Technische Hochschule Rosenheim
Technical University of Applied Sciences

Objektorientierte Programmierung
Kapitel 5 – List, Set und Map

Prof. Dr. Kai Höfig

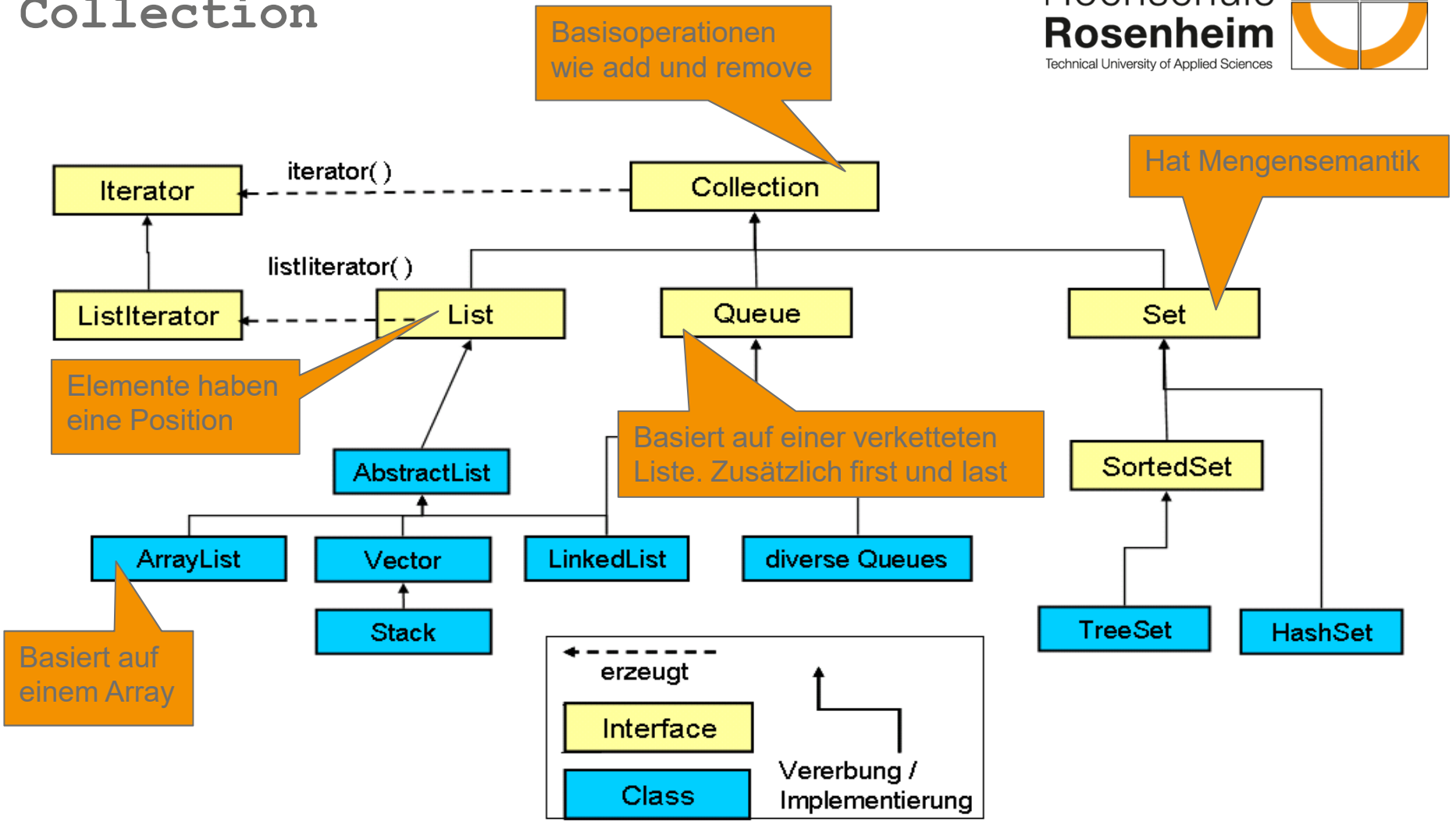
Motivation

Technische Hochschule Rosenheim
Technical University of Applied Sciences

- **Bisher haben wir**
  - Gesehen, was der Vorteil ist, Implementierungen gegen ein Interface zu entwickeln.
  - Eigene Implementierungen von Datenstrukturen in unterschiedlichen Varianten gesehen:
    - Mittels Array
    - Mittels Block Array
    - Mittels verketteter Liste
    - LIFO, FIFO Stack
    - (Binär-) Bäume
  - Mittels Generics typsichere generische Implementierungen solcher Datenstrukturen kennengelernt
  - Mit `Object`, `Comparable`, `Comparator`, `Iterable` und `Iterator` den Einsatzzwecke generischer Implementierungen erweitert
- **Jetzt lernen wir die existierenden generischen Datenstrukturen des Java Collection Frameworks kennen**

Speichern einzelner Elemente: `Collection`

Technische Hochschule Rosenheim
Technical University of Applied Sciences



https://docs.oracle.com/javase/8/docs/technotes/guides/collections/overview.html

Interface Collection<E> (1)

Technische Hochschule Rosenheim
Technical University of Applied Sciences

```
boolean add(E e)
```
Ensures that this collection contains the specified element (optional operation).

```
boolean addAll(Collection<? extends E> c)
```
Adds all of the elements in the specified collection to this collection (optional operation).

```
void    clear()
```
Removes all of the elements from this collection (optional operation).

```
boolean isEmpty()
```
Returns true if this collection contains no elements.

```
boolean contains(Object o)
```
Returns true if this collection contains the specified element.

```
boolean containsAll(Collection<?> c)
```
Returns true if this collection contains all of the elements in the specified collection.

```
boolean equals(Object o)
```
Compares the specified object with this collection for equality.

```
int     hashCode()
```
Returns the hash code value for this collection.

Interface Collection<E> (2)

Technische Hochschule Rosenheim
Technical University of Applied Sciences

```
Iterator iterator()
```
Returns an iterator over the elements in this collection

```
boolean  remove(Object o)
```
Removes a single instance of the specified element from this collection, if it is present (optional operation).

```
boolean  removeAll(Collection<?> c)
```
Removes all this collection's elements that are also contained in the specified collection (optional operation).

```
boolean  retainAll(Collection<?> c)
```
Retains only the elements in this collection that are contained in the specified collection (optional operation).

```
int      size()
```
Returns the number of elements in this collection.

```
Object[] toArray()
```
Returns an array containing all of the elements in this collection.

```
T[]      toArray(T[] a)
```
Returns an array containing all of the elements in this collection whose runtime type is that of the specified array.

Wiederholung: Liste und Set

Technische Hochschule Rosenheim
Technical University of Applied Sciences

- Wir kennen
  - Liste als *sequenziellen* Container der dynamisch wachsen und schrumpfen kann.
  - Set bzw. Binärbaum als *duplikatfreien* Container der einzigartige Elemente speichert.

```
List<String> entries = new LinkedList<>();
while (true) {
    System.out.print("Eingabe: ");
    String line = br.readLine();

    if (line == null)
        break;

    entries.add(line);
}

System.out.println("\nEingabe: " + entries);
```

```
Set<String> set = new TreeSet<>();
while (true) {
    System.out.print("Eingabe: ");
    String line = br.readLine();

    if (line == null)
        break;

    if (set.contains(line)) {
        System.out.println("vergeben.");
        continue;
    } else {
        entries.add(line);
        set.add(line);
    }
}
```

- *Immer dann die richtige Wahl, wenn zur Entwicklungszeit die Anzahl der Elemente nicht bekannt ist und/oder Mengensemantik benötigt wird.*

Assoziatives Datenfeld: Map

Technische Hochschule Rosenheim
Technical University of Applied Sciences

- Das assoziative Datenfeld (engl. *map*) speichert zu einem Schlüsselobjekt *K* genau ein Wertobjekt *V (K -> V).* In Java ist diese Datenstruktur als generisches Interface definiert:

```
interface Map<K, V> {
    void put(K key, V value);
    V get(K key);
    boolean containsKey(K key);
}
```

- Auffallend ist dabei, dass die Map über zwei Typvariablen verfügt:
  - **K** für den Schlüsseltyp (key) (= wie *Set*, speichert Wert genau einmal)
  - **V** für den Wertetyp (value) (mehrfach)

- Häufig vorkommende Datenstruktur in der Informatik, um zu einem *Key* schnell den dazugehörigen *Value* zu berechnen:

```
Map<Integer,String> pcodes = new TreeMap<Integer,String>();
pcodes.put(83101,"Thansau");
pcodes.put(83026,"Rosenheim");
pcodes.put(83022,"Rosenheim");
```

Interface `Map<K,V>` (1)

Technische Hochschule Rosenheim
Technical University of Applied Sciences

`Boolean containsKey(Object key)`
Returns true if this map contains a mapping for the specified key.

`Boolean containsValue(Object value)`
Returns true if this map maps one or more keys to the specified value.

`V get(Object key)`
if this map contains a mapping from a key k to a value v such that `Objects.equals(key, k)`, then this method returns v; otherwise it returns null.

`Boolean isEmpty()`
Returns true if this map contains no key-value mappings.

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

`V put(K key, V value)`
Associates the specified value with the specified key in this map (optional operation).

`void putAll(Map<? extends K,? extends V> m)`
Copies all of the mappings from the specified map to this map (optional operation).

Interface Map<K,V> (2)

Technische Hochschule Rosenheim
Technical University of Applied Sciences

```
Set<Map.Entry<K,V>> entrySet()
```
Returns a Set view of the mappings contained in this map.

```
V                   remove(Object key)
```
Removes the mapping for a key from this map if it is present (optional operation).

```
default boolean     remove(Object key, Object value)
```
Removes the entry for the specified key only if it is currently mapped to the specified value.

```
default V           replace(K key, V value)
```
Replaces the entry for the specified key only if it is currently mapped to some value.

```
default boolean     replace(K key, V oldValue, V newValue)
```
Replaces the entry for the specified key only if currently mapped to the specified value.

```
int                 size()
```
Returns the number of key-value mappings in this map.

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

Beispiel Vorgesetze

Technische Hochschule Rosenheim
Technical University of Applied Sciences

```
Map<Person,Person> vorgesetzte = new HashMap<>();

Person chef= new Person("Maria");
Person a1=new Person("Klaus");
Person a2 = new Person("Kathrin");
Person al = new Person("Michael");

vorgesetzte.put(chef, null);
vorgesetzte.put(al,chef);
vorgesetzte.put(a1,al);
vorgesetzte.put(a2,al);

System.out.println(vorgesetzte);

System.out.println(vorgesetzte.get(new Person("Michael")));
```

{Chapter05.Person@17c68925=Chapter05.Person@19dfb72a, Chapter05.Person@19dfb72a=null, Chapter05.Person@7e0ea639=Chapter05.Person@17c68925, Chapter05.Person@3d24753a=Chapter05.Person@17c68925}

{Michael=Maria, Maria=null, Klaus=Michael, Kathrin=Michael}

Mit überschriebener `toString` Methode

null

Maria

Wenn `equals` und `hashCode` richtig implementiert sind

Objektgleichheit mittels `equals`

Technische Hochschule Rosenheim
Technical University of Applied Sciences

- Gleichheit prüfen mittels `equals`
- In Java können zwei Objekte mit equals auf inhaltliche Gleichheit verglichen werden:

```
public class MeineKlasse {
    int attribut;
    public boolean equals(Object o) {
        // 1. Das_selbe_ Objekt?
        if (o == this)
            return true;
        // 2. Passt die Klasse?
        if (!(o instanceof MeineKlasse))
            return false;
        // umwandeln...
        MeineKlasse other = (MeineKlasse) o;
        // 3. Attribute vergleichen
        if (this.attribut != other.attribut)
            return false;
        return true;
    }
}
```

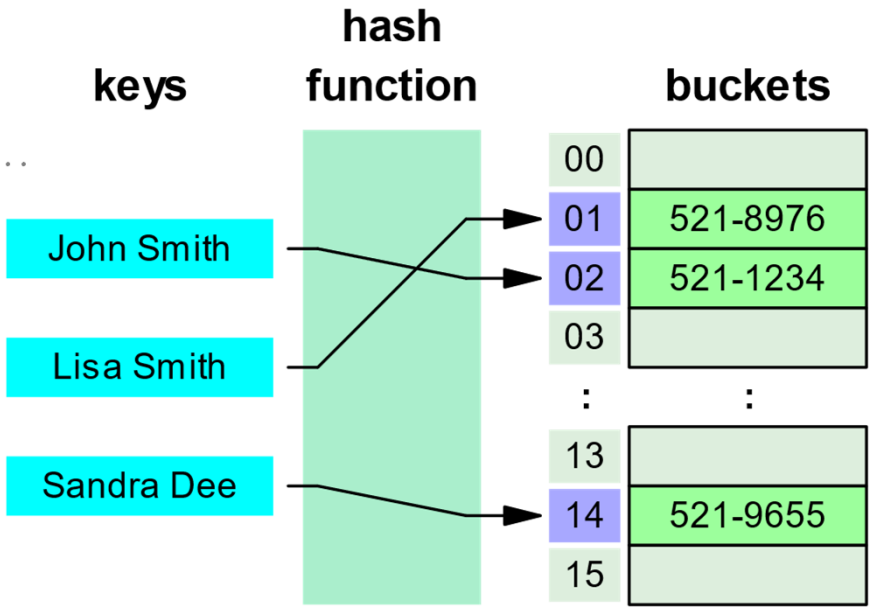
Exkurs: Effizienz durch Hashing

Technische Hochschule Rosenheim
Technical University of Applied Sciences

- Verwende Array wegen schnellem Direktzugriff
- Hashfunktion um von Schlüssel zu "Schublade" zu gelangen
- Object.hashCode
  https://docs.oracle.com/javase/10/docs/api/java/lang/Object.html#hashCode()
- Definiert für alle API Klassen (String, Double, etc.)
- Für eigene Klassen: hashCode implementieren:

```
public class MeineKlasse implements Comparable<MeineKlasse> {
    int attribut;
    String s;
…

    public int hashCode() {
        return s.hashCode() + attribut;  // zum Beispiel...
    }
}
```

- Eigenschaften einer guten Hash Funktion
  - Verteilt alle möglichen Eingaben gleichmäßig auf Buckets
  - Leicht zu berechnen



Hash Funktion aus Hilfsbibliothek

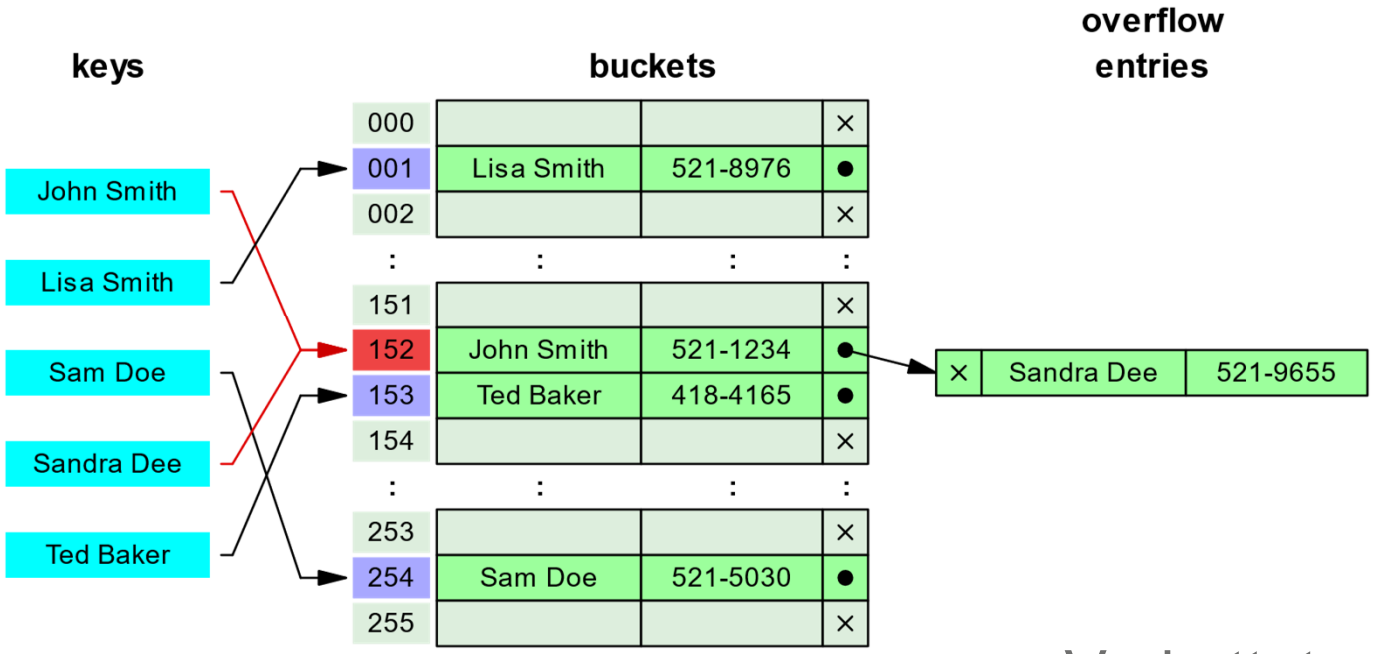
Technische Hochschule Rosenheim
Technical University of Applied Sciences

- In der Praxis: Verwendung von Hilfsbibliothek
- Z.B. `org.apache.commons.lang3.builder.HashCodeBuilder`

```
import org.apache.commons.lang3.builder.HashCodeBuilder;

public class MeineKlasse implements Comparable<MeineKlasse> {
    int attribut;
    String s;
…
    public int hashCode() {
        // wähle zwei beliebige ungerade Zahlen
        HashCodeBuilder b = new HashCodeBuilder(17, 19);

        // füge alle wichtigen Elemente an
        b.append(attribut).append(s);

        return b.hashCode();
    }
}
```

Kollisionen

- Indizes aus Hash können *kollidieren*
- Verwende eine Liste für ein Bucket statt einzelner Elemente



Verkettete Liste

Key Value

Implementierung einer HashMap mittels eines Array Index (Skizziert)

Technische Hochschule Rosenheim
Technical University of Applied Sciences

- Wie bildet man nun einen Hashwert eines Schlüssels auf einen Arrayindex ab? Immerhin ist der Wertebereich von `int` durchaus groß -- zum einen würde der Speicher für ein solch großes Array nicht reichen, zum anderen wäre dieses dann vermutlich überwiegend leer.

- Man behilft sich anders: Man wählt zunächst Arraygrößen aus, welche eine Zweierpotenz darstellen, also z.B. 4, 8, 32, 256 usw. Es gibt also entsprechend Arrayindizes von 0..3, 0..8, 0..31, usw. Ein mathematischer Kniff (und das Wissen um das Binärsystem) helfen hier: Der Index im Array wird berechnet als `(array.length - 1) & key.hashCode()`, wobei & der Bitweise UND Operator ist.

- Ein Beispiel: Ein Objekt habe den hashCode von 42, das Array habe eine Länge von 16:

- Falls Kollisionen auftreten, kann pro Array-Eintrag eine `List` verwendet werden.

```
Person[] array = new Person[16];
Person p = new Person("Klaus");
int hash = p.hashCode();  // z.B. 42

int index = (array.length - 1) & hash;

// array.length - 1 in binary:  001111
// hash             in binary:  101010
// 15 & 42       bitweise UND:  001010
//                       ...zwischen 0..16

System.out.println(index);
```

Beispiel HashMap

Technische Hochschule Rosenheim
Technical University of Applied Sciences

Person Klaus
Name="Klaus"
Einkommen=200
hashCode=72578462
hashCode als Binärzahl=10001010011011101011001111**10**

`buckets.length()-1` als Binärzahl **11**

`buckets.length()-1 & „Klaus".hashCode` als Binärzahl **10**

→ *Klaus kommt in Bucket No.2*

Generic Array creation

```
// Array von Generics
@SuppressWarnings("unchecked")
List<Entry<String, Integer>>[] buckets = new List[4];
for(int i=0;i<4;i++)
    buckets[i]=new ListImpl<Entry<String, Integer>>();
```

```
// Hinzufügen von Klaus
buckets[2].add(new Entry<String,Integer>("Klaus",200));
```

0	*null*
1	*null*
2	(Maria,400)→(Klaus,200)→*null*
3	*null*

Implementierung mittels `TreeMap`

Technische Hochschule Rosenheim
Technical University of Applied Sciences

- Eine Hash-Map ist also wieder eine Datenstrukturen mit Vor- und Nachteilen.
  - Buckets können zügig gefunden werden über Hashing
  - Hinzufügen von Elementen leicht
  - Suchen von Elementen vereinfacht, wenn das Hashing gut funktioniert
  - Löschen leicht (Auffinden und überspringen)

- Implementierung als TreeMap ebenfalls möglich.
  - Auffinden von Elementen noch besser

```
class Entry<K extends Comparable<K>, V> implements Map.Entry<K, V>, Comparable<Entry<K, V>> {
    K key;
    V value;

    Entry<K, V> left, right;

…
}
```

  - *Wieso eigentlich keine Buckets mit Trees statt Listen anlegen?*

Für eine HashMap haben wir `hashCode` und `equals` benötigt zum Einfügen und auffinden. Jetzt brauchen wir zusätzlich `compareTo`, damit wir für unseren Baum vergleichen können.

Objektvergleich mittels `compareTo`

Technische Hochschule Rosenheim
Technical University of Applied Sciences

- Für die Verwendung in einer TreeMap, muss ein Entry über seinen Key vergleichbar sein.

```
public class MeineKlasse implements Comparable<MeineKlasse> {
    int attribut;
…
    public int compareTo(MeineKlasse other) {
        // 0 bei Gleichheit, negativ wenn kleiner als other
        if (this.attribut == other.attribut)
            return 0;
        else if (this.attribut < other.attribut)
            return -1;
        else
            return 1;

        // `alternativ sehr viel kuerzer:`
        // return this.attribut - other.attribut;
    }
}
```

Ansonsten wieder wie gehabt, nur diesmal als Key/Value

Technische Hochschule Rosenheim
Technical University of Applied Sciences

Keine Wurzel, Baum ist leer

Den Key gibt es schon, wir überschreiben den value

Der einzufügende Key ist „kleiner“ als der aktuelle Knoten

→ einfügen

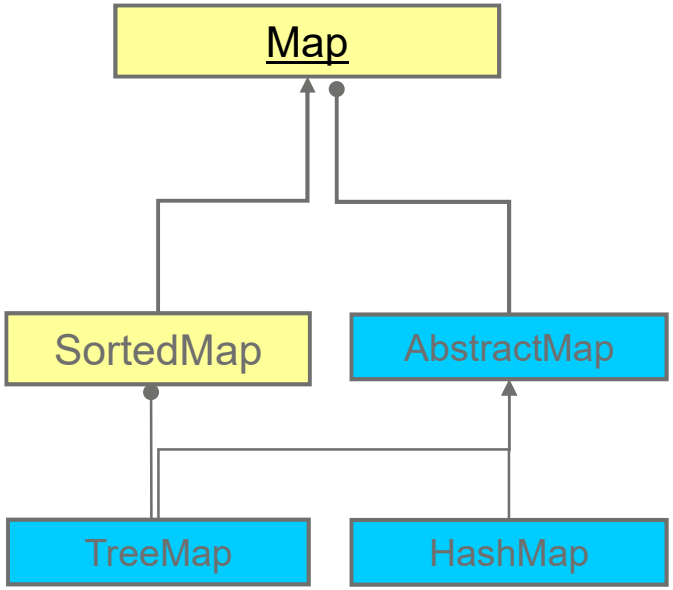
→ Oder links lang

Sonst rechts lang

```
@Override
public void put(K key, V value) {
    if (root == null) {
        root = new Entry<>(key, value);
        return;
    }

    Entry<K, V> it = root;
    while (it != null) {
        int c = it.key.compareTo(key);
        if (c == 0) {
            // update!
            it.value = value;
            return;
        } else if (c < 0) {
            if (it.left == null) {
                it.left = new Entry<>(key, value);
                return;
            } else {
                it = it.left;
            }
        } else {
            if (it.right == null) {
                it.right = new Entry<>(key, value);
                return;
            } else {
                it = it.right;
            }
        }
    }
}
```

Assoziative Speicher: `Interface <Map>`

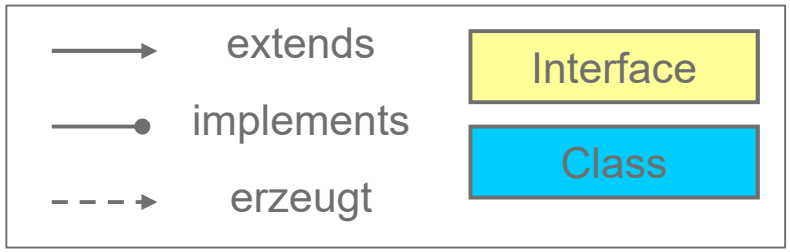
Technische Hochschule Rosenheim
Technical University of Applied Sciences



Listen und Mengen speichern Elemente, assoziative Speicher dagegen (Key-Value)-Paare!

Implementierung über einen Binärbaum

Implementierung über ein Hash-Verfahren



https://docs.oracle.com/javase/8/docs/api/java/util/Map.html

Java Collection Framework

Technische Hochschule Rosenheim
Technical University of Applied Sciences

- **`List`** als sequenzielle Datenstruktur
- **`Set`** als duplikatfreie (ungeordnete) Datenstruktur
- `List` und `Set` erweitern **`Collection`**, was wiederum **`Iterable`** erweitert, d.h. alle sind iterierbar via **`Iterator`**
- **`Map`** als assoziative Datenstruktur
- Realisierungen in der Java API:
  - **`ArrayList`** und **`LinkedList`**
  - **`TreeSet`** und **`HashSet`**
  - **`TreeMap`** und **`HashMap`**

Zusammenfassung

Technische Hochschule Rosenheim
Technical University of Applied Sciences

- *Die grundlegenden Datenstrukturen in der Informatik sind*
  - `List` ist eine sequenzielle Datenstruktur, realisiert z.B. als `ArrayList` oder `LinkedList`
  - `Set` ist eine duplikatfreie (ungeordnete) Menge, realisiert z.B. als `HashSet` oder `TreeSet`
  - `Map` ist eine assoziative Datenstruktur, welche Schlüssel auf Werte abbildet, realisiert z.B. als `HashMap` oder `TreeMap`

- *Beim Programmieren:*
  - definieren Sie Variablen als Schnittstellen
  - initialisieren Sie die Variablen von Klassen der Java API
  - z.B. `Set<String> s = new TreeSet<>()`
  - Vermeiden Sie die Verwendung von raw types (unparametrisierten generischen Klassen), verwenden Sie also z.B. immer `List<...>` statt `List`.

- *Werden Datenstrukturen mit eigenen Klassen verwendet, so sollte unbedingt*
- `equals` zur Prüfung auf Wertgleichheit implementiert werden
- `hashCode` implementiert Werden, sofern Hashing verwendet wird
- `Comarable<T>` implementiert werden, sofern Objekte vergleichbar sein sollen.
- Collections sind `Iterable`, man kann diese also in `for-each` Schleifen verwenden, oder einen `Iterator` zur Traversierung erhalten.