hochschule mannheim

Fakultät für Informatik

Programmierung 2 (PR2)

07 - Collections

Prof. Dr. Frank Dopatka

Hochschule Mannheim University of Applied Sciences

hochschule mannheim

Container & Collections: Allgemeine Klassifizierung

Was ist eine Containerklasse?

- Sie dient der Speicherung und Verwaltung von Objektmengen.
  - Einfachstes Beispiel: Array
  - Selbst programmierte Beispiele:
    Listen, FiFo/LiFo, Ringpuffer, Hashes, Tree, Trie, Graphen
- Das Java Collection Framework (JCF) im Paket **`java.util`** beinhaltet eine große Anzahl vordefinierter Containerklassen.
  - Die Anwendungsklassen können diese Klassen benutzen oder von ihnen erben.
  - Die Idee besteht in der Definition gemeinsamer Schnittstellen.
  - Diese werden, je nach Verwendungszweck, unterschiedlich implementiert.

hochschule mannheim

Was ist eine Containerklasse?

- Es gibt 2 Arten von Containern:
  - Collection als reine Sammlung von Objekten,
    z.B. Studentenliste
  - Map als Abbildung zwischen Objekten.
    z.B. Telefonverzeichnis
- Es gibt 2 Arten von Collections:
  - Eine Sequenz in Form einer List:
    - Legt eine bestimmte Reihenfolge ihrer Elemente fest.
    - Erlaubt, daß dasselbe Element auch mehrfach auftritt.
  - Eine Menge in Form eines Set:
    - Ein Element kann nicht mehrfach vorkommen.
    - Es ist keine bestimmte Reihenfolge der Elemente definiert.

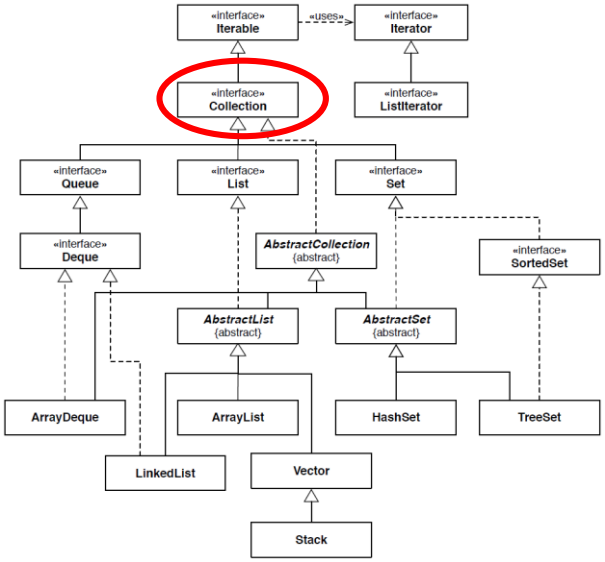
hochschule mannheim

Was ist eine Container-Klasse?

- In einer Containerklasse ist die Zahl der Elemente ist nicht im Voraus festgelegt auch nicht begrenzt.
- Objekte von Containerklassen können dynamisch zur Laufzeit wachsen im Gegensatz zu Arrays, deren Größe bei der Erzeugung festgelegt werden muss.
- Im Java Collection Framework sind Sammlungen auf 4 verschiedene Arten realisiert:
  - Mit Hilfe von Arrays als **`ArrayList`**, **`Vector`**,
  - als verkettete Listen als **`LinkedList`**,
  - mit Hilfe sortierter Bäume als **`TreeSet`** und
  - mit Hilfe von Hashing als **`HashSet`**.
- Für Mengen kleiner, nicht-negativer ganzer Zahlen gibt es daneben eine spezielle Bit-Vektor-Implementierung: **`BitSet`**
  - **`BitSet`** implementiert die Schnittstelle Collection nicht!

hochschule mannheim

Klassengeflecht von Collection



Basis-Methoden des Interfaces Collection<E>

- Einfügen eines Elements x in die Sammlung.
  - bei Mengen: kein Mehrfacheintrag
  - bei Listen: ggf. mit Angabe der Einfügestelle
- Löschen eines Elements x aus der Sammlung.
- Löschen aller Elemente der Sammlung.
- Abfrage, ob x Element der Sammlung ist.
- Bestimmung der Anzahl der Elemente in der Sammlung.
- Abfrage, ob die Sammlung leer ist.
- Durchlaufen aller Elemente der Sammlung.
  - bei Mengen: Reihenfolge ist unbestimmt

hochschule mannheim

Basis-Methoden des Interfaces Collection<E>

- **`boolean add(E e)`**
  Fügt ein Element e hinzu.
- **`boolean addAll(Collection<? extends E> c)`**
  Fügt alle Elemente aus c hinzu.
- **`boolean remove(Object o)`**
  Entfernt das gegebene Objekt o, falls vorhanden.
- **`boolean removeAll(Collection<?> c)`**
  Entfernt alle Elemente, die in der gegebenen Collection c enthalten sind.
- **`boolean retainAll(Collection<?> c)`**
  Entfernt alle Elemente, die nicht in der gegebenen Collection c enthalten sind.

Basis-Methoden des Interfaces Collection<E>

- **`void clear()`**
  Entfernt alle Elemente.
- **`boolean contains(Object o)`**
  Prüft, ob das gegebene Objekt in der Collection enthalten ist.
- **`boolean containsAll(Collection<?> c)`**
  Prüft, ob alle Elemente der gegebenen Collection enthalten sind.
- **`int size()`**
  Gibt Anzahl der Elemente zurück.
- **`boolean isEmpty()`**
  `size() == 0`
- **`Object[] toArray()`**
  Konvertiert die Collection in ein Object-Array.
- **`T[] toArray(T[] a)`**
  Konvertiert die Collection in ein Array.

Generics

hochschule mannheim

Was ist Typsicherheit?

- Typsicherheit bezeichnet den Zustand einer Programmausführung, bei dem die Datentypen gemäß ihren Definitionen in der benutzten Programmiersprache verwendet werden und keine Typverletzungen auftreten.
- Werden dementsprechend Typfehler spätestens zur Laufzeit erkannt, spricht man von typsicheren Sprachen.
- Typsicherheit herzustellen ist Aufgabe des Compilers bzw. Interpreters.

hochschule mannheim

Was ist Typprüfung?

- Als Typprüfung bezeichnet man dabei den Vorgang, die Verwendung von Datentypen innerhalb des Typsystems zu prüfen, um etwaige Typverletzungen festzustellen.
- Hierbei müssen u.a. bei Zuweisungen die beteiligten Typen nicht notwendig identisch sein, da auch ganze Zahlen existierenden Gleitkommavariablen zugewiesen werden können.
- Bei den typisierten Sprachen gibt es solche
  - mit Typprüfungen während der Kompilierung (statisch typisiert) und
  - solche in denen Typprüfungen primär zur Laufzeit stattfinden (dynamisch typisiert).

Was ist Typprüfung?

- Java unterstützt die statische Typisierung, da z.B. die Typsicherheit von primitiven Datentypen bereits vom Bytecode-Compiler **`javac`** geprüft wird.
- Java unterstützt die dynamische Typisierung, da sie auch dynamische Typprüfungen mit dem **`instanceof`**-Operator zulässt.
- Sprachen wie JavaScript, PHP und Ruby hingegen sind vollständig dynamisch typisiert.

Das Problem der Element-Typen

- Einerseits sind Containerklassen dazu da, beliebige Elementtypen zuzulassen.
- Andererseits soll bei der Deklaration der Einfügeoperation Typ für das Argument angegeben werden.
- Lösung1 bis Java 1.4: Verwendung von polymorphen Behältern
  - Alle Elemente haben den Datentyp **`Object`**.
  - Alle Klassen sind Unterklassen von Object.
  - Die Typsicherheit geht jedoch verloren.
- Lösung 2 ab Java 5: Generische Datentypen
  - Die Schnittstellen-Deklaration einer Collection erhält den Elementtyp selbst als Parameter; diesen nennt man generischen Typ-Parameter.
  - Beispiel: **`List<T>`**
    `List<T>` ist ein generischer Typ mit **`T`** als Typ-Parameter.

hochschule mannheim

Lösung 1 mit der im Folgenden näher betrachteten ArrayList

```
import java.util.ArrayList;
public class test01 {

  @SuppressWarnings("unchecked")
  public static void main(String[] args) {
    ArrayList liste1=new ArrayList();
    liste1.add(new Integer(2));
    liste1.add(new Double(2.4));
    liste1.add(new String("Hallo"));
    liste1.add(new Hund("Bello"));

    for(Object o:liste1){
      System.out.println(o);
    }
  }
}
```

Problems | Javadoc | Declaration | Console

```
<terminated> test01 [Java Application] C:\Programme\jre\bin\java
2
2.4
Hallo
Ich bin Bello der Hund
```


hochschule mannheim

Lösung 2 mit der im Folgenden näher betrachteten generischen ArrayList

```
public static void main(String[] args) {
  ArrayList<Tier> liste1=new ArrayList<Tier>();
  liste1.add(new Integer(2));
  liste1.add(new Double(2.4));
  liste1.add(new String("Hallo"));
  liste1.add(new Hund("Bello"));
```

The method add(Tier) in the type ArrayList<Tier> is not applicable for the arguments (Integer)

hochschule mannheim

Lösung 2 mit der im Folgenden näher betrachteten generischen ArrayList

```
import java.util.ArrayList;
public class test02 {
  public static void main(String[] args) {
    ArrayList<Tier> liste1=new ArrayList<Tier>();
    liste1.add(new Hund("Bello"));
    liste1.add(new Katze("Susi"));
    liste1.add(new Hund("Hasso"));
    liste1.add(new Katze("Nicki"));

    for(Tier t:liste1){ // nur Tiere drin!
      // alle Methoden eines Tiers direkt verfügbar
      System.out.println(t.gibLaut());
    }
  }
}
```

```
Bello: wow
Susi: miau
Hasso: wow
Nicki: miau
```


hochschule mannheim

Eigenschaften von Generics

- Generics erlauben es, vom konkreten Typ zu abstrahieren.
- Der Typ einer Variable, eines Parameters, eines Rückgabewertes etc. ist selbst ein Variable, die man Typ-Variable nennt.
- Klassen und Methoden haben zusätzliche Typ-Parameter, welche die Typ-Variablen setzen.
- Der Verwender kann Typ-Parameter setzen und damit die Klassen und Methoden parametrieren.
- Klassen mit Typ-Parametern nennt man generische Typen bzw. generische Klassen.
- Methoden mit Typ-Parametern nennt man generische Methoden.

Eigenschaften von Generics

- Seit Java 7 ist die Typ-Inferenz für Generics deutlich verbessert worden:
  - In den meisten Situationen muss man den Typ-Parameter nur noch bei der Deklaration setzen.
  - Bei der Objekterzeugung kann die Angabe des Typ-Parameters durch den sogenannten diamond „<>“ ersetzt werden.

```
List<String> list = new ArrayList<>();
Map<Integer, String> map = new HashMap<>();
Set<?> set = new HashSet<>();
```


Die Collections
Listen & Sets

Zusätzliche Methoden auf List

- **`boolean addAll(int index, Collection<? extends E> c)`**
  Fügt alle Elemente von c an der gegebenen Stelle ein und verschiebt die existierenden Elemente.
- **`E get(int index)`**
  Gibt das Element an der gegebenen Position zurück.
- **`E set(int index, E element)`**
  Ersetze das Element an der gegebenen Position.
- **`void add(int index, E element)`**
  Füge an der gegebenen Position ein Element ein.

Zusätzliche Methoden auf List

- **`E remove(int index)`**
  Entferne das Element an der Position.
- **`int indexOf(Object o)`**
  Position des gegebenen Objekts als erster Treffer.
- **`int lastIndexOf(Object o)`**
  Position des gegebenen Objekts als letzter Treffer.
- **`ListIterator<E> listIterator()`**
  Liefert einen ListIterator.
- **`ListIterator<E> listIterator(int index)`**
  Liefert einen ListIterator ab der gegebenen Position.
- **`List<E> subList(int fromIndex, int toIndex)`**
  Erzeugt eine Sub-Liste beginnend bei **`fromIndex`** inklusive und endend bei **`toIndex`** exklusive.

Die Klasse ArrayList

- Eine ArrayList realisiert die Sequenz durch ein Datenfeld von Objekt-Referenzen.
  - Das Feld ist immer groß genug, um alle Elemente zu halten.
  - Seine Größe **`capacity`** wird bei **`add()`** usw. automatisch angepaßt.
    - Falls erforderlich wird eine Kopie des Feldes in ein größeres Feld angelegt.
    - Das neue Feld ist im Allgemeinen größer als nötig, um häufiges Kopieren zu vermeiden.
- Konstruktoren:
  - **`ArrayList(int initialCapacity)`**
    erzeugt eine Sequenz mit gegebener initialer Kapazität.
  - **`ArrayList()`**
    mit einer initialen Kapazität von 10.

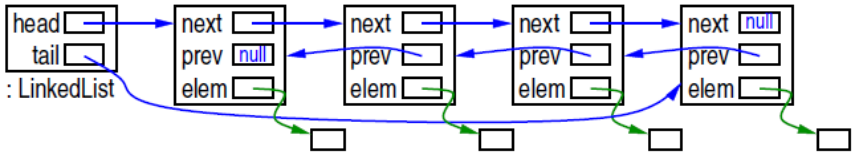
Die Klasse ArrayList: Ein einfaches Beispiel

```
List<String> list = new ArrayList<>();

list.add("Element A");
list.add("Element B");
list.add("Element C");
list.add("Element A");
list.add("Element B");

for (String element : list) {
   System.out.println(element);
}
```

```
Element A
Element B
Element C
Element A
Element B
```

Die Klasse LinkedList

- Realisierung als doppelt verkettete Liste:



- Jedes Element kennt seinen Vorgänger und Nachfolger.
- Konstruktor: **`LinkedList()`**
- Zusätzliche Operationen dieser Klasse:
  - **`addFirst()`, `addLast()`**:
    fügt vorne / hinten an
  - **`getFirst()`, `getLast()`**:
    liefert erstes / letztes Element
  - **`removeFirst()`, `removeLast()`**:
    entfernt erstes / letztes Element

Die Klasse LinkedList: Beispiel einer Auftragsbearbeitung

```
import java.util.LinkedList;

class Auftragsbearbeitung {
  private LinkedList<Auftrag> auftragsListe
                = new LinkedList<Auftrag>();

  public void auftragAnnehmen(Auftrag a) {
    auftragsListe.addLast(a);
  }

  public void naechstenAuftragBearbeiten() {
    Auftrag a = auftragsListe.removeFirst();
    // bearbeite den Auftrag...
  }
}
```

Zusätzliche Methoden auf Set

- **`Set<E>`**
  - Keine!
  - Die Schnittstelle sichert lediglich die Eindeutigkeit der Elemente einer Collection zu!

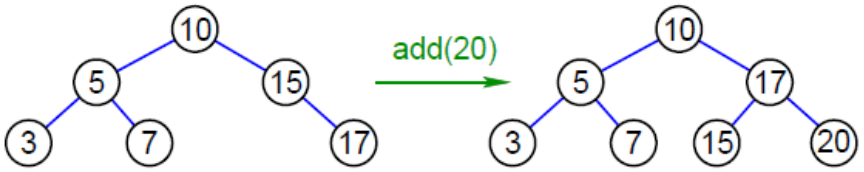
- **`SortedSet<E>`**
  - **`E first()`**
    Gibt das kleinste Element der Menge zurück.
  - **`E last()`**
    Gibt das größte Element der Menge zurück.

Die Klasse HashSet

- Konstruktoren:
  - **`HashSet(int initialCapacity, float loadFactor)<E>`**
    - Die initiale Größe der Hashtabelle wird hier spezifiziert.
    - Wenn der Füllfaktor größer als loadFactor wird, so wird eine größere Tabelle angelegt.
  - **`HashSet(int initialCapacity)<E>`**
    - Benutzt als Standardwert für den loadFactor 0.75
  - **`HashSet()<E>`**
    - Die Größe der Hashtabelle ist hier initial 16.
- Das Verhalten hash-basierter Sets ist undefiniert, wenn der Wert nach dem Hinzufügen verändert wird.
  - Schlimmstenfalls werden die Objekte nie mehr wiedergefunden.
  - Daher sollte man vorsichtig bei veränderbaren Objekte in Sets sein.

Die Klasse TreeSet

- Realisierung als balancierter binärer Suchbaum.
  - Jeder Knoten verweist auf maximal zwei Unterbäume.
  - Elemente im rechten / linken Unterbaum alle größer / kleiner als der aktuelle Knoten.
- Das Balancierung verhindert eine Entartung des Baums zur Liste.
  - Der Aufwand für die Operationen ist proportional zur Baumhöhe.
  - Das Balancierung garantiert, dass die Höhe nur logarithmisch wächst.
- Aber nach welchen Kriterien werden komplexe Objekte wie Studenten in dem Baum sortiert?



Listen vs. Sets

Klasse	Ausführungszeit `add()`	`remove()`	`get()`	`contains()`	Durchlauf-reihenfolge
`ArrayList`	$\mathcal{O}(1)$	$\mathcal{O}(n)$	$\mathcal{O}(1)$	$\mathcal{O}(n)$	Einfügung
`LinkedList`	$\mathcal{O}(1)$ *)	$\mathcal{O}(n)$	$\mathcal{O}(n)$	$\mathcal{O}(n)$	Einfügung
`TreeSet`	$\mathcal{O}(\log n)$	$\mathcal{O}(\log n)$	—	$\mathcal{O}(\log n)$	Ordnung
`HashSet`	$\mathcal{O}(1)$	$\mathcal{O}(1)$	—	$\mathcal{O}(1)$	unspezif.

*) schneller als `ArrayList`

- ↪ $\mathcal{O}(1)$: konstante Ausführungszeit, unabhängig von der Größe
- ↪ $\mathcal{O}(\log n)$: Zeit wächst logarithmisch mit d. Anzahl der Elemente
- ↪ $\mathcal{O}(n)$: Zeit wächst linear mit der Anzahl der Elemente

Listen vs. Sets

- Listen werden verwendet, wenn mehrfache Einträge erlaubt sein sollen und/oder wenn es auf die Einfügereihenfolge ankommt.
  - Bei einer **`LinkedList`** ist das Einfügen und Löschen schneller, der wahlfreie Zugriff aber langsamer als bei einer ArrayList.
  - Eine **`ArrayList`** ist zu bevorzugen bei kleineren Sequenzen, auf die häufig wahlfrei lesend zugegriffen wird.
- Sets werden verwendet, wenn keine doppelten Einträge vorkommen sollen.
  - Die Operationen eines **`HashSet`** sind performanter als bei einem **`TreeSet`**; benötigt dafür jedoch ggf. mehr Speicher.
  - Das Durchlaufen ist bei einem HashSet i.d.R. ineffizienter als bei einem TreeSet; ein HashSet ist ohne feste Reihenfolge.

Sortieren mit Comparable & Comparator

Die Klasse TreeSet

- Konstruktoren:
  - **`TreeSet()`**
    Die Menge ist nach natürlicher Ordnung der Elemente geordnet: Die Methode **`compareTo()`** der zu sortierenden Elemente wird verwendet.
  - **`TreeSet(Comparator<E> c)`**
    Die Ordnung in der Menge wird durch **`compare()`**-Methode von c festgelegt.

Geordnet vs. sortiert

- Wenn eine Liste geordnet ist, dann hat die Reihenfolge der Elemente eine bestimmte Bedeutung und wird von der Collection erhalten.
- Wenn eine Liste sortiert ist, dann folgt die Reihenfolge der Elemente einem Sortierkriterium.
- Eine sortierte Liste ist immer geordnet.
- Eine geordnete Liste kann sortiert sein, muss aber nicht.

Die natürliche Ordnung

- Alle Elemente müssen dazu Schnittstelle **`Comparable`** implementieren:

```
public interface Comparable<T> {
  public int compareTo(T o);
}
```

- **`compareTo()`** legt eine Ordnung auf Objekten fest, indem die Methode als Ergebnis liefert:
  - < 0, falls this „kleiner als“ o
  - = 0, falls this „gleich“ o
  - > 0, falls this „größer als“ o
- Die Methode kann eine **`ClassCastException`** werfen, falls der Typ von o keinen Vergleich zuläßt
- String und alle Wrapper-Klassen implementieren **`Comparable`**

Grenzen der natürlichen Ordnung

- Was passiert aber, wenn man einen Studenten wahlweise
  - nach seiner Matrikelnummer oder
  - nach seinem Nachnamen, Vornamen, Geburtsdatum

  sortieren will?
- In diesem Fall ist eine einzige feste Implementierung in der Studenten-Klasse selbst unzureichend.

Selbst definierte Ordnung über Vergleicher-Objekte

- Dem Konstruktor der zu ordnenden Menge wird ein **`Comparator`** übergeben:

```
public interface Comparator<T> {
  public int compare(T o1,T o2);
}
```

- **`compare(T o1,T o2)`** legt eine Ordnung auf Objekten fest, indem die Methode als Ergebnis liefert:
  - < 0, falls o1 „kleiner als“ o2
  - = 0, falls o1 „gleich“ o2
  - > 0, falls 01 „größer als“ o2
- Die Methode kann eine **`ClassCastException`** werfen, falls der Typ von o1 oder o2 keinen Vergleich zuläßt

Bereits vorhandene Collections nachträglich sortieren

```
class ReverseComparator implements Comparator<String> {

   public int compare(String o1, String o2) {
      return o1.compareTo(o2) * -1;
   }
}
```

Bereits vorhandene Collections nachträglich sortieren

```
List<String> list = new ArrayList<>();
list.add("Seeler");
list.add("Mueller");
list.add("Zander");
list.add("Beckenbauer");
list.add("Schumacher");

Collections.sort(list);
System.out.println(list);

Collections.sort(list, new ReverseComparator());

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

```
[Beckenbauer, Mueller, Schumacher, Seeler, Zander]
[Zander, Seeler, Schumacher, Mueller, Beckenbauer]
```


Iteratoren

Wozu braucht man Iteratoren?

```
public static void main(String[] args) {
  ArrayList<String> daten=new ArrayList<String>();
  daten.add("Uli");
  daten.add("Hans");
  daten.add("Frank");
  for(String x:daten){
    System.out.println(x);
    daten.remove(x);
  }
}
```

```
Uli
Exception in thread "main" java.util.ConcurrentModificationException
        at java.util.ArrayList$Itr.checkForComodification(Unknown Source)
        at java.util.ArrayList$Itr.next(Unknown Source)
        at Laufen.main(Laufen.java:10)
```


Wozu braucht man Iteratoren?

- Collections und Maps sind fail-fast:
  Sie entdecken konkurrierende Zugriffe und werfen eine **`ConcurrentModificationException`**, wenn eine solche vorkommt.
- Daher darf man während einer Iteration nur über einen Iterator die Collection verändern, nicht aber am Iterator vorbei.
- Direkte Änderungen während des Iterierens sind verboten.
- Man sollte die **`ConcurrentModificationException`** nicht fangen und sich auch nicht darauf verlassen, dass sie geworfen wird.

Die Lösung mit einem Iterator

```
public static void main(String[] args) {
  ArrayList<String> daten=new ArrayList<String>();
  daten.add("Uli");
  daten.add("Hans");
  daten.add("Frank");
  Iterator<String> it=daten.iterator();
  while (it.hasNext()){
    String x=it.next();
    System.out.println(x);
    it.remove();
  }
  System.out.println(daten);
}
```


Die Erweiterung eines ListIterators

- **`ListIterator`** erweitert **`Iterator`** um zusätzliche Methoden
- **`boolean hasPrevious()`**
  Gibt an, ob es noch vorhergehende Elemente gibt.
- **`T previous()`**
  Geht zum vorhergehenden Element und gibt es zurück.
- **`int nextIndex()`**
  Gibt die Position des nächsten Elements an.
- **`int previousIndex()`**
  Gibt die Position des vorhergehenden Elements an.
- **`set(T element)`**
  Ersetzt das aktuelle Element.
- **`add(T element)`**
  Fügt hinter dem aktuellen Element ein weiteres ein.

Die Map

Was sind Maps?

- Maps dienen dazu, Schlüssel auf Werte abzubilden.
- Die Schlüssel und auch die Werte können dabei beliebige Objekte sein.
- Maps realisieren einen assoziativen Speicher:
  - Deren Zugriff erfolgt nicht über eine Adresse wie einen Index oder eine Referenz, sondern über das inhaltliche Kriterium eines Schlüssels
- Beispiel: E-Mail Verzeichnis

Schlüssel		Wert
"Meier"	→	"hans.meier@uni-siegen.de"
"Müller"	→	"mueller123@gmx.de"
"Huber"	→	"u94302@aol.com"

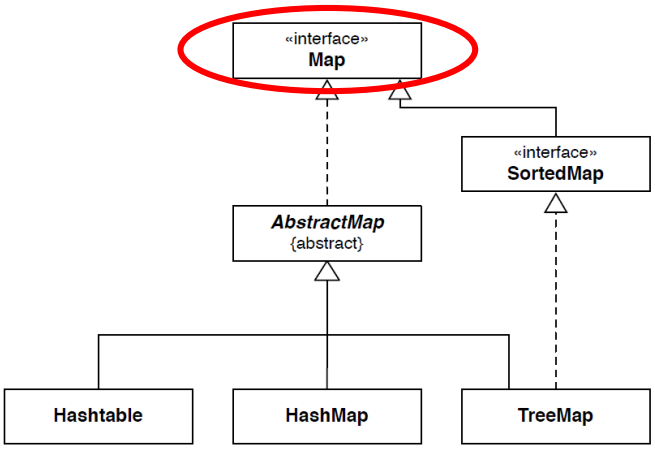
hochschule mannheim

Eigenschaften von Maps

- Eine Map kann als Menge von Paaren (Schlüssel, Wert) betrachtet werden, wobei jeder Schlüssel höchstens einmal vorkommen darf.
- Für einen Schlüssel gibt es entweder gar keinen oder genau einen Wert in der Map.
- Die Sortierung ist beliebig.
- Falls ein Schlüssel/Wert-Paar eingefügt wird, dessen Schlüssel schon in der Map existiert, so wird
  - kein neuer Eintrag angelegt, sondern der vorhandene Eintrag wird geändert und damit wird
  - dem Schlüssel ein neuer Wert zugeordnet.
- Wird ein Schlüssel gelöscht, so auch der zugehörige Wert.

hochschule mannheim

Klassengeflecht von Map



Basis-Methoden des Interfaces Map<K,V>

- **`int size()`**
  Anzahl der Elemente.
- **`boolean isEmpty()`**
  `size() == 0`
- **`boolean containsKey(Object key)`**
  Testet, ob der Schlüssel **`key`** vorhanden ist.
- **`boolean containsValue(Object value)`**
  Testet, ob Wert **`value`** vorhanden ist.
- **`V get(Object key)`**
  Liest den Wert zum Schlüssel **`key`**.
- **`V put(K key, V value)`**
  Setzt Schlüssel **`key`** und Wert **`value`**.
- **`V remove(Object key)`**
  Entfernt den Wert zu **`key`** und damit **`key`** selbst.

Basis-Methoden des Interfaces Map<K,V>

- **`void putAll(Map<? extends K, ? extends V> m)`**
  Fügt eine ganze Map hinzu.
- **`void clear()`**
  Löscht die Map.

Basis-Methoden des Interfaces Map<K,V>

- Die folgenden Methoden werden als Sichten (views) auf die Map bezeichnet.
- Sichten sind keine eigenen Objekte, sondern nur andere Sichtweisen der Map.
- Veränderungen der Map wirken sich auf die Sichten aus und umgekehrt.
- Da Map keine Iteratoren unterstützt, dienen die Sichten zum Durchlaufen der Schlüssel, Werte bzw. der (Schlüssel,Wert)-Paare.

- **`Set<K> keySet()`**
  Alle Schlüssel als Set.
- **`Collection<V> values()`**
  Alle Werte als Collection.
- **`Set<Map.Entry<K, V>> entrySet()`**
  Alle Einträge als spezielles Set.

Implementierungen des Interfaces Map

- Java bietet zwei verschiedene Implementierungen von Abbildungen:
  - **`TreeMap`**
    Verwaltung der Schlüsselmenge durch binären Suchbaum.
    - Konstruktoren:
      - **`TreeMap()`**
      - **`TreeMap(Comparator<K> c)`**
  - **`HashMap`**
    Verwaltung der Schlüsselmenge durch Hashing.
  - Konstruktoren:
    - **`HashMap()`**
    - **`HashMap(int initialCapacity, float loadFactor)`**
    - **`HashMap(int initialCapacity)`**

Properties
als spezielle Maps

Wieso Properties?

- Die Parameter-Liste in Interfaces ist in der Regel starr.
- Bei der Implementierung können die Parameter aber variabel sein; denn das Öffnen einer Dateiverbindung unterscheidet sich in seiner Implementierung vom Öffnen einer Datenbankverbindung.
- Um variable Parameter zu ermöglichen, wurden die Properties eingeführt.
- Properties können sehr leicht direkt in Dateien abgelegt oder aus Dateien geladen werden.

Die Property-Klasse

java.util

Class Properties

java.lang.Object
  java.util.Dictionary<K,V>
    java.util.Hashtable<Object,Object>
      java.util.Properties

All Implemented Interfaces:

Serializable, Cloneable, Map<Object,Object>

Die Methoden einer Property

Modifier & Type	Method and Description
`String`	**`getProperty`**`(String key)` Searches for the property with the specified key in this property list.
`void`	**`load`**`(InputStream inStream)` Reads a property list (key and element pairs) from the input byte stream.
`void`	**`load`**`(Reader reader)` Reads a property list (key and element pairs) from the input character stream in a simple line-oriented format.
`void`	**`loadFromXML`**`(InputStream in)` Loads all of the properties represented by the XML document on the specified input stream into this properties table.
`void`	**`setProperty`**`(String key, String value)` Calls the `Hashtable` method `put`.
`void`	**`store`**`(OutputStream out, String comments)` Writes this property list (key and element pairs) in this `Properties` table to the output stream in a format suitable for loading into a `Properties` table using the **`load(InputStream)`** method.
`void`	**`store`**`(Writer writer, String comments)` Writes this property list (key and element pairs) in this `Properties` table to the output character stream in a format suitable for using the **`load(Reader)`** method.
`void`	**`storeToXML`**`(OutputStream os, String comment)` Emits an XML document representing all of the properties contained in this table.

hochschule mannheim

Die vererbten Methoden von der Hashtable

Modifier & Type	Method and Description
boolean	**contains**(**Object** value) Tests if some key maps into the specified value in this hashtable.
boolean	**containsKey**(**Object** key) Tests if the specified object is a key in this hashtable.
boolean	**containsValue**(**Object** value) Returns true if this hashtable maps one or more keys to this value.
V	**get**(**Object** key) Returns the value to which the specified key is mapped, or `null` if this map contains no mapping for the key.
V	**put**(**K** key, **V** value) Maps the specified `key` to the specified `value` in this hashtable.
V	**remove**(**Object** key) Removes the key (and its corresponding value) from this hashtable.
int	**size**() Returns the number of keys in this hashtable.

hochschule mannheim

Begriffsübersicht