

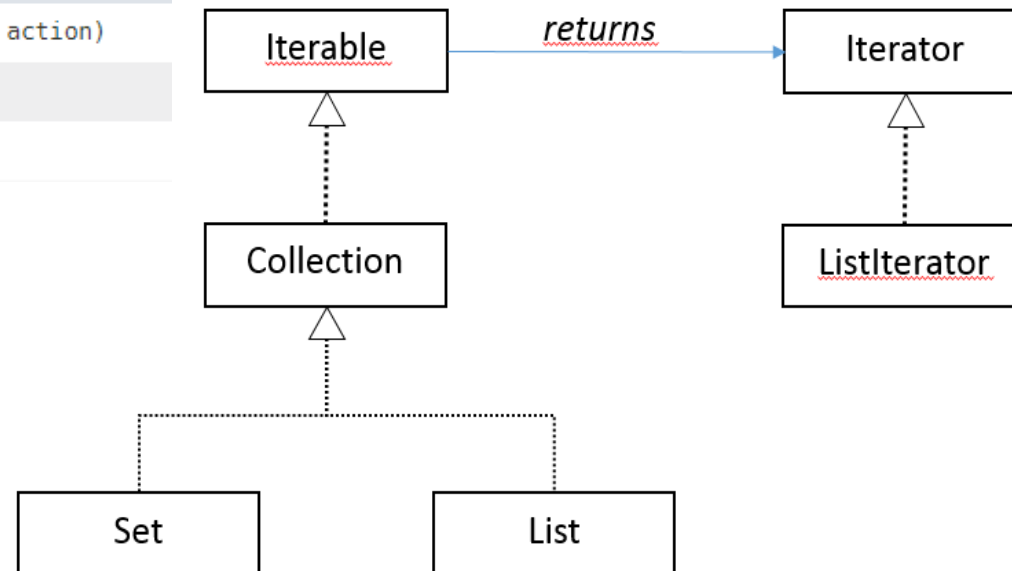
Iteratoren

Algorithmen und Datenstrukturen 2

- Grüne Farbe: Bitte im Script nachtragen

Iterable Interface

Modifier and Type	Method
default void	<code>forEach(Consumer<? super T> action)</code>
<code>Iterator<T></code>	<code>iterator()</code>
default <code>Splitter<T></code>	<code>spliterator()</code>



```

public interface Collection<E> {
    ...
    Iterator<E> iterator();
    ...
}
  
```

```

public interface Iterator<E> {
    boolean hasNext();
    E next();
    void remove();
}
  
```

Verwendung und Kurzschreibweise für Iteratoren (Kap. 3.3)

- Der erzeugte Byte-Code ist genau gleich

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

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

Möglichkeiten, um auf alle Elemente einer Collection zuzugreifen

```
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)  
    ...  
}
```

Lösen Sie die Lernaufgaben a.) – c.) im Abschnitt 3.3 (Seite 1 unten)

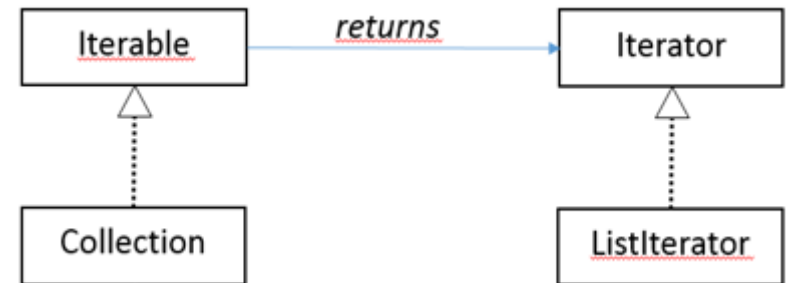
List-Iterator

```
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
}
```

```
public interface Iterator<E> {
    boolean hasNext();
    E next();
    void remove();
}
```



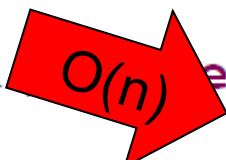
Externer Iterator auf unsere MyLinkedList

```
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() { throw new UnsupportedOperationException(); }  
}
```

Das ist schlecht, weil:

Externer Iterator auf unsere MyLinkedList

```
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() { throw new UnsupportedOperationException(); }  
}
```

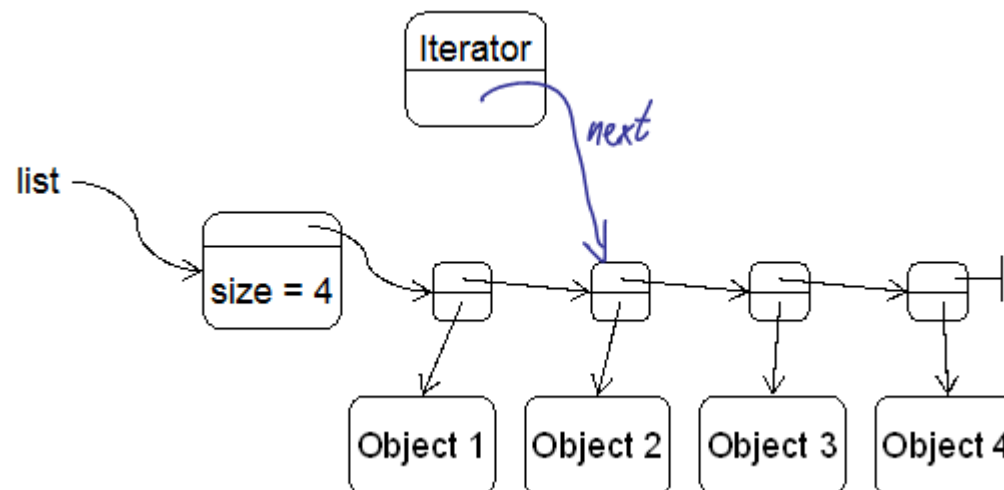


Das ist schlecht, weil: Der Zugriff über die `get(index)` – Methode braucht eine Schleife ($O(n)$). Somit liegt der Zugriff auf n Elemente in $O(n^2)$.

Interner Iterator in der MyLinkedList

- Direkter Zugriff auf interne Struktur der Liste
- Innere Iterator-Klasse, welche Iterator-Interface implementiert
- next-Zeiger jeweils auf den nächsten Node, dessen Element zurückgegeben wird

```
public interface Iterator<E> {  
    boolean hasNext();  
    E next();  
    void remove();  
}
```

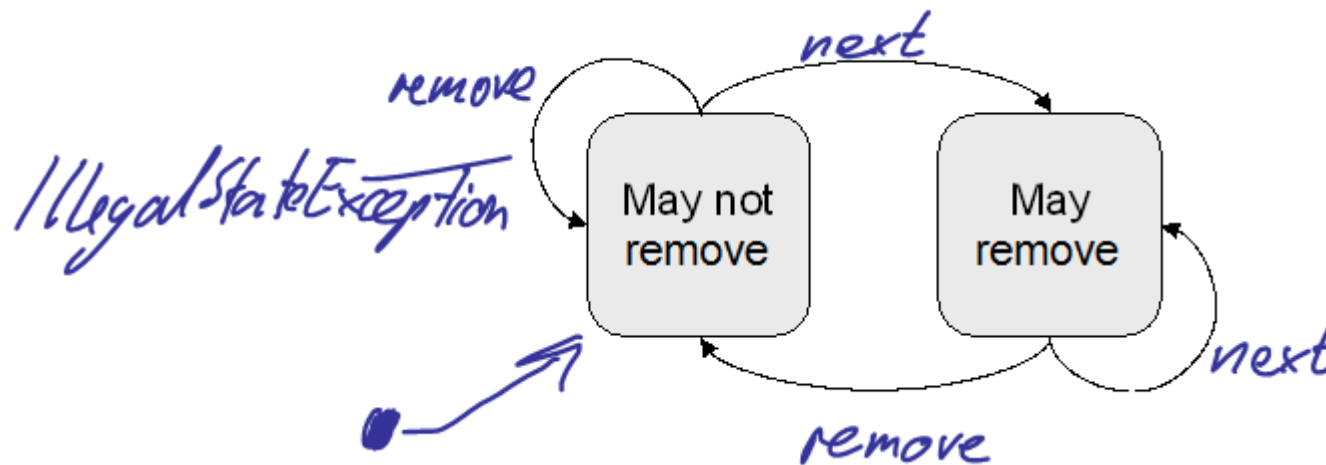


Implementieren Sie einen internen Iterator (Script, Seite 2 oben) von Hand

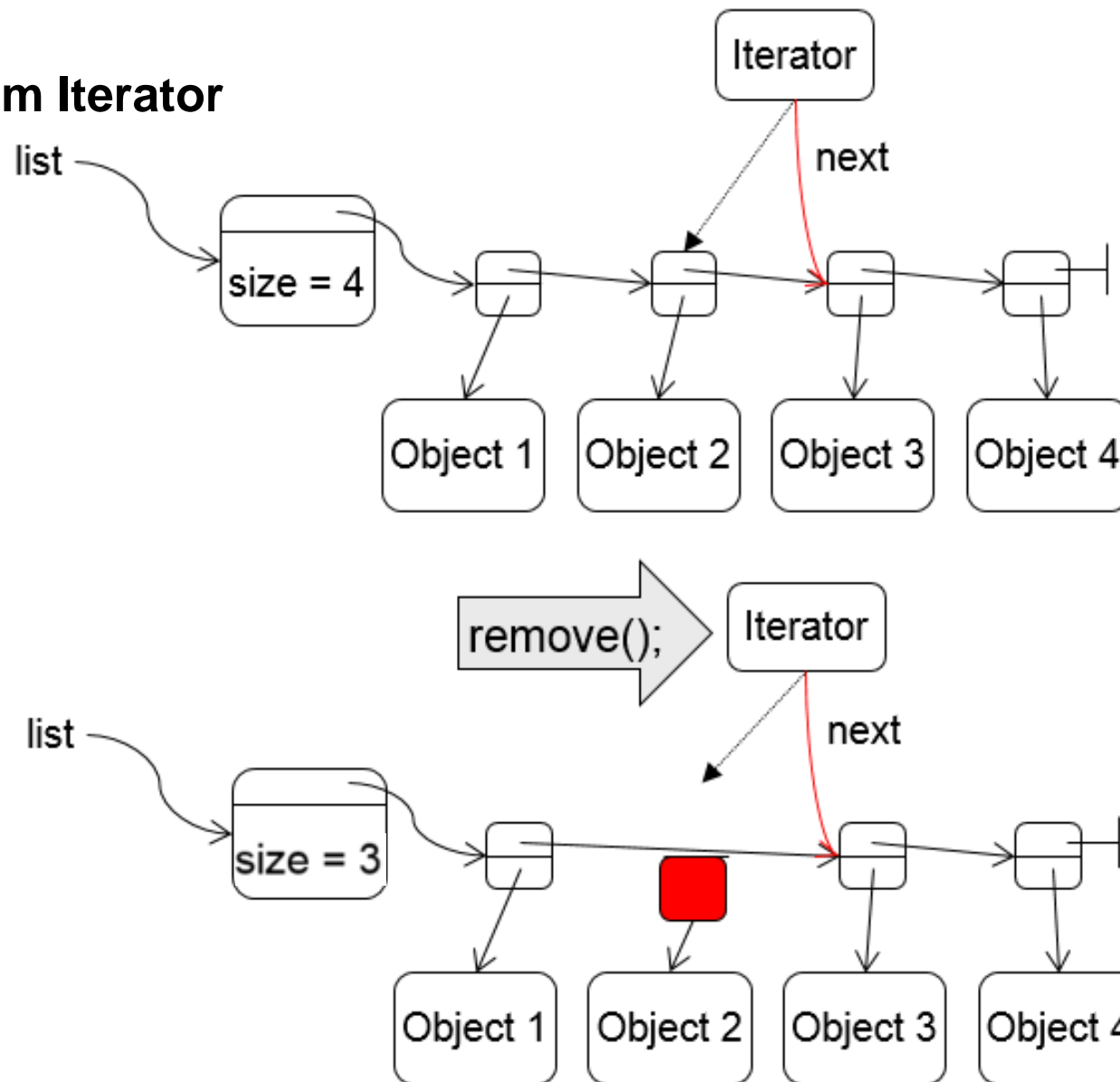
- next-Zeiger jeweils auf den Node, dessen Element als nächstes zurückgeliefert wird
- Der Aufruf von next() führt dazu, dass das aktuelle Element zurückgegeben wird und der next-Zeiger ein Element weiter springt
- Es wird eine «NoSuchElementException» geworfen, falls der next-Zeiger auf keinen gültigen Node mehr zeigt

Löschen auf dem Iterator

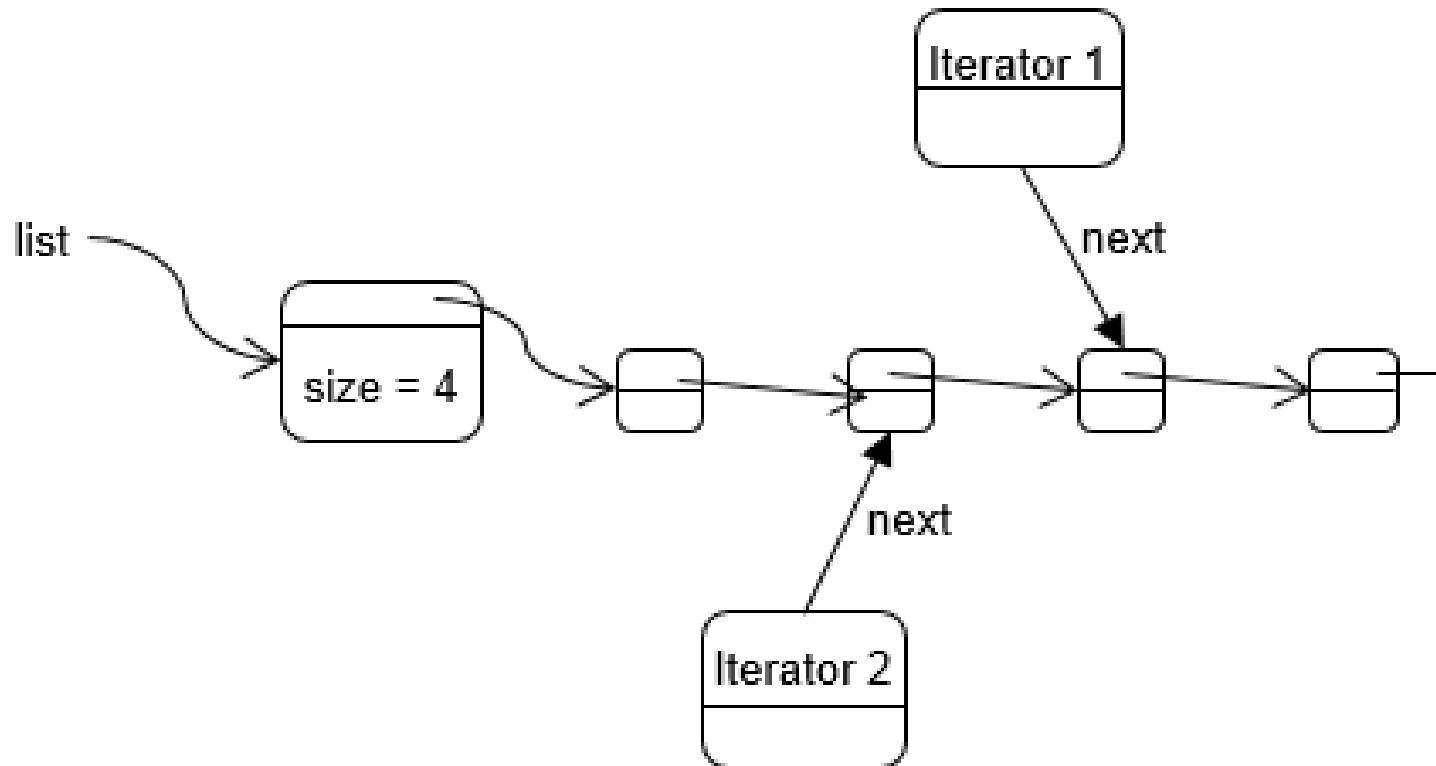
- Methode `remove()` löscht zuletzt zurückgegebenes Element (von `next()`)
- Auf ein `next()`- Aufruf darf maximal ein `remove()` erfolgen.



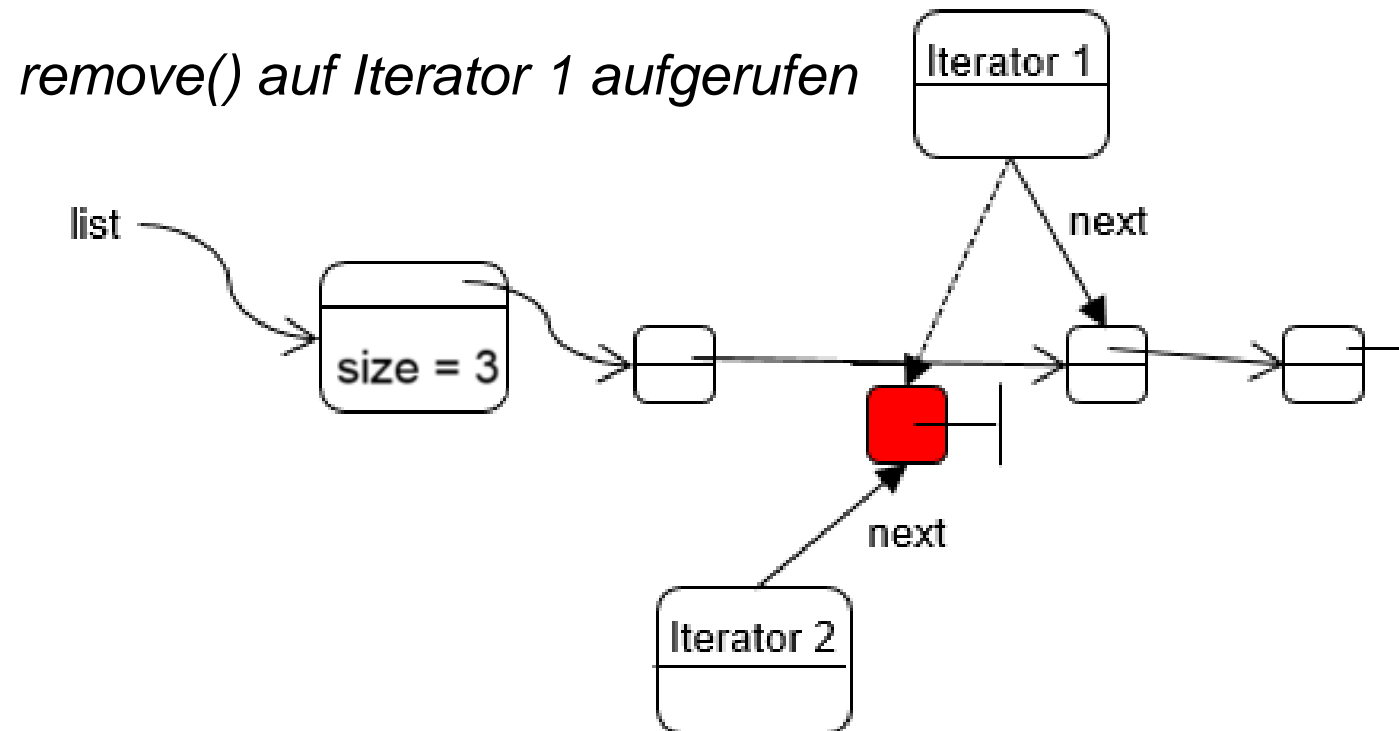
Löschen auf dem Iterator



Mögliches Problem: Mehrere Iteratoren



Mögliches Problem: Mehrere Iteratoren (2)



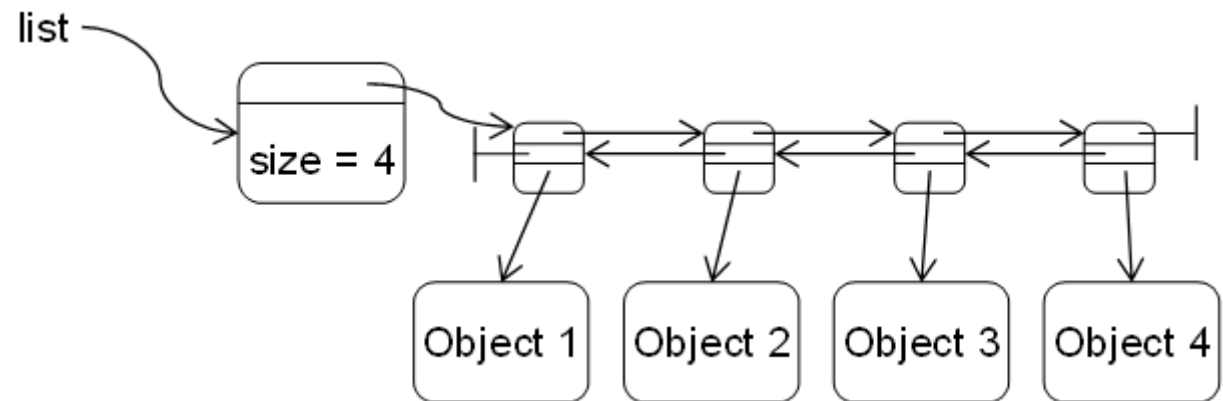
Einfache Lösung: Modification Counter / Generationszähler

- Liste hält eine Zählvariable «modCount» (Modification Counter)
- Listenstruktur verändernde Operationen **erhöhen diesen Wert jeweils um 1**:
 - Add
 - Remove
- Iterator kopiert bei Instanziierung den modCount
- Iterator prüft regelmässig, ob die beiden modCount (Liste und Iterator) die selben sind:
 - next()- Operationen
 - remove() – Operationen
- ConcurrentModificationException, falls Werte unterschiedlich sind:
 - **Ausnahme**: iterator.remove() inkrementiert auch den Iterator ModCount

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

Doppelt verkettete Listen (Varianten)

Doppelt verkettete Liste:



Doppelt verkettete Ringliste mit Dummy-Head:

