Dietrich Boles

Algorithmen und Datenstrukturen

spielend gelernt

mit dem Java-Hamster-Modell

Version 1.0

18.04.2005

Vorwort

Ursprünglich war das Java-Hamster-Modell dazu gedacht, Programmieranfänger auf spielerische Art und Weise in die (imperative) Programmierung einzuführen. Es sollten nicht so sehr – wie in vielen existierenden Lehrbüchern der Fall – technische Feinheiten der Programmentwicklung, sondern vielmehr das Lösen von Problemen durch Computerprogramme im Vordergrund stehen. Der erste Band des Java-Hamster-Buches mit dem Titel „Programmieren spielend gelernt mit dem Java-Hamster-Modell", der 1999 in der ersten Auflage im Teubner-Verlag erschienen ist, verfolgt genau dieses Ziel.

Heutzutage kommt jedoch sowohl in der Ausbildung als auch in der Industrie der objektorientierten Programmierung eine hohe Bedeutung zu, da sie eine Menge von Vorteilen gegenüber anderen Programmierparadigmen verspricht. Daher habe ich das Java-Hamster-Modell dahingehend erweitert, dass es auch in die objektorientierte Programmierung mit der Programmiersprache Java einführt. Das entsprechende Buch, der zweite Band des Java-Hamster-Buches, mit dem Titel „Objektorientierte Programmierung spielend gelernt mit dem Java-Hamster-Modell" ist im September 2004 in der ersten Auflage ebenfalls im Teubner-Verlag erschienen.

In Band 2 wird im Vorwort angekündigt, dass es noch einen dritten Band geben wird, der in die parallele Programmierung einführt und dabei das Thread-Konzept von Java nutzt. Aktuell arbeite ich an diesem Buch, das den Titel „Parallele Programmierung spielend gelernt mit dem Java-Hamster-Modell" haben wird.

In letzter Zeit habe ich mich jedoch im Rahmen meiner Lehrtätigkeit an der Universität Oldenburg viel mit grundlegenden Algorithmen der Informatik beschäftigt. Dabei ist mir aufgefallen, dass das Hamster-Modell auf hervorragende Art und Weise dazu benutzt werden kann, die Ideen und Konzepte derartiger Algorithmen zu veranschaulichen, man spricht auch von „Algorithmenvisualisierung" oder „Algorithmenanimation". Das Problem ist nämlich, dass sich die Dynamik des Verhaltens von Algorithmen in der traditionellen Darstellung als statischer Text unterstützt durch wenige Bildern nur schwer vermitteln lässt. Genau hier helfen das Java-Hamster-Modell und der Hamster-Simulator. Durch die Aktionen der Hamster lassen sich bestimmte Vorgänge im Territorium demonstrieren und in Kombination mit dem Debugger des Hamster-Simulators kann eine Zuordnung der Aktionen zu Funktionen bzw. Methoden erfolgen.

Klassisches Beispiel sind Sortieralgorithmen. Beim Sortieren geht es darum, Mengen von Daten (bspw. Zahlen) in eine bestimmte Reihenfolge zu bringen. In der Literatur existieren Dutzende von verschiedenen Algorithmen, die diese Aufgabe mehr oder weniger schnell bewältigen. Ein leistungsfähiger Algorithmus nennt sich bspw.

Quicksort. Dieser arbeitet jedoch rekursiv und viele Programmieranfänger tun sich schwer damit, das grundlegende Prinzip dieses Algorithmus zu verstehen. An dieser Stelle greifen nun die Hamster ein. Sie sortieren zwar keine Zahlen, sondern Körnerhaufen, aber Sie zeigen Ihnen dabei im Territorium durch das entsprechende Umsortieren der Haufen, auf welche Art und Weise der Quicksort-Algorithmus prinzipiell vorgeht. Damit sollten es Ihnen dann auch nicht weiter schwer fallen, den grundsätzlichen Quicksort-Algorithmus zu verstehen.

Diese zusätzliche Eigenschaft des Java-Hamster-Modells, ein hervorragende Hilfsmittel zur Algorithmenvisualisierung darzustellen, möchte ich natürlich nicht ungenutzt lassen. Deshalb arbeite ich mit Unterstützung von Informatik-Studierenden parallel zum dritten Band des Java-Hamster-Buches an einem vierten Band, der den Titel „Algorithmen und Datenstrukturen spielend gelernt mit dem Java-Hamster-Modell" tragen wird und den Sie in Version 1.0 jetzt vor sich haben. Dieser vierte Band wird voraussichtlich nie in Buchform erscheinen, sondern frei verfügbar im WWW zugänglich sein. Er wird auch nie „fertig" sein, sondern stetig fortgeschrieben werden. Ich beginne mit den Sortieralgorithmen. Es gibt jedoch viele weitere interessante, wichtige und nützliche Algorithmen. Diese werde ich – soweit es mir meine Zeit zulässt – dann nach und nach hinzufügen.

Die Vorstellung eines Algorithmus wird dabei jeweils in mehrere Teile aufgeteilt: eine textuelle Beschreibung, der allgemein gültige Sourcecode des Algorithmus und ein Hamster-Programm, das den Algorithmus in einer Hamster-spezifischen Art visualisiert. Wenn Sie sich mit einem Algorithmus beschäftigen, sollten Sie dieses Hamster-Programm im Hamster-Simulator starten und sich dadurch den Algorithmus demonstrieren lassen. Sie werden sehen, es hilft Ihnen enorm, das Prinzip des Algorithmus zu verstehen.

Wenn Sie selber Ideen zur Visualisierung bestimmter Algorithmen haben, scheuen Sie sich nicht, diese umzusetzen. Ich würde mich freuen, wenn Sie mir Ihre Arbeiten zukommen lassen würden und ich sie anderen „Hamster-Fans" zur Verfügung stellen könnte. Meine EMail-Adresse lautet: `boles@informatik.uni-oldenburg.de`.

Nun wünsche ich allen Leserinnen und Lesern viel Spaß beim Kennenlernen wichtiger Algorithmen der Informatik. Die Hamster freuen sich schon darauf, sie Ihnen näher zu bringen!

Oldenburg, im April 2005 Dietrich Boles

Inhaltsverzeichnis

Kapitel 1

Einleitung

Es gibt zahlreiche Lehrbücher zu Algorithmen und Datenstrukturen [SS04, Sed03, SG02, Lan02]. Dieses ist ein weiteres, aber es hebt sich von der Mehrzahl der existierenden Büchern dadurch ab, dass es kein reines Print-Buch ist. Vielmehr enthält es zusätzlich Hamster-Programme, die die vorgestellten Algorithmen visualisieren. Daher sollten Sie, wenn Sie dieses Buch lesen, auf Ihrem Computer den Hamster-Simulator gestartet haben und die entsprechenden Hamster-Programme jeweils ausführen.

Dieses erste Kapitel enthält zunächst in Abschnitt 1 eine kurze Einführung in die grundlegende Thematik dieses Buch. Anschließend werden in Abschnitt 2 Voraussetzungen und Ziele des Buches beschrieben. Abschnitt 3 geht auf den Aufbau des Buches ein.

1.1 Algorithmen und Datenstrukturen

Als *Datenstruktur* wird die Organisation von Daten (Variablen) für ihre Verarbeitung durch Programme bezeichnet. Während wir in Band 1 des Java-Hamster-Buches noch ausschließlich mit einzelnen Variablen, also einfachen Datenstrukturen, gearbeitet haben, haben wir in Band 2 Konzepte kennen gelernt, Variablen zu größeren Einheiten zusammenzufassen.

Die grundlegendsten solcher komplexen Datenstrukturen stellen Arrays und Klassen dar.

- Arrays ermöglichen die Zusammenfassung mehrerer Variablen gleichen Typs zu einer Einheit.
- Klassen ermöglichen die Zusammenfassung mehrerer Variablen unter Umständen unterschiedlichen Typs zu einer neuen Einheit.

Auf diesen Konzepten aufbauend wurden in Kapitel 9.3 von Band 2 bereits weiterführende Datenstrukturen vorgestellt:

- *Verkettete Listen* vermeiden das Problem von Arrays, dass bei ihrer Erzeugung bereits die Anzahl der zusammenzufassenden Variablen angegeben werden muss.

- *Stapel* organisieren die in ihnen gespeicherten Daten nach dem Aktenhaufen-Prinzip: Es kann immer nur auf den zuletzt abgespeicherten Wert zugegriffen werden.

Mit diesen und weiteren Datenstrukturen werden wir uns in diesem Buch ausführlich auseinandersetzen.

Derartige Datenstrukturen bilden die Grundlage von Algorithmen. Der Begriff des Algorithmus wurde in Band 1 des Java-Hamster-Buches ausführlich analysiert. Ein Algorithmus ist eine Arbeitsanleitung zum Lösen eines Problems bzw. einer Aufgabe, die so präzise formuliert ist, dass sie von einem Computer ausgeführt werden kann.

Wir werden in diesem Buch viele grundlegende, wichtige und nützliche Algorithmen untersuchen, die jeder Programmierer kennen sollte, wie bspw. das Sortieren oder Suchen. Warum ist das notwendig, werden jetzt vielleicht diejenigen von Ihnen fragen, die bereits die JDK-Klassenbibliothek kennen. Hierin sind doch bereits die wichtigsten Algorithmen implementiert und werden uns Programmierern zur Verfügung gestellt. Das ist in der Tat richtig. Nichtsdestotrotz sprechen einige Gründe dafür, dass wir uns damit nicht zufrieden geben:

- Nicht in jeder Situation ist ein über das JDK zur Verfügung gestellter Algorithmus auch der beste. Wenn es bspw. um das Sortieren von Datenmengen geht, implementiert das JDK mit Quicksort einen Sortieralgorithmus, der im Durchschnitt schneller ist als andere Sortieralgorithmen. Weiss man jedoch, dass die zu sortierende Datenmenge bereits nahezu sortiert ist oder viele gleiche Daten enthält, gibt es schnellere Alternativen.

- Neben der eigentlichen Implementierung werden wir auch eine Analyse der Algorithmen durchführen. Diese gibt Auskünfte darüber, wie gut (in einem bestimmten technischen Sinne) ein Algorithmus in einer bestimmten Situation ist.

- Viele der vorgestellten Algorithmen enthalten grundsätzliche Lösungsansätze, die sich in anderen Zusammenhängen wiederverwenden lassen. Programmieren hat auch viel mit Erfahrung zu tun. Das Studium der grundlegenden Algorithmen der Informatik hilft Ihnen als Programmieranfänger, sich diesen Erfahrungsschatz aufzubauen.

1.2 Voraussetzungen und Ziele dieses Buches

Band 1 [Bol02] und Band 2 [BB04] des Java-Hamster-Buches führen in das Java-Hamster-Modell und die imperative und objektorientierte Programmierung ein. Diese beiden Bücher bilden damit die Voraussetzung zum Verständnis dieses Buches.

Hauptziel dieses Buches ist die Vorstellung elementarer Datenstrukturen und die Einführung in grundlegende, nützliche Algorithmen der Informatik. Alle Algorithmen werden jeweils in einer allgemein gültigen und in einer Hamster-spezifischen Form eingeführt. Und hierin liegt auch das Besondere dieses Buches. Das Hamster-Modell ist hervorragend dazu geeignet, Algorithmen zu „visualisieren“. In der Hamster-spezifischen Algorithmusvariante zeigen Ihnen die Hamster im Territorium, wie die Algorithmen grundsätzlich funktionieren, und machen es Ihnen damit leichter, auch den allgemein gültigen Sourcecode zu verstehen.

Dadurch hilft das Hamster-Modell, ein großes Problem für Programmieranfänger zu lösen. Gute Algorithmen sind häufig etwas komplizierter und nicht immer einfach zu verstehen. In klassischen Lehrbücher findet man oft lediglich eine einführende Beschreibung, ein kleines Beispiel und dann den Sourcecode. Dieses Buch geht einen entscheidenen Schritt weiter. Sie können jeweils die Hamster-spezifische Algorithmusvariante im Hamster-Simulator starten. Die Hamster demonstrieren Ihnen dann Schritt für Schritt durch das Umordnen von Körnerhaufen, das Ablaufen bestimmter Wege oder ähnliche Aktionen, wie der Algorithmus funktioniert. Und durch die Aktivierung des Debuggers können Sie zusätzlich noch mitverfolgen, welcher Teil des Sourcecodes für welche Aktion im Territorium verantwortlich ist.

Weitere Ziele dieses Buches bestehen darin, Ihnen wichtige Tipps für den Entwurf eigener Algorithmen zu geben. Außerdem werden Sie kennen lernen, wie man Algorithmen analysieren und ihre Leistungsfähigkeit bewerten kann.

1.3 Aufbau dieses Buches

Dieses Buch wird wahrscheinlich nie in einem Verlag erscheinen. Es ist ein reines eBook. Es wird auch nie „fertig“ werden. Vielmehr wird es fortlaufend von mir ergänzt werden.

Sie haben aktuell die Version 1.0 des Buches vor sich. In dieser Version werden grundlegende Sortieralgorithmen vorgestellt. Falls Sie informiert werden möchten, wenn eine neue Version erscheint, schauen Sie regelmäßig auf die Website `http://www.java-hamster-modell.de/` oder tragen sich dort in den Java-Hamster-Newsletter ein.

Kapitel 2

Sortieren

Die sicher am intensivsten in der Informatik untersuchten Algorithmen sind Sortieralgorithmen. Sie sind insbesondere deshalb interessant, weil Sortieren zu den häufigsten Teilaufgaben beim Lösen bestimmter Probleme gehört.

Als *Sortieren* bezeichnet man dabei das Ordnen einer Menge von Datensätzen, d.h. diese sollen in eine bestimmte Reihenfolge gebracht werden. Die Datensätze können eine einfache Struktur aufweisen, wie Zahlen. Sie können jedoch auch komplex sein, bspw. Personen mit Name, Alter und Adresse.

Das Kriterium, nach dem geordnet werden soll, bezeichnet man als *Schlüssel*. Personen könnten bspw. nach ihrem Namen oder nach ihrem Alter sortiert werden. Entsprechend ist der Name oder das Alter der Sortierschlüssel.

Eine Voraussetzung beim Sortieren ist, dass eine eine (totale) Ordnung auf den Schlüsseln existiert, d.h. dass für zwei beliebige Schlüssel a und b gilt, das a kleiner als b oder b kleiner als a ist. Bestehen die Schlüssel aus Zahlen oder Zeichenketten wird im Allgemeinen nach der bekannten numerischen bzw. lexikographischen Ordnung sortiert.

In diesem Abschnitt werden die sechs bekanntesten Sortieralgorithmen vorgestellt:

- SelectionSort: Sortieren durch Auswählen
- BubbleSort: Sortieren durch Vertauschen
- InsertionSort: Sortieren durch Einfügen
- ShellSort: Verbessertes Sortieren durch Einfügen
- QuickSort: Sortieren durch rekursives Zerlegen
- MergeSort: Sortieren durch Mischen

Zu jedem Algorithmus werden zunächst die generelle Idee skizziert und der allgemein gültige Sourcecode eingeführt und erläutert. Sortiert werden dabei Arrays mit `int`-Werten, und zwar gemäß der numerischen Ordnung in aufsteigender Reihenfolge. Für jeden Algorithmus wird das folgende Interface implementiert:

```
public interface SortierAlgorithmus {
  // sortiert das uebergebene Array in aufsteigender Reihenfolge
  public void sortiere(int[] zahlen);
}
```

Anschließend wird jeweils ein Hamster-Programm vorgestellt, das den entsprechenden Sortieralgorithmus visualisiert, d.h. die Hamster zeigen Ihnen, wie der Algorithmus prinzipiell funktioniert. Sortiert werden dabei Körnerhaufen, und zwar von Hamstern spezieller Klassen, die das folgende Interface implementieren:

```
public interface SortierHamster {
  // Der Standard-Hamster steht mit Blickrichtung OST irgendwo im
  // Territorium.  Ein Vertretungshamster soll die Koernerhaufen
  // bis zur nächsten Wand in aufsteigender Reihenfolge sortieren.
  public void sortiereKoernerHaufen();
}
```

Die Hamster-Programme sind nicht primär zum Lesen, sondern zum Ausführen gedacht. Machen Sie daher folgendes:

- Laden Sie sich die Programme von der Hamster-Website herunter (Datei `Sortieren.zip`).
- Entpacken Sie die Datei in den Unterordner `Programme` des Ordners, in dem sich der Hamster-Simulator befindet.
- Starten Sie den Hamster-Simulator.
- Laden Sie eines von mehreren vorbereiteten Territorien aus dem Ordner `Sortieren`. Diese Territorien enthalten zu sortierende Körnerhaufen.
- Führen Sie die Sortierprogramme aus und lassen Sie sich von den Hamster demonstrieren und erläutern, wie die entsprechenden Sortieralgorithmen funktionieren.

Nach der Vorstellung jedes Sortieralgorithmus folgt noch eine kurze Analyse der Eigenschaften des Algorithmus. Insbesondere wird analysiert, wie schnell der Algorithmus ist.[1]

Weiterhin wird die Eigenschaft der Stabilität betrachtet. Ein Sortieralgorithmus heisst *stabil*, wenn die relative Reihenfolge gleicher Schlüssel beibehalten wird. Diese

[1] In dieser Version von Band 4 des Java-Hamster-Buches sind die Analysen noch sehr kurz. In einer Folgeversion wird ein Kapitel eingeschoben, dass sich generell mit der Analyse von Algorithmen beschäftigt. Danach werden die Analysen der Sortieralgorithmen nochmal überarbeitet und ausführlicher vorgestellt.

Eigenschaft ist von Interesse, wenn die Datensätze nicht nur aus Schlüsseln bestehen. Nehmen wir bspw. wieder zu sortierende Personen mit Name und Alter. Angenommen, die Personen seien bereits alphabetisch nach ihren Namen sortiert. Werden sie nun nach ihrem Alter sortiert, so sind nachher bei stabilen Verfahren Personen mit dem gleichen Alter auch weiterhin alphabetisch geordnet.

Weiterhin interessant ist die Frage, ob ein bestimmter Sortieralgorithmus *in-place* arbeitet. Man sagt, ein Algorithmus arbeitet *in-place*, wenn er außer dem für die Speicherung der zu sortierenden Daten benötigten Speicher nur eine konstante – d.h. von der Größe der zu sortierenden Datenmenge unabhängige – Menge von Speicher benötigt.

Zum Schluss des Kapitels folgt eine kurze Zusammenfassung sowie ein paar Anmerkungen zu weiteren Aspekten der Sortierung. Insbesondere wird gezeigt, wie die Programme verändert werden müssen, wenn anstelle von Arrays mit `int`-Werten, Arrays mit Zeichenketten bzw. beliebigen Objekten sortiert werden sollen.

2.1 Hilfsklassen

In den folgenden Abschnitten werden die einzelnen Sortieralgorithmen vorgestellt und es werden objektorientierte Hamster-Programme entwickelt, die die Funktionsweise der Algorithmen demonstrieren. Die Programme bzw. Klassen greifen dabei auf einige Hilfsklassen zurück, die in diesem Abschnitt vorgestellt werden.

2.1.1 AllroundHamster

Die erweiterte Hamster-Klasse `AllroundHamster` erweitert den Befehlsvorrat der Hamster um einige nützliche Befehle.

```
// die Klasse erweitert den Befehlssatz eines normalen
// Hamsters um viele nuetzliche Befehle
public class AllroundHamster extends Hamster {

  protected boolean mitErlaeuterungen;
    // gibt an, ob textuelle Erlaeuterungen ausgegeben werden sollen

  // initialisiert einen neuen AllroundHamster mit den
  // uebergebenen Werten
  public AllroundHamster(int r, int s, int b, int k,
                         boolean mitErlaeuterungen) {
    super(r, s, b, k);
    this.mitErlaeuterungen = mitErlaeuterungen;
  }

  // initialisiert einen neuen AllroundHamster mit den
  // uebergebenen Werten
  public AllroundHamster(int r, int s, int b, int k) {
```

```
  this(r, s, b, k, false);
}

// initialisiert einen neuen AllroundHamster mit den
// Attributwerten eines bereits existierenden Hamsters
public AllroundHamster(Hamster existierenderHamster,
                       boolean mitErlaeuterungen) {
  this(existierenderHamster.getReihe(),
       existierenderHamster.getSpalte(),
       existierenderHamster.getBlickrichtung(),
       existierenderHamster.getAnzahlKoerner(),
       mitErlaeuterungen
      );
}

// initialisiert einen neuen AllroundHamster mit den
// Attributwerten eines bereits existierenden Hamsters
public AllroundHamster(Hamster existierenderHamster) {
  this(existierenderHamster, false);
}

// gibt eine Erlaeuterung auf den Bildschirm aus
public void erlaeuterung(String text) {
  if (this.mitErlaeuterungen) {
    this.schreib(text);
  }
}

// der Hamster dreht sich um 180 Grad
public void kehrt() {
  this.linksUm();
  this.linksUm();
}

// der Hamster dreht sich nach rechts
public void rechtsUm() {
  this.kehrt();
  this.linksUm();
}

// der Hamster laeuft "anzahl" Schritte, maximal jedoch
// bis zur naechsten Mauer;
// geliefert wird die tatsaechliche Anzahl gelaufener
// Schritte
public int vor(int anzahl) {
  int schritte = 0;
  while (this.vornFrei() && anzahl > 0) {
    this.vor();
    schritte = schritte + 1;
    anzahl = anzahl - 1;
  }
  return schritte;
}
```

```
// der Hamster legt "anzahl" Koerner ab, maximal jedoch
// so viele, wie er im Maul hat;
// geliefert wird die tatsaechliche Anzahl abgelegter
// Koerner
public int gib(int anzahl) {
  int abgelegteKoerner = 0;
  while (!this.maulLeer() && anzahl > 0) {
    this.gib();
    abgelegteKoerner = abgelegteKoerner + 1;
    anzahl = anzahl - 1;
  }
  return abgelegteKoerner;
}

// der Hamster frisst "anzahl" Koerner, maximal jedoch
// so viele, wie auf der aktuellen Kachel liegen
public int nimm(int anzahl) {
  int gefresseneKoerner = 0;
  while (this.kornDa() && anzahl > 0) {
    this.nimm();
    gefresseneKoerner = gefresseneKoerner + 1;
    anzahl = anzahl - 1;
  }
  return gefresseneKoerner;
}

// der Hamster legt alle Koerner, die er im Maul hat,
// auf der aktuellen Kachel ab;
// geliefert wird die Anzahl abgelegter Koerner
public int gibAlle() {
  int abgelegteKoerner = 0;
  while (!this.maulLeer()) {
    this.gib();
    abgelegteKoerner = abgelegteKoerner + 1;
  }
  return abgelegteKoerner;
}

// der Hamster frisst alle Koerner auf der aktuellen Kachel;
// geliefert wird die Anzahl gefressener Koerner
public int nimmAlle() {
  int gefresseneKoerner = 0;
  while (this.kornDa()) {
    this.nimm();
    gefresseneKoerner = gefresseneKoerner + 1;
  }
  return gefresseneKoerner;
}

// der Hamster laeuft bis zur naechsten Mauer;
// geliefert wird die Anzahl ausgefuehrter Schritte
public int laufeZurWand() {
```

```
    int schritte = 0;
    while (this.vornFrei()) {
      this.vor();
      schritte = schritte + 1;
    }
    return schritte;
  }

  // der Hamster testet, ob links von ihm die Kachel frei ist
  public boolean linksFrei() {
    this.linksUm();
    boolean frei = this.vornFrei();
    this.rechtsUm();
    return frei;
  }

  // der Hamster testet, ob rechts von ihm die Kachel frei ist
  public boolean rechtsFrei() {
    this.rechtsUm();
    boolean frei = this.vornFrei();
    this.linksUm();
    return frei;
  }

  // der Hamster testet, ob hinter ihm die Kachel frei ist
  public boolean hintenFrei() {
    this.kehrt();
    boolean frei = this.vornFrei();
    this.kehrt();
    return frei;
  }

  // der Hamster dreht sich so lange um, bis er in die
  // uebergebene Blickrichtung schaut
  public void setzeBlickrichtung(int richtung) {
    while (this.getBlickrichtung() != richtung) {
      this.linksUm();
    }
  }

  // der Hamster laeuft in der Spalte, in der er
  // gerade steht, zur angegebenen Reihe;
  // Voraussetzung: die Reihe existiert und
  // es befinden sich keine Mauern
  // im Territorium bzw. auf dem gewaehlten Weg
  public void laufeZuReihe(int reihe) {
    if (reihe == this.getReihe()) return;
    if (reihe > this.getReihe()) {
      this.setzeBlickrichtung(Hamster.SUED);
    } else {
      this.setzeBlickrichtung(Hamster.NORD);
    }
    while (reihe != this.getReihe()) {
```

```
      this.vor();
    }
  }

  // der Hamster laeuft in der Reihe, in der er
  // gerade steht, zur angegebenen Spalte;
  // Voraussetzung: die Spalte existiert und
  // es befinden sich keine Mauern
  // im Territorium bzw. auf dem gewaehlten Weg
  public void laufeZuSpalte(int spalte) {
    if (spalte == this.getSpalte()) return;
    if (spalte > this.getSpalte()) {
      this.setzeBlickrichtung(Hamster.OST);
    } else {
      this.setzeBlickrichtung(Hamster.WEST);
    }
    while (spalte != this.getSpalte()) {
      this.vor();
    }
  }

  // der Hamster laeuft zur Kachel (reihe/spalte);
  // Voraussetzung: die Kachel existiert und
  // es befinden sich keine Mauern
  // im Territorium bzw. auf dem gewaehlten Weg
  public void laufeZuKachel(int reihe, int spalte) {
    laufeZuReihe(reihe);
    laufeZuSpalte(spalte);
  }
}
```

2.1.2 KoernerHaufenSortierHamster

Hamster, die die eigentliche Sortierung der Körnerhaufen durchführen, werden von von der Klasse `KoernerHaufenSortierHamster` abgeleiteten Klassen erzeugt, die wiederum von der Klasse `AllroundHamster` abgeleitet ist und nützliche Methoden zum Sortieren bereitstellt.

```
// Basisklasse der eigentlichen Hamster-Sortier-Klassen
public class KoernerHaufenSortierHamster extends AllroundHamster {

  protected int startSpalte = 0;
    // Spalte, in der der Sortier-Hamster anfangs steht

  // Konstruktor
  protected KoernerHaufenSortierHamster(boolean erlaeuterungen) {
    super(Hamster.getStandardHamster(), erlaeuterungen);
    this.startSpalte = this.getSpalte();
  }

  // ermittelt und liefert die Anzahl der zu sortierenden
```

```
  // Koernerhaufen
  protected int ermittleAnzahlKoernerHaufen() {
    // bis zur naechsten Mauer
    int anzahlKoernerHaufen = 0;
    while (this.vornFrei()) {
      this.vor();
      anzahlKoernerHaufen++;
    }
    // und zurueck (Seiteneffekte beseitigen)
    this.kehrt();
    int speicher = anzahlKoernerHaufen;
    while (speicher > 0) {
      this.vor();
      speicher = speicher - 1;
    }
    this.kehrt();
    return anzahlKoernerHaufen;
  }

  // laeuft zum angegebenen Koernerhaufenindex
  protected void laufeZuIndex(int index) {
    this.laufeZuSpalte(this.startSpalte + index + 1);
  }

  // liefert den aktuellen Index des Koernerhaufens, auf dem der Hamster steht
  protected int liefereIndex() {
    return this.getSpalte() - this.startSpalte - 1;
  }

  // liefert die Anzahl an Koernern des Koernerhaufens, auf dem der Hamster steht
  protected int liefereAnzahlKoerner() {
    return Territorium.getAnzahlKoerner(this.getReihe(), this.getSpalte());
  }

}
```

2.1.3 MarkierungsHamster

Um die Funktionsweise der jeweiligen Sortieralgorithmen zu demonstrieren, werden Hamster eingesetzt, die bestimmte Körnerhaufen markieren. Diese Hamster werden von der Klasse `MarkierungsHamster` erzeugt.

```
// zum Markieren bestimmter Koernerhaufen
public class MarkierungsHamster extends AllroundHamster {

  private int startSpalte;
    // Spalte, in der der Hamster erzeugt wird

  private int koernerHaufenReihe;
    // Reihe, in der die Koernerhaufen liegen
```

```
  // Konstruktor
  public MarkierungsHamster(int reihe, int spalte, int koernerHaufenReihe,
                            boolean mitErlaeuterungen) {
    super(reihe, spalte, Hamster.NORD, 0, mitErlaeuterungen);
    this.startSpalte = spalte;
    this.koernerHaufenReihe = koernerHaufenReihe;
  }

  // laeuft zum angegebenen Koernerhaufenindex
  public void markiereIndex(int index) {
    this.laufeZuSpalte(this.startSpalte + index + 1);
    this.setzeBlickrichtung(Hamster.NORD);
  }

  // liefert den markierten Koernerhaufenindex
  public int liefereIndex() {
    return this.getSpalte() - this.startSpalte - 1;
  }

  // liefert die Anzahl an Koernern des markierten Koernerhaufens
  public int liefereAnzahlKoerner() {
    return
      Territorium.getAnzahlKoerner(this.koernerHaufenReihe, this.getSpalte());
  }

  // kehrt zur Startspalte zurueck
  public void laufeZuStartSpalte() {
    this.laufeZuSpalte(this.startSpalte);
    this.setzeBlickrichtung(Hamster.NORD);
  }
}
```

2.1.4 BooleanHamster

Hamster der Klasse `BooleanHamster` repräsentieren die beiden Wahrheitswerte. Schaut ein Boolean-Hamster nach Nordern kennzeichnet er den Wert `true`, schaut er nach Osten, kennzeichnet er den Wert `false`.

```
// demonstriert die Wahrheitswerte wahr und falsch
public class BooleanHamster extends AllroundHamster {

  // Konstruktor
  public BooleanHamster(int reihe, int spalte, boolean wahr,
                        boolean mitErlaeuterungen) {
    super(reihe, spalte, Hamster.NORD, 0, mitErlaeuterungen);
    this.deuteAn(wahr);
  }

  // deutet den angegebenen Wert an
  public void deuteAn(boolean wahr) {
    if (wahr) {
```

```
      this.setzeBlickrichtung(Hamster.NORD);
    } else {
      this.setzeBlickrichtung(Hamster.OST);
    }
  }

  // liefert den angedeuteten Wert
  public boolean istWahr() {
    return this.getBlickrichtung() == Hamster.NORD;
  }
}
```

2.2 SelectionSort: Sortieren durch Auswählen

Einer der einfachsten Sortieralgorithmen wird „Sortieren durch Auswählen“ – im Englischen *SelectionSort* – genannt.

2.2.1 Algorithmus

Der SelectionSort-Algorithmus lässt sich folgendermaßen skizzieren:

Gegeben sei ein Array mit n Elementen, das in aufsteigender Reihenfolge sortiert werden soll:

- Suche das kleinste Element im Array. Tausche es gegen das erste Element aus.
- Suche das zweitkleinste Element im Array. Tausche es gegen das zweite Element aus.
- Suche das drittkleinste Element im Array. Tausche es gegen das dritte Element aus.
- ...
- Suche das (n-1)kleinste Element im Array. Tausche es gegen das (n-1)ste Element aus.

Das Array besteht also aus einem sortierten vorderen Teil, der in jedem Schritt um ein Element wächst, und einem unsortierten hinteren Teil, der entsprechend schrumpft. D.h. bei der Suche nach dem x-kleinsten Element muss nur noch der unsortierte hintere Teil des Arrays ab dem x-ten Element durchsucht werden, da die ersten x-1 Elemente bereits sortiert sind. Genauer gesagt, muss in jedem Schritt jeweils das kleinste Element des unsortierten Teils gesucht und mit dem ersten Element des unsortierten Teils vertauscht werden.

Die folgende Klasse `SelectionSort` implementiert das Interface `SortierAlgorithmus` entsprechend des SelectionSort-Algorithmus:

```
public class SelectionSort implements SortierAlgorithmus {

  // sortiert das übergebene Array in aufsteigender Reihenfolge
  // gemaess dem SelectionSort-Algorithmus
  public void sortiere(int[] zahlen) {
    for (int aktIndex = 0; aktIndex < zahlen.length - 1; aktIndex++) {
      int minIndex = sucheKleinstesElement(zahlen, aktIndex);
      tauscheElemente(zahlen, aktIndex, minIndex);
    }
  }

  // sucht im Array ab dem angegebenen Index das kleinste Element
  // und liefert dessen Index
  private int sucheKleinstesElement(int[] zahlen, int abIndex) {
    int minIndex = abIndex;
    for (int suchIndex = abIndex+1; suchIndex < zahlen.length; suchIndex++) {
      if (zahlen[suchIndex] < zahlen[minIndex]) {
        // neues kleinstes Element gefunden
        minIndex = suchIndex;
      }
    }
    return minIndex;
  }

  // vertauscht im Array die Elemente der angegebenen Indizes
  private void tauscheElemente(int[] zahlen, int index1, int index2) {
    int speicher = zahlen[index1];
    zahlen[index1] = zahlen[index2];
    zahlen[index2] = speicher;
  }

  /* der SelectionSort-Algorithmus in kompakter Form
  public void selection(int[] zahlen) {
    for (int aktIndex = 0; aktIndex < zahlen.length - 1; aktIndex++) {
      int minIndex = aktIndex;
      for (int suchIndex = aktIndex+1; suchIndex < zahlen.length; suchIndex++) {
        if (zahlen[suchIndex] < zahlen[minIndex]) {
          minIndex = suchIndex;
        }
      }
      int speicher = zahlen[aktIndex];
      zahlen[aktIndex] = zahlen[minIndex];
      zahlen[minIndex] = speicher;
    }
  }
  */
}
```

2.2.2 Visualisierendes Hamster-Programm

Die folgende Hamster-Klasse `SelectionSortHamster` implementiert das Interface `SortierHamster` und visualisiert dabei den SelectionSort-Algorithmus. Voraussetzung ist, dass es im Territorium zwei nicht durch Mauern blockierte Reihen unterhalb des Standard-Hamsters gibt. Während ein Körnerhaufensortierhamster als Vertretungshamster des Standard-Hamsters (roter Hamster) jeweils den unsortierten Teil der Körnerhaufen nach dem kleinsten Element durchsucht, markiert in der ersten Reihe unterhalb der Körnerhaufenreihe ein Markierungshamster (grüner Hamster) den aktuellen Index, d.h. er zeigt auf das erste Element des unsortierten Teils. Ein weiterer Markierungshamster (gelber Hamster), eine weitere Reihe darunter, markiert das aktuell kleinste Element während der Suche nach dem kleinsten Element.

```
// Demonstration des SelectionSort-Algorithmus
public class SelectionSortHamster
  extends KoernerHaufenSortierHamster
  implements SortierHamster
{

  private MarkierungsHamster aktIndexHamster = null;
    // markiert den Start des unsortierten Teils

  private MarkierungsHamster minIndexHamster = null;
    // markiert den jeweils kleinsten Koernerhaufen im unsortierten Teil

  private int anzahlKoernerHaufen = 0;
    // Anzahl der zu sortierenden Koernerhaufen

  // Konstruktor
  public SelectionSortHamster(boolean mitErlaeuterungen) {
    super(mitErlaeuterungen);
    this.erlaeuterung(
      "Ich sortiere die Koernerhaufen auf der Basis des " +
      "SelectionSort-Algorithmus.");
    this.aktIndexHamster =
      new MarkierungsHamster(this.getReihe()+1, this.getSpalte(),
                             this.getReihe(), mitErlaeuterungen);
    this.aktIndexHamster.erlaeuterung(
      "Ich markiere den linken Koernerhaufen des unsortierten Teils.");
    this.minIndexHamster =
      new MarkierungsHamster(this.getReihe()+2, this.getSpalte(),
                             this.getReihe(), mitErlaeuterungen);
    this.minIndexHamster.erlaeuterung(
      "Ich markiere den aktuell kleinsten Koernerhaufen im unsortierten Teil.");
  }

  // Der Standard-Hamster steht mit Blickrichtung OST irgendwo im Territorium.
  // Ein Vertretungshamster soll die Koernerhaufen bis zur nächsten Wand in
  // aufsteigender Reihenfolge sortieren.
  // Voraussetzung: Unterhalb des Standard-Hamsters existieren zwei nicht
  // durch Mauern blockierte Reihen.
  public void sortiereKoernerHaufen() {
    this.erlaeuterung(
      "Ich zaehle nun die Anzahl der zu sortierenden Koernerhaufen.");
    this.anzahlKoernerHaufen = this.ermittleAnzahlKoernerHaufen();
```

```
    this.erlaeuterung(
      "Die Anzahl der zu sortierenden Koernerhaufen betraegt " +
      this.anzahlKoernerHaufen +
      ",\nd.h. die Indizes der Haufen liegen zwischen 0 und " +
      (this.anzahlKoernerHaufen-1) +
      ".");

    // durchlaufe die Menge der Koernerhaufen von links nach rechts;
    // merke dir den aktuellen Koernerhaufen
    for (int aktIndex=0; aktIndex<this.anzahlKoernerHaufen-1; aktIndex++) {
      // suche den kleinsten Koernerhaufen im unsortierten Teil
      this.sucheMinKoernerHaufen(aktIndex);
      // tausche den aktuellen Koernerhaufen mit dem kleinsten
      // Koernerhaufen im unsortierten Teil
      this.aktUndMinKoernerHaufenTauschen();
      this.erlaeuterung(
        "Die Koernerhaufen zwischen den Indizes 0 und " +
        aktIndex +
        " sind nun sortiert.");
    }
    beendeSortierung();
  }

  private void sucheMinKoernerHaufen(int aktIndex) {
    // zunaechst ist der aktuelle Koernerhaufen auch der kleinste
    // im unsortierten Teil
    this.aktIndexHamster.markiereIndex(aktIndex);
    this.minIndexHamster.markiereIndex(aktIndex);
    this.erlaeuterung(
      "Ich suche nun den kleinsten Koernerhaufen im unsortierten Teil,\n" +
      "d.h. zwischen den Indizes " +
      aktIndex +
      " und " +
      (this.anzahlKoernerHaufen-1) +
      ".");
    // suche im unsortierten Teil der Koernerhaufen den kleinsten
    for (int suchIndex=aktIndex+1;
         suchIndex<this.anzahlKoernerHaufen;
         suchIndex++) {
      this.laufeZuIndex(suchIndex);
      if (this.liefereAnzahlKoerner() <
          this.minIndexHamster.liefereAnzahlKoerner()) {
        // neuer kleinster Koernerhaufen gefunden
        this.minIndexHamster.markiereIndex(suchIndex);
      }
    }
    this.erlaeuterung(
      "Der kleinste Koernerhaufen zwischen den Indizes " +
      aktIndex +
      " und " +
      (this.anzahlKoernerHaufen-1) +
      " befindet sich bei Index " +
      this.minIndexHamster.liefereIndex() +
```

```
      ".");
  }

  private void aktUndMinKoernerHaufenTauschen() {
    if (this.aktIndexHamster.liefereIndex() !=
        this.minIndexHamster.liefereIndex()) {
      this.erlaeuterung(
        "Ich tausche nun die markierten Koernerhaufen der Indizes " +
        this.aktIndexHamster.liefereIndex() +
        " und " +
        this.minIndexHamster.liefereIndex() +
        ".");
      this.laufeZuIndex(this.minIndexHamster.liefereIndex());
      int minKoerner = this.nimmAlle();
      this.laufeZuIndex(this.aktIndexHamster.liefereIndex());
      int aktKoerner = this.nimmAlle();
      this.gib(minKoerner);
      this.laufeZuIndex(this.minIndexHamster.liefereIndex());
      this.gib(aktKoerner);
    } else {
      this.erlaeuterung(
        "Der Koernerhaufen links im unsortierten Teil ist auch der kleinste.\n" +
        "Daher brauche ich nicht zu tauschen.");
    }
  }

  private void beendeSortierung() {
    this.laufeZuSpalte(this.startSpalte);
    this.setzeBlickrichtung(Hamster.OST);
    this.erlaeuterung("Sortierung erfolgreich beendet!");
  }
}
```

Mit dem folgendem objektorientierten Hamster-Programm (`SortierenMitSelectionSort`) können Sie sich von den Hamstern den SelectionSort-Algorithmus demonstrieren lassen.

```
void main() {
  Hamster paul = Hamster.getStandardHamster();
  String antwort =
    Hamster.getStandardHamster().liesZeichenkette(
      "Moechten Sie textuelle Erlaeuterungen (ja/nein)?");
  SortierHamster sortierer = new SelectionSortHamster(antwort.equals("ja"));
  sortierer.sortiereKoernerHaufen();
}
```

2.2.3 Analyse des Algorithmus

Die vorgestellte Implementierung des SelectionSort-Algorithmus ist stabil[2]. SelectionSort arbeitet in-place.

[2] In der Literatur finden sich auch Implementierungen, die nicht stabil sind.

Sei n die Anzahl an zu sortierenden Elementen. Die Anzahl an Vergleichen zwischen den Array-Elementen beträgt $\approx n^2/2$. Das gilt in allen Fällen,

- wenn das Array bereits aufsteigend sortiert ist,
- wenn das Array absteigend sortiert ist,
- wenn die Anordnung der Elemente im Array zufällig ist.

Die Anzahl an Vertauschungen von Array-Elementen beträgt

- 0, wenn das Array bereits aufsteigend sortiert ist,
- $n - 1$, wenn das Array absteigend sortiert ist,
- im Mittel $\approx n/2$, wenn die Anordnung der Elemente im Array zufällig ist.

Insbesondere bedeutet dass, das jedes Array-Element maximal einmal bewegt wird.

Im Web finden sich zahlreiche weitere Informationen zum SelectionSort-Algorithmus, z.B. unter

- `http://de.wikipedia.org/wiki/Selectionsort`
- `http://www.sortieralgorithmen.de/selectsort/index.html`

2.3 BubbleSort: Sortieren durch Vertauschen

Eine Variante des SelectionSort-Algorithmus, die man sehr häufig in Lehrbüchern findet, ist der BubbleSort-Algorithmus.

2.3.1 Algorithmus

Der BubbleSort-Algorithmus lässt sich folgendermaßen skizzieren:

Gegeben sei ein Array mit n Elementen, das in aufsteigender Reihenfolge sortiert werden soll:

- Das Array wird von hinten nach vorne durchlaufen. Dabei werden jeweils benachbarte Elemente miteinander verglichen. Ist das rechte Element kleiner als das linke, werden die beiden vertauscht.
- Diese Durchläufe werden so lange wiederholt, bis in einem Durchlauf keine Vertauschung stattgefunden hat. Dann ist das Array sortiert.

Durch die fortlaufenden Vertauschungen wird im ersten Durchlauf das kleinste Element des Array ganz nach links transportiert, im zweiten Durchlauf wird das zweitkleinste an seine korrekte Position gebracht, usw. Das heisst, das Prinzip von BubbleSort ist identisch mit dem Prinzip von SelectionSort, nur die Suche nach dem jeweils kleinsten Element erfolgt auf eine andere Art und Weise.

Die folgende Klasse `BubbleSort` implementiert das Interface `SortierAlgorithmus` entsprechend des BubbleSort-Algorithmus:

```
public class BubbleSort implements SortierAlgorithmus {

  // sortiert das uebergebene Array in aufsteigender Reihenfolge
  // gemaess dem BubbleSort-Algorithmus
  public void sortiere(int[] zahlen) {
    boolean tauschStattgefunden = true;
    int anzahlSortierteZahlen = 0;
    do {
      tauschStattgefunden = false;
      for (int aktIndex = zahlen.length-1;
           aktIndex > anzahlSortierteZahlen;
           aktIndex--) {
        if (zahlen[aktIndex] < zahlen[aktIndex-1]) {
          tauscheElemente(zahlen, aktIndex, aktIndex-1);
          tauschStattgefunden = true;
        }
      }
      anzahlSortierteZahlen++;
    } while (tauschStattgefunden);
  }

  // vertauscht im Array die Elemente der angegebenen Indizes
  private void tauscheElemente(int[] zahlen, int index1, int index2) {
    int speicher = zahlen[index1];
    zahlen[index1] = zahlen[index2];
    zahlen[index2] = speicher;
  }

  /* der BubbleSort-Algorithmus in kompakter Form
  public void sortiere(int[] zahlen) {
    boolean tauschStattgefunden = true;
    int anzahlSortierteZahlen = 0;
    do {
      tauschStattgefunden = false;
      for (int aktIndex = zahlen.length-1;
           aktIndex > anzahlSortierteZahlen;
           aktIndex--) {
        if (zahlen[aktIndex] < zahlen[aktIndex-1]) {
          int speicher = zahlen[aktIndex];
          zahlen[aktIndex] = zahlen[aktIndex-1];
          zahlen[aktIndex-1] = speicher;
          tauschStattgefunden = true;
        }
```

```
      }
      anzahlSortierteZahlen++;
    } while (tauschStattgefunden);
  }
  */
}
```

2.3.2 Visualisierendes Hamster-Programm

Die folgende Hamster-Klasse `BubbleSortHamster` implementiert das Interface `SortierHamster` und visualisiert dabei den BubbleSort-Algorithmus. Voraussetzung ist, dass es im Territorium zwei nicht durch Mauern blockierte Reihen unterhalb des Standard-Hamsters gibt. Ein Körnerhaufensortierhamster als Vertretungshamster des Standard-Hamsters (roter Hamster) durchläuft – so lange Vertauschungen stattgefunden haben – von rechts nach links den unsortierten Teil der Körnerhaufen und führt dabei notwendige Vertauschungen durch. In der ersten Reihe unterhalb der Körnerhaufenreihe markiert ein Markierungshamster (grüner Hamster) den rechten Index des sortierten Teils. Ein Boolean-Hamster (gelber Hamster) in der zweiten Reihe deutet an, ob im aktuellen Durchlauf Vertauschungen stattgefunden haben (Blickrichtung ist Nord) oder nicht (Blickrichtung ist Ost).

```
// Demonstartion des BubbleSort-Algorithmus
public class BubbleSortHamster
  extends KoernerHaufenSortierHamster
  implements SortierHamster {

  private MarkierungsHamster sortiertIndexHamster = null;
    // markiert das Ende des sortierten Teils

  private BooleanHamster tauschStattgefundenHamster = null;
    // zeigt an, ob in einem Durchlauf ein Tausch stattgefunden hat

  private int anzahlKoernerHaufen = 0;
    // Anzahl der zu sortierenden Koernerhaufen

  // Konstruktor
  public BubbleSortHamster(boolean mitErlaeuterungen) {
    super(mitErlaeuterungen);
    this.erlaeuterung(
      "Ich sortiere die Koernerhaufen auf der Basis des BubbleSort-Algorithmus.");
    this.sortiertIndexHamster =
      new MarkierungsHamster(this.getReihe() + 1, this.getSpalte(),
                             this.getReihe(), mitErlaeuterungen);
    this.sortiertIndexHamster.erlaeuterung(
      "Ich markiere das Ende des sortierten Teils.");
    this.tauschStattgefundenHamster =
      new BooleanHamster(this.getReihe() + 2, this.getSpalte(),
                         false, mitErlaeuterungen);
    this.tauschStattgefundenHamster.erlaeuterung(
```

```
    "Ich deute an, ob in einem Durchlauf ein Tausch stattgefunden hat.");
}

// Der Standard-Hamster steht mit Blickrichtung OST irgendwo im Territorium.
// Ein Vertretungshamster soll die Koernerhaufen bis zur nächsten Wand in
// aufsteigender Reihenfolge sortieren.
// Voraussetzung: Unterhalb des Standard-Hamsters existieren zwei nicht
// durch Mauern blockierte Reihen.
public void sortiereKoernerHaufen() {
  this.erlaeuterung(
    "Ich zaehle nun die Anzahl der zu sortierenden Koernerhaufen.");
  this.anzahlKoernerHaufen = this.ermittleAnzahlKoernerHaufen();
  this.erlaeuterung(
    "Die Anzahl der zu sortierenden Koernerhaufen betraegt " +
    this.anzahlKoernerHaufen +
    ",\nd.h. die Indizes der Haufen liegen zwischen 0 und " +
    (this.anzahlKoernerHaufen-1) +
    ".");

  do {
    this.tauschStattgefundenHamster.deuteAn(false);
    // laufe nach vorne bis zum Anfang des unsortierten Teils
    this.erlaeuterung(
      "Ich transportiere nun den kleinsten Koernerhaufen im unsortierten Teil," +
      "\nd.h. zwischen den Indizes " +
      (this.sortiertIndexHamster.liefereIndex()+1) + " und " +
      (anzahlKoernerHaufen-1) + ", zum Index " +
      (this.sortiertIndexHamster.liefereIndex()+1) +
      ".");
    for (int aktIndex = anzahlKoernerHaufen-1;
         aktIndex > this.sortiertIndexHamster.liefereIndex()+1;
         aktIndex--) {
      this.laufeZuIndex(aktIndex);
      this.setzeBlickrichtung(Hamster.WEST);
      // vergleiche benachbarte Koernerhaufen und tausche,
      // falls der hintere Haufen kleiner ist
      if (anzahlKoernerAktIndex() < anzahlKoernerAktIndexMinus1()) {
        this.erlaeuterung(
          "Der Koernerhaufen beim Index " +
          aktIndex +
          " ist kleiner als der Koernerhaufen beim Index " +
          (aktIndex-1) +
          ".\nDaher tausche ich sie.");
        koernerHaufenTauschen();
        this.tauschStattgefundenHamster.deuteAn(true);
      }
    }
    this.sortiertIndexHamster.markiereIndex(
      this.sortiertIndexHamster.liefereIndex()+1);
  } while (this.tauschStattgefundenHamster.istWahr());
  this.erlaeuterung(
    "Im letzten Durchlauf hat kein Tausch mehr stattgefunden," +
    "\nd.h. die Koernerhaufen sind sortiert.");
```

```
    beendeSortierung();
  }

  private void koernerHaufenTauschen() {
    int koernerAnzahl1 = this.nimmAlle();
    this.vor();
    int koernerAnzahl2 = this.nimmAlle();
    this.gib(koernerAnzahl1);
    this.kehrt();
    this.vor();
    this.gib(koernerAnzahl2);
  }

  private void beendeSortierung() {
    this.laufeZuSpalte(this.startSpalte);
    this.setzeBlickrichtung(Hamster.OST);
    this.erlaeuterung("Sortierung erfolgreich beendet!");
  }

  private int anzahlKoernerAktIndex() {
    return Territorium.getAnzahlKoerner(this.getReihe(), this.getSpalte());
  }

  private int anzahlKoernerAktIndexMinus1() {
    return Territorium.getAnzahlKoerner(this.getReihe(), this.getSpalte()-1);
  }
}
```

Mit dem folgendem objektorientierten Hamster-Programm (`SortierenMitBubbleSort`) können Sie sich von den Hamstern den BubbleSort-Algorithmus demonstrieren lassen.

```
void main() {
  Hamster paul = Hamster.getStandardHamster();
  String antwort =
    Hamster.getStandardHamster().liesZeichenkette(
      "Moechten Sie textuelle Erlaeuterungen (ja/nein)?");
    SortierHamster sortierer = new BubbleSortHamster(antwort.equals("ja"));
  sortierer.sortiereKoernerHaufen();
}
```

2.3.3 Analyse des Algorithmus

Der BubbleSort-Algorithmus ist stabil und arbeitet in-place.

Sei n die Anzahl an zu sortierenden Elementen. Die Anzahl an Vergleichen zwischen den Array-Elementen beträgt

- $n-1$, wenn das Array bereits aufsteigend sortiert ist,

- $\approx n^2/2$, wenn das Array absteigend sortiert ist,
- im Mittel $\approx n^2/2$, wenn die Anordnung der Elemente im Array zufällig ist.

Die Anzahl an Vertauschungen von Array-Elementen beträgt

- 0, wenn das Array bereits aufsteigend sortiert ist,
- $\approx n^2/2$, wenn das Array absteigend sortiert ist,
- im Mittel $\approx n^2/2$, wenn die Anordnung der Elemente im Array zufällig ist.

Im Web finden sich zahlreiche weitere Informationen zum BubbleSort-Algorithmus, z.B. unter

- `http://de.wikipedia.org/wiki/Bubblesort`
- `http://www.iti.fh-flensburg.de/lang/algorithmen/sortieren/networks/bubble.htm`
- `http://www.sortieralgorithmen.de/bubblesort/index.html`
- `http://olli.informatik.uni-oldenburg.de/fpsort/bubble.html`

2.4 InsertionSort: Sortieren durch Einfügen

Der Sortieralgorithmus „Sortieren durch Einfügen“ – im Englischen *InsertionSort* genannt – ist der Sortieralgorithmus, den die meisten Menschen intuitiv benutzen, wenn sie bei Kartenspielen ihre Karten in der Hand sortieren: Die Karten werden der Reihe nach betrachtet und an der entsprechenden Position einsortiert.

2.4.1 Algorithmus

Der InsertionSort-Algorithmus lässt sich folgendermaßen skizzieren:

Gegeben sei ein Array mit n Elementen, das in aufsteigender Reihenfolge sortiert werden soll. Die Elemente werden ab Index 1 der Reihe nach von links nach rechts betrachtet.

- Merke dir das Element bei Index i.
- Suche links von Index i den so genannten *Einfügeindex* des Elementes. Die Suche starte dabei bei Index i-1 und läuft von rechts nach links. Es wird jeweils verglichen, ob das Element am Suchindex kleiner oder gleich dem gemerkten Element ist. Ist das nicht der Fall, wird das Element des Suchindex um eine Position nach rechts verschoben. Wird ein solches Element bei Index j gefunden, lautet der Einfügeindex j+1. Wird überhaupt kein solches Element gefunden, lautet der Einfügeindex 0.

- Füge das gemerkte Element beim Einfügeindex in das Array ein.

Das Array besteht also aus einem sortierten vorderen Teil, der in jedem Durchlauf um ein Element wächst, und einem unsortierten hinteren Teil, der entsprechend schrumpft. Betrachtet wird in jedem Durchlauf das erste Element des unsortierten Teils, das in den bereits sortierten Teil einsortiert wird.

Die folgende Klasse `InsertionSort` implementiert das Interface `SortierAlgorithmus` entsprechend des InsertionSort-Algorithmus:

```
public class InsertionSort implements SortierAlgorithmus {

  // sortiert das uebergebene Array in aufsteigender Reihenfolge
  // gemaess dem InsertionSort-Algorithmus
  public void sortiere(int[] zahlen) {
    // durchlaufe alle Zahlen ab der zweiten
    for (int aktIndex=1; aktIndex<zahlen.length; aktIndex++) {
      int aktuelleZahl = zahlen[aktIndex];
      int einfuegeIndex = sucheEinfuegeIndex(zahlen, aktuelleZahl, aktIndex);
      zahlen[einfuegeIndex] = aktuelleZahl;
    }
  }

  // sucht und liefert den Index des Arrays, wo
  // die aktuelle Zahl eingefuegt werden muss, und verschiebt alle
  // größeren Zahlen jeweils um eine Position nach hinten im Array
  private int sucheEinfuegeIndex(int[] zahlen, int zahl, int aktIndex) {
  int vergleichsIndex = aktIndex-1;
  while (vergleichsIndex >= 0 && zahlen[vergleichsIndex] > zahl) {
    // Zahl im Array um eine Position nach hinten verschieben
    zahlen[vergleichsIndex+1] = zahlen[vergleichsIndex];
    vergleichsIndex--;
  }
  return vergleichsIndex+1;
  }

  /* der InsertionSort-Algorithmus in kompakter Form
  public void sortiere(int[] zahlen) {
  for (int aktIndex=1; aktIndex<zahlen.length; aktIndex++) {
    int aktuelleZahl = zahlen[aktIndex];
    int vergleichsIndex = aktIndex-1;
    while (vergleichsIndex >= 0 && zahlen[vergleichsIndex] > aktuelleZahl) {
      zahlen[vergleichsIndex+1] = zahlen[vergleichsIndex];
      vergleichsIndex--;
    }
    zahlen[vergleichsIndex+1] = aktuelleZahl;
  }
  }
  */
}
```

2.4.2 Visualisierendes Hamster-Programm

Die folgende Hamster-Klasse `InsertionSortHamster` implementiert das Interface `SortierHamster` und visualisiert dabei den InsertionSort-Algorithmus. Voraussetzung ist, dass es im Territorium zwei nicht durch Mauern blockierte Reihen unterhalb des Standard-Hamsters gibt. In der ersten Reihe unterhalb der Körnerhaufenreihe markiert ein Markierungshamster (grüner Hamster) jeweils den Körnerhaufen, der in den sortierten Teil einsortiert werden soll. Ein Körnerhaufensortierhamster als Vertretungshamster des Standard-Hamsters (roter Hamster) deponiert in jedem Durchlauf zunächst den betrachteten Körnerhaufen um zwei Reihen nach unten. Anschließend sucht er im linken Teil den Einfügeindex und verschiebt dabei unter Umständen größere Körnerhaufen nach rechts. Hat er den Einfügeindex gefunden, holt er die deponierten Körner und legt sie auf der entsprechenden Kachel ab.

```
// Demonstration des InsertionSort-Algorithmus
public class InsertionSortHamster
  extends KoernerHaufenSortierHamster
  implements SortierHamster
{

  private MarkierungsHamster aktIndexHamster = null;
    // markiert den aktuell betrachteten Koernerhaufen

  private int anzahlKoernerHaufen = 0;
    // Anzahl der zu sortierenden Koernerhaufen

  private int anzahlDeponierteKoerner = 0;
    // Koerneranzahl des aktuellen Koernerhaufens

  // Konstruktor
  public InsertionSortHamster(boolean mitErlaeuterungen) {
    super(mitErlaeuterungen);
    this.erlaeuterung(
      "Ich sortiere die Koernerhaufen auf der Basis des " +
      "InsertionSort-Algorithmus.");
    this.aktIndexHamster =
      new MarkierungsHamster(this.getReihe() + 1, this.getSpalte(),
                             this.getReihe(), mitErlaeuterungen);
    this.aktIndexHamster.erlaeuterung(
      "Ich markiere jeweils den Koernerhaufen,\n" +
      "der an der richtigen Position eingefuegt werden soll.");
    this.anzahlDeponierteKoerner = 0;
  }

  // Der Standard-Hamster steht mit Blickrichtung OST irgendwo im Territorium.
  // Ein Vertretungshamster soll die Koernerhaufen bis zur nächsten Wand in
  // aufsteigender Reihenfolge sortieren.
  // Voraussetzung: Unterhalb des Standard-Hamsters existieren zwei nicht
  // durch Mauern blockierte Reihen.
  public void sortiereKoernerHaufen() {
```

```
    this.erlaeuterung(
      "Ich zaehle nun die Anzahl der zu sortierenden Koernerhaufen.");
    this.anzahlKoernerHaufen = ermittleAnzahlKoernerHaufen();
    this.erlaeuterung(
      "Die Anzahl der zu sortierenden Koernerhaufen betraegt " +
      this.anzahlKoernerHaufen +
      ",\nd.h. die Indizes der Haufen liegen zwischen 0 und " +
      (this.anzahlKoernerHaufen-1) +
      ".");

    // durchlaufe die Menge der Koernerhaufen von links nach rechts;
    // merke dir den aktuellen Koernerhaufen
    for (int aktIndex = 1; aktIndex < this.anzahlKoernerHaufen; aktIndex++) {
      this.aktIndexHamster.markiereIndex(aktIndex);
      this.erlaeuterung(
        "Ich suche nun den Einfuegeindex des " +
        "markierten Koernerhaufens bei Index " +
        this.aktIndexHamster.liefereIndex() +
        ".\nDazu deponiere ich zunaechst die " +
        this.aktIndexHamster.liefereAnzahlKoerner() +
        " Koerner des markierten Koernerhaufens.");
      this.deponiereKoerner();
      this.erlaeuterung(
        "Ich suche nun zwischen den Indizes 0 und " +
        (aktIndex-1) +
        " den ersten Koernerhaufen von rechts,\n" +
        " der weniger oder gleich " +
        this.anzahlDeponierteKoerner +
        " Koerner besitzt.\n" +
        "Rechts von diesem befindet sich der Einfuegeindex.\n" +
        "Groessere Haufen verschiebe ich dabei jeweils " +
        "um einen Index nach rechts.");
      int einfuegeIndex = this.sucheEinfuegeIndex();
      this.erlaeuterung(
        "Einfuegeindex gefunden. Er liegt bei Index " +
        einfuegeIndex +
        ".\nIch hole nun die deponierten Koerner und lege sie hier ab.");
      this.fuegeDeponierteKoernerEin(einfuegeIndex);
    }
    this.beendeSortierung();
  }

  private void deponiereKoerner() {
    // die Koerner werden zwei Reihen darunter deponiert
    this.laufeZuIndex(this.aktIndexHamster.liefereIndex());
    this.anzahlDeponierteKoerner = this.nimmAlle();
    this.setzeBlickrichtung(Hamster.SUED);
    this.vor(2);
    this.gib(this.anzahlDeponierteKoerner);
    this.kehrt();
    this.vor(2);
    this.linksUm();
  }
```

```
  private int sucheEinfuegeIndex() {
    // sucht und liefert den Index der Koernerhaufen, wo
    // der aktuelle Koernerhaufen - sprich die deponierten Koerner -
    //  eingefuegt werden muss, und verschiebt alle
    // größeren Koernerhaufen jeweils um eine Position nach rechts
    this.vor();
    while (this.liefereIndex() >= 0 &&
           this.liefereAnzahlKoerner() > this.anzahlDeponierteKoerner) {
      int anzahl = this.nimmAlle();
      this.kehrt();
      this.vor();
      this.gib(anzahl);
      this.kehrt();
      this.vor();
      this.vor();
    }
    return this.liefereIndex()+1;
  }

  private void fuegeDeponierteKoernerEin(int index) {
    // holt die deponierten Koerner und fuegt sie an der uebergebenen
    // Position in die Menge der Koernerhaufen ein
    this.laufeZuKachel(this.getReihe() + 2, this.aktIndexHamster.getSpalte());
    int anzahl = this.nimmAlle();
    this.laufeZuKachel(this.getReihe() - 2, this.startSpalte + index + 1);
    this.gib(anzahl);
  }

  private void beendeSortierung() {
    this.laufeZuSpalte(this.startSpalte);
    this.setzeBlickrichtung(Hamster.OST);
    this.erlaeuterung("Sortierung erfolgreich beendet!");
  }
}
```

Mit dem folgendem objektorientierten Hamster-Programm (`SortierenMitInsertionSort`) können Sie sich von den Hamstern den InsertionSort-Algorithmus demonstrieren lassen.

```
void main() {
  Hamster paul = Hamster.getStandardHamster();
  String antwort =
    Hamster.getStandardHamster().liesZeichenkette(
      "Moechten Sie textuelle Erlaeuterungen (ja/nein)?");
  SortierHamster sortierer = new InsertionSortHamster(antwort.equals("ja"));
  sortierer.sortiereKoernerHaufen();
}
```

2.4.3 Analyse des Algorithmus

Der InsertionSort-Algorithmus ist stabil und arbeitet in-place.

Sei n die Anzahl an zu sortierenden Elementen. Die Anzahl an Vergleichen zwischen den Array-Elementen beträgt

- $n - 1$, wenn das Array bereits aufsteigend sortiert ist,
- $\approx n^2/2$, wenn das Array absteigend sortiert ist,
- im Mittel $\approx n^2/4$, wenn die Anordnung der Elemente im Array zufällig ist.

Die Anzahl an „halben Vertauschungen“ (Bewegungen) von Array-Elementen beträgt

- 0, wenn das Array bereits aufsteigend sortiert ist,
- $\approx n^2/2$, wenn das Array absteigend sortiert ist,
- im Mittel $\approx n^2/4$, wenn die Anordnung der Elemente im Array zufällig ist.

Im Web finden sich zahlreiche weitere Informationen zum InsertionSort-Algorithmus, z.B. unter

- `http://de.wikipedia.org/wiki/Insertionsort`
- `http://www.iti.fh-flensburg.de/lang/algorithmen/sortieren/insertion.htm`
- `http://www.sortieralgorithmen.de/insertsort/index.html`

2.5 ShellSort: Verbessertes Sortieren durch Einfügen

Eine Variante des InsertionSort-Algorithmus, die im Allgemeinen schneller ist, nennt sich *ShellSort*.

2.5.1 Algorithmus

Schauen Sie sich nochmal den InsertionSort-Algorithmus an. Sein Problem ist, dass beim Suchen des Einfügeindex größere Elemente immer nur jeweils um einen Index nach hinten verschoben werden. Wenn sich das kleinste Element zufällig am Ende eines Arrays der Länge n befindet, dann sind n-1 Verschiebeoperationen notwendig, bevor das kleinste Element anschließend ganz vorne einsortiert wird. Die Anzahl dieser Verschiebeoperationen, die beim Transport eines Elementes an seine korrekte Position notwendig sind, werden beim Shellsort-Algorithmus reduziert.

Die Grundidee des ShellSort-Algorithmus ist dabei die, dass der InsertionSort-Algorithmus in mehreren Schritten wiederholt angewendet wird. In einem Schritt werden dabei nicht alle Elemente des Arrays, sondern nur Elemente mit einem bestimmten Abstand d im Array betrachtet und sortiert.

Habe d bspw. den anfänglichen Wert 4. Dann werden die Elemente an den Positionen 0, 4, 8, 12, 16, ... sortiert, ebenso die Elemente an den Positionen 1, 5, 9, 13, 17, ... sowie 2, 6, 10, 14, 18, ... und 3, 7, 11, 15, 19, ... Nach diesem Schritt besteht das Array also im Prinzip aus 4 sortierten Teil-Arrays. Es wird auch 4-sortiert bzw. allgemein d-sortiert genannt.

Im zweiten Schritt wird d verkleinert und die Sortierung wiederholt. Für d=2 werden bspw. die Teil-Arrays mit den Elementen an den Positionen 0, 2, 4, 6, 8, 10, 12, 14, 16, 18, ... und 1, 3, 5, 7, 9, 11, 13, 15, 17, 19, ... sortiert. Anschließend ist das Array 2-sortiert.

Die Verkleinerung von d und die anschließende d-Sortierung wird so lange wiederholt bis d=1 gilt. Im Fall d=1 wird ja der normale InsertionSort-Algorithmus ausgeführt und das Array ist damit sortiert.

Der Klou an diesem Prinzip ist, dass bei großem d kleine Elemente, die sich hinten im Array befinden, über größere Entfernungen – also mit weniger Verschiebeoperationen – nach vorne bewegt werden können.

Für viele Programmieranfänger ist der Shellsort-Algorithmus nicht auf Anhieb verständlich. Aus diesem Grund möchte ich an einem Beispiel eine alternative Beschreibung vorstellen. Zu sortieren sei ein Array mit den Elementen `5 9 2 4 3 1 7 9 8 2 4 3`.

Habe d den Anfangswert 4. Dann kann man sich das Array vorstellen als eine 2-dimensionale Matrix mit 4 Spalten:

```
5 9 2 4
3 1 7 9
8 2 4 3
```

Diese Matrix wird nun spaltenweise mittels des InsertionSort-Algorithmus sortiert und es ergibt sich:

```
3 1 2 3
5 2 4 4
8 9 7 9
```

Das eigentliche Array ist damit 4-sortiert und hat die Gestalt `3 1 2 3 5 2 4 4 8 9 7 9`.

d wird nun im zweiten Schritt auf den Wert 2 gesetzt. Dann kann man sich das Array vorstellen als eine Matrix mit 2 Spalten:

```
3 1
2 3
5 2
4 4
8 9
7 9
```

Diese Matrix wird wiederum spaltenweise mittels des InsertionSort-Algorithmus sortiert und es ergibt sich:

```
2 1
3 2
4 3
5 4
7 9
8 9
```

Das eigentliche Array ist damit 2-sortiert und hat die Gestalt `2 1 3 2 4 3 5 4 7 9 8 9`.

Im letzten Schritt wird d auf den Wert 1 gesetzt. Die entsprechende Matrix hat nur eine Spalte, die mittels des InsertionSort-Algorithmus sortiert wird. Damit ist die Sortierung abgeschlossen und es ergibt sich das sortierte Array `1 2 2 3 3 4 4 5 7 8 9 9`.

Eine Frage, die sich noch stellt, ist die Wahl der Werte von d, auch *Distanzfolge* genannt. Hierzu gibt es viele Untersuchungen, inwieweit die Laufzeit des ShellSort-Algorithmus durch geeignete Distanzfolgen verbessert werden kann.

Die einfachste Distanzfolge – auch *Distanzfolge nach Shell* genannt – lautet ..., 64, 32, 16, 8, 4, 2, 1. Sie entspricht also den Zweierpotenzen. Problem dieser Distanzfolge ist, dass Elemente an ungeraden Positionen erst am Schluss mit Elementen an geraden Positionen verglichen werden. Bessere Ergebnisse liefern die Distanzfolge nach Papernov-Stasevich (..., 511, 255, 127, 63, 31, 15, 7, 3, 1) und die Distanzfolge nach Knuth (..., 1093, 364, 121, 40, 13, 4, 1). In allen Fällen wird das anfängliche d auf die größte Zahl der Folge gesetzt, die kleiner als die Länge des zu sortierenden Arrays ist.

Die folgende Klasse `ShellSort` implementiert das Interface `SortierAlgorithmus` entsprechend des ShellSort-Algorithmus:

```
public class ShellSort implements SortierAlgorithmus {

  // sortiert das uebergebene Array in aufsteigender Reihenfolge
  // gemaess dem ShellSort-Algorithmus
```

```
public void sortiere(int[] zahlen) {
  sortiere(zahlen, new ShellDistanz(zahlen.length));
  //sortiere(zahlen, new PapernovStasevichDistanz(zahlen.length));
  //sortiere(zahlen, new KnuthDistanz(zahlen.length));
}

// sortiert das uebergebene Array in aufsteigender Reihenfolge
// gemaess dem ShellSort-Algorithmus und der uebergebenen Distanz
public void sortiere(int[] zahlen, Distanz distanzObjekt) {
  int distanz = distanzObjekt.berechneNaechsteDistanz();
  while (distanz > 0) {
    for (int abIndex = 0; abIndex < distanz; abIndex++) {
      insertionSort(zahlen, abIndex, distanz);
    }
    distanz = distanzObjekt.berechneNaechsteDistanz();
  }
}

// analog dem InsertionSort-Algorithmus; es werden jedoch nur die
// Elemente ab "abIndex" beruecksichtigt, die voneinander eine
// Entfernung "distanz" haben
public void insertionSort(int[] zahlen, int abIndex, int distanz) {
  // durchlaufe alle Zahlen ab der zweiten
  for (int aktIndex = abIndex+distanz;
           aktIndex < zahlen.length;
           aktIndex += distanz) {
    int aktuelleZahl = zahlen[aktIndex];
    int einfuegeIndex =
      sucheEinfuegeIndex(zahlen, aktuelleZahl, aktIndex, distanz);
    zahlen[einfuegeIndex] = aktuelleZahl;
  }
}

// sucht und liefert vor der aktuellen Zahl den Index des Arrays, wo
// die aktuelle Zahl eingefuegt werden muss, und verschiebt alle
// größeren Zahlen jeweils um "distanz" Positionen nach hinten im Array
private int sucheEinfuegeIndex(int[] zahlen, int zahl,
                                 int aktIndex, int distanz) {
  int vergleichsIndex = aktIndex - distanz;
  while (vergleichsIndex >= 0 && zahlen[vergleichsIndex] > zahl) {
    // Zahl im Array um eine Position nach hinten verschieben
    zahlen[vergleichsIndex + distanz] = zahlen[vergleichsIndex];
    vergleichsIndex -= distanz;
  }
  return vergleichsIndex + distanz;
}

/* der ShellSort-Algorithmus in kompakter Form
public void sortiere(int[] zahlen, Distanz distanzObjekt) {
  int distanz = distanzObjekt.berechneNaechsteDistanz();
  while (distanz > 0) {
    for (int abIndex = 0; abIndex < distanz; abIndex++) {
      for (int aktIndex=abIndex+distanz;
```

```
                     aktIndex < zahlen.length;
                     aktIndex += distanz) {
          int aktuelleZahl = zahlen[aktIndex];
          int vergleichsIndex = aktIndex-distanz;
          while (vergleichsIndex >= 0 && zahlen[vergleichsIndex] > aktuelleZahl) {
            zahlen[vergleichsIndex+distanz] = zahlen[vergleichsIndex];
            vergleichsIndex -= distanz;
          }
          zahlen[vergleichsIndex+distanz] = aktuelleZahl;
        }
      }
      distanz = distanzObjekt.berechneNaechsteDistanz();
    }
  }
  */
}

// Berechnung der ShellSort-Distanz
interface Distanz {
  public int berechneNaechsteDistanz();
}

// Distanz nach D.L. Shell (1959)
class ShellDistanz implements Distanz {

  int distanz;

  ShellDistanz(int arrayLaenge) {
    this.distanz = 1;
    while (this.distanz < arrayLaenge) {
      this.distanz = this.distanz * 2;
    }
  }

  public int berechneNaechsteDistanz() {
    this.distanz = this.distanz / 2;
    return this.distanz;
  }
}

// Distanz nach Papernov-Stasevich (1965)
class PapernovStasevichDistanz implements Distanz {

  int distanz;

  PapernovStasevichDistanz(int arrayLaenge) {
    this.distanz = 1;
    while (this.distanz < arrayLaenge) {
      this.distanz = (this.distanz + 1) * 2 - 1;
    }
  }

  public int berechneNaechsteDistanz() {
```

```
    this.distanz = (this.distanz + 1) / 2 - 1;
    return this.distanz;
  }
}

// Distanz nach Knuth
class KnuthDistanz implements Distanz {

  int distanz;

  KnuthDistanz(int arrayLaenge) {
    this.distanz = 1;
    while (this.distanz < arrayLaenge) {
      this.distanz = this.distanz * 3 + 1;
    }
  }

  public int berechneNaechsteDistanz() {
    this.distanz = (this.distanz - 1) / 3;
    return this.distanz;
  }
}
```

2.5.2 Visualisierendes Hamster-Programm

Die folgende Hamster-Klasse `ShellSortHamster` implementiert das Interface `SortierHamster` und visualisiert dabei den ShellSort-Algorithmus. Voraussetzung ist, dass es im Territorium drei nicht durch Mauern blockierte Reihen unterhalb des Standard-Hamsters gibt. Die Visualisierung erfolgt analog zur Visualisierung des InsertionSort-Algorithmus. Dabei markieren in der zweiten Reihe unterhalb der Körnerhaufenreihe eine Menge von Hamstern die Elemente des aktuell betrachteten Teil-Arrays.

```
public class ShellSort implements SortierAlgorithmus {

  // sortiert das uebergebene Array in aufsteigender Reihenfolge
  // gemaess dem ShellSort-Algorithmus
  public void sortiere(int[] zahlen) {
    sortiere(zahlen, new ShellDistanz(zahlen.length));
    //sortiere(zahlen, new PapernovStasevichDistanz(zahlen.length));
    //sortiere(zahlen, new KnuthDistanz(zahlen.length));
  }

  // sortiert das uebergebene Array in aufsteigender Reihenfolge
  // gemaess dem ShellSort-Algorithmus und der uebergebenen Distanz
  public void sortiere(int[] zahlen, Distanz distanzObjekt) {
    int distanz = distanzObjekt.berechneNaechsteDistanz();
    while (distanz > 0) {
      for (int abIndex = 0; abIndex < distanz; abIndex++) {
        insertionSort(zahlen, abIndex, distanz);
```

```
    }
    distanz = distanzObjekt.berechneNaechsteDistanz();
  }
}

// analog dem InsertionSort-Algorithmus; es werden jedoch nur die
// Elemente ab "abIndex" beruecksichtigt, die voneinander eine
// Entfernung "distanz" haben
public void insertionSort(int[] zahlen, int abIndex, int distanz) {
  // durchlaufe alle Zahlen ab der zweiten
  for (int aktIndex = abIndex+distanz;
           aktIndex < zahlen.length;
           aktIndex += distanz) {
    int aktuelleZahl = zahlen[aktIndex];
    int einfuegeIndex =
      sucheEinfuegeIndex(zahlen, aktuelleZahl, aktIndex, distanz);
    zahlen[einfuegeIndex] = aktuelleZahl;
  }
}

// sucht und liefert vor der aktuellen Zahl den Index des Arrays, wo
// die aktuelle Zahl eingefuegt werden muss, und verschiebt alle
// größeren Zahlen jeweils um "distanz" Positionen nach hinten im Array
private int sucheEinfuegeIndex(int[] zahlen, int zahl,
                                 int aktIndex, int distanz) {
  int vergleichsIndex = aktIndex - distanz;
  while (vergleichsIndex >= 0 && zahlen[vergleichsIndex] > zahl) {
    // Zahl im Array um eine Position nach hinten verschieben
    zahlen[vergleichsIndex + distanz] = zahlen[vergleichsIndex];
    vergleichsIndex -= distanz;
  }
  return vergleichsIndex + distanz;
}

/* der ShellSort-Algorithmus in kompakter Form
public void sortiere(int[] zahlen, Distanz distanzObjekt) {
  int distanz = distanzObjekt.berechneNaechsteDistanz();
  while (distanz > 0) {
    for (int abIndex = 0; abIndex < distanz; abIndex++) {
      for (int aktIndex=abIndex+distanz;
                 aktIndex < zahlen.length;
                 aktIndex += distanz) {
        int aktuelleZahl = zahlen[aktIndex];
        int vergleichsIndex = aktIndex-distanz;
        while (vergleichsIndex >= 0 && zahlen[vergleichsIndex] > aktuelleZahl) {
          zahlen[vergleichsIndex+distanz] = zahlen[vergleichsIndex];
          vergleichsIndex -= distanz;
        }
        zahlen[vergleichsIndex+distanz] = aktuelleZahl;
      }
    }
    distanz = distanzObjekt.berechneNaechsteDistanz();
  }
```

```
  }
  */
}

// Berechnung der ShellSort-Distanz
interface Distanz {
  public int berechneNaechsteDistanz();
}

// Distanz nach D.L. Shell (1959)
class ShellDistanz implements Distanz {

  int distanz;

  ShellDistanz(int arrayLaenge) {
    this.distanz = 1;
    while (this.distanz < arrayLaenge) {
      this.distanz = this.distanz * 2;
    }
  }

  public int berechneNaechsteDistanz() {
    this.distanz = this.distanz / 2;
    return this.distanz;
  }
}

// Distanz nach Papernov-Stasevich (1965)
class PapernovStasevichDistanz implements Distanz {

  int distanz;

  PapernovStasevichDistanz(int arrayLaenge) {
    this.distanz = 1;
    while (this.distanz < arrayLaenge) {
      this.distanz = (this.distanz + 1) * 2 - 1;
    }
  }

  public int berechneNaechsteDistanz() {
    this.distanz = (this.distanz + 1) / 2 - 1;
    return this.distanz;
  }
}

// Distanz nach Knuth
class KnuthDistanz implements Distanz {

  int distanz;

  KnuthDistanz(int arrayLaenge) {
    this.distanz = 1;
    while (this.distanz < arrayLaenge) {
```

```
      this.distanz = this.distanz * 3 + 1;
    }
  }

  public int berechneNaechsteDistanz() {
    this.distanz = (this.distanz - 1) / 3;
    return this.distanz;
  }
}
```

Mit dem folgendem objektorientierten Hamster-Programm (`SortierenMitShellSort`) können Sie sich von den Hamstern den ShellSort-Algorithmus demonstrieren lassen.

```
void main() {
  Hamster paul = Hamster.getStandardHamster();
  String antwort =
    Hamster.getStandardHamster().liesZeichenkette(
      "Moechten Sie textuelle Erlaeuterungen (ja/nein)?");
  SortierHamster sortierer = new ShellSortHamster(antwort.equals("ja"));
  sortierer.sortiereKoernerHaufen();
}
```

2.5.3 Analyse des Algorithmus

Der ShellSort-Algorithmus ist nicht stabil. ShellSort arbeitet in-place.

Eine Analyse des ShellSort-Algorithmus ist schwierig und hängt natürlich auch von der gewählten Distanzfolge ab. Es kann gezeigt werden, dass für eine Vielzahl von Distanzfolgen die Anzahl an Vergleichen zwischen den Array-Elementen im unglücklichsten Fall $\approx n^{1.5}$ beträgt. n ist dabei die Anzahl an zu sortierenden Elementen.

Im Web finden sich zahlreiche weitere Informationen zum ShellSort-Algorithmus, z.B. unter

- `http://de.wikipedia.org/wiki/Shellsort`
- `http://www.iti.fh-flensburg.de/lang/algorithmen/sortieren/shell/shell.htm`
- `http://www.sortieralgorithmen.de/shellsort/index.html`

2.6 QuickSort: Sortieren durch rekursives Zerlegen

Der heutzutage wohl am häufigsten eingesetzte Sortieralgorithmus nennt sich *QuickSort*. Es existieren einige Varianten dieses Algorithmus, dessen grundlegende Version, die in diesem Abschnitt vorgestellt wird, bereits 1960 von C. A. R. Hoare entwickelt wurde. Der Erfolg von QuickSort beruht darauf, dass er zum einen leicht verständlich und zum anderen im Mittel schneller ist, als jeder andere Sortieralgorithmus.

2.6.1 Algorithmus

QuickSort zerlegt (man spricht auch von *Partitionierung*) das zu sortierende Array in zwei Teil-Arrays, die durch ein Referenz-Element – auch *Pivot-Element* genannt – getrennt werden. Alle Elemente des einen Teil-Arrays sind dabei kleiner oder gleich alle Elemente des anderen Teil-Arrays größer oder gleich dem Wert des Pivot-Elementes. Das bedeutet gleichzeitig, dass das Pivot-Element an seinem korrekten Platz im Array steht. Anschließend wird QuickSort rekursiv für die beiden Teil-Arrays aufgerufen. Die Rekursion endet, wenn das Teil-Array nur noch aus einem Element besteht.

Etwas ausführlicher lässt sich das zugrunde liegende Prinzip des Algorithmus folgendermassen skizzieren:

1. Das zu sortierende (Teil-)Array habe mehr als ein Element.
2. Dann wähle ein beliebiges Element des Arrays – das so genannte *Pivot-Element*.
3. Positioniere das Pivot-Element an seine endgültige Position p im Array.
4. Sorge dabei dafür, dass alle Elemente des Arrays links vom Pivot-Element wertmäßig kleiner oder gleich diesem sind und
5. dass alle Elemente des Arrays rechts vom Pivot-Element wertmäßig größer oder gleich diesem sind.
6. Rufe QuickSort rekursiv für das Teil-Array vor dem Pivot-Element auf.
7. Rufe QuickSort rekursiv für das Teil-Array nach dem Pivot-Element auf.

Eine einfache Strategie zur Implementierung der Partitionierung (Punkte 1 bis 5) ist folgende: Als Pivot-Element wird willkürlich das rechte Element des (Teil-)Arrays gewählt. Von links wird das Array nach einem Element durchsucht, das größer als das Pivot-Element ist. Von rechts wird das Array nach einem Element durchsucht, das kleiner als das Pivot-Element ist. Diese beiden Elemente sind offensichtlich jeweils im falschen Teil-Array und werden daher getauscht. Mit dieser Suche und dem Austausch wird fortgefahren, bis sich die Suchzeiger treffen bzw. kreuzen. Dann ist sicher gestellt, dass alle Elemente links vom linken Zeiger kleiner oder gleich und alle Elemente rechts vom rechten Zeiger größer oder gleich dem Pivot-Element sind. Als Index des Pivot-Elementes wählt man abschließend den Index, auf den der linke Zeiger zeigt, und tauscht dieses Element (das ja größer oder gleich dem Pivot-Element ist) mit dem Pivot-Element ganz rechts im Array.

Die Wahl des Pivot-Elementes ist eine viel diskutierte Eigenschaft des QuickSort-Algorithmus, da davon maßgeblich die Geschwindigkeit des Algorithmus beeinflusst wird. Wir haben das Element am rechten Rand gewählt. Unglücklich ist, wenn das gewählte Element das größte oder kleinste Element des Arrays ist, weil dies eine sehr ungleichmäßige Zerlegung des Arrays in Teil-Arrays und einen häufigen rekursiven Aufruf des Algorithmus bedingt.

In der Literatur wird oft das mittlere Element genommen. Prinzipiell ist die Wahl jedoch beliebig. Man kann auch bspw. drei Elemente auswählen und davon das Element mit dem mittleren Wert nehmen. Der oben beschriebene Partitionierungsalgorithmus muss in diesen Fällen einfach dahingehend abgeändert werden, dass nach der Wahl des Pivot-Elementes dieses zunächst mit dem Element ganz rechts im (Teil-)Array getauscht wird.

Die folgende Klasse `QuickSort` implementiert das Interface `SortierAlgorithmus` entsprechend des QuickSort-Algorithmus:

```
public class QuickSort implements SortierAlgorithmus {

  // sortiert das uebergebene Array in aufsteigender Reihenfolge
  // gemaess dem QuickSort-Algorithmus
  public void sortiere(int[] zahlen) {
    quickSort(zahlen, 0, zahlen.length-1);
  }

  // der Quicksort-Algorithmus wird auf dem Array zwischen den
  // angegebenen Indizes ausgefuehrt
  private void quickSort(int[] zahlen, int linkerIndex, int rechterIndex) {
    if (linkerIndex < rechterIndex) {
      int pivotIndex = zerlege(zahlen, linkerIndex, rechterIndex);
      quickSort(zahlen, linkerIndex, pivotIndex-1);
      quickSort(zahlen, pivotIndex+1, rechterIndex);
    }
  }

  // liefert den Index des Pivot-Elementes und ordnet das Array innerhalb
  // der angegebenen Indizes so um, dass alle Zahlen links vom Index
  // kleiner oder gleich und alle Zahlen rechts vom Index groesser
  // oder gleich dem Pivot-Element sind
  private int zerlege(int[] zahlen, int linkerIndex, int rechterIndex) {
    int pivotIndex = waehlePivotIndex(zahlen, linkerIndex, rechterIndex);
    int pivotWert = zahlen[pivotIndex];
    // das Pivot-Element kommt nach ganz rechts im Array
    tauschen(zahlen, pivotIndex, rechterIndex);
    int l = linkerIndex-1;
    int r = rechterIndex;
    // ordne das Array so um, dass jeweils alle Elemente links vom
    // Zeiger l kleiner und alle Elemente rechts vom Zeiger r groesser
    // als das Pivot-Element sind
    do {
      l++;
      while (l <= rechterIndex && zahlen[l] <= pivotWert) l++;
      r--;
```

```
      while (r >= linkerIndex && zahlen[r] >= pivotWert) r--;
      if (l < r) {
        tauschen(zahlen, l, r);
      }
    } while (l < r);
    // platziere das Pivot-Element an seine korrekte Position
    if (l < rechterIndex) {
      tauschen(zahlen, l, rechterIndex);
      return l;
    } else {
      return rechterIndex;
    }
  }

  // waehlt einen beliebigen Index zwischen den angegebenen Indizes
  private int waehlePivotIndex(int[] zahlen, int linkerIndex, int rechterIndex) {
    // in diesem Fall einfach der mittleren Index
    return (linkerIndex + rechterIndex) / 2;
    /* Alternative 1:
      return rechterIndex;
    */
    /* Alternative 2 (mittleres von drei Elementen):
      int index1 = (linkerIndex + rechterIndex) / 2;
      int index2 = (linkerIndex + index1) / 2;
      int index3 = (index1 + rechterIndex) / 2;
      if (zahlen[index1] <= zahlen[index2] &&
          zahlen[index2] <= zahlen[index3])
        return zahlen[index2];
      if (zahlen[index2] <= zahlen[index3] &&
          zahlen[index3] <= zahlen[index1])
        return zahlen[index3];
      return zahlen[index1];
    */
  }

  // tauscht die Elemente des Arrays an den angegebenen Indizes
  private void tauschen(int[] zahlen, int index1, int index2) {
    if (index1 != index2) {
      int help = zahlen[index1];
      zahlen[index1] = zahlen[index2];
      zahlen[index2] = help;
    }
  }
}
```

2.6.2 Visualisierendes Hamster-Programm

Die folgende Hamster-Klasse `QuickSortHamster` implementiert das Interface `SortierHamster` und visualisiert dabei den QuickSort-Algorithmus. Voraussetzung ist, dass es im Territorium zwei nicht durch Mauern blockierte Reihen unterhalb des Standard-Hamsters gibt. In der ersten Reihe unterhalb der Körnerhaufenreihe markieren zwei Markierungshamster jeweils den linken und rechten Rand des aktuell betrachteten Teil-Arrays. Der rechte Hamster zeigt also auch auf das Pivot-Element. Zwei weitere Markierungshamster in der Reihe darunter entsprechen den Suchzeigern des Partitionierungsalgorithmus.

```
// Demonstration des QuickSort-Algorithmus
public class QuickSortHamster
  extends KoernerHaufenSortierHamster
  implements SortierHamster
{

  private MarkierungsHamster linkerIndexHamster = null;
    // markiert den linken Rand des aktuellen Teil-Arrays

  private MarkierungsHamster rechterIndexHamster = null;
    // markiert den rechten Rand des aktuellen Teil-Arrays
    // (gleichzeitig auch das Pivot-Element!)

  private MarkierungsHamster linkerZeigerHamster = null;
    // markiert waehrend der Partitionierung das aktuell
    // betrachtete linke Element

  private MarkierungsHamster rechterZeigerHamster = null;
    // markiert waehrend der Partitionierung das aktuell
    // betrachtete rechte Element

  private int anzahlKoernerHaufen = 0;
    // Anzahl der zu sortierenden Koernerhaufen

  // Konstruktor
  public QuickSortHamster(boolean mitErlaeuterungen) {
    super(mitErlaeuterungen);
    this.erlaeuterung(
      "Ich sortiere die Koernerhaufen auf der Basis des QuickSort-Algorithmus.");
    this.linkerIndexHamster =
      new MarkierungsHamster(this.getReihe()+1, this.getSpalte(),
                             this.getReihe(), mitErlaeuterungen);
    this.linkerIndexHamster.erlaeuterung(
      "Ich markiere den linken Rand des aktuell betrachteten Teil-Arrays.");
    this.rechterIndexHamster =
      new MarkierungsHamster(this.getReihe()+1, this.getSpalte(),
                             this.getReihe(), mitErlaeuterungen);
    this.rechterIndexHamster.erlaeuterung(
     "Ich markiere den rechten Rand des aktuell betrachteten Teil-Arrays " +
     " und\ngleichzeitig auch den aktuellen Pivot-Haufen.");
    this.linkerZeigerHamster =
      new MarkierungsHamster(this.getReihe()+2, this.getSpalte(),
                             this.getReihe(), mitErlaeuterungen);
    this.linkerZeigerHamster.erlaeuterung(
      "Ich markiere waehrend der Partitionierung den aktuell " +
      "betrachteten linken Haufen.");
    this.rechterZeigerHamster =
      new MarkierungsHamster(this.getReihe()+2, this.getSpalte(),
```

```
                          this.getReihe(), mitErlaeuterungen);
  this.rechterZeigerHamster.erlaeuterung(
    "Ich markiere waehrend der Partitionierung den aktuell " +
    "betrachteten rechten Haufen.");
}

// Der Standard-Hamster steht mit Blickrichtung OST irgendwo im Territorium.
// Ein Vertretungshamster soll die Koernerhaufen bis zur nächsten Wand in
// aufsteigender Reihenfolge sortieren.
// Voraussetzung: Unterhalb des Standard-Hamsters existieren zwei nicht
// durch Mauern blockierte Reihen.
public void sortiereKoernerHaufen() {
  this.erlaeuterung(
   "Ich zaehle nun die Anzahl der zu sortierenden Koernerhaufen.");
  this.anzahlKoernerHaufen = this.ermittleAnzahlKoernerHaufen();
  this.erlaeuterung(
    "Die Anzahl der zu sortierenden Koernerhaufen betraegt " +
    this.anzahlKoernerHaufen +
    ",\nd.h. die Indizes der Haufen liegen zwischen 0 und " +
    (this.anzahlKoernerHaufen-1) +
    ".");
  this.quickSort(0, this.anzahlKoernerHaufen-1);
  this.beendeSortierung();
}

// der Quicksort-Algorithmus wird auf dem Array zwischen den
// angegebenen Indizes ausgefuehrt
private void quickSort(int linkerIndex, int rechterIndex) {
  if (linkerIndex < rechterIndex) {
    this.linkerIndexHamster.markiereIndex(linkerIndex);
    this.rechterIndexHamster.markiereIndex(rechterIndex);
    this.erlaeuterung(
      "Ich partitioniere nun das Teil-Array zwischen den Indizes " +
      linkerIndex +
      " und " +
      rechterIndex +
      ".");
    int pivotIndex = zerlege(linkerIndex, rechterIndex);
    this.erlaeuterung(
      "Die Partitionierung des Teil-Arrays zwischen den Indizes " +
      linkerIndex +
      " und " +
      rechterIndex +
      " ist abgeschlossen.\n" +
      "Der Pivot-Haufen befindet sich bei Index " +
      pivotIndex +
      ".");
    quickSort(linkerIndex, pivotIndex-1);
    quickSort(pivotIndex+1, rechterIndex);
  }
}

// liefert den Index des Pivot-Elementes und ordnet das Array innerhalb
```

```
// der angegebenen Indizes so um, dass alle Zahlen links vom Index
// kleiner oder gleich und alle Zahlen rechts vom Index groesser
// oder gleich dem Pivot-Element sind
private int zerlege(int linkerIndex, int rechterIndex) {
  int pivotIndex = waehlePivotIndex(linkerIndex, rechterIndex);
  this.laufeZuIndex(pivotIndex);
  int pivotWert = this.liefereAnzahlKoerner();
  this.erlaeuterung(
    "Ich stehe nun auf dem Pivot-Haufen. Hier liegen " +
    pivotWert +
    " Koerner.");
  // das Pivot-Element kommt nach ganz rechts ins Array
  this.erlaeuterung(
    "Ich tausche nun den Pivot-Haufen mit dem rechten " +
    "Haufen des\naktuellen Teil-Arrays (gelber Markierungshamster).");
  this.tauschen(pivotIndex, rechterIndex);
  int l = linkerIndex-1;
  this.linkerZeigerHamster.markiereIndex(l);
  int r = rechterIndex;
  this.rechterZeigerHamster.markiereIndex(r);
  // ordne das Array so um, dass jeweils alle Elemente links vom Zeiger l
  // kleiner gleich und alle Elemente rechts vom Zeiger r groesser
  // gleich dem Pivot-Element sind
  this.erlaeuterung(
    "Das Teil-Array zwischen den Indizes " +
    linkerIndex +
    " und " +
    rechterIndex +
    " wird nun so umgeordnet,\ndass jeweils alle Haufen links vom " +
    "hellblauen Markierungshamster kleiner gleich\n" +
    "und alle Elemente rechts vom violetten Markierungshamster " +
    "groesser gleich\ndem Pivot-Haufen (gelber Markierungshamster) sind.");
  do {
    l++;
    this.linkerZeigerHamster.markiereIndex(l);
    while (l <= rechterIndex &&
           this.linkerZeigerHamster.liefereAnzahlKoerner() <= pivotWert) {
      l++;
      this.linkerZeigerHamster.markiereIndex(l);
    }
    r--;
    this.rechterZeigerHamster.markiereIndex(r);
    while (r >= linkerIndex &&
           this.rechterZeigerHamster.liefereAnzahlKoerner() >= pivotWert) {
      r--;
      this.rechterZeigerHamster.markiereIndex(r);
    }
    if (l < r) {
      this.erlaeuterung(
        "Der Haufen bei Index " +
        l +
        " ist groesser als der Pivot-Haufen\nund der Haufen bei Index " +
        r +
```

```
        " ist kleiner als der Pivot-Haufen.\nSie werden daher getauscht.");
      tauschen(l, r);
    }
  } while (l < r);
  // platziere das Pivot-Element an seine korrekte Position
  if (l < rechterIndex) {
    this.erlaeuterung(
      "Die Umordnung des Teil-Arrays ist abgeschlossen.\n" +
      "Nun muss ich nur noch den Pivot-Haufen (gelber Markierungshamster) " +
      "an seine\nkorrekte Position bei Index " +
      l +
      " (hellblauer Markierungshamster) stellen.");
    tauschen(l, rechterIndex);
    return l;
  } else {
    this.erlaeuterung(
      "Die Umordnung des Teil-Arrays ist abgeschlossen.\n" +
      "Auch der Pivot-Haufen steht bereits an seiner korrekten " +
      "Position bei Index " +
      rechterIndex +
      ".");
    return rechterIndex;
  }
}

// waehlt einen beliebigen Index zwischen den angegebenen Indizes
private int waehlePivotIndex(int linkerIndex, int rechterIndex) {
  // in diesem Fall einfach der mittleren Index
  return (linkerIndex + rechterIndex) / 2;
  /* Alternative 1:
    return rechterIndex;
  */
  /* Alternative 2 (mittleres von drei Elementen):
    int index1 = (linkerIndex + rechterIndex) / 2;
    int index2 = (linkerIndex + index1) / 2;
    int index3 = (index1 + rechterIndex) / 2;
    if (zahlen[index1] <= zahlen[index2] &&
        zahlen[index2] <= zahlen[index3])
      return zahlen[index2];
    if (zahlen[index2] <= zahlen[index3] &&
        zahlen[index3] <= zahlen[index1])
      return zahlen[index3];
    return zahlen[index1];
  */
}

// tauscht die Elemente des Arrays an den angegebenen Indizes
private void tauschen(int index1, int index2) {
  if (index1 != index2) {
    this.laufeZuIndex(index1);
    int koerner1 = this.nimmAlle();
    this.laufeZuIndex(index2);
    int koerner2 = nimmAlle();
```

```
      this.gib(koerner1);
      this.laufeZuIndex(index1);
      this.gib(koerner2);
    }
  }

  private void beendeSortierung() {
    this.laufeZuSpalte(this.startSpalte);
    this.setzeBlickrichtung(Hamster.OST);
    this.erlaeuterung("Sortierung erfolgreich beendet!");
  }
}
```

Mit dem folgendem objektorientierten Hamster-Programm (`SortierenMitQuickSort`) können Sie sich von den Hamstern den QuickSort-Algorithmus demonstrieren lassen.

```
void main() {
  Hamster paul = Hamster.getStandardHamster();
  String antwort =
    Hamster.getStandardHamster().liesZeichenkette(
      "Moechten Sie textuelle Erlaeuterungen (ja/nein)?");
  SortierHamster sortierer = new QuickSortHamster(antwort.equals("ja"));
  sortierer.sortiereKoernerHaufen();
}
```

2.6.3 Analyse des Algorithmus

Der QuickSort-Algorithmus ist nicht stabil. QuickSort arbeitet in-place.

Die Geschwindigkeit von QuickSort ist abhängig von der Gleichmäßigkeit der Partitionierung. Wenn die Wahl des Pivot-Elementes bewirkt, dass die beiden Teil-Arrays in jedem Rekursionsschritt in etwa gleich groß sind, benötigt QuickSort $\approx n\ log(n)$ Vergleiche. Im ungünstigsten Fall sind jedoch $\approx n^2$ Vergleiche notwendig, nämlich genau dann wenn ein Teil-Array stets nur aus einem Element und das andere aus den restlichen Elementen besteht. n ist dabei die Anzahl an zu sortierenden Elementen.

Im Web finden sich zahlreiche weitere Informationen zum QuickSort-Algorithmus, z.B. unter

- `http://de.wikipedia.org/wiki/Quicksort`
- `http://www.iti.fh-flensburg.de/lang/algorithmen/sortieren/quick/quick.htm`
- `http://www.sortieralgorithmen.de/quicksort/index.html`
- `http://olli.informatik.uni-oldenburg.de/fpsort/quicksort.html`

2.7 MergeSort: Sortieren durch Mischen

MergeSort ist wie QuickSort ein rekursiver Sortieralgorithmus.

2.7.1 Algorithmus

Die grundlegende Idee des MergeSort-Algorithmus ist die, dass das zu sortierende Array in zwei Teil-Arrays zerlegt wird und diese durch rekursive Anwendung des Algorithmus sortiert und anschließend gemischt werden. Die Rekursion endet, wenn ein Teil-Array nur noch aus einem Element besteht. In diesem Fall ist es ja sortiert.

Mischen zweier sortierter Teil-Arrays bedeutet, dass diese zu einem sortierten Array verschmolzen werden. Dazu werden die Teil-Arrays zunächst in ein Hilfs-Array kopiert. Anschließend werden die beiden Teil-Arrays elementweise durchlaufen. Das jeweils kleinere Element wird zurückkopiert. Zum Schluss muss dann noch der Rest eines der beiden Teil-Arrays zurückkopiert werden.

Die folgende Klasse `MergeSort` implementiert das Interface `SortierAlgorithmus` entsprechend des MergeSort-Algorithmus:

```
public class MergeSort implements SortierAlgorithmus {

  private int[] hilfsArray;

  // sortiert das uebergebene Array in aufsteigender Reihenfolge
  // gemaess dem MergeSort-Algorithmus
  public void sortiere(int[] zahlen) {
    hilfsArray = new int[zahlen.length];
    mergeSort(zahlen, 0, zahlen.length-1);
  }

  // der MergeSort-Algorithmus wird auf dem Array zwischen den
  // angegebenen Indizes ausgefuehrt
  private void mergeSort(int[] zahlen, int linkerIndex, int rechterIndex) {
    if (linkerIndex < rechterIndex) {
      int mittlererIndex = (linkerIndex + rechterIndex) / 2;
      mergeSort(zahlen, linkerIndex, mittlererIndex);
      mergeSort(zahlen, mittlererIndex+1, rechterIndex);
      mischen(zahlen, linkerIndex, mittlererIndex, rechterIndex);
    }
  }

  // mischt die zwei (sortierten) Teil-Arrays von linkerIndex bis
  // mittlererIndex und mittlererIndex+1 bis rechterIndex
  private void mischen(int[] zahlen, int linkerIndex, int mittlererIndex,
                       int rechterIndex) {

    // beide Teil-Arrays in das Hilfsarray kopieren
    for (int i=linkerIndex; i<=rechterIndex; i++) {
      hilfsArray[i] = zahlen[i];
```

```
    }

    // jeweils das kleinere Element der beiden Teil-Arrays zurückkopieren
    int linkerZeiger = linkerIndex;
    int rechterZeiger = mittlererIndex + 1;
    int aktuellerZeiger = linkerIndex;
    while (linkerZeiger <= mittlererIndex && rechterZeiger <= rechterIndex) {
      if (hilfsArray[linkerZeiger] <= hilfsArray[rechterZeiger]) {
        zahlen[aktuellerZeiger++] = hilfsArray[linkerZeiger++];
      } else {
        zahlen[aktuellerZeiger++] = hilfsArray[rechterZeiger++];
      }
    }

    // falls vorhanden Reste des ersten Teil-Arrays zurückkopieren
    while (linkerZeiger <= mittlererIndex) {
      zahlen[aktuellerZeiger++] = hilfsArray[linkerZeiger++];
    }

    // falls vorhanden Reste des zweiten Teil-Arrays zurückkopieren
    while (rechterZeiger <= rechterIndex) {
      zahlen[aktuellerZeiger++] = hilfsArray[rechterZeiger++];
    }

  }
}
```

2.7.2 Visualisierendes Hamster-Programm

Die folgende Hamster-Klasse `MergeSortHamster` implementiert das Interface `SortierHamster` und visualisiert dabei den MergeSort-Algorithmus. Voraussetzung ist, dass es im Territorium drei nicht durch Mauern blockierte Reihen unterhalb des Standard-Hamsters gibt. In der ersten Reihe unterhalb der Körnerhaufenreihe markieren zwei Markierungshamster jeweils den linken und rechten Rand des aktuell betrachteten Teil-Arrays. Darunter die Reihe wird als Ablagereihe benutzt. Hierhin werden vor dem Mischen die Körnerhaufen der beiden beteiligten Teil-Arrays transportiert. Unterhalb dieser Reihe markieren zwei weitere Markierungshamster beim Mischen das jeweils betrachtete Element der beiden Teil-Arrays. Ein fünfter Markierungshamster (weiß) markiert in der ersten Reihe unterhalb der Körnerhaufenreihe die Kachel, in die jeweils beim Mischen der kleinere der beiden betrachteten Körnerhaufen der Teil-Arrays transportiert werden muss.

```
// Demonstration des MergeSort-Algorithmus
public class MergeSortHamster
  extends KoernerHaufenSortierHamster
  implements SortierHamster
{

  private MarkierungsHamster linkerIndexHamster = null;
```

```
    // markiert den linken Rand des aktuellen Teil-Arrays

  private MarkierungsHamster rechterIndexHamster = null;
    // markiert den rechten Rand des aktuellen Teil-Arrays

  private MarkierungsHamster linkerZeigerHamster = null;
    // markiert waehrend des Mischens das aktuell
    // betrachtete Element des ersten Teil-Arrays

  private MarkierungsHamster rechterZeigerHamster = null;
    // markiert waehrend des Mischens das aktuell
    // betrachtete Element des rechten Teil-Arrays

  private MarkierungsHamster aktuellerZeigerHamster = null;
    // markiert waehrend der Partitionierung das aktuell
    // betrachtete rechte Element

  private int anzahlKoernerHaufen = 0;
    // Anzahl der zu sortierenden Koernerhaufen

  // Konstruktor
  public MergeSortHamster(boolean mitErlaeuterungen) {
    super(mitErlaeuterungen);
    this.erlaeuterung(
      "Ich sortiere die Koernerhaufen auf der Basis des MergeSort-Algorithmus.");
    this.linkerIndexHamster =
      new MarkierungsHamster(this.getReihe()+1, this.getSpalte(),
                             this.getReihe(), mitErlaeuterungen);
    this.linkerIndexHamster.erlaeuterung(
      "Ich markiere den linken Rand des aktuell betrachteten Teil-Arrays.");
    this.rechterIndexHamster =
      new MarkierungsHamster(this.getReihe()+1, this.getSpalte(),
                             this.getReihe(), mitErlaeuterungen);
    this.rechterIndexHamster.erlaeuterung(
      "Ich markiere den rechten Rand des aktuell betrachteten Teil-Arrays.");
    this.linkerZeigerHamster =
      new MarkierungsHamster(this.getReihe()+3, this.getSpalte(),
                             this.getReihe()+2, mitErlaeuterungen);
    this.linkerZeigerHamster.erlaeuterung(
      "Ich markiere waehrend des Mischens das aktuell betrachtete Element\n" +
      "des ersten Teils der Hilfskoernerhaufenreihe.");
    this.rechterZeigerHamster =
      new MarkierungsHamster(this.getReihe()+3, this.getSpalte(),
                             this.getReihe()+2, mitErlaeuterungen);
    this.rechterZeigerHamster.erlaeuterung(
      "Ich markiere waehrend des Mischens das aktuell betrachtete Element\n" +
      "des zweiten Teils der Hilfskoernerhaufenreihe.");
    this.aktuellerZeigerHamster =
      new MarkierungsHamster(this.getReihe()+1, this.getSpalte(),
                             this.getReihe(), mitErlaeuterungen);
    this.aktuellerZeigerHamster.erlaeuterung(
      "Ich markiere waehrend des Mischens das Element der Körnerhaufenreihe,\n" +
      "in das die Körner der Hilfskoernerhaufenreihe zurueckkopiert werden.");
```

```
  }

  // Der Standard-Hamster steht mit Blickrichtung OST irgendwo im Territorium.
  // Ein Vertretungshamster soll die Koernerhaufen bis zur nächsten Wand in
  // aufsteigender Reihenfolge sortieren.
  // Voraussetzung: Unterhalb des Standard-Hamsters existieren drei nicht
  // durch Mauern blockierte Reihen.
  public void sortiereKoernerHaufen() {
    this.erlaeuterung(
      "Ich zaehle nun die Anzahl der zu sortierenden Koernerhaufen.");
    this.anzahlKoernerHaufen = this.ermittleAnzahlKoernerHaufen();
    this.erlaeuterung(
      "Die Anzahl der zu sortierenden Koernerhaufen betraegt " +
      this.anzahlKoernerHaufen +
      ",\nd.h. die Indizes der Haufen liegen zwischen 0 und " +
      (this.anzahlKoernerHaufen-1) +
      ".");
    this.mergeSort(0, this.anzahlKoernerHaufen-1);
    this.beendeSortierung();
  }

  // der MergeSort-Algorithmus wird auf dem Array zwischen den
  // angegebenen Indizes ausgefuehrt
  private void mergeSort(int linkerIndex, int rechterIndex) {
    if (linkerIndex < rechterIndex) {
      int mittlererIndex = (linkerIndex + rechterIndex) / 2;
      mergeSort(linkerIndex, mittlererIndex);
      this.erlaeuterung(
        "Die Koernerhaufen von Index " +
        linkerIndex + " bis " + mittlererIndex +
        " sind (nun) sortiert.");
      mergeSort(mittlererIndex+1, rechterIndex);
      this.erlaeuterung(
        "Die Koernerhaufen von Index " +
        (mittlererIndex+1) + " bis " + rechterIndex +
        " sind (nun) sortiert.");
      mischen(linkerIndex, mittlererIndex, rechterIndex);
    } else {
      this.linkerIndexHamster.markiereIndex(linkerIndex);
      this.rechterIndexHamster.markiereIndex(rechterIndex);
    }
  }

  // mischt die zwei (sortierten) Teil-Arrays von linkerIndex bis
  // mittlererIndex und mittlererIndex+1 bis rechterIndex
  private void mischen(int linkerIndex, int mittlererIndex,
                       int rechterIndex) {
    this.linkerIndexHamster.markiereIndex(linkerIndex);
    this.rechterIndexHamster.markiereIndex(rechterIndex);
    this.erlaeuterung(
      "Es folgt nun das Mischen der Koernerhaufen von Index " +
      linkerIndex + " bis " + mittlererIndex +
      " und " +
```

```
    (mittlererIndex+1) + " bis " + rechterIndex +
    ".\n" +
    "Zunaechst werden dazu die Koernerhaufen in eine Hilfsreihe verschoben.");

  // beide Teil-Arrays in das Hilfsarray (zwei Zeilen darunter) kopieren
  int linkerZeiger = linkerIndex;
  this.linkerZeigerHamster.markiereIndex(linkerZeiger);
  int rechterZeiger = mittlererIndex + 1;
  this.rechterZeigerHamster.markiereIndex(rechterZeiger);

  for (int i=linkerIndex; i<=rechterIndex; i++) {
    this.laufeZuIndex(i);
    int koerner = this.nimmAlle();
    this.setzeBlickrichtung(Hamster.SUED);
    this.vor(2);
    this.gib(koerner);
    this.setzeBlickrichtung(Hamster.NORD);
    this.vor(2);
  }

  // jeweils das kleinere Element der beiden Teil-Arrays zurückkopieren
  int aktuellerZeiger = linkerIndex;
  this.aktuellerZeigerHamster.markiereIndex(aktuellerZeiger);
  this.erlaeuterung(
    "Beim Mischen wird der kleinere der beiden markierten " +
    "Koernerhaufen der Hilfsreihe an\ndie markierte Kachel der " +
    "eigentlichen Reihe (weißer Markierungshamster) zurueckkopiert.");

  while (linkerZeiger <= mittlererIndex && rechterZeiger <= rechterIndex) {
    if (linkerZeigerHamster.liefereAnzahlKoerner() <=
        rechterZeigerHamster.liefereAnzahlKoerner()) {
      transportiereKoerner(linkerZeigerHamster.liefereIndex());
      linkerZeiger++;
      this.linkerZeigerHamster.markiereIndex(linkerZeiger);
    } else {
      transportiereKoerner(rechterZeigerHamster.liefereIndex());
      rechterZeiger++;
      this.rechterZeigerHamster.markiereIndex(rechterZeiger);
    }
    aktuellerZeiger++;
    if (aktuellerZeiger <= rechterIndex) {
      this.aktuellerZeigerHamster.markiereIndex(aktuellerZeiger);
    }
  }

  // falls vorhanden Reste des ersten Teil-Arrays zurückkopieren
  while (linkerZeiger <= mittlererIndex) {
    transportiereKoerner(linkerZeigerHamster.liefereIndex());
    linkerZeiger++;
    this.linkerZeigerHamster.markiereIndex(linkerZeiger);
    aktuellerZeiger++;
    if (aktuellerZeiger <= rechterIndex) {
      this.aktuellerZeigerHamster.markiereIndex(aktuellerZeiger);
```

```
      }
    }

    // falls vorhanden Reste des zweiten Teil-Arrays zurückkopieren
    while (rechterZeiger <= rechterIndex) {
      transportiereKoerner(rechterZeigerHamster.liefereIndex());
      rechterZeiger++;
      this.rechterZeigerHamster.markiereIndex(rechterZeiger);
      aktuellerZeiger++;
      if (aktuellerZeiger <= rechterIndex) {
        this.aktuellerZeigerHamster.markiereIndex(aktuellerZeiger);
      }
    }
  }

  // transportiert die Koerner aus
  private void transportiereKoerner(int index) {
    this.laufeZuIndex(index);
    this.setzeBlickrichtung(Hamster.SUED);
    this.vor(2);
    int koerner = this.nimmAlle();
    this.setzeBlickrichtung(Hamster.NORD);
    this.vor(2);
    this.laufeZuIndex(aktuellerZeigerHamster.liefereIndex());
    this.gib(koerner);
  }

  private void beendeSortierung() {
    this.laufeZuSpalte(this.startSpalte);
    this.setzeBlickrichtung(Hamster.OST);
    this.erlaeuterung("Sortierung erfolgreich beendet!");
  }
}
```

Mit dem folgendem objektorientierten Hamster-Programm (`SortierenMitMerge-Sort`) können Sie sich von den Hamstern den MergeSort-Algorithmus demonstrieren lassen.

```
void main() {
  Hamster paul = Hamster.getStandardHamster();
  String antwort =
    Hamster.getStandardHamster().liesZeichenkette(
      "Moechten Sie textuelle Erlaeuterungen (ja/nein)?");
  SortierHamster sortierer = new MergeSortHamster(antwort.equals("ja"));
  sortierer.sortiereKoernerHaufen();
}
```

2.7.3 Analyse des Algorithmus

Der MergeSort-Algorithmus ist stabil. MergeSort arbeitet nicht in-place.

MergeSort erfordert unabhängig vom Aufbau des zu sortierenden Arrays, d.h. auch im unglücklichsten Fall, $\approx n\ log(n)$ Vergleiche der Array-Elemente. n ist dabei die Anzahl an zu sortierenden Elementen. Allerdings ist zusätzlicher zu n proportionaler Speicherplatz erforderlich.

Im Web finden sich zahlreiche weitere Informationen zum MergeSort-Algorithmus, z.B. unter

- `http://de.wikipedia.org/wiki/Mergesort`
- `http://www.iti.fh-flensburg.de/lang/algorithmen/sortieren/merge/merge.htm`
- `http://www.sortieralgorithmen.de/mergesort/index.html`

2.8 Zusammenfassung und Anmerkungen

In diesem Kapitel wurden die sechs bekanntesten Sortieralgorithmen vorgestellt und analysiert und mit Hilfe von objektorientierten Hamster-Programmen visualisiert.

Für genügend große Arrays ist QuickSort der im Mittel schnellste Sortieralgorithmus. Im unglücklichsten Fall kann jedoch MergeSort schneller sein. Für den Fall, dass das Array schon gut vorsortiert ist, sind InsertionSort und BubbleSort schnellere Alternativen.

ShellSort und QuickSort sind nicht stabil, die anderen vier Algorithmen sind stabil. MergeSort ist der einzige der vorgestellten Algorithmen, der nicht in-place ist.

Zahlreiche weitere Informationen zu Sortieralgorithmen finden Sie im Web, z.B. unter

- `http://de.wikipedia.org/wiki/Sortieralgorithmus`
- `http://www.iti.fh-flensburg.de/lang/algorithmen/sortieren/sortalgo.htm`
- `http://www.sortieralgorithmen.de`

2.8.1 Sortieren von Objekten

In den obigen Abschnitten werden die Sortieralgorithmen vorgestellt, indem Arrays mit `int`-Werten sortiert werden. Sie funktionieren jedoch auch zum Sortieren von Objekten beliebiger Klassen, wenn ein Schlüssel festgelegt und eine totale Ordnung auf dem Schlüssel definiert wird.

Zu diesem Zweck stellt Java bereits ein vordefiniertes Interface zur Verfügung, das Interface `Comparable`. Das Interface hat folgende Gestalt:

```
public interface Comparable {
  public int compareTo(Object o);
}
```

Wenn Sie Objekte einer bestimmten Klasse sortieren möchten, lassen Sie diese Klasse das Interface `Comparable` implementieren. Die Methode `compareTo` sollte dabei folgendermaßen implementiert werden:

- Sie liefert einen negativen Wert, wenn der Schlüssel des Objektes, für das die Methode aufgerufen wird, kleiner ist als der Schlüssel des als Parameter übergebenen Objektes.
- Sie liefert den Wert 0, wenn der Schlüssel des Objektes, für das die Methode aufgerufen wird, gleich dem Schlüssel des als Parameter übergebenen Objektes ist.
- Sie liefert einen positiven Wert, wenn der Schlüssel des Objektes, für das die Methode aufgerufen wird, größer ist als der Schlüssel des als Parameter übergebenen Objektes.

Bspw. könnte eine Klasse `Person`, bei der Personen nach ihrem Alter sortiert werden sollen, folgendermaßen definiert werden:

```
public class Person implements Comparable {
  String name;
  int alter;
  ...
  public int compareTo(Object o) {
    Person anderePerson = (Person)o;
    return this.alter - anderePerson.alter;
  }
}
```

Bei den Sortieralgorithmen ist folgendes anzupassen. Zunächst wird das Interface `SortierAlgorithmus` geändert. Sortiert werden sollen nicht mehr Arrays mit `int`-Werten, sondern Arrays mit `Comparable`-Objekten:

```
public interface SortierAlgorithmus {
  // sortiert das Array in aufsteigender Reihenfolge
  public void sortiere(Comparable[] array);
}
```

Weiterhin müssen die Vergleiche der Array-Elemente ($<$, $<=$, $>$, $>=$, $==$ und $!=$) innerhalb der einzelnen Algorithmen ersetzt werden durch entsprechende Aufrufe der `compareTo`-Methode. Dies sei am SelectionSort-Algorithmus beispielhaft illustiert:

```
public class SelectionSort implements SortierAlgorithmus {

  public void selection(Comparable[] array) {
    for (int aktIndex = 0; aktIndex < array.length - 1; aktIndex++) {
      int minIndex = aktIndex;
      for (int suchIndex = aktIndex+1;
           suchIndex < array.length;
           suchIndex++) {
        if (array[suchIndex].compareTo(array[minIndex]) < 0) {
          minIndex = suchIndex;
        }
      }
      Comparable speicher = array[aktIndex];
      array[aktIndex] = array[minIndex];
      array[minIndex] = speicher;
    }
  }
}
```

2.8.2 Sortieren von Zeichenketten

Zeichenketten sind in Java Objekte der vordefinierten Klasse `String`. Diese implementiert (glücklicherweise) das im vorherigen Abschnitt vorgestellte Interface `Comparable`, stellt also die Methode `compareTo` zur Verfügung. Verglichen werden dabei paarweise die entsprechenden Zeichen, und zwar gemäß der lexikographischen Ordnung[3].

[3] Grundlage sind die Unicode-Werte der Zeichen.

Literaturverzeichnis

[BB04] Boles, Dietrich und Cornelia Boles: *Objektorientierte Programmierung spielend gelernt mit dem Java-Hamster-Modell.* Teubner, 2004.

[Bol02] Boles, Dietrich: *Programmieren spielend gelernt mit dem Java-Hamster-Modell.* Teubner, 2002.

[Lan02] Lang, Hans W.: *Algorithmen in Java.* Oldenbourg, 2002.

[Sed03] Sedgewick, Robert: *Algorithmen in Java.* Pearson-Studium, 2003.

[SG02] Solymosi, Andreas und Ulrich Grude: *Grundkurs Algorithmen und Datenstrukturen in Java.* Vieweg, 2002.

[SS04] Saake, Gunter und Kai-Uwe Sattler: *Algorithmen und Datenstrukturen: Eine Einführung mit Java.* dpunkt, 2004.