

# **Praktische Informatik 1**

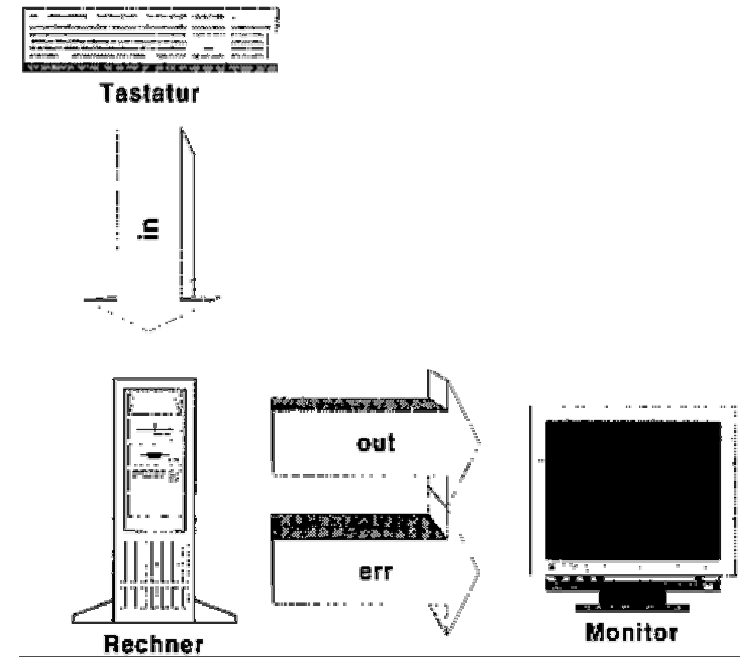
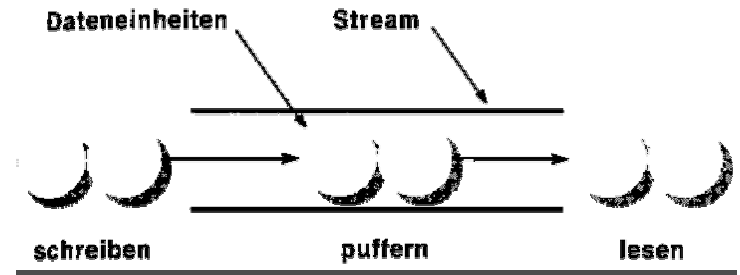
## **Prozeduren, Funktionen, JavaDoc, Datenströme**

Thomas Röfer

- Unterprogramme, Prozeduren und Funktionen
- Formale Parameter
- Überladen von Funktionen
- Rekursion
- Dokumentieren von Quelltexten
- Datenströme, Dateisystem
- Musterlösungen, Übungsblatt

# Datenströme

- Datenströme leiten Daten seriell in ein Programm hinein bzw. wieder heraus
- Sie können umgeleitet werden. So kann die Eingabe aus einer Datei kommen bzw. die Ausgabe in eine Datei geschrieben werden
- Standardeingabe
  - System.in (Instanz von InputStream)
- Standardausgabe
  - System.out (Instanz von PrintStream)
- Standardfehlerausgabe
  - System.err (Instanz von PrintStream)



# Byte-Datenströme in Java

- *InputStream* (Basisklasse)
  - *FileInputStream*, *ByteArrayInputStream*, *FilterInputStream*, *PushbackInputStream* ...
  - Typische Methoden
    - Lesen: *int read()*, *int read(byte[] b)*, *int read(byte[] b, int off, int len)*
    - Überspringen von Daten: *void skip(long n)*
    - Schließen des Datenstroms: *close()*
- *OutputStream* (Basisklasse)
  - *FileOutputStream*, *ByteArrayOutputStream*, *FilterOutputStream*, *PrintStream* ...
  - Typische Methoden
    - Schreiben: *void write(int b)*, *void write(byte[] b)*, *void write(byte[] b, int off, int len)*
    - Wegschreiben der Daten: *flush()*
    - Schließen des Datenstroms: *close()*

# Unicode-Datenströme in Java

- *Reader* (Basisklasse)
  - *BufferedReader*, *FilterReader*, *InputStreamReader*, *StringReader* ...
  - Typische Methoden
    - Lesen: *int read()*, *int read(char[] c)*, *int read(char[] c, int off, int len)*
    - Überspringen von Zeichen: *void skip(long n)*
    - Schließen des Datenstroms: *close()*
- *Writer* (Basisklasse)
  - *BufferedWriter*, *FilterWriter*, *OutputStreamWriter*, *PrintWriter*, *StringWriter* ...
  - Typische Methoden
    - Schreiben: *void write(int c)*, *void write(char[] c)*, *void write(char[] c, int off, int len)*, *write(String s)*, *write(String s, int off, int len)*
    - Wegschreiben der Zeichen: *flush()*
    - Schließen des Datenstroms: *close()*

# Datenströme in Java

- Methode *int read()*
  - Liefert das nächste gelesene Byte/Zeichen zurück
  - oder -1, wenn das Ende des Datenstroms erreicht wurde (*End Of File/EOF*)
  - Kann eine *IOException* erzeugen, wenn Fehler beim Lesen auftreten
- Methode *String readLine()* (*BufferedReader*)
  - Liefert eine gesamte Textzeile zurück
  - oder *null*, wenn das Ende des Datenstroms erreicht wurde
  - Kann eine *IOException* erzeugen, wenn Fehler beim Lesen auftreten
- Paket *java.io*
  - Datenstrom-Klassen befinden sich im Paket *java.io* (z.B. *java.io.FileInputStream*)
  - *java.io.* vor den Klassennamen kann eingespart werden, wenn der Quelltext eine Importanweisung am Anfang enthält: *import java.io.\**

# Beispiel 1

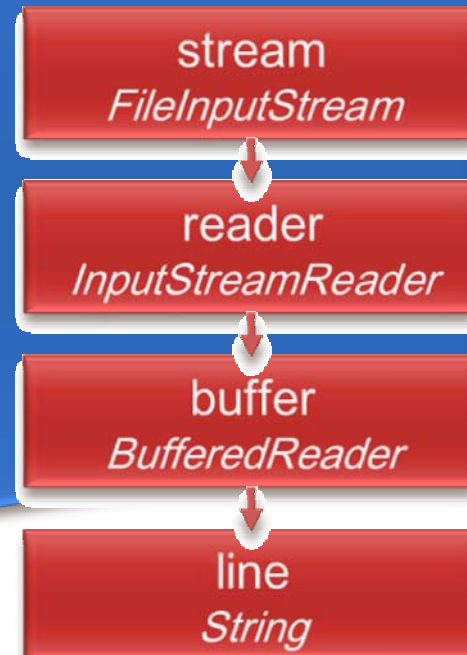
```
import java.io.*;

class Streams {
    static int length(String fileName)
        throws FileNotFoundException, IOException {
        FileInputStream stream = new FileInputStream(fileName);
        int chars = 0;
        int c = stream.read();
        while (c != -1) {
            ++chars;
            c = stream.read();
        }
        return chars;
    }
}
```



## Beispiel 2

```
static void more(String fileName) throws FileNotFoundException, IOException {  
    FileInputStream stream =      new FileInputStream(fileName);  
    InputStreamReader reader =    new InputStreamReader(stream, "ISO-8859-1");  
    BufferedReader buffer =      new BufferedReader(reader);  
    System.out.print("\f");      // Konsole löschen  
    String line = buffer.readLine(); // erste Zeile lesen  
    int count = 1;  
    while (line != null) {      // solange noch Zeilen da  
        System.out.println(count + "\t" + line); // ausgeben  
        if (++count % 20 == 1) { // alle 20 Zeilen  
            System.in.read();    // auf Eingabe warten  
            System.out.print("\f"); // und Konsole löschen  
        }  
        line = buffer.readLine(); // nächste Zeile lesen  
    }  
}
```



# Dateisystem

- Ein Dateisystem organisiert Daten auf einem Datenträger
  - Festplatte, CD/DVD, USB-Stick...
- Dateien
  - Enthalten die eigentlichen Daten
- Verzeichnisse
  - Enthalten Informationen über Name, Speicherort auf dem Datenträger, Platzverbrauch, Zugriffsrechte usw. der enthaltenen Dateien und Verzeichnisse
  - Ein Verzeichnis ist das Wurzel- oder Hauptverzeichnis
- Verweise
  - Zeigen auf Dateien/Verzeichnisse, die eigentlich woanders eingetragen sind
  - Können auch ein Dateisysteme in ein anderes „einhängen“



# Pfadnamen

- Auf Dateien und Verzeichnisse wird über Pfadnamen zugegriffen
- Diese zählen die Verzeichnisse auf, über die man zu der/dem gewünschten Datei/Verzeichnis gelangt
- Sie sind durch einen *Verzeichnistrenner* getrennt (z.B. \ oder /)
- Sie sind entweder *absolut* zu einer Dateisystemwurzel (beginnen mit Verzeichnistrenner) oder *relativ* zum aktuellen Verzeichnis
- Das aktuelle Verzeichnis heißt "."
- Das übergeordnete Verzeichnis eines Verzeichnisses heißt ".."
- Beispiele
  - `/home/roeper/Desktop` (Unix, absolut)
  - `C:\Users\Thomas Röfer\Desktop` (Windows, absolut)
  - `C:\Users\Thomas Röfer\Documents\..\Desktop` (Windows, absolut)
  - `uebung09/FreeDB.java` (Relativ, unterhalb des aktuellen Verzeichnisses)
  - `../Nachbar` (eine Datei/ein Verzeichnis im Elternverzeichnis des aktuellen Verzeichnisses)

# Die Klasse `java.io.File`

- *`new File(String pathName)`*
  - Erzeugt ein Objekt, über das man Informationen über die Datei/das Verzeichnis abrufen kann
  - Die Datei/das Verzeichnis muss (noch) nicht existieren
- *`boolean exists()`*
  - Existiert die Datei überhaupt?
- *`String getName()`, `String getAbsolutePath()`, `String getCanonicalPath()`*
  - Pfadloser Name der Datei/des Verzeichnisses, absoluter Pfad, eindeutiger Pfad (".", ".." beseitigt)
- *`long length()`*
  - Länge dieser Datei
- *`boolean delete()`*
  - Löscht diese Datei und gibt *true* zurück, wenn erfolgreich
- *`boolean renameTo(String dest)`*
  - Benennt diese Datei um und gibt *true* zurück, wenn erfolgreich

# Die Klasse `java.io.File`

- *`boolean isFile()`, `boolean isDirectory()`*
  - Repräsentiert dieses Objekt eine Datei oder ein Verzeichnis?
- *`File[] listFiles()`*
  - Ist dies ein Verzeichnis, liefert die Funktion alle Unterverzeichnisse
- *`boolean mkdir()`, `boolean mkdirs()`*
  - Erzeugt dieses Verzeichnis und bei Bedarf alle Verzeichnisse auf dem Weg dorthin (`mkdirs()`) und gibt *true* zurück, wenn erfolgreich
- *`long getTotalSpace()`, `long getFreeSpace()`*
  - Wie viel Platz bietet der Datenträger, auf dem die Datei, das Verzeichnis liegt und wie viel ist davon noch frei?
- *`static File[] listRoots()`*
  - Liefert alle Dateisystemwurzeln dieses Systems
- *`static char separatorChar`, `static String separator`*
  - Der Verzeichnistrenner

# Dateisystemoperationen

```
static long dir(File file, String indentation) {  
    if(file.isDirectory()) {  
        System.out.println("[dir]\t" +  
            indentation + file.getName());  
        File[] files = file.listFiles();  
        long size = 0;  
        for (File f : files) {  
            size += dir(f, indentation + "  ");  
        }  
        return size;  
    } else {  
        System.out.println(file.length() + "\t"  
            + indentation + file.getName());  
        return file.length();  
    }  
}
```

```
static void dir(String path) {  
    long size = dir(new File(path), "");  
    System.out.println(size);  
}
```

# Musterlösung zu Übungsblatt 6

- Ein-/Ausgabespezifikation vergessen bzw. beim Testen ignoriert
- Tests nicht mit eigenem, kleinen Beispiel, daher Ergebnisse schlecht nachvollziehbar
  - An Tests soll man erkennen können, ob das Programm wie gewünscht funktioniert
  - Daher muss man die richtigen Ergebnisse kennen

Praktische Informatik I WS 2007/08

## Übungsblatt 6 Musterlösung

### Aufgabe 1 Könnt' ich doch so schreiben wie... (100%)

a) Eine Methode, die feststellt, ob bei zwei Indizes  $i$  und  $j$  im Text die gleichen  $N$ -Gramme beginnen, d.h. die Wörter bei  $i \dots i + n - 1$  und  $j \dots j + n - 1$  paarweise gleich sind.

`areNGramsEqual` durchläuft die beiden  $N$ -Gramme bei  $i$  und  $j$  und testet, ob die jeweiligen Worte gleich sind. Es ist erforderlich, dass  $i, j \in [0 \dots \text{words.length} - n]$  und  $n \geq 0$ . Die Schleife bricht ab, wenn das Ende der  $N$ -Gramme erreicht wurde oder wenn zwei ungleiche Worte gefunden wurden. Wurde das Ende der  $N$ -Gramme erreicht, sind diese gleich, ansonsten sind sie ungleich.

```
1 class NGram {
2     static boolean areNGramsEqual(String[] words, int n, int i, int j) {
3         int k = 0;
4         while (k < n && words[i + k].equals(words[j + k])) {
5             k = k + 1;
6         }
7         return k == n;
8     }
```

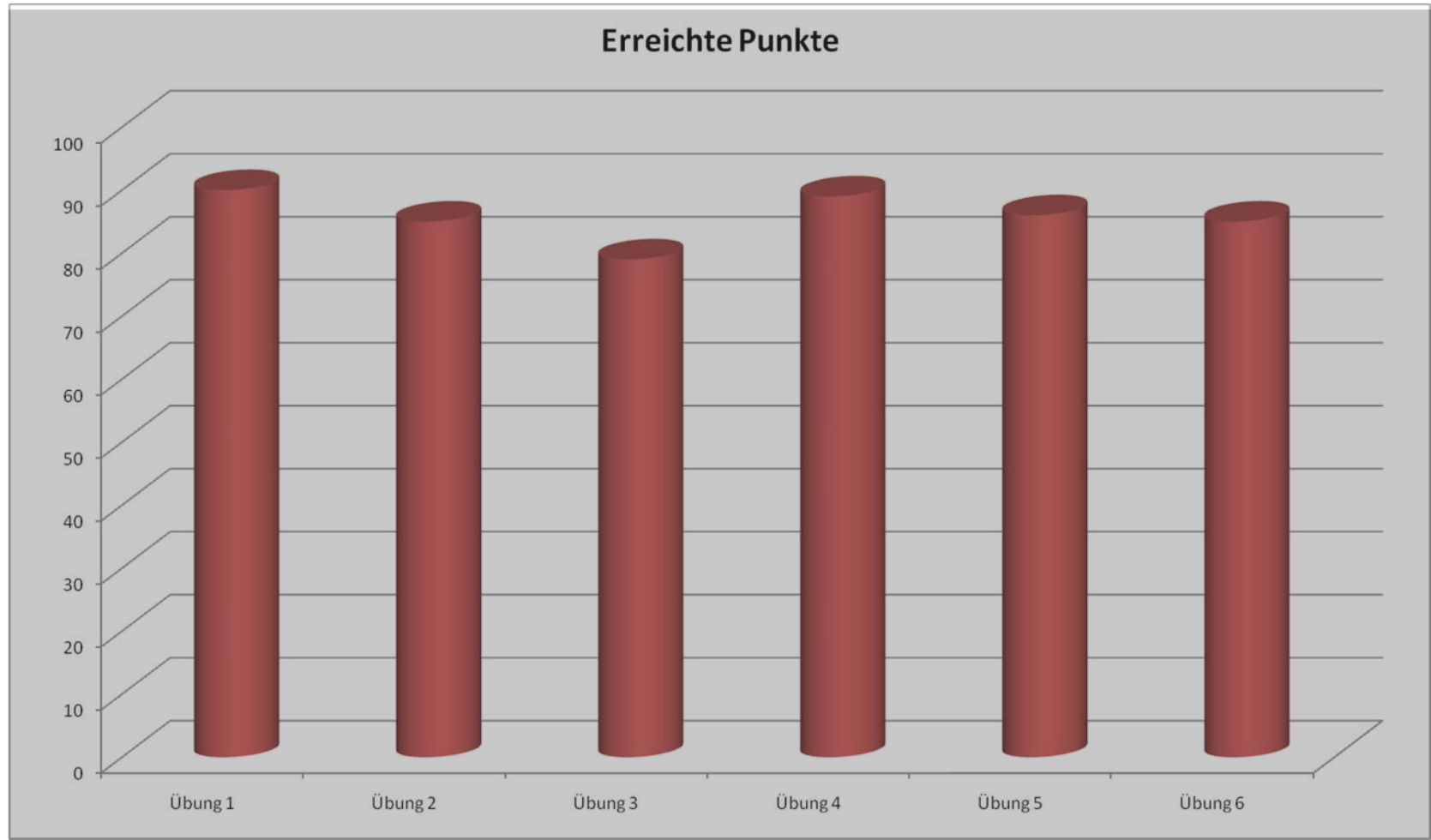
Tests. Ist  $n = 0$ , sind alle  $N$ -Gramme gleich, ebenso wie zwei  $N$ -Gramme an derselben Stelle unabhängig von  $n$  immer gleich sind. Des Weiteren wurde der Fall zweier gleicher  $N$ -Gramme an unterschiedlichen Stellen getestet, sowie der Vergleich zweier  $N$ -Gramme, die sich erst im letzten Wort unterscheiden. Dieser Test zeigt auch, dass der Vergleich eines  $N$ -Gramms am Ende des Textes funktioniert.

```
1 String[] text = {"testing", "testing", "one", "two", "one", "two",
2                 "testing", "testing", "one", "two", "three"};
3 NGram areNGramsEqual(text, 0, 0, 2)
4 true (boolean)
5 NGram areNGramsEqual(text, 3, 4, 4)
6 true (boolean)
7 NGram areNGramsEqual(text, 3, 1, 7)
8 true (boolean)
9 NGram areNGramsEqual(text, 3, 2, 8)
10 false (boolean)
```

b) Eine Methode, die zählt, wie oft das  $N$ -Gramm bei Index  $i$  im Text vorkommt.

`countEqualNGrams` durchläuft den Text vom Wort mit Index 0 bis zum Wort mit Index  $\text{words.length} - n - 1$  und erhöht den Zähler `count`, wenn ein gleiches  $N$ -Gramm gefunden wurde. Die Obergrenze ist künstlich beschränkt, um kein  $N$ -Gramm zu finden, das bei  $\text{words.length} - n$  beginnt, denn dieses ist auf jeden Fall eine Sackgasse. Es müssen  $i \in [0 \dots \text{words.length} - n]$  und  $n \geq 0$  gelten.

# Halbzeitstand





# Musterlösung zu Übungsblatt 7

- Aufgabe 1
  - Teilweise Zwischenschritte (Typerweiterungen) vergessen
  - Teilweise falsche Auswertungsreihenfolge
- Aufgabe 2
  - Befehlszähler nicht in R0
    - Dadurch Überschreiben von Befehlszähler unmöglich
  - Hochzählen des Befehlszählers nach der Ausführung des Befehls
    - MRI R0, 0 führt das Programm an Adresse 3 (!) fort
  - Illegale Instruktion?
  - Maschinenprogramme ignorieren teilweise die Werte der Parameter R1 und R2

Praktische Informatik I WS 2007/08

## Übungsblatt 7

Musterlösung

### Aufgabe 1 Typisch Ausdrücke (30%)

Gibt die Auswertungsreihenfolge und das Ergebnis der nachfolgenden Ausdrücke an und ermittelt dabei auch jeweils den Typ der (Teil-)Ausdrücke. Dabei gelte jeweils:  
 byte b = 2; int i = 5, j = 10, k = 20; float r = 5.0f; double d = 2.5;

Alle Ausdrücke sind syntaktisch korrekt. Das Auflösen von Klammern wurde nicht explizit aufgeführt, aber es gilt immer  $() : type \mapsto type$ . Für die Typanalyse wird die Schreibweise aus der Vorlesung verwendet, damit man alle Typen und Typumwandlungen in einem Ausdruck schreiben kann und dafür keine bessere Schreibweise existiert. In Pl-3 wird diese Schreibweise eine andere Bedeutung haben und der Typ eines Teilausdrucks ist einfach das, was hinter dem letzten  $\mapsto$  steht. Bei der Auswertung von Ausdrücken wurde auf die Darstellung von Umwandlungen zwischen verschiedenen Ganzzahltypen verzichtet. Der jeweils ausgewertete Teilterm ist unterstrichen.

a)  $j = (int) (r * b) + i \ll j$   
 Typen der (Teil-)Ausdrücke:

- $b : byte, i : int, j : int, r : float$
- $r * b : float \times (((byte \mapsto short) \mapsto int) \mapsto float) \mapsto float$
- $(int) (r * b) : float \times (((byte \mapsto short) \mapsto int) \mapsto float) \mapsto float \mapsto int$
- $(int) (r * b) + i : ((float \times (((byte \mapsto short) \mapsto int) \mapsto float) \mapsto float) \mapsto int) \times int \mapsto int$
- $(int) (r * b) + i \ll j : (((float \times (((byte \mapsto short) \mapsto int) \mapsto float) \mapsto float) \mapsto int) \times int \mapsto int) \times int \mapsto int$
- $j = (int) (r * b) + i \ll j : (int \times (((float \times (((byte \mapsto short) \mapsto int) \mapsto float) \mapsto float) \mapsto int) \times int \mapsto int) \times int \mapsto int) \times int \mapsto int$

Auswertung:  $j = (int) (r * b) + i \ll j$   
 $j = (int) (5.0f * 2) + i \ll j$   
 $j = (int) (5.0f * 2) + i \ll j$   
 $j = (int) (5.0f * 2.0f) + i \ll j$   
 $j = (int) 10.0f + i \ll j$   
 $j = 10 + 5 \ll j$   
 $j = 10 + 5 \ll j$   
 $j = 15 \ll j$   
 $j = 15 \ll 10$   
 $j = 15360$

b) "Hallo" + 1 + (j - i % (j + k))  
 Typen der (Teil-)Ausdrücke:

- $i : int, j : int, k : int, "Hallo" : String, 1 : int$



# Übungsblatt 9

- Drei Klassen implementieren
  - Spur auf CD
  - CD-Album (enthält Spuren)
  - CD-Sammlung (enthält CD-Alben)
- Alle drei Klassen enthalten Methoden, um ihren Inhalt auf der Konsole auszugeben
  - Ausgabe-Methoden stützen sich aufeinander ab
- Konstruktor von CD-Album ruft Informationen von freedb.org ab und generiert daraus die Spuren
  - Länge einer Spur kann man aus den Framepositionen und der Gesamtlänge der CD berechnen
- Dokumentieren mit JavaDoc

Praktische Informatik I WS 2007/08

## Übungsblatt 9

Abgabe: 16.01.08

### Aufgabe 1 PI-Tunes (100%)

Da Audio-CDs normalerweise keine Informationen über die Titel der CD und die Titel der einzelnen Spuren (z.B. Musiktitel) enthalten, holen sich gängige Abspielprogramme diese Information aus dem Internet, z.B. vom kommerziellen Server *gracenote.com* oder von freien Servern wie *freedb.org*. Die CDs werden dabei anhand der *Disk-ID* identifiziert. Da auf CDs kein solches Merkmal gespeichert ist, wird die Disk-ID aus anderen Informationen auf der CD berechnet. Leider ist die Disk-ID nicht eindeutig, so dass z.B. *freedb.org* die CDs auf verschiedene Kategorien verteilt, um die Wahrscheinlichkeit von Konflikten zu minimieren. Die Kategorien sind *blues*, *classical*, *country*, *data*, *folk*, *jazz*, *newage*, *reggae*, *rock*, *soundtrack* und *misc*.

Um die Disk-ID berechnen zu können, liest eine Abspiel-Software die Anzahl der Spuren  $n$ , die Gesamtdauer in Sekunden  $d$  und die Anfangspositionen der einzelnen Spuren in *Frames*  $f_i$ ,  $i \in [1 \dots n]$  aus (1 Frame =  $\frac{1}{75}$  Sekunde). Für die Berechnung der Disk-ID werden die Anfangspositionen der Spuren in Sekunden umgerechnet (mit Abrunden):

$$t_i = \left\lfloor \frac{f_i}{75} \right\rfloor \quad (1)$$

Es wird eine Prüfsumme über diese Spuranfänge gebildet. Die Funktion  $q$  berechnet dabei die Quersumme (im Dezimalsystem) einer Zahl:

$$s = \left( \sum_{i=1}^n q(t_i) \right) \bmod 255 \quad (2)$$

Die Gesamtdauer aller Spuren auf der CD in Sekunden wird wie folgt berechnet:

$$g = d - t_1 \quad (3)$$

Die Disk-ID ist eine 32-Bit-Zahl. In die Bits 0-7 wird die Anzahl der Spuren  $n$  eingetragen. Die Bits 8-23 nehmen die Gesamtdauer der Spuren  $g$  auf. In den Bits 24-31 wird die Prüfsumme über die Spuranfänge  $s$  abgelegt. Die Disk-ID wird immer als 8-stellige Hexadezimalzahl dargestellt. Die Ziffern  $a-f$  werden dabei klein geschrieben.

Auf der PI-1-Seite findet ihr das Archiv *uebung09.zip*, das die Klasse *FreeDB* enthält. Ein Aufruf wie z.B. *FreeDB.read("misc", "940b1a0a")* liefert eine Reihung von Zeichenketten zurück, in der jeder Eintrag die Form *Schlüssel=Wert* hat. In dem Beispiel wurden die Daten aus der Seite <http://www.freedb.org/freedb/misc/940b1a0a> extrahiert, wobei *misc* die Kategorie ist und *940b1a0a* die Disk-ID.

Implementiert die folgenden drei Klassen:

- a) Eine Klasse, die Informationen über eine Spur auf einer CD repräsentiert. Sie soll eine Methode bereitstellen, die die laufende Nummer der Spur auf der CD, den Titel und die Länge der Spur in Minuten und Sekunden auf der Konsole ausgibt.