

Programmierzertifikat Objekt-Orientierte Programmierung mit Java

Vorlesung 09: Mengen

Peter Thiemann

Universität Freiburg, Germany

SS 2008

Inhalt

Mengen

HashSet

LinkedHashSet

CopyOnWriteArraySet

EnumSet

SortedSet

NavigableSet

Mengen

- ▶ Eine Menge ist eine Ansammlung von Elementen
 - ▶ ohne Duplikate
 - ▶ ohne Berücksichtigung der Reihenfolge
- ▶ Hinzufügen eines bereits vorhandenen Elements hat keine Wirkung.
- ▶ Set Interface
 - ▶ hat **die gleichen Methoden** wie Collection Interface
 - ▶ aber andere Verträge für add und addAll

Beispiel (Mengen)

```
// Erinnerung: Collection<Task>  
assert mondayTasks.toString().equals ("[code logic, phone Paul]");  
assert phoneTasks.toString().equals ("[phone Mike, phone Paul]");  
  
// Menge aus Collection  
Set<Task> phoneAndMondayTasks = new TreeSet<Task> (mondayTasks);  
phoneAndMondayTasks.addAll (phoneTasks);  
// doppelte Elemente fallen weg  
assert phoneAndMondayTasks.toString().equals(  
    "[code logic, phone Mike, phone Paul]");
```

Arbeiten mit Mengen

- ▶ Set ist Interface
- ▶ Methoden arbeiten mit allen Implementierungen
- ▶ Unterschiede der Implementierungen: unterschiedliche Komplexität der Operationen (einfügen, löschen, Durchlauf, etc)
- ▶ Job des Programmierers:
Auswahl der geeigneten Implementierung nach Notwendigkeiten der Aufgabe

Sechs Implementierungen von Mengen

Implementierungen von Set<E>

- ▶ HashSet<E>
- ▶ LinkedHashSet<E>
- ▶ CopyOnWriteArraySet<E>
- ▶ EnumSet<E **extends** Enum<E>>

Implementierungen von SortedSet<E> und NavigableSet<E>

- ▶ TreeSet<E>
- ▶ ConcurrentSkipListSet<E>

HashSet

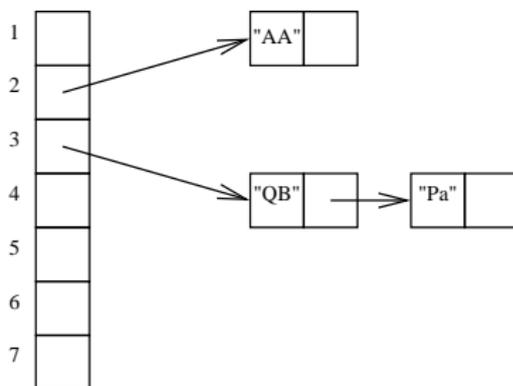
HashSet

- ▶ Erzeugung durch Konstruktor
HashSet (int initialCapacity)
- ▶ Implementierung durch *Hashtabelle*
- ▶ Array indiziert durch eine *Hashfunktion*, die aus den Elementen ausgerechnet wird
- ▶ Beispiel (aus `java.lang.String`)

```
int hashCode() {  
    int hash = 0;  
    for (char ch : str.toCharArray()) {  
        hash = hash * 31 + ch;  
    }  
    return hash;  
}
```

Kollisionen

- ▶ Problem: hashCode ist nicht injektiv
 - ▶ Es kommt zu *Kollisionen*.
 - ▶ Beispiel: `"QB".hashCode() == "Pa".hashCode()`
 - ▶ (Kollisionsfreiheit prinzipiell nicht möglich für Typen mit mehr als 2^{32} Werten.)
- ▶ Eine Lösung: Überlauflisten



Aufwand (Einfügen und Suchen)

- ▶ Ohne Kollisionen:
Aufwand für Einfügen, Suchen und Löschen eines Elements ist konstant
 - ▶ Je voller die Hashtabelle, desto wahrscheinlicher sind Kollisionen
 - ▶ Beim Einfügen, Suchen und Löschen kommt der Durchlauf der Überlauf-liste hinzu
- ⇒ Je voller die Hashtabelle, desto langsamer der Zugriff
- ⇒ Reorganisation bei gewissem Füllungsgrad (*rehashing*):
z.B. kopieren der Elemente in neues, doppelt so großes Array

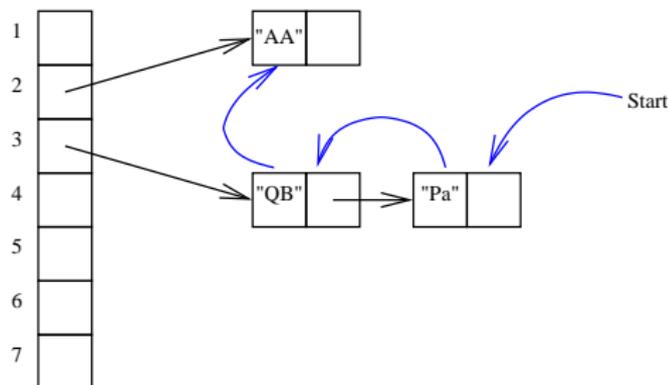
Aufwand (Durchlauf)

- ▶ Der Iterator muss jeden Eintrag der Hashtabelle untersuchen
- ⇒ Aufwand = Größe der Hashtabelle + Anzahl der Einträge

LinkedHashSet

HashSet mit schnellem Iterator

- ▶ Subklasse von HashSet
- ▶ Zusätzliche Garantie:
Iterator durchläuft die Elemente in der Reihenfolge, in der sie eingefügt wurden
- ▶ Implementiert durch zusätzliche verkettete Liste



Aufwand

- ▶ Einfügen, Suchen, Löschen wie `HashSet`
 - ▶ Extra Aufwand (größere Konstante und erhöhter Speicherbedarf) für Verwaltung der verketteten Liste
 - ▶ Aufwand für Iterator = Anzahl der Einträge
 - ▶ Insbesondere: `next()` läuft in konstanter Zeit
- ⇒ `LinkedHashSet` lohnt nur, falls der schnellere Iterator wichtig ist

CopyOnWriteArraySet

Mengenimplementierung für Spezialanwendungen

- ▶ Menge implementiert durch ein Array
 - ▶ **Jede Änderung des Inhalts bewirkt die Erzeugung eines neuen Arrays!**
- ⇒ Aufwand für Einfügen $O(n)$
- ⇒ Aufwand für Suchen $O(n)$
- ⇒ Aufwand für `Iterator.next()` $O(1)$
- ▶ Vorteil von `CopyOnWriteArraySet`
 - ▶ Nebenläufige Verwendung möglich ohne das Lesen zu verlangsamen
 - ▶ (Geeignet für subject/observer Muster)

EnumSet

Spezielle Eigenschaften von Enum

- ▶ Jede Enumeration hat die Eigenschaften
 - ▶ die Anzahl m der Elemente ist bekannt
 - ▶ jedes Element hat eine feste Nummer `value.ordinal()`
- ▶ Implementierung durch **Bitset**
 - ▶ Array von 32-Bit Zahlen
 - ▶ Jede Bitstelle steht für ein Element
 - ▶ Mengenoperationen durch Bitmanipulation
 - ▶ Bit setzen: einfügen
 - ▶ Bit löschen: Element entfernen
 - ▶ Bitweise-und: Schnittmenge
 - ▶ Bitweise-oder: Vereinigung
- ▶ Iterator liefert Elemente in natürlicher Reihenfolge
- ▶ Aufwand
 - ▶ Einfügen, Entfernen: $O(1)$
 - ▶ `clear`, `isEmpty`, `size`: $O(m)$
 - ▶ Iterator `next`: $O(m)$

Statische Factory Methoden

- ▶ Menge aus den angegebenen Elementen

```
<E extends Enum<E>> EnumSet<E> of (E first, E... rest);
```

- ▶ Menge mit einem Intervall von Elementen

```
<E extends Enum<E>> EnumSet<E> range (E from, E to);
```

- ▶ Menge mit allen Elementen

```
<E extends Enum<E>> EnumSet<E> allOf (E from, E to);
```

- ▶ Leere Menge

```
<E extends Enum<E>> EnumSet<E> noneOf (E from, E to);
```

Verwendung der Factory Methoden

```
enum Season { SPRING, SUMMER, AUTUMN, WINTER }  
  
EnumSet<Season> s1 = EnumSet.of (Season.SPRING, Season.AUTUMN);  
  
EnumSet<Season> s2 = EnumSet.range (Season.SPRING, Season.AUTUMN);  
  
EnumSet<Season> all_seasons = EnumSet.allOf (Season.class);  
  
EnumSet<Season> no_seasons = EnumSet.noneOf (Season.class);
```

Manipulation von EnumSet

- ▶ Kopie eines EnumSet

```
<E extends Enum<E>> EnumSet<E> copyOf (EnumSet<E> s);
```

- ▶ Umwandlung Collection → EnumSet (mindestens ein Element erforderlich)

```
<E extends Enum<E>> EnumSet<E> copyOf (Collection<E> s);
```

- ▶ Komplement eines EnumSet

SortedSet

Mengen über geordneten Elementen

- ▶ Iterator durchläuft Elemente in aussteigender Ordnung

```
interface SortedSet<E> {  
    public E first();  
    public E last();  
    public Comparator<? super E> comparator();  
    public SortedSet<E> subSet (E fromElement, E toElement);  
    public SortedSet<E> headSet (E toElement);  
    public SortedSet<E> tailSet (E fromElement);  
}
```

Ab Java 6:

- ▶ NavigableSet<E> als Subinterface von SortedSet<E>
- ▶ gleiche Implementierungen

Konstruktion

- ▶ `TreeSet<E>` implementiert `SortedSet<E>`
- ▶ Konstruktoren
 - ▶ `TreeSet ()`
verwendet die natürliche Ordnung
 - ▶ `TreeSet (Comparator<? super E> comparator)`
verwendet die durch `comparator` definierte Ordnung

Beispiel: Verschmelzen zweier sortierter Tasklisten

- ▶ Vorher mit zwei Iteratoren implementiert

```
Set<Task> naturallyOrderedTasks = new TreeSet<Task> (mondayTasks);  
naturallyOrderedTasks.addAll (tuesdayTasks);  
assert naturallyOrderedTasks.toString().equals(  
    "[code db, code gui, code logic, phone Mike, phone Paul]");
```

- ▶ Aufwand zum Verschmelzen von Mengen mit insgesamt m Elementen
 - ▶ Verschmelzen von zwei sortierten Listen: $O(m)$
 - ▶ Verschmelzen über `TreeSet<Task>`: $O(m \log m)$.

Erweitertes Beispiel: Aufgaben mit Prioritäten

```
public enum Priority { HIGH, MEDIUM, LOW }

public final class PriorityTask implements Comparable<PriorityTask> {
    private final Task task;
    private final Priority priority;
    PriorityTask(Task task, Priority priority) {
        this.task = task;
        this.priority = priority;
    }
    public Task getTask() { return task; }
    public Priority getPriority() { return priority; }
```

Erweitertes Beispiel: Aufgaben mit Prioritäten (Forts.)

```
public int compareTo(PriorityTask pt) {  
    int c = priority.compareTo(pt.priority);  
    return (c != 0) ? c : task.compareTo(pt.task);  
}  
public boolean equals(Object o) {  
    if (o instanceof PriorityTask) {  
        PriorityTask pt = (PriorityTask)o;  
        return task.equals(pt.task) && priority.equals(pt.priority);  
    } else return false;  
}  
public int hashCode() { return task.hashCode(); }  
public String toString() { return task + ": " + priority; }  
}
```

Arbeiten mit PriorityQueue

```
NavigableSet<PriorityTask> priorityTasks = new TreeSet<PriorityTask>();  
priorityTasks.add (new PriorityTask (mikePhone, Priority.MEDIUM));  
priorityTasks.add (new PriorityTask (paulPhone, Priority.HIGH));  
priorityTasks.add (new PriorityTask (databaseCode, Priority.MEDIUM));  
priorityTasks.add (new PriorityTask (interfaceCode, Priority.LOW));  
  
assert priorityTasks.toString().equals(  
    "[phone Paul: HIGH, code db: MEDIUM, phone Mike: MEDIUM, code gui: LOW]"
```

Teilbereiche mit SortedSet-Methoden

```
public SortedSet<E> subSet (E fromElement, E toElement);  
public SortedSet<E> headSet (E toElement);  
public SortedSet<E> tailSet (E fromElement);
```

- ▶ `subSet` liefert alle Elemente \geq `fromElement` und $<$ `toElement`
- ▶ `headSet` liefert alle Elemente $<$ `toElement`
- ▶ `tailSet` liefert alle Elemente \geq `fromElement`
- ▶ Die Argumente selbst müssen weder in der Menge noch im Ergebnis enthalten sein.
- ▶ Das `fromElement` *kann* dazugehören.
- ▶ Das `toElement` gehört *nie* zum Ergebnis.

Beispiele mit SortedSet-Methoden

- ▶ Mit den Teilbereichsmethoden kann eine Taskliste nach Prioritäten unterteilt werden.
- ▶ Dafür ist eine “kleinste” Task erforderlich:

```
public class EmptyTask extends Task {  
    public EmptyTask() {}  
    public String toString() { return "";}  
}
```

Beispiele mit SortedSet-Methoden (Forts.)

```
PriorityTask firstLowPriorityTask =  
    new PriorityTask (new EmptyTask(), Priority.LOW);  
SortedSet<PriorityTask> highAndMediumPriorityTasks =  
    priorityTasks.headSet (firstLowPriorityTask);  
assert highAndMediumPriorityTasks.toString().equals(  
    "[phone Paul: HIGH, code db: MEDIUM, phone Mike: MEDIUM]");
```

Oder auch

```
PriorityTask firstMediumPriorityTask =  
    new PriorityTask (new EmptyTask(), Priority.MEDIUM);  
SortedSet<PriorityTask> mediumPriorityTasks =  
    priorityTasks.subSet (firstMediumPriorityTask, firstLowPriorityTask);  
assert mediumPriorityTasks.toString().equals(  
    "[code db: MEDIUM, phone Mike: MEDIUM]");
```

Grenzen der Teilbereichsmethoden

- ▶ Angenommen, gesucht ist die Menge aller Aufgaben mit
 - ▶ mittlerer Priorität
 - ▶ bis `mikePhone` inklusive
- ▶ Zur Verwendung von `subSet` müsste die nächstgrößere Task zu `mikePhone` konstruiert werden.
- ▶ Kann mit Trick erreicht werden.
(Was ist der Nachfolger von "Mike" in der natürlichen Ordnung auf `String`?)
- ▶ Besser: Verwende `NavigableSet`!

Eigenschaften der Teilbereiche

- ▶ Die Teilbereiche sind **keine** neuen, eigenständigen Mengen!
- ▶ Einfügen und Löschen in die unterliegende Menge wirkt auch auf den Teilbereichen

```
PriorityTask logicCodeMedium =
    new PriorityTask (logicCode, Priority.MEDIUM);
priorityTasks.add (logicCodeMedium);
assert mediumPriorityTasks.toString().equals(
    "[code db: MEDIUM, code logic: MEDIUM, phone Mike: MEDIUM]");
```

- ▶ und umgekehrt:

```
mediumPriorityTasks.remove (logicCodeMedium);
assert priorityTasks.toString().equals(
    "[phone Paul: HIGH, code db: MEDIUM, phone Mike: MEDIUM, code gui: LOW]");
```

NavigableSet

Weiterentwicklung von SortedSet

```
interface NavigableSet<E> extends SortedSet<E> {  
    public E pollFirst();  
    public E pollLast ();  
    public NavigableSet<E> subSet (E fromElement, boolean fromInclusive,  
                                  E toElement, boolean toInclusive);  
    public NavigableSet<E> headSet (E toElement, boolean toInclusive);  
    public NavigableSet<E> tailSet (E fromElement, boolean fromInclusive);  
    public E ceiling (E e);  
    public E floor (E e);  
    public E higher (E e);  
    public E lower (E e);  
    public NavigableSet<E> descendingSet();  
    public Iterator<E> descendingIterator();  
}
```

Erste und letzte Elemente

```
public E pollFirst();  
public E pollLast ();
```

- ▶ liefern das erste bzw letzte Element der Menge und **entfernen es**
- ▶ oder `null`, falls leer
- ▶ Unterschiede zu `first` und `last`:
 - ▶ werfen Exception, falls Menge leer
 - ▶ Elemente werden nicht entfernt
- ▶ Beispiel: Die nächst-fällige Aufgabe

```
PriorityTask nextTask = priorityTasks.pollFirst();  
assert nextTask.toString().equals("phone Paul: HIGH");
```

Teilbereiche

```

public NavigableSet<E> subSet (E fromElement, boolean fromInclusive,
                               E toElement, boolean toInclusive);
public NavigableSet<E> headSet (E toElement, boolean toInclusive);
public NavigableSet<E> tailSet (E fromElement, boolean fromInclusive);

```

- ▶ Verbesserung der Methodend von `SortedSet<E>`:
Zugehörigkeit der `from` und `to` Elemente kann separat spezifiziert werden.
- ▶ Beispiel: Alle MEDIUM-Priorität Aufgaben \leq `mikePhone`

```

PriorityTask mikePhoneMedium =
    new PriorityTask (mikePhone, Priority.MEDIUM);
NavigableSet<E> closedInterval = priorityTasks.subSet(
    firstMediumPriorityTask, true, mikePhoneMedium, true);
assert closedInterval.toString().equals(
    "[code db: MEDIUM, phone Mike: MEDIUM]");

```

Bestmögliche Treffer

- ▶ `E ceiling (E e)`
liefert das kleinste Element $\geq e$ oder `null`
- ▶ `E floor (E e)`
liefert das größte Element $\leq e$ oder `null`
- ▶ `E higher (E e)`
liefert das kleinste Element $> e$ oder `null`
- ▶ `E lower (E e)`
liefert das größte Element $< e$ oder `null`

Beispiel (Bestmögliche Treffer)

- ▶ Aufgabe: Finde in einer Menge von `String` die größten drei Strings \leq "x-ray".

```
NavigableSet<String> stringSet = new TreeSet<String> ();  
Collections.addAll (StringSet, "abc", "cde", "x-ray", "zed");  
String last = stringSet.floor ("x-ray");  
assert last.equals ("x-ray");  
String secondToLast = last == null ? null : stringSet.lower (last));  
String thirdToLast =  
secondToLast == null ? null : stringSet.lower (secondToLast);
```

Navigieren in umgekehrter Reihenfolge

```
public NavigableSet<E> descendingSet();  
public Iterator<E> descendingIterator();
```

- ▶ Verwendung der umgekehrten Ordnung, ohne zuvor einen Comparator explizit zu konstruieren.

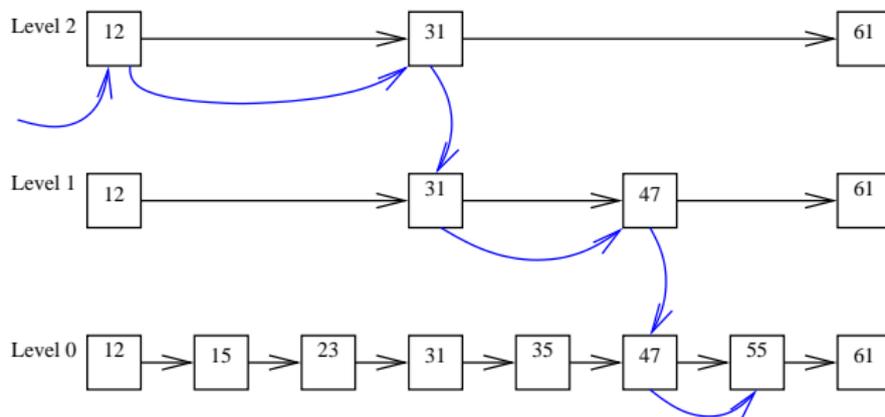
```
NavigableSet<String> headSet = stringSet.headSet (last, true);  
NavigableSet<String> reserveHeadSet = headSet.descendingSet();  
assert reserveHeadSet.toString().equals("[x-ray, cde, abc]");  
String conc = " ";  
for (String s : reserveHeadSet) {  
    conc += s + " ";  
}  
assert conc.equals(" x-ray cde abc ");
```

TreeSet<E>

- ▶ Implementierung von NavigableSet<E>
- ▶ durch balancierten binären Suchbaum
- ▶ Verwendet Rot-schwarz Baum (*red-black tree*)

ConcurrentSkipListSet<E>

► Implementierung von NavigableSet<E>



- Level 0 enthält alle Elemente
- Level $n + 1$ enthält eine Teilliste von Level n
- Suchen: Beginne bei höchstem Level
- Einfügen: Beginne bei 0; werfe eine Münze; falls Kopf, füge ins nächsthöhere Level ein; usw

Zusammenfassung: Aufwand

	add	contains	next
HashSet	$O(1)$	$O(1)$	$O(h)$
LinkedHashSet	$O(1)$	$O(1)$	$O(1)$
CopyOnWriteArraySet	$O(n)$	$O(n)$	$O(1)$
EnumSet	$O(1)$	$O(1)$	$O(m)$
TreeSet	$O(\log n)$	$O(\log n)$	$O(\log n)$
ConcurrentSkipListSet	$O(\log n)$	$O(\log n)$	$O(1)$

- ▶ h ist die Kapazität der Hashtabelle
- ▶ m ist die Anzahl der Element der Enumeration