n|w

Java Collections (Sets & Bags)

Algorithmen und Datenstrukturen 2

Lösungsbesprechung

- Equals-Methode / == vs. Equals
- Im Normalfall keine UnsupportedOperationExceptions mehr
- Kapazität vs. Size (Kapazität nicht Default-Kapazität)
- Aufräumen, wenn Element gelöscht wird (Memory Leak Test)
- Sorted muss eine "Ordnungsrelation" definiert haben
- e.equals(null) => Immer false oder NullPointerException, deshalb: e == null
- Comparable / compareTo
- Size-Variable (nicht O(n))
- Binäre Suche mit Parameter (Arrays.*binarySearch(data, 0, size(), o))*
- toArray-Methode (kopieren des Arrays)

Wildcard Generics - Extends

- In Java leiten (u.a.) die Klassen `Integer` und `Double` von `Number` ab
- **Extends**: Ist eine Obergrenze => «Producer Extends» (Producer aus Sicht der Collection)

```
public static void main(String[] args) {
    Collection<? extends Number> numbers = createCollectionExtendingNumber();
    numbers.add(Integer.valueOf(1)); // Compile error
    numbers.add(Double.valueOf(3.5)); // Compile error
    numbers.add(new Zero()); // Compile error
    Integer i = numbers.iterator().next(); // Compile error
    Number num = numbers.iterator().next(); // The only thing I know!
}

private static Collection<? extends Number> createCollectionExtendingNumber() {
    Collection<? extends Number> collection;
    collection = new ArrayList<Number>();  // Number "extends" Number (in this context)
    collection = new ArrayList<Integer>(); // Integer extends Number
    collection = new ArrayList<Double>();  // Double extends Number
    return collection;
}

private static class Zero extends Number{
```

Wildcard Generics - Super

- In Java leiten (u.a.) die Klassen `Integer` und `Double` von `Number` ab
- **Super**: Ist eine Untergrenze => «Consumer Super» (Consumer aus Sicht der Collection)

```
private static void superExample() {
    Collection<? super Integer> numbers = createCollectionSuperInteger(); // Contains Integers, Numbers or Objects
    Integer i = numbers.iterator().next(); // Compile error (returns Object)
    Number num = numbers.iterator().next(); // Compile error (returns Object)
    Object o = numbers.iterator().next(); // Only thing I know here: Common class Object

    numbers.add(new Object());
    numbers.add(new Zero());
    numbers.add(Integer.valueOf(1)); // The only thing I know
}

private static Collection<? super Integer> createCollectionSuperInteger() {
    Collection<? super Integer> collection;
    collection = new ArrayList<Integer>();  // Integer "extends" Integer (in this context)
    collection = new ArrayList<Number>();  // Integer extends Number
    collection = new ArrayList<Object>(); // Number extends Object
    collection = new ArrayList<Double>(); // Compile error
    return collection;
}
```

Wildcard Generics - Eselsbrücke

- **PECS: Producer extends, Consumer super (immer aus Sicht der Collection)**
  - Anwendungsbeispiel:

```
public class Collections {
  public static <T> void copy(List<? super T> dest, List<? extends T> src) {
      for (int i = 0; i < src.size(); i++)
        dest.set(i, src.get(i));
  }
}
```

- Stackoverflow Beitrag: https://stackoverflow.com/questions/4343202/difference-between-super-t-and-extends-t-in-java

Generics

- `UnsortedSet<E>:` Kann alle Objekte der Klasse E (und Sub-Klassen) darin ablegen.

- `SortedSet<E extends Comparable<E>>:` Kann Objekte der Klasse E (und Sub-Klassen) darin ablegen, die das Comparable-Interface implementieren und sich das Objekt **nur mit Objekten der Klasse E** vergleichen lässt.

- `SortedBag<E extends Comparable<? super E>>:` Kann Objekte der Klasse E (und Sub-Klassen) darin ablegen, die das Comparable-Interface implementieren und sich das Objekt **mit Objekten der Klasse E oder einer Super-Klasse** vergleichen lässt.

Generics

```
public class MyInteger extends Number
implements Comparable<MyInteger> {

private int val =0;


public MyInteger(int val){

this.val = val;

}


@Override

public int compareTo(MyInteger o) {

return Integer.compare(val, o.val);

}
```

```
public class MyInteger2 extends Number
implements Comparable<Number> {

private int val =0;


public MyInteger2(int val){

this.val = val;

}

@Override

public int compareTo(Number o) {

return Integer.compare(val,
o.intValue());
```

Generics

```
public class SortedBag<E extends Comparable<E>>:
```

- `SortedBag<MyInteger> bag = new SortedBag<>(); // OK`
- `SortedBag<MyInteger2> bag = new SortedBag<>(); // Compile Error`

```
public class SortedBag<E extends Comparable<? super E>>:
```

- `SortedBag<MyInteger> bag = new SortedBag<>(); // OK`
- `SortedBag<MyInteger2> bag = new SortedBag<>(); // OK`

Quiz: Problem mit MyInteger & SortedBag (funktioniert korrekt)

```
public class MyInteger extends Number
implements Comparable<MyInteger> {

private int val =0;
public MyInteger(int v){this.val = v;}

public void setValue(int v){this.val = v;}

@Override
public int compareTo(MyInteger o) {
return Integer.compare(val, o.val);
```

```
SortedBag<MyInteger> bag = new SortedBag<>();
MyInteger one = new MyInteger(1);
MyInteger two = new MyInteger(2);
MyInteger three = new MyInteger(3);
bag.add(one);
bag.add(two);
bag.add(three);
System.out.println(bag.contains(three)); // True
// something happens here… (no elements removed)
System.out.println(bag.contains(three)); // False
```

Quiz: Problem mit MyInteger & SortedBag (funktioniert korrekt)

```
public class MyInteger extends Number
implements Comparable<MyInteger> {

private int val =0;
public MyInteger(int v){this.val = v;}

public void setValue(int v){this.val = v;}

@Override
public int compareTo(MyInteger o) {
return Integer.compare(val, o.val);
```

```
SortedBag<MyInteger> bag = new SortedBag<>();
var one = new MyInteger(1);
var two = new MyInteger(2);
var three = new MyInteger(3);
bag.add(one);
bag.add(two);
bag.add(three);
System.out.println(bag.contains(three)); // True
two.setValue(10);
System.out.println(bag.contains(three)); // False
```

Bemerkungen zu Collections

		UnsortedBag	SortedBag	UnsortedSet	SortedSet
	Inhalt nach dem Einfügen von: 12, 5, 28, 47, 12, 8				
Asyptotische Komplexität im **Worst Case** (n Elemente in der Collection)	add(E e);				
	contains(Object o)				
	remove(Object o)				
	Besonders gut geeignet für:				

Bitte für alle Varianten ergänzen

Bemerkungen zu Collections

		UnsortedBag	SortedBag	UnsortedSet	SortedSet
	Inhalt nach dem Einfügen von: 12, 5, 28, 47, 12, 8	12, 5, 28, 47, 12, 8	5, 8, 12, 12, 28, 47	12, 5, 28, 47, 8	5, 8, 12, 28, 47
Asyptotische Komplexität im **Worst Case** (n Elemente in der Collection)	add(E e);	$O(1)$ kein suchen, kein Verschieben	$O(n)$ suchen $O(\log n)$, einf. $O(n)$	$O(n)$ suchen $O(n)$, einf. $O(1)$	$O(n)$ suchen $O(\log n)$, einf. $O(n)$
	contains(Object o)	$O(n)$	$O(\log n)$	$O(n)$	$O(\log n)$
	remove(Object o)	$O(n)$ suchen $O(n)$, entfernen $O(1)$	$O(n)$ suchen $O(\log n)$, entfernen $O(n)$	$O(n)$ suchen $O(n)$, entfernen $O(n)$	$O(n)$ suchen $O(\log n)$, entfernen $O(n)$
	Besonders gut geeignet für:	häufiges Hinzufügen	häufiges Suchen	Set-Semantik, wenn für die Elemente keine Ordnungsrelation definiert ist	Set-Semantik "Duplikate ausgeschlossen"

Bemerkungen zu Collections

- Unsorted: Auch ohne Comparable implementierbar
- Nur wenig Gründe für UnsortedSet