# 252-0027
# Einführung in die Programmierung

# 14. Java Collections Framework

*Manuela Fischer, Malte Schwerhoff*

**Departement Informatik**
**ETH Zürich**

# 14. Java Collections Framework

# Datenstruktur

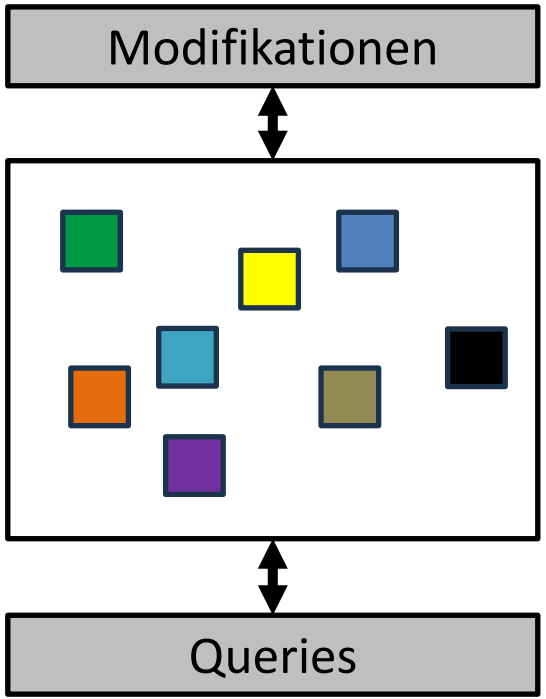
**Strukturiert** abgespeicherte **Daten**, so dass

- Queries («Anfragen») beantwortet und
- Modifikationen vorgenommen

werden können.

Struktur bestimmt über

- Art der Operationen
- Effizienz der Operationen



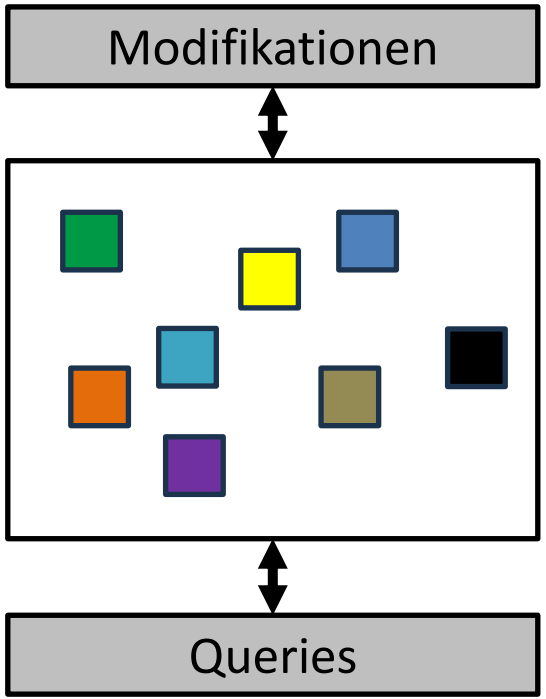
# Datenstruktur

Typische Queries

- `contains` (Suche: ist Element enthalten)
- `size` (Grösse)
- `toString` (String-Repräsentation)

Typische Modifikationen

- `add` (Element einfügen)
- `remove` (Element löschen)
- `clear` (alle Elemente löschen)



# Eigenschaften von Datenstrukturen

- **Geordnet** («**ordered**»): mit Reihenfolge
  - es gibt ein erstes und ein letztes Element
  - oft geordnet nach Einfügereihenfolge oder nach Grösse (sortiert)
- **Indexierbar** («**indexable**»): Elemente sind effizient indexierbar (Random Access)
- **Sortiert** («**sorted**»): angeordnet nach Grösse der Elemente
- **Duplikat-frei** («**duplicate-free**»): jedes Element erscheint nur max. einmal
- **Assoziativ** («**associative**»): assoziiert mit jedem Element einen Schlüssel
  - (`key`, `value`)-Paare gespeichert statt nur `value`
  - Zugriff via `key`

# Java Collections Framework

- Datenstrukturen
  - Interfaces, abstrakte Klassen und Klassen
  - Auswahl bezüglich Verfügbarkeit und Effizienz von Operationen
  - Anpassbar bezüglich Art der Operation (z.B. wie wird sortiert?)

- Service-Klasse **`java.util.Collections`**
  - Nicht zu verwechseln mit dem Interface `java.util.Collection`
  - Viel Funktionalität wie z.B. `Collections.sort()`
  - Ähnlich wie `Arrays` (`Arrays.deepCopy()`, `Arrays.toString()`, …)

# Service-Klasse `java.util.Collections`

**Package** java.util

## Class Collections

java.lang.Object
    java.util.Collections

---

```
public class Collections
extends Object
```

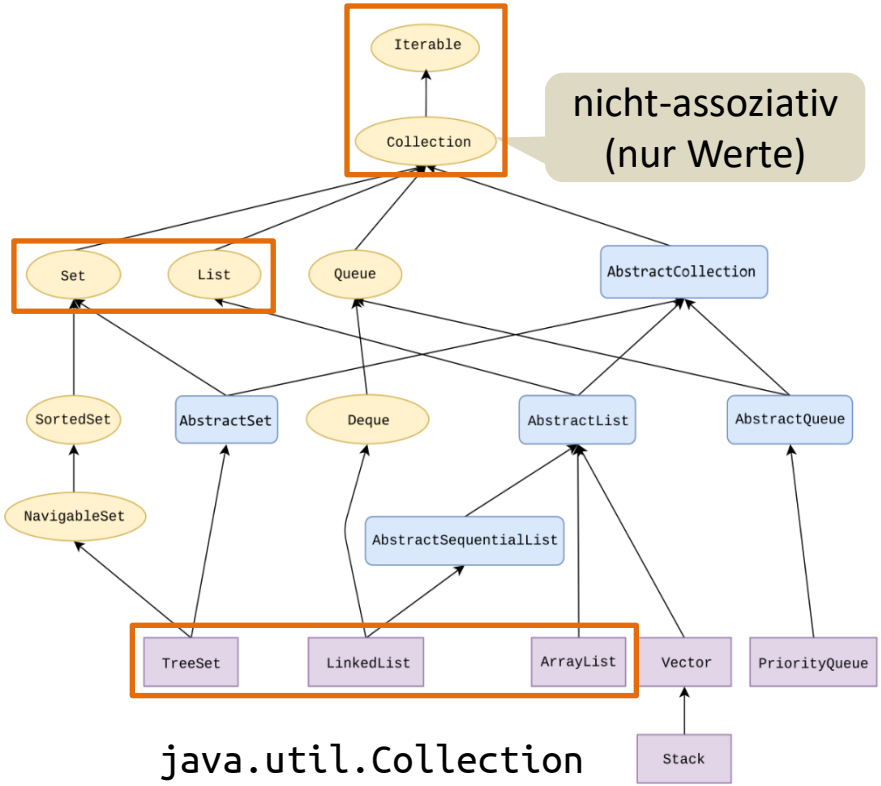
This class consists exclusively of static methods that operate on or return collections. It contains polymorphic algorithms that operate on collections, "wrappers", which return a new collection backed by a specified collection, and a few other odds and ends.

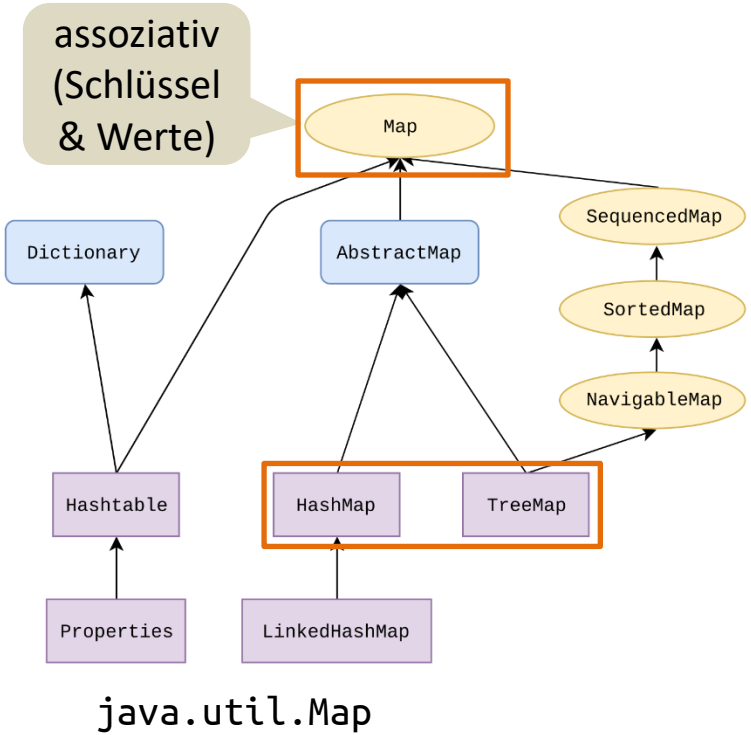
The methods of this class all throw a `NullPointerException` if the collections or class objects provided to them are null.

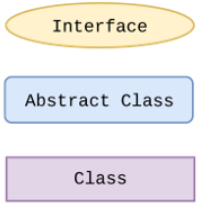
The documentation for the polymorphic algorithms contained in this class generally includes a brief description of the *implementation*. Such descriptions should be regarded as *implementation notes*, rather than parts of the *specification*. Implementors should feel free to substitute other algorithms, so long as the specification itself is adhered to. (For example, the algorithm used by `sort` does not have to be a mergesort, but it does have to be *stable*.)

https://docs.oracle.com/en/java/javase/21/docs/api/java.base/java/util/Collections.html

# Datenstrukturen: Collection & Map




Grafik von https://en.wikipedia.org/wiki/Java_collections_framework

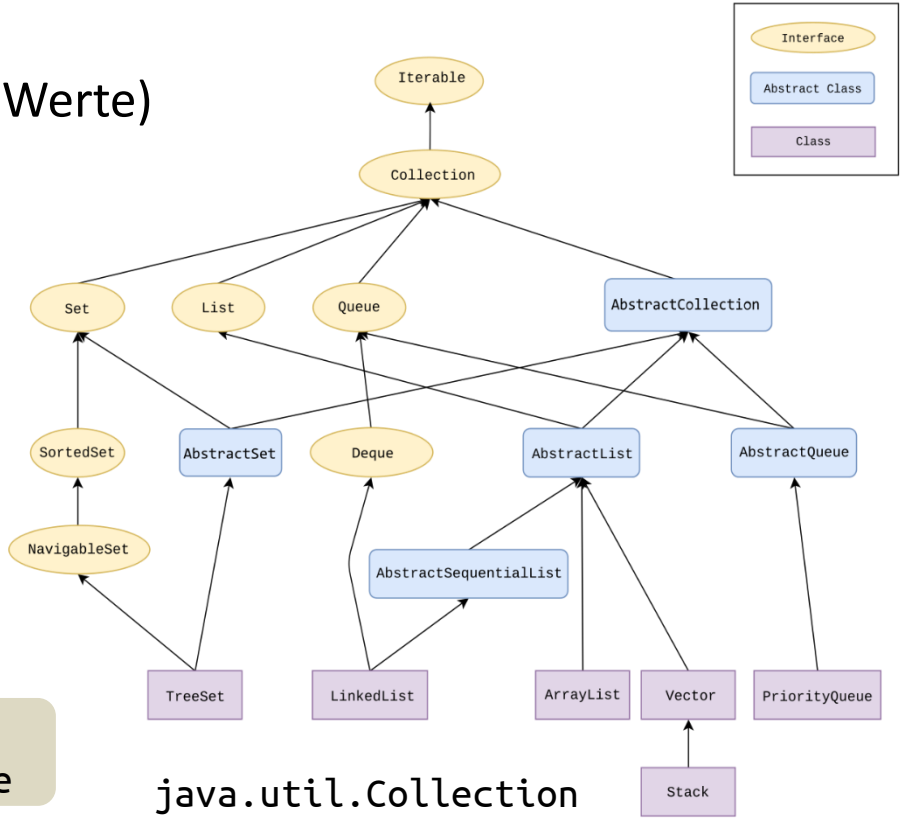
# Interface Collection («Ansammlung»)

- Nicht-assoziativ (keine Schlüssel, nur Werte)

Typparameter, mehr dazu im Kapitel zu Generics

```
public interface Collection<E> ... {
  public void add(E element);
  public T remove(Object e);
  public void clear();
  public boolean contains(Object e);
  public boolean isEmpty();
  public int size();
  ...
}
```

Lineare Suche nach Element e




java.util.Collection

# 14. Java Collections Framework

# Interface `List` («Liste»)

- Nicht-assoziativ (keine Schlüssel, nur Werte)
- Geordnet: 0-basierten Index

```
public interface List<E> extends Collection<E> {
  public void add(int index, E element);
  public int indexOf();
  public E get(int index);
  public E remove(int index);
  public E set(int index, E element);
  ...
}
```

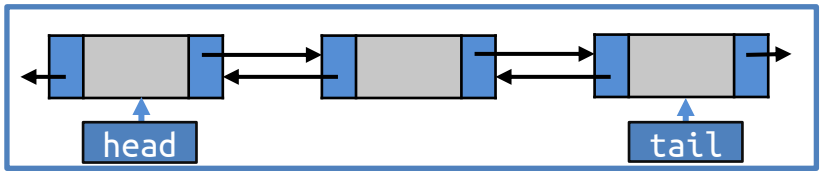
`0 <= index < size()`, sonst `IndexOutOfBoundsException`

Mit Vorsicht zu geniessen; können ineffizient sein!

**Warnung:** Nicht alle angebotenen Methoden sollten tatsächlich auch verwendet werden!

https://docs.oracle.com/en/java/javase/21/docs/api/java.base/java/util/List.html

# Klasse LinkedList<E>




- Doppelt verkettete Liste
- Index-basierter Zugriff ineffizient
- Effizientes **Einfügen/Löschen am Anfang und Ende** der Liste in $\Theta(1)$
- Geeignet für Implementation von Stack + Queue

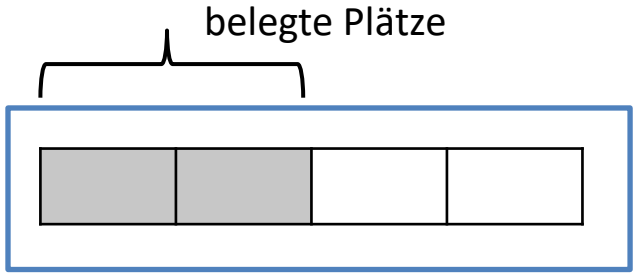
```
public class LinkedList<E> ... {
  public void addFirst(E element) { ... }
  public void addLast(E element) { ... }
  public E getFirst() { ... }
  public E getLast() { ... }
  public E removeFirst() { ... }
  public E removeLast() { ... }
  public boolean contains(Object o) { ... }
  ...
}
```

Lineare Suche

Mehr hier: https://docs.oracle.com/en/java/javase/21/docs/api/java.base/java/util/LinkedList.html

# Klasse `ArrayList<E>`

- Dynamisches (wachsendes) Array: Grösse kann sich zur Laufzeit beliebig ändern
- Einfügen/Löschen am Anfang ineffizient: Verschieben der späteren Elemente
- **Einfügen/Löschen am Ende effizient** in amortisiert $\Theta(1)$
- **Random Access**: effizienter Index-basierter Zugriff in $\Theta(1)$
- Mitführen der belegten Plätze; falls kein Platz mehr, werden Elemente in neues (um Faktor 1.5) grösseres Array kopiert.




```
public class LinkedList<E> ... {
  public void add(E element) { ... }
  public E removeLast() { ... }
  public E get(int index) { ... }
  public int indexOf(Object o) { ... }
  ...
}
```

Lineare Suche

Geeignet für Listen mit viel Index-Zugriffen und nur wenig Modifikationen

Mehr hier: https://docs.oracle.com/en/java/javase/21/docs/api/java.base/java/util/ArrayList.html

# Neue leere Liste erstellen

```
ArrayList<String> names = new ArrayList<String>();
LinkedList<Integer> numbers = new LinkedList<Integer>();
```

Typ kann auf rechter Seite auch weggelassen werden (**Diamond Operator** <>):
Typ wird inferiert vom Compiler

```
ArrayList<String> names = new ArrayList<>();
LinkedList<Integer> numbers = new LinkedList<>();
```

Typparameter muss der gleiche sein auf linker und rechter Seite!

```
ArrayList<Object> o = new ArrayList<String>(); // COMPILER ERROR!
```

# Beispiel

```
ArrayList<String> schools = new ArrayList<String>();
schools.add("ETH Zurich");
schools.add("Imperial College");
schools.add("MIT");
schools.add("ETH Lausanne");
for (int i = 0; i < schools.size(); i++) {
  if (schools.get(i).startsWith("ETH")) {
    System.out.println(schools.get(i));
  }
}
```

```
ETH Zurich
ETH Lausanne
```

# Neue Liste erstellen von Collection

- Konstruktoren von `ArrayList<E>` und `LinkedList<E>` können Collection `Collection<E>` als Argument erhalten: Kopie wird erstellt
- `Arrays.asList(...)` erstellt Collection (bestehend aus Argumenten), welche dem Konstruktor von `LinkedList` und `ArrayList` übergeben werden kann

```
Collection<String> c = Arrays.asList("aa", "b", "c");
ArrayList<String> a = new ArrayList<>(c);   // copy of c
LinkedList<String> l = new LinkedList<>(a); // copy of a
LinkedList<Integer> ll = new LinkedList<>(Arrays.asList(2, 9));
```

# Listen und Typen

- `LinkedList<E>` und `ArrayList<E>` und `List<E>` sind neue Typen; verwenden als
  - Attribut, lokale Variable

```
class Course {
  private ArrayList<String> studentNames;
  ...
}
```

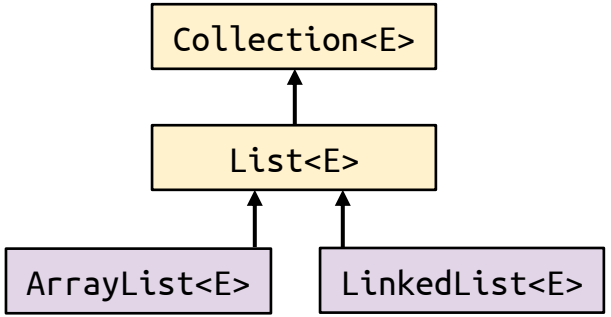
  - Parameter

```
void removeNouns(List<String> words) {
  ...
}
```

  - Rückgabewert

```
LinkedList<Integer> readIntsFromConsole() {
  ...
}
```

# Collections und Subtyping




Erinnerung:

- Referenzvariable für Klasse K kann auf Instanzen aller Subklassen von K verweisen
  - Umgekehrt nur mit explizitem Downcast
- Referenzvariable für Interface X kann auf Instanzen aller Klassen, die X implementieren, verweisen

Gilt auch für generische Klassen/Interfaces mit dem **gleichen Typparameter**:

```
List<String> names = new ArrayList<String>();
Collection<Integer> numbers = new LinkedList<Integer>();
ArrayList<String> strings = (ArrayList<String>) names;
```

Impliziter Upcast

Expliziter Downcast

# Achtung: Subtyping und Typparameter

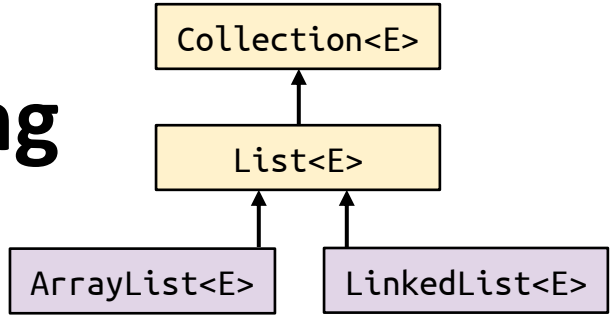
- Auch wenn B Subtyp ist von A, dann ist L<B> kein Subtyp von L<A>

```
ArrayList<Object> objects = new ArrayList<String>();     // ERROR!
List<Object> objects = new ArrayList<Integer>();         // ERROR!
```

- Aber Collection L<A> kann sowohl Instanzen vom Typ A als auch B speichern

```
ArrayList<Object> objects = new ArrayList<Object>();
objects.add(new Object());
objects.add("hello");
objects.add(1);
```

# Collections-Typen: Empfehlung




- Flexibel (allgemein) bleiben:
  - Mit allgemeinstem Typ arbeiten
  - Später über konkreten Typ entscheiden, ohne Programm zu ändern
  - Wahl der Datenstruktur aufschieben
- Achtung: Kann Einfluss auf die Laufzeit haben!

```
void f(List<Integer> list) {
  ...
  list.set(list.size() - 1, 42);
  ...
}
```

```
List<Integer> list1 = new ArrayList<>();
List<Integer> list2 = new LinkedList<>();
...
f(list1); // faster
f(list2); // slower!!
```

# Was passiert hier?

```
List<Integer> a = new ArrayList<>();
for (int i = 1; i <= 10; i++) { a.add(10 * i); }
for (int i = 0; i < a.size(); i++) { a.remove(i); }
```

```
List<Integer> a = new ArrayList<>();
for (int i = 1; i <= 5; i++) { a.add(2 * i); }
for (int i = 0; i < a.size(); i++) { a.add(i, 42); }
```

```
ArrayList<Character> a = new ArrayList<>(Arrays.asList('r' , 'o', 't'));
for (int i = a.size() - 1; i >= 0; i--) {
  if (i % 2 == 0) { a.add(a.get(i)); }
  else { a.add(0, a.get(i)); }
}
```

# Vorsicht vor Iteration und Modifikation

Vorsicht beim Iterieren über Liste, wenn Liste gleichzeitig modifiziert wird!

```
ArrayList<Integer> numbers = new ArrayList<>();
numbers.add(0);
for (int i = 0; i < numbers.size(); i++) {
  numbers.add(i); // infinite loop!
}
```

```
ArrayList<Integer> numbers = new ArrayList<>(Arrays.asList(0, 1, 2));
int size = numbers.size();
for (int i = 0; i < size; i++) {
  numbers.remove(i);  // IndexOutOfBoundsException:
}                     // Index 2 out of bounds for length 1
```

# Beispiel (Variante 1)

Methode `intersect`, die zwei Listen vom Typ `ArrayList<Integer>` als Argumente nimmt und eine neue Liste zurückgibt, die nur die Elemente der 1. Liste enthält, die auch in der 2. Liste vorkommen (in der gleichen Reihenfolge wie in der 1. Liste).

```
ArrayList<Integer> a = new ArrayList<>(Arrays.asList(1, 2, 3, 4, 1, 1));
ArrayList<Integer> b = new ArrayList<>(Arrays.asList(1, 6, 4));
ArrayList<Integer> c = intersect(a, b); // c == [1, 4, 1, 1]
```

```
ArrayList<Integer> intersect(ArrayList<Integer> a, ArrayList<Integer> b) {
  ArrayList<Integer> result = new ArrayList<>(); // empty list
  for (int i = 0; i < a.size(); i++) {
    if (b.contains(a.get(i))) { result.add(a.get(i)); }
  }
  return result;
}
```

# Beispiel (Variante 2)

Vorsicht beim Ändern von Listen, über die man gerade iteriert.

Methode `intersect`, die zwei Listen vom Typ `ArrayList<Integer>` als Argumente nimmt und eine neue Liste zurückgibt, die nur die Elemente der 1. Liste enthält, die auch in der 2. Liste vorkommen (in der gleichen Reihenfolge wie in der 1. Liste).

```
ArrayList<Integer> a = new ArrayList<>(Arrays.asList(1, 2, 3, 4, 1, 1));
ArrayList<Integer> b = new ArrayList<>(Arrays.asList(1, 6, 4));
ArrayList<Integer> c = intersect(a, b); // c == [1, 4, 1, 1]
```

```
ArrayList<Integer> intersect(ArrayList<Integer> a, ArrayList<Integer> b) {
  ArrayList<Integer> result = new ArrayList<>(a); // copy of a
  for (int i = 0; i < result.size(); i++) {
    if (!b.contains(result.get(i))) { result.remove(i);   i--;}
  }
  return result;
}
```

Entfernen ändert Indizes.
Ohne diese Anweisung wäre der Rückgabewert `[1, 3, 4, 1, 1]`.

# Beispiel (Variante 3)

Methode `intersect`, die zwei Listen vom Typ `ArrayList<Integer>` als Argumente nimmt und eine neue Liste zurückgibt, die nur die Elemente der 1. Liste enthält, die auch in der 2. Liste vorkommen (in der gleichen Reihenfolge wie in der 1. Liste).

```
ArrayList<Integer> a = new ArrayList<>(Arrays.asList(1, 2, 3, 4, 1, 1));
ArrayList<Integer> b = new ArrayList<>(Arrays.asList(1, 6, 4));
ArrayList<Integer> c = intersect(a, b); // c == [1, 4, 1, 1]
```

```
ArrayList<Integer> intersect(ArrayList<Integer> a, ArrayList<Integer> b) {
  ArrayList<Integer> result = new ArrayList<>(a); // copy of a
  result.retainAll(b); // only keep the elements that are in b
  return result;
}
```

# Suche

- `LinkedList`: mit `contains(Object)` in linearer Zeit (ineffizient!)

```
List<Integer> l = new LinkedList<>(Arrays.asList(9, 2, 3, 4, 2, 1));
System.out.println(l.contains(1)); // true
```

- `ArrayList`: mit `contains(Object)` / `indexOf(Object)` in linearer Zeit (ineffizient!)

```
List<Integer> a = new ArrayList<>(Arrays.asList(9, 2, 3, 4, 2, 1, 1));
System.out.println(a.contains(1)); // true
System.out.println(a.indexOf(1)); // 5
```

- **Sortierte `ArrayList`**: mit `Collections.binarySearch` in logarithmischer Zeit

```
List<Integer> a = new ArrayList<>(Arrays.asList(9, 2, 3, 4, 2, 1, 1));
Collections.sort(a); // sortieren!
if (Collections.binarySearch(a, 9) < 0) System.out.println("no luck!");
```

Mehr Details folgen!

**Achtung:** Binäre Suche in Java auch auf sortierter `LinkedList` erlaubt, aber nicht effizient!
Binäre Suche auf nicht-sortierter `List` führt zu fehlerhaften Resultaten!

# 14. Java Collections Framework

# Vergleiche von Elementen

- Gleichheit
  - `==` für primitive Typen
  - `equals`-Methode für alle Referenztypen (muss existieren, da in `Object` definiert)
- Natürliche Ordnung
  - `>` `<` `>=` `<=` für primitive Typen
  - `compareTo`-Methode für Referenztypen (kann implementiert werden, muss aber nicht)

```
class E {
  public int compareTo(E other) { ... }
  ...
}
```

```
e.compareTo(other)
  < 0 wenn e < other
  > 0 wenn e > other
  = 0 wenn e == other
```

# compareTo für String

```
e.compareTo(other)
 < 0 wenn e < other
 > 0 wenn e > other
 = 0 wenn e == other
```

- Gewisse Klassen, z.B. `String`, haben `compareTo` implementiert
- Dort entspricht natürliche Ordnung der (aufsteigend) **lexikographischen Ordnung**

```
System.out.println("a" < "world"); // ERROR; < not defined
System.out.println("hello".compareTo("world")); // -15
System.out.println("world".compareTo("hello")); // 15
System.out.println("12".compareTo("8"));        // -7
```

```
String a = "Alice";
String b = "Bob";
if (a.compareTo(b) < 0) { ... }
```

# Vergleich von Vergleichen

```
e.compareTo(other)
 < 0 wenn e < other
 > 0 wenn e > other
 = 0 wenn e == other
```

| Primitive Typen | Referenztypen |
|---|---|
| `if (a < b) { ... }` | `if (a.compareTo(b) < 0) { ... }` |
| `if (a <= b) { ... }` | `if (a.compareTo(b) <= 0) { ... }` |
| `if (a == b) { ... }` | `if (a.compareTo(b) == 0) { ... }` |
| `if (a != b) { ... }` | `if (a.compareTo(b) != 0) { ... }` |
| `if (a >= b) { ... }` | `if (a.compareTo(b) >= 0) { ... }` |
| `if (a > b) { ... }` | `if (a.compareTo(b) > 0) { ... }` |

Für selbst-definierte Klassen müssen wir `compareTo`-Methode zur Verfügung stellen!

# compareTo für unsere Klassen

```
class Person {
  String name;
  int age;
  Person(String name, int age) {
    this.name = name;
    this.age = age;
  }
}
```

```
Person p = new Person("Ann", 21);
Person q = new Person("Ben", 19);
if (p.compareTo(q) < 0) { // ERROR
  ...
}
```

*cannot find method compareTo(Person)*

**Delegation** an String-Klasse: Personen sollen anhand des Namens verglichen werden.

```
// in class Person
public int compareTo(Person other) {
  return name.compareTo(other.name);
}
```

```
System.out.println(p.compareTo(q));
```

```
-1
```

# Sortieren mit java.util.Collections

| | | |
|---|---|---|
| static <T extends Comparable<? super T>> void | sort(List<T> list) | Sorts the specified list into ascending order, according to the natural ordering of its elements. |
| static <T> void | sort(List<T> list, Comparator<? super T> c) | Sorts the specified list according to the order induced by the specified comparator. |

Mehr Details folgen!

- Service-Klasse `java.util.Collections`
- Statische Methode `Collections.sort(List)` sortiert Liste (Liste wird verändert!)
- Verwendet `compareTo`-Methode für den Vergleich von Elementen

```
String[] w = {"my", "dog", "has", "no", "fleas"};
ArrayList<String> words = new ArrayList<>(Arrays.asList(w));
Collections.sort(words);
System.out.println(words);
```

```
["dog", "fleas", "has", "my", "no"]
```

https://docs.oracle.com/en/java/javase/21/docs/api/java.base/java/util/Collections.html#sort(java.util.List)

# Was ist der Output?

```
Object[] a = {"two", "three", "101", "2", "1010"};
ArrayList<Object> s = new ArrayList<>(Arrays.asList(a));
Collections.sort(s);
System.out.println(s);
```

# Sortieren von Elementen unserer Klassen

```
class Person {
  String name;
  int age;
  Person(String name, int age) {
    this.name = name;
    this.age = age;
  }
  public int compareTo(Person o) {
     return name.compareTo(o.name);
  }
}
```

```
ArrayList<Person> a = new ArrayList<>();
a.add(new Person("Ben", 19));
a.add(new Person("Zoe", 25));
a.add(new Person("Ann", 21));
Collections.sort(a); // COMPILER ERROR
```

*method sort is not applicable*

Trotz `compareTo`-Methode können wir Liste mit `Person`-Instanzen nicht sortieren.

**Problem:** Wir müssen deklarieren, dass wir `compareTo` implementieren!
→ Interface `Comparable` implementieren

# Interface Comparable<E>

| | | |
|---|---|---|
| static <T extends Comparable<? super T>> void | sort(List<T> list) | Sorts the specified list into ascending order, according to the natural ordering of its elements. |

```
public interface Comparable<E> {
  public int compareTo(E other);
}
```

- Interface `Comparable<E>` erlaubt Vergleiche von Instanzen der Klasse `E`
- Wenn `List<E>` mit `Collections.sort` sortiert werden soll, so muss die Klasse `E` das Interface `Comparable<E>` implementieren und `compareTo` definieren/erben

```
public class Person implements Comparable<Person> {
  ...
  public int compareTo(Person other) { ... }
}
```

# Weiteres Beispiel: Sortierbare 2D-Punkte

```
public class Point implements Comparable<Point> {
  private int x;
  private int y;
  public Point(int x, int y) {
    this.x = x;
    this.y = y;
  }
  public int compareTo(Point other) {
    if (x < other.x || (x == other.x && y < other.y)) return -1;
    if (x == other.x && y == other.y) return 0;
    return 1;
  }
}
```

```
e.compareTo(other)
 < 0 wenn e < other
 > 0 wenn e > other
 = 0 wenn e == other
```

NullPointerException falls other == null

# Konsistenz von `equals` und `compareTo`

- Für jede Klasse sollte `compareTo` konsistent mit `equals` sein:
  Für alle Instanzen `e1`, `e2`:
  `( e1.compareTo(e2) == 0) == e1.equals(e2)`

```
e.compareTo(other)
  < 0 wenn e < other
  > 0 wenn e > other
  = 0 wenn e == other
```

- "Ausnahme" bei `null`-Referenzen:
  - `e1.compareTo(null)` wirft `NullPointerException`
  - `e1.equals(null)` ist `false`

# Zusammenfassung: Sortieren und Comparable

```
static <T extends Comparable<? super T>>          sort(List<T> list)
void
```

Sorts the specified list into ascending order, according to the natural ordering of its elements.

```
public class ElementType implements Comparable<ElementType> {
  public int compareTo(ElementType other) { ... }
}
```

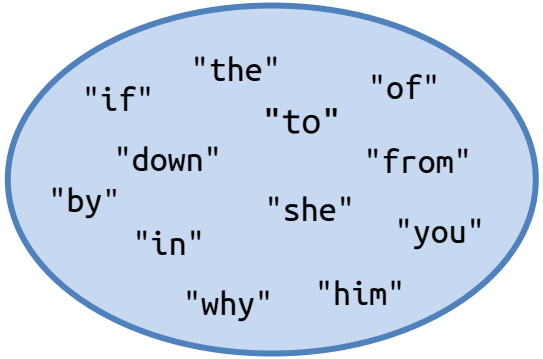
```
List<ElementType> l = ...;
Collections.sort(l);
```

Typische `compareTo`-Methoden:

- Delegation: Vergleich basierend auf Attribut:
  `return attributeName.compareTo(other.attributeName);`
- Vergleich basierend auf String-Repräsentation (`toString`):
  `return toString().compareTo(other.toString());`

# 14. Java Collections Framework

# Interface Set («Menge»)




- Nicht-assoziativ (keine Schlüssel, nur Werte)
- Keine Duplikate (Einfügen erfordert Suche!)
- Nicht geordnet, keine Reihenfolge, kein Index-basierter Zugriff

```
public interface Set<E> extends Collection<E> {
  public void add(E element);
  public boolean remove(Object o);
  public boolean cotains(E element);
  public boolean isEmpty();
  ...
}
```

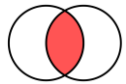
https://docs.oracle.com/en/java/javase/21/docs/api/java.base/java/util/Set.html

# Typische Mengenoperationen

```
Set<Integer> a = ...; // a == {1, 2, 3, 4}
Set<Integer> b = ...; // b == {3, 4, 5, 6}
```

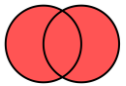
- Schnittmenge («Intersection») $A \cap B$

```
a.retainAll(b);    // a == {3, 4}
```



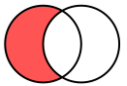
- Vereinigung («Union») $A \cup B$

```
a.addAll(b);       // a == {1, 2, 3, 4, 5, 6}
```

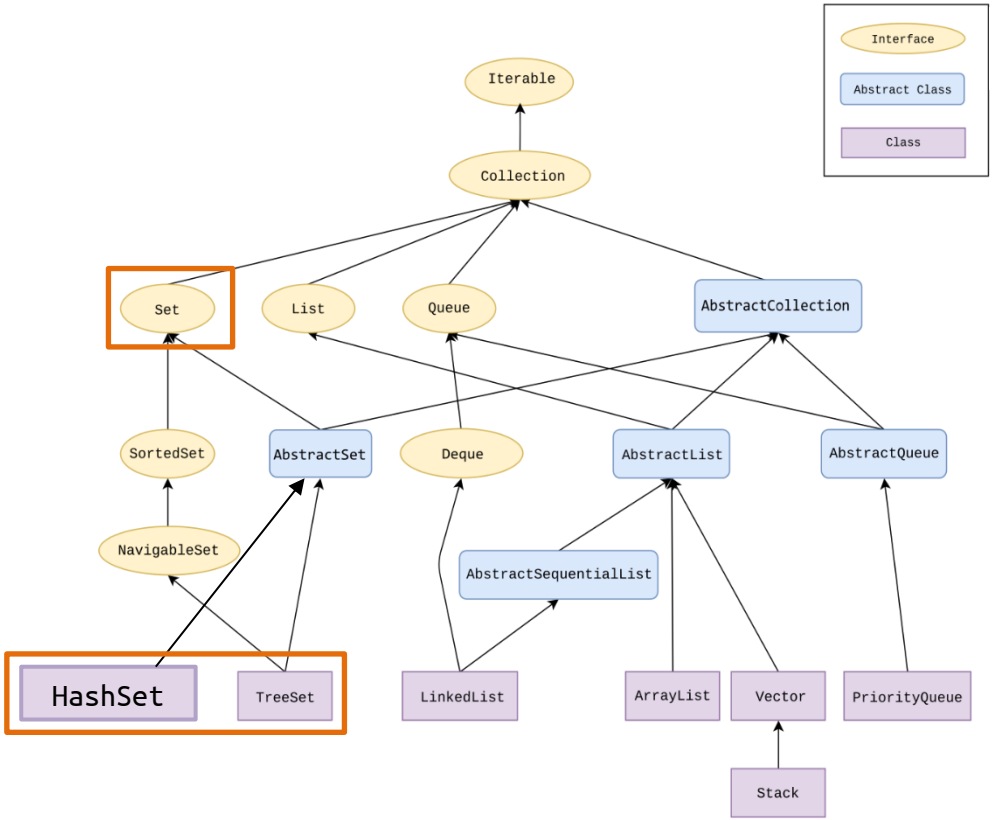


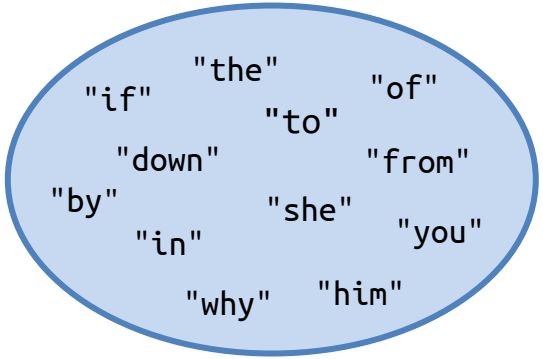
- Differenz («Difference») $A \setminus B$

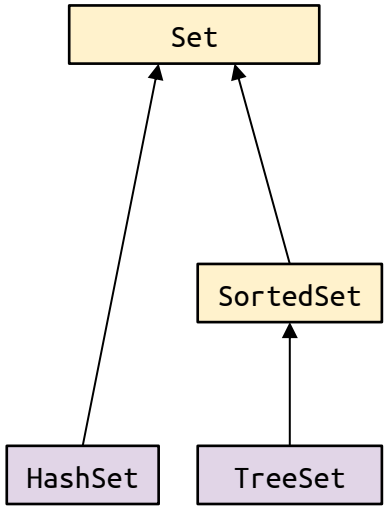
```
a.removeAll(b);    // a == {1, 2}
```



# Set-Implementationen

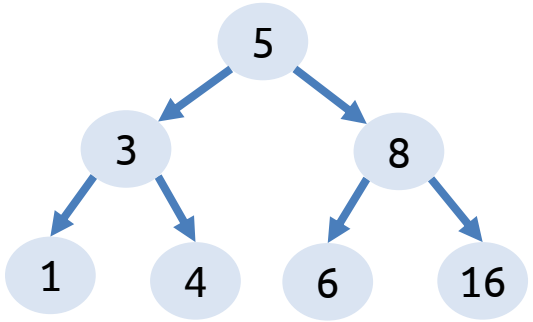


{4, 5, 16, 8, 6, 1, 3}

# TreeSet<E>: Idee




- Elemente **sortiert** in balanciertem binären Suchbaum
- Einfügen/Suchen/Löschen in **logarithmischer Zeit**
- Elemente brauchen **Ordnungsrelation**
  a. E muss Interface `Comparable<E>` und Methode `compareTo(E)` implementieren
  b. Beim Erstellen des `TreeSet`s muss Vergleichsmethode mitgegeben werden (später!)

```
int[] a = {4, 4, 5, 16, 8, 6, 1, 6, 1, 6, 1, 3, 3};
Set<Integer> set = new TreeSet<Integer>();
for (int i = 0; i < a.length; i++) {
  set.add(a[i]);
}
System.out.println(set);
```

Ausgabe in natürlicher Reihenfolge gegeben durch `compareTo`

```
[1, 3, 4, 5, 6, 8, 16]
```

# Vergessene compareTo-Methode

```
class Point {
  int x;
  int y;
  Point (int x, int y) { ... }
  public boolean equals(Object o) { ... }
}
```

```
class E implements Comparable<E> {
  public boolean equals(Object o) { }
  public int compareTo(E o) { }
}
```

```
Set<Point> p = new TreeSet<>(); // OK!
points.add(new Point(5, 4));   // NOT OK!
```

*Runtime Error: ClassCastException*
*Class Point cannot be cast to java.lang.Comparable*

**Warnung:** Vergessene `compareTo`-Methode führt (unglücklicherweise!!) zu Laufzeitfehler und nicht zu Compilerfehler!

**Tipp:** Für `TreeMap<E>` immer direkt `compareTo` und `equals` in `E` implementieren!

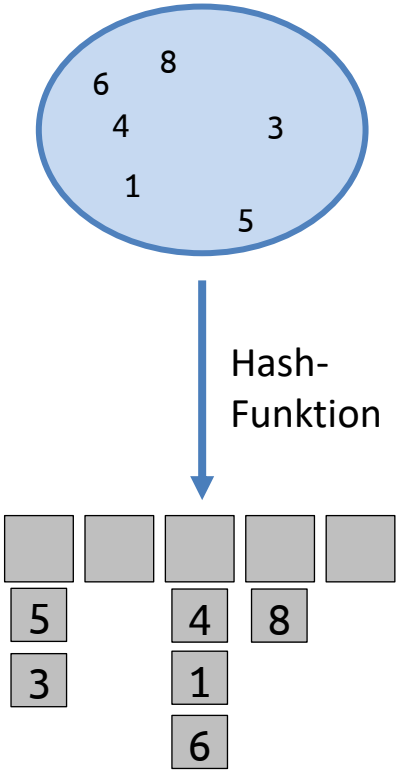
# TreeSet: Ordnungsabfragen

```
TreeSet<E> s;
E e;
```

- Elemente in umgekehrter Reihenfolge: `s.descendingSet()`
- Minimum/Maximum: `s.first()` & `s.last()`
- Strikt kleineres/grösseres Element: `s.lower(e)` & `s.higher(e)`
- Kleineres/grösseres Element: `s.floor(e)` & `s.ceiling(e)`
- Alle strikt kleineren/grösseren Elemente: `s.headSet(e)` & `s.tailSet(e)`
- Alle kleineren/grösseren Elemente: `s.headSet(e, true)` & `s.tailSet(e, true)`

https://docs.oracle.com/en/java/javase/21/docs/api/java.base/java/util/TreeSet.html

# Hash-Tabelle: Idee




- Hash-Tabelle: Sequenz (Array) von Buckets
- Hash-Funktion bildet Elemente auf Bucket (Array-Index) ab
- Alle Elemente mit dem gleichen Hash (Bucket-Index) werden in einer (verketteten) Liste abgespeichert
- Probleme …
  - Müssen Hash-Funktion definieren für unsere Elemente
  - Berechnung des Hash-Werts kann ineffizient sein
  - Falls zu viele Elemente im gleichen Bucket sind, werden Operationen extrem ineffizient
  - Keine (realistischen) theoretischen Garantien
- … aber dennoch schnell in Praxis falls sinnvolle Hash-Funktion!

# HashSet: Idee

- Elemente in nicht-bekannter Reihenfolge in Hash-Tabelle
- Einfügen/Suchen/Löschen in Praxis schnell
  (aber ohne theoretische Garantie!)
- Elemente brauchen keine Vergleiche (`compareTo`) aber **Hash-Funktion** (`hashCode`)

```
int[] a = {4, 4, 5, 55, 66, 111, 1, 1, 66};
Set<Integer> set = new HashSet<Integer>();
for (int i = 0; i < a.length; i++) {
  set.add(a[i]);
}
System.out.println(set);
```

Ausgabe in beliebiger Reihenfolge

```
[1, 66, 4, 5, 55, 111]
```

https://docs.oracle.com/en/java/javase/21/docs/api/java.base/java/util/HashSet.html

# Hash-Funktion hashCode()

### hashCode

```
public int hashCode()
```

Returns a hash code value for the object. This method is supported for the benefit of hash tables such as those provided by `HashMap`.

The general contract of `hashCode` is:

- Whenever it is invoked on the same object more than once during an execution of a Java application, the `hashCode` method must consistently return the same integer, provided no information used in `equals` comparisons on the object is modified. This integer need not remain consistent from one execution of an application to another execution of the same application.
- If two objects are equal according to the `equals` method, then calling the `hashCode` method on each of the two objects must produce the same integer result.
- It is *not* required that if two objects are unequal according to the `equals` method, then calling the `hashCode` method on each of the two objects must produce distinct integer results. However, the programmer should be aware that producing distinct integer results for unequal objects may improve the performance of hash tables.

**Implementation Requirements:**

As far as is reasonably practical, the `hashCode` method defined by class `Object` returns distinct integers for distinct objects.

# Hash-Funktion hashCode()

- Definiert in der Klasse `Object`
- Wunschdenken (nicht möglich!): Kollisionsfreiheit
  `o1.equals(o2) == (o1.hashCode() == o2.hashCode())`
- Muss folgendes Verhalten haben:
  1. Wiederholter Aufruf von `o.hashCode()` innerhalb einer Ausführung liefert den gleichen Wert, ausser die Attribute (die zur Berechnung vom Hash-Code beitragen) haben sich geändert.
  2. Wenn `o1.equals(o2)`, dann `o1.hashCode() == o2.hashCode()`
- **Konsequenz/Regel**:
  Wird eine der Methoden `hashCode` und `equals` überschrieben in einer Klasse, so muss auch die andere überschrieben werden!

# Beispiel hashCode

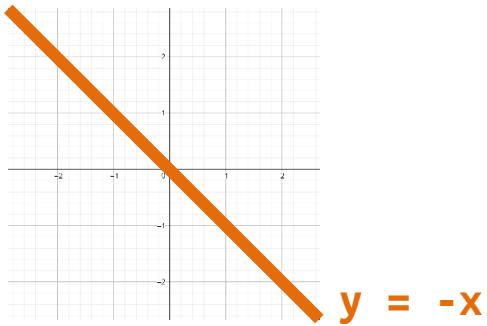
```
class Point {
  int x;
  int y;
  ...
  @Override
  public boolean equals(Object other) {
    if (!(other instanceof Point)) {
      return false;
    } else {
      return x == ((Point) other).x && y == ((Point) other).y;
    }
  }
  @Override
  public int hashCode() {
    return x * 31 + y;
  }
}
```

p.equals(q) impliziert
p.hashCode() == q.hashCode()

# Warnung zu HashSet!

- Sinnvolle Hash-Funktion ist entscheidend für Effizienz
  - Default nur dann okay, wenn `equals` nicht überschrieben!
  - Triviale Hash-Funktion `return 1;` erfüllt Regel (und maximiert Kollisionen!)
  - Ob Hash-Funktion sinnvoll ist, hängt auch von der Verteilung der Werte ab!

```
class Point {
  int x; int y;
  public int hashCode() {
    return Integer.valueOf(x + y).hashCode();
  }
}
```




- `HashSet` **nicht geeignet** für Anfragen zu **Ordnung/Sortierung** der Werte (nächst-grösserer Wert; alle kleineren Werte; Maximum, …) → `TreeSet`, `HashSet` nur für Operationen `add`, `remove` & `contains`

# Keine Duplikate in Sets

**add**

`public boolean add(E e)`

Adds the specified element to this set if it is not already present. More formally, adds the specified element `e` to this set if the set contains no element `e2` such that `Objects.equals(e, e2)`. If this set already contains the element, the call leaves the set unchanged and returns `false`.

```
Set<String> unis = new TreeSet<>();
System.out.println(unis.add("MIT"));             // true
System.out.println(unis.add("ETH Zurich"));      // true
System.out.println(unis.add("ETH Zurich"));      // false
System.out.println(unis.add("ETH Lausanne"));    // true
System.out.println(unis.add("MIT"));             // false
System.out.println(unis.add("EPFL"));            // true
System.out.println(unis);
```

```
[EPFL, ETH Lausanne, ETH Zurich, MIT]
```

# Duplikate: Warnung 1

```
Set<Point> p = new HashSet<>();
Point p1 = new Point(2, 2);
Point p2 = new Point(2, 2);
Point p3 = p2;
p.add(p1);
p.add(p2);
p.add(p3);
System.out.println(p); // [(2, 2), (2, 2)]
```

```
class Point {
  int x;
  int y;
  public Point(int x, int y) {
    this.x = x;
    this.y = y;
  }
  public String toString() {
    return "(" + x + ", " + y + ")";
  }
}
```

- Falls die Methode `equals` nicht überschrieben wird, so werden beim Einfügen in ein `Set` die Adressen der Objekte verglichen (Default-Implementation).
- Objekte mit dem gleichen Wert aber unterschiedlicher Adresse können dann gleichzeitig in der Menge sein.

**Sofern nicht Adressen verglichen werden sollen, `equals`-Methode überschreiben** (und `hashCode`-Methode für `HashSet`)!

# Duplikate: Warnung 2

```
Set<Point> points = new HashSet<>();
Point p1 = new Point(1, 2);
Point p2 = new Point(2, 2);
points.add(p1);
points.add(p2);
p1.x = 2;
System.out.println(points);
```

```
[(2, 2), (2, 2)]
```

```
class Point {
  public int hashCode() {
    return x * 31 + y;
  }
  public boolean equals(Object o) {
    return o instanceof Point ?
           ((Point) o).x == x &&
           ((Point) o).y == y :
           false;
  }
}
```

- Falls Objekte, die sich bereits in einem `Set` befinden, verändert werden, so wird das `Set` nicht neu auf Duplikate überprüft.
- Ein Objekt kann so zu einem Duplikat eines anderen Objekts in der Menge werden.

**Objekte in einem `Set` nicht ändern, sondern entfernen, ändern und neu einfügen!**

# Beispiel: Priority Queue

In Java implementiert als Heap (und nicht als Suchbaum!)

- Priority Queue («Prioritätswarteschlange») mit Operationen
  - Wert einfügen
  - Kleinsten Wert entfernen
  - Kleinsten Wert abfragen
- Wie implementieren wir eine Priority Queue mit den Datenstrukturen, die wir bereits kennen?
  - ~~`LinkedList`~~ Unsortierte Daten nicht hilfreich für wiederholtes Finden des Min.
  - ~~`ArrayList`~~ Unsortierte Daten nicht hilfreich für wiederholtes Finden des Min.
  - ~~Sortierte `LinkedList`~~ Einfügen schwierig/ineffizient
  - ~~Sortierte `ArrayList`~~ Einfügen schwierig/ineffizient
  - ~~`TreeSet`~~ Minimum abfragen/entfernen nicht möglich mit Hash-Tabelle
  - `HashSet`

java.util.PriorityQueue gibt es bereits, müssen sie nicht selber implementieren. Es dient hier nur als Übung.

# Beispiel: Priority Queue (Teil 2)

```
class Task implements Comparable<Task> {
  String name;
  int priority;
  public Task(String n, int p) {
    name = n;
    priority = p;
  }
  public String toString() {
    return name + " (" + priority + ")";
  }
  public int compareTo(Task t) {
    return t.priority - priority;
  }
  ...
}
```

Tasks mit grossem `priority` kommen früher in der Ordnung

```
TreeSet<Task> todo = new TreeSet<>();
todo.add(new Task("Sport", 2));
todo.add(new Task("Sleep", 6));
todo.add(new Task("EProg", 4));
todo.add(new Task("Sport", 5));
while (!todo.isEmpty()) {
  Task t = todo.first();
  todo.remove(t);
  System.out.println(t);
}
```

```
Sleep (6)
Sport (5)
Eprog (4)
Sport (2)
```

# 14. Java Collections Framework

# Iterieren über alle Elemente

- Bisher: Iterieren über alle Elemente einer Liste/eines Arrays:
  einfach, da die Elemente einen Index haben…
- Über Mengen und andere ungeordnete Datenstrukturen können wir nicht mit Hilfe eines Index iterieren
- Braucht neue Technik: for-each-Schleife
  - Iteriert über alle Elemente einer Collection oder eines Arrays

```
for (E element : c) { // c is a collection of type Collection<E>
  // loop body
  // do something with element
}
```

# Beispiel for-each-Schleife

```
static void printAll(Collection<String> c) {
  for (String element : c) {
    System.out.print(element + " ");
  }
}
```

```
Set<String> s = new HashSet<>(Arrays.asList("Zoe","Anna","Vera"));
printAll(s); // Zoe Vera Anna (arbitrary order)
```

```
Set<String> ss = new TreeSet<>(Arrays.asList("Zoe","Anna","Vera"));
printAll(ss); // Anna Vera Zoe (natural order according to compareTo)
```

```
List<String> l = new LinkedList<>(Arrays.asList("Zoe","Anna","Vera"));
printAll(l); // Zoe Anna Vera (order of insertion)
```

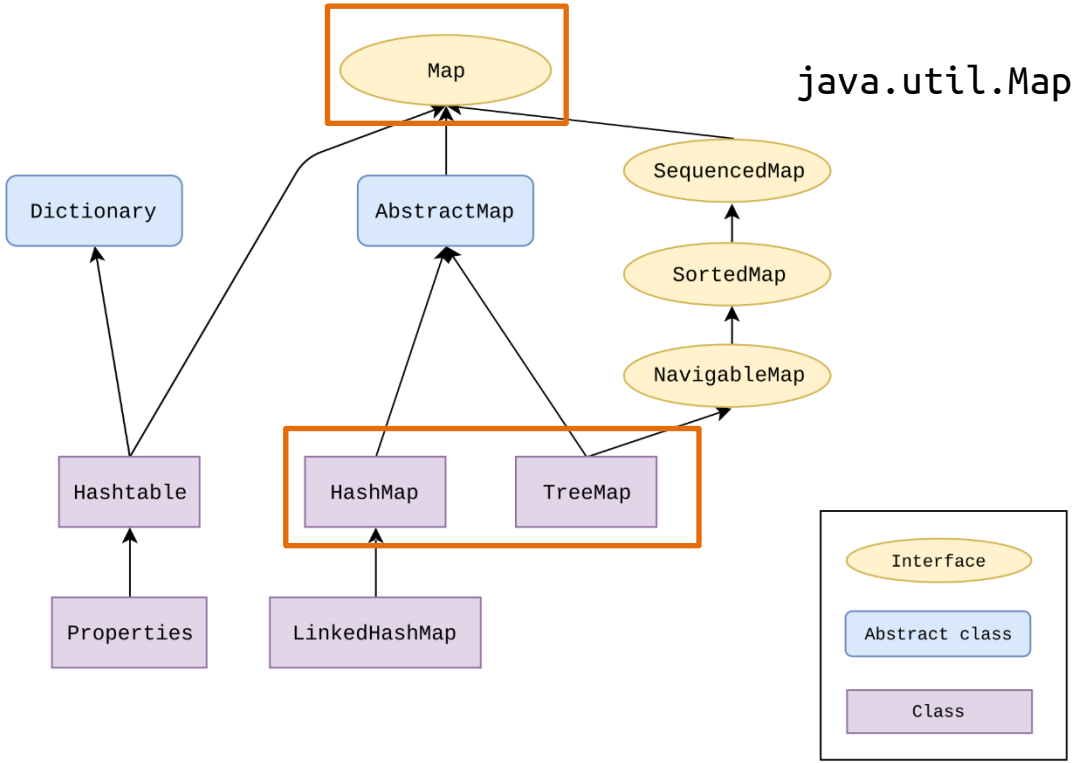
# Reihenfolge der Iteration (for-each-Schleife)

```
for (E element : Collection<E>) {
  // loop body
  // do something with element
}
```

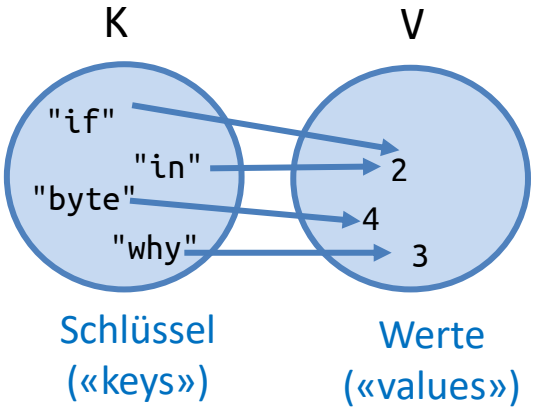
- `ArrayList`/`LinkedList`/Array:
  aufsteigender Index
- `TreeSet`:
  aufsteigend nach natürlicher Ordnung (`compareTo`)
- `HashSet`:
  nicht definiert
- `LinkedHashSet`:
  in Einfügereihenfolge (für EProg nicht relevant)

# 14. Java Collections Framework

# Map Interface («Abbildung»)



# Map Interface («Abbildung»)



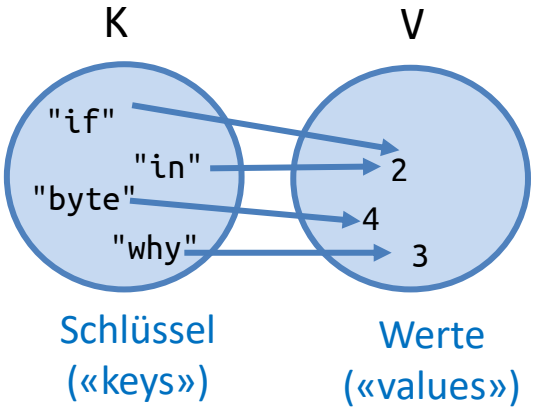

- Assoziativ (Schlüssel-Wert-Paare)
  - bildet Schlüssel auf Werte ab: Abbildung keys → values
  - Jeder Schlüssel hat einen assoziierten Wert
- Keine Schlüssel-Duplikate (Einfügen erfordert Suche!)
  - Jeden Schlüssel gibt es nur einmal: Schlüssel sind eine Menge!
  - Werte kann es mehrmals geben: Werte sind keine Menge!
- Nicht geordnet, keine Reihenfolge, kein Index-basierter Zugriff
- Oft auch Dictionary genannt und als Tabelle dargestellt!

```
if → 2
in → 2
byte → 4
why → 3
```

# Map Interface («Abbildung»)

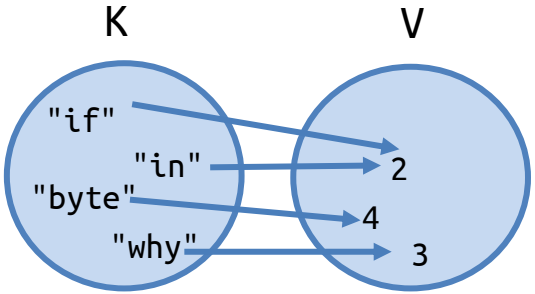
```
public interface Map<K, V> {
  public V put(K key, V value);
  public V replace(K key, V value);
  public V remove(Object key);
  public V get(Object key);
  public V getOrDefault(Object key, V defVal);
  public boolean cotainsKey(Object key);
  public boolean cotainsValue(Object value);
  public Set<K> keySet();
  public Collection<V> values();
  public boolean isEmpty();
  public int size();
  ...
}
```

2 Typparameter: für Schlüssel und Werte.



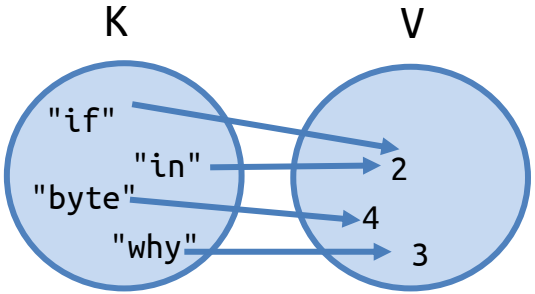

https://docs.oracle.com/en/java/javase/21/docs/api/java.base/java/util/Map.html

# Beispiel Map




```
Map<String, Integer> numberOfLetters = ...; // empty map
numberOfLetters.put("if", 2);
numberOfLetters.put("in", 2);
numberOfLetters.put("byte", 4);
numberOfLetters.put("why", 3);
System.out.println(numberOfLetters.get("why")); // 3
System.out.println(numberOfLetters.get("whyy")); // null
System.out.println(numberOfLetters.getOrDefault("why", -1)); // 3
System.out.println(numberOfLetters.getOrDefault("whyy", -1)); // -1
System.out.println(numberOfLetters.containsKey("whyy")); // false
System.out.println(numberOfLetters.containsValue(-1)); // false
```

# getOrDefault

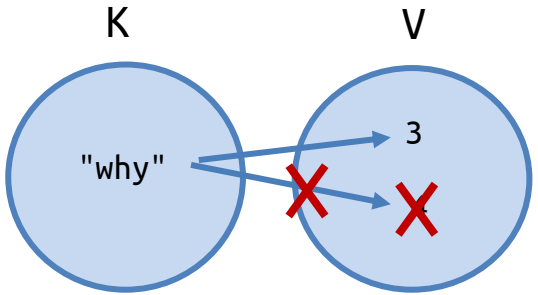



```
int nr;
if (numberOfLetters.get("whyy") == null) {
  nr = -1;
} else {
  nr = numberOfLetters.get("whyy");
}
```

```
int nr = numberOfLetters.getOrDefault("whyy", -1);
```

- Falls Schlüssel existiert, so wird assoziierter Wert zurückgegeben
- Falls Schlüssel nicht existiert, dann wird übergebener Default-Wert zurückgegeben

# Wert überschreiben

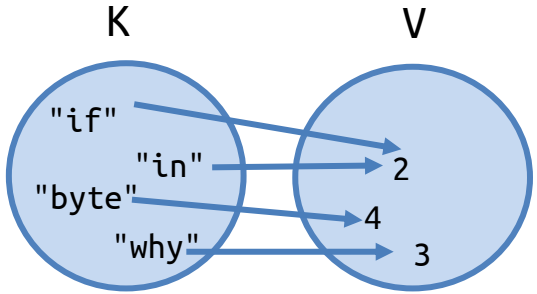


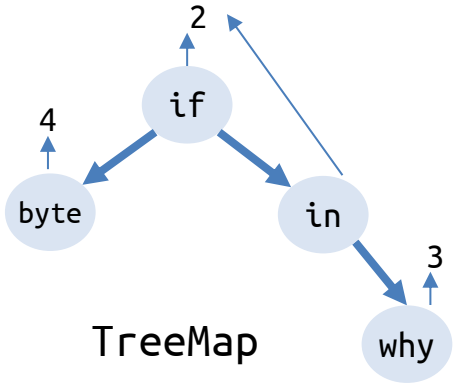

```
numberOfLetters.put("why", 4);
numberOfLetters.remove("why");
numberOfLetters.put("why", 3);
```

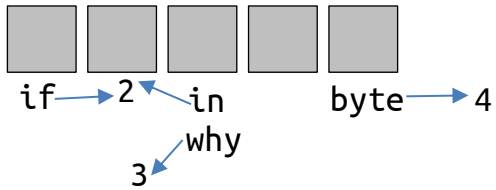
```
numberOfLetters.put("why", 4);
numberOfLetters.put("why", 3);
```

- Falls der Schlüssel bereits existiert, so überschreibt `put` den Wert.
- Kein Löschen des Schlüssels durch `remove` nötig!
- Ändern des Werts problemlos; Ändern des Schlüssels kann zu Problemen führen (ähnlich wie beim Ändern des Werts bei `Set`)

# Implementationen Map




- Schlüssel sind `Set`:
  als `TreeSet` (→ `TreeMap`) oder `HashSet` (→ `HashMap`) gespeichert
- Jeder Schlüssel hat Verweis auf seinen assoziierten Wert

https://docs.oracle.com/en/java/javase/21/docs/api/java.base/java/util/TreeMap.html
https://docs.oracle.com/en/java/javase/21/docs/api/java.base/java/util/HashMap.html

# `TreeMap`: Ordnungsabfragen

```
TreeSet<K, V> m;
K k;
```

- Elemente in umgekehrter Reihenfolge: `s.descendingMap()` & `s.descendingKeySet()`
- Min/Max: `s.firstEntry()` & `s.firstKey()` & `s.lastEntry()` & `s.lastKey()`
- Strikt kleineres/grösseres Element: `s.lowerKey(e)` & `s.higherEntry(e)`
- Kleineres/grösseres Element: `s.floorKey(e)` & `s.ceilingEntry(e)`
- Alle strikt kleineren/grösseren Elemente: `s.headMap(e)` & `s.tailMap(e)`
- Alle kleineren/grösseren Elemente: `s.headMap(e, true)` & `s.tailMap(e, true)`

`Entry` in Methoden-Name liefert Schlüssel-Wert-Paar; `Key` nur Schlüssel.
Aufzählung hier nicht vollständig.

Alle Abfragen und Modifikationen effizient!

# Maps Map<K, V>: Typen

- Alle Referenztypen möglich für K und V
  - Wrapper für primitive Typen
  - Typ kann auch durch Interface definiert sein, muss keine konkrete Klasse sein

- Beliebige Konstruktionen möglich!

```
Map<Shape, Double> umfang = new TreeMap<Shape, Double>();
Map<Set<Shape>, Integer> anzahl = new HashMap<>();
```

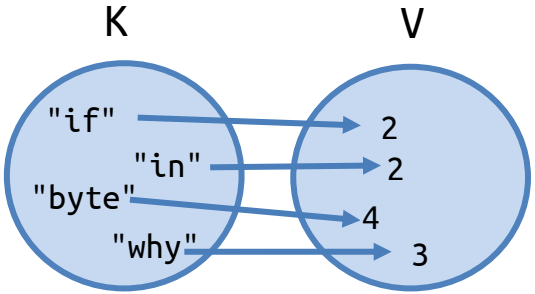
Mit oder ohne Diamond Operator

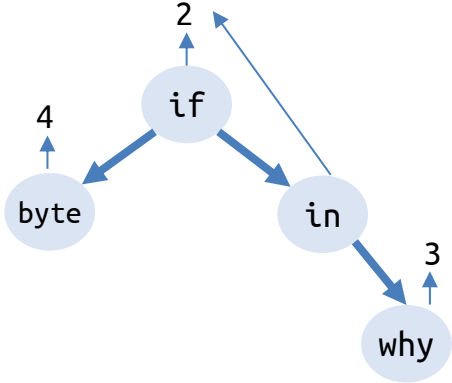
# Beispiel TreeMap

```
Map<String, Integer> t = new TreeMap<>();
t.put("if", 2);
t.put("in", 2);
t.put("byte", 4);
t.put("why", 3);
System.out.println(t);
System.out.println(t.higherKey("im"));
System.out.println(t.lastEntry());
```

```
{byte=4, if=2, in=2, why=3}
in
why=3
```

Natürliche Ordnung der Schlüssel, hier alphabetisch, da Strings!



# Mehrere Werte pro Schlüssel: Teil 1

```
class Record {
  String name;
  int grade;
  public Record(String name, int grade) {
    this.name = name;
    this.grade = grade;
  }
}
```

**K1 → V1 & V1'**
**K2 → V2 & V2'**
**K3 → V3 & V3'**
**K4 → V4 & V4'**

Wir möchten: `Map<K, V, V'>`

Stattdessen: `Map<K, Record>`

```
Map<String, Record> students = new HashMap<>();
students.put("11919040", new Record("Manuela", 1));
students.put("09123434", new Record("Malte", 6));
students.get("09123434").grade++;
System.out.println(students);
```

```
{11919040=Record@15db9742, 09123434=Record@2a139a55}
```

# Mehrere Werte pro Schlüssel: Teil 2

```
class Record {
  String name;
  int grade;
  ...
  public String toString() {
    return name + ":" + grade;
  }
}
```

**K1 → V1 & V1'**
**K2 → V2 & V2'**
**K3 → V3 & V3'**
**K4 → V4 & V4'**

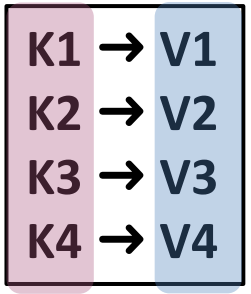
Wir möchten: `Map<K, V, V'>`

Stattdessen: `Map<K, Record>`

```
Map<String, Record> students = new TreeMap<>();
students.put("11919040", new Record("Manuela", 1));
students.put("09123434", new Record("Malte", 6));
students.get("09123434").grade++;
System.out.println(students);
```

```
{11919040=Manuela:1, 09123434=Malte:7}
```

# keySet() und values()




`Map<K, V>`

- **`keySet()`** gibt Referenz auf **`Set<K>`** aller Schlüssel zurück
  - Muss Typ `Set<K>` oder Supertyp zugewiesen werden; keiner konkreten Set-Klasse
  - Set darf nicht verändert werden: kein Hinzufügen/Entfernen von Elementen

- **`values()`** gibt Referenz auf Ansammlung **`Collection<V>`** aller Werte zurück
  - Muss Typ `Collection<V>` zugewiesen werden; keiner konkreten `Collection`-Klasse
  - Collection darf nicht verändert werden: kein Hinzufügen/Entfernen von Elementen
  - Werte erscheinen mehrmals, falls sie mehrmals referenziert (assoziiert) wurden

# Was ist der Output für myMap?

```
void mumble(Map<String, String> map) {
   Map<String, String> result = new TreeMap<String, String>();
   for (String key : map.keySet()) {
      if (key.compareTo(map.get(key)) < 0){
         result.put(key, map.get(key));
      } else {
         result.put(map.get(key), key);
      }
   }
   System.out.println(result);
}
```

```
myMap:
{  zero=null,
   eight=acht,
   four=vier,
   one=eins,
   two=zwei    }
```

# Beispiel v1

```
Map<String, String> dict = new TreeMap<>();
dict.put("Traene", "tear");
dict.put("Recht", "right");
dict.put("reissen", "tear");
dict.put("Glaeser", "glasses");
dict.put("rechts", "right");
dict.put("Brille", "glasses");
System.out.println(dict);
Set<String> germanWords = dict.keySet();
Collection<String> englishWords = dict.values();
System.out.println(germanWords);
System.out.println(englishWords);
```

Alphabetisch nach Schlüssel

```
{Brille=glasses, Glaeser=glasses, Recht=right, Traene=tear, rechts=right, reissen=tear}
[Brille, Glaeser, Recht, Traene, rechts, reissen]
[glasses, glasses, right, tear, right, tear]
```

# Beispiel v2

```
Map<String, String> dict = new TreeMap<>();
dict.put("tear", "Traene");
dict.put("right", "Recht");
dict.put("tear", "reissen");
dict.put("glasses", "Glaeser");
dict.put("right", "rechts");
dict.put("glasses", "Brille");
for (String englishWord : dict.keySet()) {
  System.out.println(englishWord + " -> " + dict.get(englishWord));
}
```

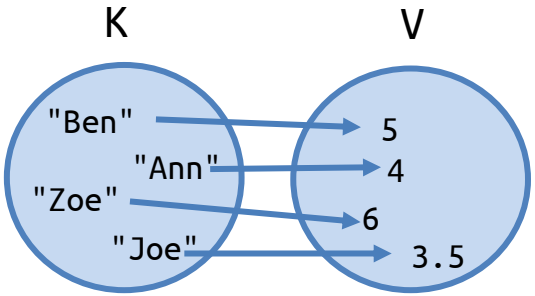
```
glasses -> Brille
tight -> rechts
tear -> reissen
```

Alternative Möglichkeit, um direkt über die Menge von Paaren zu iterieren wird hier nicht diskutiert.

# Beispiel: Wer hat Note 5.0?

```
Map<String, Double> grades = new HashMap<>();
grades.put("Ben", 5.0);
grades.put("Ann", 4.0);
grades.put("Zoe", 6.0);
grades.put("Joe", 3.5);
```

```
for (String name : grades.keySet()) {
  if (grades.get(name) == 5.0) {
    System.out.println(name);
  }
}
```

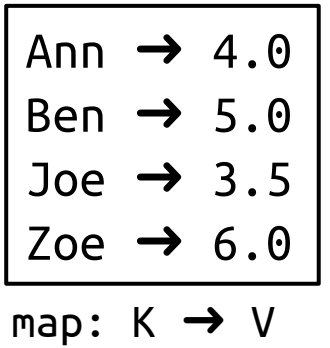



```
Ann → 4.0
Ben → 5.0
Joe → 3.5
Zoe → 6.0
```

Abspeichern in Datenstruktur statt neu berechnen?

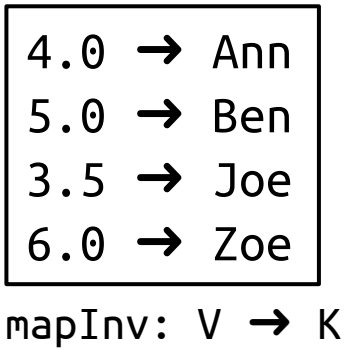
# Umkehrabbildung?

```
Map<String, Double> map = new HashMap<>();
map.put("Ben", 5.0);
map.put("Ann", 4.0);
map.put("Zoe", 6.0);
map.put("Joe", 3.5);
```




```
Map<Double, String> mapInv = new HashMap<>();
for (String key : map.keySet()) {
  mapInv.put(map.get(key), key);
}
```

```
mapInv.put(4.0, "Ann");
mapInv.put(5.0, "Ben");
mapInv.put(3.5, "Joe");
mapInv.put(6.0, "Zoe");
```




Idee: Abbildung umkehren, so dass keys values werden und umgekehrt

# Umkehrabbildung? Problem!

```
Map<String, Double> map = new HashMap<>();
map.put("Ben", 5.0);
map.put("Ann", 5.0);
map.put("Zoe", 6.0);
map.put("Joe", 3.5);
```

Ann → 5.0
Ben → 5.0
Joe → 3.5
Zoe → 6.0

map: k → v

```
Map<Double, String> mapInv = new HashMap<>();
for (String key : map.keySet()) {
  mapInv.put(map.get(key), key);
}
```

```
mapInv.put(5.0, "Ann");
mapInv.put(5.0, "Ben");
mapInv.put(3.5, "Joe");
mapInv.put(6.0, "Zoe");
```

5.0 → Ben
3.5 → Joe
6.0 → Zoe

mapInv: v → k

Problem: Mehrere Personen mit gleichen Note möglich; Information geht verloren!

# Umkehrabbildung

```
Map<String, Double> map = ...;
```

- Umkehrabbildung bildet values auf Menge von keys ab  `mapInv: v → Set<K>`

```
Map<Double, Set<String>> mapInv = new HashMap<>();
for (String k : map.keySet()) {
  Double v = map.get(k); // store value to avoid repeated search
  if (!mapInv.containsKey(v)) { // no entry yet in inverse map
    mapInv.put(v, new HashSet<>()); // add v -> { }
  }
  mapInv.get(v).add(k); // add k to set associated with v
} // NEW VERSION: every value is stored with a set of keys!
```

```
Map<Double, Set<String>> mapInv = new HashMap<>();
for (String k : map.keySet()) {
  Double v = map.get(k);
  mapInv.put(v, mapInv.getOrDefault(v, new HashSet<>()));
  mapInv.get(v).add(k);
} // NEW VERSION (shorter)
```

```
5.0 → {Ann, Ben}
3.5 → {Joe}
6.0 → {Zoe}
```

# Genügende Note?

```
5.0 → {Ann, Ben}
3.5 → {Joe}
6.0 → {Zoe}
```

- Mit HashMap: hoffnungslos…

```
Map<Double, Set<String>> mapInv = new HashMap<>();
// inverse map
for (String key : map.keySet()) {
  Double v = map.get(k);
  mapInv.getOrDefault(v, new HashSet<>()).add(k);
}
// iterate over all possible passing grades ?!?
for (Double grade = 4.0; grade <= 6.0; grade += ?? ) {
  if (mapInv.containsKey(grade)) {
    for (String name : mapInv.get(grade)) {
      System.out.println(name);
    }
  }
}
```

# Genügende Note?

```
5.0 → {Ann, Ben}
3.5 → {Joe}
6.0 → {Zoe}
```

- Mit `TreeMap`: einfach und elegant!

```
Map<Double, Set<String>> mapInv = new TreeMap<>();
// inverse map
for (String key : map.keySet()) {
  Double v = map.get(k);
  mapInv.put(v, mapInv.getOrDefault(v, new HashSet<>()));
  mapInv.get(v).add(k);
}
for (String name : mapInv.tailMap(4.0, true)) {
  System.out.println(name);
}
```

# Umkehrabbildung: Hash vs Tree

`Map<V, Set<K>>`

- **`HashMap`**`<V, Set<K>>`
  - Beliebige/keine Ordnung auf `v`
  - Alle `k` mit `v` grösser als `x` nicht möglich!

- **`TreeMap`**`<V, Set<K>>`
  - `v` sortiert gemäss natürlicher Ordnung (`compareTo`)
  - Alle `k` mit `v` grösser als `x` möglich: Welche Studenten haben genügende Note?

# 14. Java Collections Framework

# Unterschiedliche Ordnungen?

- Ziel: Studenten und ihre Noten ausgeben
  i. Aufsteigend sortiert nach Namen
  ii. Absteigend sortiert nach Namen
  iii. Sortiert nach Note

```
Ann → 4.0
Ben → 5.0
Joe → 3.5
Zoe → 6.0
```

- Variante i. ist direkt durch natürliche Ordnung der Strings in `TreeMap` gegeben.

```
TreeMap<String, Double> grades = ...; // add entries
for (String name : grades.keySet()) { // alphabetical order of keys
  System.out.println(name + ": " + grades.get(name));
}
```

- Variante ii.? Können `compareTo`-Methode in String-Klasse nicht ändern…

```
TreeMap<String, Double> grades = ...; // add entries
for (String name : grades.descendingKeySet()) {
  System.out.println(name + ": " + grades.get(name));
}
```

Glück gehabt. Brauchen allgemeinere Lösung!

# Vergleichsobjekte

- Interface `Comparator<E>`
  - Methode `int compare(E o1, E o2);` mit Verhalten
    - Rückgabewert < `0` wenn `o1` vor `o2` kommt
    - Rückgabewert > `0` wenn `o1` nach `o2` kommt
    - Rückgabewert `0` wenn `o1` und `o2` in der Ordnungsrelation gleich sind
- Vergleichsobjekt: `Comparator<E>`-Instanz (Instanz einer Klasse, die Interface implementiert)
  - definiert eine Ordnungsrelation für `E`-Instanzen
  - Eine Klasse pro Ordnungsrelation
  - Instanz übergeben an Sortiermethode oder an Konstruktor von `TreeSet` & `TreeMap`

https://docs.oracle.com/en/java/javase/21/docs/api/java.base/java/util/Comparator.html

# Rückwärts alphabetisch: Vergleichsobjekt

```
public class StringReverseComparator implements Comparator<String> {
  public int compare(String s1, String s2) {
    return -s1.compareTo(s2);
  }
}
```

- 2 Optionen: das Vergleichsobjekt übergeben an
  a. Sortiermethode als 2. Parameter:

     ```
     Collections.sort(myArrayList<String>, new StringReverseComparator());
     ```

  b. den Konstruktor von `TreeSet`/`TreeMap` als Parameter

     ```
     TreeSet<String> s = new TreeSet<>(new StringReverseComparator());
     ```

# Rückwärts alphabetisch: a. Sortieren

```
class StringReverseComparator implements Comparator<String> {
  public int compare(String s1, String s2) {
    return -s1.compareTo(s2);
  }
}
```

| | | |
|---|---|---|
| static <T extends Comparable<? super T>> void | sort(List<T> list) | Sorts the specified list into ascending order, according to the natural ordering of its elements. |
| static <T> void | sort(List<T> list, Comparator<? super T> c) | Sorts the specified list according to the order induced by the specified comparator. |

```
ArrayList<String> names = ...;
Collections.sort(names, new StringReverseComparator());
System.out.println(names);
```

# Rückwärts alphabetisch: b. TreeSet/TreeMap

```
class StringReverseComparator implements Comparator<String> {
  public int compare(String s1, String s2) {
    return -s1.compareTo(s2);
  }
}
```

```
TreeSet<String> names = new TreeSet<>(new StringReverseComparator());
...
System.out.println(names);
```

```
TreeMap<String, Double> grades = new TreeMap<>(new StringReverseComparator());
...
for (String name : grades.keySet()) {
  System.out.println(name + ": " + grades.get(name));
}
```

# Nach Note sortieren? 1. Versuch

- Idee: Noten als Schlüssel mit Vergleichsklasse für `Double`

```
class GradeComparator implements Comparator<Double> {
  public int compare(Double d1, Double d2) {
    return d1.compareTo(d2);
  }
}
```

```
TreeMap<Double, String> grades = new TreeMap<>(new GradeComparator());
...
for (Double grade : grades.keySet()) {
  System.out.println(grade + ": " + grades.get(grade));
}
```

**Problem:** Jetzt kann jede Note nur noch einmal vorkommen…

# Nach Note sortieren: Lösung (Teil 1)

- Idee: neue Klasse für Studenten (Paar von Name und Note) …

```
class Student {
  String name;
  double grade;
  public Student(String name, double grade) {
    this.name = name;
    this.grade = grade;
  }
}
```

- … und Vergleichsklasse…

```
class StudentGradeComparator implements Comparator<Student> {
  public int compare(Student s1, Student s2) {
    return s1.grade < s2.grade ? -1 :
           s1.grade > s2.grade ?  1 : s1.name.compareTo(s2.name);
  }
}
```

# Nach Note sortieren: Lösung (Teil 2)

- … so sortieren in `ArrayList`

```
ArrayList<Student> students = new ArrayList<>();
...
Collections.sort(students, new StudentGradeComparator());
for (Student s : set) {
  System.out.println(s.name + ": " + s.grade);
}
```

- … oder so in `TreeSet` speichern (jetzt nicht mehr assoziativ, keine `Map`!)

```
TreeSet<Student> students = new TreeSet<>(new StudentGradeComparator());
...
for (Student s : students) {
  System.out.println(s.name + ": " + s.grade);
}
```

# Bemerkung: Anonyme Klassen

- Bisher:
  Pro Ordnungsrelation braucht man eine Klasse…
- Alternative:
  Vergleichsmethode kann direkt im Parameter definiert werden
  als anonymes Objekt einer anonymen Klasse

```
TreeSet<Student> students = new TreeSet<>(new Comparator<Student>() {
                                  public int compare(Student s1, Student s2) {
                                    return s1.grade.compareTo(s2.grade);
                                  }
                                });
```

# `E.compareTo(E)` vs. `compare(E, E)`

- Parameter
  - `compareTo` hat nur einen Parameter: Aufruferobjekt ist impliziter Parameter
  - `compare` hat zwei Parameter: die zwei zu vergleichenden Objekte

- Wo implementiert?
  - `compareTo` in Klasse `E`, welche das Interface `Comparable<E>` implementiert
    - Natürliche Ordnung in Elementklasse definiert
    - Elemente haben immer die gleiche Ordnungsrelation, egal wo
  - `compare` in Vergleichsklasse, die das Interface `Comparator<E>` implementiert
    - Von Klienten (die die Elemente sortieren wollen) implementiert
    - Bietet Flexibilität bei der Auswahl der Ordnung: je nach Kontext anders sortieren

# Was ist der Output?

```
public class StudentComparatorLength implements Comparator<Student> {
  public int compare(Student s1, Student s2) {
    return s1.name.length() - s2.name.length();
  }
}
```

```
Comparator<Student> htc; // how to compare
htc = new StudentComparatorLength();
Set<Student> scores = new TreeSet<Student>(htc);
scores.add(new Student("Kim", 38));
scores.add(new Student("Lisa", 94));
scores.add(new Student("Roy", 87));
scores.add(new Student("Marty", 43));
scores.add(new Student("Marisa", 72));
System.out.println(scores);
```

# 14. Java Collections Framework

# Wiederholung: for-each-Schleife

```
for (E element : collection) {
  // loop body
  // do something with element
}
```

- `ArrayList` & `LinkedList` & Array:
  aufsteigender Index
- `TreeSet` & `TreeMap` & sortiertes Array :
  aufsteigend nach natürlicher Ordnung (`compareTo`)
- `HashSet` & `HashMap` :
  nicht definiert
- `LinkedHashSet` & `LinkedHashMap` :
  in Einfügereihenfolge (für EProg nicht relevant)

# Beispiel: Häufige Wörter (Teil 1)

- Ein Programm, das einen Text von einer Datei liest und alle Wörter, die öfters als 1000-mal in dem Text vorkommen, alphabetisch sortiert ausgibt…

```
Scanner input = new Scanner(new File("myText.txt"));
Map<String, Integer> wordCount = new TreeMap<String, Integer>();
while (input.hasNext()) {
  String word = input.next();
  wordCount.put(word, wordCount.getOrDefault(word, 0) + 1);
}
for (String word : wordCount.keySet()) {
  if (wordCount.get(word) > 1000) {
    System.out.println(word);
  }
}
```

# Beispiel: Häufige Wörter (Teil 2)

- ...und aus der Datenstruktur entfernt.

```
...
for (String word : wordCount.keySet()) {
  if (wordCount.get(word) > 1000) {
    System.out.println(word);
    wordCount.remove(word);
  }
}
```

```
java.util.ConcurrentModificationException
```

**Problem:** Die ***for-each*-Schleife ist read-only**, d.h. die Datenstruktur (über die wir iterieren) darf in der Schleife nicht modifiziert werden.

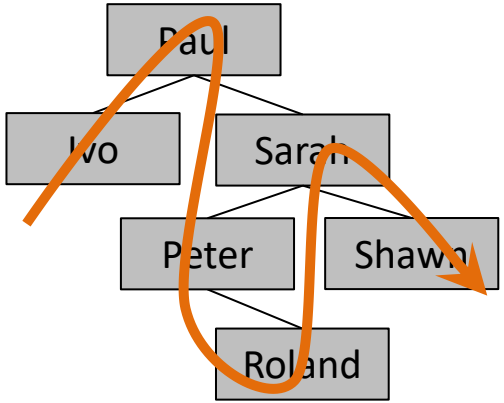
# Beispiel: Häufige Wörter (Teil 3)


- ...und aus der Datenstruktur entfernt.

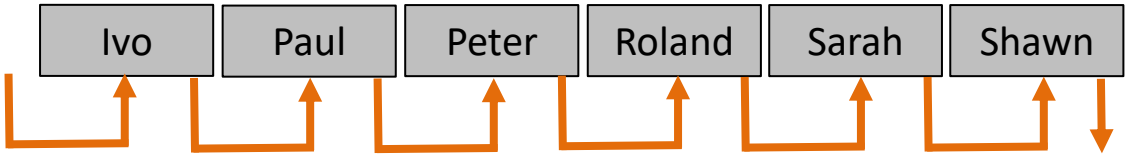
```
...
Set<String> wordSet = new TreeSet(wordCount.keySet()); // copy
for (String word : wordSet) {
  if (wordCount.get(word) > 1000) {
    System.out.println(word);
    wordCount.remove(word);
  }
}
```

**Hacky Lösung:** Die ***for-each*-Schleife** über eine Kopie der Schlüssel iterieren lassen: das funktioniert (manchmal), aber ist schlechter Stil!!

# Ordnung einer Datenstruktur



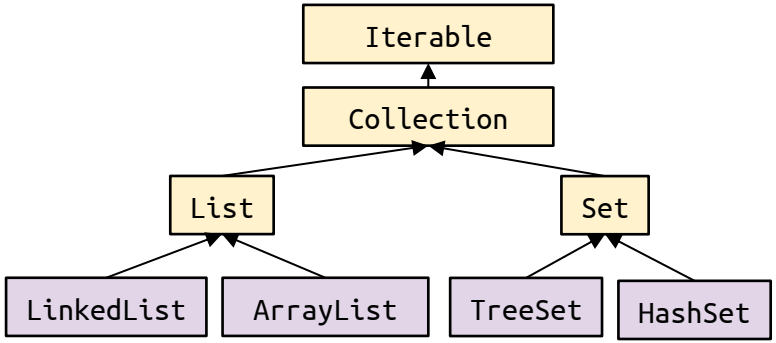

Datenstruktur (hier `TreeSet`)




Ordnung (hier Inorder-Traversierung des Baums)

- Jede Datenstruktur speichert intern eine **Ordnung**: die Reihenfolge, in der über die Datenstruktur iteriert wird (z.B. mit einer for-each-Schleife)
  - Kann man sich vorstellen wie eine verkettete Liste über die Elemente
- Interne Ordnung kann nach aussen geteilt werden via einem **Iterator-Objekt**
  - entkoppelt Iterieren über Datenstruktur von den Details der Darstellung

# Iterator-Objekt einer Collection




- Iterator kann verwendet werden, um über eine Collection zu iterieren.
- Jede Klasse, die `Iterable<T>` implementiert, (insbesondere jede Subklasse von `Collection<T>`) besitzt eine Methode `iterator()`, die ein Iterator-Objekt vom Typ `Iterator<T>` zurückgibt

```
Collection<String> collection = ...;
Iterator<String> it = collection.iterator(); // iterator object for collection
```

https://docs.oracle.com/en/java/javase/21/docs/api/java.base/java/lang/Iterable.html

# Iterator («iterator»)

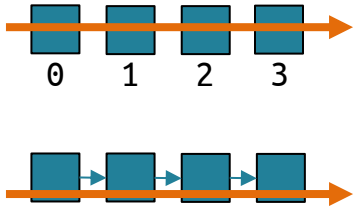
- `Iterator<T>` ist ein Interface in `java.util`
- Iterator-Objekt erlaubt einem Klienten, über die Elemente der Datenstruktur (in der intern gegebenen Reihenfolge) zu iterieren
  - Erinnert sich an die aktuelle Position (enthält den Zustand der Iteration)

| | |
|---|---|
| `hasNext()` | gibt `true` zurück, falls es noch mindestens ein Element gibt |
| `next()` | rückt zum nächsten Element vor und gibt das aktuelle Element zurück (wirft `NoSuchElementException` falls `!hasNext()`) |
| `remove()` | entfernt das letzte von `next()` zurückgegebene Element (wirft `IllegalStateException` falls `next()` noch nie aufgerufen wurde) |

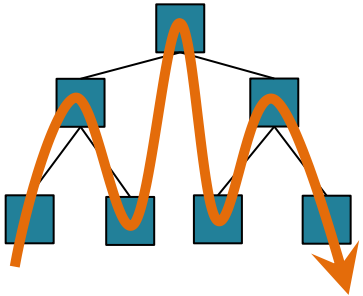
**Achtung:** Kein Einfügen oder Ändern des Elements möglich via Iterator, nur Entfernen!

Mit Subklasse `ListIterator<T>` ist Einfügen möglich, wird in EProg aber nicht diskutiert.
https://docs.oracle.com/en/java/javase/21/docs/api/java.base/java/util/Iterator.html

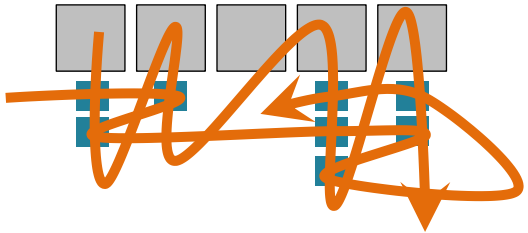
# Datenstrukturen und ihre Iteratoren




`ArrayList` & `LinkedList`:
aufsteigender Index



`TreeSet`:
Inorder-Traversierung

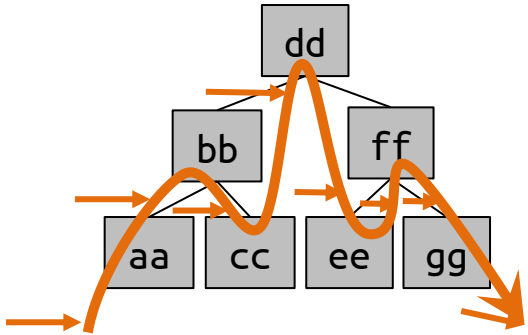


`HashSet`:
beliebige Reihenfolge

Im Folgenden Fokus hauptsächlich auf `TreeSet`, da nicht-trivial und nicht-arbiträr.

# Ausgabe

```
Set<String> words = new TreeSet<String>();
... // words == {aa, bb, cc, dd, ee, ff, gg}
Iterator<String> it = words.iterator();
while (it.hasNext()) {
  System.out.print(it.next());
}
```



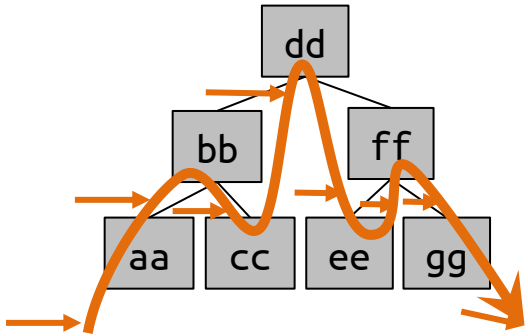

```
for (String word : words) {
  System.out.print(word);
}
```

Intuition (Teil 1)

- Iterator zeigt anfänglich vor das erste Element, dann jeweils zwischen zwei Elemente und final hinter das letzte Element
- `next()` überspringt das Element nach der Iterator-Position und gibt das gerade übersprungene Element zurück

# Alles löschen

```
Set<String> words = new TreeSet<String>();
... // words == {aa, bb, cc, dd, ee, ff, gg}
Iterator<String> it = words.iterator();
while (it.hasNext()) {
  it.next();
  it.remove();
}
```



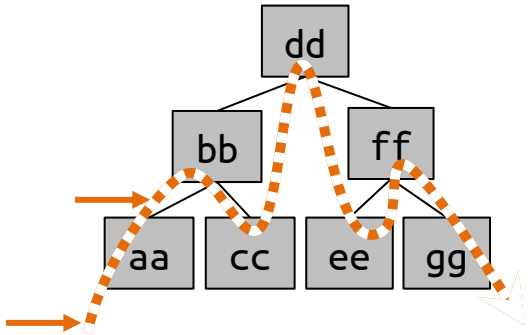

Intuition (Teil 2)

- `remove()` entfernt (zuletzt übersprungene) Element direkt vor Iterator-Position
- `remove()` darf nur aufgerufen werden, nachdem `next()` aufgerufen wurde (sonst `IllegalStateException`)

# Alles löschen: So nicht!

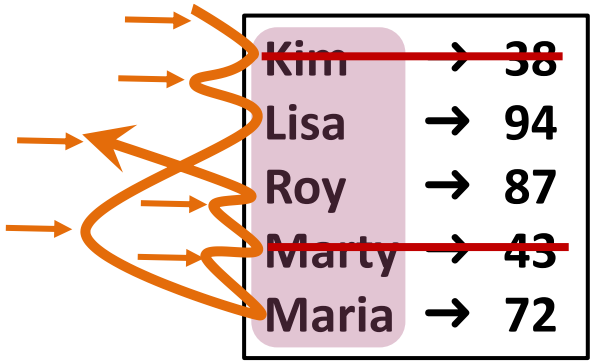
```
Set<String> words = new TreeSet<String>();
... // words == {aa, bb, cc, dd, ee, ff, gg}
Iterator<String> it = words.iterator();
while (it.hasNext()) {
  words.remove(it.next());
}
```

```
java.util.ConcurrentModificationException
```




**Achtung:** Datenstruktur darf nur via Iterator (`it.remove()`) verändert werden, sonst ist Iterator nicht mehr aktuell.

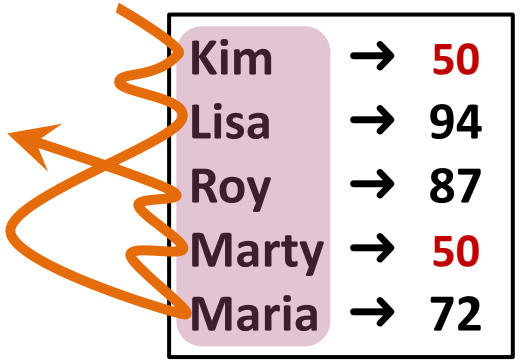
# Iteration über Map




```
// remove all students with score < 50 from map
Map<String, Integer> scores = new TreeMap<>();
// ...
Iterator<String> it = scores.keySet().iterator();
while (it.hasNext()) {
  String student = it.next();
  if (scores.get(student) < 50) {
    it.remove();
  }
}
```

- Iteration via Iteration über Schlüssel-Menge: `map.keySet().iterator()`
- Zugriff auf Value via `get`-Methode

# Ändern der Werte möglich!




```
// raise failing students' score to 50
Map<String, Integer> scores = new TreeMap<>();
// ...
Iterator<String> it = scores.keySet().iterator();
while (it.hasNext()) {
  String student = it.next();
  if (scores.get(student) < 50) {
    scores.replace(student, 50);
  }
}
```

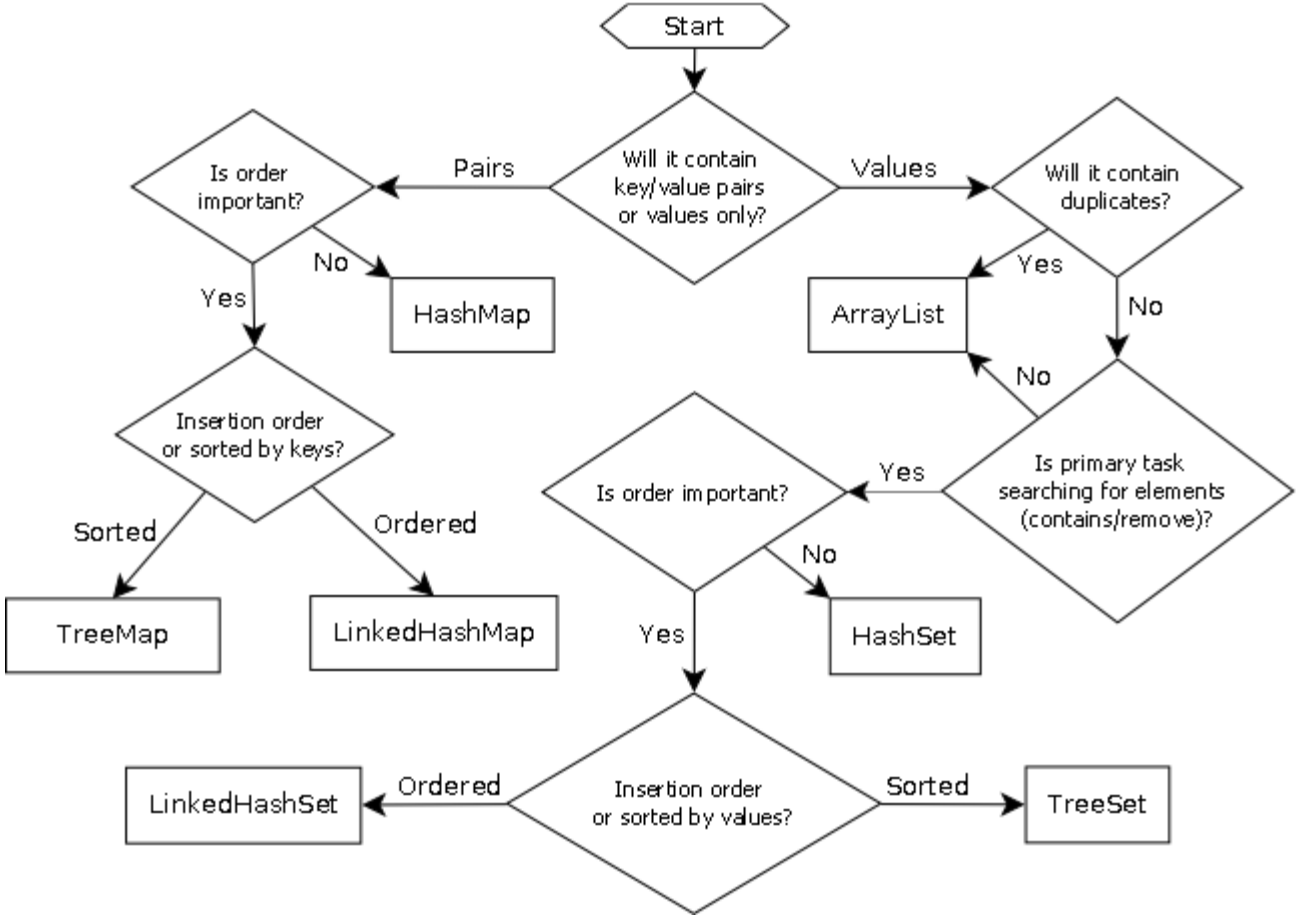
```
for (String student : scores.keySet()) {
  if (scores.get(student) < 50) {
    scores.replace(student, 50);
  }
}
```

- Ändern der Map via `replace`-Methode erlaubt (auch in for-each-Schleife!)
- Ändert nur Wert, nicht Schlüssel, und ändert deshalb Iterator-Reihenfolge nicht
- Einfügen (via `put`-Methode) nicht erlaubt: `ConcurrentModificationException`

# 14. Java Collections Framework

| | LinkedList | ArrayList | Sortierte ArrayList | TreeSet | HashSet | TreeMap | HashMap |
|---|---|---|---|---|---|---|---|
| Duplikate | | | | | | | |
| Assoziativ | (grundsätzlich möglich durch Speichern von Paaren als Werte) | | | | | | |
| Index | (ineffizient) | | | | | | |
| Geordnet | | | | | | | |
| Sortiert | | | | | | | |
| Min/Max | (ineffizient) | | | | | | |
| Suche | | | | | | | |
| Einfügen | (nur am Anfang/Ende) | (nur am Ende) | | | | | |
| Löschen | | | | | | | |

# Java Map/Collection Cheat Sheet




https://web.archive.org/web/20200802192003/http://www.sergiy.ca/guide-to-selecting-appropriate-map-collection-in-java/