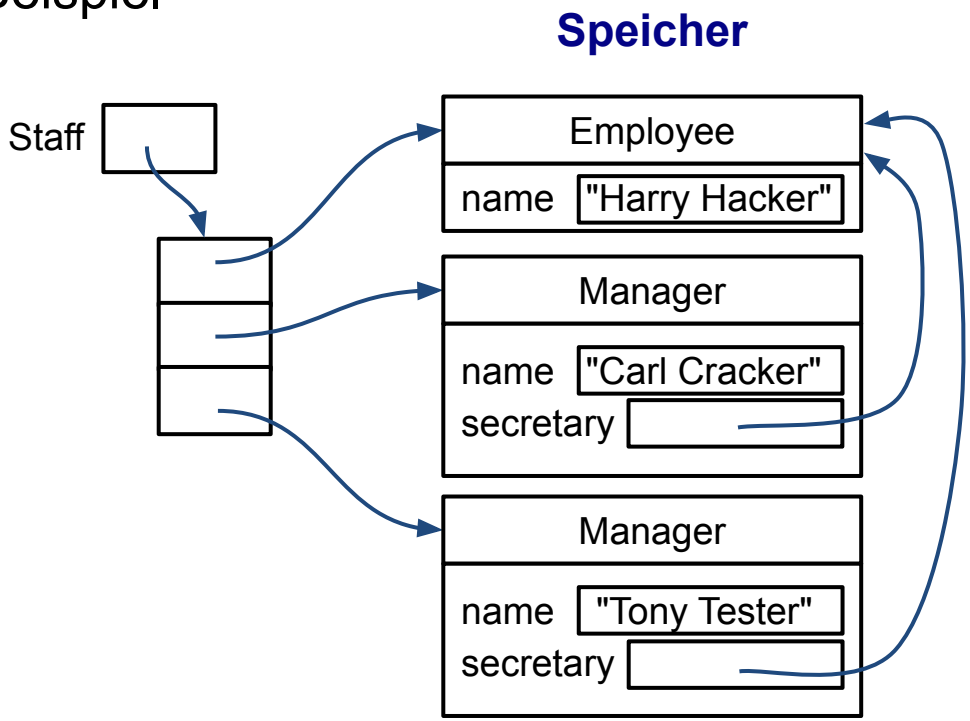
- Ziel: einmal erzeugte Objekte sollen auch über das Ende des Programms hinaus gespeichert bleiben ("Lightweight Persistence")
- Persistenz: Langfristige Speicherung von Objekten mit ihren Zuständen und Beziehungen, so dass ein analoger Zustand im Arbeitsspeicher wiederhergestellt werden kann
- Serialisierung: Umwandlung des Zustands eines Objekts in einen *Byte-Strom* bzw. umgekehrt (Deserialisierung)
  - der Byte-Strom lässt sich dann in eine Datei ausgeben bzw. von dort wieder einlesen
  - das Objekt kann dabei Referenzen auf Arrays und andere Objekte enthalten, die automatisch mit serialisiert werden

9.4. Serialisierung von Objekten

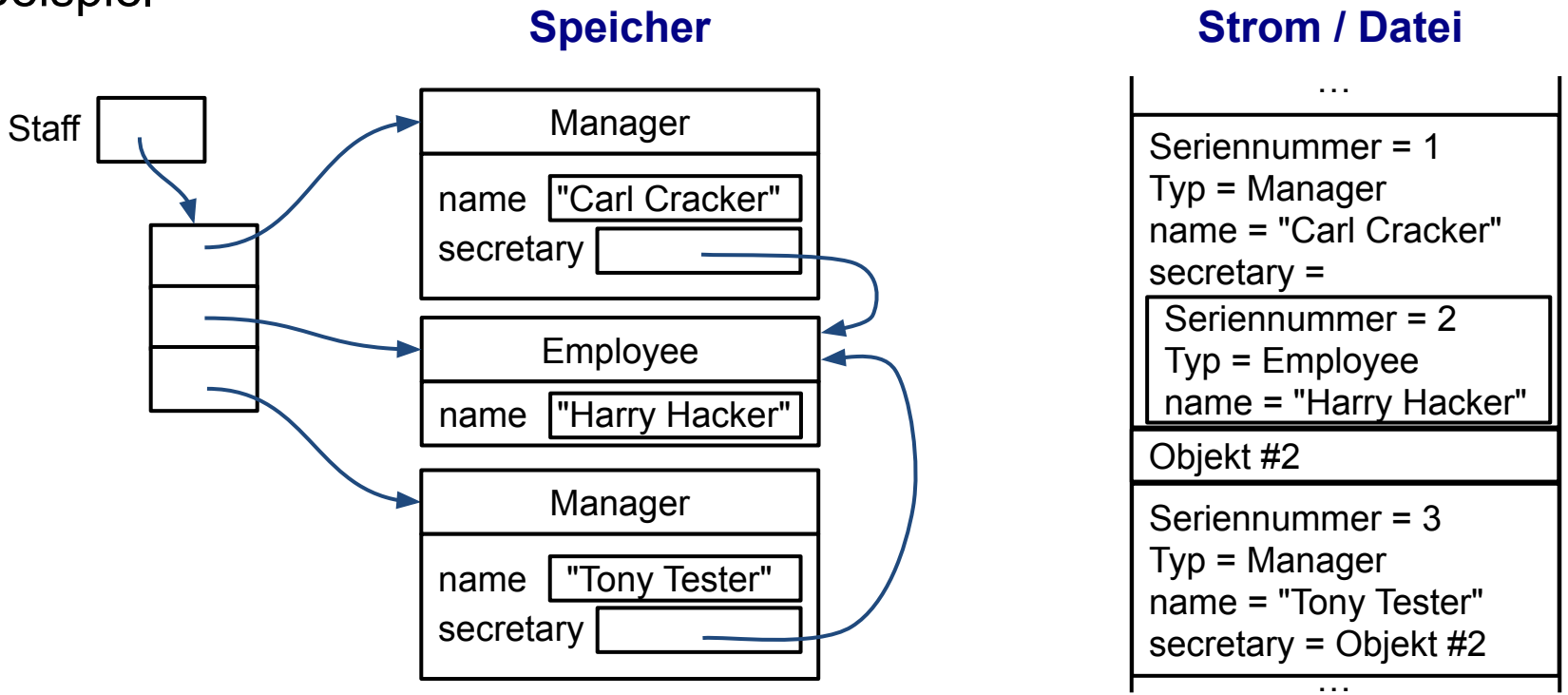
Was geschieht bei der Serialisierung eines Objekts / Arrays?

1. Erzeuge eine eindeutige Seriennummer für das Objekt und schreibe diese in den Strom
2. Schreibe Information zur Klasse in den Strom, u.a. Klassenname, Attributnamen und -typen
3. Für alle Attribute des Objektes (bzw. Elemente des Arrays):
   - falls keine Referenz: schreibe den Wert in den Strom
   - sonst: Z = Ziel der Referenz
     - falls Z noch nicht in diesen Strom serialisiert wurde:
       - serialisiere Z (Rekursion)
     - sonst: schreibe die Seriennummer von Z in den Strom

9.4. Serialisierung von Objekten
Beispiel



9.4. Serialisierung von Objekten
Beispiel



9.4. Serialisierung von Objekten

Voraussetzung für die Serialisierbarkeit von Objekten

1. Die Klasse muß die Schnittstelle **`Serializable`** implementieren
   - **`Serializable`** besitzt weder Methoden noch Attribute (Interface **`Serializable`** in **`java.io`**),
   - die Schnittstelle dient nur der Markierung einer Klasse als serialisierbar (*Marker-Interface*)
2. Zudem müssen alle Referenzen in dem Objekt wieder auf serialisierbare Objekte verweisen:

```
class Person implements Serializable {
    private String name;  // String ist serialisierbar
    private Address adresse;
}
class Address implements Serializable {  // ...
```

9.4. Serialisierung von Objekten

Die Klasse `ObjectOutputStream`

- Realisiert die Serialisierung von Objekten
- Konstruktor: `ObjectOutputStream(OutputStream out) throws IOException`
- `void writeObject(Object obj) throws IOException`
  serialisiert `obj` in den Ausgabestrom
- `void reset() throws IOException`
  löscht alle Information darüber, welche Objekte bereits in den Strom geschrieben wurden.
  Nachfolgendes `writeObject()` schreibt Objekte erneut in den Strom
- zusätzlich: alle Methoden der Schnittstelle `DataOutput`

9.4. Serialisierung von Objekten

Was schreibt **`writeObject(obj)`** in den Ausgabestrom?

- Falls **`obj`** noch nicht in den Strom geschrieben wurde:
  - neben **`obj`** werden auch alle von **`obj`** aus erreichbaren Objekte serialisiert
  - es wird also immer ein ganzer Objekt-Graph serialisiert
  - **`obj`** heißt Wurzelobjekt des Objekt-Graphen
  - die Referenzen zwischen den Objekten werden bei der Deserialisierung automatisch wiederhergestellt
- Falls **`obj`** bereits in den Strom geschrieben wurde (und kein **`reset()`** ausgeführt wurde), wird nur ein ”Verweis“ (Seriennummer) auf das schon im Strom befindliche Objekt geschrieben

9.4. Serialisierung von Objekten
Die Klasse `ObjectInputStream`

- Konstruktor: **`ObjectInputStream(InputStream in) throws IOException`**
- **`Object readObject() throws IOException`**
  liest das nächste Objekt aus dem Eingabestrom
  - falls Objekt bereits vorher gelesen wurde (ohne **`reset()`**):
    Ergebnis ist Referenz auf das schon existierende Objekt
  - sonst: Objekt und alle in Beziehung stehenden Objekte lesen
    - Objekte werden neu erzeugt, besitzen denselben Zustand und dieselben Beziehungen wie die geschriebenen Objekte
    - Ergebnis ist Referenz auf das Wurzelobjekt
  - i.d.R. explizite Typkonversion des Ergebnisses notwendig
- zusätzlich: alle Methoden der Schnittstelle **`DataInput`**

9.4. Serialisierung von Objekten
Die Schnittstellen `DataOutput` und `DataInput`

- Einige Methoden von DataOutput:
  - **`void writeInt(int v) throws IOException`**
    schreibt ganze Zahl in den Strom (in Binärform: 4 Bytes)
  - **`void writeDouble(double v) throws IOException`**
    schreibt Gleitkomma-Zahl (in Binärform: 8 Bytes)
- Einige Methoden von DataInput:
  - **`int readInt() throws IOException`**
  - **`double readDouble() throws IOException`**
  - bei Leseversuch am Dateiende: **EOFException**

9.4. Serialisierung von Objekten
Beispiel: Studentendatei (1/3)

```
import java.io.*;
class Name implements Serializable {
  String name;
  String vorname;
  public Name(String n, String vn) { name = n; vorname = vn; }
}

class Student implements Serializable {
  Name name;
  int matrNr;
  double Note;
  public Student(String n, String vn, int mn) {
    name = new Name(n, vn); matrNr = mn;
  }
  public void setNote(double note) { this.note = note; }
```

9.4. Serialisierung von Objekten
Beispiel: Studentendatei (2/3)

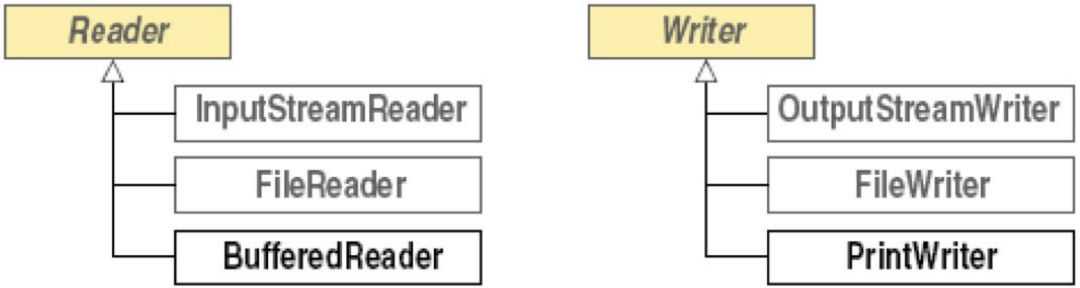
```
public static void main(String[] args) {
  ObjectOutputStream oos = null;
  try {
    Student s = new Student("Hugo", "Test", 12345678);
    oos = new ObjectOutputStream( new FileOutputStream("out.ser") );
    oos.writeObject(s);  // Schreibe Objekt s
    s.setNote(3.7);
    oos.reset();   // sonst wird nur eine weitere Referenz geschrieben
    oos.writeObject(s);  // Schreibe Objekt s nochmal
  }
  catch (/*...*/) {/*...*/}
  finally { /*...*/ oos.close();  }
```

9.4. Serialisierung von Objekten
Beispiel: Studentendatei (3/3)

```
    ObjectInputStream ois = null;
    try {
       ois = new ObjectInputStream( new FileInputStream("out.ser") );
       // Objekte von Datei einlesen und Typ umwandeln
       Student s1 = (Student) ois.readObject();
       Student s2 = (Student) ois.readObject();
       System.out.println(s1);
       System.out.println(s2);
       System.out.println(s1 == s2);  // Was wird hier ausgegeben?
    }
    catch (/*...*/) {/*...*/}
    finally { /*...*/ ois.close();  }
}
```

9.5. Formatierte Text-Ein-/Ausgabe

- Java-Ströme arbeiten byte-orientiert, nicht zeichen-orientiert
- d.h. Daten werden im Strom binär übertragen, nicht als Text
- Für die text-basierte Ein-/Ausgabe stellt Java zusätzliche Klassen zur Verfügung, u.a.: **`Reader`** und **`Writer`** stellen Basis-Methoden für die zeichenweise Ein-/Ausgabe zur Verfügung



9.5. Formatierte Text-Ein-/Ausgabe
Beispiel (1/4)

```
import java.io.*;
class Student {
  String name, vorname;
  int matrNr;  double note;

  public Student(String name, String vorname, int matrNr) {
    this.name=name;  this.vorname = vorname;  this.matrNr = matrNr;
  }
  public Student(BufferedReader reader) throws IOException {
    try {
      String line = reader.readLine();
      String[] fields = line.split(",");
      name = fields[0];  vorname = fields[1];
      matrNr = Integer.parseInt(fields[2]);
      note = Double.parseDouble(fields[3]);
    }
```

9.5. Formatierte Text-Ein-/Ausgabe
Beispiel (2/4)

```
    catch (NullPointerException e) {
      throw new IOException("Unerwartetes Dateiende");
    }
    catch (NumberFormatException e) {
      throw new IOException("Falsches Elementformat");
    }
    catch (IndexOutOfBoundsException e) {
      throw new IOException("Zu wenig Datenelemente");
    }
  }

  public void writeToStream(PrintWriter pw) {
    pw.println(name + "," + vorname + "," + matrNr + "," + note);
    pw.flush();
  }
```

9.5. Formatierte Text-Ein-/Ausgabe
Beispiel (3/4)

```
public static void main(String[] args) {
  PrintWriter pw = null;
  try {
    Student s = new Student("Hugo", "Test", 12345678);
    s.writeToStream(new PrintWriter(System.out));
    pw = new PrintWriter( new FileWriter("out.txt") );
    s.writeToStream(pw);
  }
  catch (FileNotFoundException e) {/*...*/}
  catch (IOException e) {/*...*/}
  finally {
    if (pw != null) pw.close();   // Keine IOException
  }
```

9.5. Formatierte Text-Ein-/Ausgabe
Beispiel (4/4)

```
    BufferedReader reader = null;
    try {
      reader = new BufferedReader( new FileReader("out.txt") );
      Student s1 = new Student(reader);
      System.out.println("Name,Vorname,matrNr,Note:");
      Student s3 = new Student(new BufferedReader(
                                    new InputStreamReader(System.in) ) );
      System.out.println(s1);
      System.out.println(s3);
    }
    catch (FileNotFoundException e) {/*...*/}
    catch (IOException e) {/*...*/}
    finally {
      try { reader.close(); } catch (Exception e) {/*...*/}
    }
  }
```