Di., 29.9.2009 Informatik 12 M (Kr)

1. Klausur 12/I (A)

Dauer: 4 Schulstunden Name: ___www.r-krell.de___

Hilfsmittel: normaler Taschenrechner,

* *Achte auf sorgfältige Darstellung mit vollständigem, nachvollziehbarem Lösungsweg!* *
* *Kommentiere deine Programme!* *

1 Einfache Sortierverfahren in einer Reihung ganzer Zahlen:

a) Sortiere die *anzahl=6* Zahlen **43 25 12 18 32 41** von Hand (bitte Umspeicherungen bzw. Vertauschungen durch Pfeile deutlich machen und den Inhalt der Reihung nach jedem Durchgang vollständig hinschreiben! Unterstreiche dort den jeweils sicher sortierten Teil. Notiere alle vom Programm ausgeführten Durchgänge, auch wenn die Zahlen schon richtig stehen!)
   a1) mit dem BubbleSort-Verfahren (Sortieren durch Austausch von Nachbarn)
   a2) mit dem EinSort-Verfahren (Sortieren durch Einfügen)

b) Schreibe für die Klasse *ZahlenVw* die Methode *tausche* in Java (für die Reihung *reihe* ganzer Zahlen)!

```
public class ZahlenVw
{
 int[] reihe = int[50];
 int anzahl; //Füllgrad von reihe

 private void tausche (int i, int j)
 {
```

c) Beim Bubblesort-Verfahren in a1) wurde in den letzten Durchgängen nichts mehr vertauscht, sodass hier das Verfahren schon früher hätte beenden können. Schreibe die Methode *public void bubbleSortV2()* in Java, die keine feste Zahl von Durchgängen ausführt, sondern nur genau dann endet, wenn im vorigen Durchgang keine Vertauschungen mehr vorgenommen wurden. (Tipp: Benutze eine lokale Variable *boolean getauscht*. Und: Ganze Zahlen können z.B. mit *if (reihe[i] < reihe[j]) ...* verglichen werden!). Gib außerdem mit kurzer Begründung an, ob das neue BubbleSort auch aufwändiger sein kann als das alte!

d) Gegeben ist nebenstehender Programmtext. Führe einen Bleistifttest mit den Werten aus a) durch und beschreibe, an welches der uns bekanntenVerfahren dieser Algorithmus erinnert (mit kurzer Begründung). Vergleiche auch den Aufwand beider Version! Was ist besser?

```
for (int durchgang=0; durchgang<anzahl-1; durchgang++)
{
 for (int i=durchgang+1; i<anzahl; i ++)
 { if (reihe[durchgang] > reihe[i]) { tausche (durchgang, i); }
}
}
```

e) Auch beim Lotto werden die sechs gezogenen ganzen Zahlen nachher aufsteigend sortiert genannt.
   e1) Erläutere in Deutsch, wie die drei Teile Vorbereitung, Ziehung und Ausgabe arbeiten und wieso am Ende die sechs Lottozahlen sortiert erscheinen, wo es doch weder Vergleiche noch Vertauschungen gibt.
   e2) Offenbar hat das Simulationsprogramm doch noch Macken: Ganz gelegentlich gehen Zahlen verloren – jedenfalls werden ab und zu weniger als sechs Lottozahlen ausgegeben. Erkläre und verbessere entsprechend!
   e3) Weil keine geschachtelten *for*-Schleifen auftreten, würde sich der Aufwand bei Ziehung von 12 statt 6 Zahlen höchstens verdoppeln -- nicht vervierfachen. Ist das Verfahren daher besser als alle unsere Sortierverfahren und sollte auch auf das Sortieren von Kontakten umgeschrieben werden? Überlege und erkläre!

```
public void lottoSimulation ()
{
 int nummerAufKugel;
 boolean[] gezogen = new boolean[50];

 for (int i=1; i<=49; i++) // Vorbereitung
 {
  gezogen [i] = false;
 }

 for (int j=0; j<6; j++)   // 6 Ziehungen
 {
  nummerAufKugel = Hilfe.zufall(1, 49);
  gezogen[nummerAufKugel] = true;
 }

 for (int i=1; i<=49; i++) // Ausgabe
 {
  if (gezogen[i] == true)
  {
   System.out.print(" "+i);
  }
 }
}
```

2 Im Supermarkt werden die Artikel per Computer verwaltet („Warenwirtschaftssystem“) .

a) Von jedem Artikel sind die ganzzahlige europäische Artikelnummer (EAN), die Artikelbezeichnung (also der Name), der Bestand (=Anzahl der vorrätigen Artikel dieses Typs) und der Verkaufspreis (VK) gespeichert. Schreibe den Anfang einer passenden Java-Klasse mit allen Datenfeldern (=Attribute =globale Variable) und einem Konstruktor, der die Datenfelder füllt.

b) Das gesamte Sortiment umfasst *artikelZahl* verschiedene Artikel und soll in einer Reihung *reihe* verwaltet werden; die *artikelZahl* bleibt immer unter 4000. Schreibe auch für das Sortiment den Anfang einer geeigneten Klasse mit globalen Variablen sowie der Methode *public void rein (Artikel neuerArtikel)*, die einen bereits erzeugten und mit Werten gefüllten neuen Artikel ins Sortiment aufnimmt – und zwar hinten auf dem nächsten freien Platz der Reihung!

c) Beim Sortieren wird eine Methode *vergleicheMit* benutzt.

c1) In welche Klasse sollte diese Methode am besten? (Antwort mit kurzer Begründung)

c2) Schreibe die Methode *vergleicheMit* in Java, wenn die Schriftzeichen ‘e’, ‘n’ oder ‘g’ das Sortierkriterium bestimmen – für die Sortierung nach der EAN-Nummer, nach dem Namen oder nach dem Gesamtwert aller gerade vorrätigen Artikel dieses Typs (so haben bei einem Bestand = 20 alle 20 Dosen „Erasco grüne Erbsen“ à 1.48 € den Gesamtwert 29.60).

d) Wenn immer nur nach dem gleichen Kriterium (z.B. immer nur nach der EAN) absteigend sortiert sein soll (Artikel mit der höchsten EAN links, kleinste ganz rechts), kann schon direkt bei der Eingabe auf die richtige Ordnung geachtet werden. Statt einen neuen Artikel wie in b) hinten in die Reihung einzufügen, könnte man ihn direkt an der richtigen Stelle in die bereits geordnete Reihung einfügen (und muss einige Inhalte in der Reihung schieben):

d1) Schreibe eine passende Methode *public void sortRein (Artikel neuerArtikel)*

d2) Schätze ab: Wie groß ist hier der maximale Aufwand zum Erstellen einer Reihung aus *artikelZahl* Artikeln? (Anzahl der Vergleiche und Anzahl der Umspeicherungen *))

e) Wenn die Reihung nach Artikelnummern sortiert ist, kann besonders schnell nach einer Artikelnummer gesucht werden.

e1) Notiere in Java die binäre Suche aus dem Unterricht, angepasst auf die absteigende Sortierung der Artikel, wenn -1 oder der Index des gefundenen Artikels innerhalb der *reihe* zurückgegeben werden soll.

e2) An der binären Suche aus dem Unterricht könnte stören, dass innerhalb der while-Schleife zwei Vergleiche bzw. *if*-Abfragen durchgeführt werden. Deshalb wird die nebenstehende Version vorgeschlagen. Teste mit der Suche nach (1) Artikel mit EAN=18, (2) Artikel mit EAN=42 in der Reihung mit den Artikelnummern 43 41 32 25 18 12 von Hand und erläutere, ob/warum das Verfahren (nur bei bestimmten Suchvorgängen nicht?) klappt!

```
public int suche (Artikel suchwert, char kriterium)
{                                                  // zu Aufgabe 2 e2)
 int li = 0;    int re = artikelZahl-1;    int mi;
 while (li<re)
 {
  mi = (li + re) / 2;
  if (suchwert.vergleicheMit(reihe[mi], kriterium) <= 0)
    {  li = mi;  }  // nicht = mi+1! rechts weiter suchen
   else
    {  re = mi - 1;  }  // große Werte links weiter suchen
 }
 if (suchwert.vergleicheMit(reihe[li], kriterium)==0)
 {  return (li); }  else {  return (-1); }
}
```

f) Ab sofort soll das Sortiment doch wieder aufsteigend statt bisher absteigend geordnet werden, d.h. vorne in der Reihung soll künftig der Artikel mit der kleinsten EAN stehen. Da bisher schon geordnet war, braucht man nicht wirklich neu sortieren, sondern kann mehrfach *tausche* aufrufen. Schreibe die Methode *kehreUm* und nenne wieder den Aufwand*) in Abhängigkeit von der *artikelZahl*.

3 Alle Supermarkt-Artikel aus Aufgabe 2 sind auch – noch absteigend nach Artikelnummern geordnet – in der Datei „alle.art“ auf der Festplatte gespeichert. Im Herbst werden einige Saison-Artikel aus dem Sortiment genommen. Dazu schickt die Konzernzentrale eine Datei „raus.ean“, in der aber nur die Nummern(!) aller zu entfernenden Artikel stehen – zum Glück ebenfalls absteigend geordnet. Entwickle einen Algorithmus (mit kurzer Beschreibung in Deutsch, einem Struktogramm des Vergleichs- bzw. Entferne-Vorgangs und dann mit vollständigem, kommentiertem Java-Text) für eine Methode *public void bereinigeSortiment()*, die die beiden genannten Dateien zum Lesen öffnet und die neue Datei „winter.art“ der auch im Winter noch verfügbaren Artikel auf die Festplatte schreibt. Es darf/soll darauf vertraut werden, dass alle Dateien mindestens einen Eintrag enthalten und nur vorher vorhandene Artikel raus sollen! Nenne auch den Aufwand der Bereinigung*).

---

*) Hier interessiert begründet, ob der Aufwand konstant ist oder linear mit der *artikelZahl* oder etwa proportional zur Wurzel aus der Artikel-Zahl, zu $log_2(artikelZahl)$, zu $artikelZahl^2$ o.ä. steigt oder fällt.

Hilfe: Typische Konstrukte für das Schreiben und Lesen von (Objekt-)Dateien

Schreiben (auf Festplatte/in Datei)

```
try
{
 ObjectOutputStream datei = new ObjectOutputStream(
   new FileOutputStream ("name.dat"));

 //folgenden Befehl wiederholen, um mehrere Objekte zu
 //schreiben (artikel sei ein Objekt vom Typ Artikel)

 datei.writeObject (artikel); //schreibt ein Objekt



 datei.close();
}
catch (IOException ex)
{..}
```

Lesen (von Festplatte/aus Datei)

```
try  //Fehler, falls Datei nicht vorhanden oder close ver-
sagt
{
 ObjectInputStream datei = new ObjectInputStream(
   new FileInputStream ("name.dat"));
 try //Fehler, wenn Datei zu Ende
 {
   //folgenden Lesebefehl immer wieder wiederholen:
   artikel = (Artikel) datei.readObject();  //liest Objekt
 }
 catch (EOFException eex) //EOF = End of File =..
 {                        //..= Dateiende
   datei.close();  //Datei schließen, da zu Ende
 }
}
catch (Exception ex) //für IO- oder ClassNotFound-Ex.
{..}
```

Die Art der Wiederholstrukturen richtet sich danach, wie die einzelnen Komponenten zur Verfügung stehen. Passende Variable (wie *artikel*) müssen vorher definiert (und vorm Schreiben mit Inhalt gefüllt) sein, während sie beim Lesen gefüllt werden. -- Zu Beginn der Klasse ist **import java.io.*;** nötig (und die verwendete *Artikel*-Klasse muss den Zusatz **implements java.io.Serializable** erhalten).