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;
```

fügt ein Element der ArrayList hinzu

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));
    }
```

holt ein Element aus der ArrayList

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);
  }
```

cast von Objekt auf Schrank

Shape davon

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 bei SUN inzwischen die Sammlungsklassen typisiert haben. Deklariert und definiert man

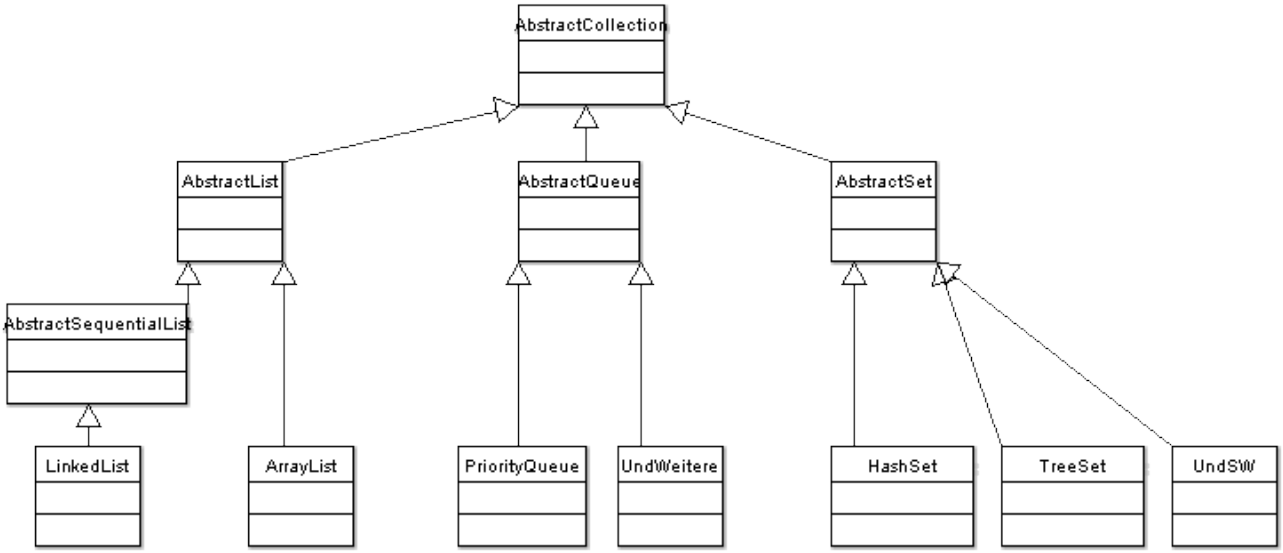
```
ArrayList<Schrank> schraenke = new ArrayList<Schrank>();
```

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

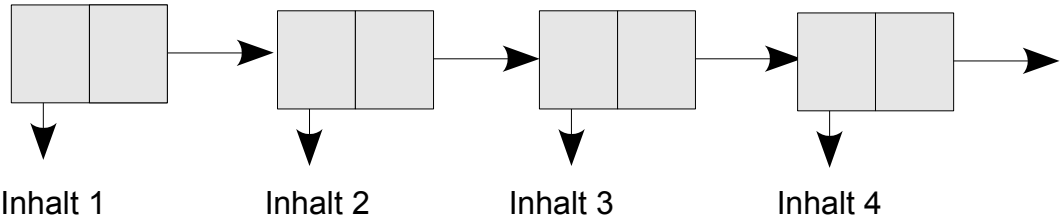
... stellt man fest, dass es für ein Diagramm viel zu viele implementierende Klassen gibt.

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):



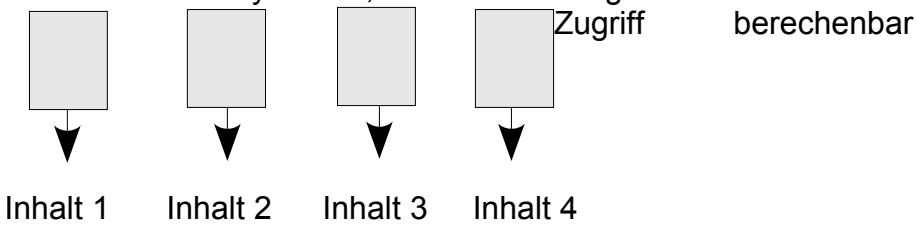
Man greift auf die Daten sequentiell zu. Beginnend beim Kopf kann man zunächst auf das erste Element zugreifen, es enthält neben den gespeicherten Inhalten eine Information, wo das nächste Element der Liste zu finden ist. So geht es weiter bis zum Ende der Liste, bei dem **`next`** nicht existiert, also 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.

Interessant wird's ...

... erst, wenn man sich mit dem Iterator beschäftigt, da dann nicht nur ein Entwurfsmuster der OO betrachtet werden kann, sondern außerdem auch eine weitere Reduzierung der Unterschiede für weitere Datentypen möglich wird.
Wir nutzen dann nicht mehr den bei der LinkedList beschriebenen Zugriff über die Indizes, sondern ***benutzen*** den Iterator, der von den Sammlungsklassen bereitgestellt wird. Der Programmtext ändert sich dann bei ArrayList folgendermaßen:
notwendiger Import:

```
import java.util.Iterator;
```

Schleife:

```
for (Iterator it= schraenke.iterator(); it.hasNext();)
  schrankwand.append(((Schrank) it.next()).gibAktuelleFigur(), false);
```

Im Schleifenkopf bedeuten die erkennbaren Änderungen

- „Schleifenvariable“ ist hier der von der Klasse ArrayList bereitgestellte Iterator, den wir durch die Methode `iterator()` bekommen
- Laufbedingung ist der Wert der Methode `hasNext()`
- Der Iterationsschritt wird durch den Iterator selbst gesteuert; wie er das macht, bleibt ihm selbst überlassen

Zugriff auf die Elemente

- Wir greifen mit der Methode `next()` auf die Elemente des Iterators zu
- Der cast durch `(Schrank)` ist notwendig, da der Iterator nur `Object` zurückliefern kann, es sei denn man verwendet einen typisierten Iterator.

typisierter Iterator

In der Variante mit einem typisierten Iterator) lautet die Schleifendefinition:

```
for (Iterator<Schrank> it= schraenke.iterator(); it.hasNext();)
  schrankwand.append(it.next().gibAktuelleFigur(), false);
```

Nun sind die Änderungen beim Verwenden anderer Klassen zur Speicherung der Schrankobjekte gering. Probieren Sie außer `ArrayList` und `LinkedList` auch `Vector`, `HashSet` und `TreeSet`. Beispiel HashSet: Bei einem Wechsel auf diesen völlig anderen Sammlungstyp wird in der Klasse nur beim `import` und beim Erzeugen der Klasse `HashSet` geändert. Alles andere bleibt!
Die Vielfalt der möglichen Klassen bleibt verwirrend, wenn man sich nicht ausführlich mit ihren Unterschieden beschäftigt. Insbesondere der Unterschied zwischen Vector und ArrayList ist gering und hat damit zu tun, dass beide eine unterschiedliche Entwicklungsstufe des JAVA – Systems darstellen. Vector ist eine ältere Variante, die für JAVA 2 neu implementiert wurde. Der bemerkenswerte Unterschied, dass Vector „synchronized“ ist, macht sich bei den Zugriffen von mehreren threads[1] auf dasselbe Sammlungsobjekt bemerkbar und wird eben nur dann relevant, wenn man mit threads arbeitet.

1 threads sind nebenläufige Prozesse. Bei einem laufenden JAVA – Programm heißt das, dass mehrere Anwendungskomponenten dadurch quasi gleichzeitig aktiv sind, dass sie sich die Prozessorzeit teilen. Gleichzeitige Zugriffe auf dieselben Daten können dabei zu Inkonsistenzen führen.

Eine selbst geschriebene Klasse Warteschlange

Idee und Vorlage: Guido Krüger in GoTo Java 2[1]

Zunächst definieren wir, was die Eigenschaften einer Warteschlange sein sollen.

- Man kann sich hinten anstellen.
- Wenn man vorn in der Warteschlange steht, kommt man als nächster dran.

Über diese zwingenden Eigenschaften hinaus, bauen wir noch ein:

- Man kann die Länge der Warteschlange erfahren.
- Man kann feststellen, ob die Warteschlange leer ist.
- Man kann die Warteschlange in einem Schritt komplett leeren.
- Die Warteschlange hat einen Iterator

Damit definieren wir eine Schnittstelle, das Interface der Queue-Collection[2].

```
import java.util.Iterator;
public interface Queue
{
  // Gibt true für eine leere queue zurück.
  public boolean isEmpty();

  // Fügt das Element obj am Ende der Queue an.
  public boolean add(Object obj);

  /**
   * Liefert das erste Element der Queue und entfernt es
   * daraus. Falls die Queue leer ist, erhält man null.
   */
  public Object retrieve();

  // Liefert die Anzahl der Elemente der Queue.
  public int size();

  // Entfernt alle Elemente aus der Queue.
  public void clear();

  // Liefert einen Iterator über alle Elemente der Queue.
  public Iterator iterator();
}
```

Schnittstellenkopf wie bei Klassen

Die Klasse Warteschlange implementiert nun diese Schnittstelle, also lautet der Kopf

```
public class Warteschlange implements Queue
```

Für das Realisieren der Verkettungsstruktur bauen wir uns eine einfache Klasse ListenElement. Da sie extern nicht benötigt wird[3], schreiben wir eine ***innere Klasse*** (Fachausdruck, auch lokale Klasse). Sie kann das Sichtbarkeitsattribut private haben.

```
    private class ListenElement{
        public Object inhalt = null;            // Zeiger auf den Inhalt
        public ListenElement nachfolger = null; // Zeiger auf den Nachfolger
    }
```

1 In diesem Projekt arbeiten wir noch ohne Exceptions. In einem weiteren Schritt sollten diese eingebaut und besprochen werden.

2 Vergleichen Sie mit dem Collection – interface!

3 Über die hier beschriebene Eigenschaft hinaus bieten innere Klassen eine wichtige weitere Eigenschaft gegenüber normalen Klassen: Da sie innerhalb der Umgebung ihrer äußeren Klasse definiert werden, gehören sie zu dieser Umgebung und haben Zugriff auf alle Eigenschaften der äußeren Klasse. Auf die sich daraus ergebenden Möglichkeiten wird hier aber nicht eingegangen.

An Instanzvariablen benötigen wir:

```
protected ListenElement anfang;
protected ListenElement ende;
protected int anzahl;
```

Den Zeiger auf den Anfang benötigen wir trivialerweise und zwingend, den auf das Ende nur, um das Anhängen einfach zu realisieren. Auch das Attribut anzahl ist nicht zwingend, es erleichtert aber die Bestimmung von size.
Beachten Sie, dass die Elemente in der queue nicht die Inhalte sind, sondern eben Listenelemente.
Der Konstruktor initialisiert nur diese Werte standardmäßig (könnte man sich also eigentlich sparen).

```
public Warteschlange()
{
    anfang = null;                // die Warteschlange ist am Anfang leer.
    ende = null;
    anzahl = 0;
}
```

Die folgenden Methoden sind sehr einfach:

```
public boolean isEmpty(){ return anzahl == 0; }
public int size() { return anzahl; }
```

Betrachten wir die add – Methode etwas genauer, dann stellt man sehr bald fest, dass eine Fallunterscheidung notwendig ist:

```
public boolean add(Object obj)
{
  if (isEmpty()) {                  // Die Schlange ist leer
    anfang = new ListenElement(); // Ein neues Listenelement wird angelegt
    ende = anfang;  // Das erste Element ist gleichzeitig auch das letzte
  } else {                                // anderenfalls:
    ende.nachfolger = new ListenElement(); // Ein neues Element wird erzeugt,
    ende = ende.nachfolger;                // und angehängt.
  }
  ende.inhalt = obj;                 // Inhalt wird zugewiesen
  ende.nachfolger = null;           // Als Ende kennzeichnen
  anzahl++;                         // Anzahl wird angepasst.
  return true;
}
```

hat es geklappt?

Und das Entfernen:

```
public Object retrieve() {
    if (isEmpty()) {
        return null;  // Hier liegt ein fehlerhafter Zugriff vor
    }
// Da der Anfang verändert wird, muss der Inhalt bis zum return gesichert werden
    Object dasObjekt = anfang.inhalt;
    anfang = anfang.nachfolger;     // Die Schlange rückt auf
    anzahl--;                       // Länge korrigieren
    if (isEmpty()) {                // jetzt leer?
        ende = anfang;
    }
    return dasObjekt;
}
```

Nun geht das Leeren sehr einfach:

```
    public void clear() { while (!isEmpty()) retrieve(); }
```

Interessant wird das Erstellen des Iterators. Hier könnte man sinnvollerweise auch eine anonyme Klasse verwenden, darauf wird hier aber verzichtet. Bemerkenswert ist insbesondere der Zugriff auf die Variablen und die andere innere Klasse der äußeren Klassendefinition!

```
    public Iterator iterator(){ return new WarteschlangenIterator(); }

    /**
     * interne Klasse WarteschlangenIterator
     */
    private class WarteschlangenIterator
    implements Iterator
    {
        private ListenElement aktuell;
        private WarteschlangenIterator(){
            aktuell=kopf;
        }
        public boolean hasNext() {
            return (aktuell!=null) ;
        }
        public Object next()
        throws NoSuchElementException
        {
            if (aktuell==null) throw new NoSuchElementException();
            else {
                Object temp=aktuell.inhalt;
                aktuell=aktuell.nachfolger;
                return temp;
            }
        }
        public void remove() {}
    }
```

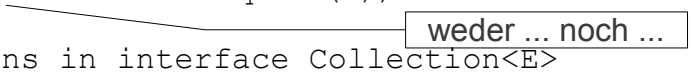
Aufgabe:

Schreiben Sie eine Methode contains(obj), die der folgenden Methode der Klassendefinition von LinkedList aus der Javadoc entspricht:

contains

public boolean contains(Object obj)

Returns true if this list contains the specified element. More formally, returns true if and only if this list contains at least one element e such that (obj==null ? e==null : o.equals(e)).



Specified by: contains in interface Collection<E>
Specified by: contains in interface List<E>
Overrides: contains in class AbstractCollection<E>
Parameters:
o - element whose presence in this list is to be tested.
Returns:
true if this list contains the specified element.
(und ich ergänze:) **false** if it does not.

Warteschlangen und Individuen

Eine interessante Frage stellt sich, wenn man Warteschlangen nicht unabhängig von einer konkreten Anwendung betrachtet, sondern bei den sich anstellenden Objekten z.B. an Personen denkt. Wir werden dann fordern, dass sich eine Person, die schon in der Schlange steht, nicht noch einmal anstellen kann.
Arbeitet man mit mehreren Warteschlangen, wie sie beispielsweise bei einem Supermarkt an der Kasse vorzufinden sind, dann würde man die Anforderungen sogar noch darum erweitern, dass ein Objekt in keiner der Warteschlangen sich zusätzlich anstellen kann.
In diesem Fall muss man ernsthaft darüber nachdenken, in welcher Klasse man die Absicherung ansiedelt:
Es ist sicher eine Eigenschaft eines konkreten Person – Objektes, dass es sich nicht gleichzeitig in zwei oder mehr Warteschlangen befinden kann. Daher könnte man der Person ein Attribut „**`angestellt`**“ zuordnen, das dann beim Anstellen und Verlassen der Warteschlange entsprechend verwaltet wird.
Aus der Sicht einer einzelnen Warteschlange ist eine Verwaltung mit einer Methode **`contains(Object obj)`** angemessen, die auch intern beim **`add(Object obj)`** verwendet wird.

```
    public boolean contains(Object obj){
        for (Iterator it=iterator();it.hasNext();){
            if (it.next()==obj) return true;}
        return false;
    }
```

und:

```
    public boolean add(Object obj){
        if (isEmpty()) {
            // Ein neues Listenelement wird angelegt:
            anfang = new ListenElement();
            // Das erste Element ist gleichzeitig auch das letzte:
            ende = anfang;
        } else {
            // Ein neues Element wird erzeugt,
            ende.nachfolger = new ListenElement();
            // und angehängt
            ende = ende.nachfolger;
        }
        if (contains(obj)) return false;   // keine Wiederholung!
        else {
            ende.inhalt = obj;             // Inhalt wird zugewiesen
            ende.nachfolger = null;        // Als Ende kennzeichnen
            anzahl++;                      // Anzahl wird angepasst.
            return true;
        }
    }
```

sets

Die Frage, ob man Elemente mehrfach in einer Sammlung vorfinden kann, ist von allgemeinerer Bedeutung und findet sich in einem besonderen Sammlungstyp wieder, den **`sets`**. Der Begriff stammt von mathematischen Anwendungen, er bedeutet im Deutschen Menge. In einer Menge kann ein Element höchstens einmal enthalten sein.
Dabei ist es übrigens in der Informatik nicht zwingend, dass dies intern auch wirklich so realisiert wird. Es wäre durchaus denkbar, wenn mehrfaches add für dasselbe Objekt nach außen dadurch keine erkennbare Veränderung macht, dass beispielsweise

Methoden wie size() nur einmal zählen und bei remove alle entfernt werden.