

Datei „IntSortReihe3.java“

```
1 // Verschiedene einfache Sortierverfahren + BucketSort
2 // Hier nur noch Sortierfunktionalität; Ein-/Ausgabe bzw. Menüs sind
3 // ausgelagert in eigene Klasse IntSortGUI3 (weiter unten abgedruckt)
4 // Java jdk 1.1.18 -- R. Krell, 12.10.01
5
6 import stiftUndCo.*; // für Zufallsgenerator
7
8 public class IntSortReihe3
9 {
10 // Deklaration global benötigter Objekte und der Variablen erzeugt
11 int[] reihung, kopie;
12 boolean erzeugt = false;
13
14 public void erzeuge()
15 // Initalisierungsmethode: erzeugt alle benötigten Objekte einmal
16 {
17     if (! erzeugt) // kein neues Erzeugen bei versehentlichem Mehrfachaufruf
18     {
19         reihung = new int[11]; // nimmt die zu sortierenden Zahlen auf
20         kopie = new int[11]; // Kopie der reihung, um gleiche Zahlen mit versch.
21                               // Methoden sortieren zu können
22         erzeugt = true;
23     }
24 }
25
26 public void aufsteigendFüllen () // jetzt public für Aufruf von außerhalb der Klasse
27 // füllt die reihung (und kopie) so, dass zwischen Nachbarn <= gilt
28 {
29     reihung[0] = Hilfe.zufall(10,20);
30     for (int i=1; i < reihung.length; i++)
31     {
32         reihung[i] = reihung [i-1] + Hilfe.zufall(0,6);
33         kopie[i] = reihung[i];
34     }
35 }
36
37 public void absteigendFüllen ()
38 // füllt die reihung (und kopie) so, dass zwischen Nachbarn >= gilt
39 {
40     reihung[0] = Hilfe.zufall(80,100);
41     for (int i=1; i < reihung.length; i++)
42     {
43         reihung[i] = reihung [i-1] - Hilfe.zufall(0,6);
44         kopie[i] = reihung[i];
45     }
46 }
47
48 public void zufälligFüllen ()
49 // füllt die reihung (und kopie) mit völlig zufälligen Werten zwischen 10 und 99
50 {
51     for (int i=0; i < reihung.length; i++)
52     {
53         reihung[i] = Hilfe.zufall(10,100);
54         kopie[i] = reihung[i];
55     }
56 }
57
58 public boolean istSortiert()
59 // wird wahr, wenn die reihung aufsteigend sortiert ist
60 {
61     boolean okay = true;
62     int stelle = 0;
63     do
64     {
```

```

65     if (reihung[stelle] > reihung[stelle+1])
66     {
67         okay = false;
68     }
69     stelle = stelle + 1;
70 } while (okay && (stelle < reihung.length-1));
71 return (okay);
72 }
73
74 public void wiederherstellen()
75 // füllt die beim Füllen vorsorglich angelegte kopie in die evtl. durch
76 // Sortieren inzwischen veränderte reihung
77 {
78     for (int i=0; i<reihung.length; i++)
79     {
80         reihung[i] = kopie[i];
81     }
82 }
83
84 public String zeigeReihe ()
85 // erzeugt eine Zeichenkette mit allen Elementen der Reihung.
86 // Schreibt nicht mehr direkt auf den Bildschirm wie altes zeigeAlle!
87 {
88     String zchnkette = new String();
89     zchnkette = "Reihung: ";
90     for (int i=0; i < reihung.length; i++)
91     {
92         zchnkette = zchnkette + " " +reihung[i];
93     }
94     return (zchnkette);
95 }
96
97 private void tausche (int i, int j)
98 // vertauscht innerhalb der reihung die beiden Inhalte an den Positionen i und j
99 // Methode wird von den meisten Sortierverfahren (s.u.) benutzt
100 {
101     int zwischen; // hier int -- immer vom Typ der Komponenten der reihung!
102
103     zwischen = reihung[i];
104     reihung[i] = reihung[j];
105     reihung[j] = zwischen;
106 }
107
108 public void bubbleSort() // BubbleSort, 1. Version
109 // Sortieren durch Vertauschen unmittelbarer Nachbarn
110 // beim ersten Durchgang "perlt" die größte Zahl nach hinten, so dass
111 // spätere Durchgänge immer ein Element früher enden können und so
112 // der sortierte Teil von hinten nach vorne wächst
113 {
114     int durchgang, stelle;
115
116     for (durchgang=1; durchgang<reihung.length; durchgang++)
117     // hier mit fester Durchgangszahl = Elementzahl - 1
118     {
119         for (stelle=0; stelle<reihung.length-durchgang; stelle++)
120         // Durchgänge beginnen immer bei 0, enden aber immer früher
121         {
122             if (reihung[stelle] > reihung[stelle+1])
123             {
124                 // Nachbarn in falscher Reihenfolge werden vertauscht
125                 tausche (stelle, stelle+1);
126             }
127         }
128     }
129 }
130
131 public void bubbleSort2() // BubbleSort, verbesserte Version
132 // Sortieren durch Vertauschen unmittelbarer Nachbarn

```

```

133 // wie vor, allerdings: wurde in einem Durchgang nichts mehr vertauscht,
134 // so sind keine weiteren Durchgänge nötig und es ist alles sortiert.
135 // NB: War anfangs hinten die kleinste Zahl, sind alle Durchgänge nötig!
136 {
137     int durchgang, stelle;
138     boolean vertauscht;
139
140     durchgang=1;
141     do {
142         vertauscht = false; // in diesem Durchgang wurde noch nicht getauscht
143         for (stelle=0; stelle<reihung.length-durchgang; stelle++)
144             {
145                 if (reihung[stelle] > reihung[stelle+1])
146                 {
147                     tausche (stelle, stelle+1);
148                     vertauscht = true; // jetzt hat ein Tausch stattgefunden
149                 }
150             }
151         durchgang = durchgang + 1;
152     } while (vertauscht && (durchgang<reihung.length));
153     // höchstens so viele Durchgänge wie beim normalen bubbleSort;
154     // aber vorzeitiges Ende, wenn zuletzt nichts mehr vertauscht wurde
155 }
156
157
158 public void shakerSort() // ShakerSort
159 // wie BubbleSort, aber mit Durchgängen abwechselnd in beide Richtungen:
160 // beim Durchgang nach rechts "perlt" die größte Zahl nach oben
161 // beim Durchgang nach links "perlt" jetzt die kleinste Zahl nach unten,
162 // so dass möglicherweise noch mehr Durchgänge gespart werden können
163 {
164     int durchgang, stelle;
165     boolean vertauscht;
166
167     durchgang=1;
168     do {
169         vertauscht = false;
170         // Durchgang nach rechts:
171         for (stelle=durchgang-1; stelle<reihung.length-durchgang; stelle++)
172             {
173                 if (reihung[stelle] > reihung[stelle+1])
174                 {
175                     tausche (stelle, stelle+1);
176                     vertauscht = true;
177                 }
178             }
179         if (vertauscht)
180         {
181             // falls noch nötig, Durchgang nach links
182             vertauscht = false;
183             for (stelle=reihung.length-durchgang-2; stelle>=durchgang-1; stelle--)
184                 {
185                     if (reihung[stelle] > reihung[stelle+1])
186                     {
187                         tausche (stelle, stelle+1);
188                         vertauscht = true;
189                     }
190                 }
191         }
192         durchgang = durchgang + 1; // Hin- und Zurück = 1 (Doppel-)Durchgang
193     } while (vertauscht && (durchgang<=reihung.length/2));
194 }
195
196 public void minSort() // MinSort
197 // Sortieren durch Auswahl des jeweils kleinsten Elements im Rest
198 // und Tausch dieses Elements nach vorne. Nach dem 0. Durchgang steht
199 // also das kleinste Element ganz links auf Platz 0, beim nächsten Durchgang
200 // kommt das nächstkleinere Element auf Platz 1 usw., so dass die Sortierung

```

```

201 // von links nach rechts wächst
202 {
203     int minStelle, durchgang, stelle;
204
205     for (durchgang=0; durchgang<reihung.length-1; durchgang++)
206     {
207         minStelle = durchgang; // vorderstes Element im unsortierten Teil zum Vergleich
208         for (stelle=durchgang+1; stelle<reihung.length; stelle++)
209         {
210             if (reihung[stelle] < reihung[minStelle])
211             {
212                 // wird ein kleineres Element gefunden, so wird dessen Index gemerkt
213                 minStelle = stelle;
214             }
215         }
216         // das gefundene kleinste Element wird nach vorne (auf den Platz mit
217         // dem Index durchgang) getauscht, sofern es nicht schon dort steht:
218         if (minStelle > durchgang)
219         {
220             tausche (durchgang, minStelle);
221         }
222     }
223 }
224
225 public void einSort() // EinSort
226 // in den hinteren, sortierten Teil der reihung wird jeweils das letzte Element
227 // davor richtig eingefügt, wodurch der sortierte Teil nach vorne wächst.
228 // Damit Platz zum Einfügen ist, werden die sortierten Elemente solange
229 // um eins nach vorne gezogen, bis das neue Element in die Lücke gehört.
230 {
231     int sortAb, stelle;
232     int zwischen; // hier int -- immer vom Typ der Komponenten der Reihung
233     boolean aufhören;
234
235     sortAb = reihung.length-1; // anfangs gilt nur das letzte Element als sortiert
236     do
237     {
238         zwischen = reihung[sortAb-1]; // letztes unsortiertes Element rausnehmen
239
240         aufhören = false;
241         stelle = sortAb;
242         do
243         {
244             if (zwischen > reihung[stelle])
245             {
246                 reihung[stelle-1] = reihung[stelle]; //Nachfolgende Inhalte nach vorne schieben
247                 stelle = stelle + 1;
248             }
249             else
250             {
251                 aufhören = true;
252             }
253         } while ((aufhören==false)&&(stelle<reihung.length));
254
255         reihung[stelle-1] = zwischen; // Element an richtiger Stelle rein
256         sortAb = sortAb - 1; // sortierter Teil ist um eins nach vorne gewachsen
257     } while (sortAb > 0); // äußere do-Schleife
258 }
259
260
261 public void bucketSort() // Bucketsort
262 // Kein einfaches Sortierverfahren, da zusätzlicher Speicherplatz für kontrollle
263 // nötig. Zählt in kontrollle, ob/wie oft ein Element der reihung auftrat und
264 // gibt dann in Schleife 3 die Sortierung aus.
265 // Wichtig: Verfahren funktioniert nur, wenn die in der Reihung auftretenden
266 // Werte im Indexbereich von kontrollle -- hier also zwischen 0 und 199 -- liegen!
267 // Dafür wächst der Zeitaufwand höchstens linear mit reihung.length!
268 {
269     int[] kontrollle = new int[200];

```

```

270     int stelle;
271
272     for (int index=0; index<kontrolle.length; index++)
273     { //Schleife 1: Initialisiert kontrolle
274         kontrolle[index] = 0;
275     }
276     for (stelle=0; stelle<reihung.length; stelle++)
277     { //Schleife 2: zählt Werte aus reihung
278         kontrolle [reihung[stelle]] = kontrolle [reihung[stelle]] + 1;
279     }
280     stelle = 0;
281     for (int index=0; index<kontrolle.length; index++)
282     { //Schleife 3: füllt Sortierung in die reihung
283
284         for (int zähler=1; zähler<=kontrolle[index]; zähler++)
285         { //mehrfach gezählte Werte müssen auch
286             reihung[stelle] = index; //wieder mehrfach in die reihung!
287             stelle = stelle + 1;
288         }
289     }
290 }
291
292 }

```

Datei „IntSortGUI3.java“

```

1 // Ein-/Ausgabe-Teile zur Bedienung von IntSortReihe3
2 // (=Sortieren ganzer Zahlen) über eine grafische Benutzeroberfläche ('GUI').
3 // Java jdk 1.1.18 -- R. Krell, 12.10.01
4
5 import java.awt.*;
6 import java.awt.event.*;
7
8 public class IntSortGUI3 extends Frame // Frame ist Fensterklasse
9 {
10     IntSortReihe3 reihe; // Objekt für Funktionalität definieren
11     Label ausgabezeile; // Objekt für Ausgabe definieren
12
13     public void starte() // Fenster initialisieren und beschreiben
14     {
15         //WindowsListener hinzufügen, damit Schließknopf funktioniert
16         addWindowListener (
17             new WindowAdapter ()
18             {
19                 public void windowClosing (WindowEvent ereignis)
20                 { //ersetzt bisher leere Methode
21                     setVisible (false);
22                     dispose();
23                     System.exit(0);
24                 }
25             }
26         ); // runde Klammer vom Windowlistener geschlossen;
27
28         // Fenster mit Titel versehen
29         setTitle("Sortieren ganzer Zahlen");
30
31         // Fenstergröße festlegen
32         setSize (420,400);
33
34         // Knöpfe definieren und beschriften
35         Button btNeuA = new Button ("Neu-aufsteigend");
36         Button btNeuB = new Button ("Neu-absteigend");
37         Button btNeuZ = new Button ("Neu-zufällig");
38         Button btSort1 = new Button ("BubbleSort");
39         Button btSort2 = new Button ("BubbleSort2");

```

```

40 Button btSort3 = new Button ("ShakerSort");
41 Button btSort4 = new Button ("MinSort");
42 Button btSort5 = new Button ("EinSort");
43 Button btSort6 = new Button ("BucketSort");
44 Button btPrüf = new Button ("Sortierung prüfen");
45 Button btOrig = new Button ("Originalreihe wieder herstellen");
46 Button btEnde = new Button ("Programm beenden");
47
48 // Layout: Ausgabefenster und Knöpfe anordnen
49 ausgabezeile = new Label(); // wird später je nach Knopfdruck
50 // mit Ausgabe(-text) gefüllt
51 ausgabezeile.setAlignment (Label.CENTER);
52
53 Panel tastenreihe = new Panel();
54 tastenreihe.setLayout (new FlowLayout (FlowLayout.CENTER,20,20));
55 tastenreihe.add (btNeuA);
56 tastenreihe.add (btNeuB);
57 tastenreihe.add (btNeuZ);
58 tastenreihe.add (btSort1);
59 tastenreihe.add (btSort2);
60 tastenreihe.add (btSort3);
61 tastenreihe.add (btSort4);
62 tastenreihe.add (btSort5);
63 tastenreihe.add (btSort6);
64 tastenreihe.add (btPrüf);
65 tastenreihe.add (btOrig);
66 tastenreihe.add (btEnde);
67
68 setLayout (new GridLayout (2,1)); // Fenster horizontal halbieren
69 add (ausgabezeile); // oben: Ausgabe
70 add (tastenreihe); // unten: alle Knöpfe wie vorstehend angegeben
71
72 setVisible(true);
73
74 //ganz oben definierte Sortierklasse erzeugen und initialisieren
75 reihe = new IntSortReihe3();
76 reihe.erzeuge();
77
78 //Funktion der Knöpfe festlegen
79
80 btNeuA.addActionListener (
81     new ActionListener ()
82     {
83         public void actionPerformed (ActionEvent e)
84         {
85             reihe.aufsteigendFüllen();
86             ausgabezeile.setText (reihe.zeigeReihe());
87         }
88     }); // runde Klammer (von addActionListener)
89
90 btNeuB.addActionListener (
91     new ActionListener ()
92     {
93         public void actionPerformed (ActionEvent e)
94         {
95             reihe.absteigendFüllen();
96             ausgabezeile.setText (reihe.zeigeReihe());
97         }
98     }); // runde Klammer (von addActionListener)
99
100 btNeuZ.addActionListener (
101     new ActionListener ()
102     {
103         public void actionPerformed (ActionEvent e)
104         {
105             reihe.zufälligFüllen();
106             ausgabezeile.setText (reihe.zeigeReihe());
107         }
108     }); // runde Klammer (von addActionListener)

```

```

109
110 btSort1.addActionListener (
111     new ActionListener ()
112     {
113         public void actionPerformed (ActionEvent e)
114         {
115             reihe.bubbleSort();
116             ausgabezeile.setText (reihe.zeigeReihe());
117         }
118     }); // runde Klammer (von addActionListener)
119
120 btSort2.addActionListener (
121     new ActionListener ()
122     {
123         public void actionPerformed (ActionEvent e)
124         {
125             reihe.bubbleSort2();
126             ausgabezeile.setText (reihe.zeigeReihe());
127         }
128     }); // runde Klammer (von addActionListener)
129
130 btSort3.addActionListener (
131     new ActionListener ()
132     {
133         public void actionPerformed (ActionEvent e)
134         {
135             reihe.shakerSort();
136             ausgabezeile.setText (reihe.zeigeReihe());
137         }
138     }); // runde Klammer (von addActionListener)
139
140 btSort4.addActionListener (
141     new ActionListener ()
142     {
143         public void actionPerformed (ActionEvent e)
144         {
145             reihe.minSort();
146             ausgabezeile.setText (reihe.zeigeReihe());
147         }
148     }); // runde Klammer (von addActionListener)
149
150 btSort5.addActionListener (
151     new ActionListener ()
152     {
153         public void actionPerformed (ActionEvent e)
154         {
155             reihe.einSort();
156             ausgabezeile.setText (reihe.zeigeReihe());
157         }
158     }); // runde Klammer (von addActionListener)
159
160 btSort6.addActionListener (
161     new ActionListener ()
162     {
163         public void actionPerformed (ActionEvent e)
164         {
165             reihe.bucketSort();
166             ausgabezeile.setText (reihe.zeigeReihe());
167         }
168     }); // runde Klammer (von addActionListener)
169
170 btPrüf.addActionListener (
171     new ActionListener ()
172     {
173         public void actionPerformed (ActionEvent e)
174         {
175             ausgabezeile.setText (reihe.zeigeReihe()+"
176 sortiert="+reihe.istSortiert());
177         }

```

```

178     }); // runde Klammer (von addActionListener)
179
180     btOrig.addActionListener (
181         new ActionListener ()
182         {
183             public void actionPerformed (ActionEvent e)
184             {
185                 reihe.wiederherstellen();
186                 ausgabezeile.setText (reihe.zeigeReihe());
187             }
188         }); // runde Klammer (von addActionListener)
189
190     btEnde.addActionListener (
191         new ActionListener ()
192         {
193             public void actionPerformed (ActionEvent e)
194             {
195                 setVisible (false);
196                 dispose();
197                 System.exit(0);
198             }
199         }); // runde Klammer (von addActionListener)
200 } // starte
201
202 }

```

Datei „IntSortReiheStart.java“

```

1 public class IntSortReiheStart
2 {
3     public static void main (String[] s)
4     {
5         IntSortGUI3 r;
6         r = new IntSortGUI3();
7         r.starte();
8     }

```

