HAW Hamburg
Fakultät TI
Technik und Informatik

Programmiermethodik 1
Programmiertechnik

Collections

Wiederholung

- Einführung
- Exception-Typen
- catch und finally
- Exceptions werfen
- Logging

Ausblick für heute

Use Cases

- Ich habe eine Menge von Dingen und möchte diese effizient verwalten (performanter Lese- und Schreibzugriff).
- Ich benötige unterschiedliche Container-Typen (mit unterschiedlichen Stärken und Schwächen) für unterschiedliche Daten.
- Ich möchte auf strukturierte Art Objekte vergleichen (z.B. zum Sortieren).

Agenda

- Collections-Framework
- Verkettete Liste und Array-Liste
- Iteratoren
- Vergleichen
- Menge
- Map
- Collections-Operationen

Einführung

Motivation

- häufige Anforderung: mehrere Dinge verwalten
- bisher kennengelernt: Arrays
- Nachteile von Arrays
  - feste Länge
  - Einfügen "in der Mitte"
  - kein echter Referenztyp
  - sehr starr, nicht für alle Anwendungsfälle geeignet
- Lösung: `Collection`-Framework
  - Datenstrukturen
  - Operationen

Collection-Framework

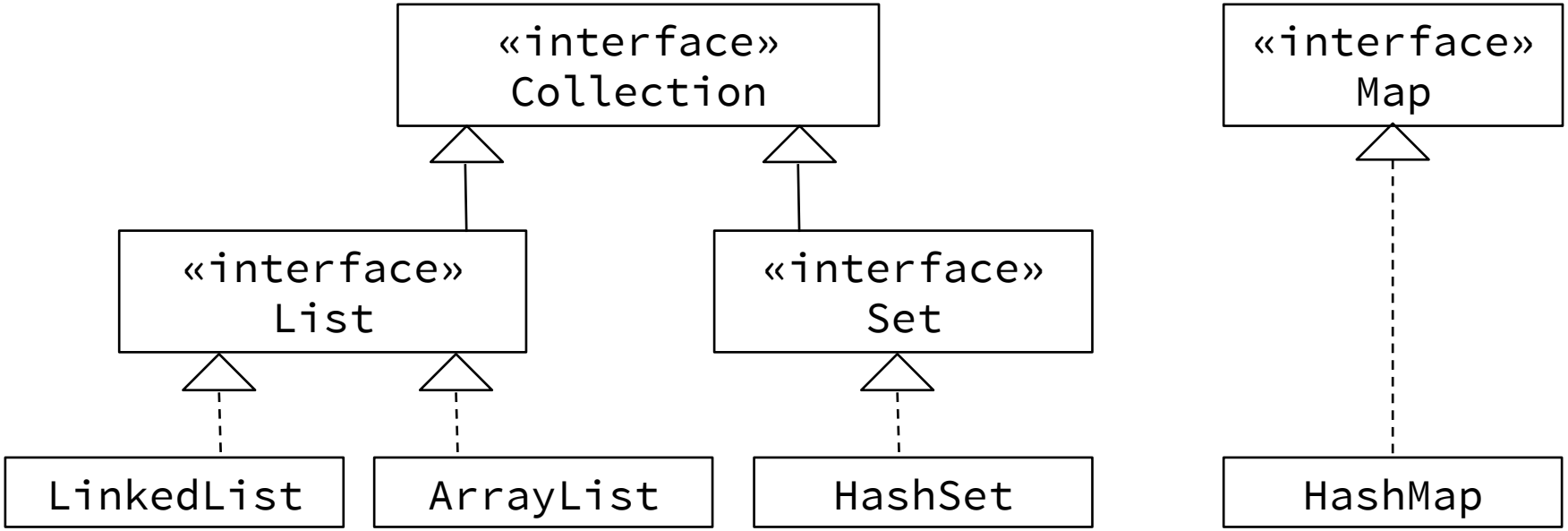
- Die Klassen des `Collection`-Frameworks ("`Collection`-Klassen") im Package `java.util ...`
  - sind Containerobjekte (wie Arrays)
  - haben eine veränderliche Elementanzahl
    - keine fest vorgegebene Länge
  - können mit einer `foreach`-Schleife durchlaufen werden (wie Arrays)
  - können Objekte aller Referenztypen als Elemente speichern
    - bis Java 1.4: nur Elementtyp `Object`
    - ab Java 1.5.: generische Typen

Generische Typen

- werden ausführlich in Programmiermethodik 2 behandelt
- hier zum Verständnis notwendig
- Beispiel: `List<T>`
  - Deklaration einer Liste von Objekten vom Typ T
  - Typ T? Kenne ich nicht
  - Gibt es auch nicht – tatsächlich verwendeten Typ einsetzen
  - etwa: `List<String>` = Liste von Strings

Collection-Framework

- Listen (`List`) ordnen die Elemente an
  - erstes .. letztes Element



Mengen (`Set`) speichern die Elemente ungeordnet und ohne Duplikate

Abbildungen (`Map`) speichern Element-Paare (Schlüssel, Wert)

Wichtige Methoden des Interfaces Collection


```
boolean add(Elementtyp element)
```

- fügt das Element zur `Collection` hinzu
- liefert `true`, wenn das Element eingefügt wurde
- nur bei `Set` kann `false` vorkommen

```
boolean remove(Elementtyp element)
```

- löscht das Element aus der `Collection`
- liefert `true`, wenn das Element gefunden und gelöscht wurde

```
boolean contains(Elementtyp element)
```

- liefert `true`, wenn das Element in der `Collection` enthalten ist

Wichtige Methoden des Interfaces Collection


```
int size()
```

- liefert die Anzahl an Elementen in dieser `Collection`

```
Object[] toArray()
```

- liefert ein neues Array mit allen Elementen der `Collection`
- bei Listen muss die Reihenfolge erhalten bleiben.

```
Elementtyp[] toArray(Elementtyp[] array)
```

- speichert die Elemente der `Collection` im Array `array`
- falls groß genug, sonst wird ein neues Array zurückgeliefert

Verkettete Liste und Array-Liste

Listen

- erste Kategorie von `Collections`: `Listen`
- Interface `List` von Interface `Collection` abgeleitet
- Eigenschaften
  - Elemente haben Reihenfolge
  - Elemente können mehrfach vorkommen (an unterschiedlichen Positionen)

Wichtige Methoden des Interfaces List


```
Elementtyp get(int index)
```

- liefert das Element an Listenposition `index`
- löst eine `IndexOutOfBoundsException` aus, wenn der `index` ungültig ist, also wenn `(index < 0 || index >= size())`

```
Elementtyp set(int index, Elementtyp element)
```

- ersetzt das Element an Listenposition `index` durch `element`
- liefert das alte Element zurück.

Wichtige Methoden des Interfaces List


```
int indexOf(Elementtyp element)
```

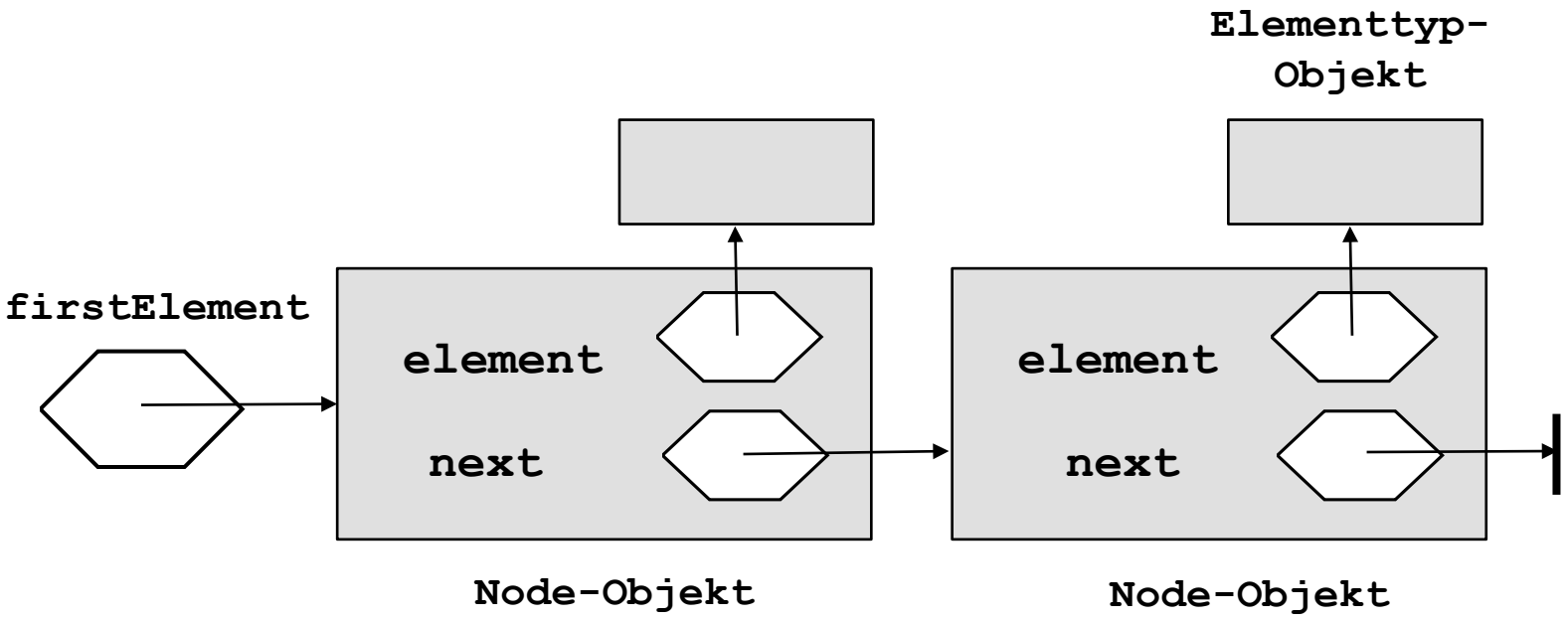
- liefert den Index (Listenposition) des ersten Vorkommens von `element` in der Liste
- oder `-1`, falls `element` nicht in der Liste enthalten ist
- zum Vergleichen wird die `equals`-Methode des Elementtyps verwendet!

Referenzimplementierungen

- Interface List hat zwei Referenzimplementierungen
- Klasse `LinkedList`: verkettete Liste
- Klasse `ArrayList`: verhält sich wie ein Array

Klasse LinkedList

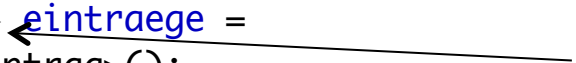
- Implementierung des Interfaces `List` durch Objektzeiger



Beispielanwendung: Telefonliste


```
/**
 * Eine Telefonliste verwaltet Einträge von Kontakten.
 */
public class Telefonliste {
  /**
   * Die Liste der Einträge wird als Collection (List) verwaltet.
   **/
  private List<TelefonlistenEintrag> eintraege =
      new LinkedList<TelefonlistenEintrag>();

  /**
   * Es sollen Telefonnummern und dazugehörige Namen in die Liste
eingetragen
   * bzw. verändert werden.
   */
  public void eintragHinzufuegen(String name, String nummer) {
    eintraege.add(new TelefonlistenEintrag(name, nummer));
  }
```



hier wird der Typ der Elemente der Liste festgelegt

Konvertierung

- `Collections` können in andere Typen überführt werden

```
List<TelefonlistenEintrag> eintraege =
        new LinkedList<TelefonlistenEintrag>();
```

- Konvertierung in Array:

```
Object[] arrayVonEintraegen = eintraege.toArray();
```

- Problem: Typ `Object`, daher besser:

```
TelefonlistenEintrag[] arrayVonEintraegen =
        new TelefonlistenEintrag[eintraege.size()];
eintraege.toArray(arrayVonEintraegen);
```

- Konvertierung von LinkedList in ArrayList

```
List<TelefonlistenEintrag> eintraegeAlsArrayList =
        new ArrayList<TelefonlistenEintrag>(eintraege);
```

Klasse ArrayList

- Implementierung des Interface `List` durch ein `Array`
  - mit automatischer Größenanpassung
- Beispielanwendung: Telefonliste
  - einzige Änderung gegenüber `LinkedList`-Version:

```
private ArrayList<PhoneListEntry> eintraege = new
        ArrayList<PhoneListEntry>();
```

anstelle von

```
private LinkedList<PhoneListEntry> eintraege = new
        LinkedList<PhoneListEntry>();
```

- der restliche Code bei der Verwendung bleibt derselbe wie für `LinkedList`
  - es werden nur Methoden aus dem `Collection`-Interface verwendet

Vergleich ArrayList und LinkedList

- Eigenschaften von `ArrayList`:
  - Indexzugriff auf Elemente (z.B. `get(10)`) ist überall schnell
  - Einfügen und Löschen ist nur am Listenende schnell, am Listenanfang langsam
- Eigenschaften von `LinkedList`:
  - Indexzugriff auf Elemente ist an den Enden schnell, in der Mitte langsam
    - wegen Java-Implementierung mit doppelten Zeigern
    - jeweils auf Erstes Element + Nachfolger und letztes Element + Vorgänger
  - Einfügen und Löschen ohne Indexzugriff ist überall schnell

Initialisierung nicht-leerer Listen

- bisher bekannt: Methode add(Typ element):

```
List<String> liste = new ArrayList<String>();
liste.add("Jan");
liste.add("Hein");
liste.add("Klaas");
```

- elegantere Möglichkeit

```
List<String> listeMitInit =
        Arrays.asList("Jan", "Hein", "Klaas", "Pit");
```

Übung: Liste

- Schreiben Sie Quellcode, der folgendes macht
- Erzeugen einer Liste (Implementierung: verkettete Liste) mit den Einträgen 23, 42, 12, 11
- Ausgaben des Listeninhalts auf der Konsole
  - so soll es aussehen: {23, 42, 12, 11}
- Ausgaben des zweiten Elements auf der Konsole

Iteratoren

Durchlaufen von Collections

- bisher: foreach-Schleife oder Schleife mit Zähler

```
Collection<String> stadtteile =
        new LinkedList<String>(
                Arrays.asList("Hafencity", "Wandsbek", "Altona"));
for ( String stadtteil : stadtteile ) {
        System.out.print(stadtteil + " ");
}
```

oder

```
for ( int i = 0; i < stadtteile.size(); i++ ) {
        System.out.print( stadtteile.get(i) + " " );
}
```

Durchlaufen von Collections

- Probleme
  - gleicher Zugriff für unterschiedliche `Collection`-Container
    - Mengen? Elemente haben keine Reihenfolge und daher keinen Index
  - kein Zeiger auf das nächste Listenelement benutzbar
    - notwendig zum Löschen

Iterator

- Lösung: Interface `Iterator<Elementtyp>`
- wird erzeugt durch eine `Collection`-Methode

```
Iterator<String> stadtteilIterator = stadtteile.iterator();
```

- Iteration über alle Elemente einer Collection
- Zusicherung: alle Elemente werden besucht
- keine Zusagen zur Reihenfolge der Elemente
- Iterator "zeigt" zu jedem Zeitpunkt auf ein Element

Methoden


```
public boolean hasNext()
```

- liefert `true`, wenn der `Iterator` auf ein existierendes Element zeigt

```
public Elementtyp next()
```

- liefert das aktuelle Element und geht zum nächsten Element weiter
- falls nicht vorhanden: `NoSuchElementException`

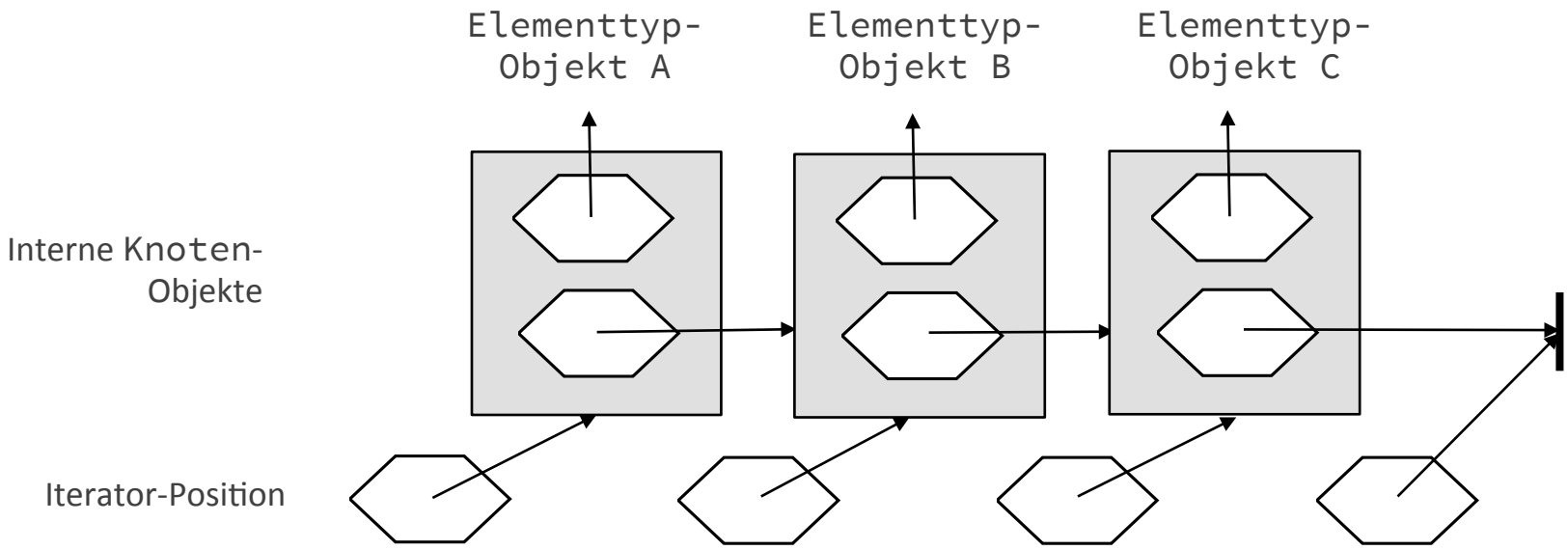
```
public void remove()
```

- entfernt das zuletzt mit `next()` aktuelle Element aus der `Collection`
- zeigt dann auf den nachfolgnden Knoten falls nicht vorhanden: `IllegalStateException`

Iteratoren


```
Iterator<String> iter = liste.iterator();
while (iter.hasNext()) {
      String stadtteil = iter.next();
      System.out.println(stadtteil);
}
```

Iteratoren



`hasNext()`	`true`	`true`	`true`	`false`
`next()`	Elementtyp-Objekt A	Elementtyp-Objekt B	Elementtyp-Objekt C	`NoSuch-Element-Exception`
`remove()`	entfernt Objekt A	entfernt Objekt B	entfernt Objekt C	`Illegal-State-Exception`

Beispiel: Nummer in Telefonliste finden


```java
/**
 * Es soll nach einem Namen gesucht werden. Ergebnis: Telefonnummer oder null,
 * falls nicht gefunden.
 */
public String getNummer(String name) {
  // "Traditionelle" for-Schleife
  for (TelefonlistenEintrag eintrag : eintraege) {
    if (eintrag.getName().equals(name)) {
      return eintrag.getNummer();
    }
  }

  // Iterator mit while-Schleife
  Iterator<TelefonlistenEintrag> it =
      eintraege.iterator();
  while (it.hasNext()) {
    TelefonlistenEintrag eintrag = it.next();
    if (eintrag.getName().equals(name)) {
      return eintrag.getNummer();
    }
  }

  // Iterator mit for-Schleife
  for (it = eintraege.iterator(); it.hasNext();) {
    TelefonlistenEintrag eintrag = it.next();
    if (eintrag.getName().equals(name)) {
      return eintrag.getNummer();
    }
  }

  return null;
}
```

Übung: Iteratoren

- Durchlaufen Sie die `List<Integer> liste` (*23,42,11,12*) zweimal
  - einmal mit einer `for`-Schleife
  - einmal mit einer `while`-Schleife
- Verwenden Sie in beiden Fällen `Iteratoren`

Vergleichen

Motivation

- häufige Anforderung (nicht nur bei `Collections`): Vergleichen zweier Objekte
- bisher kennengelernt: `equals()`
  - nicht vergessen: pro Klasse überschreiben
  - nur Vergleich "gleich" oder "ungleich"
  - fehlt: größer oder kleiner
- Möglichkeit 1: eigene Methode implementieren
  - z.B. `boolean istGroesser(Bruch andererBruch)`
  - Nachteil: keine Standardisierung
- besser: Standard-Interface `Comparable<T>`

Interface Comparable

- definiert in `java.lang`
- Generisches Interface
  - d.h. automatische Definition von `Comparable<Typ>`
- einzige Methode:
  `int compareTo(Typ anderesObjekt)`
- Aufrufer-Objekt (`this`) wird mit einem Objekt derselben Klasse (`anderesObjekt`) verglichen
- es wird ein **`int`**-Wert wird als Ergebnis zurückgeliefert:
  - > 0: wenn `this > anderesObjekt`
  - < 0: wenn `this < anderesObjekt`
  - 0: wenn `this` und `anderesObjekt`gleich sind

Beispiel: Sortieren der Telefonlisteneinträge


```java
/**
 * Ein Eintrag in einer Telefonliste besteht aus einem Namen und einer Nummer.
 */
public class TelefonlistenEintrag implements
    Comparable<TelefonlistenEintrag> {
  /**
   * Name des Eintrags.
   */
  private final String name;

  /**
   * Telefonnummer des Eintrags.
   */
  private final String nummer;

  @Override
  public int compareTo(TelefonlistenEintrag andererEintrag) {
    return getName().compareTo(andererEintrag.getName());
  }
```

Implementieren des Interfaces

diese Methode muss implementiert werden

String implementiert ebenfalls das Interface, Wiederverwendung

Interface Comparable

- Anwendung
  - viele Einsatzmöglichkeiten
  - z.B. Sortieren: `Collections.sort(eintraege);`
- Beispiel

```
System.out.println(telefonliste);
telefonliste.sortieren();
System.out.println(telefonliste);
```

- liefert

Mehmet Scholl: 1121718, Ikke Häßler: 736712027, Zizu Zidane: 674237423
Ikke Häßler: 736712027, Mehmet Scholl: 1121718, Zizu Zidane: 674237423

Interface Comparator

- manchmal `Comparable` nicht ausreichend
  - Sortieren von Objekten einer Klasse, die nicht das Interface **`Comparable`** implementieren
  - Implementierung verschiedener Sortierkriterien
- Lösung
  - Definition einer Klasse, die das Interface `Comparator<ImplementingClass>` (Package: `java.util`) implementiert mit der einzigen Methode

    ```
    public int compare(Typ objekt1, Typ object2)
    ```

  - Rückgabewerte wie bei `compareTo`

Beispiel: Komparator für Telefonliste


```java
/**
 * Komparator für Telefonlisten-Einträge.
 */
public class TelefonlisteEintragComparator implements
    Comparator<TelefonlistenEintrag> {

  @Override
  public int compare(TelefonlistenEintrag eintrag1,
      TelefonlistenEintrag eintrag2) {
    return eintrag1.compareTo(eintrag2);
  }

}
```

- Sortieren

```java
  /**
   * Sortieren der Einträge nach Name
   */
  public void sortieren() {
    // Verwendung des Interfaces Comparable in TelefonlistenEintrag
    Collections.sort(eintraege);

    // Verwendung eines Komparators
    Collections.sort(eintraege,
        new TelefonlisteEintragComparator());
  }
}
```

Übung: Vergleichen

- Erweitern Sie die Klasse Bruch, sodass sie das Interface `Comparable` implementiert

```
/**
 * Ein Bruch besteht aus einem Zähler und einem Nenner.
 */
class Bruch{

  /**
   * Zähler.
   */
  int zaehler;

  /**
   * Nenner.
   */
  int nenner;
```

Menge

Motivation

- Erinnerung:
  - Container aus Interface `List` können das gleiche Element mehrfach beinhalten (mit unterschiedlichem Index)
- manchmal gewünscht:
  - jedes Element nur einmal in Container
  - mathematische Bezeichnung: Menge
  - Umsetzung in Java: Interface `Set`

Interface Set

- ebenfalls vom Interface `Collection` abgeleitet
  - wie das Interface `List`
- Erweiterung der `Collection`-Schnittstelle
  - zusätzliche Anforderung an die implementierenden Klassen
  - eine Duplikate zulassen
  - zu keinem Zeitpunkt darf es zwei Element-Objekte `x` und `y` geben, für die `x.equals(y) == true` gilt!
- Einträge sind nicht geordnet

Achtung Aufpassen

- Achtung
  - sollten Element-Objekte "von außen" so verändert werden, dass `x.equals(y)` eintritt, gerät ein Set in einen undefinierten Zustand!

Referenzimplementierung: HashSet

- Klasse `HashSet` implementiert des Interface `Set`
- wieder: Verwendung von generischen Typen
  - `HashSet<Elementtyp>`

Beispiel: Anzahl verschiedener Wörter in Text


```java
/**
 * Liefert die Anzahl der Wörter in einem Text. Identische Wörter werden nicht
 * doppelt gezählt.
 */
public class WoerterZaehlen {
  /**
   * Berechne Anzahl Wörter in Text.
   */
  public int zaehleWoerter(String text) {
    // Menge erzeugen
    Set<String> wordSet = new HashSet<String>();
    // Text bei Leerzeichen auftrenne -> regulärer Ausdrücke (siehe PM2)
    for (String word : text.split("\\s+")) {
      wordSet.add(word);
    }
    // Anzahl Wörter = Anzahl Elemente in Menge.
    return wordSet.size();
  }

  /**
   * Programmeinstieg.
   */
  public static void main(String[] args) {
    WoerterZaehlen woerterZaehlen = new WoerterZaehlen();
    String text =
        "Wenn Fliegen fliegen fliegen Fliegen Fliegen     nach.";
    System.out.format("Text: '%s'.\n", text);
    System.out.format("Anzahl Wörter in Text: %d.\n",
        woerterZaehlen.zaehleWoerter(text));
  }
}
```

Map

Motivation

- Informationen lassen sich häufig so darstellen:
  - Menge von Schlüsseln (z.B. String oder Zahl)
  - zu jedem Schlüssel ein Wert (kann auch Objekt sein)
- Liste und Menge für diese Anforderung noch optimal geeignet
- besser
  - Wörterbuch oder Map/Abbildung
  - Java: Interface `Map`

- Hinweis: falls Schlüssel = aufsteigender Zahlenwert → Array oder Liste auch geeignet
  - eindeutige Abbildung: Index → Element
  - Beispiele
    - `myArray[0]`
    - `myList.get(5);`

Interface Map

- Map-Idee: Verallgemeinerung des Indextyps
  - Index = Schlüssel ("key")
  - Element = Wert ("value")
  - Schlüssel und Wert können jeweils beliebigen Datentyp haben
  - eindeutige Abbildung: Schlüssel→ Wert
  - Speicherung von Paaren (Schlüssel, Wert)
- Beispiel Telefonliste:
  - Name: Schlüssel
  - Nummer: Wert

Referenzimplementierung HashMap

- Einträge sind nicht geordnet
- Schlüssel und Werte müssen Referenztypen sein
- wieder: Verwendung von generischen Typen

```
HashMap<KeyType,ValueType>
```

Wichtige Methoden

`public ValueType put(KeyType key, ValueType value)`

- speichert den Wert `value` unter dem Schüssel `key` in der `HashMap`
- Rückgabe: ein evtl. vorhandener alter Wert oder `null`

`public ValueType get(Object key)`

- liefert den unter dem Schlüssel `key` gespeicherten Wert
- oder `null`, falls der Schlüssel in der `HashMap` nicht vorkommt

`public ValueType remove(Object key)`

- löscht den unter dem Schlüssel `key` gespeicherten Eintrag (Schlüssel, Wert) und liefert den Wert
- falls der Schlüssel `key` in der `HashMap` nicht vorhanden ist, passiert nichts und `null` wird zurückgegeben

Beispiel: Telefonliste mit Map


```java
/**
 * Eine Telefonliste verwaltet Einträge von Kontakten.
 */
public class TelefonlisteMap {
  /**
   * Die Liste der Einträge wird als Map verwaltet.
   **/
  private Map<String, String> eintraege =
      new HashMap<String, String>();

  /**
   * Es sollen Telefonnummern und dazugehörige Namen in die Liste eingetragen
   * bzw. verändert werden.
   */
  public void eintragHinzufuegen(String name, String nummer) {
    eintraege.put(name, nummer);
  }

  /**
   * Es soll nach einem Namen gesucht werden. Ergebnis: Telefonnummer oder null,
   * falls nicht gefunden.
   */
  public String getNummer(String name) {
    return eintraege.get(name);
  }

  /**
   * Es soll ein Eintrag aus der Liste gelöscht werden.
   */
  public void eintragEntfernen(String name) {
    eintraege.remove(name);
  }
```

Gleichheit

- alle Referenztypen können als Schlüssel (`key`) verwendet werden
- alle `Collections`-Klassen verwenden die `equals()`-Methode eines Objekts zur Feststellung der Gleichheit
  - eine geeignete `equals()`-Implementierung für den Typ des Schlüssels muss vorhanden sein!
  - Default der Klasse `Object:` Test auf Identität der Objekte
  - Telefonbuch: `equals()`-Methode der Klasse `String` wird verwendet
    - also keine Redefinition nötig
- viele `Collection`-Klassen verwenden auch die `hashCode()`-Methode zur Optimierung
  - Vorsicht bei Redefinition

Umwandlung HashMap


```
public Set<KeyTyp> keySet()
```

- liefert die Menge aller Schlüssel einer `Map` als `Set` (generischer Typ!)

```
public Collection<ValueTyp> values()
```

- liefert die Werte einer `Map` als allgemeine `Collection`

```
public Set<Map.Entry<KeyTyp,ValueTyp>> entrySet()
```

- liefert alle Einträge der `Map` als Menge mit Elementtyp `Map.Entry<KeyTyp,ValueTyp>`
- Klasse `Map.Entry` definiert die beiden Methoden
- `KeyTyp getKey()`: liefert den Schlüssel des Eintrags
- `ValueTyp getValue()`: liefert den Wert des Eintrags

Interface Map

- Umwandlungen in andere Container liefern nur eine Sicht auf die `Map`, d.h.
  - Änderungen an der `Collection` wirken sich auf die `Map` aus
  - sehr effiziente Erzeugung, weil keine Daten kopiert werden
- Beispiele für `Collection`-Sichten auf eine `Map`
  - Namen und Nummern jeweils getrennt sammeln:

```
Set<String> namen = eintraegeMap.keySet();
Collection<String> nummern = eintraegeMap.values();
```

- Das gesamte Telefonbuch ausgeben:

```
for (Map.Entry<String, String> eintrag: eintraegeMap.entrySet()){
        System.out.format("%s: %s\n", eintrag.getKey(),
        eintrag.getValue());
}
```

Übung: Menge und Map

- Schreiben Sie eine Klasse `SchweizerNummernKontoBank`
- eine solche Bank verwaltet Konten
- ein Konto besteht aus einer Kontnummer (`int`) und einem Kontostand (`float`)
- Schreiben Sie Methoden
  - zum Hinzufügen eines Kontos
  - zum Setzen des Kontostands eines Kontos
  - zum Auslesen des Kontostands eines Kontos

Collections-Operationen

Collections

- Sammlung von Algorithmen als statische Methoden der Klasse `java.util.Collections` (ähnlich wie Klasse `Arrays`)
- Beispielauswahl:

```
static void sort(List<Elementtyp> list)
```

  - Liste `list` nach aufsteigender Elementgröße sortieren
  - Elementtyp muss das Interface `Comparable` implementieren

```
static Elementtyp max(Collection<Elementtyp> coll)
```

  - liefert das größte Element der Collection
  - Elementtyp muss das Interface `Comparable` implementieren

```
static int binarySearch(List<Elementtyp> list, <Elementtyp> key)
```

  - `key` in der *sortierten* Liste `list` suchen (Vergleiche: `Comparable`)
  - Ergebnis
    - >= 0 Index der ersten Fundstelle von `key` in `list`
    - < 0 `key` in `list` nicht gefunden

Zusammenfassung

- Collections-Framework
- Verkettete Liste und Array-Liste
- Iteratoren
- Vergleichen
- Menge
- Map
- Collections-Operationen