17. Java Collections

Generic Types, Iterators, Java Collections, Iterators

Organizing Data

- Data Structures that we know
  - Arrays – Fixed-size sequences
  - Strings – Sequences of characters
  - Linked Lists (up to now: self-made for a fixed element type)

Today:

- General Collection Concept of the Java API
  - ArrayList on generic element types – more dynamic than arrays
  - LinkedList, Sets, Queues

General Map Concept of the Java API

6 "API = Application Programming Interface"

Generic List in Java: java.util.List

- import java.util.ArrayList;
- import java.util.List;
- ...
- // List of 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

Type Parameters ("Parameteric Polymorphism")

In Java you can parameterize a class with a type

// ListNode with generic value type T
class ListNode <T>{
  T value;
  ListNode<T> next;

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

placeholder T
concrete type (string) replaces T in the ListNode used.

Use:

ListNode<String> n = new ListNode<String>("ETH", null);
**Example: Generic Stack**

```java
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(){...}
}
```

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

**Stack of Integers**

- Java generics can only operate on objects
- Fundamental types `int, float` (etc.) are no objects
- Java offers wrapper classes for fundamental types, e.g. type `Integer`
- Java provides *autoboxing* and automatically wraps a fundamental type into a wrapper class, where necessary.

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

**Sorted List?**

```java
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;
        }
        ...
    }
}
```

```java
Sorted List?
public class SortedList<T>
```

```java
Sorted List!
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>` makes sure that method `T.compareTo` exists.
Interfaces

An interface defines functionality of a potential implementation by a class.

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

Any class `T` that implements `Comparable<T>` is required to implement all methods of `Comparable<T>`.

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

Comparable Gifts

```java
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; }
    }
}
```

Gifts Sorted

```java
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 and Wrapper Classes

The wrapper classes `Integer` and `Double` implement the interface `Comparable`.

In Java, classes can only inherit from (extend a) single class, but they can implement several interfaces.
### Java Collections / Maps

- **Collection**
  - Queue
  - List
  - Set
    - SortedSet
    - HashSet
  - Map
    - SortedMap
    - HashMap
    - LinkedHashMap

### Interface Collection<E> (Excerpt)

- `boolean add(E e)` : Inserts e into the Collection, returns if the collection has changed.
- `boolean contains(Object o)` : returns, if o is contained in the collection.
- `boolean remove(Object o)` : Removes a single instance of the objects o from the collection. Returns if o was present.
- `boolean isEmpty()` : returns if the collection is empty
- `int size()` : Returns the number of elements stored in the collection.
- `Iterator<E> iterator()` : Returns an iterator that can be used to iterate over the elements of the collection

### Why so many Collections?

Collection defines the *common interface* of different possible implementations.

Different applications / algorithms require different operations, potentially in addition to those defined in interface of the collection: random access, insert at the beginning / the end, etc.

**Beispiel**

An undo-function in a texteditor is implemented using operations push and pop. A matrix-multiplication requires random access.

Collection defines the *common interface* of different possible implementations.

**Different data structures** (arrays, linked lists, trees, etc.) differ in their *suitability for different operations*.

**Beispiel**

Linked Lists are very well suited for insertion and deletion but inappropriate for random access (i.e. access via index). For Array-based data structures, rather the reverse is true.
The interface `Iterator<E>` provides methods for traversing all elements of a collection. Every collection offers an iterator.

- `boolean hasNext()`: Returns if there are more elements to iterate via this iterator.
- `E next()`: Returns the next element available for iteration
- `void remove()`: Returns the last element returned by this iterator from the collection.

```java
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 proceeds
    Out.print(s);
```

Equivalent short-form of the for-loop above:

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

In addition to interface `Collection`:

- random access
  - `E get(int index)`
  - `E set(int index, E element)`
  - `int indexOf(Object o)`
- insertion and deletion at position
  - `void add(int index, E element);`
  - `void remove(int index);`

Implementationen: `ArrayList`, `LinkedList`

---

`ArrayList` versus `LinkedList`

run time measurements for 10000 operations (on [code] expert)

<table>
<thead>
<tr>
<th></th>
<th>ArrayList</th>
<th>LinkedList</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>add()</code></td>
<td>469 µs</td>
<td>1787 µs</td>
</tr>
<tr>
<td><code>remove()</code></td>
<td>37900 µs</td>
<td>761 µs</td>
</tr>
<tr>
<td><code>indexOf()</code></td>
<td>1840 µs</td>
<td>2050 µs</td>
</tr>
<tr>
<td><code>get()</code></td>
<td>426 µs</td>
<td>110600 µs</td>
</tr>
<tr>
<td><code>set()</code></td>
<td>31 ms</td>
<td>301 ms</td>
</tr>
<tr>
<td><code>indexOf()</code></td>
<td>38 ms</td>
<td>141 ms</td>
</tr>
<tr>
<td><code>remove()</code></td>
<td>228 ms</td>
<td>1080 ms</td>
</tr>
<tr>
<td><code>add()</code></td>
<td>648 µs</td>
<td>757 µs</td>
</tr>
<tr>
<td><code>remove()</code></td>
<td>58075 µs</td>
<td>609 µs</td>
</tr>
</tbody>
</table>
**Interface Set<E>**

Set: a collection that has no duplicates: each element can occur at once once. No random access.

Implementations:

- **HashSet<E>:** Data structure that supports insertion and very efficient search (contains) for elements.
- **LinkedHashMap<E>:** Data structure that supports insertion and efficient search and the respects the insertion order on iterators.
- **TreeSet<E>:** Data structure that supports insertion and efficient search and that stores data in a sorted way (elements must be comparable).

**Set<E> and List<E>**

run time measurements for 10000 operations (on [code]expert

<table>
<thead>
<tr>
<th></th>
<th>List</th>
<th>HashSet</th>
<th>LinkedSet</th>
<th>TreeSet</th>
</tr>
</thead>
<tbody>
<tr>
<td>Insert</td>
<td>350µs</td>
<td>958µs</td>
<td>930µs</td>
<td>1126µs</td>
</tr>
<tr>
<td>Iterate</td>
<td>360µs</td>
<td>394µs</td>
<td>345µs</td>
<td>555µs</td>
</tr>
<tr>
<td>Contains</td>
<td>49953µs</td>
<td>380µs</td>
<td>380µs</td>
<td>960µs</td>
</tr>
<tr>
<td>Contains not</td>
<td>304289µs</td>
<td>179µs</td>
<td>203µs</td>
<td>400µs</td>
</tr>
</tbody>
</table>

**PriorityQueue<E>**

A queue where always the smallest element is at the front (ready for extraction).

- `void add(E e)` inserts the element into the priority queue
- `E remove()` extracts the first element of the priority queue

<table>
<thead>
<tr>
<th></th>
<th>PriorityQ</th>
<th>TreeSet</th>
</tr>
</thead>
<tbody>
<tr>
<td>Insert</td>
<td>423µs</td>
<td>1714µs</td>
</tr>
<tr>
<td>Extract Smallest</td>
<td>2400µs</td>
<td>2000µs</td>
</tr>
</tbody>
</table>

**Look for a data set**

Example: we store all students of this class in a data structure.

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

We want to find students by legi number as quick as possible.

Welche Datenstruktur? `LinkedHashSet<Student>`?
Problem
Which data-structure? LinkedHashSet<Student>?
The problem: the Set does not know by which criterion it should search, and actually it can only do contains. and even this does not work well:

```java
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.
```

[Remark Aside]
You can change the data structure Student such that at least contains works (if you want ...)
```
class Student{
    String name;
    String id;
    public Student(String name, String id){
        this.name = name; this.id = id;
    }
    public int hashCode(){ // search criterion
        return id.hashCode();
    }
    public boolean equals(Object other){ // comparison criterion
        return id.equals(((Student)other).id);
    }
}
```

Associative Datastructure
Associative data structures store pairs: key (search criterion) / value (data)

<table>
<thead>
<tr>
<th>Key-Value Pairs</th>
</tr>
</thead>
<tbody>
<tr>
<td>Key</td>
</tr>
<tr>
<td>123-456-789</td>
</tr>
<tr>
<td>007-420-312</td>
</tr>
</tbody>
</table>

Map<K,V>: Table that can be searched for key in an efficient way.

List versus Maps

<table>
<thead>
<tr>
<th>List / Array</th>
<th>Map</th>
</tr>
</thead>
<tbody>
<tr>
<td>0 → obj1</td>
<td>&quot;18-101-008&quot; → obj1</td>
</tr>
<tr>
<td>1 → obj2</td>
<td>&quot;18-389-221&quot; → obj2</td>
</tr>
<tr>
<td>2 → obj3</td>
<td>&quot;18-761-891&quot; → obj3</td>
</tr>
<tr>
<td>3 → obj4</td>
<td>&quot;17-234-365&quot; → obj4</td>
</tr>
<tr>
<td>4 → obj5</td>
<td>&quot;18-120-861&quot; → obj5</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
</tr>
</tbody>
</table>
**Interface Map<K, V> (Excerpt)**

V put(K key, V value) associates the specified value with the specified key in this map.

V get(Object key) returns the value to which the specified key is mapped (null otherwise).

V remove(Object key) removes the mapping for a key from this map if present.

Collection<V> values() returns a Collection view of the values contained in this map.

Set<K> keySet() returns the Set view of the keys contained in this map.

**Implementations of Map<K, V>**

HashMap<K, V> Associative container storing key-value pairs. No order guarantees. Null key and null value allowed.

LinkedHashMap<K, V> Associative container with an order guarantee: the insertion order is retrieved.

TreeMap<K, V> Associative container with an order guarantee: the map is sorted according to the natural ordering of its keys.

**Overview**

<table>
<thead>
<tr>
<th>Implementation</th>
<th>Interface</th>
<th>Order</th>
<th>Duplicate</th>
</tr>
</thead>
<tbody>
<tr>
<td>ArrayList</td>
<td>List</td>
<td>Index</td>
<td>yes</td>
</tr>
<tr>
<td>LinkedList</td>
<td>List, Queue</td>
<td>Index</td>
<td>yes</td>
</tr>
<tr>
<td>PriorityQueue</td>
<td>Queue</td>
<td>Priority</td>
<td>yes</td>
</tr>
<tr>
<td>HashSet</td>
<td>Set</td>
<td>none</td>
<td>no</td>
</tr>
<tr>
<td>LinkedHashSet</td>
<td>Set</td>
<td>insertion</td>
<td>no</td>
</tr>
<tr>
<td>TreeSet</td>
<td>Set</td>
<td>natural order</td>
<td>no</td>
</tr>
<tr>
<td>HashMap</td>
<td>Map</td>
<td>none</td>
<td>no</td>
</tr>
<tr>
<td>LinkedHashMap</td>
<td>Map</td>
<td>insertion</td>
<td>no</td>
</tr>
<tr>
<td>TreeMap</td>
<td>Map</td>
<td>natural order</td>
<td>no</td>
</tr>
</tbody>
</table>

**Beispiel**

```java
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("?");
}
```
**Decision**

- **TreeMap**
  - sorted
  - order important

- **HashMap**
  - not important

- **LinkedHashMap**
  - order important

- **Ordered**
  - ArrayList
  - random access

- **PriorityQueue**
  - order important

- **Sorted**
  - TreeSet
  - LinkedHashMap

- **HashSet**
  - not important

- **Not Sorted**
  - LinkedList
  - no random access

**Application Example: Sensors!**

(Many) sensors deliver (a lot of) measurements

<table>
<thead>
<tr>
<th>id</th>
<th>Standort</th>
<th>typ</th>
<th>...</th>
</tr>
</thead>
<tbody>
<tr>
<td>69</td>
<td>Turm</td>
<td>humid</td>
<td>...</td>
</tr>
<tr>
<td>2282</td>
<td>Keller</td>
<td>temp</td>
<td>...</td>
</tr>
<tr>
<td>124</td>
<td>Turm</td>
<td>temp</td>
<td>...</td>
</tr>
<tr>
<td>2</td>
<td>Kessel</td>
<td>humid</td>
<td>...</td>
</tr>
</tbody>
</table>

**Task:** we want to output the temperatures provided by the sensors (sorted by time stamp) with the sensor locations.

Which data structure do we use for the *table of sensors*?

**HashMap<Integer, Sensor>* (map: id → Sensor)

because we require a quick lookup for sensor by id.

**Sensors!**

Sensors deliver measurements

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

Note the "wrong" order of the data (not ordered by time stamp)
Which data structures do we use for the table of measurements? 

**PriorityQueue<Measurement>** with the following comparison method:

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

because with this data structure we efficiently insert and extract the measurements sorted by time.

**ArrayList<Temperature>** or **LinkedList<Temperature>**

Which data structure do we use for storing the table (timestamp / location / temperature)?

**ArrayList<Temperature>** with:

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

because this is the simplest data structure we know in order to iterate through the data.