

Listen

Martin Wirsing

in Zusammenarbeit mit
Michael Barth, Philipp Meier und Gefei Zhang

02/05

Ziele

- Standardimplementierungen für Listen kennenlernen
- Listeniteratoren verstehen

M. Wirsing: Listen

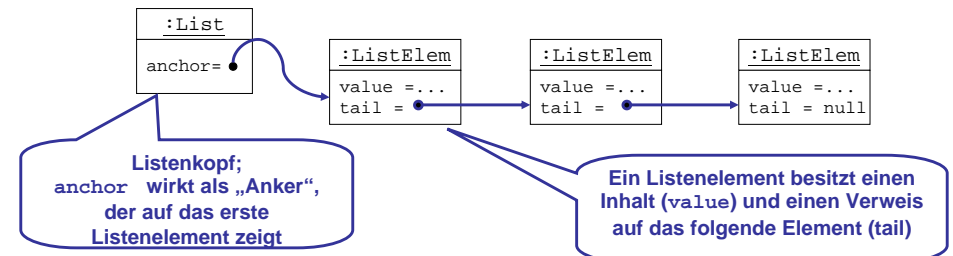
Die Rechenstruktur der Listen

- Eine **Liste** ist eine **endliche Sequenz von Elementen**, deren Länge (im Gegensatz zu Reihungen) durch Hinzufügen und Wegnehmen von Elementen geändert werden kann.
- **Standardoperationen für Listen** sind:
 - Löschen aller Elemente der Liste
 - Zugriff auf und Änderung des ersten Elements
 - Einfügen und Löschen des ersten Elements
 - Prüfen auf leere Liste, Suche nach einem Element
 - Berechnen der Länge der Liste, Revertieren der Liste
 - Listendurchlauf
- Die **Javabibliothek** bietet Standardschnittstellen und -Klassen für Listen an:
 - interface List, class LinkedList, ArrayList
 die weitere Operationen enthalten, insbesondere den direkten Zugriff auf Elemente durch Indizes wie bei Reihungen
 - **! Problematisch: Führt zur Vermischung von Reihung und Liste**

M. Wirsing: Listen

Listenimplementierung: Einfach verkettete Listen

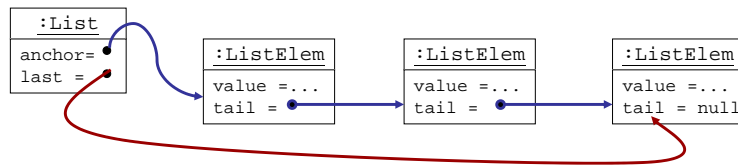
- Eine einfach verkettete Liste ist eine Sequenz von Objekten, wobei jedes Element auf seinen Nachfolger in der Liste zeigt.
- Unterschiedliche Implementierungen:
 1. Realisierung des Anfügens vorne in konstanter Zeit:



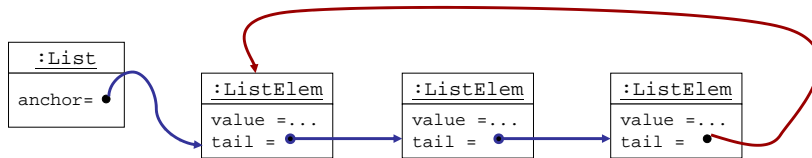
M. Wirsing: Listen

Einfach verkettete Listen

2. Realisierung des Anfügens vorne und hinten in konstanter Zeit:

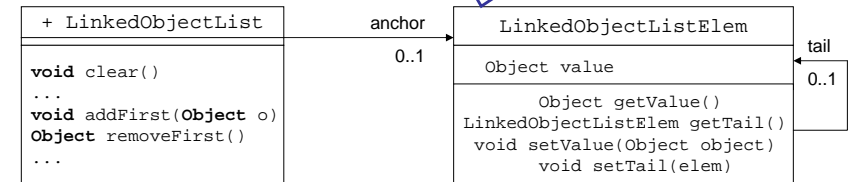


3. Zirkuläre Liste:



Einfach verkettete Listen: UML-Entwurf

Nicht public!! Dadurch ist die Klasse nur im gleichen Paket sichtbar, aber nicht für externe Benutzer



```
anchor = new LinkedObjectListElem(o, anchor);
```

Einfach verkettete Listen in Java

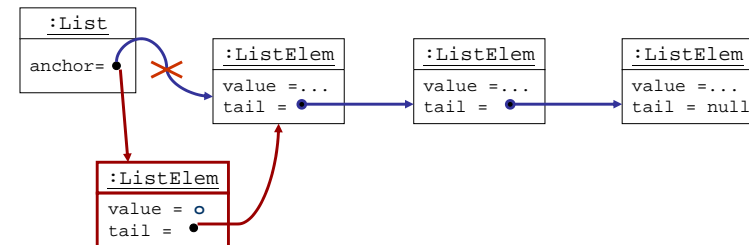
```
public class LinkedObjectList
{ private LinkedObjectListElem anchor;
  ...
}
```

Nicht public!! Dadurch ist die Klasse nur im gleichen Paket sichtbar, aber nicht für externe Benutzer

```
class LinkedObjectListElem
{ private Object value;
  private LinkedObjectListElem tail;
  LinkedObjectListElem getTail() { return tail; }
  void setTail(LinkedObjectListElem elem) { tail = elem; }
  ...
}
```

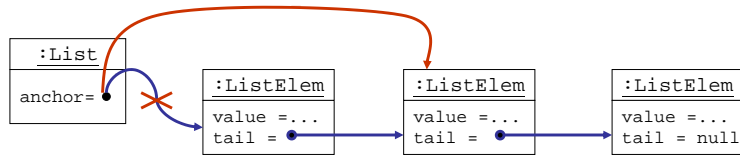
Ersetzt das nächste Listenelement durch elem und ändert dadurch den Rest der Liste (nicht für externe Benutzer!)

Einfügen eines Objekts o am Anfang der Liste



```
public void addFirst(Object o)
{
  anchor = new LinkedObjectListElem(o, anchor);
}
```

Entfernen des ersten Elements

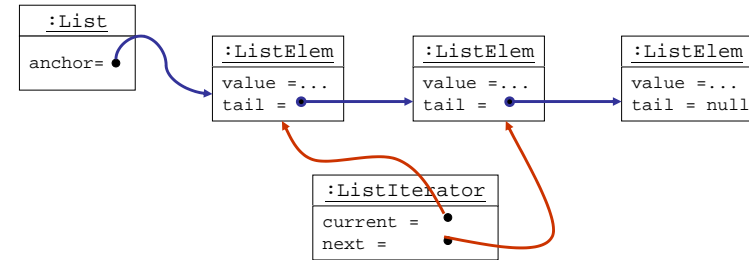


```
public Object removeFirst() throws NoSuchElementException
{
    if (anchor == null)
    {
        throw new NoSuchElementException();
    }
    else
    {
        Object result = anchor.getValue();
        anchor = anchor.getTail();
        return result;
    }
}
```

Ausnahme, wenn Liste leer

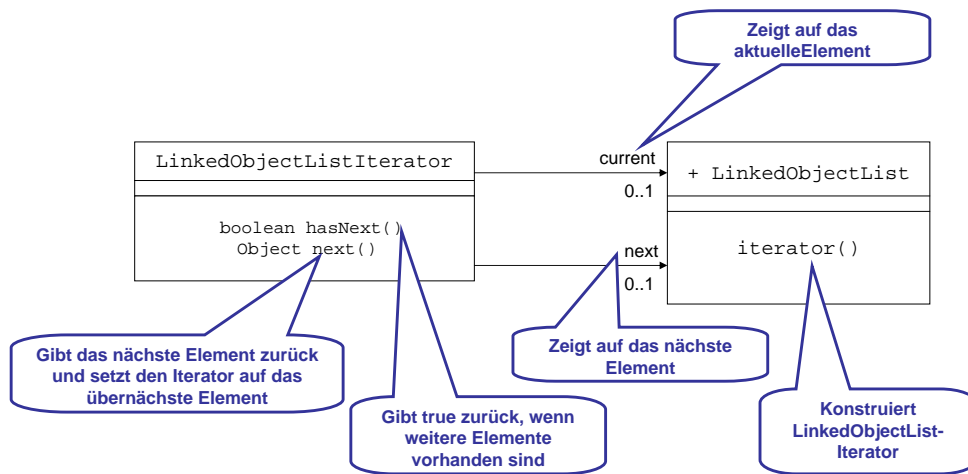
Gibt das „alte“ erste Elem zurück

Listendurchlauf mit Listeniterator



- Ein Listeniterator ermöglicht den Zugriff auf die Elemente einer verketteten Liste
- Ein Listeniterator schützt die Liste während des Zugriffs vor (unkontrollierten) Änderungen
- Ein Listeniterator kapselt eine Position in der Liste

Listendurchlauf: Listeniterator in UML



Listeniterator in Java

```
class LinkedObjectListIterator
{
    protected LinkedObjectListElem currentElem;
    protected LinkedObjectListElem nextElem;

    LinkedObjectListIterator(LinkedObjectListElem elem)
    {
        nextElem = elem;
    }

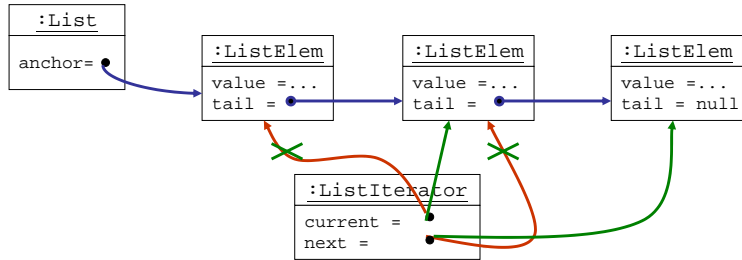
    public boolean hasNext()
    {
        return nextElem != null;
    }

    ...
}
```

Weiterschalten des Listeniterators in Java

```
public Object next() throws NoSuchElementException
{
    if (nextElem == null)
    {
        throw new NoSuchElementException();
    }
    currentElem = nextElem;
    nextElem = nextElem.getTail();
    return currentElem.getValue();
}
}
```

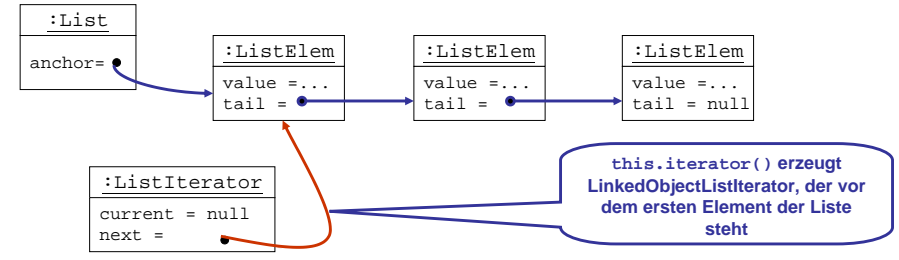
Schaltet den Iterator weiter und gibt den value eines LinkedListObjectElem zurück!



Konstruktion eines Listeniterators in Java

```
class LinkedListList
{
    private LinkedListListElem anchor;
    ...

    public LinkedListListIterator iterator() {
        return new LinkedListListIterator(this.anchor);
    }
}
```



Listendurchlauf mit Iteratoren

Schema:

```
Iterator iter = iterator();
while (iter.hasNext())
{
    << mache etwas >>
    iter.next();
}
```

erzeuge Iterator

Solange ein nächstes Element existiert

Schalte zum nächsten Element weiter

Beispiele für Listeniteration: Länge der Liste

```
public int size()
{
    int result = 0;
    LinkedListListIterator iter = iterator();
    while (iter.hasNext())
    {
        result++;
        iter.next();
    }
    return result;
}
```

erzeuge Iterator

Solange ein nächstes Element existiert

Schalte zum nächsten Element weiter

Beispiele für Listeniteration: Suche in der Liste

```

public boolean contains(Object o)
{
    LinkedObjectListIterator iter = iter;
    while (iter.hasNext())
    {
        if (iter.next().equals(o))
            return true;
    }
    return false;
}
    
```

Beispiele für Listeniteration: Revertieren der Liste

```

public void reverse()
{
    LinkedObjectListElem newAnchor = null;
    LinkedObjectListIterator iter = iterator();
    while(iter.hasNext())
    {
        newAnchor =
            new LinkedObjectListElem(iter.next(), newAnchor);
    }
    anchor = newAnchor;
}
    
```

Beispiele für Listeniteration: Listenvergleich

- Die Listenvergleichsoperation

```
public boolean equals(Object o)
```

prüft, ob zwei Listenobjekte die gleiche Länge haben und ihre Elemente jeweils den gleichen Wert (value) besitzen.

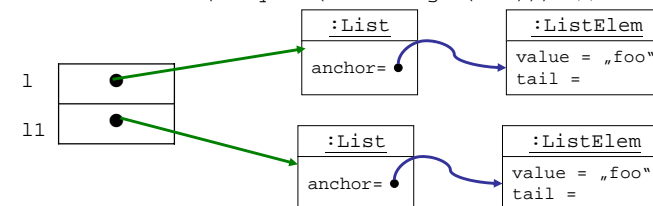
- Sind die Längen unterschiedlich oder sind die Listenelemente nicht alle „equals“ zueinander, so ist das Ergebnis false.
- Das Ergebnis ist auch false, wenn o nicht vom Typ LinkedObjectList ist.

Beispiele für Listeniteration: Listenvergleich

Beispiel: Folgendes sollte beim Testen für `l = new LinkedObjectList("foo")` gelten:

```

assertTrue(l.equals(l)); //l ist mit sich selbst gleich
assertFalse(l.equals("foo")); //falscher Typ
LinkedObjectList l1 = new LinkedObjectList("foo");
assertTrue(l.equals(l1)); //l, l1 haben das gleiche Element
l1.addFirst("baz");
assertFalse(l.equals(l1)); //falsche Laenge
assertFalse(l.equals(new Integer(123))); //verschiedenes Element
    
```



Beispiele für Listeniteration: Listenvergleich

```
public boolean equals(Object o)
{
    try
    {
        LinkedObjectListIterator iter1 = iterator();
        LinkedObjectListIterator iter2 =
            ((LinkedObjectList)o).iterator();
        while (iter1.hasNext() && iter2.hasNext())
        {
            if (!iter1.next().equals(iter2.next()))
                return false;
        }
        if (iter1.hasNext() != iter2.hasNext())
            return false;
        else
            return true;
    }
    catch (Exception e)
    {
        return false;
    }
}
```

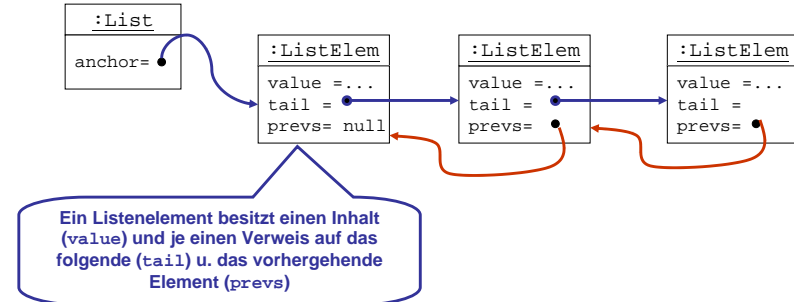
Erzeuge 2 Iteratoren

Vergleiche u. schalte weiter, solange beide noch ein nächstes Element haben

false, falls o kein Listenobjekt

Verfeinerung: Doppelt verkettete Listen

- Doppelt verkettete Listen können auch von rechts nach links durchlaufen werden.

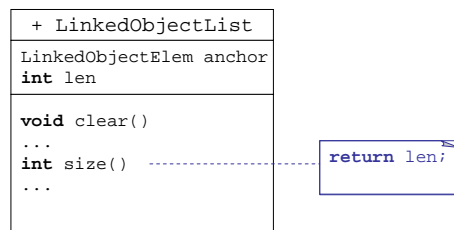


Ein Listenelement besitzt einen Inhalt (value) und je einen Verweis auf das folgende (tail) u. das vorhergehende Element (prevs)

- Die Standardlistenklasse von Java ist doppelt verkettet implementiert.

Verfeinerung: Zeiteffiziente einfach verkettete Listen

- Durch Hinzufügen eines Attributs für die Länge der Liste erhält die Abfrage nach der Größe der Liste konstante Zeitkomplexität:



Zusammenfassung

- Listen werden in Java als einfach oder doppelt verkettete oder auch als zirkuläre und Ringlisten realisiert. Zur Implementierung definiert man eine Klasse `LinkedList`, mittels eines Ankers (`anchor`) auf Objekte der Klasse `ListElem` zeigt. Diese sind über die `tail`- und `prevs`-Zeiger miteinander verknüpft.
- Der Listendurchlauf wird mit Hilfe der Klasse `ListIterator` realisiert. Iteratorobjekte wandern sequenziell durch die Liste.