Technische Hochschule Rosenheim
Technical University of Applied Sciences

Objektorientierte Programmierung
Kapitel 7 – Collections

Prof. Dr. Kai Höfig

Motivation

Technische Hochschule Rosenheim
Technical University of Applied Sciences

"Reusability is one of the great promises of object-oriented technology. Unfortunately, it's a promise that often goes unrealized. The problem is that reuse isn't free; it isn't something you get simply because you're using object-oriented development tools. Instead, it's something you must work hard at if you want to be successful."
(Scott Ambler)

- **Generischer, wiederverwertbarer Code** durch *Schnittstellen, Spezialisierung, Polymorphie*, etc.
  - Konflikt: Flexibilität vs. Typsicherheit.
  - Typüberprüfung teils erst zur Laufzeit.

- Welche generischen Datenstrukturen und Algorithmen bietet die **Java Standard Library** bereits?

Inhalt

Technische Hochschule Rosenheim
Technical University of Applied Sciences

- Einführung: Java Generics
  - Details in "Programmieren 3", Fokus: Generics aus Sicht des Nutzers
- Einführung: Java Collection API
- Listen
- Mengen / Sets
- Assoziative Speicher / Maps
- Schnittstellen `Iterable` und `Iterator`
- Algorithmen mit Collections

Ohne Generics ("Raw Types")

Technische Hochschule Rosenheim
Technical University of Applied Sciences

- **Ziel:**
  - Generische Klasse Bag, die beliebiges Objekt als Inhalt aufnehmen kann.
- **Versuch:**

```
public class Bag {
 private Object content;
 public Bag(Object content) {
     this.content = content;}
 public Object getContent() {
     return content; }
 public void setContent(Object c){
     this.content = c;}
}
```

Verwendung der Klasse Bag**:**

```
Long bigNumber = 1111111111L;
Bag b1 = new Bag(bigNumber);
Bag b2 = new Bag("Hallo");

// later on
Long val =  (Long) b1.getContent();
String s =  (String) b2.getContent();
```

Kein Compiler-error, wenn b2 ein Long enthält!

- **Mögliche Verbesserungen**
  - Teile Compiler beim Initialisieren mit für welchen Inhaltstyp die Instanz von Bag verwendet werden soll.
  - Der Compiler kann dann überwachen, dass wirklich nur der gewünschte Inhaltstyp hinzugefügt wird.
  - Beim Entnehmen kann man sich sicher sein, dass der gewünschte Datentyp in der Bag liegt.

Generische Klassen

Technische Hochschule Rosenheim
Technical University of Applied Sciences

- **Deklaration** eines *generischen Typs T* für eine Klasse
  - "Parametrisierung eines Datentyps"
  - Ersetze `Object` stets durch `T`

- **Verwenden** eines generischen Datentyps
  - Es entstehen 2 *parametrisierte Typen* mit den *Typparametern* Long und String.
  - Kein Typecast notwendig!

```
public class Bag<T> {
  private T content;
  public Bag(T content) {
       this.content = content;
  };
  public T getContent() {
       return content;
  }
  public void setContent(T c) {
       this.content = c;}
}
```

```
Long bigNumber = 1111111111L;
Bag<Long> b1 = new Bag<Long>(bigNumber);
Bag<String> b2 = new Bag<String>("Hallo");

// later on
Long val = b1.getContent();
String s = b2.getContent();
```

Hinweis für Compiler, hier wird quasi der Platzhalter T mit einem Typ belegt, der ab dann fest ist.

Inhalt

Technische Hochschule Rosenheim
Technical University of Applied Sciences

- Einführung: Java Generics
  - Details in "Programmieren 3", Fokus: Generics aus Sicht des Nutzers
- Einführung: Java Collection API
- Listen
- Mengen / Sets
- Assoziative Speicher / Maps
- Schnittstellen `Iterable` und `Iterator`
- Algorithmen mit Collections

Warum nicht immer ein Array verwenden?

Technische Hochschule Rosenheim
Technical University of Applied Sciences

- Array = Effektive Art, Referenzen auf Objekte zu sammeln:
  - Sequentielle Aneinanderreihung im Speicher

- Vorteile
  - Kann primitive Typen direkt enthalten.
  - Direkter Zugriff auf Einzelelemente über Index.
  - Länge fest, Abfrage der Länge möglich

- Nachteile
  - Hat eine **unveränderliche Länge**.
  - Löschen von Elementen **hinterlässt Lücken**.

Java Container / Collections bieten weitere Implementierungen, z.B. verkettete Listen (siehe Übungsblatt05)

Datenstrukturen in Java

Technische Hochschule Rosenheim
Technical University of Applied Sciences

- Java Collection API
  - Sammlung wichtiger, häufig verwendeter Datenstrukturen.
  - Alle speichern Objekte anderer Referenztypen als Elemente (keine primitiven Typen)
  - Struktur / Gliederung der verschiedenen Klassen: Musterbeispiel für Einsatz von Vererbung, Schnittstellen und Generics!

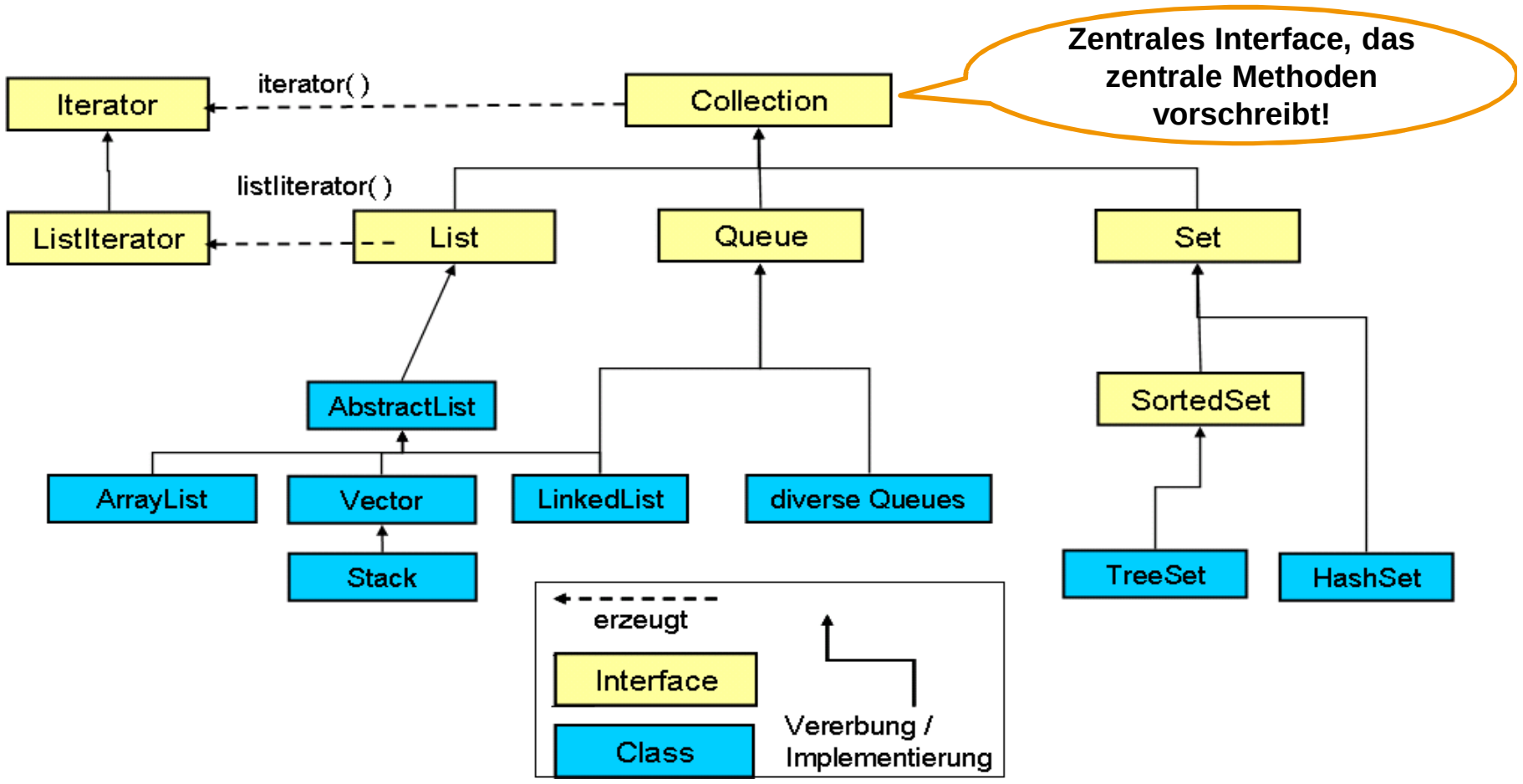
- Java Collection API gliedert sich in 2 Teile

  - **Collection-Datenstrukturen:**
    - Speichert ***Einzelelemente***
    - Alle Klassen aus dieser Gruppe implementieren Interface **`Collection`**.

  - **Map-Datenstrukturen:**
    - Speichert ***(Key, Value)-Paare***
    - Alle Klassen aus dieser Gruppe implementieren das Interface **`Map`**.

Speichern einzelner Elemente: `Collection`

Technische Hochschule Rosenheim
Technical University of Applied Sciences



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.

Listen: Interface `List`

Technische Hochschule Rosenheim
Technical University of Applied Sciences

- Datenstruktur Liste:
  - Sequenz von Daten, bei der die Elemente eine feste Reihenfolge besitzen.
  - Duplikate sind erlaubt.

- Interface `List`:
  - Erweiterung von `Collection`: Schreibt **zusätzliche** Methoden für Liste vor.
  - Vor allem Methoden mit Bezug zur Position eines Elements, z.B. `get(i)`.
  - Es existieren verschiedene Implementierungen des Interface.

- Konkrete Unterklassen
  - `ArrayList`
    - Implementierung basierend auf Arrays.
    - Schneller Zugriff auf beliebige Positionen.
    - Bei hoher Anzahl von Elementen wird automatisch ein größeres Arrays angelegt.
  - `LinkedList`
    - Implementierung basierend auf (doppelt) verketteter Liste.
    - Schnelles Einfügen und Entfernen von Elementen "in der Mitte".
    - Kann beliebig wachsen.

Interface List<E> extends Collection<E> (1)

Technische Hochschule Rosenheim
Technical University of Applied Sciences

void **add(int index, E element)**
Inserts the specified element at the specified position in this list (optional operation).

boolean **addAll(int index, Collection<? extends E> c)**
Inserts all of the elements in the specified collection into this list at the specified position (optional operation).

T **get(int index)**
Returns the element at the specified position in this list.

int **indexOf(Object o)**
Returns the index in this list of the first occurrence of the specified element, or -1 if this list does not contain this element.

int **lastIndexOf(Object o)**
Returns the index in this list of the last occurrence of the specified element, or -1 if this list does not contain this element.

Interface List<E> extends Collection<E> (2)

Technische Hochschule Rosenheim
Technical University of Applied Sciences

E **remove(int index)**
Removes the element at the specified position in this list (optional operation).

E **set(int index, T element)**
Replaces the element at the specified position in this list with the specified element (optional operation).

List<E> **subList(int fromIndex, int toIndex)**
Returns a view of the portion of this list between the specified fromIndex, inclusive, and toIndex, exclusive.

ListIterator<E> **listIterator()**
Returns a list iterator of the elements in this list (in proper sequence).

ListIterator<E> **listIterator(int index)**
Returns a list iterator of the elements in this list (in proper sequence), starting at the specified position in this list.

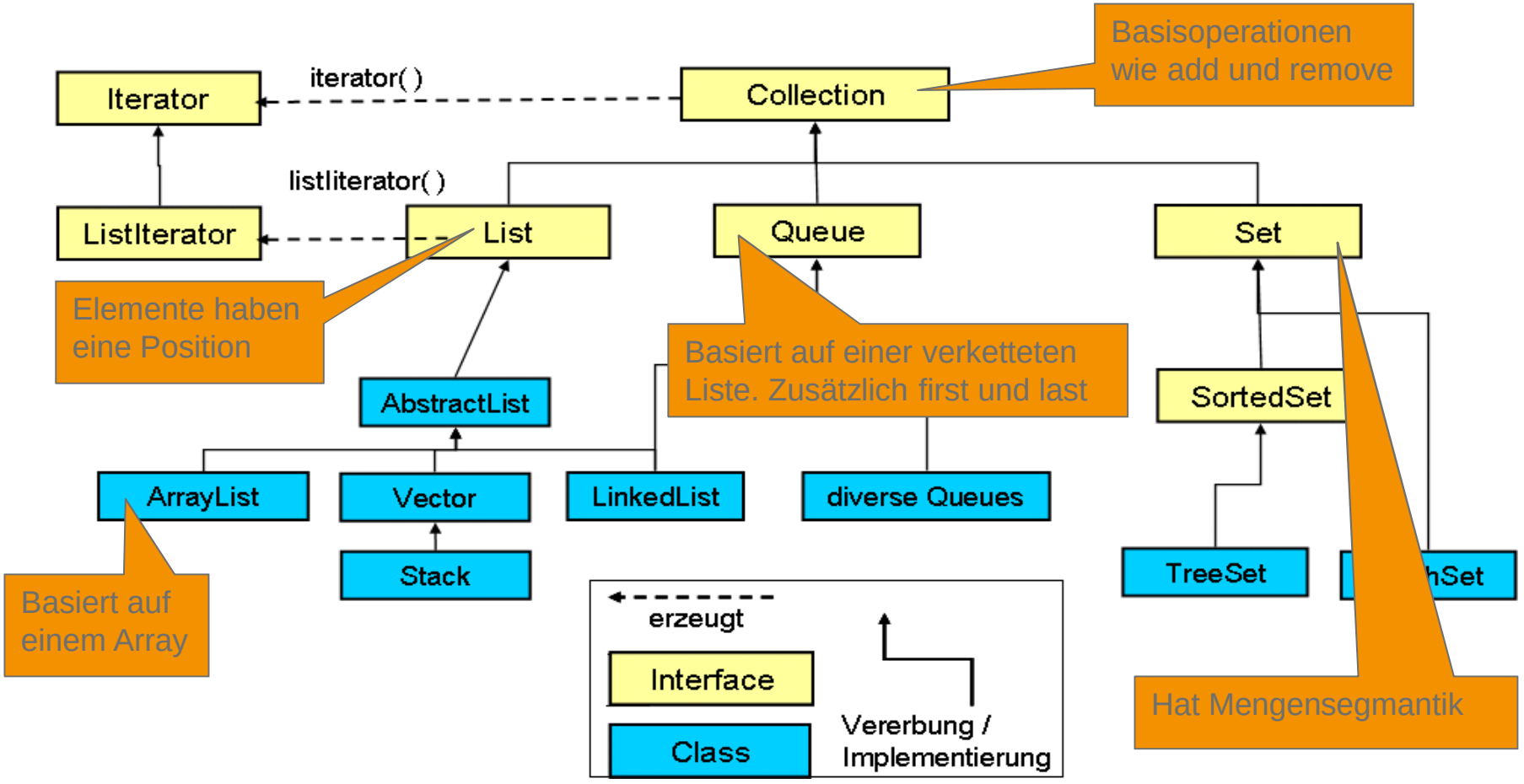
Beispiel: ArrayList

Technische Hochschule Rosenheim
Technical University of Applied Sciences

```java
// create empty list
List<Person> list = new ArrayList<>();

// add a person
list.add(new Person("Maier", 33));

// add a player (subclass of person)
list.add(new Player("Müller", 28, 4, 2, 5));

// is person stored in list?
Person p = new Person("Maier", 33);
boolean b1 = list.contains(p);

// index where "Maier" is stored?
int index = list.indexOf(p);

// get object at start of list and remove it
Person p1 = list.get(0);
list.remove(0);

// convert into array
Person[] persons = list.toArray(new Person[list.size()]);
```

Speichern einzelner Elemente: `Collection`

Technische Hochschule Rosenheim
Technical University of Applied Sciences



Listen: Interface `Set`

Technische Hochschule Rosenheim
Technical University of Applied Sciences

- **Gemeinsamkeiten** zu Listen
  - Mengen und Listen speichern beliebige Elemente (→Generics).
  - Mengen und Listen implementieren *alle* Methoden der Schnittstelle `Collection<T>`, fügen aber teilweise eigene Methoden hinzu.

- **Unterschiede** zu Listen
  - Jedes Element darf nur **einmal** vorkommen (**keine Duplikate!)**
  - Anpassung, um die Restriktion bzgl. Duplikaten umzusetzen.
  - Bsp: `boolean add(Collection E e)` fügt e nur zur Menge hinzu, falls es noch nicht vorhanden ist. Ansonsten Rückgabe von `false`.

- **Frage:** Wann gelten 2 Elemente/Objekte als gleich?
  - Wenn Ergebnis von `equals(..) true` ist → **Überschreiben!**

- Methoden, siehe offizielle Java Dokumentation:
  - `interface Set<E> extends Collection<E>`
  - https://docs.oracle.com/javase/8/docs/api/java/util/Set.html

Set Implementierungen

Technische Hochschule Rosenheim
Technical University of Applied Sciences

- **HashSet**
  - Implementierung verwendet Hashtabelle
  - keine Ordnung der Elemente
  - schneller beim Einfügen u. Löschen
    - **add**, **remove** und **contains** benötigt **im Schnitt konstante Zeit**

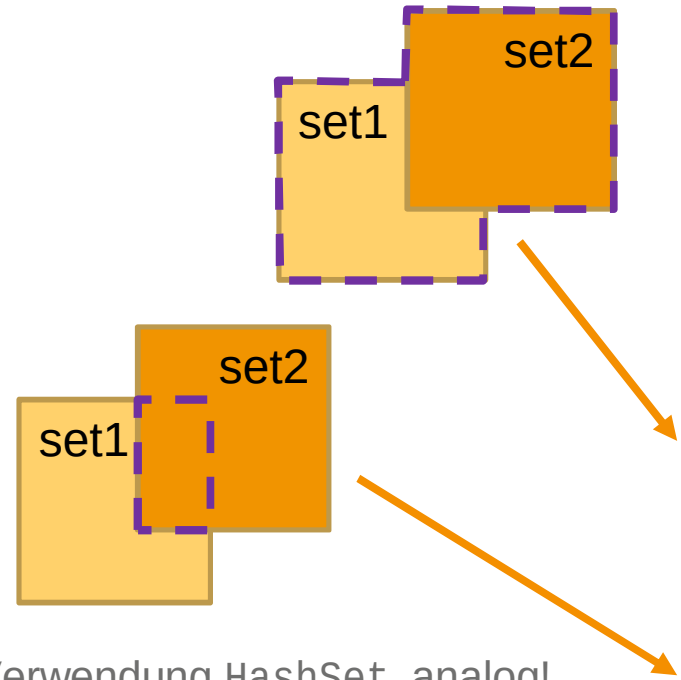
- **TreeSet**
  - Implementierung verwendet balancierten Baum
  - garantiert Ordnung der Elemente
  - **add**-, **remove**- und **contains**-Methoden
    - Benötigen höchstens **logarithmisch viel Zeit** im Verhältnis zur Anzahl der Elemente.

- Details: Siehe "Algorithmen und Datenstrukturen"

Beispiel `TreeSet`: Vereinigungs- und Schnittmenge

Technische Hochschule Rosenheim
Technical University of Applied Sciences

- `ClassCastException` falls Klasse `Person` nicht `Comparable<Person>` implementiert.




```
// generate 3 persons
Person p1 = new Person("Merkel", 61);
Person p2 = new Person("Gabriel", 56);
Person p3 = new Person("Seehofer", 66);

// generate 2 sets
TreeSet<Person> set1 = new TreeSet<Person>();
set1.add(p1);
set1.add(p2);
TreeSet<Person> set2 = new TreeSet<Person>();
set2.add(p2);
set2.add(p3);

// Vereinigungsmenge / union
TreeSet<Person> union = new TreeSet<Person>();
union.addAll(set1);
union.addAll(set2);

// Schnittmenge / intersection
TreeSet<Person> intersection = new
TreeSet<Person>();
intersection.addAll(set1);
intersection.retainAll(set2);
```

- Verwendung `HashSet` analog!
  - Empfehlung: Neben `equals(.)` auch `hashCode()` implementieren.

Interface `Set<E>`: Weitere Details

Technische Hochschule Rosenheim
Technical University of Applied Sciences

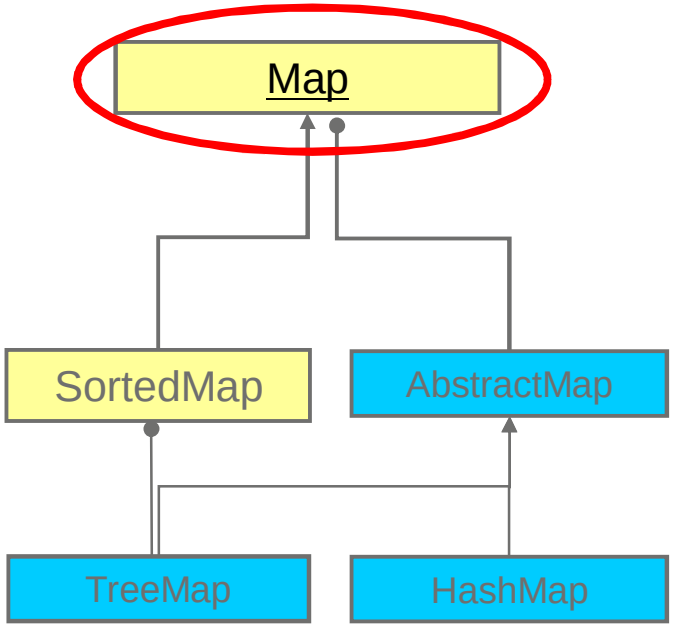
- `TreeSet` sortiert Elemente bzgl. ihrer Ordnung
  - Ordnung der Elemente **entweder** durch `Comparable<E>` **oder** `Comparator<E>`.
  - `HashSet<E>` macht das nicht.
  - Sollen die Elemente also in einer bestimmten Reihenfolge durchlaufen werden, ist ein TreeSet vorzuziehen.

- `TreeSet<E>` implementiert weitere Schnittstellen
  - Implementiert z.B. `first(), last()` des Interface `SortedSet<E>` sowie die Möglichkeit über alle Elemente in einer korrekten Ordnung zu iterieren.

- Es gibt weitere "Set"-Implementierungen.
  - Bsp: `LinkedHashSet<E>` ist wie `TreeSet<E>`, behält aber die Einfüge-Reihenfolge bei.

- Dokumentation
  - https://docs.oracle.com/javase/8/docs/api/

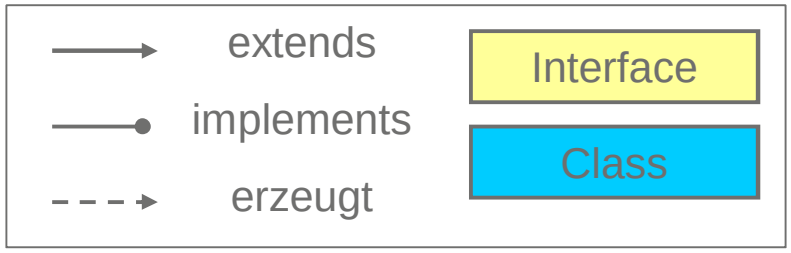
Assoziative Speicher: `Interface <Map>`



Implementierung über
einen Binärbaum

Implementierung über
ein Hash-Verfahren

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



Assoziative Speicher: `Interface Map<K,V>`

Technische Hochschule Rosenheim
Technical University of Applied Sciences

- Key-Value Paar
  - Jedes Element der Datenstruktur besteht aus einem *Schlüssel / Key* und einem *Wert / Value*.
  - Annahme: Schlüssel sind eindeutig!
  - Beispiele für Key-Value-Paare in der Praxis?

- Verallgemeinerung von Arrays
  - Key beim Array ist die "Einfügeposition"

- `Interface<Map>` hat 2 generische Typen
  - Typ der Schlüssel (K)
  - Typ der Werte (V)

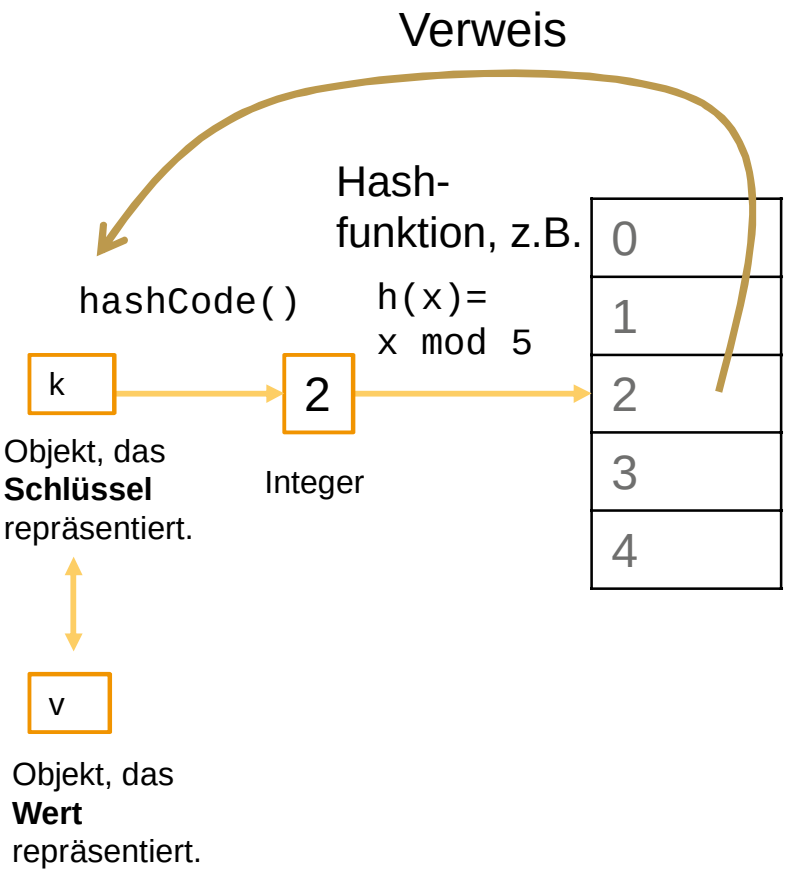
- Methoden, siehe offizielle Java Dokumentation:
  - https://docs.oracle.com/javase/8/docs/api/java/util/Map.html

Implementierung eines assoziativen Speichers

- Wichtigste Implementierungen von `Map` sind die konkreten Klassen `HashMap<E>` und `TreeMap<E>`

- `HashMap<K,V>`
  - Basiert wie `HashSet<E>` auf einer Hashtabelle.
  - `HashSet<E>` ist ein `HashMap<K,V>` mit `K=E`. Der Wert `V` wird nicht weiter verwendet.
  - Wichtig: Implementiere `hashCode()` und `equals()`!

- `TreeMap<K,V>`
  - Basiert wie `TreeSet<E>` auf einem binären Suchbaum.
  - `TreeSet<E>` ist eine `TreeMap<K,V>` mit `K=E`. Der Wert `V` wird nicht weiter verwendet.
  - Wichtig: Implementiere Interface `Comparable<E>` bzw. `Comparator<E>`!



HashMap: Funktionsweise

Beispiel HashMap

Technische Hochschule Rosenheim
Technical University of Applied Sciences

- Vorgesetzter eines Politikers
  - **Schlüssel**: Klasse `Person`
  - **Wert**: Ebenfalls Klasse `Person`. Objekt, das den Vorgesetzten repräsentiert.
  - Bsp.: "Merkel" ist Vorsitzender von "Gabriel"

- Hinweis
  - Schlüssel und Wert werden meist durch verschiedene Klassen repräsentiert.

```
Person p1 = new Person("Merkel", 61);
Person p2 = new Person("Gabriel", 56);
Person p3 = new Person("Seehofer", 66);

// generate map
Map<Person, Person> map =
          new HashMap<Person, Person>();

// adding 2 entries
map.put(p2, p1);
map.put(p3, p1);

// Wer ist Vorgesetzter von p2?
Person vorgesetzter = map.get(p2);

// Gibt es einen Eintrag/Schlüssel für p3?
boolean b1 = map.containsKey(p3);

// get all keys
Set<Person> keys = map.keySet();

// get all values
Collection<Person> values = map.values();
```

`Interface Map<K,V>`: **Weitere Details**

Technische Hochschule Rosenheim
Technical University of Applied Sciences

- `Map.Entry<K,V>`
  - "Spezielle" innere Klasse, siehe Programmieren3
  - Jedes Objekt repräsentiert ein Key-Value-Paar.

- Schlüssel / Keys müssen immer **immutable** sein!
  - Collections / Datenstrukturen setzen das implizit voraus.
  - Ändert man Schlüssel (*mutable Objekte*), so wird unter Umständen interne Ordnung der Datenstruktur zerstört!
    - Beispiel: Das Ergebnis von `hashCode()` ändert sich.

```
Map<Person, Person> map = new HashMap<Person, Person>();
map.put(p2, p1);
p2.setName("Müller");
boolean b2 = map.containsKey(p2); ??
```

Inhalt

Technische Hochschule Rosenheim
Technical University of Applied Sciences

- Einführung: Java Generics
  - Details in "Programmieren 3", Fokus: Generics aus Sicht des Nutzers
- Einführung: Java Collection API
- Listen
- Mengen / Sets
- Assoziative Speicher / Maps
- Schnittstellen `Iterable` und `Iterator`
- Algorithmen mit Collections

Durchlaufen von Collections mittels `for`-Schleife

Technische Hochschule Rosenheim
Technical University of Applied Sciences

- Bei Arrays gibt es eine "`foreach`"-Schleife

```
int[] arr = {1, 2, 5, 7};
for (int val : arr) {
    System.out.println(val);
}
```

- **Jede** `Collection`, **jede** `Map`, **jede** Datenstruktur, die Interface `Iterable` implementiert, kann ebenfalls mit "`foreach`"-Schleife durchlaufen werden.
  - Interface `Iterable` schreibt nur die Methode `iterator()` vor.

```
Map<String, Integer> map = new HashMap<String, Integer>();

for (Map.Entry<String,Integer> e : map.entrySet()) {
    String key = e.getKey();
    Integer val = e.getValue();
}
```

Iterieren über alle Daten mit `Iterator`

Technische Hochschule Rosenheim
Technical University of Applied Sciences

- `List<String>`

```
List<String> list = new ArrayList<String>();
Iterator<String> iter = list.iterator();
while (iter.hasNext()) {
    String temp = iter.next(); // do s.th
}
```

Liefert Iteratorobjekt für das erste Element

- `Set<String>`

```
Set<String> set = new TreeSet<String>();
Iterator<String> iter = set.iterator();
while (iter.hasNext()) {
    String temp = iter.next(); // do s.th
}
```

- **Vorteil** eines Iterators:
  - Identisches Vorgehen für alle Datenstrukturen!
  - Funktioniert für alle Klassen, die `Collection` implementieren!

Iterieren über `Map`: for-Schleife

Technische Hochschule Rosenheim
Technical University of Applied Sciences

- `Map` implementiert **nicht** die Schnittstelle `Collection`
  - Problem: `Map` hat keine Methode `iterator()`!
- `Map` enthält anstelle von Elementen Key-Value-Paare

- **Lösung**: Umwandeln in Menge von
  - **Schlüsseln**: `keySet(..)` liefert Menge aller Schlüssel.

```
Map<String, Integer> map = new HashMap<String, Integer>();
Set<String> set = map.keySet();
Iterator<String> iter = set.iterator(); // über key
while (iter.hasNext()) {
     String key  = ??
     Integer val = ??
}
```

Benötigt man nur die Werte, kann man auch mit `values()` eine `Collection` der Werte bekommen.

  - **Key-Value-Paaren**: `entrySet(..)` liefert Menge aller Key/Values

```
Map<String, Integer> map = new HashMap<String, Integer>();
Set<Map.Entry<String,Integer>> entries = map.entrySet();
Iterator<Map.Entry<String,Integer>> iter = entries.iterator();
while (iter.hasNext()) {
    Map.Entry<String, Integer> entry = iter.next();
    String key = entry.getKey();
    Integer val = entry.getValue();
}
```

`Map.Entry`: Innere Klasse, siehe Prg3!

Inhalt

Technische Hochschule Rosenheim
Technical University of Applied Sciences

- Einführung: Java Generics
  - Details in "Programmieren 3", Fokus: Generics aus Sicht des Nutzers
- Einführung: Java Collection API
- Listen
- Mengen / Sets
- Assoziative Speicher / Maps
- Schnittstellen `Iterable` und `Iterator`
- Algorithmen mit Collections

Sortieren von Listen

Technische Hochschule Rosenheim
Technical University of Applied Sciences

- Wie sortiert man ***beliebige*** Listen?
  - Algorithmus soll für `LinkedList`, `ArrayList`, etc. funktionieren, **egal** welches *Element* gespeichert wird!

- Zutaten
  - **Paarweises Vergleichen** von 2 Elementen → `Comparable` bzw. `Comparator`
  - **Typvariablen / Generics**: Funktionen arbeiten soweit als möglich auf beliebigen Typen
    - Alternativ: Alles muss auf Objekten der Klasse `Object` funktionieren ("Raw Types")
  - Geschicktes Programmieren **gegen allgemeine Interfaces** wie `Iterator` oder `Collection`.

- Es wird exemplarisch der Algorithmus ***Quicksort*** betrachtet:
  - *Hinweis:* Es geht nicht um Algorithmen per se, sondern wie man OOP einsetzen kann, um generische Algorithmen zu entwickeln.

QuickSort: Rekursive Methode `quickSort(..)`

Technische Hochschule Rosenheim
Technical University of Applied Sciences

```java
/**
 * Das ist der QuickSort
 *
 * @param list  zu sortierende Liste
 * @param cmp   Vergleichsfunktion (Comparator)
 * @return      sortierte Liste
 */
public static <T> List<T> quickSort(List<T> list, Comparator<T> cmp) {
    if (list.size() <= 1)
        return list;

    List<T> l1 = new ArrayList<T>();
    List<T> l2 = new ArrayList<T>();

    partition(list, l1, l2, cmp); // teilen mit vergleichen

    l1 = quickSort(l1, cmp);      // Teile rekursiv sortieren
    l2 = quickSort(l2, cmp);

    List<T> result = union(l1, l2);  // sortierte Teile aneinanderhängen

    return result;
}
```

QuickSort: Methode `partition(..)`

Technische Hochschule Rosenheim
Technical University of Applied Sciences

```java
/**
 * partition teilt die Liste mit vergleichen
 * @param list  zu teilende Liste
 * @param l1    Teilliste mit kleineren Elementen / vorher leer
 * @param l2    Teilliste mit größeren Elementen / vorher leer
 */
public static <T> void partition(List<T> list, List<T> l1,
                                 List<T> l2, Comparator<T> cmp) {
    if (list.size() == 0 ) return;
    Iterator<T> i = list.iterator();
    T first = i.next();  // Vergleichsobjekt

    while (i.hasNext()) {
        T element = i.next();
        if (cmp.compare(element, first) < 0)
            l1.add(element);
        else
            l2.add(element);
    }
    // erstes Element an kürzere Liste anhängen
    if (l1.size() < l2.size())
        l1.add(first);
    else
        l2.add(first);
}
```

QuickSort: Methode `union(..)` und Testmethode

Technische Hochschule Rosenheim
Technical University of Applied Sciences

```java
/**
 * union vereint zwei (sortierte) Listen durch Aneinanderhängen
 * @param l1     Teilliste mit kleineren Elementen
 * @param l2     Teilliste mit größeren Elementen
 * @return       vereinte (sortierte) Liste
 */
public static <T> List<T> union(List<T> l1, List<T> l2) {
    List<T> result = new ArrayList<T>();
    result.addAll(l1);
    result.addAll(l2);
    return result;
}
// Test
public static void main(String[] args) {
    List<Integer> unsortedList = new ArrayList<Integer>();
    unsortedList.addAll(Arrays.asList( new Integer[] {3, 10, 7, 1, 6, 2, 5, 8}));
    System.out.println("Unsortiert:" + unsortedList.toString());

    List<Integer> sortedList = quickSort(unsortedList, new IntegerComparator());
    System.out.println("Sortiert:" + sortedList.toString());
}
```