03 Iteratoren - Arbeitsblatt

1. Verschieden Möglichkeiten zum Iterieren

a) Sei `int sumOf(Collection<Integer> c)` eine Methode, die alle in der Collection `c` enthalten Elemente aufsummiert. Programmieren Sie die Methode auf mindestens zwei verschiedene Arten. Welches sind die Vor- und Nachteile?

toArray:

```
private static int sumToArray(Collection<Integer> c){
  int sum = 0;
  Integer[] A = c.toArray(new Integer[c.size()]);
  for(int i=0; i<A.length; i++) {
     sum += A[i];
  }
  return sum;
}
```

Nachteil: Alles wird kopiert, braucht also O(n) zusätzlichen Speicher. Vorzeitiges Abbrechen beim iterieren spart asymptotisch keine Zeit, da bereits alles einmal kopiert wurde.

Iterator:

```
private static int sumForLoop(Collection<Integer> c) {
   int sum = 0;
   for (Integer e : c)
      sum += e;
   return sum;
}

private static int sumIterator(Collection<Integer> c) {
   int sum = 0;
   Iterator<Integer> it = c.iterator();
   while (it.hasNext()) {
      sum += it.next();
   }
   return sum;
}
```

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?

```
private static <T> void removeSecondElemIterator(Collection<T> c, T v) {
  int cnt = 0;
  Iterator<T> it = c.iterator();
  while (it.hasNext() && cnt < 2) {
     T e = it.next();
     if (e.equals(v)) {
        cnt++;
        if (cnt == 2) it.remove();
     }
  }
}
```

Einzige Möglichkeit, um frühzeitig abzubrechen und Collection direkt zu verändern.

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

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

2. Einfacher ListIterator

Untenstehend finden Sie eine zu einfache Art, einen `Iterator` auf einer verlinkten Liste zu implementieren. Warum ist diese Implementierung nicht gut?

```
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() { … }
}
```

Das Problem hier ist die list.get() Methode. Bei jedem Aufruf wird von vorne bis zum Index next++ durch die Liste gegangen. Um auf jedes Element in der Liste einmal zuzugreifen (1mal komplett über die Liste iterieren) braucht man darum O(n^2) Zeit, statt nur O(n):

O(1) + O(2) + O(3) + … + O(n) = O(n^2)

Aus diesem Grund wird ein ListIterator direkt in der Liste selber implementiert. Dann kann mit einem next-Zeiger direkt auf die nächste Node zugegriffen werden und es muss nicht jedesmal die teure get() Methode aufgerufen werden.