# 3. Iteratoren

## 3.1. Motivation

Das Interface `Collection` abstrahiert verschiedenste Arten von *Sammlungen* wie *Mengen* oder *Listen*, implementiert z.B. mit einem Array oder als verkettete Liste. Es gibt in Anwendungen solcher *Collections* oft Aufgaben, die *für alle Elemente der Collection* zu lösen sind. Beispiele könnten sein: Summieren aller Zahlen in einer Menge oder das Entfernen aller Strings, die mit einem Grossbuchstaben beginnen, aus einer Liste. *Iteratoren* sind die Abstraktion, mit der Elemente einer *Collection* gelesen werden können.

## 3.2. Lernziele

- Sie können verschiedene Arten des Zugriffs auf alle Elemente einer *Collection* oder einer *Liste* unterscheiden, Vor- und Nachteile angeben und für spezifische Anwendungen sinnvoll auswählen.
- Sie können zu einer *Collection* einen *Iterator* programmieren.
- Sie können die möglichen Konflikte erklären, die entstehen, wenn eine *Collection* während der Iteration verändert wird und können eine Lösung erläutern und programmieren.
- Sie können für eine Liste einen *ListIterator* implementieren.

## 3.3. Operationen auf allen Elementen einer Collection

Möchte man eine Operation mit allen Elementen einer Collection durchführen, oder zumindest mit so vielen, bis man ein bestimmtes Ziel erreicht hat, genügen die drei grundsätzlichen Operationen `add`, `contains` und `remove` nicht. Deswegen sind im Interface `Collection` mehrere Methoden definiert, die auf verschiedene Möglichkeiten Zugriff auf den kompletten Inhalt der Datensammlung geben:

```
public interface Collection<E> {
  Object[] toArray();                          // includes new Object[size()]
  <T> T[] toArray(T[] a);                      // reuses a, if large enough, otherwise new T[]
  <T> T[] toArray(IntFunction<T[]> gen);       // calls gen to create an array to be filled

  Stream<E> stream();                          // create streams
  Stream<E> parallelStream();

  void forEach(Consumer<? super E> action); // calls action for each element in collection

  Iterator<E> iterator();                      // creates iterator (Interface see below)
  ...
}

public interface Iterator<E> {
 boolean hasNext();                            // checks if there are more elements available
 E next();                                     // returns next element
 void remove();                                // removes least recently returned element
}
```

Schliesslich gibt es auch noch die Kurzschreibweise mit einer `for`-Schleife, die vom Compiler exakt wie die Iteration mit einem `Iterator` übersetzt wird:

```
for (int e : c) { do something with e }
```

```
Iterator<Integer> it = c.iterator();
while (it.hasNext()) {
  e = it.next(); do something with e
}
```

entspricht dem erzeugten Byte Code

### Lernaufgabe

a) Programmieren Sie eine Methode `int sumOf(Collection<Integer> c)`, die alle in der Collection `c` enthalten Elemente aufsummiert. Welche der oben gezeigten Möglichkeiten benutzen Sie? Warum?

```
return c.parallelStream().mapToInt(Integer::valueOf).sum();
```

ist parallelisierbar!

oder

```
int sum = 0;
for (Integer i : c) sum += i;
```

ist einfach

b) Programmieren Sie eine Methode `void removeSecondElem(Collection<Integer> c, int v)`, die genau eines der Elemente v entfernt, falls es mehr als eines gibt: Wenn beim Aufruf c das Element v ein- oder keinmal enthält, soll c nicht verändert werden. Ansonsten wird die Anzahl enthaltener v-Elemente um eins reduziert. Nehmen Sie an, dass c insgesamt sehr viele Elemente enthält. Welche der gezeigten Zugriffs-Möglichkeiten benutzen Sie? Warum?

Iterator - kann als einziger aufhören, wenn 2. v gefunden
- kann als einziges Elemente entfernen

```
Iterator<Integer> it = c.iterator();
int vcnt = 0;
while (it.hasNext && cnt < 2) {
    if (it.next() == v) {
        cnt++;
        if (cnt == 2) it.remove();
    }
}
```

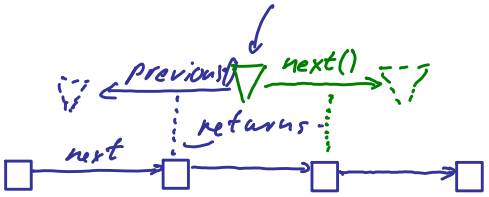
c) Ordnen Sie die folgenden Eigenschaften einer oder mehreren der Möglichkeiten zu, mit denen auf alle Elemente einer Collection zugegriffen werden kann:

1. Es werden immer *alle* Elemente der Collection bearbeitet. Vorzeitiger Abbruch, wie z.B. bei der sequenziellen Suche, ist *nicht* möglich.
2. Unterstützt funktionales Programmieren mit immutable Collections.
3. Es entsteht eine vollständige Kopie der Datenstruktur (nicht der enthaltenen Elemente). Diese kann verändert werden, ohne dass die originale Datenstruktur davon beeinflusst wird.
4. Einzelne bearbeitete Elemente können bei Bedarf direkt aus der Collection entfernt werden.
5. Der Aufwand sowohl für Speicher als auch für Laufzeit entspricht *O*(n), wobei n die Anzahl der Elemente in der Collection ist.
6. Erlaubt unter gewissen Randbedingungen sehr leicht Multi-Core Prozessoren durch parallele Verarbeitung auszunutzen.

### 3.4. ListIterator

Der im vorigen Kapitel gezeigte einfache `Iterator` wird für Listen zu einem `ListIterator` erweitert, der komfortables Bearbeiten von Listen erlaubt:

ListIterator: logische Iteratorposition zwischen 2 Elementen




```
public interface List<E> extends Collection<E> {
  ...
  ListIterator<E> listIterator();
  ListIterator<E> listIterator(int index);
}

public interface ListIterator<E> extends Iterator<E> {
  // Query Operations
  boolean hasNext();             // as for simple Iterator
  E next();
  boolean hasPrevious();         // moves in opposite direction
  E previous();
  int nextIndex();               // returns position left and right of Iterator
  int previousIndex();

  // Modification Operations
  void remove();                 // removes least recently by next or previous returned elem.
  void set(E e);                 // replaces least recently returned element
  void add(E e);                 // adds a new element at the current iterator position
}
```

### 3.5. Iterator auf LinkedList

Unten finden Sie eine zu einfache Art, einen `Iterator` auf einer verlinkten Liste zu implementieren:

```
class MyIterator<E> implements Iterator<E> {
  private List<E> list;
  private int next = 0;

  MyIterator(List<E> list) { this.list = list; }

  public boolean hasNext() { return next < list.size(); }

  public E next() { return list.get(next++); }

  public void remove() { … }
}
```

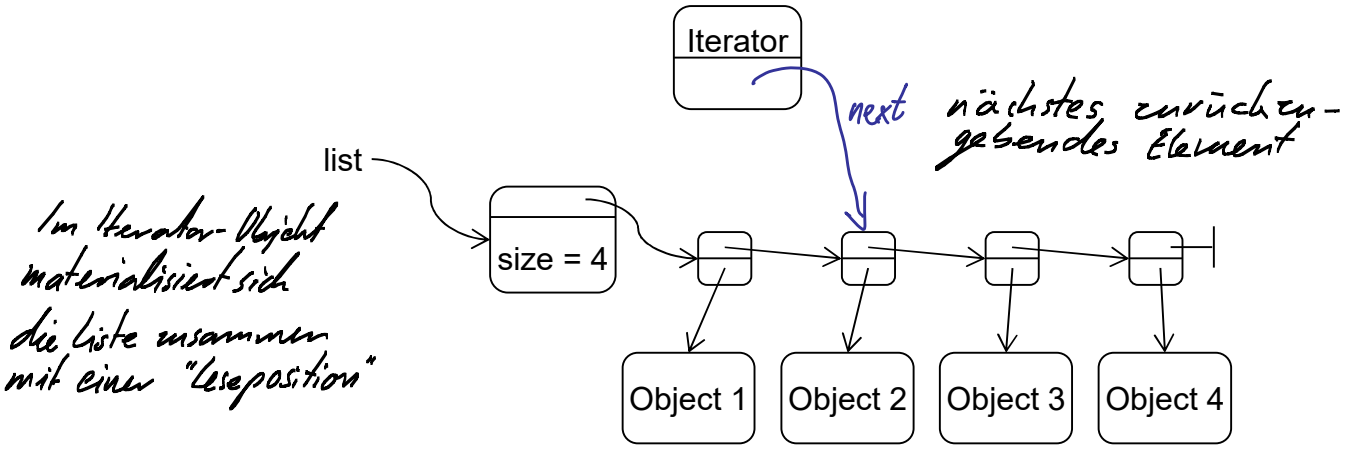
vgl. auch AbstractList ← AbstractSequentialList

O(n) (bei verlinkten Listen)

Das ist schlecht weil:

Der Zugriff mittels get(index) braucht eine Schleife also O(n)
Aufwand. Analyse von n Elementen braucht also $O(n^2)$ Operationen.

Das lässt sich natürlich deutlich besser lösen, wenn man den `Iterator` innerhalb der Listen-Klasse implementiert. Dann kann dieser direkt auf die `Node`-Objekte zugreifen und mit ihnen arbeiten:

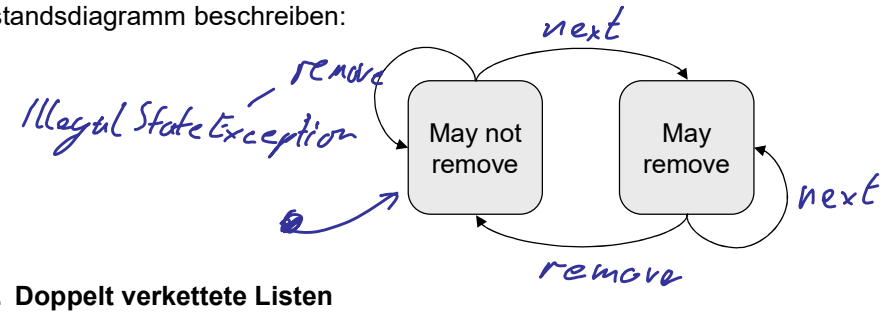



```
class MyIterator<E> implements Iterator<E> {
  private Node<E> next = first;

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

  public E next() {
     if (next == null) throw new NoSuchElementException();
     E e = next.elem;
     next = next.next;
     return e;
  }

  public void remove() {
    throw new UnsupportedOperationException("Don't know yet how to do this.");
  }
}
```

### 3.6. Remove-Methode im Iterator

Bisweilen möchte man Elemente mit einer bestimmten Eigenschaft aus einer Collection entfernen. Dann benutzt man einen `Iterator`, um alle Elemente zu prüfen. Ergibt die Prüfung eines Elements, dass man es entfernen möchte, benutzt man dazu idealerweise gerade wieder den `Iterator`, der ja noch „weiss“, wo in der Liste sich das Element befindet.

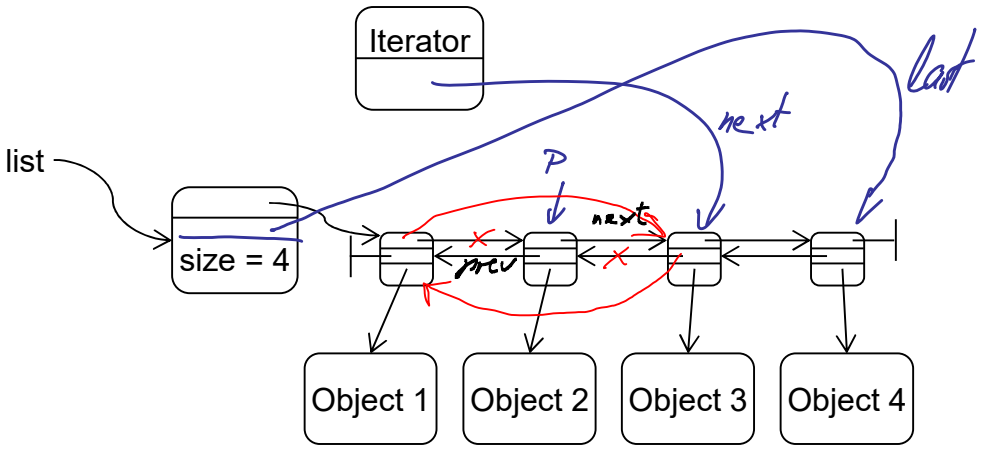
Die `remove`-Methode im Interface `Iterator` ist so definiert, dass nach einem Aufruf von `next` genau das dabei zurückgegebene Element entfernt werden kann. Anschliessend darf `remove` erst wieder aufgerufen werden, wenn zwischenzeitlich `next` aufgerufen wurde[1]. Diese Spezifikation kann man gut mit einem Zustandsdiagramm beschreiben:




Programmieren mit "State-Variable"
z.B.:
boolean mayRemove = false;

### 3.7. Doppelt verkettete Listen

Mit der bisher betrachteten `Iterator`-Implementierung ist es möglich aber umständlich, das zuletzt zurückgegebene Element zu entfernen, weil dafür das `Node`-Objekt *vor* dem zuletzt zurückgegebenen benötigt wird und Spezialfälle berücksichtigt werden müssen. Eine sehr effektive Lösung dieser Schwierigkeit besteht darin, die `Node`-Objekte zusätzlich mit einem *Rückwärtszeiger* zu verketten:




```
private static class Node<E> {
  private E elem;
  private Node<E> prev, next;

  private Node(E elem) { this.elem = elem; }

  private Node(Node<E> prev, E elem, Node<E> next) {
    this.prev = prev; this.elem = elem; this.next = next;
  }
}
```

[1] Sie wissen ja mittlerweile sicher, wie Sie in der Java-Dokumentation solche Details finden und nachlesen können.

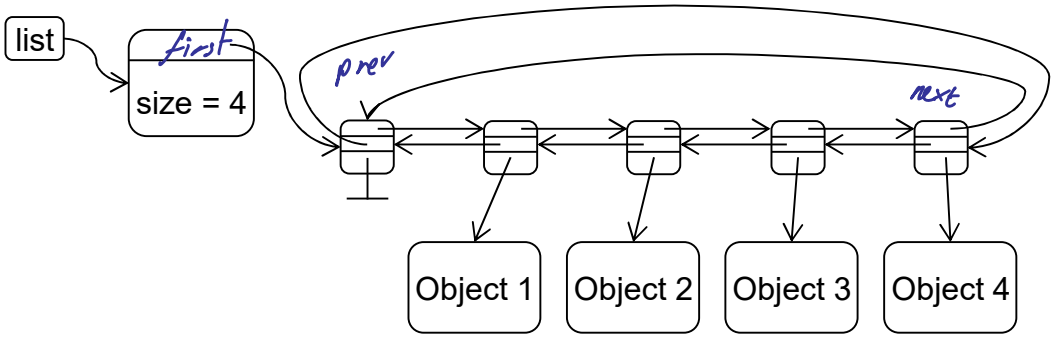
### 3.8. Remove aus doppelt verketteter Liste

Damit kann man die `remove`-Methode gut implementieren. Lästig sind jedoch die Spezialfälle an den Rändern…

```
class MyIterator<E> implements Iterator<E> {
  private boolean mayRemove = false;
  private Node<E> next = first;

  public boolean hasNext() { ... }
  public E next() { ... mayRemove = true; ... }
  public void remove() {
     if (!mayRemove) throw new IllegalStateException();
     Node<E> p = next != null ? next.prev : last;
     if(p.prev != null) p.prev.next = p.next;
     else first = p.next;
     if(p.next != null) p.next.prev = p.prev;
     else last = p.prev;
     mayRemove = false; size--;
  }
}
```

### 3.9. Zyklisch verkettete Liste mit Kopfelement (Ringliste)

Eine elegante Art, die Unterscheidung von Spezialfällen während der Ausführung zu vermeiden, ist ein zusätzliches unbenutztes `Node`-Objekt einzuhängen und damit einen Kreis zu schliessen:




Die Methode `remove` wird damit einfacher, `hasNext` etwas komplizierter:

```
private Node<E> next = first.next;
public boolean hasNext() {
    return next != first;

}
public void remove() {
    if (!mayRemove) throw new IllegalStateException;
    Node<E> p = next.prev;
    p.next.prev = p.prev;
    p.prev.next = p.next;
    mayRemove = false; size--;
}
```

```
if (modCount != list.modCount)
    throw, ConcurrentModification
    new        Exception
```

### 3.10. Concurrent Modification

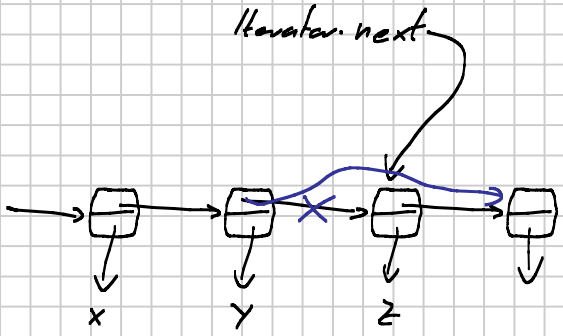
Zu einer Liste können jederzeit beliebig viele `Iterator`-Objekte bestehen. Jedes davon hat Referenzen auf `Node`-Objekte der Liste und ist in der Annahme programmiert, dass diese `Node`-Objekte auch korrekt Teil der Liste sind.

Das stimmt nicht mehr, wenn zum Beispiel über die `remove`-Methode der Liste selbst oder eines anderen `Iterator`-Objekts das als nächstes zurückzugebende Element (`.next`) entfernt wird. s. nächste Seite (→ Tafel)

*Lösungskonzept:* Um solche Probleme in den Griff zu bekommen gibt es einen verbreiteten Ansatz. Man führt als Attribut der Liste einen *Generationszähler* für die Daten. Dieser hat beim Erzeugen einer neuen Liste den Wert 0 und wird bei jeder Veränderung (hinzufügen, entfernen) der Daten um 1 erhöht.

In den *Iteratoren* wird die Nummer derjenigen Generation gespeichert, auf die sich der aktuelle Zustand des *Iterators* bezieht. Stimmt dieser Wert nicht mehr mit der aktuellen Generation der Liste überein, wird der *Iterator* ungültig und kann nicht mehr verwendet werden. Gemäss Spezifikation sollen in diesem Fall die Methoden `next` und `remove` eine `ConcurrentModificationException` werfen.

Zu beachten bleibt ein Spezialfall: Wird mittels der `remove`-Methode eines `Iterators` ein Element aus der Liste entfernt, muss natürlich auch dabei der Generationenzähler der Liste erhöht werden, damit andere `Iteratoren` die Zustandsänderung bemerken. Der entfernende `Iterator` selbst aber „weiss, was er tut“ und sollte in der Lage sein, weiter zu funktionieren. Dazu muss die im `Iterator` abgelegte Generationsnummer in diesem Fall ebenfalls nachgeführt werden.




l.remove(z)
it.next() → z
obwohl nicht mehr in der Liste!