



Java Intro

Michael Inden

Freiberuflicher Consultant und Trainer

https://github.com/Michaeli71/JAVA_INTRO

Speaker Intro



- Michael Inden, Jahrgang 1971
- Diplom-Informatiker, C.v.O. Uni Oldenburg
- ~8 ¼ Jahre **SSE** bei Heidelberger Druckmaschinen AG in Kiel
- ~6 ¾ Jahre **TPL, SA** bei IVU Traffic Technologies AG in Aachen
- ~4 ¼ Jahre **LSA / Trainer** bei Zühlke Engineering AG in Zürich
- ~3 Jahre **TL / CTO** bei Direct Mail Informatics / ASMIQ in Zürich
- **Freiberuflicher Consultant, Trainer und Konferenz-Speaker**
- Autor und Gutachter beim dpunkt.verlag

E-Mail: michael.inden@hotmail.ch

Blog: <https://jaxenter.de/author/minden>

Kurse: **Bitte spricht mich an!**



https://github.com/Michaeli71/JAVA_INTRO



Agenda



- **PART 1: Schnelleinstieg Java**
 - Erste Schritte in der JShell
 - Schnelleinstieg
 - Variablen
 - Operatoren
 - Fallunterscheidungen
 - Schleifen
 - Methoden
 - Rekursion



- **PART 2: Strings**
 - Gebräuchliche String-Aktionen
 - Suchen und Ersetzen
 - Formatierte Ausgaben
 - Mehrzeilige Strings
 - Einstieg Reguläre Ausdrücke
- **PART 3: Arrays**
 - Gebräuchliche Array-Aktionen
 - Mehrdimensionale Arrays
 - Beispiel: Flood Fill



- **PART 4: Klassen & Objektorientierung**
 - Basics
 - Textuelle Ausgaben
 - Gleichheit == / equals()
 - Klassen ausführbar machen
 - Imports & Packages
 - Information Hiding
 - Vererbung und Overloading und Overriding
 - Die Basisklasse Object
 - Interfaces & Implementierungen



- **PART 5: Collections**
 - Einführung
 - Schnelleinstieg Listen
 - Praxisbeispiel Stack und Queue selbst gebaut
 - Schnelleinstieg Sets
 - Schnelleinstieg Maps
 - Generics
 - Basisinterfaces für Container
 - Iteratoren
 - Sortierung – sort() + Comparator



- **PART 6: Ergänzendes Wissen**
 - Sichtbarkeits- und Gültigkeitsbereiche
 - Primitive Typen und Wrapper
 - Enums
 - ?-Operator
 - switch
 - Besonderheiten in Schleifen break und continue
 - Vererbung und Polymorphie
 - Varianten innerer Klassen
 - Records



- **PART 7: Exception-Handling**
 - Schnelleinstieg
 - Exceptions selbst auslösen
 - Eigene Exception-Typen definieren
 - Propagation von Exceptions
 - Automatic Resource Management
 - Checked / Unchecked Exceptions
- **PART 8: Dateiverarbeitung**
 - Verzeichnisse und Dateien verwalten
 - Daten schreiben / lesen
 - CSV-Dateien einlesen



- **PART 9: Einstieg in Lambdas und Streams**
 - Syntax von Lambdas
 - Lambdas im Einsatz mit `filter()`, `map()` und `reduce()`
 - Lambdas im Einsatz mit `Collectors.groupingBy()`
 - `takeWhile()` / `dropWhile()`
- **PART 10: Datumsverarbeitung**
 - Einführung Datumsverarbeitung
 - Zeitpunkte und die Klasse `LocalDateTime`
 - Datumswerte und die Klasse `LocalDate`
 - Zeit und die Klasse `LocalTime`



PART 5:

Collections



- Immer wieder muss man mehrere Objekte verwalten, dazu haben wir bisher Arrays kennengelernt. Allerdings bieten diese nicht allzu viel Komfort, insbesondere keine automatische Größenanpassung
 - Deswegen gibt es weitere Möglichkeiten, Objekte anderer Klassen zu speichern und zu verwalten.
 - Um Daten in eigenen Applikationen sinnvoll zu speichern und performant darauf zugreifen zu können, muss man geeigneter Datenstrukturen nutzen.
 - In Java werden Listen, Mengen und Schlüssel-Wert-Abbildungen durch sogenannte Containerklassen realisiert.
-



- **Für Sammlungen von Elementen muss sich zwischen Listen und Mengen entscheiden.**
 - Für Daten, die eine Reihenfolge der Speicherung erfordern und auch (mehrfach) gleiche Einträge enthalten dürfen, setzen wir Listen ein.
 - Möchte an doppelte Einträge automatisch verhindern, so stellt ein Set eine geeignete Wahl dar.
 - **Es gibt diverse Anwendungsfälle, in denen man Abbildungen von Objekten auf andere Objekte realisieren muss. Man spricht hier von einem Mapping von Schlüsseln auf Werte. Dazu nutzt man sinnvollerweise Maps.**
-



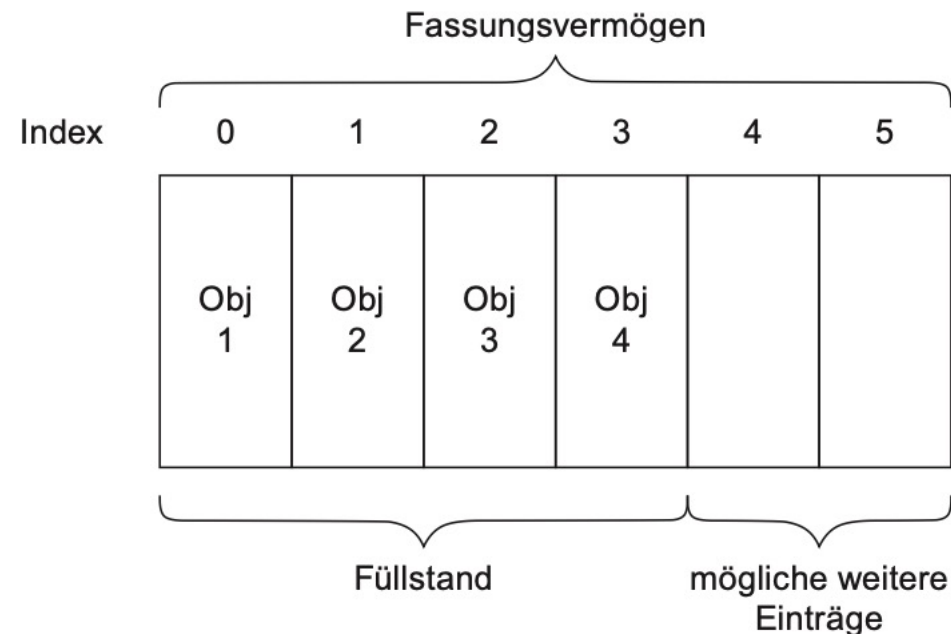
Listen



Listen



- Unter einer Liste versteht man eine über ihre Position geordnete Folge von Elementen – dabei können auch identische Elemente mehrfach vorkommen.
- Die ArrayList kann man sich wie ein größenveränderliches Array vorstellen.
- Sie ermöglicht einen indizierten Zugriff und erlaubt das Hinzufügen und Entfernen von Elementen





- **Erzeugen mit new:**

```
jshell> var names = new ArrayList<String>()  
names ==> []
```

- **Es entsteht ein neuer Container. Durch die Angabe `ArrayList<String>` erstellt Java eine `ArrayList`, die NUR zur Verwaltung von Strings dient.**

- **Erzeugen mit Collection-Factory-Methoden**

```
jshell> var unmodifiableNames = List.of("Jim", "James")  
unmodifiableNames ==> [Jim, James]
```

- **Heterogene Zusammensetzung möglich (wie früher ohne Generics)**

```
jshell> List.of(0, "ABC", 42.195, true)  
$55 ==> [0, ABC, 42.195, true]
```



- **Hinzufügen**

```
jshell> var names = new ArrayList<String>()  
names ==> []
```

```
jshell> names.add("Tim")  
$86 ==> true
```

```
jshell> names.add("Tom")  
$87 ==> true
```

```
jshell> names  
names ==> [Tim, Tom]
```



- **Hinzufügen an Position**

```
jshell> names  
names ==> [Tim, Tom]
```

```
jshell> names.add(0, "Anton")
```

```
jshell> names.add(0, "Andreas")
```

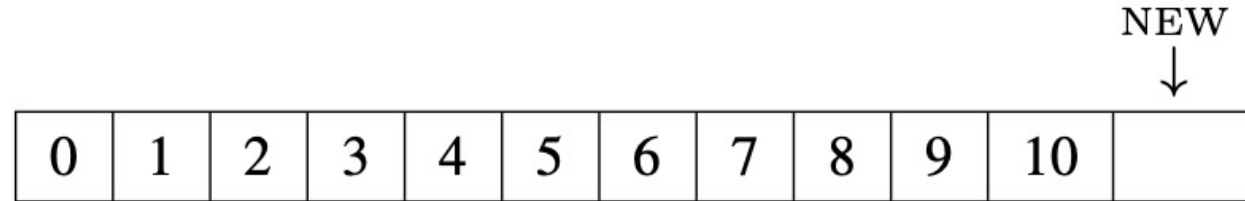
```
jshell> names.add(4, "Last")
```

```
jshell> names  
names ==> [Andreas, Anton, Tim, Tom, Last]
```

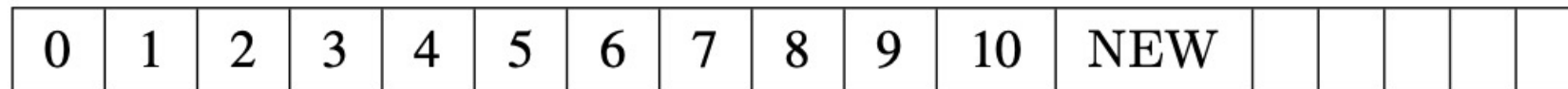
Besonderheit beim Hinzufügen



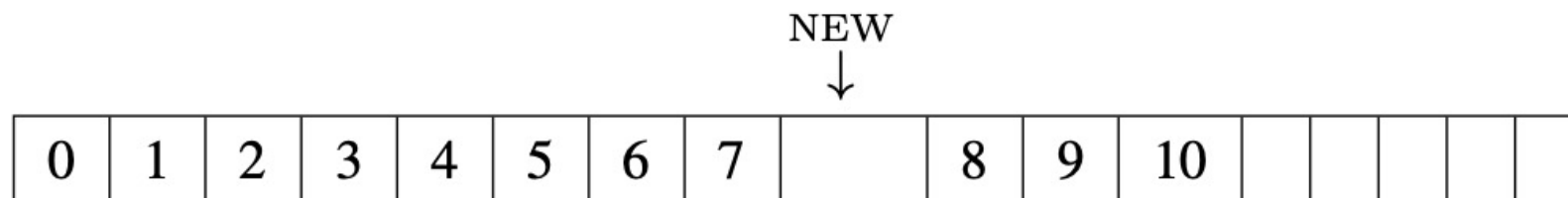
- **Hinzufügen an letzter freier Position:**



Das zugrunde liegende Array ist jetzt komplett belegt und es ist kein Platz für ein weiteres Element vorhanden. Soll erneut ein Element hinzugefügt werden, muss als Folge das Array in seiner Größe angepasst werden:



- **Hinzufügen mittendrin**





- **Auf Elemente zugreifen**

```
jshell> var numbers = List.of(0, 1, 2, 3, 4, 5, 6, 7, 8, 9)
numbers ==> [0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
```

```
jshell> numbers.get(5)
$60 ==> 5
```

```
jshell> numbers.get(0)
$61 ==> 0
```

```
jshell> numbers.get(9)
$62 ==> 9
```

- **Wieviele Elemente?**

```
jshell> numbers.size()
$63 ==> 10
```



- **Bereichsverletzung (auch bei negativem Index)**

```
jshell> var numbers = List.of(0, 1, 2, 3, 4, 5, 6, 7, 8, 9)
numbers ==> [0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
```

```
jshell> numbers.get(42)
| Exception java.lang.ArrayIndexOutOfBoundsException: Index 42 out of bounds for
length 10
|       at ImmutableCollections$ListN.get (ImmutableCollections.java:680)
|       at (#57:1)
```

```
jshell> numbers.get(-42)
| Exception java.lang.ArrayIndexOutOfBoundsException: Index -42 out of bounds
for length 10
|       at ImmutableCollections$ListN.get (ImmutableCollections.java:680)
|       at (#58:1)
```

Listen durchlaufen



- **positionsbasiert**

```
jshell> var names = List.of("Andy", "Mike", "Tim")
names ==> [Andy, Mike, Tim]
```

```
jshell> for (int i = 0; i < names.size(); i++)
...>     System.out.println(names.get(i))
Andy
Mike
Tim
```

- **elementbasiert**

```
jshell> for (String name : names)
...>     System.out.println(name)
Andy
Mike
Tim
```



- **Ändern**

```
jshell> var names = List.of("Andreas", "Anton", "Tim", "Tom", "Last")
names ==> [Andreas, Anton, Tim, Tom, Last]
```

```
jshell> var modifiableNames = new ArrayList<>(names)
modifiableNames ==> [Andreas, Anton, Tim, Tom, Last]
```

```
jshell> modifiableNames.set(1, "Mike")
$98 ==> "Anton"
```

```
jshell> modifiableNames
modifiableNames ==> [Andreas, Mike, Tim, Tom, Last]
```

Listen – Basisfunktionalitäten: Löschen



```
jshell> modifiableNames  
modifiableNames ==> [Andreas, Mike, Tim, Tom, Last]
```

```
jshell> modifiableNames.remove(2)  
$100 ==> "Tim"
```

```
jshell> modifiableNames  
modifiableNames ==> [Andreas, Mike, Tom, Last]
```

```
jshell> modifiableNames.remove("Tom")  
$102 ==> true
```

```
jshell> modifiableNames.remove("Tom")  
$103 ==> false
```

```
jshell> modifiableNames.clear()
```

```
jshell> modifiableNames  
modifiableNames ==> []
```



- **Basisfunktionalitäten in Utility-Klasse Collections**

```
var numbers = new ArrayList<>(List.of(7, 5, 3, 1, 6, 4, 2));
```

| | | |
|--------------------------|---|---|
| numbers.size() | → | 7 |
| Collections.min(numbers) | | 1 |
| Collections.max(numbers) | | 7 |

- **Nettigkeiten**

| | | |
|-------------------------------|---|-----------------------|
| Collections.reverse(numbers); | → | |
| System.out.println(numbers); | | [2, 4, 6, 1, 3, 5, 7] |
| Collections.sort(numbers); | | |
| System.out.println(numbers); | | [1, 2, 3, 4, 5, 6, 7] |



- **Multi-Append**

```
jshell> var cities = new ArrayList<>(List.of("Zürich", "Luzern"))  
cities ==> [Zürich, Luzern]
```

```
jshell> cities.addAll(List.of("Bremen", "Aachen"))  
$107 ==> true
```

```
jshell> cities.addAll(List.of("Bremen", "Aachen", "Kiel"))  
$108 ==> true
```

- **Multi-Remove**

```
jshell> cities.removeAll(List.of("Bremen", "Aachen"))  
$110 ==> true
```

```
jshell> cities  
cities ==> [Zürich, Luzern, Kiel]
```



- **Enthaltensein prüfen (contains())**

```
jshell> var values = List.of(0, "ABC", 42.195, true)
values ==> [0, ABC, 42.195, true]
```

```
jshell> values.contains("ABC")
$113 ==> true
```

```
jshell> values.contains(42.195)
$114 ==> true
```

```
jshell> values.contains(false)
$115 ==> false
```



- **Teilbereich als Liste liefern (subList())**

```
jshell> var numbers = List.of(0,1,2,3,4,5,6,7,8,9)
numbers ==> [0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
```

```
jshell> var first3 = numbers.subList(0, 3)
first3 ==> [0, 1, 2]
```

```
jshell> var last4 = numbers.subList(numbers.size() - 4, numbers.size())
last4 ==> [6, 7, 8, 9]
```



- **Teilbereich als Liste liefern (subList())**

```
jshell> var numbers = List.of(0,1,2,3,4,5,6,7,8,9)
numbers ==> [0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
```

```
jshell> var first3 = numbers.subList(0, 3)
first3 ==> [0, 1, 2]
```

```
jshell> var last4 = numbers.subList(numbers.size() - 4, numbers.size())
last4 ==> [6, 7, 8, 9]
```

```
jshell> last4.addAll(List.of(11, 13, 17))
| Exception java.lang.UnsupportedOperationException
|       at ImmutableCollections.uoe (ImmutableCollections.java:142)
|       at ImmutableCollections$AbstractImmutableCollection.addAll
(ImmutableCollections.java:148)
|       at (#120:1)
```

Listen – Teile extrahieren / Sublisten



- **Teilbereich als Liste liefern (subList()) => ACHTUNG: VIEW**

```
jshell> var numbers = new ArrayList<>(List.of(0,1,2,3,4,5,6,7,8,9))  
numbers ==> [0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
```

```
jshell> var first3 = numbers.subList(0, 3)  
first3 ==> [0, 1, 2]
```

```
jshell> var last4 = numbers.subList(numbers.size() - 4, numbers.size())  
last4 ==> [6, 7, 8, 9]
```

```
jshell> last4.addAll(List.of(11, 13, 17))  
$124 ==> true
```

```
jshell> last4  
last4 ==> [6, 7, 8, 9, 11, 13, 17]
```

```
jshell> numbers  
numbers ==> [0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 11, 13, 17]
```



Stack und Queue im Selbstbau

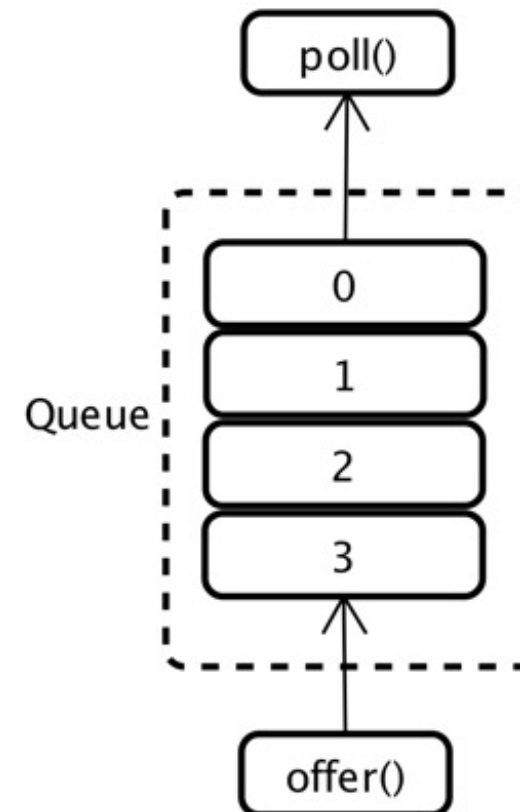
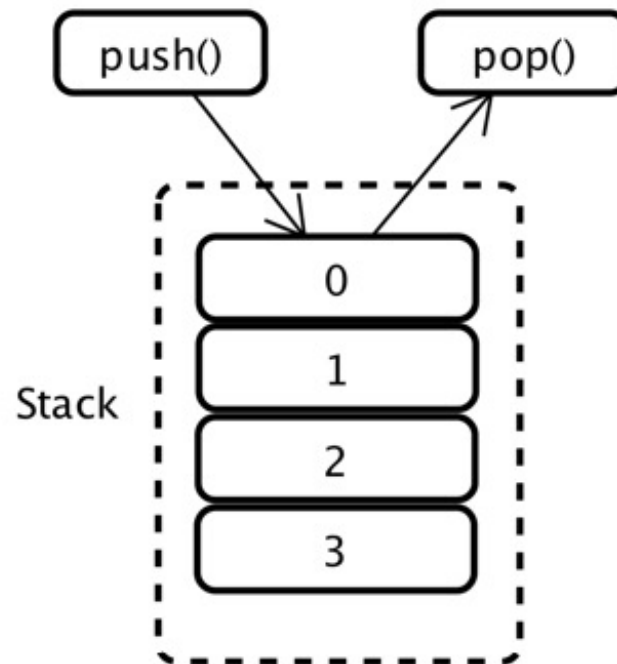


Listen – Definition eigener Datenstrukturen



- **Stack: Schreibtischablage, LIFO**
- **Queue: Schlange an der Kasse, FIFO / FCFS**

Neue Aufträge werden oben abgelegt und der oberste Auftrag wird als Nächstes bearbeitet.



Neue Wartende reihen sich immer hinten ein und jeder kommt der Reihe nach dran.

Praxisbeispiel: Stack selbst realisiert



```
public interface IStack<E>
{
    void push(E elem);
    E pop();
    E peek();
    boolean isEmpty();
}
```

Praxisbeispiel: Stack selbst realisiert



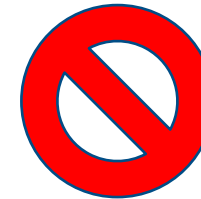
```
public class MyStack<E> implements IStack<E>
{
    private final ArrayList<E> values = new ArrayList<>();

    @Override
    public void push(final E elem) {
        values.add(elem);
    }

    @Override
    public E pop() {
        return values.remove(values.size() - 1);
    }

    @Override
    public E peek() {
        return values.get(values.size() - 1);
    }

    @Override
    public boolean isEmpty() {
        return values.isEmpty();
    }
}
```



Praxisbeispiel: Queue selbst realisiert



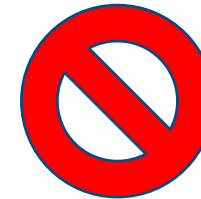
```
public class MyQueue<E>
{
    private final ArrayList<E> values = new ArrayList<>();

    public void enqueue(final E elem) {
        values.add(elem);
    }

    public E dequeue() {
        return values.remove(0);
    }

    public E peek() {
        return values.get(0);
    }

    public boolean isEmpty() {
        return values.isEmpty();
    }
}
```



Praxisbeispiel: Queue selbst realisiert



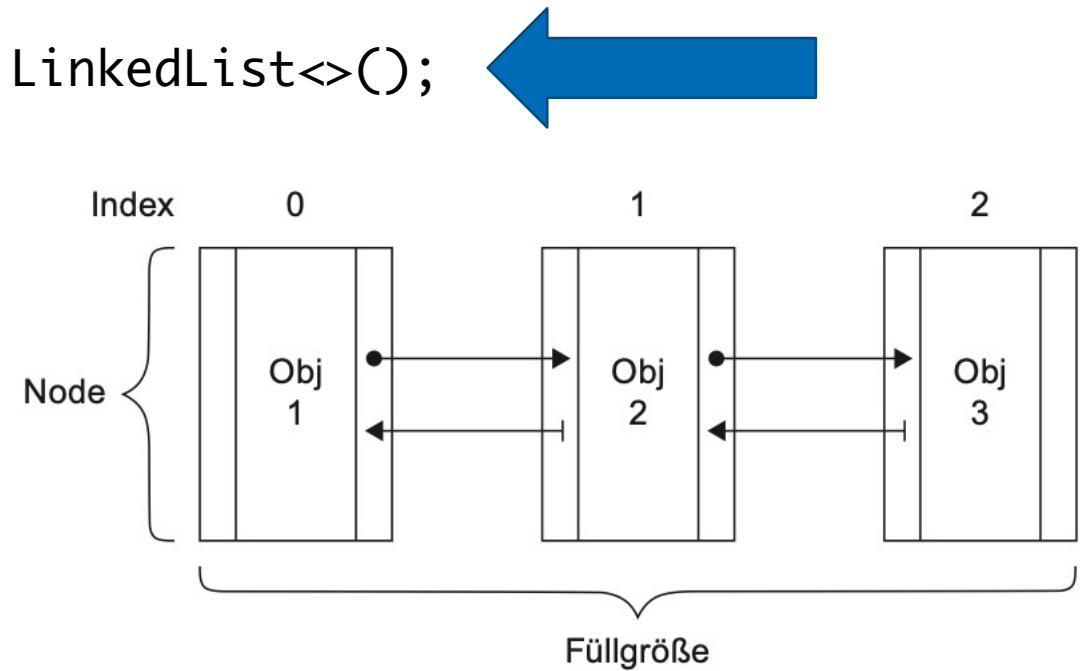
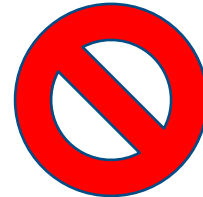
```
public class MyQueue<E>
{
    private final LinkedList<E> values = new LinkedList<>();

    public void enqueue(final E elem) {
        values.add(elem);
    }

    public E dequeue() {
        return values.remove(0);
    }

    public E peek() {
        return values.get(0);
    }

    public boolean isEmpty() {
        return values.isEmpty();
    }
}
```



Praxisbeispiel: Queue selbst realisiert



```
public static void main(final String[] args)
{
    final MyQueue<String> waitingPersons = new MyQueue<>();

    waitingPersons.enqueue("Marcello");
    waitingPersons.enqueue("Michael");
    waitingPersons.enqueue("Karthi");

    while (!waitingPersons.isEmpty())
    {
        if (waitingPersons.peek().equals("Michael"))
        {
            // Am Ende "neu anstellen" und verarbeiten
            waitingPersons.enqueue("Michael again");
            waitingPersons.enqueue("Last Man");
        }
        final String nextPerson = waitingPersons.dequeue();
        System.out.println("Processing " + nextPerson);
    }
}
```

Processing Marcello
Processing Michael
Processing Karthi
Processing Michael again
Processing Last Man



**Was kann man sonst
noch so machen?**



- **Ausflug Fibonacci-Zahlen (rekursive Definition):**

```
jshell> long fib(int n)
...> {
...>     if (n <= 1)
...>         return n;
...>
...>     return fib(n-2) + fib(n-1);
...> }
| created method fib(int)
```

```
jshell> fib(5)
$68 ==> 5
```



- **Fibonacci Berechnungs-Trick:**

```
jshell> var fib_numbers = new ArrayList<>(List.of(0, 1))  
fib_numbers ==> [0, 1]
```

```
jshell> for (int i = 0; i < 10; i++) {  
    ...>     int lastIdx = fib_numbers.size();  
    ...>     fib_numbers.add(fib_numbers.get(lastIdx - 2) +  
    ...>                                     fib_numbers.get(lastIdx - 1));  
    ...> }
```

```
jshell> fib_numbers  
fib_numbers ==> [0, 1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89]
```




Mengen (HashSet)



Sets – Basisfunktionalitäten



- Ein HashSet ist eine Sammlung (Menge) von Elementen.
- Mathematisches Konzept => keine Duplikate
- Somit bilden Sets eine ungeordnete, duplikatfreie Datenstruktur, bieten **aber keinen** indizierten Zugriff.
- Stattdessen einige Mengenoperationen insbesondere die Berechnung von Vereinigungs-, Schnitt-, Differenz- und symmetrischen Differenzmengen:





- **Erzeugen mit new und Collection-Factory-Methoden**

```
jshell> var names = new HashSet<String>()  
names ==> []
```

```
jshell> var unmodifiableNames = Set.of("Jim", "James")  
unmodifiableNames ==> [James, Jim]
```

- **Heterogene Zusammensetzung möglich (wie früher in Java ohne Generics)**

```
jshell> var differentTypes = Set.of(0, "ABC", 42.195, true)  
differentTypes ==> [ABC, 42.195, true, 0]
```



- **Hinzufügen**

```
jshell> names.add("Tim")  
$141 ==> true
```

```
jshell> names.add("Tom")  
$142 ==> true
```

```
jshell> names  
names ==> [Tom, Tim]
```

```
jshell> names.add("Tim")  
$144 ==> false
```

```
jshell> names.addAll(List.of("Mike", "Peter"))  
$146 ==> true
```

```
jshell> names  
names ==> [Mike, Tom, Tim, Peter]
```



- **Löschen**

```
jshell> names.remove("Tim")  
$149 ==> true
```

```
jshell> names  
names ==> [Mike, Tom, Peter]
```

```
jshell> names.removeAll(Set.of("Peter", "Tom"))  
$151 ==> true
```

```
jshell> names  
names ==> [Mike]
```

```
jshell> names.clear()
```

```
jshell> names  
names ==> []
```



- Enthaltensein prüfen (**contains()**)

```
jshell> var differentTypes = Set.of(0, "ABC", 42.195, true)
differentTypes ==> [ABC, 42.195, true, 0]
```

```
jshell> differentTypes.contains("ABC")
$138 ==> true
```

```
jshell> differentTypes.contains("PETER")
$139 ==> false
```



- **Basisfunktionalitäten: Minimum / Maximum / Anzahl an Elementen**

```
jshell> var numbers = Set.of(11, 2, 3, 4, 5, 6, 7, 8, 9)
numbers ==> [5, 6, 7, 8, 9, 11, 2, 3, 4]
```

```
jshell> Collections.min(numbers)
$134 ==> 2
```

```
jshell> Collections.max(numbers)
$135 ==> 11
```

```
jshell> numbers.size()
$136 ==> 9
```



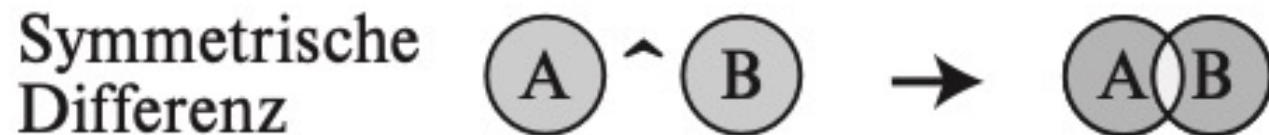
- **elementbasiert durchlaufen**

```
jshell> var differentTypes = Set.of(0, "ABC", 42.195, true)
differentTypes ==> [ABC, 42.195, true, 0]
```

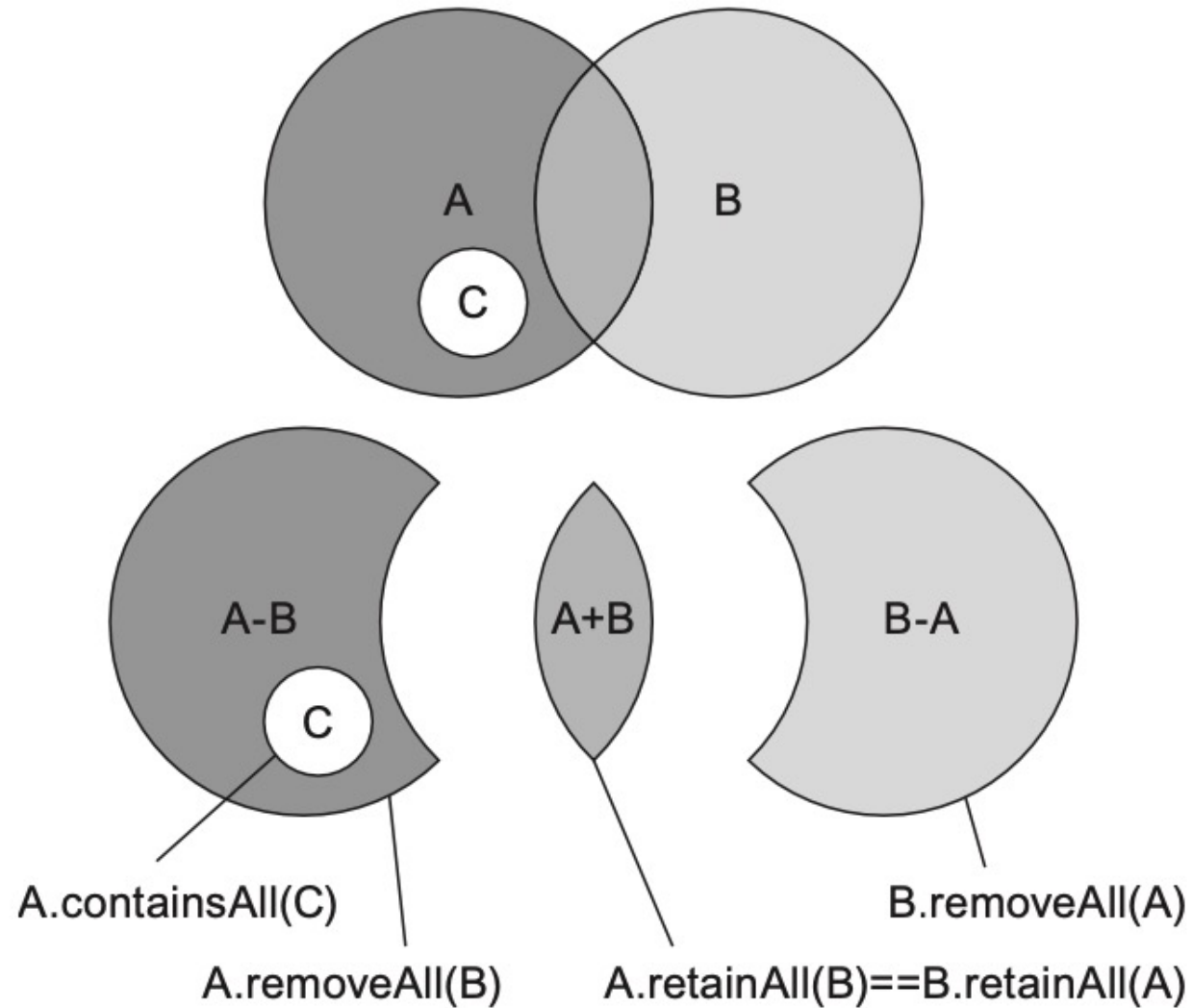
```
jshell> for (var value : differentTypes)
    ...>     System.out.println(value)
ABC
42.195
true
0
```




- **Mengenoperationen: Berechnung von Vereinigungs-, Schnitt-, Differenz- und symmetrischen Differenzmengen:**

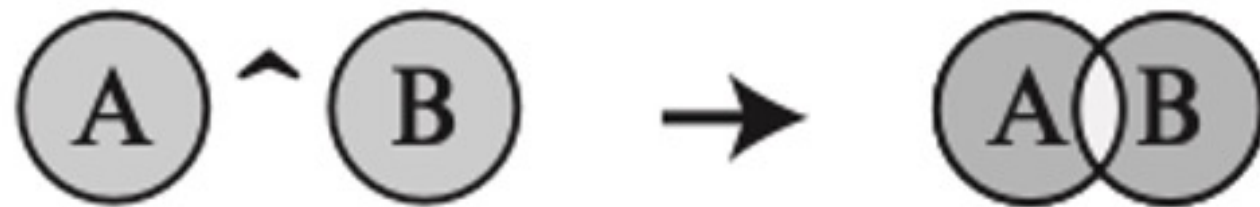


Vordefinierte Mengenoperationen





**Wie kann man die
symmetrische Differenz
ermitteln?**





- **Erst 2 x Differenz berechnen, dann Vereinigung**

```
jshell> HashSet<Integer> symDiff(HashSet<Integer> values1,  
...>                               HashSet<Integer> values2)  
...> {  
...>     // Differenzmengen bilden  
...>     HashSet<Integer> diff1_2 = new HashSet<>(values1);  
...>     diff1_2.removeAll(values2);  
...>     HashSet<Integer> diff2_1 = new HashSet<>(values2);  
...>     diff2_1.removeAll(values1);  
...>  
...>  
...>     // Vereinigung bilden  
...>     HashSet<Integer> result = new HashSet<>();  
...>     result.addAll(diff1_2);  
...>     result.addAll(diff2_1);  
...>     return result;  
...> }  
| created method symDiff(HashSet<Integer>,HashSet<Integer>)
```

Symmetrische Differenz – mal ausprobieren



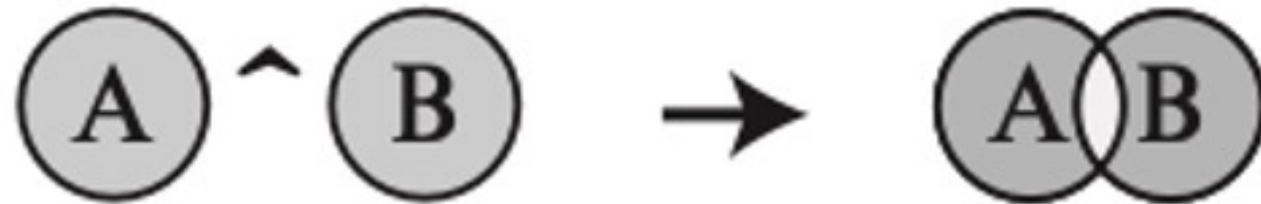
```
jshell> var values1 = new HashSet<>(Set.of(1,2,3))  
values1 ==> [1, 2, 3]
```

```
jshell> var values2 = new HashSet<>(Set.of(3,4,5))  
values2 ==> [3, 4, 5]
```

```
jshell> symDiff(values1, values2)  
$82 ==> [1, 2, 4, 5]
```



Geht das nicht einfacher?





- **Schnittmenge berechnen, Vereinigung bilden und Schnittmenge abziehen**

```
jshell> HashSet<Integer> symDiff_shorter(HashSet<Integer> values1,  
                                         HashSet<Integer> values2)  
  
...> {  
...>     // Schnittmenge bilden  
...>     HashSet<Integer> intersection = new HashSet<>(values1);  
...>     intersection.retainAll(values2);  
...>  
...>     // Vereinigung bilden und Schnittmenge abziehen  
...>     HashSet<Integer> result = new HashSet<>();  
...>     result.addAll(values1);  
...>     result.addAll(values2);  
...>     result.removeAll(intersection);  
...>     return result;  
...> }  
| created method symDiff_shorter(HashSet<Integer>,HashSet<Integer>)
```

```
jshell> symDiff_shorter(values1, values2)  
$84 ==> [1, 2, 4, 5]
```




Schlüssel-Wert-Abbildungen (HashMap)



Maps – Basisfunktionalitäten



- In der `ArrayList` sind Elemente als geordnete Sammlung gespeichert und man kann indiziert (Typ `int`) darauf zugreifen.
 - Eine `HashMap` hingegen speichert Abbildungen von Schlüsseln auf Werte, sogenannte »Schlüssel- Wert«-Paare.
 - zugrunde liegende Idee, jedem gespeicherten Wert einen eindeutigen Schlüssel zuzuordnen
 - Als Beispiel bilden Telefonbücher Namen auf Telefonnummern ab. Es gibt keinen indizierten Zugriff, sondern dieser erfolgt über den Schlüssel.
 - Maps werden auch als Dictionary oder als Lookup-Tabelle bezeichnet
-

HashMap



- **Erzeugen mit new**

```
jshell> var cityInhabitantsMap = new HashMap<String, Integer>()  
cityInhabitantsMap ==> {}
```

- **Erzeugen mit Collection-Factory-Methoden**

```
jshell> var unmodifiableMap = Map.of("Bern", 170_000, "Berlin", 3_500_000)  
unmodifiableMap ==> {Berlin=3500000, Bern=170000}
```

HashMaps -- Basisfunktionalitäten



- **Hinzufügen einzelner Abbildungen**

```
jshell> cityInhabitantsMap.put("Zürich", 400_000);  
$169 ==> null
```

```
jshell> cityInhabitantsMap.put("Hamburg", 2_000_000);  
$170 ==> null
```

```
jshell> cityInhabitantsMap.put("Kiel", 250_000);  
$171 ==> null
```

```
jshell> cityInhabitantsMap  
cityInhabitantsMap ==> {Kiel=250000, Hamburg=2000000, Zürich=400000}
```

- **Anzahl an Abbildungen ermitteln**

```
jshell> cityInhabitantsMap.size()  
$172 ==> 3
```

HashMaps -- Basisfunktionalitäten



- **Hinzufügen mehrerer Abbildungen**

```
jshell> cityInhabitantsMap.putAll(unmodifiableMap)
```

```
jshell> cityInhabitantsMap  
cityInhabitantsMap ==> {Bern=170000, Kiel=250000, Berlin=3500000,  
Hamburg=2000000, Zürich=400000}
```

- **Zugriff über Schlüssel (auch mit Fallback)**

```
jshell> cityInhabitantsMap.get("Zürich")  
$184 ==> 400000
```

```
jshell> cityInhabitantsMap.get("Oldenburg")  
$185 ==> null
```

```
jshell> cityInhabitantsMap.getOrDefault("Oldenburg", 175_000)  
$186 ==> 175000
```

HashMap



```
cityInhabitantsMap ==> {Bern=170000, Kiel=250000, Berlin=3500000,  
Hamburg=2000000, Zürich=400000}
```

- **Prüfen, ob ein Eintrag existiert (containsKey())**

```
jshell> cityInhabitantsMap.containsKey("Zurich")  
$175 ==> true
```

```
jshell> cityInhabitantsMap.containsKey("Bremen")  
$176 ==> false
```

- **Prüfen, ob ein Wert existiert (containsValue())**

```
jshell> cityInhabitantsMap.containsValue(400000)  
$177 ==> true
```

```
jshell> cityInhabitantsMap.containsValue(1234567)  
$178 ==> false
```



- **Aktualisieren einzelner Abbildungen**

```
jshell> cityInhabitantsMap.put("Bern", 180_000)
$187 ==> 170000
```

- **Aktualisieren mehrerer Abbildungen**

```
jshell> cityInhabitantsMap.putAll(Map.of("Kiel", 262_000,
...>                                     "Aachen", 265_000))
```

```
jshell> cityInhabitantsMap
cityInhabitantsMap ==> {Bern=180000, Kiel=262000, Berlin=3500000,
Zürich=400000, Hamburg=2000000, Zürich=400000, Aachen=265000}
```



- **Löschen**

```
jshell> cityInhabitantsMap.remove("Bern")  
$190 ==> 180000
```

```
jshell> cityInhabitantsMap.remove("Kiel")  
$191 ==> 262000
```

```
jshell> cityInhabitantsMap.remove("Aachen")  
$192 ==> 265000
```

```
jshell> cityInhabitantsMap  
cityInhabitantsMap ==> {Berlin=3500000, Zürich=400000, Hamburg=2000000,  
Zürich=400000}
```

```
jshell> cityInhabitantsMap.clear()
```

```
jshell> cityInhabitantsMap  
cityInhabitantsMap ==> {}
```

HashMaps -- Durch die Elemente iterieren



- Bei HashMaps existiert kein indizierter Zugriff
- Allerdings lassen sich die Abbildungen mit folgenden Methoden auslesen:
 1. `entrySet()` – Erzeugt eine Menge mit allen Schlüssel-Wert-Paaren.
 2. `keySet()` – Liefert eine Menge hinterlegten Schlüsseln.
 3. `values()` – Liefert eine Liste hinterlegten Werten.

```
cityInhabitantsMap ==> {Zürich=400000, Kiel=265000, Bremen=550000}
```

```
jshell> cityInhabitantsMap.keySet()  
$198 ==> [Zürich, Kiel, Bremen]
```

```
jshell> cityInhabitantsMap.values()  
$199 ==> [400000, 265000, 550000]
```

```
jshell> cityInhabitantsMap.entrySet()  
$200 ==> [Zürich=400000, Kiel=265000, Bremen=550000]
```

HashMaps -- Durch die Elemente iterieren



- **Alle Abbildungen ablaufen:**

```
jshell> for (Map.Entry entry : cityInhabitantsMap.entrySet())  
    ...> {  
    ...>     System.out.println(entry);  
    ...> }  
Zürich=400000  
Kiel=265000  
Bremen=550000
```

- **Aufbereitung der Schlüssel:**

```
jshell> String.join("-", cityInhabitantsMap.keySet())  
$201 ==> "Zürich-Kiel-Bremen"
```

- **Gesamteinwohner ermitteln (mit Vorgriff Lambdas & Streams):**

```
jshell> cityInhabitantsMap.values().stream().mapToInt(n->n).sum()  
$202 ==> 1215000
```



Generics



Containerklassen und Typen der gespeicherten Objekte



- Ursprünglich waren die Containerklassen des Collections-Frameworks wie Listen, Sets und Maps untypisiert.
 - Somit konnten dort Objekte beliebigen Typs verwaltet werden. Jedoch sind derartige heterogene Container nur für wenige Anwendungsfälle nützlich.
 - Viel öfter ist gewünscht, gleichartige Objekte, also diejenigen eines bestimmten Typs, zu speichern – man spricht von einer homogenen Zusammensetzung.
-

Containerklassen und Typen der gespeicherten Objekte



- Ohne das Sprachfeature Generics oder eine selbst geschriebene Containerklasse kann man diese Forderung nur durch eine geeignete Namensgebung, etwa `personList`, ausdrücken, nicht aber vom Compiler sicherstellen lassen.
- Seit JDK 5 lässt sich die typsichere Definition von Containerklassen mithilfe von Generics ohne weiteren eigenen Implementierungsaufwand erreichen. Es muss lediglich eine Typangabe bei der Definition einer Containerklasse erfolgen. Basierend darauf kann vom Compiler sichergestellt werden, dass dort nur gewünschte Typen verwaltet werden.

```
new ArrayList<String>()
```

```
new HashSet<Person>()
```



- **Um die Problematik nicht typsicherer Container zu rekapitulieren bzw. besser nachvollziehen zu können, betrachten wir als Beispiel die Datenspeicherung von Person-Objekten in einer ArrayList ohne Typangabe.**

```
jshell> ArrayList personList = new ArrayList()  
personList ==> []
```
 - **Man spricht auch von einem sogenannten Raw Type**
 - **Die Zugriffsmethoden, etwa `Object get(int)` und `add(Object)`, sind alle mit Eingabeparametern oder Rückgabewerten des Typs `Object` definiert.**
 - **Dadurch können Objekte beliebiger Typen verarbeitet werden.**
-



- **Nehmen wir an, es wären zwei Personen und ein String in der Liste gespeichert:**

```
jshell> for (int i = 0; i < personList.size(); i++) ...> {  
    ...>     // Explizite Typumwandlung notwendig zum Methodenaufruf  
    ...>     Person person = (Person) personList.get(i);  
    ...>     System.out.println(person.name() + " aus " + person.city());  
    ...> }
```

Max aus Musterstadt
Moritz aus Musterstadt

| Exception java.lang.ClassCastException: class java.lang.String cannot be
cast
to class REPL.\$JShell\$31\$Person



- Seit Java 5 kann und sollte man bei der Definition einer Containerklasse den zu verwaltenden Typ in spitzen Klammer angeben:

// Typsichere Definition mit Generics

```
final ArrayList<Person> personList = new ArrayList<Person>();
```

```
personList.add(new Person("Max", LocalDate.now(), "Musterstadt"));
```

```
personList.add(new Person("Moritz", LocalDate.now(), "Musterstadt"));
```

```
//personList.add("Sarah vom Auetal?); // Compile-Error
```

Definition eigener generischer Klassen



Oftmals ist es darüber hinaus wünschenswert, eigene typisierte Klassen erstellen zu können, etwa einen Datencontainer für Wertepaare beliebiger Typen. Wir definieren den Container `Pair` mit zwei formalen Typparametern `T1` und `T2`, die als Platzhalter für konkrete Typen beim Einsatz dienen:

```
public final class Pair<T1, T2>
{
    private final T1 first;
    private final T2 second;

    public Pair(final T1 first, final T2 second) {
        this.first = first;
        this.second = second;
    }

    public final T1 getFirst() {
        return first;
    }

    public final T2 getSecond() {
        return second;
    }
}
```

`Pair<String, Person>`

`Pair<String, Integer>`



- **Der Container Pair zur Rückgabe mehrerer Werte:**

```
public static void main(String[] args)
{
    Pair<String, List<String>> result = calcComplexResult();
    System.out.println("first: " + result.first);
    System.out.println("second: " + result.second);
}

static Pair<String, List<String>> calcComplexResult()
{
    return new Pair<>("INFO", List.of("VALUE1", "VALUE2"));
}
```

- =>

```
first: INFO
second: [VALUE1, VALUE2]
```



Basis-Interfaces für Collections





- Bislang haben wir die Containerklassen wie `ArrayList`, `HashSet` und `HashMap` direkt ohne Abstraktion genutzt.
- Das ist oftmals eher schlechtes Design, war aber zum Einstieg in Datenstrukturen ein Kompromiss, um nicht gleich zu viele Themen behandeln zu müssen.
- Normalerweise definiert man die Datenstrukturen auf Basis der jeweiligen Interfaces `List<E>`, `Set<E>` bzw. `Map<K, V>`, etwa wie folgt:

```
jshell> List<String> names = new ArrayList<>()  
names ==> []
```

```
jshell> Set<Integer> numbers = new HashSet<>()  
numbers ==> []
```

```
jshell> Map<String, Integer> mapping = Map.of()  
mapping ==> {}
```



- **List und Set besitzen noch eine gemeinsame Basis: Collection**

```
jshell> Collection<String> names = new ArrayList<>()  
names ==> []
```

Das Interface `Collection<E>` definiert die Basis für diverse Containerklassen, die das Interface `List<E>` bzw. `Set<E>` erfüllen und somit Listen bzw. Mengen repräsentieren. Das Interface `Collection<E>` bietet *keinen* indizierten Zugriff, aber folgende Methoden:

- `int size()` – Ermittelt die Anzahl der in der Collection gespeicherten Elemente.
- `boolean isEmpty()` – Prüft, ob Elemente vorhanden sind.
- `boolean add(E element)` – Fügt ein Element zur Collection hinzu. Gibt `true` zurück, wenn sich die Collection ändert – bei Sets ist dies bei Duplikaten nicht der Fall.
- `boolean addAll(Collection<? extends E> collection)` – Fügt der Collection alle übergebenen Elemente hinzu.



Das Interface `List<E>`

Das Interface `List<E>` bildet die Basis für alle Listen und bietet *zusätzlich* zu den Methoden des Interface `Collection<E>` folgende indizierte, 0-basierte Zugriffe:

- `E get(int index)` – Ermittelt das Element der Liste an der Position `index`.
- `void add(int index, E element)` – Fügt das Element `element` an der Position `index` der Liste ein.
- `E set(int index, E element)` – Ersetzt das Element an der Position `index` der Liste durch das übergebene Element `element` und liefert das zuvor an dieser Position gespeicherte Element zurück.³
- `E remove(int index)` – Entfernt das Element an der Position `index` der Liste und liefert das gelöschte Element zurück.
- `int indexOf(Object object)` und
- `int lastIndexOf(Object object)` – Mit diesen Methoden wird die Position eines gesuchten Elements zurückgeliefert. Die Gleichheit zwischen dem Suchelement und den einzelnen Elementen der Liste wird mit der Methode `equals(Object)` überprüft. Die Suche startet dabei entweder am Anfang (`indexOf(Object)`) oder am Ende der Liste (`lastIndexOf(Object)`).

Interface Set



- Im Gegensatz zum Interface `List<E>` sind im Interface `Set<E>` keine Methoden zusätzlich zu denen des Interface `Collection<E>` vorhanden.
 - Allerdings wird ein anderes Verhalten für die Methoden `add(E)` und `addAll(Collection<E>)` vorgeschrieben.
 - Dieser Unterschied zwischen `Set<E>` und dem zugrunde liegenden Interface `Collection<E>` ist nötig, um Duplikatfreiheit zu garantieren, selbst dann, wenn der Menge das gleiche Objekt mehrfach hinzugefügt wird.
-



Iteratoren





- **Iteratoren werden in Java vielfach verwendet, etwa für Schleifen**
- **Ein Iterator ist eine Art Zeiger, der auf das aktuelle Element verweist und das nächste Element eines Datencontainers liefern kann.**
- **Alle Datenstrukturen, die das Interface `Collection<E>` erfüllen, bieten über die Methode `iterator()` Zugriff auf das Interface `java.util.Iterator<E>`, das zwei Methoden bietet:**
 - `hasNext()` und
 - `next()`.

Iteratoren im (naiven) Einsatz



```
jshell> var names = List.of("Tim", "Tom", "Mike")
names ==> [Tim, Tom, Mike]
```

```
jshell> Iterator<String> it = names.iterator()
it ==> java.util.ImmutableCollections$ListItr@34a245ab
```

```
jshell> it.next()
$5 ==> "Tim"
```

```
jshell> it.next()
$6 ==> "Tom"
```

```
jshell> it.next()
$7 ==> "Mike"
```

```
jshell> it.next()
| Exception java.util.NoSuchElementException
|       at ImmutableCollections$ListItr.next (ImmutableCollections.java:375)
|       at (#9:1)
```

Iteratoren – typisches Einsatzmuster



```
// Holen eines Iterators
Iterator<T> it = ds.iterator()
while (it.hasNext())
{
    // Zugriff auf nächstes Element
    T element = it.next();
    // do something with element
}
```

Iteratoren im korrekten Einsatz



```
jshell> var names = List.of("Tim", "Tom", "Mike", "Andy")
names ==> [Tim, Tom, Mike, Andy]
```

```
jshell> Iterator<String> it = names.iterator()
it ==> java.util.ImmutableCollections$ListItr@17d10166
```

```
jshell> while (it.hasNext())
...> {
...>     String nextValue = it.next();
...>     System.out.println(nextValue);
...> }
```

```
Tim
Tom
Mike
Andy
```

Eigene Iteratoren (zur Fakultätsberechnung)



```
jshell> class FactorialIterator implements Iterator<Integer>
...> {
...>     private int n;
...>     private int result = 1;
...>     private int iteration = 1;
...>
...>     public FactorialIterator(int n) {
...>         this.n = n;
...>     }
...>
...>     public boolean hasNext() {
...>         return iteration <= n;
...>     }
...>
...>     public Integer next() {
...>         result *= iteration;
...>         iteration++;
...>         return result;
...>     }
...> }
| created class FactorialIterator
```



- **Handgestrickt**

```
jshell> Iterator<Integer> fac = new FactorialIterator(5)
fac ==> FactorialIterator@504bae78
```

```
jshell> fac.next()
$13 ==> 1
```

```
jshell> fac.next()
$14 ==> 2
```

```
jshell> fac.next()
$15 ==> 6
```

```
jshell> fac.next()
$16 ==> 24
```

```
jshell> fac.next()
$17 ==> 120
```

```
jshell> fac.next()
$18 ==> 720
```

Mit Schleife

```
jshell> var fac = new FactorialIterator(5)
fac ==> FactorialIterator@5eb5c224
```

```
jshell> while (fac.hasNext())
...>     System.out.println(fac.next())
1
2
6
24
120
```



ArrayIterator selbstgestrickt



```
public class ArrayIteratorAdapter implements Iterator<T>
{
    int currentPosition;
    final T[] adaptee;

    public ArrayIteratorAdapter(final T[] adaptee)
    {
        this.adaptee = adaptee;
        this.currentPosition = 0;
    }

    // TODO
    public boolean hasNext()
    public T next()
}
```

ArrayIterator – Beispiellösung



```
public class ArrayIteratorAdapter<T> implements Iterator<T>
{
    int currentPosition;
    final T[] adaptee;

    public ArrayIteratorAdapter(final T[] adaptee) {
        this.adaptee = adaptee;
        this.currentPosition = 0;
    }

    @Override
    public boolean hasNext() {
        return currentPosition < adaptee.length;
    }

    @Override
    public T next() {
        final T next = adaptee[currentPosition];
        currentPosition++;
        return next;
    }
}
```



Sortierung





- Mitunter ist es praktisch, wenn die in einer Containerklasse verwalteten Daten sortiert vorliegen.
 - Für Arrays und Listen gibt es keine Automatik. Um diese sortiert zu halten, wird ein manueller Schritt notwendig. Hierbei unterstützt, wie schon gesehen, die Methode `sort()`.
 - Natürliche Ordnung und `Comparable<T>` – Sofern zu speichernde Objekte das Interface `Comparable<T>` erfüllen, können sie darüber ihre Ordnung, d. h. ihre Reihenfolge untereinander, beschreiben. Diese Reihenfolge wird auch als natürliche Ordnung bezeichnet, da sie durch die Objekte selbst bestimmt wird.
-



Sortierungen und das Interface `Comparable<T>`

Oftmals besitzen Werte oder Objekte eine natürliche Ordnung: Das gilt etwa für Zahlen und Strings. Für komplexe Typen ist die Aussage »kleiner« bzw. »größer« nicht immer sofort ersichtlich, lässt sich aber bei Bedarf selbst definieren.

Dazu erlaubt das Interface `Comparable<T>` typsichere Vergleiche und deklariert die Methode `compareTo(T o)` folgendermaßen:

```
public interface Comparable<T>
{
    public int compareTo(T o);
}
```

Das Vorzeichen des Rückgabewerts bestimmt die Reihenfolge der Elemente:

- `= 0`: Der Wert 0 bedeutet Gleichheit des aktuellen und des übergebenen Objekts.
- `< 0`: Das aktuelle Objekt ist kleiner als das übergebene Objekt.
- `> 0`: Das aktuelle Objekt ist größer als das übergebene Objekt.

Diverse Klassen im JDK (alle Wrapper-Klassen, `String`, `Date` usw.) implementieren das Interface `Comparable<T>` und sind damit automatisch sortierbar.



- Teilweise benötigt man zusätzlich zur natürlichen Ordnung weitere oder alternative Sortierungen, etwa wenn man Personen nicht nach Nachname, sondern alternativ nach Vorname und Geburtsdatum ordnen möchte.
 - Diese ergänzenden Sortierungen können mithilfe von Implementierungen des Interface Comparator<T> festgelegt werden.
 - Der Vorteil ist, dass man Anwendungsklassen nicht überfrachtet, sondern Sortierungen in eigenständigen Vergleichsklassen definiert wird.
 - Es lassen sich von der natürlichen Ordnung abweichende Sortierungen realisieren und auch Objekte von Klassen sortieren, für die keine natürliche Ordnung existiert, weil Comparable<T> nicht implementiert ist.
-

Vordefinierte Sortierungen in Comparator<T>



- `naturalOrder()`
- `reverseOrder()`
- `nullsFirst()` / `nullsLast()`

```
jshell> var names = Arrays.asList("A", null, "B", "C", null, "D")  
names ==> [A, null, B, C, null, D]
```

```
jshell> Comparator<String> naturalOrder = Comparator.naturalOrder();  
naturalOrder ==> INSTANCE
```

```
jshell> var nullsFirst = Comparator.nullsFirst(naturalOrder);  
nullsFirst ==> java.util.Comparators$NullComparator@18e8568
```

```
jshell> names.sort(nullsFirst)
```

```
jshell> names  
names ==> [null, null, A, B, C, D]
```



ACHTUNG: JAVA Fallstrick ...

```
jshell> var names = List.of("Mike", "Andy", "Peter", "Jim", "Tim")
names ==> [Mike, Andy, Peter, Jim, Tim]
```

```
names.sort(Comparator.naturalOrder())
| Exception java.lang.UnsupportedOperationException
```

```
jshell> var modifiable = new ArrayList<>(names)
modifiable ==> [Mike, Andy, Peter, Jim, Tim]
```

```
jshell> modifiable.sort(Comparator.naturalOrder())
```

```
jshell> modifiable
modifiable ==> [Andy, Jim, Mike, Peter, Tim]
```

Natürliche Ordnung / Umgedrehte Sortierung



```
jshell> var modifiable = new ArrayList<>(names)
modifiable ==> [Mike, Andy, Peter, Jim, Tim]
```

```
jshell> modifiable.sort(Comparator.naturalOrder())
```

```
jshell> modifiable
modifiable ==> [Andy, Jim, Mike, Peter, Tim]
```

```
jshell> modifiable.sort(Comparator.reverseOrder())
```

```
jshell> modifiable
modifiable ==> [Tim, Peter, Mike, Jim, Andy]
```

Comparator<T>



- **comparing()** – Definiert einen Komparator basierend auf der Extraktion zweier Werte, die sich mit **Comparable<T>** vergleichen lassen.
- **thenComparing(), thenComparingInt()/-Long() und -Double()** – Hintereinanderschaltung von Komparatoren

```
Comparator<Person> byName = Comparator.comparing(Person::getName);  
Comparator<Person> byAge = Comparator.comparing(Person::getAge);  
  
// Kombination von Komparatoren  
Comparator<Person> byNameAndFirstname = byName.  
                                         thenComparing(byFirstname);  
Comparator<Person> byNameAndAge = byName.thenComparing(byAge);
```

Spezielle Sortierung nach letztem Buchstaben / nach Länge



- **byLastChar**

```
jshell> modifiable.sort(Comparator.comparing(name -> name.charAt(name.length() - 1)))
```

```
jshell> modifiable  
modifiable ==> [Mike, Tim, Jim, Peter, Andy]
```

- **Nach Länge**

```
jshell> modifiable.sort(Comparator.comparing(String::length))
```

```
jshell> modifiable  
modifiable ==> [Jim, Tim, Mike, Andy, Peter]
```

Case-Insensitive Sortierung



```
jshell> var names = new ArrayList<>(List.of("tim", "TOM", "MIKE", "Michael",  
"andreas", "STEFAN"))  
names ==> [tim, TOM, MIKE, Michael, andreas, STEFAN]
```

```
jshell> names.sort(Comparator.comparing(String::toLowerCase))
```

```
jshell> names  
names ==> [andreas, Michael, MIKE, STEFAN, tim, TOM]
```

Spezielle Sortierung nach Länge sowie auch absteigend



- Nach Länge auf- und absteigend

```
jshell> var names = List.of("Tim", "Tom", "Michael", "Andy", "James")
names ==> [Tim, Tom, Michael, Andy, James]
```

```
jshell> var modifiable = new ArrayList<>(names)
modifiable ==> [Tim, Tom, Michael, Andy, James]
```

```
jshell> modifiable.sort(Comparator.comparing(String::length))
```

```
jshell> modifiable
modifiable ==> [Tim, Tom, Andy, James, Michael]
```

```
jshell> modifiable.sort(Comparator.comparing(String::length).reversed())
```

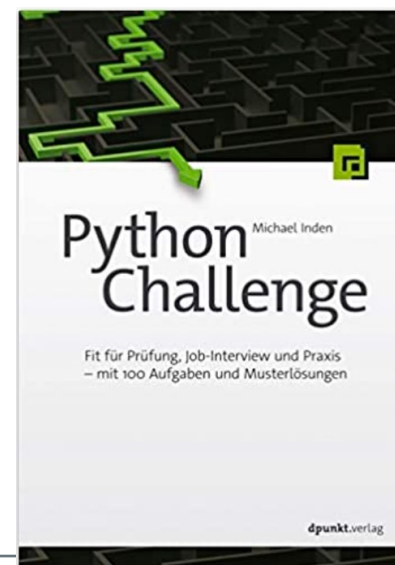
```
jshell> modifiable
modifiable ==> [Michael, James, Andy, Tim, Tom]
```



Exercises Part 5

https://github.com/Michaeli71/JAVA_INTRO







Thank You
