Abstrakte Datentypen, zunächst: Keller

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 Lagerhallen 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 Speicherns kümmern muss.

Abstrakte Datentypen sind solche Datenspeicher, auf die nur mittels weniger festgelegter Methoden zugegriffen werden kann, und wo die interne Verwaltung vorm Benutzer verborgen bleibt.

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.zeige1()* 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 *Datentyp speicher = new Datentyp()* definiert und erzeugt, d.h. neu und leer angelegt. Danach ist der Zugriff nur mit *rein, istLeer, raus* oder *zeige1* möglich (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 hier die gleichen deutschen Namen verwendet werden sollen). 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 *zeige1* 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.

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.

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 solche 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, *zeige1* 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 bei der Implementation, 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 Beschreibungen 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*: ▮K × ▮E → ▮K mit *rein*: (*k, e*) ↦ *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 Beschreibung passt eher zur prozeduralen Denkweise, weniger zur objektorientierten.

- 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 (Stapel, LIFO-Speicher) (stack)

Konstruktor	*new Keller()*	(new, create)
Vorher	./.	
Nachher	Es wurde ein neuer, leerer Keller erzeugt	
Auftrag	*rein (Object neuesElement) : void*	(push, add, put)
Vorher	Es existiert ein Keller	
Nachher	Das Objekt neuesElement wurde dem Keller ‘oben’ hinzugefügt	
Anfrage	*istLeer() : boolean*	(empty, 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	*zeige1() : Object* (peek, top, get, look)
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* (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 (wie oben beschrieben) oder entfernt es, ohne es nochmal zu nennen. 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 *zeige1*), sodass hier so implementiert werden soll. Auch wird hier darauf verzichtet, innerhalb von *raus* oder *zeige1* automatisch zu prüfen, ob der Keller vielleicht leer ist (und notfalls *null* statt eines Fehlers auszugeben) -- das soll der Anwender mit *istLeer* machen. Manche Implementation enthalten hingen die zusätzliche Kontrolle.
Ob wie oben der Ergebnistyp wie im UML-Diagramm und in Pascal nachgestellt wird, oder vielleicht auch javaartig zuerst genannt wird, sollte ebenfalls letztlich egal sein. Man sollte die Beschreibung in verschiedenen Varianten verstehen können, also auch in der mir etwas besser gefallenden Version wie etwa:

Anfrage	*boolean istLeer()*
Vorher	Es existiert ein Keller
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 Datentyps gefordert wird, aber keine weiteren Voraussetzungen (wie „nicht-leer“) nötig sind.

Ein Wort noch zum verwendeten Elementtyp: Oben 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. Im Interesse der Typsicherheit sollte man statt *Object* aber besser den jeweiligen, speziellen Elementtyp angeben. Dann könnte Java schon beim Kompilieren prüfen, ob der Keller nur richtig verwendet wird. Auch dafür braucht nur eine Klasse geschrieben werden (mit dem Platzhalter *Elementtyp* für den späteren Elementtyp), der man beim Definieren und Erzeugen eines Objekts zur Laufzeit des Programms dann in spitzen Klammen den konkreten Elementtyp übergibt

Und schließlich noch ein weiterer Hinweis: Wenn man sicherstellen will, dass alle Keller- (und auch die künftigen Schlangen-)Implementationen wirklich die 4 oben genannten Methoden in gleicher Schreibweise verwirklichen, kann man dies über ein Java-Interface garantieren, dass dann mit *implements* zur Kontrolle aufgerufen wird.