



13 Collections-Framework



SEW 3

SJ 2016/17

DI Thomas Helml



Inhalt

- ① 13.1 Grundlagen zum Collections Framework
- ① 13.2 Das Interface Collection
- ① 13.3 Mit Listen arbeiten
- ① 13.4. Listen sequentiell durchlaufen
- ① 13.5 Hash-Tabellen und Bäume
- ① 13.6 Sets - Collections vom Typ Set
- ① 13.7 Maps - Collections vom Typ Map<K, V>
- ① 13.8 Beispiel



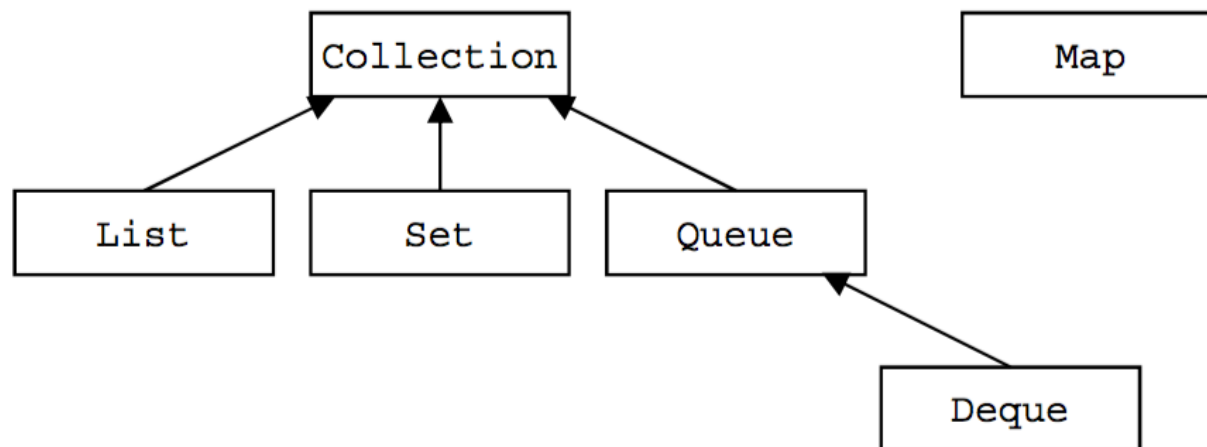
13.1. Was sind Collections?

- ④ ***Collections*** (=Container) ***sind Datenstrukturen***, die eine Gruppe von Daten zu einer Einheit zusammenfassen
- ④ ***Elemente*** = Daten der Collection
- ④ Zugriff auf Elemente mit entsprechenden Methoden
- ④ ***Beispiele:***
 - ④ Liste: Duplikate erlaubt, eventuell sortiert, Zugriff wahlweise oder sequenziell
 - ④ Menge: keine Duplikate



Interfaces-Hierarchie

- ① in `java.util`: Hierarchie von Interfaces
- ① jedes Interface wird von mehreren Klassen implementiert
- ① Klassenname beinhaltet Name des Interface





Interfaces-Hierarchie

Collection	Basis-Interface, grundlegende Methoden
List	Beliebig große Liste, auf Elemente kann sequentiell od. wahlfrei zugegriffen werden
Set	Menge von Elementen ohne Duplikate, Mengenoperationen
Queue	Warteschlange, nur sequentieller Zugriff (z.B. FIFO), kein wahlfreier Zugriff
Deque	double ended queue – Doppelt verkettete Liste, kein wahlfreier Zugriff
Map	Schlüsse-Werte-Paare



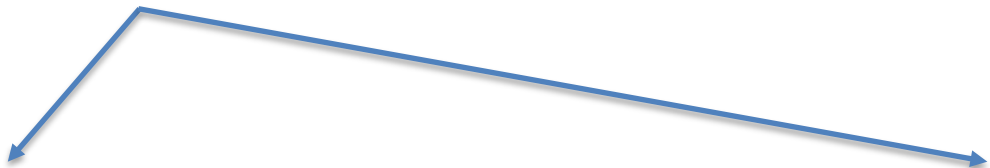
Vorteile des Collections-Frameworks

- ① Collections sind ***nicht synchronisiert!***
 - ① Achtung bei Nutzung in mehreren Threads!
 - ① ab Java8 gibt es die *Stream-API* für synchronisierten Zugriff



Einheitliche Datentypen

- ① Datenelemente müssen gleichen Typ haben = ***generischer Datentyp***



```
ArrayList<String> arrList = new ArrayList<String>();  
for (int i=1; i<=10;i++)  
    arrList.add("Obj"+i);  
  
arrList.add(new Integer(12)); // Compiler Fehler!
```



Einheitliche Datentypen

- ① Klassen, die Typargumente besitzen, nennt man ***generische Klassen***
- ① durch Angabe eines Datentyps entsteht ein ***generischer Datentyp***
- ① kann der Compiler aus dem Kontext das Typargument erkennen, kann der ***Diamond-Operator*** `<>` verwendet werden



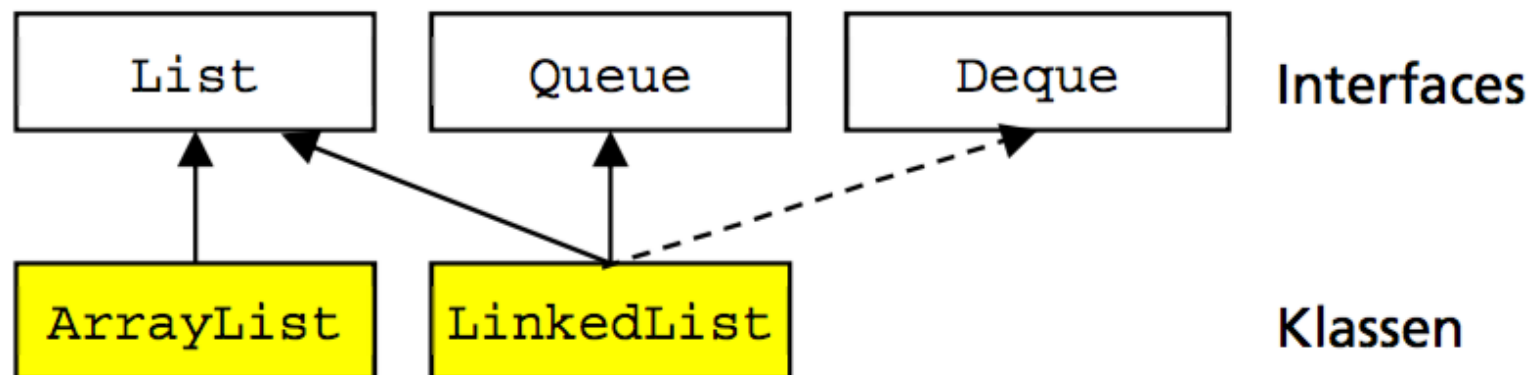
13.2. Das Interface Collection

Methoden des Basis-Interface Collection<E>

boolean add(E o)	Fügt Element der Collection hinzu
void clear()	Löscht alle Elemente der Collection
boolean contains(Object o)	Prüft, ob Element in Collection vorhanden ist
boolean equals(Object o)	Prüft, ob Object mit Collection gleich ist
int hashCode()	Hashcode der Collection
boolean isEmpty()	Prüft, ob Collection leer ist
Iterator<E> iterator()	Iterator-Objekt über Elemente der Collection
boolean remove(Object o)	Entfernt übergebenes Objekt
int size()	Anzahl der Elemente in Collection
Object[] toArray()	Alle Elemente der Collection als Array
<T> T[] toArray(T[] a)	Alle Elemente der Collection mit Laufzeittyp T

13.3 Mit Listen arbeiten

- ① **Listen** eignen sich für
 - ① **geordnete Mengen**
 - ① **wahlfreier** Zugriff (Index)
 - ① **sequentieller** Zugriff (der Reihe nach)



Eine Liste bearbeiten

- ① ***ArrayList*** implementiert eine Liste, deren Elemente linear hintereinander gespeichert werden (intern: Array)
 - ① erstes Element (analog zu Arrays): Index 0
 - ① Unterschied zu Arrays:
 - ① `ArrayList` ist dynamisch, d.h. zur Laufzeit lassen sich Elemente löschen + hinzufügen
 - ① beim Löschen/Einfügen innerhalb der Liste werden alle nachfolgenden Elemente kopiert



Eine Liste bearbeiten

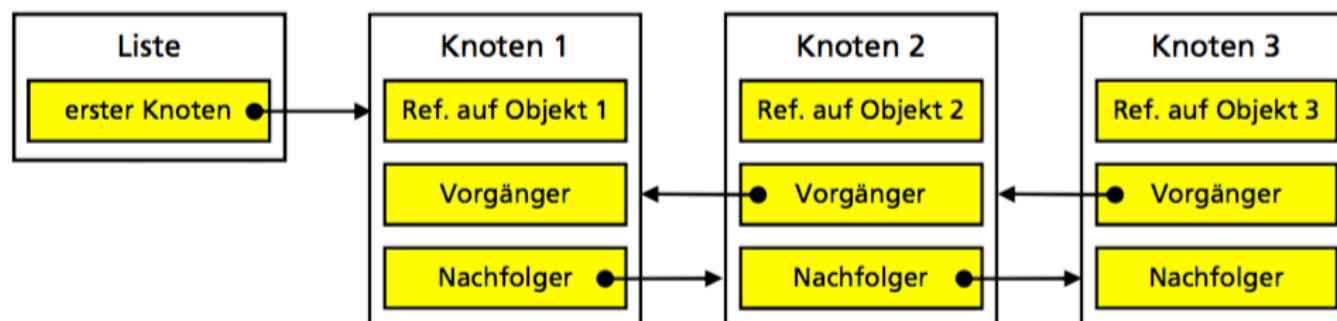
`ArrayList` implementiert ***zusätzlich*** zum Interface `Collection` folgende Methoden:

<code>void add(int index, E element)</code>	Fügt <code>element</code> an der Position <code>index</code> ein
<code>E get (int index)</code>	Gibt das Element an der Position <code>index</code> zurück
<code>int indexOf(Object o)</code>	Ermittelt den Index des <i>ersten Vorkommens</i> von <code>o</code>
<code>E set (int index, E element)</code>	Ersetzt das Element an der Position <code>index</code> durch <code>element</code>

Doppelt verkettete Listen erstellen

IT LinkedList

- IT Jedes Element wird in einem „Knoten“ gespeichert
- IT Pro Knoten: Referenz auf Element, vorigen und nächsten Knoten (doppelt verkettete Liste)
- IT Vorteil: Sehr schnell bei Einfügen/Löschen





13.4 Listen sequentiell durchlaufen

- ④ Sequenzieller Durchlauf einer Collection geschieht mittels eines ***Positionszeigers***
- ④ diesen Positionszeiger nennt man ***Iterator***
- ④ Es kann ***mehrere Iteratoren gleichzeitig*** geben
- ④ Klassen, die iterierbar sind, müssen das Interface `Iterable` implementieren
 - ④ nur eine Methode: `Iterator iterator()`
 - ④ gibt Referenz auf `Iterator`-Objekt retour
- ④ Interface `Iterator` bzw. `ListIterator` definieren
Iterator-Funktionalität



Positionszeiger vom Typ `Iterator<E>` verwenden

Interface `Iterator`

<code>boolean hasNext()</code>	prüft, ob weitere Elementie in Collection sind
<code>E next()</code>	holt aktuelles Element aus Collection und setzt Positionszeiger auf nächstes Element
<code>void remove()</code>	entfernt Element aus Collection, welches beim letzten Aufruf von next zurückgegeben wurde



Positionszeiger vom Typ `ListIterator<E>` verwenden

Interface `ListIterator`

<code>boolean hasPrevious()</code>	prüft, ob es Vorgängerelement gibt (rückwärts durchlaufen)
<code>E previous()</code>	liefert vorheriges Element
<code>int nextIndex()</code>	liefert Index des Elements, dass nach Aufruf von <code>next</code> aktuelles Element wird
<code>int previousIndex()</code>	liefert Index des Elements, dass nach Aufruf von <code>previous</code> aktuelles Element wird
<code>void add(E o)</code>	fügt <code>o</code> vor dem nächsten Element in die Liste ein
<code>void set(E o)</code>	ersetzt das zuletzt über <code>next/previous</code> mit <code>o</code>



Verwendung eines Iterators

```
// Iterator-Objekt „besorgen“
Iterator<...> iter = arrList.iterator();

// Durchlauf durch gesamte Collection
while(iter.hasNext()) {
    // aktuelles Element
    curr = iter.next();
    // mach was mit dem Element
    ...
}
```



Foreach-Schleifen bei Collections

- ① foreach Schleifen können bei Collections verwendet werden (wie bei Arrays!), wenn
 - ① Elemente nur ausgelesen und nicht verändert werden
 - ① Liste sequentiell beim ersten Element beginnend durchlaufen wird
 - ① nur **eine** Liste durchlaufen wird



Foreach-Schleifen bei Collections

```
ArrayList<String> arrList = new ArrayList<String>
```

```
// Liste befüllen
```

```
for (int i=1; i <= 10; i++)  
    arrList.add("Obj" + i);
```

```
// sprich: „for each element in arrList“
```

```
for (String element : arrList)  
    System.out.println(element);
```



Beispiel

- ① Erstelle eine **Klasse Pupil** (Name, Vorname, Klasse, Katalognummer, eindeutige Schüler-ID über alle Schüler)
- ① Erstelle ein **Testklasse**, welche die **Klasse PupilStorage** ausführlich testet
- ① Implementiere die Klasse **PupilStorage**, mit folgender Funktionalität (je eine Methode), die jeweils mit ArrayList bzw. LinkedList funktionieren soll
 1. **createPupils(int x)** : x Schüler werden erzeugt und in einer Collection gespeichert (Tip: generiere Namen + KatNr. mit Schleife)
 2. **deletePupil(int id)**: Lösche einen bestimmten Schüler anhand seiner SchülerID
 3. **searchName(String name)**: Suchen eines Schülers anhand seines Namens
 4. **printAll()**: Ausgeben aller Schüler (bei LinkedList auch umgekehrte Reihenfolge)
 5. **getPupilsAsArray()**: gibt alle Schüler in einem Feld zurück

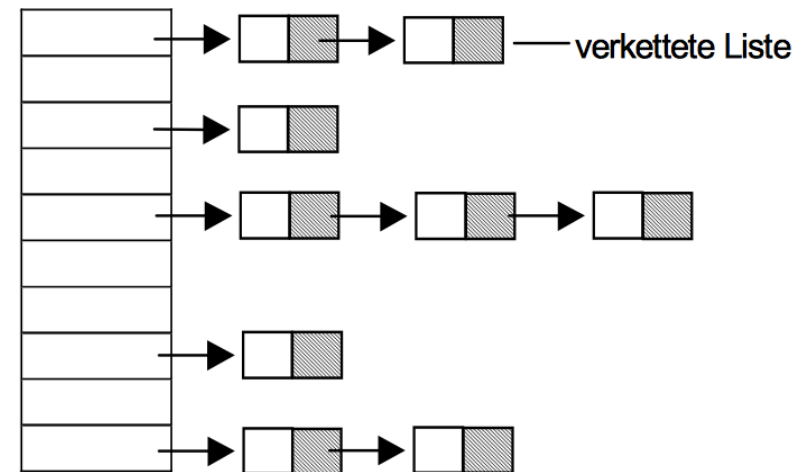


13.5 Hash-Tabellen und Bäume

- ④ Suche bei größeren Datenmengen (Arrays bzw. verkettete Listen gespeichert) sehr zeitaufwendig
- ④ Lösung: Hash-Tabelle
 - ④ Position, wo Element gespeichert wird,, wird berechnet
 - ④ Hashcode: schnell berechnet, hängt vom Element ab
 - ④ Hash-Tabelle ist Array von verketteten Listen (Buckets)
 - ④ Ist Hash-Tabelle voll wird sie umkopiert in doppelt so große
 - ④ Ladefaktor: gibt an bei welchem Füllstand umkopiert wird (default: 75%)

Hash-Tabellen

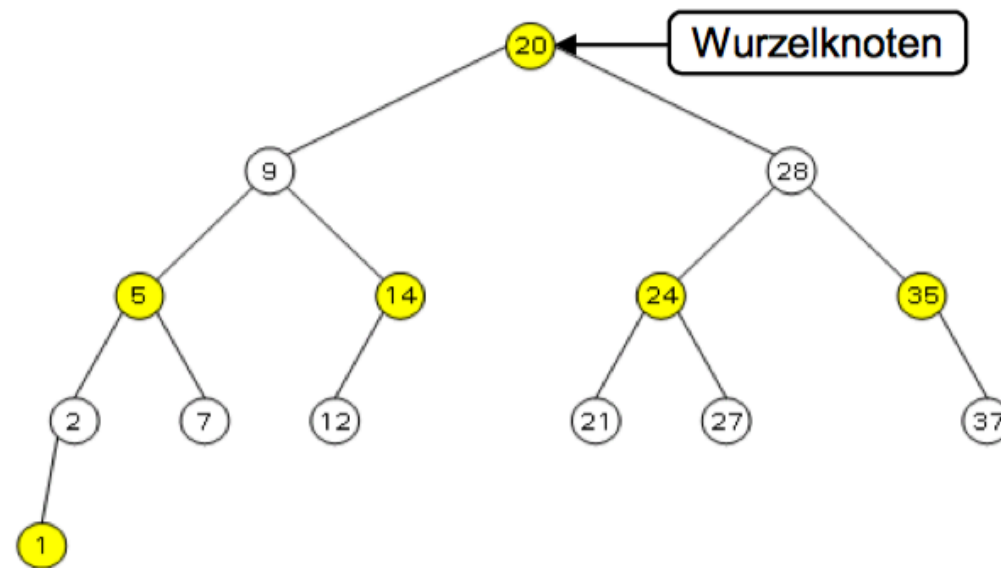
- ❶ Hash-Tabelle ist Array von verketteten Listen (=Buckets)
- ❷ Anzahl der verketteten Listen abhängig von Anzahl der zu speichernden Elemente
- ❸ in verkettete Liste werden jeweils die Elemente mit dem gleichen Hashcode aufgenommen
- ❹ Position eines Elements = *Hashcode modulo Gesamtzahl*



Aufbau einer Hash-Tabelle

Baumstrukturen

① Ideal für sortierte Daten

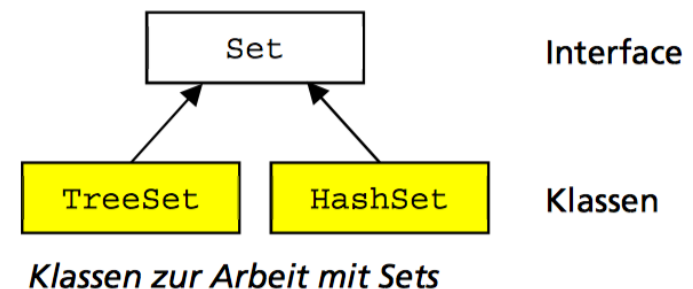


*Daten in einer Baumstruktur speichern
(die Zahlen sind willkürlich gewählt)*



Sets – Collections vom Typ Set

- ① Interface `Set` von `Collection` abgeleitet
- ① Unterschiede zu Listen
 - ① Keine doppelten Elemente (Math. Mengen)
 - ① Elemente sind gleich, wenn `o1.equals(o2)` `true` liefert
 - ① Elemente in Sets keine festgelegte Reihenfolge





Für schnellen Zugriff HashSet erstellen

- ④ HashSets immer dann, wenn kurzer Zugriff notwendig
- ④ Basieren auf Hash-Tabellen
- ④ Konstruktoren:
 - ④ `HashSet ()`
 - ④ Default Kapazität: 101 Elemente
 - ④ Ladefaktor: 75%
 - ④ `HashSet(int initialCapacity)`
 - ④ Kapazität aufrunden zu nächster Primzahl (hängt mit Berechnung zusammen)
 - ④ `HashSet (int initialCapacity, float loadFactor)`
 - ④ Ladefaktor: gibt in % an, ab welchem Füllgrad das HashSet vergrößert wird
 - ④ Je höher Füllgrad, desto langsamer der Zugriff



HashCode zur Speicherung im HashSet berechnen

- ① Methode `hashCode()` berechnet Hashcode eines Objekt
- ① Wenn Objekte in HashSet gespeichert werden, muss `hashCode()` überschrieben werden
- ① Auch `equals()` muss/sollte überschrieben werden, da sonst nur auf Objektgleichheit (gleiche Objektreferenz!), nicht aber auf Inhaltsgleichheit geprüft wird!



Beispiel HashSet

```
public class Person {
    private String lastname;
    private String prename;
    private int personalNr;

    ... //Getter- und Setter-Methoden

    //Standardkonstruktor
    public Person() {
        this("", "", 0);
    }
    public Person (String lastname,
                   String prename, int personalNr) {
        setLastname(lastname);
        setPrename(prename);
        setPersonalNr(personalNr);
    }
}
```



Beispiel HashSet

```
public int hashCode()
{
    return  getLastname().hashCode() +
           getPrenome().hashCode() +
           getPersonalNr();
}

public boolean equals(Object o) {
    if ((o == null) || (o.getClass() != this.getClass()))
        return false;
    else {
        Person obj = (Person)o;
        return ((obj.getLastname().equals(getLastname())) &&
                (obj.getPrenome().equals(getPrenome())) &&
                (obj.getPersonalNr() == getPersonalNr()));
    }
}
```



Mit TreeSet Baumstrukturen erstellen

- ① TreeSet: geordnete Speicherung
- ① d.h. Sortierreihenfolge muss vorgegeben sein!
 - ① bei bestimmten Klassen ist Sortierreihenfolge vorgehen (z.B. `String`)
- ① eigene Klassen: Interface `Comparable` implementieren!
 - ① `int compareTo(T o)`
 - ① Rückgabewert: gleich 0, größer >0, kleiner <0



Mit TreeSet Baumstrukturen erstellen

`E first()`

Liefert das erste (kleinste) Element des TreeSets

`E last()`

Liefert das letzte (größte) Element des TreeSets

`SortedSet<E> headSet(E toElement)`

Gibt den Teil des TreeSets zurück, dessen Elemente kleiner sind als ToElement

`SortedSet<E> subSet(E fromElement, E toElement)`

Gibt den Teil des TreeSets zurück, dessen Elemente größer sind als fromElement und kleiner als toElement sind

`SortedSet<E> tailSet(E fromElement)`

Gibt Deine Teil des TreeSets zurück, dessen Elemente größer oder gleich fromElement sind



Beispiel TreeSet

```
public class Person2
    extends Person
    implements Comparable<Person2>
{
    public Person2() {
        super();
    }

    public Person2 ( String lastname,
                     String prename,
                     int personalNr) {
        super(lastname, prename, personalNr);
    }
}
```



Beispiel TreeSet

```
public int compareTo(Person2 o) {
    if ((o == null) ||
        (o.getClass() != getClass()))
        return -1;

    int i = getLastname().compareTo(o.getLastname());

    if (i != 0)
        return i;
    else {
        i = getPrenome().compareTo(o.getPrenome());

        if (i != 0)
            return i;
        else {
            i = getPersonalNr() - o.getPersonalNr();
            return i;
        }
    }
}
```




Beispiel TreeSet

```
public static void main(String[] args) {
    TreeSet<Person2> personTreeSet = new TreeSet<Person2>();

    Person2 personA = new Person2("Meier", "Heinz", 33);
    ... //weitere Objekte definieren
    Person2 personE = new Person2("Geier", "Norbert", 35);

    personTreeSet.add(personA); ... // alle weiteren Objekte hinzufuegen
    personTreeSet.add(personE);

    System.out.println("Ausgabe mit Positionszeiger");

    Iterator iter = personTreeSet.iterator();
    while (iter.hasNext()) {
        Person2 x = (Person2)iter.next();
        System.out.println(x.getLastname() + ", " + x.getPrenome() + " - Personal-Nr: " +
                           x.getPersonalNr());
    }
    System.out.println("\nAusgabe in einer foreach-Schleife");

    for (Person2 x : personTreeSet)
        System.out.println(x.getLastname() + ", " + x.getPrenome() + " - Personal-Nr: " +
                           x.getPersonalNr());
}
```



13.7 Maps – Collections vom Typ $\text{Map}\langle K, V \rangle$

- ① Map = Tabelle mit Schlüssel-Werte-Paare $\langle K, V \rangle$
 - ① Wert (Value) wird gemeinsam mit Schlüssel (Key) gespeichert – jeder Schlüssel hat GENAU einen Wert
 - ① über Schlüssel kann Wert schnell gefunden werden
 - ① wird neuer Wert mit gleichem Schlüssel eingefügt, so wird bestehender Wert überschrieben
- ① $\langle K, V \rangle$ bedeutet, dass 2 Typargumente angehen werden muss
 - ① K ... KeyType
 - ① V ... ValueType



Methoden des Interface Map

<code>void clear()</code>	Entfernt alle Wertepaare der Map
<code>boolean isEmpty()</code>	Liefert den Wert <code>true</code> , wenn die Map kein Schlüssel-Wert- Paar enthält
<code>V remove(Object key)</code>	Entfernt das Objekt, das durch den übergebenen Schlüssel bezeichnet ist
<code>boolean equals(Object o)</code>	Vergleicht das übergebene Objekt mit dieser Map
<code>int hashCode()</code>	Gibt den Hashcode für diese Map zurück
<code>int size()</code>	Gibt die Anzahl der Wertepaare der Map zurück
<code>boolean containsKey(Object key)</code>	Liefert den Wert <code>true</code> , wenn der übergebene Schlüssel in der Map enthalten ist
<code>boolean containsValue(Object value)</code>	Liefert den Wert <code>true</code> , wenn die Map einen oder mehrere Schlüssel zu dem übergebenen Objekt besitzt
<code>Set <Map.Entry<K,V>> entrySet()</code>	Gibt ein Set zurück, das die Schlüssel-Wert-Paare der Map in Entry-Objekten enthält. Entry-Objekte sind Objekte, die Schlüssel-Wert-Paare speichern. Über die Methoden <code>getKey</code> , <code>getValue</code> und <code>setValue</code> kann auf Schlüssel und Wert zugegriffen werden.
<code>V get(Object key)</code>	Liefert den Wert zu dem übergebenen Schlüssel aus dieser Map
<code>Set<K> keySet()</code>	Gibt ein Set zurück, das die Schlüssel der Map enthält
<code>V put(K key, V value)</code>	Verbindet den übergebenen Wert mit dem übergebenen Schlüssel in dieser Map
<code>Collection<V> values()</code>	Erzeugt ein Collection-Objekt mit den Werten dieser Map



Methoden des Interface Map

- ④ Iterator ist nicht implementiert
- ④ Foreach-Schleife ist möglich, für Ausgabe
- ④ wenn Iterator benötigt wird:
 - ④ `entrySet` erzeugt Set mit `EntrySet`-Objekten
 - ④ `keySet` erzeugt Set mit Schlüsseln
 - ④ `values` erzeugt Collection mit Werten



Mit HashMaps arbeiten

- ④ HashMap speichert Schlüssel-Werte-Paare in Form einer Hash-Tabelle
- ④ hashCode wird aus Schlüssel berechnet
- ④ bei Kollision (gleicher hashCode)->verkettete Liste
 - ④ `HashMap()`
 - ④ `HashMap(int initialCapacity)`
 - ④ `HashMap(int initialCapacity, float loadFactor)`



Mit TreeMaps arbeiten

- ④ TreeMaps speichern Schlüssel-Werte-Paare in Baumform
- ④ Sortierung: nur Wert des Schlüssels, nicht Element!
- ④ über Konstruktoren kann Sortierreihenfolge festgelegt werden:
 - ④ `TreeMap()`
 - ④ Schlüssel in natürlicher Reihenfolge sortiert
 - ④ `TreeMap(Comparator c)`
 - ④ Comparator Objekt wird übergeben, in dem die Sortierung über eigene compare-Methode selbst festgelegt wird
 - ④ `TreeMap(Map m)`
 - ④ erzeugt TreeMap aus Map



Beispiel TreeMap

```
public static void main(String[] args) {
    HashMap<String, Person2> hashmap = new HashMap<String, Person2>(11);

    Person2 personA = new Person2("Meier", "Heinz", 33);
    String keyA = "16233686"; ... //weitere Schluessel und Objekte erzeugen

    hashmap.put(keyA, personA); ... //weitere Schluessel-Wert-Paare hinzufügen

    System.out.println("\nHashMap sequentiell durchlaufen:");

    for (Map.Entry<String, Person2> e : hashmap.entrySet())
    {
        String key = e.getKey();
        Person2 person = e.getValue();
        System.out.println("Schluessel: " + key +
                           " Name: " + person.getLastname() + ", " +
                           person.getPrenome() + " Personal-Nr: " +
                           person.getPersonalNr());
    }
}
```



Beispiel TreeMap

```
//Aus dem HashMap eine TreeMap erzeugen
TreeMap<String, Person2> treemap = new TreeMap<String, Person2>(hashmap);

System.out.println("\nTreeMap sequentiell durchlaufen:");

for (Map.Entry<String, Person2> e : treemap.entrySet()) {
    String key = e.getKey();
    Person2 person = e.getValue();

    System.out.println("Schluessel: " + key + " Name: " +
        person.getLastname() +
        ", " + person.getPrenome() +
        " Personal-Nr: " +
        person.getPersonalNr());
}
}
```




Beispiel TreeMaps

HashMap sequentiell durchlaufen:

```
Schluessel: 35243534 Name: Geier, Norbert Personal-Nr: 35  
Schluessel: 64376657 Name: Schneider, Guenther Personal-Nr: 25  
Schluessel: 45674576 Name: Schneider, Bernd Personal-Nr: 41  
Schluessel: 16233686 Name: Meier, Heinz Personal-Nr: 33  
Schluessel: 68832346 Name: Geier, Norbert Personal-Nr: 49
```

TreeMap sequentiell durchlaufen:

```
Schluessel: 16233686 Name: Meier, Heinz Personal-Nr: 33  
Schluessel: 35243534 Name: Geier, Norbert Personal-Nr: 35  
Schluessel: 45674576 Name: Schneider, Bernd Personal-Nr: 41  
Schluessel: 64376657 Name: Schneider, Guenther Personal-Nr: 25  
Schluessel: 68832346 Name: Geier, Norbert Personal-Nr: 49
```



13.8 Übung

- ① Definieren Sie die Klasse `Book`. Sie dient zum Speichern der Daten über ein Buch und hat die Eigenschaften `author`, `title` und `issue`. Erstellen Sie Getter- und Setter-Methoden zum Speichern und Auslesen der Attribute. Im Konstruktor werden die übergebenen Werte gesetzt.
- ② Erstellen Sie eine Anwendung mit dem Namen `Exercise`.
- ③ Definieren Sie ein Array, in dem sechs Objekte der Klasse `Book` gespeichert werden sollen.
- ④ Lassen Sie im Konstruktor der Klasse `Exercise` sechs Objekte der Klasse `Book` erstellen und in dem Array speichern. Dieses Array wird für die folgenden Übungsaufgaben ⑥ - ⑧ benötigt.
- ⑤ Für jeden der nachfolgenden Aufgabenteile ⑥ bis ⑧ ist eine Methode `part6 ... part8` zu programmieren. Der Aufruf der entsprechenden Methode erfolgt in Abhängigkeit vom Parameter, der der Anwendung übergeben wird. Der Parameter entspricht der Aufgabennummer. Wird das Programm z. B. mit Parameter 6 aufgerufen, soll die Methode `part6` ausgeführt werden. Erstellen Sie im Konstruktor eine `if`-Verzweigung, in der Sie die entsprechende Methode aufrufen.

13.8 Übung

- ⑥ Erzeugen Sie für die Verwaltung der Bücher eine ArrayList. Fügen Sie in die ArrayList die sechs Buchobjekte des Arrays ein und geben Sie diese unsortiert, sortiert und in umgekehrter Reihenfolge sortiert aus. Die Ausgabe sollte den folgenden Aufbau besitzen:

```
Goethe: "Faust I"           Auflage: 20000 Stueck
Schiller: "Wilhelm Tell"    Auflage: 10000 Stueck
...
Fontane: "Effi Briest"      Auflage: 10000 Stueck
*** in umgekehrter Reihenfolge ***
Schiller: "Wilhelm Tell"    Auflage: 10000 Stueck
...
Fontane: "Effi Briest"      Auflage: 10000 Stueck
*** in sortierter Reihenfolge ***
Fontane: "Effi Briest"      Auflage: 10000 Stueck
...
Schiller: "Wilhelm Tell"    Auflage: 10000 Stueck
```



13.8 Übung

Um eine Sortierung in der gewünschten Form zu erhalten, müssen Sie in die Klasse `Book` die Methode `compareTo` implementieren.

- ⑦ Verwenden Sie für die Bücherverwaltung eine `HashMap`. Füllen Sie diese mit den Büchern des Arrays. Als Schlüssel soll der laufende Index verwendet werden. Geben Sie anschließend die Bücher der `HashMap` aus (ohne Index).

Definieren Sie nun eine `TreeMap`, die die gleichen Elemente enthält wie die `HashMap`, und geben Sie sie ebenfalls aus.

- ⑧ In dieser Aufgabe sollen die Bücher in Collections vom Typ `Set` verwaltet werden. Arbeiten Sie zuerst mit einem `HashSet`. Übernehmen Sie die Bücher aus dem Array und zeigen Sie diese an. Fügen Sie dann ein neues Buch hinzu, das in der Liste bereits existiert. Erscheint dieses Buch in der Anzeige doppelt, ist das ein Fehler (ein `Set` kann jedes Objekt nur einmal enthalten). Lesen Sie in diesem Kapitel nach, wie Sie dieses Problem lösen können.

Speichern Sie nun die Objekte des `HashSet` in einem `TreeSet` und zeigen Sie die Objekte des `TreeSet` an.