

# 19 Collections Framework

Collection = Containerklasse, die andere Objekte enthält.

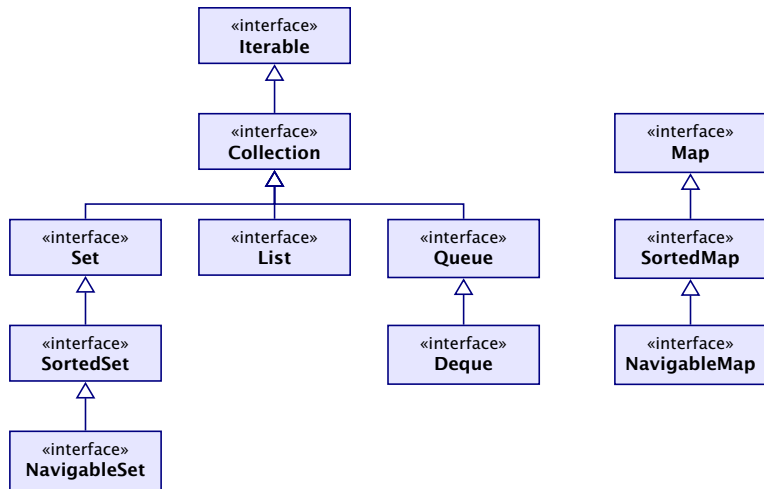
## Inhalte:

- ▶ Schnittstellen
- ▶ Implementierungen
- ▶ Algorithmen

## Vorteile:

- ▶ Einheitlicher Zugriff auf Containerobjekte.
- ▶ Abstraktion von den Implementierungsdetails.
- ▶ Effiziente Standardimplementierungen.
- ▶ Durch Einhalten vorgegebener APIs können unabhängige Containerklassen zusammenarbeiten.

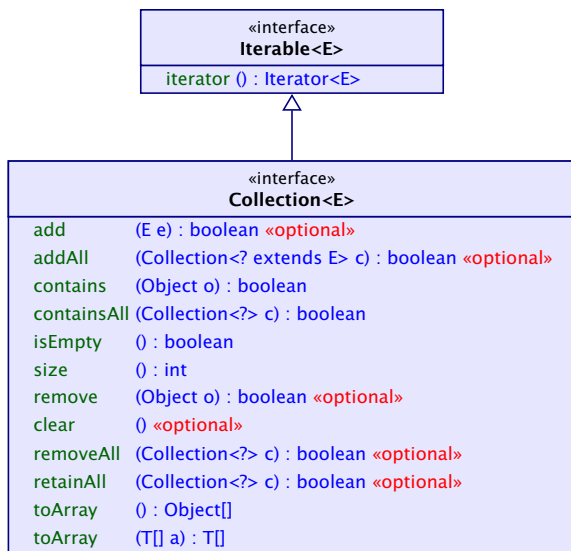
# Interfaces - Maps and Collections



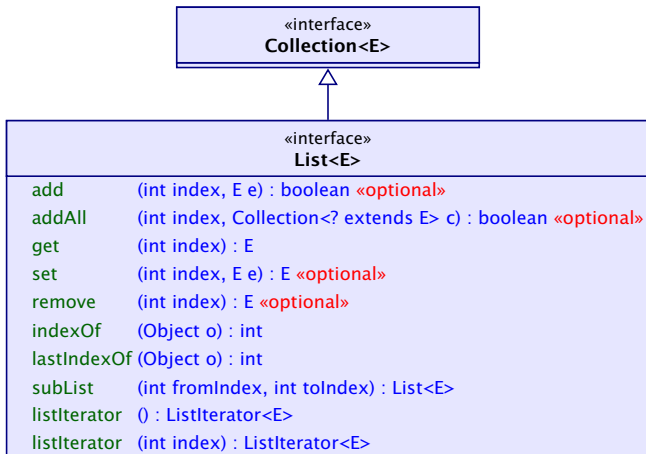
# Überblick

- ▶ **Collection.** Allgemeine Containerschnittstelle; das Framework bietet keine konkrete Implementierung;
- ▶ **Set.** Ein Container, der keine Duplikate enthält.
- ▶ **List.** Ein geordneter Container; man kann Elemente an vorgegebenen Positionen einfügen; Duplikate sind erlaubt.
- ▶ **Queue.** Ein Container, der zusätzliche Einfüge, und Abfragemöglichkeiten bietet; realisiert (spezielle) Ordnung der Elemente (z.B. **FIFO**, **LIFO**, **PriorityQueue**)
- ▶ **Map** Assoziativer Speicher. Bildet Schlüssel auf Werte ab. Schlüssel können nicht doppelt vorkommen.

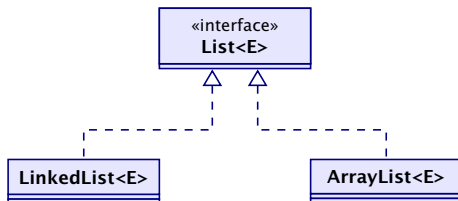
# Collection Interface



# List Interface



# List – Implementierungen



## Laufzeiten:

	<i>add</i>	<i>remove</i>	<i>get</i>	<i>contains</i>
ArrayList	$O(1)$	$O(n)$	$O(1)$	$O(n)$
LinkedList	$O(1)$	$O(1)$	$O(n)$	$O(n)$

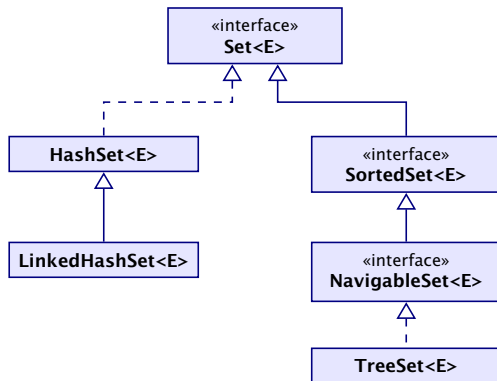
# Set Interface(s)

Das **Set**-interface enthält nur Methoden aus dem **Collection**-Interface. Diese haben aber teilweise eine andere Bedeutung, da Duplikate nicht erlaubt sind.

Zusätzliche Methoden von **SortedSet**:

«interface» <b>SortedSet&lt;E&gt;</b>	
<b>first</b>	() : E
<b>last</b>	() : E
<b>headSet</b>	(E toElement) : SortedSet<E>
<b>tailSet</b>	(E fromElement) : SortedSet<E>
<b>subSet</b>	(E fromElement, E toElement) : SortedSet<E>
<b>comparator</b>	() : Comparator<? super E>

# Set – Implementierungen



- ▶ `TreeSet<E>` (`Comparator<E> c`) erzeugt eine sortierte Menge, in dem Elemente gemäß `c` sortiert sind.



# Set – Laufzeiten

## Laufzeiten:

	<i>add</i>	<i>contains</i>	<i>next</i>
HashSet	$O(1)$	$O(1)$	$O(h/n)$
LinkedHashSet	$O(1)$	$O(1)$	$O(1)$
TreeSet	$O(\log n)$	$O(\log n)$	$O(\log n)$

$h$  ist die Anzahl der Buckets in der HashSet-Implementierung.

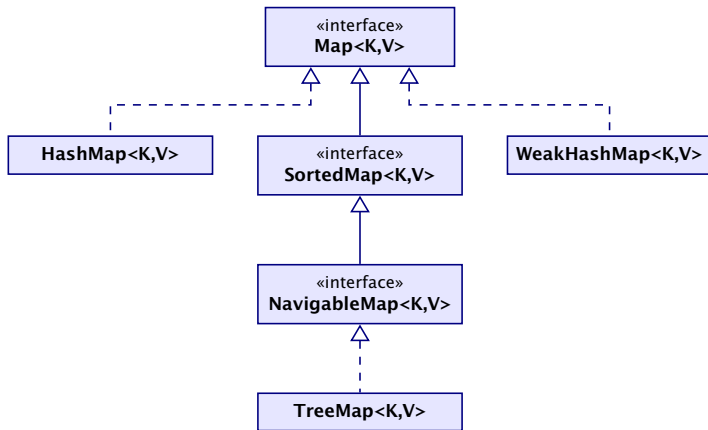
Die Laufzeiten für die hashbasierten Verfahren setzen eine gute `hashCode()`-Funktion voraus.

# Map Interface

«interface» <b>Map&lt;K,V&gt;</b>	
put	(K key, V value) : V «optional»
putAll	(Map<? extends K,? extends V> m) «optional»
remove	(Object o) : V «optional»
clear	() «optional»
containsKey	(Object o) : boolean
containsValue	(Object o) : boolean
isEmpty	() : boolean
size	() : int
get	(K key) : V
entrySet	() : Set<Map.Entry<K,V>>
keySet	() : Set<K>
values	() : Collection<V>

- ▶ Schlüssel werden auf Werte abgebildet;
- ▶ kein Schlüssel kommt doppelt vor; Werte eventuell schon
- ▶ in einer **SortedMap** sind die Schlüssel sortiert

# Map – Implementierungen



# Map – Laufzeiten

## Laufzeiten:

	<i>get</i>	<i>containsKey</i>	<i>next</i>
HashMap	$O(1)$	$O(1)$	$O(h/n)$
WeakHashMap	$O(1)$	$O(1)$	$O(h/n)$
LinkedHashMap	$O(1)$	$O(1)$	$O(1)$
TreeMap	$O(\log n)$	$O(\log n)$	$O(\log n)$

$h$  ist die Anzahl der Buckets in der HashSet-Implementierung.

Die Laufzeiten für die hashbasierten Verfahren setzen eine gute `hashCode()`-Funktion voraus.

## Collections – Tricks

- ▶ Löschen, der Elemente `[100,...,200)` in einer Liste:  
`list.subList(100,200).clear();`
- ▶ Entfernen von Duplikaten aus einer `Collection`:  
`Collection<Type> c = ...;`  
`Collection<Type> noDups = new HashSet<Type>(c);`
- ▶ Löschen aller Strings, die mit `'f'` anfangen:  
`SortedSet<String> dictionary = ...;`  
`dictionary.subSet("f", "g").clear();`

## Arrays - asList

Die Funktion `asList(T[] a)` gibt ein Listenvue auf ein Array zurück. Damit können Funktionen, die eine `Collection` erwarten auch mit einem Array aufgerufen werden:

```
1 Integer[] arr = {3,7,12,-5};
2 Collections.sort(Arrays.asList(arr));
3 for (int i=0; i<arr.length; i++) {
4     System.out.println(arr[i]);
5 }
```

- ▶ die resultierende Liste ist in der Größe fixiert; `remove()`, `add()` werfen Exceptions;
- ▶ mit `set()` kann man die Liste aber verändern.

# Eigene Kollektionen

- ▶ Zum Erstellen eigener Kollektionen erbt man von abstrakten Klassen, die einen Großteil der Implementierung bereitstellen (z.B. `removeAll()` über wiederholten Aufruf von `remove()` etc.)
- ▶ Dann werden nur einige Funktionen implementiert.
- ▶ Für zusätzliche Effizienz können auch weitere Funktionen überschrieben werden.

# Eigene Kollektionen – Übersicht

- ▶ `AbstractCollection` benötigt `iterator` und `size`.
- ▶ `AbstractSet` benötigt `iterator` und `size`.
- ▶ `AbstractList` benötigt `get` und `size` und (optional) `set`, `remove`, `add`.
- ▶ `AbstractSequentialList` benötigt `listIterator` und `size`.
- ▶ `AbstractMap` benötigt `entrySet (View)`; (optional) `put` falls veränderbar



## Beispiel - asList

### Idee:

- ▶ Speichere Arrayreferenz in Attribut lokaler Klasse.
- ▶ Übersetze Listenbefehle in entsprechende Arraybefehle.
- ▶ typisch für Adapterklassen

```
1 public static <T> List<T> asList(T[] a) {
2     return new MyArrayList<T>(a);
3 }
4 private static class MyArrayList<T> extends
5     AbstractList<T> {
6     private final T[] a;
7     MyArrayList(T[] array) {
8         a = array;
9     }
```

## Beispiel – asList

```
10     public T get(int index) {
11         return a[index];
12     }
13     public T set(int index, T element) {
14         T oldValue = a[index];
15         a[index] = element;
16         return oldValue;
17     }
18     public int size() {
19         return a.length;
20     }
21 }
```