Listen

Martin Wirsing

in Zusammenarbeit mit
Michael Barth, Fabian Birzele und Gefei Zhang

http://www.pst.informatik.uni-muenchen.de/lehre/WS0506/infoeinf/

WS 05/06

Ziele

- Standardimplementierungen für Listen kennenlernen
- Listeniteratoren verstehen

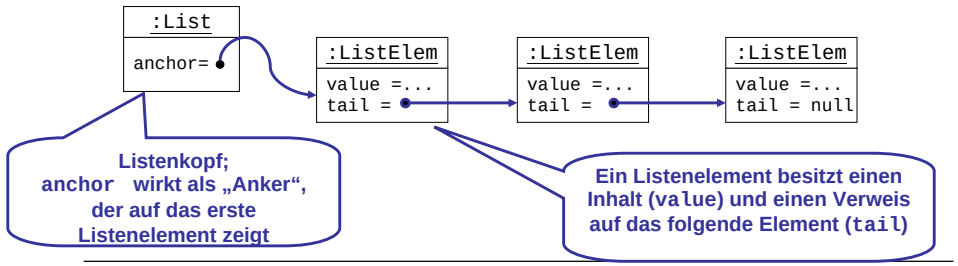
Die Rechenstruktur der Listen

- 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

Listenimplementierung: Einfach verkettete Listen

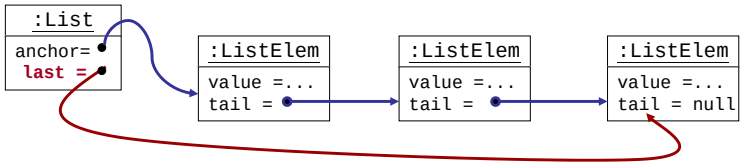
- Eine einfach verkettete Liste ist eine Sequenz von Objekten, wobei jedes Element auf seinen Nachfolger in der Liste zeigt.
- Unterschiedliche Implementierungen:

  1. Realisierung des Anfügens vorne in konstanter Zeit:

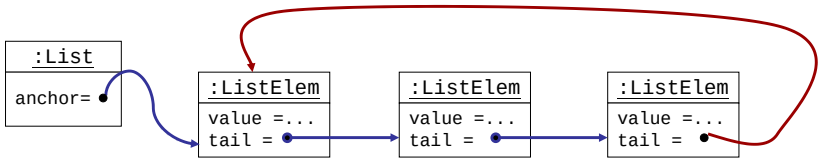


Einfach verkettete Listen

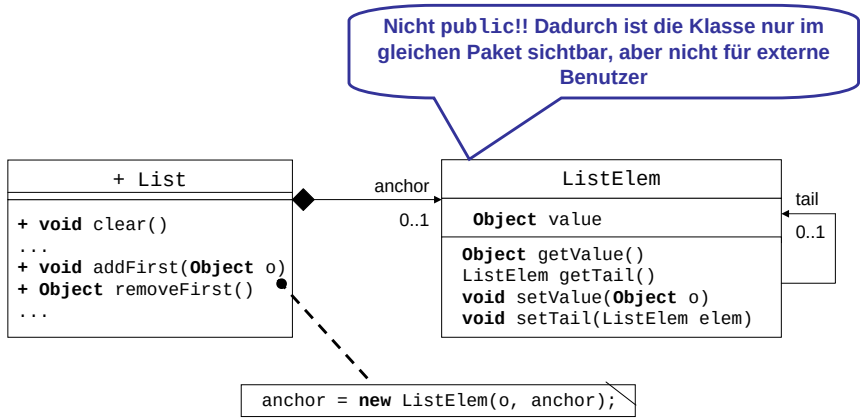
2. Realisierung des Anfügens vorne und hinten in konstanter Zeit:



3. Zirkuläre Liste:



Einfach verkettete Listen: UML-Entwurf



Einfach verkettete Listen in Java

```
public class List
{ private ListElem anchor;

//Konstruktoren
  public List()
  {     anchor = null;
  }

  public List(Object o)
  {     anchor = new ListElem(o);
  }

  ...
}
```

Erzeugung einer leeren Liste

Erzeugung einer einelementigen Liste

Starke Aggregation: Die „Teile“ werden im Konstruktor des „Ganzen“ erzeugt.

Einfach verkettete Listen in Java

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

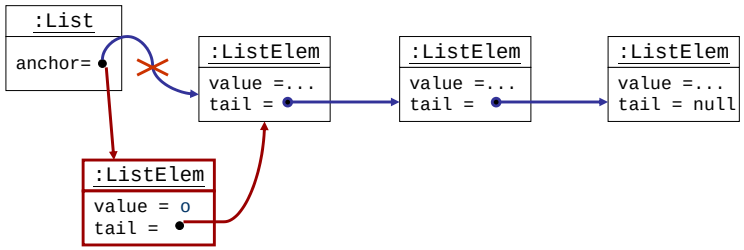
Nicht public!!
Dadurch ist die Klasse nur im gleichen Paket sichtbar, aber NICHT für externe Benutzer!

Konstruktor

Zugriff auf das Attribut value

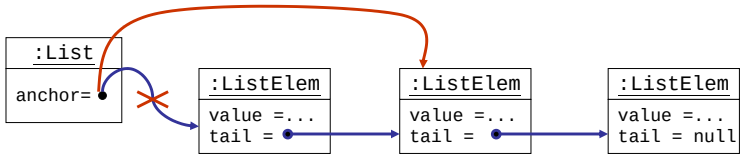
Ersetzt das nächste Listenelement durch elem und ändert dadurch den Rest der Liste (nicht für externe Benutzer!)

Einfügen eines Objekts o am Anfang der Liste




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


Entfernen (und Zurückgeben) des ersten Elements



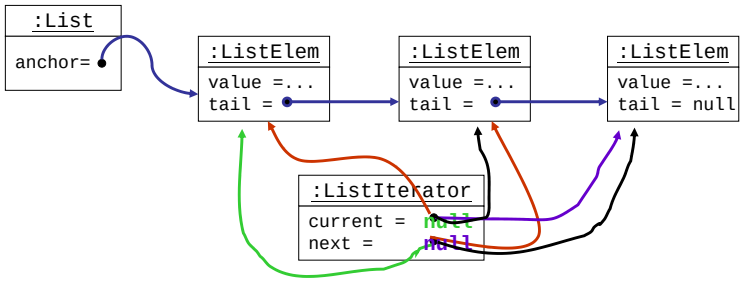

```
public Object removeFirst() throws NoSuchElementException {
        if (anchor == null)
        {       throw new NoSuchElementException();
        }
        else
        {       Object result = anchor.getValue();
                anchor = anchor.getTail();
                return result;
        }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

Ausnahme, wenn Liste leer

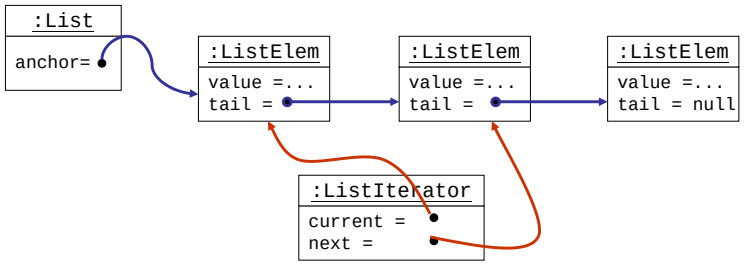
rettet den Wert des ersten Elements

Gibt den Wert des „alten“ ersten Elements zurück

Listendurchlauf mit Listeniterator

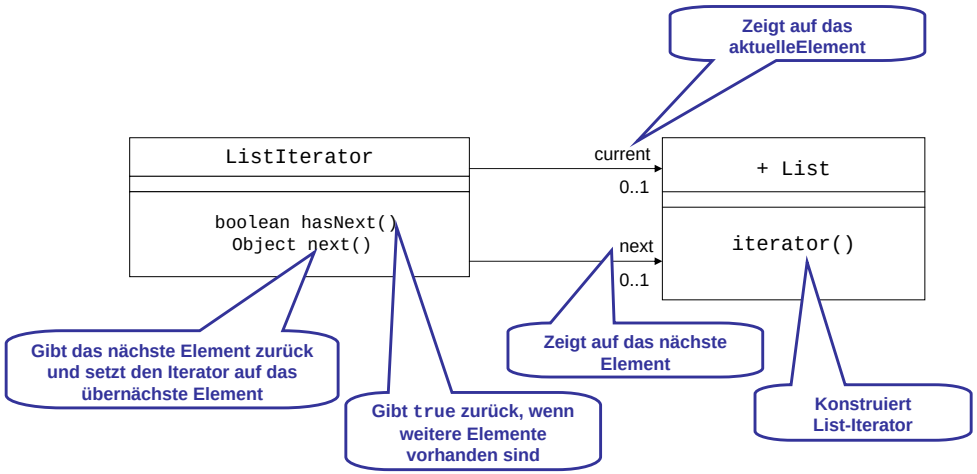


Listendurchlauf mit Listeniterator



- 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 ListIterator
{ protected ListElem currentElem;
  protected ListElem nextElem;

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

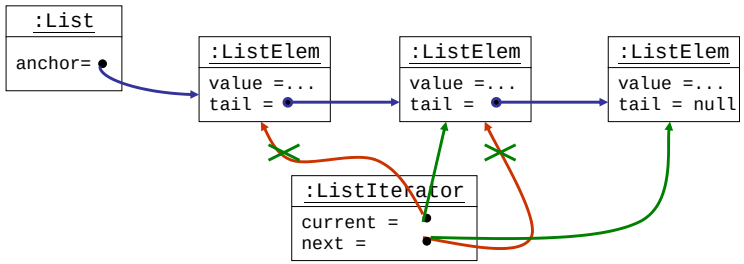
hasNext() ergibt true, wenn es ein nächstes Element gibt

Weiterschalten des Listeniterators in Java

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

Ausnahme, wenn kein nächstes Element exisitiert

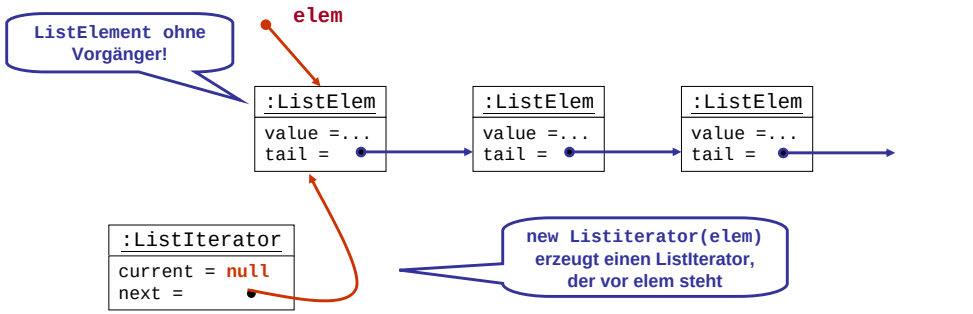
Schaltet den Iterator weiter und gibt den **value** eines ListElem zurück!



Konstruktor für `ListIterator`

```
ListIterator(ListElem elem) {
      nextElem = elem;
}
```

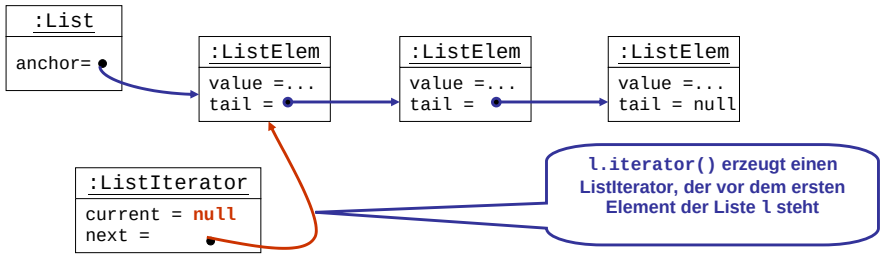
Das nächste Element ist elem (und das „current“ Element ist null)



Erzeugung eines Listeniterators in der Klasse `List`

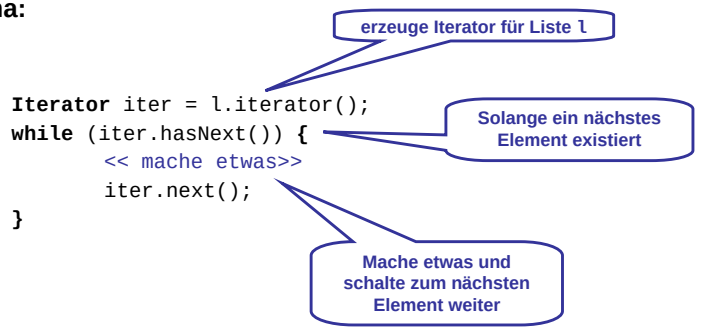
```
class List
{ private ListElem anchor;
  ...

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

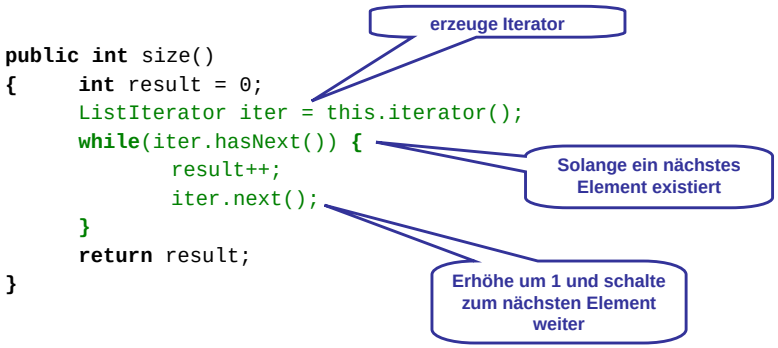


Listendurchlauf mit Iteratoren

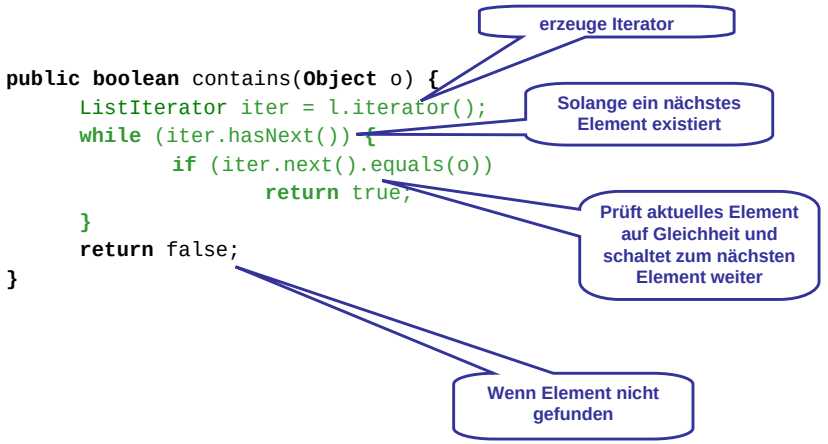
Schema:



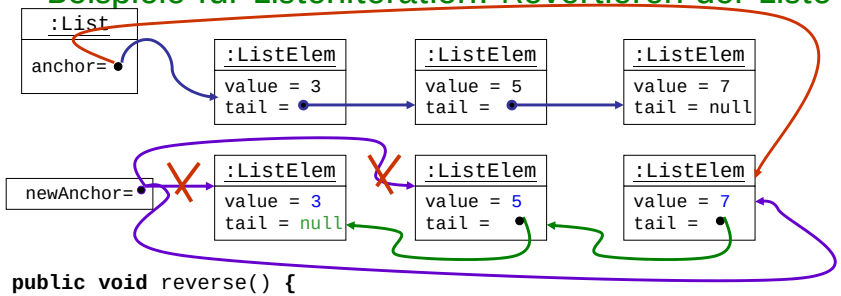
Beispiele für Listeniteration: Länge der Liste



Beispiele für Listeniteration: Suche in der Liste



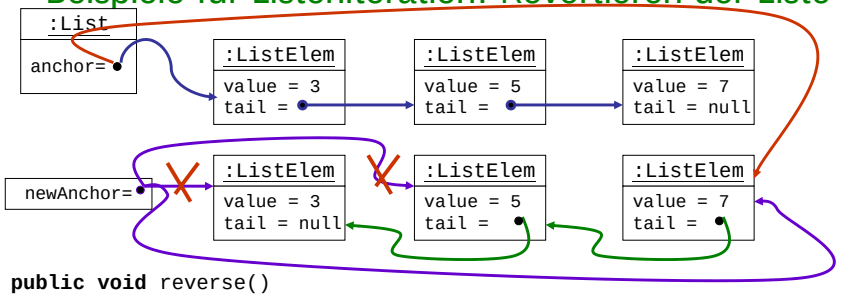
Beispiele für Listeniteration: Revertieren der Liste




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

erzeugt neues Listenelement mit umgedrehtem Verweis

Beispiele für Listeniteration: Revertieren der Liste




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

erzeugt neues Listenelement mit umgedrehtem Verweis

Beispiele für Listeniteration: Listenvergleich

- Die Listenvergleichsoperation

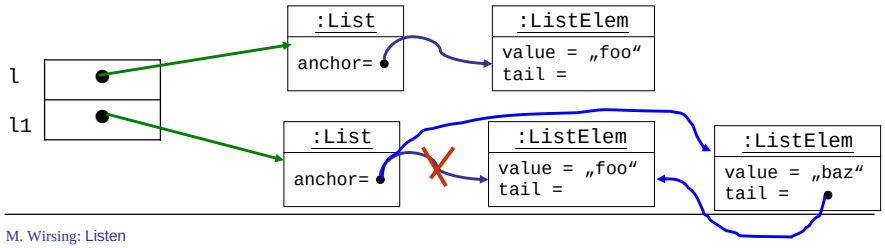
  `public boolean equals(Object o)`

  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 `List` ist.

Beispiele für Listeniteration: Listenvergleich

Beispiel: Folgendes sollte beim Testen gelten:

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



Beispiele für Listeniteration: Listenvergleich

```
public boolean equals(Object l) {
      try {
            ListIterator iter1 = this.iterator();
            ListIterator iter2 = ((List)l).iterator();

            while (iter1.hasNext() && iter2.hasNext()){
                  if (!iter1.next().equals(iter2.next()))
                        return false;
            }
            return iter1.hasNext() = iter2.hasNext();
      }
      catch(Exception e)
      {     return false;
      }
}
```

Erzeuge 2 Iteratoren

return false, falls this oder l kein Listenobjekt

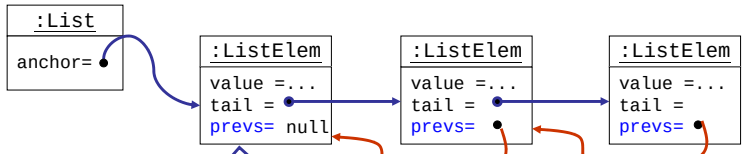
solange beide noch ein nächstes Element haben

Vergleiche u. schalte beide Iteratoren weiter

return true, falls nach dem Ende von while beide Listen keine weiteren Elemente (d.h. die gleiche Länge) haben

Verfeinerung: Doppelt verkettete Listen

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

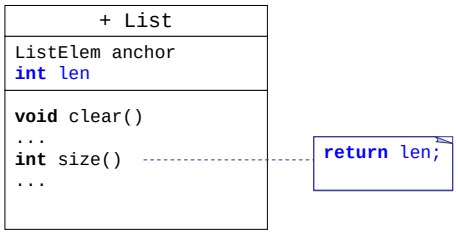


Ein Listenelement besitzt einen Inhalt (value) und je einen Verweis auf das folgende (tail) u. das vorhergehende Element (prevs)

- 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 `List`, 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 sequentiell durch die Liste.