

Mit Character- und Bytestreams arbeiten

Grundlagen zu Streams

Was sind Streams?

Die sequenziellen Ein- und Ausgaben werden in Java mit **Streams** (Strömen) realisiert.

- ✓ Mit einem **Input-Stream** (Eingabestrom) können Daten aus einer beliebigen Quelle gelesen werden, z. B. aus der Tastatur, einer Datei, einem String oder einer Netzwerkverbindung.
- ✓ Mit einem **Output-Stream** (Ausgabestrom) können Daten in ein beliebiges Ziel geschrieben werden, z. B. auf den Bildschirm, in eine Datei oder über eine Netzwerkverbindung.

Sie können jeden Datenstrom nach dem gleichen Prinzip verarbeiten, egal woher er kommt und für welches Ziel er bestimmt ist. Ein Datenstrom kann die Daten aber immer nur in eine Richtung senden.

Seit Java 8 gibt es als Erweiterung der Collections eine neue Stream-API, die auf Lambda-Ausdrücken basiert. Diese hat nichts mit den hier behandelten Character- und Bytestreams zu tun.



Ergänzende Lerninhalte: *Funktionale Programmierung.pdf*

Eine Einführung zu Lambda-Ausdrücken finden Sie im oben angegebenen BuchPlus-Dokument.

Stream-Typen

Grundsätzlich gibt es zwei Typen von Streams.

- ✓ Zeichenorientierte Streams, sogenannte **Character-Streams**, transportieren Unicode-Zeichen mit einer Transportbreite von 16 Bit. Sie stehen für die Ein- und Ausgabe von Texten zur Verfügung.
- ✓ Byteorientierte Streams, sogenannte **Byte-Streams**, transportieren jeweils 8 Bit. Hiermit wird die Ein- und Ausgabe von Daten, die im Byte-Format dargestellt werden, beispielsweise ausführbare Programmdateien, durchgeführt.

Streams und ihre Klassen

Die Klassen zur Verarbeitung von Streams sind im Package `java.io` zusammengefasst. Die Superklassen für die Ein- und Ausgabe der Streams sind abstrakte Klassen. Abhängig von der Superklasse werden verschiedene Interfaces verwendet:

- ✓ `Closable` zum Schließen von Ein- bzw. Ausgabeströmen
- ✓ `Readable` zum Lesen von Character-Streams

- ✓ Flushable zum Leeren von Ausgabeströmen
- ✓ Appendable, um Daten an Character-Streams anzuhängen

Stream-Typ	Transportbreite	Für die Eingabe verwendete ...		Für die Ausgabe verwendete ...	
		... Superklasse	... Interfaces	... Superklasse	... Interfaces
Character-Stream	16 Bit (1 Unicode-Zeichen)	Reader	Closable, Readable	Writer	Closable, Flushable, Appendable
Byte-Stream	8 Bit (1 Byte)	InputStream	Closable	OutputStream	Closable, Flushable

Abstrakte Klassen können nicht instanziiert werden. Von den Superklassen ist eine Vielzahl von Klassen abgeleitet, die die Verbindung zu den speziellen Ein- und Ausgabeströmen herstellen.

Die Namen der Klassen, die von den beiden Superklassen für Character-Streams abgeleitet sind, enden auf `Reader` bzw. `Writer`. Entsprechend enden die Namen der Klassen, die von den beiden Superklassen für Byte-Streams abgeleitet sind, auf `InputStream` bzw. `OutputStream`.

Fehlerbehandlung

Bei fast allen Ein- bzw. Ausgabeoperationen können Fehler auftreten, z. B. beim Öffnen, Lesen und Schreiben in eine Datei. Solche Fehler lösen eine Ausnahme vom Typ `IOException` aus. Aus diesem Grund müssen all diese Operationen in einem `try-catch`-Block eingeschlossen sein. Tritt z. B. beim Schreiben in eine Datei ein Fehler auf, kann der `catch`-Block verwendet werden, um eine entsprechende Meldung auszugeben und die Datei zu schließen.

Schließen von Ressourcen

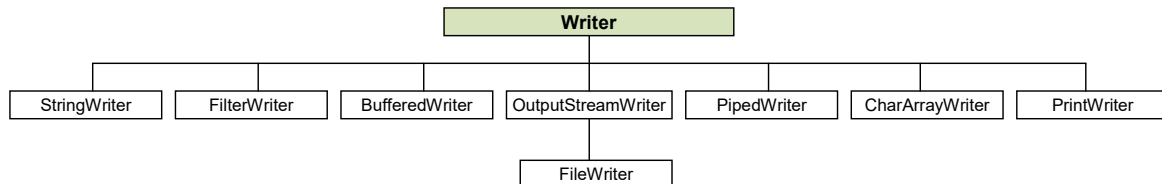
Bei der Arbeit mit Streams geöffnete Ressourcen müssen wieder geschlossen werden. Dies gilt für jede Art von Ressource wie beispielsweise Dateien und Datenbankzugriffe. Da das korrekte Schließen der Ressourcen sehr aufwendig ist, bietet Java seit der Version 7 mit den **selbst-schließenden Ressourcen** hier eine Vereinfachung des Programmcodes. Durch eine Erweiterung des Statements `try` (`try with resources`) werden alle in der `try`-Anweisung deklarierten Ressourcen durch die VM automatisch geschlossen, wenn der `try`-Block abgearbeitet wurde. Der Compiler erstellt den dafür benötigten Code von selbst. Die automatisch zu schließenden Ressourcen werden dabei in runden Klammern angegeben ①.

```
try (OutputStream out = new FileOutputStream(...)) { ①
    out.write(...);
}
```

Character-Streams schreiben

Hierarchie der Klassen für die Ausgabe

Von der Superklasse `Writer` werden alle Klassen zur Ausgabe von Character-Streams abgeleitet, z. B. die Klasse `FileWriter` zum Schreiben in eine Datei.



Methoden der abstrakten Superklasse `Writer`

Die folgenden Methoden werden in der Klasse `Writer` zur Verfügung gestellt.

<code>Writer()</code>	Konstruktor der Klasse <code>Writer</code> , der das Öffnen des Ausgabestroms realisiert	
<code>close()</code>	Schließt den Ausgabestrom	*
<code>flush()</code>	Leert den Ausgabestrom	*
<code>write(char[] cbuf, int off, int len)</code>	Schreibt einen Teil eines Arrays von Zeichen	*
<code>write(char[] cbuf)</code>	Schreibt ein Array von Zeichen in den Ausgabestrom	
<code>write(int c)</code>	Schreibt ein einzelnes Zeichen (2 Byte) in den Ausgabestrom, wobei als Zeichen die zwei niederwertigen Bytes des <code>int</code> -Parameters interpretiert werden	
<code>write(String str)</code>	Schreibt eine Zeichenkette in den Ausgabestrom	
<code>write(String str, int off, int len)</code>	Schreibt beginnend beim Offset <code>off</code> eine Zeichenkette der Länge <code>len</code> in den Ausgabestrom	
<code>append(char c)</code>	Hängt ein Zeichen an; Rückgabewert: <code>Writer</code>	
<code>append(CharSequence csq)</code>	Hängt eine Zeichenfolge an; Rückgabewert: <code>Writer</code> . <code>CharSequence</code> beschreibt eine nur lesbare Folge von Zeichen.	
<code>append(CharSequence csq, int start, int end)</code>	Hängt einen Teil einer Zeichenfolge an, beginnend mit dem Zeichen an der Stelle des Index <code>start</code> und endend ein Zeichen vor dem Index <code>end</code> ; Rückgabewert: <code>Writer</code>	

Die mit * gekennzeichneten Methoden sind abstrakt und werden von den Klassen implementiert, die von der Klasse `Writer` abgeleitet wurden.

Abgeleitete Klassen der Klasse `Writer`

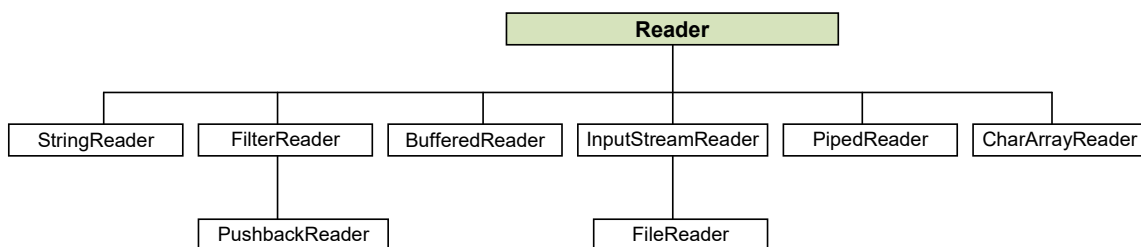
Die abgeleiteten Klassen der Klasse `Writer` stellen eine Verbindung zu den verschiedenen Ausgabegeräten her oder können als Filter eingesetzt werden:

<code>BufferedWriter</code>	Schreibt Zeichen, Strings oder Arrays in einen gepufferten Character-Stream und ermöglicht so ein effektives Schreiben
<code>CharArrayWriter</code>	Die Zeichen werden in ein Character-Array geschrieben. Dieser Puffer (das Array) wächst automatisch während des Schreibens.
<code>FilterWriter</code>	Schreibt gefilterte Character-Streams (abstrakt)
<code>OutputStreamWriter</code>	Schreibt die Zeichen in einen Ausgabestrom und wandelt dabei den Character-Stream in einen Byte-Stream um
<code>FileWriter</code>	Eine von <code>OutputStreamWriter</code> abgeleitete Klasse, die die Ausgabe in eine Datei realisiert
<code>PipedWriter</code>	Dient zur Ausgabe in eine Pipe. Pipes werden zur Realisierung der Ein-/Ausgabeströme von Threads (gleichzeitig ausgeführte Programmmodule) verwendet. Die Arbeit mit Threads ist nicht Gegenstand dieses Buches.
<code>PrintWriter</code>	Gibt alle primitiven Datentypen und Strings im Textformat aus
<code>StringWriter</code>	Schreibt Zeichen in einen String-Puffer

Character-Streams lesen

Hierarchie der Klassen für die Eingabe

Von der Superklasse `Reader` werden alle Klassen zur Eingabe von Character-Streams abgeleitet, z. B. die Klasse `FileReader` zum Lesen aus einer Datei.



Methoden der abstrakten Superklasse `Reader`

In der Klasse `Reader` werden folgende Methoden zur Verfügung gestellt.

<code>Reader()</code>	Konstruktor, der das Öffnen des Eingabestroms realisiert	
<code>close()</code>	Schließt den Eingabestrom	*

<code>mark()</code>	Markiert die aktuelle Position im Eingabestrom	
<code>boolean markSupported()</code>	Überprüft, ob der Eingabestrom Markierungen unterstützt	
<code>reset()</code>	Zurücksetzen des Eingabestroms	
<code>long skip(long n)</code>	Überspringt die nächsten <code>n</code> Zeichen, wobei <code>n</code> größer als 0 sein muss	
<code>int read()</code>	Liest ein einzelnes Zeichen aus dem Eingabestrom und gibt dieses als <code>int</code> -Wert zurück. Gibt die <code>read</code> -Methode den Wert <code>-1</code> zurück, ist das Dateiende erreicht. Für Leseoperationen in der Schleife können Sie diese Eigenschaft als Abbruchkriterium nutzen.	
<code>int read(char[] cbuf)</code>	Liest ein oder mehrere Zeichen aus dem Eingabestrom in das als Parameter übergebene <code>char</code> -Array. Der zurückgegebene <code>int</code> -Wert entspricht der Anzahl der übertragenen Zeichen.	
<code>int read(char[] cbuf, int off, int len)</code>	Liest Zeichen aus dem Eingabestrom in einen Teil des als Parameter übergebenen <code>char</code> -Arrays	*

Die mit * gekennzeichneten Methoden sind abstrakt und werden von den abgeleiteten Klassen implementiert.

Bevor die Methode `mark` ausgeführt wird, sollte mithilfe der Methode `markSupported` geprüft werden, ob der Eingabestrom das Markieren unterstützt. Ist ein Eingabestrom markiert, kann mittels der Methode `reset` der Eingabestrom auf die Position zurückgesetzt werden, die mit `mark` gekennzeichnet wurde. Ist keine Position markiert, wird versucht, den Eingabestrom auf die Anfangsposition zurückzusetzen. Werden die Methoden `mark` bzw. `reset` aufgerufen, ohne dass dies von der verwendeten Klasse unterstützt wird, kommt es zu einer `IOException`.

Abgeleitete Klassen der Klasse `Reader`

Von der Klasse `Reader` sind weitere Klassen abgeleitet, die eine Verbindung zur Datenquelle herstellen bzw. die gelesenen Daten filtern:

<code>BufferedReader</code>	Liest die Zeichen, Strings oder Arrays aus einem Character-Stream, puffert diese und ermöglicht so ein effektives Lesen
<code>CharArrayReader</code>	Die Zeichen werden in ein Character-Array gelesen.
<code>FilterReader</code>	Filtert Character-Streams während des Lesens (abstrakt)
<code>PushbackReader</code>	Von der Klasse <code>FilterReader</code> abgeleitete Klasse, die die erlaubten Zeichen liest und sie wieder in den Eingabestrom zurückschiebt
<code>InputStreamReader</code>	Der Eingabestrom wird als Byte-Stream gelesen und in einen Character-Stream umgewandelt.

<code>FileReader</code>	Von der Klasse <code>InputStreamReader</code> abgeleitete Klasse, die das Lesen aus einer Datei realisiert
<code>PipedReader</code>	Dient zum Lesen aus einer Pipe. Pipes werden zur Realisierung der Ein-/Ausgabeströme von Threads (gleichzeitig ausgeführte Programmmodule) verwendet. Die Arbeit mit Threads ist nicht Gegenstand dieses Buches.
<code>StringReader</code>	Liest die Zeichen aus einem String-Puffer

Mit Character-Streams und Textdateien arbeiten

Die Klassen `OutputStreamWriter` und `FileWriter`

- ✓ `OutputStreamWriter` ist eine abstrakte Klasse und kann als Superklasse für die Klassen verwendet werden, für die eine Umwandlung von Character-Streams in Byte-Streams durchgeführt werden soll.
- ✓ Die Klasse `FileWriter` ist von der Klasse `OutputStreamWriter` abgeleitet und wird für die Ausgabe in Dateien eingesetzt.

Konstruktoren der Klasse `FileWriter`

Bevor Sie mit einer Datei arbeiten können, muss diese geöffnet werden. Diese Aufgabe übernimmt der Konstruktor. Der Aufruf des Konstruktors erzeugt eine Datei mit dem als Parameter übergebenen Namen ①, sofern diese noch nicht existiert. Gibt es bereits eine Datei mit diesem Namen, wird sie überschrieben. Wollen Sie Daten an eine vorhandene Datei anhängen, nutzen Sie den Konstruktor ② und geben Sie als zweites Argument `true` an. Dann bleiben die vorhandenen Daten erhalten und neue Daten können hinzugefügt werden.

Auch die Übergabe eines Objekts der Klasse `File` ③ als Argument ist zulässig.

```
FileWriter(String fileName) ①
FileWriter(String fileName, boolean append) ②
FileWriter(File file) ③
```

Ein `FileWriter`-Objekt erzeugen

Mit `new` können Sie ein `FileWriter`-Objekt erstellen.

```
FileWriter fw = new FileWriter("testCharFile.dat");
```

- ✓ Als Argument wird dem Konstruktor ein String mit dem Namen der Datei übergeben.
- ✓ Geben Sie im Dateinamen kein Verzeichnis an, wird das aktuelle Benutzerverzeichnis verwendet. Das Benutzerverzeichnis können Sie bei Bedarf über die Methode `System.getProperty("user.dir")` ermitteln, die dieses Verzeichnis als String liefert.

Beispiel: `com\herdt\java9\kap17\FileWriterTest.java`

Im Beispiel legt ein `FileWriter`-Objekt eine Datei `testCharFile.dat` im aktuellen Benutzerverzeichnis an und schreibt einen String in die Datei. Danach wird die Datei wieder geschlossen.

```
① try
{
②   FileWriter fw = new FileWriter("testCharFile.dat");
③   fw.write
      ("1. Zeile: Test Ausgabe \r\n2. Zeile: in eine Datei");

④   fw.close();
}
① catch (IOException ioex)
{
⑤   System.out.println(ioex.getMessage());
}
```

- ① Ein- und Ausgabe-Operationen müssen immer in einem `try-catch`-Block stehen, um dabei auftretende `IOExceptions` abfangen zu können und einen Programmabsturz zu verhindern.
- ② Durch den Aufruf des Konstruktors wird die Datei `testCharFile.dat` im aktuellen Benutzerverzeichnis geöffnet. Ist die Datei noch nicht vorhanden, wird sie neu angelegt. Anderenfalls wird die bestehende Datei überschrieben.
- ③ Mit der Methode `write` wird ein String in die Datei geschrieben.
- ④ Die Datei wird mit der `close`-Methode geschlossen.
- ⑤ Tritt beim Öffnen oder Schreiben in die Datei eine Exception auf, wird mithilfe der Methode `getMessage` die entsprechende Fehlerbeschreibung ermittelt und angezeigt.

Die Klassen `InputStreamReader` und `FileReader`

Die Klasse `InputStreamReader` ist eine abstrakte Superklasse für Eingabeströme, für die eine Umwandlung von Byte- in Character-Streams durchgeführt werden soll. Die daraus abgeleitete Klasse `FileReader` ermöglicht das Lesen aus einer Datei.

Konstruktoren der Klasse `FileReader`

Dem Konstruktor kann der Dateiname als String ① oder als `File`-Objekt ② übergeben werden. Ist die Datei nicht vorhanden, wird eine Exception vom Typ `FileNotFoundException` ausgelöst.

```
FileReader(String fileName) ①
FileReader(File file) ②
```

Beispiel: `com\herdt\java9\kap17\FileReaderTest.java`

In diesem Beispielprogramm werden aus der geöffneten Datei `testCharFile.dat` die einzelnen Zeichen sequenziell gelesen und an einen String angehängt. Anschließend wird der String ausgegeben.

```
String text = "";
int x = 0;
① try
{
②   FileReader fr = new FileReader("testCharFile.dat");
③   while ((x = fr.read()) != -1)
④     text += (char)x;
⑤   fr.close();
}
① catch (IOException io)
{
⑥   System.out.println(io.getMessage());
}
System.out.println(text);
```

- ① Die Anweisungen zum Öffnen und Lesen der Datei werden in einen `try-catch`-Block eingeschlossen. Wird beim Öffnen der Datei beispielsweise die Datei nicht gefunden, tritt eine `FileNotFoundException` auf. Die Klasse `IOException` ist die Superklasse für alle Ausnahmen, die während der Ein- und Ausgabe auftreten können. Sie fängt somit auch die `FileNotFoundException` ab.
- ② Die Datei `testCharFile.dat` wird geöffnet.
- ③ In der Schleife wird zeichenweise aus der Datei gelesen.
- ④ Das gelesene Zeichen wird an den String `text` angehängt. Da die Methode `read` einen Rückgabewert vom Typ `int` liefert, muss vor dem Anfügen an den String eine Typumwandlung erfolgen.
- ⑤ Die Datei wird geschlossen.
- ⑥ Tritt eine `IOException` ein, wird der Text der entsprechenden Fehlermeldung ausgegeben.

Brückenklassen zur Konvertierung von Character- in Byte-Streams

Die Klassen `InputStreamReader` und `OutputStreamWriter` stellen sogenannte **Brückenklassen** zwischen Character-Streams und Byte-Streams dar.

- ✓ Mit ihrer Hilfe erfolgt beim Lesen eine Umwandlung von Byte-Streams in Character-Streams.
- ✓ Beim Schreiben werden Character-Streams in Byte-Streams umgewandelt.

Durch die Umwandlung von Unicode-Zeichen in Bytes wird der Platzbedarf beim Speichern in der Datei reduziert.

Character-Streams puffern

Ein- und Ausgabeströme puffern

Ein- und Ausgabeströme werden meist gepuffert, um die Ein- und Ausgabe effektiver zu gestalten. Bei einem Eingabestrom werden beispielsweise Daten in einen Puffer (Zwischenspeicher) gelesen und dort gesammelt. Von dort werden sie an ihr Ziel weitergeleitet, wenn der Puffer voll ist oder wenn eine Methode zum Leeren des Puffers aufgerufen wird. So werden größere Mengen an Daten transportiert, was wesentlich effektiver ist, da die Zugriffe auf Ein- und Ausgabegeräte zeitaufwendig sind.

Ausgabeströme puffern

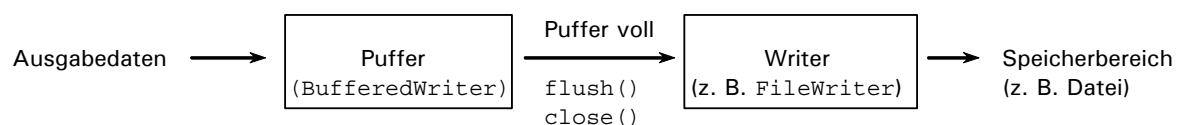
Die Klasse `BufferedWriter` dient der Pufferung von Ausgabeströmen. Alle über die Methode `write` erzeugten Ausgaben werden so lange in einem Puffer gespeichert, bis dieser voll ist oder die Methode `flush` aufgerufen wird. Die Methode `flush` bewirkt, dass alle im Puffer befindlichen Daten zum Ausgabegerät transportiert werden. Dabei wird der Ausgabepuffer geleert. Mit der Methode `close` wird ebenfalls der Ausgabepuffer geleert und zusätzlich der Ausgabestrom geschlossen.



Der Character-Stream muss nach Beendigung der Ausgaben mit der Methode `close` geschlossen werden, da sonst möglicherweise im Puffer befindliche Daten nicht ausgegeben werden und Ressourcen des Betriebssystems belegt bleiben.

Character-Streams zur Pufferung von Ausgabeströmen schachteln

Für die Pufferung von Ausgabeströmen ist es erforderlich, Character-Streams zu schachteln. Dem Konstruktor der Klasse `BufferedWriter` muss ein weiteres `Writer`-Objekt übergeben werden, z. B. vom Typ `FileWriter`. Wird nun die `write`-Methode für den `BufferedWriter` aufgerufen, werden die Daten nicht gleich an den `FileWriter` weitergeleitet, sondern erst dann, wenn der Puffer geleert wird.



Die Klasse `BufferedWriter` besitzt folgende Konstruktoren:

Über den zweiten Parameter `sz` des Konstruktors ① kann die Größe des Puffers angegeben werden.

```

BufferedWriter(Writer out)
BufferedWriter(Writer out, int sz) ①
  
```

Beispiel: `com\herdt\java9\kap17\BufferedWriterTest.java`

Ein Zeichen-Array `ca` mit 5 Elementen soll 100-mal in die Datei `testCharBuffer.dat` ausgegeben werden. Die Ausgabe erfolgt gepuffert.

```

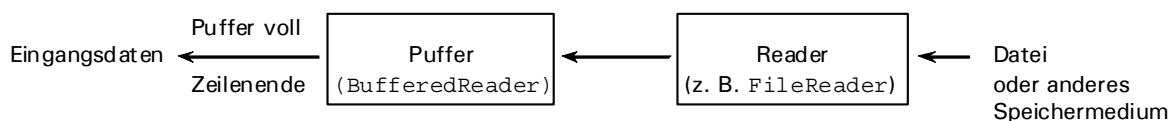
① char[] ca = {'k', 'n', 'r', '0', '1'};
② try
  {
③   BufferedWriter bw =
       new BufferedWriter(new FileWriter("testCharBuffer.dat"));
④   for (int i = 0; i < 100; i++)
       {
⑤     bw.write(ca);
⑥     bw.newLine();
       }
⑦   bw.close();
  }
  catch (IOException io) {...

```

- ① Das Feld `ca` vom Typ `char` wird deklariert und initialisiert.
- ② Ein `try-catch`-Block schließt die Verwendung der Stream-Klassen ein.
- ③ Die Datei `testCharBuffer.dat` wird geöffnet und mit dem `FileWriter`-Objekt dem Konstruktor der Klasse `BufferedWriter` als Parameter übergeben. Für die Operationen wird nun nicht mehr das `FileWriter`-Objekt verwendet, sondern das `BufferedWriter`-Objekt.
- ④ Die Schleife wird 100-mal durchlaufen.
- ⑤ Das Character-Array wird ausgegeben.
- ⑥ Das Zeilenende-Kennzeichen wird ausgegeben.
- ⑦ Bis zu dieser Anweisung befinden sich noch alle Daten im Ausgabepuffer. Erst mit dieser Anweisung werden die Daten in die Datei geschrieben, und der Stream und die Datei werden geschlossen.

Eingabeströme puffern

Das Prinzip ist das gleiche wie bei Ausgabeströmen, nur in entgegengesetzter Richtung.



Da mehrere Bytes auf einmal gelesen werden, wird die Anzahl der Zugriffe auf den Datenträger geringer und die Lesegeschwindigkeit erhöht sich. Die Klasse `BufferedReader` besitzt zwei Konstruktoren.

- ✓ Dem Konstruktor muss ein `Reader`, z. B. ein `FileReader`-Objekt, übergeben werden, für den der `BufferedReader` dann die Pufferung der Daten vornimmt. Die Größe des Puffers entspricht der Standardgröße.

```

BufferedReader (Reader in)
BufferedReader (Reader in, int sz)

```

- ✓ In einem zweiten Parameter kann die Größe für den Puffer festgelegt werden.

Die Klasse `BufferedReader` besitzt neben den geerbten `read`-Methoden noch die Methode `readLine`, die eine Textzeile in den Eingabestrom liest und als `String` zurückgibt. Als Zeilenende werden `\n`, `\r` und `\r\n` erkannt, aber nicht zurückgegeben. Konnte keine Zeile gelesen werden oder ist das Ende der Datei erreicht, liefert die Methode den Wert `null`.

```
String readLine()
```

Nutzen Sie die Klasse `BufferedReader` nicht nur, um die Eingabe zu puffern, sondern auch dann, wenn Sie als Ergebnis des Lesens einen `String` benötigen. Die `read`-Methoden aller anderen Reader-Klassen liefern als Ergebnis nur ein Charakter-Array bzw. ein Zeichen.

Beispiel: `com\herdt\java9\kap17\BufferedReaderTest.java`

Wie im folgenden Beispiel demonstriert wird, kann mithilfe der Methode `readLine` eine Datei in einer Schleife zeilenweise gelesen und verarbeitet werden.

```
String s;
BufferedReader br = null;
① try
{
②   br = new BufferedReader(new FileReader("testCharBuffer.dat"));
③   try
   {
④     while ((s = br.readLine()) != null)
⑤       System.out.println(s);
   }
   finally
   {
⑥     if (br != null)
       br.close();
   }
}
catch (IOException io)
{
  System.out.println(io.getMessage());
}
```

- ① Die Eingaben sind in einem `try-catch`-Block einzuschließen, um Ein- und Ausgabebefehle abzufangen.
- ② Ein Objekt der Klasse `BufferedReader` wird erzeugt. Dem Konstruktor wird als Parameter ein Objekt der Klasse `FileReader` übergeben, das ebenfalls in dieser Anweisung generiert wird. Das `FileReader`-Objekt wird benötigt, um die Verbindung zur Datei `testCharBuffer.dat` herzustellen. Für die weitere Arbeit wird das `BufferedReader`-Objekt verwendet.
- ③ Tritt beim Einlesen der Daten eine Exception auf, wird über diesen `try-finally`-Block sichergestellt, dass die Datei wieder ordnungsgemäß geschlossen wird und die Ressourcen freigegeben werden.

- ④ In einer Schleife wird die Datei zeilenweise gelesen. Eine gelesene Zeile wird jeweils in der String-Variablen `s` gespeichert. Liefert die Methode `readLine` den Wert `null`, wird die Schleife beendet, da das Dateiende erreicht wurde.
- ⑤ Die gelesene Zeichenkette wird auf dem Standardausgabegerät (z. B. Bildschirm) ausgegeben.
- ⑥ Der Eingabestrom (und damit auch die Datei `testCharBuffer.dat`) wird geschlossen, wenn er in Zeile ② erfolgreich geöffnet werden konnte (d. h. wenn er ungleich `null` ist).

Eingaben von der Konsole als String lesen

Die Klasse `BufferedReader` können Sie verwenden, um Eingaben von der Konsole als String zu erhalten.

```
BufferedReader br =  
    new BufferedReader(new InputStreamReader(System.in));  
String s = br.readLine();
```

- ✓ Erzeugen Sie ein Objekt dieser Klasse und übergeben Sie dem Konstruktor ein Objekt der Klasse `InputStreamReader`, der die Konvertierung in den Unicode durchführt.
- ✓ Dem Konstruktor der Klasse `InputStreamReader` übergeben Sie das Standard-eingabegerät `System.in`.
- ✓ Mithilfe der Methode `readLine` können Sie anschließend die Tastatureingaben als Strings empfangen. Auch Umlaute werden damit richtig übertragen.

Mit Character-Streams primitive Datentypen schreiben und lesen

Primitive Datentypen schreiben

In den bisher erläuterten `Writer`-Klassen sind nur die `write`-Methoden der Superklasse `Writer` verfügbar.

Die Klasse `PrintWriter` besitzt für die Ausgabe jedes primitiven Datentyps, für Strings und Objekte eine eigene Methode.

- ✓ Mit den Methoden `print` und `println` wird die übergebene Variable bzw. das Objekt ausgegeben.
- ✓ Die Methode `println` hängt zusätzlich noch ein Zeilenende-Kennzeichen an die Ausgabe an.

```
print(boolean b)  
print(double d)  
print(Object obj)  
print(String s)  
...  
println(boolean b)  
println(double d)  
println(Object obj)  
...
```

Die Methode `printf` ermöglicht eine formatierte Ausgabe. Dem ersten Parameter wird der auszugebende Text mit Formatierungszeichen und Platzhaltern übergeben, dem zweiten Parameter eine Argumentenliste, die der Anzahl der Platzhalter entsprechen muss.

```
printf(String format, Object...args)
```

Die Konstruktoren für die Klasse `PrintWriter` verlangen als Argumente ein `Writer`, ein `OutputStream`, ein `File` oder einen Dateinamen als `String`.

<code>PrintWriter(File file)</code>	
<code>PrintWriter(OutputStream out)</code>	*
<code>PrintWriter(String fileName)</code>	
<code>PrintWriter(Writer out)</code>	*

Die Ausgaben mit dem `PrintWriter` erfolgen gepuffert. Standardmäßig wird der Ausgabepuffer nicht geleert.

Die mit * gekennzeichneten Konstruktoren stehen jeweils auch in einer Variante mit einem zweiten Parameter `autoFlush` zur Verfügung. Der Parameter `autoFlush` gibt an, ob der Ausgabepuffer automatisch geleert werden soll, wenn die Methode `println` ausgeführt wurde (Wert `true`). Wird der Parameter nicht angegeben oder besitzt er den Wert `false`, erfolgt die Leerung des Puffers erst, wenn er voll ist oder der Datenstrom geschlossen wird.

Beispiel: `com\herdt\java9\kap17\PrintWriterTest.java`

In diesem Beispiel wird für 10 Radiuswerte der Flächeninhalt für Kreise berechnet und zeilenweise in die Datei `testCharPrint.dat` ausgegeben.

```

try
{
①  PrintWriter pw = new PrintWriter("testCharPrint.dat");
②  pw.println("Ausgabe des Flächeninhalts für Kreise mit");
    //String schreiben

    for (int r = 1; r <= 10; r++)
    {
③  pw.print("Radius r = " + r + ": "); //String schreiben
④  pw.println(Math.PI * r * r);      //double-Wert schreiben
    }
⑤  pw.close();
}
catch (IOException io)
{
    System.out.println(io.getMessage());
}

```

- ① Ein Objekt der Klasse `PrintWriter` wird erzeugt. Als Parameter wird dem Konstruktor der Name der Datei `testCharPrint.dat` übergeben.
- ② Ein String mit Zeilenendezeichen wird mittels der `println`-Methode ausgegeben.
- ③ Ein String wird mithilfe der `print`-Methode ausgegeben.

- ④ Der Flächeninhalt für den aktuellen Radius `r` wird errechnet. Durch die Anweisung `println` wird der `double`-Wert ausgegeben.
- ⑤ Der Ausgabepuffer wird geleert und der Datenstrom geschlossen.

Die Anweisungen ③ und ④ können komfortabler mit der `printf`-Methode codiert werden:

```
...
pw.printf("Radius r = %d: %g%n", r, PI * r * r);
//formatierte Ausgabe
...
```

Primitive Datentypen lesen

Eine vordefinierte Klasse zum Lesen von primitiven Datentypen gibt es nicht. Sie können dieses Problem aber wie folgt lösen:

- ✓ Für die Eingabe von Strings können Sie die bereits bekannte Klasse `BufferedReader` verwenden.
- ✓ Möchten Sie Zahlen eingeben, lesen Sie diese erst als String ein und wandeln Sie dann den String in das gewünschte Zahlenformat um.

Beispiel: `com\herdt\java9\kap17\KonsoleReaderTest.java`

Im Beispiel wird gezeigt, wie Sie eine Zahl vom Typ `int` aus der Standardeingabe lesen können. Das Einlesen einer Zahl vom Typ `double` erfolgt analog.

```
try
{
①  BufferedReader br =
    new BufferedReader(new InputStreamReader(System.in));
    System.out.println("Geben Sie eine Integerzahl ein: ");
②  String s = br.readLine();
③  int x = Integer.parseInt(s);
    System.out.println("Die eingegebene Zahl heisst: " + x);
}
catch (IOException io)
{
    System.out.println(io.getMessage());
}
④ catch (NumberFormatException ex)
{
    System.out.println("Fehlerhafte Integerzahl gelesen: " +
        ex.getMessage());
}
```

- ① Für die Eingabe wird die Standardeingabe (`System.in` – die Tastatur) verwendet, die einen Byte-Stream liefert. Die Umwandlung in 16-Bit-Unicode wird durch den `InputStreamReader` vorgenommen, dem `system.in` übergeben wird. Der `BufferedReader` ermöglicht eine gepufferte String-Eingabe. Dazu muss ihm das `InputStreamReader`-Objekt übergeben werden.

- ② Die Methode `readLine` liest den Inhalt einer Zeile als String. Das bedeutet bei der Tastatureingabe, dass alle Zeichen bis zum Betätigen der `↵`-Taste gelesen werden.
- ③ Da ein `int`-Wert eingegeben werden soll, muss der gelesene String über die Methode `parseInt` der Wrapper-Klasse `Integer` umgewandelt werden. Tritt bei der Umwandlung ein Fehler auf, war der eingegebene Wert kein `int`-Wert und das Programm verzweigt zum `catch`-Block ④.

Character-Streams filtern

Aus- und Eingabeströme filtern

Mithilfe der Klasse `FilterWriter` bzw. `FilterReader`, die als abstrakte Klassen definiert sind, können Sie eigene Filter definieren, beispielsweise um Kleinbuchstaben in Großbuchstaben umzuwandeln. Zu diesem Zweck wird jeweils eine neue Klasse definiert, die von der Klasse `FilterWriter` bzw. `FilterReader` abgeleitet wird.

- ✓ Die Klasse besitzt Konstruktoren, denen ein `Writer`- bzw. `Reader`-Objekt als Parameter übergeben werden muss.
- ✓ Der Konstruktor einer eigenen Filterklasse muss den Konstruktor der Superklasse aufrufen, um das übergebene Objekt zu initialisieren.

```
FilterWriter(Writer out)
FilterReader(Reader in)
```

Weiterhin besitzen die Klassen `write`- bzw. `read`-Methoden, die mit entsprechenden eigenen Methoden überschrieben werden können. Auch bei der Überlagerung der `write`- bzw. `read`-Methoden muss die entsprechende Methode der Superklasse aufgerufen werden. Die Parameter, die Sie der Superklassenmethode übergeben, können vorher so verändert werden, dass sich die gewünschte Filterung ergibt.

Gelesene Zeichen oder Zeichenfolgen wieder zurückschreiben

Von der Klasse `FilterReader` ist die Klasse `PushbackReader` abgeleitet. Haben Sie beispielsweise ein Zeichen oder eine Zeichenfolge gelesen, die Sie lediglich auf bestimmte Kriterien prüfen wollten, können Sie diese Zeichen mit der Klasse `PushbackReader` wieder in den Eingabestrom zurückstellen.

Dazu werden die gelesenen Zeichen in einem sogenannten **Pushback-Puffer** zwischengespeichert. Mit dem Aufruf des Konstruktors wird der entsprechende Datenstrom verbunden.

- ✓ Konstruktor ① stellt einen Puffer für ein Zeichen bereit.
- ✓ Bei Konstruktor ② wird die Größe des Puffers durch den zweiten Parameter bestimmt.

```
PushbackReader(Reader in) ①
PushbackReader(Reader in, int size) ②
```

Zusätzlich zu den von der Reader-Klasse geerbten Methoden ist die Methode `unread` implementiert, die das Zurückschreiben in den Eingabestrom realisiert. Die Zeichen, die zurückgestellt werden sollen, werden der Methode als Parameter übergeben.

```
unread(char[] cbuf)
unread(char[] cbuf, int off, int len)
unread(int c)
```

Character-Streams für Strings und Character-Arrays

In Strings und Character-Arrays schreiben

Im Unterschied zur Klasse `FileWriter` ist das Ausgabeziel der Klassen `CharArrayWriter` und `StringWriter` nicht eine Datei, sondern ein Character- bzw. ein String-Buffer. Diese Puffer wachsen automatisch, wenn Daten in den Strom geschrieben werden. Beide Klassen besitzen zwei Konstruktoren.

- ✓ Der Parameter `initialSize` kann für die Festlegung der Puffergröße genutzt werden.
- ✓ Wird keine Größe als Parameter angegeben, wird ein Standardwert verwendet.

```
CharArrayWriter()
CharArrayWriter(int initialSize)
StringWriter()
StringWriter(int initialSize)
```

Da beide Klassen direkt von der Klasse `Writer` abgeleitet sind, stehen die bekannten `write`-Methoden zur Verfügung. Zusätzlich verfügen beide Klassen über spezielle Methoden.

Methoden der Klasse `CharArrayWriter`

<code>char[] toCharArray()</code>	Kopiert die Daten des Streams in ein Character-Array
<code>String toString()</code>	Konvertiert die Daten des Streams in einen String
<code>void writeTo(Writer out)</code>	Schreibt die Daten des Streams in einen anderen Character-Stream
<code>int size()</code>	Gibt die aktuelle Puffergröße zurück
<code>void reset()</code>	Leert den Puffer

Methoden der Klasse `StringWriter`

<code>StringBuffer getBuffer()</code>	Gibt den aktuellen Inhalt des Streams als String-Buffer zurück
<code>String toString()</code>	Konvertiert den aktuellen Inhalt des Streams in einen String

Beispiel: `com\herdt\java9\kap17\StringWriterTest.java`

In dem folgenden Beispiel wird ein `StringWriter` für das Zwischenspeichern von Zeichenketten eingesetzt. Mit jedem Aufruf der `write`-Methode wird ein weiterer String angefügt, dessen Länge jeweils um 1 gegenüber seinem Vorgänger vermindert ist. Ist die Länge 0 erreicht, wird der Inhalt des `StringBuffers` in einen einzigen String umgewandelt und angezeigt.

```
String s = "Java macht Spass";
① StringWriter sw = new StringWriter();
② int l = s.length();
  for (int i = 0; i < l; i++)
  {
③   s = s.substring(0, s.length() - 1);
④   sw.write(s + "\n");
  }
⑤ System.out.println(sw.toString());
```

- ① Ein Objekt der Klasse `StringWriter` wird erzeugt.
- ② Durch die Anzahl der Zeichen der Zeichenkette `s` wird die Anzahl der Schleifendurchläufe bestimmt.
- ③ Das letzte Zeichen der Zeichenkette `s` wird abgeschnitten.
- ④ Nun werden die verkürzte Zeichenkette und ein Zeilenumbruch in den `StringWriter` geschrieben.
- ⑤ Der Inhalt des `StringWriters` wird auf der Konsole ausgegeben.

Aus Strings und Character-Arrays lesen

Für das Lesen von `Character-Arrays` und `Strings` können die Klassen `CharArrayReader` und `StringReader` angewendet werden. Von ihrer Superklasse `Reader` erben sie alle Methoden und besitzen folgende Konstruktoren:

```
CharArrayReader(char [] buf)
CharArrayReader(char [] buf, int offset, int length)
StringReader(String s)
```

Beispiel: `com\herdt\java9\kap17\StringReaderTest.java`

Das folgende Beispiel liest einen String zeichenweise und gibt jedes gelesene Zeichen in einer Zeile aus.

```
String s = "Java macht Spass";
① StringReader sr = new StringReader(s);
  int z;
  try
  {
②   while ((z = sr.read()) != -1)
```

```

③ System.out.println((char)z);
}
catch (IOException io)
{
    System.out.println(io.getMessage());
}

```

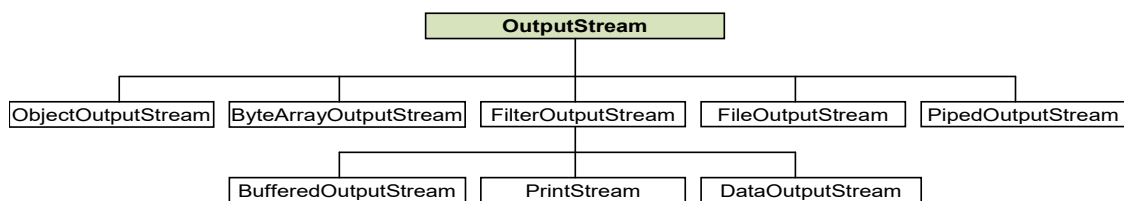
- ① Ein Objekt der Klasse `StringReader` wird definiert. Dem Konstruktor wird die Zeichenkette `s` übergeben, die als Quelle des Streams dient.
- ② Aus dem `StringReader` wird in einer Schleife jeweils ein Zeichen gelesen. Die Methode `read` liefert den ASCII-Code des gelesenen Zeichens als Integer-Zahl zurück. Wenn alle Zeichen gelesen sind, gibt die Methode `read` den Wert `-1` zurück, wodurch die Schleife beendet wird.
- ③ Das gelesene Zeichen wird ausgegeben. Da die Variable `z` eine Integer-Zahl ist, muss sie zuvor in ein Zeichen umgewandelt werden.

Mithilfe der Klassen `StringWriter` und `StringReader` lassen sich Strings auf die gleiche Weise verarbeiten wie andere Datenströme.

Mit Byte-Streams arbeiten

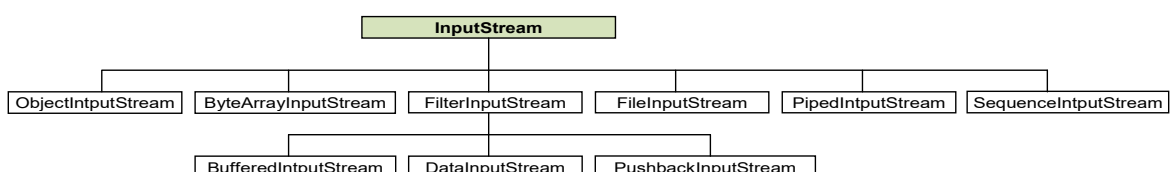
Hierarchie der Klassen für die Ausgabe

Byte-Streams transportieren jeweils 8 Bit, d. h. 1 Byte. Byte-Streams sind bereits seit der ersten Java-Version implementiert und stellen die grundlegende Ein- und Ausgabe von Daten dar. Alle Klassen zur Ausgabe von Byte-Streams werden von der Superklasse `OutputStream` abgeleitet, z. B. die Klasse `FileOutputStream` zum Schreiben in eine Datei.



Hierarchie der Klassen für die Eingabe

Von der Superklasse `InputStream` werden alle Klassen zur Eingabe von Byte-Streams abgeleitet, z. B. die Klasse `FileInputStream` zum Lesen aus einer Datei.



Klassen für Byte- und Character-Streams im Vergleich

Viele der Klassen haben ein Pendant in den Character-Streams, von denen sie sich nur durch die Verarbeitungsbreite unterscheiden. Sie verfügen meist über die gleichen Methodenaufrufe und werden analog verwendet. Die folgende Tabelle zeigt die sich entsprechenden Klassen für Byte- und Character-Streams.

Ausgabe-Streams	
Byte-Streams	Character-Streams
OutputStream	Writer
FileOutputStream	FileWriter
BufferedOutputStream	BufferedWriter
ByteArrayOutputStream	CharArrayWriter
FilterOutputStream	FilterWriter
PipedOutputStream	PipedWriter
PrintStream	PrintWriter

Eingabe-Streams	
Byte-Streams	Character-Streams
InputStream	Reader
FileInputStream	FileReader
BufferedInputStream	BufferedReader
ByteArrayInputStream	CharArrayReader
FilterInputStream	FilterReader
PipedInputStream	PipedReader
PushbackInputStream	PushbackReader

Einige Klassen, wie `StringReader` und `StringWriter`, haben keine vergleichbaren Klassen in den Byte-Streams implementiert, ebenso wie die Klassen `InputStreamReader` und `OutputStreamWriter`, die als Schnittstelle zwischen beiden Stream-Typen fungieren. Die Klassen `ObjectInputStream` und `ObjectOutputStream` haben kein Pendant in den Character-Streams, da diese mit Binärdaten arbeiten.

Standardeingabe und -ausgabe mit Byte-Streams

Als Vertreter für Byte-Stream-Klassen werden hier nur die Klassen `InputStream` und `PrintStream`, die für die Standardeingabe und -ausgabe verwendet werden, betrachtet.

Die Ein- und Ausgabe über die Standard-E/A-Geräte (meist Tastatur und Bildschirm) ist in der Klasse `System` implementiert. Da diese Klasse im Package `java.lang` definiert ist, muss bei Verwendung der Standard-E/A kein zusätzliches Package eingebunden werden. Die Standard-E/A arbeitet mit Objekten der Stream-Klassen `InputStream` und `PrintStream`.

Typ des Byte-Streams	Definition der Klassenkonstanten	Aufruf
Standard-Eingabestrom	<code>public static final InputStream in</code>	<code>System.in</code>
Standard-Ausgabestrom	<code>public static final PrintStream out</code>	<code>System.out</code>
Standard-Fehlerausgabestrom	<code>public static final PrintStream err</code>	<code>System.err</code>

In den Klassen `InputStream` und `PrintStream` sind die `read`- und `print`-Methoden implementiert, die Sie für die Ein- und Ausgabe der verschiedenen Datentypen auf Standardgeräte nutzen können.

Beispiel: `com\herdt\java9\kap17\StdIO1.java`

Die `print`-Methode gibt auf dem Bildschirm eine Zeichenkette aus, führt aber keinen Zeilenvorschub aus. Die Methode `read` liest die Tastatureingaben in ein `byte`-Array, wenn die Eingabe mit der `↵`-Taste abgeschlossen wurde. Mit der Methode `println` wird zusätzlich zur Ausgabe ein Zeilenvorschub durchgeführt.

```
byte[] b = new byte[1];
try
{
    System.out.print("Bitte geben Sie ein Zeichen ein: ");
    System.in.read(b);
    System.out.println((char)b[0] + " hat den ASCII-Code " + b[0]);
}
catch (IOException io)
{
    System.out.println(io.getMessage());
}
```

Falls Sie mehrere Ein- und Ausgabe-Anweisungen codieren, können Sie die Klasse `System` statisch importieren und somit von der verkürzten Schreibweise bei den Ein- und Ausgabe-Anweisungen profitieren.

```
import static java.lang.System.*;
...
out.print("Bitte geben Sie ein Zeichen ein: ");
in.read(b);
out.println((char)b[0] + " hat den ASCII-Code " + b[0]);
...
```

Byte-Streams mit den Methoden `print` und `println` ausgeben

Die Methoden `print` und `println` sind mit verschiedenen Parameterlisten implementiert und können für jeden primitiven Datentyp und für die Klassen `String` und `Object` aufgerufen werden, z. B.:

```

print(char b)           println(char b)
print(double d)        println(double d)
print(int i)           println(int i)
print(Object obj)      println(Object obj)
print(String s)        println(String s)
...                   ...

```

Byte-Streams mit der Methode `read` lesen


Die Methode `read` kann ohne Parameter oder mit einem `byte`-Array aufgerufen werden.

```

int read()
int read(byte[] b)
int read(byte[] b, int off, int len)

```

- ✓ Ohne Parameter wird das nächste Zeichen aus dem Eingabestrom gelesen.
- ✓ Wird `read` mit dem `byte`-Array aufgerufen, werden maximal so viele Zeichen gelesen, wie das Array aufnehmen kann. Der Rest verbleibt im Eingabestrom (Eingabepuffer).

Die eingegebenen Zeichen werden in jedem Fall erst beim Betätigen der -Taste eingelesen.

Mitunter verbleiben im Eingabepuffer noch Zeichen, die nicht mehr benötigt werden. Um alle noch im Eingabepuffer befindlichen Zeichen zu überspringen, können Sie den nebenstehenden Methodenaufruf verwenden.

```
System.in.skip(System.in.available());
```

Die Methode `available` liefert dabei die Anzahl der noch im Eingabepuffer befindlichen Zeichen.



Durch den Parametertyp `byte` ist es nicht möglich, Umlaute einzulesen und richtig zu interpretieren. Umlaute lassen sich nicht in einer Variablen vom Typ `byte` darstellen.

Beispiel: `com\herdt\java9\kap17\StdIO2.java`

In diesem Programmbeispiel ist analog zum vorherigen Beispiel eine Ausgabe mit einer Eingabeaufforderung, eine Eingabe und die Ausgabe des ersten eingegebenen Zeichens als Zeichen und als ASCII-Code realisiert. Diese Anweisungsfolge wird nach dem Löschen des Eingabepuffers noch einmal wiederholt, um die Wirkung der Methode `skip` zu zeigen.

```

...
import static java.lang.System.*;
...
byte[] b = new byte[1];
try
{
    out.print("Bitte geben Sie ein Zeichen ein: ");
    in.read(b);

```

```


out.println((char)b[0] + " hat den ASCII-Code " + b[0]);
out.print("Bitte geben Sie noch ein Zeichen ein: ");
in.skip(in.available());
in.read(b);
out.println((char)b[0] + " hat den ASCII-Code " + b[0]);
}
catch (IOException ioex)
...

```

- ▶ Starten Sie die Anwendung.
- ▶ Geben Sie nach der Eingabeaufforderung mehrere Zeichen ein.
- ▶ Testen Sie das Programm auch mit Umlauten. Dadurch, dass sich Umlaute nicht in einem Byte darstellen lassen, erhalten Sie negative Werte für den ASCII-Code eines solchen Zeichens.
- ▶ Kommentieren Sie die Zeile `in.skip...` aus und geben Sie bei der ersten Eingabeaufforderung mehrere Zeichen ein.

Übung

Arbeit mit Streams

Level		Zeit	ca. 80 min
Übungsinhalte	<ul style="list-style-type: none"> ✓ Anwendung von <code>BufferedReader</code> und <code>BufferedWriter</code> ✓ Filterung von Streams ✓ Verwendung von <code>PrintWriter</code> 		
Übungsdateien	<i>valuesIn.dat, PrintWriterTest.java</i>		
Ergebnisdatei	<i>Measurements.java, valuesOut.dat, ReadWriteData.java, personal.dat, MyFilterWriter.java, FilterCharacters.java, filter.dat, PrintWriterTest2.java</i>		

1. Lesen Sie in einem Programm *Measurements.java* 20 Messreihen zu je 10 Messwerten aus der Datei *valuesIn.dat*. Geben Sie die einzelnen Messwerte durch Leerzeichen getrennt am Bildschirm aus. Schließen Sie jede Messreihe mit einem Hinweis *Ende der Messreihe xx* ab und geben Sie die nächste Messreihe in einer neuen Zeile aus. Schreiben Sie alle Daten in die Datei *valuesOut.dat*.
Die Ein- und Ausgabe soll gepuffert erfolgen. Die Dateien *valuesIn.dat* und *valuesOut.dat* sollen sich im Ordner `com\herdt\java9\kap17` befinden bzw. dort gespeichert werden.
Um bei der Bildschirmausgabe alle Zahlen rechtsbündig untereinander zu schreiben, können Sie in der `printf`-Methode die Codierung wie im folgenden Beispiel verwenden: `%4d` bewirkt, dass der angegebene Parameter auf 4 Zeichen Breite rechtsbündig geschrieben wird.
2. Erstellen Sie eine Anwendung *ReadWriteData.java* zur Erfassung von Personendaten. Dazu sind über die Tastatur folgende Daten abzufragen: Name, Geschlecht, Größe (in cm) und Gewicht (in kg). Verwenden Sie dazu die Klasse `BufferedReader` und nutzen Sie nicht die Klasse `StdInput`!

Schreiben Sie die eingegebenen Werte in eine Datei *personal.dat* im Ordner *com\herdt\java9\kap17*. Verwenden Sie dazu die überladene Methode `println` der Klasse `PrintWriter`.

Das Programm soll so gestaltet werden, dass Daten von mehreren Personen erfasst werden können.

- Erstellen Sie eine Filterklasse `MyFilterWriter`, welche die Ausgaben in eine Datei so filtert, dass nur Zahlen und Groß- und Kleinbuchstaben in die Datei geschrieben werden. Alle anderen Zeichen sind durch das Zeichen `*` zu ersetzen. Gehen Sie dabei wie folgt vor:

- ✓ Leiten Sie Ihre Filterklasse von der Klasse `FilterWriter` ab.
- ✓ Rufen Sie im Konstruktor Ihrer Filterklasse den Konstruktor der Superklasse mit dem übergebenen `Writer`-Objekt auf.
- ✓ Überschreiben Sie die drei `write`-Methoden der Klasse `FilterWriter`.

Testen Sie Ihre Filterklasse in einem Programm *FilterCharacters.java*. Dazu soll über die Tastatur eine Zeichenkette eingegeben werden, die dann gefiltert in die Datei *filter.dat* im Ordner *com\herdt\java8\kap17* geschrieben wird.

Hinweise:

- ✓ Die Filterung braucht nur in der Methode `write(int c)` durchgeführt zu werden. Das zu schreibende Zeichen ist durch Aufruf der `write`-Methode der Superklasse auszugeben. In den anderen beiden `write`-Methoden können Sie dann diese `write`-Methode aufrufen.
 - ✓ Die ASCII-Codes für die herauszufilternden Zeichen liegen in folgenden Bereichen:

kleiner 48	und
zwischen 57 und 65	und
zwischen 90 und 97	und
größer 122	
 - ✓ Das Zeichen `*` hat den ASCII-Code 42.
- Erstellen Sie auf Basis des Programms *PrintWriterTest.java* das Programm *PrintWriterTest2.java* und verwenden Sie zur Erstellung des `PrintWriter`-Objekts den Konstruktor mit den Parametern `out` und `autoflush`. Testen Sie den Parameter `autoflush` für die Werte `true` und `false`. Um die Auswirkung des Wertes `false` auf dem Standardausgabegerät zu testen, können Sie beispielsweise eine einfache Zählschleife einfügen.