TEKO
SCHWEIZERISCHE FACHSCHULE

Objektorientiertes Programmieren 2

Eingaben und Ausgaben

Stephan Kessler, BSc in Systems Engineering

Kontakt: stephan.kessler@edu.teko.ch

Vorbereitung

Lesen Sie folgende Abschnitte im Buch ***Sprechen Sie Java?*** als Vorbereitung für dieses Thema. Beachten Sie, dass bei Angaben von Unterkapiteln nur diese auch gelesen werden müssen:

- Ausnahmebehandlung (S. 255-266)
- Datenströme (S. 279-300)

Exceptions

- Beim Ablauf eines Programmes können Fehler während der Ausführung des Programmes auftreten. In Java werden diese Situationen **Exceptions** genannt.

  - Beispiel: Das Programm dividiert mit einer Int-Variable, die zur Laufzeit auf 0 gesetzt wurde. Eine Null-Divison ist mathematisch nicht möglich.
    → *ArithmeticException*

  - Eine Datei wird zum Lesen geladen. Die Datei ist jedoch nicht auf dem Speichermedium vorhanden.
    → *IOException*

  - In einer Methode wird als Parameter ein Objekt aufgenommen und in der Methode wird auf dieses Objekt zugegriffen. Beim Methodenaufruf wird jedoch ein nicht instanziiertes Objekt als Argument übergeben.
    → NullPointerException

Aufgabe

- Erstellen Sie ein Programm mit einer main-Methode.
- Deklarieren Sie in Ihrer main-Methode zwei Int-Variablen: *divident*, *divisor*
- Setzten Sie für die Variablen folgende Werte: *divident* = 10, *divisor* = 2
- Speichern Sie folgende Berechnung in eine weitere Int-Variable ab: *divident / divisor*
- Geben Sie mit einem weiteren Befehl das Ergebnis an die Konsole aus.
- Führen Sie Ihr Programm aus.
- Ändern Sie nun den Wert des *divisors* zu 0 (anstelle 2).
- Führen Sie Ihr Programm aus.
- Was wird ausgegeben? Wird die Ausgabe der Variable durchgeführt?

Exceptions

```
int divident = 10;
int divisor = 0;

int ergebnis = divident / divisor
```

Sie bemerken, dass bei einer unbehandelten Exception das Programm beendet wird. Dies ist notwendig, da das Programm nicht weiss, wie es mit dieser besonderen Situation umzugehen hat.

Exceptions

```
int divident = 10;
int divisor = 0;

try {
    int ergebnis = divident / divisor;
} catch (ArithmeticException e) {
    System.out.println(e.getMessage());
}
```

Mit den Schlüsselwörtern ***try*** und ***catch*** kann der Programmier angebene, wie in so einer Situation reagiert werden soll.

Exceptions

```java
int divident = 10;
int divisor = 0;

try {
    int ergebnis = divident / divisor;
} catch (ArithmeticException e) {
    System.out.println(e.getMessage());
}
```

Im ***try***-Bereich wird der Befehl ausgeführt, welcher eine Exception auswerfen kann. Hier können auch weitere Befehle stehen, die keine Exception auswerfen. z.B die Ausgabe der Variable *ergebnis*.

Exceptions

```
int divident = 10;
int divisor = 0;

try {
    int ergebnis = divident / divisor;
} catch (ArithmeticException e) {
    System.out.println(e.getMessage());
}
```

In den Klammern der ***catch***-Zeile wird die zu abfangende Exception als Parameter angegeben. Exceptions sind in Java spezielle Klassen und können beim Eintreten der Exception als Objekt aufgenommen werden. Im ***catch***-Bereich steht in diesem Beispiel nun das Objekt *e* zur Verfügung.

Ihnen stehen die entsprechenden Methoden der Exception-Klasse zur Verfügung.

Exceptions

```
int divident = 10;
int divisor = 0;

try {
    int ergebnis = divident / divisor;
} catch (ArithmeticException e) {
    System.out.println(e.getMessage());
}
```

Im ***catch***-Bereich wird beschrieben, was bei einer Exception-Situation durchgeführt werden soll. Da es sich um einen Bereich handelt, können auch hier mehrere Befehle hinterlegt werden.

In diesem Beispiel wird die Exception-Nachricht auf die Konsole ausgegeben. Das Programm fährt ausserhalb des ***try/catch***-Blockes weiter, wenn kein Unterbruch im ***catch***-Bereich (z.B. *return*) vorhanden ist.

Exceptions

```
String eingabe = new java.util.Scanner(System.in).nextLine();
int divident = 10;
int divisor = 0;

try {
    divisor = Integer.parseInt(eingabe);
    int ergebnis = divident / divisor;
} catch (ArithmeticException e) {
    System.out.println(e.getMessage());
} catch (NumberFormatException e) {
    System.out.println("Falsche Eingabe!");
}
```

Sie haben auch die Möglichkeit mehrere ***catch***-Blöcke zu verwenden, sollte die Situation bestehen, dass verschiedene Exceptions in einem ***try***-Bereich eintreten können.

In diesem Beispiel wird der *divisor* durch eine Benutzer-Eingabe gesetzt. Der String-Wert wird in einen Integer umgewandelt. Wie Sie bereits wissen, kann bei dieser Umwandlung ein Fehler auftreten, wenn der Benutzer z.B. für die Eingabe Buchstaben verwendet.
→ *NumberFormatException*

Exceptions

```
String eingabe = new java.util.Scanner(System.in).nextLine();
int divident = 10;
int divisor = 0;

try {
    divisor = Integer.parseInt(eingabe);
    int ergebnis = divident / divisor;
} catch (ArithmeticException e) {
    System.out.println(e.getMessage());
} catch (NumberFormatException e) {
    System.out.println("Falsche Eingabe!");
}
```

Beachten Sie dabei, dass *try*-Blöcke bei eine Exception **unterbrochen** werden. Sind in einem *try*-Block mehrere Anweisungen, kann es durchaus bei einem Fehlerfall vorkommen, dass nicht alle Anweisungen ausgeführt werden.

Im oberen Beispiel wird bei einer falschen Eingabe beim Parsing-Befehl eine *NumberFormatException* ausgeworfen. Durch den Sprung zum entsprechenden *catch*-Block wird die Zeile *int ergebnis = divident / divisor* **nicht mehr ausgeführt**.

Exceptions

```
// überprüfe und erstelle ggf. Datei

try {
    // erzeuge Zugriff auf Datei
} catch (IOException e) {
    // erstelle Datei
}
```

Da Exceptions **Ausnahmezustände** behandeln, sollten diese nicht für den normalen Programmablauf (controll flow) verwendet werden (**schlechter Programmier-Stil**).

Wenn Sie z.B. ein Programm schreiben, welches erst eine Datei erstellen soll, wenn diese noch nicht existiert, dann sollten Sie die Überprüfung der Datei nicht mit einem Try-Catch-Block durchführen, da es keinen Ausnahmezustand darstellt, dass die Datei fehlt.

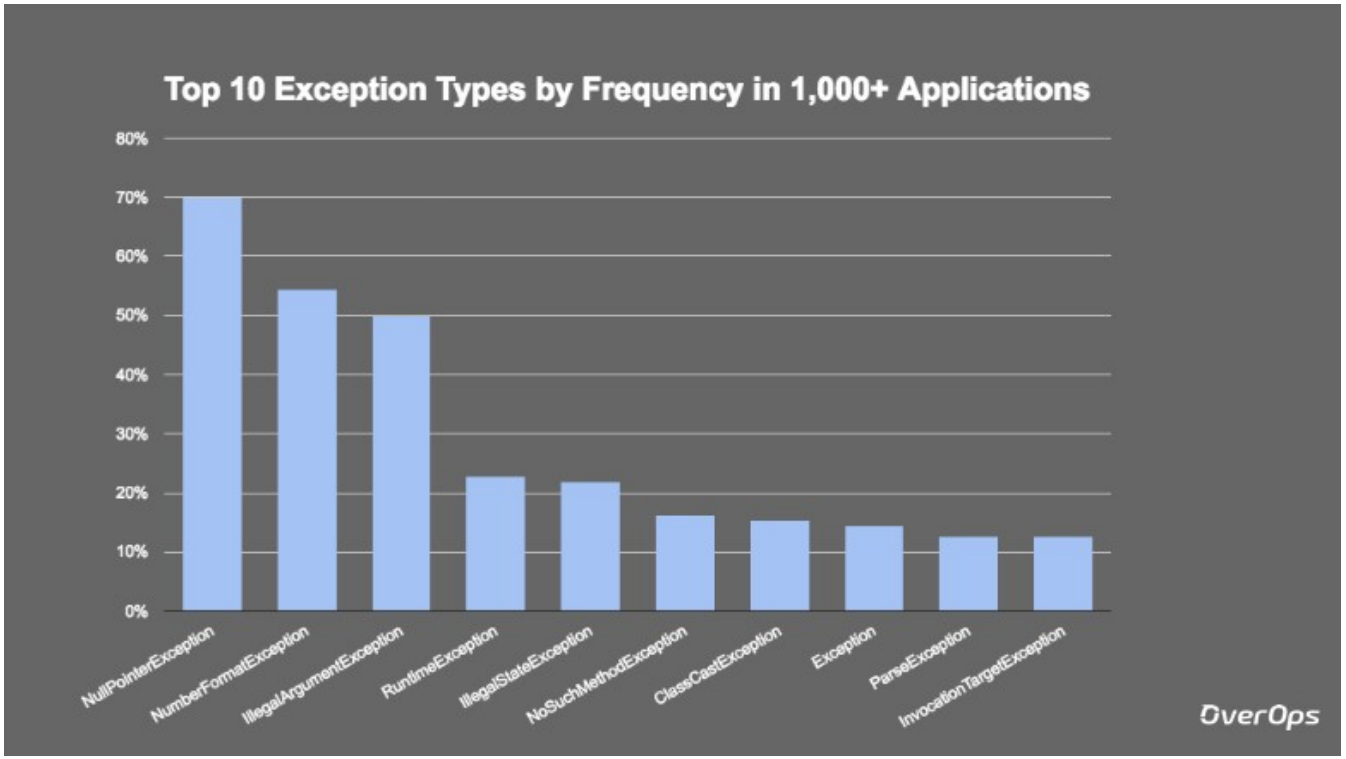
Exceptions

- **ArithmeticException**
  Arithmetic error, such as divide-by-zero.
- **ArrayIndexOutOfBoundsException**
  Array index is out-of-bounds.
- **ArrayStoreException**
  Assignment to an array element of an incompatible type.
- **UnsupportedOperationException**
  An unsupported operation was encountered.
- **NumberFormatException**
  Invalid conversion of a string to a numeric format.
- **IllegalMonitorStateException**
  Illegal monitor operation, such as waiting on an unlocked thread.
- **IllegalStateException**
  Environment or application is in incorrect state.
- **IllegalThreadStateException**
  Requested operation not compatible with the current thread state.
- **IndexOutOfBoundsException**
  Some type of index is out-of-bounds.
- **NegativeArraySizeException**
  Array created with a negative size.
- **NullPointerException**
  Invalid use of a null reference.
- **IllegalArgumentException**
  Illegal argument used to invoke a method.
- **SecurityException**
  Attempt to violate security.
- **StringIndexOutOfBounds**
  Attempt to index outside the bounds of a string.
- **ClassCastException**
  Invalid cast.

Aufgabe

- Erweitern Sie das Programm des Praktikums „Fliessender Taschenrechner“ (Vorgabe) so, dass falsche Eingaben über das Abfangen der Exception behandelt werden. Sollte der Benutzer eine falsche Eingabe tätigen, soll im Ergebnisfenster eine entsprechende Nachricht dargestellt werden.

Null und Referenzen



by overops.com
https://blog.overops.com/the-top-10-exceptions-types-in-production-java-applications-based-on-1b-events/

NullPointerException ist die am häufigsten auftretende Exception-Situation. Sobald eine Referenz auf ein „Null-Objekt“ zeigt und diese Referenz verwendet wird, wird eine *NullPointerException* ausgeworfen.

Null und Referenzen

```
MeineKlasse mein_objekt1;
mein_objekt1 = new MeineKlasse();
```

Wie schon angedeutet sind Variablen, welche ein Objekt halten, **Referenzen zum Objekt**. (Sie beinhalten nicht das Objekt, sondern die Referenz zum Objekt („Link“)).

In diesem Beispiel besitzt die Variable *mein_objekt1* die Referenz zum Objekt, welches durch die Zeile *new MeineKlasse()* instanziiert wurde.

Aufgabe

```
MeineKlasse mein_objekt1;
MeineKlasse mein_objekt2;
MeineKlasse mein_objekt3;

mein_objekt1 = new MeineKlasse();
mein_objekt2 = mein_objekt1;
mein_objekt3 = mein_objekt1;
mein_objekt3.text = "Hallo";
mein_objekt2.text = "Welt";

System.out.println(mein_objekt3.text);
System.out.println(mein_objekt1.text);
```

- Mit der beschriebenen Logik, was wird an die Konsole ausgegeben?

Null und Referenzen

```
MeineKlasse mein_objekt1;
mein_objekt1.rechne();
mein_objekt1 = new MeineKlasse();
```

Da nach der Deklaration noch keine Objekt-Referenz der Variable zugewiesen wurde, besitzt die Variable eine Referenz zur **Null-Referenz**.

Null und Referenzen

```
MeineKlasse mein_objekt1;
mein_objekt1.rechne();
mein_objekt1 = new MeineKlasse();
```

Zugriffe auf das Objekt in diesem Moment führen daher auf eine *NullPointerException*. → Die Referenz zum Objekt ist noch nicht vorhanden.

Null und Referenzen

```
MeineKlasse mein_objekt1;
mein_objekt1 = new MeineKlasse();
mein_objekt1.rechne();
mein_objekt1 = null;
mein_objekt1.rechne();
```

Das Schlüsselwort ***null*** steht für die Null-Referenz. Daher können Sie die Referenz mit diesem Schlüsselwort wieder zurücksetzten.

In diesem Beispiel wird durch die letzte Zeile eine *NullPointerException* ausgeworfen.

Null und Referenzen

```
if(mein_objekt1 != null){
    mein_objekt1.rechne();
}
```

Das Schlüsselwort können Sie aber auch für die Abfrage einer Null-Referenz nutzen.

In diesem Beispiel wird die Methode *rechne()* ausgeführt, wenn *mein_objekt* auf ein Objekt zeigt bzw. keine Null-Referenz besitzt.

Garbage Collector

```
MeineKlasse mein_objekt1;
mein_objekt1 = new MeineKlasse();
mein_objekt1.rechne();
mein_objekt1 = null;
```

Java behält den Überblick über alle Referenzen und bemerkt, wenn keine Referenz zu einem Objekt mehr besteht. Im obigen Beispiel ist dies nach der letzten Zeile der Fall.

Da die Referenz in keiner weiteren Variable abgespeichert ist und die einzige Variable, die die Referenz besitzt, mit *null* überschrieben wird, kann auf dieses Objekt nicht mehr zugegriffen werden.

Da Java dies bemerkt, wird das Objekt auch aus dem Arbeitsspeicher gelöscht (Garbage Collector).

Schreiben von Zeichen

```
String output_text = "Hello World";
String output_path = "output.txt";

try {
    FileWriter writer = new FileWriter(output_path);
    writer.write(output_text);
    writer.close();
} catch(IOException e){
    System.out.println(e.getMessage());
}
```

In Java können Sie Zeichen mit der Klasse ***FileWriter*** in eine Datei schreiben. Da Sie dazu auf den Datenträger zugreifen, können IOException-Fehler ausgeworfen werden. Daher wird hier ein *try/catch*-Block benötigt.

Schreiben von Zeichen

```
String output_text = "Hello World";
String output_path = "output.txt";

try {
    FileWriter writer = new FileWriter(output_path);
    writer.write(output_text);
    writer.close();
} catch(IOException e){
    System.out.println(e.getMessage());
}
```

Mit einem String geben Sie den Pfad der Datei an. Dabei können Sie einen relativen oder absoluten Pfad verwenden. Der relative Pfad geht vom Ort des ausgeführten Programmes aus während der absolute Pfad vom Anfang der gesamten Pfadstruktur (Root-Verzeichnis) ausgeht.

In diesem Beispiel wurde ein relativer Pfad verwendet. Die Datei *output.txt* wird an dem Ort erstellt, wo sich die ausgeführte Datei befindet.

Beispiel absoluter Pfad (Windows): „C:\Datei\MeinProgramm\Daten\Text\output.txt“
Beispiel relativer Pfad (Windows): „Daten\Text\output.txt“

Schreiben von Zeichen

```
String output_text = "Hello World";
String output_path = "output.txt";

try {
    FileWriter writer = new FileWriter(output_path);
    writer.write(output_text);
    writer.close();
} catch(IOException e){
    System.out.println(e.getMessage());
}
```

Mit der Methode *write*() können Sie einen String in die Datei schreiben. Jeder zusätzliche *write*-Befehl fügt der Datei weitere Zeichen hinzu. Die Zeichen werden dabei auf der gleichen Zeile weitergeführt (keine neue Zeile).

Schreiben von Zeichen

```
String output_text = "Hello World";
String output_path = "output.txt";

try {
    FileWriter writer = new FileWriter(output_path);
    writer.write(output_text);
    writer.close();
} catch(IOException e){
    System.out.println(e.getMessage());
}
```

Sobald der Schreibvorgang beendet wurde und der Zugriff auf die Datei nicht mehr benötigt wird, muss mit der Methode *close()* der Schreibzugriff der Datei wieder freigegeben werden.

Aufgabe

- Schreiben Sie ein Programm welches eine Datei mit folgende Zahlenreihe füllt: 2, 4, 6, 8, 10, 12, 14 … 1000
- Fügen Sie nach jeder Zahl das zeichen *„\n“* ein. Dieses Zeichen verursacht, dass weitere Zeichen auf einer neuen Zeile dargestellt werden. Jede Zahl sollte dadurch auf einer eigenen Zeile stehen.

Lesen von Zeichen

```
String input_path = "output.txt";

try {
    FileReader reader = new FileReader(input_path);
    char[] input = new char[50];
    reader.read(input);
    System.out.println(input);
    reader.close();
} catch (IOException e) {
    System.out.println(e.getMessage());
}
```

Mit der Klasse *FileReader* können Sie Zeichen aus einer Datei lesen.

Lesen von Zeichen

```
String input_path = "output.txt";

try {
    FileReader reader = new FileReader(input_path);
    char[] input = new char[50];
    reader.read(input);
    System.out.println(input);
    reader.close();
} catch (IOException e) {
    System.out.println(e.getMessage());
}
```

Ihnen steht dabei die Methode *read()* zur Verfügung, welche ein Char-Array als Argument aufnimmt und dieses mit den Zeichen in der Datei füllt.

Lesen von Zeichen

```
String input_path = "output.txt";

try {
    FileReader reader = new FileReader(input_path);
    char[] input = new char[50];
    reader.read(input);
    System.out.println(input);
    reader.close();
} catch (IOException e) {
    System.out.println(e.getMessage());
}
```

Auch beim Lesen muss mit der Methode *close()* am Schluss der Zugriff wieder frei gegeben werden.

Schreiben Buffered

```
String output_text = "Hello World";
String output_path = "output_txt";

try {
    FileWriter writer = new FileWriter(output_path);
    BufferedWriter buffered_writer = new BufferedWriter(writer);
    buffered_writer.write(output_text);
    buffered_writer.newLine();
    buffered_writer.write(output_text);
    buffered_writer.close();
} catch (IOException e) {
    System.out.println(e.getMessage());
}
```

Ein Zugriff auf eine Datei kann relativ lange dauern. Daher ist es ungünstig, wenn nach jedem *write*-Befehl die Datei verändert wird. Mit der *BufferedWriter*-Klasse wird dies verhindert. Diese Klasse schreibt erst, wenn eine gewisse Menge an Zeichen sich angesammelt hat oder der Zugriff auf die Datei beendet wird (Befehl *close*).

Diese Klasse ist eine Erweiterung von *FileWriter*. *FileWriter* wird daher beim Konstruktor als Argument aufgenommen.

Schreiben Buffered

```
String output_text = "Hello World";
String output_path = "output_txt";

try {
    FileWriter writer = new FileWriter(output_path);
    BufferedWriter buffered_writer = new BufferedWriter(writer);
    buffered_writer.write(output_text);
    buffered_writer.newLine();
    buffered_writer.write(output_text);
    buffered_writer.close();
} catch (IOException e) {
    System.out.println(e.getMessage());
}
```

Mit dieser Klasse steht Ihnen auch die Methode *newLine()* zur Verfügung, wodurch weitere Zeichen auf einer separaten Zeile erstellt werden. Diese Methode kann auch mehrfach ausgeführt werden.

Lesen Buffered

```
String input_path = "output.txt";

try {
    FileReader reader = new FileReader(input_path);
    BufferedReader buffered_reader = new BufferedReader(reader);
    String line = buffered_reader.readLine();
    System.out.println(line);
    buffered_reader.close();
} catch (IOException e) {
    System.out.println(e.getMessage());
}
```

Um auch beim Lesen von Zeichen aus Dateien den effektiven Datei-Zugriff gering zu halten, existiert die Klasse *BufferedReader*.

Lesen Buffered

```
String input_path = "output.txt";

try {
    FileReader reader = new FileReader(input_path);
    BufferedReader buffered_reader = new BufferedReader(reader);
    String line = buffered_reader.readLine();
    System.out.println(line);
    buffered_reader.close();
} catch (IOException e) {
    System.out.println(e.getMessage());
}
```

Hierbei wird Ihnen die Methode *readLine()* angeboten, wodurch Sie direkt eine ganze Zeile in einen String abspeichern können. Dabei wird beim wiederholten Aufruf **die nächste** Zeile gelesen. Sie können sich das mit einem versteckten Zeiger vorstellen, welcher bei jedem *readLine*-Befehl auf die nächste Zeile verschoben wird. Überschreitet diese Methode die letzte Zeile, wird anstelle eines Strings die *null*-Referenz zurückgegeben.

Überblick Ströme

Zeichenströme

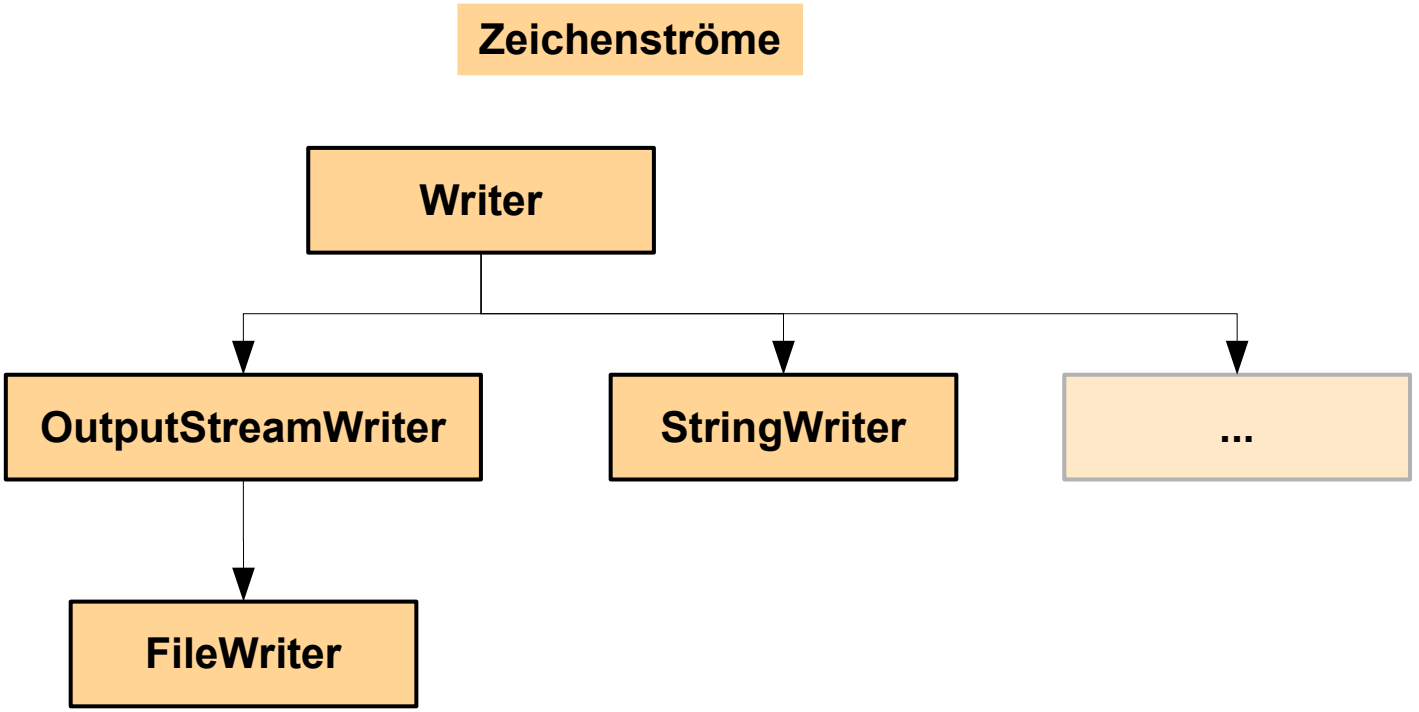
Byteströme

In Java können Sie den Ressourcen-Zugriff in zwei Arten Unterteilen:

- **Zeichenströme**: Zeichen in Form von ASCII oder Unicode. Wird oft für das Schreiben und Lesen von menschlich lesbaren Informationen genutzt (z.B. Abspeichern von Namen in eine Datei). Beispiel: *Hans \nPeter \nPaula*

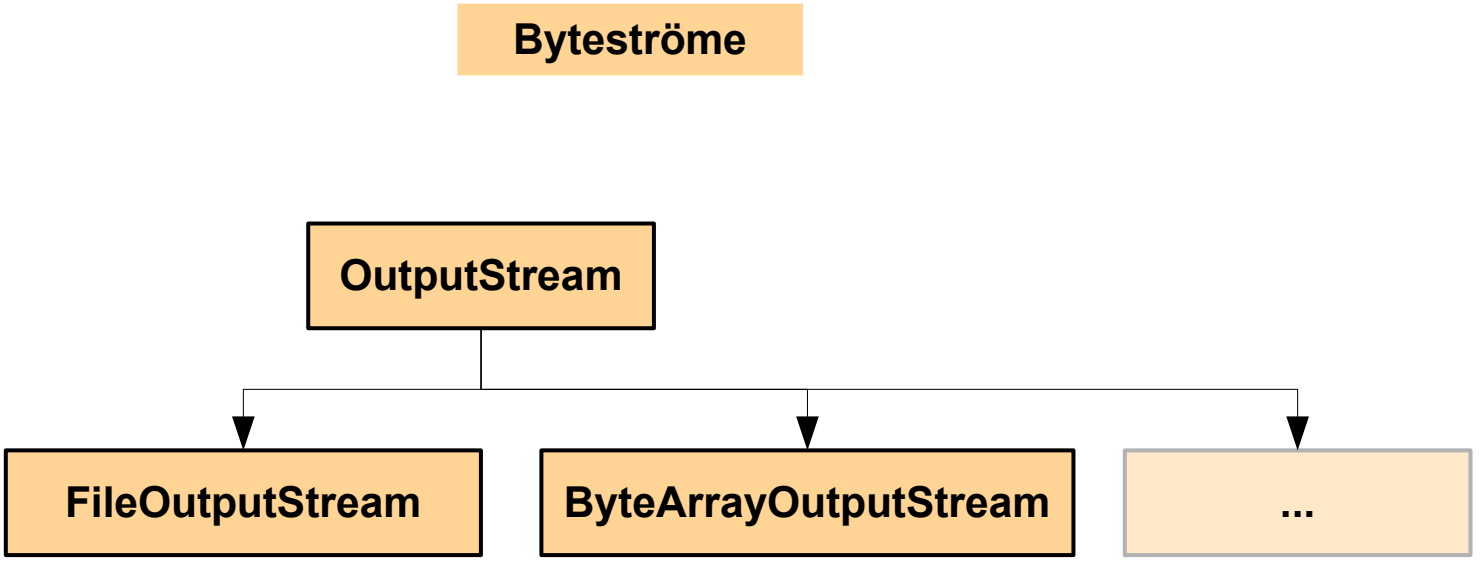
- **Byteströme**: Werte in der Grundform ohne Berücksichtigung von Zeichentabellen (ASCII bzw. Unicode). Direktes betrachten der Werte ist erschwert. Liegt im binären System und ist als Byte-Code gruppiert. Kann auch in hexadezimaler Form betrachtet werden.
  Beispiel: *4FEA127DD13FA8*

Überblick Ströme



Zeichenströme haben Sie in den vorherigen Folien mit den Klassen *FileReader*, *FileWriter*, *BufferedReader* und *BufferedWriter* kennen gelernt. Diese bedienen sich entsprechend der abstrakten Klassen *Reader* und *Writer* (und deren Unterklassen).

Überblick Ströme



Bei Byteströmen gibt es Operationen, um einzelne Bytes oder ganze Bytefolgen zu schreiben und zu lesen. Alle Byteströme sind von den abstrakten Klassen *Inputstream* und *Outputstream* abgeleitet.