13 Collections-Framework

SEW

DI Thomas Helml

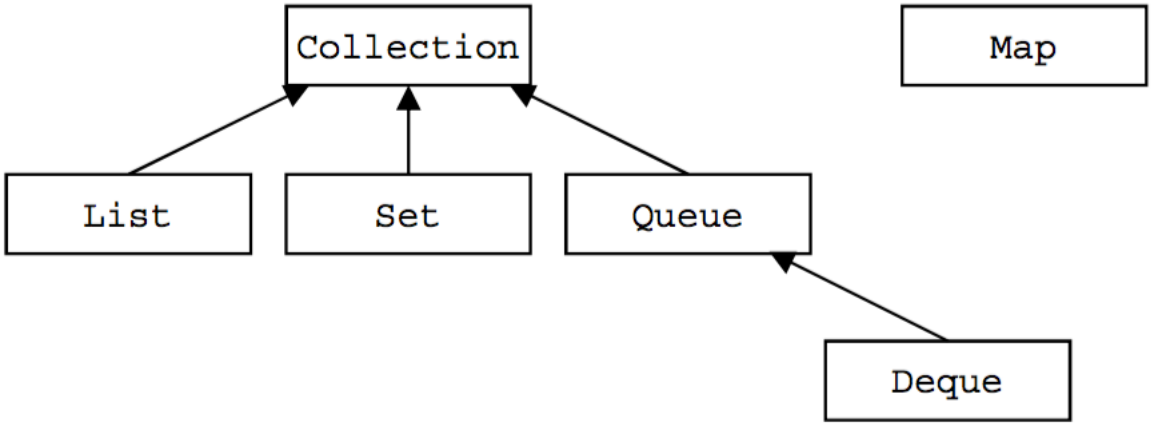
Inhalt

13.1 Was sind 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

Interfaces-Hierarchie

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



Interfaces-Hierarchie

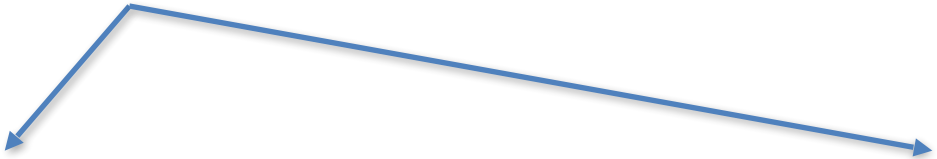
Collection	Basis-Interface, grundlegende Methoden
`List`	Beliebig große Liste, auf Elemente kann sequentiell od. wahlfrei zugegriffen werden
`Set`	Menge von Elementen ohne Duplikate, Mengenoperationen
`Queue`	Warteschlange, nur sequentieller Zugriff (z.B. FIFO), kein wahlfreier Zugriff
`Deque`	**d**ouble **e**nded **que**ue – 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

Einheitliche Datentypen

- 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

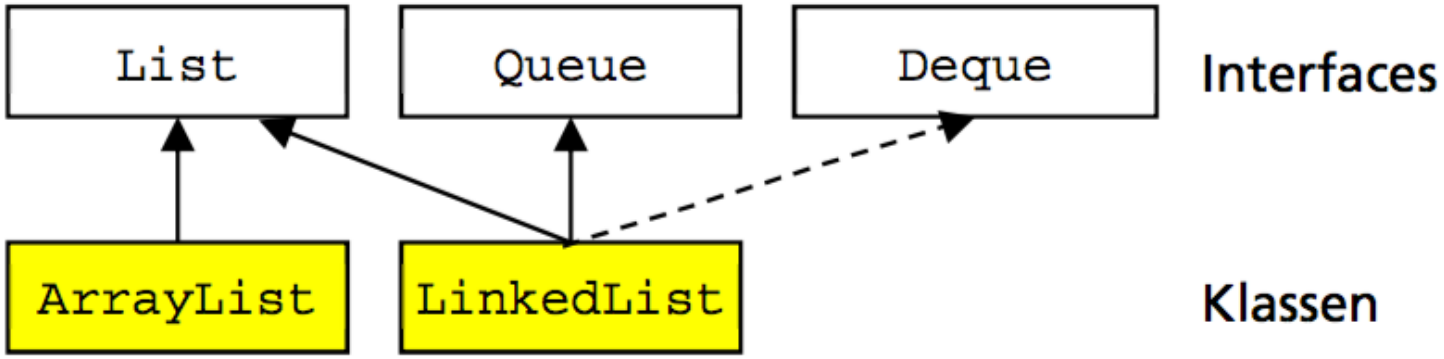
- Klassen, die Typargumente besitzen, nennt man ***generische Klassen***
- durch Angabe eines Datentyps entsteht ein ***generischer Datentyp***
- kann der Compiler aus dem Kontext das Typargument erkennen, kann der ***Diamond-Operator*** <> verwendet werden

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

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



Eine Liste bearbeiten

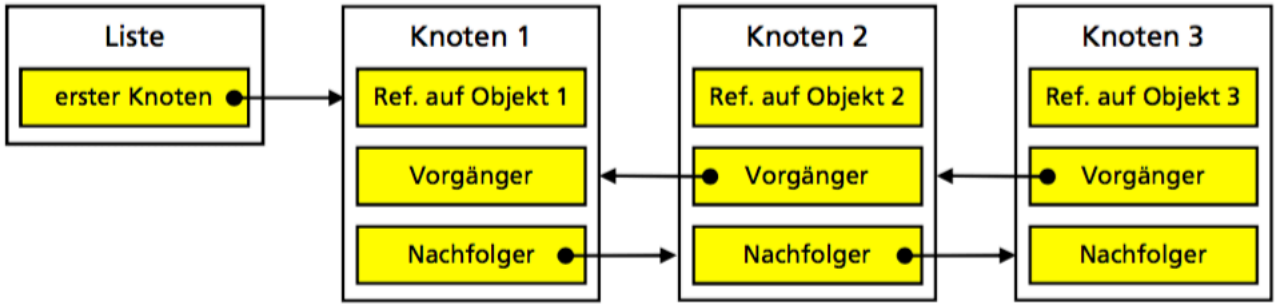
- ***`ArrayList`*** implementiert eine Liste, deren Elemente linear hintereinander gespeichert werden (intern: Array)
  - erstes Element (analog zu Arrays): Index 0
  - Unterschied zu Arrays:
    - `ArrayList` ist dynamisch, d.h. zur Laufzeit lassen sich Elemente löschen + hinzufügen
  - beim Löschen/Einfügen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

Eine Liste bearbeiten

`ArrayList` implementiert ***zusätzlich*** zum Interface `Collection` folgende Methoden:	
`void add(int index, E element)`	Fügt `element` an der Position `index` ein
`E get (int index)`	Gibt das Element an der Position `index` zurück
`int indexOf(Object o)`	Ermittelt den Index des *ersten Vorkommens* von `o`
`E set (int index, E element)`	Ersetzt das Element an der Position `index` durch `element`

Doppelt verkettete Listen erstellen

- `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



13.4 Listen sequentiell durchlaufen

- 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 `Iterator<E>` verwenden

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

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

- 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

```
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

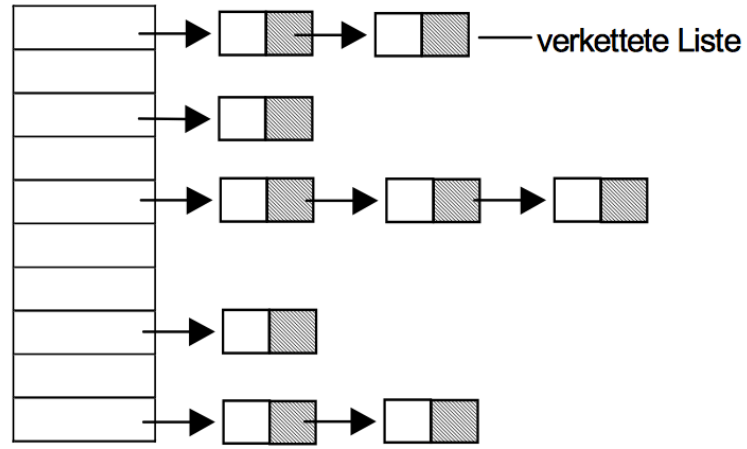
- Erstelle eine ***Klasse Smurf***(Name, Vorname, PilzNr., eindeutige Schlumpf-ID über alle Schlümpfe)
- Erstelle ein ***Testklasse***, welche die ***Klasse SmurfStorage*** ausführlich testet
- Implementiere die Klasse *SmurfStorage*, mit folgender Funktionalität (je eine Methode), die jeweils mit ArrayList bzw. LinkedList funktionieren soll
  1. ***createSmurfs(int x)*** : x Schlümpfewerden erzeugt und in einer Collection gespeichert (Tip: generiere Namen + KatNr. mit Schleife)
  2. ***deleteSmurf(int id):*** Lösche einen bestimmten Schlumpf anhand seiner SchülumpfID
  3. ***searchName(String name):*** Suchen eines Schlumpfes anhand seines Namens
  4. ***printAll():*** Ausgeben aller Schlümpfe (bei LinkedList auch umgekehrte Reihenfolge)
  5. ***getSmurfAsArray():*** gibt alle Schlümpfe in einem Feld zurück

13.5 Hash-Tabellen und Bäume

- 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-Tabellen

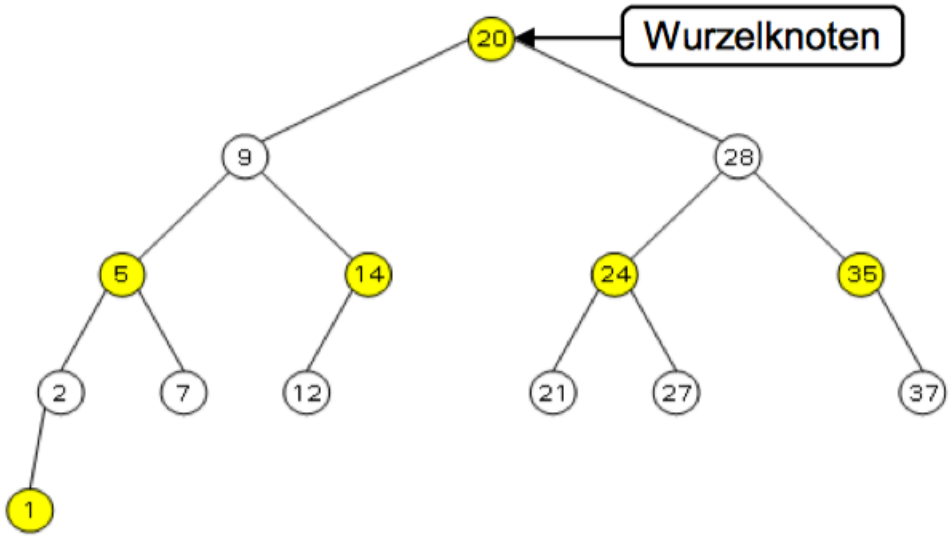
- Hash-Tabelle ist Array von verketteten Listen (=Buckets)
- Anzahl der verketteten Listen abhängig von Anzahl der zu speichernden Elemente
- in verkettete Liste werden jeweils die Elemente mit dem gleichen Hashcode aufgenommen
- Position eines Elements = *Hashcode modulo Gesamtzahl*



Aufbau einer Hash-Tabelle

Baumstrukturen

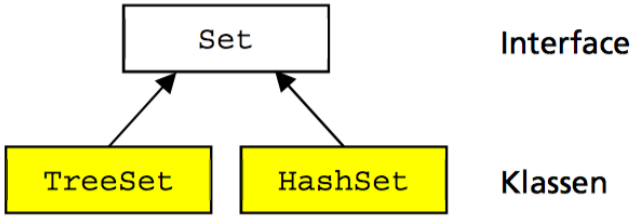
- Ideal für sortierte Daten



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

13.6 Sets - Collections vom Typ `Set`

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



Klassen zur Arbeit mit Sets

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

Hashcode zur Speicherung im HashSet berechnen

- 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!

Methode .hashCode()

- 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() +
            getPrename().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.getPrename().equals(getPrename())) &&
              (obj.getPersonalNr() == getPersonalNr()));
    }
  }
}
```

Mit TreeSet Baumstrukturen erstellen

- `TreeSet`: geordnete Speicherung
- d.h. Sortierreihenfolge muss vorgegeben sein!
  - bei bestimmten Klassen ist Sortierreihenfolge vorgehen (z.B. `String`)
- eigene Klassen: Interface `Comparable` implementieren!
  - `int compareTo(T o)`
  - 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);
   }
```

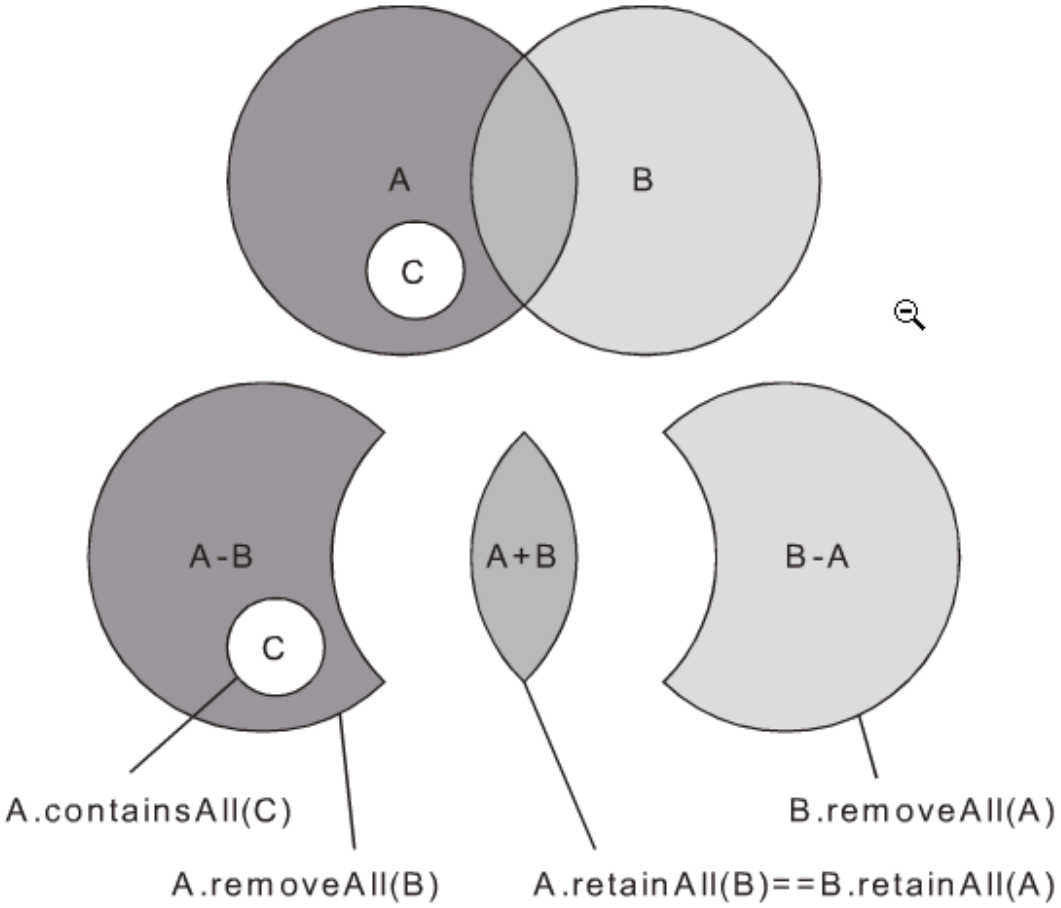
Beispiel TreeSet

```
    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 = getPrename().compareTo(o.getPrename());

            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.getPrename() + " - Personal-Nr: " +
                              x.getPersonalNr());
      }
      System.out.println("\nAusgabe in einer foreach-Schleife");

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

Sets - Mengenoperationen



Sets - Mengenoperationen

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

13.7 Maps – Collections vom Typ Map<K,V>

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

Methoden des Interface Map

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

Methoden des Interface Map

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

Mit HashMaps arbeiten

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

Mit TreeMaps arbeiten

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

Beispiel TreeMaps

```
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.getPrename() + " Personal-Nr: " +
                       person.getPersonalNr());
  }
```

Beispiel TreeMaps

```
  //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.getPrename() +
                       " 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.

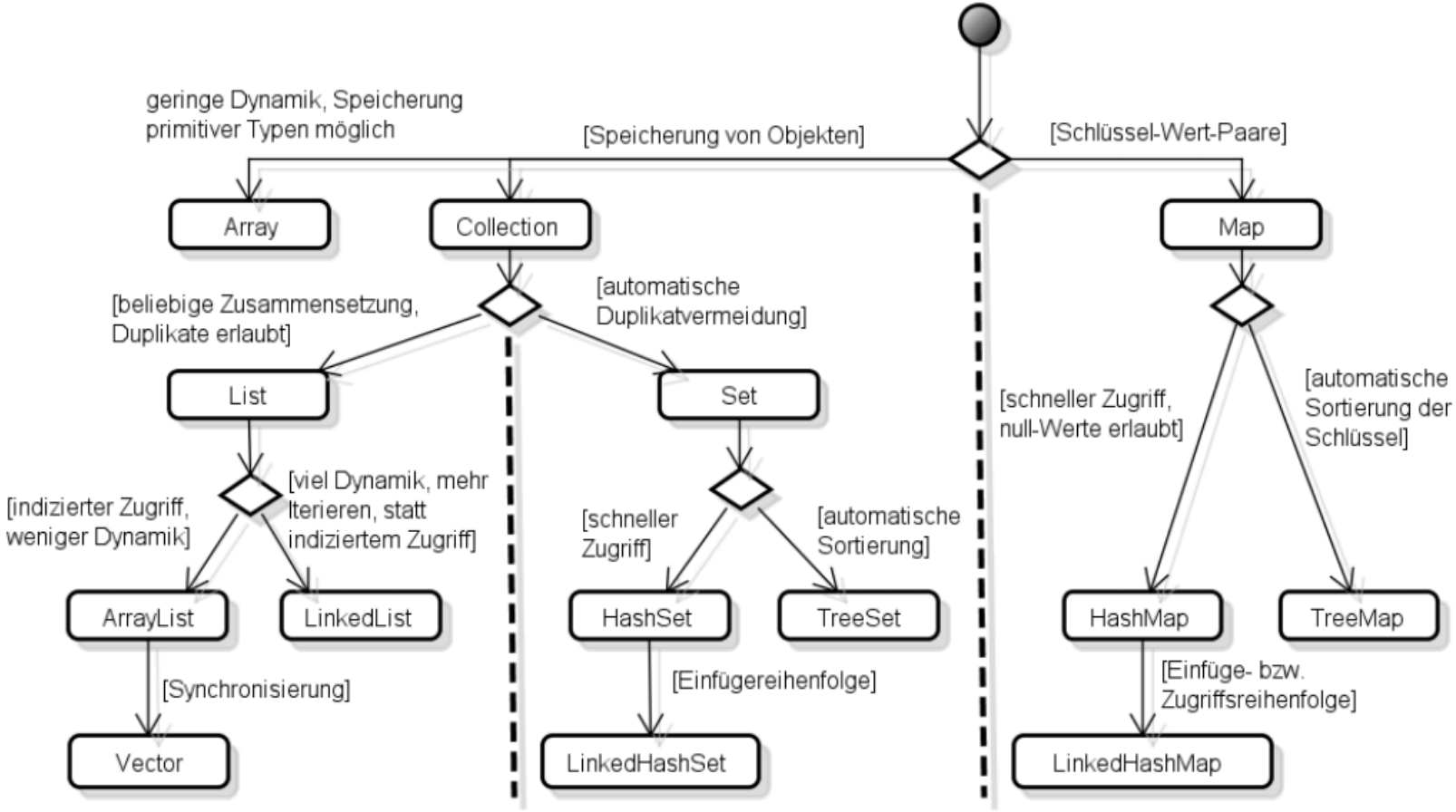
⑦ Verwenden Sie für die Bücherverwaltung eine HashMap. Füllen Sie diese mit den Büchern des Arrays. Als Schlüssel soll der laufende Index verwendet werden. Geben Sie anschließend die Bücher der HashMap aus (ohne Index).

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 HashSets in einem TreeSet und zeigen Sie die Objekte des TreeSets an.

13.9 Auswahl der passenden Collection



13.10 Klasse Collections

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

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)

Queue

- Queues/Schlangen
  - Klassisches Beispiel Warteschlange
  - Interface `Queue ab Java 1.5`
  - normalerweise FIFO-Prinzip (First In, First Out)
  - Spezielle Implementierung: DeQueue
    - Double Ended Queues
    - also Schlangen an denen an beiden Enden Elemente angefügt und entnommen werden können

Queue

- `E element()`
  - liefert das Element vom Kopf der Queue, entfernt es aber nicht
  - ist Queue leer => `NoSuchElementException`
- `E peek()`
  - wie `element()` - nur `null`, falls leer
- `E remove()`
  - wie `element()` - nur mit entfernen des Elements
- `E poll()`
  - wie `peek()` - nur mit entfernen des Elements
- `boolean offer(E obj)`
  - fügt Element in Queue ein

Stack

- Stack = Stapel
  - LIFO Speicher: Last in, First Out
  - Operationen:
    - push: auf Stapel legen
    - pop: von Stapel nehmen
    - 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());
    }
}
```