

Inhaltsverzeichnis

1	Dateiverarbeitung in Java.....	2
1.1	Dateien aus Sicht des Dateisystems	2
1.2	Verschiedene Arten des Dateizugriffs.....	3
1.2.1	Sequentieller vs. wahlfreier Zugriff	3
1.2.2	Zugriff nach Dateneinheit	3
2	Lesen und Schreiben von Textdateien	5
2.1	Zeichenweises Lesen von Textdateien	5
2.2	Methoden für <code>Path</code> Objekte	6
2.3	Zeilenweises Lesen von Textdateien.....	7
2.4	Schreiben von Textdateien	9
3	Lesen und Schreiben von serialisierten Objekten	10
3.1	Begriffsbestimmung	10
3.2	Serialisierung und Deserialisierung.....	10
3.2.1	Prinzip der Serialisierung.....	10
3.2.2	Prinzip der Deserialisierung.....	11
3.2.3	Versionierung.....	11
3.3	Realisierung von Serialisierung und Deserialisierung in Java	12
3.3.1	Voraussetzungen	12
3.3.2	Serialisierung (Schreiben).....	12
3.3.3	Deserialisierung (Lesen)	13

1 Dateiverarbeitung in Java

Bisher haben wir Daten nur in den zu einer Methode, zu einem Objekt oder zu einer Klasse gehörigen Variablen gespeichert. Zwar ist der lesende und schreibende Zugriff auf Variablen bequem und schnell zu realisieren, doch spätestens beim Verlassen des Programms gehen alle Variableninhalte verloren.

Eine Möglichkeit Daten eines Programms dauerhaft zu Speichern (man spricht auch von „persistieren“) besteht natürlich darin, sie im Dateisystem in einer Datei abzulegen. Eine weitere Möglichkeit wäre die Speicherung in einem Datenbanksystem auf das später eingegangen wird.

1.1 Dateien aus Sicht des Dateisystems

Aus Sicht des Dateisystems ist eine Datei einfach eine Folge von Bytes, die im hierarchischen Verzeichnisbaum unter einem bestimmten Pfadnamen (inklusive Dateiname) abgelegt sind. Zusätzlich zu den Nutzdaten speichert das Dateisystem außerdem noch sogenannte Metadaten, wie z.B. das Erstellungs- und Änderungsdatum, die Zugriffsrechte und weiteres.

Dateianfang

Dateiende

1.Byte	2.Byte	3.Byte	...	n.Byte
--------	--------	--------	-----	--------

Zusätzlich zu den Nutzdaten speichert das Dateisystem noch Metadaten, wie den Pfadnamen, Erstellungs- und Änderungsdaten, Zugriffsrechte, etc.

Wie der Inhalt einer Datei zu interpretieren ist, hängt vom erstellenden Anwendungsprogramm ab. Die Nutzdaten können einen Text bestehend aus Zeichen einer bestimmten Textkodierung (wie. z.B. ASCII, ISO8859_1, oder Unicode im Format UTF-8, etc.) darstellen oder aber ein Foto, eine Audiosequenz, usw.

Um auf eine Datei zuzugreifen, benötigt man ihren Dateipfad. Dieser enthält per Definition nicht nur den Pfad zum Verzeichnis der Datei, sondern auch den Dateinamen selbst.

Man unterscheidet absolute und relative Pfade.

Ein absoluter Pfad beginnt immer an der Wurzel des Verzeichnisbaums, d.h. bei Windows Systemen beginnt er mit einem Laufwerksbuchstaben¹:

```
D:\Users\stk\Documents\java\Prog_Projects\Dateien\Datei.xxx
```

Ein relativer Pfad geht immer vom „aktuellen“ Verzeichnis aus. Das ist standardmäßig das Verzeichnis, in dem eine Java Applikation gestartet wird. Einen relativen Pfad erkennt man daran, dass er nicht mit dem Trennzeichen² „\“ beginnt. Beispiele:

```
stk\Documents\java\Prog_Projects\Dateien\Datei.xxx
```

```
Datei.xxx
```

¹ Unix/Linux Systeme kennen keine Laufwerksbuchstaben. Ein absoluter Pfad beginnt einfach mit einem „/“

² Das Trennzeichen bei Unix/Linux Systemen ist „/“, anders als bei Windows

1.2 Verschiedene Arten des Dateizugriffs

1.2.1 Sequentieller vs. wahlfreier Zugriff

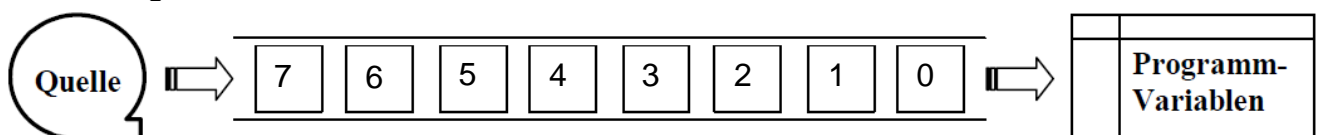
Erfolgt der Zugriff auf eine Datei (lesend oder schreibend) konsequent vorwärtsschreitend vom Anfang Richtung Ende der Datei, ohne den „Zugriffs-Cursor“ auf beliebige Positionen in der Datei zu verschieben, dann spricht man von sequentiellem Zugriff. Das bedeutet, dass der Zugriffs-Cursor nach einem Lese- bzw. Schreibvorgang immer hinter dem gerade gelesenen bzw. geschriebenen Datenelement steht.

Beim wahlfreiem Zugriff kann mit entsprechenden Befehlen zu beliebigen Positionen innerhalb der Datei navigiert werden, um auf beliebige Datenelemente lesend zuzugreifen bzw. vorhandene Daten zu überschreiben oder am Ende der Datei Daten anzufügen.

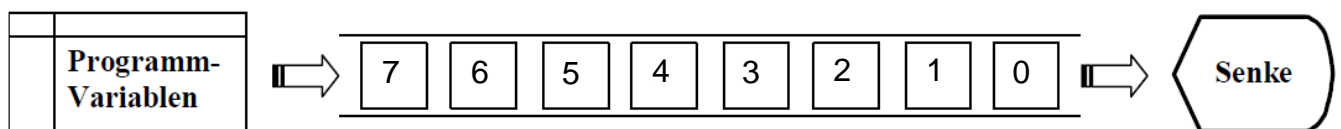
Im Weiteren werden wir uns nur mit sequentiellem Verarbeiten von Dateien beschäftigen. Bei den meisten Dateien, wie z.B. Textdokumenten wird der wahlfreie Zugriff effizienter dadurch gewährleistet, dass die komplette Datei für die Bearbeitung in den Arbeitsspeicher geladen wird.

1.2.2 Zugriff nach Dateneinheit

Lesender Zugriff



Schreibender Zugriff



Je nachdem welches Datenformat der Inhalt einer Datei hat, d.h. je nachdem wie die Daten zu interpretieren sind, als ausführbare Datei (z.B. Java Class-Datei), als Text oder als Foto etc., werden die im Bild dargestellten Dateneinheiten unterschiedlich sein.

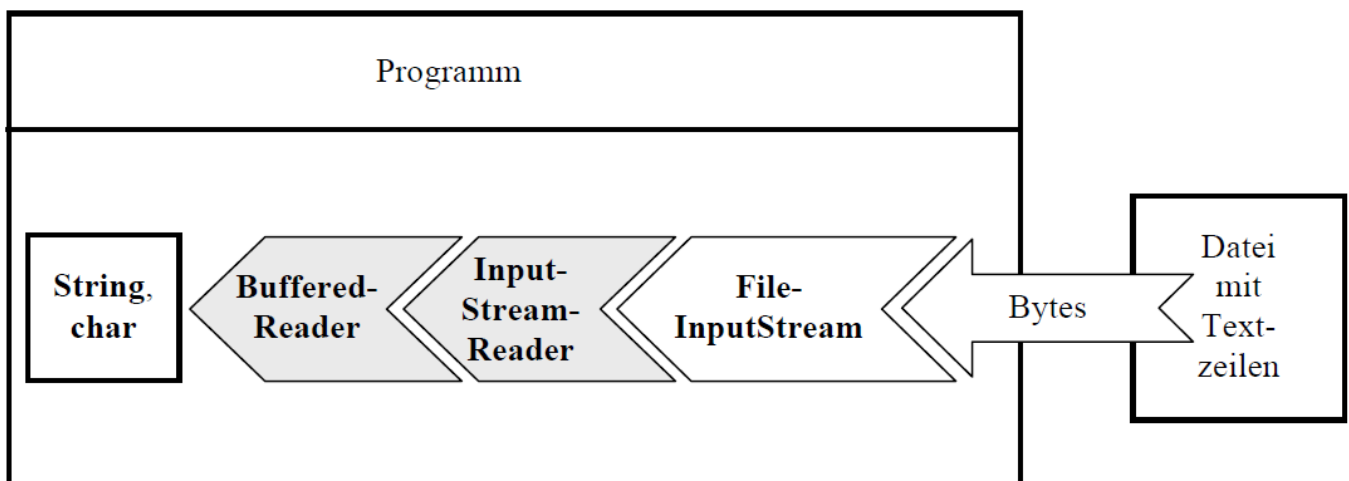
Dateien die byteweise bzw. blockweise zugegriffen werden: Dateien die prinzipiell eine Bytestruktur besitzen (wie z.B. der Bytecode in einer Java Class-Datei) oder solche, die aus Sicht der Programmiersprache keinem Standarddatenformat entsprechen, werden byteweise oder in Blöcken von vielfachen eines Bytes (z.B. 512 Byte) gelesen oder geschrieben. Die Interpretation der Bytes ist dann Sache des Anwendungsprogrammierers. Java stellt entsprechende Klassen für diese Zugriffsart zur Verfügung (z.B. `FileInputStream`, `FileOutputStream`).

Textdateien: Liest man eine Textdatei, so ist die Dateneinheit ein einzelnes Zeichen oder eine Textzeile. Je nach Kodierung der Textdatei kann ein einzelnes Zeichen aus 1 Byte bestehen (ASCII Code und ISO8859_1), aus 2 Byte bestehen (UTF-16 Format von Unicodezeichen) oder aus 1 bis 4 Byte bestehen (UTF-8 Format von Unicodezeichen). Um es dem Programmieren zu ermöglichen eine Textdatei unabhängig von der Kodierung zeichenweise bzw. zeilenweise zu lesen, bietet die Klassenbibliothek von Java entsprechende Klassen (z.B. `FileReader`, `FileWriter`, bzw. `BufferedReader`, `BufferedWriter`).

Dateien die elementare Daten enthalten, die entsprechend ihrer internen Java Kodierung abgelegt sind: Man kann Inhalte von Variablen primitiver Java Datentypen auch entsprechend ihrer Kodierung, wie sie im Arbeitsspeicher vorliegt, in Dateien speichern. Solche Dateien können nur mit einem Programm wieder gelesen werden, das die genaue Struktur des Dateiinhalts kennt. Die Speicherung von numerischen Daten in dieser Art ist, im Vergleich zur Speicherung in Textdateien, sehr effizient, sowohl hinsichtlich des Speicherbedarfs, als auch hinsichtlich der Performanz. Allerdings sind solche Dateien nur sehr schlecht portabel zwischen verschiedenen Programmen bzw. Plattformen. Entsprechende Klassen sind z.B. `DataInputStream`, `DataOutputStream`.

Dateien die Java Objekte enthalten, die in serialisierter Form abgelegt sind: Man kann Java Objekte auch in sogenannter serialisierter Form in Dateien speichern (s.u. Kapitel 3). Die Speicherung von Objekten in dieser Form ist, im Vergleich zur Speicherung in Textdateien oder Datenbanken, sehr effizient, sowohl hinsichtlich des Speicherbedarfs, als auch hinsichtlich der Performanz. Allerdings sind solche Dateien nur sehr schlecht portabel zwischen verschiedenen Programmen bzw. Plattformen. Entsprechende Klassen sind z.B. `ObjectInputStream`, `ObjectOutputStream`.

Kombination der Bibliotheksklassen nach dem „Baukastenprinzip“ am Beispiel Textdatei lesen:



Wie man im Bild erkennen kann, werden auch Textdateien tatsächlich mit Hilfe der `FileInputStream` Klasse byteweise gelesen. Die Bytes werden je nach Kodierung der Textdatei von der Klasse `InputStreamReader` zu Zeichen zusammengesetzt. Die Klasse `BufferedReader` sorgt dann noch für ein effizienteres Lesen von Zeichen und Zeilen. Ohne Pufferung könnte jeder einzelne Lesebefehl zu einem Zugriff auf das externe Laufwerk führen, was selbstverständlich sehr ineffizient wäre.

2 Lesen und Schreiben von Textdateien

2.1 Zeichenweises Lesen von Textdateien

Beispiel:

```
import java.io.*;
import java.nio.charset.StandardCharsets;
import java.nio.file.*;
import input.Eingabe;

public class LeseZeichenweise
{
    public static void main(String[] args)
    {
1      Path dateiPfad = null;
2      int iZeichen;

3      dateiPfad = Paths.get("Dateien/Text.txt");

4      try (BufferedReader reader =
        Files.newBufferedReader(dateiPfad, StandardCharsets.UTF_8))
        {
5          iZeichen = reader.read(); // versuche erstes Zeichen
                                   // zu lesen
                                   // wiederhole solange Dateiende nicht erreicht
6          while (iZeichen != -1)
            {
7              System.out.print((char)iZeichen);
              iZeichen = reader.read(); // lese nächstes Zeichen
            }
        }
        catch (UTFDataFormatException oe)
        {
            System.out.printf("FormatFehler beim Öffnen: %s\n",
                              oe.getMessage());
        }
        catch (Exception oe)
        {
            System.out.printf("Fehler beim Öffnen: %s\n",
                              oe.getMessage());
        }
    }
}
```

Anmerkungen:

1	Das Interface <code>Path</code> wurde mit Java 7 eingeführt. Es dient zum Speichern und bearbeiten von Verzeichnis-/Dateipfaden
2	Obwohl die Variable im Allg. ein Zeichen speichern soll, muss der Datentyp <code>int</code> verwendet werden, da außer Zeichen auch der Wert -1 vorkommen kann (s.u.)
3	Um aus einem Pfadstring ein <code>Path</code> -Objekt zu erzeugen, wird die Klassenmethode <code>get</code> der Hilfsklasse <code>Paths</code> verwendet. Innerhalb von Java-Programmen kann auch bei Windows-Systemen das Pfadtrennzeichen <code>/</code> statt <code>\</code> verwendet werden.

4	<p>Beim Erzeugen des <code>BufferedReader</code> Objektes wird versucht eine Verbindung zu der im <code>Path</code>-Parameter angegebenen Datei herzustellen. Hierbei können selbstverständlich Fehler auftreten, die zu einer Exception führen würden. Z.B. kann es sein, dass die angegebene Datei nicht existiert oder das Leserecht nicht gegeben ist.</p> <p>→ Eine Ausnahmebehandlung ist erforderlich!</p> <p>Seit Java SE 7 gibt es dafür eine elegante Möglichkeit in Form einer erweiterten <code>try</code>-Anweisung: "try with resources": alle hinter <code>try</code> in runden Klammern aufgeführten Ressourcen (in diesem Fall die Datei) werden automatisch geschlossen, sobald der <code>try</code>-Block abgearbeitet wurde, unabhängig davon, ob eine Exception erzeugt wurde oder nicht. (siehe Skript 24_OOP_Exceptions Kapitel 3.2.3)</p> <p>Um das <code>BufferedReader</code> Objekt zu erzeugen, wird hier kein Konstruktor verwendet, sondern die Klassenmethode <code>newBufferedReader</code> der Hilfsklasse <code>Files</code>. Dies hat den Vorteil, dass alle weiteren notwendigen Objekte für den Dateizugriff (<code>InputStreamReader</code>, <code>FileInputStream</code>, s.o.) implizit mit erzeugt werden.</p> <p>Wichtig ist die Angabe der korrekten Dateikodierung der zu lesenden Textdatei.</p>
5	<p>Mit Hilfe des <code>BufferedReader</code> Objektes (hier <code>reader</code>) kann nun mit der Methode <code>read()</code> sequentiell das jeweils nächste Zeichen gelesen werden. Ist das Dateiende erreicht, so liefert <code>read()</code> statt einem Zeichen den Integer Wert -1.</p>
6	<p>Wurde als letztes ein gültiges Zeichen gelesen, d.h. <code>iZeichen</code> hat einen Wert ungleich -1, so ist offensichtlich das Dateiende noch nicht erreicht.</p>
7	<p>Für die Ausgabe auf dem Bildschirm wird <code>iZeichen</code> in den Datentyp <code>char</code> umgewandelt, damit das entsprechende Zeichen ausgegeben wird, statt der Zahl.</p>

2.2 Methoden für `Path` Objekte

Für `Path` Objekte stehen eine Reihe von Methoden für häufig benötigte Auswertungen zur Verfügung.

Beispiel:

```
Path p = Paths.get("C:/Windows/Fonts/Datei.xxx");
System.out.println(p.toString()); // C:\Windows\Fonts\Datei.xxx
System.out.println(p.isAbsolute()); // true
System.out.println(p.getRoot()); // C:\
System.out.println(p.getParent()); // C:\Windows\Fonts
System.out.println(p.getNameCount()); // 3
System.out.println(p.getName(p.getNameCount() - 1)); // Datei.xxx
System.out.println(p.getFileName()); // Datei.xxx
```

Möchte man feststellen, in welchem aktuellen Verzeichnis das laufende Programm „sich befindet“, so kann man das mit folgendem Befehl tun:

```
System.out.println(Paths.get(".").toRealPath());
```

2.3 Zeilenweises Lesen von Textdateien

Textdateien sind im Allgemeinen in Zeilen strukturiert; als Zeilen-Endekennung dienen die Zeichen: '\r'\n' (bei Windows Systemen 2 Zeichen), '\n' (bei Unix/Linux Systemen 1 Zeichen).

Zum Lesen einer kompletten Zeile wird die Methode `readLine()` verwendet.

```
public class LeseZeilenweise
{
    static void leseDatei()
    {
        Path dateiPfad = null;
        String eineZeile;
        String[] zeilenTeile;

        dateiPfad = Paths.get("./Dateien/namen.txt");

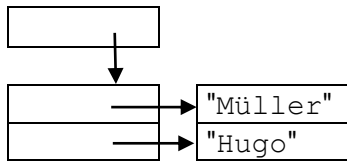
        try (BufferedReader reader = Files.newBufferedReader(dateiPfad,
            StandardCharsets.UTF_8))
        {
1       eineZeile = reader.readLine(); // liest die nächste Zeile
2       while (eineZeile != null)
        {
            // Die folgende Anweisung trennt die Bestandteile der Zeile
            // und speichert die Teile in einen String Array
3       zeilenTeile = eineZeile.split("[ \t]");

            System.out.printf("Name = %s, Vorname = %s\n",
4       zeilenTeile[0], zeilenTeile[1]);

            eineZeile = reader.readLine(); // liest die nächste Zeile
        }
    }
    catch (Exception oe)
    {
        System.out.println("Fehler beim Öffnen der Datei: " +
            oe.getMessage());
    }
}

/**
 * @param args
 * Kurzbeschreibung:
 */
public static void main(String[] args)
{
    leseDatei();
}
```


Anmerkungen:

	Beim Erzeugen des <code>BufferedReader</code> Objektes für den prinzipiellen Zugriff auf die Datei ändert sich nichts gegenüber 2.1
1	Die Methode <code>readLine()</code> liest eine komplette Zeile inklusive der Zeichen für die Zeilen-Endekennung (s.o.). Der Rückgabewert (<code>String</code>) enthält die Textzeile aus der Datei ohne die Zeichen für die Zeilen-Endekennung. Enthält eine Datei eine Zeile nur eine Zeilen-Endekennung, so ist der Rückgabewert ein leerer String (<code>""</code>). Ist bereits das Dateiende erreicht und es wird trotzdem <code>readLine()</code> aufgerufen, so ist der Rückgabewert <code>null</code> .
2	Wurde als letztes eine gültige Zeile gelesen, d.h. <code>eineZeile</code> hat einen Wert ungleich <code>null</code> , so ist offensichtlich das Dateiende noch nicht erreicht.
3	<p>Besteht eine Textzeile aus mehreren Teilen (z.B. Wörtern oder Spalten), die durch ein Trennzeichen voneinander getrennt werden (Bsp. CSV-Dateien: <u>C</u>haracter <u>S</u>eparated <u>V</u>alues), so könnte man mit dem uns bekannten <code>StringTokenizer</code> arbeiten.</p> <p>Seit Java 4 wird allerdings die Verwendung der Objektmethode <code>split</code> der Klasse <code>String</code> empfohlen: <code>public String[] split(String regex)</code></p> <p>Die Trennung der Spalten einer Zeile kann durch ein Zeichen, durch verschiedene alternative Zeichen oder eine Zeichenfolge definiert sein. Mit dem Parameter <code>regex</code> kann man mit Hilfe sogenannter „regulärer Ausdrücke“³ (regular expressions) sehr flexibel die Trennzeichenfolge spezifizieren.</p> <p>Im Bsp. hat der reguläre Ausdruck folgende Syntax <code>[\t]</code></p> <ul style="list-style-type: none"> - Die Klammern bedeuten, dass es sich um eine Menge von alternative Zeichen handelt - Das Trennzeichen kann ein Leerzeichen oder ein Tabulatorzeichen sein. <p>Würden mehrere Leer- oder Tabulatorzeichen hintereinander zwischen zwei Spalten stehen, so hätte dies zur Folge, dass leere Spalten entstehen. Möchte man in diesem Fall leere Spalten als Ergebnis verhindern, könnte man folgenden regulären Ausdruck verwenden <code>[\t]+</code>, das bedeutet ein Trennzeichenfolge besteht aus einem oder mehreren Leer- oder Tabulatorzeichen.</p> <p>Das Ergebnis der <code>split</code>-Methode ist ein <code>String</code>-Array, im Bsp.:</p> <pre>eineZeile sei "Müller Hugo" zeilenTeile</pre> 
4	Über das <code>String</code> -Array <code>zeilenTeile</code> kann jetzt auf die einzelnen Spalten der Zeile, z.B. <code>zeilenTeile[0]</code> , zugegriffen werden.

³ Reguläre Ausdrücke sind eine sehr mächtiges und flexibles Mittel um Zeichenmuster (patterns) zu beschreiben und werden aus diesem Grund vielfältig angewendet, z.B. auch zur Überprüfung der formalen Korrektheit von Benutzereingaben. Siehe auch <https://docs.oracle.com/javase/tutorial/essential/regex/index.html>

2.4 Schreiben von Textdateien

Zum Schreiben von Textdateien kann die Klasse `BufferedWriter` verwendet werden. Prinzipiell kann das `BufferedWriter` Objekt in gleicher Weise erzeugt werden, wie ein `BufferedReader` Objekt (s.o. 2.1). Die `newBufferedWriter` Methode der Klasse `Files` besitzt allerdings noch weitere Parameter:

```
public static BufferedWriter newBufferedWriter(Path path, Charset cs,
                                             OpenOptions ... options) throws IOException
```

Die wichtigsten `StandardOpenOptions` sind in folgender Tabelle beschrieben:

Option	Bedeutung
APPEND	Die Datei wird zum Schreiben geöffnet. Neue Zeichen werden am Ende der Datei angefügt.
CREATE	Die Datei wird erzeugt, wenn sie noch nicht existiert.
TRUNCATE_EXISTING	Falls die Datei bereits existiert und zum Schreiben geöffnet wird, dann wird ihr Inhalt gelöscht. Die Option wird ignoriert, falls die Datei zum Lesen geöffnet wird.
CREATE_NEW	Die Datei wird erzeugt, wenn sie noch nicht existiert. Ist sie schon vorhanden, dann wird eine <code>IOException</code> ausgelöst.
WRITE	Wenn die Datei bereits existiert, so wird ihr Inhalt nicht gelöscht, aber beginnend vom Dateianfang überschrieben. Ist sie nicht vorhanden, dann wird eine <code>IOException</code> ausgelöst.

Beispiel

```
static void schreibeInDatei()
{
    Path dateiPfad = null;
    String eineZeile;

    dateiPfad = Paths.get("../Dateien/text.txt");

    try (BufferedWriter writer = Files.newBufferedWriter(dateiPfad,
        StandardCharsets.UTF_8, StandardOpenOption.CREATE,
        StandardOpenOption.TRUNCATE_EXISTING))
    {
        eineZeile = Eingabe.getString("Textzeile:");
        while (eineZeile != null)
        1   {
        2       writer.write(eineZeile); // schreibt den Text in die Datei
           writer.newLine();           // Fügt eine Zeilenendekennung ein
           eineZeile = Eingabe.getString("Textzeile:");
        3   }
        catch (IOException oe)
        {
        4   System.out.println("Fehler beim Öffnen: " + oe.getMessage());
        }
        catch (Exception oe)
        {
            System.out.println("Fehler beim Zugriff: " + oe.getMessage());
        }
    }
}
```

3 Lesen und Schreiben von serialisierten Objekten

3.1 Begriffsbestimmung

Unter Persistenz versteht man das dauerhafte, nicht-flüchtige Speichern von Daten auf einem Datenträger, sodass die Daten auch nach dem Beenden des Programms verfügbar sind.

Übliche Datenträger sind Dateisysteme oder Datenbanken.

Persistenz ist also prinzipiell unabhängig von einer bestimmten Technologie.

Serialisierung bezeichnet die Überführung eines Java-Objekts in eine sequenzielle Darstellungsform. Die Serialisierung ist vorallem deshalb notwendig, da Objekte im Allgemeinen auch Verweise enthalten, die im Arbeitsspeicher in Form von Speicheradressen vorliegen. Die Speicheradressen verlieren aber bei einer Persistierung ihren Sinn, da beim späteren Lesen der persistierten Objekte diese Speicheradressen ihre Gültigkeit verloren haben. Nur in serialisierter Form können Objekte sinnvoll in eine Datei geschrieben und damit persistiert werden⁴ oder über ein Netzwerk transportiert werden.

In diesem Dokument wird die Serialisierung von Objekten zum Zwecke der persistenten Speicherung in Dateien dargelegt.

3.2 Serialisierung und Deserialisierung

3.2.1 Prinzip der Serialisierung

Durch den Vorgang der Serialisierung wird die JVM-interne Darstellung eines Objekts in einen externen Datenstrom (Stream) umgewandelt.

Das Objekt liegt also anschließend in 2 Formen vor: einmal als "normales" Objekt im Hauptspeicher und einmal in serialisierter Form auf einem externen Medium. Beide Formen existieren unabhängig voneinander, sodass sich nachträgliche Änderungen am Hauptspeicherobjekt nicht auf das serialisierte Objekt auswirken und umgekehrt ebenfalls nicht.

Folgende Daten werden in den Stream geschrieben:

- die Klasse des Objekts
- die Signatur der Klasse
- alle Attributs-Variablen des Objekts
- alle Attributsvariablen aller Oberklassen des Objekts, also alle geerbten Attributsvariablen

Folgende Daten werden *nicht* in den Stream geschrieben:

- alle Klassenvariablen (`static` - sie gehören nicht zum Objekt, sondern zur Klasse)
- alle transienten Attributsvariablen (`transient` - sie sind "flüchtig", ergeben sich z.B. zur Laufzeit dynamisch)
- alle Methoden / der Code (bleiben in der `.class`-Datei und müssen beim Deserialisieren über den Klassennamen wieder zugeordnet werden)

Um die Komplexität dieses Vorgangs anzudeuten sei Folgendes angemerkt:

- zur Laufzeit müssen alle Attributsvariablen des Objekts ermittelt werden, auch die geerbten
- alle Klassen- und transienten Attribute müssen aussortiert werden

⁴ Bei der Speicherung von Objekten in einem Datenbanksystem (DBS) ist ebenfalls eine Umstrukturierung der Objektdaten notwendig. Dies gilt vorallem auch für relationale DBS. Mit dem Einsatz von OO-Datenbanken wurde versucht den Umstrukturierungsaufwand, der sowohl beim Speichern als auch beim Lesen notwendig ist, zu minimieren.

- Attribute können selbst wieder Objekte sein, die nach diesen Regeln serialisiert werden müssen
- sämtliche Objektreferenzen müssen beim Einlesen der Objekte wieder rekonstruiert werden

3.2.2 Prinzip der Deserialisierung

Für sämtliche serialisierten Objekte muss auch die Umkehrung dieses Vorgangs möglich sein, also die Deserialisierung. Ein Ausgabestream muss wieder in ein Hauptspeicherobjekt zurückverwandelt werden können, damit das Programm ohne Informationsverlust damit weiterarbeiten kann.

Da im Ausgabestream keine Informationen zu den Methoden der Klasse vorhanden sind, muss beim Deserialisieren diese Information erneut zugeordnet werden. Dieser Vorgang ist komplex und soll hier nicht im Detail vorgestellt werden. Nur einige Hinweise dazu:

Zur Neubildung eines Objekts wird die Reflection-API von Java benutzt. Die Klassen `Class` und `Object` spielen dabei eine zentrale Rolle. Über diese lassen sich Eigenschaften von Klassen und Objekten bestimmen.

Das bedeutet, dass die JVM bei der Deserialisierung die Klasse kennen muss, deren Objekt sie wieder herstellen soll. Sollte dies nicht der Fall sein, so wird eine `ClassNotFoundException` geworfen, die im Programm abgefangen werden muss.

3.2.3 Versionierung

Als zusätzliche Problematik kommt hinzu, dass durch die Serialisierung die Daten eines Objekts von der zugehörigen Class-Datei getrennt werden. Bei der Deserialisierung müssen Attribut- und Methodendefinitionen aber wieder passgenau den serialisierten Daten zugeordnet werden, d.h. die zugehörige Klasse darf sich in der Zwischenzeit nicht verändert haben.

Diese Problematik versucht Java durch einen Versionierungsmechanismus zu lösen:

Das Interface `Serializable` enthält eine `long`-Konstante `serialVersionUID`. Diese wird beim Serialisieren von der Methode `writeObject` (s.u.) als Hashcode berechnet, der den Klassennamen, Signatur, Methoden und Konstruktoren mit einbezieht und diese in den Stream schreibt.

Bei der Deserialisierung wird diese VersionsID mit der aktuellen VersionsID der Klasse verglichen. Stimmen diese nicht überein, so wird die Deserialisierung mit einer `InvalidClassException` abgebrochen. Auf diese Weise können selbst minimale Veränderungen wie das Hinzufügen einer einfachen Methode die serialisierten Daten für alle Zeiten unbrauchbar machen, weil sie nicht mehr deserialisiert werden können.

Diesen impliziten Versionierungsmechanismus kann man umgehen, indem man die `serialVersionUID` selbst explizit festlegt. Zu diesem Zweck kann man die symbolische Konstante `static final long serialVersionUID` anlegen. Diese wird dann von der Methode `writeObject` benutzt. Allerdings ist der Programmierer dann selbst dafür verantwortlich für die Kompatibilität der Klasse mit den serialisierten Objekten zu sorgen.

3.3 Realisierung von Serialisierung und Deserialisierung in Java

3.3.1 Voraussetzungen

Ein Objekt, das serialisiert werden soll, muss auch serialisierbar sein: seine Klasse muss das Interface `Serializable` implementieren. `Serializable` ist ein sogenanntes Markierungsinterface, das keine zu implementierende Methode enthält.

Beispiel:

```
public class Auto implements Serializable
{
    // static final long serialVersionUID = 4711L;
    private int iPs;
    private String sMarke;
    ...
}
```

3.3.2 Serialisierung (Schreiben)

Zum Erzeugen eines Ausgabestreams dient die Klasse `ObjectOutputStream`. Ihr Konstruktor erwartet einen `OutputStream` als Parameter oder eine davon abgeleitete Klasse.

`ObjectOutputStream` kennt u.a. die Methode `public void writeObject (Object o)`, der ein beliebiges Objekt zur Serialisierung übergeben werden kann. `writeObject` erzeugt ggf. eine `IOException`, die abgefangen werden muss.

Für die Serialisierung von einfachen Datentypen existieren jeweils eigen Methoden, z.B. `writeInt (int i)`.

Beispiel:

```
public class SerialDemo
{
    public static void main(String[] args)
    {
        Auto einAuto = new Auto();
        einAuto.setMarke("BMW");
        einAuto.setPS(180);
        Path datei = Paths.get("./Dateien/daten.ser");
        try (ObjectOutputStream oos =
            new ObjectOutputStream (
                new BufferedOutputStream(
                    Files.newOutputStream(datei,
                        StandardOpenOption.CREATE,
                        StandardOpenOption.TRUNCATE_EXISTING)))
        {
            oos.writeObject(einAuto);
        }
        catch (IOException io)
        {
            System.out.println("Fehler: " + io.getMessage());
        }
    }
}
```

3.3.3 Deserialisierung (Lesen)

Zum Erzeugen eines Eingabestreams dient die Klasse `ObjectInputStream`. Ihr Konstruktor erwartet einen `InputStream` als Parameter oder eine davon abgeleitete Klasse.

`ObjectInputStream` kennt u.a. die Methode `public final Object readObject ()`, die aus dem `ObjectInputStream` liest. Neben einer `IOException` muss aus o.g. Gründen auch eine `ClassNotFoundException` abgefangen werden.

Beispiel:

```
public class SerialDemo
{
    public static void main(String[] args)
    {
        Auto a1;
        Path datei = Paths.get("./Dateien/daten.ser");

        try (ObjectInputStream ois =
            new ObjectInputStream (
                new BufferedInputStream(
                    Files.newInputStream(datei))))
        {
            a1 = (Auto)ois.readObject();

            System.out.println("A1: "+a1.getMarke());
        }
        catch (ClassNotFoundException cnf)
        {
            System.out.println("Fehler: " + cnf.getMessage());
        }
        catch (IOException io)
        {
            System.out.println("Fehler: " + io.getMessage());
        }
    }
}
```

Beispiel 2: Lesen bis zum Dateiende: Das Dateiende kann beim Lesen vom `ObjectInputStream` nur durch die `EOFException` erkannt werden

```
public class SerialDemo_v2
{
    public static void main(String[] args)
    {
        Auto einAuto;
        String sMarke;
        Path datei = Paths.get("./Dateien/daten.ser");
        try (ObjectOutputStream oos =
            new ObjectOutputStream (
                new BufferedOutputStream(
                    Files.newOutputStream(datei, StandardOpenOption.CREATE,
                                           StandardOpenOption.TRUNCATE_EXISTING))))
        {
        }
```

```
{
    sMarke = Eingabe.getString("Geben Sie eine Automarke ein: ");
    while (sMarke != null)
    {
        einAuto = new Auto();
        einAuto.setsMarke(sMarke);
        einAuto.setiPs(Eingabe.getInt("Leistung in PS = "));
        oos.writeObject(einAuto);
        sMarke = Eingabe.getString("Geben Sie eine Automarke ein: ");
    }
}
catch (IOException io)
{
    System.out.println("Fehler: " + io.getMessage());
}

Auto a1;

try (ObjectInputStream ois =
    new ObjectInputStream (
        new BufferedInputStream(
            Files.newInputStream(datei))))
{
    do
    {
        a1 = (Auto)ois.readObject();

        System.out.printf("Marke: %-10s PS: %4d \n", a1.getsMarke(), a1.getiPs());

    } while (true); // Schleife endet durch EOFException
}
catch (EOFException eof)
{
    System.out.println("Dateiende erreicht\n");
}
catch (IOException io)
{
    System.out.println("Fehler: " + io.getMessage());
}
catch (ClassNotFoundException cnf)
{
    System.out.println("Fehler: " + cnf.getMessage());
}
}
}
```