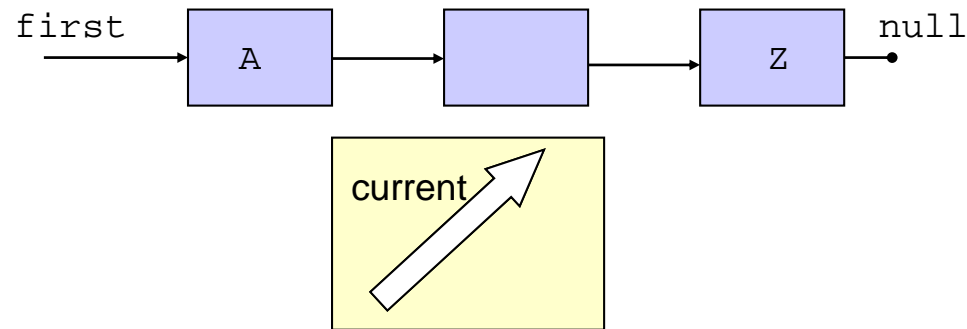
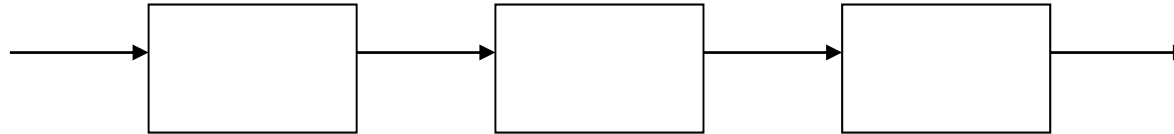


# Einfach und mehrfach verkettete Listen



- Sie wissen, was einfach und mehrfach verkettete 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 verkettet, zirkulär und sortiert
- Sie kennen die Schnittstellen *Comparable* und *Comparator* und können damit umgehen.
- Sie können mit den Java Collection Klassen umgehen

# Listen



- Abstrakter Datentyp, der eine Liste von Objekten verwaltet.
- Liste ist eine der grundlegenden Datenstrukturen in der Informatik, neben den Arrays
- definiert im Interface `java.util.List`
- implementiert in: `java.util.LinkedList` und `java.util.ArrayList`

## minimaler Satz von Operationen

Funktionskopf

`boolean add(E obj)`

`void add (int index, E obj)`

`void add(0, E obj)`

`E get (int i)`

`E get (0)`

`E get (l.size()-1)`

`E remove (int i)`

`E remove (0)`

`E remove (l.size()-1)`

`int size()`

`boolean isEmpty()`

Beschreibung

Fügt *obj* am Schluss der Liste an

Fügt *obj* an der Stelle *index* ein

Fügt *obj* am Anfang ein

Gibt Element an Stelle *i* zurück

Gibt erstes Element zurück

Gibt letztes Element zurück

Entfernt das *i*-te Element und gibt es als Rückgabewert zurück

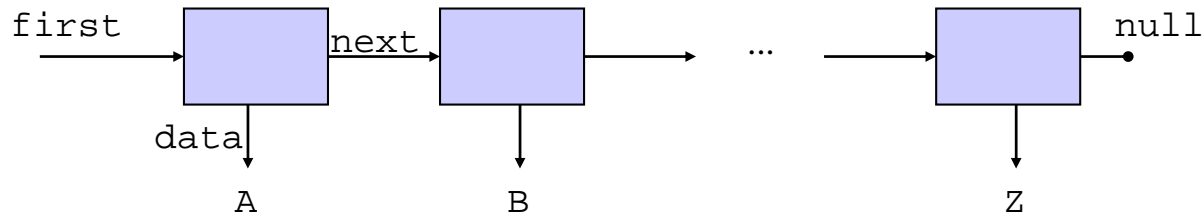
Entfernt das erste Element

Entfernt das letzte Element

Gibt Anzahl Elemente zurück

Gibt `true` zurück, falls die Liste leer ist

# Datenstruktur des Listenknotens, addFirst, addLast



Daten der Liste

```

class ListNode<E> {
    private E data;
    public ListNode<E> next;

    ListNode(E o) {
        data = o;
        next = null;
    }
}
  
```

```

class LinkedList<E> {
    private ListNode<E> first;

    public void addLast (E o) {
        ListNode<E> n = new ListNode<E>(o);
        if (first == null)
            first = n;
        else {
            ListNode<E> gf = first;
            while(gf.next!=null) gf=gf.next;
            gf.next = n;
        }
    }

    public void addFirst (E o){
        ListNode<E> n = new ListNode<E>(o);
        n.next = first;
        first = n;
    }
}
  
```

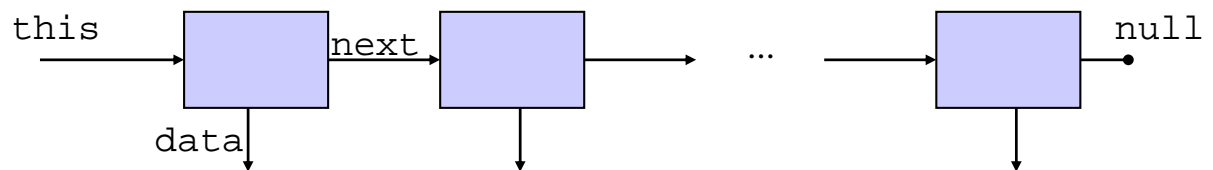
Referenz auf nächsten ListNode



## removeFirst, removeLast, getFirst, getLast

```
E removeFirst () {  
    E temp;  
    if (next != null) {  
        temp = data;  
        data = next.data;  
        next = next.next;  
    }  
    else { // next==null  
        E temp = data;  
        data = null;  
    }  
    return temp;  
}
```

Java: Brauchen uns nicht um die Entsorgung des gelöschten Elements zu kümmern.



**Übung:** Schreiben Sie die Methoden removeLast (getFirst, getLast)

# Einfügen in eine Liste: add

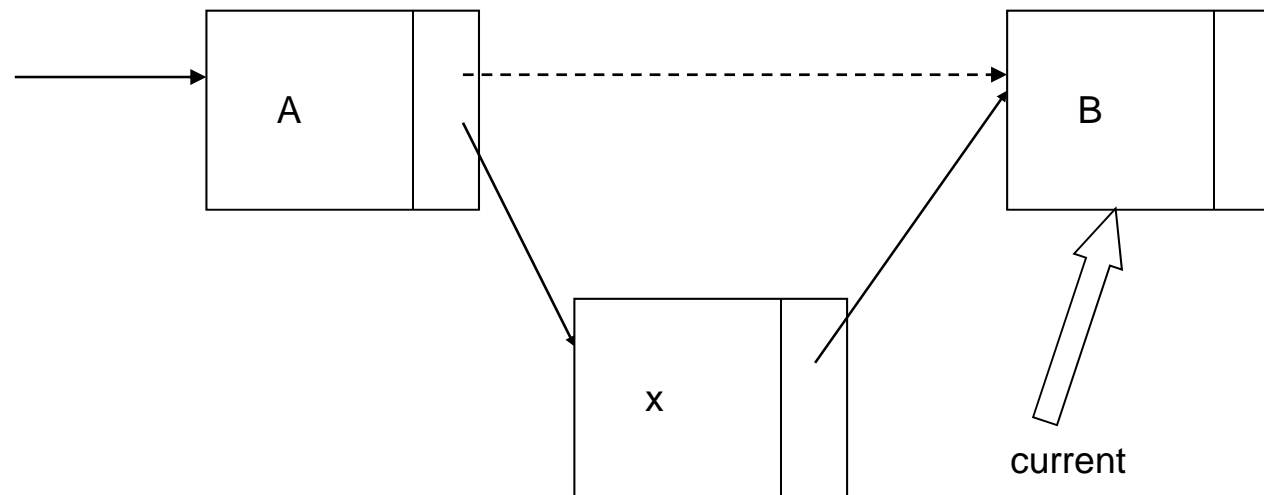
## Operationen

Methode

`void add (x)`

Beschreibung

fügt x vor current ein



Einfügen:

ein Element wird  
zwischen zwei  
Elemente eingefügt

Übung: Schreiben Sie die Methode `add (int index, Object x)`

# Löschen eines Objekts: remove

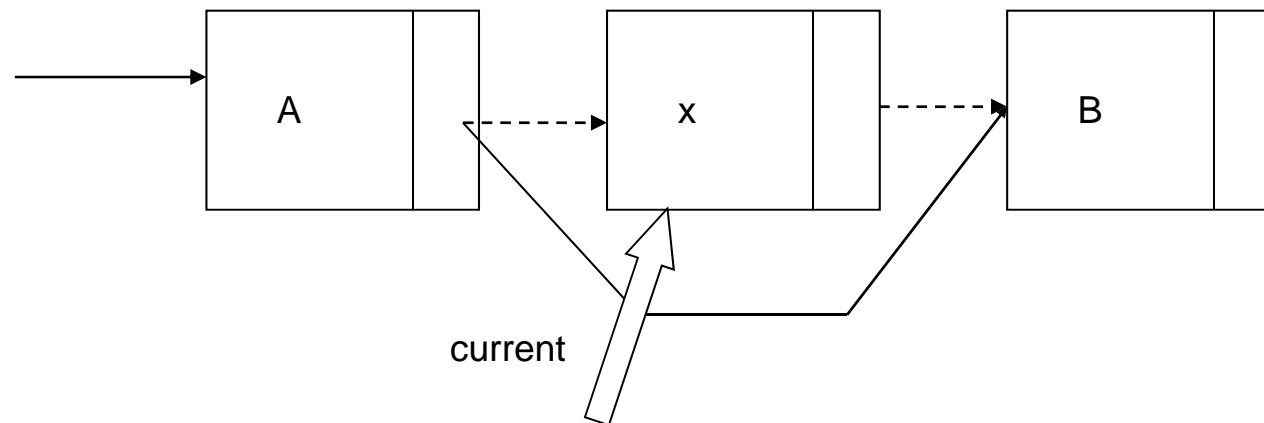
## Operationen

Methode

Beschreibung

`void remove`

löscht Element auf das *current* zeigt



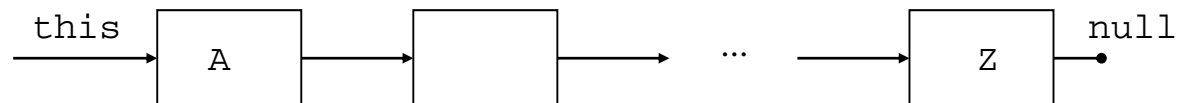
`a.next = a.next.next`

Löschen:

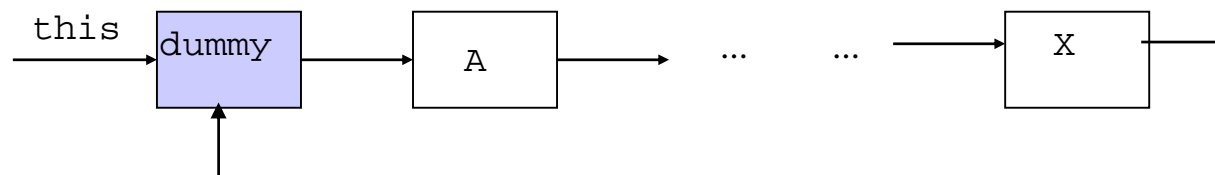
das zu löschende  
Element wird  
"umgangen"

Frage: wie findet man das Vorgänger-Objekt ?

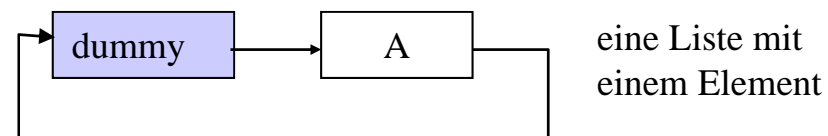
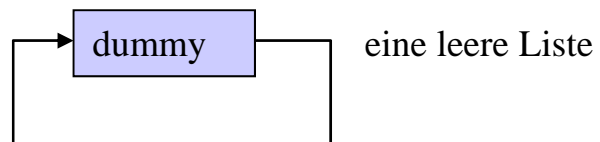
## Mitte, Anfang und Ende der Liste



- 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.
- Einführen eines Dummy-Nodes. Das letzte Element zeigt wieder auf diesen.

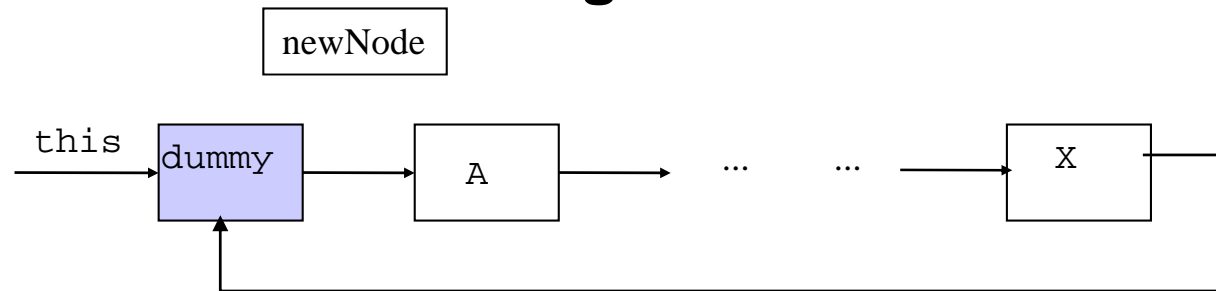


- Damit wird die Liste zu einem Ring, das erste und letzte Element sind keine Spezialfälle mehr.





# Ringliste



```

class LinkedList<E> {
    private E data;
    private LinkedList<E> next;

    LinkedList() {
        next = this;
        data = null;
    }

    void addFirst(E o) {
        LinkedList<E> newNode =
            new LinkedList<E>();
        newNode.data = o;
        newNode.next = next;
        next = newNode;
    }
}
    
```

```

void addLast(E o) {
    LinkedList<E> temp = this;
    while (temp.next!=this)
        temp = temp.next;
    LinkedList<E> newNode =
        new LinkedList<E>();
    newNode.data = o;
    newNode.next = this;
    temp.next = newNode;
}
}
    
```

**Frage:** Ist dies eine gute Implementation für eine Queue?

# Doppelt verkettete Listen

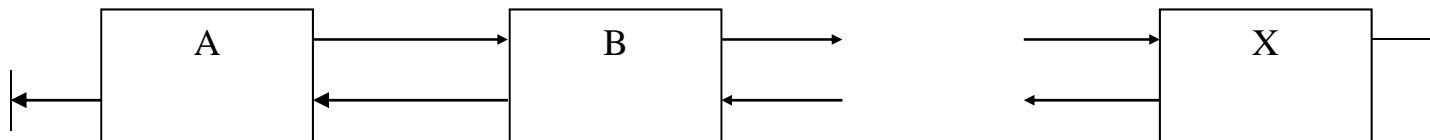
folgende Probleme treten bei einfach verketteten Listen auf:

- Der Zugang zum Listenende kostet viel Zeit (im Vergleich mit einem Zugriff auf den Listenanfang)
- Man kann sich mit `next ( )` nur in einer Richtung effizient durch die Liste “hangeln”, die Bewegung in die andere Richtung ist ineffizient.

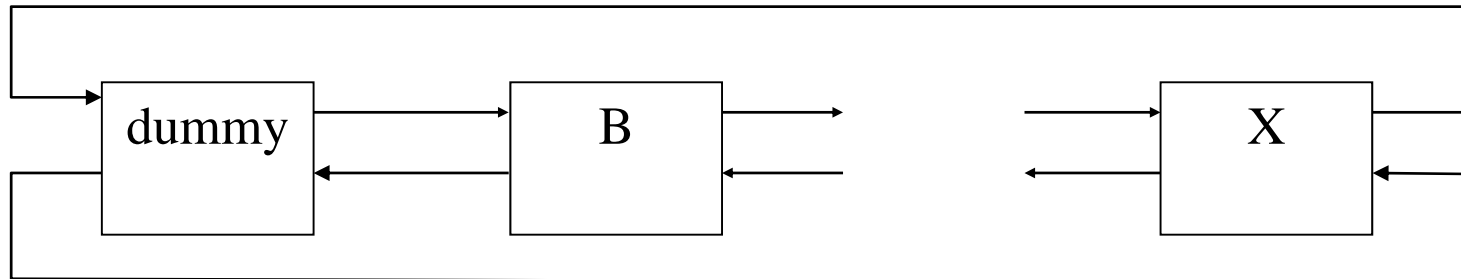
```
class List<E> {  
    E      data;  
    List<E> next, prev;  
}
```

**symmetrisch aufbauen:**

jeder Knoten hat zwei Referenzen **next** und **previous**



# Doppelt verkettete Ringliste



## Operationen bei doppelt verketteten Listen: add

## Operationen

## Methode

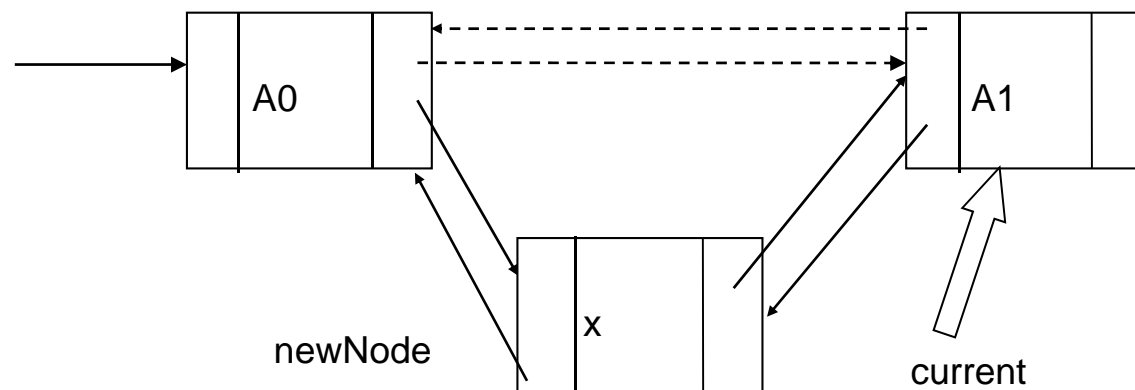
```
void add (x)
```

## Beschreibung

fügt x ein

Nachteil (gegenüber einfach verkettet):  
mehr Anweisungen

Vorteil:  
add-Operation ist an jeder  
Stelle einfach möglich



```
newNode.next = current;
newNode.prev = current.prev;
current.prev.next = newNode;
current.prev = newNode;
```

# Operationen bei doppelt verketteten Listen: remove

## Operationen

Methode

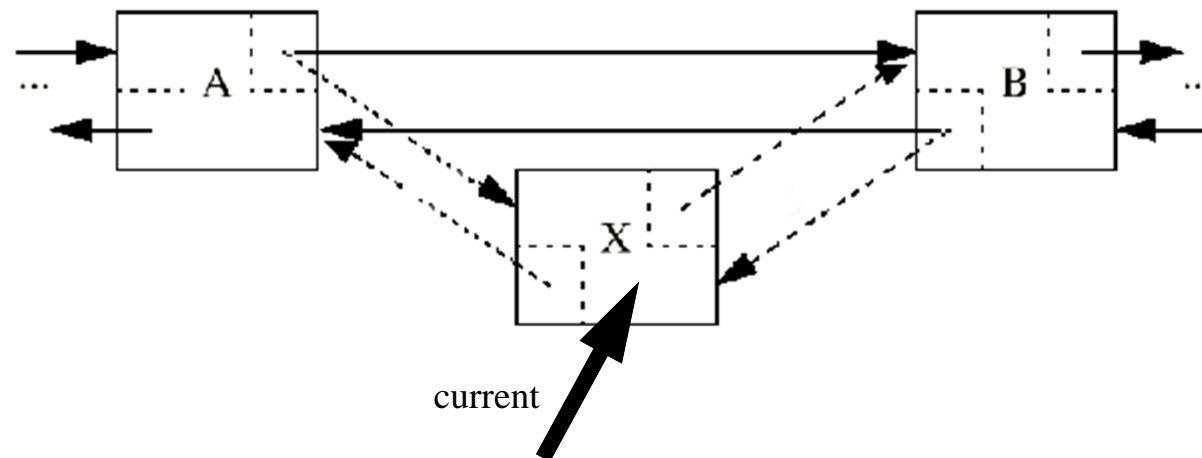
`void remove (x)`

Beschreibung

löscht x

Vorteil:

Remove-Operation ist  
nun sehr einfach

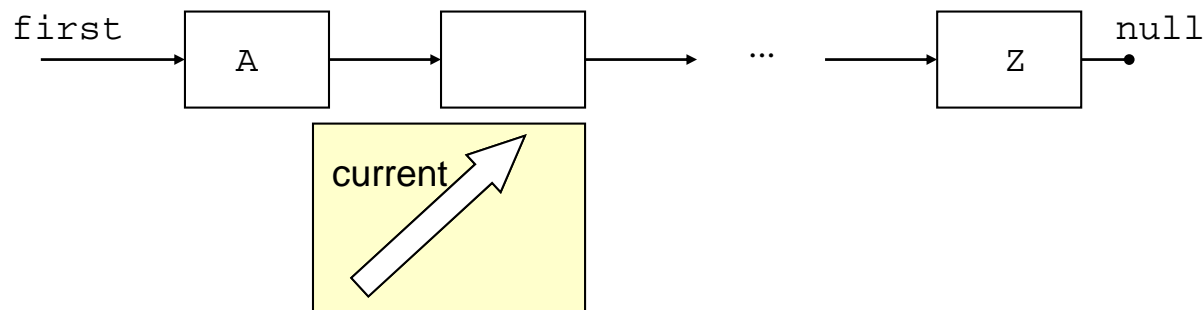


```
current.prev.next = current.next;  
current.next.prev = current.prev;  
current = current.next;
```

# Übung

1. Konstruieren Sie eine leere, doppelt verkettete Ringliste.
2. Programmieren Sie die *remove* Methode für die doppelt verkettete Ringliste
3. Programmieren Sie die *add* Methode für die doppelt verkettete Ringliste (*addAtEnd()*)

# Das Konzept des Iterators



- Der Iterator ist ein ADT, mit dem eine Datenstruktur, z.B. Liste, traversiert werden kann, ohne dass die Datenstruktur bekannt gemacht werden muss: *Information Hiding*.
- Der Iterator kennt seinen Container und es wird ein privater (current) Zeiger auf die aktuelle Position geführt.

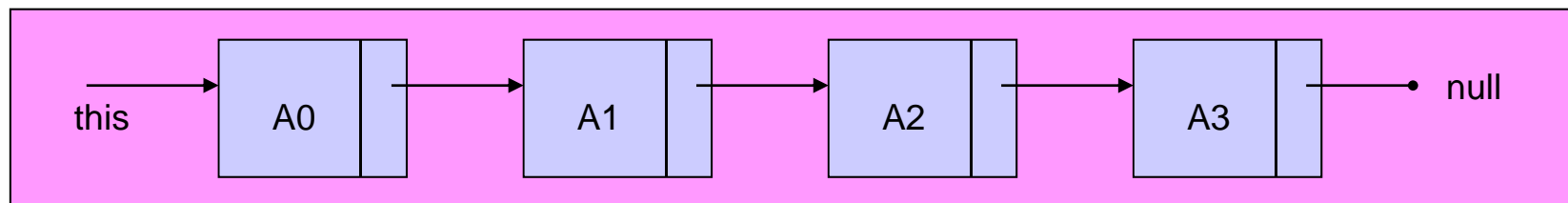
Augu

```
interface Iterator<E> {  
    boolean hasNext(); // es hat weitere Elemente  
    E next();          // liefere nächstes Element (beim 1.  
                      // Aufruf das erste)  
    void remove();     // lösche das Element  
                      // das zurückgegeben wurde  
}
```

# Iteratoren, Aufteilung der Funktionalität in Klassen

## I. **LinkedList<E>** implements **List<E>**

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



## II. **ListIterator<E>** implements **Iterator<E>**

- ermöglicht das Iterieren durch die Liste (ohne Verletzung des Information Hiding-Prinzips)

- verwaltet eine aktuelle Position:

```
private LinkedList<E> current
```

- hat eine Referenz auf seinen Container

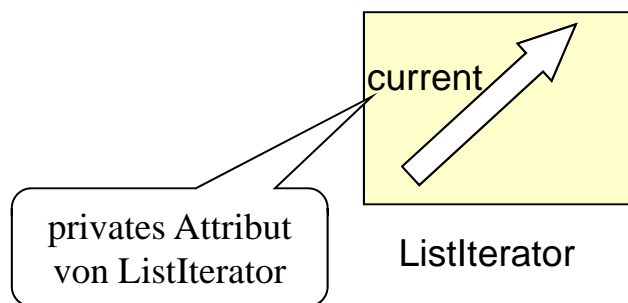
- definiert Methoden um:

- die aktuelle Position zu verschieben → `E next()`

- abzufragen, ob man am Ende angelangt ist

- `boolean hasNext()`

- gleichzeitig mehrere Iteratoren auf die gleiche Liste ansetzbar.





## ListIterator und Listen

```
class ListIterator<E> implements Iterator<E> {  
    private LinkedList<E> root;  
    private LinkedList<E> current;  
    public ListIterator(LinkedList<E> root) {  
        this.root = root;  
        current = root;  
    }  
    public E next() {  
        if (current == null) throw new  
            NoSuchElementException();  
        E temp = current.data;  
        current = current.next;  
        return temp;  
    }  
    ...  
}
```

## ListIterator und Listen, Anwendung

```
List<Integer> l = new LinkedList<Integer>();  
// Fill the list  
...  
  
Iterator<Integer> iter = l.iterator();// Gib mir einen  
                                     // Iterator  
while (iter.hasNext() {           // sind wir schon am Ende?  
    Integer i = iter.next(); // gibt den Wert zurück, auf  
                             // den current zeigt und  
                             // inkrementiert current  
    System.out.println(i);  
}
```

## ListIterator und Listen

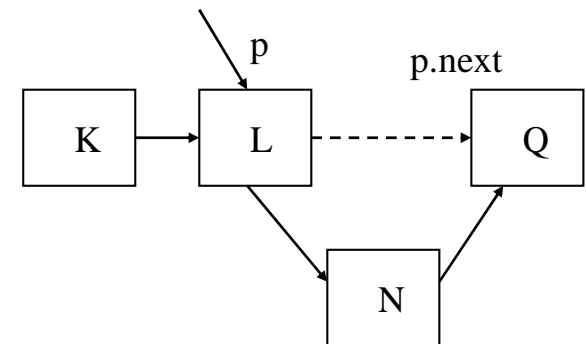
```
class LinkedList<E> {  
    ....  
    Iterator<E> iterator() {  
        // liefert einen Iterator auf den Anfang der Liste  
        return  
            new ListIterator<E>(this);  
    }  
    ....  
}
```

Übung: Wie würde ein ArrayIterator aussehen?

## 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
  - suche vor dem Einfügen die richtige Position  

```
while (p.next.data < n.data) p = p.next;
```
- Anwendung:
  - überall wo sortierte Datenbestände verwendet werden, z.B. PriorityQueue



## Sortierte Listen 2 - Comparable Interface

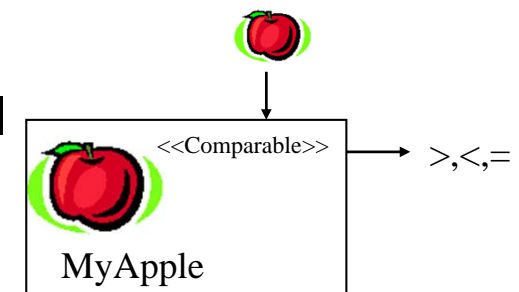
- Problem: Wie vergleicht man Objekte miteinander?  
  
⇒ Es muss etwas geben, das eine Bewertung von 2 Elementen bezüglich  $>$ ,  $=$ ,  $<$  erlaubt ... (Ordnungsrelation)
- Lösung: Das Interface `java.lang.Comparable` ist vorgesehen, zum Bestimmen der relativen (natürlichen) Reihenfolge von Objekten.

Bsp: `x.compareTo(y)`                      // if ( $x < y$ ) ⇒ negative Zahl

   // if ( $x == y$ ) ⇒ 0

   // if ( $x > y$ ) ⇒ positive Zahl

```
public interface Comparable<E> {  
    public int compareTo(E o);  
}
```



## Beispiel Comparable

```
class Apple implements Comparable<Apple> {  
    int value;  
  
    public int compareTo(Apple a) {  
        return this.value - a.value;  
    }  
}
```

## Sortierte Listen 3

Bei geordneten Listen muss eine Ordnung bezüglich der Elemente definiert sein.

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

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

**Lösung 1:** Listen, die aus Elementen bestehen, welche dieses Interface implementieren, können mit **Collections.sort** (statische Methode) automatisch sortiert werden.



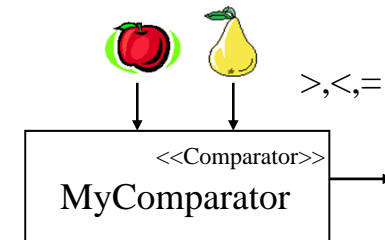
**Collections** != Collection

```
Collections.sort(List<E> list);
```

## Sortierte Listen 4

- Was macht man, wenn es gar keine *natürliche Reihenfolge* der Elemente gibt?
- Zum Beispiel eine Liste von Personen, die nach verschiedenen Kriterien (Name, Wohnort, Grösse) sortiert werden soll.
- **Lösung 2:** Das Interface ***java.util.Comparator*** ist vorgesehen für Objekte ohne natürliche Reihenfolge, oder wenn Objekte nach verschiedenen Kriterien sortiert werden sollen. z.B. Farbe, mit/ohne Wurm, etc.

```
public interface Comparator<E> {  
    public int compare(E o1, E o2);  
    public boolean equals(Object o);  
}
```





- Trick: Ein Comparator-Objekt (Object einer Klasse welche Comparator implementiert) wird bei einem Methodenaufruf übergeben

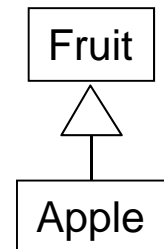
```
Collections.sort(List<E> list, Comparator<? super E> comp);
```



## Beispiel Comparator

```
class FruitComparator implements Comparator<Fruit> {  
       
    int compare(Fruit f1, Fruit f2) {  
        return f1.value - f2.value;  
    }  
}
```

```
List<Fruit> fruechtekorb = new LinkedList<Fruit>;  
...  
Collections.sort(fruechtekorb, new FruitComparator());  
  
List<Apple> apfelkorb = new LinkedList<Apfel>;  
...  
Collections.sort(apfelkorb, new FruitComparator());
```



## Vergleich Liste und Array

- **Array:** Teil der Java Sprache:
  - Benutzung sehr einfach
  - 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`, und `Vector`
  - nicht ganz so einfach in der Benutzung
  - Anzahl Elemente zur Erstellungszeit **nicht** definiert: `new LinkedList();`
  - Operationen:
    - Indizierter Zugriff möglich aber **ineffizient**: `list.get(i)`
    - Anfügen, Ersetzen, Vertauschen und **Einfügen** und **Löschen** von Elementen

# Wo werden Listen angewendet?

Listen werden angewendet für:

als grundlegende Struktur für Stack, Queue, etc.

- wenn Anzahl der Elemente a priori unbekannt
- Reihenfolge/Position relevant ist
- Einfügen und Löschen von Elementen in der Mitte häufig ist

Speicherverwaltung

- Liste der belegten Blöcke

Betriebssysteme

- Disk-Blöcken, Prozesse, Threads,

Editoren

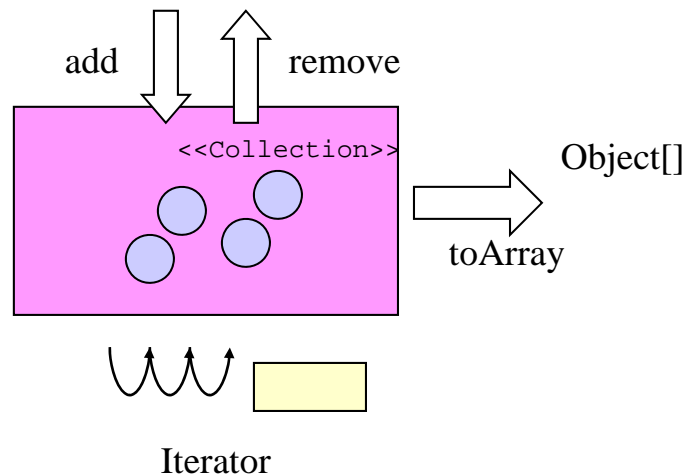
- Liste von Textstücken gleichen Fonts und Style

# Das java.util.Collection Interface

- Gemeinsames Interface für Sammlungen (Collections) von Objekten - Ausnahme Array - leider.

Abstrakter Datentyp für beliebige Objektbehälter.

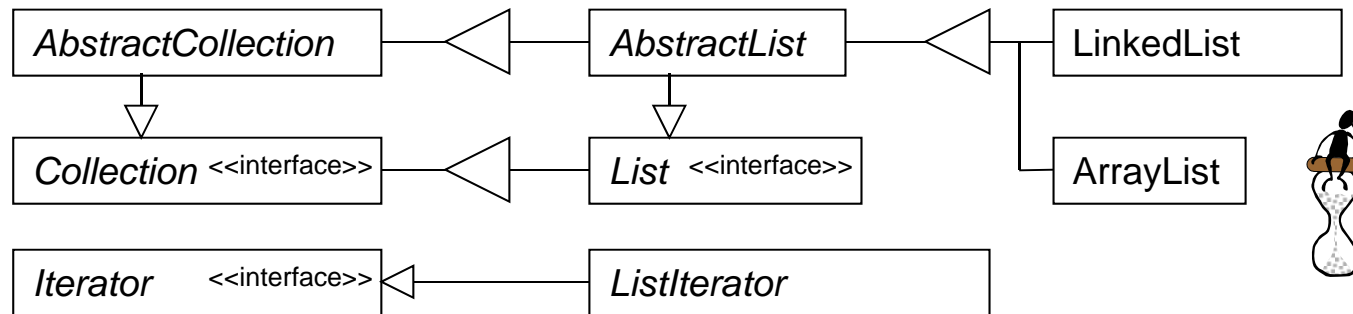
Die *add()* Methode fügt ein Element an der "natürlichen" Position hinzu.



## Operationen

Funktionskopf	Beschreibung
<code>boolean add(E x)</code>	Fügt x hinzu
<code>boolean remove (Object x)</code>	löscht das Element x
<code>boolean removeAll()</code>	löscht ganze Collection
<code>Object[] toArray()</code>	schreibt die Collection in einen Array
<code>Iterator&lt;E&gt; iterator()</code>	gibt Iterator auf Collection zurück
<code>int size()</code>	Gibt Anzahl Element zurück
<code>boolean isEmpty()</code>	Gibt true zurück, falls die Collection leer ist

# Die Klasse `java.util.LinkedList`, Klassenhierarchie



# Das java.util.List Interface

Abstrakter Datentyp der eine Liste von Elementen verwaltet.

- definiert in `java.util.List`
- implementiert in:  
`java.util.LinkedList`

## Wichtigste Operationen

### Konstruktoeren

<code>LinkedList&lt;E&gt; ();</code>	erzeuge Liste
<code>LinkedList&lt;E&gt; (Collection&lt;E&gt; c);</code>	erzeuge Liste mit Elementen der Collection

### Methoden

<code>boolean add(E obj)</code>	Fügt <i>obj</i> am Schluss der Liste an
<code>void add (int index, E o)</code>	Fügt <i>obj</i> an der Stelle <i>index</i> ein
<code>void add(0,obj)</code>	Fügt <i>obj</i> am Anfang ein

<code>E get (int i)</code>	Gibt Element an Stelle <i>i</i> zurück
<code>E get (0)</code>	Gibt erstes Element zurück
<code>E get (l.size()-1)</code>	Gibt letztes Element zurück

<code>boolean remove (Object x)</code>	löscht das Element <i>x</i>
<code>E remove (int i)</code>	Entfernt das <i>i</i> -te Element und gibt es als Rückgabewert zurück
<code>E remove (0)</code>	Entfernt das erste Element
<code>E remove (l.size()-1)</code>	Entfernt das letzte Element

<code>int size()</code>	Gibt Anzahl Element zurück
<code>boolean isEmpty()</code>	Gibt true zurück, falls die Liste leer

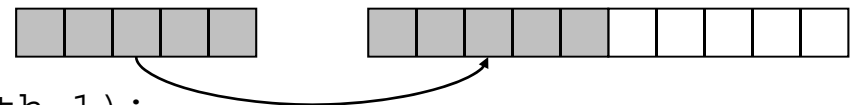
## Konkrete Klassen die List Interface implementieren

- *Vector*
  - ab JDK 1.0 vorhanden, Implementation als Array
  - -> indexierter Zugriff schnell, Einfügen und Löschen langsam
  - synchronized Aufrufe.
  - Ist veraltet.
- *ArrayList*
  - Ähnlich wie Vector
  - non-synchronized Aufrufe
- *LinkedList*
  - Implementation als Liste
  - → indexierter Zugriff langsam, Einfügen und Löschen schnell
  - non-synchronized Aufrufe

## ArrayList Implementation

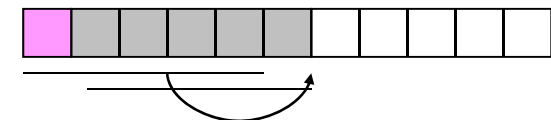
- Wenn mehr Elemente gespeichert werden sollen als im Array Platz haben, muss ein neuer Array erstellt werden und es müssen die Elemente umkopiert werden (gute Strategie Länge \* 2).

```
aNeu = new Object[a.length * 2];  
System.arraycopy(a, 0, aNeu, 0, a.length - 1);  
a = aNeu;
```



- Beim Einfügen am Anfang der Liste müssen alle nachfolgenden Elemente im Array umkopiert werden

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



- Frage: ist ArrayList für Queue oder Stack geeignet?



# Collections Framework

- **Vector, Stack, Enumeration**
- 'alte' Collection Klassen (JDK 1.0)  
– im wesentlichen **Vectors** und **Stacks**
- Interface **Enumeration** - stellt eine gute Abstraktion dar, wenn es nicht von Bedeutung ist, wie eine Collection traversiert wird oder wie genau die darunter liegende Datenstruktur aussieht
- sollten **nur noch für die Wartung** von Java 1.0 und 1.1 Programmen eingesetzt werden.

## Das Java Collections API Framework

In Java 1.2 wurden die Collection Klassen durch ein völlig **neues Set** von rund 25 Klassen im java.util-Paket ersetzt.

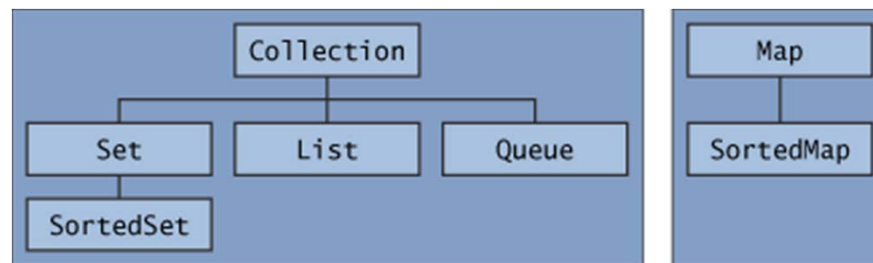
Die vier Interfaces

**Collection,**  
**List,**  
**Queue,**  
**Set,**  
**Map** (später)

definieren die verschiedenen Arten von Collections.

# Collections Framework

- In Java 1.5 wurde das Collections Framework noch einmal völlig überarbeitet.
- Alle Collection-Klassen sind generisch.
- Die folgenden Interfaces werden definiert:



- *Collection*: Ein Behälter, Sammlung von Elementen
- *Set*: Menge; kein Element doppelt vorhanden, keine Ordnung
- *List*: geordnete Sammlung von Elementen
- *Queue*: geordnete Sammlung von Elementen, typischerweise als FIFO-Speicher
- *Map*: Sammlung von Schlüssel und Wertepaaren. Schlüssel kommen nicht dupliziert vor

# Collections Framework

Die Implementationen werden aufgeteilt in:

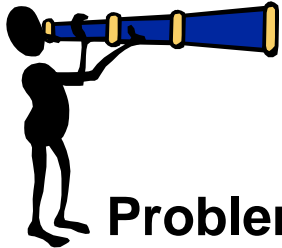
- **General-purpose implementations:** the most commonly used implementations, designed for everyday use.
- **Special-purpose implementations:** designed for use in special situations. They display nonstandard performance characteristics, usage restrictions, or behavior.
- **Concurrent implementations:** designed to support high concurrency, typically at the expense of single-threaded performance. These implementations are part of the package `java.util.concurrent`.
- **Wrapper implementations:** used in combination with other types of implementations (often the general-purpose implementations) to provide added or restricted functionality.
- **Convenience implementations:** mini-implementations typically made available via static factory methods, that provide convenient, efficient alternatives to the general-purpose implementations for special collections (such as singleton sets).
- **Abstract implementations:** skeletal implementations that facilitate the construction of custom implementations.

# Collections Framework

## General Purpose Implementations

Interfaces	General-purpose Implementations				
	Hash Table	Resizable array	Tree	Linked List	Hash Table + Linked List
Set	HashSet		TreeSet		LinkedHashSet
List		ArrayList		LinkedList	
Queue					
Map	HashMap		TreeMap		LinkedHashMap

- Eine Beschreibung aller Collection-Klassen würde zu weit führen.  
Eine gute Beschreibung findet man unter  
<http://java.sun.com/j2se/1.5.0/docs/guide/collections/index.html>



# Thread-Safe und Synchronized

## Problem

- wenn mehrere Threads gleichzeitig z.B. gleiches Element entfernen, passiert ein Unglück, z.B. kann eine Listen-Datenstruktur inkonsistent werden.
- **Thread-Safe**
  - mehrere Threads können gleichzeitig auf z.B. remove Methode zugreifen.
  - in Java einfach mit synchronized vor z.B. remove Methode → nur ein Thread darf gleichzeitig in der Methode sein
  - Nachteil:
    - meist nicht nötig
    - andere Threads werden u.U. behindert
    - synchronized kostet was
- **Neue Collection Klassen sind alle non-synchronized**
- Können mit `Collections.synchronizedList()` bei Bedarf Thread-Safe gemacht werden:

```
List<Type> list = Collections.synchronizedList(  
                                new ArrayList<Type>());
```

# Read-Only und Not Implemented

## Problem

- Listen sollen vor unbeabsichtigter Veränderung geschützt werden

## Lösung

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

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

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

## Problem

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

## Lösung

- Es wird die `UnsupportedOperationException` von diesen Methoden geworfen

## Zusammenfassung

- Datenstruktur einer Liste
- Operationen auf Listen
- Zirkuläre und doppelt verkettete Liste
- Iteratoren: zum Traversieren der Liste
- Sortierte Listen
  - Das *Comparable* und das *Comparator* Interface
- List Interface und Implementationen
- Collections Framework
  - Spezialfälle
    - Thread-Safe, Read-Only
    - Vollständigkeit der List-Interfaces