Viele Schränke bilden eine Schrankwand

Das einfache Projekt Schrankwand könnte so erweitert werden, dass die Schrankwand eine „beliebige“ Zahl von Schränken enthalten kann. In diesem Fall brauchen wir statt der Objekte schrank1, schrank2 und schrank3 eine Variable [gesucht ist also ein Datentyp] mit der wir alle Schränke gemeinsam verwalten können, also z.B. eine Liste von Schränken oder ein Array von Schränken. Wir werden uns mehrere Fälle ansehen, zunächst das Beispiel Array.

Lösung mit dem Array

Unsere Klasse Schrankwand benötigt nun eine Objektvariable zum Speichern der Schränke (wir nennen sie schraenke) und eine integer – Variable anzahl, um die Zahl der Schränke zu speichern. Wir nehmen die Deklaration in den Kopf der Klasse auf:

```
private Schrank[] schraenke;
private int anzahl;
```

Die nachgestellte eckige Klammer deklariert schraenke als ein Array von Exemplaren der Klasse Schrank. Der Konstruktor sollte nun so verändert werden, dass Schrankwände mit verschiedenen Anzahlen erzeugt werden können. Dazu benötigt er einen Parameter für die Anzahl.
Außerdem muss das Array selbst initialisiert werden, anschließend sind noch die Schrankobjekte in gewünschter Anzahl und Position zu erzeugen. Die Änderungen im Konstruktor sind also:

```
  public Schrankwand(int anzahl)
{
  xPosition = 40;
  yPosition = 80;
  farbe = "blau";
  orientierung = 0;
  istSichtbar = false;
  int schrankbreite = 60;
  this.anzahl = anzahl;
  breite = schrankbreite*anzahl;
  tiefe  = 37;
  schraenke = new Schrank[anzahl];
  for (int i=0;i<anzahl;i++){
      schraenke[i] =
         new Schrank(i*schrankbreite, 0,
                     farbe, orientierung,
                     schrankbreite, tiefe);
  }
}
```

Parameter in Variable übernehmen

Breite der Schrankwand berechnen

Schrankobjekte erzeugen

Schrankposition berechnen

Beachten Sie bitte, dass im Schleifenkopf die richtigen Werte für die Variable i verwendet werden. i wird zunächst auf der Startwert 0 gesetzt, da die erste Position im Array den Index 0 hat, die Zählvorschrift i++ zählt jeweils um 1 weiter und die Laufbedingung, mit der man den Abbruch der Schleife festlegt, muss i< anzahl lauten, da der höchste zulässige Index den Wert anzahl-1 hat. Die zulässigen Indizes gehen also von 0..anzahl-1.
Die Änderungen in **`gibAktuelleFigur()`** sind auch sehr einfacher Art. Hier muss das Einfügen in den GeneralPath über eine Schleife gesteuert werden.

```
protected Shape gibAktuelleFigur()
{
    GeneralPath schrankwand = new GeneralPath();
    for (int i=0;i<anzahl;i++){
      schrankwand.append(schraenke[i].gibAktuelleFigur(), false);
    }
```

Zugriff auf das Array über den Index

```
      return  transformiere(schrankwand);}
```

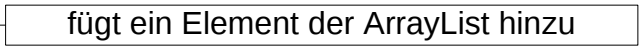
Lösung mit ArrayList

Eine ArrayList ist eine Klasse, die List implementiert (s.u.) und uns von JAVA im Paket **`java.util`** bereitgestellt wird.
Wir müssen sie zunächst importieren.

```
import java.util.ArrayList;
```

Im Kopf der Klasse heißt es nun:

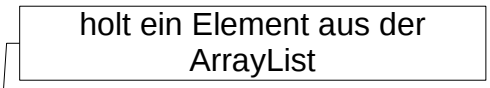
```
  private ArrayList schraenke;
  private int anzahl;
```



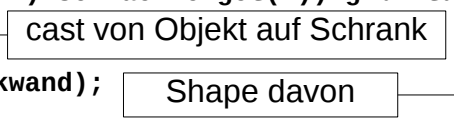
Im Konstruktor ändern sich die erzeugenden Zeilen:

```
    schraenke = new ArrayList();
    for (int i=0;i<anzahl;i++){
        schraenke.add(
                new Schrank(i*schrankbreite, 0,
                            farbe, orientierung,
                            schrankbreite, tiefe));
    }
```

und bei gibAktuelleFigur ändert sich die Zugriffsart[1]:




```
  protected Shape gibAktuelleFigur()
  {
      GeneralPath schrankwand = new GeneralPath();
      for (int i=0;i<anzahl;i++){
        schrankwand.append(((Schrank) schraenke.get(i)).gibAktuelleFigur(),
false);
      }
      return  transformiere(schrankwand);
  }
```



Neu mit "for each" und typisiert (s.u.):

```
      for (Schrank schrank : schraenke) {
        schrankwand.append(schrank.gibAktuelleFigur(), false);
      ... }
```

Die Sache mit dem cast

Die Sache mit dem cast ist zwar zum Verständnis der Typbindung von JAVA und dem statischen und dynamischen Typ wirklich interessant, in vielen Anwendungen aber so ärgerlich, dass die Entwickler inzwischen die Sammlungsklassen typisiert haben.
Deklariert und definiert man

```
   ArrayList<Schrank> schraenke = new ArrayList<Schrank>();
```

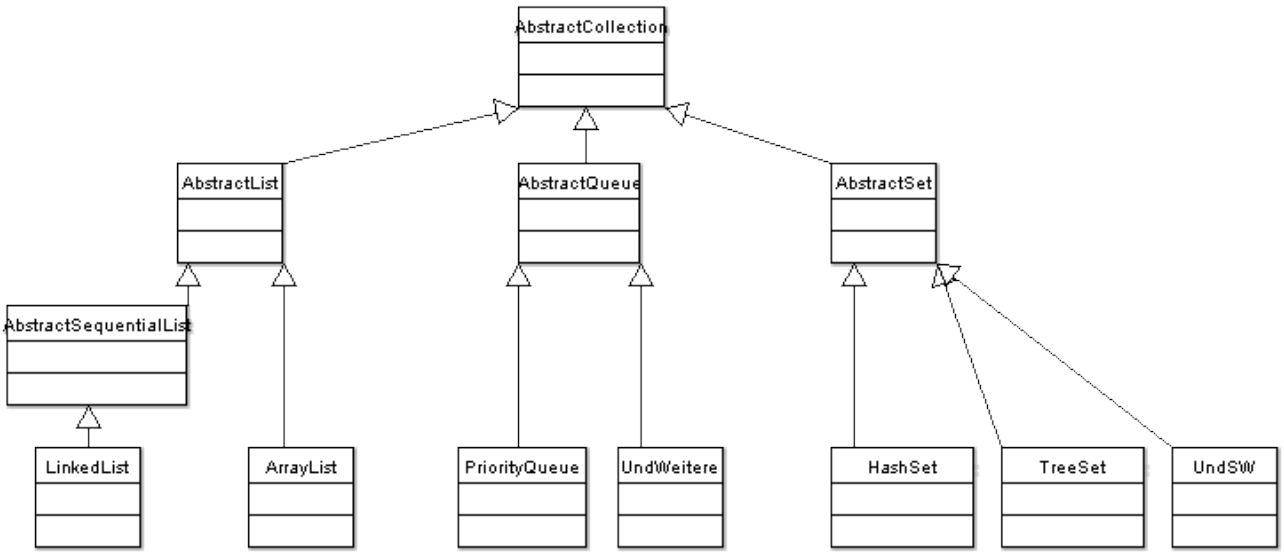
dann kann man sich den cast sparen, da die Elemente der ArrayList dann nicht vom Typ Object sind, sondern Instanzen der Klasse Schrank.

---

1 In einem hohen Maße unbefriedigend ist die hier noch verwendete – und so mögliche – Steuerung der Schleife über einen Index. Die bessere Lösung ist das Verwenden eines Iterators.
Wie der verwendet wird und dass dahinter ein Entwurfsmuster der Objektorientierung steckt, wird noch untersucht.

Tradition und Moderne ?

Das Bemerkenswerte an Arrays selbst ist, dass man viele Speicherplätze unter einem Namen direkt ansprechen kann. Welchen Wert man speziell ansprechen wollte, steuert man über den Index und die Syntax ist **`arrayname[index]`**.
Arrays stellen einen sehr traditionellen Datentyp der Informatik dar. Das bedeutet, dass viele Programmentwickler „daran gewöhnt“ sind, viel schlimmer: entsprechend bei ihrem Entwurf denken!
Betrachtet man die Geschichte von Arrays am Beispiel TURBO – PASCAL (heute Delphi), dann findet man eine Begründung für die Bedeutung dieses Typs in der Art der Speicherung von Daten bei TURBO – PASCAL. Einfache, weil berechenbare und damit schnelle Datenzugriffe waren nur in einem 64 kByte großen Speicherbereich möglich, in dem sich daher die Bereiche für „statische Arrays“ befanden. Statisch deswegen, weil ihre Größe schon zur Zeit der Programmübersetzung (*compile* – TP ist eine Compilersprache) bekannt sein musste. Man kann dann leicht aus dem Index und dem Speicherbedarf für ein Element die exakte Position jedes Elementes im Speicher direkt berechnen.
Die von JAVA bereit gestellten Sammlungstypen stellen modernere Konstrukte dar. Auch bei ihnen nutzt man das Erkennen von immer wieder auftretenden Anforderungen und das Ausformulieren dieser Anforderungen in einer bzw. mehreren Klassen.
Stellt man das Klassendiagramm zur Klasse AbstractCollection dar ...



... dann fehlen viele der Klassen und wenn man bei der Schnittstelle Collection nachsieht, ...

> Interface Collection
> All Known Subinterfaces:
> BeanContext, BeanContextServices, BlockingQueue<E>, List<E>, Queue<E>, Set<E>, SortedSet<E>
> All Known Implementing Classes:
> AbstractCollection, AbstractList, AbstractQueue, AbstractSequentialList, AbstractSet, ArrayBlockingQueue, ArrayList, AttributeList, BeanContextServicesSupport, BeanContextSupport, ConcurrentLinkedQueue, CopyOnWriteArrayList, CopyOnWriteArraySet, DelayQueue, EnumSet, HashSet, JobStateReasons, LinkedBlockingQueue, LinkedHashSet, LinkedList, PriorityBlockingQueue, PriorityQueue, RoleList, RoleUnresolvedList, Stack, SynchronousQueue, TreeSet, Vector

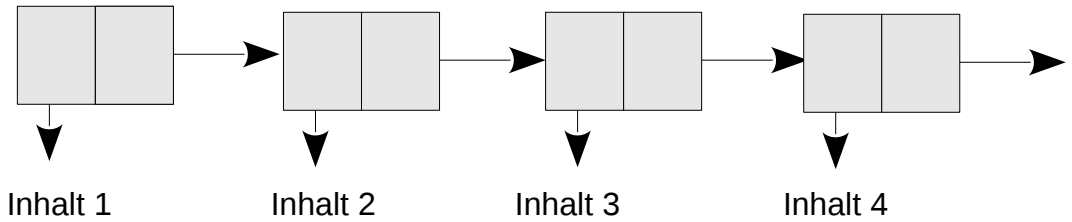
... stellt man fest, dass es für ein Diagramm viel zu viele implementierende Klassen gibt.

ArrayList – was ist das bei JAVA ?

In der javadoc zu ArrayList heißt es:
Die ArrayList ist eine Implementation des List – interface, bei dem sich die Größe ändern kann – also im Gegensatz zu statischen Arrays. Zusätzlich zu den normalen Anforderungen, die alle Klassen erfüllen müssen, die das List – interface implementieren, stellt diese Klasse Methoden zur Veränderung der benötigten Größe im Speicher bereit. In der javadoc wird auf den Zeitaufwand der Methoden eingegangen. Dabei ist von besonderer Bedeutung, dass das Anfügen von Elementen mit der Methode **`add`** mit konstantem Zeitbedarf erfolgt, worin der wesentliche Unterschied zu einer LinkedList besteht.
Zu einer ArrayList wird die Speicherkapazität (capacity) verwaltet. Deren Verwaltung erfolgt automatisch und es ist für uns – im Sinne der Kapselung – nicht bekannt, wie das erfolgt.

Und die LinkedList ?

Warum ist der Zeitbedarf bei einer LinkedList höher? Dazu sollte man sich eine grafische Darstellung einer verketteten Liste ansehen (siehe auch Aufgabe unten):



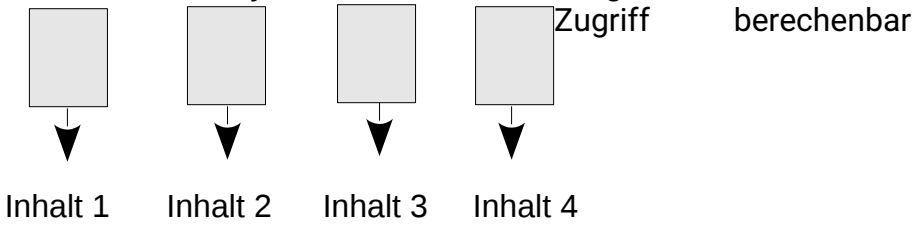
Man greift auf die Daten sequentiell zu. Beginnend beim Kopf kann man zunächst auf das erste Element zugreifen, es enthält neben den gespeicherten Inhalten eine Information, wo das nächste Element der Liste zu finden ist. So geht es weiter bis zum Ende der Liste, bei dem **`next`** nicht existiert, also z.B. auf null verwiesen wird.
Will man nun zum zwanzigsten Element, muss man auch auf alle 19 vorher zugreifen.

Aufgabe:

Berechnen Sie die Anzahl der Schritte, die nötig sind, um eine zunächst leere Liste mit 10 Elementen, 20 Elementen, 100 Elementen usw. zu füllen !

Zugriff bei Arrays

Wenn wir wirklich ein Array haben, wird hier der Zugriff direkt über den Index erfolgen.



Aufgabe:

1. Welche Schlüsse über die Art der Realisierung von LinkedList können Sie daraus ziehen, dass diese die Methoden **`addFirst`** (...), **`addLast`**(...), **`getFirst`**() und **`getLast`**() anbietet.

2. Korrigieren Sie das o.a. Bild !

Eine Lösung mit LinkedList ?

Sie sieht kaum anders aus. Insbesondere verändert sich nicht die Art der in der Schnittstelle angebotenen Zugriffe. Wenn das so ist, sollte man von der konkreten Lösung abstrahieren und das tut die Schnittstelle List! Deswegen:

Lösung mit List

Bei der Deklaration von schraenke wird nur List vorgegeben.

```
private List schraenke;          // *****Kern der Änderung*****
```

Die konkrete Implementation bleibt dann dem Konstruktor überlassen. Nur hier und beim import muss man dann bei einer Änderung der konkret verwendeten Klasse etwas ändern.

LinkedList

Die Änderungen gegenüber der ArrayList-Version sind nicht sehr groß.
Natürlich muss der Import für den Datentyp in **`import java.util.LinkedList`** geändert werden.

```
schraenke = new LinkedList();
for (int i=0;i<anzahl;i++){
    schraenke.add(new Schrank(i*schrankbreite,... ));
```

Der Rest bleibt dann – wie oben schon beschrieben – genauso wie bei der ArrayList.
Wie unwichtig diese Änderung für die Problemlösung ist, merken Sie daran, dass man eigentlich gar nicht wissen muss, was eigentlich **`ArrayList`** und **`LinkedList`** sind. Diese Problemstellung wird erst bei Anwendungen interessant, bei denen sehr oft auf große Datenbestände zugegriffen werden muss.