LUDWIG-
MAXIMILIANS-
UNIVERSITÄT
MÜNCHEN

Javakurs für Fortgeschrittene

Weitere Datenstrukturen

Kyrill Schmid
Lehrstuhl für Mobile und Verteilte Systeme

Heutige Agenda

Datenstrukturen

- Motivation
- Die Collection API
  - Interface `Collection<E>`
- Überblick

Konkrete Datenstrukturen

- Listen
- Mengen
- Schlangen
- Stack
- Abbildungen

Praxis:

- Black Jack (17 und 4)

Lernziele

- Weitere und komplexere Datenstrukturen kennenlernen
- Listen verstehen und anwenden/implementieren können
- Konzepte von Schlangen und Abbildungen verstehen

Motivation

Geeignete Strukturierung der Daten ermöglicht erst eine effiziente Verarbeitung:

- **Die Datenstruktur**:
  - Modell zur Speicherung und Verwaltung von Daten
    - Bsp.: Arrays, Listen, Schlangen, Bäume, usw.
  - Beschreibt die Organisation der Daten, also auch die Operationen
    - Beschreibung durch **Interfaces** (konkrete Implementierung verborgen)
    - Prinzip: Trennung von Beschreibung und Umsetzung
- Konkrete Implementierungen von Datenstrukturen beeinflussen **nur** Metriken wie Performance, Speicherplatzbedarf, Thread-Sicherheit, etc.

Datenstruktur	Interface	Implementierung
Liste	`List<E>`	`ArrayList`, `LinkedList`, …
Menge	`Set<E>`	`HashSet`, `TreeSet`, …
Abbildung	`Map <K,V>`	`HashMap`, `TreeMap`
Schlange	`Queue<E>`	`ArrayDeque`, `DelayQueue`

Die Collection API

Die Collection API regelt den Umgang mit den wichtigsten Datenstrukturen durch 4 Prinzipien:

- **Schnittstellen**: Definition der Datenstruktur
  - Bsp.: `List`, `Map`, `Set`, …
- **Abstrakte Basisklassen**: Minimale Anzahl an abstrakten Methoden
  - `AbstractList`, `AbstractSet`, …
- **Konkrete Klassen**: Implementieren (u. erweitern) die Grundoperationen auf den Datenstrukturen
  - Bsp.: `ArrayList`, `LinkedList`, …
- **Algorithmen**
  - Sortieren, Suchen, etc.

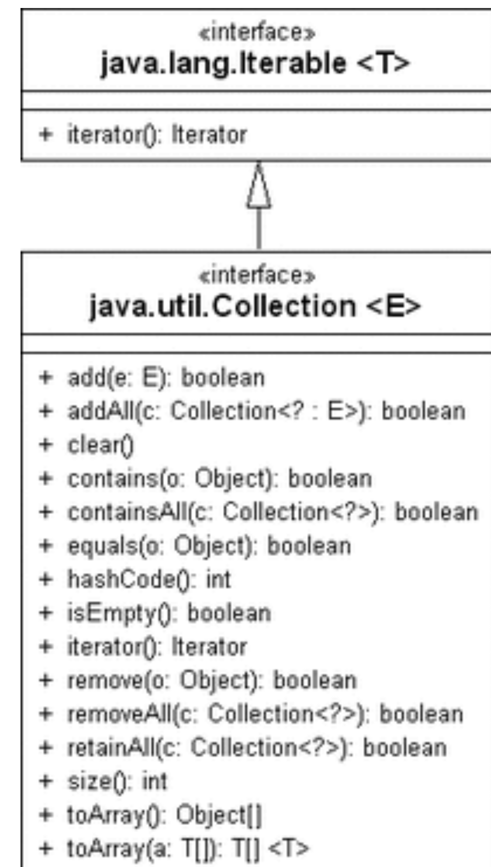
Die Collection API definiert:

- das Interface `Collection<E>` für alle Arten von Sammlungen (Listen, Mengen, Schlangen)
- Das Interface `Map<K,V>` für endliche Abbildungen

Interface `Collection`

`Collection` erweitert das Interface `Iterable` und definiert die Basis-Methoden für alle Arten von (endl.) Sammlungen :

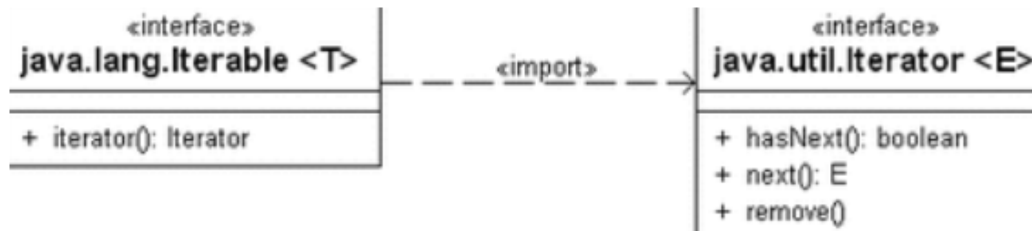
- **Einfügen**:
  - `add(element)`
  - `addAll(collection)`
- **Erfragen**:
  - `contains(object)`
  - `equals(object)`
  - `size()`
- **Löschen**:
  - `remove(object)`
  - `removeAll(collection)`
- …



Quelle: http://openbook.rheinwerk-verlag.de/javainsel9/bilder/collectionuml.gif

Einschub: Interface Iterable

Das Interface `Iterable` schreibt die Methode `iterator()` vor, die einen `java.util.Iterator` liefert.



Daher kann ein Objekt, welches `Iterable` implementiert das Ziel einer `for-each-`Schleife sein.

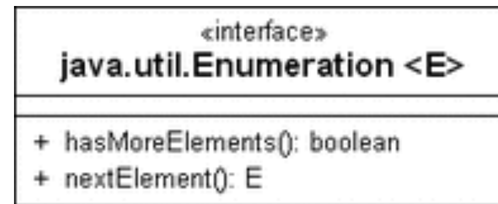
- Alle Collection-Klassen implementieren `Iterable` und damit kann eine `for-each-`Schleife immer über diese Sammlungen laufen:

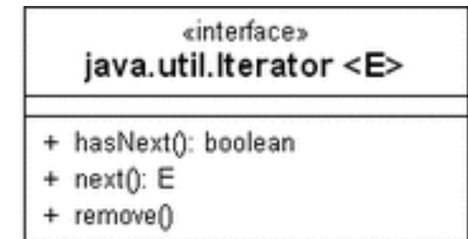
```
Collection<String> c = new LinkedList<String>();
for ( String s : c )
  System.out.println( s );

ArrayList<Double> myList = new ArrayList<Double>();
for (double d : myList)
   //do something…
```

Iterator

Mit einem Iterator können wir sequentiell durch die Daten einer Collection wandern.

- Iface Iterator bzw. Enumeration





Quelle: http://openbook.rheinwerk-verlag.de/javainsel9/bilder/enumerationiteratoruml.gif

- Mit Aufruf iterator() erhalten wir den entspr. Iterator
- Mit der statischen Methode enumeration(Collection<T> c) aus dem Paket java.util.Collections erhalten wir den Enumeration<T>
- Mit Aufruf von hasNext() bzw. hasMoreElements() können wir prüfen, ob es ein weiteres Element gibt.
- Mit Aufruf von next() bzw. nextElement() erhalten wir ein weiteres Element der Datenstruktur
- Übergehen wir ein false von hasNext()/hasMoreElements(), folgt eine NoSuchElementException.

```
Collection<String> myNameSet = new TreeSet<String>();
Collections.addAll(myNameSet, „Albert", „Einstein", „Alan", „Turing" );

for ( Iterator<String> iterator = myNameSet.iterator(); iterator.hasNext(); )
  System.out.println( iterator.next() );
```

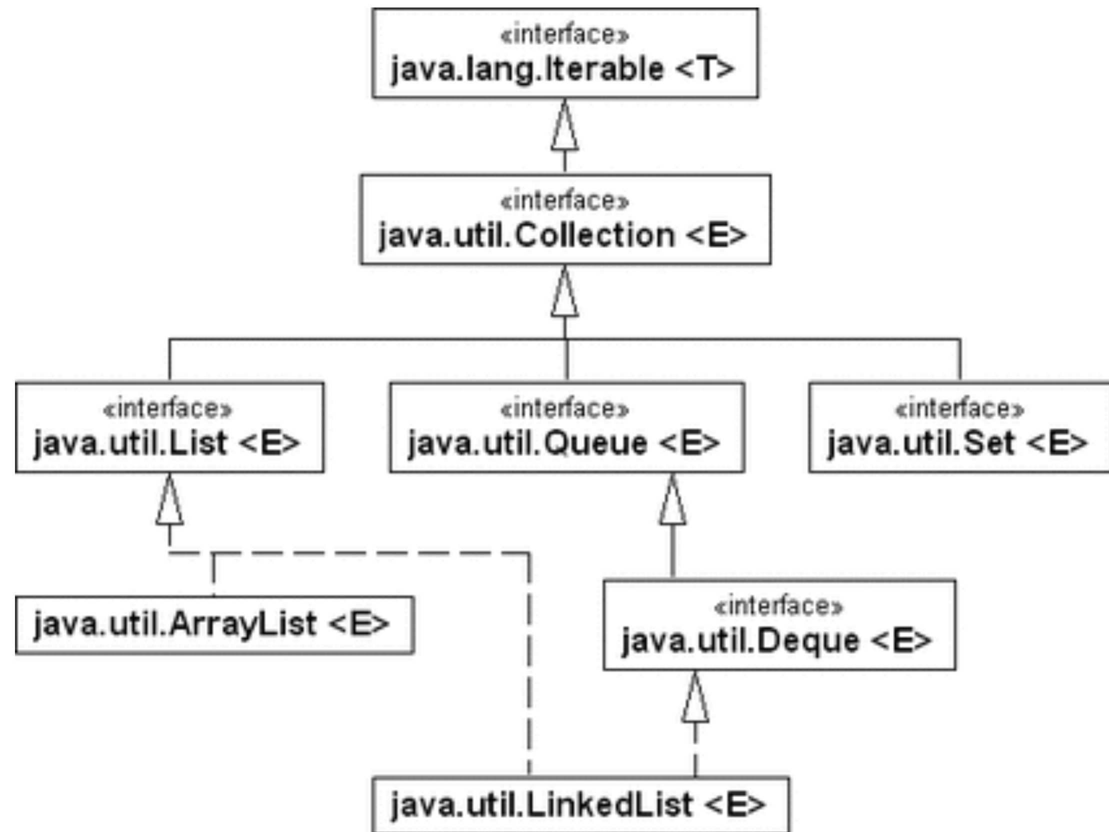
Interfaces, die Collection erweitern

Um nun die generische Schnittstelle der endlichen Sammlung (`Collection`) zu spezifizieren, existieren eine Reihe weitere Schnittstellen für konkretere Datenstrukturen:

- **List**
  - Sequenzen
- **Queue**
  - Schlangen
- **Set**
  - Mengen

Map<K,V>

- Abbildungen
- Separat zur Collection



Quelle: http://openbook.rheinwerk-verlag.de/javainsel9/bilder/collectionapiuml.gif

Listen (Allgemein)

Eine Liste

- Datenstruktur: Sequenz von Daten
  - Elemente haben feste Reihenfolge
- Generisch einsetzbar: Typspezifizierung
- Punktueller Zugriff über `get(index)`
- Sequentieller Durchlauf über `Iterator`
- Beschrieben durch das IFace `java.util.List`

Meine Liste

- Item 1
- Item 2
- Item 3
- …
- ..
- ..
- .

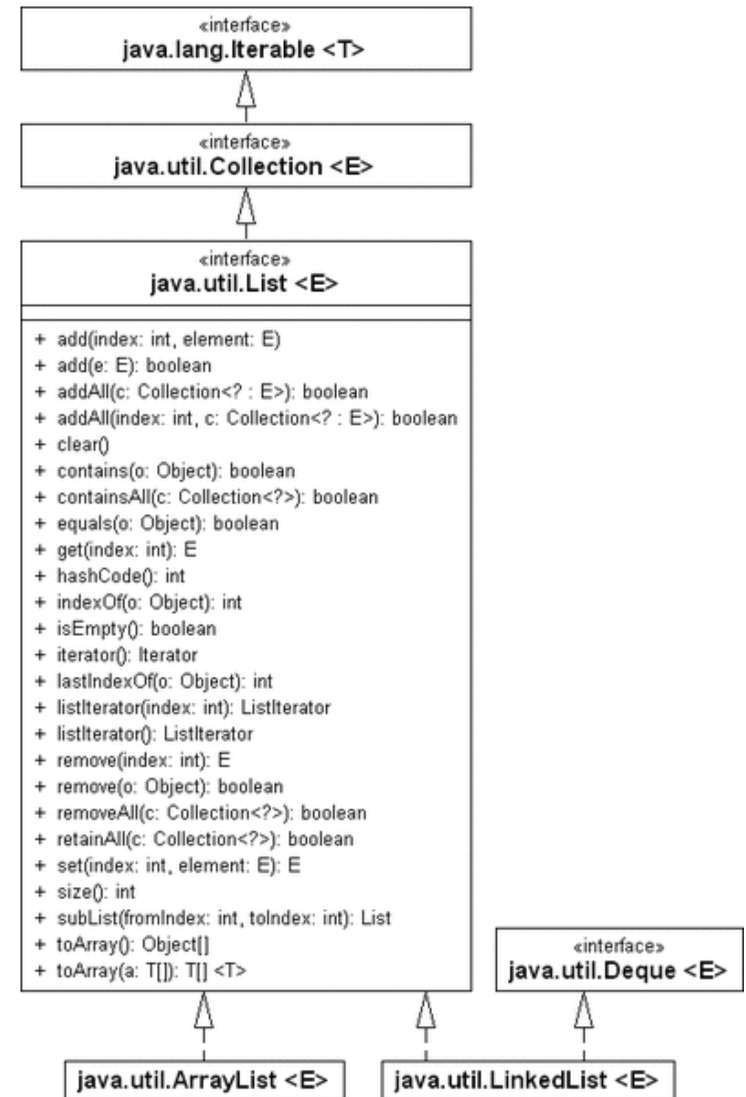
Aber: Verschiedene Ausprägungen für untersch. Zwecke:

- `ArrayList`, `Vector`, `LinkedList`, …
- Unterschied in:
  - Realisierung, Performance, Speicherplatzbedarf, Thread-Sicherheit
- Bsp.:
  - `ArrayList`: Realisiert durch Array: Schneller Zugriff auf ein Element über dessen Index
  - `LinkedList`: Realisiert durch Verkettung der Elemente: Schnelles Einfügen/Löschen von Elementen v.a. in der Mitte

Interface `List`

`List` erweitert das Interface `Collection` und definiert die Basis-Methoden für Listen:

- **Einfügen**:
  - `add(index,element)`
  - `add(element)`
  - `addAll(collection)`
- **Erfragen**:
  - `contains(object)`
  - `equals(object)`
  - `get(index)`
- **Löschen**:
  - `remove(object)`
  - `remove(index)`
  - `removeAll(collection)`
- …



Quelle: http://openbook.rheinwerk-verlag.de/javainsel9/bilder/listuml.gif

Die Klasse `LinkedList`

Implementierung verketteter Listen durch `LinkedList<E>`

- Erweiterte Methoden:
  - `addFirst(object)`
    - Element am Anfang der Liste hinzufügen
  - `addLast(object)`
    - Element am Ende der Liste hinzufügen
  - `getFirst()`
    - Erstes Element zurückliefern
  - `getLast()`
    - Letztes Element zurückliefern
  - `removeFirst()`
    - Erstes Element Löschen
  - `removeLast()`
    - Letztes Element löschen
- Einfügen/Löschen an beliebiger Stelle über den `ListIterator`

ListIterator

Die Schnittstelle `ListIterator` erweitert das IFace `Iterator` um spezielle List-Operationen:

- `add(element)`
  - Fügt Element vor den Positionszeiger ein
- `next()`
  - Liefert Element am Positionszeiger zurück und erhöht den Positionszeiger
- `remove()`
  - Entfernt das letzte Element, das von `next()` zurückgeliefert wurde
- `set(element)`
  - Ersetzt das letzte von `next()` zurückgegebene Element

Mittels der Methode `listIterator()` erhalten wir den entsprechenden `ListIterator` auf einer Liste, welcher auf das erste Element zeigt:

```
List<String> list = new ArrayList<String>();
Collections.addAll( list, "b", "c", "d" );

ListIterator<String> it = list.listIterator();

it.add( "a" );
System.out.println( list );

it.next();
it.remove();
System.out.println( list );

it.next();
it.set( "C" );
System.out.println( list );
```

Liste:[b,c,d]

Zeiger => b
Einfügen vor b: [a,b,c,d]

Rückgabe b, Zeiger: c
Löschen von b: [a,c,d]

Rückgabe c
Ändern von c: [a,C,d]

ListIterator

Anmerkungen:

- `remove()` und `set()` nur zulässig, wenn ein `next()` zuvor erfolgt ist
  - Sonst: `IllegalStateException`
- Für die andere Richtung (rückwärts) gibt es die analogen Methoden:
  - `hasPrevious()`
  - `previous()`
- Daneben gibt es noch die Methoden `nextIndex()` und `previousIndex()`
  - Geben den Index des Elements wieder, welches bei einem `next()` bzw. `previous()` zurückgeliefert werden würde.
    - Am Ende der Liste entspricht dies der Listengröße

Algorithmen auf Listen

Die Klasse `java.util.Collections` bietet zahlreiche (Hilfs-)Algorithmen für Listen:

- Listenelemente Sortieren, mischen, umdrehen, ersetzen, …
- Extremwerte bestimmen
- …

Hilfsklasse:

- Privater Konstruktor
- Alle Methoden sind statisch

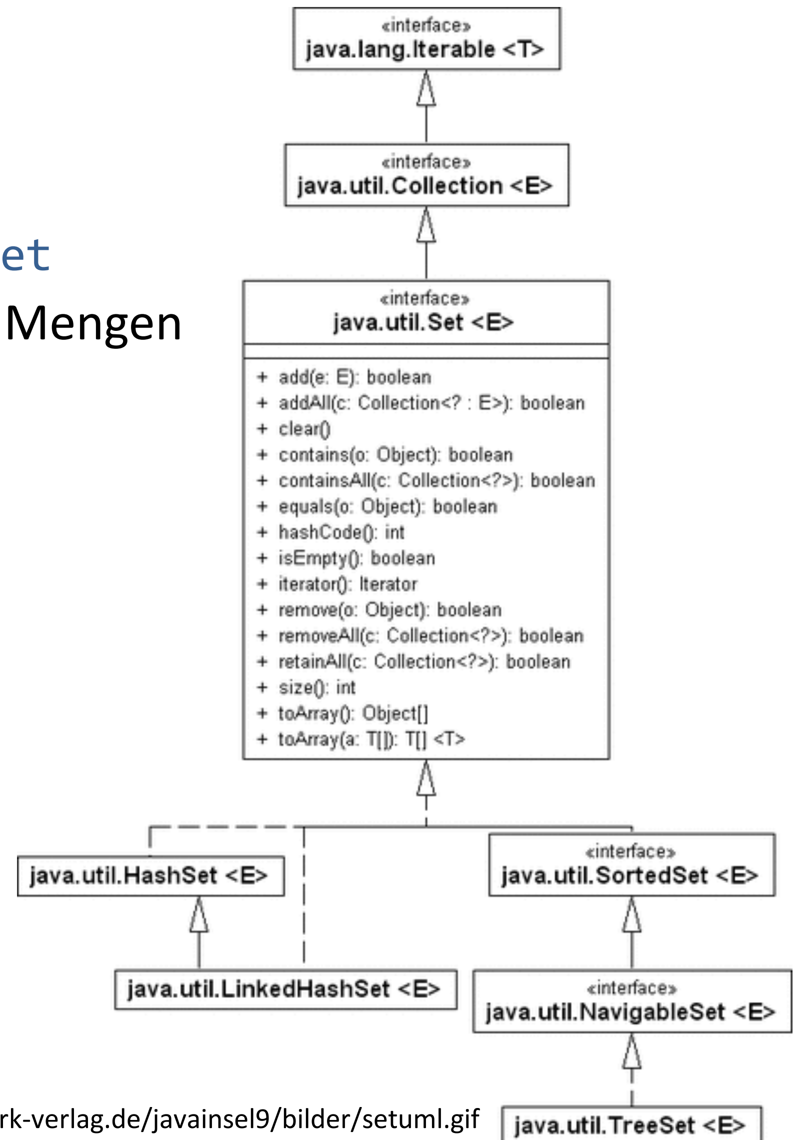
```
Beispiele:
myList = new LinkedList<>(myCollection);

// Sortieren:
Collections.sort(myList);

// Extremwerte:
Collections.max(myList);
Collections.min(myList);
```

Eine Menge

- Ungeordnete Sammlung von Elementen
  - Jedes Element darf nur einmal vorkommen!
- Beschrieben durch das IFace `java.util.Set`
- Operationen: Schnitt und Vereinigung von Mengen
- Beliebte implementierende Klassen:
  - `HashSet, TreeSet, LinkedHashSet, …`

Wichtige Methoden:

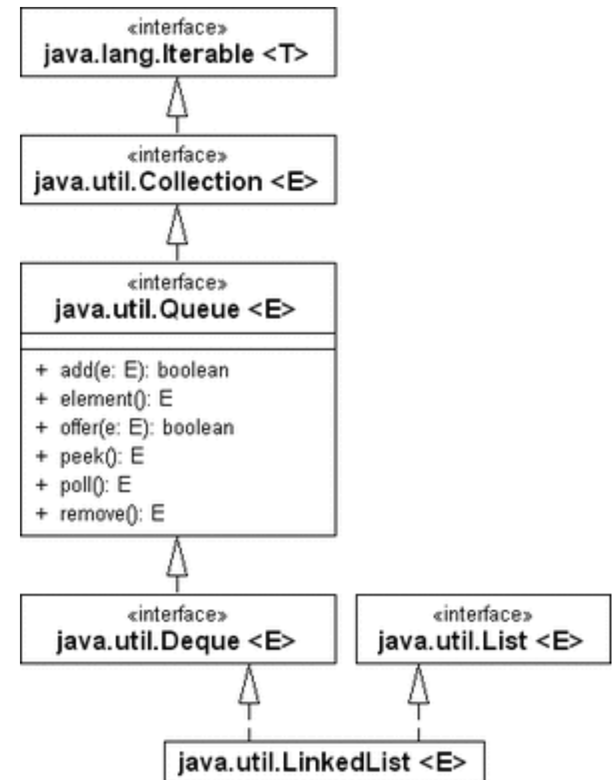
- `add(element)`
  - Einfügen falls Element noch nicht vorhanden
    - Liefert true bei Erfolg
- `clear()` löscht die Menge
- `contains(element)`
  - Prüft ob Element vorhanden



Quelle: http://openbook.rheinwerk-verlag.de/javainsel9/bilder/setuml.gif

Eine Schlange

- I.d.R. zur Implementierung von Warteschlangen nach dem FIFO Prinzip:
  - Interface `java.util.Queue`
- Implementierende Klasse: `LinkedList`



Quelle: http://openbook.rheinwerk-verlag.de/javainsel9/bilder/queueuml.gif

Wichtige Methoden:

- Einfügen:
  - `add(element) // Mit Exception`
  - `offer(element)  // False`
- Erfragen:
  - `element() // Mit NoSuchElementException`
  - `peek() //  null`
- Löschen:
  - `remove() // Mit Exception`
  - `poll() // null`

Der Stack

- Repräsentiert einen Stapel- bzw. Keller-Speicher
  - LIFO (Last-in-First-out) Datenstruktur
- Klasse: `java.util.Stack<E> extends Vector<E>`
- Implementiert u.a.: `Iterable<E>`, `Collection<E>` und `List<E>`

Wichtige Methoden:

- Einfügen:
  - `push(element) // Element auf den Stack legen`
- Erfragen:
  - `empty() // Testet, ob Elemente auf dem Stack liegen`
  - `peek() // Oberste Element wird gelesen, aber` **`nicht`** `abgeholt!`
  - `search(element)  // gibt die Position eines Elements auf dem`
`                   // Stack (von der Spitze an) wieder`
- Erfragen und Löschen:
  - `pop() // Oberste Element wird gelesen` **`und`** `abgeholt!`

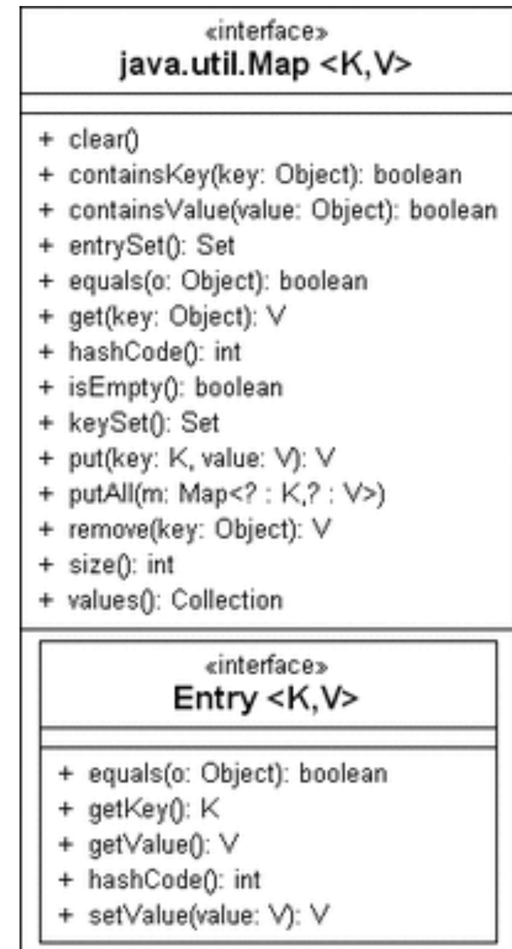
Abbildungen (Assoziative Speicher)

Eine Abbildung

- Verbindet einen Schlüssel K mit einem Wert V
- Interface: `Map<K,V>`
- Implementierende Klassen:
  - `HashMap`, `TreeMap`, `LinkedHashMap`, …

Wichtige Methoden:

- Einfügen:
  - `put(key,value)`
  - `putAll(map)`
- Erfragen:
  - `get(key)`
  - `containsKey(key)`
  - `containsValue(value)`
- Löschen:
  - `remove(key)`
  - `clear()`



Quelle: http://openbook.rheinwerk-verlag.de/javainsel9/bilder/mapentryuml.gif

Beispiel zu Abbildungen

```
HashMap<String,Color> h = new HashMap<String,Color>();

h.put( „rot", Color.red);
h.put( „gruen", Color.green);
h.put( „blau", Color.blue);
h.put( „gelb", Color.yellow);

for ( String elem : h.keySet() )
  System.out.println( elem );
```

Ausgabe:

```
rot
gruen
blau
gelb
```

Aufgabe zu Collections

Spielen Sie Black Jack (17 und 4):

Erstellen Sie ein neues Package blackjack und schreiben Sie 2 Klassen:

- **Player**:
  - Hat einen Namen und verwaltet eine Collection<Cards> mit den Karten, die der Spieler auf der Hand hat.
  - Der Spieler kann eine Karte vom Dealer fordern und diese auf die Hand nehmen
  - Außerdem kann der Spieler den Wunsch zum Karteziehen äußern
    - Nehmen Sie als einfachste Strategie an, dass der Spieler solange eine Karte fordert, bis die Summe seiner Karten > 17 ist.
      - Nehmen Sie als Vereinfachung an, dass bei Black Jack folgende Punkte gelten:

        Zahlen (2-9), Bube, Dame, König = 10 und Ass = 11

        Es gewinnt der Spieler, der am Ende am nächsten zu 21, aber nicht darüber ist.
- **Dealer**:
  - Muss zu beginn 52 Karten zur Verfügung stellen und diese mischen.
    - Verwalten Sie den Kartentyp als Enum Cards
  - Zudem muss der Dealer auf Wunsch eines Spielers die oberste Karte dem jeweiligen Spieler abgeben.

Legen Sie einen Dealer und 2 Spieler an, die abwechselnd Karten vom Stapel fordern.

Will kein Spieler mehr eine Karte ziehen, müssen die Spieler ihre Karten zeigen und den Punktestand bekannt geben. Geben Sie dies (Karten und Punktestand der Spieler) auf der Konsole aus. Verwenden Sie geeignete Collections!