Hochschule für Angewandte Wissenschaften Hamburg
Hamburg University of Applied Sciences

Programmieren

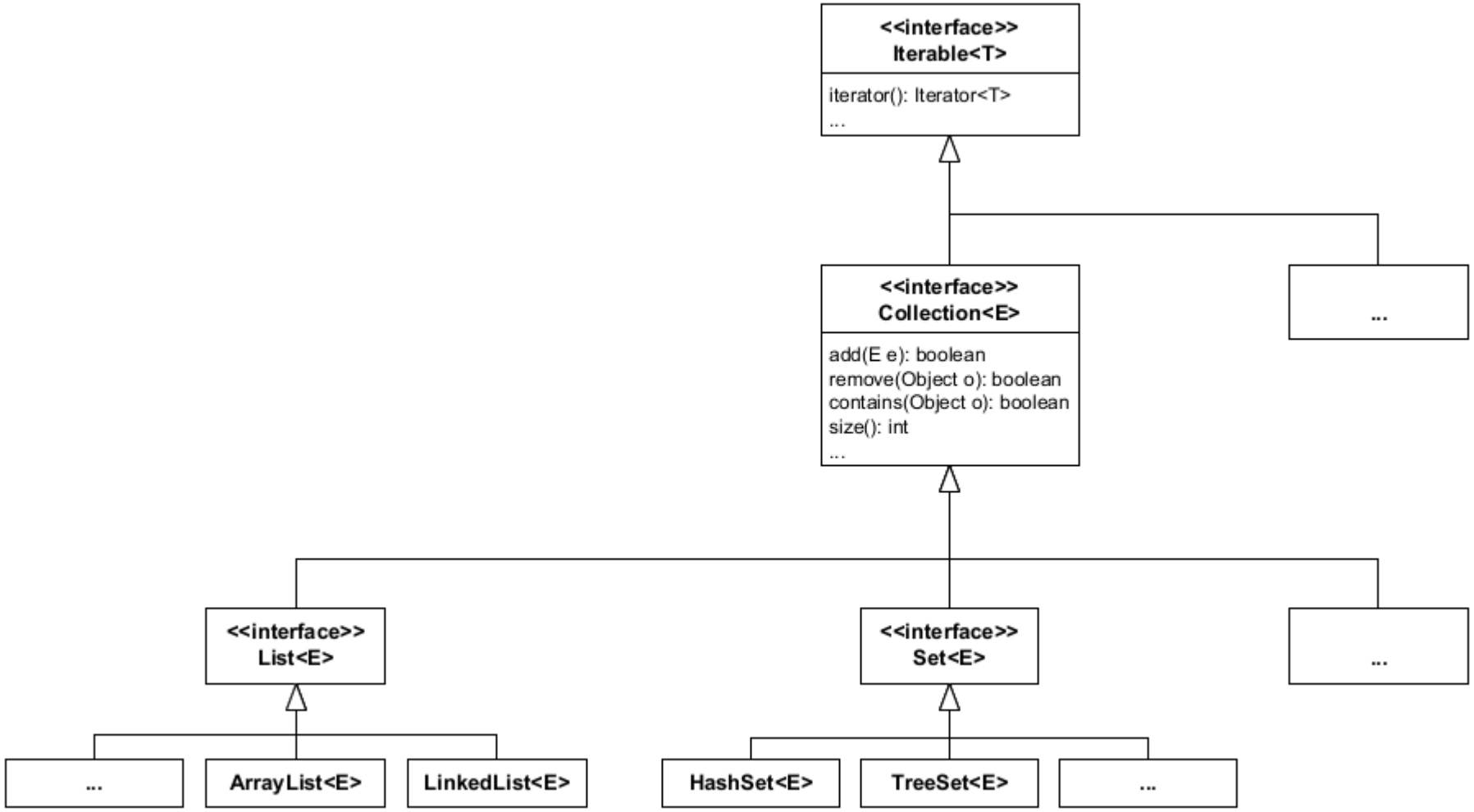
Prof. Dr. Larissa Putzar &
Thorben Ortmann

Collections

Gliederung

- Iterable & Iterator
- Collection
- List
  - ArrayList
  - LinkedList
- Set
  - HashSet
  - LinkedHashSet
  - TreeSet
  - Comparable & Comparator
  - equals
- Map
  - HashMap
- Stream

Vererbungshierarchie



Iterable & Iterator

Mit Hilfe eines *Iterators* kann man über ein *Iterable*, wie etwa eine *List*, iterieren. Das bedeutet, man kann sich jedes Element des *Iterables* nacheinander geben lassen. Der typische Code dafür sieht so aus:

```
Iterator<Integer> iterator = myIntegerList.iterator();
while (iterator.hasNext()) {
    Integer current = iterator.next();
    doSomething(current);
}
```

Iterable & Iterator

Seit Java 8 gibt es die for-each-loop und im *Iterable*-Interface die *forEach()*-Methode. Daher benutzt man den `while(iterator.hasNext()){…}` Code heute in der Regel nicht mehr.

- for-each-loop

```
for (Integer current: myIntegerList) {
    doSomething(current);
}
```

- *forEach()*-Methode

```
myIntegerList.forEach(current -> doSomething(current));
```

Collection

- Aus dem JavaDoc des *Collection*-Interfaces:

„ A collection represents a group of objects, known as its *elements*. Some collections allow duplicate elements and others do not. Some are ordered and others unordered. ”

- Das *Interface Collection<E>* bietet grundlegende Operationen für die Arbeit mit *Collections* an:

```
//Hinzufügen von Elementen
boolean add(E e) / boolean addAll(Collection c)

//Entfernen von Elementen
boolean remove(Object o) / boolean removeAll(Collection c)
```

Collection

- Das *Interface Collection<E>* bietet grundlegende Operationen für die Arbeit mit *Collections* an:

```
//enthält die Datenstruktur das Element o?
boolean contains(Object o)

//gibt die Anzahl der Elemente zurück
int size()

//gibt ein Array mit allen Elementen zurück
Object[] toArray()

//gibt einen Stream mit der Collection als Quelle zurück
Stream<E> stream()
```

List

Listen sind geordnete *Collections*, die Duplikate zulassen. Die Ordnung ist über den Index der Liste gegeben. Aus dem JavaDoc des *List*-Interfaces:

„ An ordered collection (also known as a sequence). The user of this interface has precise control over where in the list each element is inserted. The user can access elements by their integer index (position in the list), and search for elements in the list. Unlike sets, lists typically allow duplicate elements. ”

- Wichtige Listen-Implementierungen sind:
  - ArrayList
  - LinkedList

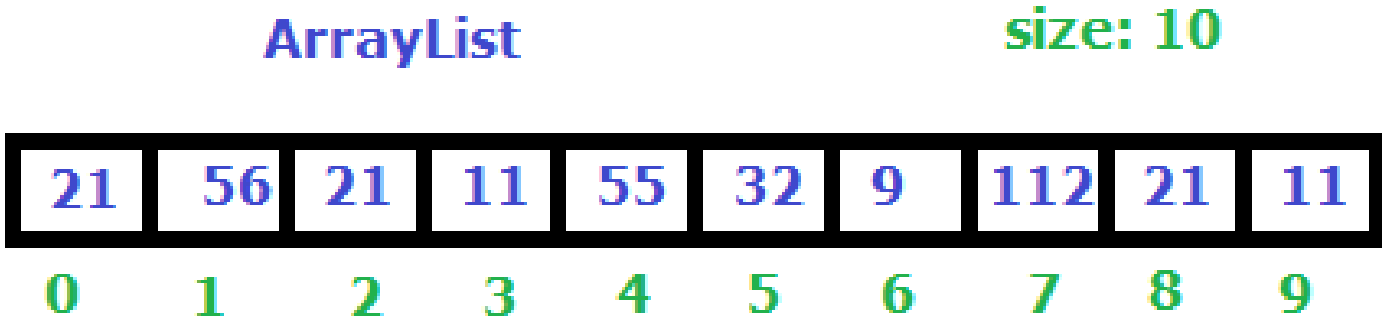
List - Operationen

- Das *Interface List<E>* bietet grundlegende Operationen für die Arbeit mit Listen an*:*

```
//gibt das Element an der Stelle index zurück
E get(int index)

//ersetzt das Element an der Stelle index und gibt das alte Element zurück
E set(int index, E element)

//fügt das Element an der Stelle index zur Liste hinzu
void add(int index, E element)

//entfernt das Element an der Stelle index aus der Liste
E remove(int index)
```

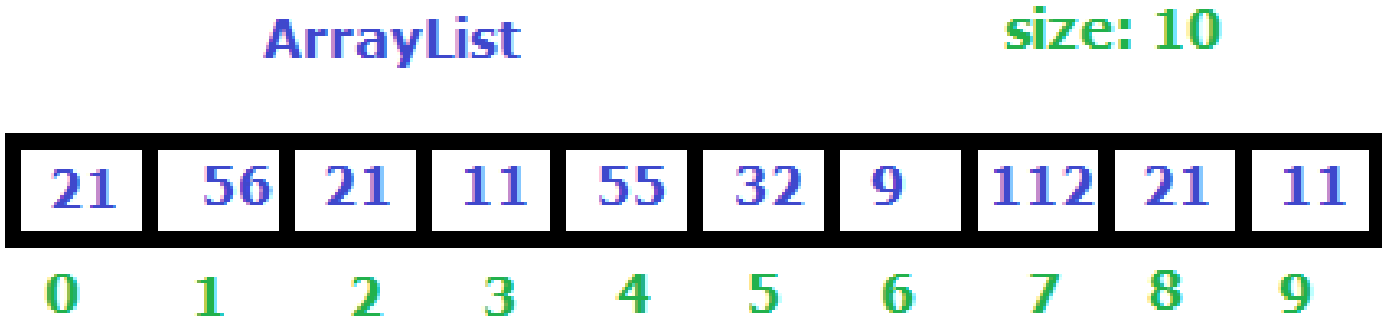
ArrayList

ArrayLists sind wie Arrays variabler Länge. *ArrayLists* arbeiten mit Arrays und bieten Methoden zur einfacheren Bedienung an. Man kann jeder Zeit ein Element hinzufügen und muss anfangs nicht wissen wie viele Elemente es geben wird.



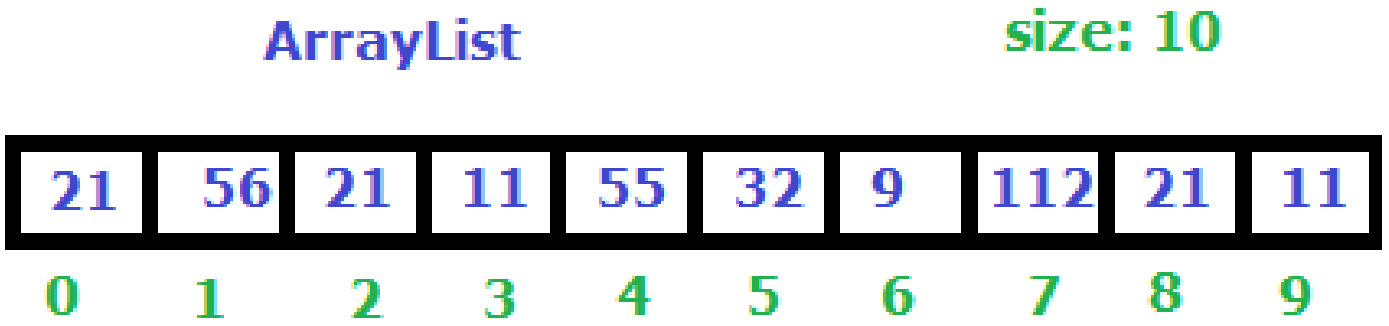
ArrayList

- Initial Size = 10
- Wird das 11 Element hinzugefügt, wird intern ein neues Array mit der verdoppelten Anzahl der Elemente erstellt und alle Daten werden kopiert.
- Man kann beim Erstellen der *ArrayList* auch eine initial Size mitgeben.



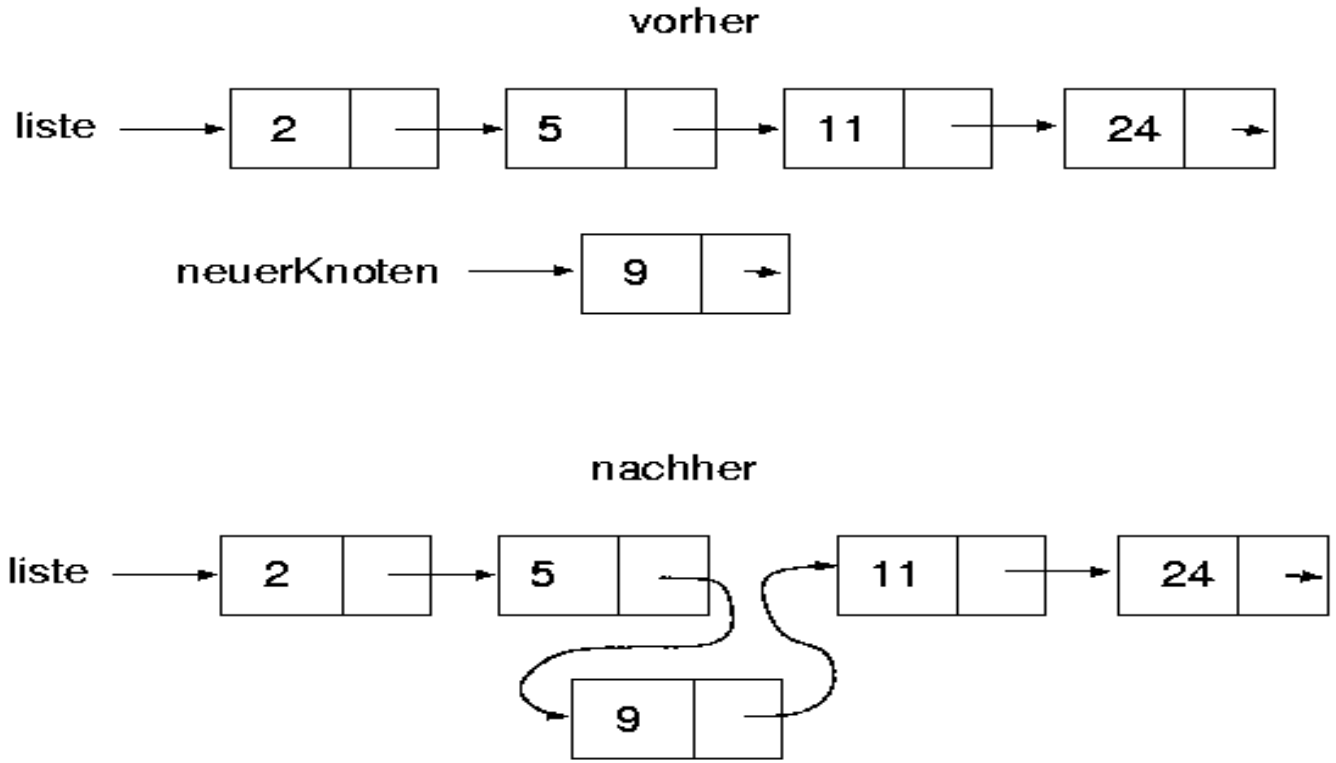
Array vs ArrayList

- Arrays sind performanter als *ArrayLists*. Allerdings man muss anfangs die Anzahl der benötigten Elemente kennen.
- *ArrayLists* sind bequemer zu bedienen und haben unterstützende Funktionen wie *contains*, *add*, *remove* etc.



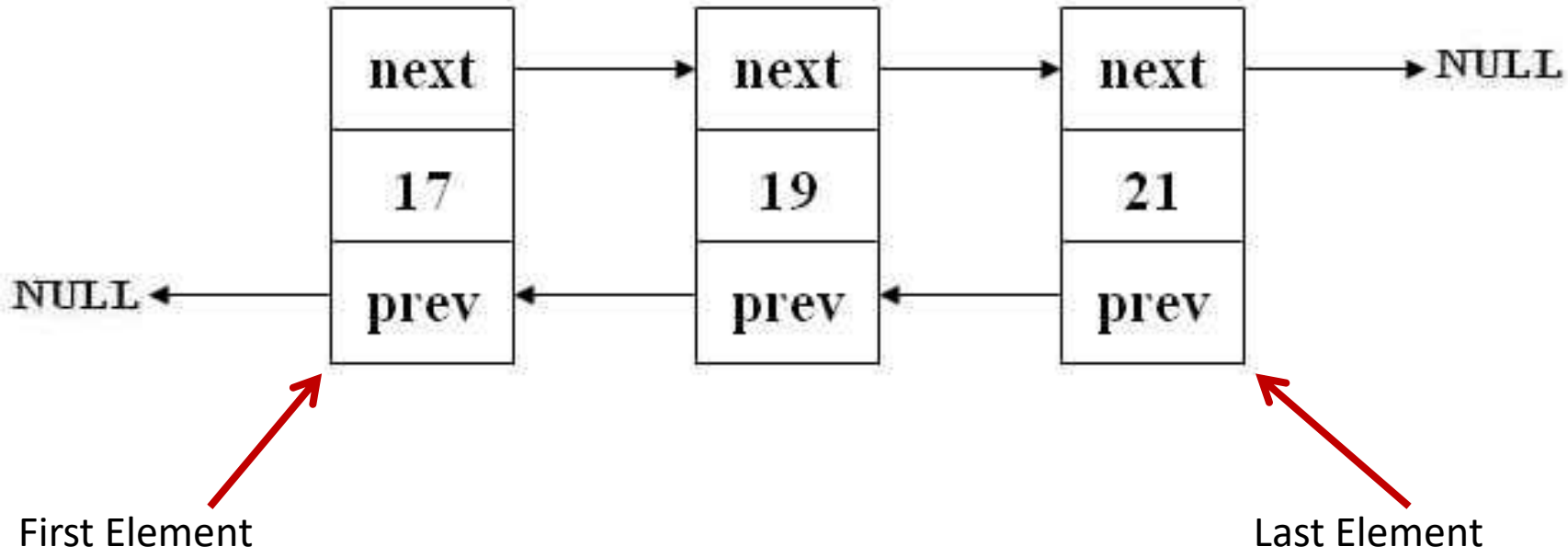
LinkedList

- Einfach verkette Liste, optimiert für schnelles Einfügen und Löschen.



LinkedList

- Java nutzt eine doppelt verkettete Liste:



ArrayList vs LinkedList

Methoden	**LinkedList**	**ArrayList**
add(E element)	*O(1)*	*O(1), worst case O(n)*
add(int index, E element)	*O(n), O(1) beim ersten Element*	*O(n)*
remove(int index)	*O(n)*	*O(n)*
remove(Object obj)	*O(n)*	*O(n)*
get(int index)	*O(n)*	*O(1)*
contains(Object obj)	*O(n)*	*O(n)*

Beispiel ArrayList

```
//erstelle eine Liste mit Strings
ArrayList<String> list = new ArrayList<String>();


//füge ein Element hinzu
list.add("Ein neues Element");


//Gib alle Elemente der Liste aus
for (int i = 0; i < list.size(); i++) {
    System.out.println(list.get(i));
}
```

- Auch möglich:

```
new ArrayList<>(List.of("Ein neues Element")).forEach(s -> System.out.println(s));
```

Set

Sets sind *Collections*, die keine Duplikate zulassen. Im Allgemeinen besitzen *Sets* keine Ordnung. Es gibt allerdings Sub-Interfaces und Implementierungen, die eine Ordnung besitzen. Aus dem JavaDoc des *Set*-Interfaces:

„ A collection that contains no duplicate elements. […] As implied by its name, this interface models the mathematical *set* abstraction. ”

- Wichtige Set-Implementierungen sind:
  - HashSet (keine Ordnung)
  - LinkedHashSet (Ordnung gemäß der Einfügereihenfolge)
  - TreeSet (natürliche Ordnung bzw. per *Comparator*)

Set - Operationen

Das *Interface Set<E>* bietet nur wenige Operationen an, die über das *Collection*-Interface hinausgehen. Die Signaturen der meisten Operationen sind identisch*:*

```
//Hinzufügen von Elementen
boolean add(E element) / boolean addAll(Collection c)


//Entfernen von Elementen
boolean remove(Object o) / boolean removeAll(Collection c)


//enthält die Datenstruktur das Element o?
boolean contains(Object o)


//gibt die Anzahl der Elemente zurück
int size()
```

Set - get

Im Gegensatz zum *List*-Interface besitzt das *Set*-Interface keine *get()*-Operation. Auf ein oder mehrere Elemente eines *Sets* kann man wie folgt zugreifen:

```
//Etwas mit dem ersten Element eines Sets tun
Iterator<String> iterator = stringSet.iterator();
if(it.hasNext()) doSomething(iterator.next());

---oder---

stringSet.stream().findFirst().ifPresent(first -> doSomething(first));


//Etwas mit allen Element eines Sets tun
for (String current : stringSet) {
    doSomething(current);

}

---oder---

stringSet.forEach(current -> doSomething(current));
```

HashSet und LinkedHashSet

- Aus dem JavaDoc von *HashSet:*
  „ This class implements the Set interface, backed by a hash table (actually a HashMap instance). It makes no guarantees as to the iteration order of the set; in particular, it does not guarantee that the order will remain constant over time. This class permits the null element. This class offers constant time performance for the basic operations (add, remove, contains and size)… ”

- Aus dem JavaDoc von *LinkedHashSet:*
  „ Hash table and linked list implementation of the Set interface, with predictable iteration order. This implementation differs from HashSet in that it maintains a doubly-linked list running through all of its entries. This linked list defines the iteration ordering, which is the order in which elements were inserted into the set (insertion-order). […] This class provides all of the optional Set operations, and permits null elements. Like HashSet, it provides constant-time performance for the basic operations (add, contains and remove)…”

TreeSet

- Aus dem JavaDoc von *TreeSet:*
  „ The elements are ordered using their natural ordering, or by a Comparator provided at set creation time, depending on which constructor is used. This implementation provides guaranteed log(n) time cost for the basic operations (add, remove and contains). ”

- Damit eine Ordnung möglich ist, müssen die Elemente des *TreeSets* entweder das Interface *Comparable* implementieren (*natural ordering*) oder bei der Erstellung wird dem *TreeSet*-Konstruktor ein *Comparator*-Objekt übergeben, das eine Ordnung der Elemente definiert.

Comparable

Comparable ist ein Interface zur Definition der Ordnung von Objekten der jeweiligen implementierenden Klasse. Es muss nur die Operation *compareTo()* implementiert werden. Sei eine Klasse *Project* mit einem *int*-Attribut *priority* und einem zugehörigen Getter gegeben. Dann könnte die *compareTo()*-Methode wie folgt aussehen:

```
// Das kleinste/erste Projekt der Ordnung, ist das Projekt mit der geringsten Priorität
@Override
public int compareTo(final Project otherProject) {
    if (priority < otherProject.priority) return -1;
    if (priority == otherProject.priority) return 0;
       return 1;
}

// Das kleinste/erste Projekt der Ordnung, ist das Projekt mit der höchsten Priorität
@Override
public int compareTo(final Project otherProject) {
    if (priority < otherProject.priority) return 1;
    if (priority == otherProject.priority) return 0;
       return -1;
}
```

Comparator

Das *Comparator*-Interface funktioniert ähnlich wie das *Comparable*-Interface. Allerdings wird es (in der Regel) nicht von der Klasse implementiert, die geordnet werden soll, sondern ist ein funktionales Interface. Ein geordnetes *TreeSet* von *Projects* lässt sich wie folgt erstellen:

```
// Das kleinste/erste Projekt der Ordnung, ist das Projekt mit der geringsten Priorität
TreeSet<Project> projectSet = new TreeSet<>(new Comparator<Project>() {
            @Override
            public int compare(Project project, Project otherProject) {
                if (project.getPriority() < otherProject.getPriority()) return -1;
                if (project.getPriority() == otherProject.getPriority()) return 0;
                return 1;
            }
        });
```

equals

Objekte können mit der *equals()-Methode* auf „Gleichheit“ verglichen werden. Wenn eine Klasse *equals()* nicht überschreibt, gibt *equals()* nur dann true zurück, wenn es sich um dasselbe Objekt handelt (Implementierung in der Klasse *Object*).

Um die Gleichheitsdefinition für Objekte einer Klasse anzupassen, kann *equals()* überschrieben werden:

```
class Project {
    private String name;

    // Projekte mit dem gleichen Namen sollen als gleich angesehen werden.
    @Override
    public boolean equals(Object o) {
    if (this == o) return true;
    if (o == null || getClass() != o.getClass()) return false;
    Project project = (Project) o;
    if (name != null) return name.equals(project.name);
    return project.name == null;
    }
}
```

Map

Maps sind eine Datenstruktur aus Schlüssel-Wert-Paaren. Schlüssel dürfen nur einmal vorkommen, Werte dagegen beliebig oft. Aus dem JavaDoc des *Map*-Interfaces:

„ An object that maps keys to values. A map cannot contain duplicate keys; each key can map to at most one value. […] The Map interface provides three collection views, which allow a map's contents to be viewed as a set of keys, collection of values, or set of key-value mappings. The order of a map is defined as the order in which the iterators on the map's collection views return their elements. Some map implementations, like the TreeMap class, make specific guarantees as to their order; others, like the HashMap class, do not. ”

- *Das Map*-Interface erweitert nicht das *Collection*-Interface, jedoch ist es Teil des Java Collections Framework.

Map - Operationen

Interface Map<K,V> :

```
//Ein Schlüssel-Wert-Paar hinzufügen
V put(K key, V value)


//Einen Wert über seinen Schlüssel bekommen
V get(Object key)


//Ein Schlüssel-Wert-Paar entfernen
V remove(Object key)


//Ein Set aller Schlüssel der Map erhalten
Set<K> keySet()


//Eine Collection aller Werte der Map erhalten
Collection<V> values()


//Ein Set von Paaren aus Schlüsseln und Werten erhalten
Set<Map.Entry<K, V>> entrySet()
```

HashMap - Beispiel

```java
// Create HashMap
HashMap<Project, LocalDate> projectDeadlines = new HashMap<>();


// Put a key-value-pair in the map
projectDeadlines.put(project, LocalDate.of(2022, 12, 31));


// Get a value by its key
LocalDate deadline = projectDeadlines.get(project);


// Print all project names
projectDeadlines.keySet().forEach(p -> System.out.println(p.getName()));


// Print all project deadlines
projectDeadlines.values().forEach(d -> System.out.println(d.toString()));
```

Collections Klasse

Die *Collections*-Klasse ist eine Utility-Klasse die hilfreiche **statische** Methoden für das Arbeiten mit *Collections* **und** *Maps* bereitstellt. Etwa zum:

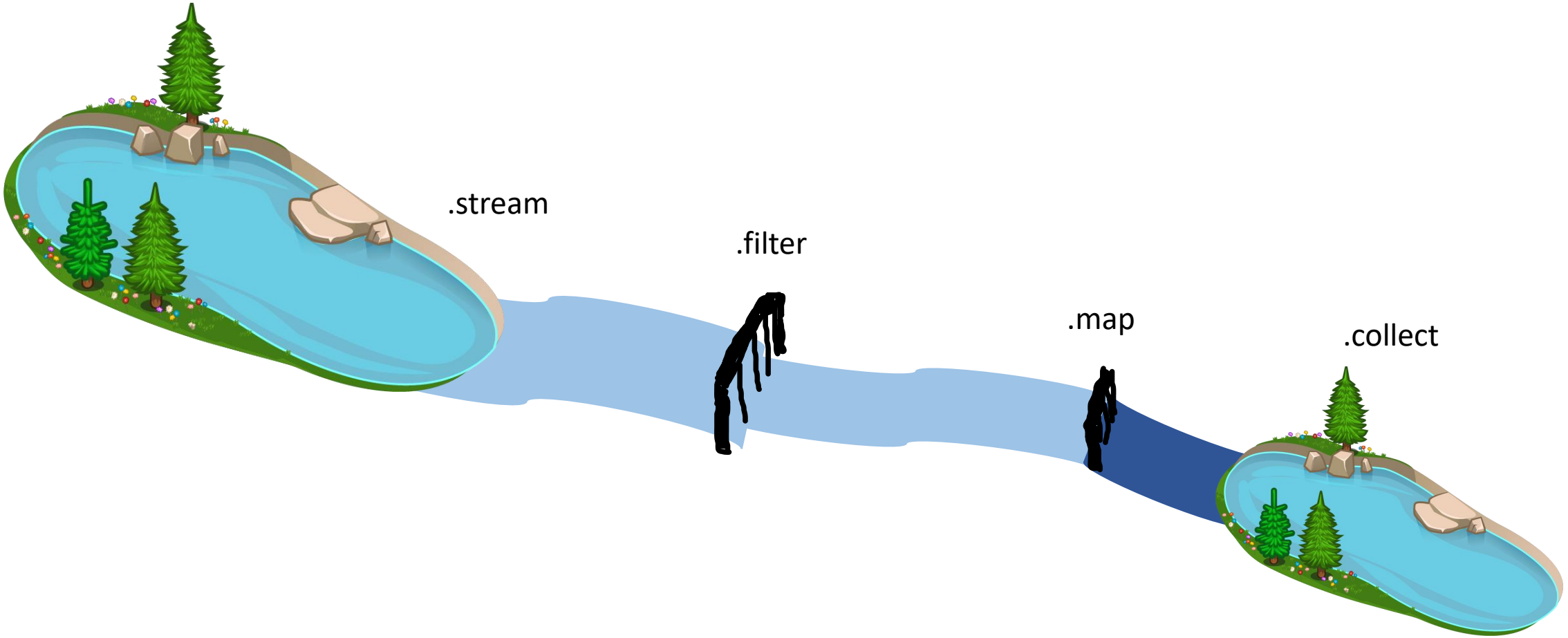
- Sortieren
- Finden des Maximums und Minimums
- Mischen
- …

Aus dem JavaDoc der *Collections*-Klasse:

„ This class consists exclusively of static methods that operate on or return collections. It contains polymorphic algorithms that operate on collections, "wrappers", which return a new collection backed by a specified collection, and a few other odds and ends. ”

Stream

Das *Interface Stream<T>* beschreibt einen „Strom“ von Elementen des Types T. Dieser „fließt“ von einer Quelle zu einem Ziel. Auf seinem Weg kann der Strom nahezu beliebig manipuliert werden.



Stream

- Ein *Stream* selbst speichert keine Elemente. Er bezieht sie aus seiner Quelle. Typische Quellen sind *Collections*, aber auch andere Quellen sind möglich.
- Nach der *Stream*-Erstellung über eine Quelle, können mittlere (intermediate) Operationen auf dem *Stream* ausgeführt werden.
- Der *Stream* wird mit einer terminalen Operation beendet.

Stream-Erstellung
- collection.stream()
- Stream.of(…)
- …

Mittlere Operatoren
- filter()
- map()
- reduce()
- sorted()
- …

Terminale Operatoren
- .collect()
- .sum()
- .average()
- …


```
Stream.of(3, 2, 1).filter(i -> i>1).sorted().collect(Collectors.toList());
```

Stream - Beispiele

```java
// Zähle, wie oft die Zahl 5 in der List vorkommt
integerList.stream().filter(i -> i.equals(5)).count();

// Multipliziere jedes Element des Sets mit 3
integerSet.stream().map(i -> i * 3).collect(Collectors.toSet());

// Errechne das Produkt aller Elemente in der Liste
integerList.stream().reduce((i1, i2) -> i1*i2).get();

// Errechne die Summe der Schlüssel der Map<Integer,Object>
integerMap.keySet().stream().mapToInt(i -> i).sum()
```