Einführung in die Informatik

Jochen Hoenicke

Software Engineering
Albert-Ludwigs-University Freiburg

Sommersemester 2014

Generische Typen

Generische Typen: Motivation

In der letzten Vorlesung hatten wir eine verkettete Liste programmiert. Das funktioniert ganz gut, auch um z.B. Zeichenketten einzufügen:

```
List list = new DoublyLinkedList();
list.add("Hallo"); // impliziete
   Typkonvertierung
```

Wenn wir allerdings eine getFirst()-Methode hinzufügen, muss man beim auslesen den Typ konvertieren.

```
String elem = (String) list.getFirst();
```

Wenn wir den falschen Typ einfügen, meldet der Compiler keine Fehler. Erst beim Auslesen schlägt die Typkonvertierung fehl.

Vergleich: Arrays

Bei Arrays gibt es für jeden Typ einen zugehörigen ArrayTyp, z.B. `String[]` um `String`-Objekte zu speichern.
Können wir etwas ähnliches für Listen tun?
Ja, es gibt generische Datentypen

Generische Liste

In Java kann eine Klasse einen Typparameter haben, der in eckigen Klammern übergeben wird.

```
public class SinglyLinkedList<E> {
    class ListEntry {
        E         data;
        ListEntry next;
    }
    public void add(E data) {...
    public E getFirst() {...
```

- `E` ist immer ein Referenztyp, d.h. er erbt von `Object`.
- `E` kann fast wie ein normaler Typ benutzt werden.
- Es gibt aber Dinge, die nicht funktionieren und zu Warnungen führen, z.B. Typkonvertierung `(E)`.

Benutzung von Generics

Wenn man die Klasse `DoublyLinkedList<E>` benutzen will, muss man angeben was `E` ist (man muss `E` instanziieren):

```
DoublyLinkedList<String> list = new
    DoublyLinkedList<String>();
list.add("Hallo");  // okay
String s = list.getFirst(); // okay
list.add((Integer) 4); // Compilerfehler
```

Der Compiler prüft dann auch, ob die Typen passen.

Wildcards

Wenn einem der Typ der Instanziierung egal ist, kann man auch Wildcards benutzen:

```
    private void printFirst(List<?> list) {
        if (!list.isEmpty())
            Object o = list.getFirst();
            System.out.println(o.toString());
        }
    }
```

Allerdings kann man dan zu solch einer Liste nichts hinzufügen (nur mit Compilerwarnungen).
Man kann auch noch den Typ von `?` genauer eingrenzen: Die Klasse `List<? implements Comparable>` speichert Objekte, die Comparable implementieren.

Vererbungshierarchie

Eine generische Liste `List<String>` erbt nicht von `List<Object>` (siehe Übung). Das hängt damit zusammen, dass `List<String>` die Methode `add(Object data)` nicht implementiert.
Allerdings ist `List<String>` eine Subklasse von

- `List<?>`,
- `List<? extends String>`,
- `List<? implements Comparable>`.
- `List<? super String>`.

Fallstricke

- Java unterstützt keine Laufzeittypen für Generics. Java merkt sich zur Laufzeit nur den Basistyp List, nicht die generischen Parameter.
- Eine Typkonvertierung z.B. auf `List<String>` kann nicht zur Laufzeit überprüft werden.
- Auch `instanceof` funktioniert nicht auf generischen Typen.
- Eine Typkonvertierung auf `E` in der Klasse $List < E >$ kann zur Laufzeit nicht überprüft werden.
- Man kann kein Array vom Typ `E[]` erzeugen.

Kompatibilität

Aus historischen Gründen gibt es in Java zu der Klasse `List<E>` den Typ `List`, der sich wie `List<Object>` verhält. Auch hier gibt es Compilerwarnungen.

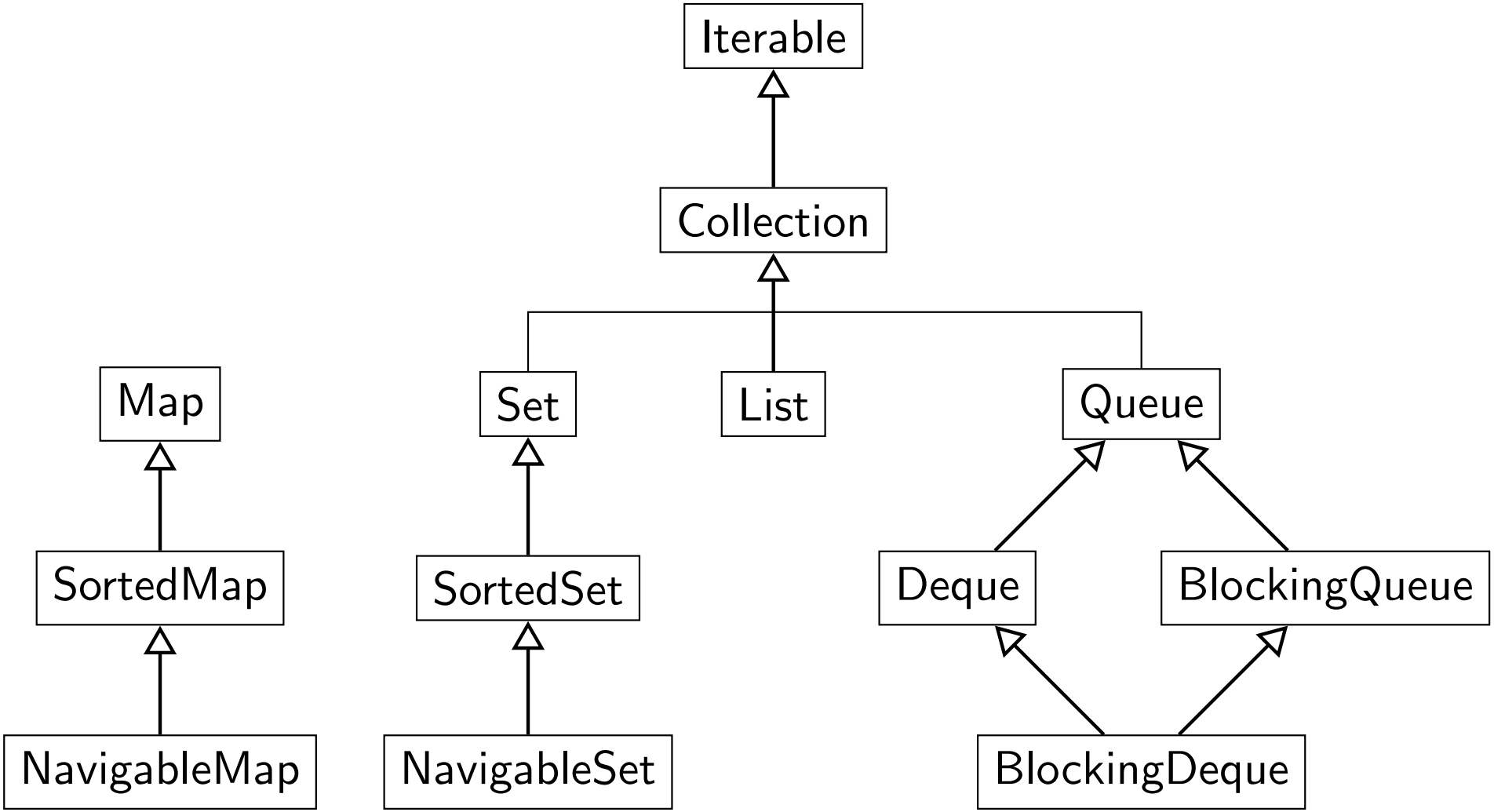
Collection-Bibliothek

Collection-Bibliothek

Die Java-Bibliothek stellt eine Reihe von Klassen zur Verfügung um Daten zu speichern.

- Container-Klassen um Objekte in verketteten Listen, Arrays, oder Bäumen zu speichern.
- Abbildungen (Maps), um Objekte miteinander zu verknüpfen
- Interfaces, die es erlauben, Objekte hinzuzufügen, entfernen, suchen, iterieren.
- Verschiedene Implementierungen, die für unterschiedliche Zwecke geeignet sind.

Collection-Hierarchie: Interfaces



Iterable

Das Interface `Iterable<E>` hat nur eine Methoden:

- `Iterator<E> iterator()` gibt einen Iterator zurück der alle Objekte aufzählt.

Der Iterator hat wiederum drei Methoden:

- `boolean hasNext()`: gibt true zurück, wenn es noch weitere Objekte gibt.
- `E next()`: gibt bei jedem Aufruf das nächste Element zurück. Wirft eine Exception, wenn es kein weiteres Element gibt.
- `void remove()` (optional): löscht das letzte Element aus der Collection.

Damit kann man alle Objekte aufzählen.

Iterable und for-Schleifen

Wenn eine Klasse `Iterable<E>` implementiert gibt es eine „schöne“ Syntax der for-Schleife:

```
Iterable<String> list = ....;
for (String s : liste) {
    ....
}
```

Hier wird der Schleifenrumpf für jedes Element der Liste einmal aufgerufen. Es ist eine Abkürzung für:

```
Iterable<String> list = ....;
Iterator<String> iterator = liste.iterator();
while (iterator.hasNext()) {
    String s = iterator.next();
    ....
}
```

Collection

Das Interface `Collection<E>` hat zusätzliche Methoden:

- `boolean add(E e)` fügt ein Element hinzu.
- `boolean addAll(Collection<? extends E> c)` fügt alle Elemente einer anderen Collection hinzu. r anderen Collection.
- `boolean clear()` löscht alle Elemente.
- `boolean contains(Object o)` prüft ob ein Element enthalten ist.
- `boolean containsAll(Collection<?> e)` prüft ob alle Elemente enthalten sind.
- `boolean isEmpty()` prüft ob die Collection leer ist.
- `boolean remove(Object o)` löscht ein Element.
- `boolean removeAll(Collection<?> c)` löscht alle Elemente eine
- `boolean retainAll(Collection<?> c)` löscht alle Elemente, die nicht in der anderen Collection liegen.
- `int size()` gibt die Anzahl der Elemente zurück.
- `T[] toArray(T[])` gibt ein Array mit allen Elementen zurück.

equals und hashCode

Die Klasse `Object` definiert zwei für Collections relevante Methoden:

- `boolean equals(Object o)` vergleicht das Objekt mit einem anderen Objekt, und gibt `true` zurück, falls die Objekte gleich sind.

  Eigenschaften `o.equals(o)== true`, `o.equals(null)== false` und `o1.equals(o2)== o2.equals(o1)` und Transitivität sollten gelten.
- `int hashCode()` gibt einen „Fingerabdruck“ des Objekts als `int` zurück. Gleiche Objekte müssen den gleichen Fingerabdruck haben. Verschiedene Objekte sollten (aus Effizienzgründen) verschiedene Fingerabdrücke haben.

  Letzteres ist nicht immer möglich, da es nicht genug Integer gibt. Es sollte aber mit großer Wahrscheinlichkeit gelten.

Collections: Übersicht

Es gibt vier Hauptarten von Collections

- `Set`: Speichert Objekte ohne Duplikate in beliebiger Reihenfolge (oder sortiert in `SortedSet`).
- `List`: Speichert Objekte mit Duplikaten in fester Reihenfolge.
- `Queue`: Speichert Objekte mit Duplikaten (mit oder ohne Reihenfolge) mit schnellem Zugriff auf das erste Element.
  Die Erweiterung `Deque` (double ended queue) erlaubt das schnelle hinzufügen und entfernen sowohl vorne, als auch hinten.
- `Map`: Speichert Assoziation zwischen Schlüssel- und Wertobjekten, mit schnellem Zugriff auf einen Wert anhand des Schlüssels. Obwohl `Map` selbst keine Collection ist, bilden die Schlüssel ein `Set` und die Werte eine `Collection`.

Implementierungen

Zu jedem Interface in der Collection-Hierarchie gibt es ein oder mehrere Implementierungen:

- `Map`: `HashMap` und `LinkedHashMap`
- `SortedMap`/`NavigableMap`: `TreeMap`
- `Set`: `HashSet` und `LinkedHashSet`
- `SortedSet`/`NavigableSet`: `TreeSet`
- `List`: `ArrayList` und `LinkedList`
- `Queue`: `PriorityQueue`.
- `Deque`: `ArrayDeque`, `LinkedList`

Sie unterscheiden sich vor allem in der Effizienz einiger Methoden.

Die Interfaces `Comparator<E>` und `Comparable<E>`

Für `SortedSet` muss man Objekte vergleichen können. Dafür gibt es das Interface `Comparator<E>`, mit einer Methode:

- `int compare(E x, E y)`: Gibt einen negativen Wert zurück, falls $x < y$, Null, wenn x = y und positiven Wert wenn $x > y$ ist. Dabei kann größer/kleiner eine beliebige totale Ordnung sein.

Statt einen `Comparator` kann eine Klasse auch eine natürliche Ordnung auf sich selbst definieren und `Comparable<E>` implementieren:

- `int compareTo(E y)` hat den gleichen Kontrakt wie $compare(this, y)$.