universität innsbruck

Iteratoren

Programmiermethodik

Lukas Kaltenbrunner, Simon Priller
Universität Innsbruck

Iteratoren allgemein

- Iterator ist ein Verhaltensmuster.
- Motivation:
  - Eine Sammlung von Elementen soll eine Möglichkeit für die Traversierung ihrer Elemente anbieten.
  - Die Traversierung sollte ohne Kenntnis der internen Struktur möglich sein.
  - Das Traversieren sollte auf unterschiedliche Art und Weise möglich sein.
- Lösung:
  - Schnittstelle, welche den Zugriff und das Traversieren definiert.
  - Spezielles Objekt (Iterator), welches diese Schnittstelle implementiert und den konkreten Zugriff sowie die Traversierung umsetzt.
- Iterator:
  - Mit Iteratoren kann eine Sammlung von Elementen traversiert werden, ohne die darunterliegende Struktur offen zu legen.
  - Es können gleichzeitig mehrere Traversierungen stattfinden.
  - Unterschiedliche Traversierungsvarianten können unterstützt werden.

Interface-Iterator

- In Java gibt es das generische Interface `Iterator<T>`.
- Iteratoren haben den gleichen Elementtyp wie die zugrundeliegende Sammlung von Elementen.
- Das Interface deklariert die folgenden Methoden:

  `hasNext()`
  - Testet, ob es weitere Elemente gibt (`true`) oder nicht (`false`).

  `next()`
  - Liefert das nächste Element und rückt den Iterator gleichzeitig um ein Element weiter, d.h. aufeinanderfolgende Aufrufe von `next` liefern immer neue Elemente.
  - Wirft eine `NoSuchElementException`, wenn keine weiteren Elemente vorhanden sind.

  `remove()`
  - Entfernt das zuletzt zurückgegebene Elemente (optional).

  `forEachRemaining()`
  - Wendet eine Aktion auf alle restlichen Elemente an.
- Ein Iterator ist verbraucht, wenn er am Ende angekommen ist.
- Für einen neuen Durchlauf muss ein neuer Iterator erzeugt werden.

Beispiel

```
public final class PositiveIntegerIterator implements Iterator<Integer> {
    private int current = 0;
    private final int n;

    public PositiveIntegerIterator(int n) {
        if (n < 0) {
            throw new IllegalArgumentException();
        }
        this.n = n;
    }

    @Override
    public boolean hasNext() {
        return current < n;
    }

    @Override
    public Integer next() {
        if (!hasNext()) {
            throw new NoSuchElementException();
        }
        return ++current;
    }
}
```

src/at/ac/uibk/pm/collection/PositiveIntegerIterator.java

Interface-Iterable

- Das Interface `Iterable<T>` schreibt die Methode `iterator()`, welche einen Iterator über Elemente vom Typ `T` zurückgibt, vor.
- Wird das Interface `Iterable` implementiert, kann die erweiterte `for`-Schleife angewendet werden.
- Das Interface `Collection` erweitert das Interface `Iterable`.

```
public class ListInventory implements Inventory {
    private List<Product> inventoryList = new ArrayList<>();
    …

    public void printInventory() {
        for (Product product : inventoryList) {
            System.out.println(product);
        }
    }
    …
}
```

Iteratoren bei Collections

Iterator auf List

```
public class ListInventory implements Inventory {
    private List<Product> inventoryList = new ArrayList<>();
    …

    public void printInventory() {
        Iterator<Product> iterator = inventoryList.iterator();
        while (iterator.hasNext()) {
            System.out.println(iterator.next());
        }
    }
    …
}
```

src/at/ac/uibk/pm/collection/productinventory/ListInventory.java

Iterator auf Map

```
public class MapInventory implements Inventory {
    private Map<Integer, Product> inventoryMap = new HashMap<>();
    …

    public void printInventory() {
        Iterator<Product> iterator = inventoryMap.values().iterator();
        while (iterator.hasNext()) {
            System.out.println(iterator.next());
        }
    }
    …
}
```

src/at/ac/uibk/pm/collection/productinventory/MapInventory.java

Interface-ListIterator

- Für Listen gibt es noch eigene Iteratoren.
  - Diese implementieren das Interface `ListIterator<T>`.
  - Das `List`-Interface deklariert die Methode `listIterator`, welche einen `ListIterator` zurückgibt.
- Das Interface `ListIterator` erweitert das Interface `Iterator`.
- Ein Listen-Iterator kann sich vorwärts und rückwärts bewegen.
- Auszug zusätzlich angebotener Methoden:
  - `previous` und `hasPrevious`, die sich auf das vorhergehende Element beziehen.
  - `previousIndex` und `nextIndex` liefern die Indexwerte des vorhergehenden und nächsten Elements.
- Ein Listen-Iterator ist nicht verbraucht, wenn er am Ende der Liste angekommen ist.

Modifikation und Iteratoren (1)

- Wird eine Collection modifiziert (z.B. neues Element einfügen), dann werden alle Iteratoren ungültig.
- Beim nächsten Zugriffsversuch auf einen Iterator nach einer Änderung wirft dieser eine `ConcurrentModificationException`.
- Wird als fail-fast bezeichnet.
  - Iterator gerät nicht irgendwann in einen inkonsistenten Zustand.
  - Iterator wird sofort unbrauchbar gemacht.
- Iteratoren reagieren nur auf strukturelle Änderungen, d.h. Ersetzen eines Elements verändert die Struktur nicht und lässt Iteratoren intakt.

Modifikation und Iteratoren (2)

- Iteratoren bieten selbst Änderungsoperationen an, bei denen der betreffende Iterator funktionsfähig bleibt.
- Alle Iteratoren definieren die Methode `remove`, die das zuletzt überquerte Element aus der Collection entfernt.
- Aber
  - Iterator muss sich zuerst bewegen, dann erst kann `remove` aufgerufen werden (löscht das letzte Element, das zurückgegeben wurde).
  - Nach einem Aufruf muss erst wieder ein Element überquert werden.
  - Bei Listen-Iteratoren ist das zuletzt überquerte Element abhängig von der Laufrichtung (kann vom Listenanfang aus gesehen vor oder hinter dem Iterator sein).

Modifikation und Iteratoren (3)

- Bei Listen-Iteratoren gibt es neben `remove` noch zusätzliche Methoden.
  - `add()` – fügt das Element in der Lücke ein, in der der Iterator steht.
  - `set()` – ersetzt das zuletzt überquerte Element durch das übergebene Element.
- Bei Änderungen über einen Iterator bleibt nur dieser eine Iterator intakt.
- Andere Iteratoren, die in der gleichen Collection unterwegs sind, werden ungültig.

Modifikation eines Iterators

```
public void removeUnavailableProducts() {
    for (Product product : inventoryList) {
        if (!product.isInStock()) {
            inventoryList.remove(product);
        }
    }
}
```

Ausgabe

```
Exception in thread "main" java.util.ConcurrentModificationException
    at java.base/java.util.ArrayList$Itr.checkForComodification(ArrayList.java:1043)
    at java.base/java.util.ArrayList$Itr.next(ArrayList.java:997)
    at ListInventory.removeUnavailableProducts(ListInventory.java:16)
    at ProductInventoryApplication.main(ProductInventoryApplication.java:33)
```

Avoid Modification During Iteration

```
public void removeUnavailableProducts() {
    Iterator<Product> productIterator = inventoryList.iterator();
    while (productIterator.hasNext()) {
        if (!productIterator.next().isInStock()) {
            productIterator.remove();
        }
    }
}
```

- Vorher: Möglichkeit einer `ConcurrentModificationException`
- Nachher: Möglichkeit ausgeschlossen

src/at/ac/uibk/pm/collection/productinventory/ListInventory.java

Bool'scher Returnwert

```
public boolean isInStock() {
    boolean result;
    if (stock <= 0) {
        result = false;
    } else {
        result = true;
    }
    return result;
}
```

Return Boolean Expressions Directly

```java
public boolean isInStock() {
    return stock > 0;
}
```

- Vorher: Code unnötig kompliziert
- Nachher: kurzer, kompakter und trotzdem gut lesbarer Code

src/at/ac/uibk/pm/collection/productinventory/Product.java

Bool'scher Vergleich

```
public void removeUnavailableProducts() {
    Iterator<Product> productIterator = inventoryMap.values().iterator();
    while (productIterator.hasNext() == true) {
        Product currentProduct = productIterator.next();
        if (currentProduct.isInStock() == false) {
            productIterator.remove();
        }
    }
}
```

Avoid Unnecessary Comparisons

```
public void removeUnavailableProducts() {
    Iterator<Product> productIterator = inventoryMap.values().iterator();
    while (productIterator.hasNext()) {
        Product currentProduct = productIterator.next();
        if (!currentProduct.isInStock()) {
            productIterator.remove();
        }
    }
}
```

- Vorher:
  - Code unnötig kompliziert
  - Unnötige Statements (Einführen neuer boolean Literale)
- Nachher:
  - Kurzer, kompakter und trotzdem gut lesbarer Code
  - Arbeitet „nativ“ mit den vorhandenen booleans

Quellen

- James Gosling, Bill Joy, Guy Steele, Gilad Bracha, Alex Buckley, Daniel Smith, Gavin Bierman: **The Java® Language Specification (***Java SE 17 Edition***)**, Oracle, 2021
- Michael Inden: **Der Weg zum Java-Profi: Konzepte und Techniken für die professionelle Java-Entwicklung**, dpunkt.verlag, 5. Auflage, 2021
- Christian Ullenboom: **Java ist auch eine Insel: Einführung, Ausbildung, Praxis**, Rheinwerk Verlag, 16. Auflage, 2022 (Java 17)
- Simon Harrer, Jörg Lenhard, Linus Dietz: **Java by Comparison: Become a Java Craftsman in 70 Examples**, The Pragmatic Programmers, LLC, 2018