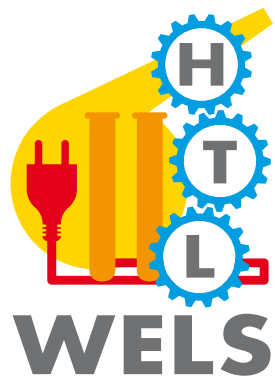


13 COLLECTIONS- FRAMEWORK

SEW 3

DI Thomas Helml

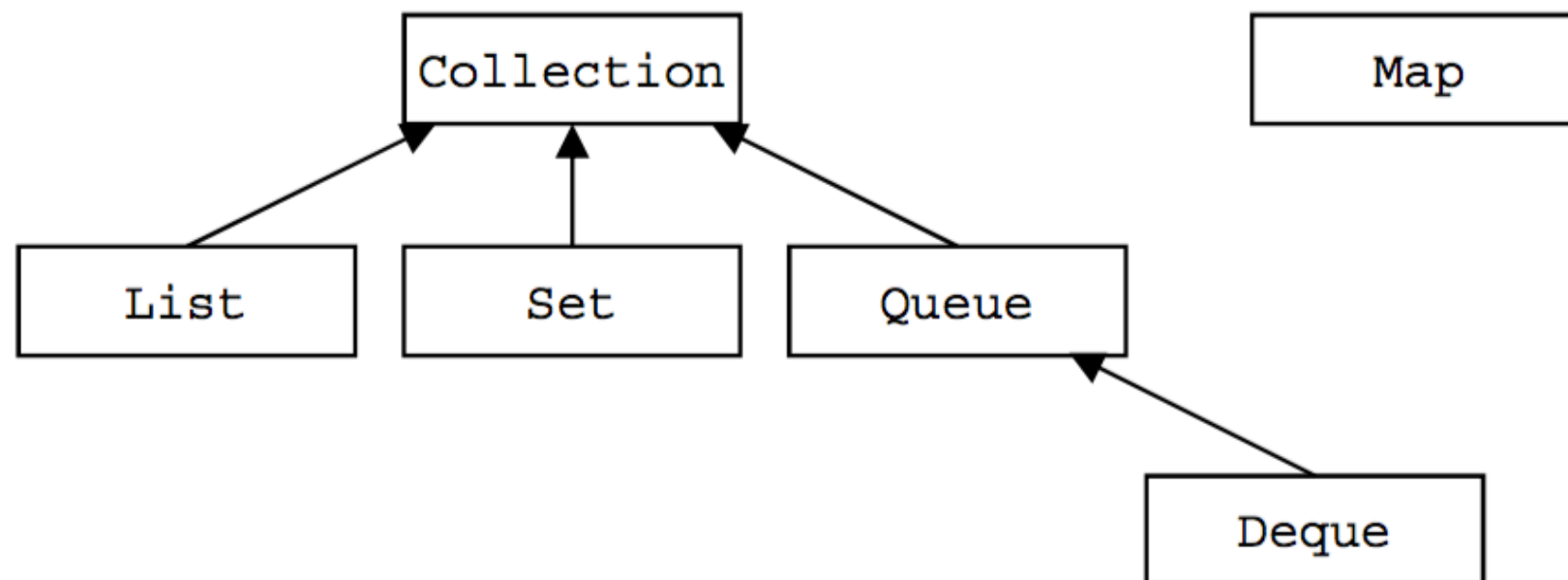


- 13.1 Grundlagen zum Collections Framework
- 13.2 Das Interface Collection
- 13.3 Mit Listen arbeiten
- 13.4. Listen sequentiell durchlaufen
- 13.5 Hash-Tabellen und Bäume
- 13.6 Sets - Collections vom Typ Set
- 13.7 Maps - Collections vom Typ Map<K,V>
- 13.8 Beispiel
- 13.9 Auswahl der passenden Collection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

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

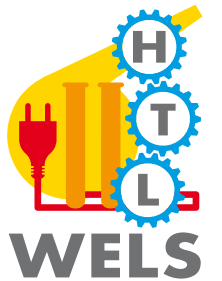
- Collections (=Container) sind Datenstrukturen, die eine Gruppe von Daten zu einer Einheit zusammenfassen
- Elemente = Daten der Collection
- Zugriff auf Elemente mit entsprechenden Methoden
- Beispiele:
 - Liste:
 - Duplikate erlaubt, eventuell sortiert, Zugriff wahlweise oder sequenziell
 - Menge: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

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





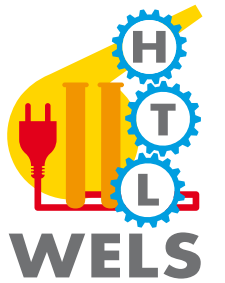
INTERFACES-HIERARCHIE



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

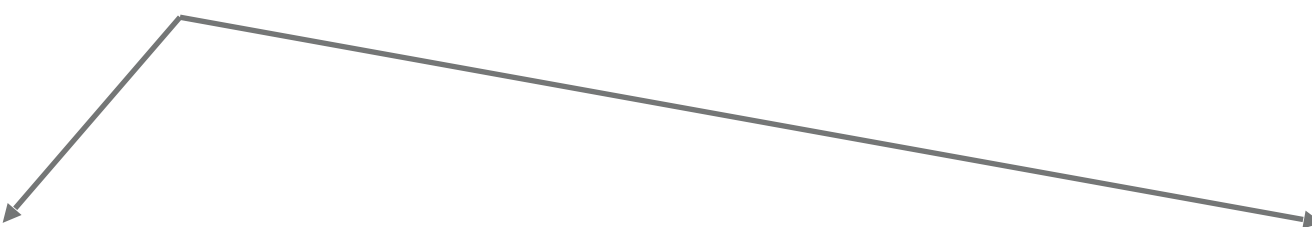


VORTEILE DES COLLECTIONS-FRAMEWORKS



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

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



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


- 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

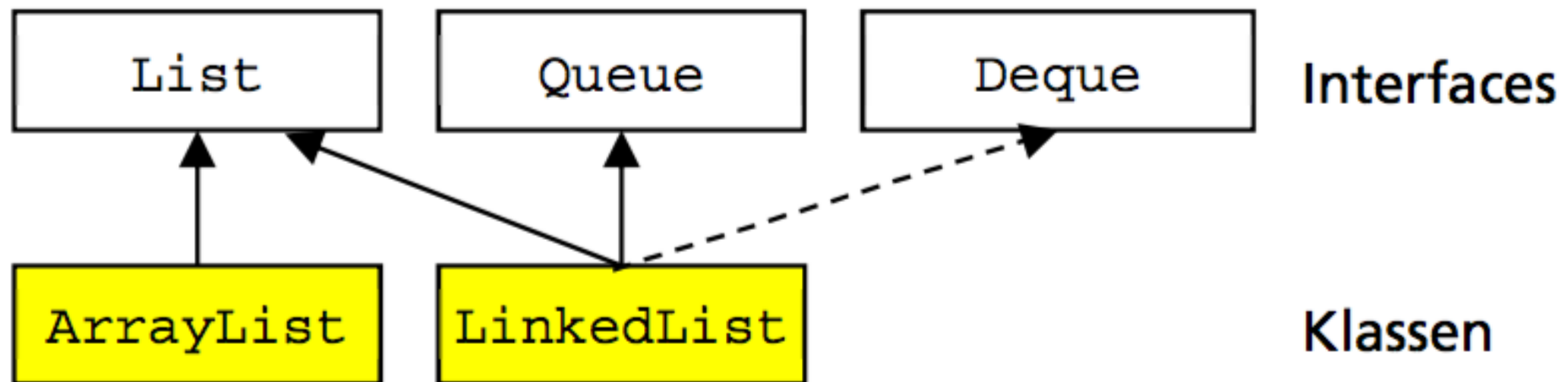
- 13.1 Grundlagen zum Collections Framework
- 13.2 Das Interface Collection
- 13.3 Mit Listen arbeiten
- 13.4. Listen sequentiell durchlaufen
- 13.5 Hash-Tabellen und Bäume
- 13.6 Sets - Collections vom Typ Set
- 13.7 Maps - Collections vom Typ Map<K,V>
- 13.8 Beispiel
- 13.9 Auswahl der passenden Collection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

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

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

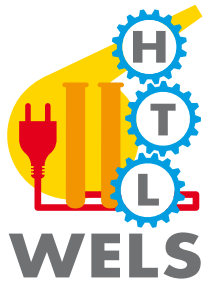
- **Listen** eignen sich für
 - **geordnete Mengen**
 - **wahlfreier** Zugriff (Index)
 - **sequentieller** Zugriff (der Reihe nach)



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



EINE LISTE BEARBEITEN

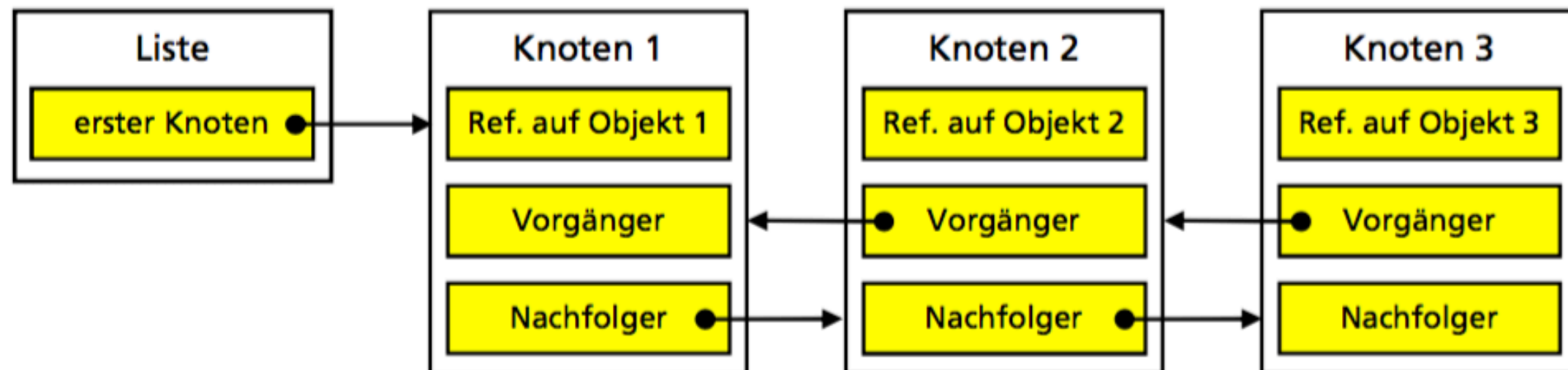


ArrayList implementiert ***zusätzlich*** zum Interface Collection folgende Methoden:

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

➤ LinkedList

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



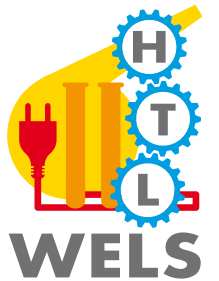


- 13.1 Grundlagen zum Collections Framework
- 13.2 Das Interface Collection
- 13.3 Mit Listen arbeiten
- **13.4. Listen sequentiell durchlaufen**
- 13.5 Hash-Tabellen und Bäume
- 13.6 Sets - Collections vom Typ Set
- 13.7 Maps - Collections vom Typ Map<K,V>
- 13.8 Beispiel
- 13.9 Auswahl der passenden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

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



POSITIONSZEIGER VOM TYP LISTITERATOR<E> VERWENDEN



Interface Iterator

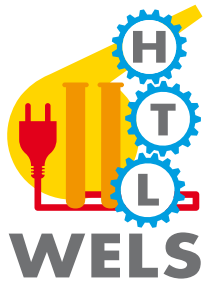
`boolean hasNext()` prüft, ob weitere Elemente in Collection sind

`E next()` holt aktuelles Element aus Collection und setzt Positionszeiger auf nächstes Element

`void remove()` 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 next aktuelles Element wird
<code>int previousIndex()</code>	liefert Index des Elements, dass nach Aufruf von previous aktuelles Element wird
<code>void add(E o)</code>	fügt o vor dem nächsten Element in die Liste ein
<code>void set(E o)</code>	ersetzt das zuletzt über next/previous mit o

```
// Iterator-Objekt „besorgen“
```

```
Iterator<...> iter = arrList.iterator();
```

```
// Durchlauf durch gesamte Collection
```

```
while(iter.hasNext()) {
```

```
    // aktuelles Element
```

```
    curr = iter.next();
```

```
    // mach was mit dem Element
```

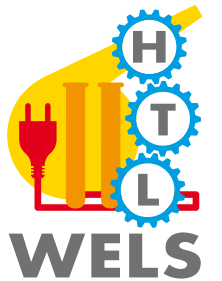
```
    ...
```

```
}
```

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



FOREACH-SCHLEIFEN BEI COLLECTIONS



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

```
// Liste befüllen
```

```
for (int i=1; i <= 10; i++)
```

```
    arrList.add("Obj" + i);
```

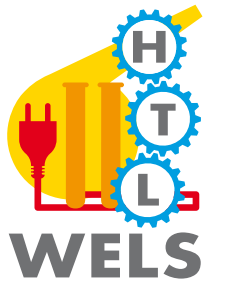
```
// sprich: „for each element in arrList“
```

```
for (String element : arrList)
```

```
    System.out.println(element);
```



BEISPIEL

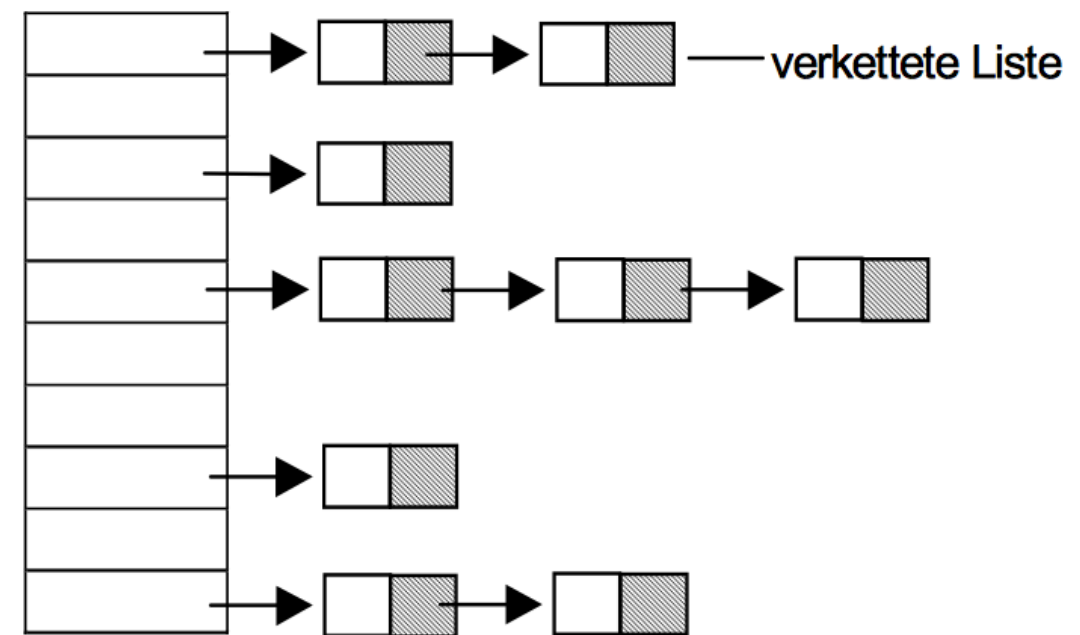


➤ Beispiel 401_Smurfs

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

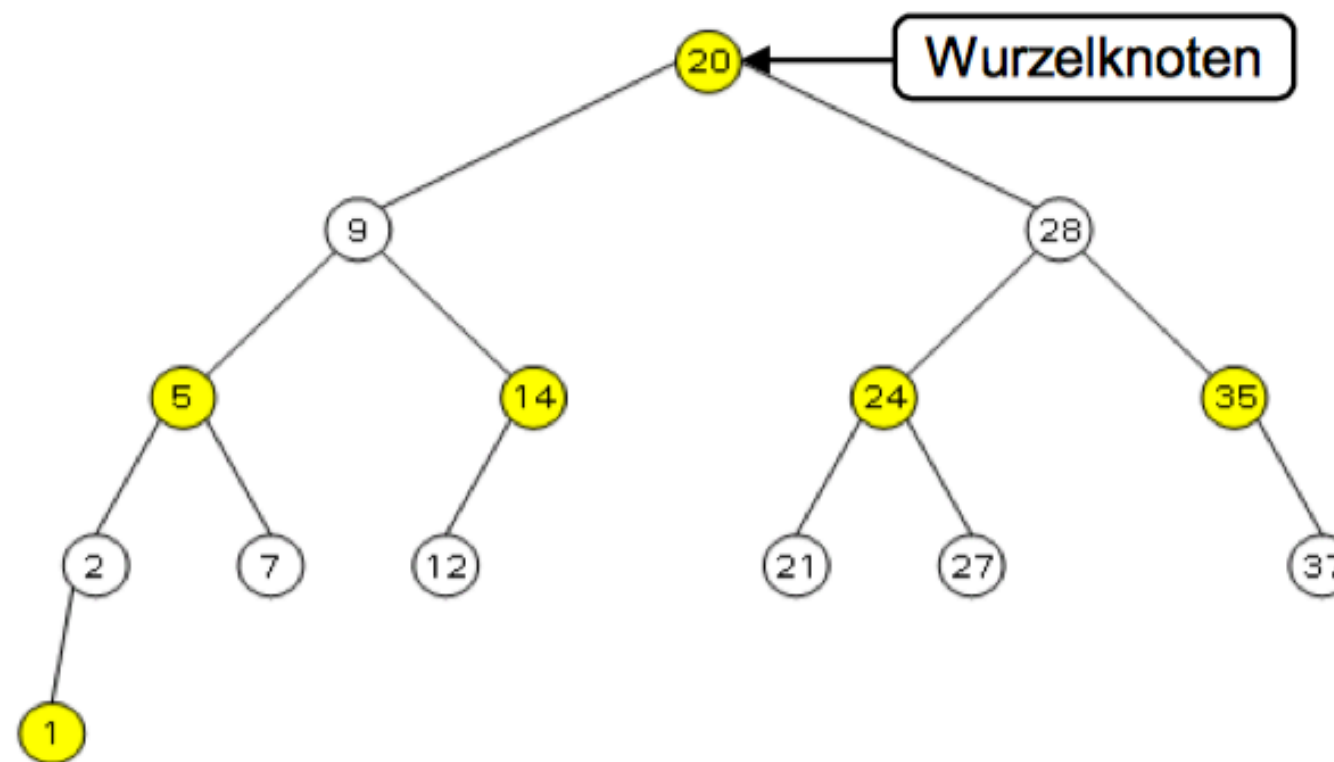
- Suche bei größeren Datenmengen (Arrays bzw. verkettete Listen gespeichert) sehr zeitaufwendig
- **Lösung: Hash-Tabelle**
 - Position, wo Element gespeichert wird, wird berechnet
 - Hashcode: schnell berechnet, hängt vom Element ab
 - Hash-Tabelle ist Array von verketteten Listen (Buckets)
 - Ist Hash-Tabelle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

- Hash-Tabelle ist Array von verketteten Listen (=Buckets)
- Anzahl der verketteten Listen abhängig von Anzahl der zu speichernden Elemente
- in verkettete Liste werden jeweils die Elemente mit dem gleichen Hashcode aufgenommen
- Position eines Elements = $\text{Hashcode} \bmod \text{Gesamtzahl}$



Aufbau einer Hash-Tabelle

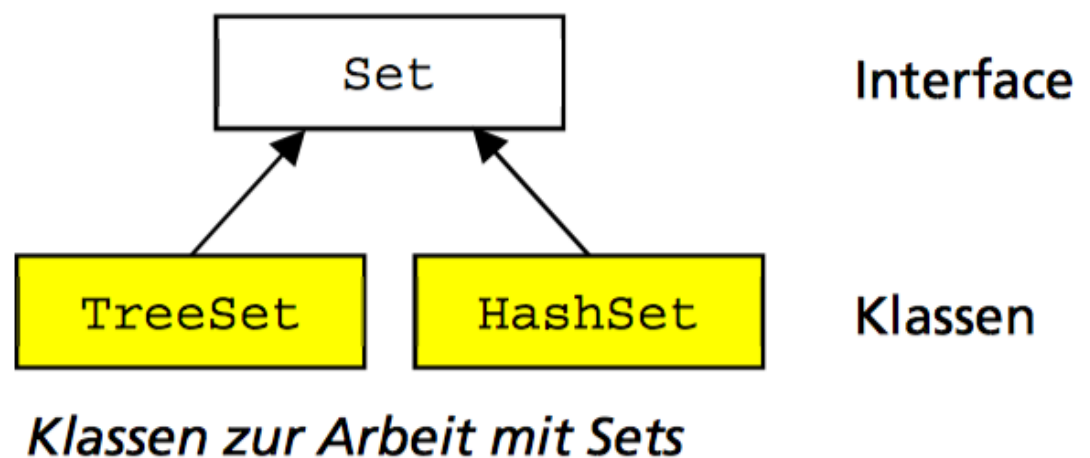
- Ideal für sortierte Daten



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

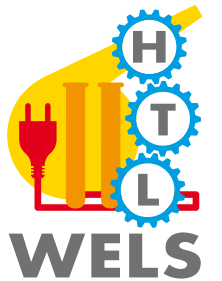
- 13.1 Grundlagen zum Collections Framework
- 13.2 Das Interface Collection
- 13.3 Mit Listen arbeiten
- 13.4. Listen sequentiell durchlaufen
- 13.5 Hash-Tabellen und Bäume
- **13.6 Sets - Collections vom Typ Set**
- 13.7 Maps - Collections vom Typ Map<K,V>
- 13.8 Beispiel
- 13.9 Auswahl der passenden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

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





FÜR SCHNELLEN ZUGRIFF HASHSET ERSTELLEN



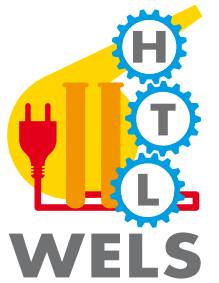
- HashSets immer dann, wenn kurzer Zugriff notwendig
- Basieren auf Hash-Tabellen
- Konstruktoren:
 - `HashSet ()`
 - Default Kapazität: 101 Elemente
 - Ladefaktor: 75%
 - `HashSet(int initialCapacity)`
 - Kapazität aufrunden zu nächster Primzahl (hängt mit Berechnung zusammen)
 - `HashSet (int initialCapacity,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

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

- Folgende Bedingungen müssen für hashCode () erfüllt sein:
 - jeder Aufruf von .hashCode () , für ein bestimmtes Objekt, muss denselben Wert zurückgeben, solange das Objekt nicht verändert wurde. Der Wert kann bei einem erneuten Programmstart anders sein
 - Sind 2 Objekte gleich (equals-Methode!), dann muss .hashCode () für beide Objekte den gleichen Wert zurück liefern
 - .hashCode() darf für 2 Objekte denselben Wert liefern, er muss nicht eindeutig sein (obwohl das von Vorteil ist)



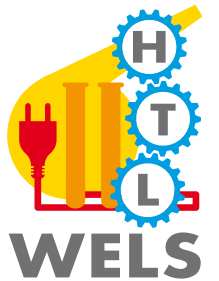
BEISPIEL HASHSET



```
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

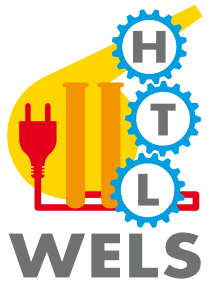


```
public int hashCode()  
{  
    return    getLastname().hashCode() +  
              getPrenome().hashCode() +  
              getPersonalNr();  
}  
  
public boolean equals(Object o) {  
    if ((o == null) || (o.getClass() != this.getClass()))  
        return false;  
    else {  
        Person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

- ④ TreeSet: geordnete Speicherung
- ④ d.h. Sortierreihenfolge muss vorgegeben sein!
 - ④ bei bestimmten Klassen ist Sortierreihenfolge vorgehen (z.B. `String`)
- ④ eigene Klassen: Interface `Comparable` implementieren!
 - ④ `int compareTo(T o)`
 - ④ Rückgabewert: gleich 0, größer >0 , kleiner <0



MIT TREESSET 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)
```

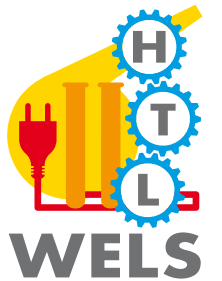
Gibt den Teil des TreeSets zurück, dessen Elemente
größer sind als fromElement und kleiner als toElement
sind

```
SortedSet<E> tailSet(E fromElement)
```

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



BEISPIEL TREESSET

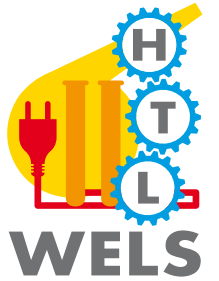


```
public class Person2
    extends Person
    implements Comparable<Person2>
{
    public Person2() {
        super();
    }

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



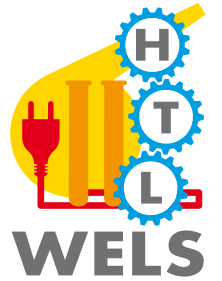
BEISPIEL TREESSET



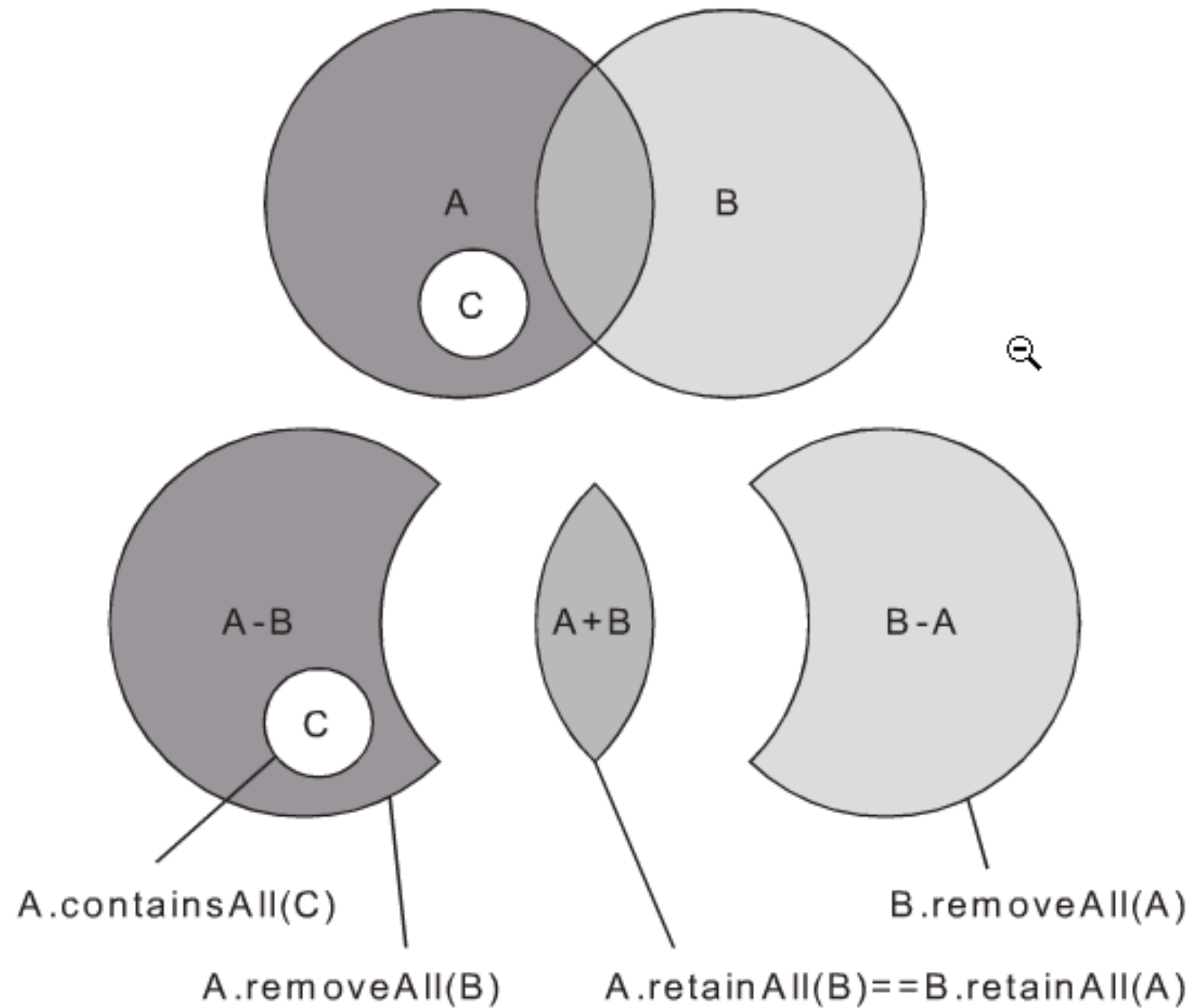
```
public int compareTo(Person2 o) {  
    if ((o == null) ||  
        (o.getClass() != getClass()))  
        return -1;  
  
    int i = getLastname().compareTo(o.getLastname());  
  
    if (i != 0)  
        return i;  
    else {  
        i = getPrenome().compareTo(o.getPrenome());  
  
        if (i != 0)  
            return i;  
        else {  
            i = getPersonalNr() - o.getPersonalNr();  
            return i;  
        }  
    }  
}
```



BEISPIEL TREESSET



```
public static void main(String[] args) {  
    Set<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());  
}
```

- ④ Methoden des Interface `Collection` funktionieren wie Mengenoperationen
- ④ `menge1.addAll(menge2)`
 - ④ Vereinigungsmenge von `menge1` und `menge2`
- ④ `menge1.containsAll(menge2)`
 - ④ `true`, wenn `menge1` eine Untermenge von `menge2` ist.
- ④ `menge1.removeAll(menge2)`
 - ④ Differenz beider Mengen gebildet.
- ④ `menge1.retainAll(menge2)`
 - ④ Es wird der Durchschnitt beider Mengen bestimmt

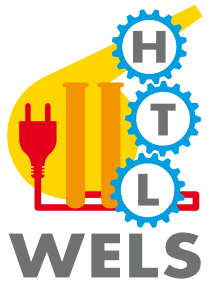


- 13.1 Grundlagen zum Collections Framework
- 13.2 Das Interface Collection
- 13.3 Mit Listen arbeiten
- 13.4. Listen sequentiell durchlaufen
- 13.5 Hash-Tabellen und Bäume
- 13.6 Sets - Collections vom Typ Set
- 13.7 Maps - Collections vom Typ $\text{Map}\langle K, V \rangle$
- 13.8 Beispiel
- 13.9 Auswahl der passenden Collection
- 13.10 Klasse Collections

- ① Map = Tabelle mit Schlüssel-Werte-Paare $<k, v>$
 - 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 $<k, v>$ 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>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

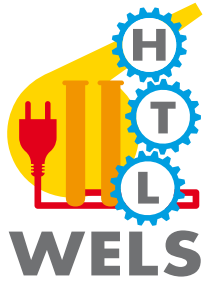
- ① Iterator ist nicht implementiert
- ① Foreach-Schleife ist möglich, für Ausgabe
- ① wenn Iterator benötigt wird:
 - ① `entrySet` erzeugt Set mit `EntrySet`-Objekten
 - ① `keySet` erzeugt Set mit Schlüsseln
 - ① `values` erzeugt Collection mit Werten

- ① HashMap speichert Schlüssel-Werte-Paare in Form einer Hash-Tabelle
- ① hashCode wird aus Schlüssel berechnet
- ① bei Kollision (gleicher hashCode) -> verkettete Liste
 - ① `HashMap()`
 - ① `HashMap(int initialCapacity)`
 - ① `HashMap(int initialCapacity, float loadFactor)`

- ① TreeMap speichern Schlüssel-Werte-Paare in Baumform
- ① Sortierung: nur Wert des Schlüssels, nicht Element!
- ① über Konstruktoren kann Sortierreihenfolge festgelegt werden:
 - ① `TreeMap()`
 - ① Schlüssel in natürlicher Reihenfolge sortiert
 - ① `TreeMap(Comparator c)`
 - ① Comparator Objekt wird übergeben, in dem die Sortierung über eigene `compare-Methode` selbst festgelegt wird
 - ① `TreeMap(Map m)`
 - ① erzeugt TreeMap aus Map



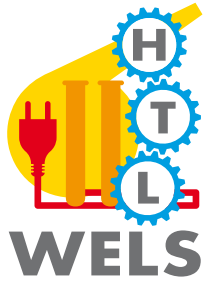
BEISPIEL TREEMAPS



```
public static void main(String[] args) {  
    Map<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 TREEMAPS



//Aus dem HashMap eine TreeMap erzeugen

```
Map<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: " +
```

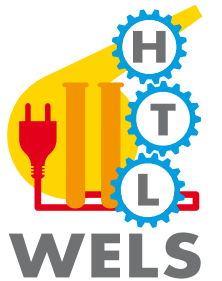
```
        person.getPersonalNr());
```

```
}
```

```
}
```



BEISPIEL TREEMAPS



.....

HashMap sequentiell durchlaufen:

Schluesssel: 35243534 Name: Geier, Norbert Personal-Nr: 35

Schluesssel: 64376657 Name: Schneider, Guenther Personal-Nr: 25

Schluesssel: 45674576 Name: Schneider, Bernd Personal-Nr: 41

Schluesssel: 16233686 Name: Meier, Heinz Personal-Nr: 33

Schluesssel: 68832346 Name: Geier, Norbert Personal-Nr: 49

TreeMap sequentiell durchlaufen:

Schluesssel: 16233686 Name: Meier, Heinz Personal-Nr: 33

Schluesssel: 35243534 Name: Geier, Norbert Personal-Nr: 35

Schluesssel: 45674576 Name: Schneider, Bernd Personal-Nr: 41

Schluesssel: 64376657 Name: Schneider, Guenther Personal-Nr: 25

Schluesssel: 68832346 Name: Geier, Norbert Personal-Nr: 49



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

- 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.

- 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
```


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

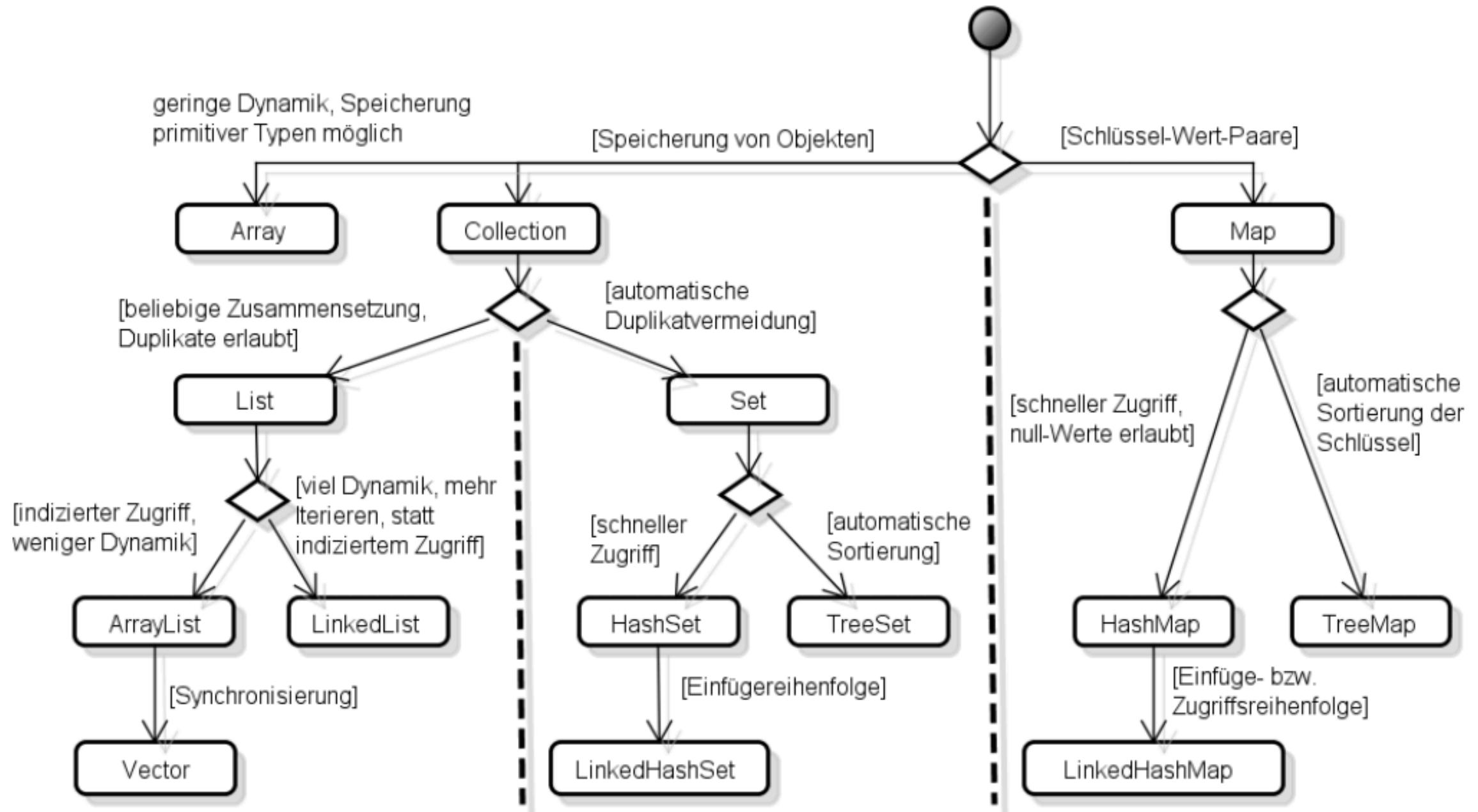
- ⑦ Verwenden Sie für die Bücherverwaltung eine `HashMap`. Füllen Sie diese mit den Büchern des Arrays. Als Schlüssel soll der laufende Index verwendet werden. Geben Sie anschließend die Bücher der `HashMap` aus (ohne Index).

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.



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





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

① Klasse Collectionss

- ① enthält nur statische Methoden
- ① Algorithmen und Hilfsmethoden
- ① je nach Algorithmus bzw. Hilfsmethode:
 - ① nur bestimmte Interfaces unterstützt
 - ① Algorithmen sind polymorph
 - ① können für verschiedene Implementierungen eines Interface-Typs verwendet werden



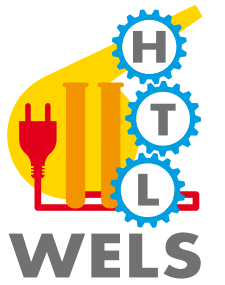
KLASSE COLLECTIONS



Methode		Erklärung	
binarySearch()		binären Suche nach einem Objekt der Collection	
copy()		Die Elemente einer Liste können in eine andere Liste kopiert werden (Achtung: Zielliste mindestens Größe der Ausgangsliste)	
fill()		Initialisieren einer Liste mit einem bestimmten Wert	
min(), max()		kleinste bzw. größtes Element der Collection (Objekte müssen das Interface Comparable implementieren)	
nCopies()		Gibt List-Objekt zurück, das n Objekte enthält	
reverse()		Reihenfolge der Collection tauschen (nur bei Listen möglich)	
shuffle()		Die Elemente der Collection werden gemischt	
sort()		Liste sortieren (Objekte müssen Interface Comparable implementieren)	



INHALT



➤ Queues/Stack

① Queues/Schlangen

① Klassisches Beispiel Warteschlange

① Interface Queue ab Java 1.5

① normalerweise FIFO-Prinzip (First In, First Out)

① Spezielle Implementierung: DeQueue

① Double Ended Queues

① also Schlangen an denen an beiden Enden Elemente angefügt und entnommen werden können

① E element ()

① liefert das Element vom Kopf der Queue, entfernt es aber nicht

① ist Queue leer => NoSuchElementException

① E peek ()

① wie element () - nur null, falls leer

① E remove ()

① wie element () - nur mit entfernen des Elements

① E poll ()

① wie peek () - nur mit entfernen des Elements

① boolean offer (E obj)

① fügt Element in Queue ein

① Stack = Stapel

① LIFO Speicher: Last in, First Out

① Operationen:

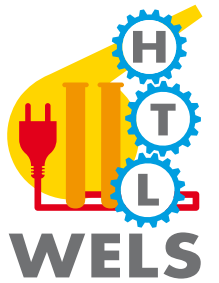
① push: auf Stapel legen

① pop: von Stapel nehmen

① peek: oberstes Element vom Stapel ansehen



STACK



```
import java.util.*;
```

```
public class Stapel
```

```
{
```

```
    Stapel ()
```

```
    {
```

```
        Stack<Integer> stapel = new Stack<Integer>();
```

```
        for(int i = 1; i <= 10; i++)
```

```
            stapel.push(new Integer(i*i));
```

```
        System.out.println(stapel.peek());
```

```
        while(!stapel.empty())
```

```
            System.out.println((Integer)stapel.pop());
```

```
    }
```

```
}
```