

⋮
Bastie – Sebastian Ritter

Make Java

• • • • • • • •

Collections - Sammlungen

Collection steht für Sammlungen einer unbestimmten Anzahl von Instanzen. Auch wenn Sammlung der korrekte deutsche Begriff wäre ist dieser jedoch so ungebräuchlich, dass wir diesen hier nicht weiter verwenden wollen.

Das Erstellen von Instanzen einer Klasse ist für das Referenzieren von einzelnen oder wenigen Objekten einfach und praktisch. In anderen Fällen ist dies jedoch nicht praktikabel. Typische Fälle hierfür ist die Verwaltung sehr großer Mengen von Instanzen oder auch wenn die Anzahl der notwendigen Instanzen nicht vorab feststeht.

Objektorientierte Programmiersprachen lösen dieses Problem, indem Ihnen eine besonderer Kategorie von Objekten zur Verfügung gestellt wird – die Collections, welche für die Verwaltung anderer Objekte entworfen wurden.

In diesem Kapitel werden Sie lernen:

- ☞☞ Die Eigenschaften der drei generischen Typen: Sortierte Listen, Sets und Verzeichnisse.
- ☞☞ Die wesentlichen Unterschiede zwischen den von Java bereitgestellten Collections und den Zugriff auf die klassischen Arrays.
- ☞☞ Wie die logische Paketstruktur der Collection Klassen und wie Sie die notwendigen Klassen importieren müssen.
- ☞☞ Wie uns die Collections erlauben mit unserem Model die reale Welt darzustellen.
- ☞☞ Welche Techniken helfen Ihnen Ihre eigenen Collections zu erstellen.

Was sind Collections

Collections erlauben es uns eine Anzahl von Instanzen identisch zu behandeln und zu verwalten. Dabei ist der Zugriff auf einzelne Objekte ebenso gegeben, wie die Möglichkeit die Instanzen in ihrer Gesamtheit zu behandeln. Zwei Beispiele:

- ☞☞ Als Angestellter in Ihrer Firma sind Sie einer von vielen. Für alle Angestellten dieser Firma muss an einem bestimmten Stichtag der Arbeitslohn überwiesen werden. Wenn Sie umziehen wird jedoch nur Ihre Adresse geändert.
- ☞☞ Sie als selbständiger Softwareentwickler haben mit vielen Kunden zu. Sie werden jedem Kunden ein individuelles Angebot auf den Leib schneiden. Die Weihnachtsgrußkarte wollen Sie jedoch an alle Ihre Kunden versenden.

Eine Collection erfüllt genau diese Aufgaben. Dabei sind Collections per default nicht synchronisiert und 0-basiert (Null-basiert), d.h. bei Zugriff durch mehrere Threads sind wir für die Synchronisation zuständig und das erste Element liegt an Position 0.

Collections - eine besondere Form von Klassen

Die Programmiersprache Java stellt eine Anzahl von vordefinierten Collections zur Verfügung. Genauso wie die meisten Klassen müssen Collections für die Verwendung initialisiert werden. Hierzu wird zuerst die Referenzvariable für unsere Collection deklariert:

```
CollectionTyp<elementTyp> referenzVariable;
```

In der Umsetzung bedeutet dies

```
LinkedList<Angestellter> arbeitnehmer;
```

Arbeitnehmer ist nunmehr unsere Referenz auf eine Liste von Instanzen des Typs Angestellter. Damit wir diese Liste auch nutzen können müssen wir noch eine Initialisierung vornehmen:

```
referenzVariable = new CollectionTyp<elementTyp> ();
```

was in unserer Umsetzung zu

```
arbeitnehmer = new LinkedList<Angestellter> ();
```

führt. Hierdurch haben wir eine leere Collection erzeugt in der wir Instanzen unserer Angestellten verwalten können.

Collections organisieren die Referenzen auf Instanzen



Ein wesentliches Faktum ist, dass in einer Collection nicht die Instanzen an sich gehalten werden sondern lediglich Referenzen auf diese Instanzen – die Adresse an welcher sich das Instanz in der JVM befindet.

Vergleichen Sie dies mit den Helium Luftballons die Ihr Kind in der Hand hält. In Wirklichkeit hält Ihr Kind nicht die Luftballons in der Hand sondern die Schnüre, die von der Hand jeweils zu genau einem Luftballon führen.

Eine Collection ist insoweit mit der Hand vergleichbar.

Kapselung der Collection

Die konkrete Implementierung der jeweiligen konkreten Collection ist vor uns versteckt. Für jede `java.util.Collection` sind lediglich einige wenige Zugriffsmethoden definiert. Zu diesen gehören:

☞☞Hinzufügen von Instanzen.

☞☞Entfernen von Instanz.

☞☞Zugriff auf eine Instanz.

☞☞Iterieren durch die Collection in einer vordefinierten Reihenfolge.

☞☞Information über die Anzahl der Instanz in der Collection.

☞ Information, ob eine Instanz in der Collection gespeichert ist.

☞ Bereinigen der Collection.

Auch wenn wir hier häufig von Instanzen die Rede ist so denken Sie bitte immer an unsere Luftballons – in der Collection stehen eigentlich nur Referenzen auf die Instanz.

Drei elementare Gruppen von Collections

Es gibt elementare Collections, die durch objektorientierte Programmiersprachen unterstützt werden sollten:

☞ Set als eine ungeordnete Sammlung von Instanzen.

☞ List als eine geordnete Sammlung von Instanzen.

☞ Map als eine Sammlung von Instanzen, die über einen eindeutigen Schlüssel ansprechbar sind.

Hierbei beinhalten Collections grds. keine abschließende Festlegung zu der Anzahl der Instanzen die in ihnen gespeichert werden können.

Set

Eine ungeordnete Sammlung von Instanzen – `java.util.Set<E>` - kann mit der Handtasche Ihrer besten Freundin¹ verglichen werden. In einer ungeordneten Sammlung kann hierbei jede Instanz nur einmal abgelegt werden (vgl. `boolean equals()`). In unserem Java SDK finden wir drei Implementationen im Collection Framework: `HashSet`, `TreeSet` und `LinkedHashSet`.

List

Eine Liste ist eine Collection, wo die Instanzen geordnet abgelegt werden. Wenn wir eine Instanz in der Collection ablegen können wir zu einem späteren Zeitpunkt diese Instanz an genau dieser Stelle in der Collection wieder auffinden. Die einzelnen Instanzen werden auf Basis ihrer Position in unserer Liste abgelegt. Anders als ein Set ist es möglich mehrere gleiche Instanzen in einer Liste abzulegen. Wenn wir eine Instanz ablegen wird diese grds. am Ende der Liste angehängen. Entfernen wir eine Instanz wird die entstehende Lücke durch aufrücken der nachfolgenden Instanzen geschlossen. Ein Arrays als einfache Collections hingegen hat u.a. diese Funktionalität nicht. Das Java SDK liefert uns die `LinkedList` und `ArrayList` mit.

Map

Eine Map ist eine Collection in der einem eindeutigen Schlüssel eine Instanz zugeordnet wird, das Schlüssel Wert Paar (key value pair). Der Schlüssel basiert hierbei in der realen Welt meist auf ein Attribut der Instanz z.B. der

¹ Sorry reine Erfahrungssache: Die Handtasche einer Frau ist (meist) wie ein schwarzes Loch...

Personalnummer unseres Angestellten. Wir haben mit dem Java SDK bereits die `HashMap`, `TreeMap` und `LinkedHashMap` zur Verfügung.

Arrays als einfache Collections

Eine einfache Sammlung die nicht zum Collection Framework gehört ist das Array. Ein Array ist insoweit vergleichbar mit einer List, da das grundsätzliche Verhalten weitestgehend identisch ist. Dabei ist ein Array nach seiner Definition jedoch nicht mehr in seiner Größe veränderbar.

Deklarieren und Definieren von Arrays

Ein Array kann für beliebige – auch primitive Datentypen – angelegt werden. Die Deklaration

```
datentyp [] variable;
```

wird durch die Definition in ihrer Größe festgelegt

```
variable = datentyp [(int) groesse];
```

z.B. legt

```
Angestellter [] imBuero = new Angestellter [15];  
double [] monatsgehaltsSumme = new double [12];
```

fest, daß in unserem Array `imBuero` maximal 15 Instanzen von `Angestellte` abgelegt werden können sowie daß wir zwölf `double` Werte in `monatsgehaltsSumme` ablegen können. Ein Array kann alternativ auch über `0` Reflection erzeugt werden. Das folgende Beispiel legt ein Array für Instanzen vom Typ `Object` in der Größe 20 an.

```
Object [] x =  
(Object []) Array.newInstance(Class.forName("java.lang.Object"), 20);
```

Zugriff auf einzelne Elemente eines Arrays

Für den direkten Zugriff auf die Instanzen eines Arrays wird die Position der Instanz in eckigen Klammern `[]` angegeben. Dadurch wird die Instanz zurückgeliefert und kann entsprechend Ihrer Klassenschnittstelle verwendet werden; bei primitiven Datentypen wird der Wert zurück geliefert.

```
imBuero [0].getPersonalnummer ();  
monatsgehaltsSumme [11];
```

Hier greifen wir auf den ersten Angestellten bzw. die Summe der Gehälter des Monats Dezember zu.

Zugriff auf alle Elemente eines Arrays

Für den Zugriff auf alle Elemente ist es notwendig auf jedes Element einzeln zuzugreifen. Hierfür nutzen wir einfach die Zählschleife `for`. Diese kann in Java in zwei Formen eingesetzt werden. Zum einem die „alte“ Form und seit Java 5 die komfortable Form:

```
// for Schleife  
for (int i = 1; i < this.fakultaet.length; i++) {  
    this.fakultaet [i] = this.fakultaet [i-1] * 2;  
}
```

```

}
// for Schleife Java 5
for (int i : this.fakultaet) {
    System.out.println(i);
}

```

Die „alte“ Form nutzt hierbei das öffentliche Attribut `length` zum Schleifenabbruch. Sollten wir über das Ende des Arrays hinaus zugreifen wollen so bekommen wir eine `IndexArrayOutOfBoundsException`. Bei Instanzen können wir zudem gleich auf die Methoden zugreifen.

```

imBuero [0].getPersonalnummer ();

```

Dabei müssen wir jedoch beachten, dass Arrays eines nicht primitiven Datentyps `grds.` alle Referenzen mit `null` initialisiert werden (`boolean` Arrays mit `false` und die anderen primitiven Datentypen mit `0`). Sollten wir also keine gültige Instanz an Index `0` abgelegt haben, bekommen wir eine `NullPointerException`.

Initialisieren des Arrayinhaltes

Das Initialisieren eines Arrays können wir bereits bei der Definition vornehmen. Hierzu werden die einzelnen Elemente in einen Block durch Komma getrennt aufgeführt. Bei Deklaration, Definition und Initialisierung in einer Anweisung:

```

int [] zahlen1bis10 = {1,2,3,4,5,6,7,8,9,10};

```

Diese Kurzschreibweise ist jedoch nicht bei „Wiederverwendung“ unserer Referenzvariablen möglich. Hierbei müssen wir die Langschreibweise nutzen:

```

zahlen1bis10 = new int [] {1,2,3,4,5,6,7,8,9,10};

```

Natürlich hätten wir die gleiche Zuweisung auch statt der Kurzschreibweise hinter der Deklaration verwenden können.

Manipulieren von Arrays

So wie der Zugriff auf ein Array durch Indexierung in eckigen Klammern erfolgt, können auch Zuweisungen erfolgen. Hierbei müssen wir bedenken, dass im Array nur die Referenzen abgelegt werden. Wir manipulieren insoweit auch nur diese Referenzen, durch setzen neuer Referenzen in ein Array:

```

imBuero [0] = new Angestellter ("Meier");
imBuero [1] = new Angestellter ("Müller");
// u.s.w.

```

Dies können wir mit dem Zugriff auf alle Elemente eines Arrays verbinden, wenn wir z.B. nur einen Konstruktor ohne Parameter haben.

```

for (Angestellter a : imBuero) {
    a = new Angestellter ();
}

```

java.util.ArrayList

Ein Array ist ideal, wenn wir bereits wissen wie viele Instanzen wir maximal bekommen werden, noch besser wenn die Anzahl der Instanzen sich nicht mehr ändert. Doch häufig ist es nur schwer oder gar unmöglich die maximale Anzahl zu

ermitteln. Selbst wenn dies möglich ist kann es uneffizient sein ein großes Array anzulegen indem nur wenige Instanzen aufgenommen werden.

Deshalb bietet uns Java dynamische Sammlungen im Collection Framework, welche mit der Anzahl der aufgenommen Instanzen wachsen oder schrumpfen. Eine der häufig verwendet Collection ist die `ArrayList`.

Verwendung der ArrayList Class am einfachen Beispiel

Das folgende Programm „de.bastie.collection.BeispielArrayListNow“ zeigt die Nutzung einer `ArrayList` um Angestellte zu verwalten. Zuerst das Programm bevor wir uns mit den einzelnen Komponenten beschäftigen werden.

```
1 package de.bastie.collection;
2
3 import de.bastie.now.Now;
4 import java.util.*;
5
6 public final class BeispielArrayListNow implements Now {
7
8     private ArrayList<Angestellter> imBuero = new
        ArrayList<Angestellter>();
9
10    public BeispielArrayListNow () {
11    }
12
13    public void init () throws Throwable {
14        imBuero.add(new Angestellter(1));
15        imBuero.add(new Angestellter(2));
16        imBuero.add(new Angestellter(3));
17    }
18
19    public void go () throws Throwable {
20        for (Angestellter a : this.imBuero) {
21            System.out.println ("Personalnummer: "+a.getPersonalnummer());
22        }
23    }
24 }
```

package und import Anweisung

In Zeile 1 geben wir an, dass unsere Klasse in dem Paket `de.bastie.collection` liegt. In Zeile 3 machen wir das Interface `Now` und in Zeile 4 alle Typen im Paket `java.util` bekannt. Wenn wir mit Collections arbeiten finden wir die notwendigen Java Typen in diesem Paket.

Namensraum unserer Klasse

Ab Zeile 6 beginnt schließlich unsere Klasse. Sie kann nicht mehr erweitert werden und implementiert das Interface `Now`. In `BeispielArrayListNow` haben wir einen Standardkonstruktor, eine Instanzvariable und zwei Methoden definiert.

In Zeile 8 deklarieren und definieren / instanzieren wir zunächst unsere Collection. Erinnern wir uns: Eine Collection muss für die Nutzung wie die meisten Klassen zuerst instanziiert werden. Der Standardkonstruktor hat keinerlei Besonderheiten. In den Zeilen 14 bis 16 fügen wir unserer `ArrayList` einige Instanzen hinzu. In der Zeile 20 führen wir schließlich unsere Zählschleife aus, um auf alle unsere Instanzen wieder zuzugreifen (Zeile 21).

Ein Blick in die ArrayList

Die Klasse ArrayList besitzt zahlreiche öffentliche Methoden – viele davon werden von allen Collection Klassen bereitgestellt.

boolean add (E instanz)

Fügt zur ArrayList eine Instanz hinzu. Diese wird hinten an die Liste angehängen. E steht dabei für den Typ den wir bei der Initialisierung unserer ArrayList mitgeben. Beispiel:

```
ArrayList<Angestellter> imBuero = new ArrayList<Angestellter>();  
imBuero.add (new Angestellter ());
```

void add (int index, E instanz)

Fügt an dem angegebenen Index eine Instanz hinzu. Die anderen Elemente werden nach hinten in der Liste verschoben.

void clear ()

Entfernt alle Instanzen aus der ArrayList. Sobald die Garbage Collection keine Referenzen mehr auf die betroffenen Instanzen findet, werden diese aus der JVM entfernt. Alternativ kann auch mit new eine neue ArrayList erzeugt werden. Beispiel:

```
ArrayList<Angestellter> imBuero = new ArrayList<Angestellter>();  
imBuero.clear ();  
imBuero = new ArrayList<Angestellter>();
```

boolean contains (Object instanz)

Wird benutzt um zu prüfen, ob die übergebene Instanz in der ArrayList bekannt ist. Dabei muss es sich um das gleiche Objekt handeln.

int size ()

Liefert die Anzahl der Instanzen welche in unserer ArrayList sind. Diese Methode ist analog zur öffentlichen Variable length in einem Array zu verwenden.

boolean isEmpty ()

Ermöglicht abzufragen, ob noch Instanzen in unserer ArrayList vorhanden sind.

E get (int index)

Liefert die Instanz an dem benannten Index zurück. E steht dabei für den Typ den wir bei der Initialisierung unserer ArrayList mitgeben

boolean remove (Object instanz)

Entfernt eine einzelne Instanz aus der ArrayList. Im folgenden Beispiel entfernen wir nacheinander jeweils das erste Element.

```
while (!imBuero.isEmpty()) {  
    imBuero.remove (imBuero.get (0))  
}
```

```
}
```

Von der ArrayList zum Array

Manchmal ist es notwendig aus einer ArrayList ein Array zu erzeugen. Die Methode `toArray` ermöglicht dabei die Referenzen in einem Array abzulegen.

```
ArrayList<Angestellter> imBuero = new ArrayList<Angestellter>();  
Angestellter [] bueroAngestellte = new Angestellter [imBuero.size()];  
imBuero.toArray (bueroAngestellte);
```

java.util.HashMap

Eine HashMap ist eine Collection in welche die Instanzen über einen eindeutigen Schlüssel abgelegt werden. Dabei können sowohl Schlüssel als auch Wert von einem beliebigen nicht primitiven Typ sein.

```
1 package de.bastie.collection;  
2  
3 import de.bastie.now.Now;  
4 import java.util.HashMap;  
5  
6 public class BeispielHashMapNow implements Now {  
7  
8     private HashMap<Integer, Angestellter> imBuero = new  
9     HashMap<Integer, Angestellter> ();  
10  
11     public BeispielHashMapNow () {  
12     }  
13  
14     public void init () throws Throwable {  
15         Angestellter a = new Angestellter(0, "Meier");  
16         this.imBuero.put(a.getPersonalnummer(), a);  
17         a = new Angestellter(1, "Müller");  
18         this.imBuero.put(a.getPersonalnummer(), a);  
19         a = new Angestellter(2, "Ritter");  
20         this.imBuero.put(a.getPersonalnummer(), a);  
21     }  
22  
23     public void go () throws Throwable {  
24         Angestellter a = this.imBuero.get(5);  
25         if (a != null) {  
26             System.out.println ("Angestellter \""+a.getNachname()+"\"");  
27         }  
28         else {  
29             System.out.println ("Falsche Personalnummer angegeben.");  
30         }  
31  
32         a = this.imBuero.get(2);  
33         if (a != null) {  
34             System.out.println ("Angestellter \""+a.getNachname()+"\"");  
35         }  
36         else {  
37             System.out.println ("Falsche Personalnummer angegeben.");  
38         }  
39  
40         for (Angestellter alle : this.imBuero.values()) {  
41             System.out.println ("Angestellter  
42             (\"+alle.getPersonalnummer()+") \" "+alle.getNachname()+"");  
43         }  
44     }  
45 }
```

package und import Anweisung

In Zeile 1 geben wir an, dass unsere Klasse in dem Paket `de.bastie.collection` liegt. In Zeile 3 machen wir das Interface `Now` und in Zeile 4 die Klasse `java.util.HashMap` bekannt.

Namensraum unserer Klasse

Ab Zeile 6 beginnt schließlich unsere Klasse. Sie kann nicht mehr erweitert werden und implementiert das Interface `Now`. In `BeispielHashMapNow` haben wir einen Standardkonstruktor, eine Instanzvariable und zwei Methoden definiert.

In Zeile 8 deklarieren und definieren / instanzieren wir zunächst unsere Collection. Erinnern wir uns: Eine Collection muss für die Nutzung wie die meisten Klassen zuerst instanziiert werden. Bei der Instanzierung müssen wir mitteilen, welchen Typ unser Schlüssel haben soll und welchen Typ wir in unserer HashMap ablegen wollen. Wie sagen hier der Schlüssel sei vom Typ `Integer` und wir werden Instanzen von `Angestellter` speichern. Für den Schlüssel selbst werden wir uns später das automatische in/outboxing zu Nutze machen.

In den Zeilen 14 bis 19 erstellen wir mehrere Instanzen vom Typ `Angestellter` und fügen diese unserer HashMap hinzu. In der Zeile 23 lassen wir uns schließlich die Instanz geben, welche unter dem Schlüssel 5 abgelegt ist – moment 5 ist vom Typ `int`?! In diesem Fall können wir ein `int` übergeben, da dieser Wert automatisch in eine `Integer` Instanz umgewandelt wird. Schließlich prüfen wir ob wir eine Instanz unter diesem Schlüssel gefunden haben (Zeile 24). Da wir keine Instanz unter diesem Index abgelegt haben landen wir im `else` Teil der Verzweigung (Zeilen 27 bis 29). In den Zeilen 31 bis 37 führen wir selbiges nochmals durch nur dass wir mit dem Index 2 erfolgreich sind.

In den Zeilen 39 bis 41 arbeiten wir schließlich mit allen Instanzen `Angestellter`. Da wir keine `null` Instanz eingestellt haben, können wir hier auf entsprechende die Prüfung verzichten.

Ein Blick in die HashMap

Object put (Object schluessel, Object wert)

Fügt zu einer HashMap eine Instanz hinzu. Diese wird unter dem übergebenen Schlüssel abgelegt. Sofern bereits ein Objekt unter diesem Schlüssel abgelegt war, wird dieses zurückgegeben. Beispiel:

```
Angestellter a= new Angestellter(2, "Ritter");
this.imBuero.put(a.getPersonalnummer(), a);
```

void clear ()

Entfernt alle Instanzen aus der HashMap. Sobald die Garbage Collection keine Referenzen mehr auf die betroffenen Instanzen findet, werden diese aus der JVM entfernt. Alternativ kann auch mit `new` eine neue HashMap erzeugt werden. Beispiel:

```
HashMap<Integer, Angestellter> imBuero =
    new HashMap<Integer, Angestellter>();
imBuero.clear ();
imBuero = new HashMap<Integer, Angestellter>();
```

boolean containsKey (Object schluessel)

Wird benutzt um zu prüfen, ob der Schlüssel bereits in der HashMap vorhanden ist. Dabei muss es sich um das gleiche Objekt handeln.

☞☞boolean containsValue (Object wert)

Wird benutzt um zu prüfen, ob der Wert bereits in der HashMap vorhanden ist. Dabei muss es sich um das gleiche Objekt handeln.

☞☞int size ()

Liefert die Anzahl der Instanzen welche in unserer HashMap sind.

☞☞boolean isEmpty ()

Ermöglicht abzufragen, ob noch mindestens ein Schlüssel Wert Paar in unserer HashMap vorhanden sind.

☞☞Object get (Object schluesel)

Liefert den Wert zurück, der unter dem übergebenen Schlüssel gespeichert wurde. Ist kein Wert vorhanden wird null zurückgegeben.

☞☞Object remove (Object schluesel)

Entfernt den Wert der unter dem Schlüssel abgelegt wurde. Ist keine Instanz unter dem Schlüssel abgelegt bekommen wir null zurück.

☞☞Collection values ()

Liefert uns eine Collection nur mit den Werten zurück. Beispiel:

```
for (Angestellter alle : this.imBuero.values()) {
    System.out.println ("Angestellter ("
        +alle.getPersonalnummer()
        +") \"
        +alle.getNachname()
        +'\n');
}
```

java.util.TreeMap

Die TreeMap ist vergleichbar mit der HashMap bietet jedoch die Sicherheit dass wir die Elemente in sortierter aufsteigender Form abgelegt werden.

Erstellen einer eigenen Collection

In diesem Abschnitt wollen wir uns mit der Erstellung einer eigenen Collection befassen. In den meisten Situationen werden die vorgegebenen Collection Klassen für unsere Bedürfnisse ausreichen. Wenn dies jedoch nicht mehr der Fall ist müssen wir uns eine eigene Collection schreiben. Dafür stehen uns grds. zwei Wege offen:

☞☞Implementierung einer neuen Collection

☞☞Erweiterung einer vorhandenen Collection

☞☞Kapselung einer vorhanden Collection

Implementierung einer neuen Collection

Für die Erstellung einer neuen Collection müssen wir das Interface `java.util.Collection` implementieren.

Erweiterung einer vorhandenen Collection

Eine vorhandene Collection zu erweitern ist wesentlich einfacher. Wir werden im folgenden eine `ArrayList` so erweitern, dass wir diese in einen `javax.swing.JTree` einhängen können:

```
44 package de.bastie.collection;
45
46 import de.bastie.now.Now;
47
48 import java.awt.BorderLayout;
49 import java.util.*;
50 import javax.swing.*;
51 import javax.swing.tree.*;
52
53 public class JTreeArrayList extends ArrayList implements TreeNode,
54     Now {
55     private TreeNode parent;
56
57     public JTreeArrayList () {}
58     public JTreeArrayList (final TreeNode parent) {
59         this.parent = parent;
60     }
61
62     public void add(int index, final Object node) {
63         if (node instanceof TreeNode) {
64             super.add(index, node);
65         }
66     }
67
68     public boolean add(Object node) {
69         if (node instanceof TreeNode) {
70             return super.add(node);
71         }
72         else {
73             return false;
74         }
75     }
76
77     public TreeNode getChildAt (int childIndex) {
78         return (TreeNode)this.get(childIndex);
79     }
80
81     public int getChildCount () {
82         return this.size();
83     }
84
85     public TreeNode getParent () {
86         return this.parent;
87     }
88
89     public int getIndex (TreeNode node) {
90         return this.indexOf(node);
91     }
92
93     public boolean getAllowsChildren () {
94         return true;
95     }
96
97     public boolean isLeaf () {
98         return false;
99     }
100
101     public Enumeration children () {
102         return new JTreeArrayListEnumeration(this);
103     }
104 }
```

```

105 private final class JTreeArrayListEnumeration implements
Enumeration{
106     private final Iterator base;
107     public JTreeArrayListEnumeration (JTreeArrayList master) {
108         this.base = master.iterator();
109     }
110     public boolean hasMoreElements () {
111         return base.hasNext();
112     }
113
114     public Object nextElement () {
115         return base.next();
116     }
117 };
118
119 // Now implementation
120 private JTreeArrayList list;
121 public void init () throws Throwable {
122     this.list = new JTreeArrayList();
123     JTreeArrayList german = new JTreeArrayList(this.list);
124     german.add(new DefaultMutableTreeNode ("http://www.Bastie.de"));
125     german.add(new DefaultMutableTreeNode
("http://www.wikibooks.de"));
126     JTreeArrayList english = new JTreeArrayList(this.list);
127     english.add(new DefaultMutableTreeNode ("http://java.sun.com"));
128     list.add(german);
129     list.add(english);
130 }
131
132 public void go () throws Throwable {
133     JFrame f = new JFrame (this.getClass().getName()+" sample");
134     f.setDefaultCloseOperation(WindowConstants.EXIT_ON_CLOSE);
135     JTree tree = new JTree (this.list);
136     f.add (tree, BorderLayout.CENTER);
137     f.pack();
138     f.setVisible(true);
139
140 }
141 }

```

package und import Anweisung

In Zeile 1 geben wir an, dass unsere Klasse in dem Paket `de.bastie.collection` liegt. In Zeile 3 machen wir das Interface `Now` und in den Zeilen 5 bis 8 die für unser Beispiel notwendigen Klassen bekannt.

Namensraum unserer Klasse

Vorab: Die Zeilen 76ff. beinhalten lediglich den Aufbau einer kleinen grafischen Oberfläche, so dass wir diese hier nicht näher betrachten werden. In den folgenden Zeilen sorgen wir lediglich dafür, dass alle Methoden unserer `TreeNode` Implementation auf Methoden unserer `ArrayList` umgelenkt werden.

In der Zeile 10 definieren wir unseren Klassentyp. Hierbei erweitern wir in unserem Beispiel eine `ArrayList` insbesondere um die Funktionalität eines Eintrags (`javax.swing.tree.TreeNode`) in einen Baum (`javax.swing.Tree`). Die Methode `public Enumeration children ()` (Zeile 58) erweist sich hier als einziges Problem, da wir keine `Enumeration` mal eben erzeugen können. Als einfache Lösung gehen wir hier über die angebotene `Iterator` Implementation der `Collection` (Zeile 62 ff.). Durch das einfache Umlenken der `Enumeration` Methoden auf unseren `Iterator` überwinden wir auch diese Hürde.

Collection als Rückgabewert

Eine Collection, wie auch ein Array, ermöglichen uns als Java Entwickler die Beschränkung „eine Methode hat nur einen Rückgabewert“ einzuhalten, ohne den ggf. vorhandenen Anforderungen nach mehreren Ergebnissen einer Methode nicht nachkommen zu können. Beispiel:

```
1 package de.bastie.collection;
2
3 import de.bastie.now.Now;
4
5 import java.util.*;
6
7 public class BeispielMehrereRueckgabewerteEinerMethode implements
  Now {
8     private LinkedList<Angestellter> imBuero = new
  LinkedList<Angestellter>();
9
10    public void init () throws Throwable {
11        Angestellter a = new Angestellter(0, "Meier");
12        this.imBuero.add(a);
13        a = new Angestellter(1, "Müller");
14        this.imBuero.add(a);
15        a = new Angestellter(2, "Ritter");
16        this.imBuero.add(a);
17    }
18
19
20    public Collection<Angestellter> getImBuero () {
21        return this.imBuero;
22    }
23
24    public void go () throws Throwable {
25        for (Angestellter a : this.getImBuero()) {
26            System.out.println(a.getNachname());
27        }
28    }
29 }
```

package und import Anweisung

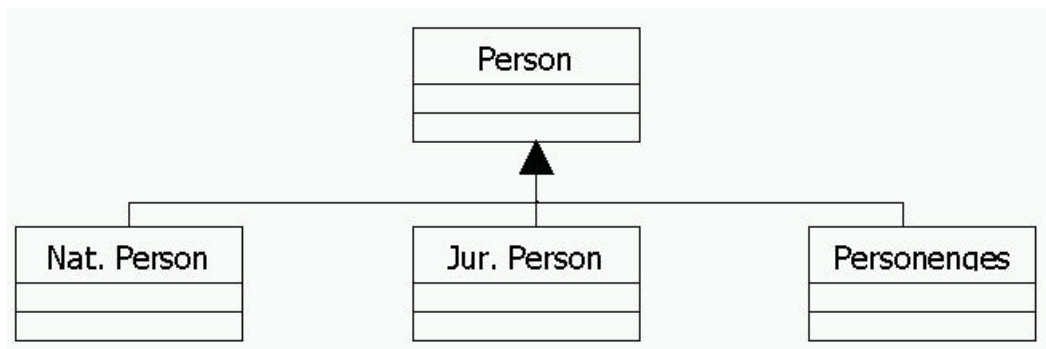
In Zeile 1 geben wir an, dass unsere Klasse in dem Paket `de.bastie.collection` liegt. In Zeile 3 machen wir das Interface `Now` und in der Zeile 5 das Paket `java.util` bekannt.

Namensraum unserer Klasse

In der `init` Methode (Zeile 10 ff.) füllen wir unsere Collection – die `LinkedList` von Zeile 8. In Zeile 25 wollen wir schließlich auf mehrere Angestellte zurückgreifen. Um den Zugriff objektorientiert zu kapseln haben wir uns eine Methode `getImBuero ()` definiert. Da in Java nur Methoden mit einem Rückgabewert erlaubt sind, haben wir diesen als `Collection<Angestellter>` definiert und können somit mehrere Objekte – genauer gesagt deren Referenzen – zurückgeben.

Collection mit unterschiedlichen Typen

Sammlungen sind in der Lage genau einen Typ aufzunehmen. Dies definieren wir durch Angabe des Typs in den spitzen Klammern (`<>`). Wollen wir nun unterschiedliche Typen innerhalb einer Sammlung ablegen haben wir scheinbar ein Problem? Etwas – tatsächlich müssen wir uns auf den größten gemeinsamen Typen einigen.



Am Beispiel dieses kleinen Klassendiagrammes lässt sich dies verdeutlichen. Sofern wir drei Instanzen jeweils von `NatuerlichePerson`, `JuristischePerson` und `Personengesellschaft` haben, können wir diese entweder allgemein in einer `Collection<Object>` ablegen oder spezieller in den größten gemeinsamen Typen `Collection<Person>` ablegen. Zusammen mit der Polymorphie ergibt sich hierdurch ein mächtiges Werkzeug um Instanzen verschiedener Typen gemeinsam zu verwalten.

Zusammenfassung Collection

Sie sollten nunmehr einiges über die Java Collection gelernt haben:

- ☞ Eine Collection ist ein spezieller Datentyp, welcher Referenzen auf Instanzen verwaltet.
- ☞ Sie können eine eigene Collection z.B. durch erweiteren der vorhandenen Klassen erzeugen.
- ☞ Eine Collection ist eine Möglichkeit mehrere Rückgabewerte einer Methode zu definieren.