

# Datenstrukturen und die **Collection** -Bibliothek *Objektorientierung mit Java*

Dominikus Herzberg, THM

Version 1.0, 21.4.2021: PiS/VW3

# Lerninhalte

Was Du über Datenstrukturen wissen musst .....	1
Was ist was und wofür? Array, List, Stapel, Set, Queue, Deque, Map .....	1
Komplexitätsbetrachtungen: Das große "O" in Raum und Zeit ( <i>Big O notation</i> ) .....	2
Veränderliche und unveränderliche Datenstrukturen: mutabel vs. immutabel .....	3
Die <b>Collection</b> -Bibliothek in Java .....	4
Das Entscheidende: <b>Collection</b> , <b>List</b> , <b>Set</b> und <b>Map</b> .....	4
Was ist eine Collection? .....	4
Beispiele für die immutable Varianten: <b>List.of()</b> , <b>Set.of()</b> , <b>Map.of()</b> .....	5
Beispiele für <b>List</b> , <b>Set</b> und <b>Map</b> .....	5
Iteration mit foreach-Variante von <b>for</b> oder Streaming .....	6
Weiterführendes und Vertiefendes .....	7

# Was Du über Datenstrukturen wissen musst

- Datenstrukturen nutzen die Möglichkeiten zusammengesetzter Datentypen, um Strukturen zur Verwaltung von Datenelementen aufzusetzen; in Java werden zusammengesetzte (d.h. nicht elementare) Datentypen mit Referenztypen aufgesetzt; in funktionalen Sprachen kommt stattdessen in aller Regel der einfachste zusammengesetzte Datentyp, das Paar (*Pair* oder 2-Tupel), zum Einsatz
- Unterschiedliche Datenstrukturen sind für verschiedene Zwecke entworfen worden, was in unterschiedlichen Organisationsformen zum Ausdruck kommt
- Der Zweck und die damit einhergehende Organisationsform verändern den Speicherbedarf für die Verwaltungsstruktur und die Zeiten, um auf Elemente zuzugreifen, sie hinzuzufügen oder sie aus der Struktur zu entfernen
- Viele Organisationsstrukturen haben Bezeichnungen erhalten, die bildhaft an die zugrunde liegende Organisationsform erinnern sollen

## Was ist was und wofür? Array, List, Stapel, Set, Queue, Deque, Map

Details auch der Implementierung lernen Sie in Algorithmen und Datenstrukturen

- Das Array ist eine lineare und indizierte Struktur fixer Länge; der Index als unmittelbarer Zugriff auf eine Position in der Folge; sehr speichereffizient
- Die Liste als lineare Struktur, die dynamisch erweitert oder verkleinert werden kann; gibt es in zwei "Geschmacksrichtungen": angepasst für indizierten Zugriff (*random access*) als `ArrayList` oder angepasst für Hinzufügen/Entfernen von Elementen als `LinkedList`
- Der Stapel (`Stack`) als eine lineare Struktur, die dynamisch nur einseitig wachsen und schrumpfen kann; angepasst fürs Einfügen/Entfernen am "oberen" Ende des Stapels (LIFO-Prinzip: *last in, first out*)
- Der Set (zu deutsch "Menge") ist eine Kollektion, die keine gleichen Elemente doppelt enthält
- Eine Queue (Puffer oder auch Warteschlange) verwaltet eingehende Elemente in einer linearen Struktur für den späteren Zugriff, meist nach dem FIFO-Prinzip (*first in, first out*); es gibt z.B. die Prioritätsqueue, bei der Elemente je nach Priorität beim Zugriff bevorzugt werden; wird z.B. verwendet zum nicht-blockierenden Datenaustausch zwischen asynchronen Methoden
- Ein Deque (ausgesprochen als "Deck") ist eine *double ended queue*, die den Zugriff von beiden Seiten einer Queue erlaubt
- Eine Map bildet sogenannte Schlüsselwerte auf damit assoziierte Werte ab; in anderen Programmiersprachen auch als "Wörterbuch" (*dictionary*) oder assoziatives Array bezeichnet. Die Map ist eine wichtige und viel eingesetzte Datenstruktur: In JavaScript ist jedes Objekt auch eine Map. Die Objektorientierung in Python basiert auf Maps (dort heißt es Dictionaries)

# Komplexitätsbetrachtungen: Das große "O" in Raum und Zeit (*Big O notation*)

Performanz-Übersicht übernommen aus [Quelle](#) (Abruf, 20.04.2021)

List	Add	Remove	Get	Contains	Next	Data Structure
ArrayList	$O(1)$	$O(n)$	$O(1)$	$O(n)$	$O(1)$	Array
LinkedList	$O(1)$	$O(1)$	$O(n)$	$O(n)$	$O(1)$	Linked List
CopyOnWriteArrayList	$O(n)$	$O(n)$	$O(1)$	$O(n)$	$O(1)$	Array

Set	Add	Remove	Contains	Next	Size	Data Structure
HashSet	$O(1)$	$O(1)$	$O(1)$	$O(h/n)$	$O(1)$	Hash Table
LinkedHashSet	$O(1)$	$O(1)$	$O(1)$	$O(1)$	$O(1)$	Hash Table + Linked List
EnumSet	$O(1)$	$O(1)$	$O(1)$	$O(1)$	$O(1)$	Bit Vector
TreeSet	$O(\log n)$	$O(\log n)$	$O(\log n)$	$O(\log n)$	$O(1)$	Red-black tree
CopyOnWriteArraySet	$O(n)$	$O(n)$	$O(n)$	$O(1)$	$O(1)$	Array
ConcurrentSkipListSet	$O(\log n)$	$O(\log n)$	$O(\log n)$	$O(1)$	$O(n)$	Skip List

Queue	Offer	Peak	Poll	Remove	Size	Data Structure
PriorityQueue	$O(\log n)$	$O(1)$	$O(\log n)$	$O(n)$	$O(1)$	Priority Heap
LinkedList	$O(1)$	$O(1)$	$O(1)$	$O(1)$	$O(1)$	Array
ArrayDeque	$O(1)$	$O(1)$	$O(1)$	$O(n)$	$O(1)$	Linked List
ConcurrentLinkedQueue	$O(1)$	$O(1)$	$O(1)$	$O(n)$	$O(n)$	Linked List
ArrayBlockingQueue	$O(1)$	$O(1)$	$O(1)$	$O(n)$	$O(1)$	Array
PriorityBlockingQueue	$O(\log n)$	$O(1)$	$O(\log n)$	$O(n)$	$O(1)$	Priority Heap
SynchronousQueue	$O(1)$	$O(1)$	$O(1)$	$O(n)$	$O(1)$	None!
DelayQueue	$O(\log n)$	$O(1)$	$O(\log n)$	$O(n)$	$O(1)$	Priority Heap
LinkedBlockingQueue	$O(1)$	$O(1)$	$O(1)$	$O(n)$	$O(1)$	Linked List

Map	Get	ContainsKey	Next	Data Structure
HashMap	$O(1)$	$O(1)$	$O(h / n)$	Hash Table
LinkedHashMap	$O(1)$	$O(1)$	$O(1)$	Hash Table + Linked List
IdentityHashMap	$O(1)$	$O(1)$	$O(h / n)$	Array
WeakHashMap	$O(1)$	$O(1)$	$O(h / n)$	Hash Table
EnumMap	$O(1)$	$O(1)$	$O(1)$	Array
TreeMap	$O(\log n)$	$O(\log n)$	$O(\log n)$	Red-black tree
ConcurrentHashMap	$O(1)$	$O(1)$	$O(h / n)$	Hash Tables
ConcurrentSkipListMap	$O(\log n)$	$O(\log n)$	$O(1)$	Skip List

# Veränderliche und unveränderliche Datenstrukturen: mutabel vs. immutabel

- Aus Java kennen Sie nur veränderliche Datenstrukturen
- Die Referenz auf die Datenstruktur bleibt gleich, beispiel Array
- Unveränderliche Datenstrukturen erlauben auch das Hinzufügen, Entfernen und Modifizieren von Datenelementen, allerdings in einem anderen Sinne: diese Operationen liefern eine neue Referenz auf eine Datenstruktur, die alles bisherige enthält *und* das ergänzte Element enthält bzw. das entfernte Element nicht mehr enthält bzw. anstelle des unmodifizierten nun ein modifiziertes Element enthält; damit bestehen die vorige und die neue Situation unabhängig voneinander
- Der Unterschied deutet sich im Interface an: ein **add** für das Hinzufügen eines Elements hat z.B. bei einer mutablen Liste den Rückgabebetyp **void**, bei einer immutablen Liste den Typ **List**
- Da sich Daten nicht ändern, sind immutable Datenstrukturen automatisch *thread safe*



Die immutablen Datentypen, die das **Collection**-Framework anbietet, unterstützen keine Operationen zum Hinzufügen, Entfernen oder Modifizieren von Elementen; nach ihrer Erstellung sind sie ausschließlich auf lesenden Zugriff ausgelegt

# Die Collection-Bibliothek in Java

- Die **Collection**-Bibliothek hat die wichtigsten Datenstrukturen zu bieten, die im Programmieralltag benötigt werden
- Zu diesen Datenstrukturen existieren zahlreiche und teils sehr spezialisierte Implementierungen (z.B. für nebenläufige Zugriffe)
- Nicht alle Datenstrukturen (wie z.B. Maps) sind Kollektionen oder als Kollektionen realisiert

Die Dokumentation zum Collections Framework ist ausgezeichnet:

<https://docs.oracle.com/en/java/javase/16/docs/api/java.base/java/util/doc-files/coll-index.html>  
(Abruf, 21.04.2021)

Von diesem Einstiegspunkt sind sowohl der *Overview* und das *Annotated API Outline* absolut lesenswert. Lassen Sie sich nicht von der scheinbar langweiligen Auflistung abschrecken.

## Das Entscheidende: Collection, List, Set und Map

Die Collection-Bibliothek hat einiges zu bieten. Als Neuling empfehle ich einen Anfang mit den folgenden Interfaces und Klassen; der Rest erschließt sich mit der Zeit.

- Die Interfaces **Collection**<E>, **List**<E>, **Set**<E>, **Map**<K,V>
- **ArrayList**<E> implementiert **List**
- **HashMap**<K,V> implementiert **Map**
- **HashSet**<E> implementiert **Set**
- **Stack**<E> (implementiert kein Interface)
- Die Klasse **Collections** mit ihren statischen Methoden

### Beachte bei der Variablendeklaration: Verhaltenstyp vs. Implementierungstyp

Wenn keine guten Gründe dagegen sprechen: Wähle für den Typ einer Variable den Verhaltenstyp (Interface) und weise ihr den Wert eines Implementierungstyps (Klasse) zu.

Beispiel: **List** ist das Interface, **ArrayList** eine implementierende Klasse

```
List<Integer> numbers = new ArrayList<>();
```

## Was ist eine Collection?

- **Collection** ist das oberste Interface in der Kollektions-Hierarchie

- Zugang über `import java.util.*;`
- Eine Kollektion ist eine Sammlung oder Gruppe von Objekten, die Objekte sind die Elemente der Kollektion
- Die erweiternden Interfaces wie z.B. `Set`, `List`, `Queue`, `Deque` charakterisieren verschiedene Arten von Kollektionen
- Diese Arten von Kollektionen erweitern `Collection`, um Besonderheiten gerecht zu werden

## Beispiele für die immutable Varianten: `List.of()`, `Set.of()`, `Map.of()`

- Mit `List.of(...)`, `Set.of(...)` und `Map.of(...)` immutable Listen, Mengen und Abbildungen erzeugen
- Werden gerne genutzt als Ersatz dafür, dass es in Java keine Literale für Liste, Mengen bzw. Abbildungen gibt
- Dann nützlich, wenn sich Daten nicht mehr ändern werden
- Der Speicherbedarf immutabler und hier nur lesbarer Datenstrukturen ist effizienter als bei mutablen Datenstrukturen

Beispiel für Einsatz von `List.of`

```
List<String> names = List.of("Alice", "Bob", "Carrie", "Don");
assert names.get(0).equals("Alice") && names.get(3).equals("Don");
names.add("Emma"); ①
```

① Wirft eine `UnsupportedOperationException`

## Beispiele für `List`, `Set` und `Map`

Ihre Entwicklungsumgebung hilft bei der raschen Orientierung, welche Methoden gibt

Beispiel für `List`

```
List<Integer> numbers = new ArrayList<>(List.of(4, 1, 9));
numbers.add(0);
numbers.addAll(List.of(5, 4));
assert numbers.get(numbers.size() - 1) == 4; ①
assert !numbers.isEmpty() && numbers.contains(9);
```

① Bei Zahlen nimmt Java Konvertierung vor, kein `.equals()` nötig zum Vergleich

### Beispiel für Set

```
Set<Integer> set = Set.of(3,5,2); ①
assert !set.isEmpty() && set.contains(3) && !set.contains(4);
Set<Integer> set = new HashSet<>(Set.of(3,5,2)); ②
assert !set.isEmpty() && set.contains(3) && !set.contains(4);
```

① In diesem Fall ist `set` immutabel

② Und hier ist `set` mutabel

### Beispiel für Map

```
Map<String, Integer> maxDays = Map.of("Jan", 31, "Feb", 29, "Mar", 31, "Apr", 30); ①
assert maxDays.get("Mar") == 31 && maxDays.get("May") == null;
Map<String, Integer> maxDays = new HashMap<>(Map.of("Jan", 31, "Feb", 29)); ②
assert maxDays.get("Jan") == 31 && maxDays.get("Aug") == null;
```

① In diesem Fall ist `maxDays` immutabel

② Und hier ist `maxDays` mutabel



`Map` kann als Ersatz für `switch/case` oder `if/else` dienen.

## Iteration mit foreach-Variante von `for` oder Streaming

### Beispiel zur Iteration über eine `Collection`

```
int sum = 0, sum2 = 0;
for(Integer number : numbers) sum += number; ①
assert sum == numbers.stream().reduce(0, (x, y) -> x + y); ②
numbers.forEach(n -> sum2 += n); ③
assert sum == sum2;
```

① `Collection` erweitert das Interface `Iterable`, das die Grundlage der `foreach`-Variante der `for`-Schleife ist

② Eine `Collection` bietet die `stream`-Methode an

③ Jedes `Iterable` hat die `forEach`-Methode



# Weiterführendes und Vertiefendes

- Oracle (März 2021). Creating Unmodifiable Lists, Sets, and Maps, <https://docs.oracle.com/en/java/javase/16/core/creating-immutable-lists-sets-and-maps.html>, F35145-01, (Abruf, 19.04.2021)

Java zeichnet sich durch eine ausgezeichnete Dokumentation aus, was auch Design-Entscheidungen einschließt. Wer etwas über das Warum des Collection-APIs erfahren möchte, findet hier eine interessante Quelle:

- Oracle. Java Collections API Design FAQ , <https://docs.oracle.com/en/java/javase/16/docs/api/java.base/java/util/doc-files/coll-designfaq.html> (Abruf, 19.04.2021)

Ebenso ist viel zu lernen aus dem Text von Wadler und seinem Co-Autoren Naftalin, die die Generics für Java entworfen haben mit dem Ziel, davon eine **Collection**-Bibliothek profitieren zu lassen.

- Naftalin, Maurice; Wadler, Philip (2006). Java Generics and Collections. O'Reilly