Java Generics & Collections

Praktikum Effizientes Programmieren (Sommersemester 2017)

Dennis Reuling

UNIVERSITÄT SIEGEN

Praktische Informatik

Agenda

Teil 1

Generics

Ein Beispiel...

Elemente nach einer Definition ordnen:

String

```
class RankString(String toRank){
    String[] ranking;
    rank(){
        for(int j = 0;
        j < ranking.length;
        j++){
            //rank
        }
    String[] getRanking(){
    return ranking;
        }
    }
}
```

Integer

```
class RankInteger(int toRank){
    int[] ranking;
    rank(){
        for(int j = 0;
        j < ranking.length;
        j++){
            //rank
        }
    int[] getRanking(){
    return ranking;
        }
    }
}
```

Ein Beispiel...

Elemente nach einer Definition ordnen:

String

```
class RankString(String toRank){
    String[] ranking;
    rank(){
        for(int j = 0;
        j < ranking.length;
        j++){
            //rank
        }
    String[] getRanking(){
    return ranking;
        }
    }
}
```

Integer

```
class RankInteger(int toRank){
    int[] ranking;
    rank(){
        for(int j = 0;
        j < ranking.length;
        j++){
            //rank
        }
    int[] getRanking(){
    return ranking;
        }
    }
}
```

- Viel Redundanz
- Unnötige LOC

Mögliche Lösung

Auf gemeinsamen Typ „zurückziehen“, hier `Object`:

```
class RankObject(Object toRank){
    Object[] ranking;
    rank(){
        for(int j = 0;
        j < ranking.length;
        j++){
            //rank
        }
    }
    Object[] getRanking(){
        return ranking;
    }
}
```

Mögliche Lösung

Auf gemeinsamen Typ „zurückziehen“, hier `Object`:

```
class RankObject(Object toRank){
    Object[] ranking;
    rank(){
        for(int j = 0;
        j < ranking.length;
        j++){
            //rank
        }
    }
    Object[] getRanking(){
        return ranking;
    }
}
```

- `Object` verhindert aber **Typsicherheit**
- Unlesbar für Entwickler, was enthält `ranking[]` ?

Mögliche Lösung

Auf gemeinsamen Typ „zurückziehen“, hier `Object`:

```
class RankObject(Object toRank){
    Object[] ranking;
    rank(){
        for(int j = 0;
        j < ranking.length;
        j++){
            //rank
        }
    }
    Object[] getRanking(){
        return ranking;
    }
}
```

- `Object` verhindert aber **Typsicherheit**
- Unlesbar für Entwickler, was enthält `ranking[]` ?

Daher ▷ **Java Generics**

Allgemeines

Idee

Typparameter einführen, um auf verschiedenen Typen von Objekten (gleich) arbeiten zu können.

Geschichte

- Ursprung: Generic Java (1998)
- Integriert: seit JDK 5 (2004)
- Zukunft: Project Valhalla[1] (t.b.a.)

Übersicht

- Ursprung: Viele Probleme lassen sich **ähnlich** lösen
- Daher: Datenstrukturen (**später** ▷ Collections) oder Methoden teilen sich Eigenschaften
- Beispiele:
  - Elemente hinzufügen, sortieren, ...
  - Operationen ausführen, addieren, ...
- Ziele:
  - Redundanz vermeiden
  - Typsicherheit gewährleisten
  - (Methoden) aufrufe sinnvoll verallgemeinern

Warum nicht **Vererbung** ?

Vererbung vs Generics

Vererbung

- Drückt **Ist-Ein** aus
- Beispiel:
```
class Dog extends Animal(){...}
```
- Wenn Eigenschaften **übernommen** werden und die Subklasse die Oberklasse *richtig* darstellt

Generics

- Drückt **Von** aus
- Beispiel:
```
class Veterinian<T>(){...}
```
- Wenn **auf** verschiedenen Typen gearbeitet werden kann (soll)
- Es werden **keine** Eigenschaften des Typparameter's in der Klasse übernommen

Ein Beispiel zur Typsicherheit

Lösung 1:

```
List list = new ArrayList();
list.add(42);
Integer i = (Integer)list.get(0);
```

Lösung 2:

```
List<Integer> list = new ArrayList<Integer>();
list.add(42);
Integer i = list.get(0);
```

Welche Lösung ist besser ? Warum ?

Vorkommen in freier Wildbahn

- Klasse:
```
class GenericTreeNode<T>(){...}
```
- Methoden:
```
void setElement(T t){...}
```
- Attribute:
```
private K key;
```

Übliche Typparameter

T Typ

E Element

K Key

V Value

N Number

Subtyping

Ist folgender Code korrekt ?

```
class Dog extends Animal(){...}

class Veterinian<T>(){...}

Veterinian<Animal> va = ... ;
Veterinian<Dog> vd = va;
```

Subtyping

Ist folgender Code korrekt ?

```
class Dog extends Animal(){...}

class Veterinian<T>(){...}

Veterinian<Animal> va = ... ;
Veterinian<Dog> vd = va;
```

Nein, aufgrund von Type Erasure (▷ später!)

Subtyping

Generell:

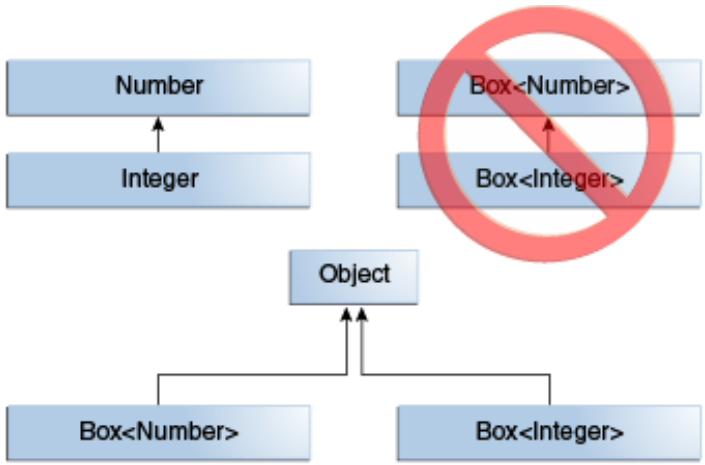

Abbildung: Vererbungsbeziehung Generics [2]

Subtyping

Generell:

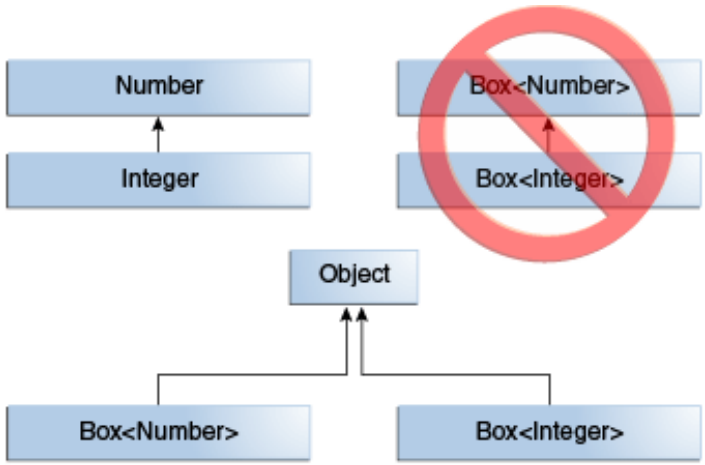

Abbildung: Vererbungsbeziehung Generics [2]

Lösung: Wildcards (mit Subtyping)

Wildcards - Allgemein

- Idee: Typparameter **einschränken**, um eingrenzen zu können
- Ansatz: Typhierarchien direkt mit dem Parameter angeben
- Möglichkeiten:
  - Upper
  - Lower
  - Unbound
- Darstellung = ?

Wildcards - Beispiele

Wildcards - Beispiele

Upper:

```
class Veterinian<? extends Animal>(){...}
```

▷ Tierärzte dürfen nur Tiere behandeln.

Wildcards - Beispiele

Upper:

```
class Veterinian<? extends Animal>(){...}
```

▷ Tierärzte dürfen nur Tiere behandeln.

Lower:

```
class Veterinian<? super Animal>(){...}
```

▷ Tierärzte dürfen nur Tiere und alle Supertypen behandeln (Katzen).

Wildcards - Beispiele

Upper:

```
class Veterinian<? extends Animal>(){...}
```

▷ Tierärzte dürfen nur Tiere behandeln.

Lower:

```
class Veterinian<? super Animal>(){...}
```

▷ Tierärzte dürfen nur Tiere und alle Supertypen behandeln (Katzen).

Unbound:

```
class Veterinian<?>(){...}
```

▷ Tierärzte behandeln alle gleich (schlecht/gut).

Wildcards - Verwendung

Richtlinie (bei Klassen):

- IN-Variable: Upper
- OUT-Variable: Lower
- Variable nicht nötig bzw `Object` reicht aus: Unbound
- IN & OUT gleichzeitig: Keine

Wiederholung

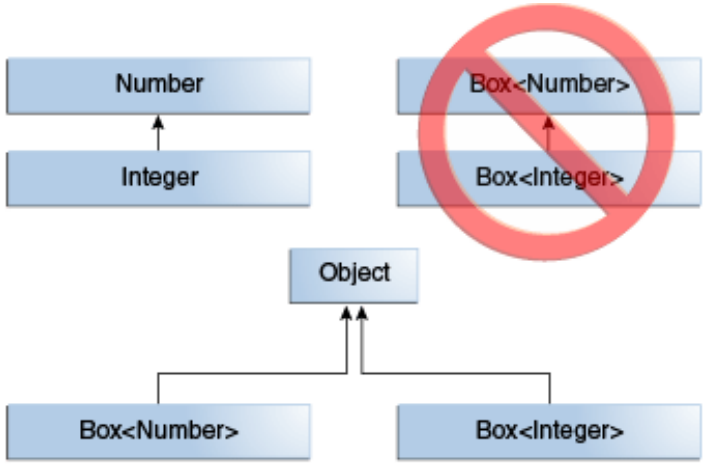

Abbildung: Vererbungsbeziehung Generics [2]

Warum `Object` als gemeinsame Superklasse ?

Grundlegendes

Um Typsicherheit(zur Compilezeit) zu gewährleisten, geht Java folgendermaßen vor:

1. Alle Typparameter werden durch die **Bounds** oder aber `Object` ersetzt, falls keine Bounds vorhanden.
2. Typkonversionen werden eingefügt, falls nötig
3. Bridge Methods werden zur Erhaltung von Polymorphie eingefügt

Beispiele

Unbound - vorher

```
class Veterinian<T>(){
    heal(T){
        //...
    }
}
```

Upper - vorher

```
class Veterinian<? extends Animal>(
    heal(?){
        //...
    }
}
```

Unbound - nachher

```
class Veterinian(){
    heal(Object){
        //...
    }
}
```

Upper - nachher

```
class Veterinian(){
    heal(Animal){
        //...
    }
}
```

Einschränkungen (Auszug)

- Primitive Typen können nicht verwendet werden (`int`, `float`, ...)
  (Ist aber dank Autoboxing meist kein Problem)
- Typparameter können nicht instanziiert werden
- Statische Attribute unterstützen keine Typparameter
- Arrays können nicht aus parametrisierten Typen abgeleitet werden

Einschränkungen (Auszug)

- Primitive Typen können nicht verwendet werden (`int`, `float`, ...)
  (Ist aber dank Autoboxing meist kein Problem)
- Typparameter können nicht instanziiert werden
- Statische Attribute unterstützen keine Typparameter
- Arrays können nicht aus parametrisierten Typen abgeleitet werden

Aber: Sie sehen einfach komplexer aus ;-)

```
public static <K, V> SortedMap<K, List<V>>
sortedClassifiedLists(Collection<? extends V> collection,
Classifier<K, V> classifier, Comparator<K> comparator) {...}
```

Teil 2

Collections

Allgemeines

Idee

Bibliothek, um Gruppen von (zusammengehörigen) Elementen (generisch) zu verwalten.
Vorbild / Inspiration: C++ STL

Geschichte

- Ursprung: Collections Package (1997)
- Integriert: seit JDK 2 (1998)

Umsetzung

Verwendet Java Generics sowie allgemeingültige **Schnittstellen**

Übersicht

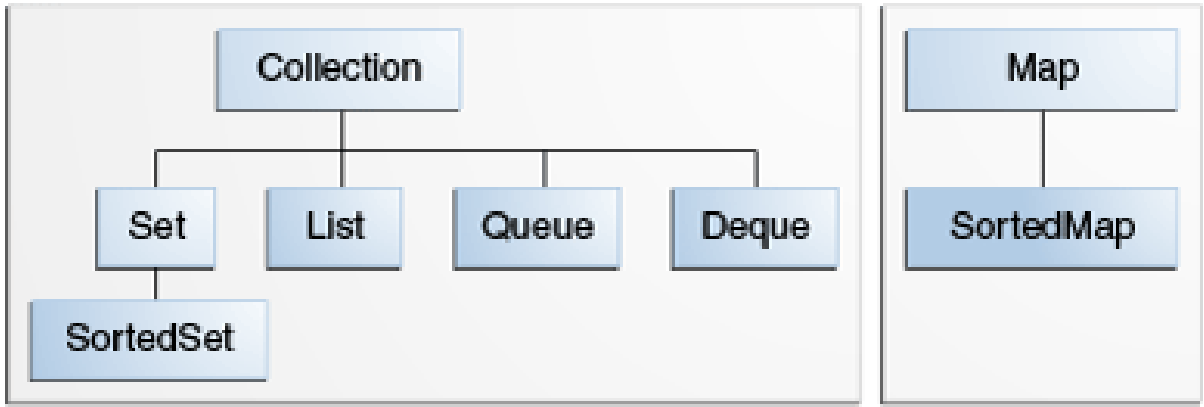

Abbildung: Java Collections Schnittstellen [2]

Collections Interface

- Bietet **allgemeingültige** Aufrufe an
- Konstruktor-Aufruf wandelt jegliche Collection in die gewünschte um, Beispiel:

```
List<String> names = ...;
Set<String> uniqueNames = new HashSet<String>(names);
```

- Basis Methoden:
  - `int size()`, `boolean isEmpty()`, `Iterator<E> iterator()`
- Element-spezifische Methoden:
  - `boolean contains(Object element)`, `boolean add (E element)`, `boolean remove(E element)`
- Gruppen-spezifische Methoden:
  - `boolean containsAll(Collection<?> c)`, `void clear()`, `Object[] toArray()`

Set

Beschreibung

Sets sind Kollektionen, die keine **Duplikate** enthalten dürfen.

Methoden

- Keine Erweiterung zu allgemeiner Collection
- Einschränkungen hinsichtlich Elemente, z.B. Einfügen von bereits vorhandenem Element nicht erlaubt (`add` liefert `false`)
- Semantik der mathematischen **Mengenlehre** nachgebildet

Beispiel (Natürliche Gleichheit)

Vorher:

```
Collection<Integer> ints = {4,1,6,8,6};
```

Nachher:

```
Set<Integer> ints = {4,1,6,8};
```

Gleichheit

- Gleichheit muss (selbst) definiert werden, um Collections sinnvoll (und performant) verwenden zu können.
- Zwei Methoden verfügbar:
  - `boolean equals(Object obj)`:
    - **Muss** `true` liefern, wenn zwei Objekte gleich sind
    - **Muss** eine Äquivalenzrelation sein
  - `int hashCode()`:
    - Berechnet einen int-Wert, der bei gleichen Objekten den gleichen Wert liefern **muss** (d.h. wenn `equals` true liefert)
    - **Könnte** bei verschiedenen Objekten den gleichen Wert liefern
- Beispiel:

```
String s1 = "hallo";
String s2 = "Hallo";
s1.equals(s2) -> false;
s1.hashCode() == s2.hashCode() -> true;
```

List

Beschreibung

Lists sind Kollektionen, die eine **Ordnung** besitzen.

Methoden

- Erweiterungen zu allgemeiner Collection: Positionsbezogene(r)
  - Zugriff
  - Suche
  - Iterator
  - Bereichs-Ansicht

Beispiel (Natürliche Ordnung)

Vorher:

```
Collection<Integer> ints = {4,1,6,8,6};
```

Nachher:

```
List<Integer> ints = {1,4,6,6,8};
```

Ordnung

- Eine Ordnung muss (selbst) definiert werden, um Collections sinnvoll (und performant) verwenden zu können.
- Zwei Möglichkeiten:
  - Comparable Interface (in o1) implementieren
    `int compareTo(Object o2)`
  - Comparator implementieren
    `int compare(T o1, T o2)`
  - Beide liefern:
    - 0, wenn $o1 = o2$
    - -1, wenn $o1 < o2$
    - 1, wenn $o1 > o2$
- Beispiel:

```
List<Integer> ints = {1,4,6,6,8};
class Invert implements Comparator<Integer>{...}
Collections.sort(ints, new Invert());
List<Integer> ints = {8,6,6,4,1};
```

Queue

Beschreibung

Queues sind Kollektionen, die eine **Verarbeitungsreihenfolge** definieren (üblich FIFO).

Methoden

- Erweiterungen zu allgemeiner Collection: Verbrauch von Elementen:
- Erstes Element holen und entfernen
- Kapazitäten können festgelegt werden

Beispiel (Kapazität 3)

Vorher:

```
Collection<Integer> ints = {4,1,6,8,6};
```

Nachher:

```
Queue<Integer> ints = {4,1,6};
```

Deque

Beschreibung

Deques sind Queues, die von **beiden** Seiten manipuliert werden.

Methoden

- Erweiterungen zu Queue:
  - Letztes Element holen und entfernen
  - Erstes/Letztes Vorkommen holen und entfernen

Map

Beschreibung

- Maps sind **keine** Kollektionen, aber teil Teil der Standard Bibliothek.
- Sie beschreiben eine Abbildung zwischen einem **eindeutigen** Schlüssel und einem Wert.

Methoden

Manipulation von Schlüsseln und Werten:

- `put(K key, V value)`
- `remove(Object key)`

Teile einer Map können als Collections abgebildet werden:

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

Map - Beispiele

Geordnet nach natürlicher Ordnung (Int):

```
Map<Integer, String> matrTOName =
        {111222 -> "Manfred Muster",
        333444 -> "Eva Muster"}
```

Komplexe Sachverhalte durch Verschachtelung:

```
Map<Uni,Map<Integer,String>> uniToStudenten =
        {UniSiegen -> {
        111222 -> "Manfred Muster",
        333444 -> "Eva Muster"}}
```

Implementierungen

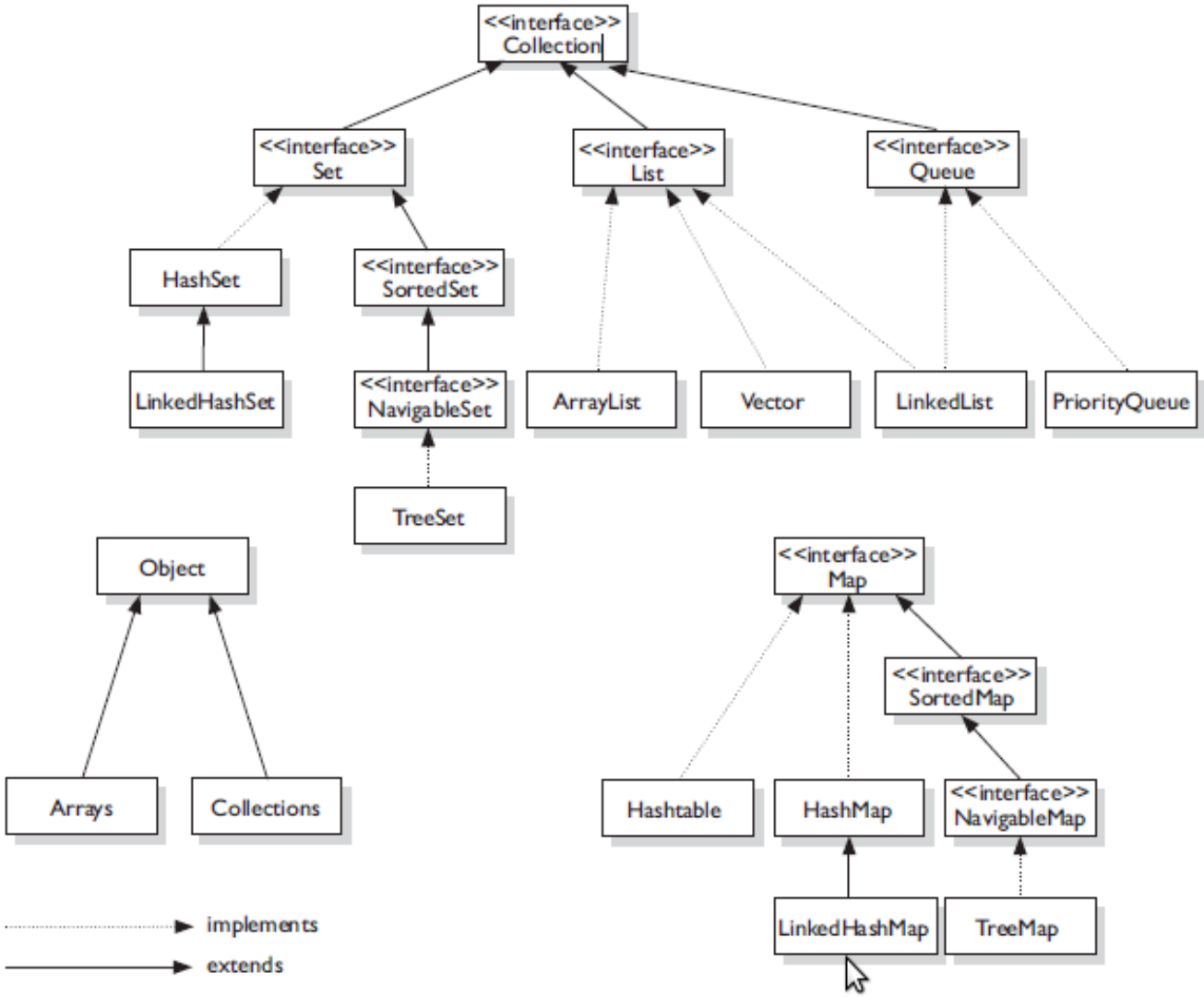

Abbildung: Java Collections Implementierungen [5]

Teil 3

Literatur

Weiterführende Literatur

1 Maurice Naftalin, Philip Wadler - Java Generics and Collections, O' Reilly Verlag, 2006

2 Oracle - Java Tutorials (`http://docs.oracle.com/javase/tutorial/`)

3 Angelika Langer - JavaFAQ (`http://www.angelikalanger.com/GenericsFAQ/JavaGenericsFAQ.html`)

4 Project Valhalla (`http://openjdk.java.net/projects/valhalla/`)

5 Java Collections Cheat Sheet (`http://pedrocardoso.eu/scjp-java-collections-cheat-sheet/`)