4 Input & Output

4.1 Einführung

Die meisten Applikationen können ihren vollen Nutzen erst dann entfalten, wenn sie in der Lage sind, Daten persistent zu speichern und sie zu einem späteren Zeitpunkt wieder abzurufen. Unsere bisherigen Programme haben ausschliesslich auf dem Hauptspeicher (**RAM** – *random access memory*) gearbeitet. In diesem Kapitel geht es darum, Programme zu befähigen, Daten extern zu speichern. Als Medien kommen beispielsweise in Frage: Festplatten, SSDs, Speicherkarten, Clouds, Datenbanken. Im weiteren soll auch ganz allgemein der Zugriff auf Daten im Web angesprochen werden.

Die Arbeit mit externen Medien gliedert sich in zwei verschiedene Aspekte:

a) Der Umgang mit der Infrastruktur im Allgemeinen. Darunter fallen etwa folgende Themen:
   - Navigation in Dateisystemen (Pfade, Verzeichnisse)
   - Verwalten von Dateien als geschlossene Einheiten (Dateien erstellen, löschen, unbenennen, verschieben)

b) Datentransfer zwischen einer Datei (oder dem Web) und dem Programm:
   - Textbasierte Dateien
   - Dateien mit Binärdaten
   - Simulation von Dateien im **RAM**

Der Datentransfer (Aspekt b) wirft einige Fragen auf, die gesonderter Beachtung bedürfen:

- *Binärdaten*: wie werden beispielsweise lange Zahlen gespeichert, die mehr als ein Byte umfassen? Erscheint beim Lesen zuerst das höchstwertige Byte oder aber das niederwertigste? Gibt es irgendwelche Standards?
- *Binärdaten*: Kann man immer nur sequentiell auf Daten zugreifen, oder gibt es auch die Möglichkeit des wahlfreien Zugriffes (*sog. random access*)?
- *Binärdaten*: Gibt es auch die Möglichkeit, ganze Objekte zu speichern (und nicht nur primitive Variablen)?
- *Textdaten*: wie wird das Zeilenende codiert? (in der Tat ist dies plattformabhängig)
- *Textdaten*: wie werden Zeichen gespeichert, die ausserhalb des Bereiches des ASCII-Zeichensatzes liegen? Wie sieht es mit Unicode aus?
- *Generell:* Lese- und Schreiboperationen auf Dateien können Ausnahmen (Exceptions) werfen. Wie kann man sicherstellen, dass Dateien in einem solchen Fall ordentlich geschlossen werden (*siehe Seite 4.7*)?

Auch der Aspekt a) steuert zumindest eine Frage zu unserem Problemkatalog bei:

- Wie werden Pfade und Dateinamen auf verschiedenen Plattformen strukturiert? Gibt es Unterschiede, beispielsweise in der Menge, der für Dateinamen erlaubten Zeichen?

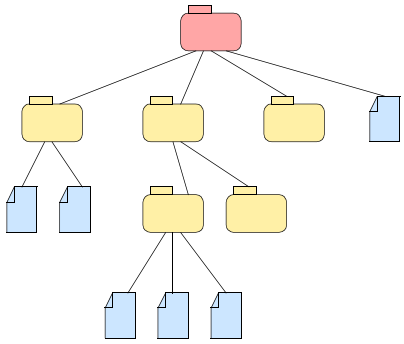
Wer ist wo?

Für dateibezogene Aufgaben sind primär die Pakete *java.nio* und *java.io* zuständig. Vergessen Sie nicht, passende Importe zu formulieren, wenn Sie nicht alle Klassennamen qualifizieren möchten.

Beachten Sie weiterhin, dass die meisten dateibezogenen Operationen *geprüfte* Ausnahmen werfen können (zur Erinnerung: das sind solche, die Sie fangen oder delegieren *müssen*).

4.2 Dateien verwalten

4.2.1 Pfade



Dateisysteme sind hierarchisch organisiert, d.h. sie bestehen aus baumartig angeordneten Elementen (Knoten):

- *Verzeichnisse* (Ordner) können weitere Elemente enthalten.
- *Dateien* sind Endknoten und enthalten keine weiteren Elemente.
- Am oberen Ende jedes Dateisystems liegt das *Wurzel-Element*.
- Jedes Element wird durch einen eindeutigen *Pfad* identifiziert.

Merke:

Ein Baum ist eine *rekursive* Struktur (d.h. ein Baum besteht aus einer Wurzel, unter welcher (Teil-)Bäume hängen. Aus diesem Grund eignen sich rekursive Algorithmen besonders gut, um Bäume zu durchlaufen.

Die Syntax von Pfaden hängt von der verwendeten Plattform ab:

Beispiel für Windows:

```
E:\ FHNW \ Module \ OOP2i \ Skripten \ Exceptions \ Kap-1_Exceptions.fm
```

Beispiel für Linux, MacOS und weitere Unix-basierte Systeme:

```
/ home / veitschegger / FHNW / Module / OOP2i / Skripten / Exceptions / Kap-1_Exceptions.fm
```

In UNIX-basierten gibt es genau eine Wurzel (`/`), während unter Windows jedes Laufwerk eine eigene Wurzel bildet. Das Dateisystem von UNIX besteht also aus einem einzigen Baum, dasjenige von Windows aus einer Menge von unabhängigen Bäumen.

Pfade können *absolut* oder *relativ* formuliert werden:

- Ein **absoluter Pfad** beginnt immer mit einem Wurzel-Element.
- Ein **relativer Pfad** beginnt mit einem Datei- oder Ordner-Namen und muss *relativ zu einem gegebenen Bezugspunkt* interpretiert werden.

  Beispiel: `Prüfungen\2024-FS\Korrektur.docx` Worauf bezieht sich dieser Pfad?

  Die Antwort kann erst gegeben werden, wenn man den Bezugspunkt kennt. Wäre der Bezugspunkt etwa als `E\:FHNW\OOP2i` bekannt, dann könnte man den absoluten Pfad angeben:
  `E\:FHNW\OOP2i\Prüfungen\2024-FS\Korrektur.docx`

Merke:

Üblicherweise verwenden Java-Programme das sogenannte Arbeitsverzeichnis (*current working directory*) als Bezugspunkt. Alle relativen Pfade werden relativ zu diesem Verzeichnis interpretiert.

Tipp (wie findet man in einem Java-Programm das Arbeitsverzeichnis?):

```
Path cwd = Path.of("").toAbsolutePath();            // als Path-Objekt...
System.out.println(cwd.normalize().toString());     // ...dann als String
```

In der ersten Zeile wird der leerer relativer Pfad "" in einen absoluten Pfad umgewandelt, woraus logischerweise der Pfad zum Arbeitsverzeichnis entsteht. In der zweiten Zeile wird mit `normalize()` sichergestellt, dass der Pfad aus Zeile 1 garantiert keine redundanten Elemente mehr enthält.

Tipp im Tipp: Sehen Sie sich die `Path`-Methode `resolve()` an. Man kann sie verwenden, um Pfade zusammenzusetzen.

Path-Objekte

Objekte der Klasse `java.nio.file.Path` repräsentieren Pfade und Dateinamen. Es handelt sich dabei im Prinzip um Strings, welche mit einigen zusätzlichen Eigenschaften angereichert wurden.

Einige Beispiele:

```
Path anchor = Path.of("E:", "FHNW", "OOP2i");                    // ein absoluter Pfad
Path further = Path.of("Skripten", "Exceptions", "Kap-1.fm");    // ein relativer Pfad

System.out.println(anchor.getParent());                 // E:\FHNW
System.out.println(anchor.resolve("Data"));             // E:\FHNW\OOP2i\Data
System.out.println(further.getFileName());              // Kap-1.fm
```

Merke:

In Windows-Filesystemen wird das Zeichen «\» verwendet, um Elemente eines Pfads voneinander zu trennen. Es muss in einem String-Literal als «\\» eingegeben werden.

4.2.2 Administrative Arbeiten rund um Dateien

Administrative Funktionen operieren auf geschlossenen Dateien. Es geht hierbei nicht darum, Daten zwischen Datei und Applikation hin oder her zu schieben, sondern darum, generelle Infrastruktur-Arbeiten zu erledigen. Darunter fallen etwa folgende Aufgaben:

- Kopieren, Verschieben und Umbenennen von Dateien
- Dateien erzeugen oder Löschen
- Überprüfen von Datei-Attributen (Existiert die Datei? Habe ich Zugriffsrechte? Liegt ein Ordner vor?)
- Navigieren im Dateisystem
- Verzeichnisse anlegen oder löschen

Die meisten Arbeiten dieser Art können Sie mit `java.nio.file.Files` erledigen. Bei dieser Klasse handelt es sich um eine rein statische Funktionssammlung. Die folgenden Tabellen bieten eine knappe Übersicht zu den obengenannten Themen. Für Details konsultieren Sie die Javadoc:

Methode	Beschreibung	throws
`copy()`	Datei kopieren. Als Quelle oder Ziel kann auch ein File-Stream verwendet werden.	`IOException`
`createFile()`	Neue leere Datei anlegen	`IOException`
`createTempFile()`	Temporärdatei anlegen (Namensvergabe automatisch oder halbautomatisch)	`IOException`
`delete()`	Datei löschen	`IOException`
`move()`	Datei verschieben oder umbenennen	`IOException`

`Files`: Kopieren, Verschieben, Umbenennen, Löschen, Erzeugen

Anmerkung zu `createTempFile()`:

Die durch diese Methoden generierten Dateinamen setzen sich aus drei Teilen zusammen:

- Präfix am Anfang des Dateinamens, darf `null` sein).
- Zufallszahl (in der Mitte des Dateinamens).
- Suffix (Endung) am Schluss der Datei.

Beispiel:

```
createTempFile("abc_", ".blah");    // erzeugt Namen, wie etwa abc_37621.blah
```

Es gibt verschiedene Varianten dieser Methode. Eine davon verwendet automatisch das von der Plattform vorgegebene Standardverzeichnis für Temporärdateien, eine andere erlaubt es Ihnen, das Temporärverzeichnis selbst zu wählen.

Die folgenden Operationen können auch mit symbolischen Links umgehen: man kann wählen, ob man einen Link auflösen (verfolgen) will oder nicht):

Methode	Beschreibung	throws
`exists()`	Existenztest	-
`getLastModifiedTime()`	Zeitpunkt der letzten Änderung (`FileTime`-Objekt)	`IOException`
`isDirectory()`	Prüfen, ob Element ein Verzeichnis ist	-
`isExecutable()`	Prüfen, ob Element ausführbar ist	-
`isReadable()`	Darf die laufende JVM diese Datei lesen?	-
`isWritable()`	Darf die laufende JVM in diese Datei schreiben?	-

`Files:` Datei-Attribute

4.2.3 Verzeichnisse

Grundoperationen

Methode	Beschreibung	throws
`createDirectory()`	Erzeugt ein Verzeichnis. Der als Parameter angegebene Pfad muss bis einschliesslich des vorletzten Elements bereits existieren. Falls der gesamte Pfad bereits existiert, schlägt die Operation fehl.	`IOException`
`createDirectories()`	Erzeugt einen ganzen Pfad von Verzeichnissen. Beliebige Teile des Pfads (auch der ganze Pfad) dürfen bereits vor der Operation existieren, ohne dass sie fehlschlägt.	`IOException`
`delete()`	Löscht ein *leeres* Verzeichnis. Nicht leere Verzeichnisse müssen zuvor entleert werden (Sicherheit!).	`IOException`

`Files:` Verzeichnisse anlegen oder löschen

Beispiele:

```
Path p = Files.createDirectory(Path.of("E:\\", "Data", "FHNW", "OOP2i"));
  // Läuft nur genau dann ordnungsgemäss, falls E:\Data\FHNW bereits existiert,
  // nicht aber E:\Data\FHNW\OOP2i. p enthält am Schluss E:\Data\FHNW\OOP2i

Path p = Files.createDirectories(Path.of("E:\\", "Data", "FHNW", "OOP2i"));
  // Baut E:\Data\FHNW\OOP2i auf, unabhängig davon, welche Teile dieses Pfads
  // bereits existieren. p enthält am Schluss E:\Data\FHNW\OOP2i
```

Verzeichnisse lesen

Methode	Beschreibung	throws
`list(Path)`	Falls es sich beim angegebenen Pfad um ein Verzeichnis handelt, wird ein **Stream** erzeugt, welcher die Elemente des Verzeichnisses enthält. Bezeichnet der Pfad eine einfache Datei, wird `NotDirectoryException` geworfen.	`IOException`
`walk(Path)`	Wie `list()`, aber der ganze Baum unter dem Pfad wird rekursiv traversiert (Reihenfolge: *pre-order*).	`IOException`

`Files:` Verzeichnisse auslesen

Beispiel 1: Ein Verzeichnis auslesen, Ausgabe der Namen der Elemente

```
List<Path> plist = Files.list(Path.of("E:\\", "FHNW")).toList();
for (Path p : plist)
  System.out.println(p.getFileName());  // nicht den ganzen Pfad anzeigen, nur das Ende
```

Beispiel 2: Ein Verzeichnis und alle seine Unterverzeichnisse auslesen (automatisch)

```
List<Path> plist = Files.walk(Path.of("E:\\", "FHNW")).toList();
for (Path p : plist)
  System.out.println(p.getFileName());  // nicht den ganzen Pfad anzeigen, nur das Ende
```

Beispiel 3: Ein Verzeichnis und alle seine Unterverzeichnisse auslesen (manuell)

```
public static void traversePath(Path p) throws IOException{
  traversePath(p, 0, "");
}

private static void traversePath(Path p, int depth, String prefix) throws IOException{
  System.out.println(prefix + p.getFileName());
  if (Files.isDirectory(p)) {                    // Rekursionsabbruch
    prefix += "  ";                              // Einrückungstiefe erhöhen
    List<Path> list = Files.list(p).toList();
    for (Path element : list) {
      traversePath(element, depth+1, prefix);    // rekursiver Abstieg
    }
  }
}
```

Die obere Methode wird aufgerufen und delegiert die Rekursion an die private untere Version. Dies ist nötig, um weitere Parameter einführen zu können, die dem Einrücken von Zeilen dienen.

Navigieren in einem Pfad

Es gibt auch andere Möglichkeiten als die `Files`-Methoden `list()` und `walk()`, um in Pfaden zu navigieren. Die Klasse `java.nio.file.Path` selbst bietet dafür einige interessante Methoden (auszugsweise):

Methode	Beschreibung	throws
`getFileName()`	Gibt das letzte Element des Pfads zurück (relativer Pfad).	-
`getParent()`	Gibt einen Pfad zurück, welcher alles ausser dem letzten Element enthält.	-
`getRoot()`	Gibt das Wurzel-Element als Pfad zurück	-
`isAbsolute()`	Ist der Pfad absolut?	-
`iterator()`	Iterator durch die einzelnen Elemente des Pfads, beginnend nahe der Wurzel	-
`normalize()`	Entfernt redundante Elemente aus dem Pfad	-
`of(String, String...)`	Konstruiert einen Pfad aus einzelnen Elementen.	`InvalidPath-Exception`
`resolve(Path other)` `resolve(String other)`	Die beiden Pfade `this` und `other` werden vereinigt. Üblicherweise ist `other` relativ und `this` relativ oder absolut. In diesem Fall wird `other` an `this` angehängt. Einige Spezialfälle sind zu beachten: • Ist `other` absolut, wird `other` zurückgegeben. • Ist `other` leer, wird `this` zurückgegeben.	`InvalidPath-Exception`
`toAbsolutePath()`	Wandelt einen relativen in einen absoluten Pfad um. Falls der Pfad bereits absolut ist, geschieht nichts. Ist der Pfad relativ, wird er um das Arbeitsverzeichnis erweitert.	-

`Path`: Navigation und Anderes

4.3 Text-Dateien

Die Klassen `BufferedWriter` und `BufferedReader` können verwendet werden, um bequem zeilenweise mit Dateien umzugehen, welche reinen Text enthalten. Dateien, die mit diesen Streams geschrieben werden, können mit beliebigen Reintext-Editoren (kein MS-Word, aber z.B. Windows-Editor) weiterverarbeitet werden.

Ein kleines Nutzungsbeispiel:

```
public static void main(String[] args) throws IOException{

  Path text = Path.of("E:\\", "FHNW", "OOP2i", "text.txt");

  BufferedWriter bw = Files.newBufferedWriter(text);  // Factory-Methode von Files
  bw.write("Zeile eins ");
  bw.newLine();
  bw.write("Zeile zwei ");          // schreibt einen String (ohne Zeilenende)
  bw.close();

  BufferedReader br = Files.newBufferedReader(text);  // Factory-Methode von Files
  String s;
  while ((s = br.readLine()) != null){     // eine Zuweisung hat einen "Rückgabewert"!
    System.out.println(s);
  }
  br.close();

}
```

Anmerkungen:

- `newLine()` schreibt einen Zeilenumbruch. Welches Format er aufweist, hängt allerdings von der Plattform ab. Unter Windows wird `"\r\n"` geschrieben, in UNIX-basierten Systemen hingegen `"\n"`.
- In diesem Beispiel werden die Dateien *ungesichert* geschlossen. Dies ist eigentlich kein guter Stil und geschieht hier aus Platzgründen. Später werden Sie sehen, wie man eine Datei *sicher* schliesst.

ASCII? Latin-1? Unicode?

In reinem Text gibt es verschiedene Zeichencodes. Damit Textdateien korrekt gelesen werden können, muss bekannt sein, welcher Code beim Schreiben verwendet wurde. Die wichtigsten Codes:

- ASCII (definiert Zeichencodes 0 bis 127, keine Umlaute, nur für Englisch geeignet).
- Latin-1 (auch: ISO-8859-1, Zeichencodes 0 bis 128, für westliche Sprachen mit lateinischem Alphabet).
- UTF-8 (Codierung, welche den ganzen Unicode-Zeichensatz platzsparend abbildet. Platzbedarf zwischen 8 und 32Bits pro Zeichen). *Verwenden Sie, wenn immer möglich UTF-8!*

Im Beispiel oben wurde bei der Erzeugung des Readers und des Writers keine Codierung angegeben. Wenn man keinen Code angibt, wird automatisch UTF-8 verwendet. Falls Sie eine andere Codierung benötigen oder wenn Sie ganz sicher sein wollen, schreiben Sie zum Beispiel folgendes:

```
BufferedWriter bw = Files.newBufferedWriter(text, Charset.forName("ISO-8859-1"));
  :
  :
BufferedReader br = Files.newBufferedReader(text, Charset.forName("ISO-8859-1"));
```

Text in einen Stream lesen

Wenn Sie die Zeilen einer Textdatei in einem Collection-Stream weiterverarbeiten möchten, benutzen Sie die Methode `Files::lines(Path)`. Sie liefert einen `Stream<String>` zurück.

Ein Beispiel:

```
Files.lines(myPath).distinct().sorted().toList();
```

4.4 Binärdateien

Alles, was nicht reiner Text ist, wird als «binär» bezeichnet (dies, obschon reiner Text selbstverständlich auch aus binärer Information besteht). Unter Binärdaten verstehen wir also zum Beispiel Bilder, Musik, Videos, Messdatenreihen, formatierter Text (z.B. Word), Daten für eine Tabellenkalkulation, compilierte Programme. Java-Quelltexte, XML, Markdown, LaTex, HTML hingegen gehören zur Welt der Reintexte.

In diesem Kapitel geht es darum, Binärdateien zu schreiben und zu lesen. Im Gegensatz zu Textdateien sind Binärdateien nicht zeilenweise organisiert. Wir finden also keine Zeilenumbrüche, welche die Funktion haben, Daten voneinander zu trennen (wohl können aber die Bytes enthalten sein, welche in reinen Texten als Umbrüche wirken, nur dass sie in einer Binärdatei keine gesonderte Bedeutung aufweisen).

Um Binärdaten sinnvoll interpretieren zu können, muss man deren Format kennen. Für öffentliche Formate sind solche Spezifikationen öffentlich zugänglich.

4.4.1 Byte-Streams

Die kleinste Informationseinheit, die aus einer Datei gelesen oder in eine Datei geschrieben werden kann, ist ein Byte (= 8Bits).[1] Die elementarsten I/O-Streams, welche Java zu bieten hat, erlauben es Ihnen, Binärdateien byteweise zu lesen und zu schreiben.

Der einfachste Weg, an byte-orientierte Streams zu gelangen, erfolgt über die Klasse `Files`:

```
static InputStream newInputStream(Path, OpenOption)      // aus einer Binärdatei lesen
static OutputStream newOutputStream(Path, OpenOption)    // in eine Binärdatei schreiben
```

I/O-Streams stellen eine Verbindung zwischen der Applikation und einer Datei her, sie *öffnet* die Datei. Wenn Sie Ihre Operationen auf der Datei beendet haben, müssen Sie die Datei wieder *schliessen*, damit keine Informationen verloren gehen können.

Ein Beispiel:

```
OutputStream out = Files.newOutputStream(...);
   :
   : diverse Schreib-Operationen
   :
out.close();
```

Dieses Beispiel ist problematisch: Die Schreib-Operationen können eine Exception werfen, und in einem solchen Fall würde der Stream nicht geschlossen. Also zweiter Versuch:

```
OutputStream out = Files.newOutputStream(...);
try{
    // diverse Schreib-Operationen
}catch (IOException e) {
    // tue irgendwas
}finally {
  out.close();
}
```

Auch diese Lösung ist nicht wirklich befriedigend, denn auch der `finally`-Block könnte eine Exception werfen. Die beste Lösung (*try-with-resources*):

```
try (OutputStream out = Files.newOutputStream(...))}
   :
   : diverse Schreib-Operationen
   :
}
```

1. Falls Sie einzelne Bits aus einer Datei analysieren wollen, müssen Sie Bitmasking- und/oder Bitshifting-Techniken einsetzen. Sie werden später besprochen.

Bytes lesen

Das Interface `java.io.InputStream` enthält unter anderem folgende Methoden:

Methode	Beschreibung	throws
`int read()`	Liest ein Byte (vorzeichenlos, d.h. im Bereich 0..255).	`IOException`
`int read(byte[] b)`	Liest Bytes in den gegebenen Buffer b. Die Anzahl wird zurückgegeben.	`IOException`
`byte[] readAllBytes()`	Liest alle verbleibenden Bytes in ein neues Array.	`IOException`
`byte[] readNBytes(int n)`	Liest maximal die durch n spezifizierte Anzahl Bytes in ein neues Array.	`IOException`
`void close()`	Datei schliessen	`IOException`

Anmerkungen:

- Wenn es darum geht, grössere Datenmengen zu lesen, sollten Sie `read(byte[])`, `readAllBytes()` oder `readNBytes()` bevorzugen. Das einfache `read()` ist dafür zu langsam.
- Jeder Stream arbeitet mit einem *File-Pointer* zusammen, welcher die zuletzt gelesene (oder geschriebene) Position markiert. Aufeinanderfolgende Operationen beginnen immer dort, wo die vorhergehende geendet hat.
- `read(byte[])` verwendet einen bereits existierenden Buffer `b`, um die eingelesenen Bytes zu speichern. Mit einer Operation können maximal `b.length` Bytes gelesen werden.
- `readAllBytes()` und `readNBytes()` erzeugen neue Arrays. Die erste dieser beiden Operationen sollten Sie nur verwenden, wenn die zu lesende Datenmenge erstens bekannt und zweitens nicht übermässig gross ist.

Informationen zur Arbeitsweise von `int read()` (*single-byte-read*):

Der Datentyp `byte` überspannt den Zahlenbereich von –128 bis +127 (8Bits, vorzeichenbehaftet). Wie wir an der Signatur von `read()` erkennen können, liefert sie einen `int` zurück. Die 8 Bits des gelesenen Bytes werden hierbei im niederwertigsten Byte des `int` abgelegt. Dies resultiert in einem *gelesenen* Wertebereich von 0 bis 255, die Zahl wird also vorzeichenlos. Um das Ende der Datei anzuzeigen, gibt `read()` den Wert –1 zurück, wenn keine Daten mehr vorhanden sind.

Wenn Sie also den originalen, vorzeichenbehafteten Wert eines Bytes erhalten wollen, müssen Sie den `int` explizit in ein `byte` casten. Vor dem Cast müssen Sie aber den gelesenen Wert zuerst auf –1 überprüfen, um das Ende der Datei zu erkennen.

Beispiele:

```
InputStream in = Files.newInputStream(Path.of("aufgaben-01/bytes.bin"));

// single-byte-read I ************************************************
int single;
while ((single = in.read()) >= 0) {    // check for end-of-file (EOF)
  System.out.println(single);          // 0  51  204  15  240  255
}

// bulk-read *********************************************************
byte[] all = in.readAllBytes();
for (byte b : all) {
    System.out.println(b);             // 0  51  -52  15  -16  -1
}

// single-byte-read II ***********************************************
int single;
while ((single = in.read()) >= 0) {    // check for end-of-file (EOF)
  System.out.println((byte)single);    // 0  51  -52  15  -16  -1
}
```

Bytes schreiben

Das Interface `java.io.OutputStream` enthält unter anderem folgende Methoden:

Methode	Beschreibung	throws
`void write(int)`	Schreibt ein einzelnes Byte (der `int`-Wert wird beschnitten)	`IOException`
`void write(byte[] b)`	Schreibt ein ganzes Array in die Datei	`IOException`
`void write(byte[] b, int start, int n)`	Schreibt `n` Werte aus dem Array in die Datei, beginnend bei `start`.	`IOException`
`void close()`	Datei schliessen	`IOException`

Anmerkungen:

- Wird eine bereits existierende Datei im Normalmodus zum Schreiben geöffnet, verliert sie *automatisch ihren gesamten ursprünglichen Inhalt*, welcher dann durch die neu geschriebenen Daten ersetzt wird (oder durch nichts, wenn man sie unverrichteter Dinge wieder schliesst):

  ```
  OutputStream out = Files.newOutputStream(path);    // öffnen im Normalmodus
  ```

  Sie haben die Möglichkeit, eine Datei in einem alternativen Modus zu öffnen (siehe Unterkapitel *Dateien öffnen*).
- Versucht man, eine nicht existierende Datei zum Schreiben zu öffnen, wird normalerweise eine neue Datei angelegt. Dies kann man verhindern (siehe Unterkapitel *Dateien öffnen*).

Dateien öffnen

Die beiden Methoden `newInputStream()` und `newOutputStream()` verfügen nebst dem obligatorisch anzugebenden Pfad einen weiteren Parameter vom Typ `OpenOptions`, mit welchem man den Öffnungsmodus spezifizieren kann. Er ist als Vararg-Parameter konstruiert. Sie können mehrere Optionen nennen.

Die wichtigsten Optionen (`java.nio.file.StandardOpenOption`):

Option	Beschreibung
`APPEND`	Datei nicht löschen, sondern neue Daten an die bestehenden anhängen.
`CREATE`	Erzeuge eine neue Datei, falls die angegebene noch nicht existiert
`CREATE_NEW`	Erzeuge immer eine neue Datei. Wirft eine Exception, falls die Datei bereits existiert.
`DELETE_ON_CLOSE`	Die Datei wird gelöscht, nachdem sie geschlossen wurde. Dies ist besonders für Temporärdateien nützlich.

Beispiele:

```
import static java.nio.file.StandardOpenOption.*;

// Datei leeren, falls sie existiert. Eine neue Datei anlegen, falls sie nicht existiert
OutputStream out = Files.newOutputStream(path);

// Falls Datei existiert: Daten anhängen. Falls sie nicht existiert: Exception
OutputStream out = Files.newOutputStream(path, APPEND);

// Falls Datei existiert: Daten anhängen. Falls sie nicht existiert: neu anlegen
OutputStream out = Files.newOutputStream(path, CREATE, APPEND);

// Neue Datei anlegen. Falls sie bereits existiert: Exception
OutputStream out = Files.newOutputStream(path, CREATE_NEW);
```

4.4.2 Mit einzelnen Bits arbeiten (Bit-Operationen)

Wie bereits weiter oben erwähnt, ist die kleinste Zugriffeinheit in Byte-Streams ein Byte. Manchmal ist es allerdings erforderlich, in einem Byte einzelne Bits zu überprüfen (*ist die vorliegende Audiodatei mono oder stereo?*) oder aber in einem zu schreibenden Byte ein bestimmtes Bit zu setzen (*setze das Fehler-Bit zurück*).

Im folgenden Abschnitt werden alle dazu nötigen Techniken beschrieben (*siehe auch: Skript OOP1, Seite 1.9*):

Bitmasking

Die zu bearbeitende ganzzahlige Variable wird mit einer sogenannten *Maske*, die vom selben Typ sein muss, über bitweises **UND**, **ODER** bzw. **XOR** verknüpft. Dabei werden in der Variablen alle Bits, welche in der Maske aktiviert wurden, auf einen definierten Zustand gesetzt.

- Mit einer **UND**-Maske werden alle Bits in der Variablen gelöscht, welche in der Maske mit 0 belegt wurden.
- Mit einer **ODER**-Maske werden alle Bits in der Variablen gesetzt, welche in der Maske mit 1 belegt wurden.
- Mit einer **XOR**-Maske werden alle Bits in der Variablen invertiert, welche in der Maske mit 1 belegt wurden.

Alle Bits der Variablen, deren Gegenpart in der Maske nicht aktiviert wurden, bleiben original erhalten.

Beispiele:

- Lösche Bit 5 der `short`-Variablen `s`.

```
s &= 0xFFDF;        // Maske: 1111'1111'1101'1111
```

- Setze die höchstwertigen drei Bits der `int`-Variablen `i` auf Eins.

```
i |= 0xE0000000;    // Maske: 1110'0000...0000
```

- Invertiere Bit 6 der `byte`-Variable `b`.

```
b ^= 0x40;          // Maske: 0100'0000
```

- Prüfe Bit 3 der `byte`-Variable `b` (*ist es gesetzt?*).

```
if ( (b & 0x08) != 0 ){        // Maske: 0000'1000
  // do something...
}
```

→ Beachten Sie, dass die Zählung der Bits rechts an der niederwertigsten Stelle mit 0 beginnt.

→ Masken werden üblicherweise in hexadezimaler Form angegeben, da sich aus Hex-Literalen einfacher auf das gewünschte Bit schliessen lässt.

→ Verwenden Sie Kombi-Zuweisungen wie in den Beispielen oben, denn damit umgehen Sie Type-Casts. Also wenn möglich nicht so:

```
b = (byte)(b ^ 0x40);     // b ^ 0x40 liefert einen int, und der passt nicht nach b
```

Bit-Shifting

Schiebe-Operationen werden entweder dazu benutzt, um Bits in eine für die weitere Verarbeitung bequemere Position zu bringen oder um billige Multiplikationen bzw. Divisionen mit Potenzen von 2 zu realisieren.

Folgende Eigenheiten der Bit-Shifting-Operationen sind zu beachten:

- `<<` schiebt nach links (rechts werden Nullen nachgefüllt)
- `>>>` schiebt nach rechts (links werden Nullen nachgefüllt)
- `>>` schiebt nach rechts (links werden Einsen nachgefüllt, falls Zahl negativ, sonst Nullen)

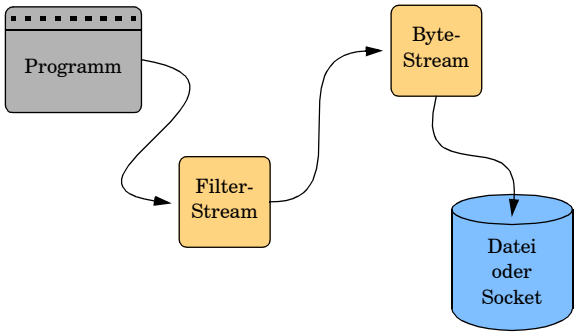
Beispiel: `u = i << 3;` `// 3 Bits nach links schieben. Entspricht einer Multiplikation mit 8`

4.4.3 Streams für primitive Datentypen

Wenn Sie Werte primitiver Datentypen wie zum Beispiel `int` oder `double` in Dateien schreiben wollen, können Sie dies mit den bisher besprochenen Möglichkeiten nur tun, indem Sie sie in Strings umwandeln und dann in eine Reintext-Datei ausgeben. Diese Strategie hat einige Nachteile:

- Das Umwandeln von Zahlen, insbesondere von Fliesskommazahlen, erfordert relativ viel Rechenleistung und ist deshalb insbesondere für grosse Datenmengen ineffizient.
- Ein `double`-Wert benötigt im **RAM** 64 Bits. Wenn er mit maximaler Genauigkeit in einen String verwandelt und so in einer Textdatei gespeichert wird, belegt er im günstigsten Fall 184 Bits, also beinahe dreimal so viel wie in der Original-Variablen (vorausgesetzt, dass pro Zeichen 1 Byte gespeichert wird).
- Bei der Umwandlung zwischen `double` und `String` können systembedingt Rundungsfehler auftreten.[2]

Dieser problematische Schritt via Strings kann in Java umgangen werden, indem man zwischen Byte-Stream und Applikation einen *Filter-Stream* legt, der eine beliebige primitive Variable in eine Sequenz von Bytes zerlegt.



Selbstverständlich existieren auch die passenden Gegenstücke, mit welchen primitive Wert verlustfrei wieder eingelesen werden können.

Wenn Sie auf diese Weise Daten Ihrer Applikation sichern wollen, müssen Sie ein klare Spezifikation entwerfen, damit die Daten später wieder korrekt eingelesen werden können.

Die Java-Bibliothek enthält in `java.io` zwei Basisklassen für Filter-Streams, `FilterInputStream` und `FilterOutputStream`. Davon abgeleitet sind `DataInputStream` bzw. `DataOutputStream`. Objekte der letzteren Klassen werden benutzt, um primitive Werte in Byte-Sequenzen zu verwandeln und umgekehrt.

Beispiel (*Lesen und Schreiben einer float-Variablen*):

```
public static void main(String[] args) throws IOException{
  Path fp = Path.of("D:\\", "FHNW", "OOP2i", "IOStreams", "float.bin");
  float x = -13.7f;

  try( DataOutputStream out = new DataOutputStream(Files.newOutputStream(fp)) ){
    out.writeFloat(x);
  }

  try( DataInputStream in = new DataInputStream(Files.newInputStream(fp)) ){
    x = in.readFloat();
    System.out.print("Der gelesene Wert: " + x);
  }
}
```

- In diesem Beispiel wird *try-with-resources* verwendet, um das Problem mit dem Schliessen von Dateien korrekt zu lösen.
- Die mit den beiden verwendbaren Methoden sind in den beiden Interfaces `java.io.DataInput` bzw. `java.io.DataOutput` spezifiziert.
- Alle zahlenverarbeitenden Methoden schreiben bzw. lesen Werte im *Big-Endian*-Format, d.h. das höchstwertige Byte (MSB) wird zuerst geschrieben/gelesen, das niederwertigste (LSB) zuletzt. Wenn Sie Fremddaten aus Dateien verarbeiten müssen, ist vorher zu prüfen, ob sie mit dem selben Codierungsschema geschrieben wurden.

2. Wenn Sie Genaueres darüber erfahren wollen, empfehlen wir Ihnen den Besuch der Vorlesung *Algorithmen & Datenstrukturen 1*.

4.5 Random-Access Dateien

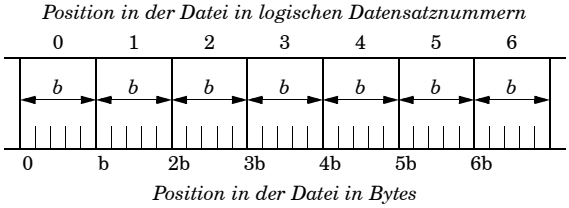
4.5.1 Grundkonzept

Die bisher besprochenen I/O-Streams eignen sich für alle Aufgaben, in welchen Daten sequentiell verarbeitet werden. Manchmal ist es allerdings wünschenswert, dass man auf Daten in beliebiger, vielleicht sogar zufälliger Reihenfolge zugreifen kann. Um dies zu gewährleisten muss man die Daten in *Datensätze* (*records*) gliedern, die alle genau gleich gross sind (z.B. Adresskartei, Messdaten-Erfassung usw.). Dies wird durch geeignetes Planen der Datenarchitektur gewährleistet.

Die Vorteile von Datensätzen mit bekannter, konstanter Grösse sind offensichtlich:

- Durch Adressrechnung kann jeder beliebige Datensatz direkt angesprungen werden, ohne dass die ganze Datei hierfür zeitraubend durchlaufen werden muss.
- Man kann jederzeit einen beliebigen Datensatz auslesen, editieren und dann wieder zurückschreiben.

Bei gegebener Datensatzlänge b errechnet sich die Startposition s des Datensatzes i wie folgt: $s = ib$ (vorausgesetzt, die Datensatznummern beginnen bei 0).



Damit dies funktioniert, muss der verwendete I/O-Stream drei Bedingungen erfüllen:

a) In einer geöffneten Datei müssen sowohl Lese- als auch Schreiboperationen möglich sein.
b) Der Dateizeiger muss frei positioniert werden können.
c) Der Stream sollte alle primitiven Datentypen unterstützen (nicht zwingend, wäre aber sehr bequem).

Objekte der Klasse `RandomAccessFile` erfüllen alle Bedingungen.

4.5.2 Die Klasse `RandomAccessFile`

Die meisten Methoden dieser Klasse kennen wir bereits (Interfaces `DataInput` und `DataOutput`):

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

boolean readBoolean();
byte readByte();
short readShort();
char readChar();
int readInt();
long readLong();
float readFloat();
double readDouble();

int readUnsignedByte();
int readUnsignedShort();
String readLine();
int skipBytes(int n);
```

```
void write(int b);
void write(byte[] b);
void write(byte[] b, int off, int len);

void writeBoolean(boolean b);
void writeByte(byte b);
void writeShort(short s);
void writeChar(char c);
void writeInt(int i);
void writeLong(long l);
void writeFloat(float f);
void writeDouble(double d);

void writeChars(String s);
void writeBytes(String s);
```

In den Konstruktoren werden eine Datei und der gewünschte Zugriffsmodus angegeben:

```
RandomAccessFile(File f, String mode);
RandomAccessFile(String n, String mode);
```

Als Modi können die Werte `"r"`, `"rw"`, `"rws"` oder `"rwd"` angegeben werden. Man beachte bei den rw-Modi, dass der Rechner versucht, eine neue Datei zu erzeugen, falls sie nicht bereits existiert. Die Modi `"rws"` und `"rwd"` erzwingen zusätzlich ein Durchschreiben auf die Datei, während `"rw"` zunächst nur auf den Buffer schreibt und eine Durchschreibeaktion nur dann auslöst, wenn es nötig ist (z.B. wenn der Buffer voll ist). *Lesen Sie hierzu Details in der offiziellen Dokumentation nach.*

Speziell von Interesse für uns sind folgende beiden Methoden:

```
void seek(long pos);
long getFilePointer();
```

Mit diesen beiden Methoden stellen Sie den Positionszeiger auf den gewünschten Wert ein, bzw. lesen die aktuelle Position aus. Der Referenzwert ist dabei jeweils der Anfang der Datei (0). Dadurch lässt sich der Zeiger auf einfache Weise auf den Anfang eines Datensatzes beliebiger Grösse einstellen. Wir gehen von folgender Situation aus:

Alle Lese- und Schreib-Operationen versetzen den Positionszeiger automatisch, so dass bei sequentiellem Bearbeiten der Datensätze nur eine Positionierung nötig ist.

4.6 Daten aus dem Web lesen

Mit I/O-Streams kann man von einer URL Daten einlesen. Hierzu ist zunächst ein `URL`-Objekt (`java.net`) mit einer gültigen Adresse zu erstellen. Anschliessend kann von der URL ein `InputStream` bezogen und gelesen werden. Dies ist der einfachste Weg.

Ein kleines Beispiel:

```
import java.net.URL;
import java.io.InputStream;

public class main_URL{

  public static void main(String[] args) throws Exception{
    URL url = new URL("https://www.fhnw.ch");          // make URL
    InputStream in = url.openStream();                 // get Stream

    for (int i=0; i<1000; ++i) {
      System.out.print((char)(in.read()));             // read...
    }
  }

}
```

4.7 Objekt-Serialisierung (nicht Prüfungsstoff)

4.7.1 Einführung

Muss ich ein Objekt, wenn ich es auf Disk schreiben will, immer manuell in seine primitiven Einzelteile zerlegen und die dann einzeln schreiben? Die Antwort lautet: nein! Mit den beiden Klassen `ObjectInputStream` und `ObjectOutputStream` (beide in `java.io`) kann man das auf mehr oder wenige bequeme Weise automatisch oder zumindest halbautomatisch erledigen (lassen). Wenn man dieses Thema in seiner Tiefe verstehen will, braucht es allerdings einen gewissen Effort. Wir werden dieses Thema in diesem Kapitel nicht in seiner ganzen Breite behandeln.

Folgendes Codefragment zeigt, wie man ganze Objekte schreiben und lesen kann. Dies funktioniert ähnlich wie mit den Filter-Streams, die wir in Abschnitt 4.4.3 kennengelernt hatten:

```
Object o = ...
Path fp = ...

// serialisieren
try( ObjectOutputStream out = new ObjectOutputStream(Files.newOutputStream(fp)) ){
  out.writeObject(o);
}

// deserialisieren
try( ObjectInputStream in = new ObjectInputStream(Files.newInputStream(fp)) ){
  o = in.readObject();
}
// die Catch-Blöcke wurden aus Platzgründen weggelassen. Es sind IOException und
// ClassNotFoundException zu fangen oder zu delegieren
```

4.7.2 Die verschiedenen Strategien der Serialisierung

Um Objekte einer Klasse serialisierbar zu machen, müssen Sie einige Vorkehrungen treffen. Dabei kann man zwischen verschiedenen Strategien wählen, welche sich hinsichtlich des zu erbringenden Programmier-Aufwandes, der Kontrolle über Serialisierungsprozess, des Speicherbedarfs eines serialisierten Objekts sowie dem Grad der Standardisierung unterscheiden:

Die drei Strategien im Überblick:

Parameter	automatisch (a)	manuell (b)	non-standard (c)
Programmier-Aufwand	klein	gross	gross
Speicherbedarf	gross	klein	minimal
Kontrolle	gering	gross	maximal
Standard erfüllt?	ja	ja	nein
was ist zu tun?	Die Klasse mus das Interface *Serializable* implementieren. Es handelt sich hierbei um ein Marker-Interface, d.h. es sind keine Methoden zu implementieren. Ein geringes Mass an Kontrolle über den Serialisierungsprozess ist möglich, indem Attribute als *transient* deklariert und so von der Serialisierung ausgeschlossen werden.	Die Klasse mus das Interface *Externalizable* implementieren. Es enthält zwei Methoden, welche die Details der Serialisierung steuern und vom Entwickler bereitzustellen sind.	Sie müssen vollständig eigene Methoden zur Serialisierung schreiben. Die Serialisierung erfolgt dann nicht über den Objekt-Serialisierer sondern in der Regel über einen einfachen Filter-Stream, mit welchem beispielsweise ein Byte-Array geschrieben wird.

a) Serialisieren mit `Serializable`

Weitgehend automatisierte Form. Die Java-VM serialisiert das betroffene Objekt in Eigenregie, wobei ein rekursives Verfahren verwendet wird: Enthält das Objekt Referenzen auf andere Objekte, werden diese Objekte ebenfalls serialisiert und gespeichert (sofern erstens die Referenz nicht als `transient` deklariert wurde, und zweitens falls die referenzierten Objekte überhaupt serialisierbar sind). Es ist daher möglich, mit dieser Strategie eine ganze Datenstruktur mit einer einzigen Operation in eine Datei zu schreiben.

Es ist zu beachten, dass bei der Serialisierung eines ganzen Objekt-Graphen, jeder einzelne Knoten für sich nach Standard serialisierbar sein muss (d.h. jedes betroffene Objekt in der Struktur muss `Serializable` oder `Externalizable` implementieren. Der Mechanismus ist intelligent genug, dass er zirkuläre Bezüge auflösen kann und jedes Objekt maximal einmal abspeichert.

b) Serialisieren mit `Externalizable`

Wenn Sie selbst die Kontrolle übernehmen wollen, wie Ihr Objekt serialisiert werden soll, müssen Sie für ihre Klasse folgende Methoden implementieren (Interface `Externalizable`):

```
void readExternal(ObjectInput)
void writeExternal(ObjectOutput)
```

Falls ein zu serialisierendes Objekt auf andere Objekte verweist, ist im Falle des Bedarfs explizit für deren Serialisierung zu sorgen. Wird für ein mit dieser Strategie zu serialisierendes Objekt `writeObject()` aufgerufen, ruft die VM ihrerseits `writeExternal()` auf, wobei der Stream, an welchem `writeObject()` aufgerufen wurde, als Parameter in `writeExternal()` eingesetzt wird.

Sie müssen also in Ihrer Implementierung von `writeExternal()` die einzelnen Attribute in den als Parameter übergebenen Stream schreiben, was aber relativ bequem ist, weil `ObjectOutput` auch das Interface `DataOutput` implementiert. Selbstverständlich gilt das alles auch in der umgekehrten Richtung.

c) Non-Standard-Serialisierung

Der Aufwand der Strategie c) ist vergleichbar mit b): Alles, was Sie in b) ihn den Stream schreiben, können Sie auch über einen Filter-Stream in einen Byte-Array-Stream schreiben. Der Unterschied zwischen den Standard- und dem Non-Standard-Verfahren besteht in erster Linie darin, dass die Java-VM bei a) und b) zusätzliche Informationen über die Identität des serialisierten Objekts mit in den Stream schreibt und deshalb unter Umständen bedeutend mehr Speicher im Ziel des Streams beansprucht. Im weiteren ist dieser Speicherbedarf auch nicht zwingend vollständig voraussagbar. Die Signaturen für Non-Standard-Serialisierer könnten wie folgt aussehen:

```
public byte[] serialize()
public void deserialize(byte[] data)
```

Tipp:

Sehen Sie sich die beiden Klassen `ByteArrayInputStream` und `ByteArrayOutputStream` in der Javadoc an. Diesen beiden Streamtypen funktionieren im Prinzip wie wie gwöhnliche Byte-Streams, aber anstatt Dateien für die Lese- und Schreiboperationen zu benutzen, werden Byte-Arrays verwendet.

Diese beiden Klassen können für die Non-Standard-Serialisierung recht nützlich sein. Ausserdem können Sie sie auch verwenden, um Stream-Operationen zu simulieren.