Aufgabe 7.1 : Textdatei einlesen und mit Zeilennummern ausgeben

In den Prärequisiten finden Sie die Klasse `PrintWithLineNumbers`. Ergänzen Sie das Programm so, dass es die angegebene Textdatei einliest und sie, mit Zeilennummern versehen, auf der Konsole wieder ausgibt. In dieser Aufgabe sollen Sie keine Arrays und auch keine Collections verwenden.

Aufgabe 7.2 : Rangieren im Sackbahnhof mit einem Logfile protokollieren

In den Prärequisiten finden Sie die Klasse `Sackbahnhof`, welche das Rangieren in einem Bahnhof mit drei Stumpengeleisen simuliert. Zu Beginn steht auf den Geleisen 1 und 2 je eine gemischte Zugskomposition mit Waggons des Typs A sowie B. Der Rangiervorgang verschiebt so lange Waggons, bis am Schluss die beiden Kompositionen nicht mehr gemischt, sondern einheitlich sind. Die Geleise werden mit drei Stacks simuliert.

Die Klasse zeigt aber unglücklicherweise nur das Endresultat ohne die dafür nötigen Spielzüge an. In dieser Aufgabe geht es darum, die 26 Züge in einer Text-Datei (als Logfile sozusagen) zu protokollieren, so dass Sie nach dem Ende des Programmes den Verlauf noch einmal in Ruhe anschauen können.

Beachten Sie dabei folgende Vorgaben und Tips:

- Das Logfile soll als Textdatei angelegt und bewirtschaftet werden. Verwenden Sie hierzu einen geeigneten Stream-Typ.
- Für jeden Spielzug soll eine eigene Zeile ausgegeben werden. Hier ein Beispiel für den Zug Nummer 15:

```
15 : verschiebe Waggon Typ B von Gleis 2 nach Gleis 3
```

- Geben Sie als Dateiname einen Namen ohne Pfad an, dann wird die Datei direkt im Eclipse-Projekt angelegt, und Sie können sie einfach finden. Wenn Sie in Eclipse das Projekt markieren und dann F5 drücken, wird die Datei in Projektbaum angezeigt, und Sie können sie mit dem Editor betrachten. Allerdings muss das Programm vorher mindestens einmal ausgeführt worden sein.
- Sie sollen die Datei am Ende des Programmes zum Lesen öffnen und den Inhalt auf der Konsole zur Kontrolle anzeigen lassen.
- Es ist am einfachsten, wenn Sie die Parameterliste von `rangieren()` um einen Parameter des gewählten Stream-Typs `BufferedWriter` erweitern, so dass Sie den Zugriff auf die Datei in der Funktion zur Verfügung haben und direkt Einträge hinzufügen können.

Aufgabe 7.3 : Verzeichnisbaum anzeigen

In den Prärequisiten finden Sie die Klasse `DirectoryTree`. Ziel dieses Programmes ist es, einen Verzeichnisbaum auf der Konsole anzuzeigen. Ergänzen Sie das Programm so, dass es folgende beiden Voraussetzungen erfüllt:

- Die Methode `printDirectoryTree(Path, ...)` soll den Baum zum übergebenen Pfad (Ordner) anzeigen.
- Das anzuzeigende Verzeichnis soll aus dem `main()`-Parameter `args[0]` übernommen werden.

Tipp: Je nach verwendeter IDE müssen Sie das Argument für `main()` in Anführungszeichen einschliessen, da sonst die einzelnen Elemente des Pfads als mehrere Argumente interpretiert werden.

Aufgabe 7.4 : PDF-Checker

Schreiben Sie ein Programm `PdfChecker`, welches den User nach einem Pfad zu einer Datei fragt und bestimmt, ob diese Datei eine PDF-(kompatible) Datei ist. Achten Sie darauf, dass Sie alle Ressourcen korrekt schliessen, und implementieren Sie User-freundliches Exception-Handling.

Identifizieren Sie PDF-Dateien nach folgenden Kriterien:

a) Anhand der Dateiendung `.pdf`

b) Anhand der ersten 5 Bytes der Datei (Signatur). PDF-kompatible Dateien beginnen mit:
`0x25 0x50 0x44 0x46 0x2D` (als ASCII-String: «`%PDF-`»)

Siehe auch: https://en.wikipedia.org/wiki/List_of_file_signatures

Aufgabe 7.5 : Binärdaten haltbar machen

Es gibt Situationen, in welchen man lange Reihen von Messwerten erfassen und in einem Array speichern möchte, um sie später statistisch auszuwerten. Die Klasse `PersistentDoubles` (*auf dem AD*) tut folgendes:

- Es erlaubt die Eingabe einer beliebigen Menge von `double`-Werten und speichert sie in einem Array ab.
- Falls der Array zu klein wird, sorgt die Funktion `enlarge()` dafür, dass ein grösseres Array angelegt und die bisherigen Werte hinüberkopiert werden.
- Um das Programm abzubrechen, wird ein kleiner Trick verwendet (es ist nämlich sehr lästig, wenn es nach jeder Zahl fragt, ob man eine weitere eingeben möchte): Sie wissen vielleicht, dass die Methode `readDouble()` der Klasse `Terminal` eine `NumberFormatException` wirft, wenn Sie etwas eingeben, das nicht als Zahl interpretiert werden kann. Im Programm wird diese Ausnahme abgefangen und das Programm beendet. Sie müssen also lediglich einen Buchstaben tippen, um das Programm zu beenden.

Das Programm hat einen Nachteil: Sobald Sie es abbrechen, sind die Daten verloren. Und stellen Sie sich vor, Sie müssten Hunderte von Messwerten eingeben. Diese Arbeit würden Sie vielleicht auf mehrere Tage verteilen. Also wäre es sinnvoll, man könnte zwischenzeitlich den Rechner abschalten, ohne dass die bisher eingegebenen Werte verloren gehen.

In dieser Aufgabe geht es darum, das gegebene Programm so zu erweitern, dass die bisher eingegebenen Daten in einer Binärdatei gespeichert werden. Konkret soll wie folgt vorgegangen werden: Es seien bisher N Zahlen eingegeben worden. In der Datei `double.dat` sollen nun $N+1$ Zahlen gespeichert werden: Die erste Zahl (`int`) gibt an, wieviele Messwerte erfasst wurden. Darauf sollen N `double`-Werte folgen.

Erweitern Sie das Programm wie folgt:

- *Beim Start des Programmes* wird geprüft, ob die Datei `doubles.dat` vorhanden ist. Ist dies nicht der Fall, wird ein Array beliebiger Grösse angelegt, und die Eingabe der Daten kann beginnen. Existiert die Datei, wird ein Array passender Grösse angelegt, und die Werte werden aus der Datei eingelesen. Die Eingabe beginnt nun nicht am Anfang, sondern beim ersten, noch nicht eingegebenen Wert.
- *Beim Beenden des Programmes* werden alle Daten auf die Datei geschrieben, wobei die erste Zahl die bisherige Länge der Datenreihe repräsentiert. Beachten Sie, dass Sie jetzt die ganze Reihe neu auf die Datei schreiben müssen.

Wenn Sie die Erweiterungen erfolgreich angebracht haben, können Sie die Messreihe in beliebigen Portionen eingeben und das Programm zwischenzeitlich beenden. Die Datei ist nicht mit einem Editor lesbar, da die Werte binär vorliegen. Wenn Sie N Werte eingegeben haben, sollte die Datei $4 + 8N$ Bytes umfassen. Das sollten Sie nachkontrollieren.

Sie können das Programm mit einer zusätzlichen Kontrolle ausstatten, indem Sie bei einem Neustart die bisherigen Werte anzeigen oder deren Durchschnitt berechnen.

Aufgabe 7.6 : I/O-Tasks

Auf dem AD finden Sie die Klassen `IOTasks` und `Person`. `IOTasks` enthält 4 Methoden, die Sie implementieren sollen. Die Aufgabenstellungen finden dort direkt im Java-Quelltext. Zu allen Teilaufgaben stehen Tests zur Verfügung.

Aufgabe 7.7 : Audiodateien vom Typ WAV analysieren *(advanced)*

In dieser Aufgabe geht es darum, WAV-Dateien zu analysieren. Hierbei handelt es sich um ein unkomprimiertes Audio-Format. Die (vereinfachte) Spezifikation finden Sie im Anhang auf der nächsten Seite.

Ihre Aufgabe besteht darin, eine Datei zu öffnen und auf der Konsole einen Output etwa der folgenden Art zu produzieren:

```
d:\Musik\Songs\SailAway.wav    :  existiert
RIFF-Chunk : Format ist OK
-------------------------------------------------
Audiofomat    : 1
Anzahl Kanäle : 2
Sampling-Rate : 48000/s
Quantisierung : 16 Bits
Spieldauer    : 5:54
-------------------------------------------------
```

Hinweise:

- WAV-Dateien enthalten Daten in einem Binärformat. Die enthalten ganzzahlige Datenwerte in Grössen von 32 bzw. 16 Bits. Die Datentypen `int` und `short` eignen sich also am besten zum Einlesen der nötigen Informationen.
- Zahlen sind im *little-endian-Format* gespeichert, also genau andersrum, als das ein `DataOutputStream` tun würde. Die Klassen `Integer` und `Short` bieten hier aber einfache Abhilfe: verwenden Sie die statischen Methode `reverseBytes()`, um die Bytes in für Sie nützliche Ordnung zu bringen.
- Für diese Aufgabe benötigen Sie nicht alle Daten des File-Headers. Nicht benötigte Informationen können Sie im Stream überspringen. Beispiel: `in.skipBytes(6)` überspringt die nächsten 6 Bytes und setzt den Zeiger auf das Byte danach.
- Die Spieldauer lässt sich wie folgt berechnen: Das zweite Feld des Daten-Chunks liefert Ihnen die Grösse g der Musikdaten in Bytes. Diesen Wert müssen Sie durch die Anzahl Kanäle, durch die Samplingrate und durch die Quantisierung (in Bytes!) dividieren, um die Spieldauer in Sekunden zu erhalten.
- *Zusatztipp:* Wenn Sie unsinnige oder scheinbar unsinnige Werte zu sehen bekommen, sollten Sie sich die gelesenen Daten erstmal in Hex anschauen. String bietet hierzu eine nützliche Formatierungsfunktion.

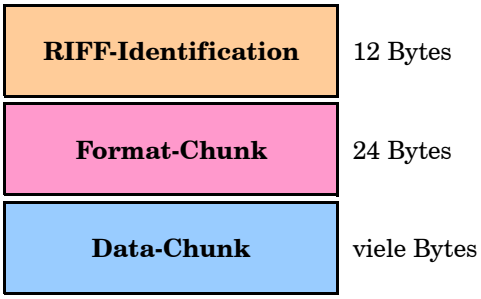
  Beispiel: `System.out.println(String.format("%08X", samplerate));`

Anhang (Auszug der WAV-Spezifikation)

Übersicht

In seiner einfachsten Form besteht eine **WAV**-Datei aus drei Chunks. Sie kann aber bis zu 12 Chunks enthalten, wenn es sich um eine spezielle erweiterte Variante handelt (komprimierte Daten, Definition von Einsprungpunkten, Playlists, MIDI-Informationen, usw).

Wir beschränken uns auf die Basis-Definition, welche von allen Programmen verstanden wird: Als Einträge in den Chunks kommen Zahlen oder Zeichenketten in Frage.



RIFF-Chunk

Identifikation als **RIFF**-Format:

Zwei der Felder enthalten Textkonstanten, welche die Datei als **WAV** identifizieren. Das mittlere Feld enthält die Dateigrösse, wobei die ersten beiden Felder nicht mitgezählt werden. Der Wert ist also um 8 Bytes zu klein. Ein typischer **RIFF**-Chunk sieht zum Beispiel so aus:

RIFF Chunk-ID	4	`"RIFF"`
Grösse der Datei	4	`size - 8`
Dateiformat	4	`"WAVE"`

```
52 49 46 46 43 8E 09 00 57 41 56 45
```

Das mittlere Feld ist als `00098E43` zu lesen und ergibt umgerechnet 626243 oder etwa 612k.

Format-Chunk

Alle wichtigen Einstellungen wie beispielsweise Sampling-Rate oder Sample-Format können im Format-Chunk eingesehen werden:

Für unkomprimierte Dateien (Normalfall) beträgt die Länge dieses Chunks immer 24 Bytes (da die ersten beiden Felder nicht gezählt werden, findet man hier normalerweise den Wert 10_H. Beispiel für ein Format-Chunk:

```
66 6D 74 20'10 00 00 00'01 00 02 00
44 AC 00 00 10 B1 02 00 04 00 10 00
```

Im ersten Teil versteckt sich die Zeichenkette `"fmt "`. Die folgenden Zahlen liegen alle im *little-endian* Format vor: Die Sampling-Rate beispielsweise ist als `0000AC44` zu lesen und ergibt, nach dezimal umgerechnet 44100.

Format Chunk ID	4	`"fmt "`
Chunk-Grösse	4	`16`
Audioformat	2	`1 (normal)`
Anzahl Kanäle	2	`mono,stereo`
Samplingrate	4	`samples/s`
Byte-Rate	4	`bytes/s`
Block-Ausrichtung	2	`bytes/frame`
Bits / Sample	2	`Quantisierun`

Die Werte *Byte-Rate* und *Block-Ausrichtung* ergeben sich aus der *Sampling-Rate*, der *Quantisierung* (Bits pro Sample) und der *Anzahl Kanäle*.

Daten-Chunk

Der Daten-Chunk besteht neben dem Header ausschliesslich aus kanalweise verschränkten Audio-Rohdaten.

Im ersten Feld befindet sich der Header für den Chunk, im zweiten Feld steht die Grösse der Rohdaten in *Bytes*. Für Stereo-Daten in 16 Bit Auflösung (Quantisierung) sind 4 Bytes pro Sample vorhanden.

Data Chunk-ID	4	`"data"`
Chunk-Grösse	4	`data-size`
Audio-Daten	s	`interleaved`