Java

Textdateien durchsuchen

Peter Nussbaumer, Alfred Nussbaumer

Java ist grundsätzlich für grafisch orientierte Anwendungen entwickelt worden; mit dem *Abstract Window Toolkit (awt)* bzw. mit den Swingkomponenten werden Dateizugriffe relativ einfach realisiert. Im Unterricht spielen jedoch kurze, textbasierte Beispiele eine wichtige Rolle. In diesem Beitrag soll die textbasierte Ein- und Ausgabe dargestellt werden.

Java-Applets können aus Sicherheitsgründen keine Dateien lesen oder schreiben. Diese Beschränkungen gelten für Applikationen nicht, sondern die Lese- und Schreibrechte entsprechen den Dateizugriffsrechten des jeweiligen Benutzers.

In den folgenden Beispielen sollen zwei Applikationen vorgestellt werden, die zeilenweise bzw. byteweise Zeichen aus Dateien auslesen, die die derzeit größte bekannte Primzahl enthalten. Die Datei **„prime5.txt"** enthält die Ziffern der Primzahl in Zeilen zu je 80 Zeichen (sie steht unter http://www.mersenne.org zum Download bereit); die Datei **„prime.txt"** enthält alle Ziffern in einer einzigen Bytefolge. Beim Lesen aus einer Datei werfen die verwendeten Methoden so genannte *„Exceptions"*, die eine geordnete Fehlerbehandlung erlauben (man beachte dazu die `try-catch` - Blöcke).

Beispiel: Die Häufigkeit von Ziffern ermitteln (zahlen.java)

```
import java.io.*;

public class zahlen {

  public static void main (String args[]) {
   info();
   try {
     zaehlen();
   }
   catch (Exception e) {
    System.out.println("Datei nicht vorhanden");
   }
  }

  public static void info() {
    System.out.println("Die Häufigkeiten der Ziffern 0,1,2 ... 9 in
der derzeit größten bekannten Primzahl werden ermittelt:");
  }

  public static void zaehlen() throws Exception
  {
    File f = new File("prime5.txt");
    int ergebnis[] = new int[10];
    for (int i=0;i<ergebnis.length;i++)
```

Der SQL-Ausdruck der Abfrage `qryPartnerMann` liefert die Personendaten aus der Tabelle `tblPerson` und aus der Tabelle `tblPartner` dazu noch das Datum der Eheschließung.

```
SELECT tblPerson. ..., tblPartner.dtmVon
FROM tblPartner
INNER JOIN tblPerson
ON tblPartner.lMann = tblPerson.lPerson
WHERE
(((tblPartner.lFrau)=
[Forms]![frmHaupt]![lPerson]))
ORDER BY tblPerson.dtmGeburtstag;
```

In der Abfrage `qryPartnerFrau` werden nur die Felder `lMann` und `lFrau` der Partnerschaftstabelle ausgewechselt.

```
SELECT tblPerson. ..., tblPartner.dtmVon
FROM tblPartner
INNER JOIN tblPerson
ON tblPartner.lFrau = tblPerson.lPerson
WHERE
(((tblPartner.lMann)=
[Forms]![frmHaupt]![lPerson]))
ORDER BY tblPerson.dtmGeburtstag;
```

Zuordnung der Wohnorte

Die Zuordnung der Wohnorte ist prinzipiell gleich wie die Zuordnung von Partnern. Sie ist einfacher, weil lediglich die zentrale Person im Spiel ist und der Vorgang vom Geschlecht unabhängig ist.

Mit der Schaltfläche ***Wohnorte*** wird das Formular `frmWohnort` geöffnet. Beim Laden werden die Datensätze, die den Schlüssel der zentralen Person enthalten, ausgefiltert.

```
Private Sub Form_Load()
  DoCmd.ApplyFilter ,
        "lPerson = Forms!frmhaupt!lPerson"
```

Die Adressen, die zugeordnet werden sollen, müssen zuerst in der Tabelle `tblAdresse` gespeichert sein. Mit der Schalfläche ***Adressen*** öffnen Sie das Formular `frmAdresse` und dort können Sie sie eintragen. Die bereits gespeicherten Adressen können dann auf dem Formular `frmWohnort` mit der Kombobox `cboAdresse` der zentralen Person zugeordnet werden. Dabei wird der Adressenschlüssel `lAdresse` in die Tabelle `tblWohnort` gespeichert. In zwei Textfeldern unter der Kombobox können Sie die Gültigkeit des jeweiligen Wohnortes einschränken.

Anzeige der Wohnorte

Alle der zentralen Person zugeordneten Wohnorte werden auf dem Unterformular `fsubWohnort` in der Listbox `lstWohnort` aufgelistet. Die Listbox ist an die Abfrage `qryWohnort` gebunden.

Der SQL-Ausdruck der Abfrage `qryWohnort` liefert alle Datensätze aus der Tabelle `tblWohnort`, die den Schlüssel der zentralen Person enthalten und alle zugeordneten Adressangaben aus der Tabelle `tblAdresse`.

```
SELECT tblWohnort. ...,
tblAdresse. ...
FROM tblAdresse INNER JOIN tblWohnort
ON tblAdresse.lAdresse = tblWohnort.lAdresse
WHERE
(((tblWohnort.lPerson)=
[forms]![frmHaupt]![lPerson]))
ORDER BY tblWohnort.dtmVon DESC;
```

Ereignisse bearbeiten und anzeigen

Mit der Schaltfläche ***Ereignisse*** wird das Formular `frmEreignis` geöffnet. Beim Laden werden genauso wie bei Wohnorten die Datensätze, die den Schlüssel der zentralen Person enthalten, ausgefiltert.

Der SQL-Ausdruck für die Selektion der Ereignisse, welche in der Listbox `lstEreignis` auf dem Unterformular `fsubEreignis` angezeigt werden sollen, ist direkt in der Eigenschaft Datensatzherkunft eingetragen.

```
SELECT lPerson, sEreignis,dtmVon, dtmBis
FROM tblEreignis
WHERE lPerson=Forms!frmHaupt!lPerson
ORDER BY dtmVon
```

Schlusswort

Das Programm ist ein Beispiel dafür, dass eine gut durchdachte Datenmodellierung auch bei einer relativ komplexen Aufgabe zu einer einfachen und übersichtlichen Lösung führen kann.

alfred.nussbaumer@schule.at Alfred Nussbaumer

```
      ergebnis[i]=0;
    try {
      FileReader fr = new FileReader(f);
      BufferedReader eingabe = new BufferedReader(fr);
      String reihe = eingabe.readLine();
      long zaehler = 0;
      while (reihe != null) {
        for (int i=0; i<reihe.length();i++)
          for (int z=0; z<10; z++)
            if (((int) reihe.charAt(i) - 48) == z)
              ergebnis[z]++;
       reihe = eingabe.readLine();
       zaehler+=1;
       if ((zaehler % 100) == 0)
         System.out.print(".");
      }
    }
    catch (Exception e) {
     System.out.println("Datei nicht vorhanden");
    }

    for (int z=0; z<10;z++)
      System.out.print("\nHäufigkeit von " + z + ": " + ergebnis[z]);
  }
}
```

Beim Einlesen aus der festgelegten Datei liefert die Character-Stream-Klasse `FileReader` einen Datenstrom, der über die Character-Stream-Klasse `BufferedReader` zeilenweise gelesen wird. So lange die eingelesene Zeichenkette nicht „`null`“ ist, werden die einzelnen Bytes analysiert und die entsprechenden Zähler im Ergebnis-Array inkrementiert. Der Zeilenzähler `zaehler` dient lediglich dazu, jede 100. Eingabezeile einen Punkt auf der Konsole auszugeben.

Beispiel: Ziffernfolgen („Muster“) suchen (muster.java)

Wie im ersten Beispiel nachgeprüft werden kann, liefern große Primzahlen alle Ziffern etwa gleich oft. Eine reizvolle Aufgabe besteht nun darin, bestimmte Ziffernfolgen zu suchen. Deshalb verwenden wir in diesem Beispiel eine Datei, die die derzeit größte Primzahl als Folge von Bytes enthält (die Zeilenenden wurden entfernt). Die Byte-Stream-Klasse `RandomAccessFile` erlaubt den Zugriff auf jedes einzelne Byte.

```
import java.io.*;

public class muster {

  public static void main (String args []) throws IOException{
    char l=0;
    byte b;
    int anz=args[0].length();
    long pos=0;
    String vergl;
    RandomAccessFile datei = new RandomAccessFile("prime.txt","r");
    do {
      try {
        datei.seek(pos);
        vergl="";
        for (int j=0;j<anz;j++) {
          l=(char) datei.readByte();
          vergl=vergl+l;
        }
        if (vergl.equals(args[0]))
          System.out.println
            (args[0]+" an der "+pos+".ten Stelle gefunden!");
        pos++;
        if (pos % 100000 == 0)
          System.out.println("--->" + pos);
      }
      catch (EOFException e) {
        l=0;
      }
    } while (l != 0);
    datei.close();
  }
}
```

Die Methode `seek()` stellt den Dateizeiger zum Lesen (bzw. Schreiben) auf die in Bytes angegebene Stelle (`pos`). Mit der Methode `readByte()` wird das dort stehende Byte gelesen. Die gelesenen Bytes werden zu einem String verkettet, der schließlich mit dem beim Programmaufruf festgelegten Muster verglichen wird.

Die Ausgabe im letzten Beispiel lässt sich deutlich beschleunigen, wenn das Lesen und Vergleichen der Zeichenkette abgebrochen wird, sobald das erste Zeichen keine Übereinstimmung mehr ergibt:

```
import java.io.*;

public class muster_1 {

  public static void main (String args []) throws IOException{
    char l=0;
    byte b;
    int anz=args[0].length();;
    long pos=0;
    boolean ok;
    int i;
    String vergl;
    RandomAccessFile datei = new RandomAccessFile("prime.txt","r");
    do {
      try {
        datei.seek(pos);
        ok=true;
        i=0;
        do {
          l=(char) datei.readByte();
          if (l != args[0].charAt(i))
            ok = false;
          i++;
        } while (i<anz && ok);
        if (ok)
          System.out.println
            (args[0]+" an der "+pos+".ten Stelle gefunden!");
        pos++;
        if (pos % 100000 == 0)
          System.out.println("--->" + pos);
      }
      catch (EOFException e) {
        l=0;
      }
    } while (l != 0);
    datei.close();
  }
}
```

Anschließende Projekte für den Unterricht:

1. Das Bestimmen der Häufigkeiten von Zeichen hat eine interessante Bedeutung in der Kryptoanalyse: Mit einem ähnlichen Programm wie `zahlen.java` lässt sich die Häufigkeit von Zeichen in einem Geheimtext ermitteln. Dies lässt bei unsicheren Verfahren Rückschlüsse auf den verwendeten Verschlüsselungsalgorithmus zu (Beispiele zur Kryptographie werden in einem späteren Beitrag behandelt).

2. Für die zur Zeit längste Primzahl ergeben sich die folgenden Ziffernhäufigkeiten:

   Häufigkeit von 0: 405083
   Häufigkeit von 1: 405614
   Häufigkeit von 2: 405068
   Häufigkeit von 3: 405928
   Häufigkeit von 4: 405491
   Häufigkeit von 5: 404915
   Häufigkeit von 6: 405154
   Häufigkeit von 7: 405308
   Häufigkeit von 8: 406672
   Häufigkeit von 9: 404713

   Die Wahrscheinlichkeit für das Auftreten einer Ziffer ist demnach etwa 0,1. Das Programm `muster.java` soll so erweitert werden, dass die Häufigkeiten für die Ziffernfolgen 11, 111, 1111, etc. ermittelt werden. Stimmen die Ergebnisse mit den erwarteten Wahrscheinlichkeiten überein?

3. Für die obigen Programme sind GUI-Applikationen (unter der Verwendung von AWT- oder Swingkomponenten) zu erstellen. Die berechneten Häufigkeiten sind als Balkendiagramme darzustellen.