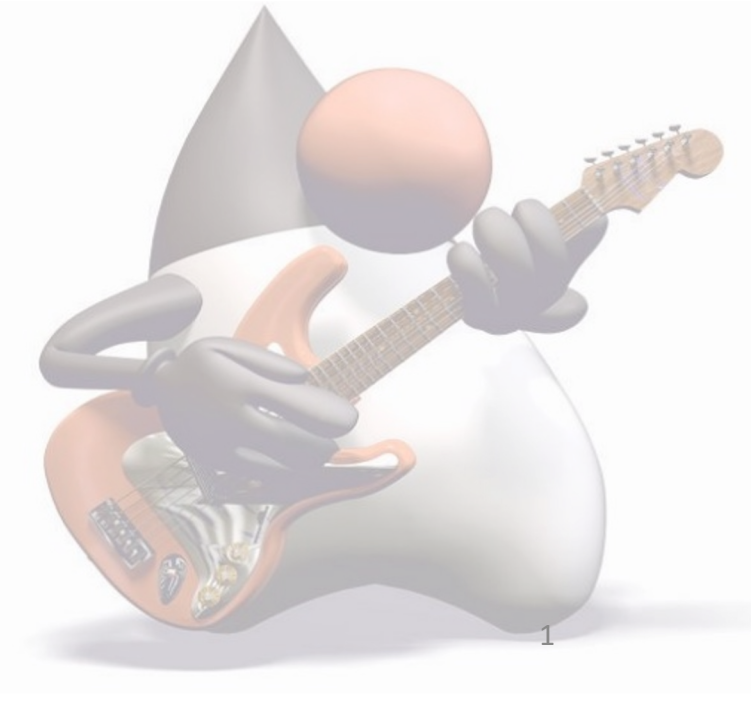
Java Fundamentals

Collections and Maps



Wieso Collections?

- Arrays haben einige Nachteile
  - statisch in der Größe
  - Bestimmte Operationen sind langsam
  - Suche nach Werten ist langsam
  - Einfügen neuer Elemente am Anfang oder in der Mitte des Arrays sind langsam und umständlich zu implementieren

Data Structures

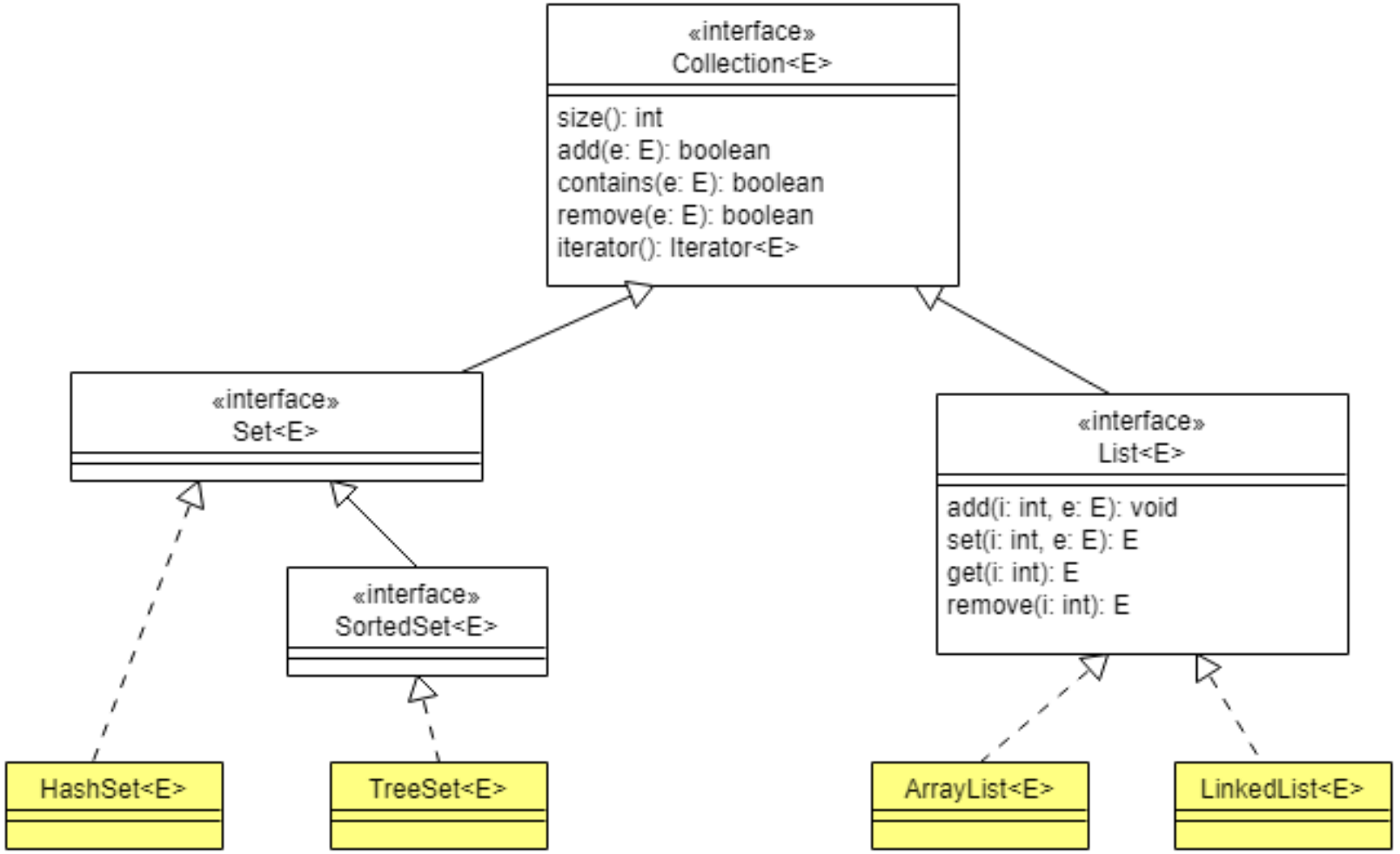
Es gibt neben Arrays eine Reihe komplexerer Datenstrukturen in Programmiersprachen

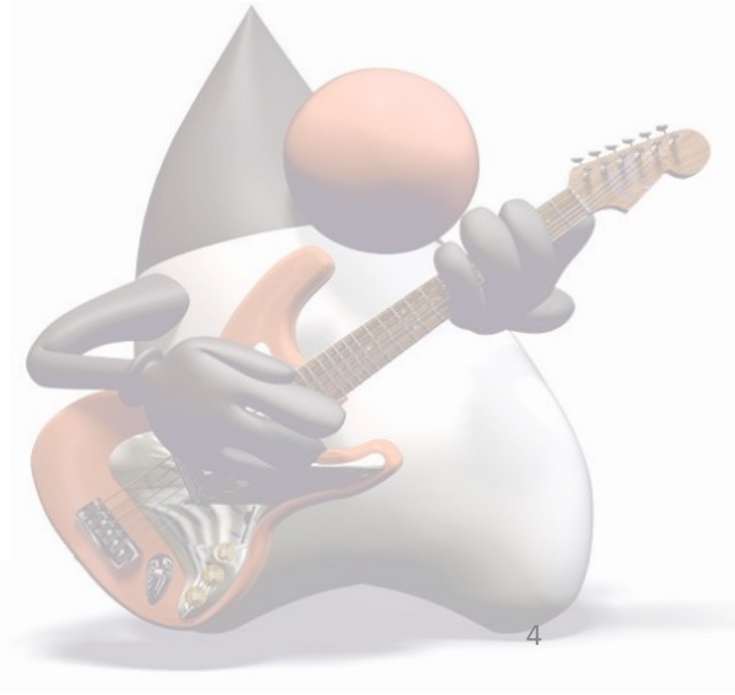
Beispiele sind

- Listen
- Maps (Assoziativarrays)
- Sets

In Java gibt es ein leistungsfähiges Collections API, das die wichtigsten Datenstrukturen implementiert

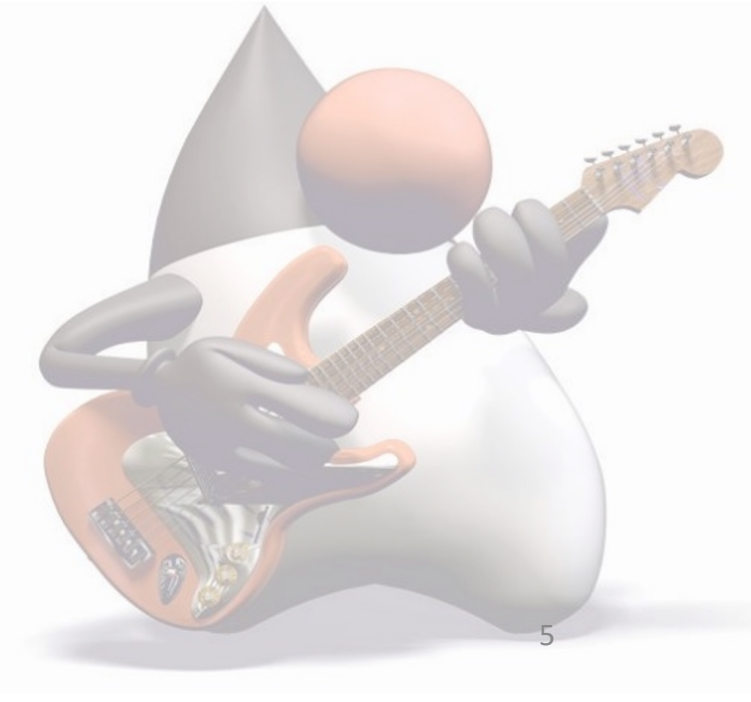
Collections





Was können die Collections?

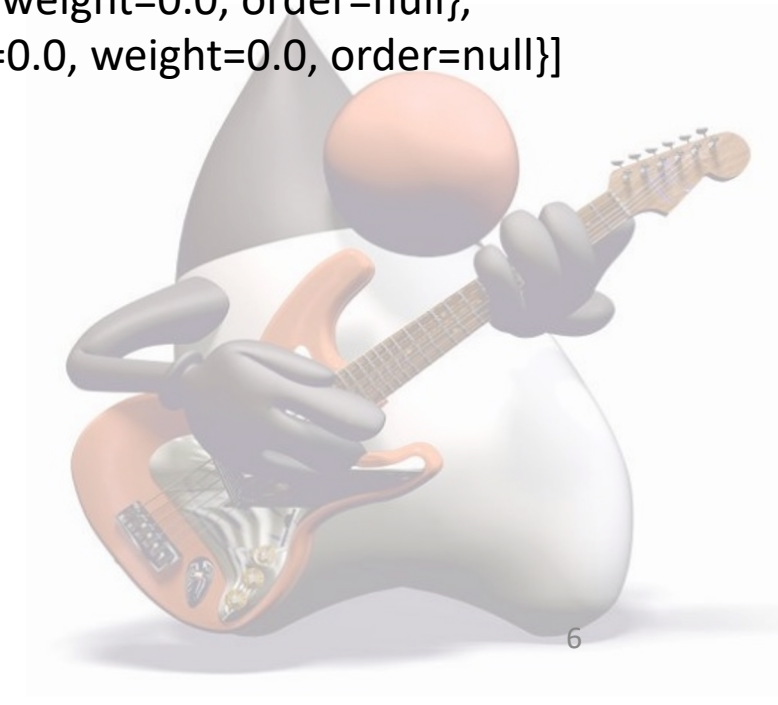
- Manche Collections sind sortierbar
- Manche Collections garantieren Eindeutigkeit
- Manche Collections können Key-Value Paare halten (Maps)



Initialisierung einer Collection

```
public static void main(String... args) {
    List<Customer> customerList = new ArrayList<>();
    customerList.add(new Customer("Peter", "Stangl"));
    customerList.add(new Customer("Michael", "Schaffler"));
    System.out.println(customerList);
}
```

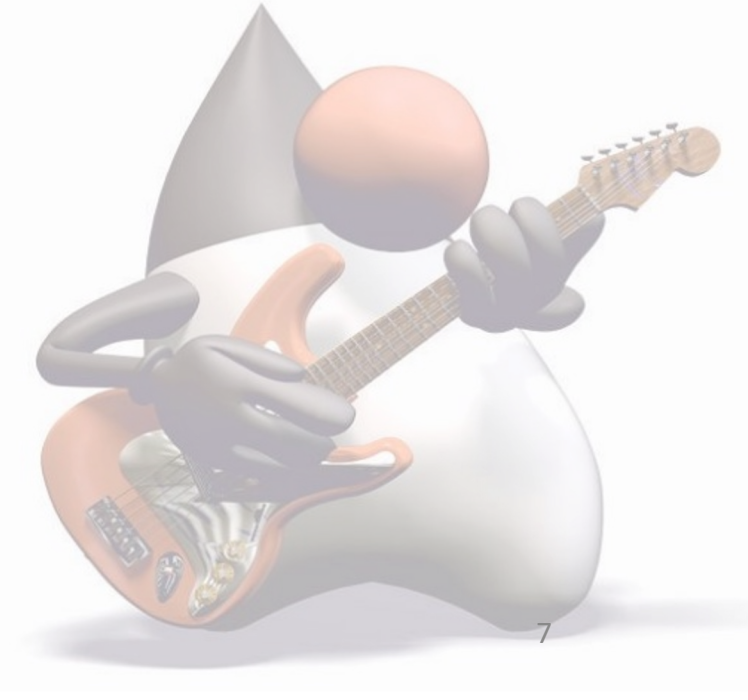
[Customer{name=Peter, lastname=Stangl, address=, city=, zipcode=0, height=0.0, weight=0.0, order=null},
Customer{name=Michael, lastname=Schaffler, address=, city=, zipcode=0, height=0.0, weight=0.0, order=null}]



List (Eigenschaften)

- Die meistgenutzte Collection
- Sie unterstützt Zugriff über den Index
- Sie ist geordnet (d.h. die Reihenfolge der Elemente bleibt erhalten)

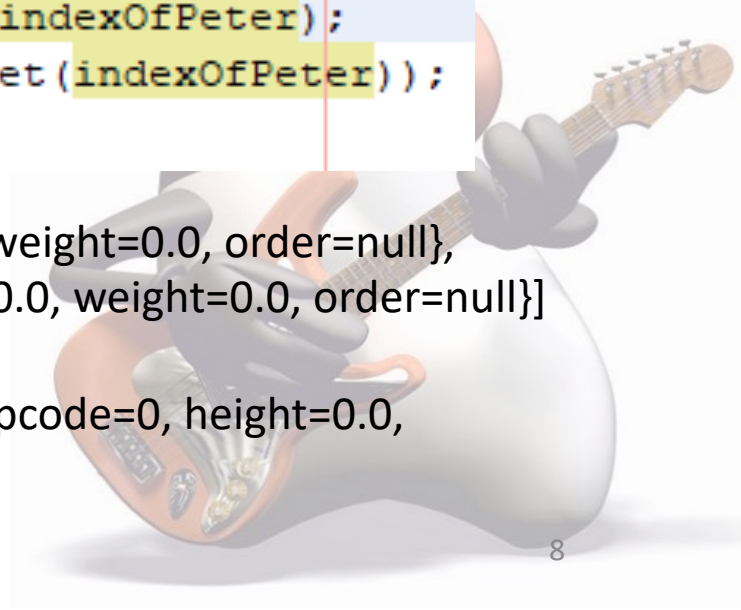
- Typen: ArrayList, LinkedList



List (Code)

```
public static void main(String... args) {
    List<Customer> customerList = new ArrayList<>();
    customerList.add(new Customer("Peter", "Stangl"));
    customerList.add(new Customer("Michael", "Schaffler"));
    System.out.println(customerList);

    Customer searchAbleCustomer = new Customer("Peter", "Stangl");
    int indexOfPeter = customerList.indexOf(searchAbleCustomer);
    System.out.println("Das Objekt befeindet sich an Stell:" + indexOfPeter);
    System.out.println("Das gefundene Objekt:" + customerList.get(indexOfPeter));
}
```



[Customer{name=Peter, lastname=Stangl, address=, city=, zipcode=0, height=0.0, weight=0.0, order=null},
Customer{name=Michael, lastname=Schaffler, address=, city=, zipcode=0, height=0.0, weight=0.0, order=null}]
Das Objekt befeindet sich an Stell:0
Das gefundene Objekt:Customer{name=Peter, lastname=Stangl, address=, city=, zipcode=0, height=0.0, weight=0.0, order=null}

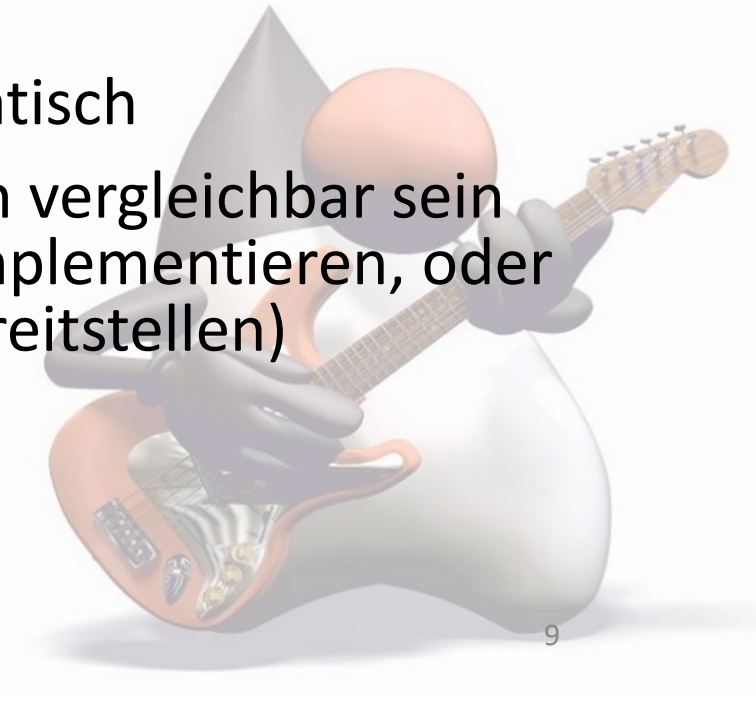
Set (Eigenschaften)

- Duplikate sind nicht erlaubt
- Keine definierte Reihenfolge
- Kein Index
- Typen: HashSet

- Hashtable Implementierung
- Sehr schnell
- Damit das funktioniert müssen Objekte HashCode und Equals implementieren

- SortedSet
  - Reihenfolge sortiert
  - Kein Index
- Typen: TreeSet

- Sortiert automatisch
- Objekte müssen vergleichbar sein (Comparable implementieren, oder Comparator bereitstellen)

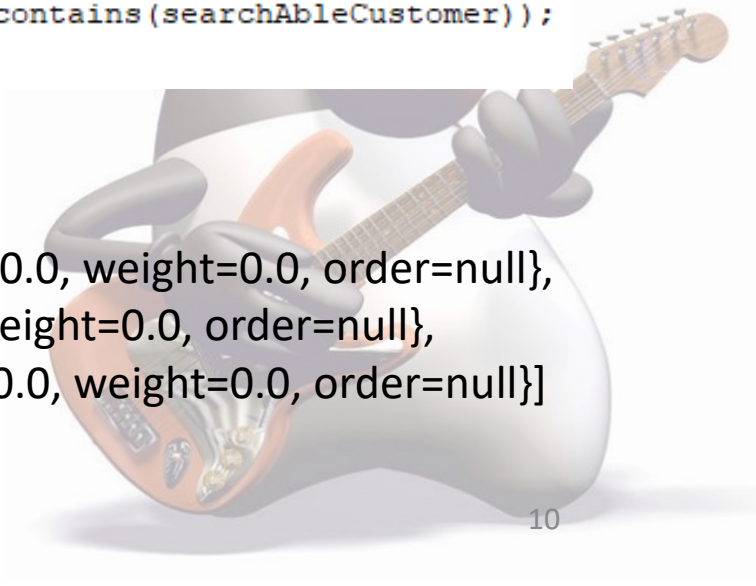


Set (Code)

```
public static void main(String... args) {
    Set<Customer> customerSet = new HashSet<Customer>();

    customerSet.add(new Customer("Peter", "Stangl"));
    customerSet.add(new Customer("Michael", "Schaffler"));
    customerSet.add(new Customer("Katarina", "Schaffler"));
    System.out.println(customerSet);

    Customer searchAbleCustomer = new Customer("Peter", "Stangl");
    System.out.println("Es gibt schon einen Kunden mit diesen Namen: " + customerSet.contains(searchAbleCustomer));
}
```

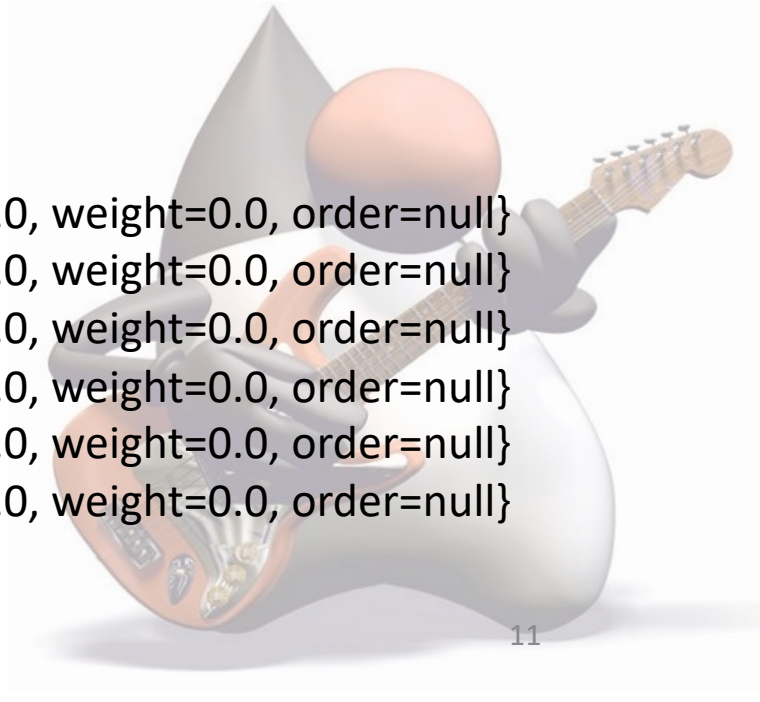


[Customer{name=Michael, lastname=Schaffler, address=, city=, zipcode=0, height=0.0, weight=0.0, order=null},
Customer{name=Peter, lastname=Stangl, address=, city=, zipcode=0, height=0.0, weight=0.0, order=null},
Customer{name=Katarina, lastname=Schaffler, address=, city=, zipcode=0, height=0.0, weight=0.0, order=null}]
Es gibt schon einen Kunden mit diesen Namen: true

Set (Code)

```
public static void main(String... args) {
    Set<Customer> set = new HashSet<Customer>();
    for (int i = 1; i < 10; i++) {
        Customer c = new Customer();
        c.setName("Customer Nr." + i);
        c.setId(i);
        set.add(c);
    }
    Iterator<Customer> it = set.iterator();
    while (it.hasNext()) {
        Customer c = it.next();
        System.out.println(c);
    }
}
```

```
Customer{name=Customer Nr.3, lastname=, address=, city=, zipcode=0, height=0.0, weight=0.0, order=null}
Customer{name=Customer Nr.4, lastname=, address=, city=, zipcode=0, height=0.0, weight=0.0, order=null}
Customer{name=Customer Nr.1, lastname=, address=, city=, zipcode=0, height=0.0, weight=0.0, order=null}
Customer{name=Customer Nr.2, lastname=, address=, city=, zipcode=0, height=0.0, weight=0.0, order=null}
Customer{name=Customer Nr.7, lastname=, address=, city=, zipcode=0, height=0.0, weight=0.0, order=null}
Customer{name=Customer Nr.8, lastname=, address=, city=, zipcode=0, height=0.0, weight=0.0, order=null}
....
```



TreeSet - Comparable Interface

```java
public class Customer implements Comparable {

    private int id;
    private String name = "";
    private String lastname = "";

    @Override
    public int compareTo(Object o) {
        if (o != null && o instanceof Customer) {
            Customer custom = (Customer) o;
            return name.compareTo(custom.name);
        }
        return -1;
    }
```

```java
public static void main(String... args) {
    Set<Customer> set = new TreeSet<Customer>();
    for (int i = 1; i < 10; i++) {
        Customer c = new Customer();
        c.setName("Customer Nr." + i);
        c.setId(i);
        set.add(c);
    }
    Iterator<Customer> it = set.iterator();
    while (it.hasNext()) {
        Customer c = it.next();
        System.out.println(c);
    }
}
```

Customer{name=Customer Nr.1, lastname=, address=, city=, zipcode=0, height=0.0, weight=0.0, order=null}
Customer{name=Customer Nr.2, lastname=, address=, city=, zipcode=0, height=0.0, weight=0.0, order=null}
Customer{name=Customer Nr.3, lastname=, address=, city=, zipcode=0, height=0.0, weight=0.0, order=null}
Customer{name=Customer Nr.4, lastname=, address=, city=, zipcode=0, height=0.0, weight=0.0, order=null}
Customer{name=Customer Nr.5, lastname=, address=, city=, zipcode=0, height=0.0, weight=0.0, order=null}
Customer{name=Customer Nr.6, lastname=, address=, city=, zipcode=0, height=0.0, weight=0.0, order=null}
…

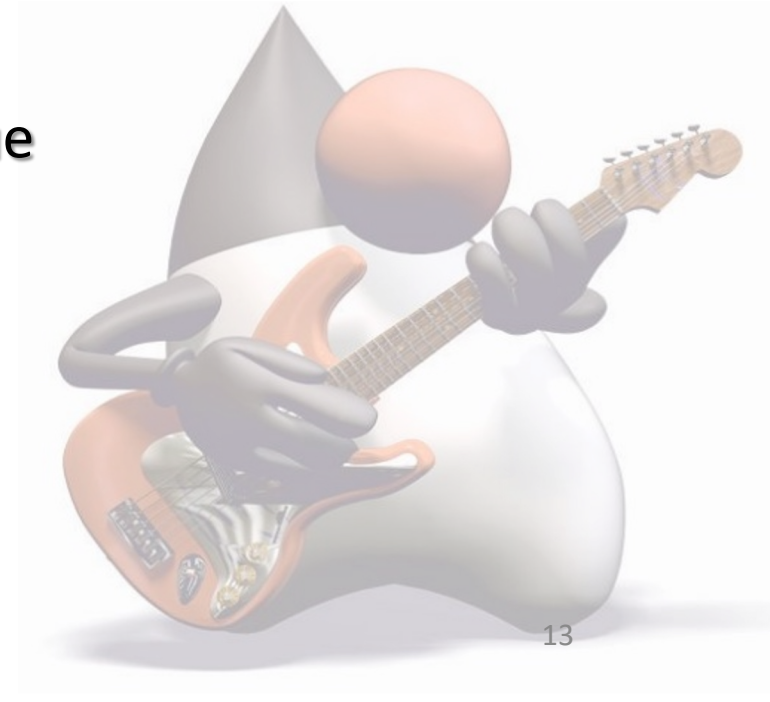
Queue / Deque (Nur für wenige Anwendungsfalle)

Queue

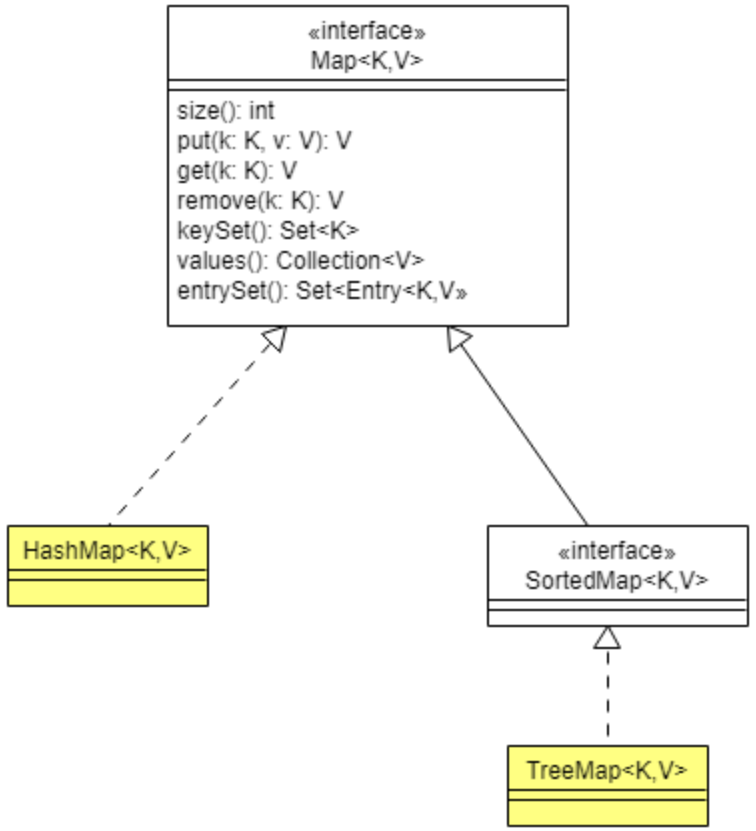
- Implementierung nach FiFo Prinzip
- Typen: PriorityQueue

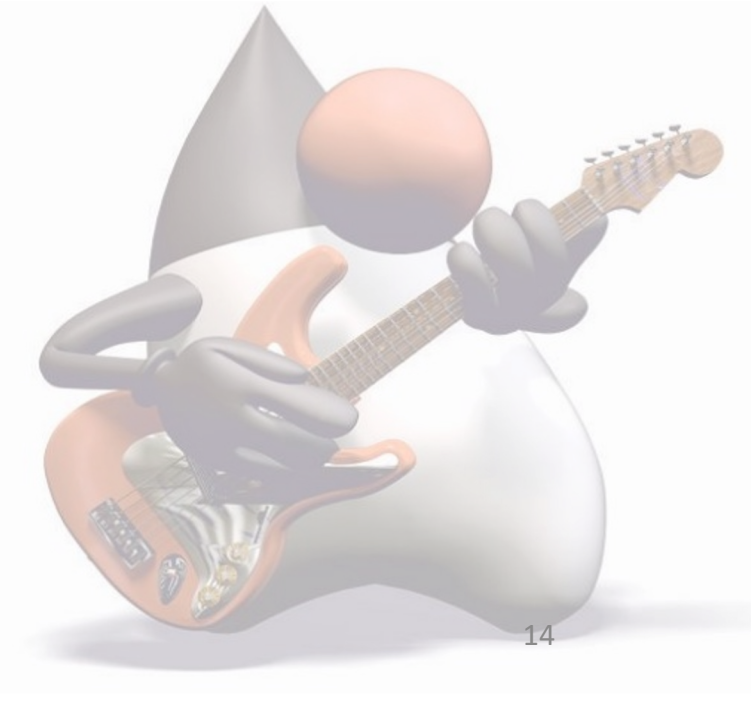
Deque

- Können FiFo - LiFo
- Typen:
  - PriorityQueue
  - ArrayDeque



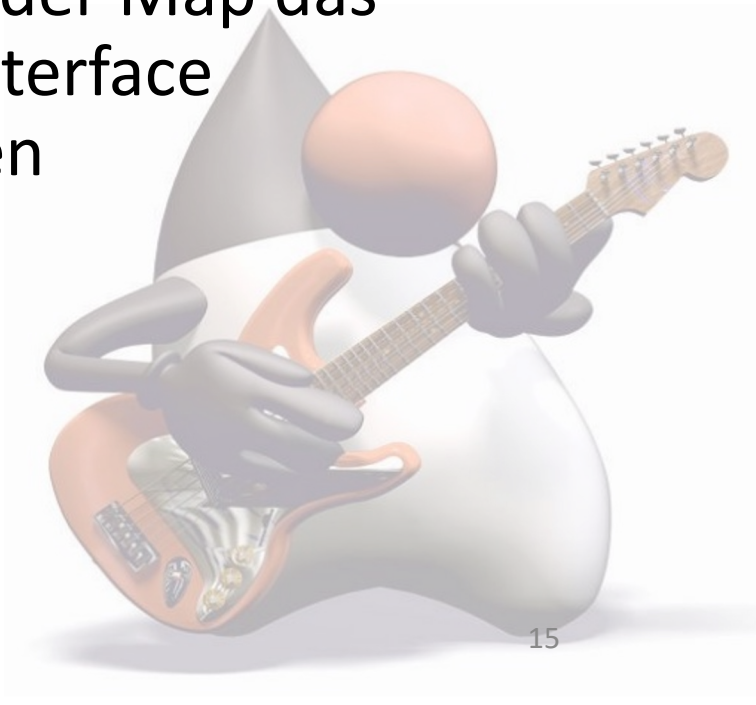
Maps


«interface»
Map<K,V>
size(): int
put(k: K, v: V): V
get(k: K): V
remove(k: K): V
keySet(): Set<K>
values(): Collection<V>
entrySet(): Set<Entry<K,V»
HashMap<K,V>
«interface»
SortedMap<K,V>
TreeMap<K,V>



Map / SortedMap

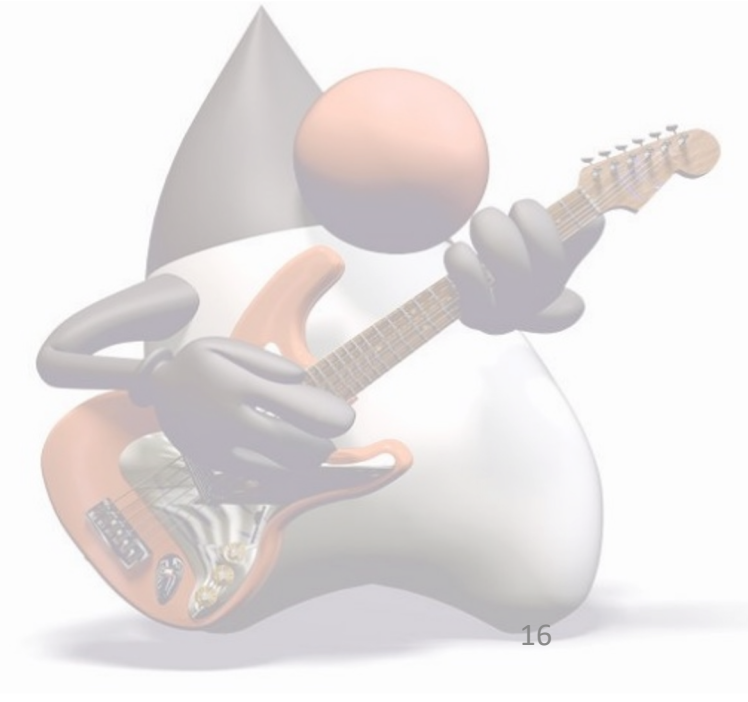
- Map
- Nicht sortiert
- SortedMap
- Immer sortiert, setzt voraus das die Objekte in der Map das Comparable Interface implementieren



Map (Code)

```
    Map<Integer, Customer> map = new HashMap<Integer, Customer>();
    for (int i = 1; i < 10; i++) {
        Customer c = new Customer();
        c.setName("Customer Nr." + i);
        c.setId(i);
        map.put(c.getId(), c);
    }
    //Accessing an element by key
    Customer customer = map.get(1);
    System.out.println(customer);

    //Iterating over all entries
    for (Entry<Integer, Customer> entry : map.entrySet()) {
        Integer key = entry.getKey();
        Customer value = entry.getValue();
    }
    //All keys
    map.keySet();
    //All Values
    map.values();
}
```

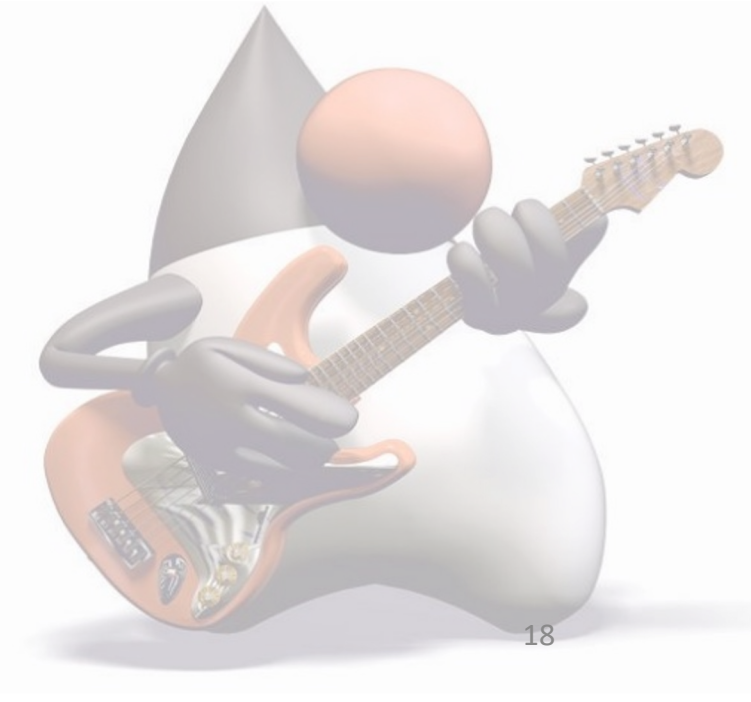


Threadsafe

- Collections sind nicht Threadsafe
- Alle Collections unter java.util.concurrent sind Threadsafe
  - BlockingQueue
  - ConcurrentMap (HashMap)
  - ConcurrentNavigableMap (TreeMap)

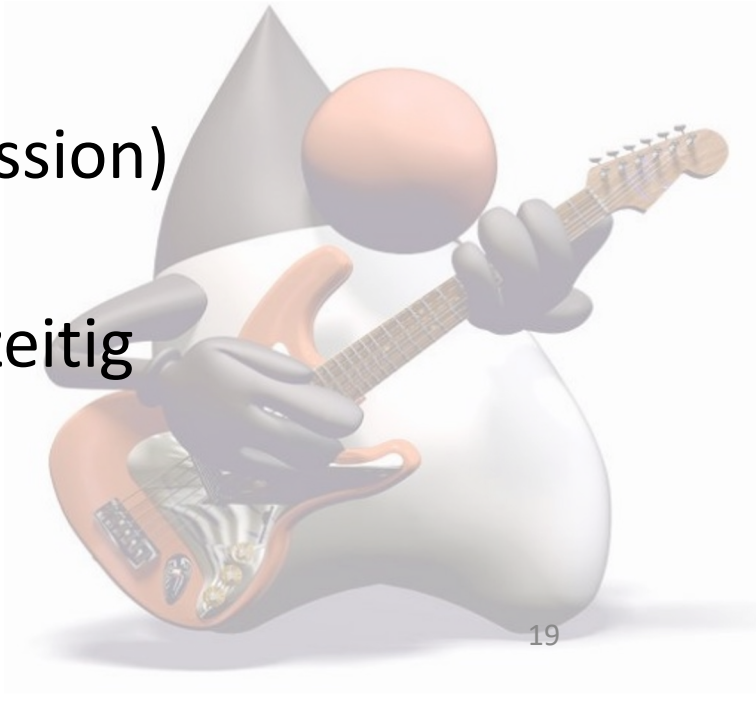
Java 8 Enhancements

- Neue replace und remove Methoden
  - replace(key,value);
  - replaceAll(BiFunction <K,V,V>);
  - remove(key,value);



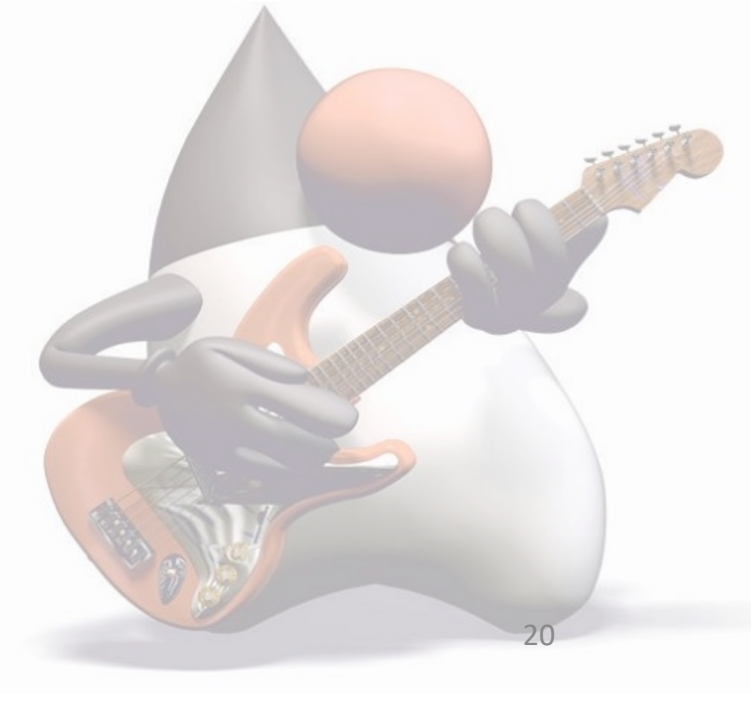
Java 8 Enhancements

- GetOrDefault Method
  - Wenn der Key nicht gefunden wird -> return default
- putIfAbsent
  - Wenn Key nicht gefunden wird in die Map geschrieben
- Map.forEach((k,v) -> sout(k+ " =" +v); (Lamda Expression)
- Verschiedene Methoden um mehrere Werte gleichzeitig auszutauschen oder zu verändern



Wann verwende ich welche Collection?

- Use Case abhängig
- Möchte ich meine Daten sortiert?
- Möchte ich meine Daten unique?
- Benötige ich meine Daten nach FI/FO?
- Habe ich Key,Value Paare?
- Usw…



Java Fundamentals

Collections and Maps