

Einfach und mehrfach verkettete Listen

- Sie haben ein intuitives Verständnis vom Landau Gross-O-Begriff.
- Sie wissen, was die einfach und mehrfach verketteten Listen sind.
- Sie kennen die wichtigsten Operationen auf Listen und wissen wie die Operationen definiert sind.
- Sie kennen das Konzept der Iteratoren und deren Implementation in Java.
- Sie kennen die speziellen Listen: doppelt verkettete, zirkulär und sortiert.
- Sie können mit den Java-Collection-Klassen umgehen.





O-Notation

O-Notation: Komplexität in Zeit und Speicher

- Beispiel Berechnung ggt:

```
c = Math.min(a,b);  
while ((a % c != 0) || (b % c != 0)) c--;
```

- Welche Laufzeit hat das Programm?
Hängt stark von Maschine, Programmiersprache und Daten ab.

Fragen:

1. **Zeitkomplexität**: Welche Zeit wird der Algorithmus benötigen?
2. **Speicherkomplexität**: Wieviel Speicher wird der Algorithmus benötigen?

Nicht absolute Zahlen, sondern als Funktion von Grösse oder Anzahl Werten: n

- Z.B. die Zeit, die der obige Algorithmus benötigt, wächst linear mit den Werten von a , b : Aufwand: $O(n)$, mit euklidischem Algorithmus aber nur: $O(\log_2 n)$

O-Notation: Definition

Definition: $T(n) = O(g(n))$ für $n \rightarrow \infty$

$$T(n) = O(g(n)) \quad \Leftrightarrow \quad \exists c, n_0 \text{ positive Konstanten: } \forall n \geq n_0: T(n) \leq c \cdot g(n)$$

Es existieren $\{\exists\}$ positive Konstanten c und n_0 ,
sodass für alle $\{\forall\}$ $n \geq n_0$ gilt $\{:\}$ $T(n) \leq c \cdot g(n)$.

g ist z.B. abhängig von
der Menge n der Daten.

Bedeutung:

- Für genügend grosse n wächst **Laufzeit-Funktion** T höchstens so schnell wie g .

Bemerkungen:

- g ist eine «asymptotische obere Schranke» für T .
- genaue Laufzeit-Funktion T wird grob nach oben abgeschätzt durch einfachere Funktion g .
- Beispiel: wenn n doppelt so gross ist, braucht der Algorithmus doppelt so lange $\Rightarrow O(n)$
- Mittels c lassen sich Unterschiede von Maschine und Programmiersprache «ausgleichen».

O-Notation: Definition (andere Sichtweise)

Die Definition der O-Notation besagt, dass wenn $T(n) = O(g(n))$, ab irgend einem n_0 die Gleichung $T(n) \leq c \cdot g(n)$ gilt.

Weil $T(n)$ und $g(n)$ Zeitfunktionen sind, ihre Werte also immer positiv sind, gilt:

$$\frac{T(n)}{g(n)} \leq c, \text{ ab irgendeinem } n_0$$

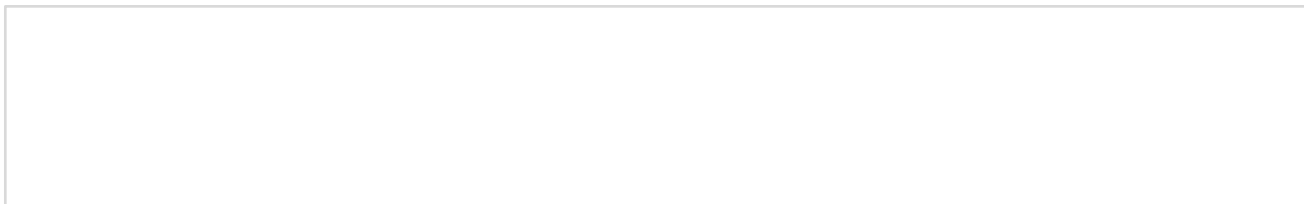
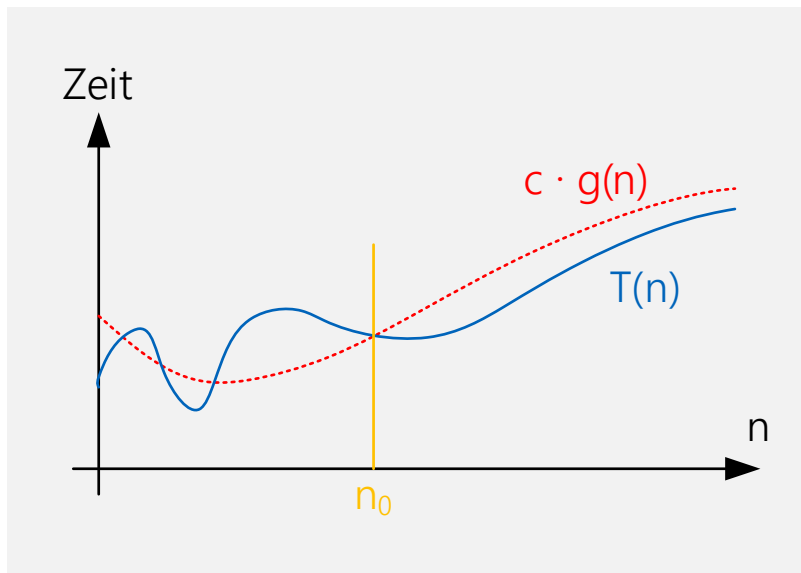
Die beiden Funktionen können besser verglichen werden, wenn man den Grenzwert berechnet.

$$\lim_{n \rightarrow \infty} \frac{T(n)}{g(n)} \left\{ \begin{array}{l} \text{Konvergiert}^{1N) \text{ die Funktion, dann gilt } T(n) = O(g(n)).} \\ \text{Wenn der Grenzwert gleich 0 ist, bedeutet dies, dass} \\ \text{g(n) sogar schneller wächst als T(n). In diesem Fall wäre} \\ \text{g(n) ungeeignet, um die obere Grenze des Wachstums} \\ \text{der Laufzeit T(n) anzugeben (obwohl korrekt).} \end{array} \right.$$

O-Notation: Definition (andere Sichtweise)

Die Definition der O-Notation besagt, dass wenn $T(n) = O(g(n))$, ab irgend einem n_0 die Gleichung $T(n) \leq c \cdot g(n)$ gilt.

Oder grafisch dargestellt:



O-Notation: Beispiel ggT(a,b)

Gezählt werden die Anzahl Divisionen im durchschnittlichen Fall:

a, b	ggT Linear*	Euklid*
$\in [1 \dots 10]$	5	3.3
$\in [1 \dots 100]$	50	6.6
$\in [1 \dots 1000]$	500	9.9
$\in [1 \dots 10000]$	5000	13.3
$\in [1 \dots 10'000'000]$	5'000'000	23.3

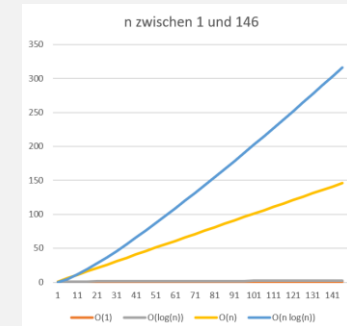
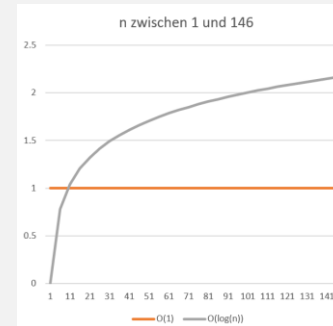
* Die Werte sind nur grob angenähert: Bestimmung genauerer Werte ist ein komplexes zahlentheoretisches Problem. Euklid wächst scheinbar «irgendwie» logarithmisch.

O-Notation: Wichtige Komplexitätsklassen

Was ist akzeptabel?

Nach Wachstumsgeschwindigkeit sortiert:

- $O(1)$ konstanter Aufwand
- $O(\log n)$ logarithmischer Aufwand
- $O(n)$ linearer Aufwand
- $O(n \cdot \log n)$ linear-logarithmischer A.
- $O(n^2)$ quadratischer Aufwand
- $O(n^k)$, $k > 1$ polynomialer Aufwand
- $O(k^n)$ exponentieller Aufwand
- $O(n!)$ faktorieller Aufwand



O-Notation: Beispiele von Algorithmen

r: Konstante; s: Statement mit Aufwand $O(1)$

- Einfache Anweisungssequenz:

```
s1; s2; s3; ...; sr
```

Aufwand in O-Notation:

- Einfache Schleifen:

```
for (int i = 0; i < n; i++) s;
```

- Geschachtelte Schleifen:

```
for (int i = 0; i < n; i++) {  
    for (int j = 0; j < n; j++) s;  
}
```

O-Notation: Rechenregeln

r, s: Konstanten; f, g: Funktionen, n: Anzahl Operationen

$$O(f \cdot g) = ?$$

r > 0

$$O(r \cdot f) = ?$$

$$O(\log_r s \cdot n) = ?$$

r > 1, s > 1

$$O(r^s \cdot n) = ?$$

O-Notation: Rechenregeln

$$O(f) > O(g)$$

$$O(f + g) = ?$$

$$s > r > 0$$

$$O(n^r) + O(n^s) = ?$$

$$r > 1$$

$$O(r^n + n^s) = ?$$

- | | | |
|----|---------------------|----------------------------------|
| 1. | $O(1)$ | konstanter Aufwand |
| 2. | $O(\log n)$ | logarithmischer Aufwand |
| 3. | $O(n)$ | linearer Aufwand |
| 4. | $O(n \cdot \log n)$ | linear-logarithmischer Aufwand |
| 5. | $O(n^2)$ | quadratischer Aufwand |
| 6. | $O(n^k)$ | für $k > 1$ polynomialer Aufwand |
| 7. | $O(k^n)$ | exponentieller Aufwand |
| 8. | $O(n!)$ | faktorieller Aufwand |

Dominanz

Komplexitätsklassen nach
Wachstumsgeschwindigkeit

O-Notation: Übung Rechenregeln

r: Konstante > 0; f: Funktion, n: Anzahl Operationen

$$f = 2^{2n} + 3n + 5$$

$$f = n^{1.00001} + 1000 \cdot \log(n)$$


$$f = 2^{1.00001n} + r \cdot 1000n$$

$$f = n^{-1} + n$$

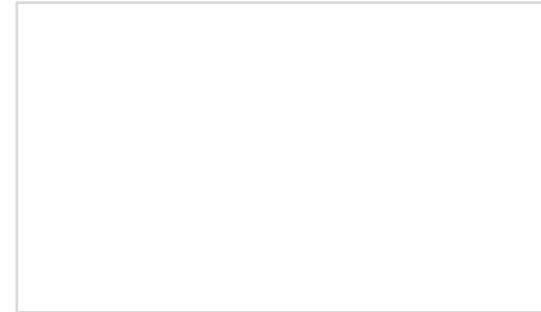
$$f = n(n - 1) / 2$$

O-Notation: Weitere Beispiele von Algorithmen


- Schleifen-Index ändert sich nicht linear:

```
i = 1;
while (i <= n) {
    s;  s = Statement; O(1)
    i = i * 2;
}
```

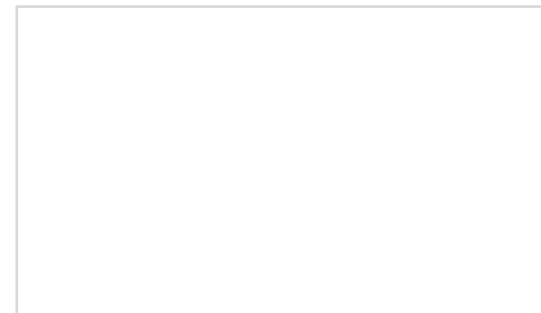
Aufwand in O-Notation:



- Schleifen-Indizes hängen voneinander ab:

```
for (i = 0; i < n; i++) {
    for (j = 0; j < i; j++) {
        s;  s = Statement; O(1)
    }
}
```

Aufwand in O-Notation:



O-Notation: Übung Schleifen

r: Konstante; f: Funktion, n: Anzahl Operationen

```
int n = r;  
while (n > 0) {  
    n = n / 3;  
}
```

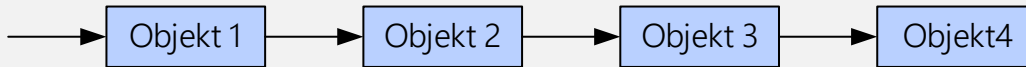
```
for (int i = 0; i < n; i++) {  
    for (int j = i; j < n; j++) {  
        f(i,j);  
    }  
}
```

```
for (int i = 0; i < n; i++) {  
    for (int j = n; j > 0; j = j/2) {  
        f(j);  
    }  
}
```



Listen, Listenode, Iterator

Listen: Als abstrakter Datentyp (ADT)



Wiederholung Lektion 1.

Die Liste ist ein abstrakter Datentyp der eine Liste von Objekten verwaltet:

- Die Liste ist eine grundlegende Datenstruktur der Informatik.
- Mit der Liste kann ein Stack variabler Grösse implementiert werden.
- Speichert Objekt oder Wert durch Typenplatzhalter bestimmt (z.B. generisch).

```
void add(Object o)
```

Fügt o am Ende der Liste an.

```
void add(int pos, Object o)
```

Fügt o an der Position pos in die Liste ein.

```
Object get(int pos)
```

Gibt das Element an Position pos zurück.

```
Object remove(int pos)
```

Entfernt das Element an Position pos und gibt es zurück.

```
int size()
```

Gibt die Anzahl Elemente zurück.

```
Boolean isEmpty()
```

Gibt true zurück, fall die Liste leer ist.

- In Java gibt es im Package `java.util` das Interface [`java.util.List`](#) mit diversen Implementationen: `LinkedList`, `ArrayList`, etc.

Listen: Einsatzbereich

Eigenschaften:

- Anzahl der Elemente zur Erstellungszeit unbekannt (sonst meist Array)
- Reihenfolge/Position ist relevant
- Einfügen und Löschen von Elementen ist unterstützt

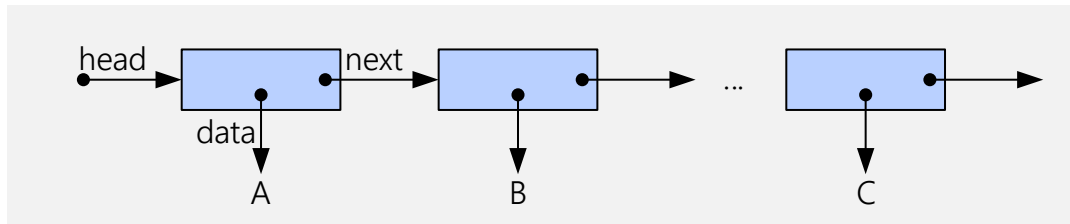
Einsatz als universelle (Hilfs-)Datenstruktur:

- Zur Implementierung von Stack, Queue, etc.

In Betriebssystemen:

- Disk-Blöcke, Prozesse, Threads, etc
- Liste der belegten/freien Memoryblöcke

Listen: Struktur des Listenknotens und der Liste



head ist auch ein Listenknoten.

```
class ListNode
{
    Object data;
    ListNode next;

    ListNode(Object o) {
        data = o;
    }
}
```

Daten der Liste.

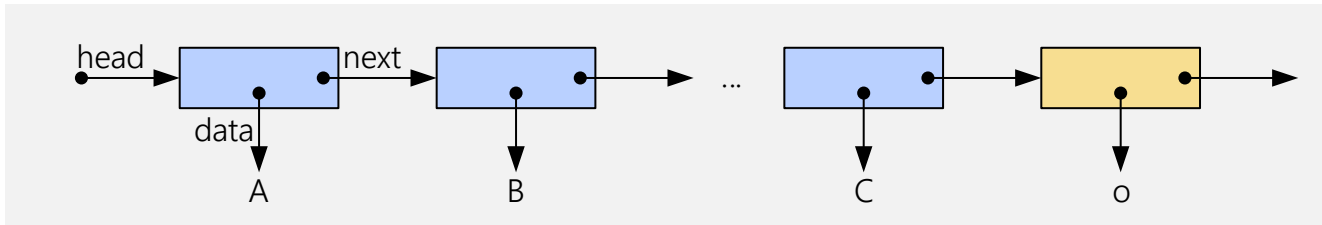
Referenz auf nächsten ListNode.

```
public class LinkedList implements List {
    private ListNode head;

    void add(Object o) { ... }
    void add(int pos, Object o) { ... }
    Object get(int pos) { ... }
    Object remove(int pos) { ... }
    Object remove(Object o) { ... }
    boolean contains(Object o) { ... }
    ...
}
```

Mögliche Implementation auf den folgenden Folien (19 – 22 und 30, 31).

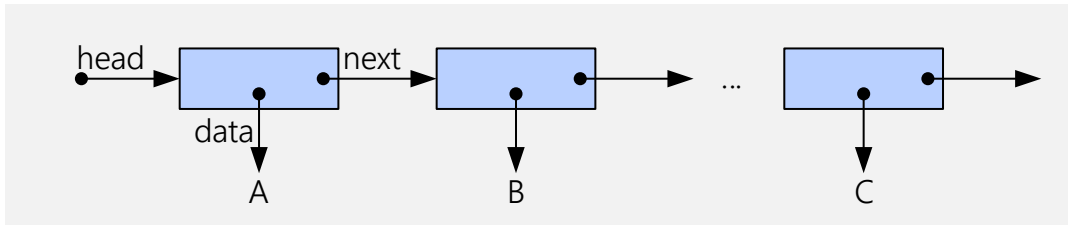
Listen: Einfügen/Add am Ende



```
void add(Object o) {  
    ListNode n = new ListNode(o);  
    ListNode f = head;  
    while (f.next != null) {  
        f = f.next;  
    }  
    f.next = n;  
}
```

Listen-Element am Schluss der Liste einfügen.

Listen: Suchen mit Position

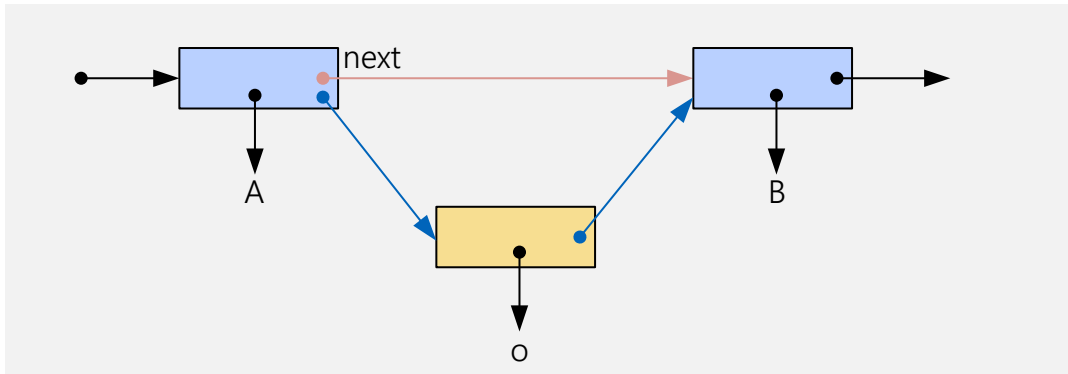


```
Object get(int pos) {  
    ListNode node = head.next;  
    while (pos > 0) {  
        node = node.next;  
        pos--;  
    }  
    return node.data;  
}
```

Suche das Listen-Element an der vorgegebenen Position.

Was passiert, falls die Liste weniger Elemente hat als gesucht?

Listen: Einfügen/Add vorgegebene Position

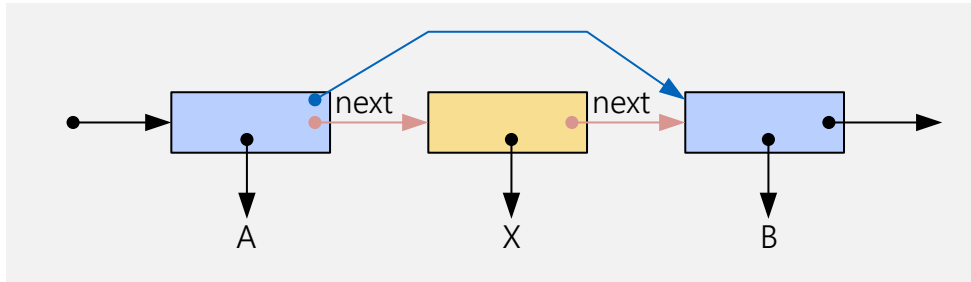


```
void add(int pos, Object o) {
    ListNode n = new ListNode(o);
    ListNode e = head,
    ListNode f = head.next;
    while ((pos > 0) && (f != null)) {
        e = f;
        f = f.next;
        pos--;
    }
    n.next = f;
    e.next = n;
}
```

Listen-Element an der vorgegebenen Position einfügen.

Was passiert, falls die Liste weniger Elemente hat als gesucht?

Listen: Entfernen/Remove vorgegebene Position

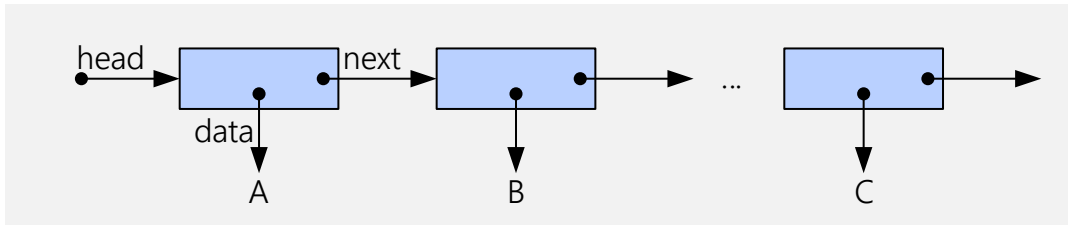


```
Object remove(int pos) {  
    ListNode n;  
    ListNode e = head,  
    ListNode f = head.next;  
    while ((pos > 0) && (f != null)) {  
        e = f;  
        f = f.next;  
        pos--;  
    }  
    e.next = f.next;  
    return f;  
}
```

Listen-Element an der vorgegebenen Position entfernen.

Was passiert, falls die Liste weniger Elemente hat als gesucht?

Listen: Traversierung

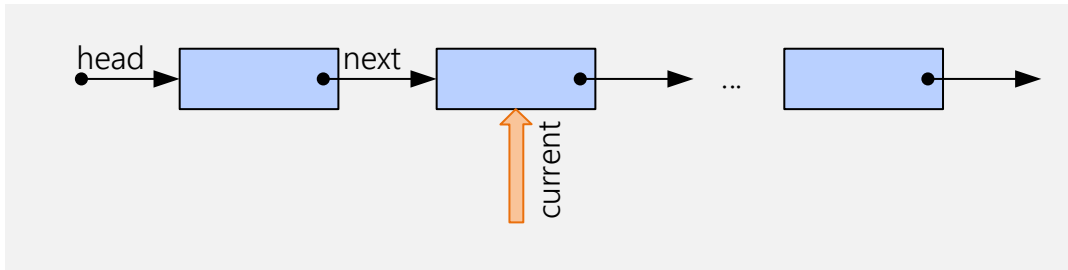


```
List list = new LinkedList();
// ... (put a lot of data into the list)

// print every element of linked list
for (int i = 0; i < list.size(); i++) {
    String element = (String)list.get(i);
    System.out.println(i + ": " + element);
}
```

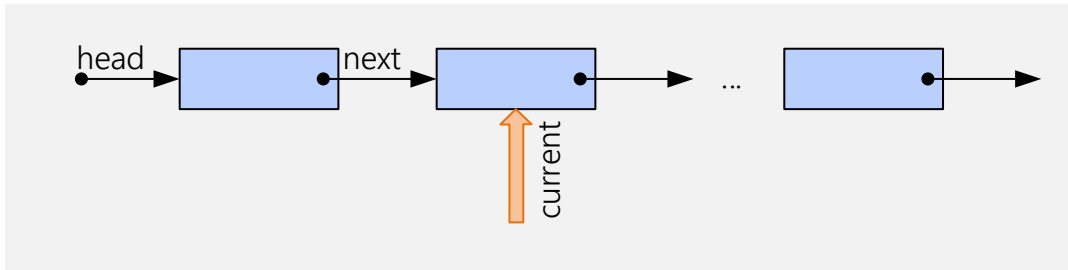
Ausgabe aller Listenelemente.
Dieser Code ist sehr ineffizient,
wenn die Listen gross sind.
Wieso?

Listen: Problem der Position



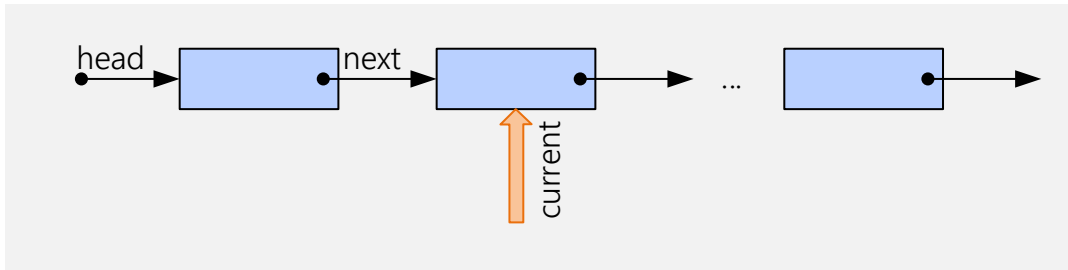
- Zum Bestimmen des i-ten Elements muss (intern) im Schnitt die halbe Liste durchlaufen werden $\Rightarrow O(n)$
- Besser wäre es, sich die Position jeweils zu merken
- Es gibt in Java ein spezielles Objekt dafür: den **Iterator**
- Allgemein: spricht man auch vom Iterator-Entwurfsmuster/Pattern

Listen: Konzept des Iterators



- Der Iterator ist ein ADT, mit dem eine Datenstruktur, traversiert werden kann, ohne dass die Datenstruktur bekannt gemacht werden muss: Information Hiding.
- Der Iterator wird nicht nur bei Listen (ListIterator) verwendet, sondern auch bei anderen ADT.
- Es wird ein privater (**current**) Zeiger auf die aktuelle Position geführt.
- Der Iterator wird im '**for (Object o: List)**'-Konstrukt erzeugt aber **versteckt**.

Listen: Interface Iterator



```
interface Iterator<E> {  
    boolean hasNext();  
    E next();  
    void remove();  
}
```

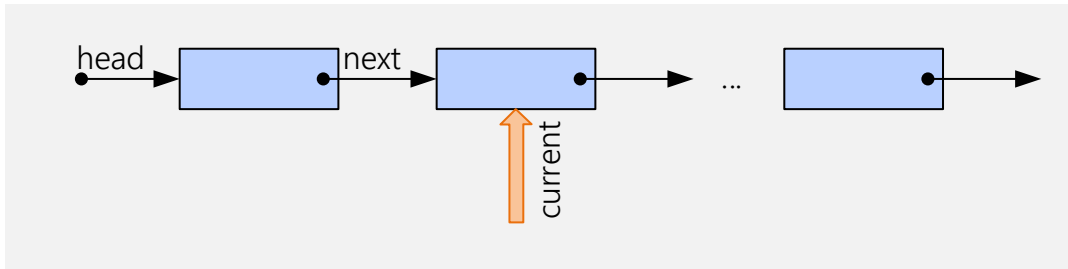
Es hat noch weitere Objekte.

Liefere nächstes Objekt.

Lösche das Element, das zurückgegeben wurde.

Das «allgemeine» Iterator-Interface von Java hat nur 4 Methoden (hier fehlt `forEachRemaining`).

Listen: ListIterator und Listen



Iterator der Liste.

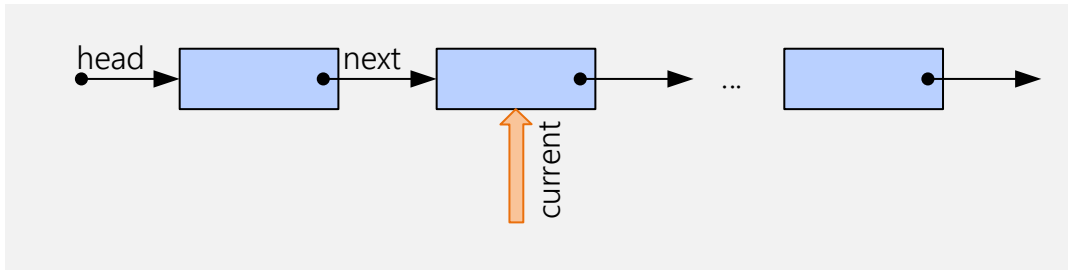
```
interface ListIterator<E> implements Iterator {
    boolean hasNext();
    E next();
    void remove();
    add(E e);
}
```

Fügt Objekt e vor dem Element, das beim nächsten next-Aufruf zurückgegeben wird, in die Liste ein.

```
class LinkedList<E> {
    ...
    ListIterator iterator();
    ...
}
```

Liefert Iterator auf den Anfang der Liste.

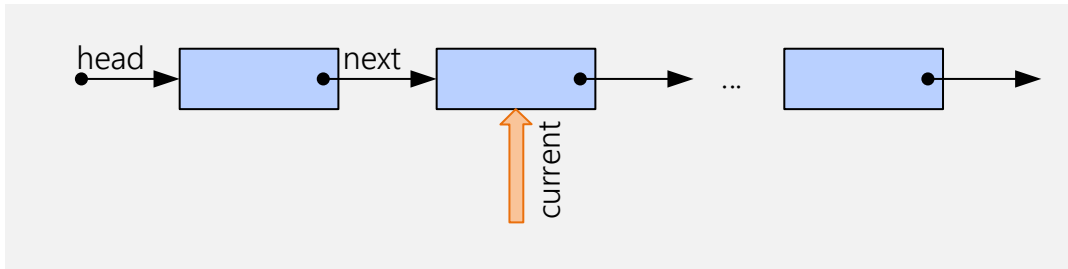
Listen: Iterator Verwendungsmuster



```
Iterator itr = list.iterator();
for (int i = 0; itr.hasNext(); i++) {
    Object element = itr.next();
    System.out.println(i + ": " + element);
}
```

Gehe durch alle Elemente der Liste und gebe Position und Wert aus.

Listen: Iterator und foreach-Schleife



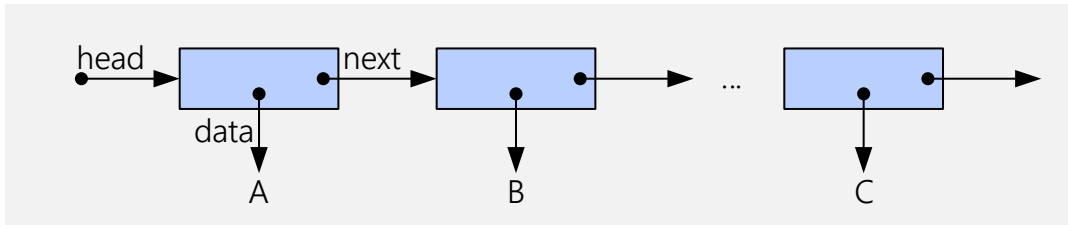
```
Iterator itr = list.iterator();
while (itr.hasNext()) {
    Object element = itr.next();
    <do something with element>;
}
```

Allgemeines Verwendungsmuster.

```
for (Object element : list) {
    <do something with element>;
}
```

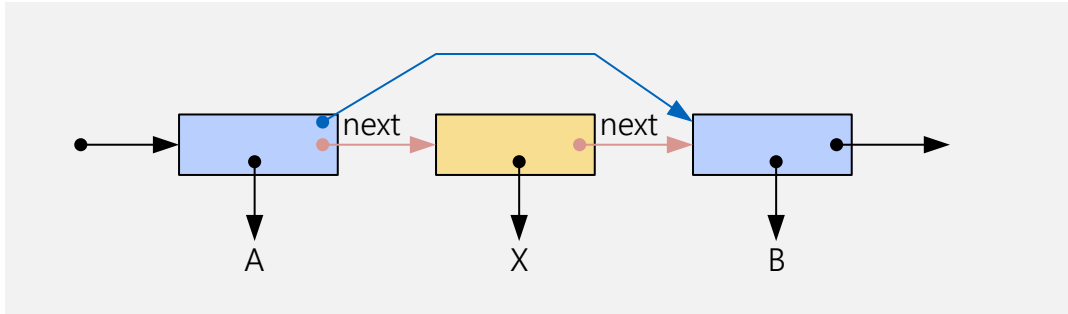
In foreach-Schleife versteckt.

Listen: Enthält/Contains Liste das Objekt



Übung: Schreiben Sie die Methode **contains**, die mittels einem Iterator überprüft, ob ein Element bereits in der Liste existiert.

Listen: Entfernen/Remove eines best. Objektes

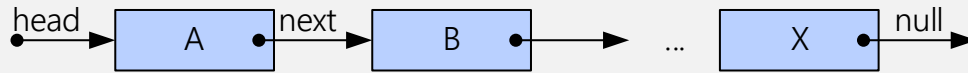


Das zu löschende Element mittels Iterator suchen und aus Liste entfernen.

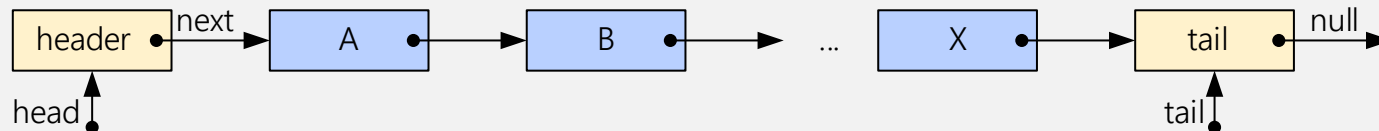
Übung: Schreiben Sie die Methode **remove** (Liste hat mindestens ein Element).

Hinweis: Das Interface Iterator hat schon eine Methode **remove()** mit dem das aktuelle Element gelöscht werden kann.

Listen: Mitte, Anfang und Ende



- Anfang der Liste: wird meist als **head** oder **root** bezeichnet, Ende der Liste: **next** zeigt auf **null**.
- Operationen müssen unterschiedlich implementiert werden, je nachdem ob sie in der Mitte, am Anfang oder am Ende der Liste angewendet werden.
- Zur Vereinfachung definiert man deshalb oft einen leeren sog. Anfangsknoten oder Header-Node und einen sog. Schwanzknoten oder Tail-Node.
- Das erste und letzte Element sind somit keine Spezialfälle mehr, jedes Element hat einen Vorgänger und einen Nachfolger.

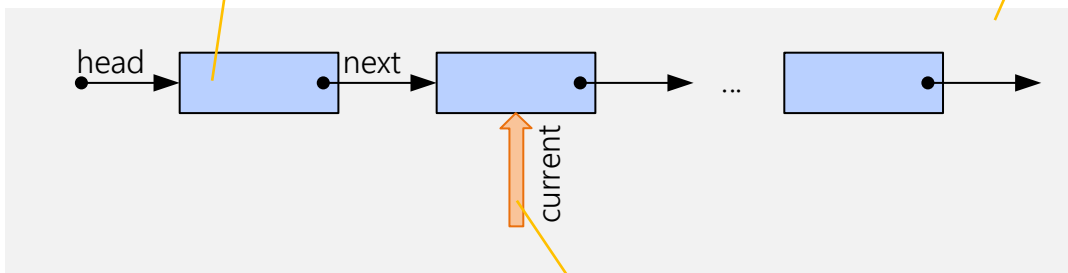


Listen: Zusammenfassung

Listen-Element: ListNode (privat)
Behälter für die Objekte (Zeiger)
Zeiger auf das nächste Element.

LinkedList implementiert List

Definiert Operationen auf Listen wie
z.B. das Einfügen, den Zugriff usw.
Zeiger auf Anfang der Liste.



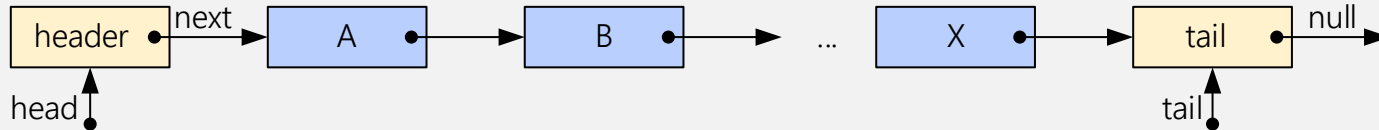
ListIterator implementiert Iterator

- Ermöglicht das Iterieren durch die Liste (ohne Verletzung des Information Hiding-Prinzips)
- Verwaltet eine aktuelle Position: private ListNode **current**
- Gleichzeitig mehrere Iteratoren auf der gleichen Liste einsetzbar.



Doppelt verkettete Liste

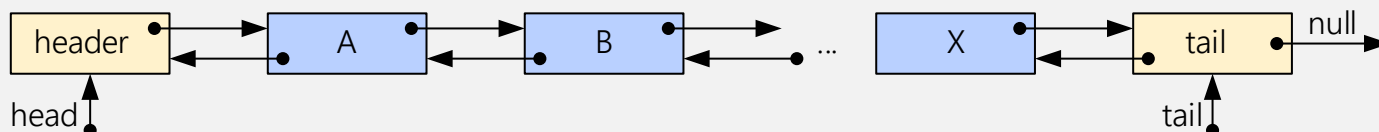
Doppelt verkettete Liste



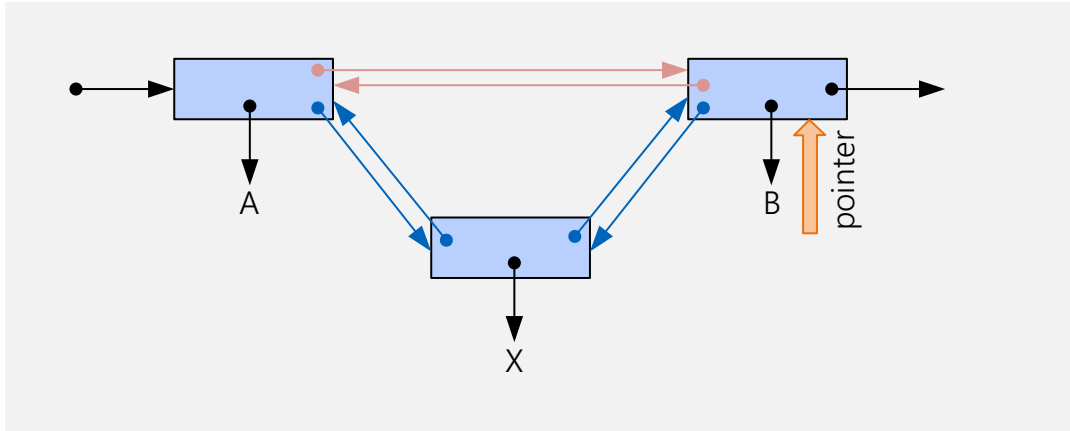
Folgende Probleme treten bei einfach verketteten Listen auf:

- Man kann sich mit `next()` nur in einer Richtung effizient durch die Liste «hangeln», die Bewegung in die andere Richtung ist ineffizient
- Der Zugang zu Elementen in der Nähe des Listenendes kostet viel Zeit (im Vergleich mit einem Zugriff auf den Listenanfang).

```
class ListNode
{
    Object data;
    ListNode next, prev;
}
```



Doppelt verkettete Liste: Einfügen/Add an definierter Position

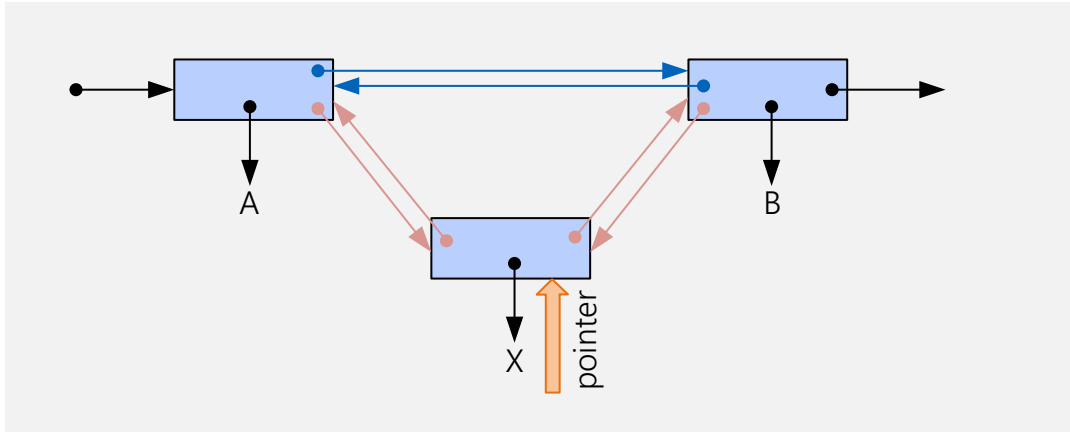


Ein Element vor dem
«pointer» einfügen
(zwischen zwei
Elementen).

Nachteil (gegenüber
einfach verkettet):
mehr Anweisungen.

Vorteil:
add-Operation ist an jeder
Stelle einfach möglich.

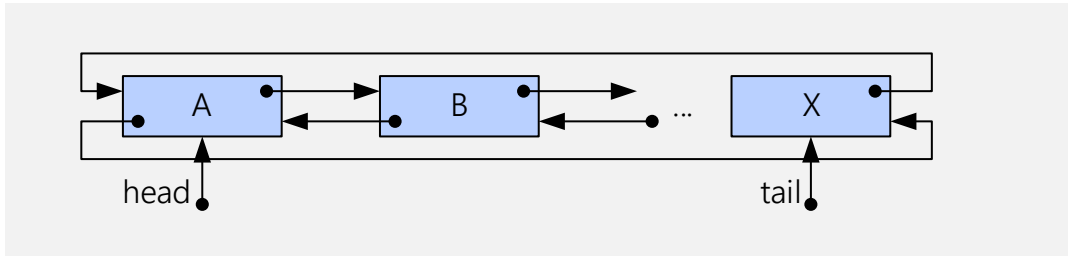
Doppelt verkettete Liste: Entfernen/Remove an definierter Position



Das «pointer»-Element löschen.

Vorteil:
remove-Operation ist nun
sehr einfach.

Zirkulär, doppelt verkettete Liste



Head- und Tail-Knoten wurden eingeführt, um sicherzustellen, dass jeder Knoten einen Vorgänger und Nachfolger hat.

Idee: wieso nicht einfach das erste Element wieder auf das Letzte zeigen lassen?

⇒ Zirkuläre, doppelt verkettete Liste.

Nur noch die leere Liste muss als Spezialfall separat behandelt werden.

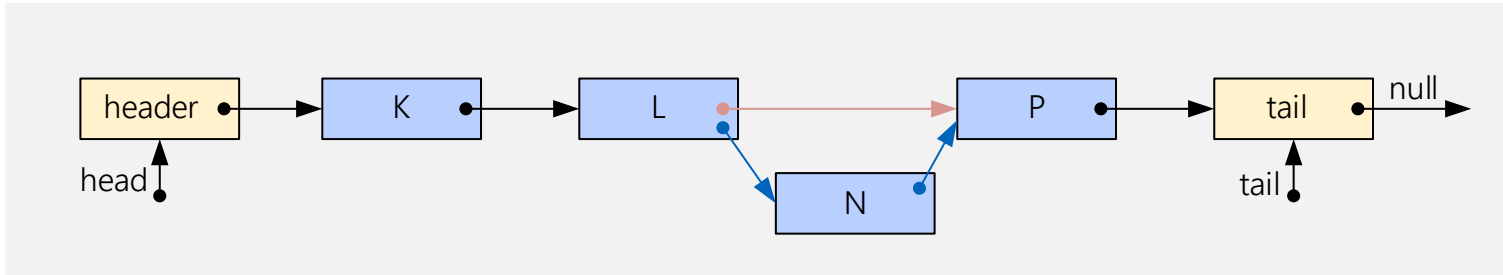


Sortierte Listen

Sortierte Listen

Wie der Name sagt: Die Elemente in der Liste sind (immer) sortiert.

Hauptunterschied: `insert()` Methode fügt die Elemente sortiert in die Liste ein.



Implementation Einfügen:
Suche vor dem Einfügen
die richtige Position.

Anwendung: Überall dort, wo sortierte Datenbestände verwendet werden, z.B. PriorityQueue.

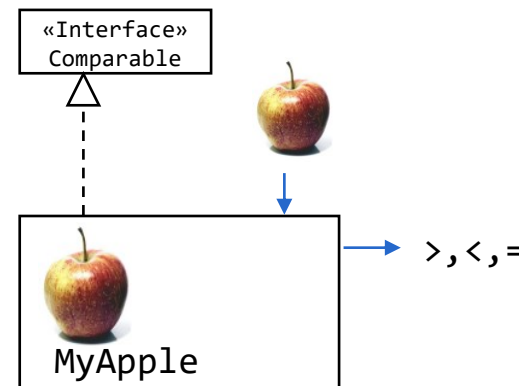
Sortierte Listen: Comparable Interface

- Problem: Wie vergleicht man Objekte miteinander?
⇒ Es muss etwas geben, das eine Beurteilung von 2 Elementen bezüglich $>$, $=$, $<$ ermöglicht
- Lösung: Das Interface **java.lang.Comparable** ist zum Bestimmen der relativen (natürlichen) Reihenfolge von Objekten vorgesehen.

x.compareTo(y)

```
public interface Comparable {  
    int compareTo(Object o);  
}
```

$x < y \Rightarrow$ negative Zahl
 $x = y \Rightarrow 0$
 $x > y \Rightarrow$ positive Zahl



Sortierte Listen: Comparable Interface - Beispiel

```
class MyApple implements Comparable {  
    int value;  
  
    int compareTo(Object o) {  
        Apple a = (Apple)o;  
        if (this.value < a.value) return -1;  
        else if (this.value > a.value) return 1;  
        else return 0;  
    }  
}
```

Oder:
`return this.value - ((Apple)o).value` falls
es nicht zu Werteüberlauf kommen kann.
Sonst `Integer.compare(this.value, a.value)`.

Sortierte Listen: Comparable Interface

Das Interface `java.lang.Comparable` wird von folgenden Klassen implementiert:

Byte, Character, Double, File, Float, Long, ObjectStreamField, Short, String, Integer, BigInteger, BigDecimal, Date, ...

Listen, die aus Objekten bestehen, welche dieses Interface implementieren, können mit `Collections.sort` (statische Methode) automatisch sortiert werden.

```
Collections.sort(List list);
```

Collections != Collection
Collection ist ein Interface.
Collections ist eine Utility Class.

Sortierte Listen: Comparator Interface

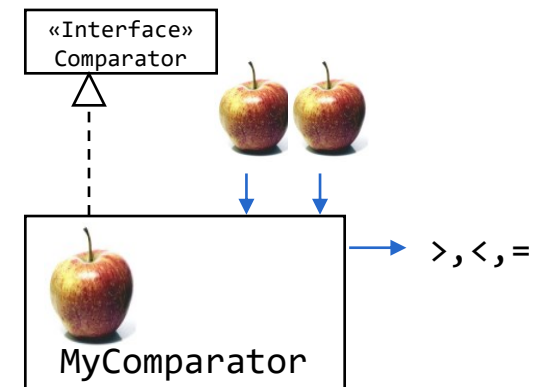
Was macht man, wenn nach einem anderem Kriterium verglichen werden soll, z.B. Anzahl Würmer je Apfel?

- **Lösung:** Das Interface `java.util.Comparator` ist vorgesehen für Objekte die nach unterschiedlichen Kriterien sortiert werden sollen.
- Es können sogar unterschiedliche Objekte sein, solange sie vergleichbar sind.

```
public interface Comparator {  
    public int compare(Object o1, Object o2);  
}
```

- Ein Comparator-Objekt (Objekt von Klasse welche Comparator implementiert) wird beim Methodenaufruf übergeben.

```
Collections.sort(List list, Comparator comp);
```



Sortierte Listen: Comparator Interface - Beispiel

Verwendet man den RawType-Comparator* (Typ nicht durch Typenplatzhalter bestimmt) dann kann mittels einem Comparator ein gemeinsames Kriterium zweier beliebiger Objekte verglichen werden (wird aber eher selten verwendet).

```
class MyComparator implements Comparator {  
  
    int compare(Object o1, Object o2) {  
        Apple a = (Apple)o1;  
        Pear p = (Pear)o2;  
        if (a.value < p.value) return -1;  
        else if (a.value > p.value) return 1;  
        else return 0;  
    }  
}
```

Wir vergleichen Äpfel mit Birnen!



* Wird in der Generic-Vorlesung Lektion 3 noch erklärt (Nerd Zone).



Arrays und Listen

Arrays und Listen - Vergleich

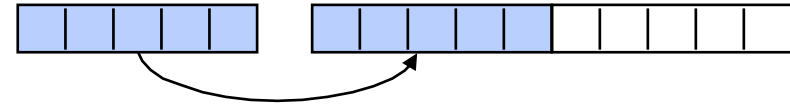
- **Array**: Teil der Java Sprache
 - Benutzung sehr einfach
 - Für alle eingebauten Typen und beliebige Objekte
 - **Anzahl Elemente muss zur Erstellungszeit bekannt sein**: `new A[10];`
 - Operationen:
 - Indizierter Zugriff sehr **effizient**: `a[i]`
 - Anfügen von Elementen, Ersetzen und Vertauschen von Elementen
 - Einfügen und Löschen mit Kopieren verbunden: **ineffizient**
- **Liste**: Klassen in Java Bibliothek: `LinkedList`
 - nicht ganz so einfach in der Benutzung
 - nur Referenztypen können in Listen verwaltet werden
 - **Anzahl Elemente zur Erstellungszeit nicht bekannt**: `new LinkedList();`
 - Operationen:
 - Indizierter Zugriff möglich aber **ineffizient**: `list.get(i)`
 - Anfügen, Ersetzen, Vertauschen und Einfügen und Löschen von Elementen

Implementation Liste mittels Array

Array Implementation der Liste in Java: **java.util.ArrayList**

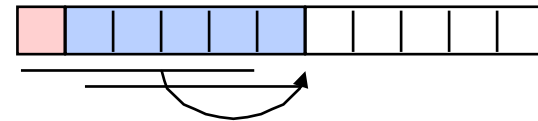
- Wenn mehr Elemente gespeichert werden sollen als im Array Platz haben, muss ein neuer Array erstellt werden und es müssen die Elemente umkopiert werden (gute Strategie Länge * 2).

```
aNeu = new Object[a.length * 2];  
System.arraycopy(a, 0, aNeu, 0, a.length);  
a = aNeu;
```



- Beim Einfügen am Anfang oder in der Mitte der Liste müssen alle nachfolgenden Elemente im Array umkopiert werden:

```
System.arraycopy(a, 0, a, 1, a.length-1);
```



- Langsam beim Einfügen und Löschen.
- Schneller beim Zugriff auf beliebiges Element.

ArrayList und LinkedList - Vergleich

- ArrayList
 - Implementation als Array
 - direkter Zugriff schnell, Mutationen langsam
 - non-synchronized Aufrufe
- LinkedList
 - schneller für Mutationen
 - langsam bei direktem Zugriff
 - non-synchronized Aufrufe
- Vector **deprecated**
 - ab JDK 1.0 vorhanden, alt
 - z.T. redundante Methoden
 - relativ langsam
 - synchronized Aufrufe



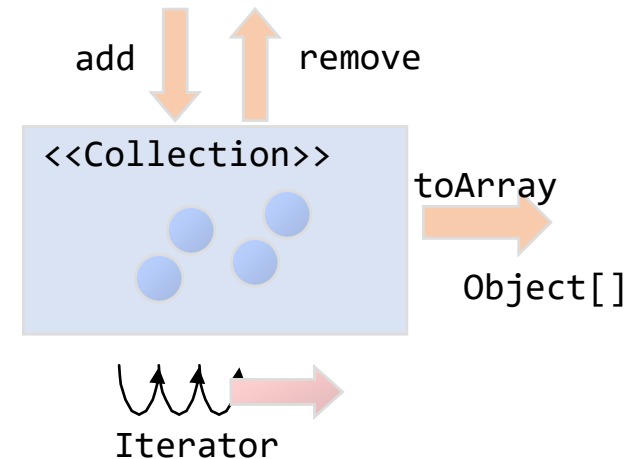
Collection-Interface, Collections-Klasse

java.util.Collection-Interface

Abstrakter Datentyp für beliebige Objektbehälter: Gemeinsames Interface für Sammlungen (Collection) von Objekten, mit Ausnahme des Arrays (leider).

Funktionskopf	Beschreibung
<code>void add(Object x)</code> <code>boolean remove (Object x)</code> <code>void removeAll()</code>	Fügt x hinzu. Löscht das Element x. Löscht alle Elemente.
<code>Object[] toArray()</code> <code>Iterator iterator()</code>	Wandelt Collection in Array um. Gibt Iterator auf Collection zurück.
<code>int size()</code> <code>boolean isEmpty()</code>	Gibt Anzahl Element zurück. Gibt true zurück, falls die Collection leer ist.

Die `add` Methode fügt ein Element an der «natürlichen» Position hinzu.



java.util.Collections-Klasse

Sammlung von Methoden für Collections. Folgende statischen Methoden von **Collections** können angewendet werden:

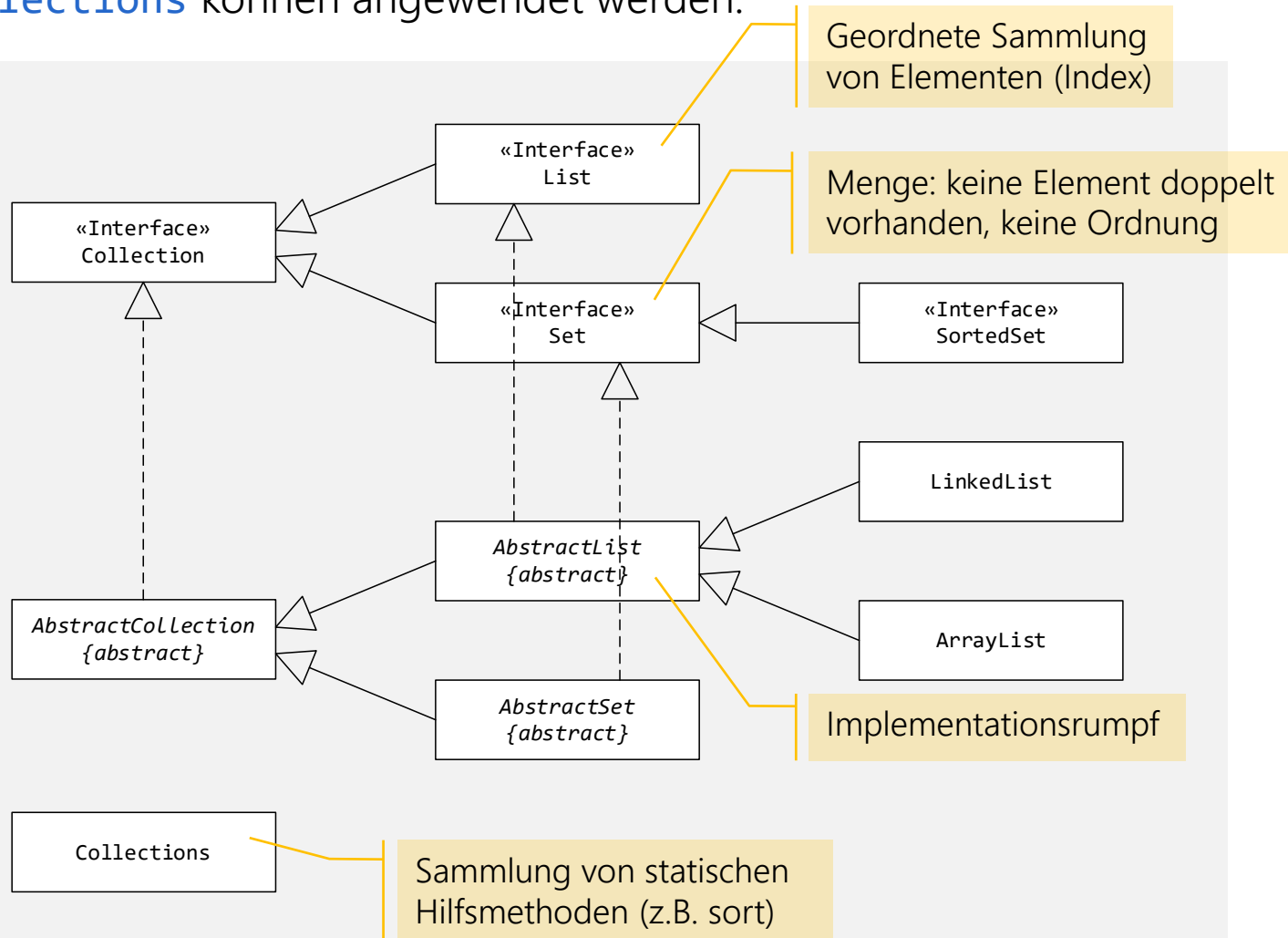
Funktionskopf	Beschreibung
<code>binarySearch(list, value)</code>	Durchsucht eine sortierte Liste nach einem Wert und gibt dessen Index zurück.
<code>copy(dest, source)</code>	Kopiert alle Elemente von einer Liste in eine andere.
<code>fill(list, value)</code>	Ersetzt alle Werte in der Liste durch den angegebenen Wert.
<code>max(list)</code>	Gibt den größten Wert in der Liste zurück.
<code>min(list)</code>	Gibt den kleinsten Wert in der Liste zurück.
<code>replaceAll(list, oldValue, newValue)</code>	Ersetzt alle Vorkommen von oldValue durch newValue.
<code>reverse(list)</code>	Kehrt die Reihenfolge der Elemente in der Liste um.
<code>rotate(list, distance)</code>	Verschiebt den Index jedes Elements um den angegebenen Abstand.
<code>sort(list)</code>	Platziert die Elemente der Liste in einer natürlich sortierten Reihenfolge.
<code>swap(list, index1, index2)</code>	Schaltet Elementwerte an den beiden angegebenen Indizes.

Beispiel:


```
Collections.replaceAll(list, "hello", "goodbye");
```

java.util.Collections-Klasse

Sammlung von Methoden für Collections. Folgende statischen Methoden von **Collections** können angewendet werden:

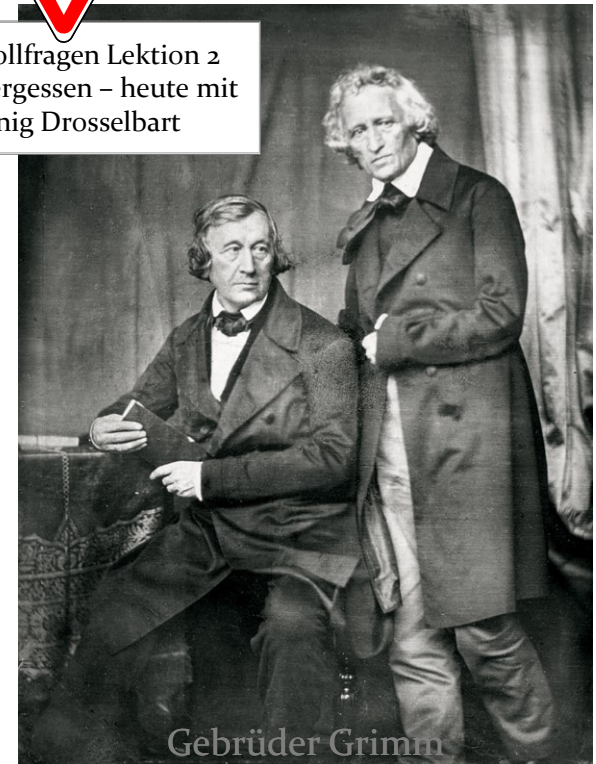


Zusammenfassung

- Einfach verkettete Liste
 - Operationen: add, remove
 - Iterator: zum Traversieren der Liste
- Doppelt verkettete Listen
- Zirkuläre, doppelt verkettete Liste
- Sortierte Listen
 - Das Comparable-Interface, die Comparator-Klasse
- Klassenhierarchie der Collection-Klassen
- Spezialfälle 
 - Read-Only
 - Thread-Safe
 - Vollständigkeit der List-Interfaces (Not Implemented)



Kontrollfragen Lektion 2
nicht vergessen – heute mit
König Drosselbart



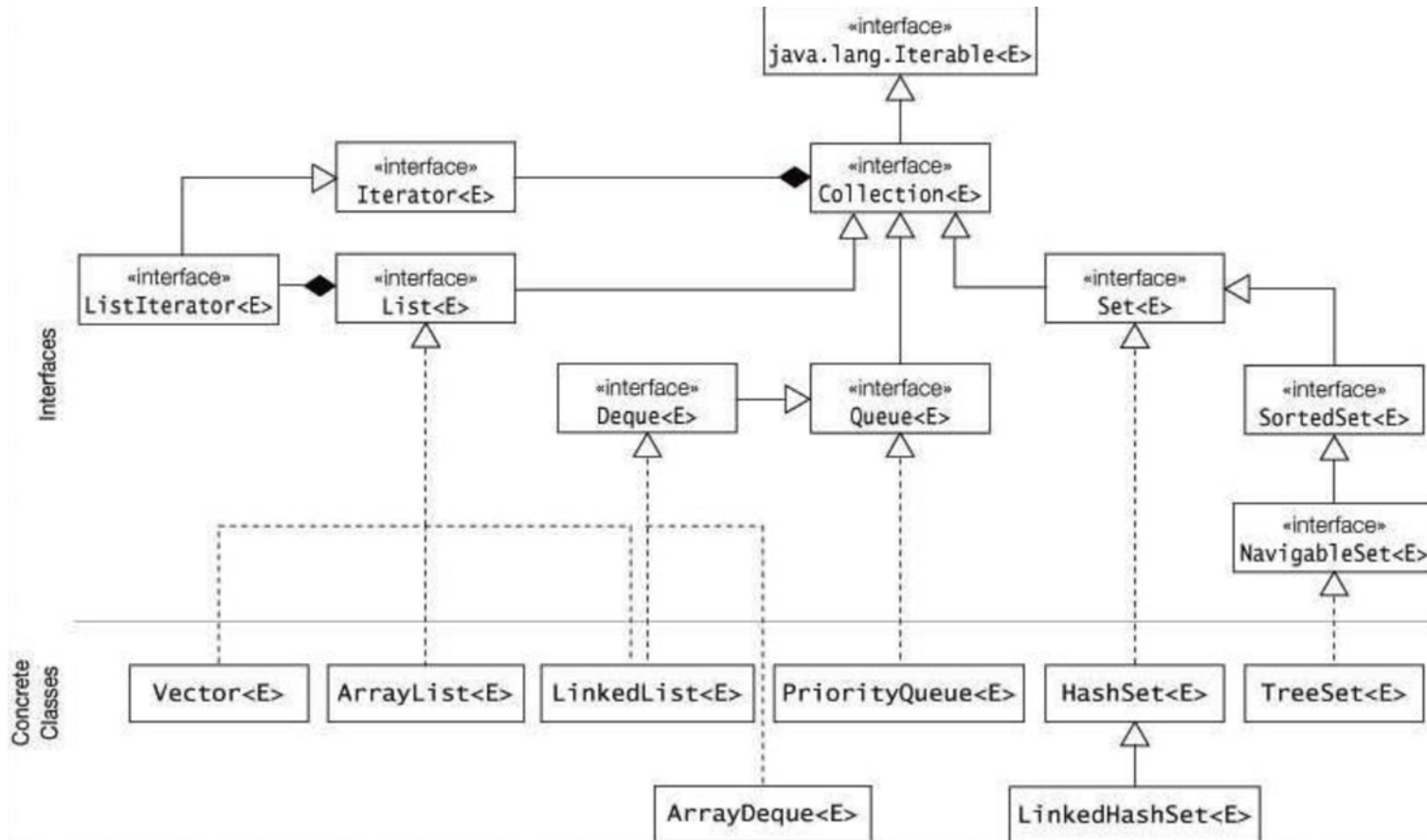


Anhang



Collections im JDK

Collections Überblick



Methoden List-Interface (1/2)



Modifier and Type	Method and Description
boolean	add(E e) Appends the specified element to the end of this list (optional operation).
void	add(int index, E element) Inserts the specified element at the specified position in this list (optional operation).
boolean	addAll(Collection<? extends E> c) Appends all of the elements in the specified collection to the end of this list, in the order that they are returned by the specified collection's iterator (optional operation).
boolean	addAll(int index, Collection<? extends E> c) Inserts all of the elements in the specified collection into this list at the specified position (optional operation).
void	clear() Removes all of the elements from this list (optional operation).
boolean	contains(Object o) Returns <code>true</code> if this list contains the specified element.
boolean	containsAll(Collection<?> c) Returns <code>true</code> if this list contains all of the elements of the specified collection.
boolean	equals(Object o) Compares the specified object with this list for equality.
E	get(int index) Returns the element at the specified position in this list.
int	hashCode() Returns the hash code value for this list.
int	indexOf(Object o) Returns the index of the first occurrence of the specified element in this list, or -1 if this list does not contain the element.
boolean	isEmpty() Returns <code>true</code> if this list contains no elements.

Methoden List-Interface (2/2)



Iterator<E>	iterator() Returns an iterator over the elements in this list in proper sequence.
int	lastIndexOf(Object o) Returns the index of the last occurrence of the specified element in this list, or -1 if this list does not contain the element.
ListIterator<E>	listIterator() Returns a list iterator over the elements in this list (in proper sequence).
ListIterator<E>	listIterator(int index) Returns a list iterator over the elements in this list (in proper sequence), starting at the specified position in the list.
E	remove(int index) Removes the element at the specified position in this list (optional operation).
boolean	remove(Object o) Removes the first occurrence of the specified element from this list, if it is present (optional operation).
boolean	removeAll(Collection<?> c) Removes from this list all of its elements that are contained in the specified collection (optional operation).
default void	replaceAll(UnaryOperator<E> operator) Replaces each element of this list with the result of applying the operator to that element.
boolean	retainAll(Collection<?> c) Retains only the elements in this list that are contained in the specified collection (optional operation).
E	set(int index, E element) Replaces the element at the specified position in this list with the specified element (optional operation).
int	size() Returns the number of elements in this list.
default void	sort(Comparator<? super E> c) Sorts this list according to the order induced by the specified Comparator .
default Spliterator<E>	spliterator() Creates a Spliterator over the elements in this list.
List<E>	subList(int fromIndex, int toIndex) Returns a view of the portion of this list between the specified fromIndex , inclusive, and toIndex , exclusive.
Object[]	toArray() Returns an array containing all of the elements in this list in proper sequence (from first to last element).
<T> T[]	toArray(T[] a) Returns an array containing all of the elements in this list in proper sequence (from first to last element); the runtime type of the returned array is that of the specified array.

Methoden ListIterator-Interface



Modifier and Type	Method and Description
void	add(E e) Inserts the specified element into the list (optional operation).
boolean	hasNext() Returns true if this list iterator has more elements when traversing the list in the forward direction.
boolean	hasPrevious() Returns true if this list iterator has more elements when traversing the list in the reverse direction.
E	next() Returns the next element in the list and advances the cursor position.
int	nextIndex() Returns the index of the element that would be returned by a subsequent call to next() .
E	previous() Returns the previous element in the list and moves the cursor position backwards.
int	previousIndex() Returns the index of the element that would be returned by a subsequent call to previous() .
void	remove() Removes from the list the last element that was returned by next() or previous() (optional operation).
void	set(E e) Replaces the last element returned by next() or previous() with the specified element (optional operation).

Funktionen

- Elementeinfügung
- Austausch
- bidirektionalen Zugriff

ListIterator wird von List erzeugt:

```
public ListIterator listIterator(int index)
```

Achtung: Index bezeichnet das erste Element welches beim erstmaligen Aufruf von **next** zurückgegeben wird.

Ist der Index ausserhalb des zulässigen Bereichs ($\text{index} < 0 \parallel \text{index} > \text{size}()$) wird eine **IndexOutOfBoundsException** geworfen.



Wrapper Klassen

Collection: Thread-Safety und Synchronized



Problem:

Wenn mehrere Threads gleichzeitig z.B. gleiches Element entfernen passiert ein Unglück, z.B. kann Listen-Datenstruktur inkonsistent werden

Lösung:

- Thread-Safe
 - mehrere Threads können z.B. gleichzeitig auf `remove` Methode zugreifen
 - in Java einfach mit `synchronized` vor z.B. `remove` Methode \Rightarrow nur ein Thread darf gleichzeitig in der Methode sein
 - Nachteil:
 - meist nicht nötig
 - andere Threads werden u.U. behindert
 - `synchronized` kostet
- Neue Collection Klassen sind alle non-synchronized
Müssen mit `Collections.synchronizedList()` bei Bedarf Thread-Safe gemacht werden:

```
List list = new LinkedList()  
list = Collections.synchronizedList(list);
```



Collection: Read-Only

Problem:

Listen sollen vor unbeabsichtigter Veränderung geschützt werden

Lösung:

können mit `Collections.unmodifiableList()` unveränderbar gemacht werden.

```
List list = new LinkedList()  
list = Collections.unmodifiableList(list);
```

Bemerkung: analoge Methoden für `Set`, `SortedSet`, `Map`, `SortedMap`

List-Interface: Not Implemented



Problem:

Das List Interface ist gross und einige Methoden machen für gewisse Implementation keinen Sinn.

Lösung:

Es wird die `UnsupportedOperationException` von diesen Methoden geworfen.