

17. Java Collections

Generische Typen, Interfaces, Java Collections, Iteratoren

Daten Organisieren

- Datenstrukturen, die wir kennen
 - Arrays – Sequenzen fixer Grösse
 - Strings – Buchstabensequenzen
 - Verkettete Listen (bisher: für festen Elementtyp selbstgemacht)

Heute:

- Allgemeines Collection Konzept der Java API (Application Programming Interface = Programmierschnittstelle)
 - ArrayList auf generischem Elementtyp – dynamischer als Arrays
 - LinkedList, Sets, Queues
- Allgemeines Map Konzept der Java API

Generische Liste in Java: java.util.List

```
import java.util.ArrayList;
import java.util.List;
...
// Liste von Strings
List<String> list = new ArrayList<String>();

list.add("abc");
list.add("xyz");
list.add(1, "123"); // Fuege 123 an Position 1 ein
System.out.println(list.get(0)); // abc
```

Generische Liste in Java: java.util.List

```
import java.util.ArrayList;
import java.util.List;
...
// Liste von Strings
List<String> list = new ArrayList<String>();
list.add("abc");
list.add("xyz");
list.add(1,"123"); // Fuege 123 an Position 1 ein
System.out.println(list.get(0)); // abc
```

Typ Parameter („Parametrischer Polymorphisms“)

In Java kann man eine Klasse mit einem Typ parametrisieren

```
// ListNode mit generischem Werttyp T
```

```
class ListNode <T> {  
    T value;  
    ListNode<T> next;  
  
    ListNode (T value, ListNode<T> next){  
        this.value = value; this.next = next;  
    }  
}
```

← Platzhalter T

Konkreter Typ string wird für T in ListNode eingesetzt.

Verwendung:

```
ListNode<String> n = new ListNode<String>("ETH", null);
```

Beispiel: Generischer Stack

```
public class Stack<T>{  
    private ListNode<T> top_node; // initialized with null  
    public void push(T value){  
        top_node = new ListNode<T>(value, top_node);  
    }  
    public T pop(){...}  
    public void output(){...}  
}
```

...

```
Stack<String> s = new Stack<String>();  
s.push("ETH");  
s.push("Hello");  
s.output(); // Hello ETH
```

Stack von Integers

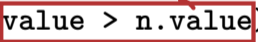
- Java Generics können nur auf Objekten operieren
- Fundamentaltypen `int`, `float` (etc.) sind keine Objekte
- Java bietet Wrapperklassen für Fundamentaltypen an, z.B. den Typ `Integer`
- Java macht *autoboxing* und packt einen Fundamentaltyp automatisch in eine Wrapperklasse ein, wo nötig.

```
Stack<Integer> s = new Stack<Integer>();  
s.push(3);           // auto boxing: int -> Integer  
int a = s.pop(a);    // auto unboxing: Integer -> int
```

Sortierte Liste?

```
public class SortedList <T>{  
    private ListNode<T> head; // initialized with null  
    ...  
    // in a sorted way (sorted ascending by value)  
    public void insert(T value){  
        ListNode<T> n = head;  
        ListNode<T> prev = null;  
        while (n != null && value > n.value){  
            prev = n;  
            n = n.next;  
        }  
        ...  
    }  
}
```

error: bad operand types
for binary operator '>'



Sortierte Liste!

```
public class SortedList <T extends Comparable<T>>{
    private ListNode<T> head; // initialized with null
    ...
    // in a sorted way (sorted ascending by value)
    public void insert(T value){
        ListNode<T> n = head;
        ListNode<T> prev = null;
        while (n != null && value.compareTo(n.value)>0){
            prev = n;
            n = n.next;
        }
        ...
    }
}
```

↑
extends Comparable<T> stellt sicher,
dass die Methode T.compareTo existiert.

Interfaces

Ein Interface (übersetzt: Schnittstelle) definiert Funktionalität einer potentiellen Implementation durch eine Klasse

```
public interface Comparable<T>
{
    public int compareTo (T o);
}
```

Jede Klasse **T**, welche **Comparable<T>** implementiert, muss die Methoden des Interfaces **Comparable<T>** anbieten.

```
public class Present implements Comparable<Present>{
    // must contain this
    public int compareTo(Present o){...}
}
```

Vergleichbare Geschenke

```
public class Present implements Comparable<Present>{
    int value;
    String content;

    public Present(int value, String content){
        this.value = value; this.content = content;
    }
    // returns if this present is more valuable than the other
    public int compareTo(Present other){
        if (this.value > other.value){ return 1;
        } else if (this.value < other.value){ return -1;
        } else { return 0; }
    }
}
```

Geschenke Sortiert

```
public class Present implements Comparable<Present>{
    ...
    public int compareTo(Present o){...}
    public String toString(){
        return content + ":" + value;
    }
}
...
```

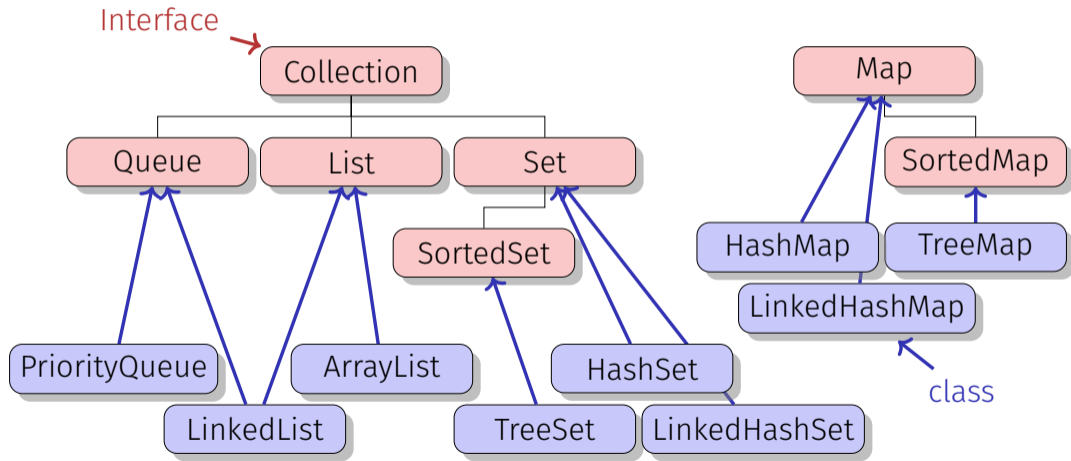
```
SortedList<Present> list = new SortedList<Present>();
list.insert(new Present("Buch",17));
list.insert(new Present("Juwelen",1000));
list.insert(new Present("Socken",12));
list.output(); // Socken:12 -> Buch:17 -> Juwelen:1000 -> NIL
```

Interfaces und Wrapperklassen

Die Wrapperklassen **Integer** und **Double** implementieren das Interface **Comparable**.

Klassen können in Java nur von einer Klasse erben (eine Klasse erweitern), aber Klassen können mehrere Interfaces implementieren.

Java Collections / Maps



Interface `Collection<E>` (Ausschnitt)

`boolean add(E e)`: Fügt `e` zur Collection hinzu, gibt zurück, ob die Collection geändert wurde.

`boolean contains(Object o)`: Gibt zurück, ob `o` in der Collection enthalten ist.

`boolean remove(Object o)`: Entfernt maximal eine Instanz des Objekts `o` von der Collection. Gibt zurück, ob `o` enthalten war.

`boolean isEmpty()`: Gibt zurück, ob die Collection leer ist

`int size()`: Gibt die Anzahl Elemente dieser Collection zurück.

`Iterator<E> iterator()`: Gibt einen Iterator zurück, mit dem die Elemente der Collection durchlaufen werden können

Vollständige Liste:

<https://docs.oracle.com/javase/10/docs/api/java/util/Collection.html>

Warum so viele Collections?

Collection definiert das **gemeinsame Interface** verschiedener möglicher Implementationen.

Verschiedene **Anwendungen / Algorithmen benötigen verschiedene Operationen**, möglicherweise zusätzlich zum Interface der Collection: Wahlfreier Zugriff, Hinzufügen am Anfang / am Ende, etc.

Eine Undo-Funktion im Texteditor ist mit Operationen push und pop implementiert. Eine Matrixmultiplikation benötigt wahlfreien Zugriff.

Warum so viele Collections?

Collection definiert das **gemeinsame Interface** verschiedener möglicher Implementationen.

Verschiedene Datenstrukturen (Arrays, Verkettete Listen, Bäume, etc.) unterscheiden sich in Ihrer **Eignung für unterschiedliche Operationen**.

Verkettete Listen sind sehr gut geeignet für Einfügen und Löschen, aber ziemlich ungeeignet für wahlfreien Zugriff (also Zugriff per Index). Bei Array-basierten Datenstrukturen ist es eher umgekehrt.

Iterator<E>

Das Interface **Iterator<E>** stellt Methoden zum Durchlaufen aller Elemente einer Collection zur Verfügung. Jede Collection bietet einen Iterator an.

boolean hasNext(): Gibt zurück, ob noch weitere Elemente auf diesem Iterator bereitstehen.

E next(): Gibt das nächste Element der Iteration zurück.

void remove(): Entfernt das zuletzt zurückgegebene Element von der Collection (muss nicht implementiert sein)

Beispiel Iterator

```
Collection<String> list = new ArrayList<String>();  
list.add("Hello");  
list.add("at");  
list.add("ETH");  
for (Iterator<String> it = list.iterator(); it.hasNext();) {  
    String s = it.next(); // Iterator fährt weiter  
    Out.print(s);  
}
```

Beispiel Iterator

```
Collection<String> list = new ArrayList<String>();  
list.add("Hello");  
list.add("at");  
list.add("ETH");  
for (Iterator<String> it = list.iterator(); it.hasNext();){  
    String s = it.next(); // Iterator fährt weiter  
    Out.print(s);  
}
```

Äquivalente **Kurzform** obiger Schleife:

```
for (String s: list){  
    Out.print(s);  
}
```

List

Zusätzlich zum Interface **Collection**:

- Wahlfreier Zugriff

```
E get (int index)
```

```
E set (int index, E element)
```

```
int indexOf(Object o)
```

- Einfügen und Löschen an Position

```
void add(int index, E element);
```

```
void remove(int index;
```

Implementationen: **ArrayList**, **LinkedList**

ArrayList versus LinkedList

Laufzeitmessungen für 10000 Operationen (auf [code]expert)

	ArrayList	LinkedList
Einfügen am Ende	469 μ s	1787 μ s
Einfügen am Anfang	37900 μ s	761 μ s
Iterieren	1840 μ s	2050 μ s
Wahlfreier Zugriff	426 μ s	110600 μ s
Einfügen in der Mitte	31ms	301ms
Enthält (erfolgreich)	38ms	141ms
Enthält (erfolglos)	228ms	1080ms
Entfernen am Ende	648 μ s	757 μ s
Entfernen am Anfang	58075 μ s	609 μ s

Interface Set<E>

Set (Menge): eine Collection, welche keine Duplikate enthält. Jedes Element kommt maximal einmal vor. Kein wahlfreier Zugriff

Implementationen:

- **HashSet<E>**: Datenstruktur, welche Einfügen und sehr effizientes Suchen (Methode **contains**) von Elementen unterstützt.
- **LinkedHashMap<E>**: Datenstruktur, welche Einfügen und effizientes Suchen unterstützt und welche beim Iterieren die **Einfügereihenfolge** respektiert.
- **TreeSet<E>**: Datenstruktur, welche Einfügen und effizientes Suchen unterstützt und welche die Daten **sortiert** speichert (Elemente müssen vergleichbar sein).

Set<E> und List<E>

Laufzeitmessungen für 10000 Operationen (auf [code]expert)

	List	HashSet	LinkedSet	TreeSet
Einfügen	350 μ s	958 μ s	930 μ s	1126 μ s
Iterieren	360 μ s	394 μ s	345 μ s	555 μ s
Enthält	49953 μ s	380 μ s	380 μ s	960 μ s
Enthält nicht	304289 μ s	179 μ s	203 μ s	400 μ s

PriorityQueue<E>

Eine Warteschlange, bei der immer das kleinste Element vorne (zum Extrahieren bereit) steht.

`void add(E e)` fügt das Element in die Prioritätswarteschlange ein

`E remove()` extrahiert das erste Element der Prioritätswarteschlange

	PriorityQ	TreeSet
Einfügen	423 μ s	1714 μ s
Kleinstes Element extrahieren	2400 μ s	2000 μ s

Einen Datensatz suchen

Beispiel: wir speichern alle Studenten dieser Vorlesung in einer Datenstruktur.

```
class Student {  
    String name;  
    String id;  
}
```

Wir wollen möglichst schnell einen Studenten nach Legi-Nr finden. Wir wollen die Studenten nach Einfügedatum ausgeben können.

Welche Datenstruktur? **LinkedHashSet<Student>**?

Problem

Welche Datenstruktur? `LinkedHashSet<Student>`?

Das Problem: das Set weiss nicht, nach welchem Kriterium es suchen muss und kann eigentlich auch nur `contains`. Aber selbst das schlägt fehl:

```
HashSet<Student> set = new HashSet<Student>();  
Student a = new Student("bobo", "123-456-789");  
Student b = new Student("bobo", "123-456-789");  
set.add(a);  
Out.println(set.contains(a)); // true  
Out.println(set.contains(b)); // false: a != b.
```

Assoziative Datenstruktur

Assoziative Datenstrukturen speichern Paare: Schlüssel (Suchkriterium) / Wert (Daten)

Schlüssel-Wert Paare

Key

Value

123-456-789 Student name = bobo, id = 123-456-789

007-420-312 Student name = pipi, id = 007-420-312, ...

...

Map<K, V>: Tabelle, welche effizient nach Schlüssel durchsucht werden kann.

List versus Maps

List / Array	Map
0 → obj1	"18-101-008" → obj1
1 → obj2	"18-389-221" → obj2
2 → obj3	"18-761-891" → obj3
3 → obj4	"17-234-365" → obj4
4 → obj5	"18-120-861" → obj5
...

Interface Map<K, V> (Ausschnitt)

V put(K key, V value) assoziiert den Wert **value** mit dem Schlüssel **key** in dieser Map.

V get(Object key) gibt den zu **key** assoziierten Wert zurück (null sonst).

V remove(Object key) entfernt das key-value pair, wenn der Schlüssel **key** vorhanden ist.

Collection<V> values() gibt die Werte der Map als Collection zurück

Set<K> keySet() gibt die Schlüssel der Map als Set zurück

Beispiel

```
HashMap<String,Integer> mountains = new HashMap<String,Integer>();
mountains.put("Matterhorn",4478);
mountains.put("Jungfrau",4158);
...
Out.print("enter mountain name: "); // enter mountain name:
String name = In.readLine(); // Eiger

Integer height = mountains.get(name);
if (height != null){
    Out.println(name + ": " + height + "m"); // Eiger: 1800m
} else {
    Out.println("?");
}
```

Implementationen von Map<K, V>

HashMap<K, V> Assoziativer Container von key-value Paaren.
Keine Reihenfolgengarantien. Null key und null value erlaubt.

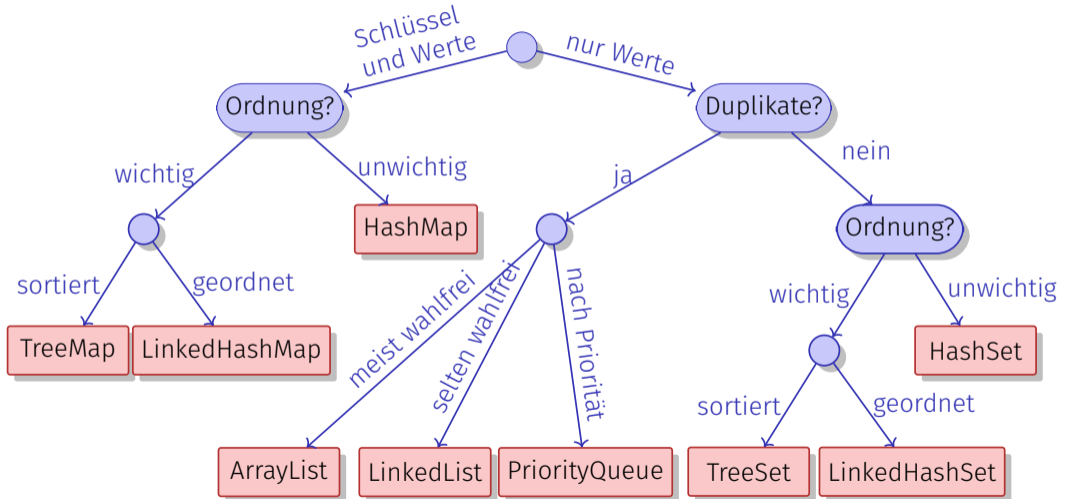
LinkedHashMap<K, V> Assoziativer Container mit
Reihenfolgengarantie: beim Iterieren wird die
Einfügereihenfolge erhalten.

TreeMap<K, V> Assoziativer Container mit
Reihenfolgengarantie: die Map ist sortiert nach der
natürlichen Ordnung der Schlüssel.

Übersicht

Implementation	Interface	Ordnung	Duplikate
ArrayList	List	Index	ja
LinkedList	List , Queue	Index	ja
PriorityQueue	Queue	Priorität	ja
HashSet	Set	keine	nein
LinkedHashSet	Set	Einfügereihenfolge	nein
TreeSet	Set	sortiert	nein
HashMap	Map	keine	nein
LinkedHashMap	Map	Einfügereihenfolge	nein
TreeMap	Map	sortiert	nein

Entscheidungshilfe



Anwendungsbeispiel: Sensoren!



id = 69

typ = humid



id = 2

typ = temp



id = 2282

typ = temp



id = 124

typ = humid



id	Standort	typ	...
69	Turm	humid	...
2282	Keller	temp	...
124	Turm	temp	...
2	Kessel	humid	...
⋮	⋮	⋮	⋮

Eigenschaft: (viele) Sensoren senden (viele) Messwerte.

Sensoren!

Sensoren liefern Messwerte

id	Zeitstempel	Wert
2282	12:34:21.000	24.80
69	12:34:20.998	40.03
2282	12:34:23.040	24.17
2282	12:34:22.010	24.30
69	12:34:25.998	41.00
2282	12:34:24.000	24.01
124	12:34:24.000	40.88
⋮	⋮	⋮

Beachte die „falsche“ Reihenfolge der Daten (nicht nach Zeitstempel geordnet)

Sensoren!

Sensoren liefern Messwerte

id	Zeitstempel	Wert
2282	12:34:21.000	24.80
69	12:34:20.998	40.03
2282	12:34:23.040	24.17
2282	12:34:22.010	24.30
69	12:34:25.998	41.00
2282	12:34:24.000	24.01
124	12:34:24.000	40.88
⋮	⋮	⋮

Beachte die „falsche“ Reihenfolge der Daten (nicht nach Zeitstempel geordnet)

```
class Sensor{
    int id;
    String loc;
    int type; // 0 (temperature)
              // or 1 (humidity)
    ...
}

class Measurement{
    int id;
    int timestamp;
    double value;
}
```

Sensoren!

Aufgabe: wir wollen alle gemessenen **Temperaturen, sortiert nach Zeitstempel, mit Ortsangabe** speichern und ausgeben.

Gewünschte Ausgabe:

Zeitstempel	Ort	Temperatur
12:34:21.000	Keller	24.80
12:34:22.010	Keller	24.30
12:34:23.040	Keller	24.17
12:34:24.000	Turm	40.88
12:34:24.000	Keller	24.01
⋮	⋮	⋮

Sensoren!

Welche Datenstruktur verwenden wir für die Tabelle der **Messdaten**?

Sensoren!

Welche Datenstruktur verwenden wir für die Tabelle der **Messdaten**?

`TreeSet<Measurement>` mit folgender Vergleichsmethode

```
class Measurement implements Comparable<Measurement>{
    int timestamp;
    ...
    public int compareTo(Measurement other){
        return new Integer(timestamp).compareTo(other.timestamp);
    }
}
```

denn damit können wir die Messdaten schnell einfügen und nach Zeitdatum sortiert extrahieren

(alternativ `PriorityQueue<Measurement>`)

Sensoren!

Welche Datenstruktur verwenden wir für die **Tabelle der Sensoren**?

Sensoren!

Welche Datenstruktur verwenden wir für die **Tabelle der Sensoren**?

`HashMap<Integer, Sensor>` (map: id \rightarrow Sensor)
denn wir benötigen schnelles Nachschlagen nach sensor id.

Sensoren!

Welche Datenstruktur verwenden wir für die Speicherung der **Tabelle** (**Zeitstempel / Ort / Temperatur**) ?

Sensoren!

Welche Datenstruktur verwenden wir für die Speicherung der **Tabelle (Zeitstempel / Ort / Temperatur)** ?

`ArrayList<Temperature>` mit

```
class Temperature {  
    Time time;  
    String location;  
    double value;  
    ...  
}
```

denn das ist die einfachste Datenstruktur, mit welcher wir die Daten einfach iterieren können.

(alternativ `LinkedList<Temperature>`)