Listen

Prof. Dr. Christian Böhm

in Zusammenarbeit mit
Gefei Zhang

http://www.dbs.ini.lmu.de/Lehre/NFInfoSW

Ziele

- Standardimplementierungen für Listen kennenlernen
- Listeniteratoren verstehen
- Unterschiede zwischen Listen und Arrays kennenlernen

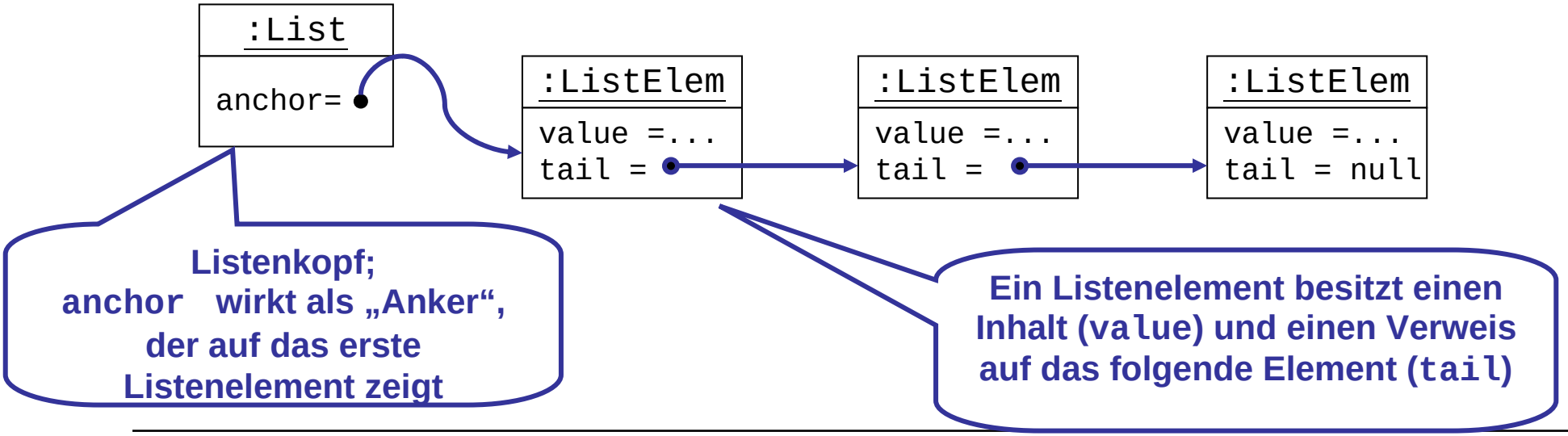
Die Rechenstruktur der Listen

- 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

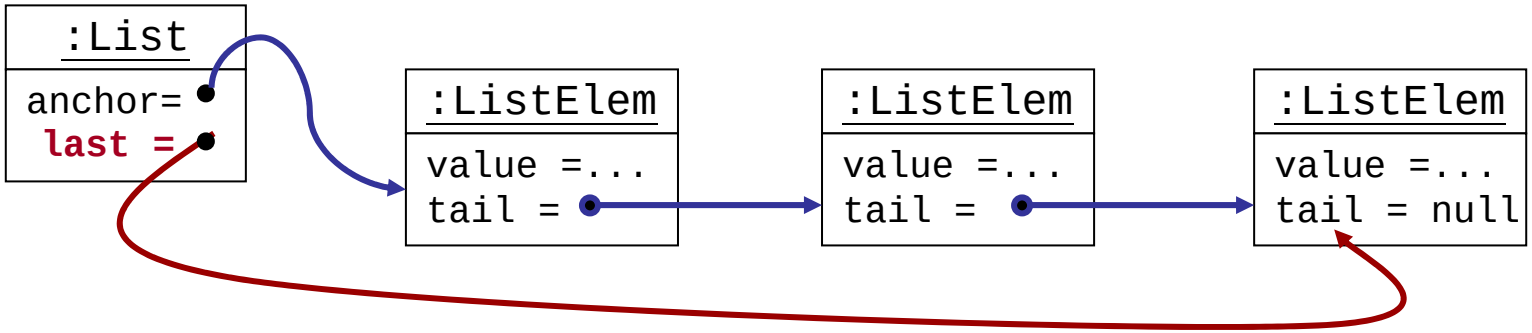
- Eine einfach verkettete Liste ist eine Sequenz von Objekten, wobei jedes Element auf seinen Nachfolger in der Liste zeigt.
- Unterschiedliche Implementierungen:

1. Realisierung des Anfügens vorne in konstanter Zeit:

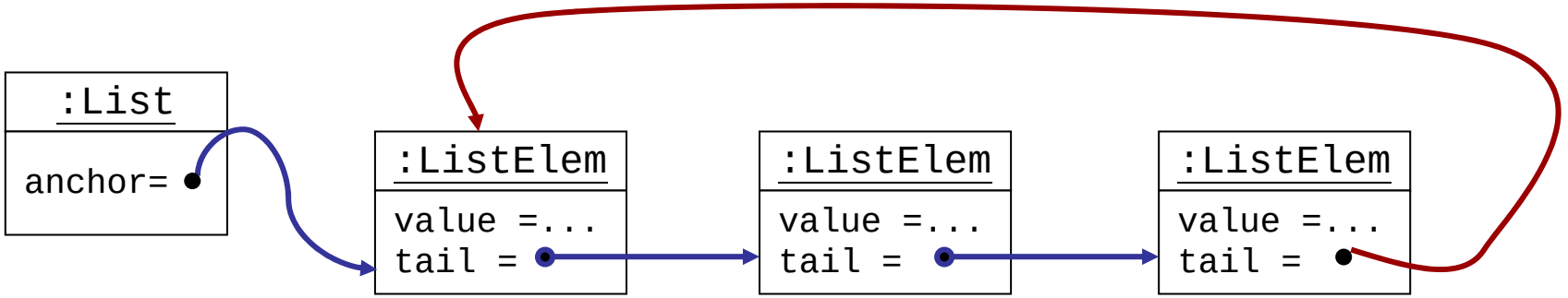


Einfach verkettete Listen

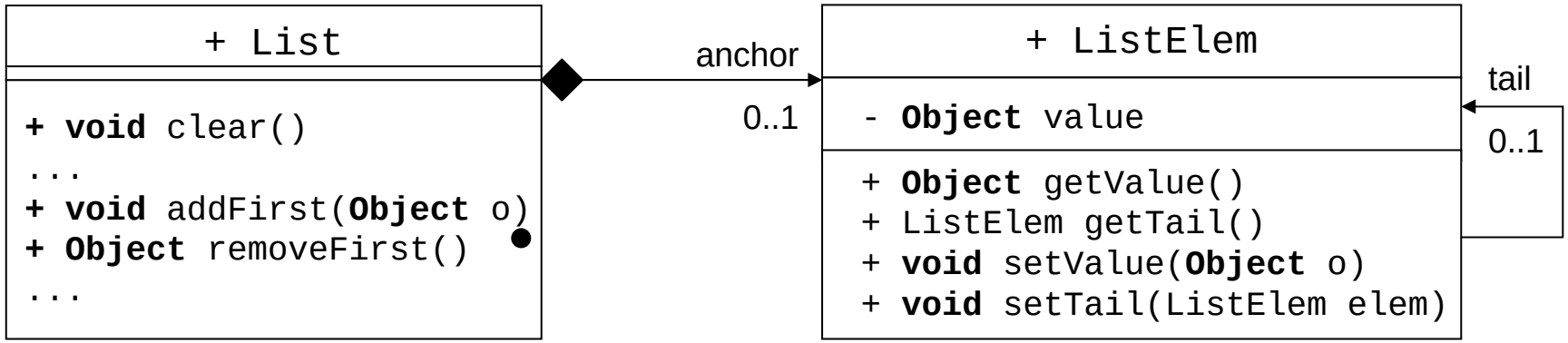
2. Realisierung des Anfügens vorne und hinten in konstanter Zeit:



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

Das Objekt o muss in einem ListElem "verpakt" werden

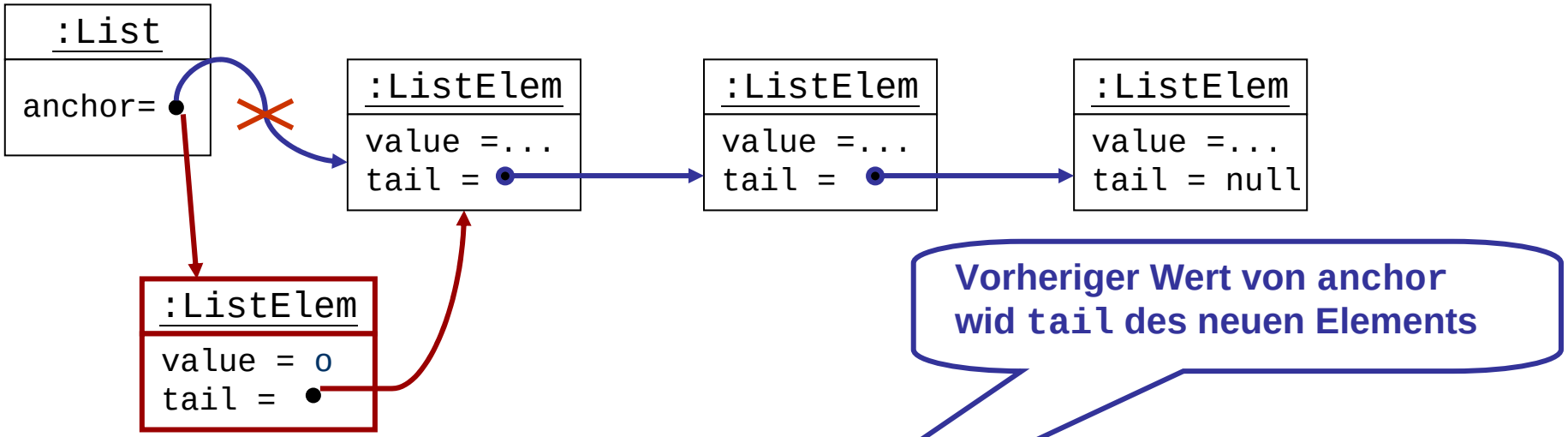
Einfach verkettete Listen in Java

```
public class ListElem
{ private Object value;
  private ListElem tail;
  public ListElem(Object o) {value = o; tail = null;}
  public ListElem(Object o, ListElem t) {value = o; tail = t;}
  public ListElem getTail() { return tail; }
  public void setTail(ListElem elem) {  ail = elem;}
   public Object getValue() {return value;}
   public void setValue(Object v) {value = v;}
   . . .
}
```

Konstruktoren

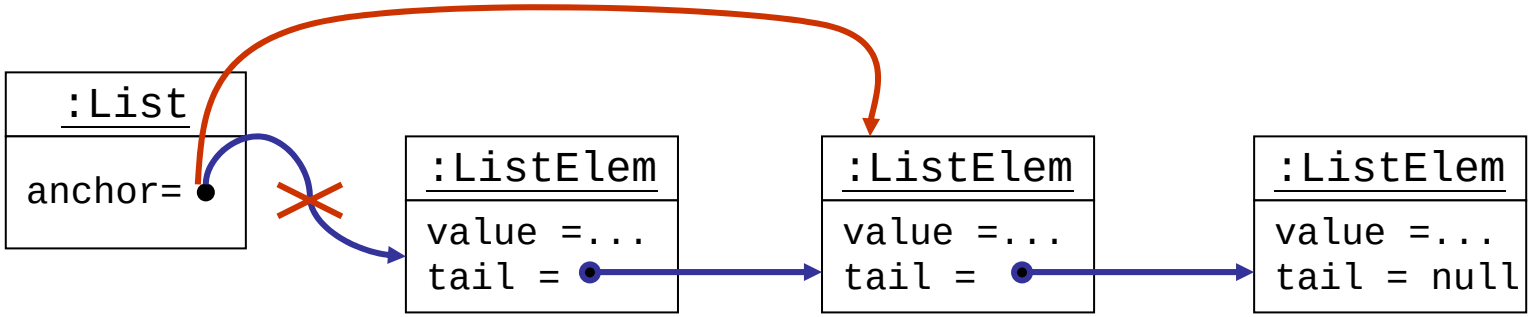
Getter- und Setter-Methoden

Einfügen eines Objekts o am Anfang der Liste




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

Entfernen (und Zurückgeben) des ersten Elements



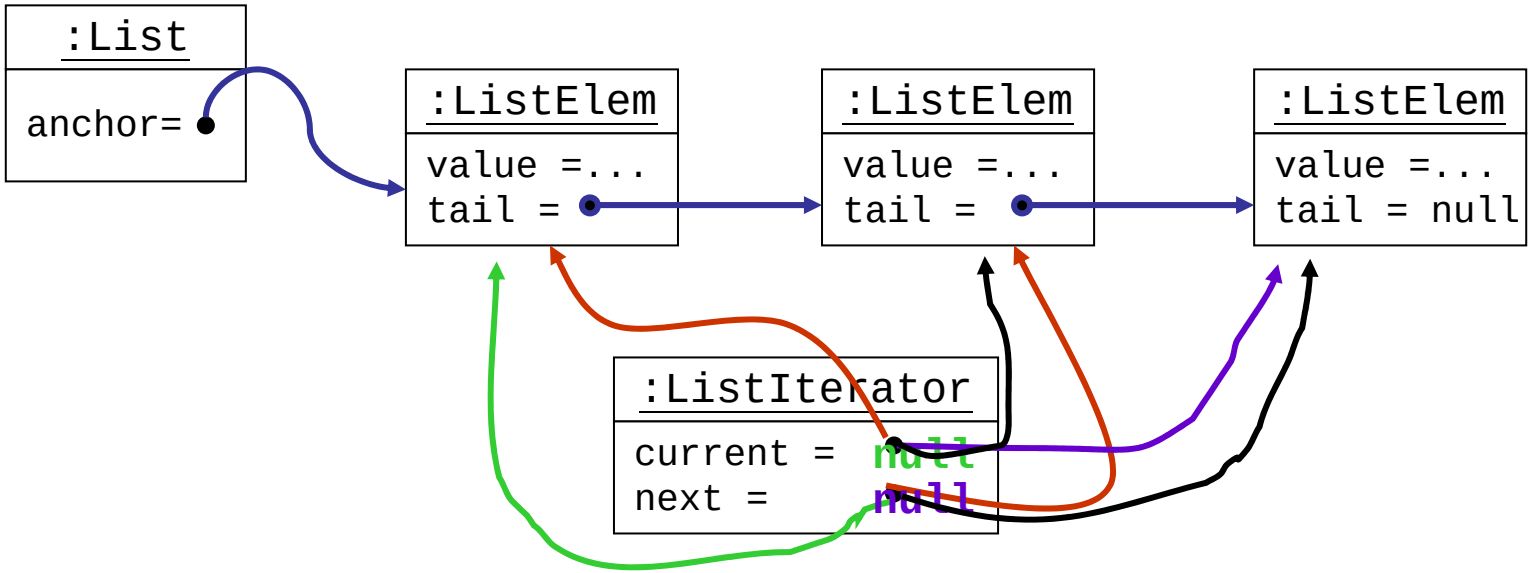

```
public Object removeFirst() throws NoSuchElementException {
        if (anchor == null)
        {       throw new NoSuchElementException();
        }
        else
        {       Object result = anchor.getValue();
                anchor = anchor.getTail();
                return result;
        }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

Ausnahme, wenn Liste leer

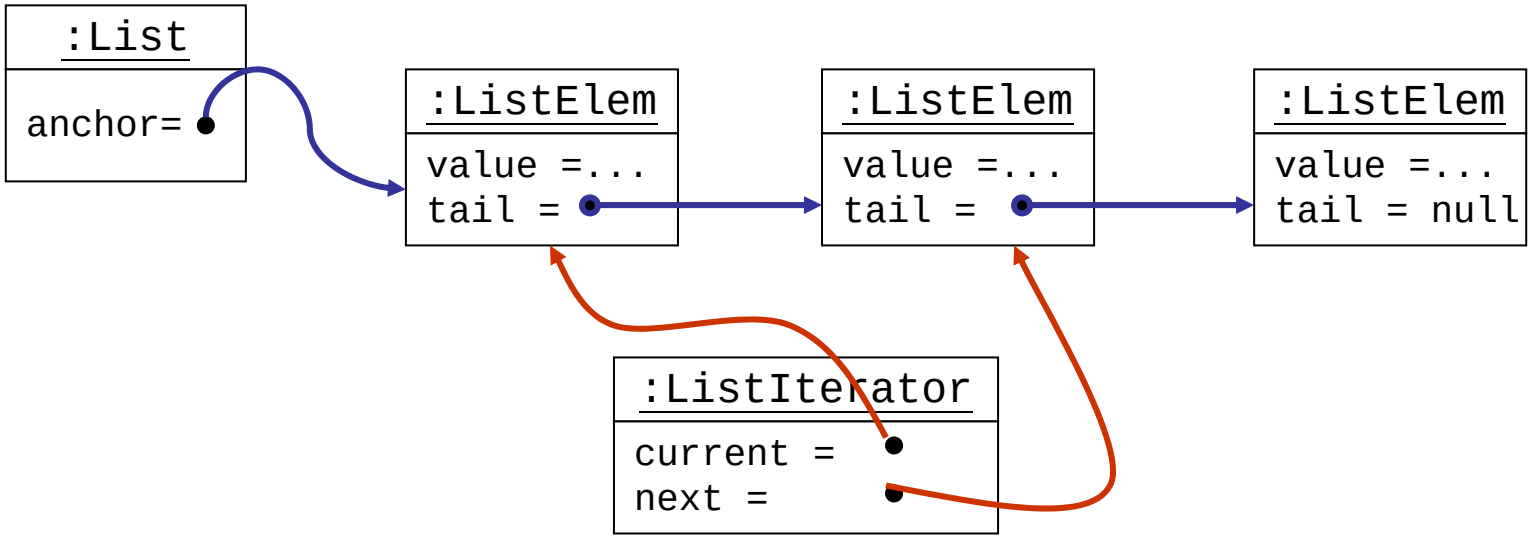
rettet den Wert des ersten Elements

Gibt den Wert des „alten“ ersten Elements zurück

Listendurchlauf mit Listeniterator

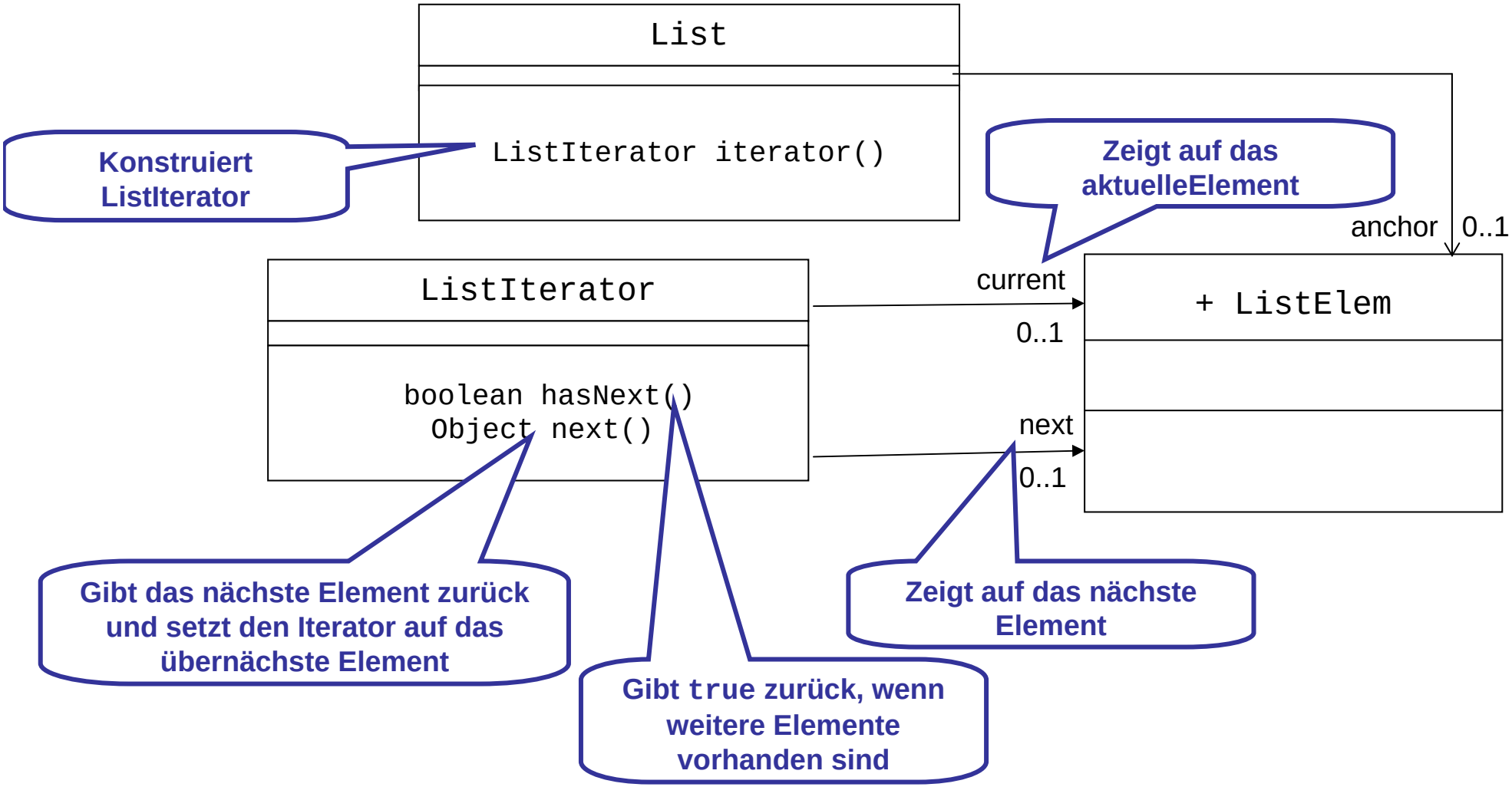


Listendurchlauf mit Listeniterator



- Ein Listeniterator ermöglicht den Zugriff auf die Elemente einer verketteten Liste
- Ein Listeniterator sollte die Liste während des Zugriffs vor (unkontrollierten) Änderungen schützen (hier nicht realisiert)

Listendurchlauf: Listeniterator in UML



Listeniterator in Java

```
class ListIterator
{ protected ListElem current;
  protected ListElem next;


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

  ...
}
```

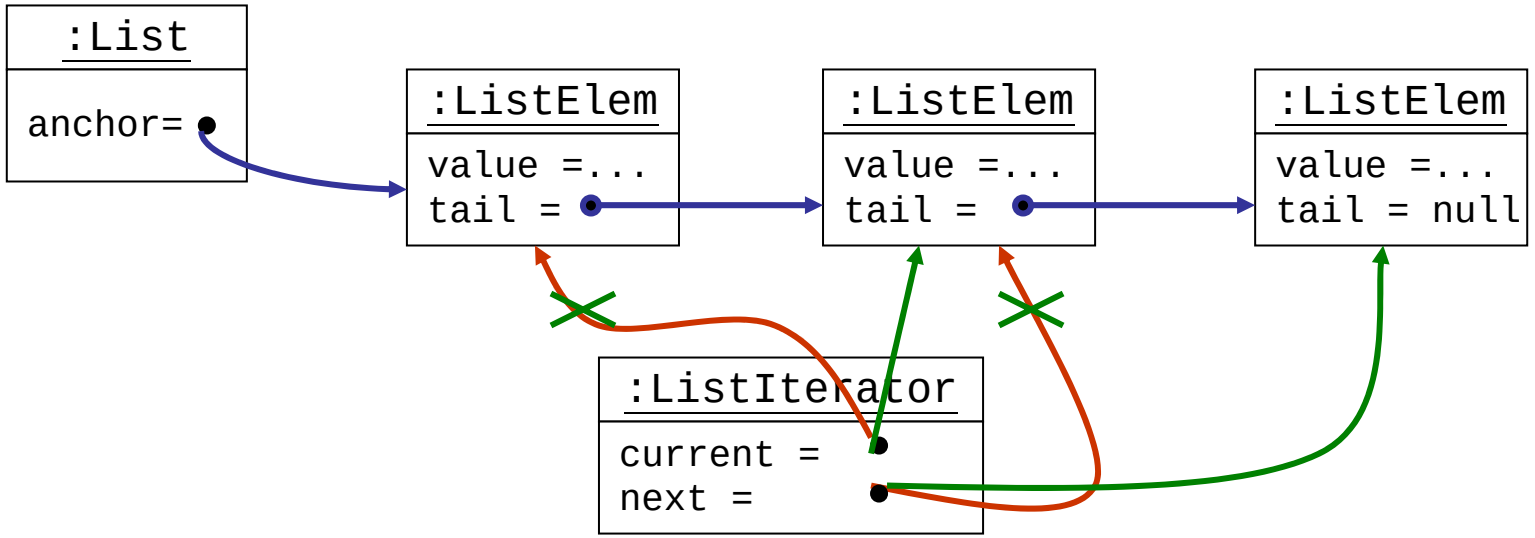
hasNext() ergibt true, wenn es ein nächstes Element gibt

Weiterschalten des Listeniterators in Java

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

Ausnahme, wenn kein nächstes Element exisitiert

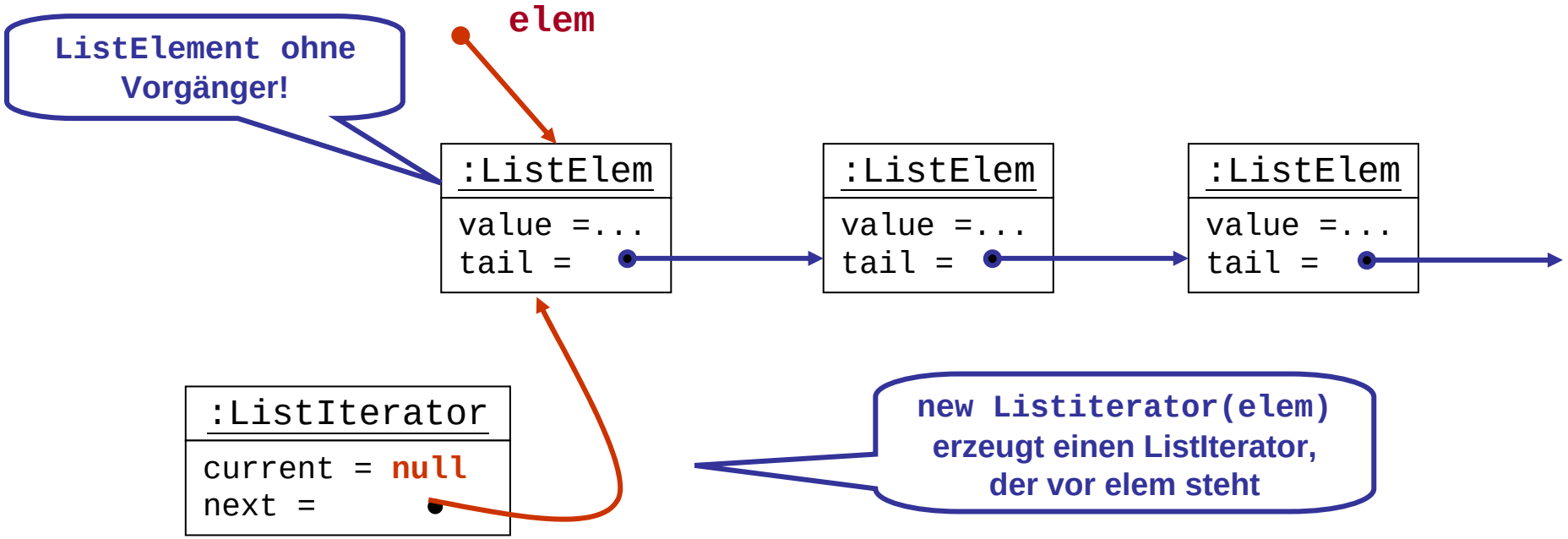
Schaltet den Iterator weiter und gibt den **`value`** eines ListElem zurück!



Konstruktor für `ListIterator`

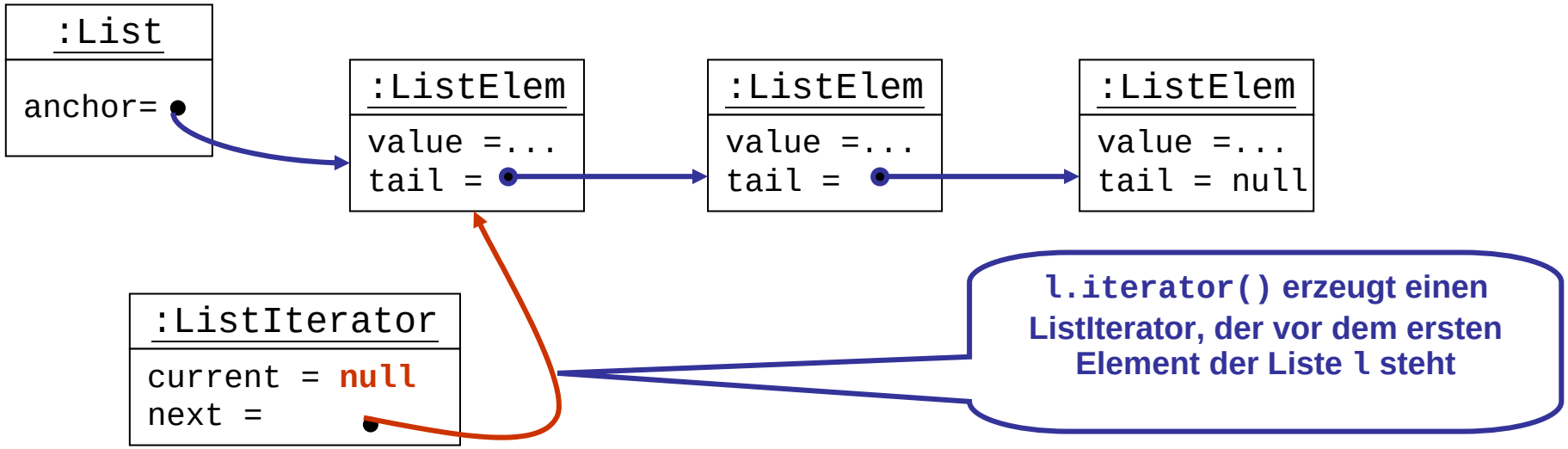
```
ListIterator(ListElem elem) {
      next = elem;
}
```

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



Erzeugung eines Listeniterators in der Klasse `List`

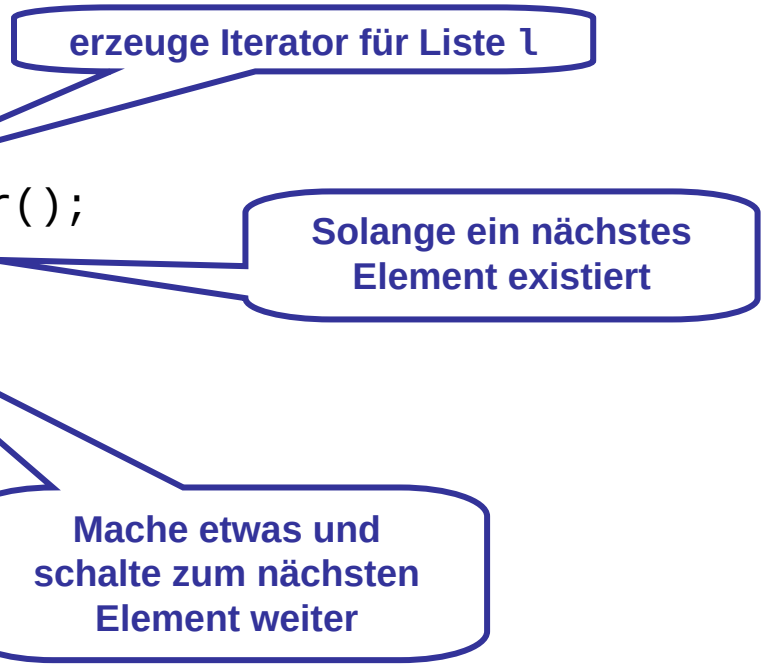
```
public class List
{ private ListElem anchor;
  ...

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



Listendurchlauf mit Iteratoren

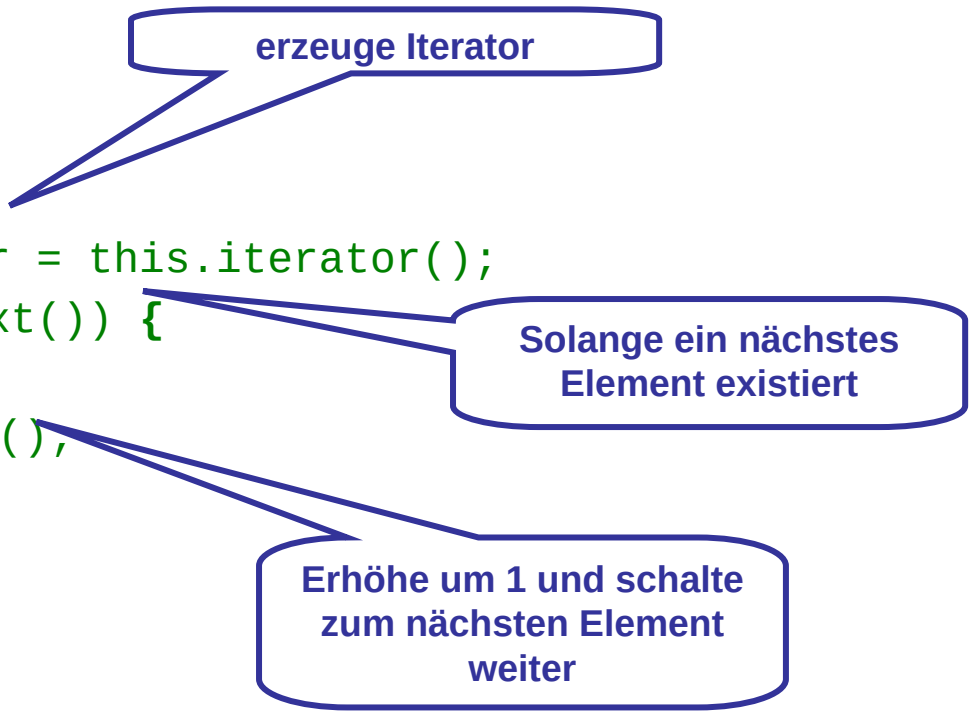
Schema:

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



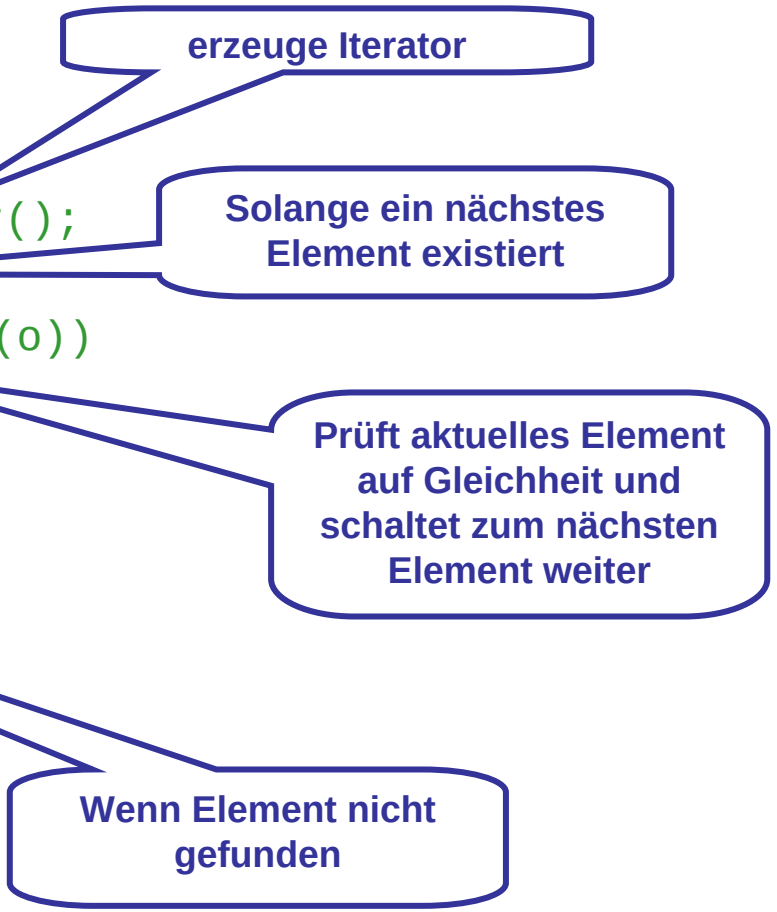
Beispiele für Listeniteration: Länge der Liste

```
public class List
{

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

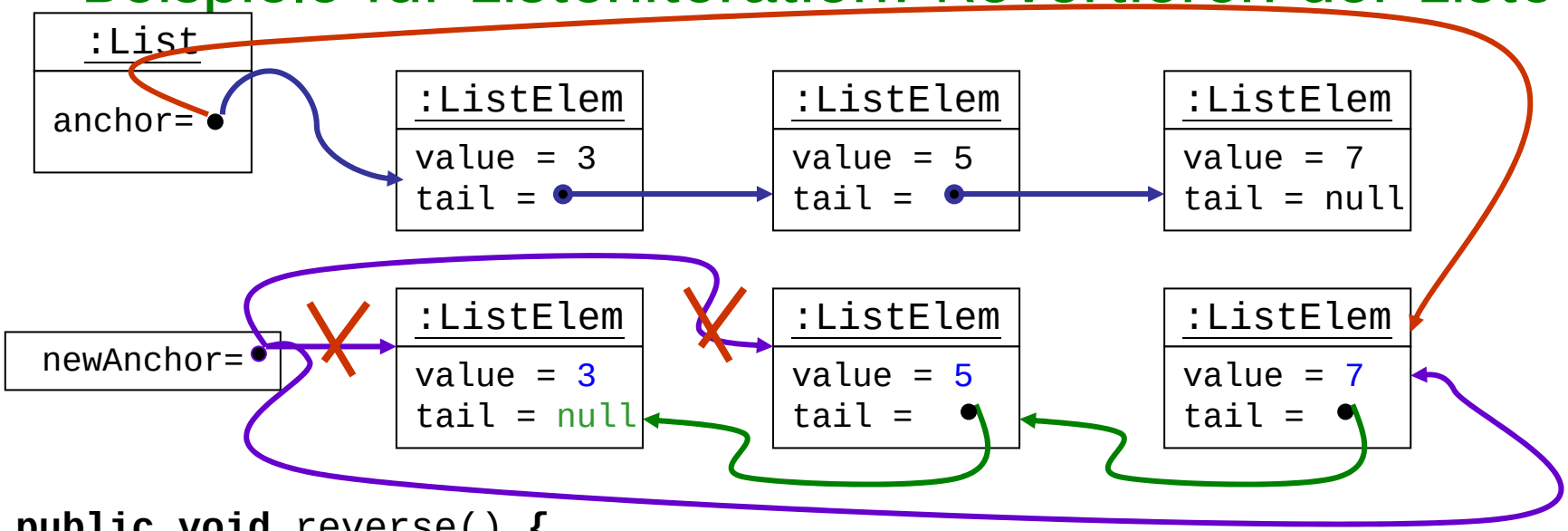


Beispiele für Listeniteration: Suche in der Liste

```
class List {
   . . .
   public boolean contains(Object o) {
         ListIterator iter = l.iterator();
         while (iter.hasNext()) {
               if (iter.next().equals(o))
                     return true;
         }
         return false;
   }
    . . .
}
```



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

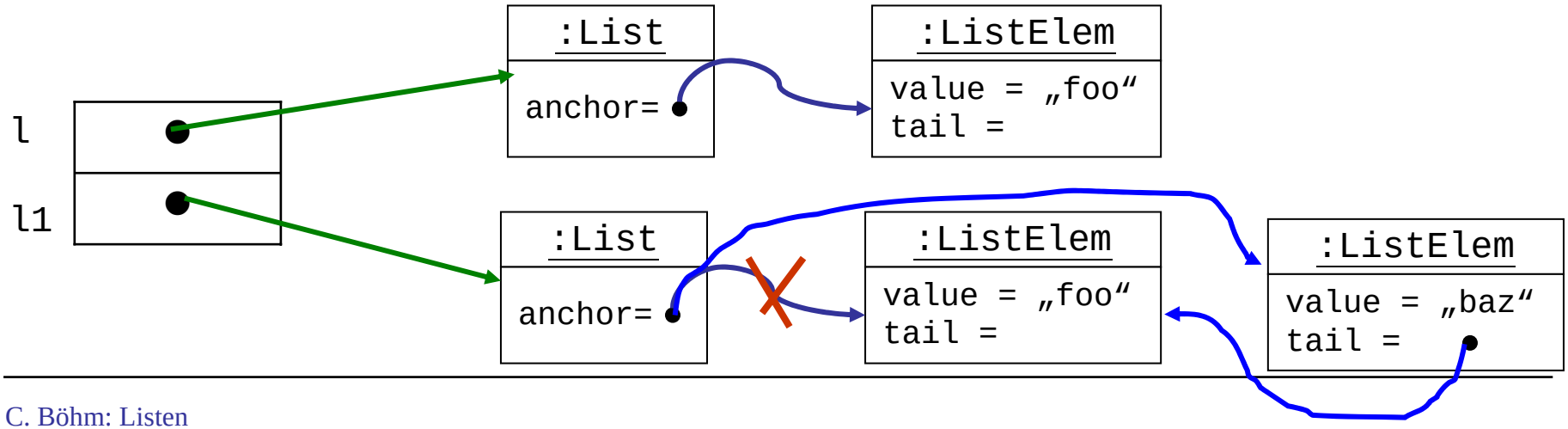
  ```
  public boolean equals(Object o)
  ```

  prüft, ob zwei Listenobjekte die gleiche Länge haben und ihre Elemente jeweils den gleichen Wert (`value`) besitzen.
- Die von der Klasse `Object` geerbte Methode `equals` wird überschrieben.
- 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");
l1.addFirst("baz");
```



Beispiele für Listeniteration: Listenvergleich

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

            while (iter1.hasNext() && iter2.hasNext()){
                  Object o1 = iter1.next();
                  Object o2 = iter2.next();
                  if (!o1.equals(o2)) return false;
            }
            return iter1.hasNext() == iter2.hasNext();
      }
      catch(Exception e)
      {     return false;
      }
}
```

Erzeuge 2 Iteratoren

return false, falls this oder l kein Listenobjekt

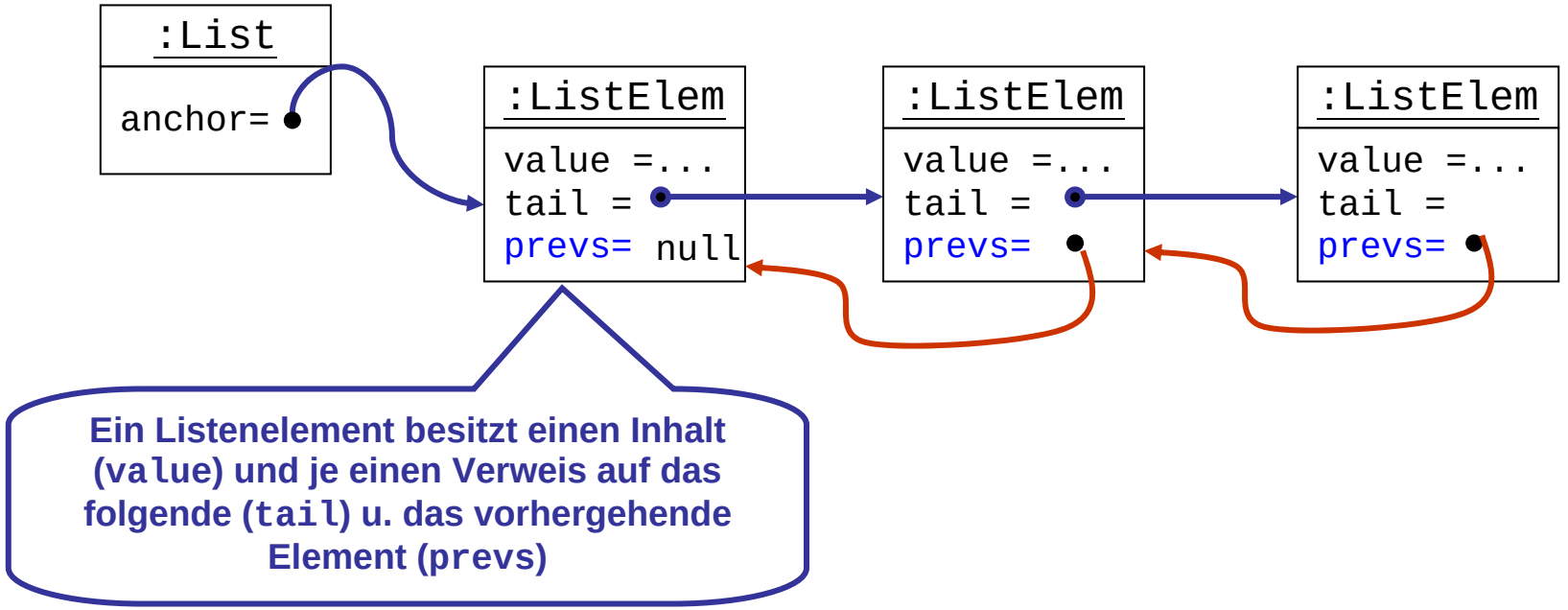
solange beide noch ein nächstes Element haben

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

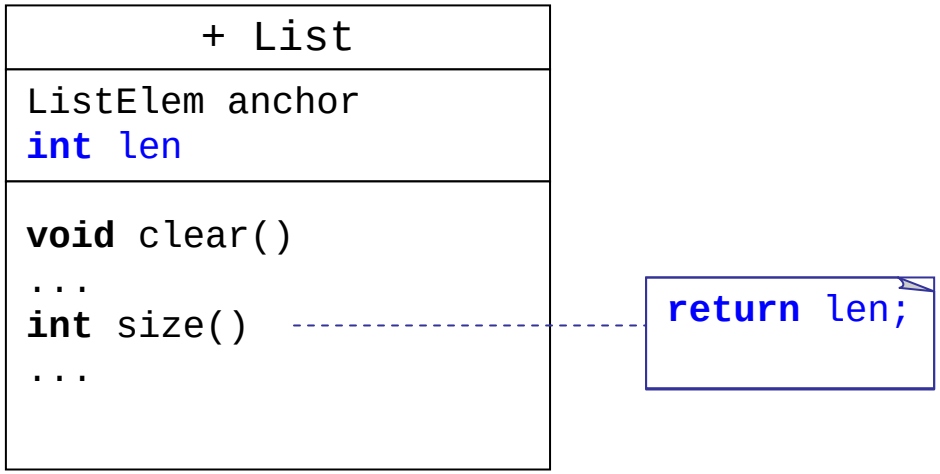
- Doppelt verkettete Listen können auch von rechts nach links durchlaufen werden.



- 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.