Programmieren II

Dr. Klaus Höppner

Hochschule Darmstadt – Sommersemester 2010

HashSet mit eigener Klasse

Wie kann man einen HashSet für eine eigene Klasse benutzen, z. B.

```
public class Person {
    private String nachname;
    private String vorname;
    private int geburtsjahr;

    public Person(String nachname, String vorname,
                                   int geburtsjahr) {
        this.nachname = nachname;
        this.vorname = vorname;
        this.geburtsjahr = geburtsjahr;
    }
}
```

Test-Programm

Zum Testen verwenden wir den folgenden Code:

```
Person p1 = new Person("Meier","Karl",1955);
Person p2 = p1;
Person p3 = new Person("Meier","Karl",1955);

Set<Person> set = new HashSet<Person>();
System.out.println(set.add(p1)); // true
System.out.println(set.add(p2)); // false
System.out.println(set.add(p3)); // true

System.out.println(set.contains(
        new Person("Meier","Karl",1955))); // false
```

Analyse

- Die Klasse `Person` überlädt nicht die Methode `equals` aus `Object`.
- Nach der Implementation von `equals` in `Object` sind zwei Objekte aber genau dann gleich, wenn sie identisch sind.
- Daher lässt sich p_2 nicht zum Set hinzufügen, da es mit p_1 identisch ist und daher beide gleich sind; p_3 wird hingegen schon, da es als eigene Instanz von `Person` nicht identisch und daher auch nicht äquivalent zur existierenden Person im Set ist. Schließlich befinden sich also *zwei* Elemente im Set.
- Die Suche nach der Person geht schief, da die zur Suche verwendete Instanz ungleich (da nicht identisch) zu allen im Set vorhandenen Personen ist.

Es fehlt daher die Definition, dass Personen mit gleichem Vor-, Nachnamen und Geburtsjahr *gleich* sind.

Definition von Gleichheit

```
@Override
public boolean equals(Object obj) {
    if (this == obj) return true;
    if (obj == null) return false;
    if (getClass() != obj.getClass()) return false;

    final Person other = (Person) obj;
    if (geburtsjahr != other.geburtsjahr) return false;
    if (nachname == null) {
        if (other.nachname != null) return false;
    } else if (!nachname.equals(other.nachname))
        return false;
    if (vorname == null) {
        if (other.vorname != null) return false;
    } else if (!vorname.equals(other.vorname))
        return false;
    return true;
}
```

equals und hashCode

Mit dieser Definition von `equals` sind zwei Personen gleich, wenn ihre Attribute gleich sind (wobei per Konvention `equals(null)` immer **`false`** ergibt).

Aber funktioniert der `HashSet` nun? Nein!
Der `HashSet` verwendet zur Kategorisierung die Methode `hashCode`. Diese wird in `Object` so definiert, dass sie von der Speicheradresse des Objekts abhängt. Daher werden Objekte, obwohl gleich (nach Definition von `equals`), trotzdem unterschiedlich einsortiert, da ihr Hash-Code verschieden sein kann.

Dies führt dazu, dass Dubletten beim Einfügen und Elemente beim Suchen nicht erkannt werden!

Überladen von hashCode

In der bisherigen Definition von `Person` wurde eine wichtige Regel verletzt:

> *Zwei Instanzen, die gleich sind, müssen den selben Hash-Code liefern!*

Hingegen ist es keine formale Bedingung, dass *ungleiche* Instanzen einen unterschiedlichen Hash-Code haben – allerdings wäre es keine gute Idee, die Methode einfach durch

```
public void hashCode() { return 0; }
```

zu überladen. Ein `HashSet` für solche Objekte würde dann zwar funktionieren, aber er wäre maximal ineffizient, weil alle Elemente in der selben Kategorie einsortiert würden.

Es empfiehlt sich daher, `hashCode` so zu überladen, dass es sich an den signifikanten Attributen orientiert.

Strategien zur Hash-Code-Berechnung

Attribut `a`	Hash-Code
byte, char, short, int	`(int) a`
long	`(int)(a ^ (a >>> 32))`
float	`Float.floatToIntBits(a)`
double	`long bits = Double.doubleToLongBits(a);` $\rightarrow$ `(int)(bits ^ (bits >>> 32))`
boolean	`a ? pattern1 : pattern2`
Object	`null == a ? 0 : a.hashCode()`

Bei mehreren Attributen

- mit `result = 1` beginnen und
- für jedes Attribut `result = 31 * result + hash` rechnen (mit `hash` nach obiger Tabelle)

Implementierung von hashCode

Nach dem vorigen Schema erhält man:

```
@Override
public int hashCode() {
    final int prime = 31;
    int result = 1;
    result = prime * result + geburtsjahr;
    result = prime * result
            + ((nachname == null) ? 0 : nachname.hashCode());
    result = prime * result
            + ((vorname == null) ? 0 : vorname.hashCode());
    return result;
}
```

Faustregel: Die Attribute, die signifikant für die Bestimmung der Gleichheit sind, werden auch für die Berechnung des Hash-Codes verwendet.

`SortedSet` – Einführung

Der normale `Set` hat einen Nachteil:

Bei einer Iteration über alle Elemente des Sets sind diese ungeordnet. Tatsächlich ist sogar die Reihenfolge der Iteration nicht zwingend reproduzierbar (und ändert sich z. B. beim Rehashing).

Dieser Nachteil wird durch das Interface `SortedSet` behoben.

Allerdings stellt `SortedSet` strengere Anforderungen an die Elemente: Für diese muss nun nicht mehr nur die *Gleichheit* definiert sein, sondern auch eine *Ordnungsfunktion*, die eine Kleiner-/Größer-Beziehung liefert.

Ordnungsfunktionen

Damit eine Klasse für einen `SortedSet` verwendet werden kann, muss sie:

- Entweder eine natürliche Ordnung besitzen, indem die Klasse das Interface `Comparable<E>` implementiert (und die zugehörige Methode `int compareTo(E e)`)
- oder dem `SortedSet` im Konstruktor eine Implementierung des Interface `Comparator<? super E>` übergeben wird (wo dann die Methode `int compare(? e1, ? e2)` implementiert wird)

Hierbei gilt:

$$\texttt{e1.compareTo(e2)}, \texttt{compare(e1,e2)} \rightarrow \begin{cases} < 0 & \text{falls } e_1 < e_2 \\ 0 & \text{falls } e_1 = e_2 \\ > 0 & \text{falls } e_1 > e_2 \end{cases}$$

TreeSet als Implementierung

Die Klasse `TreeSet<E>` implementiert das Interface `SortedSet<E>`. In diesem Interface kommen (zusätzlich dazu, dass eine Iteration über die Elemente nun sortiert ist) zwei neue Methoden hinzu:

`E first()` liefert das erste Element der Menge,

`E last()` liefert das letzte Element der Menge.

Wird dem Konstruktor kein `Comparator` als Parameter übergeben, so wird die *natürliche Ordnung* verwendet. Hierfür muss die Klasse, für die der `TreeSet` benutzt wird, `Comparable` implementieren.

Ein `TreeSet` ist als binärer Baum implementiert. Die Laufzeit für `add`, `remove` bzw. `contains` ist von der Ordnung $log(n)$.

Beispiel 1: Personen vergleichbar machen

```
class Person implements Comparable<Person> {
    ...
    @Override
    public int compareTo(Person other) {
        int result = nachname.compareTo(other.nachname);
        if (result != 0)
            return result;
        result = vorname.compareTo(other.vorname);
        if (result != 0)
            return result;
        result = Integer.valueOf(geburtsjahr).compareTo(
                                        other.geburtsjahr);
        return result;
    }
}
```

Beispiel 2: Eigener Comparator

```
import java.util.Comparator;

public class AbsCompare implements Comparator<Number> {
    public int compare(Number x, Number y) {
        double xd = x.doubleValue();
        double yd = y.doubleValue();
        return Double.valueOf(
                xd>=0 ? xd : -xd).compareTo(
                        yd>=0 ? yd : -yd);
    }
}
```

Beispiel 2: Anwendung

```
import java.util.*;

public class Main {
    public static void main(String[] args) {
        SortedSet<Integer> set =
            new TreeSet<Integer>(new AbsCompare());
        set.add(-3);
        set.add(5);
        set.add(-8);
        set.add(2);
        set.add(0);

        for (Integer i : set) {
            System.out.println(i);
        } // Ausgabe: 0 2 -3 5 -8
    }
}
```

NavigableSet

Die Klasse `TreeSet<E>` implementiert seit Java 6 nicht nur `SortedSet<E>`, sondern sogar das neue Interface `NavigableSet<E>`.
Dieses bietet einige Zusatzoperationen:

`NavigableSet<E> subset(E from, E to)` liefert alle Elemente `e` mit $from \leq e < to$.

`E floor(E e)` liefert das größte Element, das $\leq e$ ist.

`E ceiling(E e)` liefert das kleinste Element, das $\geq e$ ist.

`E lower(E e)` liefert das größte Element, das $< e$ ist.

`E higher(E e)` liefert das kleinste Element, das $> e$ ist.

Beispiel: Für einen `TreeSet<String> set` liefert `set.subset("C","D")` alle Elemente, die mit „C“ beginnen.

Assoziative Container

Assoziative Container verknüpfen einen Schlüssel (*key*) mit einem Wert (*value*).

- Telefonbuch (Name $\rightarrow$ Telefonnr.)
- Wörterbuch (deutsch $\rightarrow$ englisch)
- Kfz (Kennzeichen $\rightarrow$ Halter)
- Personalnummer $\rightarrow$ Mitarbeiter
- Buchkataloge (ISBN $\rightarrow$ Buch, Autor $\rightarrow$ Bücher)

Obige Beispiele enthalten Zuordnungen der Typen 1 zu 1 und 1 zu n.

Java unterstützt nur eindeutige Assoziationen, d. h. zu einem Schlüssel gehört maximal ein assoziierter Wert.

Die Interfaces für assoziative Container

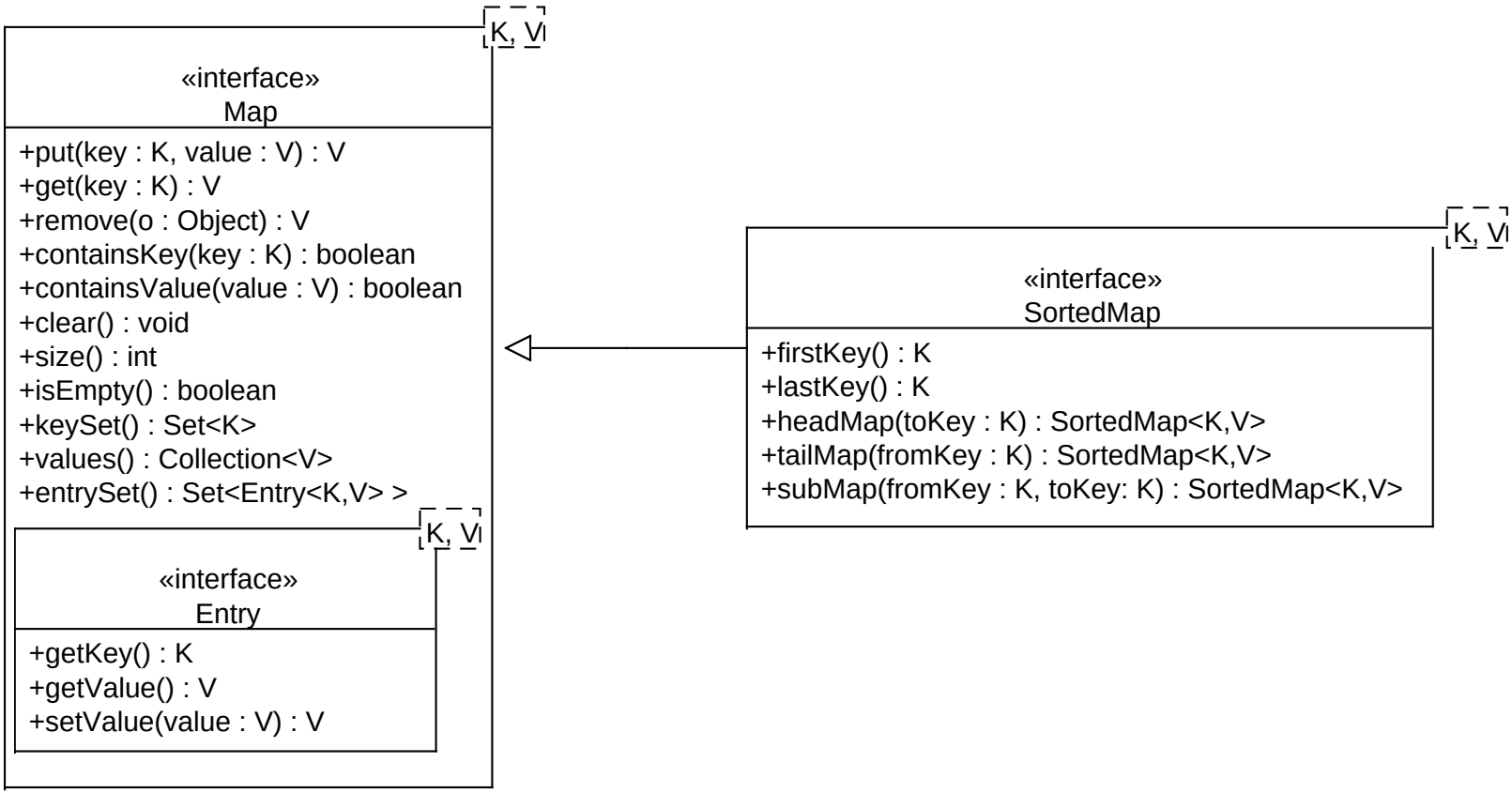
Obwohl assoziative Container häufig auch als *Collections* bezeichnet werden, sind diese vom Interface `Collection` tatsächlich unabhängig.

Assoziative Container werden durch das Interface `Map` repräsentiert. Dieses ist generisch, mit zwei Parameterklassen

- einer für den Schlüsseltyp,
- einer für den Wertetyp

Hiervon erbt das Interface `SortedMap`, bei dem die Einträge nach den Schlüsseln sortiert werden. Weiterhin gibt es seit Java 6 das Interface `NavigableMap`.

Klassen-Diagramm



Realisierung

Im Prinzip sind Maps sehr ähnlich zu Sets. So sind diese im Wesentlichen Maps nichts anderes als Sets vom Typ `Map.Entry<K,V>`, wobei für die Eindeutigkeit der Mengenelemente nur der Schlüssel berücksichtigt wird.

Dies bedeutet:

- Jeder Schlüssel darf nur einmal vorkommen (bezogen auf Äquivalenz mit `equals`).
- Bei selbst definierten Klassen müssen die Methoden `equals` (und `hashCode` bei Unterteilung nach Hashes) und `compareTo` (bei sortierten Maps) für den *Schlüsseltyp* sinnvoll definiert sein.
- Werte dürfen demgegenüber beliebig häufig vorkommen.

Methoden von Map

Das Interface `Map<K,V>` definiert u. a. folgende Methoden

`V put(K key, V val)` assoziiert den Wert `val` mit dem Schlüssel `key`. Gab es vorher schon einen Wert für `key`, so wird dieser überschrieben (und der alte Wert als Rückgabewert zurückgegeben). Gab es vorher noch keine Assoziation für `key`, so ist der Rückgabewert **`null`**.

`V get(K key)` gibt den mit `key` assoziierten Wert zurück (sonst **`null`**).

`V remove(K key)` entfernt die Assoziation für `key`. Rückgabewert ist der zuvor assoziierte Wert (**`null`**, falls keine existierte).

Methoden von Map (Forts.)

`boolean containsKey(K key)` gibt **`true`** zurück, falls der Schlüssel existiert.

`boolean containsValue(V val)` gibt **`true`** zurück, falls ein Schlüssel den angegebenen Wert assoziiert.

`void clear()` löscht alle Einträge.

`int size()` gibt die Zahl der Schlüssel-Wert-Paare zurück.

`boolean isEmpty()` gibt **`true`** bei einem leeren Container zurück.

Methoden von Map (Forts.)

`Set<K> keySet()` gibt die Menge aller Schlüssel zurück.

`Collection<V> values()` gibt alle vorhandenen Werte als `Collection` zurück. Die Reihenfolge entspricht dabei der Reihenfolge der Schlüssel bei einer Iteration über `keySet()`. Der Rückgabewert ist eine `Collection`, da jeder Wert prinzipiell beliebig häufig vorkommen darf.

`Set<Map.Entry<K,V> > entrySet()` gibt alle Schlüssel-Wertpaare als `Set` zurück.

Das Interface `Map.Entry`

Über die Methode `entrySet` erhält man eine Mengenansicht (*set view*) der Schlüssel-Werte-Paare. Hierfür wird das *innere Interface* `Entry<K,V>` von `Map<K,V>` verwendet.

Dieses kennt drei Methoden:

`K getKey()` gibt den Schlüssel zuück.

`V getValue()` gibt den Wert zurück.

`V setValue(V val)` setzt den Wert neu und gibt den vorherigen als Rückgabewert zurück.

Implementierung

Die Implementierungen von `Map` sind analog zu `Set`:

HashMap teilt die Schlüssel-Werte-Paare anhand des Hash-Codes der Schlüssel in Buckets auf. Da die Hash-Codes der Schlüssel keine Ordnung definieren, sind die Schlüssel unsortiert. Beim Wachsen der `HashMap` kommt es zu einem *Rehashing*, wenn der Füllgrad den definierten *load factor* übersteigt.

TreeMap implementiert zusätzlich `Sorted-/NavigableMap`. Es wird ein binärer Baum anhand der Schlüssel verwendet. Daher muss ein `Comparator` für den Schlüsseltyp angegeben werden, oder es wird die natürliche Ordnung der Schlüssel verwendet (hierfür muss `Comparable` implementiert sein).

Beispiel

```
import java.util.*;

public class Bsp1 {
   public static void main(String[] args) {
      Map<String,String> telefon = new HashMap<String,String>();
      telefon.put("Mustermann, Karl", "6835454");
      telefon.put("Ahrens, Karl", "484894");
      telefon.put("Fischer, Maria", "132542");
      telefon.put("Mustermann, Karl", "944889");

      for (Map.Entry<String, String> entry: telefon.entrySet()) {
         System.out.println(entry.getKey()+": "+
               entry.getValue());
      }
   }
}
```

Analyse

Im vorigen Beispiel werden vier Einträge der Telefonliste hinzu gefügt, aber nur drei davon befinden sich am Ende im Container.

Für den Schlüssel *Mustermann, Karl* werden zwei Nummern hinzugefügt. Schlüssel in der Map müssen aber eindeutig sein. Daher befindet sich am Ende nur einer der beiden Einträge in der Map.

Die Logik bei doppelten Einträgen bei `Map` unterscheidet sich vom `Set`:

- Bei `Set` „gewinnt“ das erste `add`, weitere scheitern und geben **`false`** zurück.
- Bei `Map` wird jeweils der alte Wert überschrieben, es „gewinnt“ also das letzte `put`.

Map mit eigener Klasse als Schlüssel

Für eine `Map` kann auch eine eigene Klasse als Schlüssel verwendet werden.

Hierbei gelten sinngemäß die selben Kriterien wie bei den Mengen:

- Für `HashMap` muss die Klasse `equals` und `hashCode` sinnvoll überladen,
- für `TreeMap` muss die Klasse entweder `Comparable` implementieren oder ein geeigneter `Comparator` existieren.

Neue Version der Telefonliste

```
import java.util.*;

public class Bsp2 {
  public static void main(String[] args) {
    Map<Person,String> telefon = new HashMap<Person,String>();
    telefon.put(new Person("Mustermann", "Karl",1973), "6835454");
    telefon.put(new Person("Ahrens", "Karl",1950), "484894");
    telefon.put(new Person("Fischer", "Maria",1966), "132542");
    telefon.put(new Person("Mustermann", "Karl",1973), "944889");

    for (Map.Entry<Person, String> entry: telefon.entrySet()) {
      System.out.println(entry.getKey()+": "+
          entry.getValue());
    }
  }
}
```

Das Interface SortedMap

Bei `SortedMap` sind die Paare nach dem Schlüssel sortiert.
Zusätzlich existieren drei Methoden, eine Teilmenge der Schlüssel auszuwählen:

`headMap(K key)` alle Einträge *vor* dem angegebenen Schlüssel,

`tailMap(K key)` alle Einträge *ab* dem Schlüssel (einschließlich),

`subMap(K from, K to)` alle Einträge mit Schlüssel zwischen `from` (inkl.) und `to` (exkl.).

Rückgabetyp ist jeweils `SortedMap<K,V>`.

Ab Java 6 implementiert `TreeMap` das neue Interface `NavigableMap`, das zusätzliche Methoden bietet.

Beispiel

```
SortedMap<String,String> telefon = new TreeMap<String,String>();

// Füllen

for (Map.Entry<String, String> entry:
                    telefon.tailMap("F").entrySet()) {
    System.out.println(entry.getKey()+": "
            +entry.getValue());
}

for (String name: telefon.subMap("B","H").keySet()) {
    System.out.println("Name: "+name);
}
```