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> {
   // toArray: returns an array of all elements
   Object[] toArray();
   <T> T[] toArray(T[] a);
   <T> T[] toArray(IntFunction<T[]> gen);

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

   // ForEach: Calls action for each element
   void forEach(Consumer<? super E> action);

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

public interface Iterator<E> {
   boolean hasNext();                    // checks if more elements are 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
}
```

Passend: Arbeitsblatt Aufgabe 1.

3.4. ListIterator

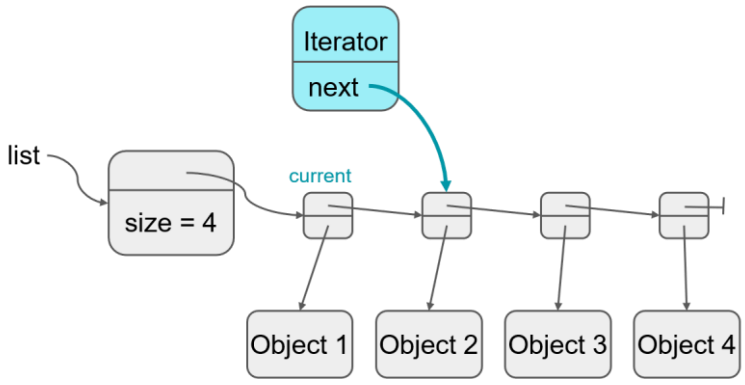
Der im vorigen Kapitel gezeigte einfache `Iterator` wird für Listen zu einem `ListIterator` erweitert, der komfortables Bearbeiten von Listen erlaubt:

```
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 element
   void set(E e);            // replaces least recently returned element
   void add(E e);            // adds a new element at the current iterator position
}
```

Analog zu den indexierten Methoden im `ListInterface`, gibt es auch einen `ListIterator`, der direkt auf einem gewünschten Index startet.

3.5. Iterator auf LinkedList

Um effizient über die Node-Objekte einer Liste zu iterieren, wird der `Iterator` direkt innerhalb der Listen-Klasse implementiert. Wichtig dabei ist, dass der Iterator einen next-Zeiger auf die nächste Node hat, aber *keinen* Zeiger auf die zuletzt zurückgegebene (current) Node.

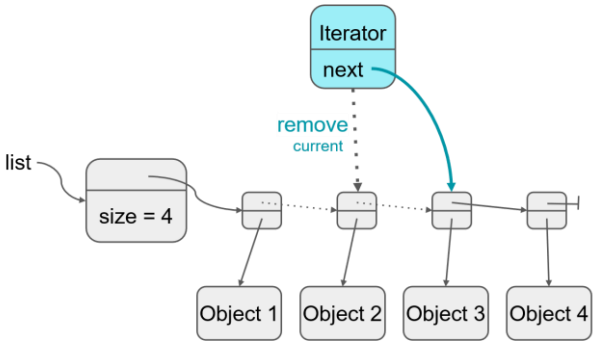


Ein Aufruf von next() gibt das Objekt zurück auf dessen Node der next-Zeiger zeigt, anschliessend wandert der Zeiger eine Position weiter.

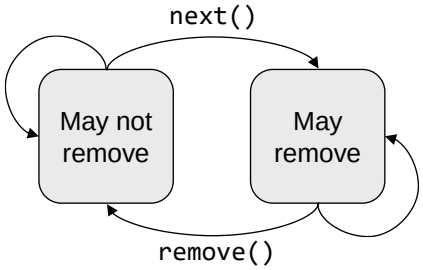
Passend: Arbeitsblatt Aufgabe 2, Programmieren Aufgabe 1.B

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:

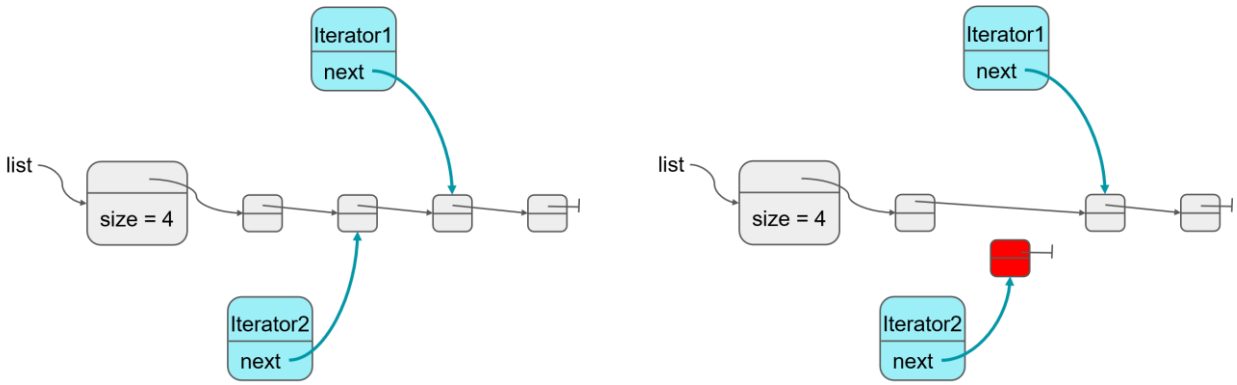


Passend: Programmieren Aufgabe 1.D

3.7. 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.



Lösungskonzept: Um solche Probleme in den Griff zu bekommen gibt es einen verbreiteten Ansatz. Man führt als Attribut der Liste einen *Generationszähler (mod-count / modification count)* 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.

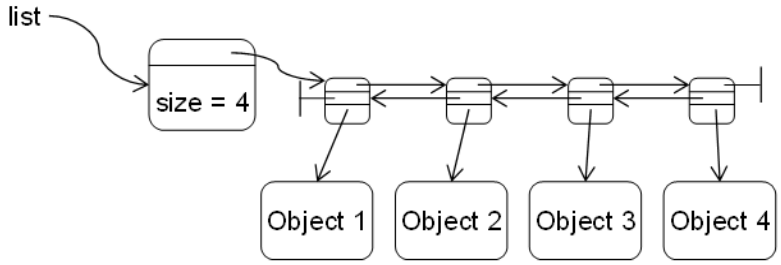
[1] Sie wissen ja mittlerweile sicher, wie Sie in der Java-Dokumentation solche Details finden und nachlesen können.

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.

Passend: Programmieren Aufgabe 1.C

3.8. 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:



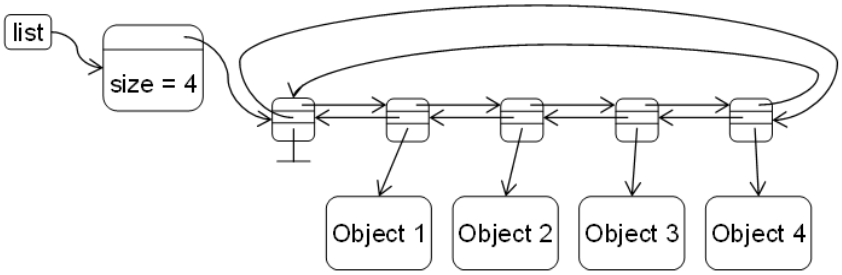
In Java würde dann die Klasse Node etwa so aussehen:

```
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;
   }
}
```

Damit kann man die `remove`-Methode einfacher implementieren und muss sich keine extra Vorgängerknoten mehr merken. Etwas Vorsicht ist bei den Spezialfällen an den Rändern geboten.

3.9. Zyklisch verkettete Liste mit Kopfelement

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:



Damit wird die Methode `remove` noch etwas einfacher.

Passend: Programmieren 2