

Programmieren I und II

Unit 3

Selbstdefinierbare Datentypen, Arrays und Collections



Prof. Dr. rer. nat.
Nane Kratzke

*Praktische Informatik und
betriebliche Informationssysteme*

- Raum: 17-0.10
- Tel.: 0451 300 5549
- Email: kratzke@fh-luebeck.de



@NaneKratzke

Updates der Handouts auch über Twitter #prog_inf
und #prog_itd

Units



**FACH
HOCHSCHULE
LÜBECK**
University of Applied Sciences

1. Semester

| | | | |
|--|--|---|---|
| Unit 1 Einleitung und Grundbegriffe | Unit 2 Grundlagen imperativer Programmierung | Unit 3 Selbstdefinierbare Datentypen und Collections | Unit 4 Einfache I/O Programmierung |
| Unit 5 Rekursive Programmierung, rekursive Datenstrukturen, Lambdas | Unit 6 Objektorientierte Programmierung und UML | Unit 7 Konzepte objektorientierter Programmiersprachen, Klassen vs. Objekte, Pakete und Exceptions | Unit 8 Testen (objektorientierter) Programme |

2. Semester

| | | | |
|---------------------------------|--|--------------------------------------|---------------------------------------|
| Unit 9 Generische Datentypen | Unit 10 Objektorientierter Entwurf und objektorientierte Designprinzipien | Unit 11 Graphical User Interfaces | Unit 12 Multithread Programmierung |
|---------------------------------|--|--------------------------------------|---------------------------------------|

Prof. Dr. rer. nat. Nane Kratzke
Praktische Informatik und betriebliche Informationssysteme

3

Abgedeckte Ziele dieser UNIT



**FACH
HOCHSCHULE
LÜBECK**
University of Applied Sciences

Kennen existierender Programmierparadigmen und Laufzeitmodelle

Sicheres Anwenden grundlegender programmiersprachlicher Konzepte (Datentypen, Variable, Operatoren, Ausdrücke, Kontrollstrukturen)

Fähigkeit zur problemorientierten Definition und Nutzung von Routinen und Referenztypen (insbesondere Liste, Stack, Mapping)

Verstehen des Unterschieds zwischen Werte- und Referenzsemantik

Kennen und Anwenden des Prinzips der rekursiven Programmierung und rekursiver Datenstrukturen

Kennen des Algorithmusbegriffs, Implementieren einfacher Algorithmen

Kennen objektorientierter Konzepte Datenkapselung, Polymorphie und Vererbung

Sicheres Anwenden programmiersprachlicher Konzepte der Objektorientierung (Klassen und Objekte, Schnittstellen und Generics, Streams, GUI und MVC)

Kennen von UML Klassendiagrammen, sicheres Übersetzen von UML Klassendiagrammen in Java (und von Java in UML)

Kennen der Grenzen des Testens von Software und erste Erfahrungen im Testen (objektorientierter) Software

Sammeln erster Erfahrungen in der Anwendung objektorientierter Entwurfsprinzipien

Sammeln von Erfahrungen mit weiteren Programmiermodellen und -paradigmen, insbesondere Multithread Programmierung sowie funktionale Programmierung

Am Beispiel der Sprache JAVA

Prof. Dr. rer. nat. Nane Kratzke
Praktische Informatik und betriebliche Informationssysteme

4

Themen dieser Unit



Referenzdatentypen

- Felder (Arrays)
- Klassen

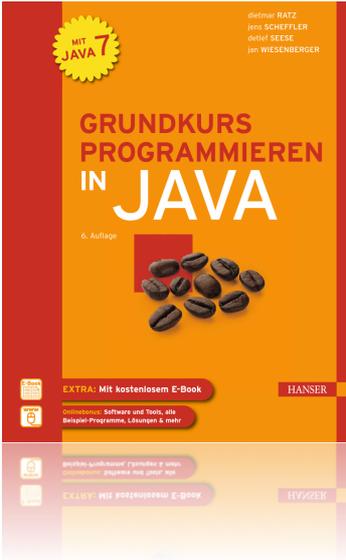
Collections

- Listen
- Stack
- Mappings

Prof. Dr. rer. nat. Nane Kratzke
Praktische Informatik und betriebliche Informationssysteme

5

Zum Nachlesen ...



Kapitel 5
Referenzdatentypen
Abschnitt 5.1 Felder (Arrays)
Abschnitt 5.2 Klassen

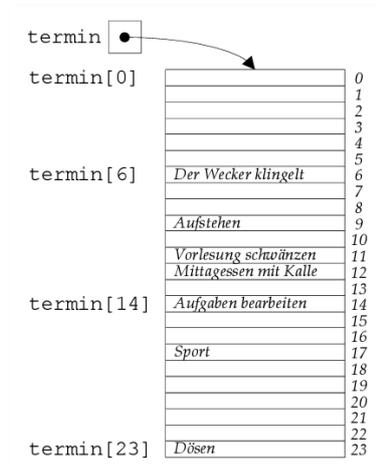
Kapitel 8
Der grundlegende Umgang mit Klassen
Abschnitt 8.3 Statische Komponenten
Abschnitt 8.4 Instantiierung

Prof. Dr. rer. nat. Nane Kratzke
Praktische Informatik und betriebliche Informationssysteme

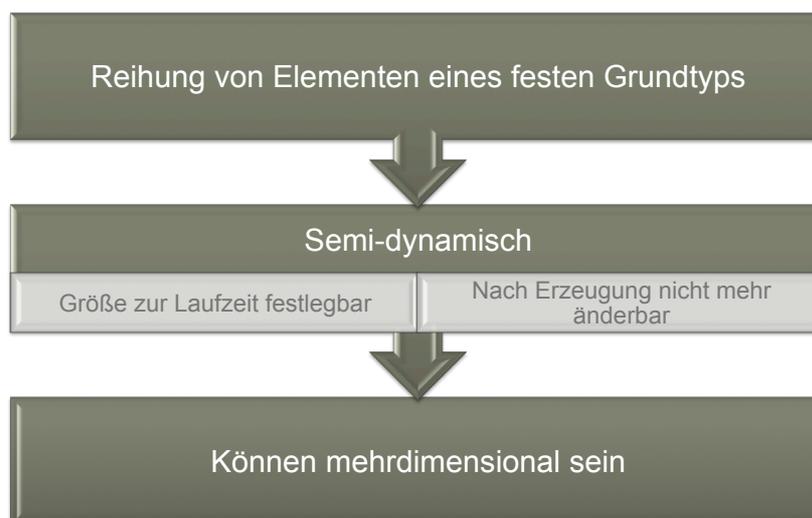
6

Felder (Arrays)

- Sie haben bislang nur primitive Datentypen kennengelernt. Korrespondierende Variablen können pro Variable genau einen Wert speichern.
- Felder (Arrays)** gestatten es, **mehrere Variablen** über einen gemeinsamen **Namen** anzusprechen
- und lediglich durch einen Index zu unterscheiden.
- Der **Index** innerhalb eines Feldes (Arrays) ergibt sich dabei aus der Position innerhalb des Feldes (Arrays), **von null aufwärts gezählt**.



Arrays



Deklaration und Erzeugung von Arrays



Variante 1: Deklarieren und Erzeugen eines Arrays in zwei Schritten:

```
Typ[] var;           // Deklaration eines Arrays  
var = new Typ[n];   // Erzeugung eines Array mit n Elementen
```

Variante 2: Deklarieren und Erzeugen eines Arrays in einem Schritt.
Die Zuweisung muss dabei unmittelbar bei der Deklaration erfolgen.

```
Typ[] var = { new Typ(), ..., new Typ() };
```

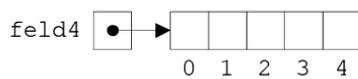
Beide Varianten erzeugen ein Reihung von n Elementen des Typs **Typ**.
Nach der Erzeugung kann n nicht mehr verändert werden, d.h. die Größe des Arrays ist nach Erzeugung unveränderlich.

Deklaration und Erzeugung von Arrays Beispiele



Festlegen des Typs:

```
int[]   feld4;
```

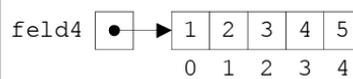


```
feld4 = new int[5]
```

Anlegen eines leeren Arrays

Anlegen der Größe:

```
feld4 = new int[5];
```



```
feld4[0]=1;  
feld4[1]=2;  
feld4[2]=3;  
feld4[3]=4;  
feld4[4]=5;
```

Befüllen eines Arrays mit Werten

Deklaration und Erzeugung von Arrays

Beispiele

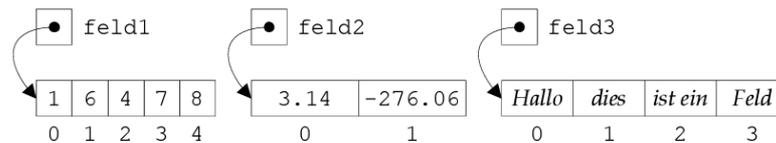


Festlegen des Typs:

```
int[]    feld1;  
double[] feld2;  
String[] feld3;
```

Anlegen der Größe:

```
feld1 = new int[5];  
feld2 = new double[2];  
feld3 = new String[4];
```



Felddeklaration, Anlegen der Größe und initiale Befüllung kann auch mittels **Feldinitialisierer (array initializer)** in einem Schritt erfolgen.

```
int[]    feld1={1 , 6 , 4 , 7 , 8};  
double[] feld2={3.14 , -276.06};  
String[] feld3={"Hallo" , "dies" , "ist ein" , "Feld"};
```

Prof. Dr. rer. nat. Nane Kratzke
Praktische Informatik und betriebliche Informationssysteme

11

Zugriff auf Array-Elemente



Schreibender Zugriff:

```
int[] prim = new int[5];    // Array mit 5 int Elementen  
  
prim[0] = 2;                // Setzen von Array-Elementen  
prim[1] = 3;  
prim[2] = 5;  
prim[3] = 7;  
prim[4] = 11;
```

prim:

| | | | | |
|---|---|---|---|----|
| 2 | 3 | 5 | 7 | 11 |
|---|---|---|---|----|

Typischer lesender Zugriff über einen Laufindex:

```
int length = prim.length    // Länge des Arrays  
for (int i = 0; i < length; i++) {  
    System.out.println(prim[i]);    // Zugriff auf Elem.  
}
```

Prof. Dr. rer. nat. Nane Kratzke
Praktische Informatik und betriebliche Informationssysteme

12

Referenzen

Arrays werden anders gespeichert, als die Ihnen bislang bekannten primitiven Datentypen. Eine Arrayvariable beinhaltet einen Verweis auf die Inhalte des Arrays, nicht die Inhalte selber! Der Unterschied fällt vor allem beim zuweisen von Werten auf.

```
x1 32    x2 a    x3 3.14  
int x1 = 32; char x2 = 'a'; double x3 = 3.14; (1)
```

```
y1 32    y2 a    y3 3.14  
int y1 = x1; char y2 = x2; double y3 = x3; (2)
```

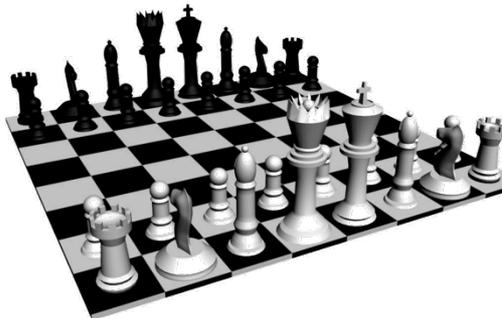
*Bei primitiven
Datentypen werden
die Werte kopiert.*

```
feld [●] → 1 6 4 7 8 (1)  
int[] kop = feld; kop [●] (2)
```

*Bei Arrays die
Referenz auf ein
Array (nicht die
Inhalte/Werte).*

Mehrdimensionale Arrays (I)

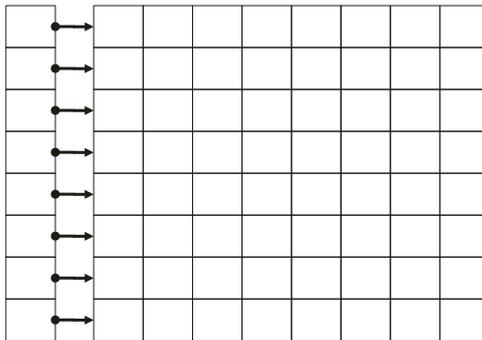
Beispiel für ein einfaches mehrdimensionales Array.



```
Schachfigur[][] schachbrett = new Schachfigur[8][8];
```

Mehrdimensionale Arrays (II)

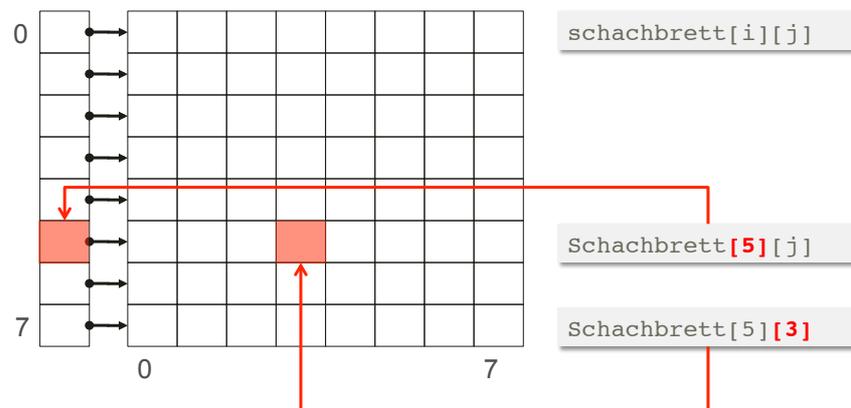
Mehrdimensionale Arrays werden als Arrays von Arrays angelegt.



```
Schachfigur[][] schachbrett = new Schachfigur[8][8];
```

Mehrdimensionale Arrays (III)

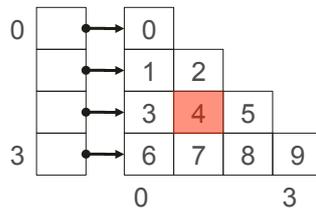
Mehrdimensionale Arrays werden als Arrays von Arrays angelegt.



Mehrdimensionale Arrays (IV)

Es ist auch möglich nicht rechteckige Arrays anzulegen.

```
int[][] a = { { 0 },
              { 1, 2 },
              { 3, 4, 5 },
              { 6, 7, 8, 9 } };
```

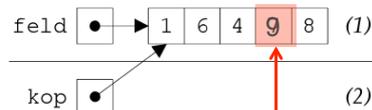


Welchen Wert hat
dieser Ausdruck?

`a[2][1] = ???`

Referenzen

Was passiert, wenn Sie eine Referenzkopie verändern?



```
int[] kop = feld;
```

```
kop[3] = 9;
```

Sie ändern sowohl die „Kopie“ als auch das „Original“ !!!

(Eigentlich gibt es keine Kopie und Original nur zwei Referenzen auf denselben Hauptspeicherbereich)

Referenztypen

FACH HOCHSCHULE LÜBECK
University of Applied Sciences

| Referenztypen | Variablen von Referenztypen | Speicher- management |
|--|---|--|
| <ul style="list-style-type: none">• Alle selbst definierten Datentypen• oder Arrays• Werden mittels new Operator erzeugt | <ul style="list-style-type: none">• enthalten Referenz auf erzeugte Objekte• nicht die Inhalte der Objekte | <ul style="list-style-type: none">• nicht referenzierte Objekte• werden durch einen Garbage Collector automatisch freigegeben  |

Prof. Dr. rer. nat. Nane Kratzke
Praktische Informatik und betriebliche Informationssysteme

19

Klassen

FACH HOCHSCHULE LÜBECK
University of Applied Sciences

- Unter einer Klasse versteht man einen selbstdefinierten Datentyp, der üblicherweise mehrere Komponenten umfasst, die mittels primitiver Datentypen ausgedrückt werden können.
- Eine Klasse kann aber auch prinzipiell Komponenten beinhalten, die wiederum Klassen sind.
- Am einfachsten macht man sich eine Klasse am Beispiel einer Adresse deutlich.

Prof. Dr. rer. nat. Nane Kratzke
Praktische Informatik und betriebliche Informationssysteme

20

Definition eigener Datentypen

Eine Adresse ist sicher ein sinnvoller Datentyp für eine Vielzahl von Anwendungen, existiert jedoch nicht in JAVA.

Eine Adresse kann jedoch aus mehreren primitiven Komponenten zusammengesetzt werden.

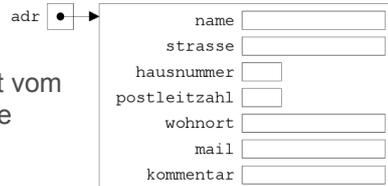
| Adresse | |
|---------------|--------|
| name: | String |
| strasse: | String |
| hausnummer: | int |
| postleitzahl: | int |
| wohnort: | String |
| mail: | String |
| kommentar: | String |

```
class Adresse {  
    public String name;  
    public String strasse;  
    public int hausnummer;  
    public int postleitzahl;  
    public String wohnort;  
    public String mail;  
    public String kommentar;  
}
```

Erzeugen von Objekten eigener Datentypen

Ein Objekt eines Referenzdatentyps wird ähnlich erzeugt wie ein Array. Mit Hilfe des new Operators.

```
Adresse adr = new Adresse();
```



Diese Zeile erzeugt ein neues Objekt vom Datentyp Adresse und speichert die Referenz auf dieses Objekt in der Variablen adr.

Auf die einzelnen Komponenten eines Datentyps kann dann auf folgende Art zugegriffen werden.

```
adr.strasse = "Mönkhofer Weg";
```

Schreibender Zugriff

```
System.out.println(adr.strasse);
```

Lesender Zugriff

adr.strasse ist zu lesen wie: Greife auf die Komponente **strasse** des Objekts zu, dessen Referenz in **adr** gespeichert ist.

Konstruktor

Um ein Adressobjekt anzulegen, kann man also wie folgt vorgehen.

```
Adresse adr = new Adresse();  
adr.name = "Max Mustermann";  
adr.strasse = "Mönkhofer Weg";  
adr.hausnummer = 239;  
  
...
```

Um sich diese Einzelinitialisierungen der Komponenten zu ersparen, wird üblicherweise ein Konstruktor definiert, der im Rahmen des Anlegens eines Objekts wie folgt aufgerufen werden kann.

```
Adresse adr = new Adresse(  
    "Max Mustermann",  
    "Mönkhofer Weg",  
    239, ...);
```

Konstruktor (II)

Ein Konstruktor belegt dabei die Komponenten eines Datentyps mit Werten. Ein Konstruktor ist eigentlich nichts weiter als eine spezielle Methode die im Rahmen der Initialisierung eines Objekts aufgerufen wird.

```
class Adresse {  
    public String name;  
    public String strasse;  
    public int hausnummer;  
  
    ...  
    // Konstruktor  
    public Adresse(String n, String s, int h) {  
        this.name = n;  
        this.strasse = s;  
        this.hausnummer = h;  
        ...  
    }  
}
```


toString() (III)

Der Aufruf:

```
Adresse adr = new Adresse("Max Mustermann", "Mönkhofer  
Weg", 239, 23562, "Lübeck");  
  
System.out.println(adr);
```

Erzeugt dann nicht, Adresse@33f42b49
sondern, die für den Leser gebräuchlichere Form:

```
Max Mustermann  
Mönkhofer Weg 239  
23562 Lübeck
```

Merke: Die `toString` Methode definiert eine textuelle Repräsentation, der Wertebelegung eines Referenztyps. Sie wird immer aufgerufen, wenn ein Objekt als Zeichenkette dargestellt werden soll.

Semantik bei Referenztypen

Zuweisung

- Zuweisung kopiert lediglich die Referenz nicht das Objekt
- Nach einer Zuweisung von *a* (Referenz auf ein Objekt *o*) an *b* zeigen also *a* und *b* auf das Objekt *o*.
- Soll tatsächlich kopiert werden, muss dies mit der **clone** Methode erfolgen.

Gleichheit

- Es wird getestet ob die Referenzen gleich sind,
- nicht ob die Inhalte gleich sind.
- Sollen nur die Inhalte verglichen werden, muss dies mit der **equals** Methode erfolgen.

equals() (I)

Bei Referenztypen ist eine weitere Besonderheit zu beachten. Die Definition der Gleichheit. Werden zwei Referenzen miteinander verglichen, so prüft JAVA ob die Referenzen auf dieselbe Speicherstelle zeigen, nicht ob die Objekte dieselben Werte haben.

So ergibt der folgende Code die Ausgabe `true` (`adr1` ist gleich `adr2`)

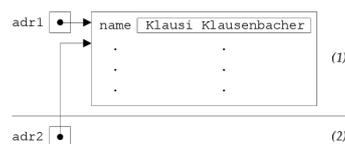
```
Adresse adr1 = new Adresse("Max Mustermann", "Mönkhofer Weg", 239, 23562, "Lübeck");  
  
Adresse adr2 = adr1;  
System.out.println(adr1 == adr2);
```

Dieser Code jedoch die Ausgabe `false` (`adr1` ungleich `adr2`)

```
Adresse adr1 = new Adresse("Max Mustermann", "Mönkhofer Weg", 239, 23562, "Lübeck");  
  
Adresse adr2 = new Adresse("Max Mustermann", "Mönkhofer Weg", 239, 23562, "Lübeck");  
  
System.out.println(adr1 == adr2);
```

equals() (II)

Im ersten Fall wird einfach nur die Referenz kopiert.



Im zweiten Fall wird ein neues Objekt `adr2` angelegt, das (zufällig) dieselben Inhalte wie das Objekt `adr1` hat. Die Inhalte stehen jedoch an unterschiedlichen Stellen im Hauptspeicher.



Der Gleichheitsoperator `==` ist jedoch auf Referenzen (Identität) und nicht auf Inhalten definiert (Wertgleichheit). JAVA selber kann nicht auf Wertgleichheit vergleichen. Jedoch kann man dies für eigene Datentypen mittels einer `equals` Methode selber implementieren.

equals() (III)

Hierzu muss im selber definierten Datentyp eine `equals` Methode eingebaut werden, die Komponenten (primitive Datentypen) paarweise miteinander auf Gleichheit vergleicht.

```
class Adresse {
    public String name;
    public String strasse;
    public int hausnummer
    ...

    public boolean equals(Adresse adr) {
        return this.name == adr.name &&
            this.strasse == adr.strasse &&
            this.hausnummer == adr.hausnummer;
    }
}
```

Der folgende Code erzeugt dann als Ausgabe `true`, d.h. `adr1` und `adr2` sind wertgleich jedoch nicht referenzgleich.

```
Adresse adr1 = new Adresse("Max Mustermann", "Mönkhofer Weg", 239, 23562,
"Lübeck");

Adresse adr2 = new Adresse("Max Mustermann", "Mönkhofer Weg", 239, 23562,
"Lübeck");

System.out.println(adr1.equals(adr2)); // Keine Anw. des == Operators
```

clone() (I)

Wie sie gesehen haben, wird mit dem Zuweisungsoperator `=` bei Referenztypen nur die Referenz, aber nicht die Inhalte dupliziert. Bei primitiven Datentypen werden hingegen tatsächlich die Inhalte dupliziert. Daher ist das Verhalten von Referenztypen und primitiven Datentypen bspw. bei Methodenaufrufen ein anderes (Stichwort: Call by Reference Verhalten, vgl. Unit 2).

Für unser Adressdatentyp sähe eine `clone` Methode beispielsweise wie folgt aus:

```
class Adresse {
    public String name;
    public String strasse;
    public int hausnummer;

    ...

    public Adresse clone() {
        return new Adresse(name, strasse, hausnummer);
    }
}
```

clone() (II)

Und könnte wie folgt aufgerufen werden:

```
Adresse adr1 = new Adresse("Max Mustermann", "Mönkhofer  
Weg", 239, 23562, "Lübeck");  
  
Adresse adr2 = adr1.clone();
```



Klassenvariablen

- Sie lernen nun die Bedeutung des Schlüsselworts **static** kennen.
- Sie haben bislang gelernt, dass Datenfelder (Variablen) eines Referenztyps ohne das Schlüsselwort **static** deklariert wurden.
- Derartige Datenfelder gehören immer zu genau einem Objekt welches mit dem **new** Operator erzeugt wird.

```
public class Adresse {  
    public String vorname;  
    public String nachname;  
    ...  
}
```

```
Adresse adr1 = new  
Adresse("Max",  
"Mustermann", ...);  
  
Adresse adr2 = new  
Adresse("Maren",  
"Musterfrau", ...);
```

Klassenvariablen und -methoden

- Datenfelder können sich aber auch auf alle Objekte einer Klasse, also die Klasse selber beziehen.
- Diese Datenfelder gelten dann für alle Objekte einer Klasse und werden bei der Datenfeld Deklaration durch das Schlüsselwort **static** gekennzeichnet.
- Gleiches gilt für Methoden.

```
public class Adresse {  
    public static String vorname;  
    public String nachname;  
    ...  
}
```

```
Adresse adr1 = new Adresse("Max",  
"Mustermann", ...);  
  
Adresse adr2 = new  
Adresse("Maren",  
"Musterfrau", ...);  
  
System.out.println(adr1.vorname);
```

Maren (obwohl doch adr1 als Max Mustermann instantiiert wurde).

Merke: Änderungen an als **static** deklarierten Datenfeldern eines Referenztyps wirken sich auf **ALLE** Objekte dieses Referenztyps aus.

Beispiel: Personen zählen

- Sie sollen nun einen Referenztyp Person entwickeln, der den Vor- und Nachnamen einer Person speichern und ausgeben kann.
- Zusätzlich soll im Referenztyp mitgezählt werden, die wievielte von wie vielen insgesamt angelegten Personen diese Person ist. Auch diese Information soll in folgender Form ausgegeben werden.



Max Mustermann (3/1089)

Personen zählen - Lösung

```
class Person {
    public String vorname;
    public String nachname;
    public static int total;           // Zählt alle angelegten Personen Objekte
    public int meine_nr;

    public Person(String vn, String nn) {
        this.vorname = vn;
        this.nachname = nn;
        this.meine_nr = ++Person.total; // auch möglich this.total
    }

    public String toString() {
        return this.vorname + " " + this.nachname +
            " (" + this.meine_nr + "/" + Person.total + ")"; // auch möglich this.total
    }
}

public class Beispiel {
    public static void main(String[] args) {
        Person p1 = new Person("Max", "Mustermann");
        Person p2 = new Person("Maren", "Musterfrau");
        System.out.println(p1);
        System.out.println(p2);
    }
}
```

Miniübung:



Gegeben seien zwei Felder a und b vom Typ `int []`.

- Warum kann man die beiden Felder a und b nicht mittels `a == b` vergleichen?
- Wie könnte ein Programmstück aussehen, das beide Felder miteinander vergleicht? Dabei seien zwei Felder genau dann gleich, wenn sie die gleiche Länge haben und alle ihre Komponenten paarweise übereinstimmen.

Miniübung:



Welche der folgenden Ausdrücke wird zu true oder false ausgewertet?

```
Adresse adr1 = new Adresse("Mustermann", "23562 HL");
Adresse adr2 = adr1;
Adresse adr3 = adr1.clone();
Adresse adr4 = new Adresse("Mustermann", "23562 Lübeck");
Adresse adr5 = new Adresse("Mustermann", "23562 HL");
```

| | | | |
|-------------------|-------|-------------------|-------|
| adr1 == adr3 | false | adr3 == adr1 | false |
| adr2 == adr1 | true | adr1.equals(adr2) | true |
| adr3.equals(adr1) | true | adr1.equals(adr3) | true |
| adr1 == adr4 | false | adr4 == adr1 | false |
| adr4.equals(adr1) | false | adr1.equals(adr4) | false |
| adr5.equals(adr1) | true | adr1.equals(adr5) | true |
| adr3.equals(adr3) | true | adr3 == adr3 | true |

Miniübung:



Der Adressdatentyp hatte einen Email und Kommentarbestandteil. Diese Anteile sollen nur ausgegeben werden, wenn Sie auch definiert sind, d.h. ungleich der leeren Zeichenkette sind. Wie müssen Sie ihre toString Methode schreiben?

```
class Adresse {
    public String name;
    public String strasse;
    public int hausnummer;
    public int postleitzahl;
    public String wohnort;
    public String mail;
    public String kommentar;
}
```

Zusammenfassung

A+

FACH
HOCHSCHULE
LÜBECK
University of Applied Sciences

- **Arrays**
 - Semistatische Datenstruktur
 - Deklaration und Initialisieren von Arrays
 - Eindimensionale Arrays
 - Zwei- und mehrdimensionale Arrays
 - Verhalten wie ein Referenzdatentyp
- **Klassen**
 - Referenzdatentypen
 - Erzeugen von Objekten mit dem `new` Operator
 - Unterschiede zu primitiven Datentypen
 - Wertgleichheit und Referenzgleichheit
 - Clonen von Objekten
 - Statische und nicht statische Datenfelder



Prof. Dr. rer. nat. Nane Kratzke
Praktische Informatik und betriebliche Informationssysteme

41

Themen dieser Unit

FACH
HOCHSCHULE
LÜBECK
University of Applied Sciences

Referenzdatentypen

- Felder (Arrays)
- Klassen

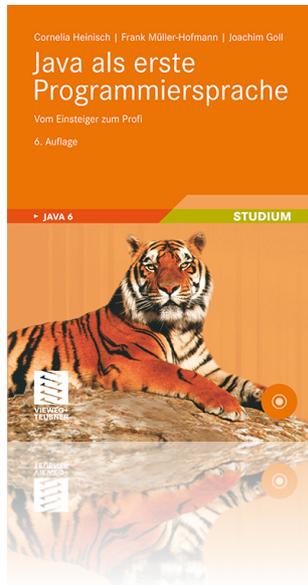
Collections

- Listen
- Stack
- Mappings

Prof. Dr. rer. nat. Nane Kratzke
Praktische Informatik und betriebliche Informationssysteme

42

Zum Nachlesen ...



Kapitel 18 Collections

Abschnitt 18.3 Listen

List

Abschnitt 18.3.3 Stapel

Stack

Abschnitt 18.6 Verzeichnisse

Map

Collections

- Im Rahmen der Programmierung benötigt man immer wieder Datenstrukturen, die sich stark ähneln.
- Z.B. muss man häufig eine Menge von Objekten in einer Liste speichern (z.B. alle Adressen von Studierenden), HTML Seiten werden in Web-Browsern bspw. häufig als Bäume verwaltet.
- Die Datenstrukturen sind dabei nicht abhängig davon, was für Arten von Objekten gespeichert werden.
- Die Art und Weise des Zugriffs ist ausschließlich abhängig von der Datenstruktur und nicht abhängig von den zu speichernden Objekttypen.
- In JAVA werden diese Arten von Datenstrukturen als Collections bezeichnet.

Collections

FACH HOCHSCHULE LÜBECK
University of Applied Sciences

| Datenstrukturen | Unterschiede zu Arrays | Ausprägungen |
|---|---|--|
| <ul style="list-style-type: none">• Verwaltung von Mengen von Daten• Daten werden gekapselt abgelegt• Zugriff auf die Daten über spezielle Methoden | <ul style="list-style-type: none">• Müssen nicht typrein sein• Können zur Laufzeit in Ihrer Größe verändert werden• Und sind damit flexibler einsetzbar | <ul style="list-style-type: none">• List – dynamische Liste• Stack – Stapel• Map – Key, Value Paare <p>Weitere werden in dieser LV nicht behandelt.</p> |

Prof. Dr. rer. nat. Nane Kratzke
Praktische Informatik und betriebliche Informationssysteme 45

Collections

FACH HOCHSCHULE LÜBECK
University of Applied Sciences

Prof. Dr. rer. nat. Nane Kratzke
Praktische Informatik und betriebliche Informationssysteme 46

Collections

List

Lineare Liste

Elemente beliebigen Typs (können jedoch auf einen Referenztyp eingeschränkt werden)

Länge zur Laufzeit veränderbar

Einfügen und Löschen von Elementen

Sequentieller und wahlfreier Zugriff



Collections

List – Einfügen und Löschen von Elementen

```
public boolean add(Object o)
```

Fügt ein Element *o* der Liste am Ende hinzu

```
public boolean add(int i, Object o)
```

```
public Object set(int i, Object o)
```

set Ersetzt ein Element an der Stelle *i* (vorheriges Element wird zurück gegeben)

Fügt ein Element *o* der Liste an der Stelle *i* hinzu. Alle folgenden Elemente werden weitergeschoben. Liefert als Rückgabe ob add erfolgreich war oder nicht.

```
public Object remove(int i)
```

Löscht das Element an der Stelle *i* in der Liste und liefert es zurück.

Collections

List – Informationen über Listen



```
public Object get(int i)
```

Liefert das i.
Element der Liste.

```
public boolean isEmpty()
```

Prüft, ob eine Liste
leer ist.

```
public int size()
```

Gibt die Anzahl
der Elemente in
einer Liste zurück.

Alle weiteren Methoden dieser Datenstruktur finden Sie unter:

<http://download.oracle.com/javase/7/docs/api/java/util/List.html>

Collections

List – Miniübung I



```
List v = new LinkedList();  
System.out.println(v);
```

[]

```
v.isEmpty();
```

true

```
v.size();
```

0

```
v.add(5);  
v.add(6);
```

[5, 6]

```
v.isEmpty();
```

false

```
v.size();
```

2

```
v.add(1, "Einschub");
```

[5, Einschub, 6]

```
v.remove(1);
```

[5, 6]

Collections

List – Sequentieller Elementzugriff



Sequentieller Elementzugriff auf eine Liste ist mittels eines Iterators möglich.

```
public Iterator iterator()
```

Liefert einen Iterator über eine Liste.

Ein Iterator bietet zwei Methoden an:

```
public boolean hasNext()
```

True, wenn noch weitere Elemente in der Aufzählung, sonst False.

```
public Object next()
```

Liefert das nächste Element der Aufzählung

Prof. Dr. rer. nat. Nane Kratzke
Praktische Informatik und betriebliche Informationssysteme

51

Collections

List – Miniübung III



```
List v = new ArrayList();  
v.add(1);  
v.add(2);  
v.add(3);
```

[1, 2, 3]

```
Iterator e = v.iterator();
```

[1,2,3]

```
e.hasNext();
```

true

```
e.next ()
```

1 und [1, 2, 3]

```
e.hasNext();
```

true

```
e.next ()
```

2 und [1, 2, 3]

```
e.hasNext();
```

true

```
e.next ()
```

3 und [1, 2, 3].

```
e.hasNext();
```

false

Prof. Dr. rer. nat. Nane Kratzke
Praktische Informatik und betriebliche Informationssysteme

52

Collections

List – Iterator Schleife über eine Liste

```
List v = new LinkedList();  
v.add(„Eins“);  
v.add(„Zwei“);  
v.add(1, „Drei“);  
  
Iterator it = v.iterator();  
  
while (it.hasNext()) {  
    System.out.println(it.next());  
}
```

Eins
Drei
Zwei

Diese Variante ohne Enumerator ginge auch:

```
for (int i = 0; i < v.size(); i++) {  
    System.out.println(v.get(i));  
}
```

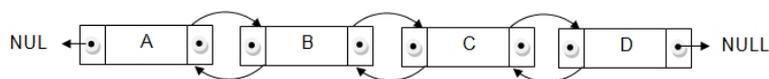
Diese Variante mit foreach Schleife ginge auch:

```
for (Object o : v) System.out.println(o);
```

Unterschiedliche Implementierungen von Listen hinter derselben Schnittstelle

Softwaretechnisch können Listen auf unterschiedlichste Arten implementiert werden.

In JAVA bietet es sich an, Objekte jeweils mit einer Referenz auf den Vorgänger und den Nachfolger in einer Liste zu verlinken. Dieses Prinzip nennt man `LinkedList`.



Man kann aber auch die Datenstruktur Array dazu nutzen, um eine Liste im Hauptspeicher zu speichern. Dieses Prinzip nennt man dann `ArrayList`.



Für den Zugriff auf die Liste und den Umgang mit der Liste ändert dies nichts.

LinkedList und ArrayList Vorteile und Nachteile



University of Applied Sciences

| | Vorteil | Nachteil |
|------------|---|--|
| LinkedList | <ul style="list-style-type: none">• Schnelles Einfügen | <ul style="list-style-type: none">• Langsamer wahlfreier Zugriff• höherer Speicherverbrauch |
| ArrayList | <ul style="list-style-type: none">• Schneller wahlfreier Zugriff• geringer Speicherverbrauch | <ul style="list-style-type: none">• Langsames Einfügen |

Da eine `ArrayList` auf die semidynamische Datenstruktur `Array` zurückgreift, kann eine Einfügeoperation nur sehr aufwändig realisiert werden, da ein `Array` zur Laufzeit nicht vergrößert werden kann.

Es muss erst ein größeres `Array` angelegt werden, dann alle Daten aus dem alten `Array` umkopiert werden, und dann das alte `Array` gelöscht werden.

Da Listen häufig sequentiell (also nicht wahlfrei) durchlaufen werden, ist es daher ratsam, `LinkedLists` zu nutzen. Erst wenn Laufzeitprobleme oder Speicherprobleme bei großen Listen auftreten, sollte man den Einsatz von `ArrayList` in Erwägung ziehen.

Prof. Dr. rer. nat. Nane Kratzke
Praktische Informatik und betriebliche Informationssysteme

55

LinkedList und ArrayList implementieren beide die List Schnittstelle



University of Applied Sciences

Egal welche Implementierung genutzt wird. Auf die Datenstruktur sollte man immer nur über die Schnittstelle `List` zugreifen.

Dies ermöglicht es, nachträglich die zugrunde liegende Listenimplementierung in einer Zeile zu ändern, ohne den Rest der Programmierung anpassen zu müssen.

Die `List` Schnittstelle ist eine klassische objektorientierte Lösung. Wie diese Mechanismen im Einzelnen funktionieren, werden wir im weiteren Verlauf der Vorlesung noch behandeln.

```
// Listen daher bitte immer so anlegen
List alist = new ArrayList();
List llist = new LinkedList();

// Niemals so (obwohl der Compiler nicht meckern würde)
ArrayList alist = new ArrayList();
LinkedList llist = new LinkedList();
```

Prof. Dr. rer. nat. Nane Kratzke
Praktische Informatik und betriebliche Informationssysteme

56

Miniübung:



Gegeben sei folgende Liste:

```
List v = new LinkedList();  
for (int i = 0; i < 10; i++) v.add(i);
```

Entwickeln Sie eine Methode `invert`, um eine `Liste` oben angegebener Art rückwärts in folgender Form als `String` zurückzugeben:

9-8-7-6-5-4-3-2-1-0

Miniübung:



Entwickeln Sie bitte zwei Methoden `firstElement` und `lastElement`, die das jeweils erste und letzte Element einer Liste zurückliefern.

Ist die Liste leer oder `null` sollen beide Methoden `null` als Ergebnis liefern.

Collections

FACH HOCHSCHULE LÜBECK
University of Applied Sciences

Map List Stack

Prof. Dr. rer. nat. Nane Kratzke
Praktische Informatik und betriebliche Informationssysteme

59

Collections

Stack

FACH HOCHSCHULE LÜBECK
University of Applied Sciences

- Kellerspeicher
- Elemente beliebigen Typs (kann auf einen Referenztyp eingeschränkt werden)
- Länge zur Laufzeit beliebig veränderbar.
- Zugriff nach dem LIFO Prinzip
Last In First Out

Prof. Dr. rer. nat. Nane Kratzke
Praktische Informatik und betriebliche Informationssysteme

60

Collections Stack – Operationen

```
public void push(Object o)
```

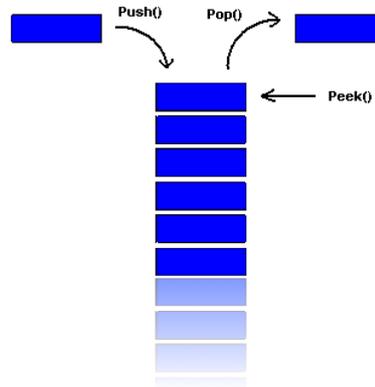
Legt ein Element o auf den Stack.

```
public Object peek()
```

Liest das oberste Element des Stacks.

```
public Object pop()
```

Liest das oberste Element des Stacks und löscht es vom Stack.



Collections Stack – Informationen über Stack

```
public boolean isEmpty()
```

Prüft, ob ein Stack leer ist.

```
public int size()
```

Gibt Anzahl der Elemente eines Stack zurück.

Alle weiteren Methoden dieser Datenstruktur finden Sie unter:

<http://download.oracle.com/javase/7/docs/api/java/util/Stack.html>

Collections Stack – Miniübung

```
Stack s = new Stack();  
s.push(1);  
s.push(2);  
s.push(3);
```

s:[1, 2, 3]

```
s.size();
```

3

```
s.isEmpty();
```

false

```
s.peek();
```

3 und s:[1, 2, 3]

```
s.pop();
```

3 und s:[1, 2]

```
s.pop();
```

2 und s:[1]

```
s.pop();
```

1 und s:[]

```
s.isEmpty();
```

true

```
s.pop();
```

Fehler: EmptyStackException

Miniübung:



Gegeben sei folgende Liste:

```
List v = new LinkedList();  
for (int i = 0; i < 10; i++) v.add(i);
```

Entwickeln Sie eine Methode `invert`, um eine `Liste` oben angegebener Art rückwärts in folgender Form als `String` zurückzugeben. Nutzen Sie die Datenstruktur `Stack`, um diese Methode zu implementieren.

9-8-7-6-5-4-3-2-1-0

Collections

FACH HOCHSCHULE LÜBECK
University of Applied Sciences

Prof. Dr. rer. nat. Nane Kratzke
Praktische Informatik und betriebliche Informationssysteme

65

Collections

Map

FACH HOCHSCHULE LÜBECK
University of Applied Sciences

- Assoziativer Speicher (A verweist auf B)
- Elemente beliebigen Typs (können auf einen Referenztyp eingeschränkt werden)
- Länge zur Laufzeit beliebig veränderbar.
- Zugriff nach dem Key Value Prinzip

Prof. Dr. rer. nat. Nane Kratzke
Praktische Informatik und betriebliche Informationssysteme

66

Collections

Map – Einfügen und Löschen von Elementen

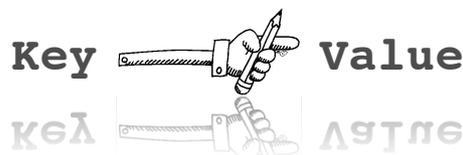


```
public Object put(Object key, Object value)
```

Fügt das **key value** Paar in die Map ein. Falls bereits ein **key value** Paar existiert wird **value** ausgetauscht und der alte **value** Wert zurückgegeben. Andernfalls wird **null** zurückgegeben.

```
public Object remove(Object key)
```

Löscht das **key value** Paar mit dem Schlüssel **key**. Gibt den unter **key** gespeichert Wert zurück oder **null**, falls der **key** nicht unter den Schlüsseln der Map gespeichert war.



Prof. Dr. rer. nat. Nane Kratzke
Praktische Informatik und betriebliche Informationssysteme

67

Collections

Map – Zugriff auf Elemente



```
public Object get(Object key)
```

Liefert den **value** aus der Map, der unter **key** abgelegt wurde. Liefert **null**, wenn der Schlüssel nicht in der Hashtable abgelegt ist.

```
public boolean containsValue(Object value)
```

Liefert **true**, wenn **value** in den Werten der Map vorhanden ist. Andernfalls **false**.

```
public boolean containsKey(Object key)
```

Liefert **true**, wenn **key** in den Schlüsseln der Map vorhanden ist. Andernfalls **false**.

Prof. Dr. rer. nat. Nane Kratzke
Praktische Informatik und betriebliche Informationssysteme

68

Collections

Map – Miniübung I



```
Map h = new HashMap();  
h.put('a', "Hello");  
h.put('c', "World");  
h.put('x', "_");  
System.out.println(h);
```

h:{ a=Hello, c=World, x=_ }

```
h.remove('c');
```

World und h:{ a=Hello,
x=_ }

```
h.remove('z');
```

null und h:{ a=Hello, x=_ }

```
h.put('x', "World");
```

_ und h:{ a=Hello,
x=World }

Prof. Dr. rer. nat. Nane Kratzke
Praktische Informatik und betriebliche Informationssysteme

69

Collections

Map – Zugriff auf alle Keys und Values



```
public List values()
```

Liefert eine Liste über alle Werte der Map.

```
public Set keySet()
```

Liefert eine Menge aller Schlüssel der Map (Set ist eine spezielle Liste in der keine doppelten Werte vorkommen).

Alle weiteren Methoden dieser Datenstruktur finden Sie unter:

<http://download.oracle.com/javase/7/docs/api/java/util/Map.html>

Prof. Dr. rer. nat. Nane Kratzke
Praktische Informatik und betriebliche Informationssysteme

70

Collections Hashtable – Miniübung II



```
Map h = new HashMap();  
h.put('a', "Hello");  
h.put('c', "World");  
h.put('x', „Strange“);
```

h: { a=Hello,
c=World,
x=Strange }

```
Iterator e = h.values().iterator();  
while (e.hasNext()) {  
    System.out.println(e.next());  
}
```

Strange
World
Hello
*Reihenfolge kann
variieren*

```
e = h.keySet().iterator();  
while (e.hasNext()) {  
    System.out.println(e.next());  
}
```

c
a
x
*Reihenfolge kann
variieren*

Prof. Dr. rer. nat. Nane Kratzke
Praktische Informatik und betriebliche Informationssysteme

71

Unterschiedliche Implementierungen von Maps hinter derselben Schnittstelle



Softwaretechnisch können Maps (ähnlich wie Listen) auf verschiedene Arten realisiert werden. Die Frage ist, wie man mit dem Schlüssel umgeht.

In JAVA werden u.a. die folgenden zwei Varianten hierzu angeboten.

- Die Schlüssel werden mittels eines Baums gespeichert und sortiert (wie dies im Einzelnen funktioniert werden wir noch im weiteren Verlauf der Vorlesung sehen). Hierzu nutzt man die Datenstruktur **TreeMap**.
- Die Schlüssel werden gehashed. Dies ermöglicht einen unsortierten aber sehr schnellen Zugriff auf die Werte innerhalb der Map. Hierzu nutzt man die Datenstruktur **HashMap**.

Wie bei `LinkedList` und `ArrayList` über die Schnittstelle `List` angesprochen werden können, so können auch `HashMap` und `TreeMap` über die Schnittstelle `Map` angesprochen werden.

Der Unterschied soll an einem kleinen Beispiel deutlich gemacht werden.

Prof. Dr. rer. nat. Nane Kratzke
Praktische Informatik und betriebliche Informationssysteme

72

Beispiel: HashMap



University of Applied Sciences

```
// Anlegen einer Map
Map<String, String> telefonbuch = new HashMap<String, String>();

// Eintragen von Werten in die Map
telefonbuch.put("Peter", "0451-123456");
telefonbuch.put("Klaus", "0451-234156");
telefonbuch.put("Armin", "0451-623145");

// Ausgeben der Map (Telefonbuch)
for (String name : telefonbuch.keySet()) {
    System.out.print ("Nummer von " + name + ":\t");
    System.out.println(telefonbuch.get(name));
}
```

Ausgabe des Programms:

Nummer von Klaus: 0451-234156

Nummer von Armin: 0451-623145

Nummer von Peter: 0451-123456

Die Ausgabe der Telefonnummer erfolgt
in zufälliger Reihenfolge der Keys.

**Die Reihung ist bei einer HashMap
nicht mal vorhersagbar!**

Prof. Dr. rer. nat. Nane Kratzke
Praktische Informatik und betriebliche Informationssysteme

73

Beispiel: TreeMap



University of Applied Sciences

```
// Anlegen einer Map
Map<String, String> telefonbuch = new TreeMap<String, String>();

// Eintragen von Werten in die Map
telefonbuch.put("Peter", "0451-123456");
telefonbuch.put("Klaus", "0451-234156");
telefonbuch.put("Armin", "0451-623145");

// Ausgeben der Map (Telefonbuch)
for (String name : telefonbuch.keySet()) {
    System.out.print ("Nummer von " + name + ":\t");
    System.out.println(telefonbuch.get(name));
}
```

Ausgabe des Programms:

Nummer von Armin: 0451-623145

Nummer von Klaus: 0451-234156

Nummer von Peter: 0451-123456

Die Ausgabe der Telefonnummer
erfolgt nun in Ordnung der Keys.

**Die Reihung ist somit bei einer
TreeMap vorhersagbar!**

Prof. Dr. rer. nat. Nane Kratzke
Praktische Informatik und betriebliche Informationssysteme

74

Miniübung:



Gegeben sind diese beiden Arrays von Matrikelnummern und Namen. Führen Sie diese beiden Arrays in einer `Map` zusammen, so dass die Namen den Matrikelnummern zugeordnet werden.

Die Arrays sind korrekt geordnet – so dass die Zuordnungen passen.

```
int[] matrnrns = { 565432, 675938, 889554, 886532 };  
String[] names = { "Max Mustermann", "Maren Musterfrau",  
                  "Tessa Loniki", "Wilder Wutz" };
```

Typsicherheit bei Collections



- Sie haben Collections bislang als einen Sammelbehälter von Werten kennengelernt, der Werte beliebigen Typs (`Object`) aufnehmen kann.
- Häufig ist dies gar nicht erforderlich und auch nicht gewollt.
- Alle vorgestellten Datentypen können in einer typfreien bzw. typgebundenen Variante genutzt werden.
- Sie haben bislang nur die typfreie vorgestellt bekommen.

Wie mache ich Collections typsicher?



Beispiel für List

```
List v = new LinkedList(); // Wertetypen in der Liste egal
```

```
List<Integer> v = new LinkedList<Integer>();  
// Es sind nur noch Integers (int) in der Liste zugelassen
```

Beispiel für Stack

```
Stack s = new Stack(); // Wertetypen im Stapel egal
```

```
Stack<String> v = new Stack<String>();  
// Es sind nur noch Zeichenketten im Stapel zugelassen
```

Wie mache ich Collections typsicher?



Beispiel für Map

```
Map h = new HashMap(); // Wertetypen im Verzeichnis egal
```

```
Map<String, Double> h = new HashMap<String, Double>();  
// Es sind nur noch Zeichenketten als Key  
// und Fließkommazahlen (double) als Wert in der Map  
// zugelassen
```

Auf die genauen Hintergründe dieses Konzepts werden wir in der Vorlesung im Teil Generizität eingehen.

Wie mache ich Collections typsicher?

| Primitiver Datentyp | Zu nutzender Datentyp in Collection (Referenztypentsprechung) |
|---------------------|---|
| char | Character |
| int | Integer |
| short | Short |
| byte | Byte |
| boolean | Boolean |
| String | String |
| double | Double |
| float | Float |

Der Java Compiler sorgt für die automatische Umsetzung von primitiven Datentypen in Referenztypen (**Autoboxing**).

Miniübung:



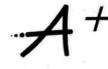
Gegeben seien folgende Quelltexte. Wie können Sie diese typsicherer machen?

```
List v = new LinkedList();  
for (int i = 1; i <= 1000; i++) v.add((double)i);
```

```
Map h = new HashMap();  
for (int i = 1; i <= 1000; i++) h.put("Key " + i, Math.pow(i, 2));
```

```
Stack s = new Stack();  
int[] is = { 9, 7, 6, 5, 10, 1000, 345, -267 };  
for (int i : is) s.push("Value: " + Math.pow(i, 2));
```

Zusammenfassung



- **List**
 - Volldynamische Datenstruktur einer Liste
 - Ausprägung als `LinkedList` und `ArrayList`
- **Stack**
 - Volldynamische Datenstruktur eines Kellerspeichers
- **Map**
 - Volldynamische Datenstruktur eines Assoziativspeichers
 - Key-Value Paare
 - Ausprägung als `HashMap` und `TreeMap`
- **Typsicherheit bei Collections**

