File-Zugriff in Java

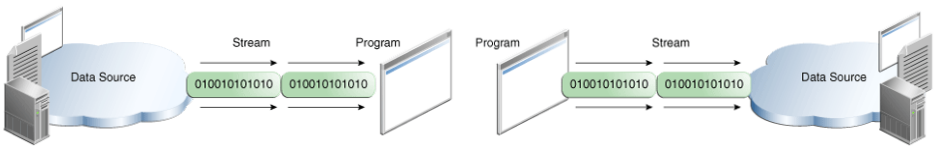
Wenn man in Java mit Files arbeiten möchte, gibt es dafür grundsätzlich zwei Möglichkeiten: `java.io` („Input/Output“) oder `java.nio` („Non-Blocking Input/Output“).

Beim `java.io`-Package handelt es sich um die ältere und klassische Variante, um mit lokalen Files zu arbeiten. Mit Java SE 7 wurde `java.nio` eingeführt, welches neue Konzepte wie den Zugriff auf entfernte Dateien, nicht-blockierendes Lesen und Schreiben und Buffer-orientierten Zugriff einführte. Dadurch werden jedoch einerseits viele Dinge verkompliziert (asynchrones Lesen und Schreiben), während andererseits manche Aspekte wiederum so simplifiziert werden, dass man kaum merkt, was im Hintergrund alles passiert. Nachdem wir außerdem mit vielen Konzepten von `java.io` wie Streams in Kontakt kommen werden, beschäftigen wir uns daher im Rahmen der Grundkompetenzen mit dem klassischen Filezugriff über `java.io`.

Streams

Input- und Output-Streams

Ein zentrales Konzept von java.io sind Streams. Ein Stream ist eine Sequenz von Daten. Es wird zwischen Input-Streams und Output-Streams unterschieden. Ein **Input-Stream** wird von Java-Programmen verwendet, um Daten von einer Quelle (z.B. ein File) nacheinander **einzulesen**. Mit einem **Output-Stream** kann ein Java-Programm Daten in ein Ziel schreiben. Folgende Grafiken[1] veranschaulichen das Konzept eines Input-Streams (links) und eines Output-Streams (rechts):



Es können unterschiedliche Daten über Streams gelesen und geschrieben werden, beispielsweise einzelne Bytes, Text (Strings) und sogar ganze Java-Objekte (Stichwort „Serialisierung“).

[1] Siehe https://docs.oracle.com/javase/tutorial/essential/io/streams.html

Abgrenzung zu java.util.stream

Wir sprechen hier von Streams aus `java.io`. Der Begriff Stream ist in Java leider doppelt belegt: es gibt zusätzlich das Package `java.util.stream`[2], welches den Zugriff auf Collections über funktionale Sprachelemente ermöglicht. `java.nio` lässt sich gut über die Funktionen von `java.util.stream` nutzen. Folgendes Beispiel listet alle Dateien und Ordner des aktuellen Working Directorys:

Beispiel aus java.nio und java.util.stream (kein Stoff für GK)

```
public static void main(String[] args) throws IOException {
	Files.list(Paths.get("./")).forEach(System.out::println);
}
```

Die Klasse `Files` aus `java.nio` bietet Hilfsmethoden für den Zugriff aufs Dateisystem, während `Paths` den Umgang mit Pfadangaben erleichtert. `Files.list` liefert eine Menge an Pfadangaben, die über `forEach` iteriert und ausgegeben werden – alles in einer Zeile Code.

Schreiben

Erweitern wir unsere ToDo-Liste um eine Speichern-Funktion:

Schreiben mit einem FileWriter

```
public class ToDoListe {
	private String name;
	private ToDoEintrag[] eintraege;

	public ToDoListe(String name) {
		this.name = name;
		this.eintraege = new ToDoEintrag[0];
	}

	public void addEintrag(String text) {
		eintraege = Arrays.copyOf(eintraege, eintraege.length + 1);
		eintraege[eintraege.length - 1] = new ToDoEintrag(text);
	}

	public void speichern(String filename) throws IOException {
		File f = new File(filename);
		FileWriter outputStream = null;
		try {
			outputStream = new FileWriter(f);
			for (int i = 0; i < eintraege.length; ++i) {
				outputStream.write(eintraege[i].getText());
				outputStream.write(System.lineSeparator());
			}
		} finally {
			if (outputStream != null)
				outputStream.close();
		}
	}

}
```

[2] Siehe https://docs.oracle.com/javase/8/docs/api/java/util/stream/package-summary.html

Die Klasse File

Die Klasse `File`[3] erlaubt die Verwaltung von Dateien. Zu beachten ist, dass beim Erzeugen eines `File`-Objektes über `new File()` **noch keine** Datei am Computer erzeugt wird. Die Klasse `File` stellt lediglich eine abstrakte Repräsentation dar, d.h. die abgebildete Datei muss (noch) gar nicht existieren. Über `exists()` kann geprüft werden, ob die Datei unter dem angegebenen Pfad überhaupt existiert. Außerdem können auch Ordner über die Klasse `File` repräsentiert werden: Über `isDirectory()` kann geprüft werden, ob es sich dabei um einen Ordner handelt. Mit `listFiles()` können alle Dateien und Sub-Ordner aufgelistet werden, die sich unter dem angegebenen Pfad befinden.

Die Klasse FileWriter

Ein `FileWriter`[4] ermöglicht das Schreiben von Zeichen (Characters). Es gibt auch andere Klassen, die das Schreiben auf Byte-Ebene ermöglichen (`FileOutputStream`[5]). Über die `write()`-Methode können also Strings geschrieben werden, wobei man beachten muss, dass der `FileWriter` standardmäßig keinen Zeilenumbruch anhängt. Daher wird in diesem Beispiel mithilfe von `System.lineSeparator()` ein Zeilenumbruch hinzugefügt.

close() und try-catch-finally

Nachdem der Konstruktor von `FileWriter` sowie die `write()`-Methode eine `IOException` werfen können, wird ein `try-catch`-Block verwendet. Streams sollten nämlich **immer** über `close()` geschlossen werden – **auch im Fehlerfall**. Daher macht man dies am besten im (optionalen) `finally`-Teil eines `try`-catch-Blockes: Statements im `finally`-Block werden immer ausgeführt, auch wenn der `try`-Block über eine Exception verlassen wurde. Nachdem `close()` wieder eine `IOException` werfen kann, wird auf einen `catch`-Block verzichtet und die Methode wird als `throws IOException` deklariert. Man könnte jedoch an dieser Stelle auch alle Exceptions abfangen und die Fehlerbehandlung direkt in der `speichern()`-Methode durchführen. Eine weitere, elegantere Möglichkeit sind try-with-resources-Statements, die etwas später vorgestellt werden.

---

[3] Siehe https://docs.oracle.com/javase/8/docs/api/java/io/File.html
[4] Siehe https://docs.oracle.com/javase/8/docs/api/java/io/FileWriter.html
[5] Siehe https://docs.oracle.com/javase/8/docs/api/java/io/FileOutputStream.html

Über Pufferspeicher (Buffers) kann der Schreibvorgang etwas performanter gestaltet werden:

Die Klasse BufferedWriter

```
public void speichern(String filename) throws IOException {
	File f = new File(filename);
	BufferedWriter outputStream = null;
	try {
		outputStream = new BufferedWriter(new FileWriter(f));
		for (int i = 0; i < eintraege.length; ++i) {
			outputStream.write(eintraege[i].getText());
			outputStream.write(System.lineSeparator());
		}
	} finally {
		if (outputStream != null)
			outputStream.close();
	}
}
```

Zwischenspeicher und flush()

Der `BufferedWriter`[6] „wrappt“ (erweitert) den `FileWriter`-Stream, sodass nicht jeder Aufruf der `write()`-Methode auch einen tatsächlichen Schreibvorgang auf die Festplatte (bzw. das jeweilige Speichermedium) durch das Betriebssystem auslöst. Schreibvorgänge sind in `java.io` immer blockierend. Das heißt, dass bei jedem Schreibvorgang darauf gewartet werden muss, bis dieser abgeschlossen ist („Blocking I/O“), was nicht sehr performant ist. Beim `BufferedWriter` wird zuerst in einen Zwischenspeicher geschrieben, der erst dann auf das Speichermedium geschrieben wird, wenn der Zwischenspeicher voll ist oder der Stream geschlossen wird. Über die Methode `flush()` kann der Zwischenspeicher auch manuell geschrieben und geleert werden.

Eine weitere Klasse, die häufig genutzt wird, ist `PrintWriter`[7]:

Die Klasse PrintWriter

```
public void speichern(String filename) throws IOException {
	File f = new File(filename);
	PrintWriter outputStream = null;
	try {
		outputStream = new PrintWriter(f);
		for (int i = 0; i < eintraege.length; ++i) {
			outputStream.println(eintraege[i].getText());
		}
	} finally {
		if (outputStream != null)
			outputStream.close();
	}
}
```

[6] Siehe https://docs.oracle.com/javase/8/docs/api/java/io/BufferedWriter.html
[7] Siehe https://docs.oracle.com/javase/8/docs/api/java/io/PrintWriter.html

Der `PrintWriter` erlaubt, ähnlich wie `System.out`, das **zeilenweise Schreiben** über die Methode `println`. Auch `PrintWriter` verwendet einen Zwischenspeicher zum Puffern. Einige Konstruktoren von `PrintWriter` erlauben das Setzen des Attributs `autoFlush`, wodurch nach jedem Schreibbefehl der Puffer geleert wird und auf das Speichermedium geschrieben wird. Der Code kann außerdem noch weiter verkürzt werden:

```
public void speichern(String filename) throws IOException {
	try (PrintWriter outputStream = new PrintWriter(filename)) {
		for (int i = 0; i < eintraege.length; ++i) {
			outputStream.println(eintraege[i].getText());
		}
	}
}
```

Viele Konstruktoren von Stream-Klassen akzeptieren auch direkt den Pfad zu einer Datei, sodass vorher kein `File`-Objekt erzeugt werden muss. Dazu gehört auch der `PrintWriter`. In diesem Beispiel erzeugt `PrintWriter` intern einen `BufferedWriter`.

Das try-with-resources-Statement

Das try-with-resources-Statement erlaubt vor dem `try`-Block noch zusätzlich die Initialisierung von Objekten, die über `close()` geschlossen werden müssen. Sie implementieren alle das Interface `AutoCloseable`[8] und werden **automatisch geschlossen** – auch im Fehlerfall. Dadurch spart man sich den `finally`-Block.

Lesen

Äquivalent zum `FileWriter` und `BufferedWriter` gibt es auch für den Lesezugriff einen `FileReader`[9] und `BufferedReader`[10]. Nachdem der `FileReader` keinen direkten zeilenweisen Zugriff erlaubt, sehen wir uns gleich den `BufferedReader` an:

Die Klasse BufferedReader

```
public void laden(String filename) throws IOException {
	this.eintraege = new ToDoEintrag[0];
	try (BufferedReader inputStream = new BufferedReader(new
						FileReader(filename))) {
		String text;
		while ((text = inputStream.readLine()) != null) {
			addEintrag(text);
		}
	}
}
```

[8] Siehe https://docs.oracle.com/javase/8/docs/api/java/lang/AutoCloseable.html
[9] Siehe https://docs.oracle.com/javase/8/docs/api/java/io/FileReader.html
[10] Siehe https://docs.oracle.com/javase/8/docs/api/java/io/BufferedReader.html

Zeilenweises Einlesen

Damit die Elemente bei mehrfachen Ladevorgängen nicht verdoppelt werden, wird zuerst das Array geleert. Wir haben hier gleich ein try-with-resources-Statement verwendet, damit zum Schluss automatisch die `close()`-Methode von `inputStream` aufgerufen wird. Der `BufferedReader` ist wieder eine Erweiterung von `FileReader`, der einen Pufferspeicher zwischenschaltet. Dadurch löst nicht jede Leseoperation automatisch einen kostspieligen Lesezugriff am Speichermedium aus. Die `readLine()`-Methode liest eine Zeile und liefert diese zurück. Wurde das Ende der Datei erreicht, wird `null` zurückgeliefert. Daher findet man häufig dieses Konstrukt über eine `while`-Schleife, die eine Zeile einliest und gleich auf `null` prüft.

Eine weitere Klasse, die oft Zum Lesen genutzt wird, ist `Scanner`[11]:

Die Klasse Scanner

```
public void laden(String filename) throws IOException {
	try (Scanner s = new Scanner(new BufferedReader(new
                                                FileReader(filename)))) {
		while (s.hasNext()) {
			addEintrag(s.next());
		}
	}
}
```

Ein `Scanner` teilt eingelesene Daten anhand von definierbaren Zeichen auf. Standardmäßig gelten alle **Whitespaces** (Zeilenumbrüche, Leerzeichen, Tabulator) als Trennzeichen. Dies kann über die Methode `useDelimiter()` geändert werden. Außerdem können nicht nur Texte, sondern auch andere Datentypen eingelesen, beispielsweise über `nextDouble()`.

Serialisierung von Objekten

Serialisierung und Deserialisierung von Objekten

Man kann in Java nicht nur Bytes, Texte und primitive Datentypen abspeichern und laden, sondern auch ganze Java-Objekte. Das Umwandeln von Java-Objekten in eine Byte-Abfolge nennt sich **Serialisierung**. Der umgekehrte Prozess, also das Umwandeln von Bytes in Java-Objekte, heißt **Deserialisierung**.

Das Serializable-Interface

Die meisten Java-Objekte können serialisiert werden. Um zu kennzeichnen, dass sie serialisiert werden können, implementiert man das `Serializable`-Interface[12]. Wir lernen Interfaces erst später kennen. Grundsätzlich können Interfaces Methoden vorschreiben, die

[11] Siehe https://docs.oracle.com/javase/8/docs/api/java/util/Scanner.html
[12] Siehe https://docs.oracle.com/javase/8/docs/api/java/io/Serializable.html

implementiert werden müssen. Das `Serializable`-Interface besitzt jedoch keine Methoden. Daher ist es ein sogenanntes **Marker-Interface** (also ein Interface ohne Methoden). Dass nicht alle Java-Objekte automatisch auch serialisiert werden können, sieht man an der Klasse `Object`[13]: Diese implementiert nämlich nicht das `Serializable`-Interface. Daher muss man eigene Klassen mit diesem Interface markieren, wenn man sie serialisieren möchte.

serialVersionUID

Implementiert man selbst das `Serializable`-Interface (`public class ABC implements Serializable`), bekommt man typischerweise eine Warnung, dass man ein statisches, finales Attribut `serialVersionUID` definieren sollte, welches man in Eclipse automatisch generieren lassen kann. Dieses Attribut dient zur Unterscheidung **verschiedener Versionen** dieser Klasse.

Folgendes Beispiel zeigt, wie eine `ToDoListe` über einen `ObjectOutputStream`[14] serialisiert werden kann:

Die Klasse ObjectOutput-Stream

```
ToDoListe t = new ToDoListe("Hallo Welt");
t.addEintrag("Einkaufen");
t.addEintrag("Hausuebung machen");
try (ObjectOutputStream outputStream = new ObjectOutputStream(new
FileOutputStream("ToDo.obj"))) {
	outputStream.writeObject(t);
} catch (IOException e) {
	System.err.println("Fehler beim Schreiben: " + e.toString());
}
```

Wichtig ist, dass die Klasse `ToDoListe` sowie alle referenzierten Klassen (`ToDoEintrag`) das Interface `Serializable` implementieren. Wie man sieht, kann man auch ganze Arrays (`eintraege`) mit einem `ObjectOutputStream` schreiben.

Umgekehrt liest ein `ObjectInputStream`[15] Objekte und Arrays ein:

Die Klasse ObjectInputStream

```
ToDoListe t = null;
try (ObjectInputStream inputStream = new ObjectInputStream(new
                                        FileInputStream("ToDo.obj"))) {
	t = (ToDoListe) inputStream.readObject();
} catch (IOException e) {
	System.err.println("Fehler beim Laden: " + e.toString());
} catch (ClassNotFoundException e) {
	System.err.println("Fehler beim Laden: " + e.toString());
}
```

[13] Siehe https://docs.oracle.com/javase/8/docs/api/java/lang/Object.html
[14] Siehe https://docs.oracle.com/javase/8/docs/api/java/io/ObjectOutputStream.html
[15] Siehe https://docs.oracle.com/javase/8/docs/api/java/io/ObjectInputStream.html

Habe ich es verstanden?

- Ich kann das Konzept von Streams erklären
- Ich kann die Klasse `File` und ihre wichtigsten Methoden erklären
- Ich kann Dateien mit einem `FileWriter`, `BufferedWriter` und `PrintWriter` schreiben und kenne die wichtigsten Methoden
- Ich kann einen `BufferedReader` und `Scanner` verwenden, um Dateien einzulesen und kenne die wichtigsten Methoden
- Ich kann das Konzept von Pufferspeichern und dessen Vorteile erklären
- Ich kann den `finally`-Block von `try-catch`-Statements erklären
- Ich kann das try-with-resources-Statement in Java verwenden
- Ich kann erklären, was man unter Serialisierung und Deserialisierung versteht
- Ich kann das `Serializable`-Interface erklären
- Ich kann über einen `ObjectOutputStream` ganze Java-Objekte serialisieren
- Ich kann über einen `ObjectInputStream` ganze Java-Objekte deserialisieren