

Programmieren II: Java

Das Java Collections-Framework

Prof. Dr. Christopher Auer

Sommersemester 2024



Wrapper-Klassen primitiver Typen

Collection-Klassen

Iteratoren

Vergleichen mit Comparable und Comparator

Inhalt

Wrapper-Klassen primitiver Typen

Primitive Typen in einer objektorientierten Sprache

Autoboxing

Zusammenfassung

Inhalt

Wrapper-Klassen primitiver Typen

Primitive Typen in einer objektorientierten Sprache

Eigenschaften von Wrapper-Klassen

Identität und Gleichheit

Veralteter Konstruktor

Primitive Typen in einer objektorientierten Sprache

- ▶ Primitive Typen spielen eine **Sonderrolle** in Java

- ▶ Zuweisung kopiert **Wert**

```
int i = 42;  
int j = i; // Wertzuweisung
```

- ▶ **Wertvergleich** über **Identität** mit ==

```
if (answer == 42)  
    ...
```

- ▶ **Kein** „.“-Operator

```
int i = 1337;  
i.toString(); // FEHLER
```


„int cannot be dereferenced“

- ▶ Wie **integriert** man primitive Typen in eine **objektorientierte Sprache**?



Primitive Typen in einer objektorientierten Sprache

- ▶ Idee: „Wrapper-Klassen“
 - ▶ Klasse je primitiver Typ
 - ▶ Instanzen „verpacken“ konkreten Wert

```
10  runIntegerWrapperExample  
11 Integer i = Integer.valueOf(42);  
12 System.out.printf("Value of %d%n", i.intValue());
```

 WrapperExamples.java

↗ `Integer.valueOf(int)` erstellt Instanz (später: **Autoboxing**)

 → Integer: value = 42

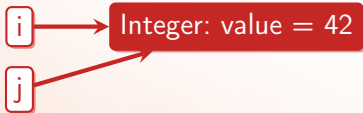
- ▶ Daher „Wrapper“ (Verpackung)



Wrapper-Klassen als Referenztyp

- ▶ Wrapper-Klasse „verpackt“ primitiven Typ in **Referenztyp**

```
Integer i = Integer.valueOf(42);  
Integer j = i;
```



i und j zeigen auf **dieselbe** Instanz

- ▶ **Zum Vergleich:** Primitive Typen arbeiten **immer** mit dem Wert

```
int i = 42;  
int j = i;
```

i = 42

j = 42



Übersicht: Wrapper-Klassen in Java

Wrapper-Klasse	Typ	Basisklasse
↗ Byte	byte	↗ Number
↗ Short	short	↗ Number
↗ Integer	int	↗ Number
↗ Long	long	↗ Number
↗ Double	double	↗ Number
↗ Float	float	↗ Number
↗ Boolean	boolean	↗ Object
↗ Character	char	↗ Object



Inhalt

Wrapper-Klassen primitiver Typen

Primitive Typen in einer objektorientierten Sprache

Eigenschaften von Wrapper-Klassen

Identität und Gleichheit

Veralteter Konstruktor

Wrapper-Klassen in Java

- ▶ Wrapper-Klassen...
 - ▶ sind **unveränderlich** (keine Setter)
 - ▶ definieren **nützliche Konstanten**

```
Integer.MAX_VALUE  
Double.POSITIVE_INFINITY
```

- ▶ definieren **nützliche statische Methoden**


```
Boolean.valueOf("true");  
Character.toUpperCase('a'); // 'A'  
Integer.parseInt("271", 8); // (271)_8
```



Wrapper-Klassen in Java

► Wrapper-Klassen...

- ermöglichen **vereinheitlichte** Behandlung primitiver Typen **als Objekte**

```
18  runPrimitivesAsObjectExample  
19 Object[] numbers = {  
20     Integer.valueOf(42),  
21     Double.valueOf(Math.PI),  
22     Byte.parseByte("01101100", 2)  
23 };  
24 out.println(Arrays.toString(numbers));
```

 WrapperExamples.java

[42, 3.141592653589793, 108]

- Implementieren  **Comparable** für Vergleiche (später mehr)
- Ermöglichen Verwendung in **Java-Collections** (auch später mehr)



Inhalt

Wrapper-Klassen primitiver Typen

Primitive Typen in einer objektorientierten Sprache


Eigenschaften von Wrapper-Klassen

Identität und Gleichheit

Veralteter Konstruktor

Identität und Gleichheit

- ▶ Ein Experiment:

```
30  runWrapperValidIdentityExample  
31 Integer i = Integer.valueOf(42);  
32 Integer j = Integer.valueOf(42);  
33 out.printf("i == j: %b%n",  
34     i == j);
```

 WrapperExamples.java




```
i == j: true
```

- ▶ `i == j` heißt `i` und `j` sind **identisch**
- ▶ **Gleiche** Referenz
- ▶ Gleichheit also wie bei **primitiven Typen**?
- ▶ Nicht darauf verlassen!



Identität und Gleichheit

► Noch ein Experiment

```
40  runWrapperNoIdentityExample
41 Integer i = Integer.valueOf(200);
42 Integer j = Integer.valueOf(200);
43 out.printf("i == j: %b\n", i == j);
```


 WrapperExamples.java

```
i == j: false
```

- Was ist jetzt **anders**?
- **Dokumentation** von  `Integer.valueOf`
- Werte von -128 bis 127 werden **cached**
- D.h.  `Integer.valueOf(i)`
 - $-128 \leq i \leq 127 \rightarrow$ Immer **gleiche** Referenz
 - sonst \rightarrow Immer **andere** Referenz
- Jede Wrapper-Klasse hat **andere** Caching-Strategien!
- Wert-Gleichheit nie mit `==` prüfen!

Identität und Gleichheit

► Wertvergleich mit equals

```
49  runWrapperEqualsExample  
50 Integer i = Integer.valueOf(200);  
51 Integer j = Integer.valueOf(200);  
52 out.printf("i.equals(j): %b%n", i.equals(j));
```

 WrapperExamples.java



```
i.equals(j): true
```

- Immer korrekt!
- Äquivalent zu

```
i.intValue() == j.intValue()
```

Inhalt

Wrapper-Klassen primitiver Typen

Primitive Typen in einer objektorientierten Sprache

Eigenschaften von Wrapper-Klassen

Identität und Gleichheit

Veralteter Konstruktor

Veralteter Konstruktor

- ▶ Wrapper-Klasse besitzen **Konstruktor**

```
Double d = new Double(3.0); // Warnung
```

„The constructor is deprecated“

- ▶ Konstruktor ist **veraltet**
- ▶ **Nicht** mehr verwenden
- ▶ **Stattdessen**: `valueOf`
- ▶ **Grund**
 - ▶ **Problem**: Konstruktor erstellt **immer** neue Instanz
 - ▶ `valueOf`-Methoden verwenden **Caching**
 - ▶ **Weniger** Speicherverbrauch
 - ▶ **Schneller**



Inhalt

Wrapper-Klassen primitiver Typen

Autoboxing

Regeln zum Autoboxing

Grenzen des Autoboxing

Autoboxing

- ▶ Konversion bisher

- ▶ Primitiv → Wrapper

```
var wrappedAnswer = Integer.valueOf(42);
```





- ▶ Wrapper → Primitiv

```
var primitiveAnswer = wrappedAnswer.intValue();
```



- ▶ Unschön: Viel Schreibarbeit

- ▶ Idee: Von Compiler generieren lassen:

- ▶ **int** gegeben,  Integer erwartet:  Integer.valueOf
 - ▶  Integer gegeben, **int** erwartet:  Integer.intValue

- ▶ Autoboxing

Autoboxing

► Autoboxing (primitiv → Wrapper)

```
9  🐉 runAutoboxingExample
10 Integer wrappedAnswer = 42;
11 out.printf("answer = %s\n", wrappedAnswer.toString());
```

📄 AutoboxingExamples.java

- Compiler erkennt: **int** gegeben, **Integer** erwartet
- Automatisches Verpacken von **int** in **Integer** mit `valueOf`

► Autounboxing (Wrapper → primitiv)

```
18 🐉 runAutounboxingExample
19 Integer wrappedAnswer = 42;
20 int unwrappedAnswer = wrappedAnswer;
21 out.printf("answer = %d\n", unwrappedAnswer);
```

📄 AutoboxingExamples.java

- Compiler erkennt: **Integer** gegeben, **int** erwartet
- Automatisches entpacken von **int** aus **Integer** mit `intValue`

Autoboxing — unter der Haube

► Autoboxing

```
Integer wrappedAnswer = 42;
```

Bytecode:

```
push 42  
invokestatic java/lang/Integer.valueOf  
astore wrappedAnswer
```

► Autounboxing

```
int unwrappedAnswer = wrappedAnswer;
```

Bytecode:

```
aload wrappedAnswer  
invokevirtual java/lang/Integer.intValue  
istore unwrappedAnswer
```



Wrapper-Klassen primitiver Typen

Autoboxing

Regeln zum Autoboxing

Grenzen des Autoboxing



Autoboxing durch Cast

► Autoboxing über expliziten Cast


```
27  runAutoboxingCastExample  
28 String s = ((Integer) 42).toString();  
29 out.println(s);
```

 AutoboxingExamples.java



- Cast heißt:  Integer wird erwartet
- (Integer)42 →  Integer.valueOf(42)

► Umgekehrt: Autounboxing über Cast

```
35  runAutounboxingCastExample  
36 Integer answer = Integer.valueOf(42);  
37 int i = (int) answer;  
38 out.println(i);
```

 AutoboxingExamples.java

- Cast heißt: **int** wird erwartet
- (**int**) answer → answer.intValue()

Autoboxing bei arithmetischen Operatoren

- ▶ +, -, /, *, usw. **erwarten** primitiven Typ
- ▶ Beispiel

44  **runAutoboxingArithmeticExample**

45 Integer i = 1;

46 Integer j = 2;

47 i = i + j;

48 out.printf("i == %s\n", i);

 AutoboxingExamples.java

i == 3

Was passiert hier?

- ▶ „i + j“ → „+“ erwartet **int**

i = i.intValue() + j.intValue();

- ▶ „i = “ → Zuweisung erwartet  **Integer** auf rechter Seite

i = Integer.valueOf(i.intValue() + j.intValue());

Autoboxing bei Vergleichsoperatoren

- ▶ <, <=, >=, > erwarten primitiven Typ

- ▶ Beispiel

54  **runAutoboxingComparisonExample**

55 Integer i = 1;

56 Integer j = 2;

57 **if** (i < j)

58 out.println("i < j");

 AutoboxingExamples.java

- ▶ „<“ erwartet primitiven Typ

```
if (i.intValue() < j.intValue())
```

- ▶ Achtung bei ==

- ▶ Funktioniert auch bei Referenztypen

```
if (i == j)
```

- ▶ Keine Umwandlung in **int**
- ▶ Vergleich der Identität


Autoboxing bei Methodenaufrufen

- ▶ Methode `printAsObject` erwartet **Referenz auf `Object`**

```
63 public static void printAsObject(Object o){  
64     out.printf("%s: %s%n",  
65         o.getClass().getSimpleName(),  
66         o.toString());  
67 }
```

AutoboxingExamples.java

- ▶ Aufruf mit **primitiven Typen**

```
77  runAutoboxingRefTypeExample  
78 printAsObject(42);  
79 printAsObject(3.1415f);  
80 printAsObject(true);
```

AutoboxingExamples.java

Autoboxing bei Methodenaufrufen

► Ausgabe

```
Integer: 42  
Float: 3.1415  
Boolean: true
```

- `printAsObject` erwartet [↗](#) `Object`
- Compiler generiert

```
printAsObject(Integer.valueOf(42));  
printAsObject(Float.valueOf(3.1415f));  
printAsObject(Boolean.valueOf(true));
```

► Regeln zu Überladung von Methoden gelten

```
public void printAsObject(int i) { ... }
```

würde bei `printAsObject(42)` aufgerufen werden

Inhalt

Wrapper-Klassen primitiver Typen

Autoboxing

Regeln zum Autoboxing

Grenzen des Autoboxing


Grenzen des Autoboxing

- ▶ Autoboxing hat seine Grenzen

```
int[] xs = new int[] {1,2,3,4};  
Integer[] ys = xs; // FEHLER
```

„incompatible types: int[] cannot be converted to Integer[]“

- ▶ ↗ **Integer-Array** kann nicht in **int-Array** konvertiert werden
 - ▶ Auch **umgekehrt** nicht möglich
 - ▶ Nur Konversion ↗ **Integer** → **int** möglich
- ▶ Lösung: Autoboxing auf Element-Ebene

```
86  runAutoboxingArrayExample  
87 int[] xs = new int[] {1,2,3,4};  
88 Integer[] ys = new Integer[xs.length];  
90 for (int i = 0; i < xs.length; i++)  
91     ys[i] = xs[i];
```

 AutoboxingExamples.java



Inhalt

Wrapper-Klassen primitiver Typen Zusammenfassung

Zusammenfassung

- ▶ Wie integriert man **primitive Typen** in eine **objektorientierte Sprache**?
- ▶ **Wrapper-Klassen**
 - ▶ „Verpacken“ primitiven Typ in Referenztyp
 - ▶ **Hilfsmethoden** (meist **statisch**)
- ▶ **Achtung**
 - ▶ Wertgleichheit nicht über „==“ prüfen!
 - ▶ **Manuelles Umwandeln** über `valueOf`!
- ▶ **Autoboxing**
 - ▶ Compiler **generiert** Code zur Umwandlung
 - ▶ **Primitiver Typ** erwartet: `Integer.intValue()`, etc.
 - ▶ **Wrapper-Typ** erwartet: `Integer.valueOf(x)`, etc.



Collection-Klassen

Motivation

ArrayList

Einschub: Generics

Übersicht

Listen

Sets

Maps

Collection-Factories

Nicht-Modifizierbare Collections

Geschachtelte Collections

Zusammenfassung

Inhalt

Collection-Klassen

Motivation

Einfache Kasse

Inhalt

Collection-Klassen

Motivation

Einfache Kasse

Einfache Kassierer-Kasse

► Aufgabe: Einfache Kassierer-Kasse

Toilettenpapier

3

Milch

1

Brot

2

Toilettenpapier: 3 EUR

Milch: 1 EUR

Brot: 2 EUR

SUMME: 6 EUR



► Liest abwechselnd **Name** und **Preis (Euro)**


► **Abbruch** bei leerem Namen

► **Ausgabe** des „Kassenbons“

► **Problem:** Wir wissen zu Beginn nicht **wie viele** Artikel kommen

Einfache Kassierer-Kasse

► Erster Ansatz: *Array*

```
99  runArrayCheckoutExample  
100 Item[] items = new Item[10];  
101 int i = 0;  
102 boolean done = false;  
104 do {  
105     Item item = readItem(scanner);  
107     if (item == null)  
108         done = true;  
109     else  
110         items[i++] = item;  
112 } while (!done);  
114 printReceipt(items);
```

 ArrayListExamples.java



Einfache Kassierer-Kasse

► Unschön

```
Item[] items = new Item[10];
```

- Array hat **feste Länge**
- Was passiert bei **mehr als zehn** Artikeln?
- [☞ ArrayIndexOutOfBoundsException!](#)

► „Lösung“


```
Item[] items = new Item[1000];
```

- Das sollte **reichen!**
- **Hoffentlich...**
- Bei wenig Artikeln: **Speicherverschwendung**
 - Einträge 0 bis $n - 1 \neq \text{null}$
 - Einträge n bis 999 $== \text{null}$
- Array-Länge ist **unveränderlich**
- Lösung: **wachsende** Datenstruktur



Einfache Kassierer-Kasse

► Zweiter Ansatz: [ArrayList](#)

```
122  runArrayListCheckoutExample  
123 ArrayList<Item> items =  
124     new ArrayList<Item>();  
125 boolean done = false;  
127 do {  
128     Item item = readItem(scanner);  
130     if (item == null)  
131         done = true;  
132     else  
133         items.add(item);  
135 } while (!done);  
137 printReceipt(items);
```

 ArrayListExamples.java



Einfache Kassierer-Kasse

► Was hat sich **verändert**?

► **Vorher**: Array

```
Item[] items = new Item[10];
```

► **Jetzt**: Instanz von **ArrayList**<Item>

```
ArrayList<Item> items = new ArrayList<Item>();
```


► **Vorher**: Schreiben in Array

```
items[i++] = item;
```

► **Jetzt**: Methode add

```
items.add(item);
```

► Vorteile

- Prinzipiell **beliebig viele** Artikel
- Keine  **ArrayIndexOutOfBoundsException**



Collection-Klassen

ArrayList

- Erstellen und Erweitern
- Elementzugriff
- Iterieren
- Einfügen
- Verändern von Elementen
- Entfernen von Elementen
- Finden von Elementen
- Weitere hilfreiche Methoden
- Vergleich Arrays und ArrayList

🔗 ArrayList

- ▶ 🔗 ArrayList unter der Haube
 - ▶ Array
 - ▶ Größe wird bei Bedarf **erweitert**
- ▶ Ähnlich zu 🔗 String vs. 🔗 StringBuilder

Feste Größe	Flexible Größe
🔗 String	🔗 StringBuilder
🔗 Array	🔗 ArrayList

- ▶ 🔗 ArrayList ist ein Generic

```
ArrayList<T> items = new ArrayList<T>();
```

- ▶ T ist zu **speichernder Typ**
- ▶ T muss Referenztyp sein

```
var numbers = new ArrayList<int>(); // FEHLER
```

„Expected reference type“

- ▶ Stattdessen 🔗 ArrayList<Integer> verwenden

Collection-Klassen

ArrayList

Erstellen und Erweitern

Elementzugriff

Iterieren

Einfügen

Verändern von Elementen

Entfernen von Elementen

Finden von Elementen

Weitere hilfreiche Methoden

Vergleich Arrays und ArrayList

Erstellen und Erweitern

- ▶ Erstellen **new** ArrayList<T>()
- ▶ Oder **new** ArrayList<T>(int initialCapacity)

```
ArrayList<String> l = new ArrayList<String>(4);
```



```
l.size() == 0
```

- ▶ [ArrayList<T>.add\(T\)](#) fügt Element hinten an

```
l.add("First!");
```

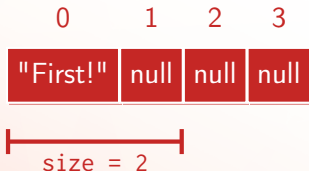


size = 1

Erweitern von `ArrayList`

- ▶ Elemente können auch **null** sein

```
l.add(null);
```



- ▶ Elemente müssen zum **Typ** passen

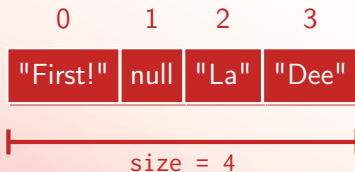
```
l.add(new Item("Salat", 2)); // FEHLER
```

- ▶ „Incompatible types Item and String“
- ▶ Wird **statisch** vom Compiler geprüft

Erweitern von ArrayList

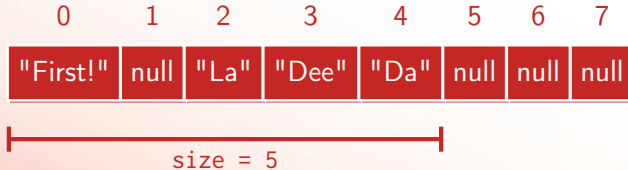
- ▶ Was passiert wenn Kapazität **nicht mehr reicht**?
- ▶ Anhängen von drei Elementen

```
l.add("La");  
l.add("Dee");
```



Erweitern von `ArrayList`

- ▶ `l.add("Da")`: `ArrayList` ...
 - ▶ erstellt intern **größeren** Array
 - ▶ **kopiert** alle bisherigen Einträge
 - ▶ hängt neues Element an



- ▶ **Vorteil**: Größe von `ArrayList` passt sich an
- ▶ **Nachteil**: Vergrößerung Kapazität kostet Zeit
- ▶ **Initiale Kapazität** geschickt wählen

Collection-Klassen

ArrayList

Erstellen und Erweitern

Elementzugriff

Iterieren

Einfügen

Verändern von Elementen

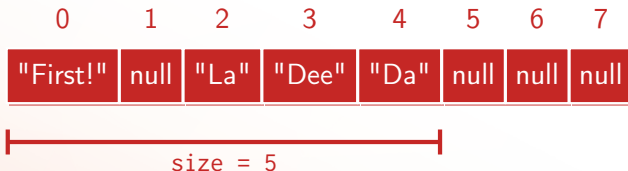
Entfernen von Elementen

Finden von Elementen

Weitere hilfreiche Methoden

Vergleich Arrays und ArrayList

Elementzugriff



► Zugriff über Index mit `get(int)`

```
l.get(0) // "First!"  
l.get(1) // null
```

- Wie bei Arrays
- [IndexOutOfBoundsException](#) bei Zugriff **außerhalb** von $0 \leq i < \text{size}$

```
l.get(5) // FEHLER
```

- **Obwohl** Array-Element prinzipiell existiert!

Collection-Klassen

ArrayList

Erstellen und Erweitern

Elementzugriff

Iterieren

Einfügen

Verändern von Elementen

Entfernen von Elementen


Finden von Elementen

Weitere hilfreiche Methoden

Vergleich Arrays und ArrayList

Iterieren


▶ Klassische `for`-Schleife

```
155  runArrayListIterateForExample  
156 for (int i = 0; i < l.size(); i++)  
157     out.println(l.get(i));
```

 ArrayListExamples.java

▶ `ArrayList` implementiert `Iterable`

▶ `for`-each-Schleife

```
164  runArrayListIterateForEachExample  
165 for (String item : l)  
166     out.println(item);
```

 ArrayListExamples.java

Collection-Klassen

ArrayList

Erstellen und Erweitern

Elementzugriff

Iterieren

Einfügen

Verändern von Elementen

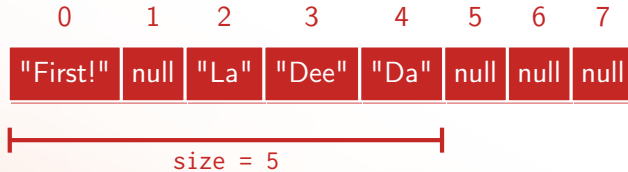
Entfernen von Elementen

Finden von Elementen

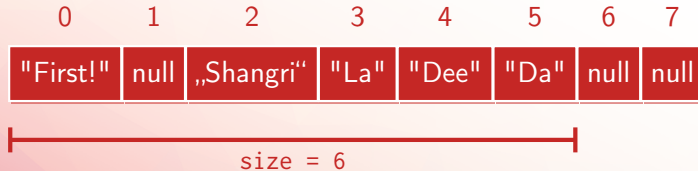
Weitere hilfreiche Methoden

Vergleich Arrays und ArrayList

Einfügen von Elementen



- ▶ `add(int index, T element)` fügt element an Stelle index ein



- ▶ `add(2, "Shangri")`
 - ▶ Alle Elemente ab index werden verschoben
 - ▶ Achtung: bei vielen Einträgen teuer
 - ▶ Besser [↗](#) `LinkedList` (später)

Collection-Klassen

ArrayList

Erstellen und Erweitern

Elementzugriff

Iterieren

Einfügen

Verändern von Elementen

Entfernen von Elementen

Finden von Elementen

Weitere hilfreiche Methoden

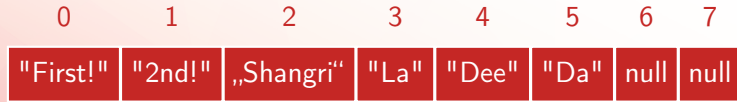
Vergleich Arrays und ArrayList

Verändern von Elementen



size = 6

- ▶ `ArrayList.set(int index, T element)` **setzt** Element an Stelle index auf element



- ▶ `l.set(1, "2nd!")`
- size = 6

Collection-Klassen

ArrayList

Erstellen und Erweitern

Elementzugriff

Iterieren

Einfügen

Verändern von Elementen

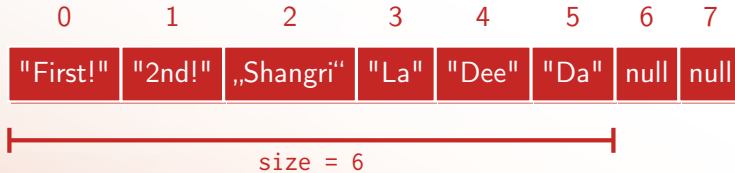
Entfernen von Elementen

Finden von Elementen

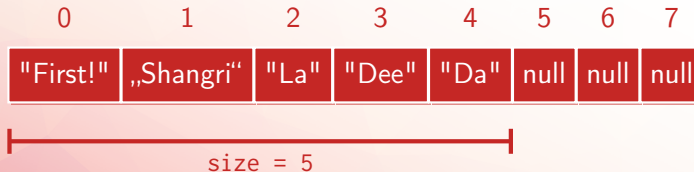
Weitere hilfreiche Methoden

Vergleich Arrays und ArrayList

Entfernen von Elementen



- ▶ `ArrayList.remove(int index)` entfernt Element an Stelle index



- ▶ `l.remove(1)`
 - ▶ Verschiebt alle Elemente rechts von index nach links
 - ▶ Achtung: Teuer bei vielen Elementen
 - ▶ Wieder besser: `LinkedList` (später)

Entfernen von Elementen

0	1	2	3	4	5	6	7
"First!"	„Shangri“	"La"	"Dee"	"Da"	null	null	null

size = 5

- ▶ **boolean** `remove(T element)` entfernt...
 - ▶ **erstes** Element `e` für das gilt
 - ▶ `e.equals(element)` wenn `e != null`
 - ▶ **Oder** `e==null` wenn `element == null`
 - ▶ Rückgabe **true** wenn **entfernt**, sonst **false**

0	1	2	3	4	5	6	7
"First!"	„Shangri“	"La"	"Da"	null	null	null	null

size = 4

- ▶ `l.remove("Dee")` (`== true`)

Collection-Klassen

ArrayList

Erstellen und Erweitern

Elementzugriff

Iterieren

Einfügen

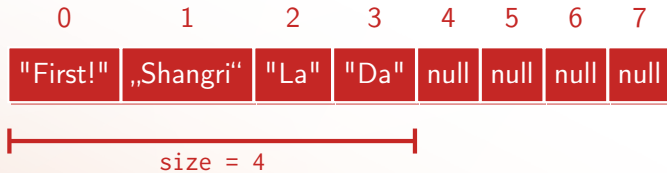
Verändern von Elementen

Entfernen von Elementen

Finden von Elementen

Weitere hilfreiche Methoden

Vergleich Arrays und ArrayList



- ▶ **boolean** `ArrayList.contains(T x)`
 - ▶ Prüft mit `equals` ob `ArrayList` `x` enthält
 - ▶ **true** wenn **ja**, sonst **false**
 - ▶ `l.contains("Shangri")== true`
 - ▶ `l.contains("Dee")== false`
- ▶ **int** `indexOf(T element)`
 - ▶ Gibt **kleinsten** Index `i` zurück für den...
 - ▶ `equals(x)` **true** liefert
 - ▶ Oder `-1` wenn `x` **nicht** gefunden wurde
 - ▶ `l.indexOf("Da")== 3`
 - ▶ `l.indexOf("Dee")== -1`
- ▶ Entsprechend `lastIndexOf` für **letzten** Index

Collection-Klassen

ArrayList

Erstellen und Erweitern

Elementzugriff

Iterieren

Einfügen

Verändern von Elementen

Entfernen von Elementen

Finden von Elementen

Weitere hilfreiche Methoden

Vergleich Arrays und ArrayList

Weitere hilfreiche Methoden

0	1	2	3	4	5	6	7
"First!"	„Shangri“	"La"	"Da"	null	null	null	null

└──────────┘
size = 4

- ▶ `clear()` entfernt alle Elemente
- ▶ `isEmpty()` **true** wenn `size()==0`, sonst **false**
- ▶ `addAll(Collection c)` fügt alle Elemente aus `c` hinzu
- ▶ `removeAll(Collection c)` entfernt alle Elemente aus `c`
- ▶ `T[] toArray()` konvertiert [↗ ArrayList](#) in `Array`
- ▶ [↗ Collections](#).`addAll(ArrayList<T> l, T[] a)` fügt alle Elemente aus `Array a` zu `l` zu

Collection-Klassen

ArrayList

Erstellen und Erweitern

Elementzugriff

Iterieren

Einfügen

Verändern von Elementen

Entfernen von Elementen

Finden von Elementen

Weitere hilfreiche Methoden

Vergleich Arrays und ArrayList

Vergleich Arrays und ArrayList

	Arrays	ArrayList[]
Deklaration	<code>T[] a</code>	<code>↗ ArrayList<T> a</code>
Erstellen	<code>new T[size]</code>	<code>new ArrayList<T>()</code>
Zugriff	<code>a[idx]</code>	<code>a.get(idx)</code>
# Elemente	fest: <code>a.length</code>	veränderlich: <code>a.size()</code>
Einfügen	nicht möglich	<code>a.add(x)</code> , <code>a.add(idx,x)</code>
Entfernen	nicht möglich	<code>a.remove(idx)</code> , <code>a.remove(x)</code>
Durchsuchen	manuell	<code>a.findFirst/Last(x)</code>
Konversion	<code>↗ Collections.addAll()</code>	<code>a.toArray()</code>
Iteration	<code>for(-each)-Schleife</code>	<code>for(-each)-Schleife</code>

Inhalt

Collection-Klassen

Einschub: Generics

Einschub: Generics

- ▶ In einer Zeit **bevor** es **Generics** gab...
 - ▶ Java-Collections arbeiteten mit ↗ **Object**-Referenzen
 - ▶ Erst mal **kein Problem** bei add/set/etc.

```
12 ArrayList items = new ArrayList();  
13 items.add(new Item("Salat", 2));  
14 items.add(new Item("Milch", 1));
```

GenericsExamples.java

- ▶ Aber: Alles „ist ein“ ↗ **Object**

```
items.add("I'm a String!");
```

- ▶ **Expliziter Cast** bei get/find/etc. notwendig

```
18 int total = 0;  
19 Item item = (Item) items.get(0);  
20 total += item.getPrice();  
22 item = (Item) items.get(1);  
23 total += item.getPrice();
```

GenericsExamples.java

Einschub: Generics

- ▶ Unschön
 - ▶ Keine Typprüfung: `item.add("I'm a String!")` möglich!
 - ▶ Immer **expliziter narrowing Cast** notwendig
 - ▶ Prüfung erst zur **Laufzeit**
- ▶ Lösung: Generics
 - ▶ `ArrayList<T>` Container für **Referenztyp T**
 - ▶ **Klassendeklaration** mit T als Typ

```
void add(T x);  
T get(int index);  
T findFirst(T x);
```

- ▶ **Spezialisierung** von T bei Instanziierung

```
ArrayList<Item> items = new ArrayList<Item>();
```

- ▶ **Ersetzt** T durch konkreten Typ Item

```
void add(Item x);  
Item get(int index);  
Item findFirst(Item x);
```

Einschub: Generics

- ▶ Besser

- ▶ Typprüfung zur Übersetzungszeit

```
items.add("I'm a String!"); // FEHLER
```

- ▶ Keine Casts mehr notwendig

```
Item item = items.get(0);
```

- ▶ Klassen und Interfaces können Generic sein

- ▶ `ArrayList<T>`: `void add(T x)`

- ▶ `Iterator<E>`: `E next()`

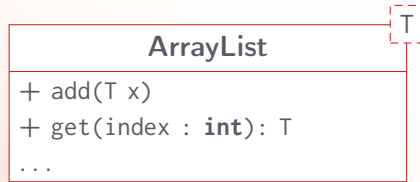
- ▶ Typparameter müssen Referenztypen sein

```
ArrayList<int> numbers = new ArrayList<int>(); // FEHLER
```

- ▶ Primitive Typen: Wrapper-Klassen und Autoboxing verwenden

```
ArrayList<Integer> numbers = new ArrayList<Integer>();  
numbers.add(23); // Autoboxing!  
int number = numbers.get(0); // Autoboxing!
```

- Darstellung von `ArrayList<T>` in UML



Inhalt

Collection-Klassen

Übersicht

Iterable

Collection

List

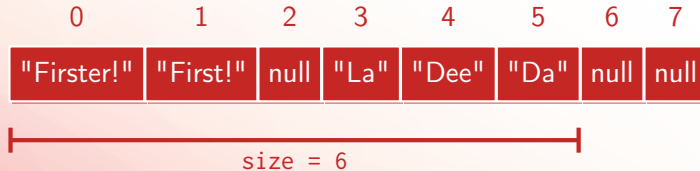
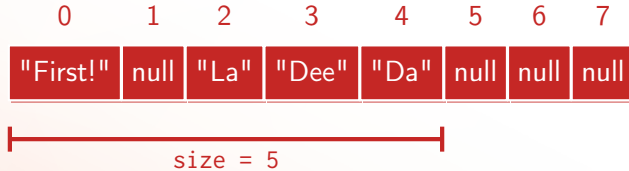
Set

Map

Vergleich

Java-Collections

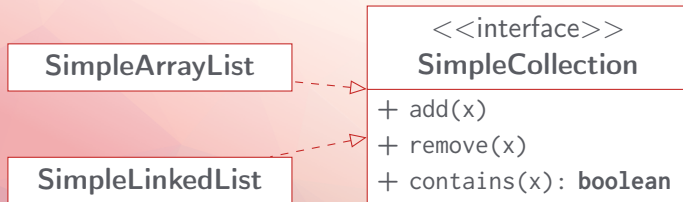
► Beispiel von vorher



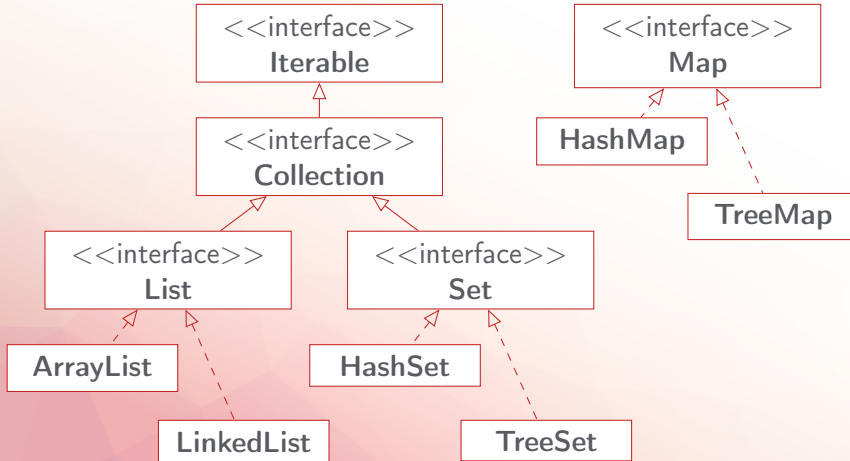
- `add(0, "Firster!")`
- **Teure Operation:** Alle Elemente werden verschoben
- Was wenn einfügen am Anfang **oft** passiert?
- Andere Datenstruktur verwenden ([↗ LinkedList](#))

Java-Collections

- ▶ **Verschiedene** Anwendungen brauchen...
- ▶ **verschiedene** Datenstrukturen
- ▶ **Aber:** Schnittstelle ist immer **ähnlich**
 - ▶ **Einfügen** d.add(x)
 - ▶ **Entfernen** d.remove(x)
 - ▶ **Durchlaufen** mit **for**-each
 - ▶ **Durchsuchen** d.contains(x)
 - ▶ ...
- ▶ Idee hinter **Java-Collections**
 - ▶ **interfaces** definieren unterstützte Operationen...
 - ▶ werden von **Datenstrukturen** implementiert



Übersicht



Inhalt

Collection-Klassen

Übersicht

Iterable

Collection

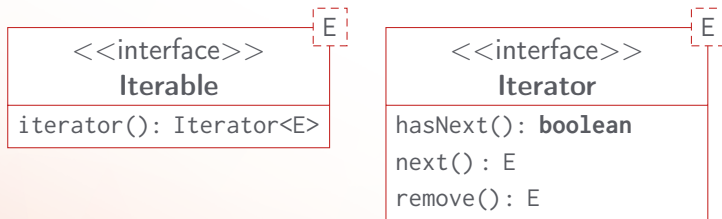
List

Set

Map

Vergleich

Iterable



- ▶ „Durchlaufbare“ Datenstrukturen
- ▶ Iteration durch **for**-each-Schleife

```
for (E element : dataStructure){  
    ...  
}
```

- ▶ Hinweis: Nicht nur für Datenstrukturen (später)
- ▶ Mathematisches Beispiel: abzählbare Mengen wie $\mathbb{N} = 1, 2, 3, \dots$

Inhalt

Collection-Klassen

Übersicht

Iterable

Collection

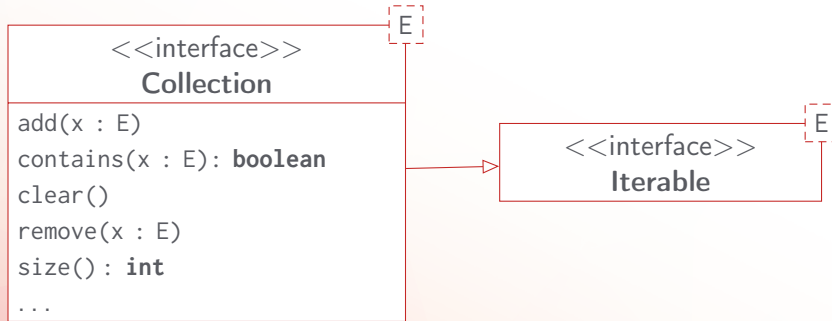
List

Set

Map

Vergleich

Collection



- ▶ Basis-Interface: Klassen die **Ansammlung** von Elementen modellieren
- ▶ **Grundlegende Operationen**: hinzufügen, entfernen, Größe und Inhalt abfragen, etc.
- ▶ **Beispiel**: Kiste/Beutel mit Gegenständen

Inhalt

Collection-Klassen

Übersicht

Iterable

Collection

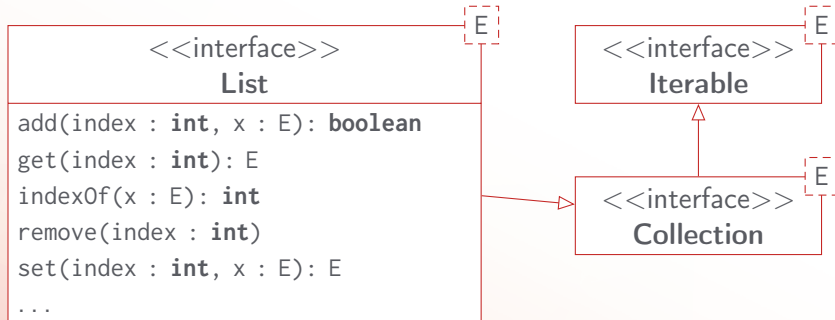
List

Set

Map

Vergleich

List



- ▶ Über **Index geordnete Liste** von Elementen
 - ▶ Zugriff über **Index** (Position)
 - ▶ Geordnet über **Index**
- ▶ Beispiele
 - ▶ ↗ `ArrayList`, ↗ `LinkedList` (siehe vorher)
 - ▶ **Rangfolge** bei einem Wettbewerb: 1. Platz, 2. Platz, etc.

Inhalt

Collection-Klassen

Übersicht

Iterable

Collection

List

Set

Map

Vergleich

Set



- ▶ [↗](#) **Collection ohne Duplikate**
 - ▶ Jedes Element **nur einmal**
 - ▶ Gleichheit über equals (und hashCode)
- ▶ **Beispiel:** Mathematische **Mengen** \mathbb{N} , $M = \{2, 3, 5, 7, 11\}$

Inhalt

Collection-Klassen

Übersicht

Iterable

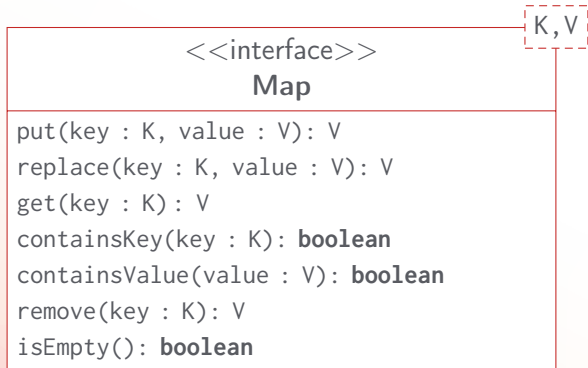
Collection

List

Set

Map

Vergleich



- ▶ Zuordnung von **Schlüsseln** (K) zu **Werten** (V)
 - ▶ Zugriff über **Schlüssel**
 - ▶ Nur **ein Wert** je Schlüssel
- ▶ Beispiele
 - ▶ Mathematische **Funktion** $f : K \rightarrow V$
 - ▶ Zuordnung Student zu Übungsgruppe: jeder Student in **höchstens** einer Übungsgruppe

Inhalt

Collection-Klassen

Übersicht

Iterable

Collection

List

Set

Map

Vergleich

Vergleich

	List	Set	Map
Zugriff über	Index	—	Schlüssel
Duplikate	ja	nein	Schlüssel eindeutig
Ordnung	Index	nur Tree -Variante	nur Tree -Variante
↗ Iterable	ja	ja	nein
Bsp. Klasse	↗ ArrayList	↗ HashSet	↗ TreeMap
Beispiel	Rangfolge	Menge	Funktion

Inhalt

Collection-Klassen

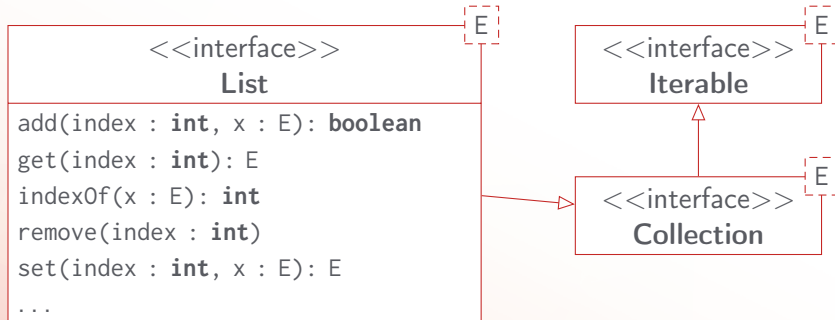
Listen

- ArrayList

- LinkedList

- ArrayList vs. LinkedList

List



- ▶ Über **Index geordnete Liste** von Elementen
 - ▶ Zugriff über **Index** (Position)
 - ▶ Geordnet über **Index**
- ▶ Implementierungen
 - ▶ ↗ **ArrayList** über **Arrays**
 - ▶ ↗ **LinkedList** über **doppelt verkettete Liste**

Inhalt

Collection-Klassen

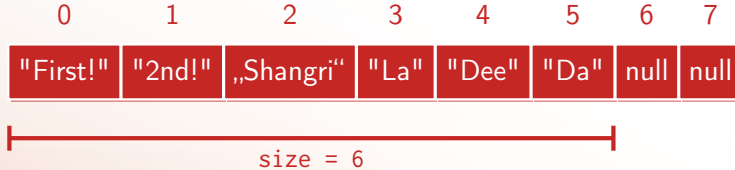
Listen

- ArrayList

- LinkedList

- ArrayList vs. LinkedList

ArrayList



- ▶ Elemente werden **intern** in **Array** abgelegt
- ▶ Arraygröße wird bei Bedarf **angepasst**
 - ▶ Erstellt neuen Array
 - ▶ Kopiert alte Einträge
- ▶ **Operationen** (n = Anzahl der Elemente)
 - ✓ `get(int i)`: direkter Zugriff über Array (schnell)
 - ✗ `add(T x)`: vergrößert nach Bedarf Array (evtl. langsam)
 - ✗ `add(int i, T x)`: verschiebt Elemente $> i$ (langsam)
 - ✗ `remove(int i)`: verschiebt Elemente $> i$ (langsam)
 - ✗ `contains(T x)`: Suche von links nach rechts (langsam)

Inhalt

Collection-Klassen

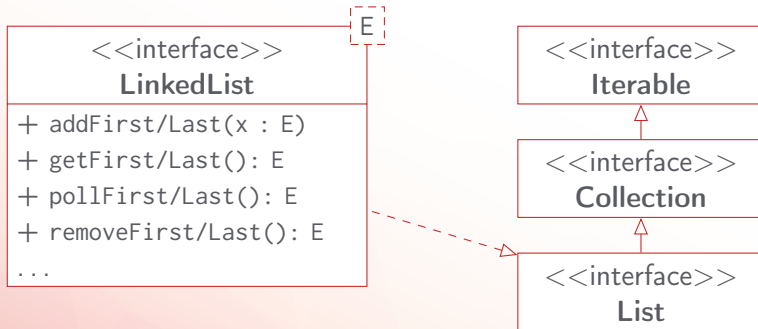
Listen

ArrayList

LinkedList


ArrayList vs. LinkedList

LinkedList



- ▶ Elemente in **doppelt-verketteter** Liste
 - ▶ Referenz auf **erstes** und **letztes** Element
 - ▶ Elemente haben Referenz auf **Vorgänger** und **Nachfolger**

LinkedList

```
10  runLinkedListExample
11 LinkedList<String> l = new LinkedList<String>();
12 l.add("La");
13 l.addLast("Da");
14 l.add(1, "Dee");
```

 LinkedListExamples.java



- ▶ `add("La")` fügt **hinten** an
- ▶ `addLast("Da")` fügt **hinten** an
 - ▶ `last` und **Nachfolger** von „La“ werden aktualisiert (**schnell**)
- ▶ `add(1, "Dee")`
 - ▶ Navigiert an Stelle 1
 - ▶ **Nachfolger** von „La“ und **Vorgänger** von „Da“ werden aktualisiert

LinkedList

```
18 l.addFirst("First!");  
19 l.removeLast();  
20 l.addLast(null);  
21 l.addLast("Last!");
```

LinkedListExamples.java



- ▶ `addFirst("First!")` aktualisiert **first** und **Vorgänger** von „La“
- ▶ `removeLast()` aktualisiert **last** und **Nachfolger** von „Dee“

LinkedList

```
26 l.set(3, "Dum");  
27 l.contains("Dee"); // true  
28 l.indexOf("Shangri"); // -1
```

LinkedListExamples.java



- ▶ `set(3, "Dum")` navigiert von first bis zu Position 3
- ▶ `contains("Dee")` durchsucht Liste von first an bis „Dee“ gefunden ist
- ▶ `indexOf("Shangri")` durchsucht Liste von first bis last (erfolglos)

LinkedList

- ▶ Durchlaufen einer [LinkedList](#) mit klassischer **for**-Schleife

```
32 for (int i = 0; i < l.size(); i++)  
33     out.println(l.get(i));
```

[LinkedListExamples.java](#)

Achtung

- ▶ `get(i)` navigiert **in jeder Iteration** von first bis zu Position `i`
- ▶ Sehr **langsam** (bei vielen Elementen)
- ▶ Besser: Durchlaufen mit **for-each**-Schleife

```
37 for (String s : l)  
38     out.println(s);
```

[LinkedListExamples.java](#)

- ▶ Verwendet **Iterator**
- ▶ **Iterator** navigiert **in jeder Iteration** zum **Nachfolger** (schnell)

Inhalt

Collection-Klassen

Listen

ArrayList

LinkedList

ArrayList vs. LinkedList

ArrayList vs. LinkedList

Operation	ArrayList	LinkedList
get(int)	✓ $\mathcal{O}(1)$	✗ $\mathcal{O}(n)$
set(int,E)	✓ $\mathcal{O}(1)$	✗ $\mathcal{O}(n)$
Hinten anhängen/löschen	✗ $\mathcal{O}(1)/\mathcal{O}(n)$	✓ $\mathcal{O}(1)$
Vorne einfügen/löschen	✗ $\mathcal{O}(n)$	✓ $\mathcal{O}(1)$
Innerhalb einfügen/löschen	✗ $\mathcal{O}(n)$	✗ $\mathcal{O}(n)$
Durchsuchen	✗ $\mathcal{O}(n)$	✗ $\mathcal{O}(n)$

► ArrayList

- ✓ Ungefähre Größe **bekannt**
- ✓ Viele Zugriffe über **Index**

► LinkedList

- ✓ Größe **unbekannt**
- ✓ Viel einfügen/löschen am **Ende/Anfang**

Inhalt

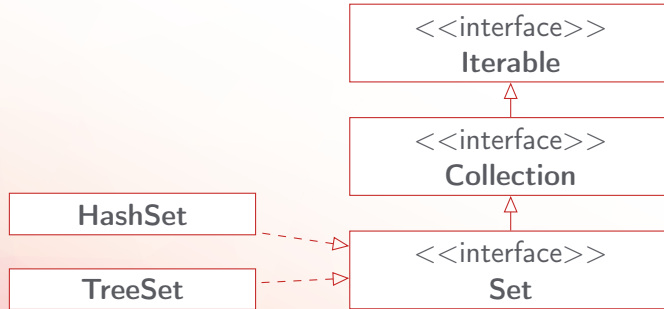
Collection-Klassen

Sets

- Sets als Mengen

- HashSet vs. TreeSet

- EnumSet



- ▶ Keine Duplikate zugelassen
 - ▶ Gleichheit über equals (und hashCode)
 - ▶ Keine Änderung bei add(x) wenn x schon enthalten

Inhalt

Collection-Klassen

Sets

Sets als Mengen

HashSet vs. TreeSet

EnumSet

Sets

☞ Set s verhalten sich wie Mengen

Mengenoperation	Set-Operation
-----------------	---------------

$A \cup \{x\}$	<code>A.add(x)</code>
----------------	-----------------------

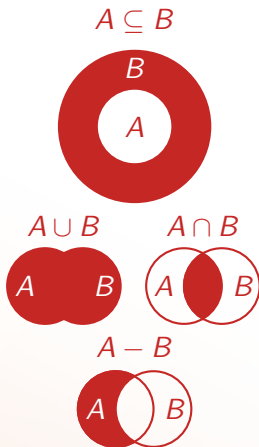
$x \in A$	<code>A.contains(x)</code>
-----------	----------------------------

$A \subseteq B$	<code>B.containsAll(A)</code>
-----------------	-------------------------------


$A \cup B$	<code>A.addAll(B)</code>
------------	--------------------------

$A \cap B$	<code>A.retainAll(B)</code>
------------	-----------------------------

$A - B$	<code>A.removeAll(B)</code>
---------	-----------------------------



Sets

```
13  runSetOperationsExample  
14 var salad = new Item("Salat", 2);  
15 var choc = new Item("Schokolade", 1);  
16 var milk = new Item("Milch", 2);  
17 var tomatoes = new Item("Tomaten", 3);  
19 var setA = new HashSet<Item>();  
20 var setB = new HashSet<Item>();  
22 setA.add(salad);  
23 setA.add(choc);  
25 setB.add(choc);  
26 setB.add(milk);
```

 SetExamples.java

```
setA = [Schokolade: 1 EUR, Salat: 2 EUR]  
setB = [Schokolade: 1 EUR, Milch: 2 EUR]
```

Sets

```
setA = [Schokolade: 1 EUR, Salat: 2 EUR]  
setB = [Schokolade: 1 EUR, Milch: 2 EUR]
```

► Erneutes hinzufügen ändert **nichts**

```
32 out.println("setA = " + setA);  
33 setA.add(salat);  
34 out.println("setA = " + setA);
```

SetExamples.java

```
setA = [Schokolade: 1 EUR, Salat: 2 EUR]  
setA = [Schokolade: 1 EUR, Salat: 2 EUR]
```

Sets

```
setA = [Schokolade: 1 EUR, Salat: 2 EUR]  
setB = [Schokolade: 1 EUR, Milch: 2 EUR]
```

► Teilmengen

```
38 out.println(setA.containsAll(setB));  
39 setA.add(milk);  
40 out.println(setA.containsAll(setB));
```

SetExamples.java

```
false  
true
```

- Zuerst: setB **keine** Teilmenge von setA
- Dann: **Hinzufügen** von milk zu setA...
- setB **ist** Teilmenge von setA



Sets

```
setA = [Schokolade: 1 EUR, Salat: 2 EUR, Milch: 2 EUR]  
setB = [Schokolade: 1 EUR, Milch: 2 EUR]
```

- ▶ Mengendifferenz $\text{setA} - \text{setB}$

```
44 setA.removeAll(setB);
```

SetExamples.java

```
setA = [Salat: 2 EUR]
```

- ▶ **Entfernt** alle Elemente aus setA ...
- ▶ die sich auch in **setB** befinden
- ▶ Tomaten in setA einfügen

```
setA.add(tomatoes);
```



Sets

```
setA = [Tomaten: 3 EUR, Salat: 2 EUR]  
setB = [Schokolade: 1 EUR, Milch: 2 EUR]
```

- Vereinigung $\text{setA} \cup \text{setB}$

51 `setA.addAll(setB);`

`SetExamples.java`

```
setA = [Schokolade: 1 EUR,  
        Milch: 2 EUR, Tomaten: 3 EUR,  
        Salat: 2 EUR]
```

- Fügt alle Elemente aus `setB...`
- in `setA` ein



Sets

```
setA = [Schokolade: 1 EUR, Milch: 2 EUR, Tomaten: 3 EUR, Salat: 2 EUR]  
setB = [Schokolade: 1 EUR, Milch: 2 EUR]
```

► Durchschnitt $\text{setA} \cap \text{setB}$

57 `setA.retainAll(setB);`

`SetExamples.java`

```
setA = [Schokolade: 1 EUR,  
        Milch: 2 EUR]
```

- Behält **nur die Elemente**, die sich ...
- in setA **und** setB befinden



Inhalt

Collection-Klassen

Sets

Sets als Mengen

HashSet vs. TreeSet

EnumSet

HashSet vs. TreeSet

	HashSet	TreeSet
Verfahren	Hashing	Rot-Schwarz-Baum
Sortiert	nein	ja
add/remove	$\mathcal{O}(1)$	$\mathcal{O}(\log n)$
contains	$\mathcal{O}(1)$	$\mathcal{O}(\log n)$
null erlaubt?	ja	nein

▶ [HashSet](#)

- ✓ Elemente müssen **nicht sortiert** vorliegen

▶ [TreeSet](#)

- ✓ Elemente müssen **sortiert** vorliegen
- ✓ Sortierung bleibt bei Operationen **erhalten**

▶ Hinweise

- ▶ Sortierung bewirkt höhere Laufzeit von $\mathcal{O}(\log n)$ in [TreeSet](#) zu $\mathcal{O}(1)$ in [HashSet](#)
- ▶ Details zu Sortierung später

Inhalt

Collection-Klassen

Sets

Sets als Mengen

HashSet vs. TreeSet


EnumSet

EnumSet

63 `enum Feature { TALL, HANDSOME, SMART, LIKABLE };`

SetExamples.java

► [EnumSet](#) für Mengen von `enum`-Werten

```
69 // snippet: enumSet
70  runEnumSetExample
71 EnumSet<Feature> myDreamPartner;
```

SetExamples.java

► Initialisierung über [EnumSet.of](#)

```
74 myDreamPartner = EnumSet.of(
75     Feature.HANDSOME, Feature.SMART);
```

SetExamples.java

```
myDreamPartner = [HANDSOME, SMART]
```

EnumSet

63 `enum Feature { TALL, HANDSOME, SMART, LIKABLE };`

SetExamples.java

► Initialisierung über `EnumSet.allOf`

81 `myDreamPartner = EnumSet.allOf(Feature.class);`

SetExamples.java

`myDreamPartner = [TALL, HANDSOME, SMART, LIKABLE]`

► Initialisierung über `EnumSet.noneOf`

87 `myDreamPartner = EnumSet.noneOf(Feature.class);`

SetExamples.java

`myDreamPartner = []`

EnumSet

```
63 enum Feature { TALL, HANDSOME, SMART, LIKABLE };
```

SetExamples.java

- Initialisierung über `EnumSet.range` (Reihenfolge in `enum`-Deklaration)

```
93 myDreamPartner = EnumSet.range(  
94     Feature.HANDSOME, Feature.LIKABLE);
```

SetExamples.java

```
myDreamPartner = [HANDSOME, SMART, LIKABLE]
```

- `EnumSet` implementiert `Set` (add, contains, remove, etc.)

```
100 myDreamPartner.remove(Feature.LIKABLE);
```

SetExamples.java

```
myDreamPartner = [HANDSOME, SMART]
```



```
63 enum Feature { TALL, HANDSOME, SMART, LIKABLE };
```

SetExamples.java

► Nützliche Methoden

- complementOf(EnumSet<E> other)

```
106 var noDreamPartner =  
107     EnumSet.complementOf(myDreamPartner);
```

SetExamples.java

```
noDreamPartner = [TALL, LIKABLE]
```

- copyOf(EnumSet<E> other) kopiert Auswahl

► Vorteile von `EnumSet<E>` zu `HashSet<E>`

- Reihenfolge wird eingehalten (nach `enum`-Deklaration)
- Schneller als `HashSet<E>`
- Praktischer durch `EnumSet`-Methoden

Inhalt

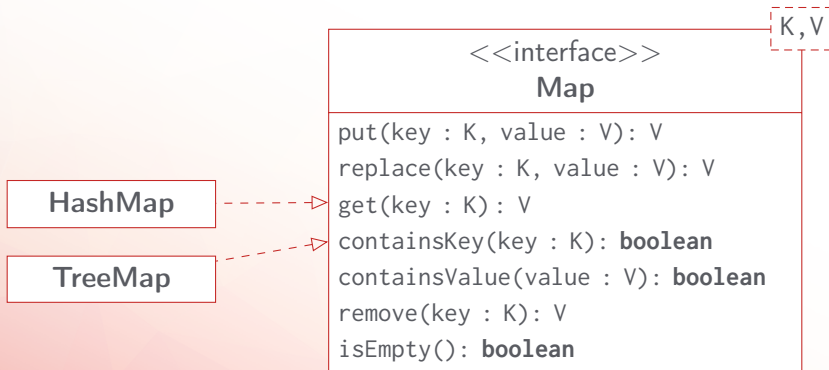
Collection-Klassen

Maps

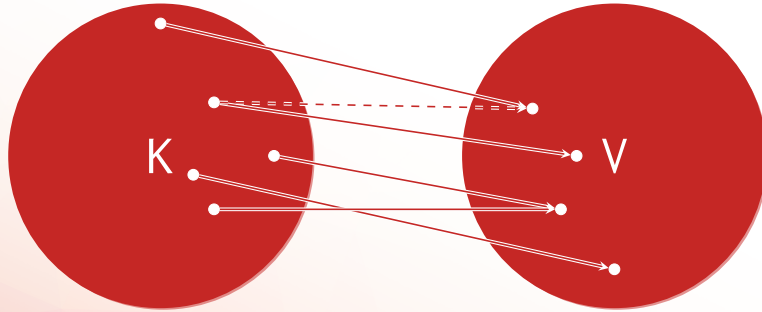
- Map-Operationen

- Views: Sichten auf Map

- HashMap vs. TreeMap



- ▶ [↗](#) `Map` (engl. „Abbildung“) bildet
 - ▶ **Schlüssel** vom Typ `K` auf...
 - ▶ **Werte** vom Typ `V` ab



- ▶ Schlüssel darf nur **einmal** vorkommen
- ▶ Werte dürfen **mehrmals** vorkommen
- ▶ Ähnlich zu **mathematischen Funktionen** $f : K \rightarrow V$
 - ▶ $f(k)$ ist **eindeutig**
 - ▶ **Aber** $f(k) = f(k')$ für $k \neq k'$ möglich
- ▶ **Nicht möglich:** Mehrere Werte für einen Schlüssel (gestrichelt)

Inhalt

Collection-Klassen


Maps

- Map-Operationen




- Views: Sichten auf Map

- HashMap vs. TreeMap

Map-Operationen: Erstellen

```
22  runMapOperationsExample  
23 var stock = new HashMap<Item,Integer>();
```

 MapExamples.java

- ▶ stock bildet ein Item auf seinen Bestand ab
- ▶  HashMap<K,V> hat zwei Typargumente
 - ▶ Schlüssel-Typ K, hier Item
 - ▶ Wert-Typ V, hier  Integer
- ▶ Zur Erinnerung
 - ▶ Typargumente müssen Referenztypen sein
 - ▶  Integer für **int** verwenden

Map-Operationen: Einfügen

```
27 stock.put(salad, 10);  
28 stock.put(choc, 50);  
29 stock.put(toiletpaper, 0);
```

MapExamples.java

```
{Schokolade: 1 EUR      = 50,  
 Toilettenpapier: 3 EUR = 0,  
 Salat: 2 EUR           = 10}
```

- ▶ `V put(K key, V value)`
 - ▶ **assoziiert** Schlüssel `key` mit Wert `value`
 - ▶ Gibt **alten Wert** zurück (wenn vorhanden, sonst **null**)

```
35 out.println(stock.put(salad, 15));
```

MapExamples.java

Gibt 10 aus

- ▶ **Praktisch:** `int` werden durch **Autoboxing** in `Integer` verpackt

Map-Operationen: Abrufen

39 `out.printf("Salat: %d Stück%n", stock.get(salad));`

MapExamples.java

Salat: 15 Stück

- ▶ `V get(K key)` liefert **Wert** zu Schlüssel `key`
- ▶ **null** wenn Eintrag **nicht vorhanden**

43 `out.printf("Milch: %d Stück%n", stock.get(milk));`

MapExamples.java

Milch: null Stück

Hier: **null** ist die Null-Referenz

Map-Operationen: containsKey und Mehrdeutigkeit von null

```
47 stock.put(milk, null);
```

MapExamples.java

```
{Schokolade: 1 EUR      = 50,  
  Milch: 2 EUR           = null,  
  Toilettenpapier: 3 EUR = 0,  
  Salat: 2 EUR           = 15}
```

- ▶ **null** als Wert zulässig
- ▶ **Problem:** Wenn `map.get(key) == null`
 - ▶ Eintrag key **nicht vorhanden?**
 - ▶ Oder: Wert für key ist **null?**
- ▶ **Besser:** Vorhandensein mit `containsKey` abfragen

```
52 if (stock.containsKey(milk))  
53     out.println("Eintrag vorhanden!");
```

MapExamples.java

Map-Operationen: remove

57 `stock.remove(milk);`

`MapExamples.java`

```
{Schokolade: 1 EUR      = 50,  
  Toilettenpapier: 3 EUR = 0,  
  Salat: 2 EUR          = 15}
```

- ▶ `V remove(K key)`
 - ▶ **entfernt** Eintrag zu `key`
 - ▶ **gibt** Wert zurück wenn vorhanden (**sonst null**)

Inhalt

Collection-Klassen

Maps

Map-Operationen

Views: Sichten auf Map

HashMap vs. TreeMap

Views: Sichten auf Map

- ▶ `Map<K,V>` kann man aus **unterschiedlichen** Perspektiven betrachten
 - ▶ `Set<K> keySet()`: **Menge** aller **Schlüssel**
 - ▶ `Collection<V> values()`: alle **Werte**
 - ▶ `Set<Map.Entry<K,V>> entrySet()` **Menge** von `Map`-Einträgen
 - ▶ `Map.Entry<K,V>` ist **innere Klasse**
 - ▶ Stellt **Tupel** (key, value) dar
 - ▶ **Keine Panik**: Beispiel kommt!
- ▶ Bevor es losgeht:

```
62 stock.put(milk, 15);
```

`MapExamples.java`

```
{Schokolade: 1 EUR      = 50,  
  Milch: 2 EUR           = 15,  
  Toilettenpapier: 3 EUR = 0,  
  Salat: 2 EUR           = 15}
```

keySet-View

```
67 for (Item item : stock.keySet())  
68     out.println(item);
```

MapExamples.java

Schokolade: 1 EUR
Milch: 2 EUR
Toilettenpapier: 3 EUR
Salat: 2 EUR

- ▶ [↗](#) **Set<K>** `keySet()` gibt Menge der **Schlüssel** zurück
- ▶ **Beispiel:** Schlüssel-Menge durchlaufen
- ▶ **Achtung:**
 - ▶ `keySet()` liefert „**Sicht**“ auf [↗](#) **Map**
 - ▶ Unterliegende Datenstruktur ist **immer noch** [↗](#) **Map**

keySet-View

```
72 Set<Item> keys = stock.keySet();  
73 keys.remove(choc);  
74 out.println(stock);
```

MapExamples.java

Milch: 2 EUR
Toilettenpapier: 3 EUR
Salat: 2 EUR

- ▶ `remove(key)` auf `keySet` entfernt Eintrag in unterliegender [Map](#)!
- ▶ `add(key)` auf `keySet` nicht unterstützt

values-View

```
78 for (Integer amount : stock.values())  
79     out.println(amount);
```

MapExamples.java

```
15  
0  
15
```

- ▶ `Collection<V> values()` liefert **alle Werte**
- ▶ Kein `Set<V>`, da Werte **mehrmals** vorkommen können
- ▶ Auch hier: `values` referenziert `Map`

```
83 Collection<Integer> amounts = stock.values();  
84 amounts.remove(0);  
85 out.println(stock);
```

MapExamples.java

```
{Milch: 2 EUR=15, Salat: 2 EUR=15}
```

entrySet-View

- ▶ Zur Erinnerung
 - ▶ ↗ Map ist eine Funktion $f : K \rightarrow V$
 - ▶ Alternativ: ↗ Map ist eine (spezielle) Relation
 - ▶ Menge von Tupeln $(k, v) \in K \times V$
- ▶ ↗ Map als Menge von Tupeln $(k, v) \in K \times V$
 - ▶ ↗ Set<Map.Entry<K,V>> entrySet()
 - ▶ ↗ Map.Entry<K,V> modelliert **Tupel** mit Schlüsseltyp K und Werttyp V

Entry	K, V
– key : K – value : V	
+ getKey(): K + getValue(): V + setValue(value : V) ...	

- ▶ ↗ Map.Entry ist innere Klasse von ↗ Map
- ▶ ↗ Set<...> weil keine Schlüsselduplikate erlaubt in ↗ Map

entrySet-View

```
89 for (Map.Entry<Item,Integer> entry : stock.entrySet())  
90     out.printf("(s, %s)%n", entry.getKey(), entry.getValue());
```

MapExamples.java

```
(Milch: 2 EUR, 15)  
(Salat: 2 EUR, 15)
```

► **Wieder gilt:** Unterliegende Datenstruktur ist ursprüngliche **Map**!

```
94 for (Map.Entry<Item,Integer> entry : stock.entrySet())  
95     entry.setValue(entry.getValue() + 5);
```

MapExamples.java

```
{Milch: 2 EUR=20, Salat: 2 EUR=20}
```

Inhalt

Collection-Klassen

Maps

Map-Operationen

Views: Sichten auf Map

HashMap vs. TreeMap

HashMap vs. TreeMap

	HashMap	TreeMap
Verfahren	Hashing	Rot-Schwarz-Baum
Sortiert	nein	ja
put/remove	$\mathcal{O}(1)$	$\mathcal{O}(\log n)$
containsKey	$\mathcal{O}(1)$	$\mathcal{O}(\log n)$

► [HashMap](#)

- ✓ Elemente müssen **nicht sortiert** vorliegen

► [TreeMap](#)

- ✓ Elemente müssen nach **Schlüssel sortiert** vorliegen
- ✓ Sortierung bleibt bei Operationen **erhalten**

► Hinweise

- Sortierung bewirkt höhere Laufzeit von $\mathcal{O}(\log n)$ in [TreeMap](#) zu $\mathcal{O}(1)$ in [HashMap](#)
- Details zu Sortierung später

Inhalt

Collection-Klassen

Collection-Factories

List


Set

Map

Unmodifiable

Collection-Factories

- ▶ Manuelles Erstellen von **Listen** aufwändig

```
23  runListCreationNoFactory  
24 var items = new ArrayList<Item>();  
25 items.add(salad);  
26 items.add(choc);  
27 items.add(milk);  
28 items.add(toiletpaper);
```

 FactoriesExamples.java

- ▶ Zur Erinnerung: Arrays haben **Literale**

```
36 Item[] itemsArray =  
37     new Item[]{salad, choc, milk, toiletpaper};
```

 FactoriesExamples.java

- ▶ Konvertierung in  **List** mit  **Arrays.asList()**

```
41 List<Item> items = Arrays.asList(itemsArray);
```

 FactoriesExamples.java

Inhalt

Collection-Klassen

Collection-Factories

List

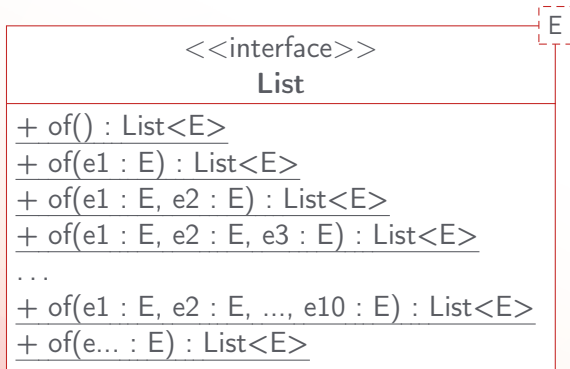
Set

Map


Unmodifiable

Collection-Factories: List

- ▶ Geht das auch **in einem**?
- ▶ **Ja**: Statische Factory-Methoden



- ▶ Beispiel

```
49  runListCreationFactory  
50 List<Item> items =  
51     List.of(salad, choc, milk, toiletpaper);
```

 `FactoriesExamples.java`

Inhalt

Collection-Klassen

Collection-Factories

List


Set

Map

Unmodifiable

Collection-Factories: Set


- ▶ Geht auch für [↗](#) Set

```
58  runSetCreationFactory  
59 Set<Item> items =  
60     Set.of(salad, choc, milk, toiletpaper);
```

 FactoriesExamples.java

[Milch: 2 EUR, Toilettenpapier: 3 EUR, Salat: 2 EUR, Schokolade: 1 EUR]

- ▶ Keine Duplikate erlaubt

```
68  runSetCreationDuplicates  
69 Set<Item> items =  
70     Set.of(salad, choc, milk, toiletpaper, milk);
```

 FactoriesExamples.java

Exception: „duplicate element: Milch: 2 EUR“

Inhalt

Collection-Klassen

Collection-Factories

List

Set

Map


Unmodifiable

Collection-Factories: Map

► Und auch für Map

►  `Map.of(key1, value1, key2, value2, ...)`

► Beispiel

```
76  runMapCreationFactory  
77 Map<Item,Integer> stock =  
78     Map.of(  
79         salad, 10,  
80         choc, 40,  
81         milk, 20,  
82         toiletpaper, 0);
```

 `FactoriesExamples.java`

```
{Milch: 2 EUR           = 20,  
  Toilettenpapier: 3 EUR = 0,  
  Salat: 2 EUR          = 10,  
  Schokolade: 1 EUR     = 40}
```

Collection-Factories: Map

- ▶ Zur Erinnerung: `Map<K,V>` kann als Menge von `Map.Entry<K,V>` aufgefasst werden
- ▶ `Map.Entry<K,V>` ist **Tupel** aus **Schlüssel** und **Wert**
- ▶ Entsprechende Factory-Methode

89  **runMapCreationEntryFactory**

```
90 Map<Item,Integer> stock =  
91     Map.ofEntries(  
92         entry(salad, 10),  
93         entry(choc, 40),  
94         entry(milk, 20),  
95         entry(toiletpaper, 0));
```

 FactoriesExamples.java

- ▶ `entry(key,value)`: **statische Methode** in `Map`-Interface
 - ▶ Erstellt `Map.Entry` aus **Schlüssel** und **Wert**
 - ▶ **Statischer Import** für kurze Schreibweise

```
12 import static java.util.Map.entry;
```

 FactoriesExamples.java

Inhalt

Collection-Klassen

Collection-Factories

List


Set

Map

Unmodifiable

Collection-Factories: Unmodifiable


- ▶ Wichtiger Hinweis: Alle mit of erstellten Collections sind **unmodifiable**
 - ▶ Erlauben **keine** Änderungen

```
102  runListFactoryModify  
103 List<Item> items = List.of(salad, choc);  
104 items.add(toiletpaper);
```

 FactoriesExamples.java

Exception: „UnsupportedOperationException“

- ▶ „Umwandlung“ in **modifizierbare** Liste mit **Konstruktor**

```
110  runListFactoryModifiable  
111 List<Item> roItems = List.of(salad, choc);  
112 var items = new ArrayList<Item>(roItems);  
113 items.add(toiletpaper);
```

 FactoriesExamples.java

[Salat: 2 EUR, Schokolade: 1 EUR, Toilettenpapier: 3 EUR]

- ▶ Entsprechend für  Set und  Map

Inhalt


Collection-Klassen

Nicht-Modifizierbare Collections

Nicht-Modifizierbare Collections

▶ [ArrayList](#), [HashMap](#) und Co. können von jedem verändert werden

▶ Beispiel

```
24  runModifiableMapExample  
25 var stock = new HashMap<Item,Integer>();  
26 stock.put(salad, 10);  
27 stock.put(choc, 50);  
28 stock.put(toiletpaper, 50);  
30 printMap(stock);
```

 UnmodifiableExample.java

▶ printMap soll nur ausgeben (macht aber **mehr**)

```
16 public static void printMap(Map<Item,Integer> stock) {  
17     out.println(stock);  
18     stock.remove(toiletpaper); // muahahaha!  
19 }
```

 UnmodifiableExample.java


Nicht-Modifizierbare Collections

► Ausgabe


```
{Schokolade: 1 EUR=50, Toilettenpapier: 3 EUR=50, Salat: 2 EUR=10}  
{Schokolade: 1 EUR=50, Salat: 2 EUR=10}
```

► Wie kann man das verhindern?

► Unmodifiable Collections

```
41  runUnmodifiableMapExample  
42 Map<Item,Integer> unmodifiableStock =  
43     Collections.unmodifiableMap(stock);  
45 printMap(unmodifiableStock);
```

 UnmodifiableExample.java

- Jetzt **UnsupportedOperationException** in printMap
- unmodifiableMap liefert **nicht-modifizierbare Sicht** auf  Map
- Exception weist auf **Programmierfehler** hin

Nicht-Modifizierbare Collections

- ▶ Klasse `Collections` beinhaltet **Hilfsmethoden**
 - ▶ **Nicht-Modifizierbare Collections**
 - ▶ `Collection<T> unmodifiableCollection(Collection<T> c)`
 - ▶ `List<T> unmodifiableList(List<T> c)`
 - ▶ `Map<T> unmodifiableMap(Map<T> c)`
 - ▶ Mehr hilfreiche Methoden
 - ▶ `binarySearch`: binäre Suche
 - ▶ `max/min`: Maximum/Minimum
 - ▶ `nCopies`: erstellt Liste mit n-mal einem Element
 - ▶ `reverse`: dreht Liste um
 - ▶ `shuffle`: mischt Liste zufällig
 - ▶ `sort`: sortiert Liste
 - ▶ `swap`: tausche Elemente
 - ▶ ...

Inhalt

Collection-Klassen

Geschachtelte Collections

Befüllen

Durchlaufen

Weitere Beispiele

Geschachtelte Collections

- Wir wollen **Kategorien** von Produkten (Items) modellieren

Kategorie	Produkte	Preis
Schokolade	Milka Vollmilch	2
	Milka Nuss	2
	Romy	3
Gemüse	Möhren	3
	Kartoffeln	2
	Salat	2
Toilettenpapier	Vella 3-lagig	3
	Happy End soft	2

Geschachtelte Collections

► Kategorien als `enum`

```
14 public enum Category {  
15     CHOCOLATE, VEGGIES, TOILETPAPER }
```

📄 NestedCollections.java

► Idee für Modellierung

- ↗ `Map` bildet `Category` auf ↗ `List` e von Produkten ab
- Geschachtelte Collection

```
Map<Category, List<Item>>
```

- Äußerer Typ: ↗ `Map` bildet `Schlüssel` `Category` auf `Wert` ↗ `List` ab
- Innerer Typ: Werte sind ↗ `List` en von Produkten

► Instanziierung

```
43 Map<Category, List<Item>> itemsForCategory =  
44     new HashMap<Category, List<Item>>();
```

📄 NestedCollections.java

Inhalt

Collection-Klassen

Geschachtelte Collections

Befüllen

Durchlaufen

Weitere Beispiele






Befüllen

Befüllen der Kategorie „Schokolade“

```
48 var chocolates = new ArrayList<Item>();  
49 chocolates.add(new Item("Milka Vollmilch", 2));  
50 chocolates.add(new Item("Milka Nuss", 2));  
51 chocolates.add(new Item("Romy", 3));  
53 itemsForCategory.put(Category.CHOCOLATE, chocolates);
```

 NestedCollections.java

► Vorgehensweise






-  List erstellen ( ArrayList)
-  List befüllen
-  List mit Schlüssel in  Map assoziieren

Befüllen

Befüllen der Kategorie „Gemüse“

```
57 var veggies = new LinkedList<Item>();  
58 veggies.add(new Item("Möhren", 3));  
59 veggies.add(new Item("Kartoffeln", 2));  
60 veggies.add(new Item("Salat", 2));  
62 itemsForCategory.put(Category.VEGGIES, veggies);
```

 NestedCollections.java


- ▶ Hier:  `LinkedList` (vorher  `ArrayList`)
 - ▶ Wertetyp ist  `List`
 - ▶ Kompatibel mit  `ArrayList` und  `LinkedList`
- ▶ Tipp: Bei Deklaration möglichst **allgemeinen** Typ angeben

Befüllen

Befüllen der Kategorie „Toilettenpapier“

```
66 itemsForCategory.put(Category.TOILETPAPER,  
67     List.of(  
68         new Item("Vella 3-lagig", 3),  
69         new Item("Happy End soft", 2)));
```

 NestedCollections.java

- ▶ Vorgehen hier
 - ▶ Keine eigene **Variable** für Liste
 - ▶ Direkte Assoziierung
 - ▶ Erstellung mit  **List**-Factory
- ▶ Kürzer
- ▶ Aber bei vielen Elementen **unübersichtlich**

Inhalt

Collection-Klassen

Geschachtelte Collections


Befüllen

Durchlaufen

Weitere Beispiele

Durchlaufen

Durchlaufen mit geschachtelter Schleife

```
19  runNestedCollectionsExample  
20 public static void printCategories(  
21     Map<Category, List<Item>> itemsForCategory){  
23     for (Category category : itemsForCategory.keySet()){  
25         out.printf("Kategorie: %s%n", category.name());  
27         List<Item> items = itemsForCategory.get(category);  
29         for (Item item : items)  
30             out.printf(" - %s%n", item);  
32         out.println();  
34     }  
35 }
```

 NestedCollections.java

Durchlaufen

▶ Äußere Schleife

```
for (Category category : itemsForCategory.keySet()) ...
```

- ▶ Durchläuft **äußere Collection** (↗ Map)
- ▶ Schleifenvariable: **Kategorie** (Schlüssel)

▶ Innere Schleife

- ▶ Durchläuft **innere Collection** (↗ List)
- ▶ **Ermittelt** Items der Kategorie

```
List<Item> items = itemsForCategory.get(category);
```

- ▶ Durchläuft die ↗ List e

```
for (Item item : items)
```

- ▶ Ähnlich zu **zwei-/mehrdimensionalen Arrays**

Ausgabe

Kategorie: TOILETPAPER

- Vella 3-lagig: 3 EUR
- Happy End soft: 2 EUR

Kategorie: VEGGIES

- Möhren: 3 EUR
- Kartoffeln: 2 EUR
- Salat: 2 EUR

Kategorie: CHOCOLATE

- Milka Vollmilch: 2 EUR
- Milka Nuss: 2 EUR
- Romy: 3 EUR

Inhalt

Collection-Klassen

Geschachtelte Collections

Befüllen

Durchlaufen

Weitere Beispiele

Weitere Beispiele

- ▶ `List<List<Double>>`: Messwerte pro Tag

```
l.get(10).get(4) // 5. Messwert am 11. Tag
```

- ▶ `Map<City, Set<Person>>`: Zuordnung Stadt zu Bewohnern

```
m.get(landshut).contains(mueller)
```

- ▶ `List<List<String>>`: Werte in einer CSV-Datei

```
l.get(2).get(5) // 6. Wert in 3. Zeile
```

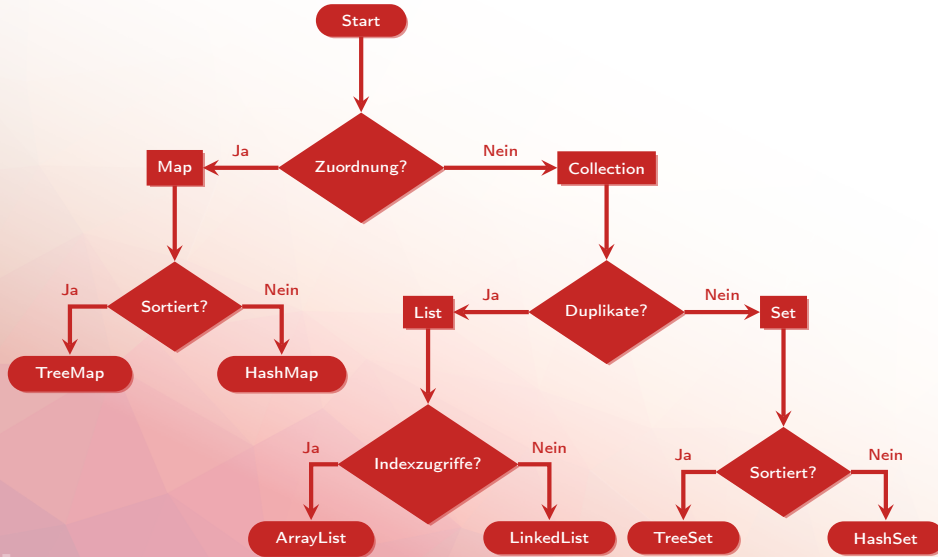
- ▶ `Map<Person, Map<Person, Relationship>>`: Personen Beziehungsstatus

```
m.get(haensel).get(gretel) // == Relationship.Sibling
```

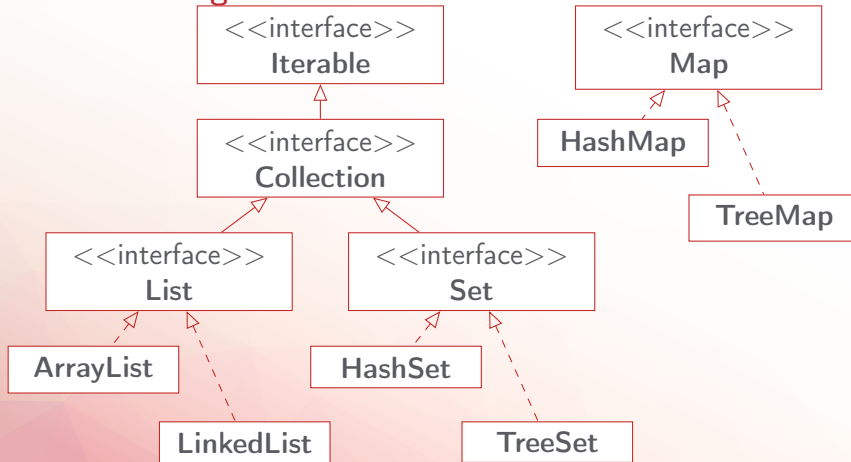
Inhalt

Collection-Klassen Zusammenfassung

Zusammenfassung



Zusammenfassung



- ▶ Je nach **Anwendung** passende **Datenstruktur** wählen
- ▶ Erstellen mit of-Factory-Methoden
- ▶ Unveränderliche Varianten über `Collections.unmodifiable*`
- ▶ Geschachtelte Collections für **bestimmte Anwendungen**

Inhalt

Iteratoren

- Iteratoren und for-each-Schleifen
- Allgemeine Verwendung von Iteratoren
- Eigene Iteratoren
- Zusammenfassung

Inhalt

Iteratoren

Iteratoren und for-each-Schleifen

Iterable und Iterator

for-each und Iterator

Inhalt

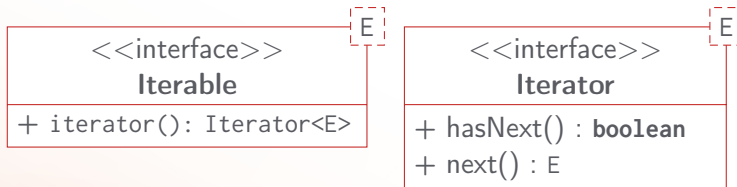
Iteratoren

Iteratoren und for-each-Schleifen

Iterable und Iterator

for-each und Iterator

Iterable und Iterator



- ▶ **Iterable**<E> — „durchlaufbare“ Strukturen
 - ▶ Erzeugen **Iterator**<E>
 - ▶ **Collection**s, „was aufzählbar ist“
- ▶ **Iterator**<E> — ein **konkreter** Durchlauf
 - ▶ E ist der **Elementtyp**
 - ▶ `hasNext()` ist **true** wenn nächstes Element **existiert**
 - ▶ `next()` **liefert** nächstes Element
 - ▶ Erster Aufruf `next()` liefert **erstes Element**
- ▶ Veranschaulichung:
 - ▶ **Iterator** entspricht in **Cursor** in Texteditor
 - ▶ `next()` liefert Zeichen **rechts neben** Cursor und **navigiert nach rechts**

Inhalt


Iteratoren

Iteratoren und for-each-Schleifen

Iterable und Iterator

for-each und Iterator

Manuelles Durchlaufen mit Iterator


```
14  runIteratorManualExample  
15 List<String> l = List.of("La", "Dee", "Da");  
17 for (Iterator<String> i = l.iterator(); i.hasNext();){  
18     String s = i.next();  
19     out.println(s);  
20 }
```

 IteratorExamples.java



La
Dee
Da


for-each-Schleifen

```
26  runIteratorForEachExample  
27 List<String> l = List.of("La", "Dee", "Da");  
29 for (String s : l)  
30     out.println(s);
```

 IteratorExamples.java

- ▶ Äquivalent zu vorheriger Version
- ▶ Allgemein

```
for ( Typ laufvariable : iterable )
```

- ▶ Typ kann auch **var** sein (wenn Typ **ablesbar**)
- ▶ iterable muss  **Iterable**<Typ> implementieren
- ▶ Oder: Array sein (klassische **for**-Schleife)
- ▶ Erzeugt **implizit** Iterator mit `iterable.iterator()`

Inhalt

Iteratoren

- Allgemeine Verwendung von Iteratoren

 - Veränderungen während der Iteration

 - Das ganze Iterator-Interface

 - ListIterator — Iterator auf Steroiden

Allgemeine Eigenschaften von Iteratoren



- ▶ Start immer am Anfang
- ▶ Nur „von links nach rechts“, keine Umkehrung möglich
- ▶ Immer nur ein Element (keine Sprünge)
- ▶ Iterator am Ende verbraucht
 - ▶ `hasNext()` liefert `false`
 - ▶ `next()` wirft `NoSuchElementException`
 - ▶ Kein Reset möglich

„Paralleles“ Durchlaufen

- ▶ **Parallele** Existenz von **Iteratoren** erlaubt


```
Iterator<String> i1 = l.iterator();  
Iterator<String> i2 = l.iterator();
```

- ▶ Existieren **unabhängig** voneinander



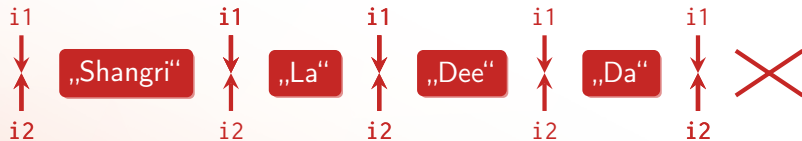
- ▶ Jeder **Iterator** verwaltet **eigenen** Fortschritt
- ▶ `next()` hat nur **Auswirkung** auf einen Iterator

„Paralleles“ Durchlaufen

```
36  runIteratorParallelExample  
37 List<String> l = List.of("Shangri", "La", "Dee", "Da");  
38 Iterator<String> i1 = l.iterator();  
39 Iterator<String> i2 = l.iterator();  
41 while (i1.hasNext() || i2.hasNext()){  
43     if (i1.hasNext())  
44         out.println("i1: " + i1.next());  
46     if (i2.hasNext())  
47         i2.next(); // überspringen  
49     if (i2.hasNext())  
50         out.println("i2: " + i2.next());  
51 }
```

 IteratorExamples.java

„Paralleles“ Durchlaufen



i1: Shangri

i2: La

i1: La

i2: Da

i1: Dee

i1: Da

Inhalt

Iteratoren

Allgemeine Verwendung von Iteratoren


Veränderungen während der Iteration

Das ganze Iterator-Interface




ListIterator — Iterator auf Steroiden

Veränderung der Struktur

- ▶ Veränderung der Datenstruktur macht alle existierenden  Iterator-Instanzen ungültig


```
58  runIteratorChangeStructureExample
59 for (Iterator<String> i = l.iterator(); i.hasNext());{
60     String s = i.next();
61     out.println(s);
62     l.add("Da");
63 }
```

 IteratorExamples.java

- ▶ Verursacht  ConcurrentModificationException
- ▶ add verändert unterliegende Datenstruktur
- ▶ Jeder aktuelle  Iterator wird ungültig
- ▶ Nächster Aufruf von next() oder hasNext() erzeugt  ConcurrentModificationException
- ▶ Auch für remove und alle Operationen, die die Datenstruktur verändern


Veränderung des Inhalts

- ▶ Veränderung des Inhalts ist erlaubt

```
69  runIteratorChangeContentExample  
70 List<String> l = new ArrayList<String>(List.of("La", "Dee", "Da"));  
71 for (Iterator<String> i = l.iterator(); i.hasNext();){  
72     l.set(0, "Dum");  
73     String s = i.next();  
74     out.println(s);  
75 }
```

 IteratorExamples.java

Dum
Dee
Da

- ▶ Keine  ConcurrentModificationException
- ▶ Grund: Struktur bleibt gleich

Inhalt

Iteratoren

Allgemeine Verwendung von Iteratoren

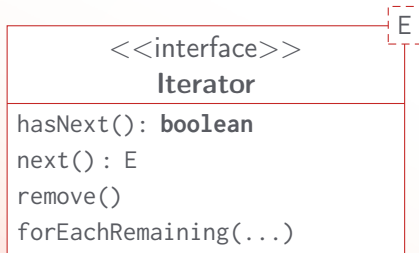
Veränderungen während der Iteration

Das ganze Iterator-Interface

ListIterator — Iterator auf Steroiden


Die ganze Wahrheit über Iterator

- ▶ [Iterator](#) kann mehr als next und hasNext




- ▶ remove
 - ▶ entfernt das **zuletzt** besuchte Element
 - ▶ Default-Implementierung: wirft [UnsupportedOperationException](#)
- ▶ forEachRemaining für uns **nicht weiter relevant**

remove mit Iterator

```
81  runIteratorRemoveExample  
82 List<String> l = new ArrayList<String>(  
83     List.of("Shangri", "La", "Dee", "Da"));  
85 for (Iterator<String> i = l.iterator(); i.hasNext();){  
86     String s = i.next();  
88     if (s.length() <= 2)  
89         i.remove();  
90 }
```


 IteratorExamples.java

[Shangri, Dee]

- ▶ Entfernt Einträge der Länge <= 2
- ▶ Hier keine  ConcurrentModificationException

remove mit zwei Iteratoren

remove macht **andere** aktive Iteratoren **ungültig**

```
100  runIteratorRemoveConcurrentExample  
101 Iterator<String> i1 = l.iterator();  
102 Iterator<String> i2 = l.iterator();  
104 while (i1.hasNext() || i2.hasNext()){  
106     if (i1.hasNext() && i1.next().length() <= 2)  
107         i1.remove();  
109     if (i2.hasNext())  
110         out.println(i2.next());  
112 }
```

 IteratorExamples.java

Shangri

FEHLER: ConcurrentModificationException

remove mit zwei Iteratoren



- ▶ `i1.next()`
- ▶ `i2.next()` (mit `out.println("Shangri");`)
- ▶ `i1.next()` und `i1.remove()`
- ▶ `i2.hasNext()` → [↗ ConcurrentModificationException](#)
- ▶ `remove` nur mit **einem** aktiven [↗](#) Iterator möglich

Inhalt

Iteratoren

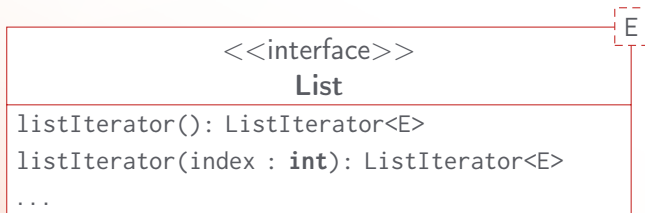
Allgemeine Verwendung von Iteratoren

Veränderungen während der Iteration

Das ganze Iterator-Interface

ListIterator — Iterator auf Steroiden

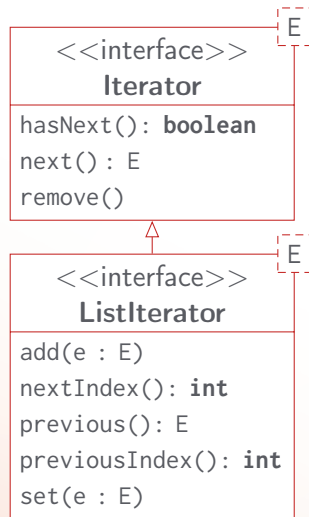
ListIterator



- ▶ `List` bietet erweiterten `Iterator` an
 - ▶ `ListIterator` erweitert `Iterator`-Interface
 - ▶ `List.listIterator()` erstellt `ListIterator`
 - ▶ `List.listIterator(int index)` erstellt `ListIterator` mit erster Position `index`

ListIterator

- ▶ Erweitert ↗ Iterator
- ▶ Hinzufügen möglich
- ▶ Laufen in **beide** Richtungen möglich
- ▶ Zugriff auf **Index**
- ▶ **Setzen** des Inhalts



Arbeiten mit ListIterator

```
119  runListIteratorLeftRightExample  
120 List<String> l = new ArrayList<String>(  
121     List.of("Shangri", "La", "Dee", "Da"));
```

 IteratorExamples.java

- ▶ Erstellen des  ListIterators vor Index 2

```
125 ListIterator<String> i = l.listIterator(2);
```

 IteratorExamples.java

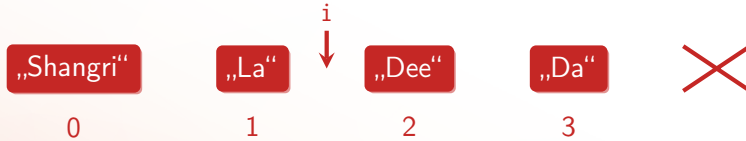


Arbeiten mit ListIterator

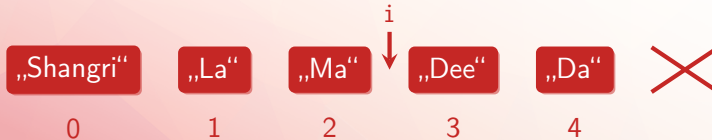


- ▶ `next()`
 - ▶ Gibt **nächstes** Element zurück (Bsp.: Dee)
 - ▶ Bewegt Iterator **hinter nächstes** Element
- ▶ `nextIndex()`
 - ▶ Index **nächstes** Element (Bsp.: 3)
 - ▶ `l.size()` wenn Iterator am **Ende**
- ▶ `previous()`
 - ▶ Gibt **vorheriges** Element zurück (Bsp.: Dee)
 - ▶ Bewegt Iterator **vor vorheriges** Element
- ▶ `previousIndex()`
 - ▶ Index **vorheriges** Element (Bsp.: 1)
 - ▶ -1 wenn Iterator am **Anfang**

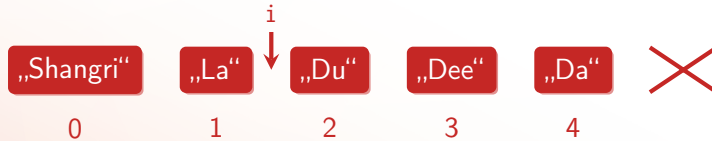
Arbeiten mit ListIterator



- ▶ `add("Ma")`
 - ▶ Neues Element an der Stelle von `ListIterator`
 - ▶ `Iterator` wird **nach** neuem Element platziert

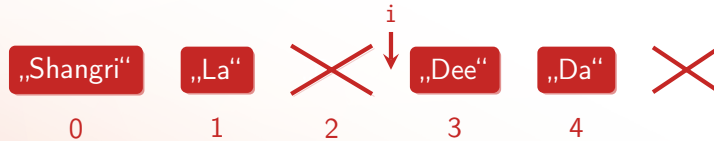


Arbeiten mit ListIterator



- ▶ `i.set`
 - ▶ Ersetzt das Element, das **zuletzt** von `next` oder `previous` **zurückgegeben** wurde
 - ▶ Vor `set` **muss** `next` oder `previous` **aufgerufen** werden
- ▶ `i.previous()` (`== "Ma"`)
- ▶ `i.set("Du")` (ersetzt "Ma" durch "Du")

Arbeiten mit ListIterator



- ▶ `i.remove`
 - ▶ Entfernt das Element, das zuletzt von `next` oder `previous` zurückgegeben wurde
 - ▶ Vor `remove` muss `next` oder `previous` aufgerufen werden
- ▶ `i.next()` (`== "Du"`)
- ▶ `i.remove()` (entfernt "Du")

Arbeiten mit ListIterator

- ▶ Achtung bei parallelem Durchlaufen
- ▶ Strukturänderungen machen anderen Iteratoren ungültig

Operation	Strukturänderung
next/previous	Nein
nextIndex/previousIndex	Nein
hasPrevious/hasNext	Nein
add	Ja
remove	Ja
set	Nein

Inhalt

Iteratoren

Eigene Iteratoren

- Eigene Klasse Iterable machen

- Fremde Klassen iterierbar machen

- Iteratoren ohne Datenstrukturen

Inhalt

Iteratoren

Eigene Iteratoren

- Eigene Klasse Iterable machen

- Fremde Klassen iterierbar machen

- Iteratoren ohne Datenstrukturen

Eigene Klasse Iterable machen

- ▶ Klasse Stock beinhaltet Ware in Lager

Stock
- items : Item[]
+ Stock(item : Item[])

- ▶ Wir wollen diese Klasse [↗](#) Iterable<Item> machen
- ▶ [↗](#) Iterator listet Waren auf

Kochrezept: Eigene Klasse **Iterable** machen

Stock
- items : Item[]
+ Stock(item : Item[])

Schritte um eigene Klasse Container **iterierbar** zu machen

1. Innere Klasse ContainerIterator definieren

- ▶ **private**
- ▶ **implements** Iterator<E>
- ▶ **Attribut** für **aktuelle Position** deklarieren
- ▶ hasNext() und next() **implementieren**

2. Container-Klasse **modifizieren**

- ▶ **implements** Iterable<E>
- ▶ iterator() **implementieren**

Stock iterierbar machen

► Klasse Stock

```
public class Stock{  
    private Item[] items;  
    public Stock(Item[] items){  
        this.items = items;  
    }  
}
```

Stock iterierbar machen: 1. Schritt

- ▶ **Innere** Klasse StockIterator deklarieren

```
26 private class StockIterator implements Iterator<Item>{
```

Stock.java

- ▶ **Aktuelle Position** des Iterators als Attribut

```
30 private int nextIndex;
```

Stock.java

- ▶ **Konstruktor**

```
34 private StockIterator(){  
35     this.nextIndex = 0;  
36 }
```

Stock.java

Stock iterierbar machen: 1. Schritt

- ▶ `hasNext()` prüft ob `nextItem` noch nicht am Ende ist

```
50 @Override
51 public boolean hasNext(){
52     return nextIndex < Stock.this.items.length;
53 }
```

Stock.java

- ▶ `next()`
 - ▶ `NoSuchElementException` wenn `hasNext() == false`
 - ▶ Sonst: nächstes Item liefern und `nextItem++`

```
40 @Override
41 public Item next(){
42     if (!hasNext())
43         throw new NoSuchElementException("End reached");
44     return Stock.this.items[nextIndex++];
45 }
46 }
```

Stock.java

Stock iterierbar machen: 2. Schritt

- ▶ Stock **implementiert** `Iterable<Item>`

```
7 public class Stock implements Iterable<Item> {
```

Stock.java


- ▶ `iterator()` **erzeugt** StockIterator-Instanz

```
19 @Override  
20 public Iterator<Item> iterator(){  
21     return new StockIterator();  
22 }
```

Stock.java

Stock iterierbar machen: Test

► Testprogramm

```
15  runStockIteratorExample  
16 Stock stock = new Stock(  
17     new Item[] {salad, choc, milk, toiletpaper});  
19 for (Item item : stock)  
20     out.println(item);
```

 OwnIteratorExamples.java

► Ausgabe

```
Salat: 2 EUR  
Schokolade: 1 EUR  
Milch: 2 EUR  
Toilettenpapier: 3 EUR
```

► Es funktioniert!

Inhalt

Iteratoren

Eigene Iteratoren

Eigene Klasse Iterable machen

Fremde Klassen iterierbar machen

Iteratoren ohne Datenstrukturen

Fremde Klassen iterierbar machen

```
String s = "YMCA!";  
for (char c : s) // FEHLER  
    out.println(c);
```

- ▶ Wir möchten die Zeichen eines Strings durchlaufen
- ▶ Problem: `String` implementiert `Iterable` nicht
- ▶ Und: `String` ist `final`
- ▶ Idee:
 - ▶ Iterierbare „Wrapper“-Klasse
 - ▶ Hat Referenz auf `String`
- ▶ Kochrezept wie oben!

IterableString

- ▶ IterableString implementiert ↗ Iterable<Character>

```
7 public class IterableString
8     implements Iterable<Character> {
```

↳ IterableString.java

- ▶ „Verpackt“ ↗ String

```
12 private String string;
14 public IterableString(String string){
15     this.string = string;
16 }
```

↳ IterableString.java

- ▶ Innere Klasse StringIterator in iterator() erzeugen

```
21 public Iterator<Character> iterator(){
22     return new StringIterator();
23 }
```

↳ IterableString.java

StringIterator

- ▶ StringIterator durchläuft Zeichen des ↗ String s

```
27 private class StringIterator
28     implements Iterator<Character>{
```

IterableString.java

- ▶ nextIndex ist Index des nächsten Zeichens

```
32 private int nextIndex;
```

IterableString.java

- ▶ Konstruktor: Start bei 0

```
36 private StringIterator(){
37     nextIndex = 0;
38 }
```

IterableString.java

StringIterator

- ▶ `hasNext()` prüft ob `nextItem` noch nicht am Ende ist

```
42 @Override
43 public boolean hasNext(){
44     return nextIndex < IterableString.this.string.length();
45 }
```

IterableString.java


- ▶ `next()`
 - ▶ `NoSuchElementException` wenn `hasNext() == false`
 - ▶ Sonst: nächsten `Character` im `String` liefern und `nextItem++`

```
49 @Override
50 public Character next(){
51     if (!hasNext())
52         throw new NoSuchElementException("End reached");
54     return IterableString.this.string.charAt(nextIndex++);
55 }
```

IterableString.java

IterableString: Test

► Test

```
37  runIterableStringExample  
38 IterableString is = new IterableString("YMCA!");  
40 for (char c : is)  
41     out.println(c);
```

 OwnIteratorExamples.java

► Ausgabe

```
Y  
M  
C  
A  
!
```

Inhalt

Iteratoren

Eigene Iteratoren

Eigene Klasse Iterable machen

Fremde Klassen iterierbar machen

Iteratoren ohne Datenstrukturen


range in Python

- ▶ Python hat range Funktion



```
s = 0
for i in range(1,100):
    s += i
```

- ▶ range liefert  Iterator
- ▶ Zahlen 1,2,...,99

- ▶ So etwas wollen wir auch in Java

```
27  runRangeIteratorExample
28 int s = 0;
29 for (int i : range(1,100))
30     s += i;
```

 OwnIteratorExamples.java

- ▶ Idee
 - ▶ Klasse Range implementiert  Iterable<Integer>
 - ▶ Liefert  Iterator für gegebenes Intervall
- ▶ Kochrezept wie oben!

Range

- ▶ Range implementiert [↗](#) `Iterable<Integer>`

```
7 public class Range implements Iterable<Integer>{
```

[↗](#) Range.java

- ▶ Hat start- und end-Index

```
11 private int start;  
12 private int end;  
13 public Range(int start, int end){  
14     this.start = start;  
15     this.end = end;  
16 }
```

[↗](#) Range.java

- ▶ Innere Klasse `RangeIterator` in `iterator()` erzeugen

```
21 @Override public Iterator<Integer> iterator(){  
22     return new RangeIterator();  
23 }
```

[↗](#) Range.java

RangeIterator

- ▶ RangeIterator **durchläuft Zahlen** start bis end-1

```
33 private class RangeIterator  
34     implements Iterator<Integer>{
```

Range.java

- ▶ next ist **nächste Zahl**

```
38 private int next;  
40 private RangeIterator(){  
41     this.next = Range.this.start;  
42 }
```

Range.java

RangeIterator

- ▶ `hasNext()` prüft ob next noch nicht am Ende ist

```
48 @Override
49 public boolean hasNext(){
50     return next < Range.this.end;
51 }
```

Range.java

- ▶ `next()`
 - ▶ `NoSuchElementException` wenn `hasNext() == false`
 - ▶ Sonst: nächste Zahl liefern und `next++`

```
55 @Override
56 public Integer next(){
57     if (!hasNext())
58         throw new NoSuchElementException("Reached end");
59     return next++;
60 }
61 }
```

Range.java


Range: Test

► Statische Factory-Methode `Range.range`

```
27 public static Range range(int start, int end){  
28     return new Range(start, end);  
29 }
```

Range.java

► Test (mit `import static Range.range`)

```
27  runRangeIteratorExample  
28 int s = 0;  
29 for (int i : range(1,100))  
30     s += i;
```

OwnIteratorExamples.java

► Ausgabe

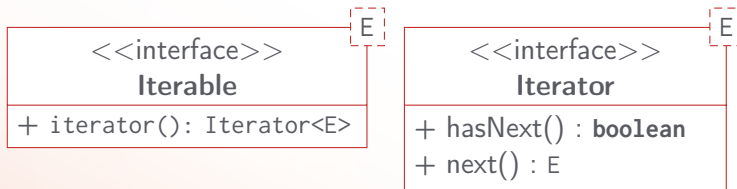
4950

Inhalt

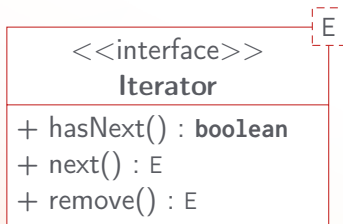
Iteratoren

Zusammenfassung

Zusammenfassung



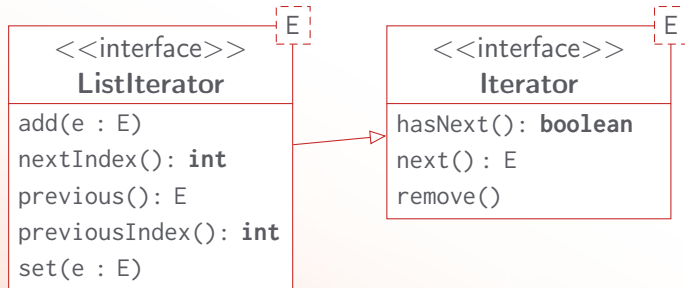
- ▶ [↗](#) **Iterable<E>**-Interface für **iterierbare** Strukturen
 - ▶ [↗](#) **Collection**s (Datenstrukturen), **aufzählbare** Strukturen
 - ▶ Unterstützen **for-each**-Schleife
- ▶ [↗](#) **Iterator<E>**
 - ▶ `hasNext()`: **true** wenn **Elemente übrig**, sonst **false**
 - ▶ `next()`: nächstes Element **und** Sprung
 - ▶ Instanz: **ein konkreter Durchlauf**
 - ▶ [↗](#) **Collections**: Am Ende verbraucht



► ↗ Collections

- Struktur ändernde Operationen (add, etc.): **alle aktiven** Iteratoren **ungültig**
- Inhalt ändernde Operationen (set, etc.): **kein** Problem
- ↗ Iterator.remove()
 - Entfernt Element, das zuletzt von next zurückgegeben wurde
 - Macht **alle anderen aktiven** Iteratoren **ungültig**

Zusammenfassung



- ▶ `List` unterstützt `ListIterator`
 - ▶ Erweitert `Iterator`
 - ▶ Hinzufügen möglich
 - ▶ Laufen in **beide** Richtungen möglich
 - ▶ Zugriff auf **Index**
 - ▶ Setzen des Inhalts
- ▶ Achtung bei Strukturänderungen

Eigene Iteratoren

► Kochrezept für Container mit Elementtyp E

1. Innere Klasse

```
private class ContainerIterator  
    implements Iterator<E>
```

- Attribut für **aktuelle Position**
- hasNext: Am Ende?
- next(): Wenn hasNext()==**false** → [↗ NoSuchElementException](#)
- **Sonst** nächstes Element liefern **und** weiterspringen

2. Container-Klasse **modifizieren**

- Container **implements** Iterable<E>
- **Instanz** von ContainerIterator in iterator erzeugen

► Fremde Klassen: **Iterierbare** „Wrapper“-Klasse

► **Iteration** ohne Datenstruktur möglich (Range)

Inhalt

Vergleichen mit Comparable und Comparator

- Motivation

- Das Comparable-Interface

- Das Comparator-Interface

- Sortierung in TreeSet und TreeMap


- Zusammenfassung

Inhalt

Vergleichen mit Comparable und Comparator Motivation

Sortieren von Zahlen

- ▶ **Gegeben:** Liste von Zahlen die sortiert werden soll
- ▶ **Zur Erinnerung:** `Collections.sort(List<T>)` sortiert `List<T>`
- ▶ **Beispiel**

```
19  runSortNumbersExample  
20 List<Integer> numbers  
21     = Arrays.asList(5,1,2,6,9,3,8,7,4);  
23 Collections.sort(numbers);  
25 out.println(numbers);
```

 ComparingExamples.java


```
[1, 2, 3, 4, 5, 6, 7, 8, 9]
```

- ▶ **Funktioniert wie erwartet**
 - ▶ `Integer/int` hat natürliche Ordnung
 - ▶ `sort` verwendet diese Ordnung

Sortieren von Items

Item
– name : String
– price : int

- ▶ Wie soll eine Liste von Items **sortiert** werden?
- ▶ Item hat **Name** und **Preis**
- ▶ **Vorschlag**: Sortierung nach **Preis**
- ▶ Versuch

```
37  runSortItemsExample  
38 List<Item> items  
39     = Arrays.asList(salad,choc,milk,toiletpaper);  
41 Collections.sort(items);  
43 out.println(items);
```

 ComparingExamples.java

Sortieren von Items

► Problem

```
Collections.sort(items); // FEHLER
```

„No suitable method found for sort(List<Item>)“

- Java **weiß nicht** wie man Items **sortiert**
- **Genauer:** Java weiß nicht, wie man zwei Items **vergleicht**
- Beispiel

```
var salad = new Item("Salat", 2);  
var choc = new Item("Schokolade", 1);
```

- Gilt „salad < choc“ → salad wird **vor** choc sortiert
- Oder „choc < salad“ → choc wird **vor** salad sortiert
- Oder „choc = salad“ → Reihenfolge **egal**
- Wie bringen wir Java bei Items zu **vergleichen**?
- Antwort: ↗ **Comparable**-Interface

Vergleichen mit Comparable und Comparator

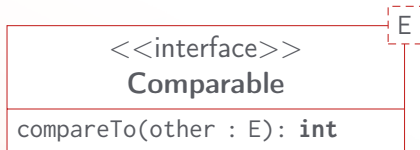
Das Comparable-Interface

- Item vergleichbar machen

- compareTo-Kochrezept

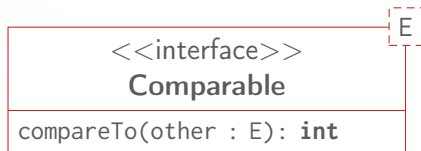
- Comparable in Wrapper-Klassen

Das Comparable-Interface



- ▶ [Comparable](#): engl. „vergleichbar“
- ▶ Definiert „natural ordering“ (natürliche Ordnung)
- ▶ Von Klassen **implementiert**, die **verglichen** werden können
- ▶ **int** compareTo(E other)
 - ▶ **Vergleicht** **this**-Objekt mit other (vgl. equals)
 - ▶ < 0 wenn „**this** $<$ other“
 - ▶ > 0 wenn „**this** $>$ other“
 - ▶ $= 0$ wenn „**this** $=$ other“

Eigenschaften von compareTo



- ▶ Seien
 - ▶ x, y, z vom Typ `E`
 - ▶ `sgn(p)` das **Vorzeichen** von p ($-1, 0, +1$)
- ▶ **Trichotomie** $((x < y) \vee (y < x) \vee (x = y))$
 $\text{sgn}(x.\text{compareTo}(y)) == -\text{sgn}(y.\text{compareTo}(x))$
- ▶ **Transitivität** $(x < y \wedge y < z \implies x < z)$
 $\text{sgn}(x.\text{compareTo}(y)) > 0 \ \&\& \ \text{sgn}(y.\text{compareTo}(z)) > 0 \implies \text{sgn}(x.\text{compareTo}(z)) > 0$
- ▶ **Konsistenz bei Gleichheit**
 $x.\text{compareTo}(y) == 0 \implies x.\text{compareTo}(z) == y.\text{compareTo}(z)$ für alle z
- ▶ **Nicht verlangt, aber stark empfohlen: Konsistenz mit equals**
 $x.\text{compareTo}(y) == 0 \implies x.\text{equals}(y)$

Vergleichen mit Comparable und Comparator

Das Comparable-Interface

- Item vergleichbar machen

- `compareTo`-Kochrezept

- Comparable in Wrapper-Klassen

Item vergleichbar machen

Item
<ul style="list-style-type: none">- name : String- price : int

- ▶ Items über Preis **vergleichbar** machen
- ▶ Item a, b: „a < b“ \Leftrightarrow a.getPrice() < b.getPrice()
- ▶ **Schritt 1:** `Comparable<Item>` implementieren

```
4 public class Item implements Comparable<Item> {
```

Item.java

- ▶ **Schritt 2:** compareTo(Item other) implementieren

```
@Override  
public int compareTo(Item other) {  
    return this.price - other.price;  
}
```

Item vergleichbar machen

```
public int compareTo(Item other) {  
    return this.price - other.price;  
}
```

- ✓ **Korrektheit:** „ $a < b$ “ $\Leftrightarrow a.getPrice() < b.getPrice() \Leftrightarrow a.getPrice() - b.getPrice() < 0$
 $\Leftrightarrow a.compareTo(b) < 0$
- ▶ Entsprechend für = und >
- ✓ **Trichotomie:** ein Item ist **billiger**, **teurer** oder kostet **gleich viel** wie ein anderes Item
- ✓ **Transitivität:** folgt aus Transitivität von < auf **int**
- ✓ **Konsistenz bei Gleichheit:** folgt aus $a.compareTo(b) == 0 \implies$
 $a.getPrice() == b.getPrice() \implies$
 $a.getPrice() - c.getPrice() == b.getPrice() - c.getPrice()$ für alle Item c

Item vergleichbar machen

```
public int compareTo(Item other) {  
    return this.price - other.price;  
}
```

✗ Konsistenz mit equals

```
Item salad = new Item("Salat", 2);  
Item milk = new Item("Milch", 2);
```

- ▶ `salad.compareTo(milk) == 0`
- ▶ `salad.equals(milk) == false`
- ▶ **Problem:** name wird **nicht** berücksichtigt
- ▶ Laut Dokumentation „nur“ **strongly recommended**
- ▶ **Trotzdem:** Natürliche Ordnung sollte **konsistent mit equals** sein

Item vergleichbar machen


```
56 @Override public int compareTo(Item other) {  
57     if (other == null)  
58         throw new IllegalArgumentException("other == null");  
60     int result = this.price - other.price;  
62     if (result == 0)  
63         result = this.name.compareTo(other.name);  
65     return result;  
66 }
```

Item.java

► Vorgehen

1. Prüfung: `other == null`
2. Preis vergleichen: `result = this.price - other.price`
3. Bei gleichem Preis: Namen vergleichen `result = this.name.compareTo(other.name);`

Item vergleichbar machen


```
49  runCompareToItemExample
50 var choc = new Item("Schokolade", 1);
51 var salad = new Item("Salat", 2);
52 var milk = new Item("Milch", 2);
54 out.printf("choc.compareTo(salad): %d\n",
55     choc.compareTo(salad));
57 out.printf("salad.compareTo(milk): %d\n",
58     salad.compareTo(milk));
```

 ComparingExamples.java

```
choc.compareTo(salad): -1
salad.compareTo(milk): 6
```

- ▶ choc < salad da Schokolade **billiger** als Salat
- ▶ salad > milk
 - ▶ Salat und Milch kosten **gleich viel**
 - ▶ Aber: "Salat" kommt **lexikographisch nach** "Milch"

Item vergleichbar machen

```
37  runSortItemsExample  
38 List<Item> items  
39     = Arrays.asList(salad,choc,milk,toiletpaper);  
41 Collections.sort(items);  
43 out.println(items);
```

 ComparingExamples.java

▶ Kein Fehler mehr

```
[Schokolade: 1 EUR, Milch: 2 EUR, Salat: 2 EUR, Toilettenpapier: 3 EUR]
```

▶ Sortierreihenfolge

- ▶ Erst nach Preis
- ▶ Dann nach Namen

Item vergleichbar machen

- ▶ Was ist wenn erst nach Namen und dann nach Preis sortiert werden soll?
- ▶ Reihenfolge umdrehen

```
int result = this.name.compareTo(other.name);  
if (result == 0)  
    result = this.price - other.price;  
return result;
```

- ▶ Ergebnis

```
[Milch: 2 EUR, Salat: 2 EUR, Schokolade: 1 EUR, Toilettenpapier: 3 EUR]
```

- ▶ Für natürliche Ordnung die „intuitive“ Variante wählen
- ▶ Für alternative Ordnung ↗ Comparator verwenden (später)

Inhalt

Vergleichen mit Comparable und Comparator

Das Comparable-Interface

Item vergleichbar machen

compareTo-Kochrezept

Comparable in Wrapper-Klassen

compareTo-Kochrezept

```
public class MyItem implements Comparable<Item>{  
    private T1 a1;  
    private T2 a2;  
    ...  
    private TN aN;  
}
```

- ▶ **Natürliche Ordnung** nach a_1, a_2, \dots, a_N
- ▶ compareTo soll a_1, a_2, \dots, a_N in dieser **Reihenfolge** vergleichen
- ▶ **Resultierende Sortierung**
 - ▶ Erst nach a_1
 - ▶ Dann nach a_2
 - ▶ ...
 - ▶ Dann nach a_N

compareTo-Kochrezept

Allgemeine Implementierung von compareTo

```
@Override public int compareTo(MyItem other){  
    if (other == null)  
        throw new IllegalArgumentException("...");  
    int result = this.a1.compareTo(other.a1);  
    if (result == 0)  
        result = this.a2.compareTo(other.a2);  
    ...  
    if (result == 0)  
        result = this.aN.compareTo(other.aN);  
    return result;  
}
```

compareTo-Kochrezept: Hinweise

```
public class MyItem implements Comparable<Item>{  
    private T1 a1;  
    private T2 a2;  
    ...  
    private TN aN;  
}
```

- ▶ Für alle T_i muss gelten
 - ▶ T_i **primitiv**: Vergleich wie `<` oder **false** < **true**
 - ▶ Oder: T_i **Referenztyp**: T_i muss `Comparable<Ti>` implementieren
 - ▶ Oder: Vergleich von T_i in `compareTo` implementieren (unschön)
- ▶ Achtung bei Referenztypen
 - ▶ a_i eventuell auf **null** prüfen
 - ▶ Behandlung bei $a_i == \text{null}$
 - ▶ Entsprechend Semantik von **null**
 - ▶ Oder: `IllegalStateException`/`IllegalArgumentException`

Vergleichen mit Comparable und Comparator

Das Comparable-Interface

Item vergleichbar machen

compareTo-Kochrezept

Comparable in Wrapper-Klassen

Comparable in Wrapper-Klassen

- ▶ Wrapper-Klassen implementieren `Comparable`

Wrapper-Typen	compareTo
<code>Byte</code> , <code>Short</code> , <code>Character</code> , <code>Integer</code> , <code>Long</code>	<code>this.v - other.v</code>
<code>Float</code> , <code>Double</code>	<code>this.v - other.v</code> <code>NaN == NaN</code> <code>x < NaN</code> <code>-0.0 < 0.0</code>
<code>Boolean</code>	<code>false < true</code>

Hinweis: `-0.0 < 0.0` erhält Konsistenz mit `equals`

- ▶ `String.compareTo` entspricht lexikographischer Ordnung

Vergleichen mit Comparable und Comparator

Das Comparator-Interface

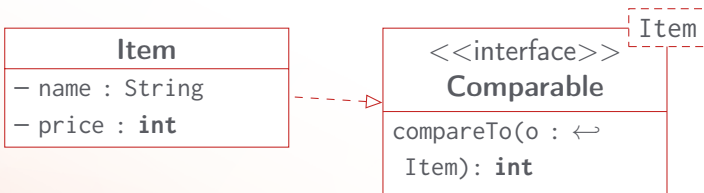
- Comparator-Interface

- Item über Comparator vergleichen

- „Richtige“ alphabetische Sortierung

- Comparator als anonyme Klasse

Alternative Sortierung von Items



- ▶ `Item` hat **natürliche Ordnung**
- ▶ `Item.compareTo` vergleicht **erst** Preis, dann **Name**
- ▶ **Aber:** Was wenn wir **erst nach Namen**, dann **nach Preis** vergleichen/sortieren wollen?
 - ▶ **Unschön:** Option in `Item.compareTo`

```
if (sortByPrice)
    // first: price, second: name
else
    // first: name, second: price
```

- ▶ **Unschön:** **Ableiten** und Überschreiben (nicht der **Sinn** von Vererbung)

Vergleichen mit Comparable und Comparator

Das Comparator-Interface

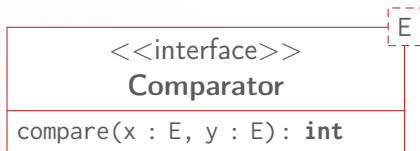
- Comparator-Interface

- Item über Comparator vergleichen

- „Richtige“ alphabetische Sortierung

- Comparator als anonyme Klasse

Comparator-Interface



- ▶ `Comparator`: engl. „Vergleicher“
 - ▶ Klassen mit `Comparator` können zwei Instanzen vom Typ `E` **vergleichen**
 - ▶ **Unterschiedliche Vergleichsalgorithmen** möglich
- ▶ **int** `Comparator.compare(E x, E y)` vs. **int** `Comparable.compareTo(E other)`
 - ▶ Gleiche **Semantik** mit `this == x` und `other == y`
 - ▶ Gleiche **Eigenschaften**: Korrektheit, Trichotomie, Transitivität, Konsistenz bei Gleichheit
 - ▶ **Unterschied**: Konsistenz mit `equals` **optional**

Vergleichen mit Comparable und Comparator

Das Comparator-Interface

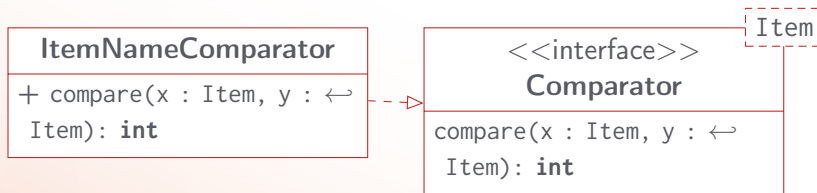
Comparator-Interface

Item über Comparator vergleichen

„Richtige“ alphabetische Sortierung

Comparator als anonyme Klasse

Item über Comparator vergleichen



- ▶ **Neue Klasse** `ItemNameComparator`
 - ▶ Implementiert `Comparator<Item>`
 - ▶ Vergleicht **erst Namen, dann Preis**
- ▶ `ItemNameComparator.compare(Item x, Item y)`
 - ▶ **Struktur** wie bei `Comparable.compareTo(Item other)`
 - ▶ Mit `x == this` und `y == other`


Item über Comparator vergleichen

ItemNameComparator.compare

```
10 @Override
11 public int compare(Item x, Item y) {
13     if (x == null || y == null)
14         throw new IllegalArgumentException("null");
16     int result = x.getName().compareTo(y.getName());
18     if (result == 0)
19         result = x.getPrice() - y.getPrice();
21     return result;
22 }
```

ItemNameComparator.java

Item über Comparator vergleichen

```
64  runComparatorItemExample
65 var choc = new Item("Schokolade", 1);
66 var salad = new Item("Salat", 2);
67 var cheapSalad = new Item("Salat", 1);
69 var comp = new ItemNameComparator();
71 out.printf("comp.compare(choc, salad): %d%n",
72     comp.compare(choc, salad));
74 out.printf("comp.compare(salad, cheapSalad): %d%n",
75     comp.compare(salad, cheapSalad));
```

 ComparingExamples.java

```
comp.compare(choc, salad): 2
comp.compare(salad, cheapSalad): 1
```

Item über Comparator vergleichen

```
comp.compare(choc, salad): 2  
comp.compare(salad, cheapSalad): 1
```


- ▶ salad < choc da "Schokolade" **alphabetisch** nach Salat kommt
- ▶ salad < cheapSalad
 - ▶ Name ist **gleich**
 - ▶ cheapSalad ist **billiger** als salad

Item über Comparator sortieren

► [List](#)<E>.sort(Comparator<E> comparator)

► Ermöglicht **sortieren** über comparator

► Beispiel

```
85  runSortItemsComparatorExample  
86 List<Item> items  
87   = Arrays.asList(salad, choc, milk, cheapSalad);  
89 items.sort(new ItemNameComparator());  
91 out.println(items);
```

 ComparingExamples.java

[Schokolade: 1 EUR, Salat: 1 EUR, Salat: 2 EUR, Milch: 2 EUR]

Vergleichen mit Comparable und Comparator

Das Comparator-Interface

Comparator-Interface


Item über Comparator vergleichen

„Richtige“ alphabetische Sortierung

Comparator als anonyme Klasse

Collator

- ▶ Hinweis: `String.compareTo` berücksichtigt keine sprach-/landspezifisches Sortierreihenfolgen
- ▶ Besser: Alphabetischer Vergleich über `Collator`
 - ▶ **implements** `Comparator<String>`
 - ▶ Berücksichtigt **locales** (länderspezifische Besonderheiten)
 - ▶ Instanz von `Collator` über `Collator.getInstance()`

```
97  runCollatorCompareExample  
98 Collator c = Collator.getInstance();  
99 out.println(c.compare("Salat", "Schokolade")); // -1
```

 ComparingExamples.java

ItemNameAlphabeticComparator

```
7 public class ItemNameAlphabeticComparator
8     implements Comparator<Item> {
10     @Override public int compare(Item x, Item y) {
11         if (x == null || y == null)
12             throw new IllegalArgumentException("...");
14         Collator c = Collator.getInstance();
16         return c.compare(x.getName(), y.getName());
17     }
18 }
```


ItemNameAlphabeticComparator.java

Sortieren über `ItemNameAlphabeticComparator`

- ▶ `ItemNameAlphabeticComparator`
 - ▶ Sortiert **nur** nach **Namen** (ignoriert Preis)
 - ▶ **Nicht konsistent** mit `equals`

```
c.compare(salad, cheapSalad) == 0
```

- ▶ Sortieren über `ItemNameAlphabeticComparator`

```
112  runSortItemsComparatorExample2  
113 items.sort(new ItemNameAlphabeticComparator());
```

 ComparingExamples.java

```
[Milch: 2 EUR, Salat: 2 EUR, Salat: 1 EUR, Schokolade: 1 EUR]
```

- ▶ Hinweis: salad kommt **vor** cheapSalad
 - ▶ `ItemNameAlphabeticComparator` ignoriert **Preis**
 - ▶ Sortierte Reihenfolge bei `compare(x,y) == 0` **nicht bestimmt**

Vergleichen mit Comparable und Comparator

Das Comparator-Interface

Comparator-Interface


Item über Comparator vergleichen

„Richtige“ alphabetische Sortierung

Comparator als anonyme Klasse

Comparator also anonyme Klasse

- ▶ Unschön
 - ▶ ItemNameAlphabeticComparator **eigene Klasse**
 - ▶ ...in **eigener Datei**
- ▶ **Alternative:** Anonyme Klasse

```
128  runSortItemsComparatorExample3  
129 items.sort(new Comparator<Item>(){  
130     public int compare(Item x, Item y) {  
131         if (x == null || y == null)  
132             throw new IllegalArgumentException("...");  
134         Collator c = Collator.getInstance();  
136         return c.compare(x.getName(), y.getName());  
137     }  
138 });
```

 ComparingExamples.java

- ▶ Oder: geschachtelte/lokale Klasse

Inhalt

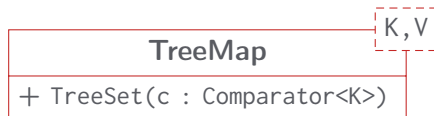
Vergleichen mit Comparable und Comparator

Sortierung in TreeSet und TreeMap

Sortierung in TreeSet

Sortierung in TreeMap

Sortierung in TreeSet und TreeMap



- ▶ [TreeSet](#) und [TreeMap](#) erhalten **Sortierung** der Einträge
 - ▶ [TreeSet](#) extends Set — nach **Elementen**
 - ▶ [TreeMap](#) extends Map — nach **Schlüssel**
- ▶ Zwei Möglichkeiten für **Sortierung**
 - ▶ Natürliche Ordnung — [Comparable](#) bei Elementen/Schlüssel
 - ▶ Comparator — [Comparator](#) über Konstruktor
- ▶ [Iteratoren](#) in sortierter Reihenfolge
- ▶ Logarithmische Laufzeit bei Einfügen, Löschen, Suchen

Inhalt

Vergleichen mit Comparable und Comparator


Sortierung in TreeSet und TreeMap

Sortierung in TreeSet

Sortierung in TreeMap

Sortierung in TreeSet

- ▶ Zur Erinnerung: Item implementiert `Comparable`
 - ▶ Erst nach Preis
 - ▶ Dann nach Name
- ▶ Beispiel

```
150  runTreeSetComparableExample  
151 TreeSet<Item> items = new TreeSet<Item>();  
152 items.add(salad);  
153 items.add(choc);  
154 items.add(milk);  
155 items.add(cheapSalad);
```


 ComparingExamples.java

```
[Salat: 1 EUR, Schokolade: 1 EUR, Milch: 2 EUR, Salat: 2 EUR]
```

- ▶ Reihenfolge entspricht natürlicher Ordnung

Sortierung in TreeSet

- ▶ Zur Erinnerung: ItemNameComparator
 - ▶ Erst nach Name (lexikographisch)
 - ▶ Dann nach Preis
- ▶ Beispiel

```
166  runTreeSetComparatorExample
167 TreeSet<Item> items = new TreeSet<Item>(
168     new ItemNameComparator());
169 items.add(salad);
170 items.add(choc);
171 items.add(milk);
172 items.add(cheapSalad);
```

 ComparingExamples.java

[Milch: 2 EUR, Salat: 1 EUR, Salat: 2 EUR, Schokolade: 1 EUR]


- ▶ Reihenfolge entspricht ItemNameComparator

Sortierung in TreeSet

- ▶ Zur Erinnerung: `ItemNameAlphabeticComparator`
 - ▶ Sortiert nach **Name** (mit [↗ Collator](#))
 - ▶ Ignoriert Preis
 - ▶ **Nicht konsistent** mit `equals`

```
c.compare(salad, cheapSalad) == 0  
salad.equals(cheapSalad) == false
```

▶ Beispiel

```
183  runTreeSetInconsistentComparatorExample  
184 TreeSet<Item> items = new TreeSet<Item>(  
185     new ItemNameAlphabeticComparator());  
186 items.add(salad);  
187 items.add(choc);  
188 items.add(milk);  
189 items.add(cheapSalad);
```

 ComparingExamples.java

```
[Milch: 2 EUR, Salat: 2 EUR, Schokolade: 1 EUR]
```

Sortierung in TreeSet

[Milch: 2 EUR, Salat: 2 EUR, Schokolade: 1 EUR]

- ▶ Wo ist cheapSalad?
 - ▶ Bei `items.add(cheapSalad)` ist salad bereits in items
 - ▶ `c.compare(salad, cheapSalad) == 0` → [TreeSet](#) „denkt“ es handelt sich um Duplikat
- ▶ [TreeSet](#) (und [TreeMap](#)) verlangen Konsistenz mit `equals`!

Inhalt

Vergleichen mit Comparable und Comparator


Sortierung in TreeSet und TreeMap

Sortierung in TreeSet

Sortierung in TreeMap

Sortierung in TreeMap

- ▶ `TreeMap<K,V>`: Sortierung über Schlüssel
 - ▶ Natürliche Ordnung — K implementiert `Comparable<K>`
 - ▶ Oder: `Comparator<K>`
- ▶ Beispiel

```
200  runTreeMapComparableExample
201 var stock = new TreeMap<Item,Integer>();
202 stock.put(salad, 20);
203 stock.put(choc, 20);
204 stock.put(milk, 10);
205 stock.put(cheapSalad, 25);
```

 ComparingExamples.java

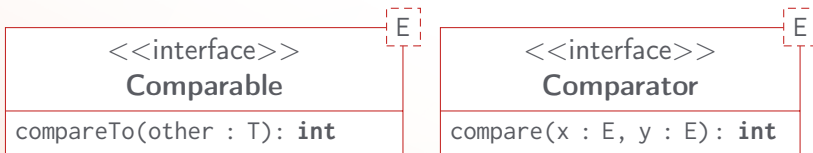
```
{Salat: 1 EUR=25, Schokolade: 1 EUR=20,
  Milch: 2 EUR=10, Salat: 2 EUR=20}
```

- ▶ Schlüssel sind über natürliche Ordnung sortiert
- ▶ `TreeMap<K,V>(Comparator<K>)` für Sortierung über `Comparator`

Inhalt

Vergleichen mit Comparable und Comparator Zusammenfassung

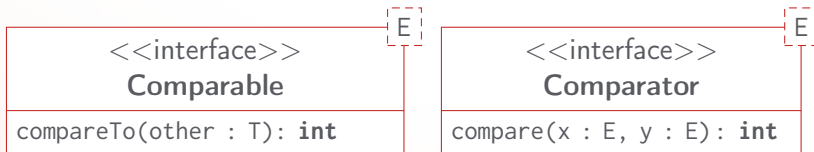
Zusammenfassung



	Comparable	Comparator
Ordnung	natürlich	Anwendungskontext
Implementierung	in Klasse	außerhalb Klasse
equals-Konsistenz	verpflichtend	optional

- **Eigenschaften:** Korrektheit, Trichotomie, Transitivität, Konsistenz bei `x.compareTo(y) == 0`

Zusammenfassung




- ▶ **Vergleich** von Elementen „x < y“

```
x.compareTo(y)
```



- ▶ **Sortieren** über natürliche Ordnung

```
Collections.sort(List<E> l)
```

- ▶ **Sortieren** über  **Comparator**

```
List<E>.sort(Comparator<E> c)
```

- ▶ **Sortierte Datenstrukturen**

- ▶  **TreeSet** — sortierte Einträge
- ▶  **TreeMap** — sortierte Schlüssel
- ▶ **Konsistenz** mit `equals` verpflichtend!