Technische Hochschule Rosenheim
Technical University of Applied Sciences

Objektorientierte Programmierung
Kapitel 2a – Verkettete Listen

Prof. Dr. Kai Höfig

Wiederholung: Felder(1)

Technische Hochschule Rosenheim
Technical University of Applied Sciences

- 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.

Wiederholung: Felder(2)

Technische Hochschule Rosenheim
Technical University of Applied Sciences

- 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.println(zs[i]);  // "0 1337 -1337"

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

Liste als sequenzielle Datenstruktur

Technische Hochschule Rosenheim
Technical University of Applied Sciences

- Da Arrays nicht dynamisch wachsen oder schrumpfen können, warden wir nun eine eigene Datenstruktur entwickeln, die das kann, eine Liste.

- Dazu definieren wir zunächst ein Interface, in dem wir die benötigten Interaktionen von Objekten unseres Listentyps definieren. Wir behandeln hier eine Liste von int-Werten.

```
public 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

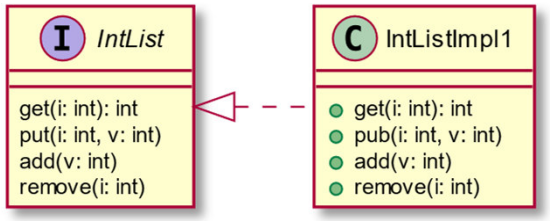
Realisierung und *is-a* Beziehung

Technische Hochschule Rosenheim
Technical University of Applied Sciences

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

```
public class IntListImpl1 implements IntList{
    public int get(int i)          { /* TODO */ return 0;}
    public void put(int i, int v) { /* TODO */ }
    public void add(int v)         { /* TODO */ }
    public void remove(int i)      { /* TODO */ }
    public int length()           { /* TODO */ return 0;}
}
```

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



- Beispiel:

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

Liste realisiert mit Array

Technische Hochschule Rosenheim
Technical University of Applied Sciences

- Wir können nun die vorhandene Array-Datenstruktur verwenden, um eine solche Listen Datenstruktur zu erstellen.

```
public class IntListImplArray implements IntList{
    private int[] zs;

    public IntListImplArray() {
        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);
    }
}
```

Verbesserte Liste mit Array und Blockgröße

Technische Hochschule Rosenheim
Technical University of Applied Sciences

- Da die zuvor dargestellte Implementierung bei jedem schreibendem Zugriff ständig neue Arrays erzugt, wird diese Implementierung schnell ineffizient. Eine Möglichkeit besteht darin, eine Blockgröße `BS` zu verwenden.

```
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;
}
```

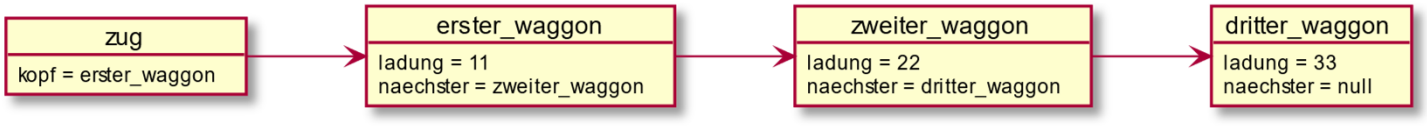
Verkettete Liste

Technische Hochschule Rosenheim
Technical University of Applied Sciences

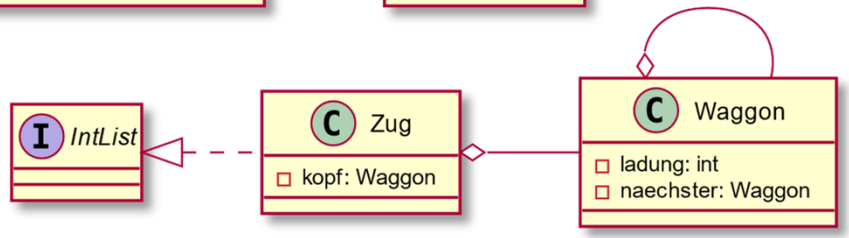
- Die vorigen Implementierungen sind
  - Schnell bei Lese- und Schreibzugriffen, aber
  - Ineffizient beim Löschen

- Wir suchen jetzt eine andere Implementierung und betrachten dabei das Beispiel eines Zuges. Abstrakt betrachtet ist ein Güterzug eine Liste:

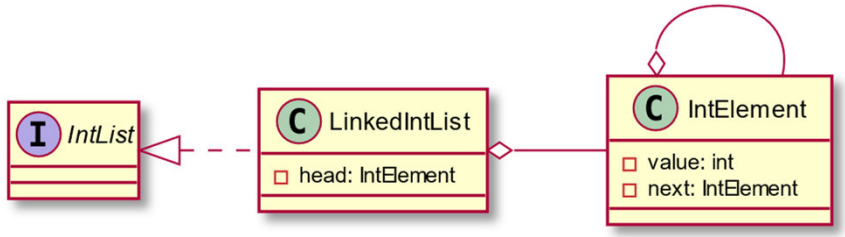
  - Fährt die Lok alleine, so ist die Liste leer.
  - Man kann problemlos Waggons eingliedern (*einfügen*), ausgliedern (*löschen*) oder anhängen (*hinzufügen*).
  - Möchte man z.B. den Inhalt des 3. Waggons, so beginnt man vorne bei der Lok und "hangelt" sich bis zum 3. Waggon durch.
  - Würde man einen "Güterzug" mit Ladung 11, 22 und 33 als Objektdiagram zeichnen, so könnte das so aussehen:



  - ..und als Klassendiagramm so



Abstrakter: eine verkettete Liste von ganzen Zahlen

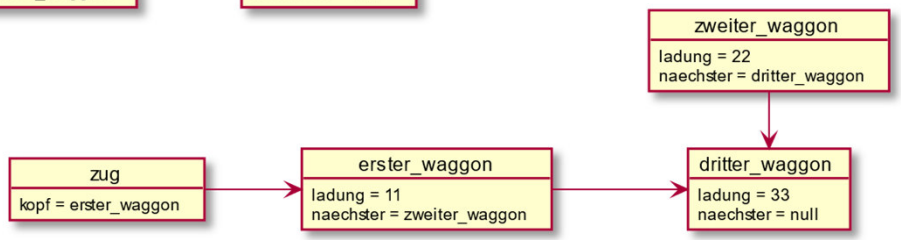


- Wir brauchen also die Klassen `IntElement` und `LinkedIntList`, wobei Letztere das Interface `IntList` implementiert.

- Die Methoden die wir zur Interaktion benötigen, müssen das Einfügen, Löschen, Lesen und Schreiben ermöglichen.

- 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).

Hier wird der zweite Waggon durch "überspringen" entfernt.



Zusammenfassung

Technische Hochschule Rosenheim
Technical University of Applied Sciences

- **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.