7. Die Klasse ArrayList

7.1 Sammlungen in Java

In der Programmierung begegnet uns häufig das Konzept der ***Sammlung***: Eine Gruppe von Objekten, die gemeinsam verwaltet werden. Während in Kapitel 4 schon die Arrays als strukturierte "Datenbehälter" mit fester Länge behandelt wurden, gehen wir jetzt einen Schritt weiter. Denn oft reicht ein einfaches Array nicht aus - zum Beispiel, wenn die Anzahl der Objekte zur Laufzeit nicht bekannt ist oder sich ständig ändert.

Die Verwaltung einer Bibliothek ist ein typisches Beispiel für den Einsatz von Sammlungen. Anfangs hat man nur ein paar Bücher zu verwalten, vielleicht 20 oder 30. Im Laufe der Jahre kommen immer mehr dazu, und die Sammlung wächst beständig.

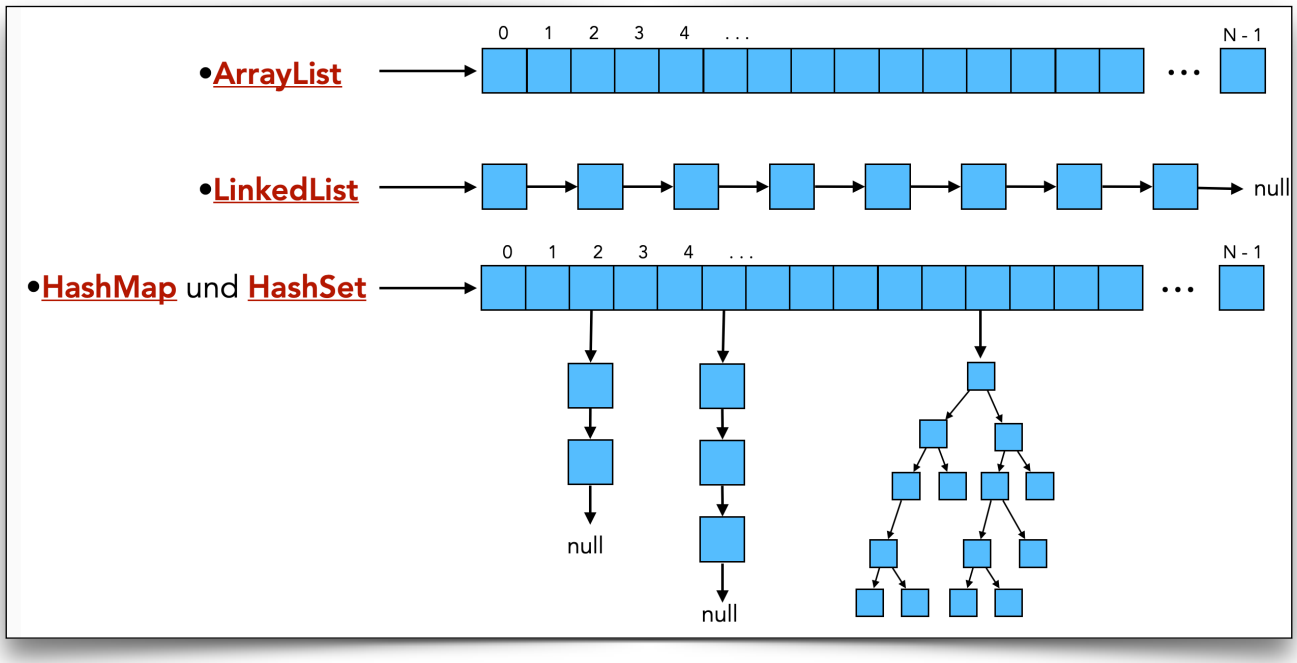
Ein anderes Beispiel ist ein Deutsch-Englisch-Lexikon, in dem sowohl nach englischen Übersetzungen für deutsche Begriffe wie auch umgekehrt nach deutschen Übersetzungen für englische Worte gesucht werden kann.

Die wichtigste und auch einfachste Sammlungsklasse ist **ArrayList**. In einem Objekt der Klasse **ArrayList** kann man zwar keine primitiven Daten wie `int`- oder `double`-Zahlen speichern, wohl aber Objekte von Klassen. Im Gegensatz zu einem Array hat eine Arrayliste keine von vornherein festgelegte Obergrenze. Eine Arrayliste kann bei Bedarf wachsen, man spricht hier von einer ***dynamischen Datenstruktur*** - im Gegensatz zu statischen Datenstrukturen wie Arrays.

7.1.1 Wichtige Sammlungsklassen in Java

- **ArrayList**: lineare Liste von Objekten
- **LinkedList**: wie ArrayList, hat aber andere Stärken: Einfügen und Löschen in der Nähe eines gegebenen Elements ist günstig, der Zugriff per Index ist aber langsam.
- **HashSet**: dient zum Verwalten von Mengen, keine Duplikate, keine feste Reihenfolge
- **HashMap**: dient zum Verwalten von Wertepaaren, zum Beispiel für ein Deutsch-Englisch-Wörterbuch.

In diesem Skript gehen wir hauptsächlich auf die wichtige Klasse **ArrayList** ein. Am Ende betrachten wir aber auch noch kurz die Klassen **LinkedList** und **HashMap**.



Diese Zeichnung stellt die vier wichtigen Sammlungsklassen graphisch in stark vereinfachter Form dar.

ArrayList: Basiert intern auf einem Array. Dieses Array ist zunächst fest (statisch), wird aber bei Bedarf automatisch vergrößert, sodass die Liste nach außen hin "dynamisch" erscheint.

LinkedList: ist Ihnen aus dem Informatik-Unterricht vielleicht als "einfach verkettete Liste" bekannt. Eine echte LinkedList ist allerdings doppelt verkettet.

HashMap: ein Array, dessen Elemente als Buckets bezeichnet werden. Wenn mehr als ein Element in einem Bucket gespeichert werden soll, verweist der Bucket auf eine einfach verkettete Liste von Elementen. Wird eine solche Liste zu lang, wird sie (seit Java 8) in einen balancierten binären Suchbaum umgewandelt.

HashSet: Eine HashMap, bei der jedes Element nur einmal vorkommen darf, wie bei einer richtigen Menge.

7.2 Erzeugung einer ArrayList

7.2.1 Einführendes Beispiel

Codebeispiel

```
1  import java.util.ArrayList;
2
3  public class Stringlist
4  {
5      ArrayList <String> wortliste;
6
7      public Stringlist()
8      {
9          wortliste = new ArrayList<>();
10     }
11 }
```

Zeile 1:

Damit wir Objekte der Klasse **ArrayList** verwenden können, muss diese Klasse zunächst aus der Java-Standardbibliothek importiert werden. Die Klasse **ArrayList** befindet sich im Paket **java.util**, daher ist die Import-Anweisung erforderlich.

Zeile 5:

Hier wird das **ArrayList**-Objekt `wortliste` deklariert. Mit der Typangabe `<String>` wird festgelegt, dass diese Arrayliste ausschließlich Objekte der Klasse **String** speichern darf. Diese Schreibweise beruht auf dem Konzept der ***Generics***.

Generics

Mit Generics kann man festlegen, welche Art von Objekten in einer Klasse oder Liste gespeichert werden darf. Bei einer `ArrayList<String>` sind das zum Beispiel nur Zeichenketten vom Typ `String`. Dadurch wird der Umgang mit solchen Listen einfacher und sicherer.

Die **Festlegung auf einen bestimmten Datentyp (**hier: `String`) hat sowohl Vor- als auch Nachteile.

Ein **Vorteil** besteht darin, dass der Datentyp der gespeicherten Elemente von vornherein feststeht. Bei `wortliste` wissen wir also immer, dass sich in der Liste nur `String`-Objekte befinden.

Der Compiler überprüft dies bereits beim Übersetzen des Programms und verhindert, dass Objekte anderer Klassen eingefügt werden. Dadurch wird der Quelltext sicherer, weniger fehleranfällig und übersichtlicher. Außerdem sind beim Auslesen der Elemente keine umständlichen Typumwandlungen erforderlich.

Ein **Nachteil** besteht darin, dass man mit einer solchen Liste keine ***heterogenen Listen*** anlegen kann, also keine Listen, die Objekte verschiedener Klassen enthalten.

Eine `ArrayList<String>` ist ausschließlich für `String`-Objekte geeignet.

Zeile 9:

Initialisierung von `wortliste`. Der ***Diamant-Operator*** `<>` ersetzt die Angabe `<String>`. Diese ist ein Beispiel für ***Typinferenz***.

Typinferenz

Typinferenz bezeichnet die Fähigkeit des Java-Compilers, den Datentyp eines Ausdrucks automatisch aus dem Kontext abzuleiten.

Vor Java 7 musste man bei der Verwendung von Generics beispielsweise schreiben:

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

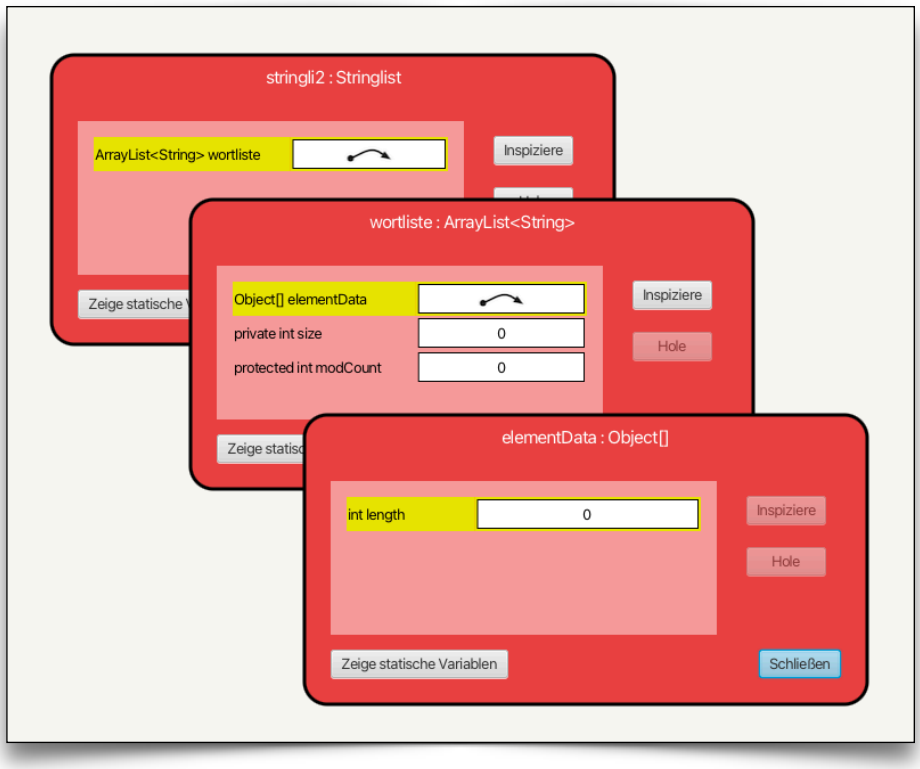
Seit Java 7 kann man den Diamant-Operator einsetzen:

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

Bei der Initialisierung des **ArrayList**-Objekts muss also nicht mehr der Datentyp (hier `<String>`) angegeben werden, sondern es reicht der Diamant-Operator `<>`. Der Compiler erkennt dann automatisch, dass hier eine `String`-Arrayliste erzeugt werden soll.

7.2.2 Untersuchung eines neuen ArrayList-Objekts

Das Objekt `wortliste` im BlueJ-Objektinspektor:



Im Objektinspektor sieht man bei **ArrayList**-Objekten drei interne Variablen:

1. elementData

Intern speichert ein **ArrayList**-Objekt seine Elemente in einem gewöhnlichen Array des Typs `Object[]`. Die Elemente dieses Arrays sind Referenzen auf die Objekte der Liste.

Entwickler haben normalerweise keinen Zugriff auf das Array `elementData`, da es sich um ein internes Implementierungsdetail handelt, das von außen nicht sichtbar ist. Der BlueJ-Objektinspektor oder andere Tools können dieses Array jedoch sichtbar machen.

2. size

Hier wird die aktuelle Größe des **ArrayList**-Objekts gespeichert.

Dabei ist darauf zu achten, dass `size` nicht identisch mit `elementData.length` ist. Eine Arrayliste kann die Länge 16 haben, wenn sie aber nur 12 Elemente enthält, hat `size` den Wert 12.

3. modCount

Mit dieser Instanzvariable wird Buch darüber geführt, wie oft das **ArrayList**-Objekt verändert wurde.

Anmerkung zu elementData und BlueJ

Dass BlueJ `elementData` anzeigen kann, widerspricht nicht dem ***Information Hiding***: Der Zugriffs-Modifizierer `private` (vom Objektinspektor nicht angezeigt!) verhindert den Zugriff aus normalem Programmcode, aber nicht die Anzeige durch Werkzeuge zur Fehlersuche (Debugger/Inspector).

7.3 Das Wachstum der Liste

Mit der Methode `add()` kann man einer Liste Objekte hinzufügen und dadurch wachsen lassen. Das Hinzufügen **primitiver Datentypen**. wie `int` oder `double` zu einer Liste ist jedoch nicht auf direktem Wege möglich. Solche Datentypen müssen vor dem Hinzufügen mithilfe der ***Wrapper-Klassen*** in Objekte umgewandelt werden.

7.3.1 Zwei Varianten der add( )-Methode

Der folgende Quelltext enthält beide Varianten der `add()`-Methode.

```
import java.util.ArrayList;

public class Stringlist
{
    ArrayList <String> wortliste;

    public Stringlist()
    {
       wortliste = new ArrayList<>();
       addDemo();
    }

    public void addDemo()
    {
       wortliste.add("Hans");        // Variante 1
       wortliste.add("Petra");
       wortliste.add("Susanne");
       wortliste.add("Maik");
       wortliste.add("Tim");
       wortliste.add(2,"Kay");       // Variante 2
    }
}
```

Variante 1:

Die erste Variante des `add()`-Befehls fügt das Element an das *Ende* der vorhandenen Liste ein. Intern - aber das sieht der Benutzer nicht - wird das Element in den nächsten freien Platz des internen Arrays `elementData` eingefügt.

Variante 2:

Die zweite Variante des `add()`-Befehls fügt das Element in der angegebenen Position ein. Die folgenden Elemente rücken dann weiter nach rechts auf, dabei wird die Liste gegebenenfalls vergrößert (dynamisches Wachstum).

Primitive Datentypen in der Liste und Auto-Boxing

```
import java.util.ArrayList;

public class WrapperDemo
{
    public static void main(String[] args)
    {
        ArrayList zahlen = new ArrayList<>();

        // Primitive int-Werte:
        int a = 12;
        int b = 25;

        // Umwandlung in Integer-Objekte und Hinzufügen zur ArrayList
        zahlen.add(Integer.valueOf(a));
        zahlen.add(Integer.valueOf(b));

        System.out.println(zahlen);
    }
}
```

Die ArrayList `zahlen` speichert keine `int`-Werte, sondern Objekte der Klasse **Integer**. Deshalb werden die primitiven Werte `a` und `b` vor dem Hinzufügen mit **Integer**.valueOf(...) in passende ***Wrapper-Objekte*** umgewandelt.

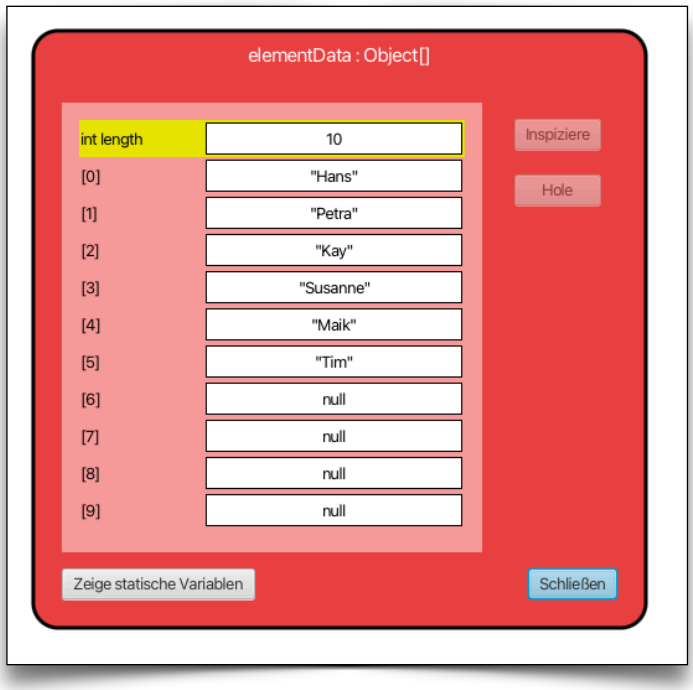
Die neueren Java-Versionen ab 5 ermöglichen auch ein einfacheres Verfahren, das als ***Autoboxing*** bezeichnet wird:

```
zahlen.add(a);
zahlen.add(b);
```

Beim Autoboxing übernimmt Java diese Umwandlung automatisch.

7.3.2 Aufbau von elementData

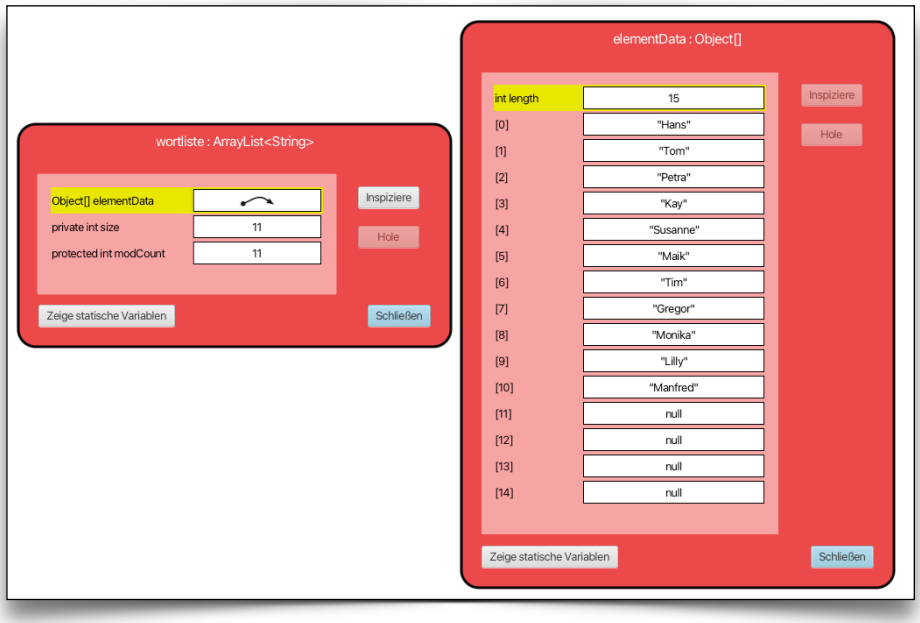
Der BlueJ-Objektinspektor zeigt uns, wie das Array `elementData` des **ArrayList**-Objekts `wortliste` genau aufgebaut ist:



"Hans" und "Petra" befinden sich an ihrer ursprünglichen Position, an Position 2 wurde "Kay" eingefügt, und die anderen drei Einträge sind nach hinten gerückt.

Interessant ist, dass das Array `elementData` eine ***Kapazität*** von zehn Elementen hat. Es ist in dem statischen Objekt-Array also noch Platz für vier weitere Elemente.

Wir fügen jetzt aber nicht vier, sondern fünf weitere Elemente zu `wortliste` hinzu. Dann untersuchen wir das Objekt wieder mit dem BlueJ-Objektinspektor.



Das Array `elementData` des **ArrayList**-Objekts `wortliste` hat jetzt eine ***Kapazität*** (`length`) von 15 Elementen, die ***Größe*** (`size`) ist aber 11.

Kapazität und Größe einer ArrayList

Die Kapazität (`length`) beschreibt die **Länge des internen Arrays**, auf dem die Liste basiert. Sie gibt also an, wie viele Elemente maximal gespeichert werden könnten, ohne dass ein neues Array angelegt werden muss.

Die Größe (`size`) hingegen gibt stets die **tatsächliche Anzahl** der aktuell in der Liste gespeicherten Objekte an.

7.3.3 "Dynamisches" Wachstum eines ArrayList-Objekts

Wir wollen uns das "dynamische" Wachstum einer Arrayliste genauer ansehen und führen dazu ein kleines Experiment durch. Warum das Wort dynamisch in Anführungszeichen steht, werden wir am Ende dieses Abschnitts sehen.

Experiment

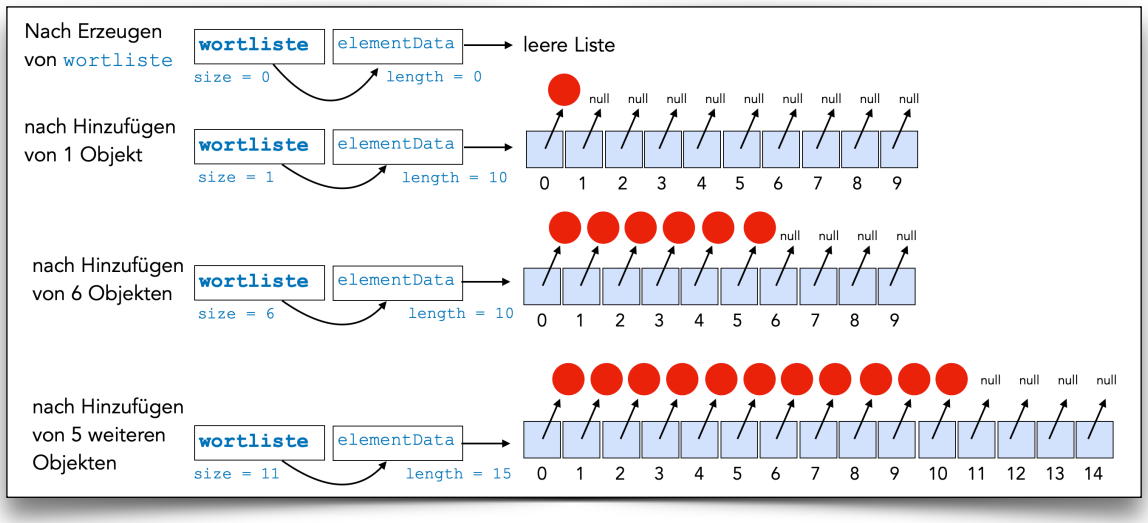
Bisher hatte das Array `elementData` eine Kapazität von 15 und eine Größe von 11. Wir fügen jetzt sechs weitere Elemente hinzu und betrachten `elementData` im BlueJ-Objektinspektor:

Das Array `elementData` umfasst jetzt 22 Elemente mit den Indizes 0 bis 21. Also ist das Array um 22 / 15 = 1,467 gewachsen, gerundet also 1,5.

In der aktuellen Java-Version wird die neue Kapazität der Liste nach der Formel

```
neue Kapazität = alte Kapazität + alte Kapazität / 2
```

berechnet. Diese Vergrößerungsregel ist aber ein **internes Implementationsdetail** und kann sich ändern. Für die Verwendung der **ArrayList**-Objekte ist nur wichtig, dass sie sich nach außen wie eine dynamisch wachsende Liste verhalten.



Dieses Bild zeigt das "dynamische" Wachstum einer ArrayList beim Hinzufügen neuer Einträge.

Nach Erzeugung von wortliste

Nach dem Erzeugen des **ArrayList-**Objekts `wortliste` ist die Größe (`size`) von `wortliste` gleich 0 und die Kapazität (`length`) des eigentlich verborgenen internen Arrays `elementData` ist ebenfalls 0. Mit dem BlueJ-Objektinspektor (oder anderen Tools) kann man dieses Array dennoch sehen und inspizieren. Aber eine Anweisung wie

```
String s = wortliste.elementData[1];
```

liefert eine Fehlermeldung des Compilers: `elementData ist nicht öffentlich`.

Nach Hinzufügen des ersten Objekts

Wenn der Liste das erste Objekt hinzugefügt wurde, erhält `size` den Wert 1, während `elementData` jetzt erst initialisiert wird und danach 10 Elemente umfasst.

Die Pfeile in den 10 Elementen von `elementData` sollen veranschaulichen, dass nicht die konkreten Objekte in `elementData` gespeichert sind, sondern Referenzen bzw. Verweise auf solche Objekte. Verweist ein Array-Element *nicht* auf ein Objekt, hat es den Wert `null`.

Nach Hinzufügen von fünf weiteren Objekten

Wenn wir noch weitere fünf Objekte zur Liste hinzufügen, werden die nächsten fünf "freien Plätze" der Liste aufgefüllt; sie enthalten jetzt Verweise auf die konkreten Objekte. Die restlichen vier Plätze der Liste enthalten weiterhin den Wert `null`.

Nach der erneuten Aufnahme von fünf Objekten

Die Liste hat noch Platz für genau vier weitere Elemente. Was passiert aber, wenn wir fünf neue Elemente hinzufügen wollen?

In diesem Fall setzt das "dynamische" Wachstum des internen Arrays `elementData` ein. Das Array der Länge 10 wird um den Faktor 1,5 verlängert, also um fünf weitere Elemente. Daher hat `length` jetzt den Wert 15. Die fünf neuen Objekte können also problemlos

aufgenommen werden. Danach hat size Wert 11, und es ist noch Platz für vier weitere Objekte in der Liste.

Simulation des "dynamischen" Wachstums mit einem statischen Array

Die folgende Java-Methode demonstriert, wie ein normales statisches int-Array `zahlen` um den Faktor 2 wachsen kann:

```
public void simuliereWachstum()
{
   int[] kopie = new int[zahlen.length * 2];
   for (int i=0; i < zahlen.length; i++)
      kopie[i] = zahlen[i];

   zahlen = kopie;
}
```

Es wird einfach lokal eine neue Array-Variable `kopie` angelegt, die doppelt so groß ist wie das ursprüngliche Array. Dann werden alle Elemente von `zahlen` in das neue Array hineinkopiert.

Der entscheidende Schritt erfolgt ganz zum Schluss: Die Referenz-Variable `zahlen`, die auf die konkreten Daten im Speicher verweist, wird mit `zahlen = kopie` auf das neu erzeugte und doppelt so große Array umgebogen.

Für den außenstehenden Betrachter ist die Referenz-Variable `zahlen` also immer noch vorhanden, nur verweist `zahlen` jetzt auf einen doppelt so großen Bereich im Speicher als vorher.

Warum "dynamisch" in Anführungszeichen steht

Von echtem dynamischem Wachstum spricht man dann, wenn eine Datenstruktur intern Element für Element wächst.

Bei einem **ArrayList**-Objekt ist das anders: Es arbeitet intern mit dem ***statischen*** Objekt-Array `elementData`.

Wird dieses Array zu klein, wird nicht *ein Element angehängt*, sondern es wird ein *neues, größeres Array erzeugt* und die bisherigen Elemente werden dorthin *kopiert*.

Ein ArrayList-Objekt wirkt nach außen also dynamisch, intern basiert es aber auf einem statischen Array.

Ein Beispiel für richtiges dynamisches Wachstum

Das folgende Quelltext-Beispiel ist noch aus einem anderen Grund hoch interessant: Es zeigt, wie man eine ***innere Klasse*** definiert. Hier schon mal vorab ein Auszug aus diesem Quelltext:

```
public class Stack
{
    public class Knoten
    {
       Object wert;
       Knoten next;

       public Knoten(Object w)
       {
          wert = w;
          next = null;
       }
    }

    private Knoten tos;

    public Stack()
    {
        tos = null;
    }

    public boolean isEmpty()
    {
       return (tos == null);
    }

    // weitere Methoden
}
```

Innere Klassen

Die innere Klasse **Knoten** ist hier rot markiert.

Alle Methoden der äußeren Klasse **Stack** können auf die innere Klasse **Knoten** zugreifen, auch auf die zwei Instanzvariablen `wert` und `next`, selbst wenn diese als `private` deklariert wären, was hier aber nicht der Fall ist.

Auch alle Klassen im gleichen Paket wie **Stack** können auf die öffentliche innere Klasse zugreifen, auch auf die beiden Instanzvariablen. Klassen außerhalb des Pakets können dagegen nicht auf die Instanzvariablen und den Konstruktor von Knoten zugreifen.

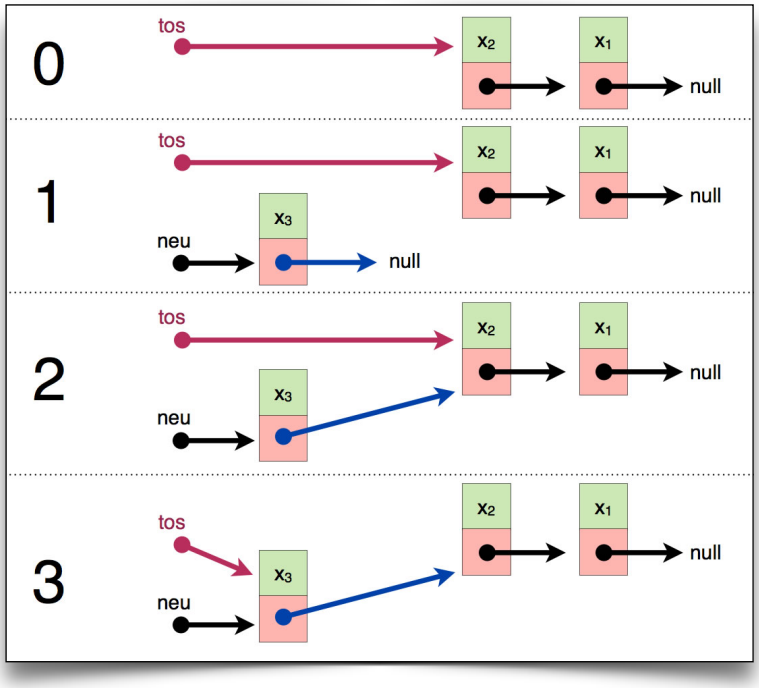
7.3.4 Echtes dynamisches Wachstum am Beispiel Stack

Das Thema "Dynamische Datenstrukturen" wird normalerweise in der Qualifikationsphase des Oberstufen-Unterrichts behandelt und ist in einigen Bundesländern auch Gegenstand der schriftlichen und mündlichen Abiturprüfungen. Auf der Seite "Ein dynamischer Stack" können Sie dies alles nachlesen, wenn Sie meinen, dass noch Nachholbedarf besteht.

Hier finden Sie eine Kurzfassung des Themas.

Die Klasse Stack

```
public class Stack
{
    public class Knoten
    {
       Object wert;
       Knoten next;

       public Knoten(Object w)
       {
          wert = w;
          next = null;
       }
    }

    private Knoten tos;

    public Stack()
    {
        tos = null;
    }

    public boolean isEmpty()
    {
       return (tos == null);
    }

     public void push(Object x)
     {
          Knoten neu = new Knoten(x);
          neu.next = tos;
          tos = neu;
     }

    public void pop()
    {
       if (!isEmpty())
          tos = tos.next;
    }

    public Object top()
    {
       if (!isEmpty())
          return tos.wert;
       else
          return null;
    }
}
```



Das Bild zeigt die Arbeitsweise der Methode `push()`.

1. Ein neuer Knoten `neu` wird erzeugt.
2. Der `next`-Zeiger des neuen Knotens wird auf `tos` gesetzt, zeigt also auf das bisherige oberste Element des Stacks.
3. Der `tos`-Zeiger wird auf das neue Element gesetzt, das jetzt das oberste des Stacks ist.

Der Stack ist "echt" dynamisch um ein Element gewachsen.

Anmerkung: Hier ist von "Zeigern" die Rede, korrekt wäre eigentlich der Begriff "Referenz-Variable", was den Text aber unübersichtlicher gemacht hätte.

7.4 Weitere wichtige Methoden

7.4.1 size()

Die Methode `size()` liefert die ***Anzahl*** der tatsächlich vorhandenen Elemente der Liste zurück. Dabei muss `size` nicht mit `length` identisch sein. Nach dem Hinzufügen des 11. Elements hatte `wortliste` eine ***Größe*** (`size`) von 11, aber eine ***Kapazität*** (`length`) von 15, und nach dem Experiment aus Abschnitt 7.3.3 war die Kapazität um den Faktor 1,5 auf 22 gewachsen, die Größe - also `size` - hatte den Wert 17.

Codebeispiel

```
public void sizeDemo()
{
   System.out.print("Die ArrayList hat eine Größe von ");
   int groesse = wortliste.size();
   System.out.println(groesse);
}
```

7.4.2 get()

Die Methode `get()` liefert das ***Objekt*** zurück, das sich an der Position `index` in der Liste befindet.

Codebeispiel

```
// neu erzeugte ArrayList nach Hinzufügen von 8 Elementen
// length = 10, size = 8

public void getDemo(int index)
{
   System.out.print("Das Wort an Position " + index + " ist: ");
   String wort = wortliste.get(index);
   System.out.println(wort);
}

public void test()
{
   getDemo(10);
}
```

Der Aufruf `get(9)` liefert eine `IndexOutOfBoundsException`. BlueJ und IntelliJ melden dann: `"Index 9 out of bounds for length 8"`.

Entscheidend ist nicht die interne ***Kapazität*** (hier 10) des zugrunde liegenden Arrays, sondern die aktuelle ***Größe*** der Liste (hier 8), also die Zahl der tatsächlich gespeicherten Elemente.

Die interne Kapazität ist für Benutzer normalerweise nicht sichtbar (Information Hiding); der BlueJ-Objektinspektor zeigt sie jedoch an. Die Fehlermeldung verwendet das Wort `length`, gemeint ist hier aber die Anzahl gültiger Elemente, also `size`.

Einsatz einer for-each-Schleife anstelle von get()

Codebeispiel

```
int pos = 0;

 for (String wort : wortliste)
 {
     System.out.print("Das Wort an der Position " + pos + " ist: ");
     System.out.println(wort);
     pos++;
 }
```

Durch die ***for-each-Schleife*** wird in diesem Beispiel ein Zugriff mit dem `get()`-Befehl überflüssig.

In der lokalen Variable `wort` steht nacheinander jedes Element von `wortliste` und kann dann weiterverarbeitet werden.

Vorteile der for-each-Schleife

1. Die Schleife startet automatisch beim ersten Element (index = 0).
2. Sie endet automatisch nach dem letzten Element (index = size-1).
3. Es ist keine Laufvariable und kein Index notwendig.

Die for-each-Schleife

Die for-each-Schleife ist die einfachste Möglichkeit, alle Elemente einer ArrayList der Reihe nach zu verarbeiten, wenn die genaue Position der Elemente keine Rolle spielt und auf den Index nicht zugegriffen werden muss.

7.4.3 set()

Mit dieser Methode kann man ein bestimmtes Element der Liste durch ein anderes *ersetzen* bzw. *überschreiben*. Die rechts stehenden Elemente rücken *nicht* weiter nach rechts, und die Größe (`size`) der Liste bleibt unverändert.

Codebeispiel 1:

```
wortliste.add("Apfel");
wortliste.add("Birne");
wortliste.add("Orange");
wortliste.set(1, "Banane");
```

Das Element `"Birne"` an Position 1 wird durch das neue Element `"Banane"` ersetzt.

Codebeispiel 2:

```
for (int i = 0; i < wortliste.size(); i++)
{
    if (wortliste.get(i).equals("Ornage"))
    {
        wortliste.set(i, "Orange");
    }
}
```

Hier wird der `set()`-Befehl genutzt, um die Liste zu durchsuchen und das falsch geschriebene Wort "Ornage" durch "Orange" zu ersetzen.

Eine for-each-Schleife ist dafür ungeeignet, weil man darin keinen Index hat und daher den Befehl `set(i, "Orange")` nicht verwenden kann.

7.4.4 remove()

In der Klasse **ArrayList** existieren zwei unterschiedliche `remove()`-Methoden, die sich in ihrer Bedeutung deutlich unterscheiden.

1. remove(int index)

Die Signatur dieser Methode lautet:

```
public E remove(int index)
```

Diese Methode entfernt das Element an der angegebenen Position. Alle rechts daneben stehenden Elemente rücken um jeweils eine Position nach links, und `size` verringert sich um 1. Das entfernte Element wird als Rückgabewert zurückgegeben.

Codebeispiel 1:

```
wortliste.add("Apfel");
wortliste.add("Birne");
wortliste.add("Orange");

String entfernt = wortliste.remove(1);
```

Anmerkung: `remove(1)` ist in diesem Beispiel unproblematisch. Würde die Liste aber int-Zahlen als Elemente enthalten, gäbe es ein Problem: Es würde nicht die Zahl 1 entfernt, sondern die Zahl, die sich an Position 1 befindet.

Die Methode remove(int index) kann aber auch ganz einfach wie eine normale manipulierende Methode aufgerufen werden, also ohne Rückgabewert.

Codebeispiel 2:

```
wortliste.add("Apfel");
wortliste.add("Birne");
wortliste.add("Orange");

wortliste.remove(1);
```

Fazit:

Die Methode remove(int index) entfernt das Element an der Position `index` und gibt dieses Element anschließend zurück. Der Rückgabewert kann in einer Variablen gespeichert werden (Codebeispiel 1), muss aber nicht verwendet werden. Java erlaubt es, den Rückgabewert einer Methode zu ignorieren (Codebeispiel 2). In diesem Fall wird das Element zwar aus der Liste entfernt, das entfernte Objekt geht aber für den weiteren Programmablauf verloren.

2. remove(Object o)

Die Signatur dieser Methode ist:

```
public boolean remove(Object o)
```

Diese Methode entfernt das erste Element aus der ArrayList, das als „gleich“ zu `o` erkannt wird. Der Vergleich erfolgt über die Methode `equals()`.

Wenn ein passendes Objekt gefunden wurde, liefert die Methode `true` zurück, ansonsten `false`.

Codebeispiel:

```
boolean erfolg = wortliste.remove("Apfel");

wortliste.remove("Birne");
```

Auch hier sind beide Varianten erlaubt, einmal die Rückgabe eines Wahrheitswerts an eine `boolean`-Variable, und einmal der einfache Aufruf ohne Rückgabe von Werten.

7.5 Noch mehr Methoden

Die Klasse **ArrayList** stellt noch eine ganze Reihe weiterer Methoden zur Verfügung, auf die wir hier aber nur kurz eingehen.

7.5.1 clear()

Diese Methode leert die Liste komplett.

7.5.2 isEmpty()

Hiermit wird geprüft, ob die Liste leer ist, also keine Elemente enthält.

7.5.3 contains(Object o)

Liefert `true`, wenn das Objekt `o` bereits in der Liste vorhanden ist.

7.5.4 indexOf(Object o)

Liefert den Index des ersten Vorkommens von `o` zurück oder `-1`, falls das Objekt nicht in der Liste enthalten ist.

7.6 Aufgaben

7.6.1 Leichte Aufgaben

Aufgabe 1 – Einkaufsliste

Erstellen Sie ein ArrayList-Objekt vom Typ `String`, das eine Einkaufsliste repräsentiert. Fügen Sie mindestens fünf Artikel hinzu.

Lassen Sie anschließend alle Artikel nummeriert auf der Konsole ausgeben (z. B. "1. Milch").

Entfernen Sie danach einen der Artikel und geben Sie die aktualisierte Liste erneut aus.

Aufgabe 2 – Zahlen auswerten

Füllen Sie eine ArrayList vom Typ `Integer` mit zehn ganzen Zahlen.

Lassen Sie alle Zahlen mit einer for-each-Schleife ausgeben.

Berechnen Sie die Summe aller Elemente sowie das Maximum und geben Sie beide Werte auf der Konsole aus.

7.6.2 Mittelschwere Aufgaben

Die mittelschweren Aufgaben sind vom Niveau her mit normalen Klausur-Aufgaben vergleichbar.

Aufgabe 3 – Duplikate entfernen

Erweitern Sie die Klasse aus Aufgabe 2 um eine Methode, die alle doppelt vorkommenden Zahlen aus der ArrayList entfernt, sodass jede Integer-Zahl am Ende nur noch einmal in der Liste steht. Die Reihenfolge der Zahlen muss aber erhalten bleiben.

Aufgabe 4 – Notenverwaltung

Erstellen Sie eine Klasse **Notenverwaltung**, die intern eine `ArrayList<Integer>` verwendet. Die Klasse soll folgende Methoden besitzen:

- `void noteHinzufuegen(int note)` (Werte = 1, 2, 3, 4, 5, 6).
- `double durchschnittBerechnen()` und
- `int besteNote()`.

Schreiben Sie eine Test-Methode, die die Klasse mit einigen Testnoten ausprobiert.

7.6.3 Anspruchsvolle Aufgabe

Bei einer Klausur würde man auf Teilaufgabe 2 verzichten.

Aufgabe 5 – Einfache Studentenverwaltung

Erstellen Sie eine Klasse **Student** mit den Attributen `name` (String) und `matrikelnummer (int)` sowie einem passenden Konstruktor und Getter-Methoden für den Namen und die Matrikelnummer.

Schreiben Sie anschließend eine Klasse **Studentenverwaltung**, die eine `ArrayList<Student>` verwaltet und folgende Methoden bereitstellt:

1. `void hinzufuegen(Student s)` – fügt einen Studenten hinzu.
2. `ArrayList<Student> sucheNachName(String name)` – gibt alle Studenten mit diesem Namen zurück (als neue ArrayList).
3. `void loescheMatrikelnummer(int mnr)` – entfernt den Studenten mit der angegebenen Matrikelnummer.
4. `void alleAusgeben()` – gibt alle gespeicherten Studenten aus.

Testen Sie die Verwaltung mit mindestens fünf Studenten, darunter zwei mit gleichem Namen.

Achten Sie auf eine einfache Fehlerbehandlung bei den Methoden. Wenn Sie noch nicht mit Exceptions gearbeitet haben, verwenden Sie einfach eine Fehlermeldung in der Konsole, gefolgt von einem return-Befehl.

7.6.4 Zusatzaufgabe für Experten

Diese Aufgabe geht deutlich über das Klausur-Niveau hinaus.

Aufgabe 6 – Notenverwaltung Plus/minus

Erstellen Sie eine Klasse **Notenverwaltung**, die intern eine `ArrayList<String>` verwendet. Die Klasse soll folgende Methoden besitzen:

- `void noteHinzufuegen(String note)` (Werte = "1+", "1", "1-", "2+", ... "6").
- `double durchschnittBerechnen()`
- `String besteNote()`.
- `String ausgeschriebeneNote()` (liefert zum Beispiel "sehr gut (minus)" oder "gut").

Schreiben Sie eine Test-Methode, die die Klasse mit einigen Testnoten ausprobiert.

Hinweise:

Eine Plus-Note hat den numerischen Wert `Note - 0,33`. Die 3+ ist also wie 2,67 zu behandeln. Eine Minus-Note hat dagegen den Wert `Note + 0,33`. Die 2- wäre also als 2,33 zu behandeln.

Alternativ könnte man Noten wie 1+, 1, 1-, 2+ durch ein Punktesystem darstellen und damit rechnen: 15, 14, 13, 12 Punkte etc.

7.6.5 Und noch zwei mittelschwere Aufgaben

Aufgabe 7 – Längstes Wort

1. Lassen Sie sich von einer KI ein Array

   ```
   String[] worte = {"Thread", "Array", "Polymorphie", ... };
   ```

   mit mindestens 30 Worten erstellen (gern auch mehr).

2. Lesen die Elemente dieses Arrays dann in eine ArrayList `wortliste` ein, zum Beispiel in einer Methode

   ```
   baueListeAuf().
   ```

3. Erstellen Sie dann eine Methode

   ```
   public String laengstesWort()
   ```

   die das längste Wort der Liste zurückliefert.

Aufgabe 8 - Bubblesort

Ergänzen Sie den Quelltext aus Aufgabe 7 um eine Methode

```
public void bubblesort()
```

welche die ArrayList nach diesem Algorithmus sortiert. Als Hilfsmethode dürfen Sie nur

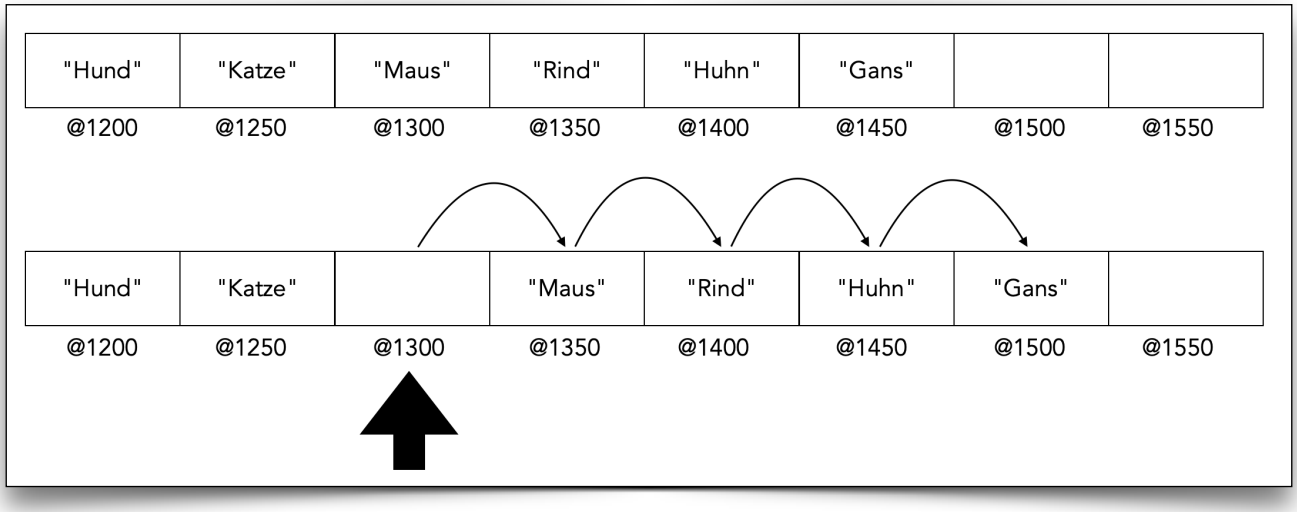
```
private void tausche(int i, int j)
```

verwenden, die zwei Elemente der ArrayList mit den entsprechenden ArrayList-Methoden vertauscht. Diese Methode müssen Sie allerdings auch erst selbst schreiben.

7.7 Die Klasse LinkedList

7.7.1 Vor- und Nachteile einer ArrayList

Betrachten wir den internen Aufbau einer kleinen Arrayliste einmal in einem vereinfachten Schema:



Dieses Bild zeigt den (vereinfacht dargestellten) internen Aufbau eines kleinen ArrayList-Objekts aus sechs Elementen. Die Zahlen unter den Kästchen sollen die relative Speicheradresse symbolisieren.

Vorteile einer ArrayList:

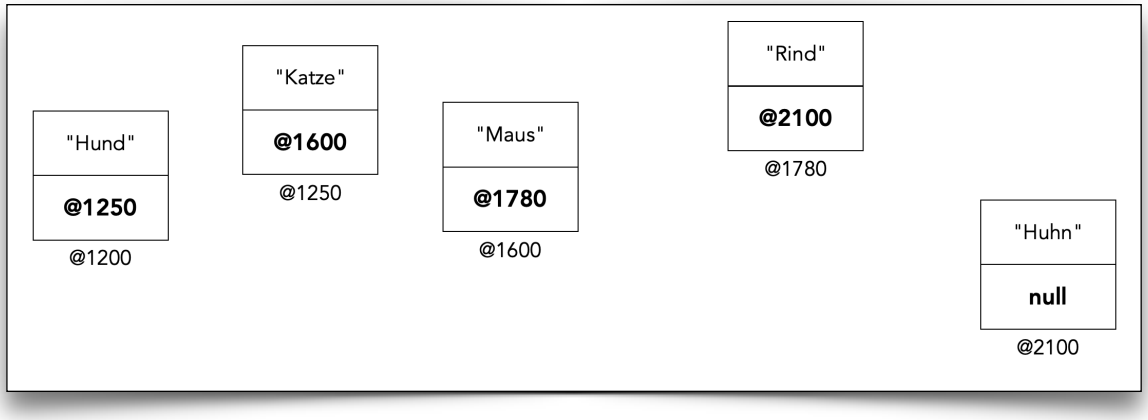
- Es ist ein direkter Zugriff auf einzelne Elemente möglich. Wenn beispielsweise mit `get(3)` auf das Element mit dem Index 3 zugegriffen werden soll, wird die Speicheradresse des ersten Elements genommen (`@1200`) und dann um 3 x 50 inkrementiert.
- Eine ArrayList kann zudem einfach mit einer for-each-Schleife durchlaufen und dabei zum Beispiel ausgegeben oder durchsucht werden.

Nachteile einer ArrayList:

- Beim Einfügen am Anfang (oder in der Mitte) müssen viele Elemente nach rechts verschoben werden.
- Beim Löschen von Elementen müssen alle nachfolgenden Elemente um eine Position nach links verschoben werden.

7.7.2 Vor- und Nachteile einer LinkedList

Aufbau einer LinkedList



Eine LinkedList besteht aus einzelnen ***Knoten***. Jeder Knoten enthält

- die eigentlichen Daten (hier zum Beispiel "Hund", "Katze" etc.)
- eine oder mehrere ***Referenzen*** auf andere Knoten (hier zum Beispiel @1250).

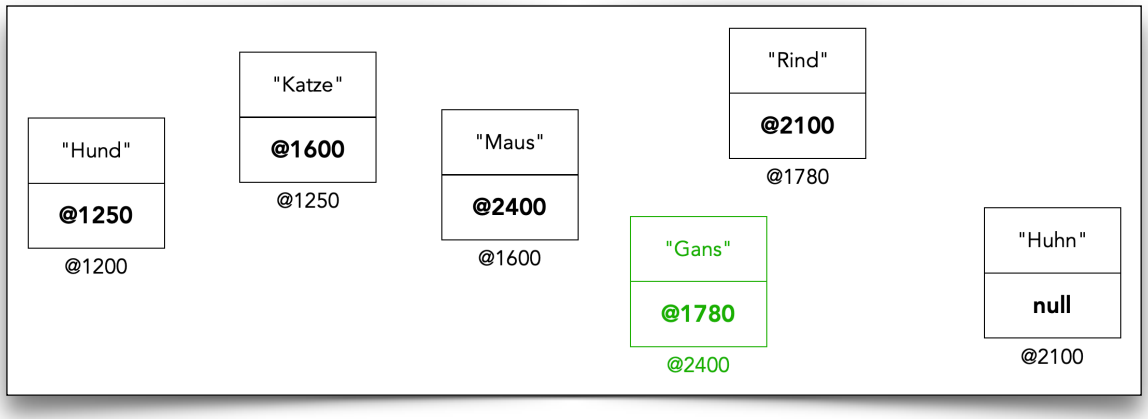
Bei einer ***einfach verketteten Liste*** besitzt jeder Knoten genau *eine* Referenz auf einen weiteren Knoten, der dann als sein ***Nachfolger*** bezeichnet wird.

Der Knoten "Hund" an Adresse @1200 besitzt eine Referenz, die auf den Nachfolger-Knoten an Adresse @1250 verweist. Dieser Knoten - "Katze" - verweist auf einen Nachfolger-Knoten an Adresse @1600.

Der Nachfolgerknoten von "Katze" kann sich also an irgendeiner Adresse im Speicher befinden. Die Adresse des Nachfolgerknotens kann nicht aus der Adresse des Vorgängerknotens berechnet werden wie bei einem Array oder einer ArrayList. Stattdessen muss der Vorgängerknoten die Adresse des Nachfolgerknotens selbst verwalten.

Einfügen in eine LinkedList

Soll ein neuer Knoten in eine solche einfach verkettete Liste eingefügt werden, so müssen lediglich die beteiligten Referenzen angepasst werden:



Das neue Element "Gans" soll zwischen "Maus" und "Rind" eingefügt werden. Dazu sind nur zwei Schritte notwendig, bei denen Referenzen "umgebogen" werden müssen:

1. Die Referenz von "Maus" muss auf das neue Element "Gans" gesetzt werden
2. Die Referenz von "Gans" wird auf "Rind" gesetzt.

Vorteile einer LinkedList

Das Einfügen und Löschen von Elementen ist sehr viel einfacher als bei einer ArrayList, es müssen lediglich ein paar Referenzen geändert werden. Ein Aufrücken vieler Elemente nach links oder rechts ist nicht erforderlich. Allerdings muss dazu die Position des einzufügenden oder zu löschenden Elements bekannt sein. Das Suchen einer solchen Position kann nämlich selbst wieder viel Zeit kosten.

Nachteile einer LinkedList

Wo Vorteile sind, sind immer auch Nachteile. Der wichtigste Nachteil besteht darin, dass **kein Direktzugriff** auf ein beliebiges Element möglich ist. Wenn man zum Beispiel das Element an Position 5000 benötigt, muss man die Liste Knoten für Knoten durchlaufen, bis diese Stelle erreicht ist - das sind dann 4999 Operationen. Der Zugriff auf ein bestimmtes Element dauert daher deutlich länger als bei einer ArrayList.

Fazit

- **ArrayList:** schnell beim Zugriff auf Elemente über deren Indizes; langsam beim Einfügen und Löschen von Elementen, aber auch beim Suchen von Elementen.
- **LinkedList:** schnell beim Einfügen und Löschen von Elementen, falls deren Position bereits bekannt ist; langsam beim Suchen von Elementen.

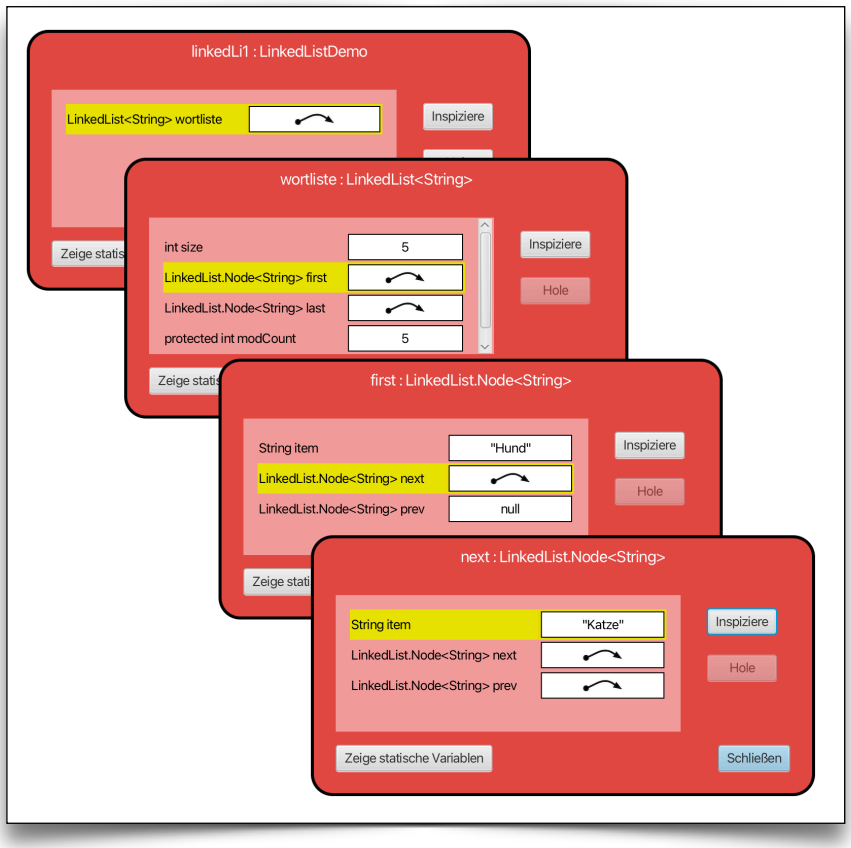
7.7.3 Untersuchung der Implementation

Wir erstellen eine kleine Java-Klasse **LinkedListDemo**:

```
import java.util.LinkedList;

public class LinkedListDemo
{
    LinkedList<String> wortliste;

    public LinkedListDemo()
    {
        wortliste = new LinkedList<>();
    }

    public void einfuegenDemo()
    {
        wortliste.add("Hund");
        wortliste.add("Katze");
        wortliste.add("Maus");
        wortliste.add("Rind");
        wortliste.add("Huhn");
    }
}
```

Dieses Programm erstellt ein Objekt `wortliste` der Klasse **LinkedList** und fügt dann mit der `add()`-Methode fünf Strings in die Liste ein.

Wenn wir in BlueJ ein Objekt der Klasse **LinkedListDemo** erzeugen und dieses Objekt mit dem Objektinspektor untersuchen, sehen wir Folgendes:



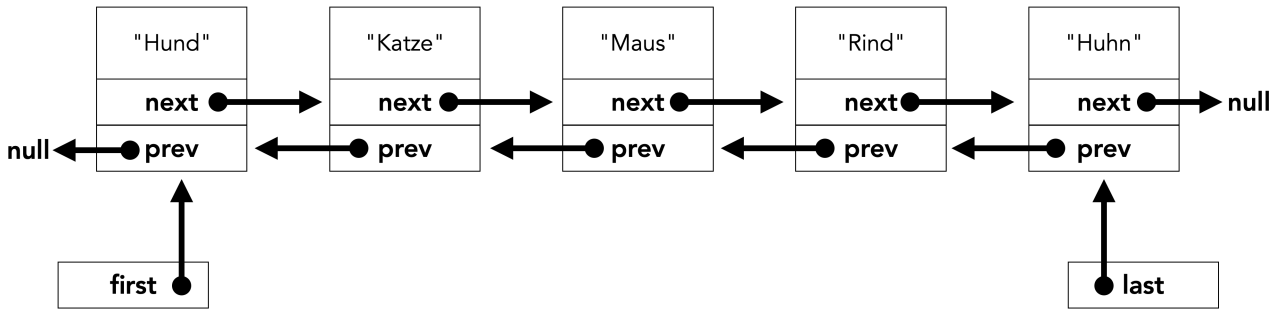
Das Objekt `wortliste` besitzt mehrere Instanzvariablen, von denen drei besonders wichtig sind: `size`, `first` und `last`.

Dabei verwaltet `size` die Anzahl der in der Liste vorhandenen Elemente, `first` ist eine Referenz auf das erste Listenelement, und `last` verweist auf das letzte Element der Liste.

Wenn man nun auf das Zeigersymbol klickt, das first repräsentiert, zeigt der Objektinspektor das erste Listenelement an. Hier sehen wir drei Instanzvariablen, nämlich `item`, `next` und `prev`.

Hier ist `item` eine Referenz auf ein Objekt (hier: ein String-Objekt), `next` ist eine Referenz auf das nächste Listenelement und `prev` eine Referenz auf das vorherige Listenelement.

Man kann also gut erkennen, dass es sich bei dieser Liste um eine ***doppelt verkettete Liste*** handelt: Jedes Element hat einen ***Vorgänger*** und einen ***Nachfolger***. Die Variable `prev` des ersten Elementes ("Hund") hat den Wert `null`, da das erste Element keinen Vorgänger hat. Entsprechend hat die Variable `next` des letzten Elementes ("Huhn") ebenfalls den Wert `null`, denn das letzte Listenelement hat keinen Nachfolger.



Dieses Bild zeigt die innere Struktur der Beispiel-Liste noch einmal in Form eines sogenannten Kästchenschemas. Im Informatik-Unterricht der gymnasialen Oberstufe sind Ihnen solche Kästchenschemata sicherlich schon begegnet. In NRW gehört die Kenntnis solcher doppelt verketteten dynamischen Listen sogar zum Pflichtstoff für das Abitur.

7.7.4 Wichtige Methoden der Klasse LinkedList

```
addFirst(E element)           // Vorne einfügen
addLast(E element)            // Ans Ende anhängen
removeFirst()                 // Erstes Element entfernen
removeLast()                  // Letztes Element entfernen
E getFirst()                  // Wert des ersten Elements liefern
E getLast()                   // Wert des letzten Elements liefern
add(int index, E element)     // Neues Element an Position index einfügen
set(int index, E element)     // Element an Position index überschreiben
remove(int index)             // Element an Position index entfernen
remove(Object o)              // Element o entfernen.
E get(int index)              // Wert des Elements an der Position index liefern
clear()                       // Liste löschen
contains(Object o)            // liefert true, wenn o in Liste enthalten ist
indexOf(Object o)             // liefert den Index des ersten Vorkommens von o
lastIndexOf(Object o)         // liefert den Index des letzten Vorkommens von o
isEmpty()                     // liefert true, wenn die Liste leer ist
size()                        // liefert die Zahl der Listenelemente
```

7.7.5 Sinnvoller Einsatz von LinkedList

Für die meisten Anwendungen verwendet man eine ArrayList, vor allem dann, wenn auf Listenelemente häufig über den Index zugegriffen wird.

Eine LinkedList lohnt sich vor allem dann, wenn nicht das schnelle Finden per Index im Vordergrund steht, sondern das häufige Einfügen und Löschen von Elementen.

7.7.6 Eine eigene dynamische Liste

In diesem Abschnitt betrachten wir die Entwicklung einer einfach verketteten Liste. Die Liste soll Objekte der Klasse **Person** speichern. Eine Klasse wie **Person** mit Name, Vorname und Alter zu erstellen, ist trivial und soll hier nicht weiter ausgeführt werden.

Die Klasse Knoten

Eine echte LinkedList besteht aus Knoten, die ihrerseits aus drei Komponenten zusammengesetzt sind:

1. Der eigentliche Inhalt oder Wert des Knotens, in unserem Beispiel also ein Objekt der Klasse **Person**
2. Einem Verweis auf den Nachfolgerknoten (oder `null`)
3. Einem Verweis auf den Vorgängerknoten (oder `null`)

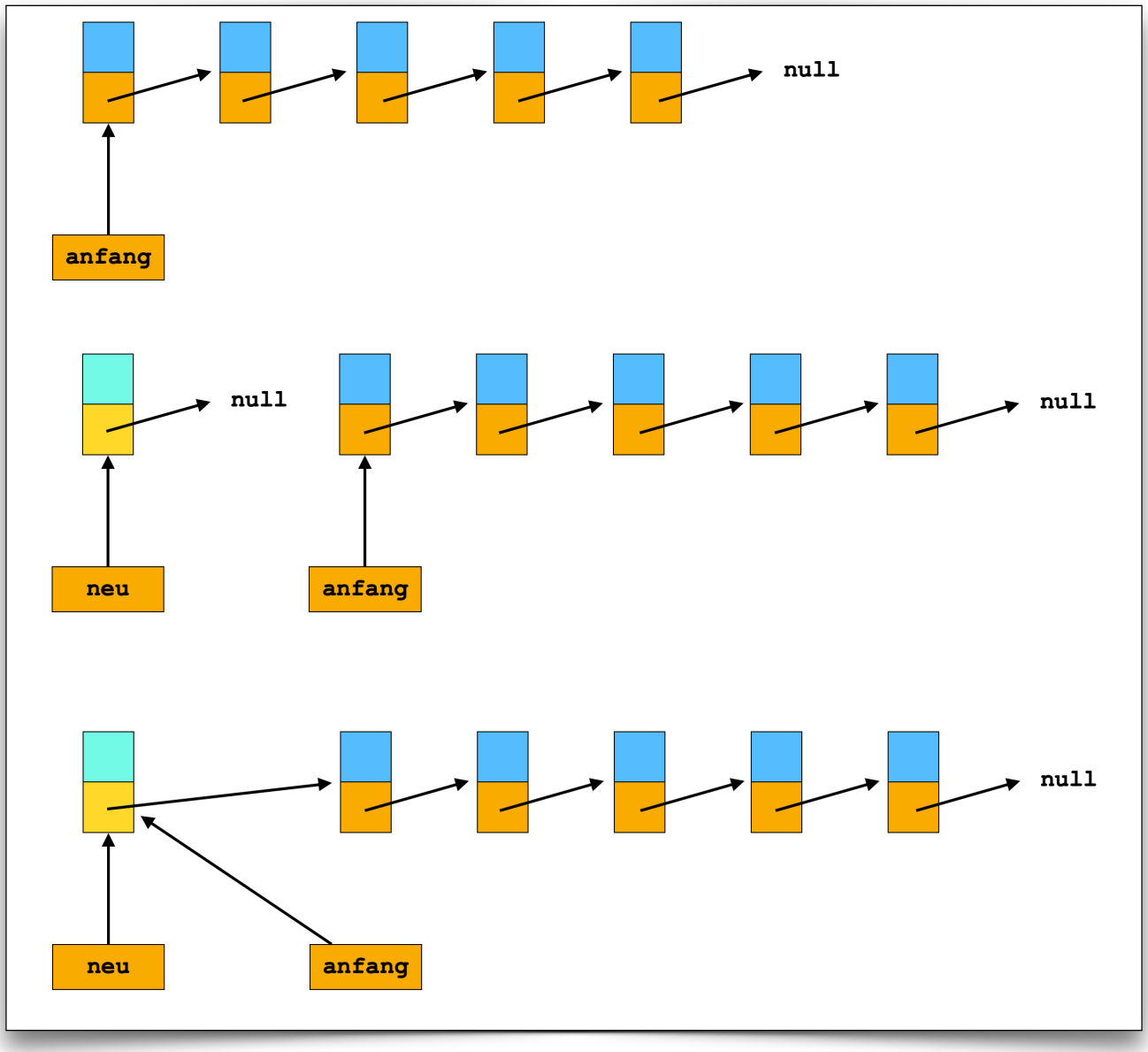
Listen der Klasse **LinkedList** sind also ***doppelt verkettet***, jeder Knoten verweist auf einen ***Vorgänger*** und einen ***Nachfolger***.

Unsere Modell-Liste, die wir hier implementieren wollen, ist dagegen nur ***einfach verkettet***. Jeder Knoten verweist auf seinen Nachfolger, nicht jedoch auf seinen Vorgänger.

```
public class Knoten
{
    public Person wert;
    public Knoten nachfolger;
}
```

Die beiden Instanzvariablen wurden als `public` deklariert, so dass man ohne Setter- und Getter-Methoden jederzeit auf sie zugreifen kann. Das entspricht zwar nicht ganz exakt den Prinzipien der OOP, vereinfacht aber die Implementation der Liste selbst, wie wir gleich noch sehen werden.

Kästchen-Darstellungen für dynamische Datenstrukturen



Diese Zeichnung stellt das Einfügen eines neuen Knotens an den Anfang der Liste graphisch mit einem sogenannten ***Kästchen-Schema*** dar. In der Klasse **LinkedList** ist die Methode `addFirst()` für diese Art des Einfügens zuständig.

Arbeitsweise von addFirst()

In der oberen Reihe sehen wir eine Liste mit fünf Knoten. Die Instanzvariable `anfang` des Listen-Objekts verweist auf das erste Listenelement. Das erste Listenelement zeigt dann auf das zweite, das zweite auf das dritte und so weiter. Das letzte Listenelement hat keinen Nachfolger mehr, seine Referenz `nachfolger` hat dann den Wert `null`.

In der zweiten Reihe wurde ein neuer Knoten erzeugt, eine Hilfsvariable `neu` verweist auf diesen Knoten, und der neue Knoten hat noch keinen Nachfolger.

In der dritten Reihe der Zeichnung ist der neue Knoten in die Liste eingefügt worden, und zwar an den Anfang der Liste. Dazu wird zunächst der Nachfolger-Zeiger des neuen Knotens auf das erste Element der Liste gesetzt:

```
neu.nachfolger = anfang;
```

Und dann wird die Instanzvariable `anfang` auf den neuen Knoten gesetzt:

```
anfang = neu;
```

Die entsprechende Java-Methode für das Einfügen an den Anfang der Liste sieht dann so aus:

```
public void addFirst(Person neuePerson)
{
   Knoten neuerKnoten = new Knoten();
   neuerKnoten.wert = neuePerson;
   neuerKnoten.nachfolger = anfang;
   anfang = neuerKnoten;
}
```

Da die Instanzvariablen `wert` und `nachfolger` der **Knoten**-Objekte `public` sind, kann hier direkt darauf zugegriffen werden, ohne Setter- oder Getter-Methoden.

Aufgabe

Ergänzen Sie die Klasse **MyLinkedList** um die Methoden

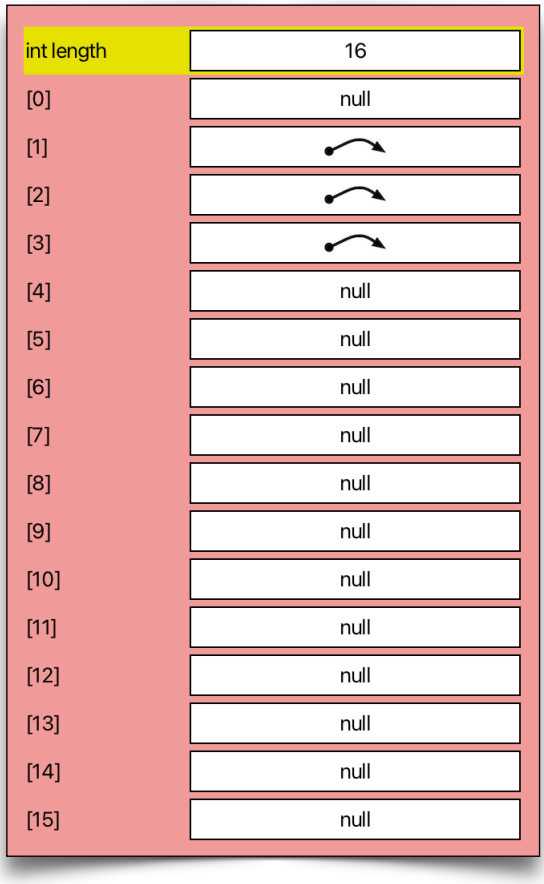
```
public void addLast(Person neuePerson)
public void add(int index, Person neuePerson)
public void show()
```

Erläutern Sie die Arbeitsweise der drei Methoden mithilfe von drei Kästchen-Schemata!

7.8 Die Klasse HashMap

7.8.1 Erforschung eines HashMap-Objekts

Die Klasse **HashMap** ist eine weitere wichtige Sammlungs-Klasse in Java. Wir wollen diese Klasse mit BlueJ und dem Objektinspektor einmal näher untersuchen. Für diesen Kurs "Einführung in die OOP" reicht jedoch eine eher oberflächliche Betrachtung aus; das Thema wird im Kurs "Algorithmen und Datenstrukturen" später vertieft.



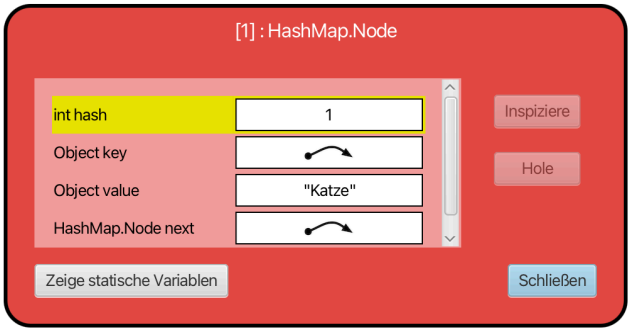

```
import java.util.HashMap;

public class HashMapDurchleuchtet
{
   private HashMap<Integer, String> map;

   public HashMapDurchleuchtet()
   {
      map = new HashMap<>();

      map.put(1,  "Katze");
      map.put(2,  "Tiger");
      map.put(3,  "Löwe");
      map.put(17, "Hund");
      map.put(18, "Wolf");
      map.put(33, "Maus");
   }
}
```

Dieses kleine Programm legt ein Objekt `map` der Klasse **HashMap** an und fügt mit der Methode `put()` sechs Einträge in die Datenstruktur ein. Jeder Eintrag besteht aus einem Schlüssel (hier eine Zahl) und einem Wert (hier ein Tiername).

Wenn wir die HashMap im Objektinspektor betrachten, erkennen wir ein Array mit 16 Elementen, genauer gesagt: ein Array aus 16 Referenzen. Die meisten dieser Referenzen haben den Wert `null`, nur wenige zeigen auf tatsächlich vorhandene Einträge.

Eines fällt hier auf: Obwohl wir *sechs* Einträge eingefügt haben, sehen wir im Array nur *drei* belegte Positionen. Wie kann das sein?

Wir schauen uns den ersten Verweis näher an, indem wir auf den Pfeil an Index 1 doppelklicken. Das Ergebnis ist rechts zu sehen.



Das Objekt, auf das hier referenziert wird, besitzt vier Instanzvariablen. Die erste - `hash` - ist der sogenannte ***Hash-Wert*** des Objekts.

Hash-Wert

Jeder Schlüssel wird intern in einen Hash-Wert umgewandelt. Ein Hash-Wert ist eine ganze Zahl, die durch die Methode `hashCode()` berechnet wird.

Bei Integer-Objekten entspricht dieser Hash-Wert in der Regel dem gespeicherten Zahlenwert. Bei komplexeren Objekten (z. B. Strings) wird der Hash-Wert durch einen speziellen Algorithmus berechnet.

Aus dem Hash-Wert wird anschließend der Index im Array bestimmt (z. B. durch eine Modulo-Operation).

Key / Value

Jeder Eintrag in einer HashMap besteht aus zwei Komponenten:

1. einem Schlüssel (`key`)
2. einem Wert (`value`)

Bei einem Deutsch-Englisch-Lexikon wäre beispielsweise der deutsche Begriff der Schlüssel (`key`), die englische Übersetzung dann der Wert (`value`).

Next

Hier kommen wir der Frage näher, die wir uns eben gestellt haben: Wieso sind nur drei Arrayelemente belegt, obwohl wir doch sechs Objekte in der Liste untergebracht haben?

Es kann vorkommen, dass verschiedene Schlüssel auf denselben Array-Index abgebildet werden. Man spricht dann von einer ***Kollision***.

Beispielsweise liefern die Schlüssel 1, 17 und 33 den gleichen Array-Index, nämlich 1. Daher befinden sich an dieser Position drei Elemente, nämlich `"Katze"`, `"Hund"` und `"Maus"`. Mit dem BlueJ-Objektinspektor kann man dies leicht überprüfen, indem man einfach auf den `next`-Pfeil klickt. Der Nachfolger von `"Katze"` ist dann `"Hund"`, und der Nachfolger von `"Hund"` ist `"Maus"`. Die Instanzvariable `next` des letzten Elements hat dann den Wert `null`.

7.8.2 Struktur einer Hash-Map

Ein Array aus Listen

Im Grunde ist eine Hash-Map ein Array aus Referenzen auf Listen. Das zugrunde liegende Array einer HashMap hat eine Anfangskapazität von 16.

0				
1	1, Katze →	17, Hund →	33, Maus →	null
2	2, Tiger →	18, Wolf →	null	
3	3, Löwe →	null		
4				
5				
6				
7				
8				
9				
10				
11				
12				
13				
14				
15				

Das erste Element (1, "Katze") hat den Schlüssel (key) 1. Bei integer-Keys gilt grundsätzlich: Integer-Wert = Hash-Wert. Bei Nicht-integer-Keys ist die Berechnung des Hash-Wertes aufwändiger.

Aus dem Hash-Wert (hier also 1) wird der Array-Index berechnet. Bei einer Anfangskapazität von 16 erfolgt die Berechnung des Index mit der einfachen Formel Hash-Wert % 16 bzw. Key % 16. Die Keys 1, 17 und 33 für "Katze", "Hund" und "Maus" liefern also den gleichen Index 1. Daher wird zunächst "Katze" in die Liste mit dem Index 1 eingefügt, dann "Hund" und schließlich "Maus".

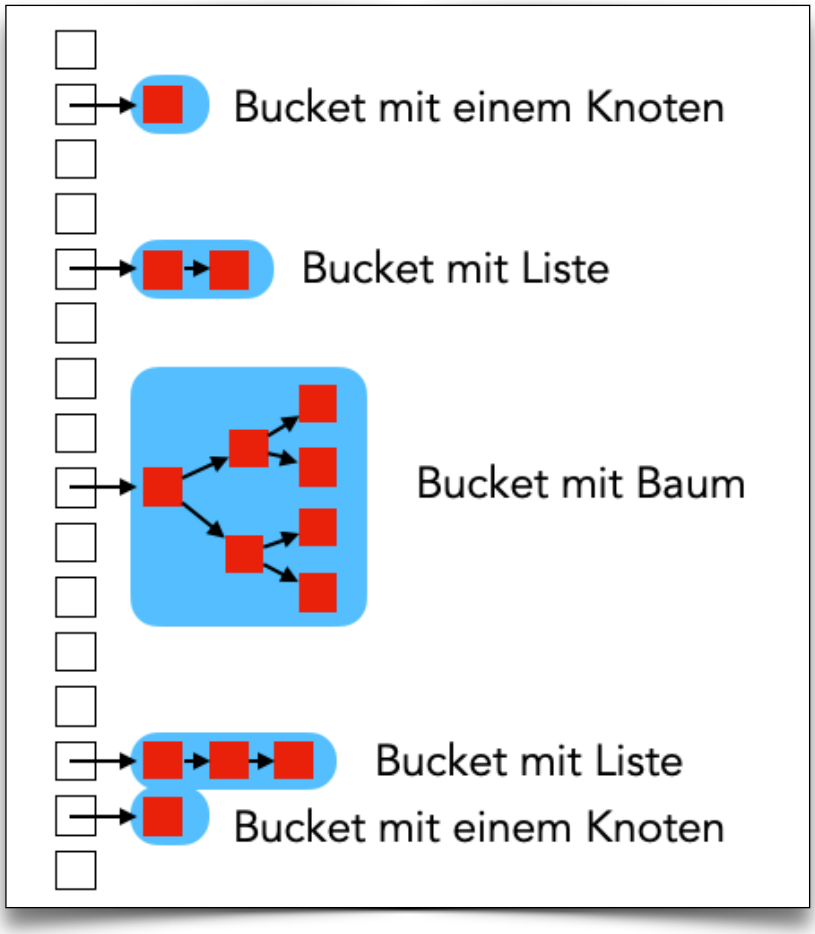
Das Element "Tiger" hat den Key 2, mit 2 % 16 erhält man den Index 2. Der Key 18 des Elementes "Wolf" liefert ebenfalls den Index 2. Das Element "Löwe" mit dem Key 3 wird an Index 3 eingeordnet.

Angenommen, wir hätten ein Element (28, "Schaf"). Wo würde dieses Element in die HashMap eingeordnet?

28 % 16 = 12, also würde das Element an Index 12 eingefügt.

Aus den Listen können Bäume werden

In der Regel kann eine Liste des HashMap-Arrays auf acht Elemente anwachsen. Wenn noch mehr Elemente hinzugefügt werden sollen, wird die lineare Liste in einen binären Suchbaum umgewandelt, und zwar in einen speziellen ausgeglichenen Binärbaum. Dieses Phänomen können wir in einer abschließenden Graphik veranschaulichen:



Die Arrayelement des HashMap-Rückgrats werden übrigens als Buckets bezeichnet.

Ganz korrekt ist die Zeichnung noch nicht. Sehen Sie den Fehler?

7.8.3 Einsatz von Hash-Maps

Eine HashMap verwendet man immer dann, wenn man zu einem Schlüssel sehr schnell einen passenden Wert finden möchte. In der HashMap werden also immer Paare gespeichert: Schlüssel → Wert.

Beispiele

Matrikelnummer → Name, Vorname, Wohnort

Benutzername → Passwort

Artikelnummer → Preis, Standort

Auf das Element einer HashMap wird also nicht über einen Index zugegriffen wie bei einem Array oder einer ArrayList, sondern über einen eindeutigen Schlüssel.

Die Reihenfolge der Einträge ist bei einer HashMap nicht wichtig. Wenn man die Elemente in einer festen Reihenfolge speichern oder über Indizes wie 0, 1, 2 etc. ansprechen möchte, ist eine HashMap nicht die geeignete Datenstruktur. Hierfür eignen sich Arrays oder lineare Listen wie ArrayList am besten.