title: 02 Verkettete Liste permalink: /02-linked-list/

Liste als sequenzielle Datenstruktur

Wiederholung: Felder

Letztes Semester hatten wir Felder (Arrays) kennengelernt (Syntaxelement `[]`) um mehrere Objekte eines Typs abzuspeichern. Felder können sowohl für *primitive Datentypen* als auch für *Referenzdatentypen* (also Objekte von Klassen, bspw. Instanzen von *Auto*) erstellt werden. Wichtig ist dabei, dass die Größe beim Erstellen entweder explizit angegeben werden muss oder durch eine Initialisierung vorgegeben ist.

```
// vier Arten ein Feld mit 3 int-Werten darin zu erzeugen:
int[] zs1 = {1, 2, 3};
int zs2[] = {1, 2, 3};
int[] zs3 = new int [] {1, 2, 3}
int[] zs4 = new int [3];


System.out.println(zs1.length);  // Ausgabe in der Kommandozeile: "3"
System.out.println(zs2.length);  // "3"
System.out.println(zs3.length);  // "3"
System.out.println(zs4.length);  // "3"
```

Bei der Variablendefinition können die `[]` dabei entweder vor den Bezeichner (also zum eigtl. Datentyp) oder hinter den Bezeichner gestellt werden, um die Variable als Array zu definieren. Im obigen Beispiel wurden die Variablen `zs1`, `zs2` und `zs3` dabei *statisch* mit den Werten 1, 2 und 3 initialisiert, `zs4` wurde hingegen mit `new` angelegt, wobei die 3 Werte mit den *Defaultwerten* initalisiert werden; diese sind `false` für Wahrheitswerte, 0 für Zahlenwerte und `null` für Referenzdatentypen.

Der Zugriff erfolgt nun lesend wie schreibend mit dem `[]`-Operator, diesmal zwingend dem Bezeichner nachgestellt. Die Länge eines Feldes ist immer über das von der JVM verwaltete `.length` Attribut zu erfahren. Eine Sonderrolle nimmt die `for-each` Schleife ein, hierbei übernimmt die JVM den Arrayzugriff, der allerdings nur *lesend* sein kann.

```
int[] zs = new int [3];
zs[1] = 1337;
zs[2] = zs[0] - zs[1];

for (int i = 0; i < zs.length; i++)
        System.out.orintln(zs[i]);  // "0 1337 -1337"

for (int z : zs) {
        System.out.println(z);  // "0 1337 -1337"
        z = 5;  // kein Syntaxfehler, aber zs bleibt unverändert!
}
```

Greift man mit dem `[]`-Operator auf einen Index zu, der außerhalb des Arrays liegt, so wird eine `ArrayIndexOutOfBoundsException` geworfen.

Liste als sequenzielle Datenstruktur

Arrays sind geeignet, wenn die Anzahl der Elemente vorher bekannt ist. Was aber, wenn diese vorher unbekannt ist, wie zum Beispiel bei einer Texteingabe für Kurznachrichten? Nutzer sollten hier im Prinzip endlos lange Texteingaben machen können.

Wir brauchen also eine Datenstruktur, welche je nach Bedarf wachsen (und schrumpfen) kann, also eine *Liste*.
Letzte Woche haben wir noch einmal die Grundzüge objektorientierter Programmierung (OOP) wiederholt. Ein weiteres wichtiges Werkzeug der OOP ist die Definition von Schnittstellen, sog. *Interfaces*.
Ein Interface ist ähnlich zu einer Klasse, doch enthält es (im Regelfall) keine Implementierung sondern nur die Methodendefinitionen.

Bleiben wir bei der Liste; diese sollte ein paar grundlegende Operationen beherrschen: das Lesen oder Schreiben von Objekten ab bestimmter Stelle (analog zum Array), sowie das Anhängen und Entfernen von Elementen.
In Java können solche Schnittstellen mit dem Schlüsselwort `interface` definiert werden:

```
interface IntList {
        // entsprechend dem []-Operator:
        int get(int i);
        void put(int i, int v);

        // die Listenlänge betreffend
        void add(int v);
        void remove(int i);

        int length();
}
```

Wichtig ist hierbei, dass die Methoden des Interfaces *keine* Implementierung haben -- die Methodendefinition endet nach der Signatur mit einem Strichpunkt.
Der Hauptvorteil von Schnittstellen ist es, dass diese ohne Wissen über ihre *Implementierung* verwendet werden können:

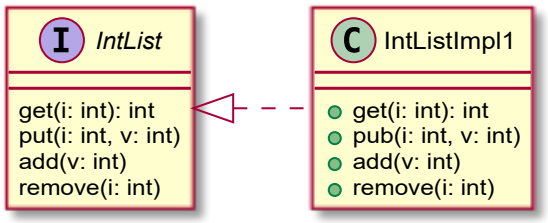
```
IntList li = ???;  // dazu kommen wir gleich!
li.add(11);
li.add(22);
li.add(33);
// li: [11, 22, 33]

System.out.println(li.length()); // 3
System.out.println(li.get(1));   // "22"
li.put(2, 44);  // li: [11, 22, 44]
li.remove(1);   // li: [11, 44]
System.out.println(li.get(1));   // "44"
```

Realisierung und *is-a* Beziehung

Möchte man nun eine Klasse schreiben, welche diese Schnittstelle erfüllt, so verwendet man in der Klassendefinition das Schlüsselwort `implements` und implementiert die vorgeschriebenen Methoden:

```
class IntListImpl1 implements IntList {
        public int get(int i)         { /* TODO */ }
        public void put(int i, int v) { /* TODO */ }
        public void add(int v)        { /* TODO */ }
        public int remove(int i)     { /* TODO */ }
        public void length()          { /* TODO */ }
}
```

Diese *Realisierung* wird in UML mit dem gestrichelten Pfeil sowie einer leeren Dreiecksspitze dargestellt:



Ist `IntListImpl1` eine Realisierung von `IntList`, gilt die *is-a* Beziehung (`IntListImpl1` *is a* `IntList`) und wir können beide wie folgt verwenden:

```
IntList li = new IntListImpl1();
System.out.println(li instanceof IntList);  // "true"

// li verwenden
```

Liste realisiert mit Array

Nun wollen wir die oben skizzierte `IntListImpl1` mit Leben erfüllen, und zwar zunächst mit einem Array. Wir stellen fest:

- Wird eine Liste neu erstellt (Konstruktor), so enthält sie keine Elemente, das Array ist also leer.
- Der lesende bzw. schreibende Zugriff kann direkt auf das Array erfolgen.
- Das Anhängen bzw. Entfernen ist komplizierter, da hier das Array verändert werden müsste: man erstellt also ein neues Array welches eins länger bzw. kürzer ist, und kopiert die Elemente entsprechend.

```
class IntListImpl1 implements IntList {
        private int[] zs;
        public IntListImpl1() { zs = new int [0]; }  // leer.

        public int get(int i) {
                return zs[i];
```

```
        }

        public void put(int i, int v) {
                zs[i] = v;
        }

        public void add(int v) {
                int[] neu = new int [zs.length + 1];
                System.arraycopy(zs, 0, neu, 0, zs.length);
                neu[zs.length] = v;
                zs = neu;
        }

        public int remove(int i) {
                int r = zs[i];
                int[] neu = new int [zs.length - 1];
                for (int j = 0, k = 0; j < zs.length; j++) {
                        if (j == i) continue;
                        neu[k++] = zs[j];
                }
                zs = neu;
                return r;
        }

        public int length() {
                return zs.length;
        }

        public String toString() {
                return Arrays.toString(zs);
        }
}
```

Blockweise Allokation

Die oben gezeigte Implementierung hat eine entscheidende Schwäche: Je länger die Liste, desto größer das Array -- und damit der Aufwand bei beim Einfügen oder Löschen.

Diesem kann man zumindest teilweise entgegenwirken, in dem man das Array *blockweise* allokiert (und bei Bedarf auch wieder freigibt).

Dazu führen wir eine Blockgröße (hier: Konstante `BS`) ein, sowie einen Zähler (hier: `len`), wie weit das Array tatsächlich gefüllt ist. Ist das Array voll, so wird neues allokiert, welches um `BS` größer ist.

```
public class IntListImpl2 implements IntList {
        public static final int BS = 4;  // Blockgröße
        private int[] zs = new int [BS];
        private int len = 0;  // wie viele Felder belegt?

        public int get(int i) {
                if (i >= len) throw new ArrayIndexOutOfBoundsException();
```

```
                return zs[i];
        }
        public void put(int i, int v) {
                if (i >= len) throw new ArrayIndexOutOfBoundsException();
                zs[i] = v;
        }
        public void add(int v) {
                if (len < zs.length) {
                        zs[len++] = v;  // mitzaehlen!
                        return;
                }
                int[] neu = new int [zs.length + BS];
                System.arraycopy(zs, 0, neu, 0, zs.length);
                neu[len++] = v;  // weiterzaehlen!
                zs = neu;
        }
        public int remove(int i) {
                int r = zs[i];
                // ab i alle eins nach links schieben
                for (int j = i+1; j < len; j++)
                        zs[j-1] = zs[j];
                len--;  // mitzaehlen!
                return r;
        }
        public int length() {
                return len;
        }
}
```

Diese Implementierung ist effizienter da über die Blockgröße gesteuert werden kann, wie oft tatsächlich allokiert wird.

> NB: Die Java Klasse `java.util.ArrayList` (link), welche das Interface `java.util.List` implementiert, arbeitet im Prinzip ähnlich: Quelltext im OpenJDK

Verkette Liste

Die beiden vorhergehenden Implementierungen haben zwar einen sehr schnellen Lese- und Schreibzugriff, je nach Anwendungsfall und Blockgröße sind sie aber ineffizient: insbesondere das Löschen eines Elementes (bzw. das Einfügen an beliebiger Stelle) erfordert bei langen Listen das Verschieben von vielen Daten.

Wir treten einen Schritt zurück und betrachten das Problem "Liste" erneut, diesmal unter Gesichtspunkten der Objektorientierung. Wir suchen also ein Modell, bei dem eine Liste beliebig lang sein kann, und bei der man an jeder Stelle ohne großen Aufwand einfügen oder entfernen kann.

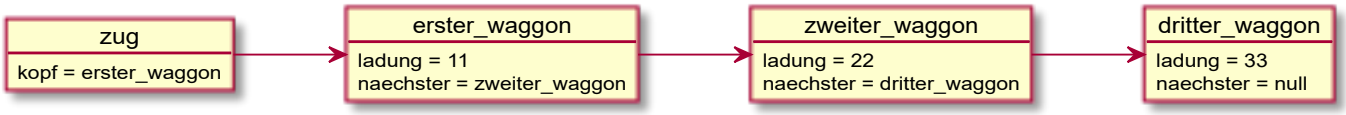
Oft hilft dabei ein Beispiel aus der realen Welt:

Quelle: Wikipedia

Abstrakt betrachtet ist ein Güterzug eine Liste:

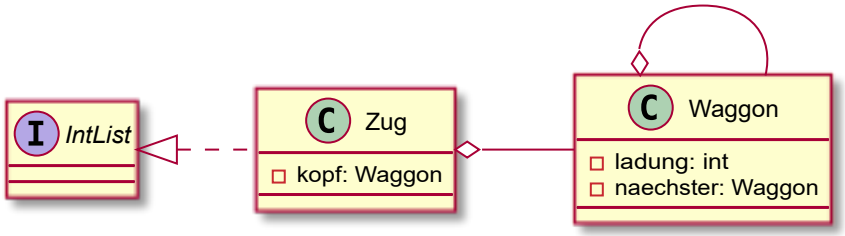
- Fährt die Lok alleine, so ist die Liste leer.
- Man kann problemlos Waggons eingliedern (*einfügen*), ausgliedern (*löschen*) oder anhängen (*hinzufügen*).
- Möchte man z.B. den Inhalt des 3. Waggons, so beginnt man vorne bei der Lok und "hangelt" sich bis zum 3. Waggon durch.

Würde man einen "Güterzug" mit Ladung 11, 22 und 33 als Objektdiagram zeichnen, so könnte das so aussehen:

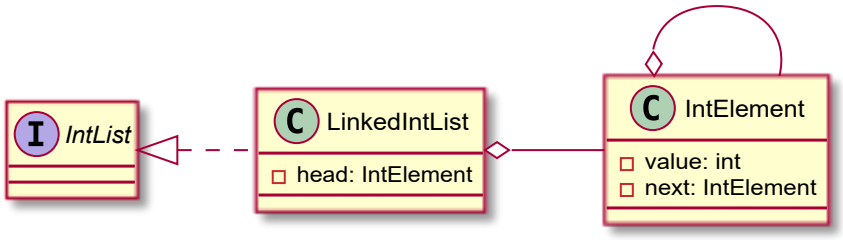


{: .figcenter}

Wir suchen also zunächst eine Klasse, welche die Lok darstellt, sowie eine Klasse welche die Wagons darstellt; die Lok soll dabei eine Liste (hier: `IntList`) sein:



Ein Zug ist also eine Verkettung von Waggons. Da nun nicht alle Nerds Zugfans sind oder Züge allgemein mögen, abstrahieren wir zur *verketteten Liste*:



Wir brauchen also die Klassen `IntElement` und `LinkedIntList`, wobei Letztere das Interface `IntList` implementiert:

```
class IntElement {
        int value;
        IntElement next;
        IntElement(int v, IntElement e) {
                value = v;
                next = e;
        }
}

class IntListImpl3 implements IntList {
        private IntElement head;

        // ... (Fortsetzung unten)
}
```

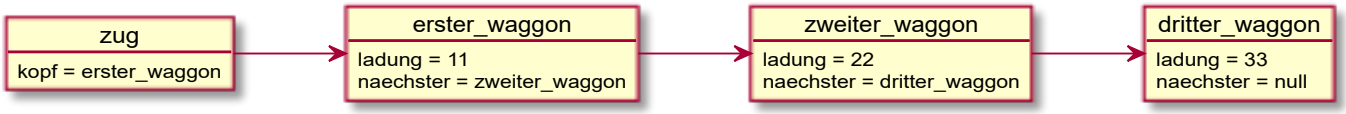
Um nun ein Element anzuhängen, müssen wir zunächst prüfen ob *überhaupt* schon ein *irgendein* Element angehängt ist.
Ist das der Fall, so müssen wir uns bis an das Ende der Liste hangeln, indem wir jeweils zum Nachfolger (`next`) gehen bevor wir anhängen:

```
public void add(int v) {
        // leere Liste: neues Element ist erstes!
        if (head == null) {
                head = new IntElement(v, null);
                return;
        }

        // "laufe" bis zum Ende, d.h. bis it.next == null!
        IntElement it = head;
        while (it.next != null)
                it = it.next;

        // nun das neue Element an das letzte anhängen
        it.next = new IntElement(v, null);
}
```

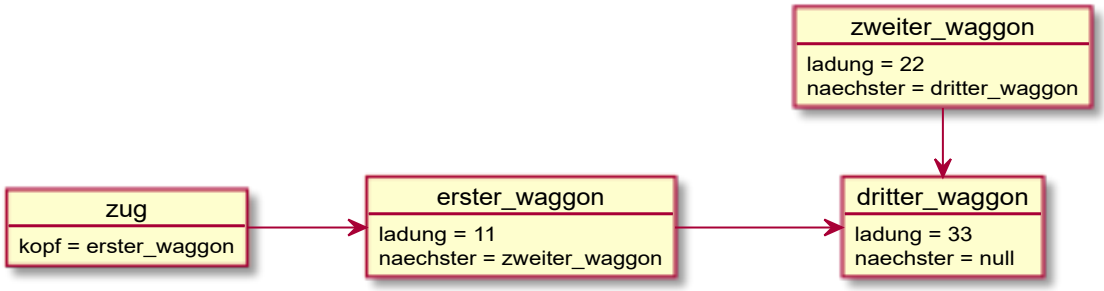
Nach ähnlichem Prinzip kann man Elemente lesen oder schreiben:

```
public int get(int i) {
        if (head == null) throw new NoSuchElementException();

        // beginne am Anfang
        IntElement it = head;
        while (i-- > 0)   // mache i Schritte
                it = it.next;

        return it.value;
}

public void put(int i, int v) {
        if (head == null) throw new NoSuchElementException();

        IntElement it = head;
        while (i-- > 0)
                it = it.next;

        // am Ziel: zuweisen
        it.value = v;
}
```

Etwas trickreicher ist nun das Löschen eines Elementes. Hierbei wird im Endeffekt nicht gelöscht, sondern ein Element durch ändern der Verlinkung unerreichbar gemacht (was wiederum die Löschung durch die JVM bewirkt).



Die Verlinkung wird nun verändert:




```
public int remove(int i) {
        // einfacher Fall: erstes Element
        if (i == 0) {
                int r = head.value;
                head = head.next;
                return r;
        }
```

```
        // vorlaufen bis "eins vor" dem zu löschenden Element
        IntElement it = head;
        while (--i > 0)
                it = it.next;

        // Verlinkung ändern, sodass das zu entfernende Element
        // nicht mehr erreichbar ist
        IntElement rem = it.next;
        it.next = rem.next;
        return rem.value;
}
```

Es verbleibt die Fingerübung, die Länge zu bestimmen, in dem man alle Elemente bis zu Ende zählt.

```
public int length() {
        IntElement it = head;
        int n = 0;
        while (it != null) {
                n++;
                it = it.next;
        }
        return n;
}
```

Zusammenfassung

- **Felder** (Arrays) haben zwar schnellen Lese- und Schreibzugriff, sind aber auf Grund Ihrer unveränderlichen Größe nicht geeignet für variable Datenmengen oder Einfüge- und Löschoperationen
- Eine **arraybasierte Liste** mit blockweiser Allokierung kann sinnvoll sein, wenn vor allem angehängt, gelesen und geschrieben wird.
- Eine **verkettete Liste** hat zwar einen langsameren Lesezugriff, kann dafür aber sehr effizient einfügen, anhängen und entfernen; damit eignet sie sich vor allem für Datenverarbeitung, bei der Datenströme sequenziell verarbeitet werden und die Anzahl der zu erwartenden Elemente unbekannt ist.
- Das **Arbeiten mit Interfaces** (Schnittstellen) stellt sicher, dass die darunterliegende Implementierung jederzeit getauscht werden kann. Bei Entwicklung im Team können so Verantwortungen klar aufgeteilt werden.

■