23. Collections

ARINKO

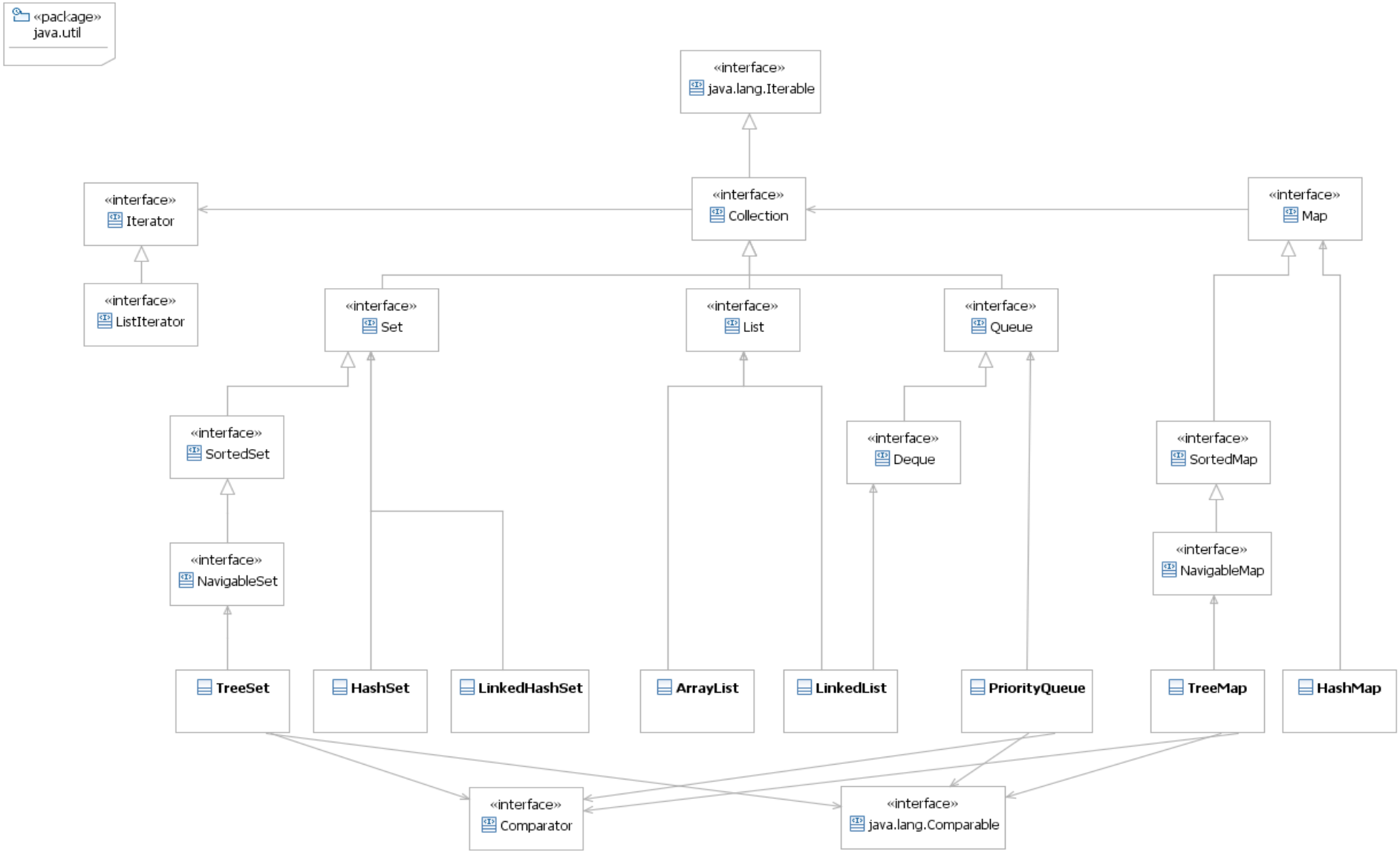
- Ein "Behälter" (Container) ist ein Objekt, das in der Lage ist, viele andere (gleichartige) Objekte aufzunehmen.
- So gesehen können Arrays als Behälter fungieren.
- Arrays haben allerdings gewisse Nachteile:
  - sie können weder wachsen noch schrumpfen
  - das bedeutet indirekt auch, dass man von vornherein wissen muss, wie viele Objekte man darin aufzubewahren denkt
  - und sie haben keine Methoden, d.h. alles Verhalten muss vom benutzenden Code kommen (das ist sehr sehr wenig objektorientiert)

- Deshalb gibt es in den meisten objektorientierten Sprachen spezielle Behälterklassen, die Aufbewahrung von Objekten komfortabler umsetzen.
- In Java nennt man diese Klassen pauschal **Collections**.
- Sie sind (seit Java 1.2) in Form des Java Collections-Frameworks implementiert.
- Das Collections-Framework findet sich im Paket `java.util`.

- Im Collection-Framework gibt es zwei grundsätzliche Collection-Typen, die jeweils nach ihren obersten Interfaces benannt sind:
  - Collections (Sammlung von Objekten)
  - Maps (Sammlung von Objekt-Paaren)
- "Collection" ist hier, analog zu Exceptions, einmal der Überbegriff und einmal der Name eines Teiles des Frameworks.

- Alle Collections sind dynamisch, d.h. sie können ihr Fassungsvermögen automatisch erweitern oder bei Bedarf schrumpfen.
- Damit Collections generell mit allen Objekten arbeiten können, ist ihr Inhaltstyp immer `Object`.
- Das bedeutet, dass jedes Objekt problemlos in eine Collection gesteckt werden kann.
- Das bedeutet aber auch, dass man beim Herausholen nur den Referenztyp `Object` erhält und nun wissen muss, um was genau es sich gehandelt hat.
- Dieses Phänomen bekommen wir erst mit dem Einsatz von Generics in den Griff (dazu später mehr).

Übersicht (ohne abstrakte Klassen)



- Iterator – Hilfsmittel zum Iterieren über eine Datenmenge
- Iterable – alles, was einen Iterator produzieren kann
- Collection – (unqualifizierte) Sammlung von Objekten
- Set – Sammlung von Objekten ohne Duplikate
- List – Sammlung von Objekten mit Reihenfolge
- Queue – Sammlung von Objekten mit Reihenfolge, die vornehmlich ganz vorne oder ganz hinten eingefügt bzw. entnommen werden (gerichtete Reihenfolge)
- Dequeue – "double ended queue" (gesprochen "deck"), Queue die von beiden Seiten befüllt und bearbeitet werden kann
- Map – Sammlung von Objektpaaren (Schlüssel + Wert)
- Sorted… - Collection, der eine Sortierung zugrunde liegt
- Navigable… - Sortierte Collection, die noch zusätzlich Ausschnitte aus Teilbereichen ermöglicht
- Comparator bzw. Comparable – Hilfsmittel zur Sortierung

- Jede Collection (nicht Map) verfügt über folgende Methoden, die bereits im Interface `java.util.Collection` deklariert sind:

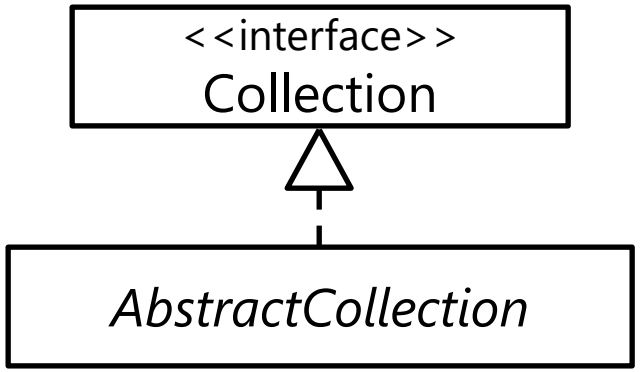
Einfügen:	`boolean add(Object o)` `boolean addAll(Collection coll)`

Löschen:	`boolean remove(Object o)`
	`boolean removeAll(Collection c)`
	`void clear()`

Abfragen:	`boolean contains(Object o)`
	`boolean containsAll(Collection coll)`
	`int size()`
	`boolean isEmpty()`

Schnittmenge:	`boolean retainAll(Collection coll)`
Iterator:	`Iterator iterator()`
Kopie in ein Array:	`Object[] toArray()` `Object[] toArray(Object[] a)`

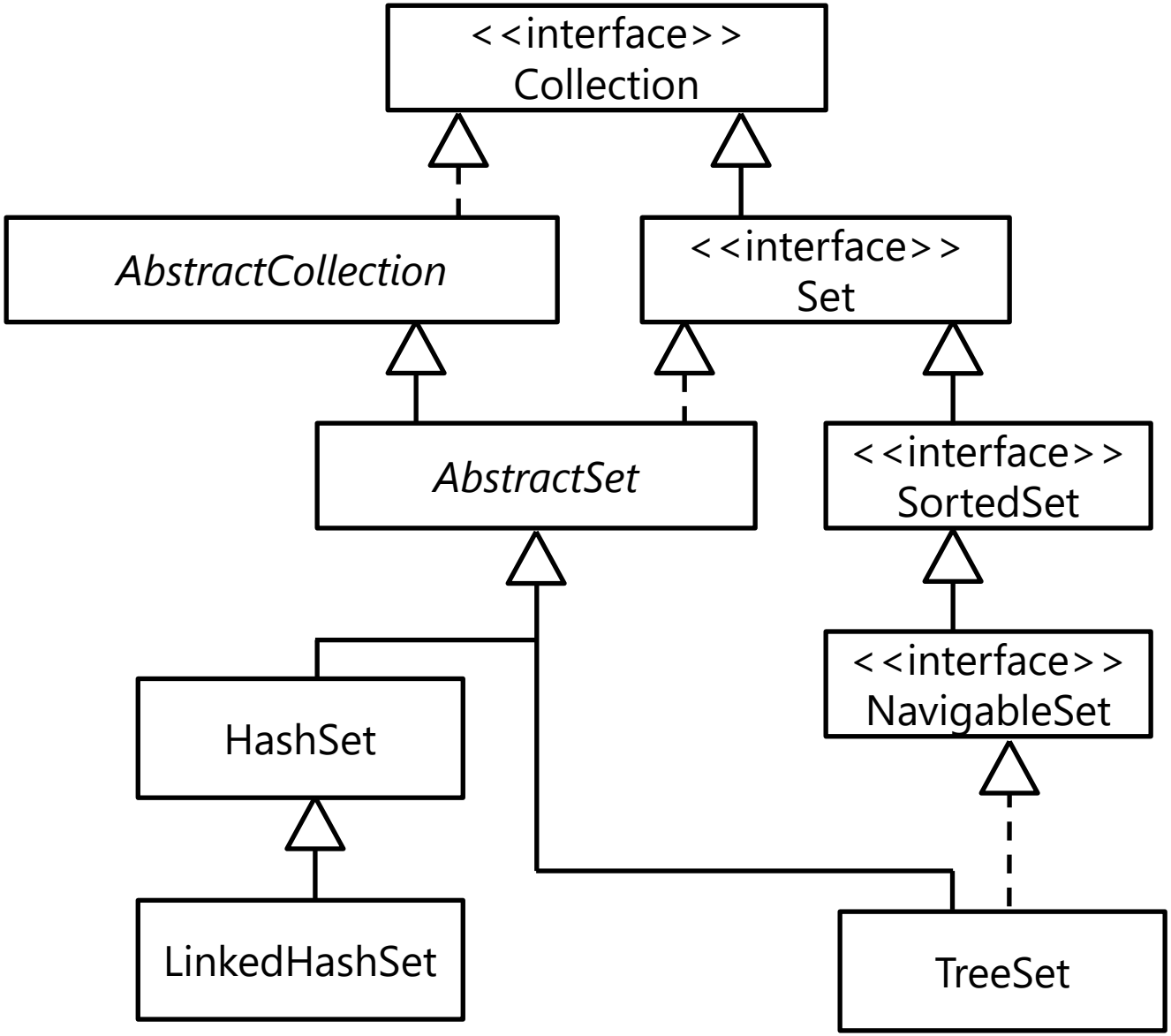
- Vom Interface `Collection` gibt es im Framework keine direkte Implementierung.
- Es existiert lediglich die abstrakte Klasse `AbstractCollection`, die gewisse Vorarbeiten für die konkreten Unterklassen, die aber weitere Interfaces implementieren, übernimmt.
- Im Open-Source-Framework CommonCollections der Apache Software Foundation findet sich aber die allgemeine Klasse `Bag` als direkte Implementierung.



- Sets sind Objektmengen.
- Sie haben gegenüber einer allgemeinen Collection die Einschränkung, dass sie keine Duplikate zulassen.
- Sets definieren keine zusätzlichen Methoden im Vergleich zum Interface Collection.
- Allerdings formuliert die Dokumentation Einschränkungen im Bezug auf Duplikate. Beispiel:

```
boolean add(Object e)
```

Adds the specified element to this set if it is not already present (optional operation). More formally, adds the specified element e to this set if the set contains no element e2 such that (`e==null ? e2==null : e.equals(e2)`). If this set already contains the element, the call leaves the set unchanged and returns false. In combination with the restriction on constructors**, this ensures that sets never contain duplicate elements**.

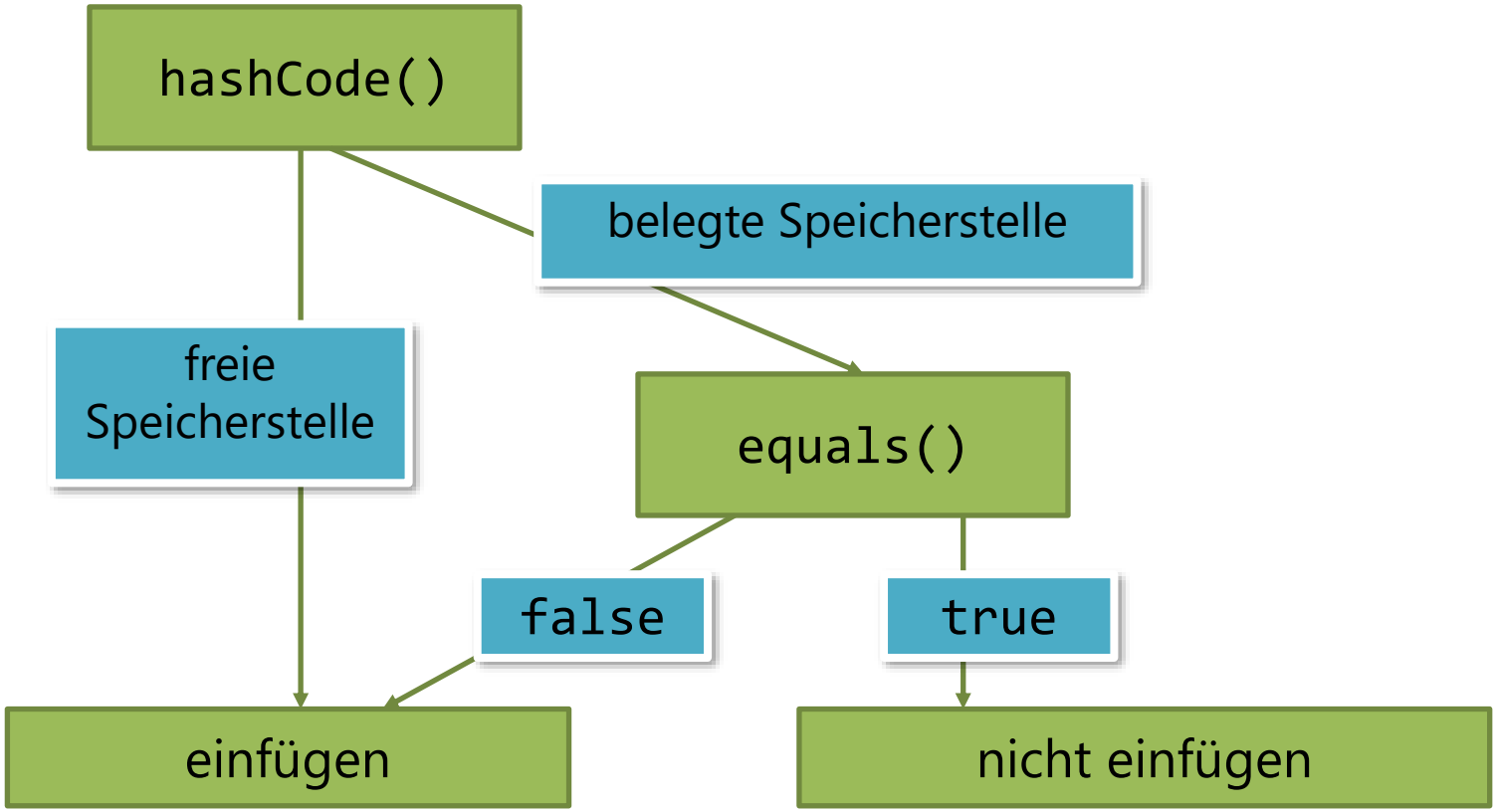

<<interface>>
Collection
AbstractCollection
<<interface>>
Set
AbstractSet
<<interface>>
SortedSet
<<interface>>
NavigableSet
HashSet
LinkedHashSet
TreeSet

- HashSets verwalten ihren Inhalt mit Hilfe einer Hashtabelle.
- Einzufügende Objekte werden mit Hilfe der von `Object` geerbten Methode `hashCode()` auf einen möglichst eindeutigen `int` Wert "reduziert", mit dem man versucht, einen geeigneten Platz in der Tabelle zu finden.
- Wenn Kollisionen auftreten, wird über `equals()` geklärt, ob es sich um einen zufällig gleichen Hashcode oder um einen absichtlich gleichen gehandelt hat.

- Bei guten Hashcode-Implementierungen sind die Operationen für Einfügen, Abfragen und Löschen in *O(1)*!
- Bei weniger guten Hashcodes oder beim Einfügen von vielen zu ähnlichen Objekten kann die Struktur "entarten" und im schlimmsten Fall ein lineares Durchsuchen nach sich ziehen, was mit *O(n)* abgeschätzt werden kann.

- Achtung: wurden `hashCode` oder `equals` nicht passend implementiert, kann das Set keine Duplikate erkennen.

- Hier nochmals zur Veranschaulichung:



- `TreeSet` arbeitet intern mit einem sortierten Binärbaum.
- Die Sortierung wird über die Sortier-Interfaces des Frameworks geregelt, dazu später mehr.
- Das `TreeSet` sortiert beim Einfügen ein, d.h. die interne Datenstruktur ist immer sortiert.
- Durch die Wahl eines Binärbaumes wird Einfügen, Entfernen und Enthaltensein mit *O(log n)* abgeschätzt.

- `LinkedHashSet` hat die Eigenschaften eines HashSets, führt aber noch eine Verzeigerung mit, um die ursprüngliche Einfügereihenfolge zu speichern.
- Performance ist minimal schlechter als bei `HashSet`, außer für die Iteration, da bei HashSets auch über zu ignorierende leere Speicherstellen iteriert wird, wohingegen `LinkedHashSet` in seiner verketteten Liste alle zu besuchenden Elemente führt.

- Oft ist die Einfügereihenfolge (nicht für den Rechner, sondern für den Menschen) wichtig. Andere Datenstrukturen bieten dies an, allerdings nicht mit der Performance eines HashSets. Ziehen Sie also LinkedHashSet immer in Betracht!

- Ein Iterator ist ein objektorientiertes Entwurfsmuster.
- Einschub: sehr grundlegende Entwurfsmuster für objektorientierte Software wurden im Buch
  "*Design Patterns: Elements of Reusable Object-Oriented Software*"
  Erich Gamma, Richard Helm, Ralph Johnson, John Vlissides – Addison-Wesley 1994
  beschrieben. Diese Muster haben heute noch Gültigkeit. Der Iterator ist eines von ihnen.
- Das Entwurfsmuster erlaubt ein Konstrukt, das es dem Programmierer möglich macht, ohne Kenntnis der genauen Implementierung einer Datenstruktur über diese zu iterieren (also in einer Schleife alle Elemente zu besuchen).
- Prinzipielle Arbeitsweise:

```
Iterator iter = collection.holeIterator();

while (iter.hatNochZuBesuchendeElemente())
{
  Element e = iter.gibDasNächsteAnstehendeElement();
  e.tuwas(); // e verarbeiten…
}
```

- Korrekt in Java:

```
HashSet hs = new HashSet();
// collection füllen...

// dann iterieren
Iterator iter = hs.iterator();
while (iter.hasNext())
{
  Object e = iter.next();
  // e verarbeiten
}
```

- Ein `Iterator` ist ein Einweg-Objekt. Wenn er durchlaufen ist, kann er nicht mehr resetiert werden.
- Die enhanced for loop von Java 5 kann auch mit Objekten arbeiten, die Iteratoren produzieren (diese sind dann `Iterable`):

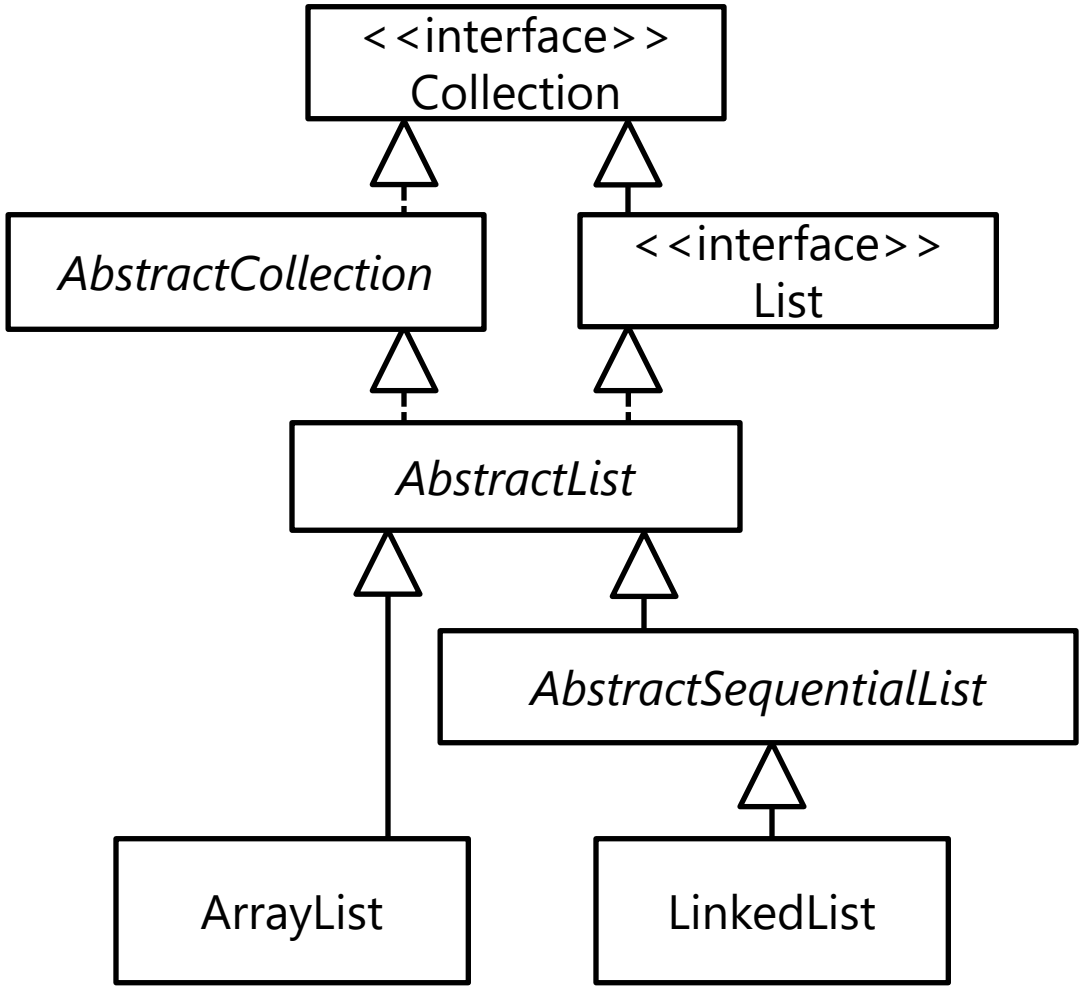
```
for (Object object : hs)
{
  // object verarbeiten
}
```

Typ von hs muss `Iterable` sein (HashSet ist das)

- Listen beachten Duplikate nicht.
- Sie konzentrieren sich auf die Reihenfolge (Achtung: damit ist nicht unbedingt Sortierung gemeint!).
- Listen sind durchnummeriert/indiziert, die Zählung beginnt analog zu Arrays bei 0.
- Listen haben alle Methoden von Collections zuzüglich einiger, die bequemer mit einem zusätzlichen Index aufrufbar sind.
- Beispiel:
  `add(Object obj)` – fügt am Ende der Liste ein [von Collection geerbt]
  `add(int index, Object obj)` – fügt das Objekt am gegebenen Index ein, der Rest "rutscht" nach hinten [neue Methode]

- Hier tauchen nur die zusätzlich definierten Methoden auf:

An einer bestimmten Stelle einfügen:	`void add(int index, Object o)` `boolean addAll(int index, Collection c)`
Ersetzen:	`Object set(int index, Object o)`
Holen und löschen:	`Object remove(int index)`
Index abfragen:	`int indexOf(Object o)` `int lastIndexOf(Object o)`
Holen:	`Object get(int index)` `List subList(int vonIndex, int bisIndex)`
List-Iteratoren:	`ListIterator listIterator()` `ListIterator listIterator(int startIndex)`


<<interface>>
Collection
AbstractCollection
<<interface>>
List
AbstractList
AbstractSequentialList
ArrayList
LinkedList

- ArrayLists arbeiten intern – voilà – mit einem Array.
- Dieses wird aber automatisch von der ArrayList bearbeitet, kann also "wachsen" (nicht wirklich, es wird umkopiert)
- Zugriffe mit get/set arbeiten mit *O(1)*.
- Hinzufügen, Löschen, Enthaltensein und einige andere im schlimmsten Fall *O(n)*.
- ArrayLists sind dann interessant, wenn man bestehende Listen manipulieren und abfragen will (viel get/set oder einfache Iteration).
- ArrayLists haben eine initiale Kapazität, die man auch per Konstruktor bestimmen kann.
- Wenn die initiale Kapazität gut gewählt wurde, werden bei großen Datenmengen weniger interne Umkopieraktionen benötigt (im besten Fall gar keine!) als bei schlecht gewählter Kapazität.
- Die default Kapazität ist 10.

- LinkedLists sind verkettete Listen.
- Einfüge- und Löschoperationen arbeiten sehr günstig, da nur Verkettungen (Referenzen) umgeleitet werden müssen: *O(n),* aber deutlich günstiger im Hinblick auf Speicherplatz.
- Besonders günstig sind Einfügeoperationen an beiden Enden: *O(1)*.
- Sehr empfehlenswert, wenn oft von einer Seite eingefügt werden soll (Stack- bzw. Queue-Verhalten).

- Seit Java 5 im Collection-Framework zu finden.
- Queues werden gerne bei der asynchronen Verarbeitung eingesetzt.
- Daher finden sich einige (hier nicht gezeigte) Implementierungen auch im Paket `java.util.concurrent`.
- Die Methoden von Queues kann man in zwei Kategorien einteilen:

1. "wehrhafte" – erzeugen Exceptions, wenn die Queue gefüllt ist oder nichts zum Löschen vorhanden ist
2. "gutmütige" – diese liefern in Extremsituationen spezielle Werte wie `null` oder `false`.

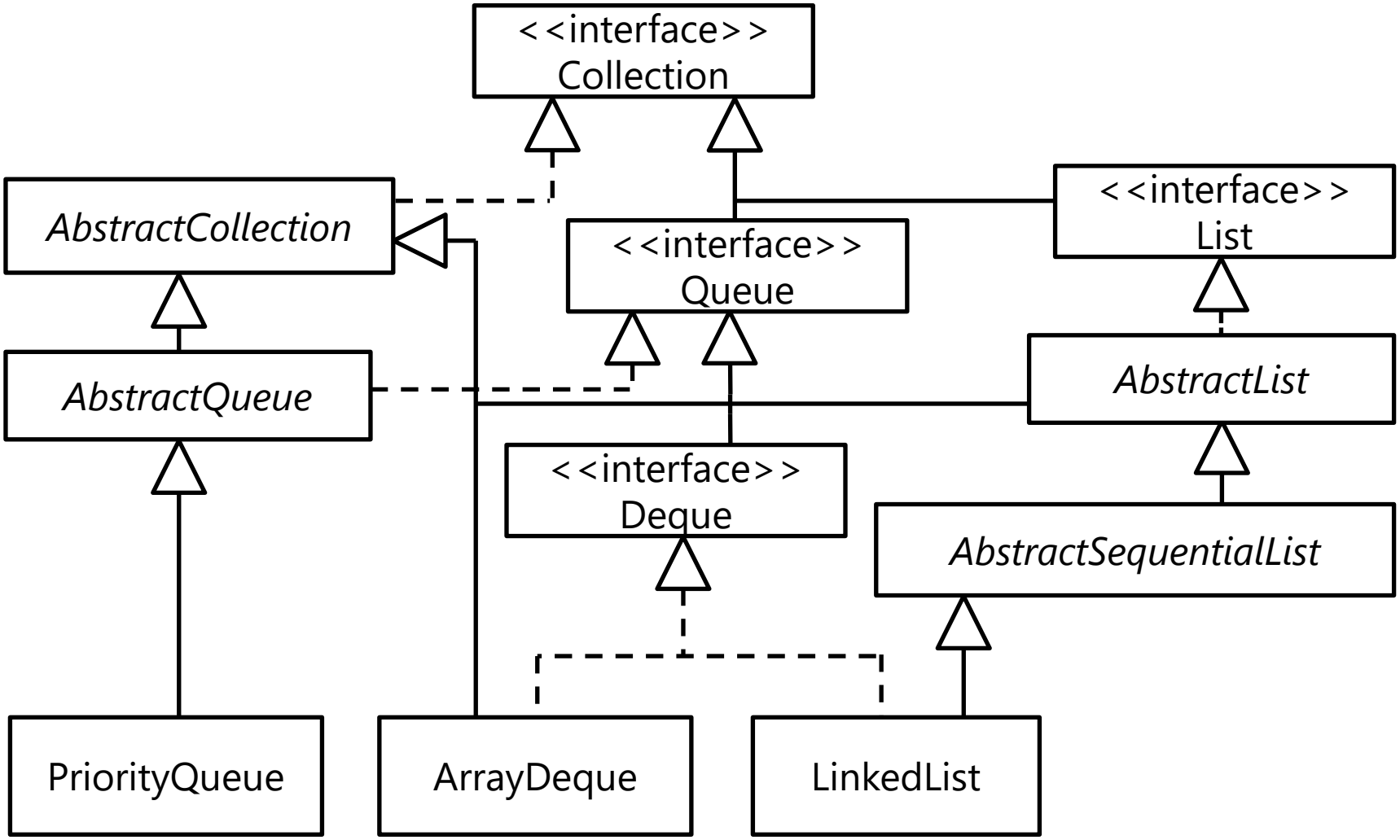
- Queues sind prinzipiell in eine Richtung gerichtet. Man fügt also immer "vorne" oder "oben" ein.

- Deque ("double ended queue", sprich "deck") sind Queues, die man an beiden Enden verwenden kann.
- Sämtliche Queue-Methoden sind nun zweifach vorhanden, zu einer `xFirst()`-Methode existiert immer eine `xLast()`.
- Deques, die über die Queue-Methoden benützt werden, zeigen FIFO-Verhalten (First In, First Out)

Queue	"wehrhaft"	"gutmütig"
Einfügen	`add(obj)`	`offer(obj)`
Entfernen	`remove()`	`poll()`
Abfragen	`element()`	`peek()`

Deque	vorne		hinten	
	"wehrhaft"	"gutmütig"	"wehrhaft"	"gutmütig"
Einfügen	`addFirst(e)`	`offerFirst(e)`	`addLast(e)`	`offerLast(e)`
Entfernen	`removeFirst()`	`pollFirst()`	`removeLast()`	`pollLast()`
Abfragen	`getFirst()`	`peekFirst()`	`getLast()`	`peekLast()`

Konkrete Queue-Implementierungen



- Zu `LinkedList`, die ebenfalls zu `List` auch noch eine `Deque` ist, ist schon einiges gesagt.

- `PriorityQueue` ist eine Vorrangwarteschlange, die Elemente werden nach Prioritäten sortiert (wie hier anzuordnen ist, bestimmt wieder die Sortierfunktionalität).
- Enqueuing und Dequeuing-Methoden: *O(log n)*
- Löschen und Enthaltensein: *O(n)*
- Abfragen (peek/element/size): *O(1)*

- `ArrayDeque` ist intern mit einem Array implementiert.
- Die meisten Methoden laufen mit nahezu konstanter Zeit *O(1)*.
- Einige spezielle remove-Methoden sowie die Abfrage auf Enthaltensein: *O(n)*.
- Laut Dokumentation ist sie schneller als `LinkedList`, wenn es nur auf das Queue-Verhalten ankommt.

```
Queue queue = new PriorityQueue();
queue.offer("One");
queue.offer("Two");
queue.offer("Three");

while (queue.peek() != null)
{
    System.out.println(queue.poll());
}
```

```
One
Three
Two
```

```
Queue queue = new ArrayDeque();
queue.offer("One");
queue.offer("Two");
queue.offer("Three");

while (queue.peek() != null)
{
    System.out.println(queue.poll());
}
```

```
One
Two
Three
```

```
Deque deque = new ArrayDeque();
deque.offer("One");
deque.offer("Two");
deque.offer("Three");

while (deque.peekLast() != null)
{
  System.out.println(deque.pollLast());
}
```

```
Three
Two
One
```

```
Deque deque = new ArrayDeque();
deque.addFirst("One");
deque.addFirst("Two");
deque.addFirst("Three");

while (true)
{
  System.out.println(deque.removeFirst());
}
```

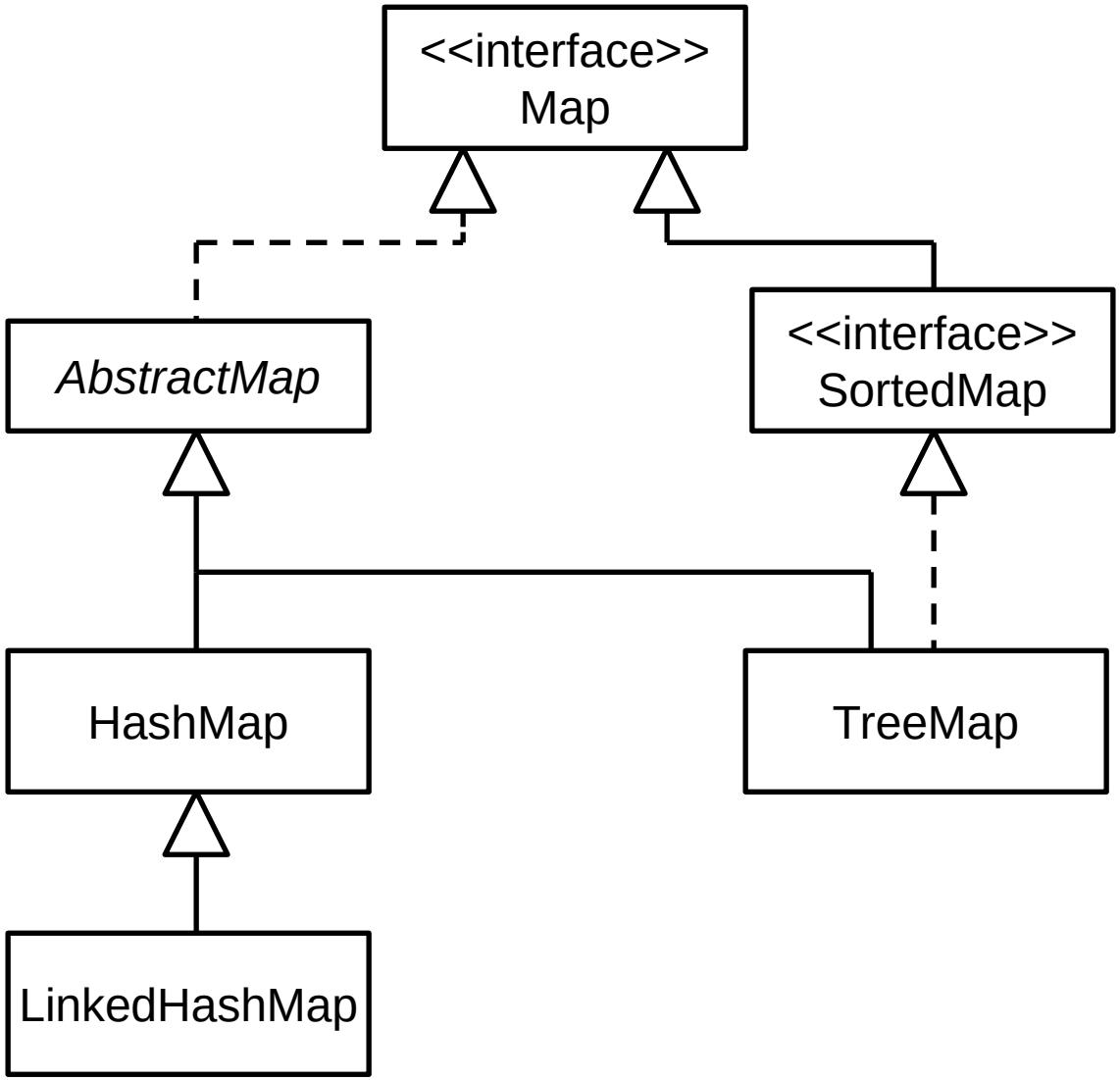
```
Three
Two
One
NoSuchElementException
```

- Maps sind zweiwertige Collections, sie bestehen immer aus einem Schlüsselobjekt (key) und einem zugeordneten Wertobjekt (value).
- Die Schlüssel müssen eindeutig sein und es kann keine 2 gleichen Schlüssel in einer Map geben.
- Wertobjekte dürfen beliebig oft vorkommen.
- Andere Programmierplattformen bezeichnen Maps beispielsweise als "assoziatives Array" oder "Dictionary".
- Maps stehen in der Hierarchie parallel zu Collection, d.h. sie haben/erben nicht die gleichen Methoden, sondern definieren einen (recht ähnlichen) eigenen Satz.

- Hier einige der wichtigsten Methoden (Map von Java 8 kennt noch ein paar mehr):

Einfügen:	`Object put(Object key, Object value)` `putAll(Map m)`
Löschen:	`Object remove(Object key)`
Leeren:	`clear()`
Direktzugriff mit Schlüssel:	`Object get(Object key)`
Enthaltene Elemente:	`boolean containsKey(Object key)` `boolean containsValue(Object o)`
Größe und Inhalt:	`int size()` `boolean isEmpty()`
Elementmengen:	`Collection values()` `Set keySet()`

Konkrete Map-Implementierungen



- Analog zu den Set-Varianten sind auch die entsprechenden Maps aufgebaut.

- `HashMap`: verwaltet die Schlüssel in einer Hash-Tabelle.
- Gleichheit der Schlüssel analog HashSet mittels `hashCode` und `equals`.

- `TreeMap`: nach Schlüsseln (!) sortierter Binärbaum.

- `LinkedHashMap`: wie `HashMap`, nur mit Bewahrung der Einfügereihenfolge.

- Für die Komplexitätsmaße gilt das gleiche wie für Sets gesagte.
- Die volle Wahrheit ist, dass die Sets die Implementierungen der Maps verwenden und dabei nur für Schlüssel und Wert immer das selbe Objekt eintragen.
- Aus der Klasse `HashSet`:

```
public HashSet()
{
        map = new HashMap();
}
public boolean contains(Object o)
{
       return map.containsKey(o);
}
```

```
Map m = new HashMap();

m.put("eins", 1);
…
m.put("sechs", 6);

for (Object key : m.keySet())
{
  System.out.println(key + ":" + m.get(key));
}
```

```
eins:1
vier:4
zwei:2
fünf:5
drei:3
sechs:6
```

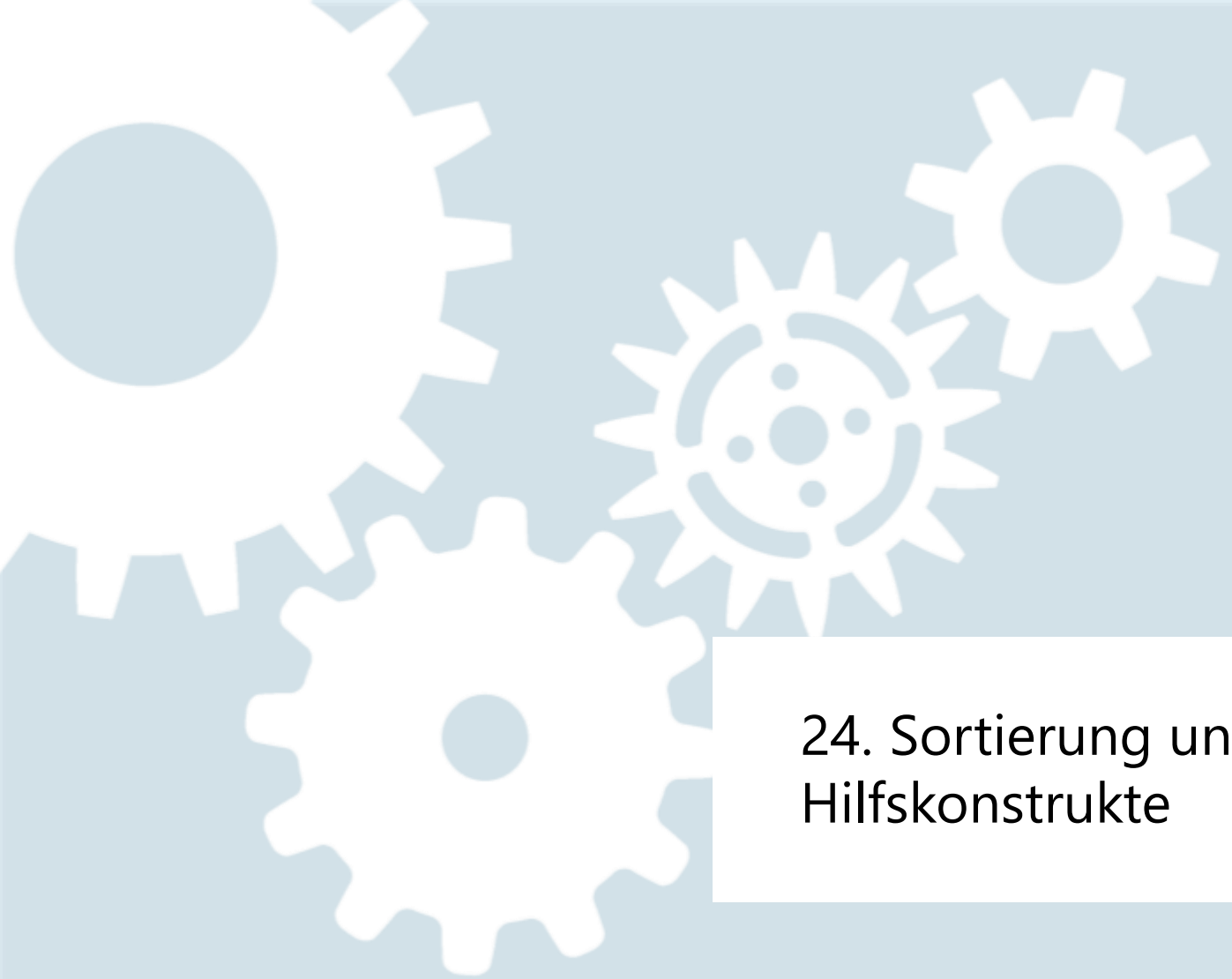
```
Map m = new TreeMap();
```

```
drei:3
eins:1
fünf:5
sechs:6
vier:4
zwei:2
```

```
Map m = new LinkedHashMap();
```

```
eins:1
zwei:2
drei:3
vier:4
fünf:5
sechs:6
```

- In Java 1.0 und 1.1 existierte das Collection-Framework noch nicht.
- Trotzdem gab es bereits Klassen, die ähnliches Verhalten implementierten.
- Leider trifft man immer wieder in Literatur und in Internetrecherche diese veralteten Klassen an – best practice ist, sie nicht zu verwenden.
- Die Klassen wurden ins Collection-Framework eingebaut, haben jetzt aber eine z.T. deutlich vergrößerte Schnittstelle, weil viele Methoden nun doppelt sind.
- Es handelt sich um diese Typen:
  - `Vector` (entspricht einer `ArrayList`)
  - `Hashtable` (entspricht einer `HashMap`)
  - `Dictionary` (abstrakte Oberklasse von `Hashtable`)
  - `Stack` (entspricht einer LIFO-Queue)
  - `Enumeration` (enspricht `Iterator`)



24. Sortierung und andere Hilfskonstrukte

ARINKO

- Sollen Objekte in einer Collection sortiert werden, benützt das Collection-Framework immer einen der zwei Typen

  java.lang.Comparable
  java.util.Comparator

- Die gute Nachricht: man muss in Java keinen Sortieralgorithmus schreiben, der ist bereits in der Plattform integriert.
- Java verwendete bis 1.6 MergeSort, seit 1.7 TimSort.
- Diese internen Sortierverfahren sind stabil und ungefähr in *O(n log n)* [worst case] bzw. *O(n)* [best case]

- Was diese Sortierroutinen aber brauchen, ist eine Entscheidung, wann umsortiert werden soll.
- Hierfür kommen die Interfaces ins Spiel.

- Beide Interfaces haben dieselbe Grundausrichtung: die darin enthaltene Methode soll bestimmen, wie zwei Objekte in einer Sortierung zueinander stehen.
- Dabei gilt für beide als Rückgabewert der Methode
  - Integer-Zahl < 0: das "Pivot"-Objekt ist kleiner als das andere
  - Integer-Zahl 0: beide Objekte sind gleich
  - Integer-Zahl > 0: das "Pivot-Objekt" ist größer als das andere.
- Die Interfaces unterscheiden sich nur darin, wo sie genau vorkommen und woher sie ihre Objekte beziehen.

- `Comparable` ist die *natürliche Sortierung* eines Fachobjektes. Das Pivot-Objekt ist das aktuelle Objekt (`this`). Das zu vergleichende wird von außen übergeben.
- `Comparator` ist ein externer Vergleicher. Beide Objekte werden von außen übergeben, das Pivot-Element ist das links in der Parameterliste stehende.

- Wird sortierenden Methoden bzw. Collections kein `Comparator` übergeben, versuchen sie die Objekte als `Comparable` zu sortieren.
- Wenn die in der Collection befindlichen Objekte dieses Interface dann nicht implementieren, kommt es zu einer `ClassCastException`.

- Die Interfaces sehen (grob) wie folgt aus. Die reale Deklaration verwendet noch Generics (dazu später mehr):

```
public interface Comparable
{
  public int compareTo(Object other);
}
```

```
public interface Comparator
{
  int compare(Object o1, Object o2);
}
```

```
Konto konto1 = new Konto("DE68-0816-4711", 250);
Konto konto2 = new Konto("DE68-0815-4712", 900);
Konto konto3 = new Konto("EN68-0815-4711", 102);
Konto konto4 = new Konto("DE69-0815-4711", 400);
Konto konto5 = new Konto("DE68-0815-4711", 848);

Set konten = new TreeSet(); // hier greift Comparable
konten.add(konto1);
konten.add(konto2);
konten.add(konto3);
konten.add(konto4);
konten.add(konto5);

for (Object kontoObj : konten)
{
  System.out.println(kontoObj);
}
```

```
Konto DE68-0815-4711, Stand: 848 EUR
Konto DE68-0815-4712, Stand: 900 EUR
Konto DE68-0816-4711, Stand: 250 EUR
Konto DE69-0815-4711, Stand: 400 EUR
Konto EN68-0815-4711, Stand: 102 EUR
```

```
public class Konto implements Comparable
{
  @Override
  public int compareTo(Object o)
  {
    if (o instanceof Konto)
    {
      Konto that = (Konto) o;
      return this.kontonummer.compareTo(that.kontonummer);
    }
    return 0;
  }

  …
}
```

```
Konto konto1 = new Konto("DE68-0816-4711", 250);
Konto konto2 = new Konto("DE68-0815-4712", 900);
Konto konto3 = new Konto("EN68-0815-4711", 102);
Konto konto4 = new Konto("DE69-0815-4711", 400);
Konto konto5 = new Konto("DE68-0815-4711", 848);

Set konten = new TreeSet(new KontostandComparator());
konten.add(konto1);
konten.add(konto2);
konten.add(konto3);
konten.add(konto4);
konten.add(konto5);

for (Object kontoObj : konten)
{
  System.out.println(kontoObj);
}
```

```
Konto EN68-0815-4711, Stand: 102 EUR
Konto DE68-0816-4711, Stand: 250 EUR
Konto DE69-0815-4711, Stand: 400 EUR
Konto DE68-0815-4711, Stand: 848 EUR
Konto DE68-0815-4712, Stand: 900 EUR
```

```
public class KontostandComparator implements Comparator
{
  public int compare(Object o1, Object o2)
  {
    if (o1 instanceof Konto && o2 instanceof Konto)
    {
       Konto k1 = (Konto) o1;
       Konto k2 = (Konto) o2;
       // einfache Implementierung mit Hilfe von Integer
       return Integer.compare(k1.getKontostand(), k2.getKontostand());
       // Alternativ, selbstgemacht
       int stand1 = k1.getKontostand();
       int stand2 = k2.getKontostand();
       if (stand1 < stand2)
         return -1;
       else
         if (stand1 > stand2)
           return +1;
       …
       // Alternativ, etwas trickreicher
       return (stand1-stand2);
    }
  }
}
```

- Sets können sich in fachlichen Sortierungen oft als hinderlich erweisen.
- WARUM?
- Alternativ kann man Listen-Implementierungen verwenden. Es gibt allerdings keine SortedList. Listen müssen immer "von außen" (um-)sortiert werden.

- `Comparator` hat einige statische und default Interfacemethoden, die recht nützlich sein können, u.a.:
  `default Comparator reversed()`
  liefert einen Comparator, der die umgekehrte Sortierung durchführt

  `default Comparator thenComparing(Comparator other)`
  liefert einen Comparator, der mehrstufig ist

  `public static Comparator naturalOrder()`
  liefert einen Comparator, der die natürliche Ordnung implementiert

  `public static Comparator reverseOrder()`
  liefert einen Comparator, der die umgekehrte natürliche Ordnung implementiert

  `public static Comparator nullsFirst(Comparator comparator)`
  liefert einen Comparator, der null-Werte an den Anfang stellt

- Für die Arbeit mit Collections (und Arrays) existieren im Collection-Framework noch 2 Hilfsklassen, die allerlei statische Methoden anbieten.

```
java.util.Collections
java.util.Arrays
```

- Beide bieten Methoden zum Durchsuchen, Sortieren, Füllen u.v.m.
- Mit Hilfe der Klasse Arrays kann man beispielsweise aus Arrays Collections (Listen) erzeugen.
- Und mit Hilfe der Klasse Collections kann man Listen extern sortieren.

- Hier ein Auszug wichtiger (allesamt statischer) Methoden:

`Collection checkedCollection( Collection c, Class type)` analog checkedMap, checkedList…	Erzeugt eine zur Laufzeit typsichere Collection, basierend auf der Typangabe.
`copy(List dest, List src)`	Kopiert eine Liste in eine andere.
`Enumeration enumeration( Collection c)`	Erzeugt eine (Altlast) Enumeration aus einer Collection. Nützlich für Altcode-Aufrufe.
`ArrayList list(Enumeration e)`	Anderer Fall: Eine aus Altcode stammende Enumeration in eine Liste wandeln.
`fill(List l, Object o)`	Füllt eine Liste auf.
`List nCopies(int n, Object o)`	Erzeugt eine Liste mit n-mal Objekt o drin.
`reverse(List list)`	Dreht eine Liste um
`shuffle(List list)`	Permutiert (de-sortiert) eine Liste. Gut für Glücksspiele…
`rotate(List list, int d)`	Rotiert eine Liste d Elemente weit

- weitere Methoden:

`Set singleton(Object o)` und singletonList etc.	Erzeugt eine Collection, die nur ein Element enthält und unveränderlich ist.
`sort(List list)`	Sortiert die Liste nach der natürlichen Ordnung.
`sort(List list, Comparator c)`	Sortiert die Liste mit Hilfe des gegebenen Comparators.
`X synchronizedX(X x)` Collection, Set, Map, List…	Erzeugt eine synchronisierte (aka thread-safe) Variante einer gegebenen Collection.
`int binarySearch(List list,` `  Object o, Comparator c)`	Sucht in einer aufsteigend sortierten Liste nach dem Index eines gegebenen Elements
`boolean disjoint(` `  Collection c1, Collection c2)`	Überprüft, ob in 2 gegebenen Collections keine gemeinsamen Elemente sind.
`X emptyX()`	Erzeugt eine leere und unveränderliche Liste, Map, Set, Iterator, Enumeration etc.

- In den gezeigten Methoden sind schon ein paar zu sehen gewesen, die spezielle (leere, einelementige) unveränderliche Collections liefern.
- Dies kann man auch gezielt für eigene, gefüllte Collections erreichen:

```
Collection unmodifiableCollection(Collection coll)
List unmodifiableList(List list)
Set unmodifiableSet(Set set)
```

  etc.
- Das ist insbesondere dann interessant, wenn Sie intern Daten in Collections halten und diese dann als Rückgabewert nach außen geben. Wer schützt ihre interne Collection davor, dass der außenstehende Code dort Eintragungen oder Löschungen macht?
- Hier gibt es in der Praxis nur 2 Ansätze:

1. Man erzeugt Kopien und gibt diese zurück ("teuer")
2. Man umhüllt (wrapped) die Collection mit einer Schutzschicht, die sie unveränderlich macht ("kostengünstig") – das sind die unmodifiable Collections.

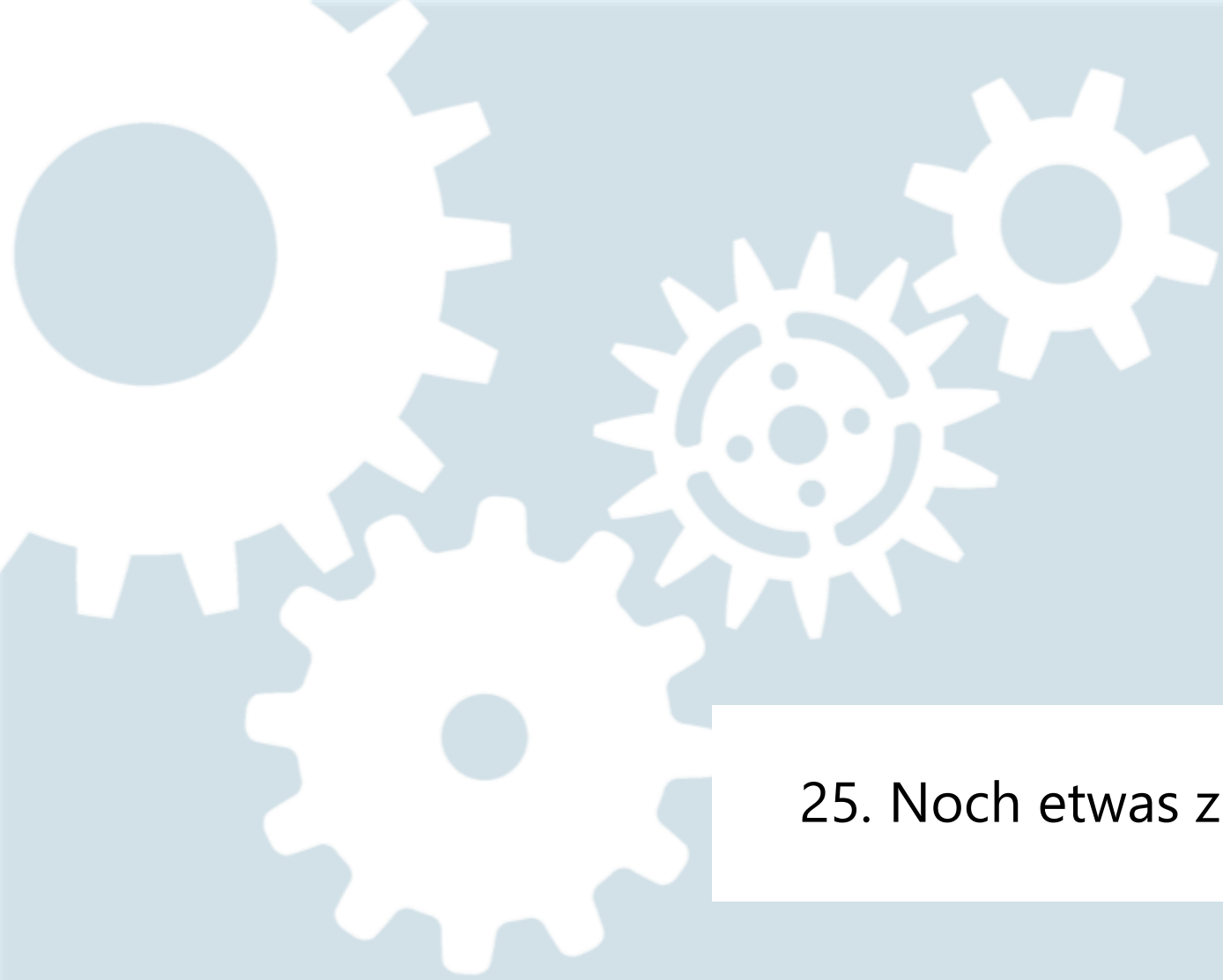
- Hier ein Auszug wichtiger (allesamt statischer) Methoden der Klasse Arrays:

`List asList(Object[] array)`	Wandelt ein Array in eine Liste um.
`int binarySearch(…)`	Div. Suchmethoden, um in Arrays einen Trefferindex zu finden.
`typ[] copyOf(typ[], int l)`	Erzeugt eine (kleinere oder größere) Kopie eines gegebenen Arrays. Wenn größer, dann wird mit Nullwerten aufgefüllt.
`boolean equals(typ[] a, typ[] b)`	Gibt true zurück, wenn beide Arrays exakt die selbe Belegung haben.
`boolean deepEquals(Object[] a, Object[] b)`	Gibt true zurück, wenn beide Arrays exakt die selbe Belegung inkl. Subarrays haben.
`fill(typ[] a, typ v)`	Füllt ein Array mit dem angegebenen Wert.
`sort(typ[] array)` `parallelSort(typ[] array)`	Sortiert ein Array (ggf. mit einem parallelen Alogrithmus) nach der natürlichen Ordnung
`sort(typ[] arr, Comparator c)` `parallelSort(…)`	Sortiert mit Hilfe des angegebenen Comparators
`String toString(typ[] array)`	Erzeugt eine String-Darstellung des Arrays.

- Im Collection-Framework wurden zwei Klassen, die eng mit Enums arbeiten, ergänzt: EnumSet und EnumMap.
- EnumSet besitzt einige nützliche statische Methoden für den Umgang mit (Teil-) Mengen von Enums.
- EnumMap ist typsicher bezüglich der Schlüssel. Dieses Feature wird allerdings auch schon von den ebenfalls mit Java 5 eingeführten Generics abgedeckt. Allerdings hat die EnumMap noch einen Vorteil: ihr Füllvermögen entspricht immer genau der Anzahl an möglichen Enums, wohingegen eine allgemeine Map u.U. eine zu große Kapazität vorhält.

- Beispiel für das EnumSet:

```
EnumSet<Muenze> muenzSet = EnumSet.of(Muenze.EIN_CENT, Muenze.ZWEI_EURO);
muenzSet = EnumSet.complementOf(muenzSet);
// jetzt sind alle Münzen außer 1 Cent und 2 Euro enthalten
```



25. Noch etwas zu Arrays

ARINKO

- Zu Arrays ist schon einiges gesagt.
- Hier nochmals die wichtigsten Nachteile gegenüber Collections:

1. Sie haben keine Methoden. Alles muss im steuernden Code außerhalb erfolgen.
2. Sie sind nicht dynamisch.
3. Manche Abfragen (Enthaltensein, leeres Array) sind deutlich teurer als bei der Wahl einer geeigneten Datenstruktur.

- Es gibt aber auch einige (manche nicht zeitlosen) Vorteile:

1. Speichereffizient. Keine zusätzliche Datenstruktur wie Hash-Buckets oder verkettete Knoten als zusätzliche Speicher-"Fresser".
2. Typsicher. (Dieses Problem werden wir mit Generics hinsichtlich Collections in den Griff bekommen)
3. Sie erlauben elementare Typen (das kann man mit Hilfe von Autoboxing angehen, ist aber nicht das selbe…)
4. Sie sind Sprachbestandteil, Collections sind Plattformbestandteil.

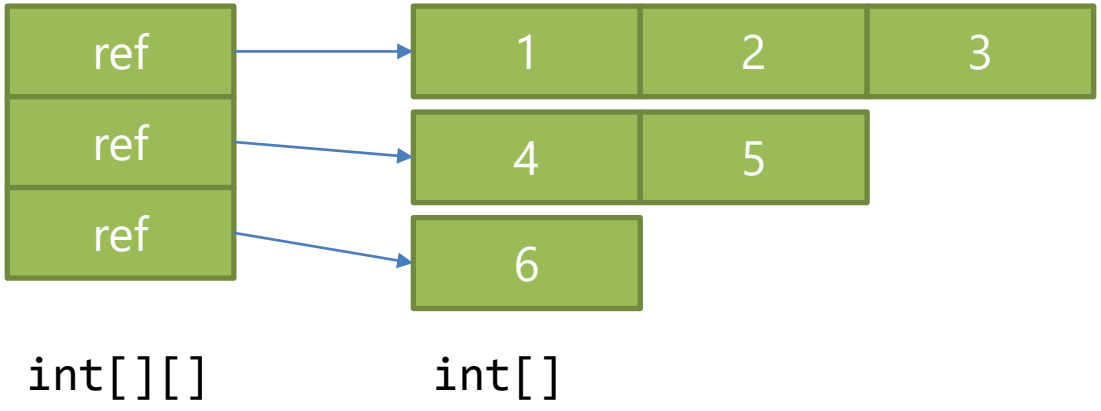
- Trotzdem: versuchen Sie mit Collections zu arbeiten, wo immer möglich.

- Arrays können wiederum Referenzen auf Arrays enthalten.
- Dies sind, exakt gesprochen, Arrays von Arrays. Lax sagt man "mehrdimensionale Arrays".
- Syntaktisch sind sie erkennbar an mehreren Sätzen von eckigen Klammern.

```
int[][] array = { {1,2,3}, {4,5}, {6} };

int x = array[2][0]; // 6, Indizierung von links (äußere Dimension) nach rechts
```

- Man erkennt, dass mehrdimensionale Arrays in Java nicht symmetrisch sein müssen!



- Das bedeutet, dass man beim Iterieren durch mehrdimensionale Arrays vorsichtig sein muss. ForEach ist beispielsweise sicher:

```
for (int[] is : array)
{
  for (int i : is)
  {
    System.out.println(i);
  }
}
```

- Das hier ist gefährlich: (wo steckt der Fehler?)

```
for (int i = 0; i < array.length; i++)
{
  for (int j = 0; j < array.length; j++)
  {
    System.out.println(array[i][j]);
  }
}
```

Es sind auch komplexe Strukturen möglich

```
Object[][][] array = new Object[][][]
        {
                new Object[1][2],
                new Object[2][3]
        };
```

```
Object[][][] array = new Object[2][4][5];
```

- Mit Java 5 ist eine etwas einfachere Behandlung für Parameter eingeführt worden, von denen man vorher noch nicht weiß, wie viele es mal werden.
- Beispiel: printf – man kann vorher nicht sagen, wie viele Parameter benötigt werden, das hängt schließlich auch von der Textschablone ab.
- Bis Java 1.4 hat man variable Argumentlisten per Überladen und/oder mit Array gestaltet. Hier ein Beispiel.

```
public static void display(int a){}
public static void display(int a1, int a2){}
public static void display(int a1, int a2, int a3){}
public static void display(int[] as){}
```

- Man findet so ein Design ab und an in der Java Plattform.
- "For convenience" wurden Methoden mit 1-3 Parameter angeboten, für n Parameter benötigt man ein Array.
- Das bürdet dem Entwickler, wenn er mehrere Parameter hat, die Erzeugung des Arrays auf:

```
display(new int[] {1,2,3,4});
```

- Variable Argumentlisten sind ebenfalls Arrays, allerdings etwas anders deklariert.
- Der Compiler erkennt daran, dass er bei Bedarf ein Array um die Parameter legen muss.

```
public static void display(int... as){}
```

- Die drei Punkte "..." werden als Ellipsen-Notation bezeichnet.
- Als Parameter kann nun direkt ein Array übergeben werden, oder Einzelemente, aus denen der Compiler ein Array formt (grau ist Compiler-Leistung):

```
display(new int[] {1,2,3,4});
```

- Die Verarbeitung findet wieder wie ein normales Array, das es auch ist, statt:

```
public static void display(int... as)
{
  for (int value : as)
  {
    System.out.println(value);
  }
}
```

- Jeder Typ kann zu einem Vararg-Array werden.
- Es kann aber in einer Parameterliste nur eine Varargdeklaration geben…
- …und diese muss der letzte Parameter sein!
- Gleichzeitig je eine Methode mit Array und mit Vararg geht nicht.

- Für das Überladen von Vararg-Methoden gilt:

1. direkt verfügbare Methode schlägt Vararg
2. Typkonvertierung schlägt Vararg
3. Boxing schlägt Vararg

- Beispiel für 1.)

```
public static void display(int... as){…}
public static void display(int a1, int a2){…}

display(1,2);
```


Praxis
Theorie