Informatik B - Objektorientierte Programmierung in Java

Vorlesung 09:
Collections 3

Inhalt

Collections des Typs Set

- Eigenschaften
  - Eine Collection vom Typ `Set` ist die Java-Repräsentation einer mathematischen Menge.
  - Sie enthält keine doppelten Elemente (Nutzdaten), d.h. es darf zu keinem Zeitpunkt zwei Elemente a und b geben, für die `a.equals(b)` wahr ist.
  - Die Elemente eines `Sets` haben keine definierte Reihenfolge. Sie kann sich durch das Einfügen und Löschen von Elementen im Laufe der Zeit ändern.

Collections des Typs Set

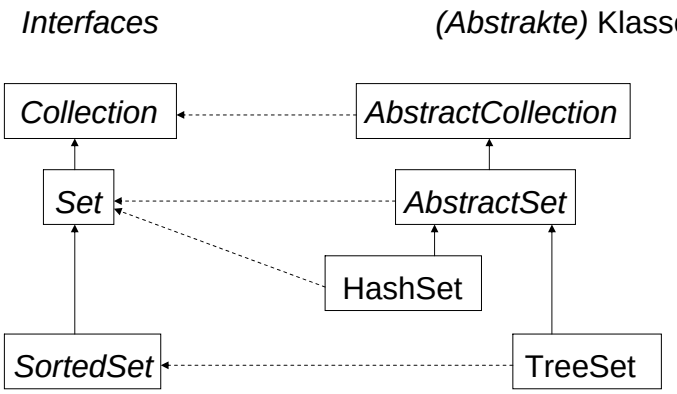
- Anmerkung
  - In allen Collections der Java-Bibliothek werden nur Verweise auf bereits vorher existierende Objekte gespeichert.
  - Besondere Vorsicht ist geboten, wenn ein `Set` dazu benutzt wird, veränderliche (engl. mutable) Objekte zu speichern.
  - Wird ein Objekt, das bereits in einem `Set` gespeichert ist, anschließend so verändert, dass der `equals`-Vergleich mit einem anderen Element des `Sets` wahr ist, so gilt das gesamte `Set` als undefiniert und darf nicht mehr benutzt werden.

Collections des Typs Set

- Operationen
  - Eine Collection vom Typ `Set<E>` besitzt alle Operationen einer allgemeinen `Collection`, insbesondere auch die typischen Mengenoperationen:
    - Einfügen aller Objekte einer zweiten Collection (Vereinigung)
      - `boolean` **`addAll`**`(Collection<? extends E> c)`
    - Entfernen aller Objekte einer zweiten Collection (Mengendifferenz)
      - `boolean removeAll(Collection<?> c)`
    - Beibehalten aller Elemente, die in einer zweiten Collection enthalten sind bzw. Entfernen aller Elemente, die nicht in der zweiten Collection enthalten sind (Durchschnitt)
      - `boolean retainAll(Collection<?> c)`

Collections des Typs Set

- Klassenhierarchie



Collections des Typs Set

- Implementierungen
  - Abstrakte Basisklasse `AbstractSet<E>`:
    - dient als Basisklasse für eigene `Set`-Implementierungen
  - Klasse `HashSet<E>`:
    - implementiert die Menge mit Hilfe einer Hash-Tabelle (siehe Maps)
    - Vor jedem Einfügen wird geprüft, ob das einzufügende Element bereits in der Hash-Tabelle enthalten ist.
  - Klasse `TreeSet<E>`:
    - siehe Sortierte Collections

Bsp: Lottotip.java

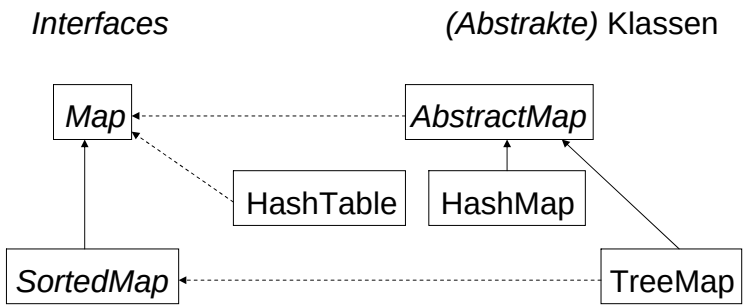
```
public static void main(String[] args)
{
     Set<Integer> set = new HashSet<Integer>(10);
     int doubletten = 0;

     //Lottozahlen erzeugen
     while (set.size() < 6)
     {
            int num = (int)(Math.random() * 49 + 1.5);
            if (!set.add(num))    ++doubletten;
     }

     //Lottozahlen ausgeben
     for (int zahl : set)  System.out.println(zahl);
     System.out.println("Ignorierte Doubletten: " + doubletten);

     SortedSet<Integer> ts = new TreeSet<Integer>();
     ts.addAll(set);
     for (int zahl : ts) System.out.println(zahl);
}
```

Map

- Eigenschaften
  - Eine `Map` realisiert einen assoziativen Speicher, der Schlüssel auf Werte abbildet.
  - Eine solche Struktur, bei der Paare mit dem Aufbau (Schlüssel,Wert) gespeichert werden, wird in der Informatik auch als Wörterbuch bezeichnet.
  - Bei den `Maps` der Java-Klassenbibliothek sind sowohl die Schlüssel als auch die Werte Objekte einer beliebigen Klasse.
  - Je Schlüssel gibt es entweder keinen oder genau einen Eintrag in der `Map`.

Map

- Klassenhierarchie



Map

- Interface `Map<K, V>`
  - Das Interface `Map` ist nicht von `Collection` abgeleitet.
  - Die Operationen `size`, `isEmpty` und `clear` entsprechen jedoch den gleichnamigen Operationen aus `Collection`.
  - Entfernen von Elementen
    - Das Entfernen eines Paares zu einem Schlüssel erfolgt mit der Operation

      `V remove(Object key)`

      Als Ergebnis wird der gespeicherte Wert zurückgegeben, wenn der Schlüssel enthalten war, ansonsten `null`.

Map

- Einfügen von Elementen
  - Das Einfügen eines Paares (Schlüssel, Wert) erfolgt mit der Operation

    `V put(K key, V value)`

    Ist der Schlüssel bereits enthalten, wird der alte Wert durch den neuen Wert ersetzt.

    Als Ergebnis wird der alte Wert zurückgegeben, wenn der Schlüssel enthalten war, ansonsten `null`.
  - Die Operation

    `void putAll(Map<? extends K,? extends V> t)`

    fügt alle Paare aus `t` ein.

Map

- Zugriff auf Elemente
  - Der Zugriff auf einen Wert über den ihm zugeordneten Schlüssel erfolgt mit der Operation
    `V get(Object key)`
    - Die Operation liefert den zu `key` gehörenden Wert oder `null`, falls kein Paar mit dem Schlüssel `key` enthalten ist.
    - Die Operation liefert auch dann `null` als Ergebnis, wenn es erlaubt ist, `null` als Wert einzutragen. In diesem Fall kann die Operation `containsKey` benutzt werden, um die beiden Fälle zu unterscheiden.
  - Zur Überprüfung, ob ein bestimmter Schlüssel oder ein bestimmter Wert enthalten ist, gibt es zwei Operationen:
    `boolean containsKey(Object key)`
    `boolean containsValue(Object value)`

Map

- Sichten (engl. views)
  - Im Vergleich zum `Collection`-Interface fällt auf, dass eine `Map` keine Operation `iterator` besitzt, um einen Iterator für den sequentiellen Durchlauf zu erzeugen.
  - Stattdessen kann sie drei unterschiedliche Collections, sog. Sichten, erzeugen, die dann ihrerseits dazu verwendet werden können, einen Iterator zu liefern:
    - `Set<K> keySet()`
      liefert die Menge aller Schlüssel. Da per Definition keine doppelten Schlüssel in einer `Map` enthalten sind, ist das Ergebnis vom Typ `Set`.
    - `Collection<V> values()`
      liefert alle Werte der `Map`. Da Werte doppelt enthalten sein können, ist der Rückgabewert lediglich vom Typ `Collection`.

Map

  - `Set<Map.Entry<K, V>> entrySet()`
    liefert eine Menge von Paaren (Schlüssel, Wert).
    Jedes Element dieser Menge ist vom Typ `Map.Entry`,
- Auf ein Objekt vom Typ `Map.Entry<K, V>` können folgende Operationen angewandt werden:
  - `K getKey()`
    liefert den Schlüssel des Eintrags als Ergebnis
  - `V getValue()`
    liefert den Wert des Eintrags als Ergebnis
  - `V setValue(V value)`
    setzt den Wert des Eintrags

Map

- Sichten (Hinweis)
  - Die von `keySet`, `value` und `entrySet` gelieferten Collections enthalten Verweise auf die Inhalte der Map. Änderungen in der Map wirken sich auf diese Collections aus und umgekehrt. Wird die Map modifiziert während ein Iterator die Collection bearbeitet, sind die Ergebnisse dieses Iterators undefiniert.
  - Die Collections unterstützen das Löschen von Elementen, das das korrespondierenden Paar aus der Map löscht: `Iterator.remove`, `Collection.remove`, `removeAll`, `retainAll` und `clear`. Nicht unterstützt: `add`, `addAll`

Map

- Implementierungen
  - Abstrakte Basisklasse `AbstractMap<K, V>`:
    - dient als Basisklasse für eigene `Map`-Implementierungen
  - Klasse `HashMap<K, V>`:
    - implementiert das Interface `Map` mit Hilfe einer Hash-Tabelle
  - Klasse `Hashtable<K, V>`:
    - Seit dem JDK 1.2 implementiert die Klasse `Hashtable` ebenfalls das Interface `Map`.
    - Im Gegensatz zur `HashMap` ist es nicht erlaubt, `null`-Werte einzutragen.
    - Die wichtigen Operationen sind synchronisiert.
  - Klasse `TreeMap<K, V>`:
    - siehe Sortierte Collections

Map

- Hash-Verfahren (dt.: gestreute Speicherung)
  - Eine Hash-Tabelle benutzt das Verfahren der Schlüsseltransformation, d.h. es wird eine Transformationsfunktion (auch Hash-Funktion genannt) zur Abbildung von Schlüsseln auf Indexpositionen eines Feldes verwendet.
  - Ziel ist es, die Transformationsfunktion durch möglichst einfache arithmetische Operationen zu realisieren und eine gleichmäßige Streuung der Einträge in der Tabelle zu erzielen.
  - Zur Auflösung von Kollisionen in der Tabelle gibt es verschiedene Strategien.

Map

- Wichtige Parameter einer Hash-Tabelle in Java sind:
  - Kapazität: Anzahl der Elemente, die insgesamt in der Hash-Tabelle untergebracht werden können
  - Ladefaktor: zeigt an, bei welchem Füllungsgrad die Hash-Tabelle vergrößert werden muss
- Das Vergrößern der Tabelle erfolgt automatisch, wenn die Anzahl der Elemente innerhalb der Tabelle größer ist als das Produkt aus Kapazität und Ladefaktor.

Map

- Eigenschaften von Schlüssel-Objekten
  - Damit die Objekte einer Klasse als Schlüssel für eine Hash-Tabelle benutzt werden können, muss die Klasse zwei Operationen zur Verfügung stellen:
    - `public boolean equals(Object o)`
    - `public int hashCode()`
  - Werden die Operationen `equals` und `hashCode` von einer Klasse nicht zur Verfügung gestellt, werden die entsprechenden Operationen der Oberklasse benutzt.
  - Stellt keine der Oberklassen die Operationen zur Verfügung, werden die Operationen der Klasse `Object` benutzt.

Map

- Eigenschaften der Operationen der Klasse `Object`:
  - Die Implementierung der Operation `equals` überprüft die Referenzgleichheit zweier Objekte nicht die Inhaltsgleichheit, und ist damit im allgemeinen nicht für die Nutzung in einer Hash-Tabelle geeignet.
  - Die Implementierung der Operation `hashCode` garantiert nur, dass mehrfache Aufrufe der Operation für ein (referenz-) gleiches Objekt den gleichen Hashcode liefern. Für zwei inhaltsgleiche Objekte wird der Hashcode im allgemeinen verschieden sein.
- Daher sollten in jeder Klasse, deren Objekte als Schlüssel für eine Hash-Tabelle benutzt werden sollen, diese beiden Operationen überschrieben, d.h. neu programmiert werden. Es muss gelten: Sind zwei Objekte bzgl. `equals` gleich, muss `hashCode` für sie denselben Hashcode liefern.

Bsp: MailAlias.java

```
public static void main(String[] args)
{
    Map<String,String> h = new HashMap<String,String>();
    //Aufbau der Aliase
    h.put("Fritz","f.mueller@test.de");
    h.put("Franz","fk@b-blabla.com");
    h.put("Paula","user0125@mail.uofm.edu");
    h.put("Lissa","lb3@gateway.fhdto.northsurf.dk");
    //Ausgabe
    Set<Map.Entry<String,String>> set = h.entrySet(); // Sicht
    Iterator<Map.Entry<String,String>> it = set.iterator();
    while (it.hasNext())
    {
          Map.Entry<String,String> entry = it.next();
          System.out.println(
                entry.getKey() + " --> " +
                entry.getValue()
          );
    }
}
```


Sortierte Collections

- Die `Set`-Collection vom Typ `HashSet` ist unsortiert, d.h. ihre Iteratoren geben die Elemente in keiner bestimmten Reihenfolge zurück.
- Die Collections der Sichten auf die Maps vom Typ `HashMap` und `Hashtable` sind ebenfalls unsortiert.
- Im Gegensatz dazu speichern die Collection vom Typ `TreeSet` und `TreeMap` ihre Elemente in einer sortierten Reihenfolge.

Sortierte Collections

- Sortierung
  - Standardmäßig erfolgt die Sortierung der Elemente nach ihrer natürlichen Ordnung.
  - Dazu muss sichergestellt werden, dass
    - alle Objekte, die in einer sortierten Collection gespeichert werden, eine `compareTo`-Operation besitzen
    - je zwei beliebige gespeicherte Objekte miteinander verglichen werden können, ohne dass es zu einem Typfehler kommt.
  - Die Klassen der Objekte müssen also das Interface `Comparable` implementieren, welches die Operation
    - `public int compareTo(Object o)`

    definiert.

Sortierte Collections des Typs Set

- Zur Realisierung von sortierten Mengen wurde aus `Set<E>` das Interface `SortedSet<E>` abgeleitet.
- Es erweitert das Interface `Set` um folgenden Operationen:
  - Zugriff auf das bezüglich der Sortierordnung erste bzw. letzte Element der Menge:
    - `E first()`
    - `E last()`

Sortierte Collections des Typs Set

- Erzeugung von Teilmengen der Originalmenge auf Basis der Sortierung:
  - `SortedSet<E> headSet(E toElement)`

    alle Elemente, die echt kleiner als das als Argument übergebene Objekt sind
  - `SortedSet<E> tailSet(E fromElement)`

    alle Elemente, die größer oder gleich dem als Argument übergebenen Objekt sind
  - `SortedSet<E> subSet(E fromElement, E toElement)`

    alle Elemente, die größer oder gleich `fromElement` und echt kleiner als `toElement` sind

Sortierte Collections des Typs Set

- Implementierung
  - Klasse `TreeSet<E>`:
    - implementiert die sortierte Menge mit Hilfe der Klasse `TreeMap`
    - Die einzufügenden Objekte werden vollständig als Schlüssel betrachtet.
    - Das zugehörige Wert-Objekt ist für alle Schlüssel identisch.

Bsp: Früchte.java

```
import java.util.SortedSet;
import java.util.TreeSet;
public class Fruechte
{
  public static void main(String[] args)
  {
      //Konstruieren des Sets
      SortedSet<String> s = new TreeSet<String>();

      s.add("Kiwi");
      s.add("Kirsche");
      s.add("Ananas");
      s.add("Zitrone");
      s.add("Grapefruit");
      s.add("Banane");
      for (String st: s) System.out.println(st);
  }
}
```

Sortierte Map

- Zur Realisierung von sortierten Maps wurde aus `Map<K, V>` das Interface `SortedMap<K, V>` abgeleitet.
- Es erweitert das Interface `Map<K, V>` um folgende Operationen, die analog zu den Operationen von `SortedSet<E>` definiert sind:
  - `K firstKey()`
  - `K lastKey()`
  - `SortedMap<K, V> headMap(K toKey)`
  - `SortedMap<K, V> tailMap(K fromKey)`
  - `SortedMap<K, V> subMap(K fromKey, K toKey)`

Sortierte Map

- Sortierung
  - Die Sortierung der Elemente in einer `SortedMap` erfolgt standardmäßig entsprechend der natürlichen Ordnung der Schlüssel.
  - Eine Klasse, deren Objekte als Schlüssel benutzt werden sollen, muss das Interface `Comparable` implementieren.

Sortierte Map

- Iteratoren
  - Die Operationen `keySet` und `entrySet` angewandt auf eine `SortedMap` liefern Collections vom Typ `Set`, deren Iteratoren ihre Elemente in aufsteigender Reihenfolge abliefern.

Sortierte Map

- Implementierungen
  - Klasse `TreeMap<K, V>`:
    - Datenstruktur
      - Als Datenstruktur wird ein Rot-Schwarz-Baum benutzt.
      - Der Rot-Schwarz-Baum ist ein balancierter Baum, d.h. spezielle Einfüge- und Löschoperationen stellen sicher, dass der Baum nicht zu einer linearen Liste entartet.

Comparable vs. …

- Seit dem JDK 1.2 wird das `Comparable`-Interface bereits von vielen der eingebauten Klassen implementiert, etwa von `String`, `Character`, `Double` usw. Die natürliche Ordnung einer Menge von Elementen ergibt sich nun, indem man alle Elemente paarweise miteinander vergleicht und dabei jeweils das kleinere vor das größere Element stellt mit `compareTo()`.

… java.util.Comparator

- Die zweite Möglichkeit, eine Menge von Elementen zu sortieren, besteht darin, an den Konstruktor der Collection-Klasse ein Objekt des Typs `Comparator` zu übergeben. `Comparator` ist ein Interface, das lediglich eine einzige Methode `compare` definiert:
- **`public int compare(Object o1, Object o2)`**
- Das übergebene `Comparator`-Objekt übernimmt die Aufgabe einer »Vergleichsfunktion«, deren Methode `compare` immer dann aufgerufen wird, wenn bestimmt werden muss, in welcher Reihenfolge zwei beliebige Elemente zueinander stehen.

Beispiel mit Comparator

- TreeTestTest
  - TreeTest.java
    - Verwaltet 2-dimensionale Punkte und sortiert diese gemäß zwei unterschiedlichen Comparatoren
  - Point.java
    - 2-dimensionaler Punkt mit Bezeichnung, z.B.: Nr. 1 (3,4)
  - EuklidPointComparator.java
    - Bestimmt den Abstands eines Punktes zum Ursprung nach euklidischer Norm.
  - DeltaXComparator.java
    - Vergleicht zwei Objekte anhand ihres X-Abstandes zum Ursprung.

Beispiel mit Comparator

- `TreeSet` kann `Comparables` einfügen und verwalten. Alternativ kann aber ein `Comparator` entsprechenden Typs angegeben werden, um die Elemente zu sortieren. Die Objekte müssen also nicht mehr von Typ `Comparable` sein. Die Implementation mit einem `Comparator` ist elegant, da dieser leicht ausgetauscht werden kann und auch von jemandem implementiert werden kann, der keinen (Programmier-) Zugriff auf die einzufügenden Objekte besitzt.