Kapitel 13

Listen

Ziele

- Implementierungen für Listen kennenlernen
- Einfach verkettete und doppelt verkettete Listen verstehen
- Listen-Implementierungen in der Java-Bibliothek kennenlernen
- Durch Listen von Objekten iterieren können

Listen und Operationen auf Listen

Eine **Liste** ist eine **endliche Folge von Elementen**, deren Länge (im Gegensatz zu Arrays) durch Hinzufügen und Wegnehmen von Elementen geändert werden kann. Es handelt sich um eine **dynamische** Datenstruktur.

Übliche Operationen auf Listen:

- Hinzufügen eines Elements am Anfang (ggf. auch am Ende) einer Liste
- Entfernen eines Elements am Anfang (ggf. auch am Ende) einer Liste
- Zugriff auf Elemente der Liste (z.B. erstes, letztes Element oder an einer bestimmten Position)
- Berechnen der Länge der Liste
- Prüfen auf leere Liste
- Listendurchlauf

Implementierungsarten für Listen

Es gibt mehrere Möglichkeiten Listen zu implementieren, z.B.

Array-Listen

- Die Listenelemente werden in einem Array gespeichert.
- Einfacher Zugriff auf Listenelemente.
- Bei Einfügeoperationen wird die Größe des Arrays bei Bedarf angepasst durch Anlegen eines neuen Arrays und Umkopieren.
- Das Array ist meist etwas größer als die repräsentierte Liste (partielle Arrays!), so dass nicht bei jeder Einfügeoperation (am Ende eines Arrays) umkopiert werden muss.
- In der Java-Bibliothek: Klasse `ArrayList`

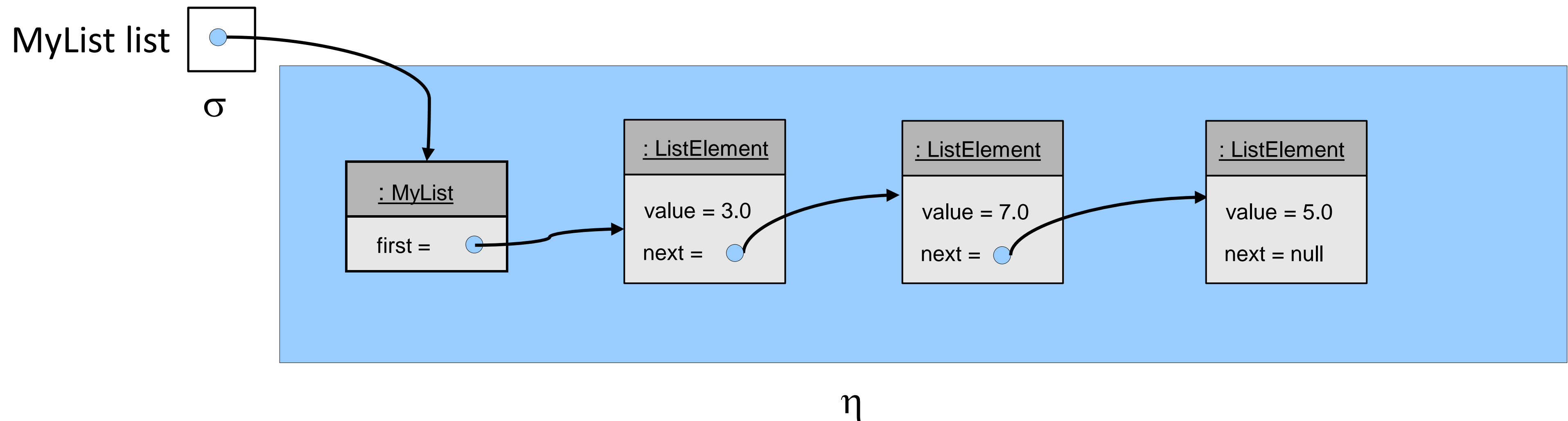
Verkettete Listen

- Die Listenelemente werden als Kette von Objekten gespeichert.
- Die Größe der Liste ist dynamisch änderbar.
- Einfügen und Löschen wird einfacher, Elementzugriff schwieriger.
- In der Java-Bibliothek: Klasse `LinkedList`

Einfach verkettete Listen

Repräsentiere Listen als Ketten von Objekten (im Heap).

Beispiel: Repräsentation der Liste `<3.0,7.0,5.0>` von `double`-Werten:



Klassen für verkettete Listen

- `MyList`: Eine Klasse für Listen, die Methoden für die verschiedenen Operationen auf Listen bereitstellt. Objekte der Klasse `MyList` haben eine Referenz auf das erste Listenelement. Die Referenz hat den Wert `null`, wenn die Liste leer ist.
- `ListElement`: Eine Klasse, deren Objekte Listenelemente darstellen. Jedes Objekt hat einen double-Wert und eine Referenz auf das nächste Listenelement. Die Referenz hat den Wert `null`, wenn es kein nächstes Element gibt.

MyList
-ListElement first
+MyList() +void addFirst(double d) +double removeFirst() +double get(int i) +int size()

ListElement
-double value -ListElement next
+ListElement(double d) +ListElement(double d, ListElement next) +double getValue() +void setValue(double d) +ListElement getNext() +void setNext(ListElement next)

Beispiel: Benutzung solcher Listen

```
public static void main(String[] args) {

   MyList list = new MyList(); // erzeuge leere Liste

   list.addFirst(5.0);  // list ist <5.0>
   list.addFirst(7.0); // list ist <7.0, 5.0>
   list.addFirst(3.0);  // list ist <3.0, 7.0, 5.0>

   // Entferne die Elemente eines nach dem anderen
   // vom Anfang der Liste aus und gib ihre Werte aus.
   while (list.size() > 0) {
      System.out.println(list.removeFirst());
   }
}
```

Implementierung der Klasse ListElement

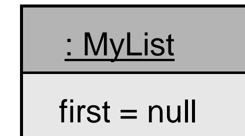
```
public class ListElement {
   // Instanzvariablen
   private double value;
   private ListElement next;

   // Konstruktoren
   public ListElement(double d) {
      this.value = d;
      this.next = null;
   }
   public ListElement(double d, ListElement next) {
      this.value = d;
      this.next = next;
   }
   public double getValue() { return this.value; }
   public void setValue(double d) { this.value = d; }
   public ListElement getNext() { return this.next; }
   public void setNext(ListElement next) { this.next = next; }
}
```

Getter- und Setter-Methoden

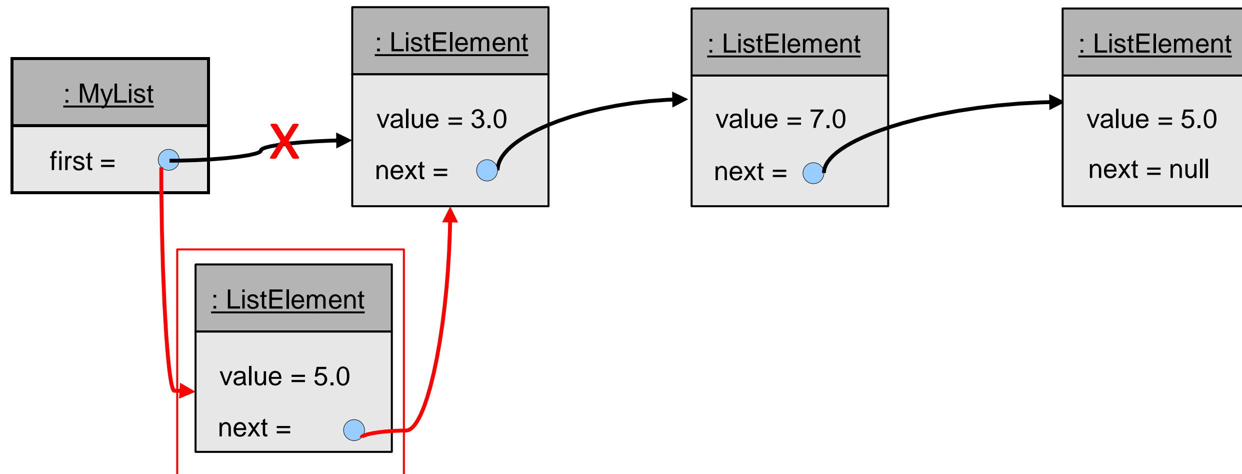
Implementierung der Klasse MyList

```
public class MyList {
   // Instanzvariable/Attribut
   private ListElement first;

   // Konstruktor
   public MyList() {
      first = null;
   }

   ...  //siehe unten

}
```

erzeugt leere Liste:



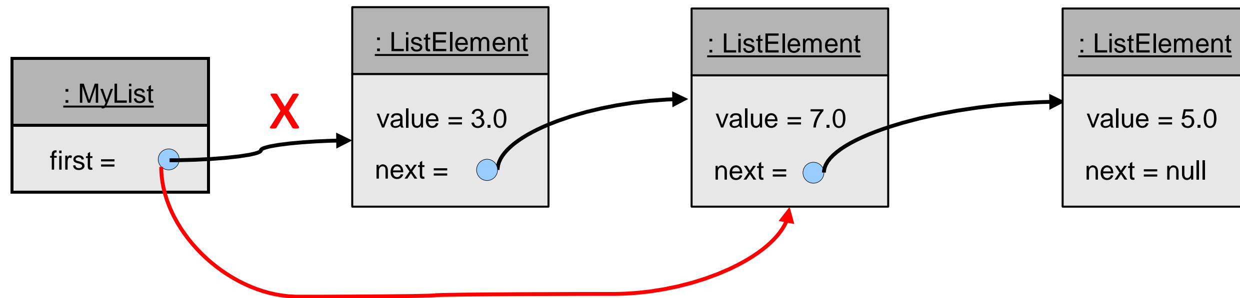
Einfügen eines Werts am Anfang der Liste




```
public class MyList {
    …

    public void addFirst(double d) {
        first = new ListElement(d, first);
    }

    …
```

Konstante Zeitkomplexität!

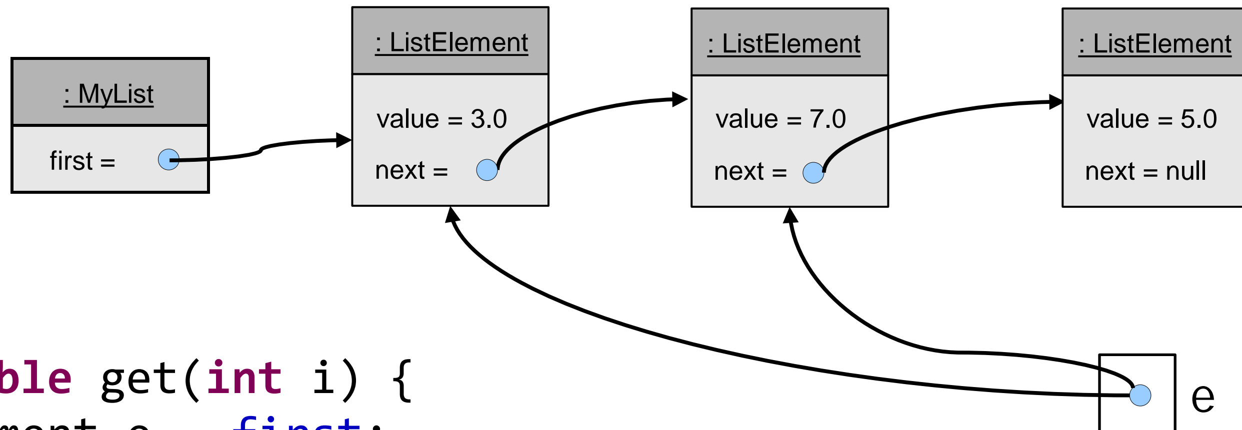
Entfernen und Zurückgeben des ersten Elements



Konstante Zeitkomplexität!

```
public class MyList {
   …
   public double removeFirst() {
      if (first == null) {
         throw new NoSuchElementException(“Die Liste ist leer!“);
      }
      double value = first.getValue();
      first = first.getNext();
      return value;
   }
```

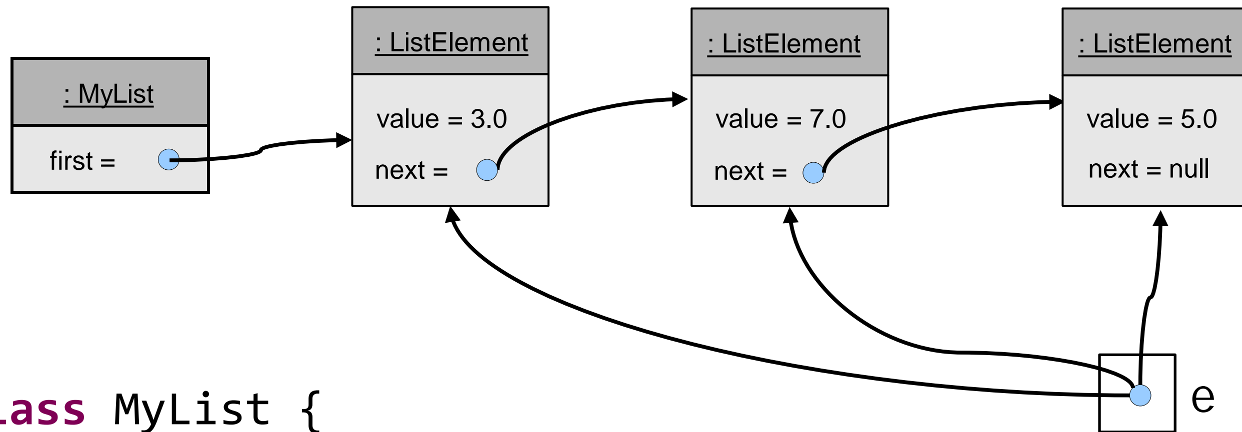
Zugriff auf Elemente der Liste




```
public double get(int i) {
    ListElement e = first;
    while (e != null && i > 0) {
        e = e.getNext();
        i--;
    }
    if (e == null) {
        throw new
        IndexOutOfBoundsException(“Falscher
        Index“);
    }
    return e.getValue();
}
```

Beachte: Laufzeit von `get(i)` ist im schlechtesten und durchschnittlichen Fall linear zu der Länge der Liste.

Berechnung der Länge der Liste

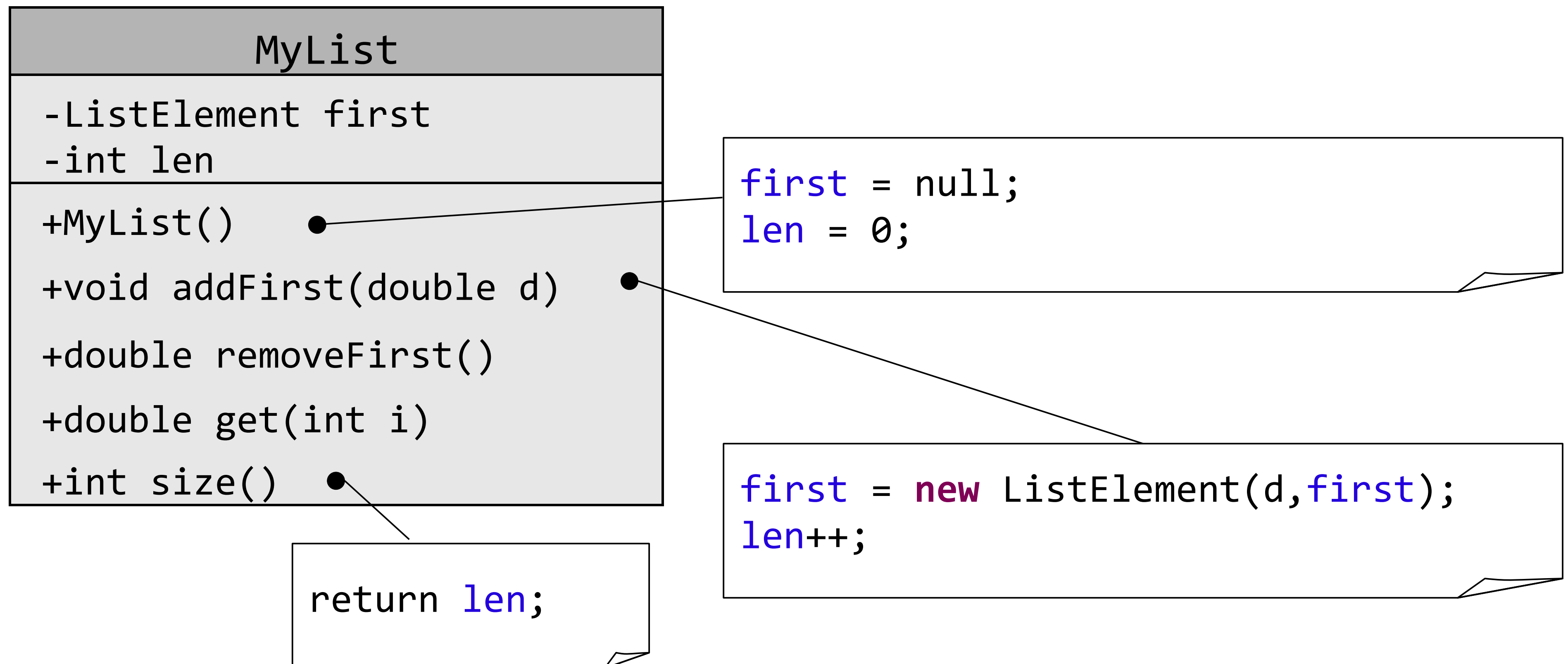



```
public class MyList {
   …
   public int size() {
      int size = 0;
      ListElement e = first;
      while (e != null) {
         e = e.getNext();
         size++;
      }
      return size;
   }
```

Beachte: Die Laufzeit von size() ist in jedem Fall linear zu der Länge der Liste.

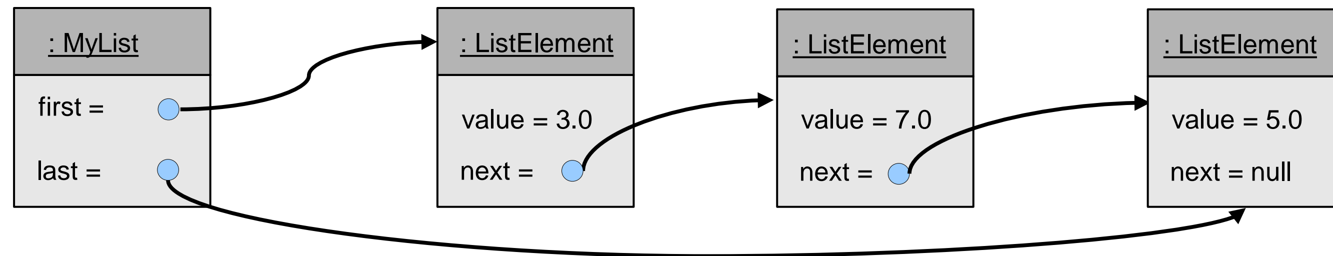
Zeiteffiziente Größenabfrage

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

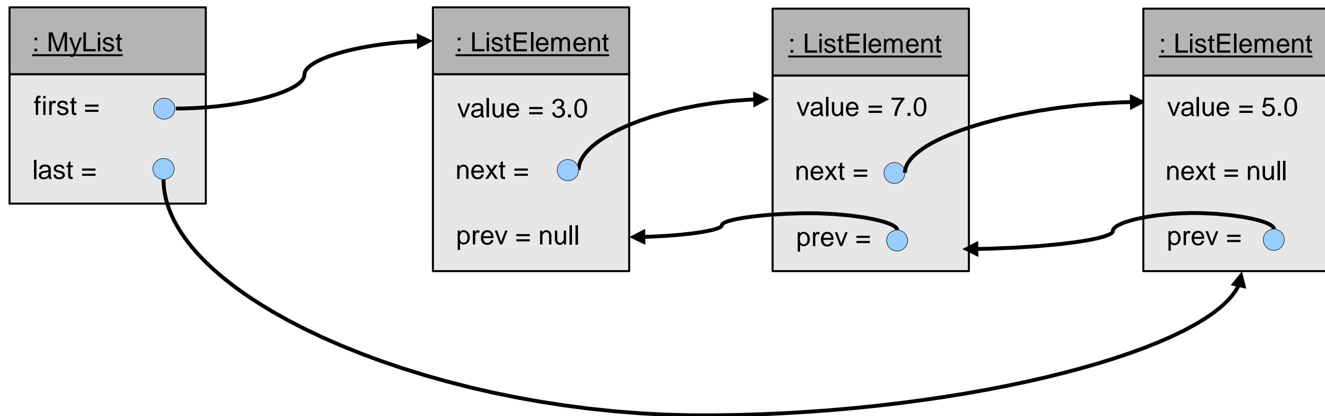


Einfügen am Ende

- Will man ein Element am Ende der Liste anfügen, so muss man erst die gesamte Liste durchlaufen, um das letzte Element zu finden (*lineare Zeit*).
- Mit einem zusätzlichen Zeiger zum letzten Element der Liste kann auch am Ende in *konstanter Zeit* eingefügt werden.
- Aber: Löschen am Ende benötigt dann immer noch lineare Zeit, da zum Auffinden des neuen letzten Elements die gesamte Liste durchlaufen werden muss.



Doppelt verkettete Listen



Löschen am Ende geht nun ebenfalls in *konstanter Zeit*.

Zusammenfassung Zeitkomplexität bei verketteten Listen

Zugriff auf i-tes Element: linear

Anfügen und Entfernen eines Elements am Anfang: konstant

Anfügen eines Elements am Ende:

- Einfach verkettete Listen: linear
- Einfach verkettete Listen mit Zeiger auf letztes Element: konstant

Entfernen eines Elements am Ende:

- Einfach verkettete Listen (mit oder ohne Zeiger auf letztes Element): linear
- Doppelt verkettete Listen: konstant

Listen in der Java-Bibliothek

Die Java-Bibliothek stellt Klassen für Listenimplementierungen zur Verfügung:

`LinkedList<E>`: doppelt verkettete Listen

`ArrayList<E>`: durch Arrays implementierte Listen

Diese Klassen können auf beliebige Elementtypen angewendet werden („generische“ Klassen) und können damit zur Implementierung von Listen für beliebige Elementtypen benutzt werden.

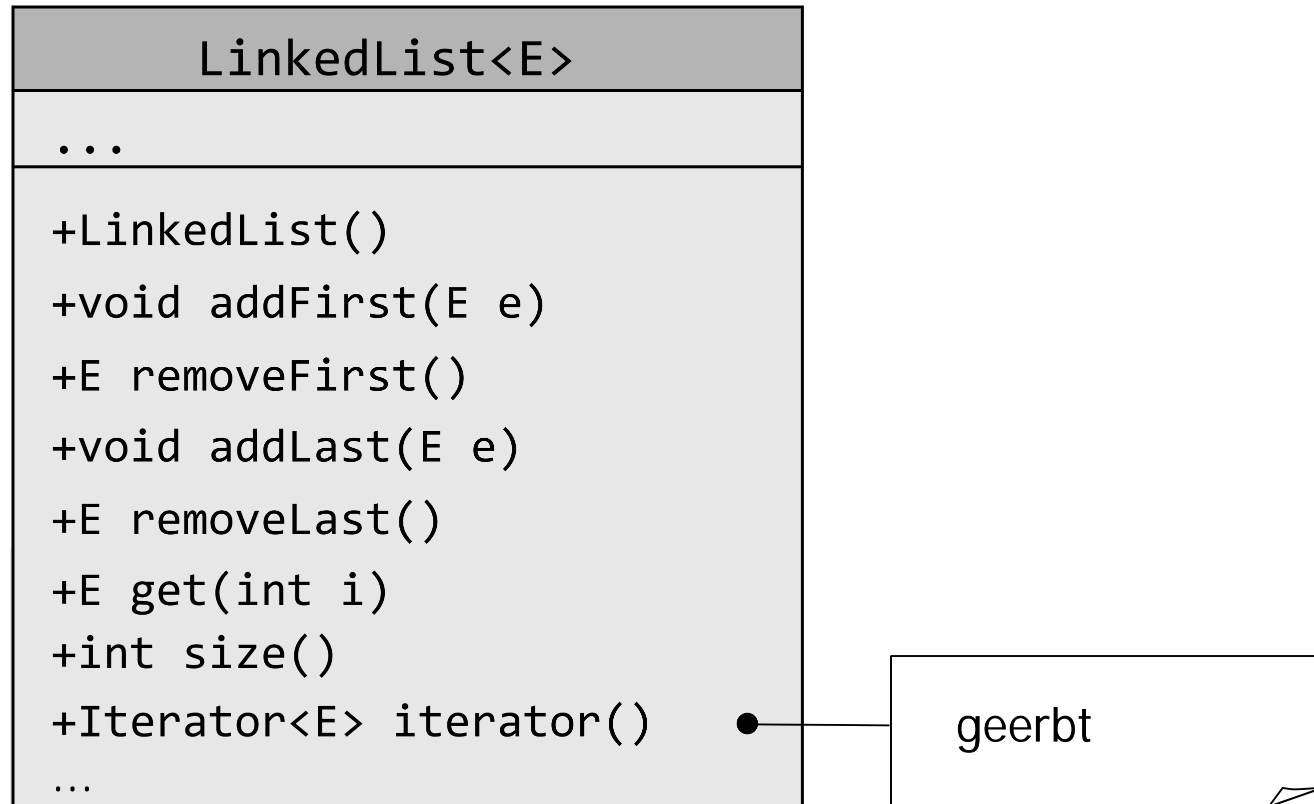
Beispiele:

```
LinkedList<BeliebigerKlassenName>
LinkedList<Point>
LinkedList<BankKonto>
LinkedList<Double>
LinkedList<Integer>
```

Beachte:

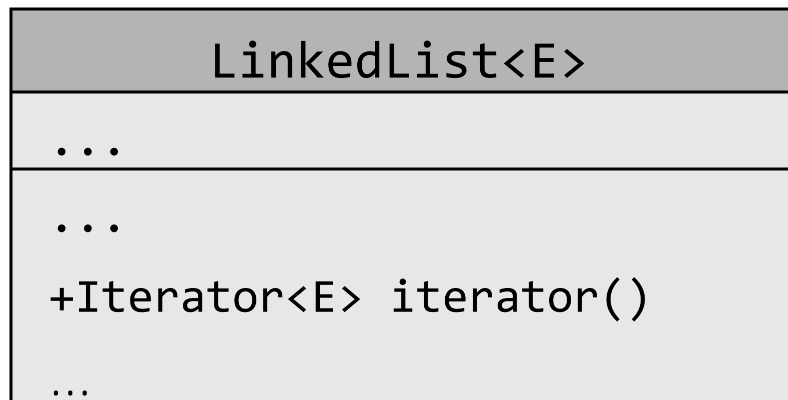
Bei Grunddatentypen sind die entsprechenden Klassen einzusetzen.

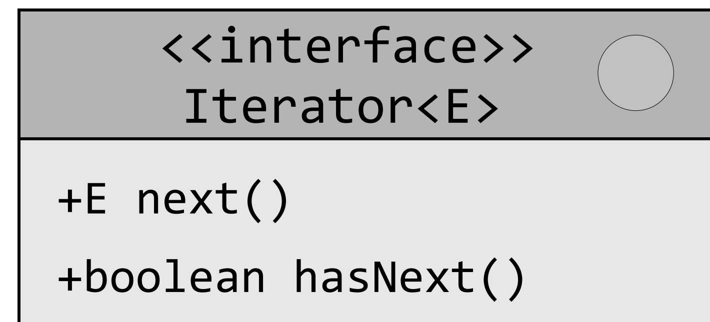
Einige Methoden der Klasse `LinkedList<E>`



Iteratoren

- Die Klassen für Listen in der Java-Bibliothek erlauben den Durchlauf von Listen mittels sogenannter **Iteratoren**.
- Ein Iterator ist ein Objekt, von dem man sich die Elemente der Liste eines nach dem anderen zurückgeben lassen kann.
- Die Methode `iterator()` erzeugt einen Iterator für eine Liste.
- Mit der Methode `hasNext()` stellt der Iterator fest, ob es ein nächstes zu besuchendes Element gibt.
- Der Aufruf der Methode `next()` auf dem Iterator liefert das nächste zu besuchende Listenelement (also zu Beginn das erste Element der Liste).
  Falls `next()` aufgerufen wird, wenn es kein nächstes Element gibt, wird eine `NoSuchElementException` geworfen.





Iteration mit Listen der Java-Standardbibliothek

Sei `C` eine Klasse und sei `list` eine Variable vom Typ `LinkedList<C>`.

Dann kann man auf folgende Art über die Liste `list` iterieren:

```
Iterator<C> it = list.iterator();
while (it.hasNext()) {
   C x = it.next();
   <Anweisungen>
}
```

Für diese häufig vorkommende Art der Iteration gibt es eine abkürzende Schreibweise:

```
for (C x : list) {
   <Anweisungen>
}
```

Verwendung von Iteratoren

```
public static void main(String[] args) {

   LinkedList<Integer> list = new LinkedList<Integer>();

   list.addFirst(new Integer(3));   // <3>
   list.addLast(new Integer(12);    // <3,12>
   list.addFirst(new Integer(72));  // <72,3,12>

   Iterator<Integer> it = list.iterator();
   while (it.hasNext()) {
      Integer k = it.next();
      System.out.println(k.intValue());
   }
}
/* Kurzform:
*
*  for (Integer k : list) {
*      System.out.println(k.intValue());
*  }
*/
```

Vergleich: Verkettete Listen und Arrays

Zeitkomplexität:

Zugriff auf i-tes Element:

Verkettete Liste: linear

Array: konstant

Anfügen und Entfernen eines Elements:

Verkettete Liste: konstant
(Beim Anfügen am Ende einen Verweis auf das letzte Element verwenden, beim Löschen am Ende mit doppelt verketteter Liste arbeiten.)

Array: Beim ersten Element linear wegen Umkopieren, beim letzten Element konstant oder, beim Anfügen linear, wenn Speicherplatz erweitert werden muss.

Folgerung:
Arrays eignen sich zur Behandlung von Sequenzen (Folgen) mit fester oder selten veränderter Anzahl von Elementen, während sich verkettete Listen besser eignen bei dynamischen Sequenzen, deren Länge sich häufig zur Laufzeit ändert.