Kapitel 5: Iterierbare Container

- Foreach-Schleife
- Interface Iterator
- Interface Iterable
- Iterator-Schleife und Foreach-Schleife
- Iterierbare generische Liste

foreach-Schleife

- Das Durchlaufen von Feldern mit einer foreach-Schleife ist eine bequeme und sichere Alternative zu der sonst üblichen Zählschleife:

```
int[] a = {1,2,3,4};

int sum = 0;
for (int x : a)
    sum += x;
```

```
int[] a = {1,2,3,4};

int sum = 0;
for (int i = 0; i < a.length; i++)
    sum += a[i];
```

- Die foreach-Schleife ist abstrakter: für jedes Element x in einem Feld a führe eine Anweisung aus. Wie auf das Feldelement zugegriffen wird, bleibt verborgen.

- Wünschenswert ist eine foreach-Schleife auch für andere Container wie z.B. Listen:

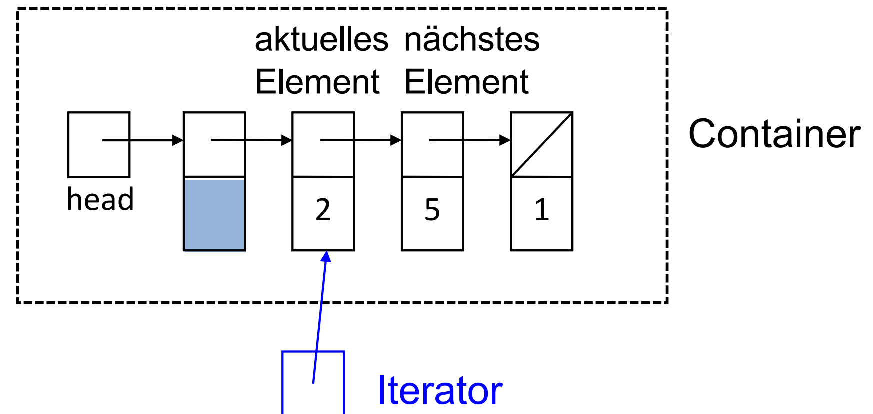
```
List<Integer> list = new LinkedList<>();
list.add(1); list.add(2); list.add(3); list.add(4);

int sum = 0;
for (int x : list)
    sum += x;
```

- Ermöglicht wird das durch die Definition sogenannter Iteratoren.

Interface Iterator aus java.util

- Ein Iterator kann man sich vorstellen wie ein Zeiger (Referenz), der auf ein aktuelles Element eines Containers verweist. Ein Iterator wird über folgende Operationen gesteuert, die in der Java-API in einem Interface festgelegt sind:

```
public interface Iterator<T> {

    boolean hasNext();
    T next();
    void remove();
}
```



- Mit hasNext() wird geprüft, ob es noch ein nächstes Element gibt.

- Mit next() wird das jeweils nächste Element des Containers zurückgeliefert.
  Der erste Aufruf liefert das erste Element, der zweite Aufruf das zweite Element usw.
  Falls es kein nächstes Element gibt, wird eine NoSuchElementException ausgelöst.

- Mit remove() wird das zuletzt mit next() gelieferte Element gelöscht.
  remove() darf nach jedem erfolgreichen next() nur einmal aufgerufen werden.

Interface Iterable aus java.lang

- Ein Container, der iterierbar sein soll, muss das Interface Iterable implementieren. Damit ist auch eine Implementierung des Iterator-Interface verknüpft.

```
public interface Iterable<T> {
    Iterator<T> iterator();
}
```

```
public interface List<T> extends Iterable<T> {
    // wie gehabt
}
```

```
public class LinkedList<T> implements List<T> {
    // ...

    public Iterator<T> iterator() {
        return new LinkedListIterator();
    }

    private class LinkedListIterator implements Iterator<T> {
        // ...
        boolean hasNext() { ... }
        T next() { ... }
        void remove() { ... }
    }
}
```

Implementierung des Iterable-Interface

Implementierung des Iterator-Interface als innere Klasse (nicht-statische geschachtelte Klasse).
Damit ist jedes erzeugte Iterator-Objekt mit dem Listen-Objekt gekoppelt.

Iterator-Schleife und foreach-Schleife

- Damit ist nun folgende Iterator-Schleife für LinkedList möglich:

```
List<Integer> list = new LinkedList<>();
list.add(1); list.add(2); list.add(3); list.add(4);

int sum = 0;
for (Iterator<Integer> it = list.iterator(); it.hasNext();) {
     int x = it.next();
     sum += x;
}
```

- Bequemerweise kann die Iterator-Schleife auch durch eine einfache foreach-Schleife ersetzt werden (der Java-Compiler übersetzt die foreach- in die obere Iterator-Schleife):

```
List<Integer> list = new LinkedList<>();
list.add(1); list.add(2); list.add(3); list.add(4);

int sum = 0;
for (int x : list)
     sum += x;
```

Iterator-Schleife mit remove

- Diese Methode löscht alle geraden Zahlen in einem Listen-Container und zeigt die Funktionsweise der Methode Iterator.remove():

```
public static void removeEven(List<Integer> list) {
    Iterator<Integer> it = list.iterator();

    while(it.hasNext())
        if (it.next() % 2 == 0) // nächstes Element ist gerade
            it.remove();

}
```

Nicht über einen Container iterieren, der verändert wird!

- Wird mit einem Iterator ein Container durchlaufen, dann ist es naheliegend, dass nicht gleichzeitig der Container (über seine Methoden) geändert werden kann, ohne die Iteratoren entscheidend zu stören.

- Folgende Schleife sollte daher eine ConcurrentModificationException auslösen:

```
int idx = 0;
for (int x : list) {
    if (x % 2 == 0)
        // Don't do this!
        list.remove(idx);
    else
        idx++;
}
```

Container list wird über list.remove(idx) verändert.

Es wird eine ConcurrentModificationException ausgelöst.

Iterierbare generische Liste

```
public interface List<T>
extends Iterable <T> {
    void add(T x);
    void add(int idx, T x);
    T set(int idx, T x);
    T get(int idx);
    void remove(int idx);
    int size();
    void clear();
    boolean isEmpty();
}
```

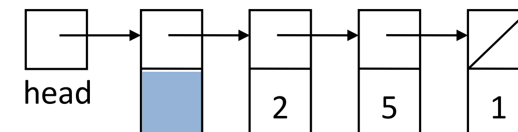
```
public class ArrayList<T>
implements List<T> {

   // ...
}
```

```
public class LinkedList<T>
implements List<T> {

   // ...
}
```

LinkedList mit Iteratoren ohne remove (1)

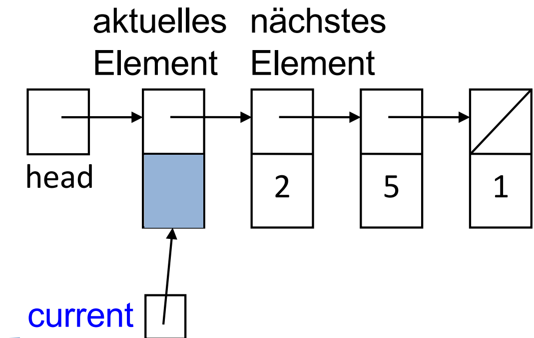
```
public class LinkedList<T> implements List<T> {

    public LinkedList() {clear();}

    public final void clear() {
        head = new Node<T>(null,null);
        size = 0;
        modCount++;
    }

    public void add(int idx, T x) {
        // ...
        modCount++;
    }

    // ...

    private static class Node<T> {
        T data;
        Node<T> next;
        Node(Node<T> p, T x) {
            data = x;
            next = p;
        }
    }

    private Node<T> head;
    private int size;
    private int modCount = 0;
}
```

Modifikationszähler wird hochgezählt, sobald sich die Struktur der verketteten Liste ändert
(d.h. Elemente werden gelöscht oder eingefügt)



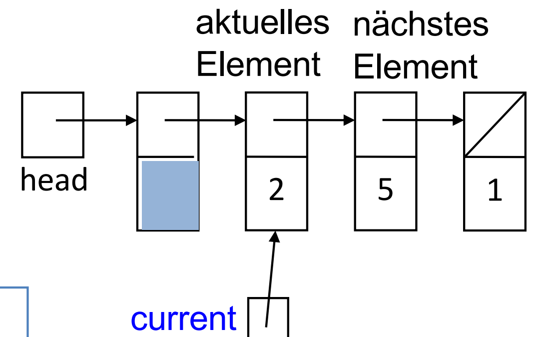
LinkedList mit Iteratoren ohne remove (2)

```
public class LinkedList<T> implements List<T> {
    // ...

    public Iterator<T> iterator() {
        return new LinkedListIterator();
    }

    private class LinkedListIterator implements Iterator<T> {
        private Node<T> current = head;
        private int expectedMod = modCount;

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

        public T next() {
            if (expectedMod != modCount)
                throw new ConcurrentModificationException();
            if (!hasNext())
                throw new NoSuchElementException();
            current = current.next;
            return current.data;
        }

        public void remove() {
            throw new UnsupportedOperationException();
        }
    }
}
```



Bei Konstruktion des Iterators wird modCount in expectedMod gespeichert.

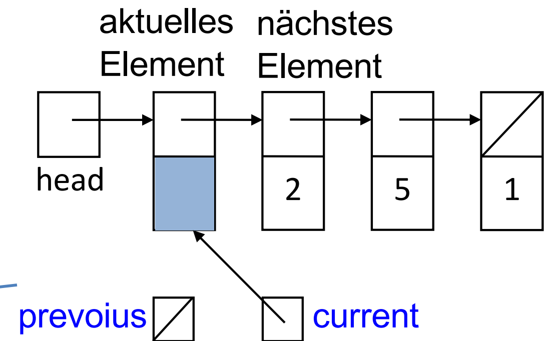
Es wird geprüft, ob der Container geändert wurde, und gegebenenfalls eine Eception ausgelöst (fail-fast iterator)



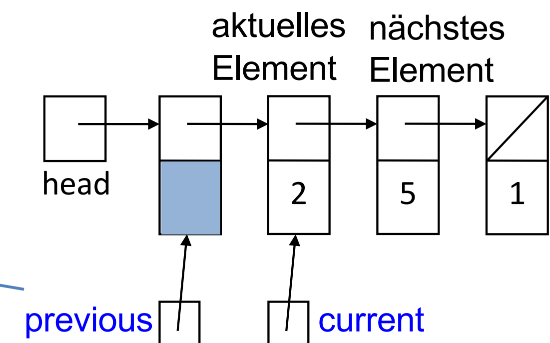
remove wird hier nicht unterstützt.
Kann auch weggelassen werden, da remove in Java 8 als default-Methode vorhanden ist.

LinkedList mit Iteratoren mit remove (1)

```
public class LinkedList<T> implements List<T> {
    // ...

    public Iterator<T> iterator() {
        return new LinkedListIterator();
    }

    private class LinkedListIterator implements Iterator<T> {
        private Node<T> current = head;
        private Node<T> previous = null;
        private boolean okToRemove = false;
        private int expectedMod = modCount;

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

        public T next() {
            if (expectedMod != modCount)
                throw new ConcurrentModificationException();
            if (!hasNext())
                throw new NoSuchElementException();
            previous = current;
            current = current.next;
            okToRemove = true;
            return current.data;
        }
        // ...
    }
}
```

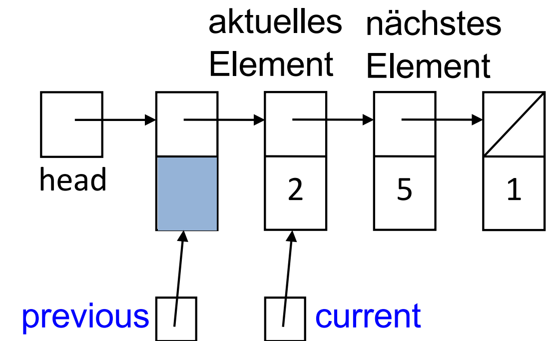


Zuätzlicher Zeiger previous, der current nach jedem next-Aufruf ein Knoten hinterherhinkt



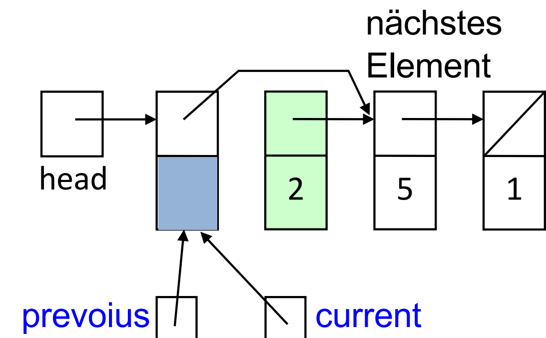
LinkedList mit Iteratoren mit remove (2)

```
public class LinkedList<T> implements List<T> {
    // ...

    private class LinkedListIterator implements Iterator<T> {
        private Node<T> current = head;
        private Node<T> previous = null;
        private boolean okToRemove = false;
        private int expectedMod = modCount;

        // ...

        public void remove() {
            if (expectedMod != modCount)
                throw new ConcurrentModificationException();
            if (!okToRemove)
                throw new IllegalStateException();
            previous.next = current.next;
            current = previous;
            okToRemove = false;
            size--;
        }
    }
}
```



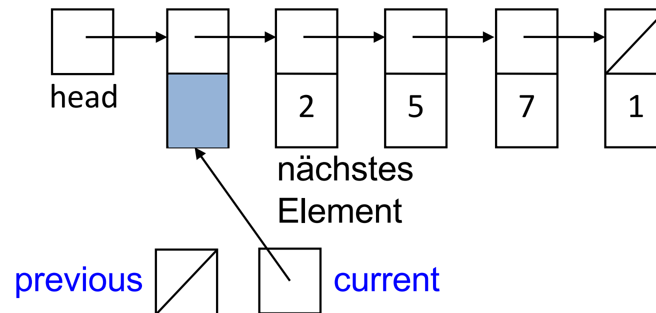
Gelöscht werden darf nur, falls zuvor erfolgreich next aufgerufen wurde.

Gelöscht wird das aktuelle Element d.h. das zuletzt mit next zurückgelieferte Element

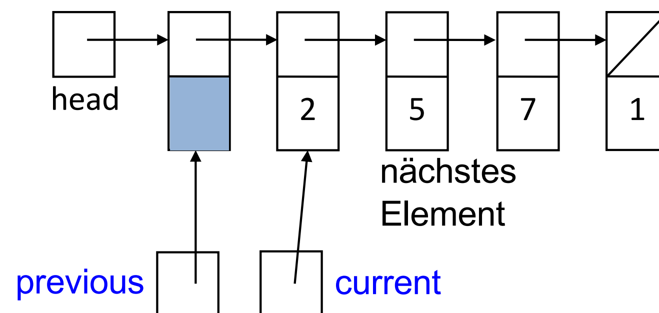


Beispiel: LinkedList mit Iteratoren mit remove (1)

```
List<Integer> list = new LinkedList<>();
list.add(2); list.add(5); list.add(7); list.add(1);
Iterator<Integer> it = list.iterator();
```

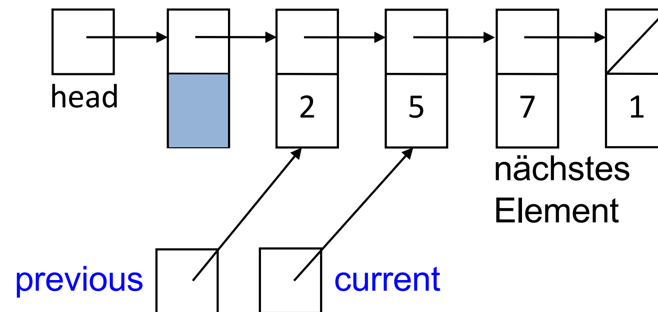



```
int x = it.next(); // x = 2
```

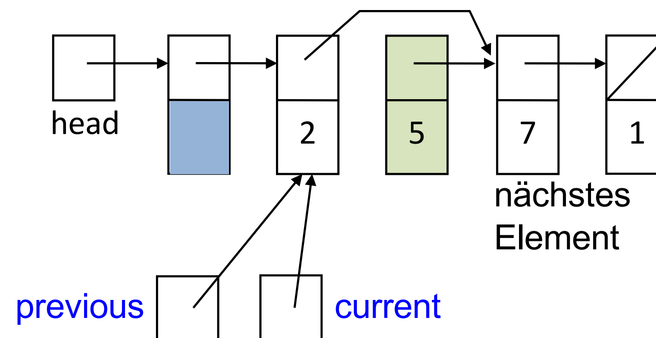


Beispiel: LinkedList mit Iteratoren mit remove (2)

```
int x = it.next(); // x = 5
```




```
it.remove();
```

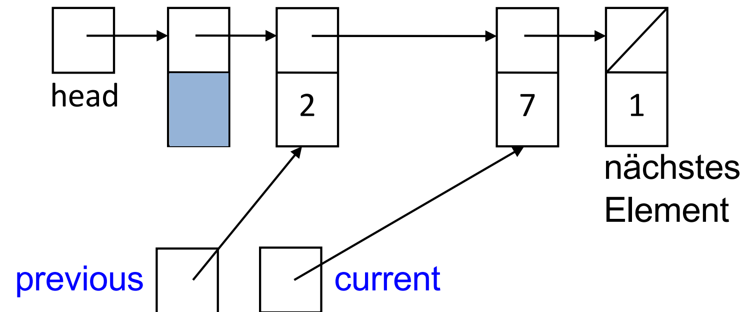


Knoten 5 kann nun vom Garbage Collector abgeräumt werden.

```
it.remove(); // jetzt nicht erlaubt!
```

Beispiel: LinkedList mit Iteratoren mit remove (3)

```
int x = it.next(); // x = 7
```



Anwendung: LinkedList mit Iterator-remove

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

    List<Integer> list = new LinkedList<>();
    for (int i = 0; i < 50; i++)
        list.add(i);

    System.out.println(l);

    // Alle geraden Zahlen loeschen:
    Iterator<Integer> it = list.iterator();
    while(it.hasNext()) {
        if (it.next() % 2 == 0)
            it.remove();
    }

    System.out.println(l);
}
```

Einschub: Innere Klassen

```
class EnclosingClass {
      int outerData = ...;

      class InnerClass{
            ... Zugriff auf outerData ...
      }

      ... new InnerClass();
      ... new InnerClass();
}
```

InnerClass ist innere (d.h. nicht-statisch geschachtelte) Klasse von EnclosingClass.

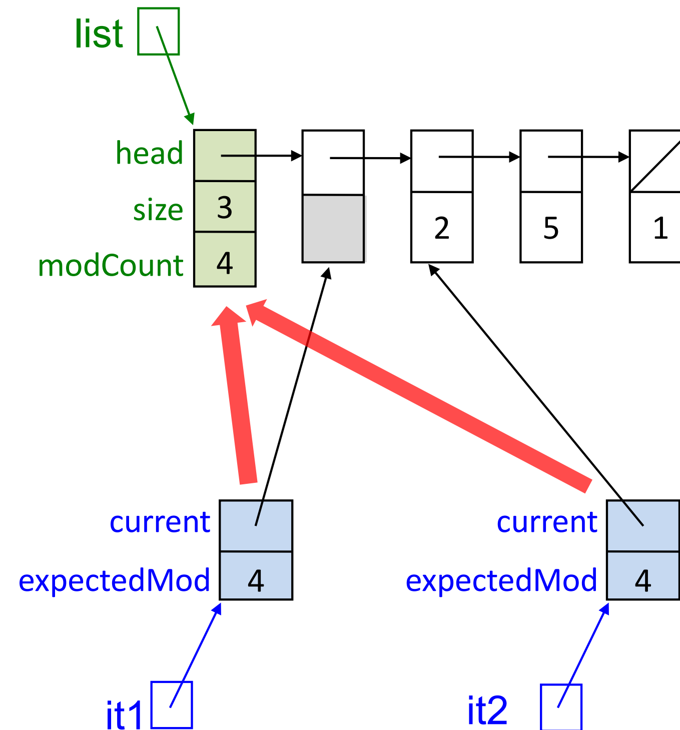
Beide InnerClass-Objekte haben Zugriff auf outerData des EnclosingClass-Objekts.

- Kopplung auf Instanzebene.
- Wird in einem EnclosingClass-Objekt ein InnerClass-Objekt erzeugt, dann ist dieses InnerClass-Objekt mit dem EnclosingClass-Objekt gekoppelt und kann auf dessen Daten zugreifen.

Illustrierung: Iterator-Klasse als innere Klasse

```
public class LinkedList<T> implements List<T> {
    private Node<T> head;
    private int size;
    private int modCount = 0;
    // ...

    public Iterator<T> iterator() {
        return new LinkedListIterator();
    }

    private class LinkedListIterator
    implements Iterator<T> {
        private Node<T> current = head;
        private int expectedMod = modCount;
        // ...
    }
}
```

```
List<Integer> list = new LinkedList<>();
list.add(2); list.add(5); list.add(1);

Iterator<Integer> it1 = list.iterator();

Iterator<Integer> it2 = list.iterator();
it2.next();
```



LinkedListIterator ist innere Klasse der generischen Klasse LinkedList<T>.

Damit haben die LinkedListIterator-Objekte it1 und it2 Zugriff auf die Daten des LinkedList-Objekts list.