



# Streams

## Programmiermethodik 2

- Einstieg
- Default-Methoden
- Iteration
- Collections-Erweiterungen



# Ausblick



- Ich möchte eine Menge von Elementen (z.B. aus einer Collection) mit der gleichen Operation verarbeiten.
- Ich möchte Datenströme (z.B. Streaming aus dem Internet) verarbeiten.

- Streams
- Streams von Elementen
- Streams von Daten
- Serialisierung



# Streams

- bisher kennengelernt (u.a.): Listen, um mehrere Elemente "hintereinander" / "in einer Kette" abzulegen
- manche Anwendungsfälle
  - sequenzielle Verarbeitung einer solchen "Kette"
  - z.B. nicht alle Elemente im Hauptspeicher
  - Beispiel: Streaming eines Videos aus dem Internet

- Streams von Informationen/Daten
- Streams von Elementen (Java 8)





# Streams von Elementen (Java 8)

`java.util.stream.Stream`

- Muster bei der Verarbeitung mit Stream:
  - Erzeugen
  - Verarbeiten (beliebig viele Operationen)
  - Terminieren

Quelle → Stream → Op 1 → Op 2 → ... → Op n → Ergebnis

Erzeugen

Verarbeiten

Terminieren

- Beispiel:

```
List<Person> erwachsene = personen.stream(). // Erzeugen  
    filter(Person::istErwachsen). // Verarbeiten  
    collect(Collectors.toList()); // Terminieren
```

- Collections: Methode `stream()`

- liefert einen Stream

```
String[] namenArray = { "Karl", "Ralph", "Andi",  
    "Andy", "Mike" };
```

```
List<String> namen = Arrays.asList(namenArray);
```

```
Stream<String> namenStream = namen.stream();
```

- Utility-Klasse `Arrays` kann auch Streams erzeugen

```
Stream<String> namenStream = Arrays.stream(namenArray);
```

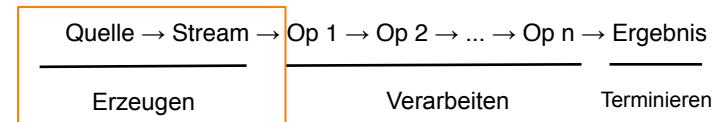
- Ausblick: Parallelisierung

- Collections-Container können auch parallelen Stream erzeugen

```
Stream<String> namenStream = namen.parallelStream();
```

- ansonsten: Stream intern parallel machen

```
Stream<String> stream = stream.parallel();
```



- Stream kann auch aus einer Liste von Werten generiert werden
  - statische Methode `of()` der Klasse `Stream`
- Beispiele

```
Stream<String> namen = Stream.of("Jan", "Hein", "Klaas");
```

```
Stream<Integer> zahlen = Stream.of(1, 4, 7, 7, 9, 7, 2);
```

- weitere Möglichkeiten:

- Stream aller Zahlen in Bereich

```
IntStream zahlen = IntStream.range(0, 100);
```

- Stream der Indizes der Buchstaben in einem String

```
IntStream zeichen = "Dies ist ein Text".chars();
```

Es gibt eigene Stream-Klassen für primitiven Datentypen (int, long, double, alle anderen Typen sind dazu kompatibel).

- auch unendliche Streams (Folgen) sind vorhanden
- Beispiel: Folge der ganzen Zahlen
- Definition:
  - Startelement
  - Berechnungsvorschrift für Nachfolge-Element

- Beispiel:

```
IntStream zahlen =
```

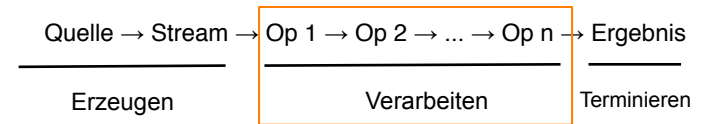
```
    IntStream.iterate(0, x -> x + 1);
```

- aber: tatsächliche Verwendung muss begrenzt werden
  - Methode `limit()`

- Beispiel:

```
zahlen.limit(10) // liefert Stream der ersten 10 Elemente
```

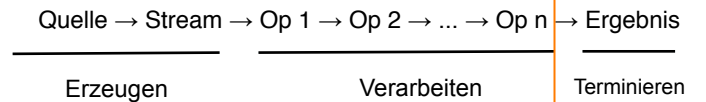
- Unterscheidung
  - zustandlose Varianten
  - zustandsbehaftete Varianten
- Beispiele
  - Filterung = zustandslos, Entscheidung für jedes Element einzeln
  - Sortierung = zustandsbehaftet, vergleich zwischen Elementen



<code>filter()</code>	Filtert alle Elemente aus dem Stream heraus, die nicht dem übergebenen <code>Predicate&lt;T&gt;</code> genügen.
<code>map()</code>	Transformiert Elemente mithilfe einer <code>Function&lt;T,R&gt;</code> vom Typ <code>T</code> auf solche mit dem Typ <code>R</code> . Im Speziellen können die Typen auch gleich sein.
<code>flatMap()</code>	Bildet verschachtelte Streams als einen flachen Stream ab.
<code>peek()</code>	Führt eine Aktion für jedes Element des Streams aus. Dies kann für Debuggingzwecke sehr nützlich sein.

<code>distinct()</code>	Entfernt alle gemäß der Methode <code>equals(Object)</code> als Duplikate erkannte Elemente aus einem Stream.
<code>sorted()</code>	Sortiert die Elemente eines Streams gemäß einem Sortierkriterium basierend auf einem <code>Comparator&lt;T&gt;</code> .
<code>limit()</code>	Begrenzt die maximale Anzahl der Elemente eines Streams auf einen bestimmten Wert. Dies ist eine Short-circuiting Operation.
<code>skip()</code>	Überspringt die ersten <code>n</code> Elemente eines Streams.





<code>forEach()</code>	Führt eine Aktion für jedes Element des Streams aus.
<code>toArray()</code>	Überträgt die Elemente aus dem Stream in ein Array.
<code>collect()</code>	Überträgt die Elemente aus dem Stream in eine Collection.
<code>reduce()</code>	Verbindet die Elemente eines Streams. Ein Beispiel ist die kommaseparierte Konkatenation von Strings. Alternativ kann man aber auch Summationen, Multiplikationen usw. ausführen, um einen Ergebniswert zu berechnen.

<code>min() / max()</code>	Ermittelt das Minimum/Maximum der Elemente eines Streams gemäß einem Sortierkriterium basierend auf einem <code>Comparator&lt;T&gt;</code> .
<code>count()</code>	Zählt die Anzahl an Elementen in einem Stream.
<code>anyMatch() / allMatch() / noneMatch()</code>	Prüft, ob es mindestens ein Element (alle Elemente/kein Element) gibt, das die Bedingung eines <code>Predicate&lt;T&gt;</code> erfüllt.
<code>findFirst() / findAny()</code>	Liefert das erste (eine beliebiges) Element des Streams, falls es ein solches gibt.

```
public class Person {  
    /**  
     * Alter der Person.  
     */  
    private int alter;  
  
    /**  
     * Name der Person  
     */  
    private String name;  
  
    public Person(String name, int alter) {  
        this.name = name;  
        this.alter = alter;  
    }  
  
    @Override  
    public String toString() {  
        return String.format("%s (%d)", name, alter);  
    }  
  
    /**  
     * Erwachsenen-Prädikat  
     *  
     * @return Liefert wahr wenn die Person älter als 18 Jahr ist.  
     */  
    public boolean istErwachsen() {  
        return alter >= 18;  
    }  
}
```

```
List<Person> personen = new ArrayList<>();  
personen.add(new Person("Micha", 43));  
personen.add(new Person("Barbara", 40));  
personen.add(new Person("Yannis", 5));  
Predicate<Person> istErwachsen = person -> person.istErwachsen();  
personen.stream().filter(istErwachsen).forEach(System.out::println);  
// Alternative mit Methodenreferenz  
personen.stream().filter(Person::istErwachsen).forEach(System.out::println);
```

– liefert

*Micha (43)*

*Barbara (40)*

# Beispiel: Mapping



```
// Mapping von Personen-Stream auf Namen-Stream
personen.stream().map(person -> person.getName())
    .forEach(name -> System.out.print(name + " "));

// Mapping von Personen-Stream auf Alter-Stream
personen.stream().map(person -> person.getAlter())
    .forEach(alter -> System.out.print(alter + " "));
```

– liefert

*Micha Barbara Yannis*

und

*43 40 5*

- Schreiben Sie einen Lambda-Ausdruck, der zum funktionalen Interface `Function<T, R>` kompatibel ist (und daher zusammen mit `map()` verwendet werden kann)
- Der Lambda-Ausdruck wandelt eine Zeichenkette in ihre Länge um

- Lesen von Zeilen aus einer Text-Datei

```
List<String> zeilen = Arrays.asList(  
    "In einem Loch im Boden, da lebte ein Hobbit."  
    "Nicht in einem feuchten, schmutzigen Loch,"  
    "wo es nach Moder riecht"  
    "und Wurmzipfel von den Wänden herabhängen." );  
Stream<String> zeilenStream = zeilen.stream();
```

- jeden String am Leerzeichen aufspalten

```
Stream<Stream<String>> text = zeilenStream.map(zeile ->  
    Stream.of(zeile.split(" ")));
```

- liefert (Stream von Stream von Strings)

<<In>, <einem>, <Loch>, <im>, <Boden, >, <da>, <lebte>, <ein>, <Hobbit.>, <<Nicht>, <in>, <einem>, <feuchten, >, <schmutzigen>, <Loch,>, <wo ...

- besser: Verflachen = innere Streams auflösen

```
Stream<Stream<String>> text = zeilenStream.flatMap(  
    zeile -> Stream.of(zeile.split(" "));
```

- liefert Stream der Wörter

# Beispiel: "Reingucken"

- Stream kann nur einmal verarbeitet werden
  - Was mache ich, wenn ich einen Zwischenstand ansehen möchte?
  - Operation: `peek()`
    - `Consumer<T>` als Parameter
    - liefert neuen Stream zurück → kann weiterverarbeitet werden
- Beispiel

```
List<Person> personen = ...  
Stream<Person> personenStream = personen.stream();  
Stream<String> alleMikes = personenStream.  
    filter(Person::istErwachsen).  
    peek(System.out::println).  
    map(Person::getName).  
    filter(name -> name.startsWith("Mi"));
```



# Beispiel: distinct() und sorted()



- Beispiel

```
Stream<Integer> zahlen = Stream.of(  
    7, 1, 4, 3, 7, 2, 6, 5, 7, 9, 8);
```

- Sortieren

```
zahlen.sorted().forEach(zahl ->  
    System.out.println(String.format("%d ", zahl)));
```

- Duplikate entfernen

```
zahlen.distinct().forEach(zahl ->  
    System.out.println(String.format("%d ", zahl)));
```

- Rückverwandlung eines Streams in eine Collection
- Beispiel

```
List<Integer> alter = alterStream.collect(  
    Collectors.toList());  
List<String> name = namenStream.collect(  
    Collectors.toCollection(ArrayList::new));
```

- Zusammenfassen aller Elemente eines Streams
- jeweils paarweise
  - alle bisherigen + das nächste
- Terminiere `reduce(T, BinaryOperator<T>)`
  - erster Parameter: Identität
  - damit wird das erste Element vereint
- Beispiel: Summe eines `int`-Streams

```
IntStream zahlen = IntStream.range(4, 23);  
int summe = zahlen.reduce(  
    0, (zahl1, zahl2) -> zahl1 + zahl2));
```

- Gegeben ist folgender Datensatz:

```
List<Person> personen =  
    Arrays.asList(new Person("Stefan", LocalDate.of(1971, MAY, 12)),  
        new Person("Micha", LocalDate.of(1971, FEBRUARY, 7)), new Person(  
            "Andi Bubolz", LocalDate.of(1968, JULY, 17)), new Person(  
            "Andi Steffen", LocalDate.of(1970, JULY, 17)), new Person(  
            "Merten", LocalDate.of(1975, JUNE, 16)));
```

Person hat jetzt Objektvariable  
geburtstag vom Typ LocalDate  
(mit Methode getMonth())

- Aufgaben
  - Filterung auf alle im Juli Geborenen
  - Extraktion des Namens
  - Ausgabe als kommagetrennte Liste (ein String)



# Streams von Daten

- engl. streams
- in Java: Objekte aus dem Package `java.io`
- abstrakte Ein-/Ausgabegeräte
- kann mit einem physikalischen Gerät verknüpft sein
  - z.B. Datei auf der Festplatte, Bildschirm
  - falls nicht: Daten bleiben im Hauptspeicher
- zwei Arten von Streams
  - Eingabeströme: Lesen
  - Ausgabeströme: Schreiben
- Daten
  - byte-basiert: 8 Bit ASCII
  - char-basiert: 16 Bit Unicode

- Konsolenfenster
- steht jederzeit automatisch zur Verfügung
- Zugriff über Klassenvariable out der Klasse System

```
static final PrintStream out;
```

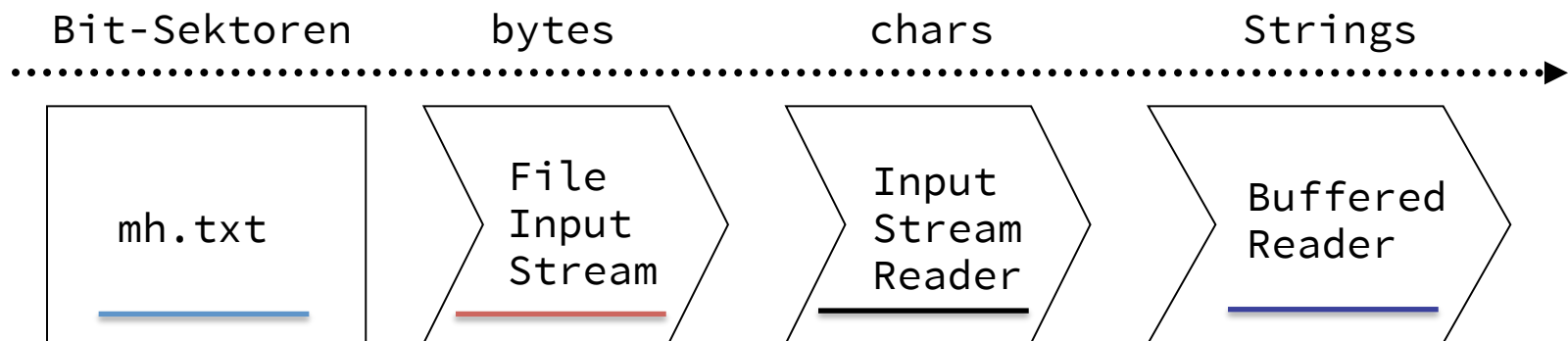
- Klasse PrintStream bietet eine Methode println(.)
- Beispiel

```
System.out.println("Hallo");
```

- Grundfunktionalität der Basis-Streams
  - Lesen
  - Schreiben
- darauf aufbauend: spezielle Streams
  - z.B. Filter, Konverter
  - benutzen Basis-Streams
- Idee zur flexiblen Kombination:
  - Verwenden der Basis-Streams
  - leiten Daten weiter
- Standard-Muster
  - Erzeugung Basis-Stream
  - Übergeben an Konstruktor speziellen Streams
  - auch genannt: Verkettung von Streams

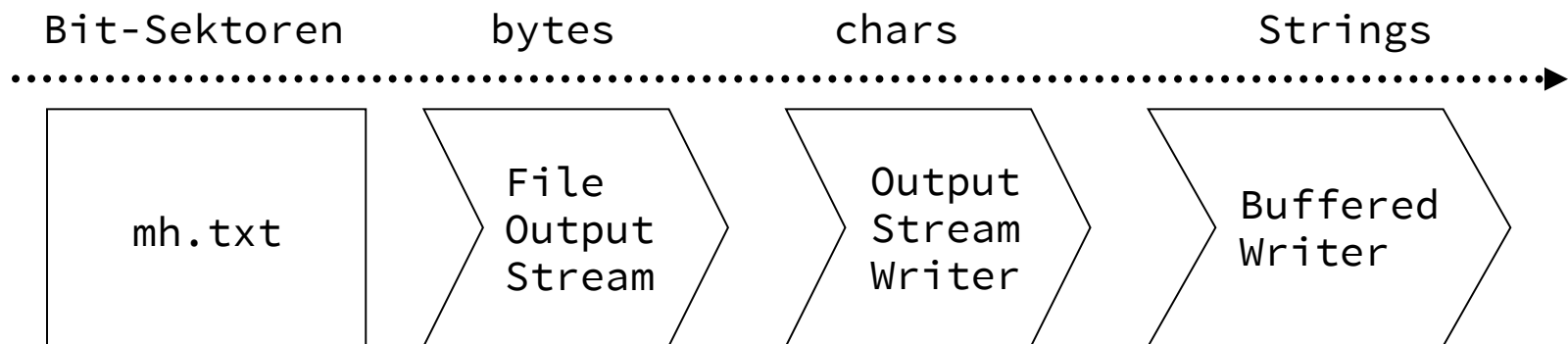


## – Beispiel 1: Verkettung zum Lesen einer Datei



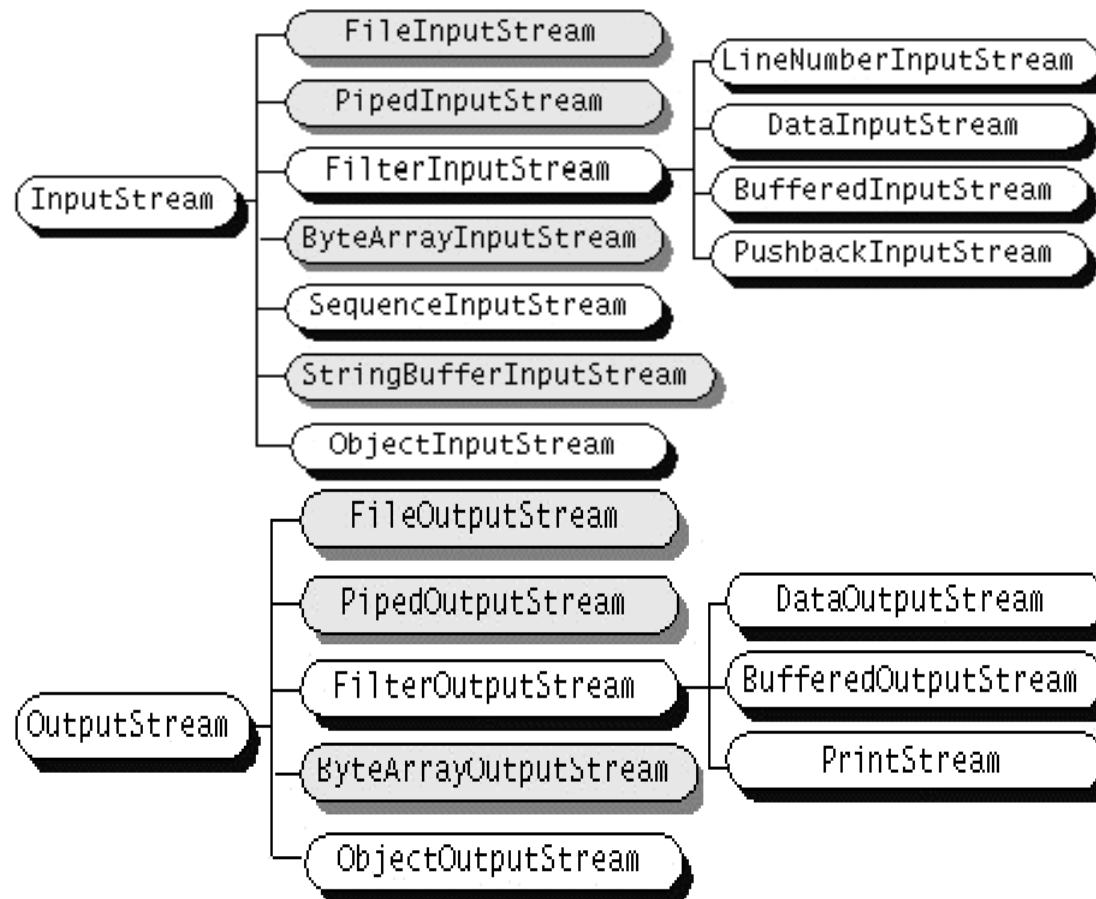
```
FileInputStream fis = new FileInputStream("mh.txt");  
InputStreamReader isr = new InputStreamReader(fis);  
BufferedReader br = new BufferedReader(isr);  
String str = br.readLine();
```

## – Beispiel 2: Verkettung zum Schreiben einer Datei



```
FileOutputStream fos = new FileOutputStream("mh.txt");  
OutputStreamWriter osw = new OutputStreamWriter(fos);  
BufferedWriter bw = new BufferedWriter(osw);  
String str = "Hallo!";  
bw.write(str); bw.newLine();
```

## – Überblick InputStreams/OutputStreams



## bis Java 6

- Schließen des Streams ist auch nach einer Exception nötig!
  - aber: oftmals Fangen verschiedener Exceptions
  - daher: `close()`-Methode in mehreren `catch`-Blöcken
  - oder im `finally`-Block
- Problem: `close()` kann selbst eine `IOException` erzeugen
  - weitere `try/catch`-Blöcke im `finally`-Block nötig
  - sehr unübersichtlich!

## – Beispiel

try-catch  
innerhalb von  
try-catch

```
FileInputStream inFile = null;
try {
    inFile = new FileInputStream("file.txt");
    // A buffer is required for the copied data
    byte[] buffer = new byte[65536];
    int len;
    // Read to buffer, write to destination
    while ((len = inFile.read(buffer)) > 0) {
        System.out.println(buffer);
    }
    inFile.close();
} catch (FileNotFoundException e) {
    // File not found
    e.printStackTrace();
    try {
        inFile.close();
    } catch (IOException e1) {
        // Error closing file
        e1.printStackTrace();
    }
} catch (IOException e) {
    // I/O error
    e.printStackTrace();
    try {
        inFile.close();
    } catch (IOException e1) {
        // Error closing file
        e1.printStackTrace();
    }
}
```

mehrere  
Aufrufe  
von close()


- Erweiterung des `try`-Blocks um die Angabe von Ressourcen

```
try (Ressource1; Ressource2; ...)  
{  
    <Anweisung>  
    ...  
}
```

- Ressource
  - Objekt, das das Interface `java.lang.AutoCloseable` implementiert
  - z.B. alle Stream-Klassen aus `java.io`
- Ressource nach Beendigung des `try`-Blocks automatisch geschlossen
  - auch nach einer `Exception`
  - automatischer Aufruf der `close()`-Methode

- Beispiel

Autoclosable  
in try-Aufruf



```
try (FileInputStream inFile = new FileInputStream("file.txt");) {  
    // A buffer is required for the copied data  
    byte[] buffer = new byte[65536];  
    int len;  
    // Read to buffer, write to destination  
    while ((len = inFile.read(buffer)) > 0) {  
        System.out.println(buffer);  
    }  
} catch (FileNotFoundException e) {  
    // File not found  
    e.printStackTrace();  
} catch (IOException e) {  
    // I/O error  
    e.printStackTrace();  
}
```

kein close()-  
Aufruf  
notwendig!

- Schreiben Sie das Skelett einer Klasse `GeraeuschSensor`
- Beim Verbinden mit dem Sensor (Konstruktor) und beim Trennen der Verbindung (`verbindungBeenden()`) kann eine `IOException` auftreten
- Stellen Sie sicher, dass der `GeraeschSensor` zusammen mit `try-with-resources` verwendet werden kann
- Schreiben Sie einen kleinen Code-Abschnitt in dem ein Objekt der Klasse erzeugt wird, ein Wert (Lautstärke) ausgelesen wird und die Verbindung zum Sensor wieder geschlossen wird





# Serialisierung

- Problem
  - Speicherung von komplexen Objekten in einem Datenstrom
  - Datei, Netz, ...
- bisher
  - Die interne Struktur jedes Objekts müsste komplett "nachprogrammiert" werden
  - Objektvariablen mit aktuellen Werten inkl. aller referenzierten Objekte
- Lösung
  - Serialisierung
  - Objekte werden "automatisch" in ein Byte-Format konvertiert und in einen Datenstrom geschrieben
- Deserialisierung
  - Java-Objekte werden aus einem Byte-Datenstrom gelesen und komplett wiederhergestellt

- Voraussetzung:
  - zu serialisierendes Objekt muss Interface `Serializable` implementieren
- Vorgehensweise bei der Programmierung:
  - Erzeugen eines `ObjectOutputStream` mit Übergabe eines existierenden `OutputStream` an den Konstruktor
    - wohin soll das Objekt geschrieben werden?
  - Methode `writeObject(Object obj)` zum Schreiben eines Objekts aufrufen
    - ggf. mehrfach für mehrere Objekte

- Folgende Daten werden in den Output-Stream geschrieben
  - Klasse des als Argument übergebenen Objekts
  - Signatur der Klasse
  - alle nicht-statischen, nicht-transienten Objektvariablen des Objekts
    - inkl. aller geerbten Objektvariablen
  - Objektvariablen können selbst wieder Objekte enthalten!

- Beispiel: Methode zum Serialisieren eines Objektes

```
private static void serialize(Serializable object, String filename) {  
    try (ObjectOutputStream dateiStream = new ObjectOutputStream(  
        new FileOutputStream(filename))) {  
        dateiStream.writeObject(object);  
    } catch (IOException e) {  
        System.out.println("Failed to serialize object");  
    }  
}
```

- Schreiben einzelner primitiver Datentypen in den selben Stream

  - `void writeBoolean (boolean value)`

  - `void writeBytes (String string)`

  - `void writeDouble (Double double)`

- Ausschließen einzelnen Objektvariablen

  - werden nicht serialisiert

  - Markierung um Schlüsselwort `transient`

  - `private transient String password;`

- Objekte welcher der folgenden Klassen dürfen serialisiert werden, welche nicht?
  - Object
  - String
  - File
  - Date
  - OutputStream
  - Scene
  - Integer
  - System

- Voraussetzungen:
  - class – Datei des Objekts muss auffindbar sein
  - sonst sind die Methoden des Objekts nicht bekannt
  - Klassenstruktur vorliegenden Objekts (Byteformat) und aktuelle Klassendefinition müssen zueinander kompatibel sein



- Beispiel: Methode zum Deserialisieren eines Byte-Streams

```
private static Object deserialize(String filename) {  
    Object object = null;  
    try (ObjectInputStream fileStream = new ObjectInputStream(  
        new FileInputStream(filename))) {  
        object = fileStream.readObject();  
    } catch (ClassNotFoundException | IOException e) {  
        System.out.println("Failed to deserialize file.");  
    }  
    return object;  
}
```

- neues Objekt erzeugen
- Objektvariablen mit Default-Werten initialisieren
- aber keinen Konstruktor aufrufen!
- serialisierte Daten lesen
- entsprechenden Objektvariablen zuweisen

- mögliches Problem
  - Klassendefinition eines serialisierten Objekts hat sich verändert hat
  - Deserialisierung schlägt fehl
  - Objekt kann nicht mehr gelesen werden!
- Java-Lösung
  - `writeObject` erzeugt aus den Klasseninformationen eine eindeutige Versionsnummer `serialVersionUID`
  - speichert diese mit ab
  - falls in der Klassendefinition eigene Konstante existiert, wird dieser Wert verwendet

```
static final long serialVersionUID = ..
```

  - Versionsnummern stimmen bei Deserialisierung nicht überein?
  - JVM verweigert Deserialisierung

- Streams
- Streams von Elementen
- Streams von Daten
- Serialisierung

- Dieser Satz Folien basiert teilweise Teilen auf folgender Literatur:  
Michael Inden: Java 8 – Die Neuerungen, dpunkt Verlag, 2014
- Dieser Satz Folien basiert teilweise auf Vorlesungsfolien von Prof.  
Martin Hübner, Hochschule für Angewandte Wissenschaften  
Hamburg