Collections

Datamining und Sequenzanalyse

Kai Dührkop, Markus Fleischauer

Arrays

Arrays

Arrays erzeugen

```
int[] arr1 = new int[3]

int[] arr2 = {1,2,3}
```

Falls möglich primitive Datentypen verwenden

```
int[] arr1 anstatt Integer[] arr1
```

java.util.Arrays

```
sort, fill, copyOf, binarySearch, toString
```

System.arraycopy

Generics

Ein Stack für Zahlen

```
public class IntegerStack {

    int[] stack;
    int pos;

    public IntegerStack(){
        stack = new int[10];
        pos = -1;
    }

    public int pop(){
        if(pos<0) throw new RuntimeException("Stack is empty");
        int backVal = stack[pos];
        pos--;
        return backVal;
    }

    public void push(int val){
        if(pos<9){
            pos++;
            stack[pos] = val;
        }else throw new RuntimeException("Stack is full");
    }
}
```

Ein universeller Stack

```
public class ObjectStack {

    Object[] stack;
    int pos;

    public ObjectStack(){
        stack = new Object[10];
        pos = -1;
    }

    public Object pop(){
        if(pos<0) throw new RuntimeException("Stack is empty");
        Object backVal = stack[pos];
        pos--;
        return backVal;
    }

    public void push(Object val){
        if(pos<9){
            pos++;
            stack[pos] = val;
        }else throw new RuntimeException("Stack is full");
    }
}
```

Zufriedenstellend?

```
public static void main(String[] args){

    ObjectStack stack = new ObjectStack();

    stack.push(1);
    stack.push(2);
    stack.push("3");
    stack.push(4);

    Integer val4 = (Integer) stack.pop(); System.out.println("4: "+val4);
    Integer val3 = (Integer) stack.pop(); System.out.println("3: "+val3);
    Integer val2 = (Integer) stack.pop(); System.out.println("2: "+val2);
    Integer val1 = (Integer) stack.pop(); System.out.println("1: "+val1);

}
```

```
4: 4
Exception in thread "main" java.lang.ClassCastException:
java.lang.String cannot be cast to java.lang.Integer
    at ObjectStack.main(ObjectStack.java:35)
```

Generics

```
public class GenericStack <T> {

    Object[] stack;
    int pos;

    public GenericStack(){
        stack = new Object[10];
        pos = -1;
    }

    public T pop(){
        if(pos<0) throw new RuntimeException("Stack is empty");
        T backVal = (T) stack[pos];
        pos--;
        return backVal;
    }

    public void push(T val){
        if(pos<9){
            pos++;
            stack[pos] = val;
        }else throw new RuntimeException("Stack is full");
    }
}
```

Generics

```
public static void main(String[] args){

    GenericStack<Integer> genStack = new GenericStack<Integer>();
    genStack.push(1);
    genStack.push("2");
    genStack.push(3);
    genStack.push(4);

    Integer val1 = genStack.pop(); System.out.println(val1);
    Integer val2 = genStack.pop(); System.out.println(val2);
    Integer val3 = genStack.pop(); System.out.println(val3);
    Integer val4 = genStack.pop(); System.out.println(val4);

}
```

```
The method push(Integer) in the type GenericStack<Integer>
is not applicable for the arguments (String)
```

Generics

- Typeinschränkungen

```
class NumberList <T extends Number>{
    public int doSomething(T number){ return number.intValue(); }
}
```

- Der Diamantoperator

```
GenericStack<String> stack = new GenericStack<>();
```

Generics

- Generische Schnittstellen

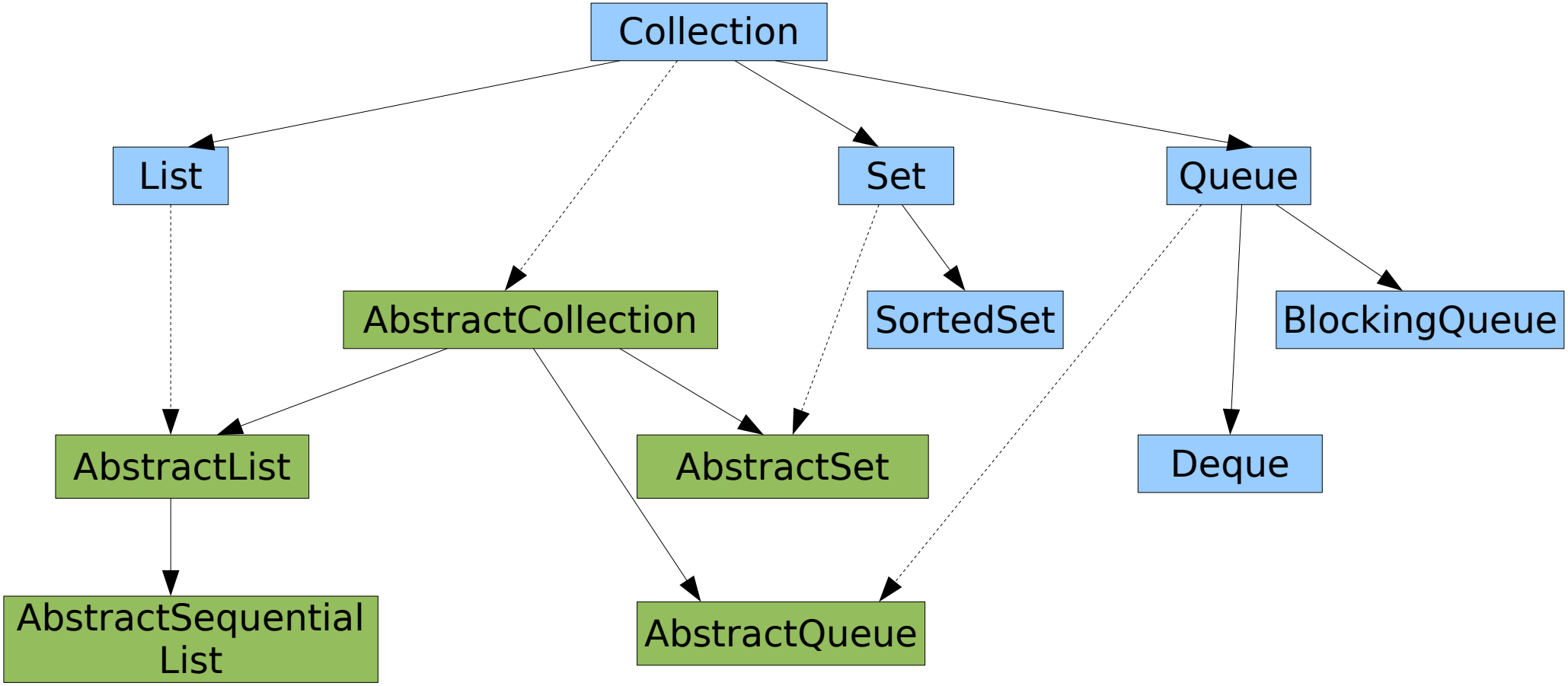
```
public class Class1 implements Comparable<Class1> {
    public int compareTo( Class1 o ) { ... };
}
```

- Generische Methoden

```
public class Tools {
    public static <T extends Number > int intVal ( T val ) {
        return number.intValue();
    }
}
```

Collections API

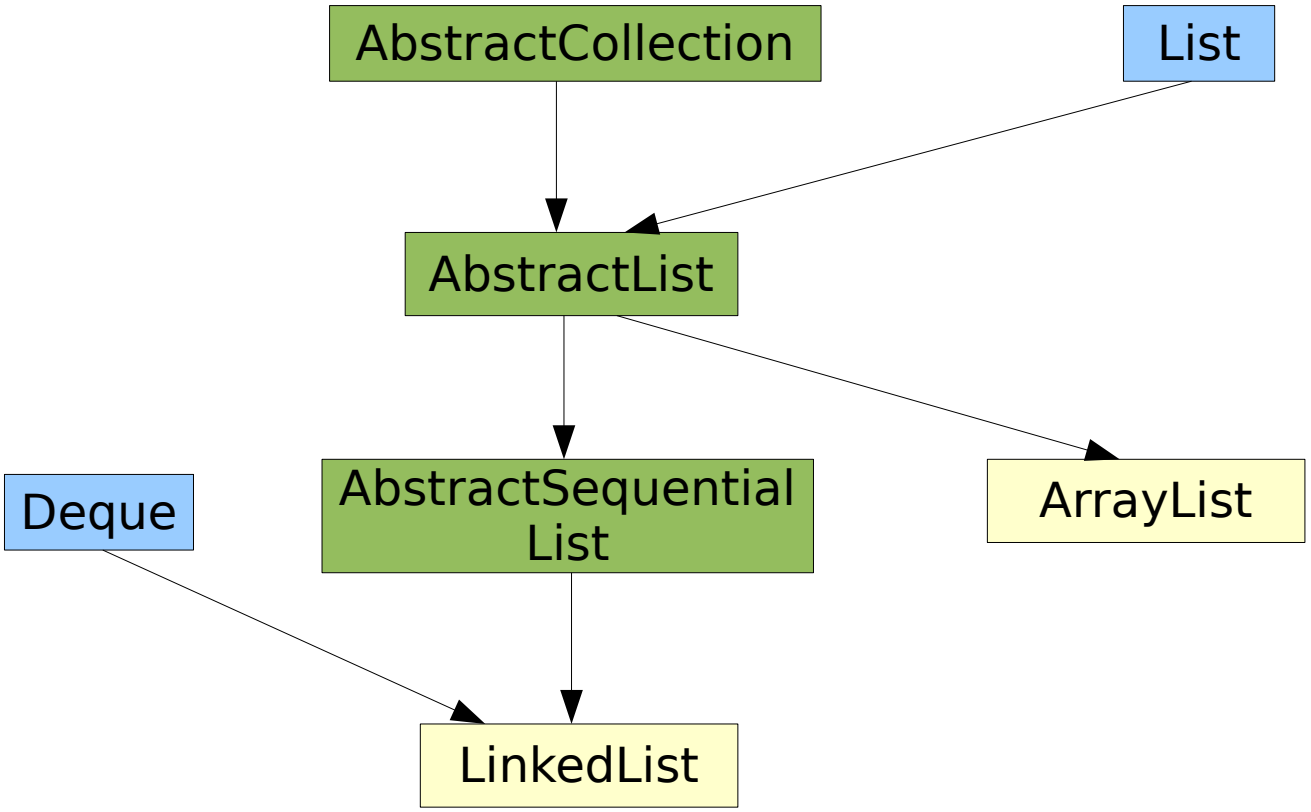
Struktur der Collections API



Was kann eine Collection?

- add / addAll
- contains / containsAll
- remove / removeAll
- size
- toArray
- isEmpty
- clear
- iterator

Listen


AbstractCollection
List
AbstractList
AbstractSequential List
Deque
ArrayList
LinkedList

Listen

Erweiterungen zu Collection

- `add (mit Index)`
- `get`
- `indexOf / lastIndexOf`
- `remove (mit Index)`
- `set`
- `subList`

Vertreter

- **ArrayList:** basiert auf Array; (in der Regel schneller)
- **LinkedList:** basiert auf verkettete Liste

Welche Liste soll ich verwenden?

schneller „random access“, Lösch- und Einfügeoperationen fast nur am Ende?

ArrayList

schneller sequentieller Zugriff, Lösch- und Einfügeoperationen in der Mitte, viele Zugriffe am Anfang/Ende?

LinkedList

Welche Liste soll ich verwenden?

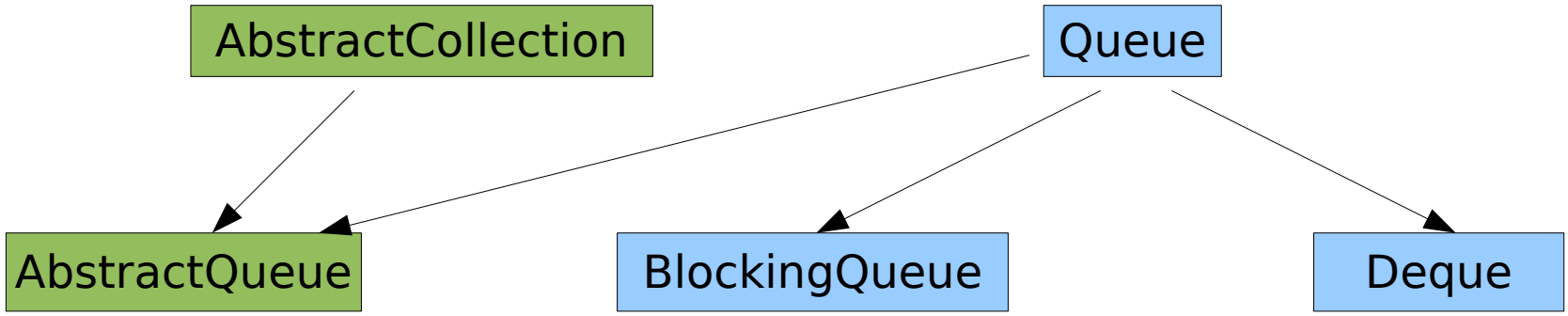
schneller „random access“, Lösch- und Einfügeoperationen fast nur am Ende?

ArrayList

schneller sequentieller Zugriff, Lösch- und Einfügeoperationen in der Mitte, viele Zugriffe am Anfang/Ende?

LinkedList

Queue



Methoden

	Einfügen	**Entnehmen**	**Löschen**
mit Exception	`add`	`element`	`remove`
Rückgabewert	`offer`	`peek`	`poll`

Vertreter

- **ConcurrentLinkedQueue:** Thread-sicher; verkettete Liste
- **PriorityQueue:** liefert das kleinste Element; Heap

BlockingQueue

Erweiterungen

- `put,take` → blockiert bis ein Element verfügbar ist
- `poll, peek, offer` mit Timeout

Vertreter

- **DelayQueue:** Eingefügte Elemente erst nach einer Zeit verfügbar
- **ArrayBlockingQueue:** max. Kapazität; Array
- **LinkedBlockingQueue:** ohne max. Kapazität; verkettete Liste
- **PriorityBlockingQueue:** liefert das kleinste Element
- **SyncronousQueue:** für den Austausch genau eines Elements

Deque

Methoden von Deque

	Mit Exception	Ohne Exception
Einfügen	addFirst / addLast	offerFirst / offerLast
Löschen	removeFirst / removeLast	pollFirst / pollLast
Entnehmen	getFirst / getLast	peekFirst / peekLast

Deque

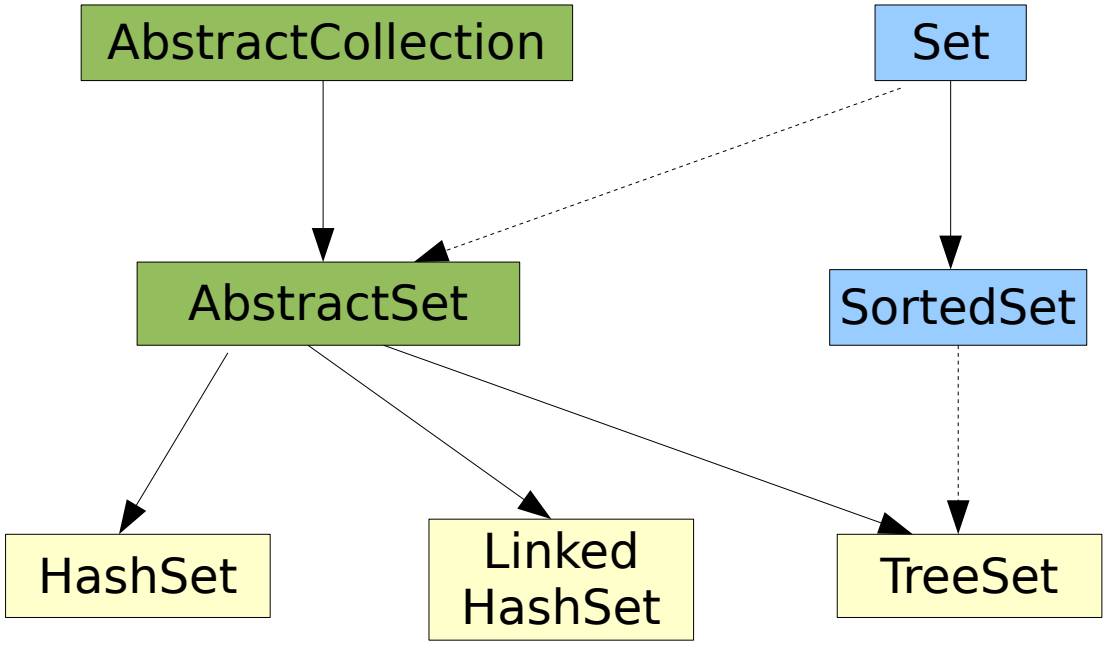
Deque als Stack

Methode Stack	Methode Deque
push	addFirst
pop	removeFirst
peek	peekFirst

Vertreter

- **ArrayDeque:** basiert auf einem Array
- **LinkedList:** …
- **LinkedBlockingDeque:** gleichzeitig BlockingQueue

Sets - Mengen


AbstractCollection
Set
AbstractSet
SortedSet
HashSet
Linked
HashSet
TreeSet

Sets

Vorhandene Sets

- **HashSet:** basiert auf Hashtable; schnell; Iteration ohne ohne feste Reihenfolge
  (contains, add, remove in O(1))
- **LinkedHashSet:** Hashtabelle + verkettete Liste; Iteration in Einfügereihenfolge
  (contains, add, remove in O(1) → mehr Overhead)
- **TreeSet:** sortiert; Iteration mit fester Reihenfolge;
  rot-schwarz Baum
  (contains, add, remove in O(log(n)))

TreeSet

Zusätzliche Methoden von TreeSet

- `first`
- `headSet`
- `comparator`
- `ceiling`
- `higher`
- `last`
- `subSet`
- `tailSet`
- `floor`
- `lower`

Erzeugen von TreeSets – Comparator

```
public class MyClass1 {

    private int val;

    public MyClass1(int val) {this.val = val;}

    public int getVal() { return val; }

    public boolean equals(Object o) { … }

}


TreeSet<MyClass1> tree = new TreeSet<>(new Comparator<MyClass1>() {

    @Override
    public int compare(MyClass1 o1, MyClass1 o2) {
        return o1.getVal() - o2.getVal();
    }

});
```

Erzeugen von TreeSets – Comparable

```
public class MyClass2 implements Comparable<MyClass2> {

    private int val;

    public MyClass2(int val) {this.val = val;}

    public int getVal() { return val; }

    public boolean equals(Object o) { … }

    @Override
    public int compareTo(MyClass2 o) {
        return this.getVal() - o.getVal();
    }

}
```

```
TreeSet<MyClass2> tree  = new TreeSet<MyClass2>();
```

Sets und equals

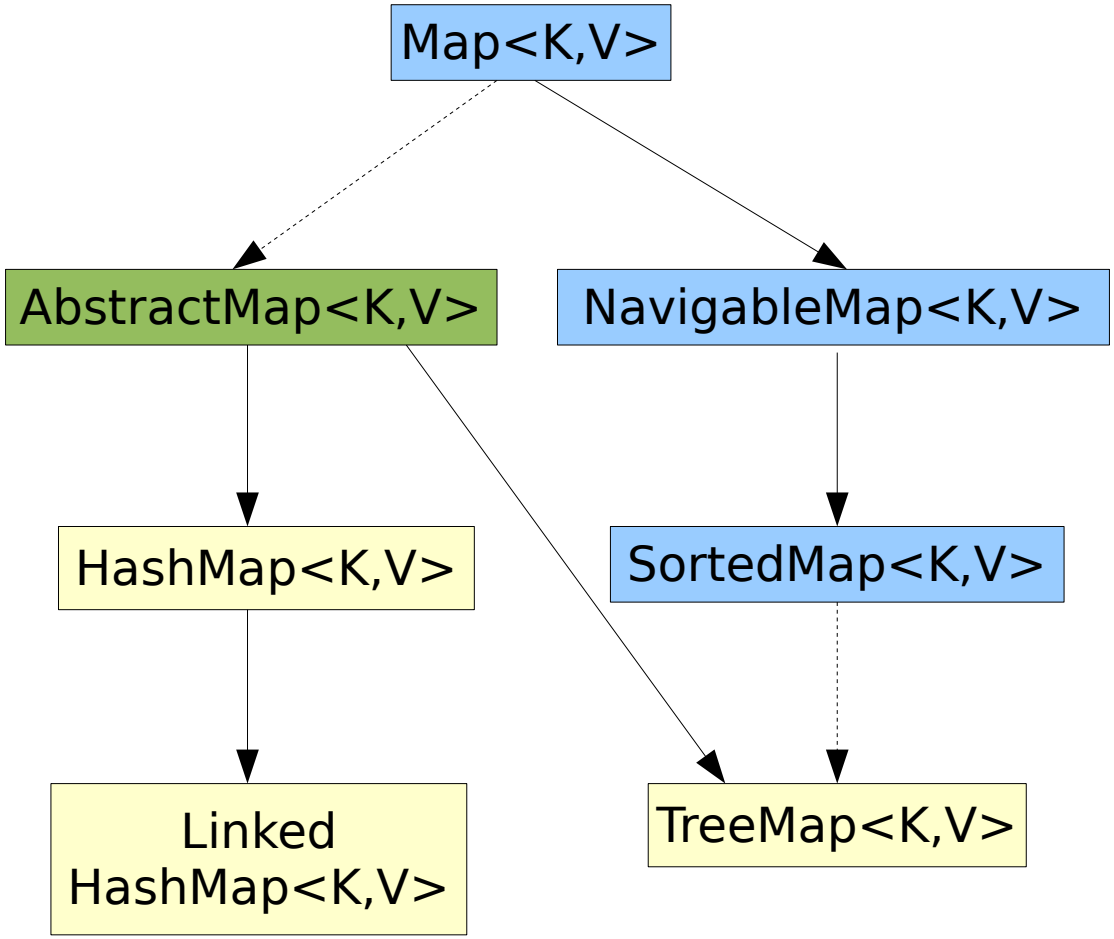
equals und TreeSets

equals = true ⟷ compare = 0 bzw. compareTo = 0

equals und HashSet / LinkedHashSet

equals = true ⟶ object1.hashCode() == object2.hashCode()
&
object1.equals(object2)

Maps – assoziativer Speicher



Maps

Methoden von Map<K,V>

- `put(K,V)`
- `get(K)`
- `containsKey(K)`
- `containsValue(V)`
- `remove(K)`
- `clear`
- `isEmpty`
- `size`
- `keySet / values`

Maps

Vorhandene Maps

- **HashMap:** schnelle Hash-Tabelle → Schlüssel ohne Reihenfolge

  ACHTUNG: Zusammenhang equals und hashCode

- **LinkedHashMap:** Hash-Tabelle + verkettete Liste → Schlüssel in Einfügereihenfolge

  ACHTUNG: Zusammenhang equals und hashCode

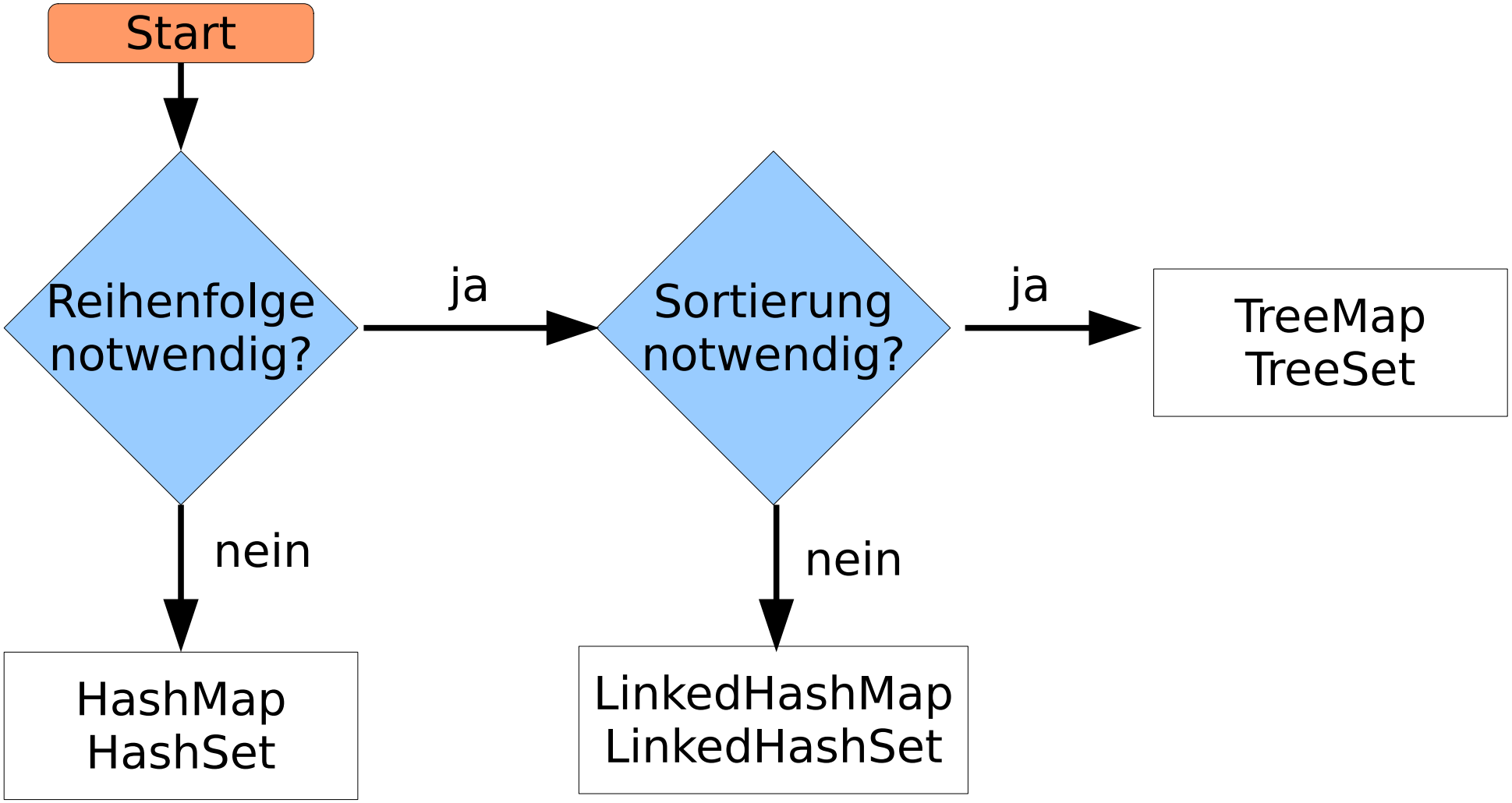
- **TreeMap:** Rot-Schwarz Baum → Schlüssel sind sortiert

Maps

Zusätzliche Methoden von TreeMap(K,V)

- `comparator`
- `firstKey()`
- `lastKey()`
- `ceilingKey(K)`
- `higherKey(K)`
- `headMap(K)`
- `tailMap(K)`
- `subMap(K,K)`
- `floorKey(K)`
- `lowerKey(K)`

Welche Map/Set soll ich nehmen?



java.util.Collections

Synchronisierte Datenstrukturen

syncronizedMap / syncronizedSet / syncronizedList / syncronizedCollection

Read Only Datenstrukturen

```
unmodifiableMap / unmodifiableSet /
unmodifiableList / unmodifiableCollection
```

Singletons (nur ein Element, unveränderbar)

```
singletonMap / singletonList / singleton
```

Dummys (leer, unveränderbar)

```
emptyMap / emptyList / emptySet
```

java.util.Collections

Operationen für Collections

- disjoint
- frequency
- min / max

Listenoperationen

- binarySearch
- replaceAll
- reverse
- rotate
- sort (Merge Sort)
- swap
- copy
- shuffle
- fill

Klassen die man meiden sollte

- Vector
- Stack
- HashTable