Lernzettel

Iteratoren und Iterable in Java

Universität:	Technische Universität Berlin
Kurs/Modul:	Einführung in die Informatik - Vertiefung
Erstellungsdatum:	September 20, 2025

Lernzettel: Iteratoren und Iterable in Java

(1) Definition.
In Java sind zentrale Konzepte für das durchlaufen von Sammlungen das Interface `Iterable<T>` und das Interface `Iterator<T>`.

- `Iterable<T>` definiert die Methode `Iterator<T> iterator();`
- `Iterator<T>` bietet nacheinander Zugriff auf Elemente über: `boolean hasNext();`, `T next();` und optional `void remove();`

(2) Funktionsweise.
Ein Iterator hält typischerweise einen internen Zustand, der den Fortschritt beim Durchlaufen der Elemente beschreibt. Wichtige Methoden:

- `boolean hasNext();` prüft, ob noch ein Element vorhanden ist.
- `T next();` gibt das nächste Element zurück und verschiebt den Zustand.
- `void remove();` entfernt das zuletzt gelesene Element; oft nicht unterstützt und dann `UnsupportedOperationException`.

Zusätzliche Hinweise:

- Bei Aufruf von `next()`, solange kein weiteres Element existiert, wird in der Regel eine `NoSuchElementException` geworfen.
- Wenn das zugrunde liegende Konstrukt während der Iteration modifiziert wird, kann eine `ConcurrentModificationException` auftreten (Fail-fast-Verhalten).

(3) Enhanced for-Schleife (for-each).
In Java kann man Sammlungen bequem mit der erweiterten for-Schleife durchlaufen:

```
for ( T e :  iterable ) { ...}
```

Dabei wird intern `iterator()` auf dem Objekt aufgerufen und `hasNext()` sowie `next()` verwendet.

(4) Implementierung eines eigenen Iterable.
Man implementiert entweder `Iterable<T>` oder verwendet vorhandene Sammlungen. Nachfolgend ein kompaktes Beispiel, das eine einfache Reihe von Zahlen als Iterable bereitstellt.

```
public class Range implements Iterable<Integer> {
  private final int start, end;
  public Range(int start, int end) { this.start = start; this.end = end; }

  public Iterator<Integer> iterator() {
    return new Iterator<Integer>() {
      private int current = start;
```

```
      public boolean hasNext() { return current <= end; }

      public Integer next() {
        if (!hasNext()) throw new java.util.NoSuchElementException();
        return current++;
      }

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



Range r = new Range(1, 5);
for (int x : r) {
  System.out.print(x + " ");
}
// Ausgabe: 1 2 3 4 5
```

(5) Praxishinweise zur Nutzung.

- Verwende Generics `Iterable<T>` und `Iterator<T>` für Typsicherheit.
- Nutze die for-each-Schleife, wenn du lediglich lesen willst.
- Falls du während der Iteration Elemente aus der zugrunde liegenden Sammlung entfernen musst, nutze `Iterator.remove()` oder sehe dir eine sicherere Alternative (z. B. Sammel-Iteratoren wie `ListIterator` an, sofern verfügbar).

(6) Beispiel mit vorhandenen Sammlungen.

```
import java.util.*;

public class Demo {
  public static void main(String[] args) {
    List<String> names = Arrays.asList("Alice", "Bob", "Carol");
    for (String name : names) {
      System.out.println(name);
    }
  }
}
```

(7) Übungsaufgaben.

- Implementiere eine eigene Klasse `Range` (wie oben) und nutze sie in einer for-each-Schleife.
- Schreibe eine MutableIterable-Klasse, bei der `remove()` tatsächlich das zuletzt gelesene Element aus einer zugrunde liegenden `List` entfernt.