Einführung

in

Dateien

(in Java)

Version 2015

Autor: Dieter Lindenberg

Inhalt

Die Klasse *File*

Ein Objekt der Klasse *File* repräsentiert einen Datei- oder Verzeichnisnamen im Dateisystem. Die Datei oder das Verzeichnis, das das File-Objekt beschreibt, muss nicht physikalisch existieren. Der Verweis wird durch einen Pfadnamen spezifiziert. Dieser kann absolut oder relativ zum aktuellen Verzeichnis angegeben werden.

Konstruktor: **`File(String name);`**

Beispiel: Erzeuge ein File-Objekt für die Datei „Testdatei.abc“ direkt auf dem Laufwerk C:
Lösung:
```
import java.io.File;      // Diese Klasse muss importiert worden sein.
……..
File f = new File("C:/Testdatei.abc");
```
Hinweis: Das in diesem Beispiel erzeugte Objekt f der Klasse File sorgt nicht automatisch dafür, dass diese Datei auch existiert. Ein Leseversuch würde bei nicht existierender Datei zu einem Fehler führen.

boolean exists()
Liefert *true*, wenn das File-Objekt eine existierende Datei oder einen existierenden Ordner repräsentiert.

long length()
Gibt die Länge der Datei in Byte zurück. *// also normalerweise nicht die Anzahl der in dieser Datei gespeicherten Elemente*

long lastModified()
Liefert den Zeitpunkt, zu dem die Datei zum letzten Mal geändert wurde. Die Zeit wird in Millisekunden ab dem 1. Januar 1970, 00:00:00 UTC, gemessen. Die Methode liefert 0, wenn die Datei nicht existiert oder ein Ein/Ausgabefehler auftritt.

boolean setLastModified(long time)
Setzt die Zeit (wann die Datei zuletzt geändert wurde). Die Zeit ist wiederum in Millisekunden seit dem 1. Januar 1970 angegeben.

Zwar lässt Java die Ermittlung und das Setzen der Zeit der letzten Änderung zu, doch gilt dies nicht unbedingt für die Erzeugungszeit. Das Betriebssystem *Windows* speichert letztere hingegen schon. Bei manchen Windows-Betriebssystemen (nicht bei allen!) wird auch gleichzeitig die Erzeugungszeit geändert, wenn man z.B. die Zeit der letzten Änderung in die Vergangenheit setzt.

boolean renameTo(File d)
Benennt die Datei in den Namen um, der durch das File-Objekt d gegeben ist. Dabei darf keine Datei mit dem neuen Namen bereits existieren (weil es sonst einen Namenskonflikt geben würde). Die aktuelle Datei, welche umbenannt werden soll, darf auch nicht geöffnet sein. Ging alles gut, wird *true* zurückgegeben.

boolean delete()
Löscht die Datei oder das leere Verzeichnis. Falls die Datei nicht gelöscht werden konnte, gibt es den Rückgabewert *false*.

void deleteOnExit()
Löscht die Datei bzw. das Verzeichnis, wenn die virtuelle Maschine korrekt beendet wird.

Aufgaben

Erstelle zunächst mit Standardprogrammen wie z.B. *Editor, Wordpad, OpenOffice* oder *LibreOffice* mehrere unterschiedliche Textdateien, die alle nur das Wort „Hallo“ enthalten! Speichere diese Dateien unter den Namen „Editortextdatei.txt“ usw. in das Projekt-Verzeichnis, in dem dein aktuelles NetBeans-Projekt steht (also das, mit dem du die folgenden Aufgaben lösen wirst)! Schreibe nun ein Java-Programm, welches nachfolgende Aufgaben löst! Vergleiche deine Ergebnisse auch immer mit den Angaben des Window-Explorers!

a) Überprüfe, ob die Datei mit dem Namen „Editortextdatei.txt“ existiert!
b) Gib die Größe der Datei mit dem Namen „Editortextdatei.txt“ (in Bytes) aus! Sicherlich wird die Angabe des Window-Explorers wesentlich größer sein. Informiere dich über den Grund dafür!
c) Notiere dir zunächst das Datum der letzten Änderung der Datei „Editortextdatei.txt“! Setze nun dieses Datum um 10 Tage zurück! Bemerkung: So lässt sich auch ein Widerspruch zwischen den Windowsangaben für Erstelldatum und Änderungsdatum herbeiführen.
d) Benenne die Datei „Editortextdatei.txt“ um! Der neue Name sei „Willi.abc“ .
e) Die Datei „Editortextdatei.txt“ soll gelöscht werden, wenn auf einen entsprechenden Button geklickt wird.
f) Die Datei „Editortextdatei.txt“ soll erst nach Beendigung deines Programmes gelöscht werden.

Lösungen

```
import java.io.File;

public class DateiAufgaben extends javax.swing.JFrame {

  File f;

  public DateiAufgaben() {
     initComponents();
     f = new File("Editortextdatei.txt");
  }


  private void btaMouseClicked(java.awt…..) {
     if (f.exists()) tfAusgabe.setText("Datei existiert");
     else tfAusgabe.setText("Datei existiert nicht");
    }


  private void btbMouseClicked(java.awt…..) {
     tfAusgabe.setText("Größe = " + f.length() +" Bytes");
  }


  private void btcMouseClicked(java.awt…..) {
     long n = f.lastModified();
     n = n - 10*24*3600*1000;
     f.setLastModified(n);
  }


  private void btdMouseClicked(java.awt……) {
     File g = new File("Willi.abc");
     f.renameTo(g);
  }


  private void bteMouseClicked(java.awt…..) {
     f.delete();
  }


  private void btfMouseClicked(java.awt…..) {
```

```
    f.deleteOnExit();
}
```

Im Umgang mit externen Dateien (d.h. beim Lesen aus der bzw. beim Schreiben in die Datei) können vielfältige Fehler auftreten: Die Datei könnte evtl. gar nicht existieren, das Ende der Datei wäre schon überschritten, die Datei kann nicht beschrieben werden (weil sie z.B. nur zum Lesen geöffnet worden ist), die Datei enthält nicht den Datentyp, den man lesen will, usw.

Aus diesem Grund werfen sehr viele Dateimethoden eine *Exception* aus. Das muss beim Methodenaufruf beachtet werden.

Damit das Programm nicht abstürzt, **müssen** diese Methoden, in denen Laufzeitfehler auftreten könnten und die deshalb eine Exception auswerfen, in einem sog. *try and catch* Block ausgeführt werden (alternativ dazu könnte man auch die *Exception* an die übergeordnete, aufrufende Instanz weiter leiten). Im *try*-Teil stehen die kritischen Anweisungen, im *catch*-Teil stehen die Anweisungen für den Fall, dass ein Fehler aufgetreten sein sollte.
Näheres dazu steht im Skript „Java Grundlagen mit NetBeans“.

Die Klasse *RandomAccessFile*

Dateien können auf zwei unterschiedliche Arten gelesen und modifiziert werden: zum einen über einen Datenstrom, der Bytes wie in einem Medien-Stream (zum Beispiel YouTube-Filme) verarbeitet, zum anderen über wahlfreien Zugriff (engl. random access). Während der Datenstrom beim Lesen eine strenge Sequenz erzwingt, ist dies beim wahlfreien Zugriff egal, da innerhalb der Datei beliebig hin und her gesprungen werden kann und ein Dateizeiger (welcher immer auf das gerade aktuelle Element zeigt) verwaltet wird, den wir setzen können. Da wir es mit Dateien zu tun haben, heißt das Ganze dann *Random Access File*, und die Klasse, die wahlfreien Zugriff anbietet, ist *java.io.RandomAccessFile*.

Im Programm-Quelltext müssen folgende Klassen importiert werden:
import java.io.IOException;
import java.io.RandomAccessFile;

Einige Methoden der Klasse RandomAccessFile

Die Klasse *RandomAccessFile* ist keine Unterklasse der Klasse *File*. Insbesondere deshalb gibt es hier auch keine *exist*- und *delete*-Methode.

RandomAccessFile(String name, String mode) throws FileNotFoundException
In diesem Konstruktor bestimmt der zweite Parameter eine Zeichenkette für den Zugriffsmodus; damit lässt sich eine Datei lesend oder schreibend öffnen. Genauer gilt:
mode = `"r"` Die Datei wird zum Lesen geöffnet.
mode = `"rw"` Die Datei wird zum Lesen und / oder Schreiben geöffnet. Eine existierende Datei wird dabei geöffnet, und Daten können an beliebigen Stellen eingefügt werden, ohne dass die Datei gelöscht wird. Existiert die Datei nicht, wird sie schon durch diesen Konstruktor auch physikalisch neu angelegt, und ihre Startgröße ist null. Einen Modus nur zum Schreiben (also etwa mode = `"w"`) gibt es nicht bzw. er funktioniert nicht.
Auch die Reihenfolge der Buchstaben ist wichtig: mode = `"wr"` wäre falsch!
Nach dem Aufruf des Konstruktors zeigt der aktuelle *Filepointer* auf den Anfang der Datei, er hat also den Wert 0.
Beispiel: *RandomAccessFile f = new RandomAccessFile("E:/Testdatei.txt", "rw");*
Falls der angegebene Dateiname noch nicht existiert, wird eine neue Datei mit diesem Namen erzeugt.

Folgende Methoden lesen jeweils ein oder mehrere Bytes und liefern das Ergebnis als primitiven Datentyp zurück. Das Attribut *final* bedeutet, dass diese Methoden in eventuellen Unterklassen nicht überschrieben werden können. Direkt nach dem Lesen eines Elementes aus der Datei wird der interne Dateizeiger automatisch entsprechend viele Speicherzellen weiter vor gerückt.

int read() throws IOException
Liest genau ein Byte und liefert es als *int* zurück.

final byte readByte() throws IOException

final int readInt() throws IOException *// int = 4-Byte-Zahl*

final char readChar() throws IOException // *char = 2-Byte-Zeichen*

final double readDouble() throws IOException *// double = 8-Byte-Zahl*

final String readLine() throws IOException
Liest eine Textzeile, die mit dem Zeilenendezeichen abgeschlossen wurde. Die letzte Zeile muss nicht so abgeschlossen sein, denn ein Dateiende zählt auch als Zeilenende. Das Zeilenendezeichen wird nicht mit als Funktionsergebnis übergeben.

void close()
Schließt eine geöffnete Datei wieder

Die obigen Methoden liefern nicht alle einen Fehler, wenn die Datei schon fertig abgearbeitet wurde und keine Daten mehr anliegen. Im Fall von *int read()* gibt es einfach den Rückgabewert –1 und keine *Exception*. Ähnliches gilt für *readLine()*. Diese Methode liefert *null* am Dateiende.

Analog zu den Lesemethoden gibt es natürlich auch Schreibmethoden, welche ebenfalls alle eine *IOException* auslösen können. Auch direkt nach dem Schreiben eines Elementes in die Datei wird der interne Dateizeiger automatisch entsprechend viele Speicherzellen weiter vor gerückt.
Hinweis: wird eine Datei zum 2. Mal neu beschrieben, so wird ihr alter Inhalt nur überschrieben. Reste vom alten Inhalt bleiben erhalten, wenn der neue Inhalt kürzer ist als der alte.

writeBoolean(boolean b)
writeChar(int n)
writeChars(String s)
writeDouble(double r)
writeFloat(float r)
writeInt(int v)
writeLong(long v)
writeShort(int v)
writeUTF(String str) Hinweis: Zusätzlich werden beim Schreiben am Anfang 2 Byte vorangestellt, welche die Länge des Strings str angeben. Das macht Probleme beim Wiederauslesen des Strings. Der ausgelesene String ist um 2 Byte länger. Beispiel: Wird der String "Hallo" gespeichert, so erhält man beim Auslesen den String " Hallo". Dabei haben die beiden ersten (in diesem Fall) nicht sichtbaren Zeichen die ASCII-CODES 0 und 5, weil das Wort "Hallo" 5 Zeichen lang ist.

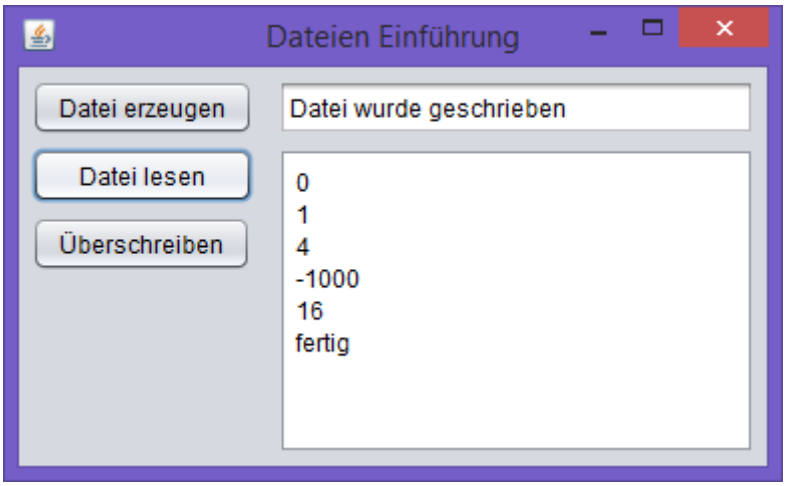
Die nächsten 3 Methoden liefern Informationen über die Datei:

long length() throws IOException
Liefert die Länge der Datei (=Anzahl der Bytes). Schreibzugriffe erhöhen natürlich den Wert.

long getFilePointer() throws IOException
Liefert die momentane Position n (das ist die Nummer des n-ten **Bytes**, egal welche Datentypen die Datei enthält) des Dateizeigers. Das erste Byte steht an der Stelle n=0.

void seek(long pos) throws IOException
Setzt die Position des Dateizeigers auf pos. Diese Angabe ist absolut und kann daher nicht negativ sein. Falls doch, wird eine Ausnahme ausgelöst.
Beispiel: file.seek(file. length()); setzt den Zeiger auf das Ende der Datei.

Das folgende kleine Programm zeigt eine erste Anwendung obiger Befehle:




```
import java.io.IOException;
import java.io.RandomAccessFile;

public class Kapitel1 extends javax.swing.JFrame {

  RandomAccessFile  f;

  public Kapitel1() {
    initComponents();
  }


  private void btDateiErzeugenMouseClicked(java.awt….) {
     try {
        f = new RandomAccessFile("Testdatei.abc", "rw");
        for (int i = 0; i < 5; i++)  f.writeInt(i * i);

        tfAusgabe.setText("Datei wurde geschrieben");
        f.close();
     }
     catch(IOException fe) {
        tfAusgabe.setText("Dateifehler beim Erzeugen");
     }
  }



  private void btDateiLesenMouseClicked(java.awt.…. evt) {
     int zahl;
     taAusgabe.setText("");
     try {
        f = new RandomAccessFile("Testdatei.abc", "r");

        while (f.getFilePointer() < f.length()) {
```

```
            zahl = f.readInt();
            taAusgabe.append(zahl+"\n");
         }

         taAusgabe.append("fertig");
         f.close();
      }
      catch(IOException fe) {
         tfAusgabe.setText("Dateifehler beim Lesen");
      }
   }



   private void btUeberschreibenMouseClicked(java.awt…) {
      taAusgabe.setText("es wurde überschrieben");
      try {
         f = new RandomAccessFile("Testdatei.abc", "rw");

         f.seek(3*4);
         // Integer-Zahlen sind 4-Byte-Zahlen. Die vierte Zahl beginnt mit dem 12. Byte
         f.writeInt(-1000);
         f.close();
      }
      catch(IOException fe) {
         tfAusgabe.setText("Dateifehler beim Ueberschreiben");
      }
   }

}
```

Aufgaben

Erstelle ein einziges Programm, welches nach und nach alle folgenden Aufgaben löst (für jede Aufgabe einen eigenen Button!):

a) Erzeuge eine Datei namens *Quadrat*, welche die ersten 10 Quadratzahlen enthält und eine Datei namens *Kubik*, welche die ersten 10 Kubikzahlen enthält!
Zusatzaufgabe: Diese beiden Dateien werden in den folgenden Teilaufgaben teilweise verändert. Damit man aber immer diese Dateien mit identischem obigen Inhalt vorfindet, soll Folgendes gemacht werden: Falls diese Dateien unter diesen Namen schon existieren sollten, sollen sie zuerst gelöscht werden (mit Hilfe der Methoden aus der Klasse File), bevor sie dann neu angelegt werden.

b) Die Elemente der Datei *Quadrat* werden in der Textarea *ta1* angezeigt und die Elemente der Datei *Kubik* werden in der Textarea *ta2* angezeigt.

c) In einem Textfeld wird eine Zahl **n** eingegeben. Es soll überprüft werden, ob diese Zahl **n** in der Datei *Quadrat* enthalten ist oder nicht. Das Ergebnis wird in Form eines ShowMessage-Dialogfensters ausgegeben.

d) In einem Textfeld wird eine Zahl **n** eingegeben. Diese Zahl **n** wird an die Datei *Quadrat* angehängt. Anschließend wird zur Kontrolle diese Datei in der Textarea *ta1* ausgegeben.

e) In einem Textfeld wird eine Nummer **n** eingegeben. Die entsprechende **n**-te Zahl in der Datei *Quadrat* wird durch die Zahl 999 ersetzt. Anschließend wird zur Kontrolle die neue Datei *Quadrat* in der Textarea *ta1* ausgegeben.

f) In einem Textfeld wird eine Zahl **n** eingegeben. Falls diese Zahl in der Datei *Quadrat* enthalten sein sollte, so wird sie durch die Zahl 999 ersetzt. Anschließend wird zur Kontrolle die neue Datei *Quadrat* in der Textarea *ta1* ausgegeben.

g) In einem Textfeld wird eine Nummer **n** eingegeben. Die entsprechende **n**-te Zahl (Nummerierung beginnt bei 0) in der Datei *Quadrat* wird gelöscht.
Leider gibt es hierfür keinen einfachen Element-Löschbefehl. Deshalb muss man folgenden, etwas umständlichen Lösungsweg einschlagen: Alle Zahlen der Datei *Quadrat* mit Ausnahme der **n**-ten werden in eine zweite Datei *Quadrat2* geschrieben. Anschließend wird die erste Datei *Quadrat* gelöscht und die zweite Datei *Quadrat2* umbenannt in *Quadrat*. Beachte die Problematik beim Umbenennen einer Datei!

h) Die beiden Dateien *Quadrat* und *Kubik* werden hintereinander gehängt zu einer dritten Datei namens *Zahlen*. Diese wird in der Textarea *ta1* dargestellt.

i) Die beiden Dateien *Quadrat* und *Kubik* werden (der Größe nach sortiert) zusammengemischt zu einer vierten Datei namens *ZahlenSortiert*. Diese wird in einer Textarea dargestellt.

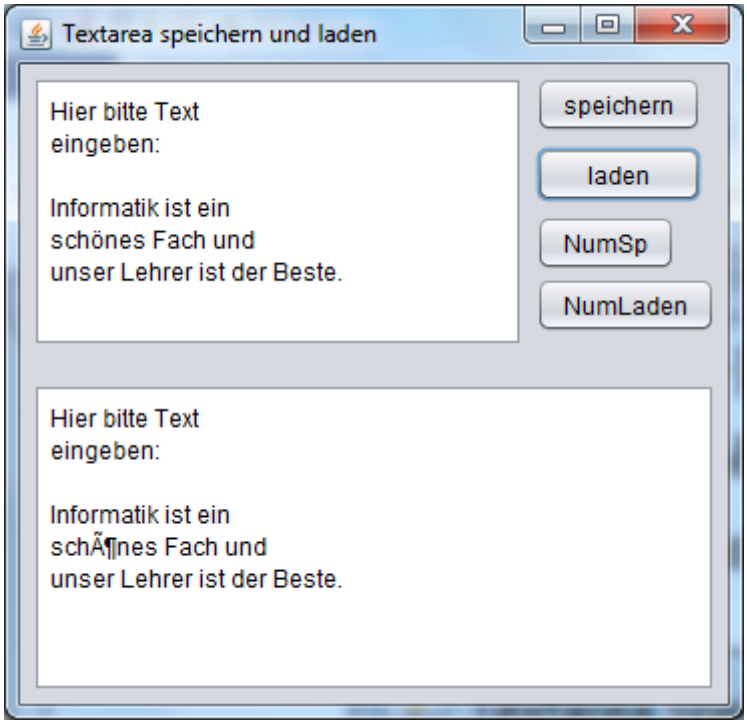
j) Erstelle ein kleines Demo-Programm (z.B. wird nach Anklicken eines Buttons das Wort „Hallo“ in einer Textarea ausgegeben), welches u.a. bei jedem Aufruf des Programms einen Zähler erhöht. Dieser Zähler muss deshalb extern gespeichert werden (in einer Datei namens „Anzahl“. Falls diese Datei noch nicht existiert, wird sie vom Programm erzeugt und mit der Zahl 0 initialisiert. Das Demo-Programm darf nur dreimal laufen. Danach erfolgt eine entsprechende Mitteilung in einem ShowMessage-Dialogfenster an den Benutzer. Wenn der Benutzer dieses Dialogfenster schließt, wird anschließend das Programm beendet (dazu reicht der Befehl **System.exit(0);**).

k) Erstelle eine Datei, welche 20 ganzzahlige Zufallszahlen zwischen 1 und 6 enthält. Schreibe anschließend eine Prozedur, welche in dieser Datei jedes Auftreten der Zahl 5 löscht (vgl. Teilaufgabe g)! Das Funktionieren deines Programms sollte mit zwei Textareas (für die beiden Dateiausgaben) kontrolliert werden können.

l) Erstelle eine Codezahl-Datei, welche nur eine einzige, fünfstellige, natürliche (Integer-) Zahl enthält. Schreibe anschließend ein Programm, welches vom Benutzer die Eingabe dieser geheimen Code-Zahl verlangt. Bei richtiger Eingabe erscheint ein Willkommensgruß, bei falscher Eingabe wird das Programm nach entsprechender Mitteilung an den Benutzer in einem ShowMessage-Dialogfenster sofort geschlossen.

m) 1. Erzeuge eine Textdatei namens *Passwort.txt*, welche nur ein einziges Wort (das Passwort) enthält.
   2. Schreibe ein Programm, welches u.a. dieses Passwort aus der obigen Datei einliest. Der Benutzer wird nach diesem Passwort gefragt. Bei falscher Antwort wird das Programm sofort (nach entsprechender Mitteilung mit Showmessage) geschlossen (dazu reicht der Befehl **System.exit(0);**).
   Bei richtiger Antwort erhält der Benutzer die Möglichkeit, dieses Passwort zu ändern. Die Änderung wird natürlich gespeichert.

n) Speichere den Inhalt einer ersten Textarea als Datei unter dem Namen „Textdatei.txt“. Kontrolliere, ob es funktioniert, indem du anschließend den Inhalt der gerade erzeugten Textdatei in eine zweite Textarea lädst.



Wie man sieht, kann es Probleme geben mit deutschen Umlauten.

o) Eine „bereits existierende“ Textdatei, die mehrere Zeilen enthält, soll so in eine zweite Textdatei kopiert werden, dass alle Zeilen nun vorangestellte Zeilennummern (beginnend mit der Nummer 1) erhalten.

p) Zwei „bereits existierende“ Textdateien sollen hintereinander gehängt werden und so eine dritte Textdatei bilden.

q) In einer „bereits existierenden“ Textdatei besteht jede Zeile nur aus einem einzigen Vornamen. Schreibe ein Programm, welches den Namen *„Klaus“* in dieser Datei sucht. Anschließend soll
a) eine entsprechende Meldung ausgegeben werden.
b) der Name *„Klaus“* gelöscht werden.

Lösungen

```
import java.io.IOException;
import java.io.RandomAccessFile;
import javax.swing.*; //für Dialog-Fenster

public class RandomAccessAufgaben extends javax.swing.JFrame {

RandomAccessFile  f;
RandomAccessFile  g;

…….

private void btaMouseClicked(java.awt…..) {
  File h = new File("Quadrat");
  if (h.exists()) h.delete();

  File k = new File("Kubik");
  if (k.exists()) h.delete();

  try {
     f = new RandomAccessFile("Quadrat", "rw");
     for (int i = 1; i <= 10; i++)  f.writeInt(i*i);
     f.close();

     g = new RandomAccessFile("Kubik", "rw");
     for (int i = 1; i <= 10; i++)  g.writeInt(i*i*i);
     // g wurde nicht geschlossen
  }
  catch(IOException fe) {
     tfAusgabe.setText("Dateifehler beim Erzeugen");
  }
}
```

```
private void btbMouseClicked(java.awt…..) {
   int zahl;
   ta1.setText("");
   ta2.setText("");

   try {
      f = new RandomAccessFile("Quadrat", "r");
      while (f.getFilePointer() < f.length()) {
```

```
            zahl = f.readInt();
            ta1.append(zahl+"\n");
         }
         f.close();

         g.seek(0);    // weil g noch geöffnet ist und der Dateizeiger von g noch am Ende
                       // der Datei steht.
         while (g.getFilePointer() < g.length()) {
            zahl = g.readInt();
            ta2.append(zahl+"\n");
         }
      }
      catch(IOException fe) {
         tfAusgabe.setText("Dateifehler beim Lesen");
      }
}
```

```
private void btcMouseClicked(java.awt…..) {
   int n = Integer.parseInt(tf.getText());
   int zahl;
   boolean enthalten = false;

   try {
      f = new RandomAccessFile("Quadrat", "r");
      while (f.getFilePointer() < f.length()) {
         zahl = f.readInt();
         if (zahl == n) enthalten = true;
      }
      if (enthalten) JOptionPane.showMessageDialog(this, "Die
                     Zahl ist in der Datei Quadrat enthalten");
      else JOptionPane.showMessageDialog(this, "Die Zahl ist
                        in der Datei Quadrat nicht enthalten");
      f.close();
   }
   catch(IOException fe) {
      tf.setText("Dateifehler");
   }
}
```

```
private void btdMouseClicked(java.awt…..) {
   int n = Integer.parseInt(tf.getText());
   try  {
      f = new RandomAccessFile("Quadrat", "rw");
      f.seek(f.length());
      f.writeInt(n);

      int zahl;
      ta1.setText("");
      f.seek(0);
      while (f.getFilePointer() < f.length()) {
         zahl = f.readInt();
         ta1.append(zahl+"\n");
      }
   }
   catch(IOException fe) {
      tf.setText("Dateifehler");
   }
}
```

```
private void btgMouseClicked(java.awt…..) {
   RandomAccessFile k;
   int n = Integer.parseInt(tf.getText());
   try  {
      f = new RandomAccessFile("Quadrat", "r");
      k = new RandomAccessFile("Quadrat2", "rw");

      int zahl;
      while (f.getFilePointer() < f.length()) {
         zahl = f.readInt();
         if (zahl != n) k.writeInt(zahl);
      }
      f.close();
      k.close();
   }
   catch(IOException fe) {
      tf.setText("Dateifehler");
   }

   File alt = new File("Quadrat");
   alt.delete();
   File neu = new File("Quadrat2");
```

```
   neu.renameTo(alt);
}



private void btjMouseClicked(java.awt…..) {
   RandomAccessFile dat;

   File anzahldatei = new File("Anzahl");
   if (! anzahldatei.exists())  {
      try  {
         dat = new RandomAccessFile("Anzahl", "rw");
         dat.writeInt(0);
         dat.close();
      }
      catch(IOException fe) {
         tf.setText("Dateifehler");
      }
   }


   try  {
      dat = new RandomAccessFile("Anzahl", "rw");
      int n = dat.readInt(); // Danach rückt der Dateizeiger auf das nächste
                             // Element
      n++;
      dat.seek(0);
      dat.writeInt(n);
      dat.close();

      if (n<=3) ta1.append("Hallo\n");
      else  {
         JOptionPane.showMessageDialog(this, "Demo-Phase ist
                                                    beendet!");
         System.exit(0);
      }
   }
   catch(IOException fe) {
      tf.setText("Dateifehler");
   }
}
```

Aufgabe m)

```
private void btm1MouseClicked(java.awt…..) {
  RandomAccessFile pwDat;

  File hilf = new File("Passwort.txt");
  if (hilf.exists())  hilf.delete();
  try  {
     pwDat = new RandomAccessFile("Passwort.txt", "rw");
     pwDat.writeUTF("ABCD");
     pwDat.close();
  }
  catch(IOException fe) {
     tf.setText("Dateifehler");
  }
}
```

```
private void btm2MouseClicked(java.awt…..) {
   RandomAccessFile pwDat;
   String wort = "";

   try  {
      pwDat = new RandomAccessFile("Passwort.txt", "rw");
      wort = pwDat.readLine();
      wort = wort.substring(2);  //weil Länge mit gespeichert wurde
      pwDat.close();
   }
   catch(IOException fe) {
      tf.setText("Dateifehler");
   }

   String eingabe;
   eingabe = JOptionPane.showInputDialog(this,"Bitte Passwort
                                                    eingeben!");
   if (! wort.equals(eingabe))   {
      JOptionPane.showMessageDialog(this,"falsches Passwort!");
      System.exit(0);
   }
   else {
      JOptionPane.showMessageDialog(this,"Willkommen");
      int wahl;
      wahl = JOptionPane.showConfirmDialog(this , "Soll
                                  Passwort geändert werden?");
      if (wahl == 0)  {    // 0 = ja
```

```
            String neuPW = "";
            neuPW = JOptionPane.showInputDialog(this,"Bitte neues
                                              Passwort eingeben!");
            File hilf = new File("Passwort.txt");
            hilf.delete();

            try {
               pwDat = new RandomAccessFile("Passwort.txt", "rw");
               pwDat.writeUTF(neuPW);
               pwDat.close();
            }
            catch(IOException fe) {
               tf.setText("Dateifehler");
            }
         }  // von if wahl == 0
      }  // von else
}
```

Aufgabe n):

```
private void btSpeichernMouseClicked(java.awt…..) {
   File h = new File("Textdatei.txt");
   if (h.exists()) h.delete();

   String inhalt;
   try  {
      f = new RandomAccessFile("Textdatei.txt", "rw");
      inhalt = ta1.getText();
      f.writeUTF(inhalt);
      f.close();
   }
   catch(IOException fe) {
      ta2.setText("");
      ta2.setText("Dateifehler");
   }
}


private void btLadenMouseClicked(java.awt…..) {
   String zeile;
   try  {
      ta2.setText("");
      g = new RandomAccessFile("Textdatei.txt", "rw");
```

```
      // Die gesamte obere Textarea wurde als ein einziger String (welcher auch
        Zeilensprünge enthalten kann) gespeichert. Die ersten beiden Bytes dieses einzigen
        Strings enthalten die Länge des gesamten Strings. Deswegen hier diese folgende
        Sonderbehandlung der ersten Zeile. Die Methode readLine liest nur bis zum
        nächsten Zeilensprung.
      zeile = g.readLine();
      zeile = zeile.substring(2);  // Byte 1 und 2  werden entfernt.
      ta2.append(zeile + "\n");
      while (g.getFilePointer() < g.length()) {
         zeile = g.readLine();
         ta2.append(zeile + "\n");
      }
      g.close();
   }
   catch(IOException fe) {
      ta2.setText("");
      ta2.setText("Dateifehler");
   }
}
```

Aufgabe o)

```
private void btoMouseClicked(java.awt…..) {
   String zeilenende = System.getProperty("line.separator");

   File h = new File("TextDatei2");
   if (h.exists()) h.delete();

   String zeile;
   int nummer = 0;
   try  {
      RandomAccessFile k = new
                           RandomAccessFile("TextDatei2","rw");
      g = new RandomAccessFile("Textdatei.txt", "rw");
      // g enthält nur einen einzigen String (mit Zeilenprüngen) und am Anfang die
      // Längenangabe in 2 Bytes.
      zeile = g.readLine();
      zeile = zeile.substring(2);
      nummer++;
      k.writeUTF(nummer + "  " + zeile + zeilenende);
```

```
      while (g.getFilePointer() < g.length()) {
          zeile = g.readLine();
          nummer++;
          k.writeUTF(nummer + "  " + zeile + zeilenende);
      }

      g.close();
      k.close();
   }
   catch(IOException fe) {
      ta2.setText("");
      ta2.setText("Dateifehler");
   }
}
```

Aufgabe o) Teil2: Lesekontrolle

```
private void btNLadenMouseClicked(java.awt…..) {
   String zeile;

   try  {
      ta2.setText("");
      g = new RandomAccessFile("TD2", "rw");

      while (g.getFilePointer() < g.length()) {
          zeile = g.readLine();
          zeile = zeile.substring(2);
          ta2.append(zeile + "\n");
      }
      g.close();
   }
   catch(IOException fe) {
      ta2.setText("Dateifehler");
   }
}
```

Datenströme

Bisher wurden nur externe Dateien, welche primitive Datentypen enthalten, behandelt. Leider konnte man bisher noch keine Objekte mit zusammenhängenden Daten speichern. Beispielsweise enthalten Personendaten üblicherweise mehrere Daten unterschiedlichen Typs, etwa Namen und Vornamen (verschieden lange Strings), Alter (Integer), Geschlecht (Char), Führerscheinbesitz (Boolean) usw. Um derartige Objekte speichern und wieder lesen zu können, benutzt man üblicherweise die beiden Klassen *FileOutputStream* und *FileInputStream* und das Interface *Serializable.*

Um mit der Klasse *FileOutputStream* (eine Unterklasse von *OutputStream*) zu arbeiten, muss sie importiert werden:

import java.io.FileOutputStream;

FileOutputStream(String name) throws *FileNotFoundException*
Dieser Konstruktor erzeugt einen *FileOutputStream,* d.h. er legt die gewünschte Ausgabedatei neu an und setzt den Dateizeiger auf den Anfang der Datei. Falls diese Datei schon existieren sollte, wird sie einfach überschrieben.

FileOutputStream(String name, boolean append) throws *FileNotFoundException*
Dieser zweite Konstruktor setzt, falls *append=true* ist, den Dateizeiger auf das Ende der Datei, falls diese bereits existiert. Andernfalls entspricht das Verhalten dem des ersten Konstruktors.
Ist der Parameter *append* nicht mit *true* belegt, wird der alte Inhalt überschrieben. Die *FileNotFoundException* wirkt vielleicht etwas komisch, wird aber dann ausgelöst, wenn zum Beispiel die Dateiangabe ein Verzeichnis repräsentiert oder die Datei gelockt ist.

public void write(int b) throws IOException
Schreibt das übergebene Byte in den FileOutputStream.

void close() throws *IOException*
Schließt den Datenstrom. Einen bereits geschlossenen Strom noch einmal zu schließen, hat keine Konsequenz.

Beispiel:

```
try {
   FileOutputStream out = new FileOutputStream("Testdat",true);
   for (int i = 0; i < 256; i++)  out.write(i);
   out.close();
}
catch (Exception e) {
     System.err.println(e.toString());
     System.exit(1);  // Parameter 1 bedeutet Fehler
}
```

FileInputStream (eine Unterklasse von *InputStream*) ist der Gegenspieler und dient zum Lesen der Daten. Ein Objekt dieser Klasse bindet eine Datei (etwa repräsentiert als ein Objekt vom Typ File) an einen Datenstrom.

Zunächst muss diese Klasse importiert werden:
import java.io.FileInputStream

FileInputStream(String name) throws *FileNotFoundException*
Erzeugt einen FileInputStream mit einem gegebenen Dateinamen.

void close() throws *IOException*
Schließt den Datenstrom. Einen bereits geschlossenen Strom noch einmal zu schließen, hat keine Konsequenz.

Serialisierung

Unter *Serialisierung* versteht man den Vorgang, ein Objekt einer nicht primitiven Klasse (also keine Integerzahlen oder Strings usw.) so in ein neues Format zu konvertieren, dass man dieses Objekt in eine Datei schreiben kann. Dabei ist natürlich auch der umgekehrte Weg eingeschlossen, also das Lesen eines in serialisierter Form in einer Datei vorliegenden Objekts.

Schreiben von Objekten

Es gibt im Paket *java.io* die Klasse *ObjectOutputStream*, mit der dieser Vorgang sehr einfach zu realisieren ist. Diese Klasse *ObjectOutputStream* besitzt einen Konstruktor, der eine Datei der Klasse *OutputStream* als Argument erwartet:

public ObjectOutputStream(OutputStream out) throws IOException

Die an den Konstruktor übergebene Datei namens *OutputStream* (bzw. genauer: das zugehörige Objekt der entsprechenden Klasse) dient als Ziel der Ausgabe. Hier kann ein beliebiges Objekt der Klasse *OutputStream* oder einer daraus abgeleiteten Klasse übergeben werden. Typischerweise wird ein *FileOutputStream* verwendet, um die serialisierten Daten in eine Datei zu schreiben.

Die Klasse *ObjectOutputStream* besitzt sowohl Methoden, um primitive Typen zu serialisieren, als auch die wichtige Methode *writeObject*, mit der ein komplettes Objekt serialisiert werden kann:

```
public final void writeObject(Object obj) throws IOException
public void writeByte(int data) throws IOException
public void writeInt(int data) throws IOException
public void writeFloat(float data) throws IOException
public void writeDouble(double data) throws IOException
public void writeBytes(String data) throws IOException
public void writeChars(String data) throws IOException
```

Während die Methoden zum Schreiben der primitiven Typen ähnlich funktionieren wie die gleichnamigen Methoden der Klasse *RandomAccessFile*, ist die Funktionsweise von *writeObject* wesentlich komplexer. Die Methode *writeObject* schreibt u.a. folgende Daten in den *OutputStream*:
Die Klasse des als Argument übergebenen Objekts und alle Attribute des übergebenen Objekts inkl. der aus allen Vaterklassen geerbten Attribute. Die Sichtbarkeit eines Attributes hat keinen Einfluss darauf, ob es von *writeObject* serialisiert wird oder nicht.

Wichtig ist weiterhin, dass ein Objekt nur dann mit *writeObject* serialisiert werden kann, wenn es das Interface ***Serializable*** implementiert.

Wir wollen uns zunächst ein Beispiel ansehen. Dazu konstruieren wir eine einfache Klasse *Time*, die eine Uhrzeit, bestehend aus Stunden und Minuten, kapselt:

```
import java.io.*;
```

```
public class Time implements Serializable {
  private int hour;
  private int minute;

  public Time(int h, int m)  {
    hour = h;
    minute = m;
  }

  public String toString()  {
    return hour + ":" + minute;
  }
}
```

Die Klasse *Time* besitzt einen öffentlichen Konstruktor und eine *toString*-Methode zur Ausgabe der Uhrzeit. Mit Hilfe eines Objekts vom Typ *ObjectOutputStream* kann ein Time-Objekt serialisiert werden:

```
import java.io.*;
import java.util.*;
…..
try {
  FileOutputStream fos = new FileOutputStream("test1.ser");
  ObjectOutputStream oos = new ObjectOutputStream(fos);
  Time zeitpunkt = new Time(10,20);
  oos.writeObject(zeitpunkt);
  oos.close();
}
catch (IOException e) {
  System.err.println(e.toString());
}
…….
```

Wir konstruieren zunächst einen *FileOutputStream*, der das serialisierte Objekt in die Datei *test1.ser* schreiben soll. Anschließend erzeugen wir einen *ObjectOutputStrea*m durch Übergabe des *FileOutputStreams* an dessen Konstruktor. Nun wird ein Time-Objekt für die Uhrzeit 10:20 konstruiert und mit *writeObject* serialisiert. Nach dem Schließen des Streams steht das serialisierte Objekt in »test1.ser«.

Wichtig an der Deklaration der Klasse *Time* ist das Implementieren des Serializable-Interfaces.

Lesen von Objekten

Nachdem ein Objekt serialisiert wurde, kann es mit Hilfe der Klasse *ObjectInputStream* wieder rekonstruiert werden.
Die Klasse *ObjectInputStream* besitzt einen Konstruktor, der einen *InputStream* als Argument erwartet, der zum Einlesen der serialisierten Objekte verwendet wird:

public ObjectInputStream(InputStream in)

Analog zu *ObjectOutputStream* gibt es Methoden zum Wiedereinlesen von primitiven Typen und eine Methode *readObject*, mit der ein serialisiertes Objekt wieder hergestellt werden kann:

public final Object readObject() throws OptionalDataException, ClassNotFoundException, IOException
public byte readByte() throws IOException
public char readChar() throws IOException
public int readInt() throws IOException
public long readLong() throws IOException
public float readFloat() throws IOException
public double readDouble() throws IOException

Das folgende Programm zeigt das Deserialisieren am Beispiel des zuletzt serialisierten und in die Datei *test1.ser* geschriebenen *Time*-Objekts:

```
try {
  FileInputStream fis = new FileInputStream("test1.ser");
  ObjectInputStream ois = new ObjectInputStream(fis);
  Time zeitpunkt = (Time) ois.readObject();
  // Beachte die Typumwandlung!
  System.out.println(zeitpunkt.toString());
  ois.close();
}
catch (ClassNotFoundException e) {
    System.err.println(e.toString());
}
catch (IOException e) {
    System.err.println(e.toString());
}
```

Hier wird zunächst ein *FileInputStream* namens ***fis*** für die Datei *test1.ser* geöffnet. Danach wird ein ObjectInputStream-Objekts namens ***ois*** erzeugt, welches sich auf die Datei test1.ser bezieht. Alle lesenden Aufrufe von ***ois*** beschaffen ihre Daten damit aus *test1.ser*. Jeder Aufruf von *ois.readObject* liest immer das nächste gespeicherte Objekt aus dem Eingabestream. Das Programm zum Deserialisieren muss also genau wissen, welche Objekttypen in welcher

Reihenfolge serialisiert wurden, um sie erfolgreich deserialisieren zu können. In unserem Beispiel ist die Entscheidung einfach, denn in der Eingabedatei steht nur ein einziges *Time*-Objekt. *ois.readObject* deserialisiert es und liefert ein neu erzeugtes *Time*-Objekt, dessen Attribute mit den Werten aus dem serialisierten Objekt belegt werden. Die Ausgabe des Programms ist demnach: 10:20

Der folgende Programmausschnitt zeigt, wie Objekte einer Klasse *Person* gespeichert und gelesen werden. Zunächst wird die Klasse *Person* vorgestellt:

```
import java.io.Serializable;

public class Person implements Serializable {
   String nachname;
   String vorname;
   char geschlecht;
   int alter;

   public Person(String n, String v, char ch, int a)  {
       setName(n);
       setVorname(v);
       geschlecht = ch;
       setAlter(a);
   }


   void setName(String n)  {
       nachname = n;
   }

   String getName()  {
       return nachname;
   }

   void setVorname(String v) {
       vorname = v;
   }

   String getVorname()  {
       return vorname;
   }

   void setAlter(int a)  {
       alter = a;
   }

   int getAlter()  {
       return alter;
   }
}
```

Nun folgt der Programmausschnitt, in welchem einige Personenobjekte gespeichert bzw. gelesen werden:

```
import java.io.IOException;
import java.io.FileInputStream;
import java.io.FileNotFoundException;
import java.io.FileOutputStream;
import java.io.ObjectInputStream;
import java.io.ObjectOutputStream;
……
……
private void btPersonenSpeichernMouseClicked(java.awt…..) {
  Person p;
  try  {
    FileOutputStream fos=new FileOutputStream("E:/Objektdatei.ser");
    ObjectOutputStream oos = new ObjectOutputStream(fos);
    p = new Person("Meier", "Tonia", 'w', 12);
    oos.writeObject(p);
    p = new Person("Müller", "Anton", 'm', 15);
    oos.writeObject(p);
    p = new Person("Lehmann", "Petra", 'w', 30);
    oos.writeObject(p);
    // weil der Datenstrom nur einmal zum Schreiben geöffnet wird, wird auch nur einmal
    // die Struktur der Klasse Person in den Datenstrom geschrieben.
    oos.close();
  }
  catch(IOException e) {    //trallala   }
}
```

```
private void btPersonenLadenMouseClicked(java.awt…..) {
  Person p;
  try  {
    FileInputStream fis = new FileInputStream("E:/Objektdatei.ser");
    ObjectInputStream ois = new ObjectInputStream(fis);
    p = (Person) ois.readObject();
    taAusgabe.append(p.nachname + "  " + p.vorname + p.alter +
                                         p.geschlecht + "\n");
    p = (Person) ois.readObject();
    taAusgabe.append(p.nachname + "  " + p.vorname + p.alter +
                                         p.geschlecht + "\n");
    p = (Person) ois.readObject();
    taAusgabe.append(p.nachname + "  " + p.vorname + p.alter +
                                         p.geschlecht + "\n");
```

```
    ois.close();
  }
  catch(Exception e) {      // trallala   }
}
```

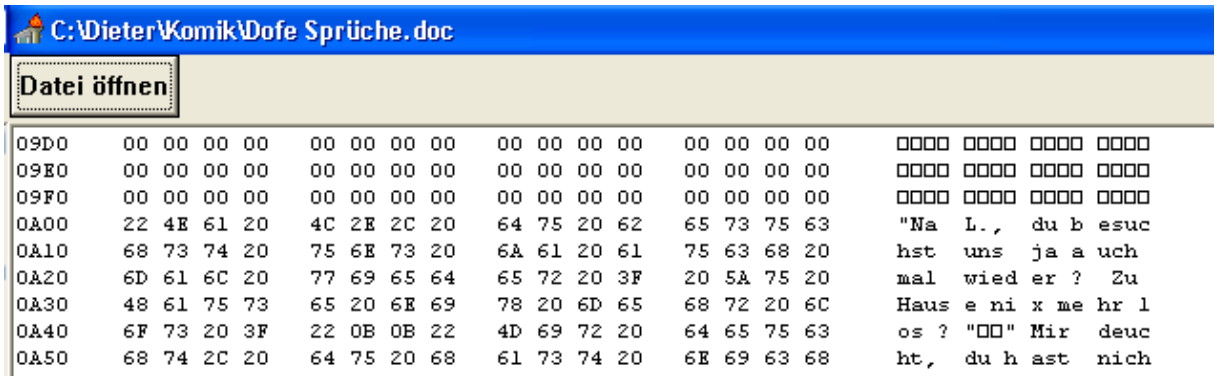
Hinweis: Obiger Datenstrom wurde nur ein einziges Mal zum Schreiben aller Daten geöffnet und auch nur ein einziges Mal zum Lesen aller Daten geöffnet. Außerdem wusste das Programm genau, wie viele Objekte man lesen konnte.

HEX-DUMP

Erstelle ein Java-Programm, mit dem man den Inhalt beliebiger Dateien auflisten kann!

In der folgenden Abbildung werden in jeder Zeile 16 Bytes einer ausgewählten Datei dargestellt. Falls es sich um ASCII-Codes von Buchstaben, Ziffern oder Satzzeichen handeln sollte, werden diese am rechten Rand dargestellt. Ansonsten sieht man am rechten Rand ein rechteckiges Kästchen.

In der linken Spalte ist hexadezimal jeweils die Nummer des ersten Elementes dieser Zeile angegeben.



1. In dieser Aufgabe soll untersucht werden, wie das Java-System Zahlen und Zeichen extern speichert.
   a) Erstelle eine Zahlendatei namens *GanzZahlen.int* ! Diese soll nur die drei int-Zahlen 1, 5 und 7 enthalten.
   b) Erstelle eine Zahlendatei namens *RealZahlen.real* ! Diese soll nur die drei double-Zahlen 1, 5 und 7 enthalten.
   c) Erstelle eine Zeichendatei namens *Zeichen.txt* ! Diese soll nur die drei Zeichen (Typ CHAR) ‚1', ‚5' und ‚7' enthalten.

   Öffne jeweils eine dieser gerade erstellten drei Dateien mit dem HEX-Dump-Programm! Kannst du erkennen, wie die entsprechenden Inhalte gespeichert wurden?

2. Untersuche mithilfe des früher geschriebenen Programms *„Hex-Dump"*, wie eine Textdatei gespeichert wird. Achte insbesondere auf das Zeilenende-Zeichen!