Lektion 20

Collections

Comparable

Assoziative Arrays (Map)

Generics (Unbounded, Upper bounded Wildcards)

Collections

Sammlungen

Wie
können mehrere
Objekte gleichen Typs
im Speicher gehalten
werden?

Wir haben bereits gesehen, dass Java für Listen eine eigene generische Klasse zur Verfügung stellt:

```
ArrayList<String> list = new ArrayList<>();
```

Dieser Liste liegt ein Array zugrunde.
Die Java Klassenbibliothek kümmert sich darum,
das zugrundeliegende Array beim Wachsen der Liste zu vergrößern.

```
public class Raum
{
  final int KAPAZITAET = 50;
  ArrayList<Student> studenten = new ArrayList<>();

  public void betrete(Student s)
  {
    if (!studenten.contains(s) && studenten.size() < KAPAZITAET)
      studenten.add(s);
  }

  public void verlasse(Student s)
  {
    studenten.remove(s);
  }




}
```

Überprüft, ob Element in Liste enthalten ist

Hängt das Element an das Ende der Liste

Entfernt das Objekt aus der Liste

Neben der ArrayList gibt es eine echte doppelt verkettete Liste namens LinkedList:

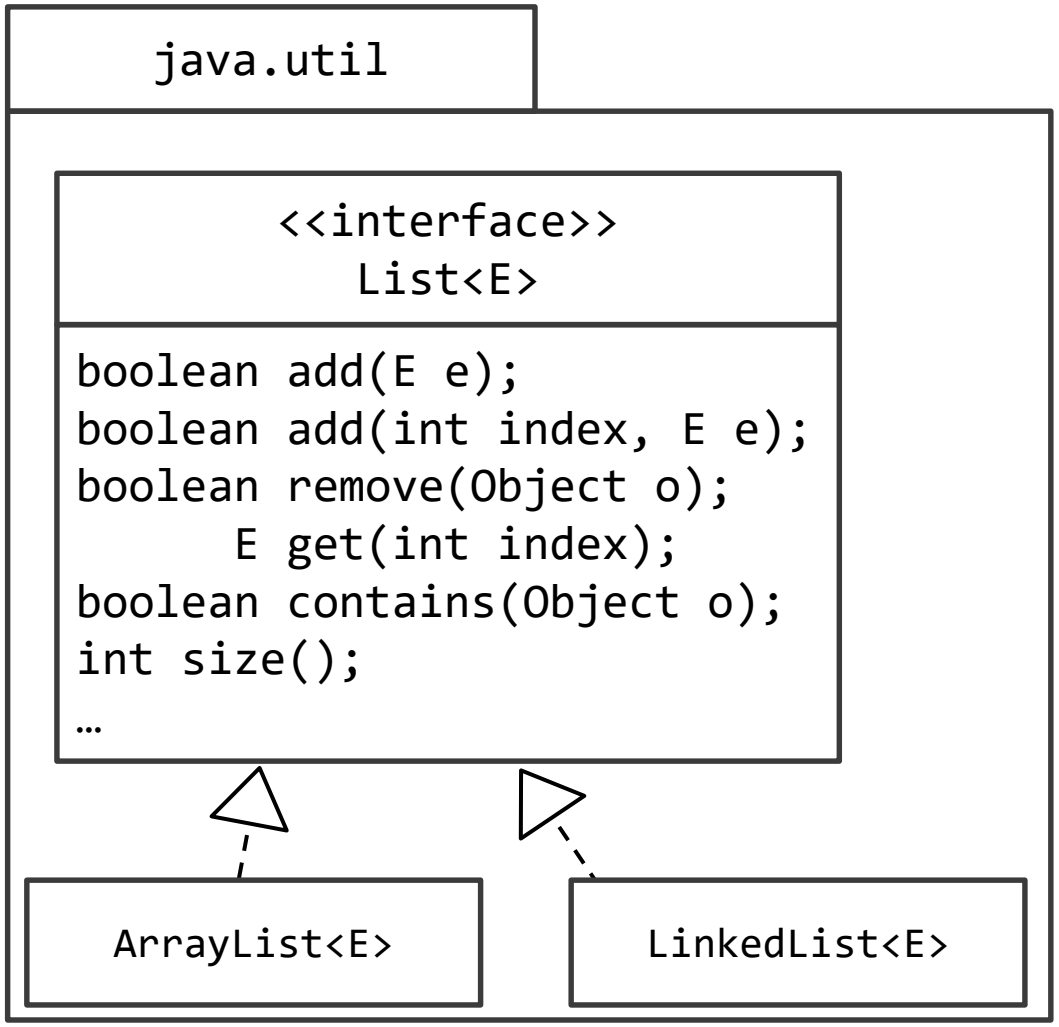
```
LinkedList<String> list = new LinkedList<>();
```

Beide Listenimplementierungen sind für unterschiedliche Szenarien unterschiedlich effizient.

Sie stellen beide einen Standardsatz an Methoden zum Hinzufügen und Entfernen von Elementen zur Verfügung.

Die gemeinsamen Methoden sind in dem List-Interface definiert.

```
List<String> list = new LinkedList<>();

List<String> list = new ArrayList<>();
```



```
public class Raum
{
  final int KAPAZITAET = 50;
  List<Student> studenten = new LinkedList<>();


  public void betrete(Student s)
  {
    if (!studenten.contains(s) && studenten.size() < KAPAZITAET)
      studenten.add(s);
  }

  public void verlasse(Student s)
  {
    studenten.remove(s);
  }




}
```

Die Implementierung der Liste genügt dem List-Interface.

Die Implementierung der Liste kann daher getauscht werden.

In der Raum-Implementierung wird vermieden,
dass sich ein Student zweimal im selben Raum befindet.

```
public void betrete(Student s)
{
    if (!studenten.contains(s) && studenten.size() < KAPAZITAET)
      studenten.add(s);
}
```

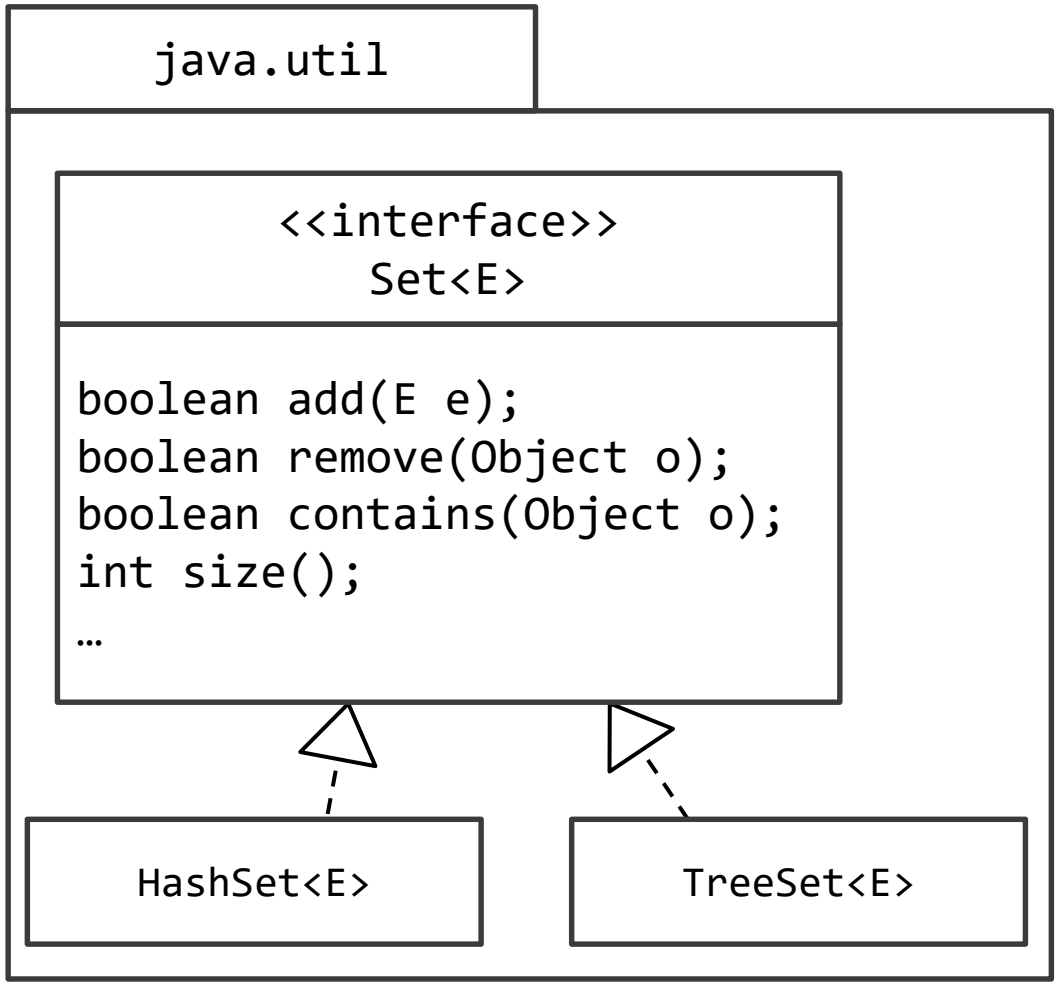
Es gibt Collections, die das von Hause aus können, sog. **Sets**.

Ein Set ist wie eine mathematische Menge:
Es gibt keine *doppelten* Elemente.

M = {1,2,3,2} 

M = {1,2,3} 

```
public class Raum
{
  final int KAPAZITAET = 50;
  Set<Student> studenten = new HashSet<>();

  public void betrete(Student s)
  {
    if (studenten.size() < KAPAZITAET)
      studenten.add(s);
  }

  public void verlasse(Student s)
  {
    studenten.remove(s);
  }
}
```



- unsortiert

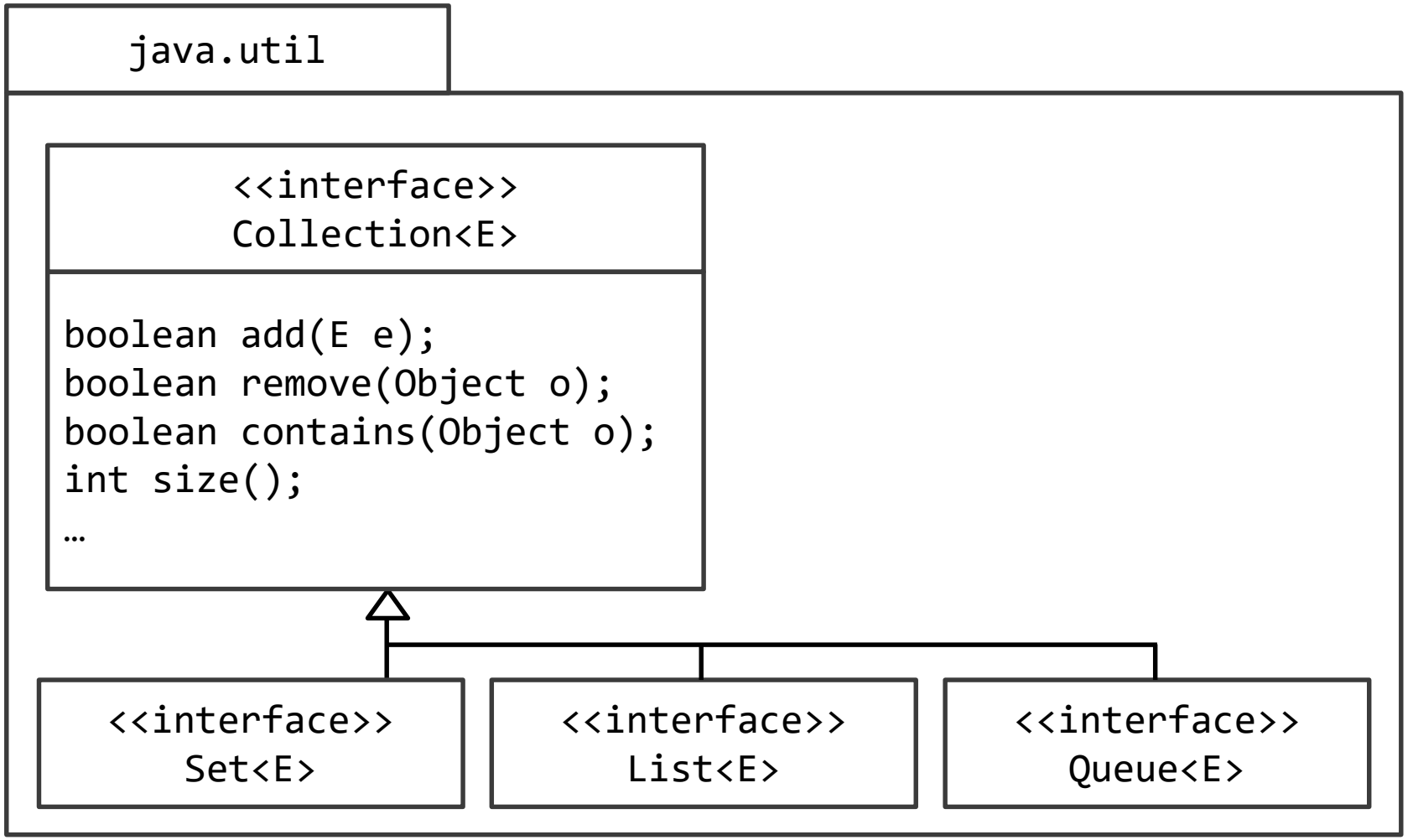
- sortiert
- E muss sortierbar, d.h. vergleichbar sein

unterschiedliche Performance je nach Szenario

Jetzt haben wir ein Set benutzt, aber die verwendeten Methoden add() und remove() sind unverändert.

Warum funktioniert das?

- Set und List sind ähnlich.
- Daher erben auch sie von einem gemeinsamen Interface.



```
public class Raum
{
  final int KAPAZITAET = 50;          (LinkedList, ArrayList, …)
  Collection<Student> studenten = new HashSet<>();

  public void betrete(Student s)
  {
    if (studenten.size() < KAPAZITAET)
      studenten.add(s);
  }

  public void verlasse(Student s)
  {
    studenten.remove(s);
  }
}
```

Wir wollen die Studenten, die gerade im Raum sind, ausgeben.

```
public class Raum
{
  List<Student> studenten = new ArrayList<>();
  …
  public void listeStudentenImRaum()
  {
    for (int i = 0; i < studenten.size(); i++)
    {
      System.out.println(studenten.get(i));
    }
  }
}
```

gibt den i-ten Studenten zurück
wie bei Arrays

Wir haben eine Aktion **für jeden** Studenten durchgeführt.

Wir können die Schleife leicht modifizieren:

```
public class Raum
{
  List<Student> studenten = new ArrayList<>();
  …
  public void listeStudentenImRaum()
  {
    for (int i = 0; i < studenten.size(); i++)
    {
      Student student = studenten.get(i);
      System.out.println(student);
    }
  }
}
```

Hier sieht man noch deutlicher, dass für jeden Studenten in der Liste eine Aktion durchgeführt wird.

Die Aktion ist unabhängig vom Schleifenindex.

Dieser Fall tritt sehr oft bei Collections auf.

Hierfür wurde eine sog. **for-each-Schleife** eingeführt
mit etwas anderer Syntax:

```
public class Raum
{
  List<Student> studenten = new ArrayList<>();
  …
  public void listeStudentenImRaum()
  {
    for (Student s : studenten)
    {
      System.out.println(s);
    }
  }
}
```

Für jedes Studentenobjekt *s* in der Liste *studenten*, tue folgendes…

Oder kurz:
Für jeden Student *s* in *studenten*, tue folgendes…

Die for-each Schleife funktioniert bei allen Collections und Arrays.

```
public class Raum {
  List<Student> studenten = new ArrayList<>();
  …
  public void listeStudentenImRaum() {
    for (int i = 0; i < studenten.size(); i++) {
      Student student = studenten.get(i);
      System.out.println(student);
    }
  }
}
```

Hier haben wir die Methode get(i) verwendet, um die Liste zu durchlaufen

```
public class Raum {
  Collection<Student> studenten = new ArrayList<>();
  …
  public void listeStudentenImRaum() {
    for (int i = 0; i < studenten.size(); i++) {
      Student student = studenten.get(i);
      System.out.println(student);
    }
  }
}
```

Diese Methode gibt es im Collection-Interface nicht!

Wie kann man Collections durchlaufen?

```
public class Raum {
  List<Student> studenten = new ArrayList<>();
  …
  public void listeStudentenImRaum() {
    for (int i = 0; i < studenten.size(); i++) {
      Student student = studenten.get(i);
      System.out.println(student);
    }
  }
}
```

Hier haben wir die Methode get(i) verwendet, um die Liste zu durchlaufen

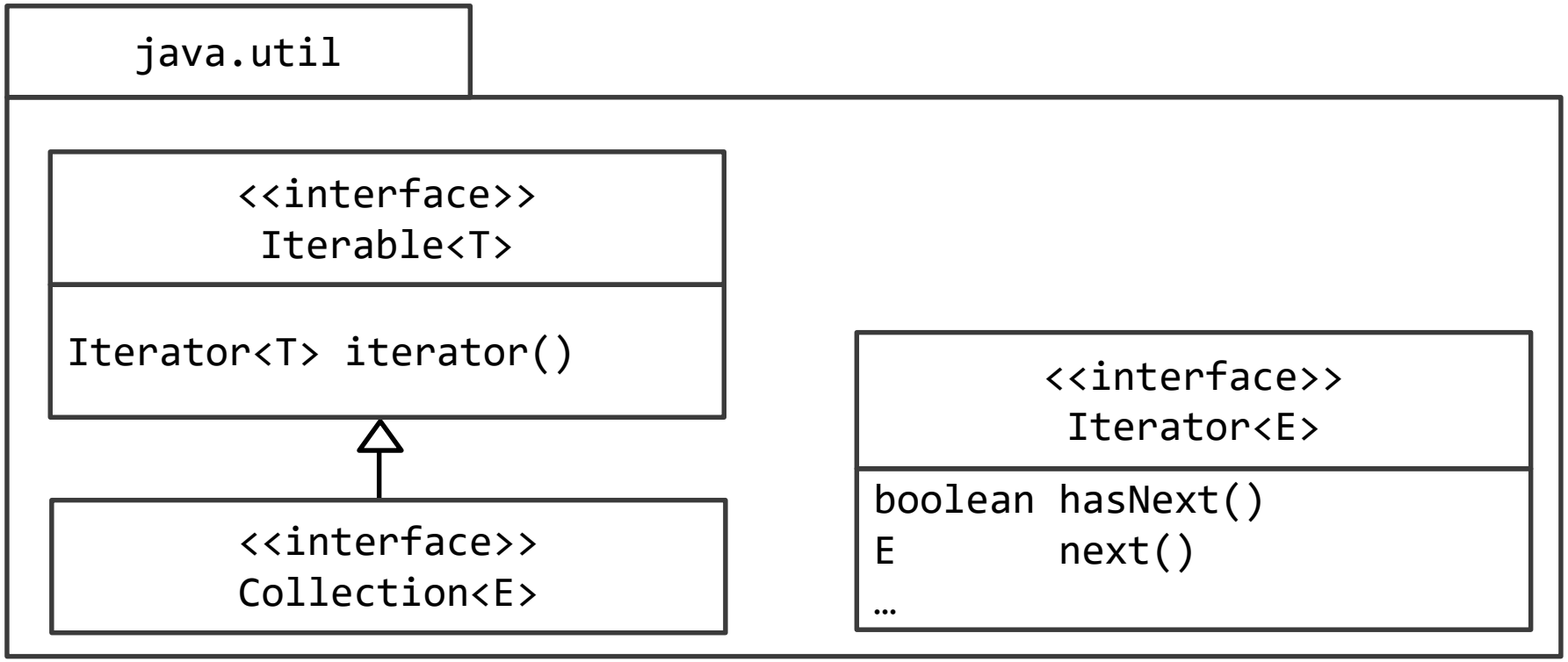
```
public class Raum {
  Collection<Student> studenten = new ArrayList<>();
  …
  public void listeStudentenImRaum() {
    for (Student s : studenten)
    {
      System.out.println(s);
    }
  }
}
```

Diese Methode gibt es im Collection-Interface nicht!

Wie kann man Collections durchlaufen?

Die for-each Schleife kann es.

Wie?



Jede Implementierung einer Collection hat eine `iterator()`-Methode, die eine Iterator-Implementierung zurückgibt, die weiß, wie die Collection durchlaufen werden muss.

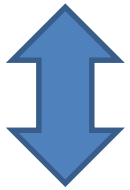
```
public class Raum {
  Collection<Student> studenten = new ArrayList<>();
  …
  public void listeStudentenImRaum() {
    Iterator<Student> iterator = studenten.iterator();


    while(iterator.hasNext())
    {

      Student student = iterator.next();
      System.out.println(student);

    }
  }
}
```

Man lässt sich einen Iterator von der Collection geben.

Solange es noch weitere Elemente in der Collection gibt…

Hole das nächste Element.
Führe eine Aktion aus.

```
Iterator<Student> iterator = studenten.iterator();
while(iterator.hasNext())
{
  Student student = iterator.next();
  System.out.println(student);
}
```



Tatsächlich ist die for-each-Schleife bei Collections eine Kurzform für obigen Code!

```
for (Student student : studenten)
{
  System.out.println(student);
}
```

Nach der Kompilierung liegt derselbe Bytecode vor.

Nehmen wir an wir wollen die Bundesligatabelle implementieren!

Wie gehen wir vor?

Die Bundesliga besteht aus 18 Mannschaften.

Jede Mannschaft besteht aus:
Platzierung
Mannschaftsname
Anzahl Spiele
Torverhältnis
Punkte

Fußball: Bundesliga

1.	Bay.München	32	75:15	82
2.	Bor.Dortmund	32	80:31	77
3.	B.Leverkusen	32	52:36	57
4.	B.M'gladbach	32	63:49	49
5.	Hertha BSC	32	41:40	49
6.	FC Schalke 04	32	46:47	48
7.	FSV Mainz 05	32	43:41	46
8.	1.FC Köln	32	36:40	41
9.	FC Ingolstadt	32	30:37	40
10.	VfL Wolfsburg	32	43:48	39
11.	Hamburger SV	32	37:44	38
12.	FC Augsburg	32	40:48	37
13.	Hoffenheim	32	38:49	37
14.	Darmstadt 98	32	36:50	35
15.	Werder Bremen	32	49:65	34
16.	Ein.Frankfurt	32	33:51	33
17.	VfB Stuttgart	32	48:69	33
18.	Hannover 96	32	29:59	22

```
public class Mannschaft
{
  int platzierung;
  String name;
  int anzahlGespielteSpiele;
  int tore;
  int gegentore;
  int punkte;

  //Konstruktor
  …

  @Override
  public String toString()
  {
    ByteArrayOutputStream baos = new ByteArrayOutputStream();
    PrintStream ps = new PrintStream(baos);
    ps.printf("%2s %14s %3s %7s %3s", platzierung, name,
      anzahlGespielteSpiele, tore + ":" + gegentore, punkte);
    ps.flush();
    return baos.toString();
  }
}
```

```
public static void main(String[] args)
{
    List<Mannschaft> tabelle = new ArrayList<Mannschaft>(18);
    tabelle.add(new Mannschaft(1, "Bay.München", 32, 75, 15, 82));
    tabelle.add(new Mannschaft(2, "Bor.Dortmund", 32, 52, 36, 77));
    tabelle.add(new Mannschaft(3, "B.Leverkusen", 32, 80, 31, 57));
    tabelle.add(new Mannschaft(4, "B.M'gladbach", 32, 64, 49, 49));
    tabelle.add(new Mannschaft(5, "Hertha BSC", 32, 41, 40, 49));
    tabelle.add(new Mannschaft(6, "FC Schalke 04", 32, 46, 47, 48));
    tabelle.add(new Mannschaft(7, "FSV Mainz 05", 32, 43, 41, 46));
    tabelle.add(new Mannschaft(8, "1.FC Köln", 32, 36, 40, 41));
    tabelle.add(new Mannschaft(9, "FC Ingolstadt", 32, 30, 37, 40));
    tabelle.add(new Mannschaft(10, "VfL Wolfsburg", 32, 43, 48, 39));
    tabelle.add(new Mannschaft(11, "Hamburger SV", 32, 37, 44, 38));
    tabelle.add(new Mannschaft(12, "FC Augsburg", 32, 40, 48, 37));
    tabelle.add(new Mannschaft(13, "Hoffenheim", 32, 38, 49, 37));
    tabelle.add(new Mannschaft(14, "Darmstadt 98", 31, 35, 48, 35));
    tabelle.add(new Mannschaft(15, "Werder Bremen", 32, 49, 65, 34));
    tabelle.add(new Mannschaft(16, "VfB Stuttgart", 32, 48, 69, 33));
    tabelle.add(new Mannschaft(17, "Ein.Frankfurt", 31, 31, 50, 30));
    tabelle.add(new Mannschaft(18, "Hannover 96", 32, 29, 59, 22));
    for (Mannschaft mannschaft : tabelle)
        System.out.println(mannschaft);

}
```

Ein Spiel muss noch nachgetragen werden.

Die Elemente der Liste treten in der Reihenfolge auf, in der sie eingefügt wurden.

Die Platzierung hängt aber von den Punkten ab.

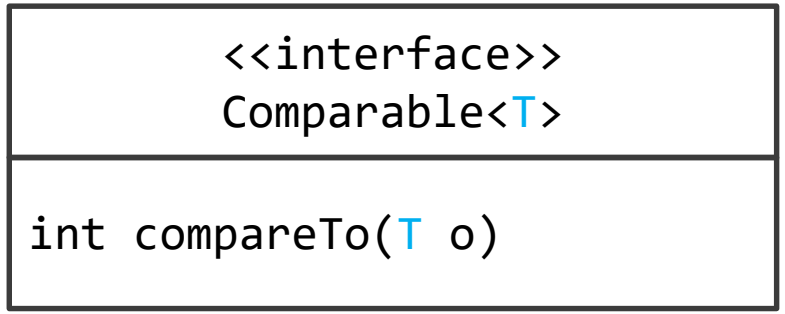
Besser man ermittelt die Platzierung nach den Punkten.

Wie kann in einer Collection festgelegt werden, wie sie sortiert werden soll?

Indem man die Elemente der Collection untereinander vergleicht.

Die Vergleichbarkeit wird i.d.R. durch einen Standardmechanismus implementiert:

Das generische Interface **Comparable**.



```
public class Mannschaft
 implements Comparable<Mannschaft>
{
  String name;
  int anzahlGespielteSpiele;
  int tore;
  int gegentore;
  int punkte;

  //Konstruktor
  …

  @Override
  public int compareTo(Mannschaft m)
  {




  }
}
```

Mit was soll eine Mannschaft verglichen werden?

Mit einer weiteren Mannschaft.

Daher ist der Typ des generischen Interfaces Mannschaft.

```
public class Mannschaft
 implements Comparable<Mannschaft>
{
  String name;
  int anzahlGespielteSpiele;
  int tore;
  int gegentore;
  int punkte;

  //Konstruktor
  …

  @Override
  public int compareTo(Mannschaft m)
  {
    if (this.punkte < m.punkte) return 1;
    else if (this.punkte > m.punkte) return -1;
    else
    {
      if (this.tore - this.gegentore < m.tore - m.gegentore) return 1;
      else if (this.tore - this.gegentore > m.tore - m.gegentore) return -1;
    }
    return 0;
  }
}
```

compareTo gibt einen int-Wert zurück.

< 0, wenn die aktuelle Mannschaft (this) früher in der Liste auftauchen soll als die Mannschaft m.

= 0, wenn es egal ist, welche der beiden Mannschaften früher auftaucht.

> 0, wenn die aktuelle Mannschaft (this) später in der Liste auftauchen soll als die Mannschaft m.

Wie erfolgt jetzt die Sortierung?

Durch den gleichen Mechanismus, durch den alle Collections sortiert werden können:

Collections.sort(collection);

```
public static void main(String[] args)
{
  List<Mannschaft> tabelle = new ArrayList<Mannschaft>(18);
  …
  tabelle.add(new Mannschaft("Darmstadt 98", 32, 36, 50, 35));
  tabelle.add(new Mannschaft("Werder Bremen", 32, 49, 65, 34));
  tabelle.add(new Mannschaft("VfB Stuttgart", 32, 48, 69, 33));
  tabelle.add(new Mannschaft("Ein.Frankfurt", 32, 33, 51, 33));
  tabelle.add(new Mannschaft("Hannover 96", 32, 29, 59, 22));

  Collections.sort(tabelle);

  for (int i = 0; i < tabelle.size(); i++)
  {
    Mannschaft mannschaft = tabelle.get(i);
    System.out.printf("%3s %s\n", i+1, mannschaft);
  }
```

Ausgabe:

```
 14   Darmstadt 98  32   36:50  35
 15  Werder Bremen  32   49:65  34
 16  Ein.Frankfurt  32   33:51  33
 17  VfB Stuttgart  32   48:69  33
 18    Hannover 96  32   29:59  22
```

```
public class Mannschaft
 implements Comparable<Mannschaft>
{
  String name;
  int anzahlGespielteSpiele;
  int tore;
  int gegentore;
  int punkte;

…

  @Override
  public int compareTo(Mannschaft m)
  {
    if (this.punkte < m.punkte) return 1;
    else if (this.punkte > m.punkte) return -1;
    else
    {
      if (this.tore - this.gegentore < m.tore - m.gegentore) return 1;
      else if (this.tore - this.gegentore > m.tore - m.gegentore) return -1;
    }
    return this.name.compareTo(m.name);
  }
}
```

Eine bessere Implementierung von compareTo stellt sicher, dass **this.compareTo(m) nur 0 zurückgibt, wenn auch this.equals(m) == true ist.**

Ansonsten kann es bspw. Probleme beim Einfügen in sortierte Sets (bspw. TreeSet) geben.

herkömmliches Array:

finger[0] -> Bild mit eingerahmtem kleinen Finger
finger[1] -> Bild mit eingerahmtem Ringfinger
…

besser:

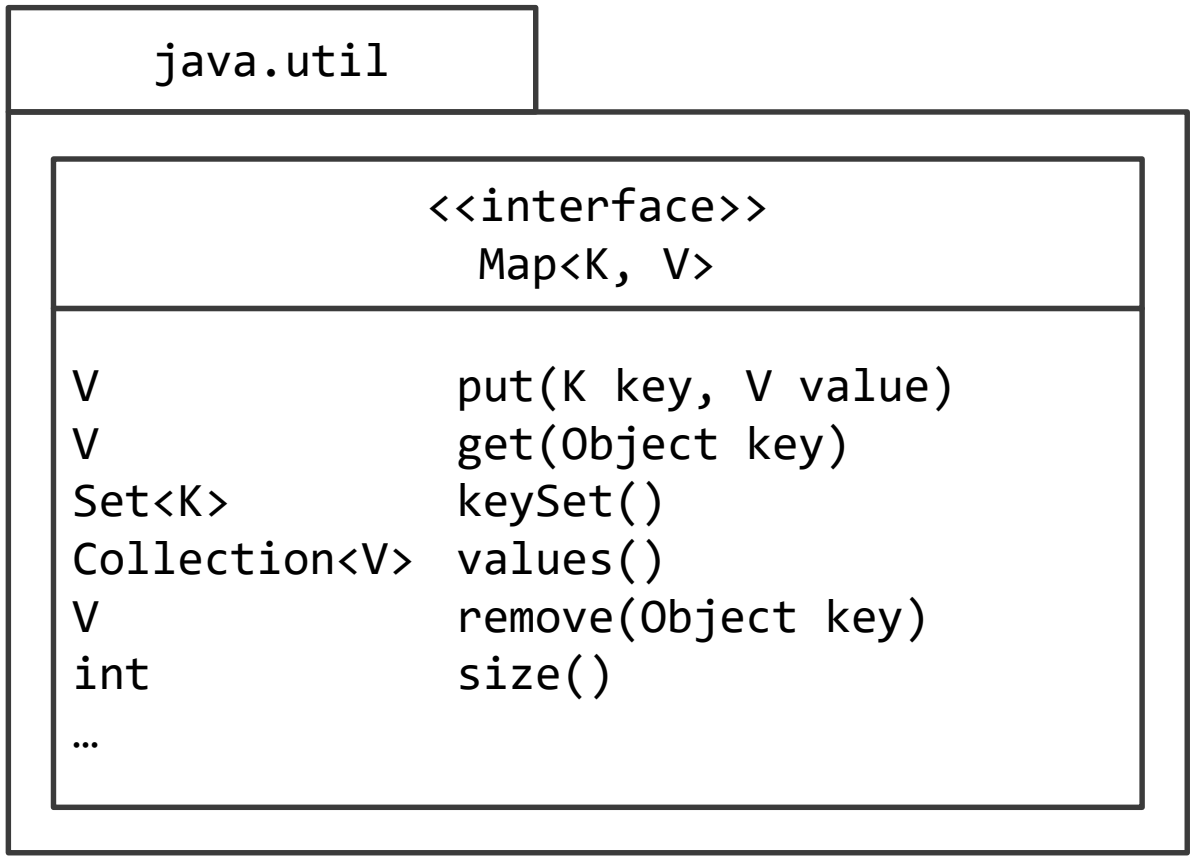
“Kleiner Finger” -> Bild mit eingerahmtem kleinen Finger
“Ringfinger” -> Bild mit eingerahmtem Ringfinger
…

Der Zugriff auf Arrays (oder Listen) erfolgt
über Indizes, d.h. über eine Zahl.

Lösung: Assoziative Arrays
(im Engl. auch Map)

Wo werden assoziative Arrays (Maps) eingesetzt?

Wörterbuch (Englischvokabel -> Deutschvokabel)
Bundesligatabelle (Platz -> Mannschaft)
Session im Webserver (Session ID des Users -> Sessiondaten)
…



In Java werden assoziative Arrays durch Maps umgesetzt.

Wir entwickeln einen Vokabeltrainer!

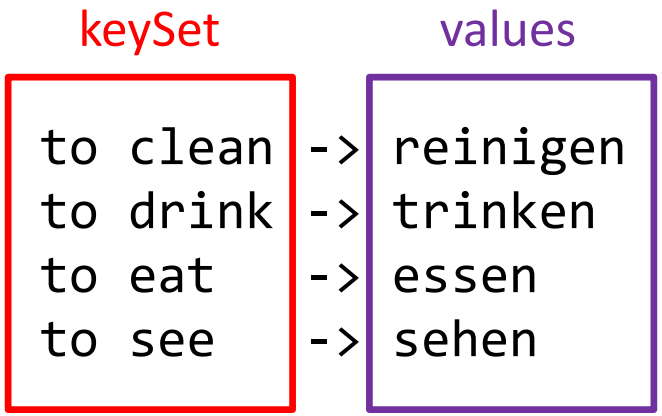
```
public class Vokabeltrainer
{
  Map<String, String> english2German = new HashMap<>();


  public void addToDictionary(String englishWord, String germanWord)
  {
    english2German.put(englishWord, germanWord);
  }

  …
```

hinterlegt das Mapping englishWord -> germanWord,
also konkret: to clean -> reinigen

```
  public static void main(String[] args)
  {
    Vokabeltrainer guessingGame = new Vokabeltrainer();
    guessingGame.addToDictionary("to clean", "reinigen");
    …
  }
```

```
public class Vokabeltrainer
{
  Map<String, String> english2German = new HashMap<>();
  String wordToGuess;

  …

  public void createRandomWordToGuess()
  {
    Set<String> keySet = english2German.keySet();
    int randomIndex = (int)(Math.random()*keySet.size());
    Iterator<String> iterator = keySet.iterator();
    int i = 0;
    while(iterator.hasNext())
    {
      String word = iterator.next();
      if (i == randomIndex)
      {
        wordToGuess = word;
        return;
      }
      i++;
    }
  }
```

liefert alle keys der Map als Set zurück



```
public class Vokabeltrainer
{
  Map<String, String> english2German = new HashMap<>();
  String wordToGuess;

  …

  public String getWordToGuess()
  {
    return wordToGuess;
  }

  public boolean guess(String guess)
  {
    String solution = english2German.get(wordToGuess);
    if (guess.equals(solution))
      return true;
    return false;
  }
```

schaut nach, ob zu dem String in der Variablen wordToGuess ein Mapping existiert

```
public class Vokabeltrainer
{
  Map<String, String> english2German = new HashMap<>();
  String wordToGuess;

  …

  public static void main(String[] args)
  {
    Vokabeltrainer guessingGame = new Vokabeltrainer();
    guessingGame.addToDictionary("to clean", "reinigen");
    guessingGame.addToDictionary("to expand", "vergrößern");
    …
    guessingGame.createRandomWordToGuess();

    Scanner scanner = new Scanner(System.in);

    System.out.println("Was heißt \"" + guessingGame.getWordToGuess() + "\" auf Deutsch?");
    String guess = scanner.nextLine();
    boolean correct = guessingGame.guess(guess);
    if (correct) System.out.println("Korrekt!");
    else System.out.println("Leider falsch!");

    scanner.close();
  }
}
```

Generics:
Unbounded Wildcards
Upper Bounded Wildcards

Die Typinformationen von generischen Klassen
sind in Java nur zur Compile-Time vorhanden,
nicht zur Laufzeit.

Die Entfernung der Typinformationen heißt auch
Type Erasure.

Zuweisungskompatibilität

```
List<String> strings = new ArrayList<>();
List<Object> objects = strings;
```

Nicht zuweisungskompatibel!

cannot convert from List<String> to List<Object>

Eine Zuweisung von Listen verschiedener Typen
wird über sogenannte Wildcards möglich.

Wildcards

- Wildcards ermöglichen, dass parametrisierbare Klassen verschiedenster Typen verarbeitet werden können:

```
List<String> strings = new ArrayList<>();
List<?> list = strings;
```

List<?> ist eine Liste von unbekanntem Typ
? ist eine unbounded Wildcard

Wir können mit Hilfe einer Unbounded Wildcard bspw. eine Methode schreiben, die beliebige Listen ausgeben kann.

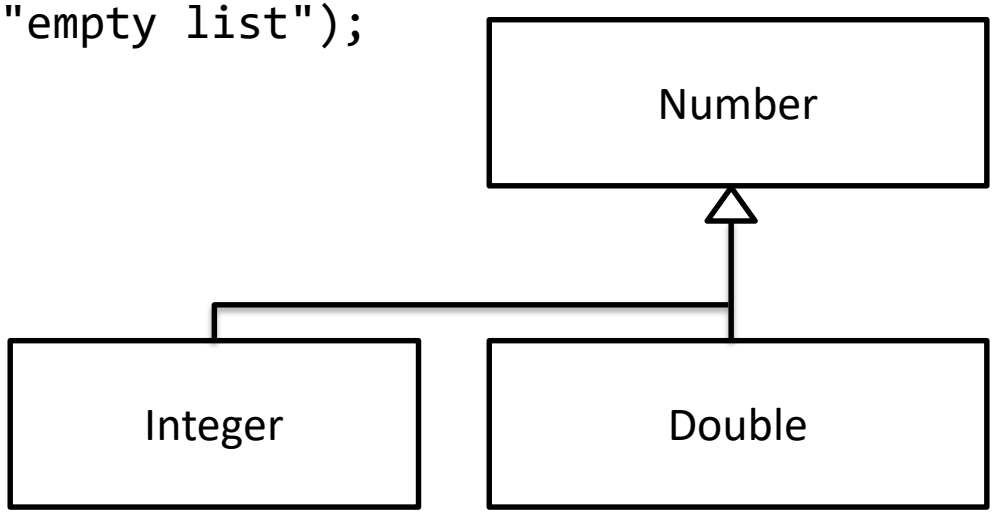
```
public static void printList(List<?> list)
{
  for (int i = 0; i < list.size(); i++)
  {
    System.out.println(list.get(i));
  }
}
```

```
public static void main(String[] args)
{
  List<String> strings = new ArrayList<>();
  strings.add("Hallo");
  strings.add("Welt");
  printList(strings);
}
```

```
public static void main(String[] args)
{
  List<Double> doubles = new ArrayList<>();
  doubles.add(Math.PI);
  doubles.add(Math.E);
  printList(doubles);
}
```

Wir wollen eine Methode schreiben, die beliebige Zahlen miteinander multipliziert!

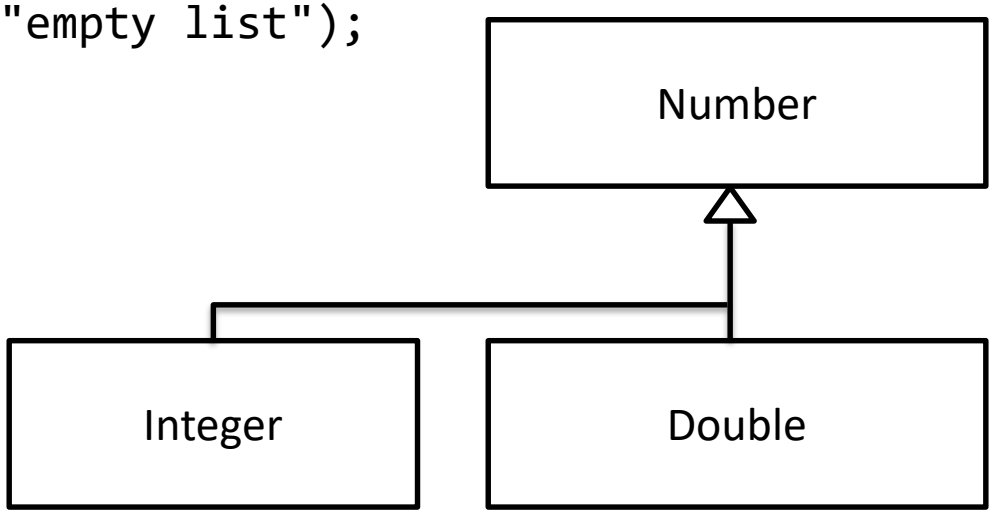
```
public static double multiplyAll(List<Number> numbers)
{
  if (numbers.isEmpty()) throw new RuntimeException("empty list");
  double result = 1;
  for (Number t : numbers)
    result = result * t.doubleValue();
  return result;
}

public static void main(String[] args)
{
  List<Double> doubles = new ArrayList<>();
  doubles.add(Math.PI);
  doubles.add(Math.E);
  multiplyAll(doubles);
}
```



List<Number> is not applicable for the arguments List<Double>

Wir können das Problem durch eine Upper Bounded Wildcard lösen!

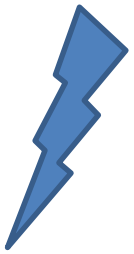
```
public static double multiplyAll(List<? extends Number> numbers)
{
  if (numbers.isEmpty()) throw new RuntimeException("empty list");
  double result = 1;
  for (Number t : numbers)
    result = result * t.doubleValue();
  return result;
}

public static void main(String[] args)
{
  List<Double> doubles = new ArrayList<>();
  doubles.add(Math.PI);
  doubles.add(Math.E);
  multiplyAll(doubles);
}
```



Es wird eine List mit Number oder Untertypen von Number akzeptiert.

Zu einer Collection mit einer **Unbounded Wildcard** können keine Elemente hinzugefügt werden:

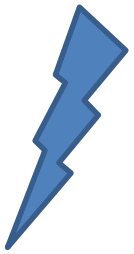
```
List<String> strings = new ArrayList<>();
strings.add("Hallo");
List<?> list = strings;
list.add("Welt");
```



Doch warum ist das so?

Bei list.add ist der Typ der Liste nicht bekannt.

Zu einer Collection mit einer **Upper Bounded Wildcard**
können keine Elemente hinzugefügt werden:



```
List<Integer> ints = Arrays.asList(new Integer[] {1,2,3});
List<? extends Number> nbelow = ints;
Number n = new Integer(5);
nbelow.add(new Integer(5));
nbelow.add(n);
```

Doch warum ist das so?

Die Liste nbelow kann bspw. vom Typ Number, Integer oder Double etc. sein.
Ist die Liste vom Typ Double ist es problematisch

- einen neuen Integer
- eine neue Number

hinzuzufügen.

Lower Bounded Wildcards

- später im Lambda Kapitel