Dateien in Java

1.1 Grundlagen

1.1.1 Streams

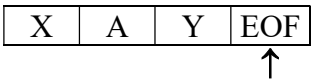
In der Datenverarbeitung fließen Datenströme (Streams) von Quellen (z.B. Tastatur) zu Senken (z.B. Drucker). Dateien sind spezielle Quellen bzw. Senken (oder beides zusammen), in die Datenströme fließen oder aus denen Datenströme herkommen. In Java gibt es über dreißig Klassen zur Verarbeitung der Datenströme.
Der Zugriff auf Dateien wird in Java über spezielle Klassen geregelt, von denen uns hier nur die Klasse **RandomAccessFile** interessiert.

1.1.2 Modell einer Datei mit wahlfreiem Zugriff

Eine Datei ist aus programmtechnischer Sicht eine Organisationseinheit, in der die einzelnen Bytes hintereinander, wie in einem Feld (array), abgelegt sind.
Der Dateianfang hat die Position 0. Das Dateiende wird mit EOF (End of File) bezeichnet.
Jede Datei besitzt automatisch einen sogenannten Dateizeiger. Beim Öffnen einer Datei zeigt dieser automatisch auf den Dateianfang (Position 0).
Werden Daten beschrieben bzw. gelesen, so verschiebt sich dieser Dateizeiger automatisch um die Anzahl der beschriebenen bzw. gelesenen Bytes und zeigt auf die nächste Stelle, von der ab nun durch einen entsprechenden Befehl gelesen oder ab der in die Datei geschrieben werden kann.

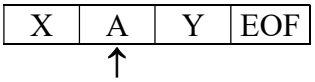
Beispiel:
Situation beim Öffnen einer schon existierenden Datei der Länge 0:

In die Datei werden dann die Zeichen 'X', 'A', 'Y' geschrieben:

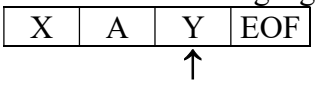


Der Dateizeiger zeigt dann auf die nächste Position (hier das Dateiende)

Jetzt soll z.B. das Zeichen A gelesen werden. Dazu muss man den **Dateizeiger** vorher an die Stelle A verschieben:



Nach dem Lesevorgang ergibt sich folgendes:



1.2 Dateiinhalte bearbeiten

1.2.1 Dateien anlegen

Wenn man eine Datei bearbeiten bzw. anlegen will, ist dazu der Pfad und Name dieser Datei nötig. Dazu wird der Name (und evtl. der Pfad) der Datei, wie z.B. `C:\\maier\\privat.txt` benutzt. Die Verzeichnisse und Dateinamen müssen durch \\ getrennt werden. Gibt man keinen Pfad an, so wird automatisch das aktuelle Verzeichnis (Ordner) als Pfad angenommen.
In Java wird auf eine Datei zugegriffen, indem man ein Objekt der Klasse RandomAccessFile erzeugt. Über die Methoden dieses Objekts wird dann auf die Datei zugegriffen.
Ein Konstruktor dieser Klasse hat als ersten Parameter als Zeichenkette den Dateinamen und als zweiten Parameter den Modus des Dateizugriffs (lesen. bzw. lesen und schreiben).

1.2.1.1 Beispiel

```
RandomAccessFile myDatei =
                    new RandomAccessFile("C:\\test1.txt","rw");
```

Modus des Dateizugriffs:
"r": Nur Lesezugriff möglich. Die Datei muss existieren, sonst gibt es einen Fehler.
"rw": Lese- und Schreibzugriff. Nur wenn die Datei nicht existiert, wird sie angelegt.

1.2.1.2 try-catch-Technik

1.2.1.2.1 Beispiel

```
import java.io.*;

public class MainDatei1 {
  public static void main(String[] args) {
    RandomAccessFile myDatei = null;
    try{
      myDatei = new RandomAccessFile("C:\\test1.txt","r");
      // schließe Datei
      myDatei.close();
    }
    catch(FileNotFoundException e){
      // Eine mögliche Ausgabe (genauere Fehlerbeschreibung)
      // mit toString()
      System.out.println("Datei nicht da:" +e.toString());
    }
    catch(IOException e){
      System.out.println("Fehler: Datei nicht schließbar:");
      // Andere mögliche Ausgabe (genauere Fehlerbeschreibung)
      // mit printStackTrace()
      e.printStackTrace();
    }
  }
}
```

Wie macht sich ein Fehler beim Anlegen oder Bearbeiten einer Datei bemerkbar und wie wird er behandelt?

Java verwendet dazu die try - catch Technik.
Wenn ein Schüler eine Fremdsprache lernt, wird er während der Lernphase nicht direkt auf "die freie Wildbahn losgelassen", sondern bekommt einen Lehrer und ein Unterrichtsgebäude zur Verfügung gestellt. Innerhalb dieses "eingezäunten Bereichs" wird ein von ihm verursachter Fehler (z.B. falsch ausgesprochenes Wort) von dem Lehrer "eingefangen" und kommentiert.
Analog wird in Java verfahren:
Anweisungen, die Fehler verursachen können (z.B. Datei kann nicht geöffnet werden, weil sie nicht vorhanden ist), werden durch try-catch "eingezäunt". Im try-Bereich wird versucht Anweisungen auszuführen. Falls dies einen Fehler verursacht, wird ein Fehlerobjekt geworfen und im catch-Bereich eingefangen (und bearbeitet), wobei der catch-Bereich aus mindestens einem catch-Block bestehen muss.

Konkret zum Beispiel oben:
In dem obigen try-Bereich können zwei Fehler passieren:
- Wenn eine Datei geöffnet werden soll, und diese Datei aber nicht existiert, wird ein Fehlerobjekt der Klasse **FileNotFoundException** geworfen (siehe: Konstruktor der Klasse RandomAccessFile in der API-Doku zur Klasse RandomAccessFile).
- Wenn die Datei geschlossen werden soll und dies nicht gelingt (Diskette, auf der sich die Datei befindet, ist nicht mehr im Laufwerk), wird ein "allgemeineres" Fehlerobjekt der Klasse IOException geworfen (siehe: Methode close() der Klasse RandomAccessFile in der API-Doku zur Klasse RandomAccessFile).

Es gibt nun zwei catch-Bereiche. Welcher fängt das Fehlerobjekt ein?
Der erste catch-Bereich, der zu dem geworfenen Fehlerobjekt **passt**, fängt das Fehlerobjekt, macht die Anweisungen des catch-Bereichs und macht dann nach dem letzten catch-Bereich weiter.
Wichtig dabei ist, dass man zuerst den catch-Bereich des spezielleren Fehlerobjekts (hier: der Klasse FileNotFoundException) bringt und dann den catch-Bereich des allgemeineren Fehlerobjekts (hier: der Klasse IOException).

Bemerkungen
1) Die Methode toString() liefert eine Zeichenkette, in der Infos über das Objekt stehen.
2) Die Methode printStackTrace() ruft intern zuerst die Methode toString auf und bringt dann noch die Zeilen des Quellcodes, wo sich ein Fehler ereignet hat.

Wichtig:
Bitte in der API-Doku von SUN die Methoden der Klasse RandomAccessFile anschauen, um Dateimanipulationen machen zu können (Dateien anlegen, Dateien lesen, Dateien beschreiben, usw.)

1.2.2 Dateizeiger verschieben

Der Dateizeiger zeigt immer an eine bestimmte Stelle der Datei. Nach einem Lese- oder Schreibvorgang wird der Dateizeiger **automatisch** verschoben.
Man kann den Dateizeiger relativ zum Dateianfang verschieben.

1.2.2.1 Beispiel

```
import java.io.*;

public class MainDatei2 {
  public static void main(String[] args) {
    RandomAccessFile myDatei = null;
    long myPosition;
    try{
      myDatei = new RandomAccessFile("C:\\test1.txt","rw");
      // Hole Position des Dateizeigers
      myPosition=myDatei.getFilePointer();
      // Verschiebe Dateizeiger um 2 in Richtung Dateiende
      myDatei.seek(myPosition+2);
      myDatei.close();
    }
    catch(FileNotFoundException e){
      System.out.println("Datei nicht da: "+e.toString());
    }
    catch(IOException e){
      System.out.println("Dateizugriff: "+e.toString());
    }
  }
}
```

1.2.2.2 Bemerkungen

1) Wenn man den Dateizeiger über das Dateiende hinaus verschiebt, wird kein Fehlerobjekt (Exception) erzeugt (ausgelöst). Siehe Java-Doku.
Wenn man den Dateizeiger vor den Dateianfang (negative Zahl als Parameter) verschiebt, gibt es einen Fehler (d.h. IOException).

2) Mit der Methode setLength(...) kann man die Länge (Größe) einer Datei festlegen.
Die Anweisung setLength(0) bedeutet also ein Löschen des Datei**inhalts**, aber nicht der Datei!

3) Mit der Methode
`long length()`
kann man die Länge (Größe) einer Datei feststellen.

1.2.3 Dateien lesen und beschreiben

1.2.3.1 Zahlen in Dateien schreiben und lesen

1.2.3.1.1 Beispiel

```
import java.io.*;

public class MainDatei3 {
  public static void main(String[] args) {
    RandomAccessFile myDatei = null;
    long myPosition;
    int zahl=5;
    int wert;

    try{
      myDatei = new RandomAccessFile("C:\\test1.txt","rw");
      // Schreibe Integer-Zahlen in Datei an Dateianfang
      myDatei.writeInt(zahl);
      myDatei.writeInt(10);
      // Verschiebe Dateizeiger an Dateianfang
      myDatei.seek(0);
      // Lese Zahl aus
      wert=myDatei.readInt();
      // Gib Zahl auf Bildschirm aus
      System.out.println("Zahl="+wert);
      myDatei.close();
    }
    catch(FileNotFoundException e){
      System.out.println("Datei nicht da: "+e.toString());
    }
    catch(IOException e){
      System.out.println("Dateizugriff: "+e.toString());
    }
  }
}
```

1.2.3.1.2 Bemerkungen

1) Mit den Anweisungen writeInt(...) und readInt() können Integer-Zahlen in die Datei geschrieben oder von der Datei gelesen werden.
Es gibt Anweisungen, mit denen Zahlen mit anderen Datentypen in die Datei geschrieben oder von der Datei gelesen werden können (näheres dazu in der API-Doku).

2) Wenn beim Lesevorgang das Dateiende erreicht wird, bevor die komplette Zahl eingelesen wurde, wird ein Fehlerobjekt (Exception) der Klasse EOFException erzeugt.

3) Im obigen Programm wurden die Zahlen 5 und dann 10 in die Datei geschrieben.
Der Programmzeiger ist danach am Dateiende. Will man jetzt die 2. Zahl 10 auslesen, muss der Dateizeiger an die Position (Stelle) 4 verschoben werden, da in Java eine Integer-Zahl 4 Byte Speicher benötigt.

1.2.3.2 Einzelne Bytes in Dateien schreiben und lesen

1.2.3.2.1 Beispiel

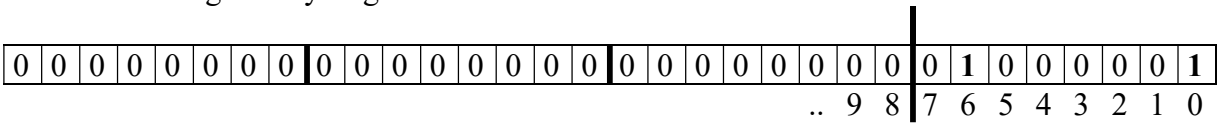
```
import java.io.*;

public class MainDatei5 {
  public static void main(String[] args) {
    RandomAccessFile myDatei = null;
    int z=65;  // Zeichen A
    try{
      myDatei = new RandomAccessFile("C:\\test1.txt","rw");
      // Schreibt wie folgt genau ein Byte in die Datei:
      // Von dem 4 Byte Integer-Wert werden die 3
      // höchstwertigen Bytes vorher entfernt.
      myDatei.write(z);
      // Schreibt wie folgt genau ein Byte in die Datei:
      // der ASCII-Wert von B kommt in die Datei
      myDatei.write('B');
      // Verschiebe Dateizeiger an Dateianfang
      myDatei.seek(0);
      // Lese Zeichenketten aus
      do{
        // Liest genau ein Byte aus der Datei.
        z=myDatei.read();
        System.out.println((char)z);
      }while(myDatei.getFilePointer()<=myDatei.length()-1);
      // alternativ: while (z != -1)
      myDatei.close();
    }
    catch(FileNotFoundException e){
      System.out.println("Fehler: Datei nicht da:
                                              "+e.toString());
    }
    catch(IOException e){
      System.out.println("Fehler: Dateizugriff:
                                              "+e.toString());
    }
  }
}
```

1.2.3.2.2 Bemerkungen

Mit den Anweisungen write(...) und read() werden einzelne Bytes in Dateien geschrieben oder gelesen. Dabei werden Integer-Werte (4 Byte) benutzt, von denen von den 4 Bytes die 3 höchstwertigen vor dem Schreiben und Lesen entfernt werden.
Wenn beim Lesevorgang das Dateiende erreicht wird, bevor die komplette Zahl eingelesen wurde, wird **kein** Fehlerobjekt (Exception) erzeugt. Näheres dazu in der API-Doku.

Beispiel:
Durch die folgenden 4 Bytes wird der integer-Wert 65 binär dargestellt.
Die höherwertigen 3 Bytes gehen von der Bitnummer 8 bis Bitnummer 31



1.2.3.3 Einzelne Zeichen (char) in Dateien schreiben und lesen

1.2.3.3.1 Beispiel

```
import java.io.*;

public class Startklasse {
  public static void main(String[] args) {
    RandomAccessFile myDatei = null;
    long myPosition;
    int zeichen='A';
    char wert;
    try{
      myDatei = new RandomAccessFile("D:\\test1.txt","rw");
      // Schreibe Integer-Zahl in Datei an Dateianfang
      myDatei.writeChar(zeichen);
      myDatei.writeChar('B');
      // Verschiebe Dateizeiger an Dateianfang
      myDatei.seek(0);
      // Lese Zahl aus
      wert=myDatei.readChar();
      // Gib Zahl auf Bildschirm aus
      System.out.println("Zeichen="+wert);
      myDatei.close();
    }
    catch(FileNotFoundException e){
      System.out.println("Fehler: Datei nicht da:
                                                "+e.toString());
    }
    catch(IOException e){
      System.out.println("Fehler: Dateizugriff:
                                                "+e.toString());
    }
  }
}
```

1.2.3.3.2 Bemerkungen

Im obigen Programm wurden die Zeichen A dann B in die Datei geschrieben.
Der Programmzeiger ist danach am Dateiende. Will man jetzt das 2. Zeichen B auslesen, muss der Dateizeiger an die Position (Stelle) 2 verschoben werden, da in Java ein Zeichen 2 Byte Speicher benötigt.

1.2.3.4 Zeichenketten in Dateien schreiben und lesen

1.2.3.4.1 Beispiel

```
import java.io.*;

public class MainDatei4 {
  public static void main(String[] args) {
    RandomAccessFile myDatei = null;
    String myString1, myString2;
    try{
      myDatei = new RandomAccessFile("C:\\test1.txt","rw");
      // Schreibe 2 Zeichenkette in Datei an Dateianfang
      myDatei.writeBytes("Hallo"+"\r\n");
      myDatei.writeBytes("wie geht es?"+"\r\n");
      // Verschiebe Dateizeiger an Dateianfang
      myDatei.seek(0);
      // Lese Zeichenketten aus
      myString1=myDatei.readLine();
      myString2=myDatei.readLine();
      // Gib Zeichenketten auf Bildschirm aus
      System.out.println("Zeichenketten="+myString1+" "
                                          +myString2);
      myDatei.close();
    }
    catch(FileNotFoundException e){
      System.out.println("Fehler: Datei nicht da:
                         "+e.toString());
    }
    catch(IOException e){
      System.out.println("Fehler: Dateizugriff:
                         "+e.toString());
    }
  }
}
```

1.2.3.4.2 Bemerkungen

1) Mit der Zeichenkette "\r\n" wird das Zeilenende (Carriage Return, Linefeed = Wagenrücklauf, Zeilenvorschub) markiert.

2) Die Anweisung readline() liest byteweise ab der Position des Dateizeigers aus der Datei. Der Lesevorgang wird beendet, wenn ein \r oder \n oder das Dateiende erreicht wird. \r bzw. \n werden weggeworfen und nicht in einer Zeichenkette gespeichert.

3) Die Anweisung writeBytes() schreibt eine Zeichenkette bytesweise in die Datei.

4) Wenn beim Lesevorgang das Dateiende erreicht wird, bevor ein Byte gelesen wurde, wird **kein** Fehlerobjekt (Exception) erzeugt, sondern null zurückgeliefert.
Näheres dazu in der API-Doku.

1.3 Dateien (nicht Dateiinhalte) bearbeiten

Da mit Hilfe der Klasse RandomAccessFile nur die Dateinhalte (und auch noch das Anlegen einer Datei) bearbeitet werden können, kann man damit keine Dateien löschen, umbenennen, usw. Dies kann man mit Hilfe der Klasse **File** realisieren. Außerdem kann man auch Dateien anlegen (was auch mit der Klasse RandomAccessFile möglich ist).

1.3.1 Beispiel

```
import java.io.*;

public class MainDatei6 {
  public static void main(String[] args) {
    File myDatei;
    boolean b;
    // Ein Objekt anlegen und mit einer Datei "verbinden"
    // Dies bedeutet nicht, dass es diese datei schon gibt!
    myDatei = new File("c:\\temp\\test1.txt");
    // Prüfen, ob diese Datei auch existiert
    b = myDatei.isFile();
    if (b == true) {
      System.out.println("Datei existiert");
    }
    else {
      System.out.println("Datei existiert nicht");
      System.out.println("deshalb wird sie jetzt angelegt");
      try {
        // Bem: Ordner c:\temp muss existieren, sonst wird
        //      Datei nicht angelegt.
        myDatei.createNewFile();
      }
      catch (IOException e) {
        System.out.println("Datei kann nicht angelegt werden");
      }
    }
    // Löschen einer Datei
    File f1 = new File("C:\\temp\\test1.txt");
    f1.delete();
    // auch möglich
    // myDatei.delete();

    // Verzeichnis erstellen
    File f2 = new File("C:\\temp\\testOrdner");
    b = f2.mkdir();
    if (b == true) {
      System.out.println("Verzeichnis wurde erstellt");
    }
    else {
      System.out.println("Verzeichnis wurde nicht erstellt,");
      System.out.println("weil es schon existiert");
    }
  }
}
```

1.3.2 Bemerkungen

1) Mit einem Konstruktor von File kann man eine "Verbindung" zu einer Datei erstellen. Diese Datei muss aber nicht schon existieren.

2) Man kann diese Datei (falls sie noch nicht existiert), mit createNewFile() anlegen. Sie hat dann den Dateinamen, der beim Konstruktor als Parameter verwendet wurde.

3) Eine Datei kann mit delete() gelöscht werden