Grundlegende Datentypen

Motivation

- Szenario: arbeitsteilige Entwicklung großer Software-Systeme im Team.
- Zerlegung solcher Software-Systeme in logisch und funktional zusammengehörige Module (oder: Komponenten).
- Module besitzen Schnittstellen bestehend aus Signaturen der Operationen und Zusicherungen (nicht im Java interface) über diese Operationen, Zweck:
  - Verwendungssicht
    - Andere können Modul allein bei Kenntnis der Schnittstelle benutzen.
    - Kenntnis der Implementierung ist nicht erforderlich.
  - Implementierungssicht
    - Modul kann allein bei Kenntnis der Schnittstelle implementiert werden.
    - Kenntnis seiner Verwendungen nicht erforderlich.
- Frage: Wie kann die Schnittstelle eines (zu entwickelnden) Moduls (eines Typs) ohne Angabe einer konkreten Implementierung vorab so spezifiziert werden, dass dessen korrekte Verwendung bzw. Implementierung möglich sind?
- Lösung: Spezifikation als sog. Abstrakter Datentyp (ADT)
  - Festlegung der auf den Typ anwendbaren Operationen (Schnittstelle)
  - Festlegung der Wirkung der Operationen (Zusicherungen)
- Vorteile:
  - Nutzende Komponenten können sich auf die Spezifikation der Wirkung verlassen, ohne die Implementierung zu kennen, sog. Design by Contract.
  - Die Implementierung einer Komponente ist für die Nutzer geheim. Sie kann ausgetauscht/verbessert werden, ohne dass die Nutzer betroffen sind, solange die spezifizierte Wirkung der Operationen erhalten bleibt, sog. Geheimnisprinzip

Ein erstes Beispiel: Menge ganzer Zahlen

- Aufgabe: Verwalte eine Menge ganzer Zahlen so, dass Zahlen eingefügt und gelöscht werden können und der Test auf Enthaltensein durchgeführt werden kann.
- Ebene der Spezifikation:
  - Abstrakteste Sicht eines Moduls: es gibt Objekte bestimmter Sorte, auf die Operationen anwendbar sind.
  - Ggfs. können sogar mehrere Sorten von Objekten eine Rolle spielen.
  - Operationen erzeugen aus gegebenen Objekten dieser Sorten neue Objekte, die ebenfalls zu einer der Sorten gehören.
- Im Beispiel:
  - Sorten: Mengen ganzer Zahlen, ganze Zahlen selbst, Wahrheitswerte.
  - Operationen: Zahlen einfügen und löschen, Test auf Enthaltensein.
- System bestehend aus Objektsorten mit dazugehörigen Operationen bezeichnet man als Datentyp.

Festlegung der Syntax: Signatur

- Um Datentyp zu beschreiben, muss man festlegen:
  - wie die Objektsorten und Operationen heißen,
  - wie viele und was für Objekte die Operationen als Argumente benötigen und
  - welche Sorte von Objekt sie als Ergebnis liefern.
- Rein syntaktischer Aspekt, wird festgelegt durch sog. Signatur:

adt $IntSet$

sorts $IntSet, Int, Boolean$

ops

$create$:		$\rightarrow IntSet$
$insert$:	$IntSet \times Int$	$\rightarrow IntSet$
$delete$:	$IntSet \times Int$	$\rightarrow IntSet$
$contains$:	$IntSet \times Int$	$\rightarrow Boolean$
$isEmpty$:	$IntSet$	$\rightarrow Boolean$

Kein Argument, liefert stets ein leeres $IntSet$; 0-stellige Funktion wird auch als ***Konstante*** bezeichnet.

Festlegung der Semantik/Wirkung von Operationen: Axiome

- Spezifikation als Abstrakter Datentyp (ADT) Beschreibung der Semantik/Wirkung von Operationen durch Angabe interessierender Aspekte der Wirkungsweise, Gesetze oder Axiome genannt.
- Gesetze: Gleichungen über Ausdrücken, die mit Hilfe der Operationssymbole gebildet werden; darin vorkommende Variablen sind implizit allquantifiziert
-

axs

$$
\begin{aligned}
&isEmpty(create) && = true\\
&isEmpty(insert(x,i)) && = false\\
&insert(insert(x,i),i) && = insert(x,i)\\
&contains(create,i) && = false\\
&contains(insert(x,j),i) && = \begin{cases} true & falls\ i = j \\ contains(x,i) & sonst \end{cases}
\end{aligned}
$$

end $IntSet$

Für alle x aus $IntSet$ und für alle i aus Int stehen links und rechts vom = gleiche Objekte. Intuitiv: mehrfaches Einfügen von i verändert die Menge x nicht.

Bewertung der Spezifikation als Abstrakter Datentyp

- Vorteile
  - Man kann Datentyp genau soweit festlegen, wie gewünscht, ohne z. B. Implementierungsdetails vorwegzunehmen.
  - Die Sprache, in der Axiome beschrieben werden, folgt formalen Regeln. Daher sind Entwurfswerkzeuge konstruierbar, die Spezifikation prüfen oder Prototyp erzeugen können.
- Nachteile
  - Bei komplexen Anwendungen wird die Zahl der Gesetze sehr groß.
  - Es ist nicht immer leicht,
  - anhand der Gesetze die intuitive Bedeutung eines Datentyps zu erkennen,
  - eine Datenstruktur anhand von Gesetzen zu charakterisieren; insbesondere ist es schwierig zu prüfen, ob die Menge der Gesetze vollständig ist.

Weitere ADT-Begriffe

- Konstruktoren: mit create und insert lassen sich alle möglichen Objekte des Typs *I*ntSet erzeugen. Beide sind zwingend erforderlich.
- Hilfskonstruktoren: erzeugen auch Datenobjekte des Typs; für Konstruktion der Objekte aber nicht zwingend erforderlich.
- Normalform eines Datenobjekts: konstruiert durch eine minimale Zahl von Konstruktoraufrufen (keine Hilfskonstruktoren).

Implementierung des ADTs IntSet

Interface der öffentlichen Operationen des ADTs *I*ntSet

```
import static java.lang.System.arraycopy;
import static java.util.Arrays.copyOf;
public interface IntSet {
        public void insert(int elem);
        public void delete(int elem);
        public boolean contains(int elem);
        public boolean isEmpty();
}
```

- Eine mögliche Implementierung als Klasse MyIntSet:
  - Verwendung von Arrays, arraycopy und copyOf.
  - create wird durch Konstruktor realisiert.
  - insert erzeugt größeres Array und fügt Element hinten an.
  - delete lässt weg und „klebt“ Teil-Arrays (vorher/danach) zusammen

```
class MyIntSet implements IntSet {
        private int[] is = new int[0];
        //public MyIntSet() { }
        public void insert(int elem) {
                if (!contains(elem)) {
                        is = copyOf(is, is.length + 1);
                        is[is.length - 1] = elem;
                }
        }
        public boolean contains(int elem) {
                for (int i = 0; i < is.length; i++)
                        if (is[i] == elem) return true;
                return false;
 }
```

- Konstruktor realisiert zusammen mit leerem Array is die im ADT spezifizierte empty-Funktionalität.
- Mengen enthalten keine Duplikate, deshalb nur einfügen, wenn elem nicht schon enthalten.

```
        public boolean isEmpty() {
                return is.length == 0;
        }
        public void delete(int elem) {
                if (contains(elem)) {
                        int[] isNew = new int[is.length - 1];
                        for (int p = 0; p < is.length; p++) {
                                if (is[p] == elem) {
                                        arraycopy(is, 0, isNew, 0, p);
                                        arraycopy(is, p+1, isNew, p, is.length-1-p);
                                        is = isNew;
                                        return;
} } } } }
```

Behälterdatentypen

- Aufgrund ihrer großer praktischer Relevanz betrachten wir im Folgenden Behälterdatentypen (engl. container types)
  - Datentypen zur Speicherung, Organisation und Verwaltung von Objekten eines Elementdatentyps.
  - Typ der Elemente im Behälter wird zum Parameter des ADTs.
  - Beispiele: Verwaltung von Daten in beliebigen Anwendungskontexten

Statische Datenstrukturen

- Um mit Mengen von Datenelementen umzugehen, haben wir bislang ein- oder mehrdimensionale Arrays benutzt.
- Array ist ein Beispiel für sog. statische Datenstrukturen, d. h. Größe der Struktur wird zum Erstellungszeitpunkt festgelegt und kann dann zur Laufzeit nicht mehr geändert werden.
- Nachteile:
  - Falls das Array zu klein ist, muss man wie in obigem Beispiel Code zum Anlegen größerer Arrays und zum Kopieren in diese schreiben. Das verlangsamt die Laufzeit.
  - Falls das Array zu groß ist, wird unnötig Platz verschwendet; dadurch werden evtl. andere Programme behindert, d. h. diese können nicht laufen oder laufen langsam. (Oben: Erzeugung eines kleineren Arrays und langsames Umkopieren.)
  - Falls Elemente in bestimmter Reihenfolge gehalten werden sollen, dann erfordert das Einfügen oder Löschen von Elementen in einem Array Kopierarbeit. (Oben: unsortierte Menge.)

Dynamische Datenstrukturen

- Anderer Ansatz:
  - Verkettung von Datenelementen, die jeweils ihren direkten Nachfolger (und ggf. Vorgänger) kennen (z. B. sog. Liste).
  - Dann ist an beliebiger Stelle Einfügen oder Entfernen von Elementen möglich, ohne dabei andere Elemente verlagern (kopieren) zu müssen.
  - Direkter Zugriff auf das i-te Element ist nicht möglich (im Unterschied zu Arrays, dort Direktzugriff in O(1) möglich). Die Liste muss von vorne beginnend bis zum i-ten Element linear durchlaufen werden, O(n).
- Liste ist Beispiel für sog. dynamische Datenstrukturen
  - können während des Programmablaufs beliebig wachsen oder schrumpfen,
  - enthalten nicht mehr Elemente als notwendig und
  - werden in Java mit Referenztypen implementiert.

Motivation für generische/parametrisierte Klassen

Beispiel: (rudimentärer) Container, der nur einen Wert speichern kann

```
public class Container {
        private int value;
        public int getValue() {
              return value;
        }
        public void setValue(int n){
              value = n;
} }
```

```
public class Container<T> {
        private T value;
        public T getValue() {
              return value;
        }
        public void setValue(T n) {
               value = n;
} }
```

- Im Code wurden alle Vorkommen von int, die Datentyp des zu verwaltenden Wertes angeben, durch T ersetzt.
- Durch Angabe des Typplatzhalters T in spitzen Klammern wird Klasse zur generischen bzw. parametrisierten Klasse

Verwendung generischer/parametrisierter Klassen

- Verwendungsbeispiele:

```
// Containter fuer Integer-Zahlen
Container<Integer> i =  new Container<Integer>();
Integer j = new Integer(3);
i.setValue(j);
// Container fuer Gleitkommazahlen
Container<Double> d =  new Container<Double>();
d.setValue(new Double(0.2));
Double d2 = d.getValue();
System.out.println(d2);
```

- Java verlangt, dass bei der „Instanziierung“ für den Platzhalter ein Klassentyp eingesetzt wird.
- Also für primitive Datentypen Wrapper-Klassen verwenden

Deklaration generischer/parametrisierter Klassen

```
public class MyClass<T, U> {
        private T variable1;
        public U calc(T t, U u) {
              ...
              return u;
        }
}
```

- Mehrere Typparameter werden durch Kommata getrennt angegeben.
- Typparameter können innerhalb der Klasse als Typen von Variablen oder Rückgabewerten oder als Parameter von Methoden genutzt werden
- Typparameter dürfen nicht in statischen Elementen verwendet werden, da verschiedene „Instanzen“ einer Klasse mit unterschiedlichen generischen Typen dennoch dieselben statischen Elemente verwenden.

```
public class MySpecialClass<E> {
        private E f;
        ...

        public void doThis(E h) {
              f = h;
              MySpecialClass<E> g = new MySpecialClass<E>();

              E e = new E();
              f.doThat();
              f.toString();
        }
}
```

- Referenzen auf Objekte vom Typ E erlaubt
- E darf als Typparameter für eine Instanzierung der parametrisierten Klasse verwendet werden
- Konstruktion eines Objekts nicht erlaubt, da Laufzeittyp unbekannt
- Aufruf spezieller Methoden der generischen Klasse nicht erlaubt, da Laufzeittyp unbekannt

```
public class ChildClass<E> extends MySpecialClass<E> {
       MySpecialClass<MyInt> c = new ChildClass<MyInt>();
       MySpecialClass<Object> o = c;
 …
```

- Polymorphie gilt auch für getypte Klassen,aber nicht für den Typparameter selbst
- Konvention für Namen der Typparameter:
    - Einzelne Großbuchstaben
    - T – Typ, K – Key, N – Number, E – Element, V – Value, S, U, V, …

Methoden und Typparameter

```
public class MethodsAndTypes {
        public static <T> Pair<T,T> duplicate(T elem) {
               return new Pair<T,T>(elem, elem);
        }
        public static void method(Container<String> elem) { … }
        public static void method(Container<Double> elem) { … }
}
```

- Typparameter können auch bei Methoden angegeben werden
- Überladen von Methoden, deren Signaturen sich nur im generischen Typ unterscheiden ist nicht erlaubt, da der Typparameter nur zur Übersetzungsaber nicht zur Laufzeit bekannt ist.

Erzeugung generischer Objekte

```
public static void main(String[] args) {
        Container<Integer> c = new Container<Integer>();
        Container<Integer> c2 = new Container<>();
        ComplexContainer<Container<Double>, Integer> c3 =
               new ComplexContainer<>();
}
```

- Sogenannter DiamondOperator <> erlaubt verkürzte Schreibweise

```
public class MyArray<T> {
        private T[] elements;
        public MyArray() {
               elements = new T[1000];
        }
        public void add(T e) { … }
}
```

- MyArray.java:5: error: generic array creation Erzeugung eines generischen Arrays nicht ohne Fehler/Warnungen möglich, da der Typparameter nur zur Übersetzungs- aber nicht zur Laufzeit bekannt ist.

Platzhalter / Beschränkungen

```
static void print(Container<?> c) {
 Object o = c.getValue();
 System.out.println(o);
}
```

- ? ist Platzhalter für beliebige Typen

```
static <T extends Number> double sum(T t1, T t2) {
       return t1.doubleValue() + t2.doubleValue();
}
```

- Number ist Typ-Obergrenze. T ist Unterklasse von Number und stellt damit auf jeden Fall doubleValue() bereit..

```
class Car { … }
class Audi extends Car { … }
class A6 extends Audi { … }
insertAudi(new MyArray<Audi>); //ok
insertAudi(new MyArray<Car>); //ok
insertAudi(new MyArray<A6>); //zu speziell
static void insertAudi(MyArray<? super Audi> array) {
        array.add(new A6());
        array.add(new Car());
}
```

- Da insertAudi als Typ-Untergrenze mit Audi aufgerufen werden könnte, kann nicht sicher in jedem Fall ein Car hinzugefügt werden

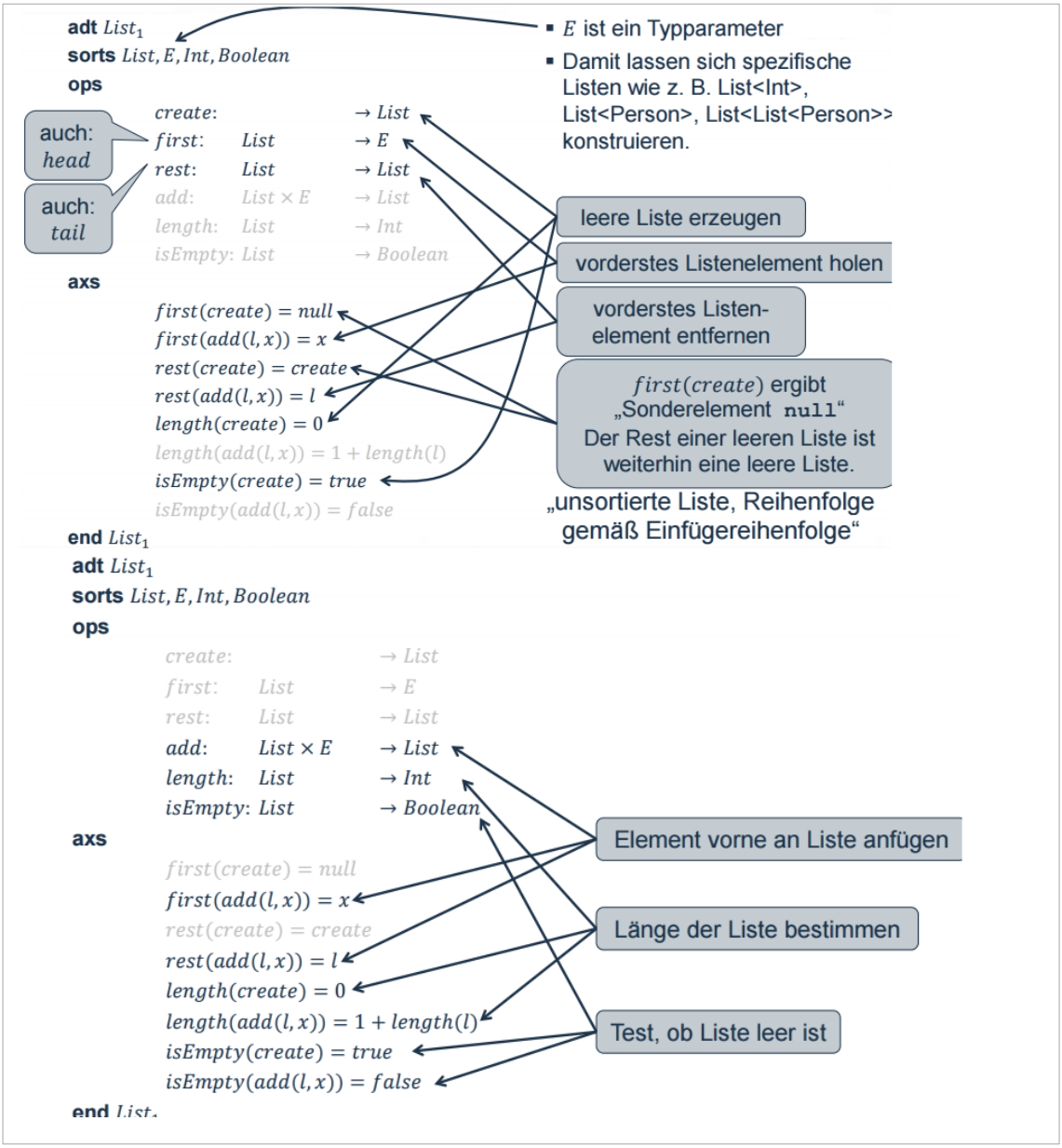
Eigenschaften generischer/parametrisierter Klassen

- Generische bzw. parametrisierte Klassen bieten sich an, wenn man feststellt, dass man mehrere Klassen für verschiedene Typen, aber mit ansonsten identischem Code benötigt.
- Vorteile:
  - Code muss nur einmal geschrieben werden.
  - Platzhalter kann durch einen beliebigen Typ ersetzt werde,der so verwendet werden kann, wie der Platzhalter im Code der generischen/parametrisierten Klasse.

Listen

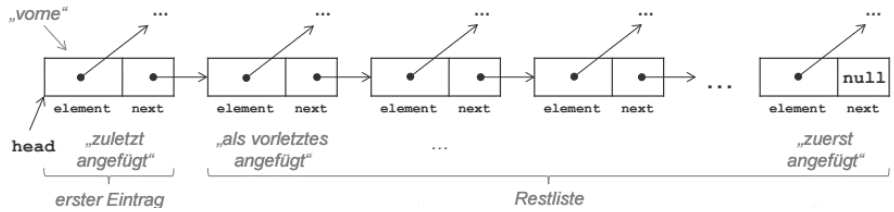
- Listen
  - enthalten endliche Folgen von Objekten eines gegebenen Grundtyps,
  - in einer definierten Ordnung: erstes, zweites … Element.
- Jedes Element hat Vorgänger und Nachfolger (außer dem ersten und letzten Element).
- Listen dürfen Duplikate enthalten.
- Man unterscheidet unsortierte und sortierte Listen (bzgl. einer Ordnung des Grundtyps).
- Listen unterscheiden sich bzgl. der angebotenen Operationen:
  - Hier zunächst elementare (unsortierte) Liste bei der Anfügen und Entfernen nur an einem Ende der Liste möglich ist (ADT $List1$).
  - Später andere Listentypen mit mehr Komfort. (Einfügen und Entfernen an beliebiger Stelle (am Anfang, in der Mitte, am Ende), Sortierung der Listenelemente, und/oder Verketten zweier Listen, Zugriff auf Anfangs- /End-/Teillisten, …)

Elementare Listen mit first, rest, add



Implementierung des ADTs *List*1 mit einfach verketteter Liste

Liste besteht aus verketteten Listeneintrags-Objekten, die jeweils „vorne“ angefügt werden:



Vorderstes Listenelement holen

```
public class List<E> {
        Entry<E> head;
        List() {…}

        public E first() {
              if (head != null)
                      return head.element;
              else
                     return null;
        }
        ...
```

```
class Entry<E> {
        E element;
        Entry<E> next;
}
```

Vorderstes Listenelement entfernen

```
public void rest() {
        if (head != null) {
              head = head.next;
        }
 }
 ...
```

Element vorne an Liste anfügen

```
public List<E> add(E newEl) {
        Entry<E> newEntr = new Entry<E>();
        newEntr.element = newEl;
        newEntr.next = head;
        head = newEntr;
        return this;
 }
 public boolean isEmpty() {
        return (head == null);
 }
```

Länge der Liste bestimmen (nach Entrekursivierung iterativ)

```
//iterative Fassung
 public int length() {
        int len = 0;
        Entry<E> e = head;
        while (e != null) {
              e = e.next;
              len++;
        }
        return len;
 }
}
```

Stapel/Keller (engl. Stack)

- Ein Stapel/Keller (engl. Stack) ist ein Spezialfall der elementaren Liste (die dem Datentyp *List*1 entspricht).
- Operationen werden hier traditionell anders benannt: top = first, push = add, pop = rest, stack = List
- Stapelprinzip (bekannt z.B. von einigen Kartenspielen und von den Methodenschachteln (siehe LE 3)):
  - Man kann nur oben etwas drauflegen oder herunternehmen. Nur oberstes Element ist sichtbar.
  - Zuletzt eingefügtes Element wird als erstes entnommen. Stapel sind sog. last-in-first-out (LIFO) Datenstrukturen.

Abstrakter Datentyp **Stack**

adt *Stack*
sorts *Stack*, *E*, *Boolean*
ops

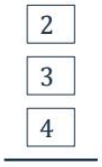
create:		→ *Stack*	← Konstruktoren
push:	*Stack* × *E*	→ *Stack*	← Konstruktoren
pop:	*Stack*	→ *Stack*	← Hilfskonstruktor
top:	*Stack*	→ *E*	← Projektionen
isEmpty:	*Stack*	→ *Boolean*	← Projektionen

axs

$isEmpty(create)$	$= true$
$isEmpty(push(s,e))$	$= false$
$pop(create)$	$= create$
$pop(push(s,e))$	$= s$
$top(create)$	$= null$
$top(push(s,e))$	$= e$

end *Stack*

Beispiel:
$push(push(push(create, 4), 3), 2)$



Implementierung des ADTs Stack mit Liste

- ADT Stack kann leicht mit Implementierungen des Datentyps $\mathit{List}1$ realisiert werden.
- Gleich lautende Operationen der Klasse List werden direkt verwendet.
- Gemäß der Namenskonventionen: top = first, push = add, pop = rest, stack = List werden die zugeordneten Listenoperationen verwendet, um die Stapel-Operationen zu realisieren.
- Grundsätzlich kann man mit jeder korrekt implementierten Liste einen Stapel implementieren.

Klasse java.util.Stack<E> in Java-Standardbibliothek

- Java stellt eine Stack-Implementierung bereit mit geringfügigen Unterschieden in der Benennung und Funktionalität der Methoden:
  - boolean empty() prüft, ob dieser Stack leer ist.
  - E peek() schaut auf das Objekt, das oben auf dem Stapel liegt, ohne es davon zu entfernen (entspricht top() in unserem ADT).
  - E pop() entfernt das oben liegende Objekt vom Stapel und gibt es als Ergebnis zurück (entspricht top() und pop() in unserem ADT).
  - E push(E item) legt das Element auf den Stapel.
  - int search(Object o) liefert die Position des Objekts auf dem Stapel (nicht Bestandteil einer reinen Stapel-Semantik).

(Warte-) Schlangen (engl. Queues)

- (Warte-) Schlangen (engl. Queues) sind spezielle Listen,
  - bei denen Elemente nur an einem Ende („vorne“) entnommen
  - und nur am anderen Ende („hinten“) angehängt werden.
- Schlangen entsprechen unserem Datentyp $\mathit{List}1$
  - ohne die Operation add
  - dafür mit einer Operation append (append),die ein Element ans Ende einer Liste hängt.
- Traditionell gibt es auch hier wieder andere Bezeichnungen:
  - Front = first, enqueue = append, dequeue = rest, Queue = List
- Schlangen sind sog. First-In-First-Out (FIFO) Datenstrukturen.

Abstrakter Datentyp *Queue*

adt *Queue*
sorts *Queue, E, Boolean*
ops

create:		$\rightarrow Queue$
front:	*Queue*	$\rightarrow E$
enqueue:	*Queue* × *E*	$\rightarrow Queue$
dequeue:	*Queue*	$\rightarrow Queue$
isEmpty:	*Queue*	$\rightarrow Boolean$

axs

$$dequeue(create) = create$$

$$dequeue(enqueue(q,x)) = \begin{cases} q & \text{falls } isEmpty(q) \\ enqueue(dequeue(q),x) & \text{sonst} \end{cases}$$

$$front(create) = null$$

$$front(enqueue(q,x)) = \begin{cases} x & \text{falls } isEmpty(q) \\ front(q) & \text{sonst} \end{cases}$$

$$isEmpty(create) = true$$

$$isEmpty(enqueue(q,x)) = false$$

$$\ldots$$

end *Queue*

Normalform:
- Nur *create* und *enqueue* sind Konstruktoren.
- Jede Schlange lässt sich nur mit diesen 2 Operationen erzeugen.

Für nicht-leere Schlange ist es unerheblich, ob man zuerst ein Element anhängt und dann eines wegnimmt oder umgekehrt.

Das vorderste (älteste) Element einer Schlange ist unabhängig davon, ob weitere Elemente mittels *enqueue* angefügt sind.

Implementierung des ADTs Queue mit Liste

- Schlangen werden mit zusätzlichem Zeiger auf das Schlusselement implementiert (für effizientes Anhängen am Ende).
- Leere Schlange: Zeiger auf erstes und letztes Element gleich null.
- Einelementige Schlange: Zeiger auf erstes und letztes Element zeigen auf dasselbe einzige Schlangenelement.
- enqueue:
  - next-Verweis des bisherigen Ende-Elements auf das neue Element zeigen lassen.
  - Dann Ende-Verweis der Schlange auf das neue Element setzen.
- dequeue:
  - Kopf-Verweis auf den Nachfolger des aktuellen Kopfs versetzen.
  - Achtung: Wenn die Schlange nur ein Element hatte, dann zeigt der
    - Ende-Verweis danach auf das entfernte Element.
    - Ende-Verweis muss ebenfalls auf null gesetzt werden

Implementierung des ADTs Queue mit Array

- Array-Implementierung ist eine Alternative bei bekannter maximaler Schlangenlänge.
- Durch Einfügen und Entfernen an verschiedenen Enden „durchwandert“ die Schlange das Array und stößt bald an einem Ende an, obwohl Gesamtgröße des Arrays durchaus ausreicht (wenn chlange höchstens so viele Elemente enthält, wie Array lang ist).
- Trick:
  - Array als zyklisch auffassen.
  - Wenn Ende erreicht, vorne wieder anfangen, sofern dort noch Plätze frei sind.
- Auf das Array-Element mit Index n-1 folgt logisch das Array-Element mit Index 0.
- Die Implementierung muss sicherstellen, dass die Schlange nicht länger als die maximale Schlangenlänge wird, damit das Ende der Schlange im zyklischen Array nicht den Kopf der Schlange überschreib

Prioritätswarteschlange (engl. Priority Queue)

- Die Elemente der Schlange erhalten zusätzlich eine Priorität.
- Beim Auslesen wird nicht das am längsten in der Schlange befindliche Element (FIFO) geliefert, sondern das Element höchster Priorität, das am längsten in der Schlange war.
- Beispiele:
  - Warteraum in Krankenhaus-Ambulanz: Patienten in akuter Gefahr werden vorgezogen.
  - Betriebssysteme: Prozesse (laufende Programme) besitzen unterschiedliche Priorität und erhalten entsprechend Ressourcen (z. B. CPU-Zeit).
- Signatur des abstrakten Datentyps $PQueue$

create:		$\rightarrow PQueue$
front:	*PQueue*	$\rightarrow (E \times Int)$
enqueue:	*PQueue* × $(E \times Int)$	$\rightarrow PQueue$
dequeue:	*PQueue*	$\rightarrow PQueue$
isEmpty:	*PQueue*	$\rightarrow Boolean$

Hilfsoperation maxprio

- Zusätzlich erforderliche Operation: Maxprio: PQueue → Int
- maxprio gehört nicht zur Signatur der Prioritätswarteschlange, weil kein Benutzer der Prioritätswarteschlange den Wert von maxprio kennen muss.
- Implementierung in Java: maxprio als private Methode vereinbaren.
- Zugehöriges Axiom:

$$maxprio(enqueue(q,(x,g))) = \begin{cases} g & \text{falls } isEmpty(q) \\ g & \text{falls } maxprio(q) < g \\ maxprio(q) & \text{sonst} \end{cases}$$

- Axiom der Schlange
  - Für eine nicht-leere Schlange ist es unerheblich, ob man zuerst ein Element anfügt und dann eines entfernt oder umgekehrt.

$$dequeue(enqueue(q,x)) = \begin{cases} q & \text{falls } isEmpty(q) \\ enqueue(dequeue(q),x) & \text{sonst} \end{cases}$$

- Axiom der Prioritätswarteschlange

$$dequeue(enqueue(q,(x,g))) = \begin{cases} q & \text{falls } isEmpty(q) \\ q & \text{falls } maxprio(q) < g \\ enqueue(dequeue(q),(x,g)) & \text{sonst} \end{cases}$$

- Axiom der Schlange
  - Das vorderste (älteste) Element einer Schlange ist unabhängig davon, ob noch weitere Elemente mittels **enqueue** angehängt wurden

$$front(enqueue(q,x)) = \begin{cases} x & \text{falls } isEmpty(q) \\ front(q) & \text{sonst} \end{cases}$$

- Axiom der Prioritätswarteschlange

$$front(enqueue(q,(x,g))) = \begin{cases} (x,g) & \text{falls } isEmpty(q) \\ (x,g) & \text{falls } maxprio(q) < g \\ front(q) & \text{sonst} \end{cases}$$

Implementierung des ADTs *P*Queue mit einer Liste

- Strategie: Einfügen sortiert nach Priorität, keine Hilfsoperation maxprio
- Abbildung:

$$create_{PQueue} = create_{List_1}$$

$$dequeue(add_{List_1}(q,(x,g))) = q$$

$$front(add_{List_1}(q,(x,g))) = (x,g)$$

$$enqueue(q,(x,g)) = \begin{cases} add_{List_1}(q,(x,g)) & \text{falls } isEmpty(q) \\ add_{List_1}(q,(x,g)) & \text{falls } q = add_{List_1}(r,(y,h)) \wedge h < g \\ add_{List_1}(enqueue(l,(x,g)),(y,h)) & \text{sonst, mit } q = add_{List_1}(l,(y,h)) \end{cases}$$

Falls (x,g) höchste Priorität hat, einfach vorne an Liste anhängen.

Sonst Listenkopf überspringen und in q gemäß Sortierung einfügen.

$$isEmpty_{PQueue} = isEmpty_{List_1}$$

Tafelübung

Einschränkung des Typparameters (Generics)

- mögliche Typen für Typparameter können eingeschränkt werden (Bounds)
  - nur Klassen, die von bestimmter Klasse erben
  - nur Klassen, die bestimmtes Interface implementieren
- Nutzen? Nutzen!
  - stellt „Mindest-Anforderung“ an Typen dar
  - konkrete Typen haben „mindestens“ die entsprechende Schnittstelle
  - erlaubt den Aufruf dieser Methoden, ohne konkreten Typ zu kennen
- in Java: Einschränkung des Typparameters mittels extends auch bei Interfaces...
- Beispiele
  - class Foo < T extends Exception > { /* ... */ }
  - class Bar < T extends Comparable <T > > { /* ... */ }

Statische generische Methoden

Folgender Code kompiliert nicht

```
public class Foo <T > {
        public Foo ( T param ) { /* ... */ }
        public static Foo <T > getFoo ( T param ) {
                if ( param == null ) { /* Fehlerbehandlung */ }
                return new Foo <T >( param );
        }
}
```

Was ist da schiefgegangen?

- bei der Instanziierung wird ein konkreter Typ für T eingesetzt
- Klassenmethoden sind nicht an ein Objekt gebunden
  - keine Instanziierung hat stattgefunden
  - für T existiert kein konkreter Typ

Folgender Code kompiliert nicht

```
public static Foo <T > getFoo ( T param ) { /* ... */ }
```

Lösung

- bei der Methodendeklaration den Typparameter mit angeben
- Compiler kann dann beim Methodenaufruf einen passenden Typ einsetzen

... im Beispiel:

```
public static <T > Foo <T > getFoo ( T param ) { /* ... */ }
// ================================================
Foo < Integer > intFoo = Foo . getFoo (4711);
```

Typlöschung

- Generics wurden erst nachträglich zu Java hinzugefügt
  - dabei wurde Augenmerk auf (Bytecode-)Kompatibilität gelegt
- Java-Laufzeitsystem kennt keine Generics im Typsystem
- deswegen notwendig: Typlöschung (Type Erasure)
  - Typparameter werden durch Object ersetzt („ausradiert“) bzw. bei Bounds durch spezifischere Typen
- explizite Typkonvertierungen werden in den Code eingefügt
- Also...
  - Bei der Übersetzung wird der Code generiert, den wir auch selbst hätten schreiben können („Schlechte Lösung (2)“). Die Typlöschung passiert aber erst nach der Typprüfung im Compiler, deswegen kann Typsicherheit zur Übersetzungszeit garantiert werden

Beispiel für Typlöschung

Code vor Typlöschung

```
class Foo <T > {
        T value ;
        public Foo ( T value ) {
                this . value = value ;
        }
        public T get () {
                return this . value
        }
}
// ===================================
Foo < Integer > f = new Foo < Integer >(13);
Integer v = f . get ();
```

Code nach Typlöschung

```
class Foo {
        Object value ;
        public Foo ( Object value ) {
                this . value = value ;
        }
        public Object get () {
                return this . value
        }
}
// ===================================
Foo f = new Foo (13);
Integer v = ( Integer ) f . get ();
```

Konsequenzen der Typlöschung

- Auf Grund der Typlöschung ist Überladen mit Typparametern nicht möglich.
- Zur Laufzeit stehen keine Typinformationen über Typparameter zur Verfügung. Folgender Code führt deshalb zu einem Fehler zur Übersetzungszeit:

```
class Foo <T > {
        /* ... */
}
// ========================
if ( f instanceof Foo < Integer >) {
        /* ... */
} else if ( f instanceof Foo < String >) {
        /* ... */
} else {
        /* ... */
}
```

Generische Arrays

Folgender Code compiliert nicht

```
class Foo <T > { /* ... */ }
// ===================================
Foo < Integer >[] foos = new Foo < Integer >[1];
```

- Arrays mit generischen Typen sind in Java problematisch...
  - Verwendung würde zu unerwartetem Verhalten führen (s.u.)
  - Grund dafür ist wieder die Typlöschung
- mögliche Lösungen sind eher Hacks als saubere Lösungen...
  - Compiler-Warnungen ignorieren/unterdrücken
  - „Hilfsklassen“ ohne Typparameter erstellen und verwenden

Unerwartetes Verhalten

Beispiel ohne Generics

```
String [] strings = new String [1];
Integer [] ints = new Integer [1];
ints [0] = 42; // Autoboxing von int nach Integer
Object [] objects = strings ;
objects [0] = ints [0]; // -> ArrayStoreException
String s = strings [0];
```

Beispiel mit Generics (Annahme: Code würde so übersetzt werden können)

```
ArrayList < String >[] strings = new ArrayList < String >[1];
ArrayList < Integer >[] ints = new ArrayList < Integer >[1];
ints [0] = new ArrayList < Integer >();
ints [0]. add (42); // Autoboxing von int nach Integer
Object [] objects = strings ;
objects [0] = ints [0];
String s = strings [0]. get (0); // -> ClassCastException
```

Erläuterung

- Der Code würde in einer ClassCastException resultieren, sobald man lesend auf ein „falsches“ Element zugreift. Eigentlich erwartet man aber schon beim Abspeichern eine ArrayStoreException.

Mögliche Lösungen

Lösung 1: Compiler-Warnungen unterdrücken

```
@SuppressWarnings (" unchecked ")
Foo < Integer >[] foos = new Foo [1];
```

Lösung 2: „Hilfsklasse“ erzeugen und verwenden

```
class Foo_Integer extends Foo < Integer > {
        /* intentionally left blank */
};
Foo_Integer [] foos = new Foo_Integer [1];
```

Generische Arrays

Folgender Code compiliert nicht

```
class Foo <T > {
        public void bar () {
                T [] array = new T [1];
        }
}
```

- Grund für den Fehler ist erneut die Typlöschung
- T ist zur Laufzeit nicht bekannt
- Java benötigt allerdings Basistyp zum Erzeugen eines Arrays

Mögliche Lösungen

Lösung 1: Compiler-Warnungen beim Zugriff/Cast unterdrücken

```
Object [] array = new Object [1];
@SuppressWarnings (" unchecked ")
T elem = ( T ) array [0];
```

Lösung 2: Compiler-Warnungen bei der Erzeugung unterdrücken (besser!)

```
@SuppressWarnings (" unchecked ")
T [] array = ( T []) new Object [1];
```

Wildcards – Motivation

Szenario

Eine Funktion soll einen Parameter eines generischen Typs haben. Der Funktion ist es aber „egal“, welcher Typparameter tatsächlich verwendet wurde; insbesondere soll der Parameter auf beliebige Typparameter passen.

Wildcards – Motivation

Folgender Code compiliert nicht

```
class Bar <T > { /* ... */ }
class Foo {
        void func1 () {
                Bar < Integer > bInt = new Bar < Integer >();
                func2 ( bInt );
                // func2 ([...]) [...] cannot be applied to func2 ([...])
        }
        void func2 ( Bar < Object > p ) { /* ... */ }
}
```

Wildcards

- das Wildcard-Symbol ? dient als Platzhalter für beliebige Typen
  - damit ist Foo<?> Basistyp „aller Foos“
  - es gehen damit aber auch Typinformationen verloren…
  - ganz grob: Lesen erlaubt, Schreiben verboten
- eine Wildcard kann mit Hilfe von Bounds eingeschränkt werden
  - ? extends Foo
    - beliebiger Typ, aber „mindestens Foo“
  - ? super Foo
    - beliebiger Typ, aber nur Oberklassen von Foo (inklusive Foo)

Lösung für unser Problem

Verwendung einer Wildcard

```
class Bar <T > { /* ... */ }
class Foo {
        void func1 () {
                Bar < Integer > bInt = new Bar < Integer >();
                func2 ( bInt );
        }
        void func2 ( Bar <? > p ) { /* ... */ }
}
```