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[6]
  - 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;                    placeholder T
  ListNode<T> next;

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

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

Use:

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

Example: Generic 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 of Integers

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

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


Sorted List?

```
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 '>'

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

```
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>` .

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


Comparable Gifts

```
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

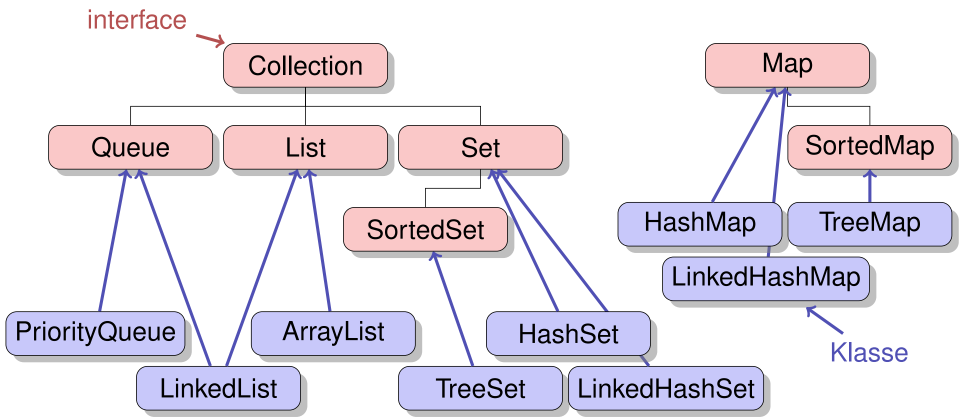
```
public class Present implements Comparable<Present>{
  ...
  public int compareTo(Present o){...}
  public String toString(){
    return content + ":" + value;
  }
}
...

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



`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 preent.

`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.

Why so many Collections?

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.

Iterator<E>

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.

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 proceedes
        Out.print(s);
}
```

Equivalent short-form of the for-loop above:

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


List

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)

ArrayList	LinkedList
469μs	1787μs
37900μs	**761μs**
1840μs	2050μs
426μs	110600μs
31ms	301ms
38ms	141ms
228ms	1080ms
648μs	757μs
58075μs	**609μs**

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)

	List	HashSet	LinkedSet	TreeSet
Insert	$350\mu s$	$958\mu s$	$930\mu s$	$1126\mu s$
Iterate	$360\mu s$	$394\mu s$	$345\mu s$	$555\mu s$
Contains	$49953\mu s$	**$380\mu s$**	**$380\mu s$**	**$960\mu s$**
Contains not	$304289\mu s$	**$179\mu s$**	**$203\mu s$**	**$400\mu s$**

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

	PriorityQ	TreeSet
Insert	**$423\mu s$**	$1714\mu s$
Extract Smallest	$2400\mu s$	$2000\mu s$

Look for a data set

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

```
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:

```
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)

Key-Value Pairs

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>`: Table that can be searched for key in an efficient way.

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>` (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 viewof the keys contained in this map

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("?");
}
```


Implementations of `Map<K,V>`

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

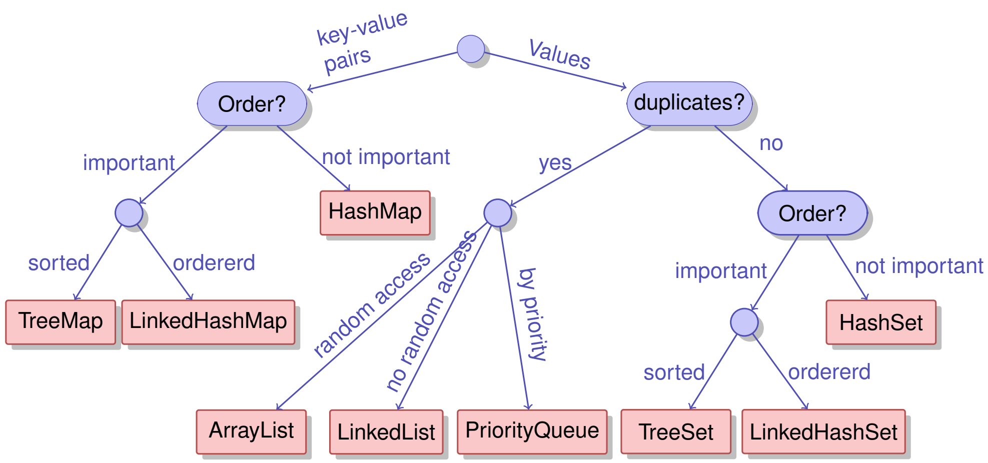
`LinkedHashMap<K,V>`Associative container with an order guarantee: the insertion order is retrieved.

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

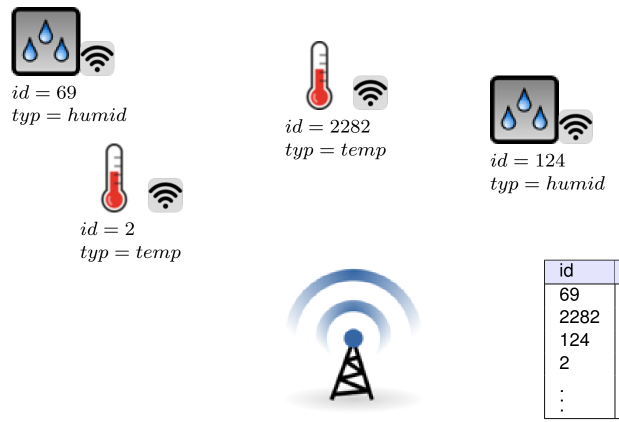
Overview

Implementation	Interface	Order	Duplicate
ArrayList	List	Index	yes
LinkedList	List , Queue	Index	yes
PriorityQueue	Queue	Priority	yes
HashSet	Set	none	no
LinkedHashSet	Set	insertion	no
TreeSet	Set	natural order	no
HashMap	Map	none	no
LinkedHashMap	Map	insertion	no
TreeMap	Map	natural order	no

Decision



Application Example: Sensors!



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

(Many) sensors deliver (a lot of) measurements

Sensors!

Sensors deliver measurements

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

Note the "wrong" order of the data (not ordered by time stamp)

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

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


Sensors!

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!

Which data structures do we use for the table of measurements?

`PriorityQueue<Measurement>`[7] with the following comparison method

```
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

[7] or `TreeSet<Measurement>`

Sensors!

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

`ArrayList<Temperature>`[8] mit

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

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

[8] or `LinkedList<Temperature>`