Die nachfolgenden Ausführungen sind im Zusammenhang mit einer Lehrerfortbildungsveranstaltung im November 2006 entstanden. Das ursprünglich erstellte Skript habe ich im Nachhinein nochmal überarbeitet und um die Anhänge erweitert, um so die Anregungen, Wünsche und Vorschläge aus der dem Referat folgenden Diskussion aufzunehmen und um die vollständige Java-Programmierung vorzuführen. Außerdem sollte der Inhalt auch für Leserinnen und Leser verständlich werden, die nicht am Vortrag teilgenommen haben.

Robert Krell:

Die abstrakten Datentypen Keller und Schlange

Im Alltag ist es ständige Praxis, dass Gegenstände an Speicherorten abgelegt bzw. irgendwo gelagert und von dort zu gegebener Zeit wieder hervorgeholt werden. Im bequemsten Fall gibt der Einlagerende seine Gegenstände beispielsweise an einer Art Gepäckschalter ab und erhält sie auf Nachfrage auch von dort wieder, ohne sich um die Details der zwischenzeitlichen Lagerung, die Einrichtung der Lagerhalle oder die interne Organisation des Speicherbetriebs kümmern zu müssen. Entsprechend einfach sollen auch im Computer Daten bzw. Objekte gespeichert werden können, ohne dass sich der Benutzer (hier der Anwendungs-Programmierer) um Einzelheiten des Lagerns kümmern muss.

Abstrakte Datentypen (ADT) sind solche Datenspeicher, auf die nur mittels weniger festgelegter Methoden zugegriffen werden kann, und wo die interne Verwaltung vorm Benutzer versteckt wird.

Ist beispielsweise *speicher* ein solches Datenspeicherobjekt, so braucht man nur vier Methoden:
- void *speicher.rein (neuesObjekt*) speichert ein neues Element, d.h. nimmt es in den *speicher* auf.
- boolean *speicher.istLeer()* fragt ab, ob der *speicher* überhaupt schon/noch Elemente enthält
- Object *speicher.raus()* entfernt ein gespeichertes Element
- Object *speicher.zeige()* liefert die Kopie bzw. den Verweis (=Adresse, Referenz) eines gespeicherten Elements, ohne es zu entfernen oder den *speicher* zu verändern.

Ein Objekt *speicher* eines solchen abstrakten Datentyps wird einmal z.B. mit *abstrakterDatentyp speicher = new abstrakterDatentyp()* definiert und erzeugt, d.h. neu und leer angelegt. Danach soll der Zugriff nur mit *rein*, *istLeer*, *raus* oder *zeige* möglich sein (wobei die Methoden natürlich auch andere Namen haben dürfen – in der Literatur sind englischsprachige Bezeichnungen üblich, die noch vorgestellt werden. Dabei haben sich leider für verschiedenen Speicher unterschiedliche Namen für Methoden mit gleicher Funktion durchgesetzt, während ich einheitlich für alle Speicher immer *rein*, *istLeer*, *zeige* und *raus* verwenden will). Der Anwender muss über das Innenleben des Speichers nichts wissen, sondern braucht nur die Funktionsweise der vier genannten Methoden kennen.

Dabei unterscheiden sich die drei Datentypen **Keller, Schlange** und **Liste** in der Funktion der Methoden – nämlich darin, ob
- das jeweils zuletzt eingefügte Element (beim Keller),
- das am längsten gelagerte Element (aus der Schlange) oder
- ein beliebiges, mit zusätzlichen Methoden wählbares Element (in der Liste)

beim nächsten Aufruf von *zeige* angezeigt oder mit *raus* entfernt wird. Für jeden der drei Speichertypen sind dabei trotz der gerade festgelegten Funktion ganz unterschiedliche interne Realisierungen möglich – diese Implementationsdetails sollen aber in jedem Fall vor dem Anwender verborgen bleiben.

Das Konzept unabhängiger Module mit klaren Schnittstellen und gekapseltem Innenleben hat sich im Alltag längst bewährt: In der Einbauküche etwa befinden sich Aussparungen genormter Größe für die Aufnahme von Herd und Kühlschrank. Hier können Fabrikate verschiedener Hersteller problemlos eingesetzt werden. Der normale Küchenbenutzer ebenso wie der aufbauende Schreiner weiß in der Regel nichts (und braucht nichts zu wissen) über die Arbeitsweise des im Kühlschrank enthaltenen Kühlaggregats. Motor und Elektrik sind im Gehäuse versteckt und können nicht berührt werden. Ein Steckeranschluss reicht. Es ist nur die Funktion wichtig: Man muss Lebensmittel einlagern können (Tür auf, Ware rein, Tür zu) und irgendwann wieder gekühlt entnehmen können (Tür auf, Ware raus, Tür zu). Auch eine Inspektion des Kühlschrankinhalts ohne Veränderung ist möglich (Tür auf, gucken, Tür zu). Der Zugriff geschieht also nur über hersteller- und implementationsunabhängige Standardmethoden. Obwohl nach außen alle Einbaukühlschränke prinzipiell gleich wirken, gibt es trotzdem Unterschiede: Kühlschränke können mit Kompressor oder nach der Absorptionsmethode arbeiten; manche Geräte mögen schneller oder langsamer kühlen, unterschiedliche Geräuschentwicklung haben oder verschieden sparsam mit der eingesetzten Energie umgehen – darin unterscheiden sich schlechte oder gute Implementationen. Trotzdem kann sich kein Küchenlieferant oder Kühlschrankbauer leisten, von den Normmaßen abzuweichen, ohne seine Absatzchancen drastisch zu senken. Trotz Preisdrucks würde kein Küchenplaner auf die Idee kommen, dadurch ein paar Cent zu sparen, dass er beispielsweise die Türachse des Kühlschranks etwas verlängert und mit dem gleichen Stift die drüber befindliche Schrankklappe hält, um so ein Scharnier weniger zu brauchen. Außer verrückten Bastlern würde niemand eine Leitung vom Kühlschranklicht in den Backofen führen, um so die Backofenbeleuchtung ohne eigenes Vorschaltgerät zu realisieren. Neben elektrischen Gefahren würde man unübersehbare Abhängigkeiten riskieren, die den späteren problemlosen Austausch einzelner Geräte erschweren. Lieber baut man in mehrere benachbarte Geräte jeweils eigene ähnliche Netzteile ein, und nimmt gerne in Kauf, dass dadurch insgesamt ein paar mehr Scharniere, Schrauben und Drähte gebraucht werden als bei einer individuell zusammen gebastelten Lösung. Die jeweils benötigte Technik wird lokal gekapselt und vorm Benutzer verborgen. Gerade dieses Konzept hat die verteilte Massenproduktion ermöglicht und mit einzelnen, für sich optimierten und getesteten Geräten zu deutlicher Qualitätsverbesserung bei viel geringeren Preisen geführt als die individuelle Einzelanfertigung. Diese Vorteile will auch die Informatik nutzen, indem dieses Konzept übernommen wird.

Keller

In vielen Fällen ist es nicht nötig, wahlfrei auf gespeicherte Daten zuzugreifen:

- Am Buffett ist es keine wirkliche Beschränkung, dass ein Gast jeweils nur den obersten Teller aus der Warmhaltevorrichtung nehmen kann (auch wenn der unterste Teller schon viel länger in der Heizröhre steckt).
- Bewohner von Mehrfamilienhäusern haben sich damit abgefunden, aus dem kleinen Keller erst die Wasserkästen und das Fahrrad ausräumen zu müssen, bevor man an den Karton mit der Weihnachts-Dekoration vom letzten Jahr kommt.
- Werden in einem Computerprogramm Methoden innerhalb von anderen Methoden aufgerufen, so ersetzen die lokalen Variablen der letzten Methode die früher geöffneter Strukturen. Erst wenn die zuletzt aufgerufene Methode beendet und ihr Datenbereich abgeräumt ist, braucht und hat man wieder Zugriff auf die Daten der früher gestarteten Methode.
- Und sucht man durch Versuch und Irrtum nach der Backtracking-Methode die Lösung eines Rätsels oder den Weg aus einem Labyrinth, so wird bei einer Sackgasse nur die zuletzt getroffene (und zuletzt gespeicherte) Entscheidung rückgängig gemacht und durch einen anderen Versuch ersetzt. Erst, wenn das auch nicht hilft, geht man noch ein Stück weiter auf dem betretenen Weg zurück.

Alle genannten Beispiele haben gemeinsam, dass die zuletzt gespeicherten Daten als erstes gebraucht bzw. als nächstes entfernt werden: Der geeignete Datenspeicher ist ein LIFO-Speicher: last in, first out. Ein solcher Speichertyp wird Keller, Stapel oder engl. **stack** genannt. Man kann sich einen Keller modellhaft als eine enge gerade und

nur an einer Seite geöffnete/zugängliche Röhre vorstellen: *rein* wirkt immer am selbem Ende des Datenspeichers, *zeige* zeigt genau das an diesem Ende befindliche und zuletzt eingekellerte Element. Selbst *raus* bedarf keiner weiteren Angabe: auch diese Methode greift automatisch immer auf das gleiche Ende des Speichers zu und entfernt das (dort) zuletzt eingekellerte Element. In der Literatur sind statt *rein, istLeer, raus* und *zeige* meist die englischsprachigen Bezeichnungen *push, isEmpty, pop* und *top* gebräuchlich. Bei anderen Datentypen haben die gleichen Methoden aber wieder andere Namen. Das erschwert aber den Vergleich über mehrere Datentypen und verhindert Vererbung und Polymorphie, sodass ich mit meinen deutschen Methodennamen arbeiten werde.

Wie die Standard-Methoden wirken, kann wie oben im frei formulierten Text beschrieben werden. Außerdem sind auch stärker formalisierte Spezifikationen üblich:

- die eher algebraisch-mengentheoretische, so genannte „axiomatische" Beschreibung, die z.B. *rein* als eine Abbildung vom Kreuzprodukt der Menge 𝕂 aller Keller und der Menge 𝔼 aller speicherbaren Elemente auf die Menge 𝕂 aller Keller auffasst, wobei aus dem bestehenden Keller *k* durch Hinzunahme des mit *rein* eingefügten Elements *e* ein neuer Keller *k'* entsteht, der jetzt ein Element mehr hat – *rein*: $\mathbb{K} \times \mathbb{E} \rightarrow \mathbb{K}$ mit *rein*: $(k, e) \mapsto k'$, wobei der Unterschied von *k* und *k'* und die Position von *e* innerhalb von *k'* noch durch zusätzlichen Text oder durch Beispieloperationen („Axiome") beschrieben werden muss. Diese Spezifikation passt eher zur prozeduralen Denkweise (da der Keller ebenfalls als Argument bzw. Parameter verwendet wird und nicht als agierendes Objekt auftritt), obwohl natürlich ein abstrakter Datentyp nicht nur unabhängig von der programmtechnischen Realisierung, sondern auch völlig unabhängig von der Programmiersprache und vom Programmierstil ist. Nur die Notation der Beschreibung lehnt sich hier (leider) an ein bestimmtes Paradigma an.

- die Vorher-Nachher-Beschreibung des Speicherobjekts Keller, wobei unter „Vorher" der Zustand des Kellers vor Ausführung der Methode genannt wird und eventuell nötige Voraussetzungen notiert werden, damit die Methode überhaupt ausgeführt werden kann. Unter „Nachher" steht dann, was die Methode bewirkt (hat)

Diese letzte Beschreibungsart soll hier verwendet werden. Ob die Nachher-Texte tatsächlich in der häufig verwendeten Vergangenheitsform am leichtesten lesbar sind, sei allerdings dahingestellt.

Keller — engl.: stack

Konstruktor	*new Keller()*	engl.: create() (z.T. auch init())
Vorher	./.	
Nachher	Es wurde ein neuer, leerer Keller erzeugt	
Auftrag	*rein (Object neuesElement) : void*	engl.: push (neuesEl.)
Vorher	Es existiert ein Keller	
Nachher	Das Objekt neuesElement wurde dem Keller 'oben' hinzugefügt	

Anfrage	*istLeer() : boolean* engl.: isEmpty()
Vorher	Es existiert ein Keller
Nachher	Die Anfrage hat den Wert *true* geliefert, wenn der Keller keine Elemente enthielt. Sonst war das Ergebnis *false*. Der Keller wurde nicht verändert.
Auftrag	*zeige() : Object* engl.: top()
Vorher	Es existiert ein nicht-leerer Keller
Nachher	Der Auftrag hat als Funktionswert das oberste (=zuletzt eingekellerte) Kellerelement geliefert. Der Keller wurde nicht verändert.
Auftrag	*raus() : Object* engl.: pop()
Vorher	Es existiert ein nicht-leerer Keller
Nachher	Der Auftrag hat das oberste Element aus dem Keller entfernt und es als Funktionswert geliefert.

Je nach Autor gibt *raus* das entfernte Element als Funktionswert zurück (*raus*: $\mathbb{K} \rightarrow \mathbb{K} \times \mathbb{E}$ wie hier beschrieben) oder entfernt es stumm, ohne es zu nennen (*raus*: $\mathbb{K} \rightarrow \mathbb{K}$). Der in Java implementierte **stack** zeigt bei *pop* das Element nochmal; außerdem bietet diese Variante in einer Reihe von Anwendungen eine gewisse Erleichterung (Verzicht auf den vorherigen zusätzlichen Aufruf von *zeige*), sodass hier so implementiert werden soll. Im Unterricht sollte aber durchaus mit verschiedenen Varianten gespielt werden. Ob wie oben der Ergebnistyp wie im UML-Diagramm und in Pascal nachgestellt angegeben wird, oder javaartig zuerst genannt wird, sollte letztlich egal sein. Lehrer und Schüler sollten die Beschreibung in verschiedenen Varianten verstehen können, also auch in der mir etwas besser gefallenden Version wie etwa:

Anfrage	*boolean istLeer()*
Nachher	Die Anfrage liefert den Wert *true*, wenn der Keller keine Elemente enthält – sonst ist das Ergebnis *false*. Der Keller wird nicht verändert.

Der verdeutlichende Zusatz „Der Keller wird nicht verändert“ kann natürlich entfallen: Änderungen, die nicht beschrieben werden, passieren nicht. Und oft wird auf die „Vorher“-Zeile ganz verzichtet, sofern nur die Existenz des Datenspeichers gefordert wird, aber keine weiteren Voraussetzungen (wie „nicht-leer“) nötig sind.

Ein Wort noch zum verwendeten Elementtyp: Von mir wurde jeweils *Object* genannt, weil es der allgemeinste in Java verwendbare Datentyp ist. Alle – auch nach beliebigen eigenen Klassen selbst erstellte – Objekte sind von diesem Typ. Daher reicht eine Kellerimplementation, um alle, auch unterschiedliche Objekte aufzunehmen. Um im Bild zu bleiben: Wir brauchen keinen Wurst-Kühlschrank, keinen Käse-Kühlschrank, keinen Wein-Kühlschrank und keinen extra Milch-Kühlschrank, um unterschiedliche Lebensmittel kühl zu lagern, sondern kommen mit einem Universal-Kühlschrank für alle Lebensmittel aus. Natürlich könnte man statt *Object* auch einen spezielleren Elementtyp angeben, würde damit aber die Verwendbarkeit des Datentyps (unnötig) beschränken. Will man in den *Object*-Keller allerdings Elemente der primitiven Datentypen *int*, *double*, *char* oder *boolean* einlagern, müssen diese bis einschließlich Java 1.42 zunächst in ein Objekt ihrer Hüll-(=Wrapper-)Klasse umgewandelt werden – z.B.

kann *x* bei *int x = 3* nicht selbst eingekellert werden, sondern muss mit *keller.rein(new Integer(x))* innerhalb einer Objekthülle eingekellert werden. Beim Rausnehmen kann mit *Integer zahlobjekt = (Integer) keller.raus()* das oberste Element entfernt und als Objekt erhalten werden; die primitive Ganzzahl 3 erhält man dann aus dem Zahlobjekt durch *x = zahlobjekt.intValue()*. Das so genannte 'type casting' mit „*(Integer)*" vor *keller.raus()* ist in Java nötig, damit das allgemeine vom Keller gelieferte Objekt wirklich der Variablen *zahlobjekt* vom spezielleren Objekttyp *Integer* zugewiesen werden kann: der Java-Programmierer wird durch diesem Zusatz gezwungen, sich über eventuelle Spezialisierungen klar zu werden. Außerdem hätte ein allgemeines *Object* keine Methode *intValue*!

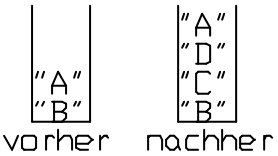
[Anmerkung: In/ab Java 1.5 (auch "Java 5" genannt) kann bei der Zuweisung von primitiven Typen an Objektvariable auf die explizite Angabe der Wrapperklasse verzichtet werden – Java 1.5 macht das dann automatisch ("autoboxing"), sodass hier auch *keller.rein(3*) oder *keller.rein(x)* erlaubt wäre. Auch ein Auto-Unboxing *int x = (Integer) keller.raus()* wäre möglich, wenn der Keller weiterhin allgemeine Objekte beinhaltet.
Außerdem sind in der neuen Java-Version auch 'generische' Datentypen möglich: Soll der Keller nur Elemente eines bestimmten Typs aufnehmen dürfen, könnte bei geeigneter Keller-Implementation zusätzlich in spitzen Klammern dieser Datentyp genannt werden. *raus* kann dann auch für den entsprechenden Datentyp sorgen, sodass kein nachträgliches Type-Casting nötig wird. So würde etwa durch *Keller<String> keller = new Keller<String>()* ein *keller* ausschließlich für String-Objekte erzeugt, bei dem später auch etwa *keller.raus().charAt(0)* ohne Type-Casting möglich ist. Ein ausführliches Beispiel findet sich im Anhang 3 ab Seite 27]

Aufgaben

0. Nennen Sie einige weitere Anwendungen für Keller (vgl. Seite 2)

1. Gegeben ist das Programmstück im Kasten. Wird es fehlerfrei ausgeführt? Zeichnen Sie den Keller nach jeder Änderung und geben Sie an, was zum Schluss in den Variablen *w* bis *z* steht!

```
Keller k = new Keller();
k.rein ("A");
k = new Keller();
k.rein ("B");
k.rein ("123");
String w = (String) k.zeige();
String x = (String) k.raus();
boolean y = k.istLeer();
String z = (String) k.raus();
w = (String) k.zeige();
```



2. Schreiben Sie ein Programmstück, das den gezeichneten Anfangszustand des Kellers in den geforderten Endzustand überführt!

3. Schreiben Sie ein Javaprogramm, das das Wort „Keller" oder ein beliebiges anderes im Programmtext vorgegebenes Wort zeichenweise einkellert und danach die einzelnen Kellerelemente (Strings von je 1 Zeichen Länge) wieder rausholt und auf der Konsole ausgibt, bis der Keller (wieder) leer ist.
(Hinweis: Haben Sie als Vorgabe zum Einkellern *String wort = "Keller";* definiert, so liefert *wort.charAt(0)* den ersten Buchstaben ('K'), *wort.charAt(1)* den nächsten ('e') und *wort.length()* die Gesamtzahl der Buchstaben des Wortes (hier 6). Zum Einkellern können Sie aus den Buchstaben durch Anhängen an einen Leerstring leicht Strings machen: ""+ 'K' = "K")

Kellerimplementationen

Nachdem wir bisher den Keller aus Benutzersicht [Sicht des Küchenbenutzers oder des aufbauenden Schreiners bzw. Sicht des (Anwendungs-)Programmierers wie in den Aufgaben 1 bis 3] kennen gelernt haben, wollen wir uns jetzt in die Rolle des Kühlschrank- bzw. Kellerherstellers hinein versetzen, und mögliche innere Aufbauten des

Kellers überlegen und diskutieren. Zuvor wollen wir aber durch ein Interface versprechen, die vier Methoden auch wirklich zu implementieren (wobei der Kommentar leider unverbindlich ist):

```
1 public interface ADT_Speicher
2 // legt nur fest, welche Methoden jeder Datenspeicher haben muss
3 {
4   public void rein (Object element); // packt das übergebene Element in den Speicher
5   public boolean istLeer(); // sagt, ob der Speicher leer ist
6   public Object zeige(); // zeigt ein Element ohne Ändern des Speichers
7   public Object raus(); // zeigt ein Element und löscht es aus dem Speicher
8 }
```

a) **statische Implementation des Kellers**

Es scheint nahe liegend, eine Reihung (Array) als interne Datenstruktur für den Keller zu verwenden: Alle eingekellerten Objekte kämen nacheinander in die einzelnen Komponenten (Felder) der Reihung. Ist z.B. sicher, dass der Keller zu keinem Zeitpunkt mehr als 100 Elemente gleichzeitig beinhalten muss, bietet sich an, das Attribut (Datenfeld)

Object [ ] reihung = new Object [100];

zu verwenden. Kommen in den zuvor leeren Keller zuerst das Objekt "A" (ältestes), dann "B" und schließlich "C" (als letztes/jüngstes Element), so sind – wenn man die Reihung von vorne füllen will – zwei Anordnungen möglich:

Obwohl die linke Füllung dadurch besticht, dass sie der Modellvorstellung vom Keller als Röhre mit einem offenen Ende (hier dem linken Rand) näher kommt und *rein*, *zeige* und *raus* ihr Element immer nur im Feld *reihung[0]* speichern oder finden, hat sie doch einen gravierenden Nachteil: Wird ein Element eingekellert oder herausgenommen, muss der gesamte übrige Kellerinhalt Element für Element verschoben werden. Dies ist unnötig aufwändig. Bei der rechts gezeigten Anordnung muss man sich zwar in einem zusätzlichen (ganzzahligen) Datenfeld *kopf* merken, an welcher Stelle das zuletzt eingekellerte Element steht. Aber *rein* und *raus* erfordern rechts nur einen 1-Schritt-Aufwand unabhängig von der aktuellen Kellerbelegung. Es muss nichts verschoben werden. Daher wird die rechts gezeigte Struktur in Java implementiert (wobei gegen mögliche Fehlbedienung keine Vorkehrung getroffen wird – der Anwender muss sorgfältig mit dem Keller umgehen und sollte z.B. nicht versuchen, auf einen leeren Keller *raus* anzuwenden!):

Keller in statischer Implementation (rechte Zeichnung von Seite 6) (im Hinblick auf eine spätere Vererbung an eine Schlange [Aufgabe 16] sind evtl. noch kleinere Änderungen sinnvoll – siehe auch Nachtrag am Endes dieses Referats!)

```
1 public class ADT_Keller implements ADT_Speicher
2 // Version 1: Keller in statischer Implementation mit einer Reihung
3 {
4   protected Object[] reihung;  // protected statt private für spätere Vererbung
5   protected int kopf;          // Index/Position des obersten (letzten) Elements
6   private final int MAX = 100; // Max. Anzahl der Kellerelemente = statische Größe
7
8   public ADT_Keller()          // Konstruktor für neuen, leeren Keller
```

```
 9    {
10      reihung = new Object[MAX]; // Anlegen der statischen Datenstruktur
11      kopf = -1;                 // negativ: noch kein oberstes Element!
12    }
13
14    public void rein (Object element) // übergebenes Element kommt in Speicher
15    {
16      kopf++;
17      reihung[kopf] = element;   // das letzte eingekellerte (=oberste) Element
18    }                            // steht am Index kopf
19
20    public boolean istLeer()     // sagt, ob der Speicher leer ist
21    {
22      return (kopf == -1);
23    }
24
25    public Object zeige()        // zeigt oberstes Element ohne Ändern des Speichers
26    {
27      return (reihung[kopf]);
28    }
29
30    public Object raus()         // zeigt oberstes El. und löscht es aus dem Speicher
31    {
32      kopf--;
33      return (reihung[kopf+1]);
34    }
35  }
```

Aufgaben

4. Führen Sie einen Schreibtisch-/Bleistifttest für die vorstehende Kellerimplementation durch, indem Sie die Belegung von *reihung* und *kopf* angeben, wenn der Keller die in Aufgabe 1 genannten Aufträge und Anfragen abarbeitet (ohne letzten Befehl aus dem Kasten von Aufgabe 1 – s. Seite 5)

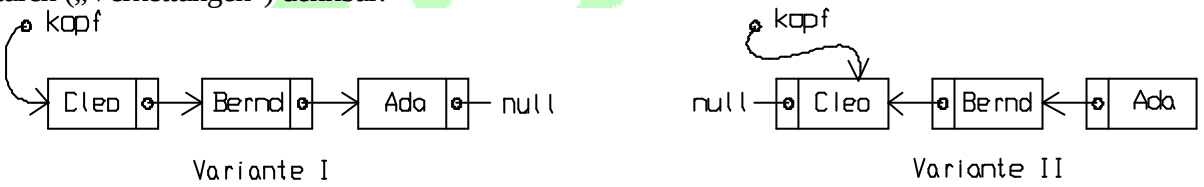
5. Auch wenn dies aufwändiger ist: Erstellen Sie eine Implementation des Kellers entsprechend der linken Zeichnung auf Seite 6 (mit Schieben). (Hinweis: Damit das Kellerende erkannt werden kann, könnte rechts immer noch ein leeres Objekt sozusagen als Schlusslicht mitgeschoben werden, das der Kontruktor anfangs mit/in *reihung[0] = null;* anlegt).

Die vorstehenden, deshalb statisch genannten Implementationen haben den Nachteil, dass die Reihung unabhängig von der tatsächlichen Belegung des Kellers einen festen Speicherplatz verbraucht: Egal, ob keines, zwei, fünf oder 98 Elemente im Keller sind, umfasst die Reihung immer genau 100 Felder und belegt entsprechend viel Platz. Und ein Überschreiten der Obergrenze ist nie möglich. (Dass in Java pro Feld der *reihung* nur die Adresse eines Objekts steht, also letztlich nur Platz für 100 Speicher-Adressen fest verbraucht wird, nicht wirklich Platz für 100 erst zur Laufzeit in ihrer Größe festgelegten Objekte, sei für Interessierte angemerkt, schmälert aber die Richtigkeit der vorstehenden Aussage nicht).

b) **dynamische Implementation des Kellers**

Wer ein Text(verarbeitungs-)-Dokument schreibt oder ändert, muss nicht vorher die Anzahl der geplanten Zeilen oder Buchstaben festlegen und entsprechenden Speicherplatz im Hauptspeicher oder auf der Festplatte reservieren: Trotzdem lässt sich das Dokument in verschiedenen Stadien immer wieder unter gleichem Namen auf der Festplatte speichern und belegt dort immer nur den gerade für die aktuelle Textmenge benötigten Platz, den das Betriebssystem in irgendwelchen freien, nicht notwendig zusammenhängenden Bereichen auf der Festplatte findet oder bei (Nicht-)Bedarf wieder für andere Dateien frei gibt. Eine solche dynamische Nutzung (allerdings des Hauptspeichers, nicht der Festplatte[1]) soll jetzt für den Keller eingerichtet werden.

Da man beim Keller von außen bekanntlich nur auf das oberste Element zugreifen darf, muss nur dieses eine Element im Programm bzw. durch die eingeführten Methoden erreichbar sein, d.h. nur sein Speicherort muss z.B. in der Variablen *kopf* im Programmtext festgehalten werden (Achtung: es wird wieder der Name *kopf* verwendet. Dieses Datenfeld ist aber nicht mehr ganzzahlig wie der gleichnamige Index für die statische Reihung aus dem letzten Abschnitt, sondern erhält unten einen anderem Typ). Wenn sich jedes Element (bzw. der umhüllende Knoten) merkt, welches jeweils eine andere Element davor im Keller war/ist, reicht dies für die Kellerverwaltung. Im Unterricht habe ich gute Erfahrung damit, dass man im Spiel einige Schüler als Elemente einkellert und sich der Lehrer (als Kopf und oberster Kellerverwalter) jeweils nur den Namen des zuletzt eingekellerten (und als erstes wieder auszukellernden) Schülers merkt. Merkt sich jeder eingekellerte Schüler seinerseits bzw. jede in den Keller aufgenommene Schülerin ihrerseits jeweils auch nur einen Namen, nämlich den der davor eingekellerten Person, kann der gesamte Keller mit den vier Methoden, *rein*, *istLeer*, *zeige* und *raus* verwaltet werden, wie man handelnd erfährt. Das Merken eines Namens kann man dabei dadurch ersetzen (und so das Spiel für die Mitschülerinnen und Mitschüler augenfällig nachvollziehbar machen), dass man mit ausgestrecktem Arm auf den zu merkenden Schüler zeigt. Werden in den bis dahin leeren Keller (der Lehrer hatte seine Hand zunächst in der Tasche stecken; in Java wird das durch das Schlüsselwort *null* [= nichts, gesprochen „nall“] symbolisiert) erst die Schülerin Ada, dann der Schüler Bernd und zuletzt Schülerin Cleo jeweils mit *rein* aufgenommen, wären sogar wieder zwei Verweis-/Zeiger-Strukturen („Verkettungen“) denkbar:



Versucht man auch Variante II im Spiel, so wird deutlich, dass beim Entfernen von Cleo der Kopf ins Leere (oder weiterhin auf den bisher von Cleo beanspruchten Speicherplatz) zeigt, ohne dass es eine Möglichkeit gibt, ihn auf den Schüler Bernd zu richten, obwohl Bernd jetzt oben im Keller stehen müsste: die einzige, die in Variante II auf Bernd zeigt, ist die Schülerin Ada, die aber ihrerseits nicht zu erreichen ist, weil nichts/niemand auf sie zeigt. Der gesamte Kellerinhalt wäre verloren, da man ihn nicht mehr im Hauptspeicher finden kann.

In Variante I sieht es zum Glück besser aus: Bevor Cleo endgültig aus dem Keller entfernt wird, kann sie (auf die Kopf ja bisher zeigt) noch mitteilen, dass Bernd der nächste ist. Kopiert man diese Information in Kopf, so zeigt Kopf ab sofort auf Bernd, der somit zum obersten Kellerelement wird. Dass Cleo als unerreichbare Karteileiche, auf die niemand mehr zeigt, noch links „rum hängt“ und unnötigerweise weiter auf Bernd zeigt, stört die Kellerverwaltung nicht. Und irgendwann räumt der Garbage Collector von Java solche unerreichbaren Karteileichen automatisch fort und gibt den Speicherplatz wieder frei, ohne dass es eines ausdrücklichen ‘dispose’-Aufrufs oder anderer Vorkehrungen im Programm bedarf.

---

[1]) obwohl eine Implementation eines Kellers in einer Datei bzw. einem ObjectInput- oder -OutputStream leicht möglich ist und eine lohnende Übung darstellt!

In einem Warenlager lässt sich die vorgestellte Verkettung mit Zeigern etwa dadurch realisieren, dass man jeden zu speichernden Gegenstand in eine rote Plastikwanne (mit Extraabteil für eine Pappkarte) füllt und alles zusammen auf eine nummerierte Position ins Lagerregal stellt. Auf dem leeren Regalplatz lag bis dahin eine Karte mit der Nummer des Lagerplatzes, die der Lagerverwalter mitnehmen und an die eine mit *kopf* bezeichnete Stelle auf seinem Schreibtisch legen kann, damit er immer weiß, wo sich die Wanne mit dem zuletzt eingelagerten Gegenstand befindet. Weil auf dem Schreibtisch aber immer nur eine Nummern-Karte liegen soll, muss diese Nummer beim Einlagern des nächsten Gegenstandes wieder weggenommen und in das Extraabteil der neuen roten Plastikwanne mit dem nächsten Objekt gesteckt werden. So „merkt" sich also die Plastikwanne im Extraabteil den Ort des vorigen Elements bzw. „zeigt" so auf die unmittelbar zuvor eingelagerte Wanne. Die Nummer ihres eigenen Platzes kommt wieder auf den Verwalterschreibtisch. Man macht sich leicht klar, dass damit das Lager auch dann verwaltet werden kann, wenn das Regal teilweise für andere Zwecke genutzt wird und die Wannen nicht alle nebeneinander, sondern kreuz und quer in der Halle verteilt stehen.

Im Computer bzw. in der Informatik-Literatur heißen die Wannen üblicherweise Knoten. Ihr Bauplan muss ein Fach für den eigentlichen Inhalt (das zu lagernde Objekt) und das Extraabteil (hier namens *zeiger* für die Adresse der letzten Wanne bzw. des vorher eingelagerten Knotens) vorsehen. Da Java in den Objektvariablen die Adressen der Objekte im Hauptspeicher speichert (und nicht die Objekte selbst), kann, ja muss also das Extraabteil als Teil eines Knotens selbst wieder vom Objekttyp Knoten sein. Insofern handelt es sich hier um eine rekursive Definition. Eine Unterscheidung von Objekten und darauf verweisende Zeigertypen ist in Java weder nötig noch möglich.

```
1  public class ADT_Knoten       // = rote Plastikwanne
2  {
3    private Object inhalt;      // Fach für den Inhalt
4    private ADT_Knoten zeiger;  // "Extraabteil" für den Verweis auf 1 anderen Knoten
5
6    public ADT_Knoten (Object neuesElement, ADT_Knoten nachbar) // Konstruktor
7    {
8      inhalt = neuesElement;
9      zeiger = nachbar;
10   }
11
12   public Object holeInhalt()
13   {
14     return (inhalt);
15   }
16
17   public ADT_Knoten holeNachbarKnoten()
18   {
19     return (zeiger);
20   }
21
22   public void ändereNachbarn (ADT_Knoten neuerNachbar) // für Schlange nötig
23   {
24     zeiger = neuerNachbar;
25   }
26 }
```

Unter Verwendung von Knoten nach vorstehendem Bauplan gelingt jetzt die dynamische Implementierung des Kellers nach Variante I. Da *kopf* auf einen Knoten zeigt bzw. einen Knoten (bzw. seine Adresse) aufnimmt, ist natürlich auch *kopf* vom Knotentyp. Im Unterricht braucht natürlich höchstens eine Methode vorgestellt werden; die

restlichen können die Schülerinnen und Schüler selbst erarbeiten und die richtige Funktion des programmierten Kellers in einem ausgeteilten Rahmenprogramm testen.

```
1  public class ADT_Keller implements ADT_Speicher
2  {
3    protected ADT_Knoten kopf; // Verankerung der Struktur
4
5    public ADT_Keller() // hier mit Extra-Konstruktor. Oder zwei Zeilen höher
6    {                   // die Variable kopf mit null initialisieren
7      kopf = null;      // um neuen leeren Keller anzulegen
8    }
9
10   public boolean istLeer() // sagt, ob der Speicher leer ist
11   {                        // genau dann wahr, wenn der Keller leer ist
12     return (kopf == null);
13   }
14
15   public Object zeige()    // zeigt/kopiert oberstes Element, ohne Keller zu ändern
16   {
17     return (kopf.holeInhalt());
18   }
19
20   public Object raus()     // zeigt oberstes El. und entfernt es aus dem Keller
21   {
22     Object altesElement = kopf.holeInhalt();
23     kopf = kopf.holeNachbarKnoten();
24     return (altesElement);
25   }
26
27   public void rein (Object neuesElement) // übergebenes Element kommt in Keller
28   {
29     ADT_Knoten neu = new ADT_Knoten (neuesElement, kopf);
30     kopf = neu;
31   }
32 }
```


Aufgaben

6. In der vorstehenden Methode *raus* wurde das alte Element – anders als bei der statischen Implementation – zunächst in einer lokalen Variable zwischen gespeichert (Zeile 22). Warum?

7. Bei der Methode *rein* gibt es keine Fallunterscheidung, ob das erste Element in einen vorher leeren Keller oder ein zusätzliches Element in einen bereits bestehenden Keller eingefügt wird. Wurde ein Fall vergessen? Ergänzen und/oder erklären Sie!

8. Führen Sie einen Schreibtisch-/Bleistifttest für die vorstehende Kellerimplementation durch, indem Sie die Knoten und Verkettungen zeichnen, wenn der Keller die in Aufgabe 1 genannten Aufträge und Anfragen abarbeitet (ohne den letzten Befehl aus dem Kasten von Aufgabe 1 – s. Seite 5). Entspricht der Keller wirklich in allen Punkten der Vorher-Nachher-Spezifikation von Seite 3?

9. Testen Sie die Keller unter der zur Verfügung gestellten Test-Oberfläche. Wieso konnte die Oberfläche schon kompiliert (bzw. in Eclipse fehlerfrei gespeichert) werden, bevor Ihr Keller fertig war? Erläutern Sie auch, warum der Anwender mit *istLeer* vor Ausführung der Methoden *zeige* oder *raus* sicher stellen sollte, dass der Keller wirklich Elemente enthält.

10. Vergleichen Sie die beiden Kellerimplementationen (Seiten 6 und 10) hinsichtlich des Speicherplatzbedarfs sowie hinsichtlich des Zeitaufwands aller Methoden. Arbeiten Sie dabei auch die Unterschiede zwischen dynamischer und statischer Realisierung heraus.

11. Die Variante II von Seite 8 wurde als ungeeignet verworfen, weil *raus* Schwiergkeiten bereitet. Zeigen Sie zur Übung, dass sich die anderen drei Methoden für diese Datenstruktur problemlos implementieren lassen.

12. Als Steuerzahler, der in der Hoffnung auf eine Erstattung die Steuererklärung frühzeitig abgibt, hat man manchmal den Eindruck, dass der Finanzamtmitarbeiter die eingehenden Anträge in einem Stapel (Keller) im Eingangskorb verwaltet. Welche Datenstruktur wäre gerechter? Wo sind Änderungen gegenüber dem Keller nötig? Notieren Sie auch die Vorher-Nachher-Spezifikation der neuen Datenstruktur (die natürlich weiterhin das Versprechen im Interface ADT_Speicher erfüllen soll).!

Schlange

Aufgabe 12 hat schon zur (Warte-)Schlange (engl. **queue**) übergeleitet: Für manche Anwendungen braucht man statt eines Kellers eben eher einen FIFO-Speicher (first in, first out), wo mit *raus* das Element entfernt wird, das bisher am längsten in der Schlange gespeichert war. Bei der Schlange wirken *rein* und *raus* also an verschiedenen Enden der Datenstruktur. Da *raus* wie bereits oben beim Vergleich der Varianten I und II (Seite 8) gesehen, weiterhin am *kopf* wirken muss, muss *rein* verändert werden, um am anderen Ende der Schlange zu wirken (das dies geht, hat schon Aufgabe 11 gezeigt). Der Kopf und die dort wirkenden Methoden *istLeer*, *zeige* und *raus* können vom Keller ohne Abstriche für die Schlange übernommen werden.

Um Mühe zu sparen, soll die Schlange alle Attribute und Methoden des (dynamischen) Kellers erben und nur *rein* überschreiben. [Anmerkung: Dieses Vorgehen ist zwar programmtechnisch geschickt, logisch aber insofern hinterfragbar, als die Schlange natürlich keine Abart oder Spezialisierung des Kellers ist, sondern ein anderer, dem Keller gleichberechtigter abstrakter Datentyp – die Schlange ist eher ein Geschwister als ein Nachkomme des Kellers. Thematisiert man dies später nach Kenntnis beider Datentypen nochmal, so bietet sich dann an, Keller und Schlange nochmal von einer gemeinsamen (abstrakten) Oberklasse *DynamischerSpeicherMitEndZugriff* abzuleiten, die nur die gleichen Methoden enthält und wo jeder Erbe 'sein' *rein* implementiert – siehe Anhang 2 ab Seite 22]

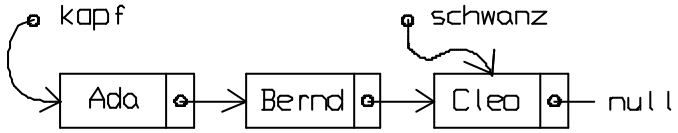
Hier aber zunächst die von der Implementationsentscheidung unabhängige Spezifikation:

Schlange — engl.: queue

Konstruktor	*new Schlange()*	engl.: create()
Nachher	Es wurde eine neue, leere Schlange erzeugt	

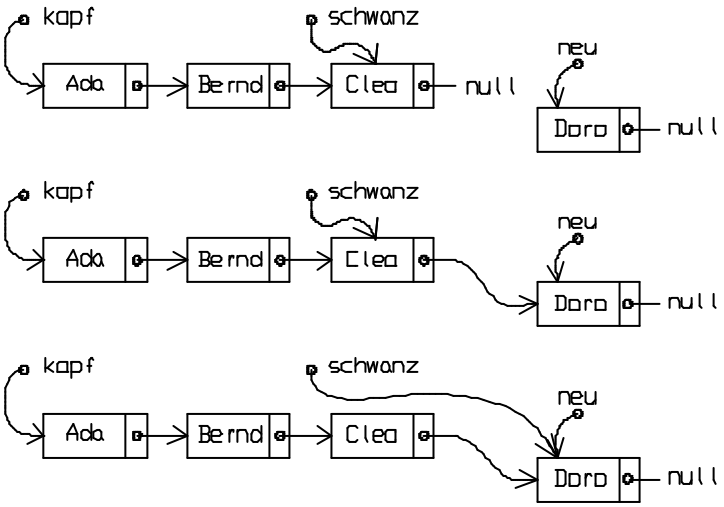
Auftrag	*rein (Object neuesElement) : void*	engl.: enqueue (neuesEl.)
Nachher	Das Objekt neuesElement wurde der Schlange 'hinten' ('am Schwanz') hinzugefügt	
Anfrage	*istLeer() : boolean*	engl.: isEmpty()
Nachher	Die Anfrage hat den Wert *true* geliefert, wenn die Schlange keine Elemente enthielt. Sonst war das Ergebnis *false*. [Die Schlange wurde nicht verändert.]	
Auftrag	*zeige() : Object*	engl.: front()
Vorher	Es existiert eine nicht-leere Schlange	
Nachher	Der Auftrag hat als Funktionswert das älteste (=zuerst 'eingeschlangte') Element geliefert. [Die Schlange wurde nicht verändert.]	
Auftrag	*raus() : Object*	engl.: dequeue()
Vorher	Es existiert eine nicht-leere Schlange	
Nachher	Der Auftrag hat das älteste (=zuerst 'eingeschlangte') Element aus der Schlange entfernt und es als Funktionswert geliefert.	

Nachdem sich beim Keller die dynamische Implementation als beste herausgestellt hat, soll die Schlange direkt dynamisch implementiert werden – und zwar (wie vor der Spezifikation mit Bedenken erklärt) als Erweiterung (Erbe) des Kellers.



Der vom Keller bereits bekannte „Zeiger" *kopf* zeigt jetzt auf das älteste Element (hier „Ada"), während ein zusätzlicher Zeiger *schwanz* auf das andere/hintere Ende mit dem zuletzt in die Schlange aufgenommenen Element zeigt (hier „Cleo"). Nur der neue Zeiger muss in der Klasse Schlange definiert werden, da *kopf* ja geerbt wird. Da am Kopf alles bleibt wie bisher, müssen die weiterhin dort wirkenden Methoden *istLeer, zeige* und *raus* nicht verändert bzw. brauchen nicht überschrieben werden.

Lediglich *rein* wird neu geschrieben und überschreibt/ersetzt dann bei der Schlange die bisherige gleichnamige Kellermethode (mit *enqueue* und *push* geht das wegen der verschiedenen Namen leider nicht!). Die Bildfolge rechts zeigt das Vorgehen beim Anfügen eines neuen Elements („Doro") an eine Schlange, die bereits Elemente enthält.



Will man allerdings das erste Element in eine bisher

leere Schlange aufnehmen, muss beachtet werden, dass *schwanz* noch auf nichts zeigt und es so insbesondere keinen Knoten *schwanz* gibt, dessen *zeiger* auf *neu* gerichtet (bzw. dessen Methode *ändereNachbarn* aufgerufen) werden kann. Vielmehr muss in diesem Fall *schwanz* direkt auf *neu* gerichtet werden – und *kopf* auch! Das nachfolgende Programm trägt dem Rechnung:

```
1  public class ADT_Schlange extends ADT_Keller
2   // implements ADT_Speicher kann entfallen, da durch Keller bereits zugesichert
3   // erweitert den Keller zur Schlange (dynamische Implementation mit Knoten)
4  {
5    private ADT_Knoten schwanz; // kopf existiert schon im und durch den Keller
6
7    public ADT_Schlange ()
8    {
9      super();        // Keller-Konstruktor aufrufen
10     schwanz = null; // zusätzlich
11   }
12
13   public void rein (Object neuesElement)  // wirkt jetzt am Ende
14   {
15     ADT_Knoten neu = new ADT_Knoten (neuesElement, null);
16     if (istLeer())
17     {
18       kopf = neu;   //1. Element: auch kopf muss auf den neuen Knoten zeigen
19     }               //hierfür darf kopf höchstens 'protected', nicht 'private' sein.
20     else
21     {
22       schwanz.ändereNachbarn(neu);  //sonst: Vorgänger muss auf neu zeigen
23     }
24     schwanz = neu;  //immer muss der Schwanz auf den letzten Knoten zeigen
25   }
26 }
```

Aufgaben

13. Ordnen Sie den drei Bildern der Bilderfolge für *rein* (Seite 12) die entsprechenden Zeilennummern des Programmtextes zu!

14. *istLeer* wird unverändert vom Keller übernommen (und sogar in Zeile 16 intern genutzt). Dabei wird nicht geprüft, ob auch *schwanz == null* ist. Stört das?

15. Der Zeiger/Knoten *schwanz* ist nicht unbedingt nötig, da der letze Knoten der Schlange (wo der Nachbar geändert werden muss) auch über den *kopf* erreicht werden kann. Schreiben Sie eine entsprechende Implementation in Java und vergleichen Sie diese mit der vorstehenden!

16. Die Schlange hätte natürlich auch statisch als Erbe des Kellers mit Reihung implementiert werden können. Erstellen Sie eine erste, vorläufige Implementation. Machen Sie sich klar, dass nach einiger Zeit (mit vielen rein-/raus-Operationen) das Ende der Reihung erreicht wird, auch wenn zu keinem Zeitpunkt gleichzeitig mehr als z.B. 5 Elemente in der Schlange sind. Sorgen Sie für Abhilfe! (Hinweis: *kopf = (kopf+1)%100* liefert den Divisionsrest der Summe nach der Division durch 100, sodass nach *kopf*=99 automatisch *kopf*=0 folgt. Ersetzen Sie im Keller(!) ggf. *kopf++* entsprechend!)

17. Testen Sie Ihre Schlangenimplementationen in der (im Anhang I) bereit gestellten Oberfläche

18. Sehen Sie sich den Quelltext der bereit gestellten Oberfläche genauer an (s. Anhang I) – insbesondere das Programmstück, das beim Betätigen der Schaltfläche „rein“ ausgeführt wird. Woher weiß Java, welches *rein* ausgeführt werden muss, das ursprüngliche Keller-*rein* oder das Schlangen-*rein*, wenn der *datenSspeicher* nach der Definition ‘nur’ vom Typ ADT_Speicher des Interfaces ist?

19. Nennen Sie einige Anwendungen für Schlangen!

Um das Programmieren der abstrakten Datenypen Keller und Schlange noch etwas mehr zu üben und gleichzeitig das Verständnis der hier ausgebreiteten Konzepte zu vertiefen, empfehle ich außerdem die Lösung folgender Aufgaben:

Aufgaben

20. Schreiben Sie drei vollständige Java-Klassen für einen *TauschKeller*, der zusätzlich zu den Standardmethoden eines normalen Kellers noch über den Auftrag *vertauscheOberste2Elemente* verfügt. Sie dürfen davon ausgehen, dass die Methode nur aufgerufen wird, wenn mindestens 3 Elemente vorhanden sind (und sollen keine Fehlerfälle berücksichtigen). Am besten implementieren Sie den *TauschKeller* als Erweiterung (Erbe) eine normalen Kellers, und zwar in drei Versionen:
    a. Innerhalb des *TauschKeller*s werden nur die Standardmethoden eines Kellers benutzt und es wird nirgends auf die konkrete Realisierung des zugrunde liegenden Kellers zurückgegriffen.
    b. Der *TauschKeller* ist Erbe des statischen Kellers mit einer Reihung und soll direkt auf die Reihung zugreifen. Standardmethoden sollen nicht verwendet werden.
    c. Der *TauschKeller* ist Erbe des dynamischen Kellers mit Knoten und soll den Tausch möglichst ausschließlich durch Umbiegen der Zeiger erreichen. Neue Knoten sollen nicht angelegt, Knoteninhalte nicht kopiert werden (allerdings dürfen durchaus zusätzliche Referenzen auf bereits bestehende Knoten eingeführt werden).

21. Eine *ErweiterteSchlange* soll über eine zusätzliche Methode *nenneElementzahl* verfügen (deren Aufruf die Schlange aber nicht dauerhaft ruinieren darf). Schreiben Sie wieder vollständige, kommentierte Java-Klassen (als Erweiterung [Erbe] einer normalen Schlange):
    a. Zählen Sie beim Aufruf von *nenneElementzahl* die Elemente der dynamischen Schlange mit einem zusätzlichen Zeiger, der nach der Reihe auf alle Knoten zeigt.
    b. Zählen Sie nur unter Verwendung der Standardmethoden der normalen Schlange. Zählen Sie mit einem geeigneten Zwischenspeicher, der während des Zählvorgangs die Elemente aufnimmt, oder erläutern Sie, wie Sie mit einem speziellen Inhalt ohne zusätzlichen Speicher die Elemente zählen wollen.
    c. Die *ErweiterteSchlange* soll über ein ganzzahliges Attribut *anzahl* verfügen, das bei jedem Aufruf von *rein* um 1 erhöht und bei jedem Aufruf von *raus* um 1 erniedrigt wird. Bei der Anfrage *nenneElementzahl* wird der Wert von *anzahl* ausgegeben. Schreiben Sie hierfür eine Java-Klasse, die nicht auf die konkrete Implementation der Oberklasse angewiesen ist.

    Vergleichen Sie die drei Realisierungen der erweiterten Schlange. Welche ist am besten?

Im Anschluss an die Behandlung von Keller und Schlange bietet sich die Besprechung von Listen an. Dort kann – etwa mit der Methode *anDenAnfang* – das erste/vorderste Element, bzw. von da aus mit *weiter* schritt- bzw. elementweise auch jedes andere Element ausgewählt werden. *zeige* und *raus* wirken bei der Liste nicht mehr automatisch auf den Anfang oder das Ende der Liste, sondern zeigen oder entfernen das so ausgewählte Element.

Und auch *rein* fügt das übergebene neue Element nicht mehr stets am Ende des Datenspeichers, sondern am besten immer gerade vor dem mit *weiter* ausgewählten Element ein. Aber das wäre Inhalt eines neuen Referats...

Weiteres Material

- Auf meiner Webseite www.r-krell.de gibt's auf den ausführlichen Seiten „Informatik mit Java" insbesondere im Teil e) mehr zu den linearen abstrakten Datentypen (Direktlink zur Übersicht: http://home.arcor.de/rkrell/if.htm#JavaInhalt) mit weiteren Beispielen und Anwendungen und einer Übersicht über entsprechende von Sun mitgelieferte Bibliotheksklassen.

- Auch die im Web veröffentlichten Informatik-Beispielaufgaben für das Zentralabitur 2007 im Gymnasium enthalten z.T. abstrakte Datentypen, z.B. Gk_Aufgabenbeispiel_3 und Lk_Aufgabenbeispiel_1 auf http://www.learn-line.nrw.de/angebote/abitur-gost-07/fach.php?fach=15.
  Aber auch auf Berufskolleg-Seiten sind die Datentypen ähnlich erläutert wie hier, so z.B. auf/in http://www.learn-line.nrw.de/angebote/abitur-wbk-08/download/if-datentypen-opjetk-ansatz-java.pdf

- Auf der Webseite von Ulrich Helmich werden in Folge 15 Keller, Schlange, Liste und sortierte Liste behandelt: http://www.u-helmich.de/inf/BlueJ/kurs121/index.html

- Bei Wikipedia findet man z.Z. nur recht knappe bzw. einseitig computerorientierte Artikel zu Kellern und Schlangen unter den Stichwörtern „Stapelspeicher" und „Warteschlange (Datenstruktur)"

- Sucht man bei Google etwa nach der Stichwortkombination „Informatik Keller Stapel Stack Schlange", so erhält man viele Links, meist auf Uni-/FH-Seiten, wo u.a. viele Folienpräsentationen als Ergänzung zu Vorlesungen angeboten werden. Leider sind die Darstellungen oft recht abstrakt. Die axiomatische Beschreibung herrscht vor.

- In Schulbüchern findet man natürlich auch Einiges. Interessant ist das neue Buch von Bernard Schriek, „Informatik mit Java – Band 2" („Nili"-Eigenverlag 2006; http://www.mg-werl.de/sum). Dort gibt's Schlangen (S. 38..51), Listen (S. 52..65) und Keller (S. 66..81) im Zusammenhang mit Projekten, die diese Strukturen verwenden. Schriek ist ein Verfechter der „hat"-, „istEin"- und ‚kennt"-Beziehungen. Folgerichtig heißt bei ihm der Verweis eines Knotens auf den nächsten nicht prosaisch *zeiger*, sondern etwa *kenntNachbar*.

- Ein allgemeines Grundgerüst für Oberflächenprogramme mit der Java-AWT gibt's auf meiner „Informatik mit Java"-Seite c), http://home.arcor.de//rkrell/if-java-c.htm#JavaEA. Ich teile diesen Text auf Papier und als java-Datei an meine Schülerinnen und Schüler aus und lasse eigene Oberflächen [wie auch die hier verwendete] durch Abwandlung/Ergänzung des Grundgerüsts entwickeln. Allerdings sollten dort [und hier im Anhang I] alle *richte..Ein*-Methoden besser als *private* deklariert sein; nur *führeAus* muss *public* sein.

Anhang

Anhang 1 : Testoberfläche

Im Folgenden wird eine mit der Java-AWT erstellte Test-Oberfläche gezeigt:

Hier die Start-Datei

```
1  // Teil des Projekts "LinADT" -- Lineare abstrakte Datentypen Keller und Schlange
2  // R. Krell (www.r-krell.de), 22.11.2006
3
4  public class ADT_Oberflaeche_Start
5  {
6    public static void main (String[] s)
7    {
8      ADT_Oberflaeche test = new ADT_Oberflaeche();
9      test.führeAus();
10   }
11 }
```

und die Datei mit der Funktionalität der Oberfläche, die ihrerseits Keller und Schlangen, aber zunächst vor allem das allgemeine Interface ADT_Speicher benutzt (s.u., Zeile 19). Der Programmtext ist nicht eingerahmt, weil hier ein Zusatzprogramm und nicht die Implementation der abstrakten Datentypen selbst gezeigt wird – die wurden oben schon vorgestellt.

```
1  // Teil des Projekts "LinADT" -- Lineare abstrakte Datentypen Keller und Schlange
2  // R. Krell (www.r-krell.de), 22.11.2006
3
4  import java.awt.*;
5  import java.awt.event.*;
6
7  public class ADT_Oberflaeche extends Frame
8  // Oberfläche zum Testen des Kellers bzw. weiterer LinADTs
9  {
10   TextField eingabe = new TextField("",8);
11   TextArea ausgabe = new TextArea("Zuerst einen LinADT mit der oberen Tastenreihe
12                                    wählen!\n",10,95);
13   CheckboxGroup cbg = new CheckboxGroup();
14   Checkbox cbZahl, cbZchn, cbText;
15   Button knKellerReih, knSchlangeReih,
16     knKellerDyn, knSchlangeDyn, knWeitere,
```

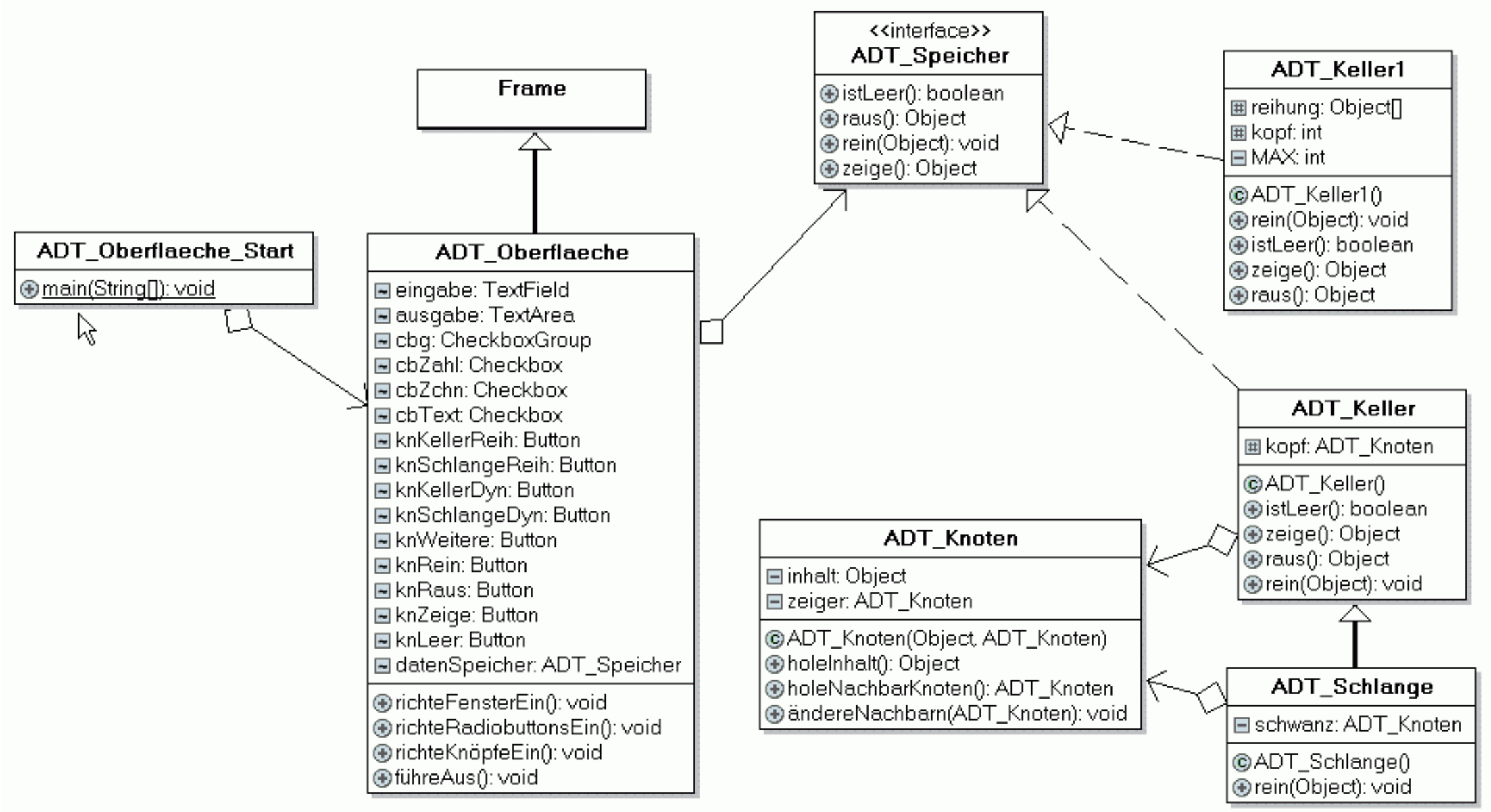
```
17      knRein, knRaus, knZeige, knLeer;// Knöpfe definieren
18
19    ADT_Speicher datenSpeicher; // Definieren eines allgemeinen LinADTs
20                                // namens datenSpeicher
21
22    private void richteFensterEin() // Fenster initalisieren und beschreiben
23    {
24      //WindowsListener hinzufügen, damit Schließknopf funktioniert
25      addWindowListener (
26        new WindowAdapter ()
27        {
28          public void windowClosing (WindowEvent ereignis)
29          {
30            setVisible (false);
31            dispose();
32            System.exit(0);
33          }
34        }
35      ); // runde Klammer vom Windowlistener geschlossen;
36
37      setTitle("Testprogramm für lineare abstrakte Datentypen (LinADTs)");
38      setSize (720,278); // Fenstergröße (Breite und Höhe in Pixeln) festlegen
39    }
40
41    private void richteRadiobuttonsEin()
42    {
43      cbZahl = new Checkbox ("Zahl", cbg, false);
44      cbZchn = new Checkbox ("Zeichen", cbg, false);
45      cbText = new Checkbox ("Text", cbg, true);
46    }
47
48    private void richteKnöpfeEin()
49    {
50      // Knöpfe erzeugen:
51      knKellerReih     = new Button ("Keller (statisch, mit Reihung)");
52      knKellerDyn      = new Button ("Keller (dyn., mit Knoten)");
53      knSchlangeReih   = new Button ("Schlange (statisch)");
54      knSchlangeDyn    = new Button ("Schlange (dynamisch)");
55      knWeitere        = new Button ("(unbelegt)");
56      knRein           = new Button ("rein");
57      knRaus           = new Button ("raus");
58      knZeige          = new Button ("zeige");
59      knLeer           = new Button ("istLeer ?");
60
61      //Funktion der Knöpfe festlegen:
62
63      knKellerReih.addActionListener (
64        new ActionListener ()
65        {
66          public void actionPerformed (ActionEvent e)
67          {
68           datenSpeicher = new ADT_Keller1 (); // Erzeugen von datenSpeicher
69             // als spezieller Keller mit Reihung
70           ausgabe.append ("Ein neuer leerer Keller wurde in/mit einer Reihung
71                            erzeugt.\n");
72           eingabe.requestFocus(); // aktviert die Eingabezeile,
73            // sodass künftige Tastatureingaben automatisch in der eingabe landen
74           eingabe.selectAll(); // macht evtl. Text in der Eingabezeile blau,
75            // sodass alter Text automatisch überschrieben wird
76          }
77        });
78
79      knKellerDyn.addActionListener (
80        new ActionListener ()
81        {
82          public void actionPerformed (ActionEvent e)
83          {
84           datenSpeicher = new ADT_Keller (); // Erzeugen von datenSpeicher
85            // als spezieller dynamischer Keller
```

```
86          ausgabe.append ("Ein neuer leerer Keller wurde dynamisch mit rekursiven
87                           Knoten erzeugt.\n");
88          eingabe.requestFocus();
89          eingabe.selectAll();
90        }
91      });
92
93    knSchlangeReih.addActionListener (
94      new ActionListener ()
95      {
96        public void actionPerformed (ActionEvent e)
97        {
98          datenSpeicher = new ADT_Schlange1 (); // Erzeugen von datenSpeicher
99          ausgabe.append ("Eine neue leere Schlange wurde in/mit einer Reihung
100                          erzeugt.\n");
101         eingabe.requestFocus();
102         eingabe.selectAll();
103       }
104     });
105
106   knSchlangeDyn.addActionListener (
107     new ActionListener ()
108     {
109       public void actionPerformed (ActionEvent e)
110       {
111         datenSpeicher = new ADT_Schlange(); // Erzeugen von datenSpeicher
112          // als dynamische Schlange
113         ausgabe.append ("Eine neue leere Schlange wurde dynamisch mit rekursiven
114                          Knoten erzeugt.\n");
115         eingabe.requestFocus();
116         eingabe.selectAll();
117       }
118     });
119
120   knRein.addActionListener (
121     new ActionListener ()
122     {
123       public void actionPerformed (ActionEvent e)
124       {
125          String ein = eingabe.getText();
126          String aus = "???";
127          Object Obj = null;
128          eingabe.requestFocus();
129          eingabe.selectAll();
130          if (ein.equals(""))
131          {
132            ausgabe.append ("** Erst Element eingeben, das rein soll! **\n");
133          }
134          else
135          {
136            if (cbZahl.getState())
137            {
138              Obj = new Double (ein);
139              aus = "Zahl "+Obj.toString();
140            }
141            else if (cbZchn.getState())
142            {
143              Obj = new Character (ein.charAt(0));
144              aus = "Zeichen '"+Obj.toString()+"'";
145              eingabe.select(0,1);
146            }
147            else if (cbText.getState())
148            {
149              Obj = ein;
150              aus = "Zeichenkette \""+ein+"\"";
151            }
152            try
153            {
154              datenSpeicher.rein (Obj);
```

```
155               ausgabe.append (""+aus+" gespeichert!\n");
156             }
157             catch (Exception f)
158             {
159               ausgabe.append ("** "+f+"\n");
160             }
161           }
162       }
163     });
164
165
166     knRaus.addActionListener (
167       new ActionListener ()
168       {
169         public void actionPerformed (ActionEvent e)
170         {
171           eingabe.requestFocus();
172           eingabe.selectAll();
173           Object Obj = null;
174           String aus = "???";
175           try
176           {
177             Obj = datenSpeicher.raus();
178             if (Obj instanceof Double)
179             {
180                aus = "Zahl "+Obj;
181             }
182             else if (Obj instanceof Character)
183             {
184                aus = "Zeichen '"+Obj+"'";
185             }
186             else if (Obj instanceof String)
187             {
188                aus = "Zeichenkette \""+Obj+"\"";
189             }
190             ausgabe.append (aus+" entfernt!\n");
191           }
192           catch (Exception f)
193           {
194             ausgabe.append ("** "+f+"\n");
195           }
196         }
197       });
198
199     knZeige.addActionListener (
200       new ActionListener ()
201       {
202         public void actionPerformed (ActionEvent e)
203         {
204           eingabe.requestFocus();
205           eingabe.selectAll();
206           Object Obj = null;
207           String aus = "???";
208           try
209           {
210             Obj = datenSpeicher.zeige();
211             if (Obj instanceof Double)
212             {
213                aus = "Zahl "+Obj;
214             }
215             else if (Obj instanceof Character)
216             {
217                aus = "Zeichen '"+Obj+"'";
218             }
219             else if (Obj instanceof String)
220             {
221                aus = "Zeichenkette \""+Obj+"\"";
222             }
223             ausgabe.append (aus+" ist vorne/oben.\n");
```

```
224           }
225           catch (Exception f)
226           {
227             ausgabe.append ("** "+f+"\n");
228           }
229         }
230       });
231
232     knLeer.addActionListener (
233       new ActionListener ()
234       {
235         public void actionPerformed (ActionEvent e)
236         {
237           eingabe.requestFocus();
238           eingabe.selectAll();
239           // Hier mal kein try-catch, obwohl NullPointerException entsteht,
240           // wenn istLeer?-Knopf gedrückt wird, bevor Keller erzeugt wurde
241           if (datenSpeicher.istLeer())
242           {
243             ausgabe.append ("Der Speicher ist im Moment leer!\n");
244           }
245           else
246           {
247             ausgabe.append ("Der Speicher ist im Moment nicht leer!\n");
248           }
249         }
250       });
251
252   }
253
254   public void führeAus ()
255   {
256     richteFensterEin();
257     richteRadiobuttonsEin();
258     richteKnöpfeEin();
259     ausgabe.setEditable(false); // Ausgabe nicht vom Benutzer änderbar
260     setLayout (new FlowLayout());
261
262     add (knKellerReih);
263     add (knKellerDyn);
264     add (knSchlangeReih);
265     add (knSchlangeDyn);
266     add (knWeitere);
267     add (eingabe);
268     add (new Label(" als"));
269     add (cbZahl);
270     add (cbZchn);
271     add (cbText);
272     add (knRein);
273     add (new Label(""));
274     add (knLeer); add (new Label(""));
275     add (knZeige); add (new Label(""));
276     add (knRaus); add (new Label(""));
277     add (ausgabe);
278     setVisible(true);
279   }
280 }
```

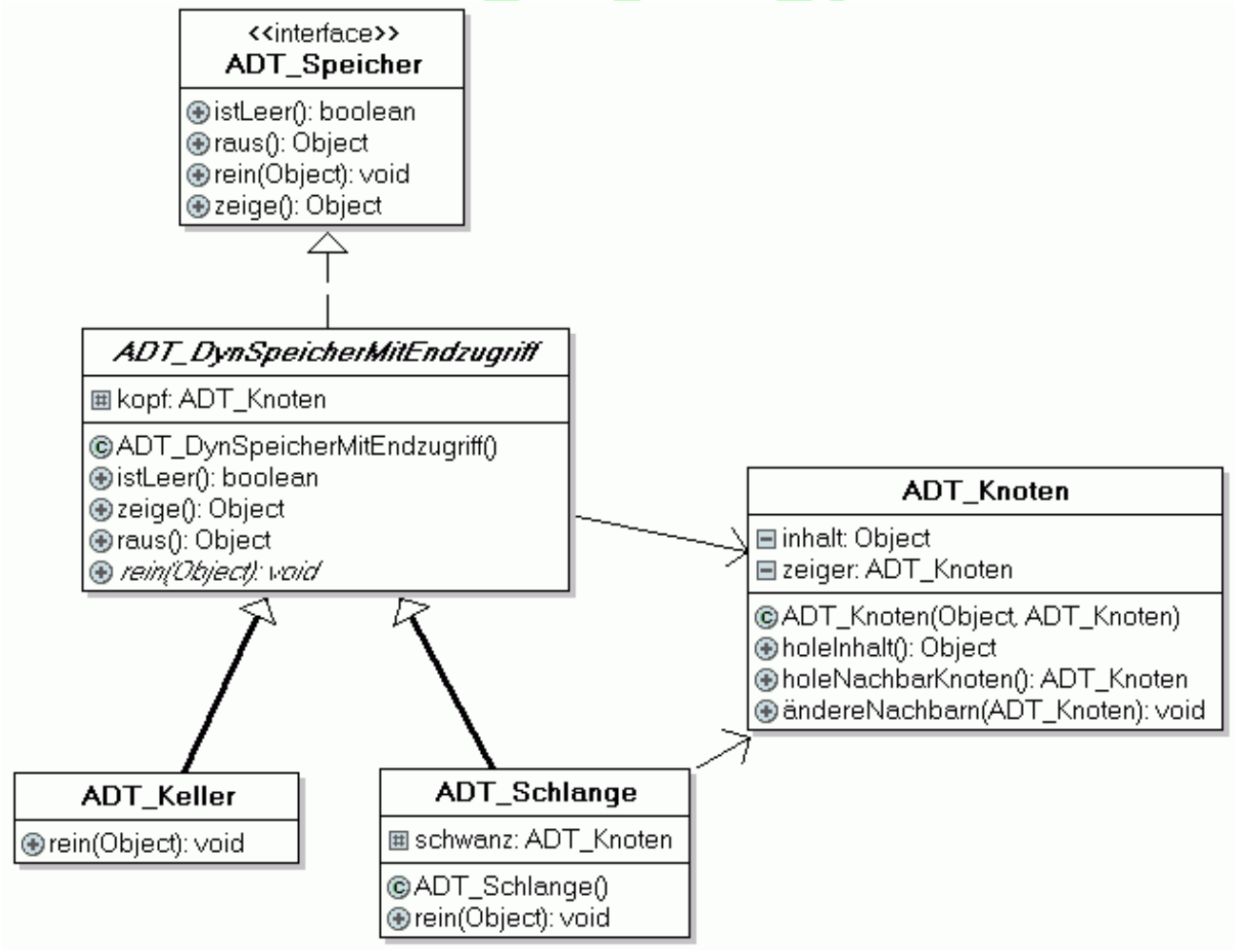
Auf der nächsten Seite folgt noch das UML-Diagramm für die Oberfläche und die drei bereits im Haupttext vorgestellten Datentyp-Implementationen (Keller1 ist der statische Keller, Schlange1 gibt's noch nicht, sondern soll in Aufgabe 16 erstellt werden). Es können natürlich auch die Keller und Schlangen aus dem folgenden Anhang 2 verwendet werden.


Frame
<<interface>>
ADT_Speicher
istLeer(): boolean
raus(): Object
rein(Object): void
zeige(): Object
ADT_Keller1
reihung: Object[]
kopf: int
MAX: int
ADT_Keller1()
rein(Object): void
istLeer(): boolean
zeige(): Object
raus(): Object
ADT_Oberflaeche_Start
main(String[]): void
ADT_Oberflaeche
eingabe: TextField
ausgabe: TextArea
cbg: CheckboxGroup
cbZahl: Checkbox
cbZchn: Checkbox
cbText: Checkbox
knKellerReih: Button
knSchlangeReih: Button
knKellerDyn: Button
knSchlangeDyn: Button
knWeitere: Button
knRein: Button
knRaus: Button
knZeige: Button
knLeer: Button
datenSpeicher: ADT_Speicher
richteFensterEin(): void
richteRadiobuttonsEin(): void
richteKnöpfeEin(): void
führeAus(): void
ADT_Keller
kopf: ADT_Knoten
ADT_Keller()
istLeer(): boolean
zeige(): Object
raus(): Object
rein(Object): void
ADT_Knoten
inhalt: Object
zeiger: ADT_Knoten
ADT_Knoten(Object, ADT_Knoten)
holeInhalt(): Object
holeNachbarKnoten(): ADT_Knoten
ändereNachbarn(ADT_Knoten): void
ADT_Schlange
schwanz: ADT_Knoten
ADT_Schlange()
rein(Object): void

Anhang 2: Keller und Schlange „als Geschwister“

Nach erfolgreicher Implementation von Keller und Schlange wie im Haupttext beschrieben und einigen weiteren Übungen und Anwendungen können auch im Unterricht in einer Wiederholungsphase die hier bereits auf Seite 11 angesprochenen Bedenken eingebracht oder aufgegriffen werden: Sachlogisch sind Keller und Schlange gleichberechtigte, in Teilen verschiedene Datentypen mit vielen Gemeinsamkeiten. Die Schlange als Erbe des Kellers zu betrachten, hat zwar die Programmierung erleichtert, ist aber inhaltlich nicht unbedingt gerechtfertigt. Das kann Anlass sein, rückblickend die Implementation nochmal zu ändern und als zusätzliche Übung und Vertiefung beide Strukturen von einem gemeinsamen Vorgänger abzuleiten – einer abstrakten (Ober-)Klasse für Datentypen, wo der Zugriff nur an den Enden des linear gedachten Speichers erfolgen kann.

Eine abstrakte Klasse kann bereits fertig programmierte Methoden enthalten (die den Nachfolgern gemeinsam sind [oder ggf. dort überschrieben werden, was hier allerdings nicht sinnvoll wäre]) und abstrakte Methoden, die (wie in einem Interface) noch ohne Rumpf sind und in den Nachfahren (Erben) ausprogrammiert werden müssen. Von einer abstrakten Klasse können keine Objekte erzeugt werden – nur von ihren Nachfahren. Das auf Seite 6 eingeführte Interface *ADT_Speicher* wird beibehalten, ebenso die Knoten wie auf Seite 9 dargestellt, weil hier zunächst mit der dynamischen Variante begonnen wird.



Hier die Java-Texte der Dateien einer möglichen dynamischen Implementation:

```
1  public abstract class ADT_DynSpeicherMitEndzugriff implements ADT_Speicher
2  // Nachtrag: Abstrakte Oberklasse für dynam. Keller und dynam. Schlange
3  {
4    protected ADT_Knoten kopf; // Verankerung der Struktur
5
6    public ADT_DynSpeicherMitEndzugriff() // Konstruktor
7    {
8      kopf = null;
9    }
10
11   public boolean istLeer() // sagt, ob der Speicher leer ist
12   {                        // genau dann wahr, wenn der Speicher leer ist
13     return (kopf == null);
14   }
15
16   public Object zeige()    // zeigt/kopiert oberstes Element, ohne Speicher zu ändern
17   {
18     return (kopf.holeInhalt());
19   }
20
21   public Object raus()     // zeigt oberstes El. und entfernt es aus dem Speicher
22   {
23     Object altesElement = kopf.holeInhalt();
24     kopf = kopf.holeNachbarKnoten();
25     return (altesElement);
26   }
27
28   public abstract void rein (Object neuesElement);
29   // übergebenes Element kommt in Speicher
30   // Methode wird von Keller und Schlange verschieden implementiert!
31 }
```

Der Keller ist ein Nachfahre dieser abstrakten Oberklasse

```
1  public class ADT_Keller extends ADT_DynSpeicherMitEndzugriff
2  // Nachtrag: Implementation, die von abstrakter Oberklasse erbt und nur rein neu
3  schreibt
4  {
5    public void rein (Object neuesElement) // neu (anstelle abstrakter Klasse)
6    {
7      ADT_Knoten neu = new ADT_Knoten (neuesElement, kopf);
8      kopf = neu;
9    }
10 }
```

und die Schlange ebenfalls:

```
1  public class ADT_Schlange extends ADT_DynSpeicherMitEndzugriff
2  // Nachtrag: Implementation, die von abstrakter Oberklasse erbt und rein neu schreibt
3  // und zusätzlich den schwanz einführt
4  {
5    protected ADT_Knoten schwanz; // kopf existiert schon durch abstrakte Oberklasse
6
7    public ADT_Schlange ()
8    {
9      super();        // Konstruktor der abstr. Oberkl. ADT_DynSpeicherMitEndzugriff
10     schwanz = null; // zusätzlich
```

```
  }

  public void rein (Object neuesElement)  // neu (anstelle abstrakter Klasse)
  {
    ADT_Knoten neu = new ADT_Knoten (neuesElement, null);
    if (istLeer())
    {
      kopf = neu;   //1. Element: auch kopf muss auf den neuen Knoten zeigen
    }               //hierfür darf kopf höchstens 'protected', nicht 'private' sein.
    else
    {
      schwanz.ändereNachbarn(neu);  // Vorgänger muss auf den neuen Knoten zeigen
    }
    schwanz = neu;  //immer zeigt auch der Schwanz auf den letzten Knoten
  }
}
```

Diffiziler ist die statische Variante: Will man die neuen Elemente auch hier am Schwanz der Schlange einfügen und den Kopf für *raus* und *zeige* beibehalten, so ist innerhalb der Reihung das rechte Ende des belegten Teils der Kopf, während links der Schwanz ist. Da bisher die Belegung der Reihung beim Index 0 angefangen hat (vgl. Seite 6, rechte Zeichnung), müsste das erste neue Element an die Position -1, wo die Reihung gar nicht definiert ist. Denkt man sich jedoch das vorderste mit dem hintersten Feld der Reihung verbunden, d.h. identifiziert man den Index -1 mit *MAX*-1 und schreibt dorthin das erste Element, so kriecht die Schlange quasi immer von rechts (Index *MAX*-1) nach links (Index 0) durch die Reihung, um dann wieder hinten aufzutauchen. Die Schlange ist leer, wenn irgendwo der *kopf* den *schwanz* ein- (noch ein letztes Element in der Schlange) bzw. überholt (leere Schlange), d.h. der Kopf um eine Position links vom Schwanz bzw. *(kopf+1) % MAX == schwanz* ist. Das bedeutet übrigens auch, dass man nur MAX-1 Felder der Reihung nutzen kann, denn die mit MAX Elementen völlig gefüllte Schlange ließe sich wegen gleicher relativer Lage von *kopf* und *schwanz* nicht von der leeren Schlange unterscheiden. Um möglichst viel Gemeinsamkeiten zu finden, muss schon der Keller an diese Idee angepasst werden.

```
public abstract class ADT_StatSpeicherMitEndzugriff implements ADT_Speicher
// Gemeinsame Oberklasse für Keller und Schlange
{
  protected final int MAX = 100; // stat. Größe;  MAX-1 = Maximale Zahl der Elemente
  protected Object[] reihung;
  protected int kopf;

  public ADT_StatSpeicherMitEndzugriff()   // Konstruktor für neuen, leeren Speicher
  {
    reihung = new Object[MAX]; // Anlegen der statischen Datenstruktur
    kopf = MAX-1;              // wegen raus; reihung[MAX-1] nicht benutzbar
  }

  public abstract void rein (Object element); // Element kommt in Speicher
   // abtrakte Klasse: Wird von Keller und Schlange verschieden implementiert

  public abstract boolean istLeer();          // sagt, ob der Speicher leer ist
   // abtrakte Klasse: Wird von Keller und Schlange verschieden implementiert

  public Object zeige()        // zeigt oberstes Element ohne Ändern des Speichers
  {
    return (reihung[kopf]);
  }
```

```
24
25    public Object raus()          // zeigt oberstes El. und löscht es aus dem Speicher
26    {
27      Object hilf = reihung[kopf];
28      kopf = (kopf+MAX-1) % MAX;  // reduziert kopf um 1. Nur nach 0 kommt wieder MAX-1
29      return (hilf);
30    }
31  }
```

Und damit gilt dann für den statischen Keller:

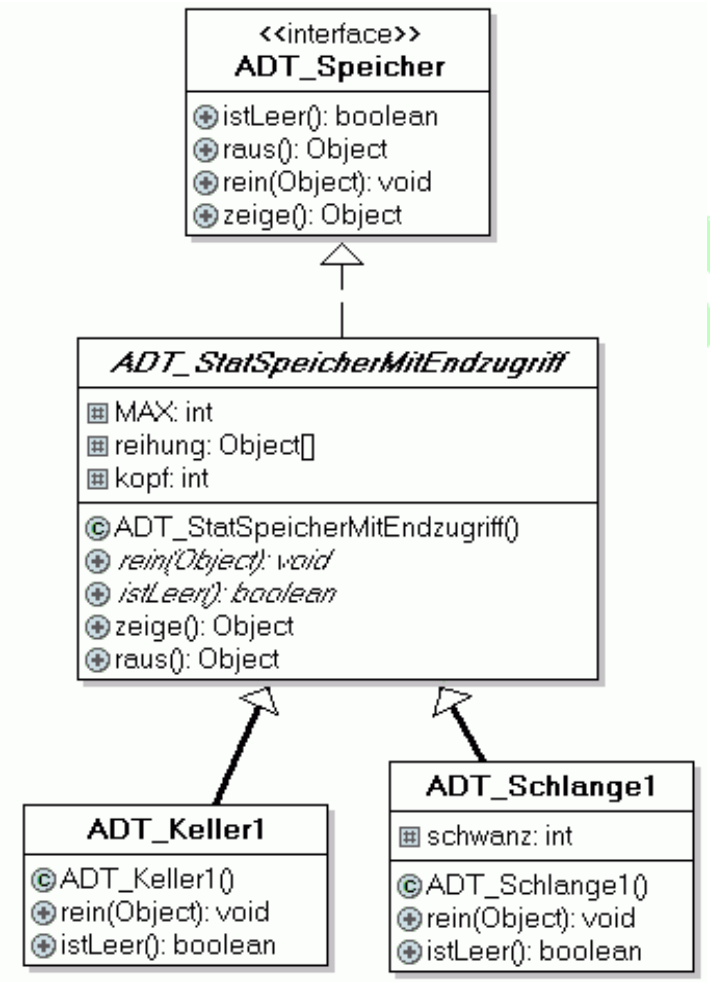
```
1  public class ADT_Keller1 extends ADT_StatSpeicherMitEndzugriff
2  // Keller als Erbe der abstrakten Oberklasse mit zusätzl. Definitionen
3  {
4
5    public ADT_Keller1()          // (eigentl. überflüssiger) Konstruktor
6    {                             // für neuen, leeren Keller
7      super();
8    }
9
10   public void rein (Object element) // übergebenes Element kommt in Speicher
11   {
12     kopf = (kopf+1) % MAX;      // statt kopf++ nötig wegen Startwert von kopf
13     reihung[kopf] = element;    // das letzte eingekellerte (=oberste) Element
14   }                             // steht am Index kopf
15
16   public boolean istLeer()      // sagt, ob der Speicher leer ist
17   {
18     return (kopf == MAX-1);     // vgl. Konstruktor und raus (aus Oberklasse)
19   }
20 }
```

und für die statische Schlange:

```
1  public class ADT_Schlange1 extends ADT_StatSpeicherMitEndzugriff
2  // Schlange als Erbe der abstrakten Oberklasse mit zusätzl. Definitionen
3  {
4    protected int schwanz;
5
6    public ADT_Schlange1()        // Konstruktor für neue, leere Schlange
7    {
8      super();
9      schwanz = 0;                // weil noch nichts da ist, steht der schwanz noch
10   }                             // 'rechts' vom künftigen 1. El. bei -1 = MAX-1
11
12   public void rein (Object element) // übergebenes Element kommt in Speicher
13   {
14     schwanz = (schwanz+MAX-1) % MAX;  // Neue Elemente stehen links von vorhandenen,
15     reihung[schwanz] = element;// das letzte eingeschlangte (=jüngste) Element
16   }                             // steht am Index schwanz
17
```

```
18    public boolean istLeer()     // sagt, ob der Speicher leer ist
19    {
20      return ((kopf+1)%MAX == schwanz);  // schwanz überholt kopf gerade
21    }
22  }
```

Die Zusammenhänge werden durch das nachfolgende UML-Diagramm nochmal etwas klarer:



Natürlich können diese Keller und Schlangen ohne weiteres mit der in Anhang 1 vorgestellten Oberfläche verwendet und getestet werden: nach außen bzw. gegenüber der Oberfläche verhalten sich die unterschiedlichen Implementationen ja völlig gleich!

Anhang 3: Typisierte Keller und Schlangen (ab Java 1.5)

Im Haupttext war ziemlich zu Beginn (Seite 4) eher am Rande vermerkt worden, dass hier Keller und Schlange für beliebige Objekte erstellt würden. Die Datenspeicher wurden mit einem Universalkühlschrank vergleichen, der praktischerweise verschiedenartige Lebensmittel aufnehmen kann. Dies verlangt natürlich vom Anwender, dass er die Übersicht über die eingekellerten oder eingeschlangten Objekte behält. Während ein menschlicher Kühlschrankbenutzer leicht sieht, zu welcher Art das entnommene Lebensmittel gehört, tut sich der Computer bzw. ein Programm da schwerer. Möglicherweise wird zur Laufzeit ein entnommenes Käsestück irrtümlich für ein Objekt vom Typ Weinflasche gehalten. Erst beim falschen Type-Casting bzw. beim Versuch, den Korkenzieher in den Käse einzudrehen, entsteht dann ein Fehler.

Jedenfalls konnten bisher verschiedenartige Objekte durchaus bunt gemischt in die gleiche Datenstruktur eingefügt werden: Dies findest seinen Niederschlag auch in der im Anhang 1 gezeigten Oberfläche, wo im abgebildeten Fall Objekte unterschiedlicher Art in die gleiche Datenstruktur eingereiht werden (vgl. Seite 16). Abfragen mit *instanceof* (Seite 19, Zeilen 178 bis 193 sowie etwas später noch Zeilen 211 bis 222) sorgen für die richtige Interpretation bei der Entnahme – zumindest, solange nur Objekte der drei abgefragten Typen eingelagert wurden, was allerdings nirgends garantiert wird.

Bei normalen Variablen prüft schon der Compiler während des Programmierens und damit vor dem Start des Programms streng, ob die im Programmtext vorgesehenen Zuweisungen und Operationen zu dem deklarierten Variablentyp passen (und schützt so vor bösen Überraschungen irgendwann während der Laufzeit eines vermeintlich richtigen Programms). Deshalb sind in Java und in vielen anderen Programmiersprachen ja auch nicht so allgemeine Variablen-Deklarationen wie etwa in JavaScript erlaubt, wo man als Typ jeder Variablen einfach ‘var’ angibt und diesen erst zur Laufzeit durch die erste Zuweisung bzw. durch den Inhalt festlegt.

Mit der Java-Version (1.)5 wurde das Konzept der generischen Klassen eingeführt, das für die mitgelieferten Bibliotheksklassen, aber auch für die selbstgeschrieben Klassen eine ausdrückliche Typisierung erlaubt, die schon vom Compiler geprüft werden kann und wird. Entscheidet man sich dafür, muss man allerdings tatsächlich verschiedene Kühlschränke für verschiedene Lebensmittelsorten bzw. verschiedene Keller oder Schlangen für Inhalte unterschiedlicher Typen benutzen. Wie schon erwähnt (Seite 5), wird der gewünschte (Objekt-)Typ in spitzen Klammern zum Klassennamen hinzugefügt. Eine Parameterisierung mit primitiven Typen wie *int* oder *char* ist nicht möglich (wohl aber mit *Integer* oder *Character*). Einzelheiten findet man in neueren Büchern über „Java 5“ bzw. zur JDK 1.5 oder z.B. auch auf den (englischsprachigen) Webseiten des Java-Herstellers Sun, insbesondere auf http://java.sun.com/j2se/1.5.0/docs/guide/language/generics.html bzw. in der Anleitung http://java.sun.com/j2se/1.5/pdf/generics-tutorial.pdf. Im Folgenden zeige ich die Realisierung dieses Konzepts beispielhaft durch entsprechend angepasste Klassen, die in der einfachen (nicht typsicheren) Form bereits im Anhang 2 vorgestellt wurden, sodass der direkte Vergleich der Quelltexte beider Konzepte leicht möglich ist.

Zunächst wird im allgemeinen Interface nochmal festgelegt/versprochen, welche Methoden jeder Datentyp haben soll.

```
// Teil des Projekts typisierte lineare abstrakte Datentypen -- nur ab Java (1.)5
// R. Krell (www.r-krell.de) -- 21.12.2006 -- Java JDK 1.5+ nötig

public interface TADT_Speicher<Elementtyp>
// legt nur fest, welche Methoden jeder Datenspeicher haben muss
{
  public void rein (Elementtyp element); // packt das Element in den Speicher
  public boolean istLeer();   // sagt, ob der Speicher leer ist
  public Elementtyp zeige();  // zeigt ein Element ohne Ändern des Speichers
  public Elementtyp raus();   // zeigt ein Element und löscht es aus dem Speicher
}
```

Wie im Anhang 2 beginne ich bei der Implementation mit der dynamischen Variante, wobei zuerst die Knoten definiert werden:

```
// Teil des Projekts typisierte lineare abstrakte Datentypen -- nur ab Java (1.)5
// R. Krell (www.r-krell.de) -- 21.12.2006 -- Java JDK 1.5+ nötig

public class TADT_Knoten <Elementtyp>      // = rote Plastikwanne
{
  private Elementtyp inhalt;               // Fach für den Inhalt
  private TADT_Knoten<Elementtyp> zeiger;  // "Kartentasche" für den Verweis

  public TADT_Knoten (Elementtyp neuesElement, TADT_Knoten<Elementtyp> nachbar)
  {                                        // Konstruktor
    inhalt = neuesElement;
    zeiger = nachbar;
  }

  public Elementtyp holeInhalt()
  {
    return (inhalt);
  }

  public TADT_Knoten<Elementtyp> holeNachbarKnoten()
  {
    return (zeiger);
  }

  public void ändereNachbarn (TADT_Knoten<Elementtyp> neuerNachbar) // für Schlange
  {
    zeiger = neuerNachbar;
  }
}
```

Mit solchen Knoten wird jetzt die abstrakte Oberklasse erstellt, von der danach der dynamische Keller ebenso wie die dynamische Schlange abgeleitet werden.

```
1  // R. Krell (www.r-krell.de) -- 21.12.2006 -- Java JDK 1.5+ nötig
2
3  public abstract class TADT_DynSpeicherMitEndzugriff<Elementtyp>
4                                          implements TADT_Speicher<Elementtyp>
5  // Abstrakte Oberfläche für neue Vererbung, jetzt typisiert
6  {
7    protected TADT_Knoten<Elementtyp> kopf; // Verankerung der Struktur
8
9    public TADT_DynSpeicherMitEndzugriff()  // Konstruktor
10   {
11     kopf = null;
12   }
13
14   public boolean istLeer() // sagt, ob der Speicher leer ist
15   {                        // genau dann wahr, wenn der Speicher leer ist
16     return (kopf == null);
17   }
18
19   public Elementtyp zeige()    // kopiert oberstes Element, ohne Speicher zu ändern
20   {
21     return (kopf.holeInhalt());
22   }
23
24   public Elementtyp raus()     // zeigt oberstes El. und entfernt es aus dem Speicher
25   {
26     Elementtyp altesElement = kopf.holeInhalt();
27     kopf = kopf.holeNachbarKnoten();
28     return (altesElement);
29   }
30
31   public abstract void rein (Elementtyp neuesElement); // in Speicher
32   // Methode wird von Keller und Schlange verschieden implementiert!
33
34 }
```

Davon abgeleitet der Keller ...

```
1  // R. Krell (www.r-krell.de) -- 21.12.2006 -- Java JDK 1.5+ nötig
2
3  public class TADT_Keller <Elementtyp> extends
4  TADT_DynSpeicherMitEndzugriff<Elementtyp>
5  // Implementation, die von abstrakter Oberklasse erbt und nur rein neu schreibt
6  {
7    public void rein (Elementtyp neuesElement) // neu (anstelle abstrakter Klasse)
8    {
9      TADT_Knoten<Elementtyp> neu = new TADT_Knoten<Elementtyp> (neuesElement, kopf);
10     kopf = neu;
11   }
12 }
```

... und, ebenfalls direkt von der abstrakten Oberklasse (und nicht vom Keller) abgeleitet, die Schlange:

```
1  // R. Krell (www.r-krell.de) -- 21.12.2006 -- Java JDK 1.5+ nötig
2
3  public class TADT_Schlange<Elementtyp> extends
4  TADT_DynSpeicherMitEndzugriff<Elementtyp>
5  // Implementation, die von abstrakter Oberklasse erbt und nur rein neu schreibt
6  // und zusätzlich den schwanz einführt
7  {
8    protected TADT_Knoten<Elementtyp> schwanz; // kopf existiert schon
9
10   public TADT_Schlange ()
11   {
12     super();         // ** hier kein super<Elementtyp>(); erlaubt !**
13                      // Konstruktor der abstr. Oberkl. TADT_DynSpeicherMitEndzugriff
14                      // aurufen
15     schwanz = null; // zusätzlich
16   }
17
18   public void rein (Elementtyp neuesElement)  // neu (anstelle abstrakter Klasse)
19   {
20     TADT_Knoten<Elementtyp> neu = new TADT_Knoten<Elementtyp> (neuesElement, null);
21     if (istLeer())
22     {
23       kopf = neu;    //1. Element: auch kopf muss auf den neuen Knoten zeigen
24     }                //hierfür darf kopf höchstens 'protected', nicht 'private' sein.
25     else
26     {                                //sonst:
27       schwanz.ändereNachbarn(neu); //Vorgänger muss auf den neuen Knoten zeigen
28     }
29     schwanz = neu;  //immer zeigt auch der Schwanz auf den letzten Knoten
30   }
31 }
```

Damit ist die programmtechnische Realisierung typsicherer dynamischer Keller und Schlangen erfolgreich angeschlossen.

Bei der statischen Variante stört, dass Reihungen (arrays) von parameterisierten Typen zwar definiert, aber nicht erzeugt werden können: So ist zwar *Elementtyp[] reihung* erlaubt, nicht aber *new Elementtyp[100]* möglich – dem Vernehmen nach, weil bestimmte Zuweisungsmöglichkeiten von Reihungen kompatibler Typen Operationen ermöglichen, die die Typsicherheit gefährden könnten. Offenbar muss die Reihung daher wie bisher für Elemente vom allgemeinsten Typ *Object* erzeugt werden. Und dann kann entweder in den beiden Methoden *zeige* und *raus* durch Type-Casting erzwungen werden, dass die der Reihung entnommenen Elemente in den *Elementtyp* umgewandelt werden – oder die ganze Reihung wird wie im Folgenden schon direkt nach dem Erzeugen durch Type-Casting zu einer Reihung vom *Elementtyp[]* umgewandelt. Beide Varianten werden compiliert, aber in beiden Fällen warnt der Java-1.5-Compiler vor unsicherem Code ('unsafe code') bzw. 'unchecked cast', weil das Type-Casting eben erst zur Laufzeit durchgeführt wird und mögliche Fehler nicht schon zur Compilierzeit entdeckt werden. Leider ist eine bessere Möglichkeit nicht in Sicht. (Die erwähnten Prpobleme treffen übrigens auch den Java-Bibliothekstyp *ArrayList*, der ebenfalls nicht mit generischem Typ erzeugt werden kann).

Hier der Programmcode (wobei die anfangs im Kommentar verdeutlichte Behandlung leerer Strukturen natürlich genauso auf die nicht-typisierte Variante aus Anhang 2 zutrifft, und die unterschiedlichen Meldungen im Bild der Oberfläche weiter unten erklären soll):

```
1  // Teil des Projekts typisierte lineare abstrakte Datentypen -- nur ab Java (1.)5
2  // R. Krell (www.r-krell.de) -- 21.12.2006 -- Java JDK 1.5+ nötig
3
4  public abstract class TADT_StatSpeicherMitEndzugriff<Elementtyp>
5                                        implements TADT_Speicher<Elementtyp>
6  // Gemeinsame Oberklasse für Keller und Schlange mit Reihung und Typisierung
7
8  // Achtung: raus aus einem leeren Speicher liefert keinen (ArrayIndexOutOfBounds-) Fehler,
9  // sondern -- da Anfang und Ende der Reihung praktisch verbunden sind -- das Element
10 // an der nächsten (verbundenen) Stelle. Gibt's dort kein Element, wird null geliefert.
11 // Da dadurch kein Fehler bemerkt wird, wird kopf wie bei einer erfolgreichen raus-Operation
12 // verändert, sodass danach der Speicher nicht mehr als leer gilt und istLeer false
13 // liefert. Erst nach Einfügen eines Elements in den Speicher in diesem Zustands gilt dieser
14 // wieder als leer! Deshalb muss der Aufruf von raus bei leerem Speicher unbedingt vom
15 // Anwender vermieden werden!
16
17 {
18   protected final int MAX = 100; // stat. Größe;  MAX-1 = Maximale Zahl der Elemente
19   protected Elementtyp[] reihung ;  // Definition mit Elementtyp problemlos
20   protected int kopf;
21
22   public TADT_StatSpeicherMitEndzugriff()   // Konstruktor für neuen, leeren Speicher
23   {
24     reihung = (Elementtyp[]) new Object [MAX]; // Anlegen der statischen Datenstruktur.
25       // Es ist allerdings nicht möglich, new Elementtyp[MAX] zu erzeugen. Es muss der
26       // Umweg über new Object[MAX] und den TypeCast gewählt werden, der beim Compilieren
27       // zur Warnung "unchecked cast" führt, weil der Cast möglicherweise erst zur
28       // Laufzeit misslingt. Hier klappt's natürlich und die Warnung muss ignoriert werden.
29     kopf = MAX-1;               // wegen raus;
30   }
31
32   public abstract void rein (Elementtyp element); // Element kommt in Speicher
33    // Methode wird von Keller und Schlange verschieden implementiert
34
35   public abstract boolean istLeer();          // sagt, ob der Speicher leer ist
36    // Methode wird von Keller und Schlange verschieden implementiert
37
38   public Elementtyp zeige()      // zeigt oberstes Element ohne Ändern des Speichers
39   {
40     return (reihung[kopf]);
41   }
42
43   public Elementtyp raus()       // zeigt oberstes El. und löscht es aus dem Speicher
44   {
45     Elementtyp hilf = reihung[kopf];
46     kopf = (kopf+MAX-1) % MAX;  // reduziert kopf um 1. Nur nach 0 kommt wieder MAX-1
47     return (hilf);
48   }
49 }
```

Und, von der vorstehenden abstrakten Oberklasse abgeleitet, wieder der Keller (dessen Programmtext keinen Anlass zu weiteren Compiler-Warnungen gibt):

```
1  // R. Krell (www.r-krell.de) -- 21.12.2006 -- Java JDK 1.5+ nötig
2
3  public class TADT_Keller1<Elementtyp>
4                          extends TADT_StatSpeicherMitEndzugriff<Elementtyp>
5  // Keller als Erbe der abstrakten Oberklasse mit zusätzl. Definitionen
6  {
7
8    public TADT_Keller1()  // Konstruktor für neuen, leeren Keller
9    {
10     super();             // hier kann und darf kein Typ übergeben werden, wie etwa
11                          // super<Elementtyp>();
12                          // Wird offenbar automatisch von Java richtig gemacht.
13   }
14
15   public void rein (Elementtyp element) // übergebenes Element kommt in Speicher
16   {
17     kopf=(kopf+1)%MAX;   // der Keller wächst zwar nur von 0 an nach rechts,
18                          // sodass kopf++ scheinbar reicht. Aber der leere Keller
19                          // beginnt wegen ererbtem raus nicht bei -1, sondern
20                          // bei MAX-1, sodass beim Schritt vom leeren Keller
21                          // zum Keller mit einem Element diese Rechnung nötig ist.
22     reihung[kopf] = element;   // das letzte eingekellerte (=oberste) Element
23   }                      // steht am Index kopf
24
25   public boolean istLeer()     // sagt, ob der Speicher leer ist
26   {
27     return (kopf == MAX-1);    // vgl. Konstruktor und raus (aus Oberklasse)
28   }
29 }
```

Und auch die statische Schlange wird direkt von der abstrakten Oberklasse abgeleitet und ist somit ein „Geschwister“ des statischen Kellers:

```
1  // R. Krell (www.r-krell.de) -- 21.12.2006 -- Java JDK 1.5+ nötig
2
3  public class TADT_Schlange1<Elementtyp>
4                              extends TADT_StatSpeicherMitEndzugriff<Elementtyp>
5  // Schlange als Erbe der abstrakten Oberklasse mit zusätzl. Definitionen
6  {
7    protected int schwanz;
8
9    public TADT_Schlange1()       // Konstruktor für neue, leere Schlange
10   {
11     super();
12     schwanz = 0;                // weil noch nichts da ist, steht der schwanz noch
13   }                             // 'rechts' vom künftigen 1. El. bei -1 = MAX-1
14
15   public void rein (Elementtyp element) // übergebenes Element kommt in Speicher
16   {
17     schwanz = (schwanz+MAX-1) % MAX;  // Neue Elemente stehen links von vorhandenen
18     reihung[schwanz] = element;   // das letzte eingeschlangte (=hinterste) Element
19   }                             // steht am Index schwanz
20
```

```
21   public boolean istLeer()     // sagt, ob der Speicher leer ist
22   {
23     return ((kopf+1)%MAX == schwanz);  // schwanz überholt kopf gerade
24   }
25 }
```

Der Test der typisierten Datenstrukturen erfordert natürlich auch ein Anpassen der Testoberfläche. Augenfällig ist im nachstehendem Bild, dass die früher (s. Seite 16) beim einzufügenden Element vorzunehmende Typwahl nach oben gewandert ist und jetzt vor dem Erzeugen der entsprechenden Datenstruktur ausgeführt werden muss (und dann für die Lebensdauer der Datenstruktur gilt): Elemente können nur noch typrichtig aufgenommen werden – wozu im folgenden Quelltext allerdings nicht nur die Typisierung der Datenstrukturen sorgt, sondern auch die in der Variable *elementTypNr* gespeicherte Wahl, die beim Betätigen der Methoden-Schaltflächen verwendet wird. Auch intern kommt man nicht mehr mit einem allgemeinen Datenspeicher aus, sondern muss für die vorgesehenen Typen verschiedene speziell typisierte Datenspeicher anlegen. Nur noch, ob der Datenspeicher durch Knopfdruck zur Laufzeit als Keller oder als Schlange (oder als neue, vom Interface *TADT_Speicher* abgeleitete Datenstruktur) erzeugt wird, bleibt – zusammen mit der Art der Implementation – offen.

Und hier – weil nicht unmittelbar zur Implementation der abstrakten Datentypen gehörend, wieder ohne Rahmen – der vollständige Programmtext von Startdatei

```
1 public class TADT_Oberflaeche_Start
2 {
3   public static void main (String[] s)
4   {
5     TADT_Oberflaeche test = new TADT_Oberflaeche();
6     test.führeAus();
7   }
8 }
```

und Oberfläche:

```
1 // Teil des Projekts typisierte lineare abstrakte Datentypen -- nur ab Java (1.)5
2 // R. Krell (www.r-krell.de) -- 21.12.2006 -- Java JDK 1.5+ nötig
3
4 import java.awt.*;
```

```
import java.awt.event.*;

public class TADT_Oberflaeche extends Frame
// Oberfläche zum Testen der typisierten Keller und Schlangen;
// Die Compiler-Warnung, dass diese Klasse keine serialVersionUID hat,
// kann ignoriert werden.
{
  TextField eingabe = new TextField("",8);
  TextArea ausgabe = new TextArea("Zuerst einen Elemettyp wählen, dann LinADT
                                   erzeugen!\n",10,95);
  CheckboxGroup cbg = new CheckboxGroup();
  Checkbox cbZahl, cbZchn, cbText;
  Button knKellerReih, knSchlangeReih,
    knKellerDyn, knSchlangeDyn,
    knRein, knRaus, knZeige, knLeer; // Knöpfe definieren

  final int NDEF = 0;       // nicht definiert: noch kein Elementtyp ausgewählt
  final int ZAHL = 1;
  final int ZCHN = 2;
  final int TEXT = 3;
  int elementTypNr = NDEF;  // merkt sich den Elementtyp des erzeugten Datenspeichers
                            // &dient später zur Auswahl des richtigen Datenspeichers

  TADT_Speicher<Double> datenSpeicherZahl;     // Zwar wäre es hier noch möglich,
                                               // statt drei
  TADT_Speicher<Character> datenSpeicherZchn;  // spezieller Speicher einen allgemein
                                               // typisierten
  TADT_Speicher<String> datenSpeicherText;     // TADT_Speicher<?> datenSpeicher zu
                                               // definieren,
    // aber in diesen könnten später keine konkreten Elemente
    // eines bestimmten Typs mit rein eingefüllt werden: der Compiler meldet Fehler
    // wegen nichtübereinstimmender Typen!

  private void richteFensterEin() // Fenster initalisieren und beschreiben
  {
    //WindowsListener hinzufügen, damit Schließknopf funktioniert
    addWindowListener (
      new WindowAdapter ()
      {
        public void windowClosing (WindowEvent ereignis)
        {
          setVisible (false);
          dispose();
          System.exit(0);
        }
      }
    ); // runde Klammer vom Windowlistener geschlossen;

    setTitle("Testprogramm für typisierte lineare abstrakte Datentypen
              -- www.r-krell.de"); // Fenster mit Titel versehen
    setSize (720,278); // Fenstergröße (Breite und Höhe in Pixeln) festlegen
  }

  private void richteRadiobuttonsEin()
  {
    cbZahl = new Checkbox ("Zahl", cbg, false);
    cbZchn = new Checkbox ("Zeichen", cbg, false);
    cbText = new Checkbox ("Text", cbg, true);
  }

  private int frageRadiobuttonsAb()
  {
    int ergebnis = NDEF;
    if (cbZahl.getState())
    {
      ergebnis = ZAHL;
    }
    else if (cbZchn.getState())
    {
```

```
74        ergebnis = ZCHN;
75      }
76      else if (cbText.getState())
77      {
78        ergebnis = TEXT;
79      }
80      return (ergebnis);
81    }
82
83    private void richteKnöpfeEin()
84    {
85      // Knöpfe erzeugen:
86      knKellerReih     = new Button ("Keller (stat., mit Reihung)");
87      knKellerDyn      = new Button ("Keller (dyn., mit Knoten)");
88      knSchlangeReih   = new Button ("Schlange (stat.)");
89      knSchlangeDyn    = new Button ("Schlange (dyn.)");
90      knRein           = new Button ("rein");
91      knRaus           = new Button ("raus");
92      knZeige          = new Button ("zeige");
93      knLeer           = new Button ("istLeer ?");
94
95      //Funktion der Knöpfe festlegen:
96
97      knKellerReih.addActionListener (
98        new ActionListener ()
99        {
100         public void actionPerformed (ActionEvent e)
101         {
102           String aus = "** Probleme beim Anlegen des neuen stat. typisierten Kellers
103                         mit Reihung **";
104           elementTypNr = frageRadiobuttonsAb(); // Der gewählte Typ wird global
105             // gemerkt und dient sowohl hier zur Auswahl des richtigen Speichers
106             // als auch später bei rein, raus und zeige zur richtigen Umwandlung
107             // der Eingabe bzw. richtigen Interpretation des Elements
108           switch (elementTypNr)
109           {
110             case ZAHL: datenSpeicherZahl = new TADT_Keller1<Double>();
111                        aus = "neuer, leerer stat. Keller mit Reihung für Kommazahlen
112                              angelegt"; break;
113             case ZCHN: datenSpeicherZchn = new TADT_Keller1<Character>();
114                        aus = "neuer, leerer stat. Keller mit Reihung für
115                              Schriftzeichen angelegt"; break;
116             case TEXT: datenSpeicherText = new TADT_Keller1<String>();
117                        aus = "neuer, leerer stat. Keller mit Reihung für
118                              Zeichenketten angelegt"; break;
119           }
120           ausgabe.append (aus+"\n");
121           eingabe.requestFocus(); // aktviert die Eingabezeile,
122            // sodass künftige Tastatureingaben automatisch in der eingabe landen
123           eingabe.selectAll(); // macht evtl. Text in der Eingabezeile blau,
124            // sodass alter Text automatisch überschrieben wird
125         }
126       });
127
128     knKellerDyn.addActionListener (
129       new ActionListener ()
130       {
131         public void actionPerformed (ActionEvent e)
132         {
133           String aus = "** Probleme beim Anlegen des neuen dyn. typisierten Kellers
134                         mit Knoten **";
135           elementTypNr = frageRadiobuttonsAb();
136           switch (elementTypNr)
137           {
138             case ZAHL: datenSpeicherZahl = new TADT_Keller<Double>();
139                        aus = "neuer, leerer dyn. Keller mit Knoten für Kommazahlen
140                              angelegt"; break;
141             case ZCHN: datenSpeicherZchn = new TADT_Keller<Character>();
142                        aus = "neuer, leerer dyn. Keller mit Knoten für Schriftzeichen
```

```
                              angelegt"; break;
            case TEXT: datenSpeicherText = new TADT_Keller<String>();
                       aus = "neuer, leerer dyn. Keller mit Knoten für Zeichenketten
                              angelegt"; break;
          }
          ausgabe.append (aus+"\n");
          eingabe.requestFocus();
          eingabe.selectAll();
        }
      });

    knSchlangeReih.addActionListener (
      new ActionListener ()
      {
        public void actionPerformed (ActionEvent e)
        {
          String aus = "** Probleme beim Anlegen der neuen stat. typisierten Schlange
                        mit Reihung **";
          elementTypNr = frageRadiobuttonsAb();
          switch (elementTypNr)
          {
            case ZAHL: datenSpeicherZahl = new TADT_Schlange1<Double>();
                       aus = "neue, leere stat. Schlange mit Reihung für Kommazahlen
                              angelegt"; break;
            case ZCHN: datenSpeicherZchn = new TADT_Schlange1<Character>();
                       aus = "neue, leere stat. Schlange mit Reihung für
                              Schriftzeichen angelegt"; break;
            case TEXT: datenSpeicherText = new TADT_Schlange1<String>();
                       aus = "neue, leere stat. Schlange mit Reihung für
                              Schriftzeichen angelegt"; break;
          }
          ausgabe.append (aus+"\n");
          eingabe.requestFocus();
          eingabe.selectAll();
        }
      });

    knSchlangeDyn.addActionListener (
      new ActionListener ()
      {
        public void actionPerformed (ActionEvent e)
        {
          String aus = "** Probleme beim Anlegen der neuen dyn. typisierten Schlange
                        mit Knoten **";
          elementTypNr = frageRadiobuttonsAb();
          switch (elementTypNr)
          {
            case ZAHL: datenSpeicherZahl = new TADT_Schlange<Double>();
                       aus = "neue, leere dyn. Schlange mit Knoten für Kommazahlen
                              angelegt"; break;
            case ZCHN: datenSpeicherZchn = new TADT_Schlange<Character>();
                       aus = "neue, leere dyn. Schlange mit Knoten für Schriftzeichen
                              angelegt"; break;
            case TEXT: datenSpeicherText = new TADT_Schlange<String>();
                       aus = "neue, leere dyn. Schlange mit Knoten für Zeichenketten
                              angelegt"; break;
          }
          ausgabe.append (aus+"\n");
          eingabe.requestFocus();
          eingabe.selectAll();
        }
      });

    knRein.addActionListener (
      new ActionListener ()
      {
        public void actionPerformed (ActionEvent e)
        {
          String ein = eingabe.getText();
```

```
212         if (!ein.equals(""))
213         {
214           try
215           {
216             String aus="???";
217             switch (elementTypNr)  // Die Eingabe wird in den Typ des
218                                    // Datenspeichers verwandelt.
219             {
220               case ZAHL: datenSpeicherZahl.rein(Double.parseDouble(ein));
221                          aus = "Die Zahl "+ein; break;
222               case ZCHN: datenSpeicherZchn.rein(ein.charAt(0)); aus="Das Zeichen '"
223                          +ein.charAt(0)+"'"; break;
224               case TEXT: datenSpeicherText.rein(ein); aus="Die Zeichenkette \""
225                          +ein+"\""; break;
226             }
227             ausgabe.append (aus+" wurde eingefügt.\n");
228           }
229           catch (Exception f)
230           {
231             ausgabe.append ("** Fehler beim Versuch, \""+ein+"\" einzufügen -- "
232                             +f+"\n");
233           }
234         }
235         else
236         {
237           ausgabe.append ("Erst gesuchtes Element eingeben!\n");
238         }
239         eingabe.requestFocus();
240         eingabe.selectAll();
241       }
242     });
243
244
245     knRaus.addActionListener (
246       new ActionListener ()
247       {
248         public void actionPerformed (ActionEvent e)
249         {
250           eingabe.requestFocus();
251           eingabe.selectAll();
252           Object el = null;
253           String aus = "???";
254           try
255           {
256             switch (elementTypNr)  // Das el wird entsprechend dem Datenspeichertyp
257                                    // interpretiert
258             {
259               case ZAHL: el = datenSpeicherZahl.raus(); aus = "Zahl "+el; break;
260               case ZCHN: el = datenSpeicherZchn.raus(); aus = "Zeichen '"+el+"'";
261                          break;
262               case TEXT: el = datenSpeicherText.raus(); aus = "Zeichenkette \""
263                          +el+"\""; break;
264             }
265             if (el !=null)
266             {
267               ausgabe.append (aus+" entfernt!\n");
268             }
269             else
270             {
271               ausgabe.append ("** Kein Element zum Entfernen vorhanden/gefunden!
272                                **\n");
273             }
274           }
275           catch (Exception f)
276           {
277             ausgabe.append ("** "+f+"\n");
278           }
279         }
280       });
```

```
281
282     knZeige.addActionListener (
283       new ActionListener ()
284       {
285         public void actionPerformed (ActionEvent e)
286         {
287           eingabe.requestFocus();
288           eingabe.selectAll();
289           Object el = null;
290           String aus = "???";
291           try
292           {
293             switch (elementTypNr)   // Das el wird enstprechend dem Datenspeichertyp
294                                     // interpretiert
295             {
296               case ZAHL: el = datenSpeicherZahl.zeige(); aus = "Zahl "+el; break;
297               case ZCHN: el = datenSpeicherZchn.zeige(); aus = "Zeichen '"+el+"'";
298                          break;
299               case TEXT: el = datenSpeicherText.zeige(); aus = "Zeichenkette \""
300                          +el+"\""; break;
301             }
302             if (el !=null)
303             {
304               ausgabe.append (aus+" ist vorne/oben.\n");
305             }
306             else
307             {
308               ausgabe.append ("** Kein Element zum Anzeigen vorhanden/gefunden!
309                                **\n");
310             }
311           }
312           catch (Exception f)
313           {
314             ausgabe.append ("** "+f+"\n");
315           }
316         }
317       });
318
319     knLeer.addActionListener (
320       new ActionListener ()
321       {
322         public void actionPerformed (ActionEvent e)
323         {
324           eingabe.requestFocus();
325           eingabe.selectAll();
326           try
327           {
328             boolean leer=true;
329             switch (elementTypNr)   // Auswahl nötig, weil es drei versch.
330                                     // Datenspeicher gibt.
331             {
332               case ZAHL: leer = datenSpeicherZahl.istLeer(); break;
333               case ZCHN: leer = datenSpeicherZchn.istLeer(); break;
334               case TEXT: leer = datenSpeicherText.istLeer(); break;
335             }
336             if (leer)
337             {
338               ausgabe.append ("Der Speicher ist im Moment leer!\n");
339             }
340             else
341             {
342               ausgabe.append ("Der Speicher ist im Moment nicht leer!\n");
343             }
344           }
345           catch (Exception f)
346           {
347             ausgabe.append ("** "+f+"\n");
348           }
349         }
```

```
350       });
351
352   }
353
354   public void führeAus ()
355   {
356     richteFensterEin();
357     richteRadiobuttonsEin();
358     richteKnöpfeEin();
359     ausgabe.setEditable(false); // Ausgabe nicht vom Benutzer änderbar
360     setLayout (new FlowLayout());
361
362     add (cbZahl);
363     add (cbZchn);
364     add (cbText);
365     add (knKellerReih);
366     add (knKellerDyn);
367     add (knSchlangeReih);
368     add (knSchlangeDyn);
369     add (new Label("Element:"));
370     add (eingabe);
371     add (knRein); add (new Label(""));
372     add (knLeer); add (new Label(""));
373     add (knZeige); add (new Label(""));
374     add (knRaus); add (new Label(""));
375     add (ausgabe);
376     setVisible(true);
377   }
378 }
```

Ob die generische Variante allerdings direkt zur Einführung der abstrakten Datentypen bei Schülerinnen und Schülern taugt, scheint mir fraglich. Vielleicht wird der Einfachheit halber doch besser herkömmlich begonnen und diese Variante nur gelegentlich – vielleicht per Schülerreferat – nachgetragen.

Die angegebenen Quelltexte können auch als .java-Dateien vom Autor bezogen werden, der sich außerdem über Kritik und Anmerkungen per Mail freut:

Robert Krell
krell@web.de