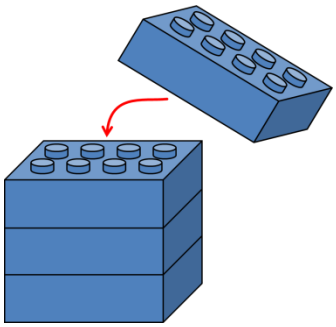
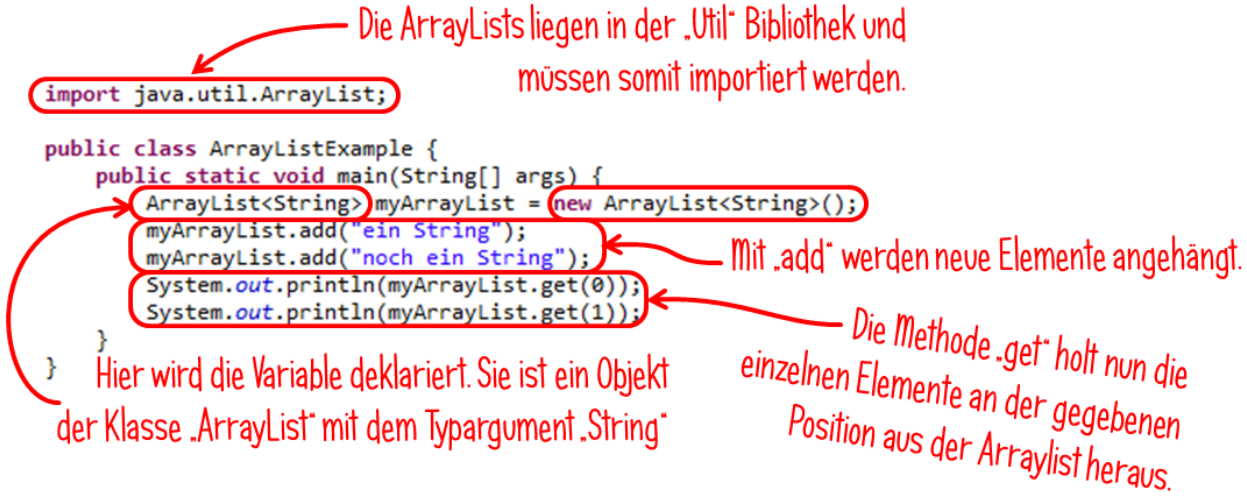
1 ArrayList – wenn man hinterher erst schlauer ist



Sehr häufig kommt es vor, dass man dynamische Listen erstellen muss – also die Größe der Liste sich während der Programmausführung ständig ändern kann. Hier stoßen die Arrays (zumindest in Java) an ihre Grenzen. Skriptorientierte Sprachen wie bspw. Javascript haben damit kein Problem, Java Programmierer müssen sich hier etwas anderes ausdenken. Das Zauberwort hier heißt „Array-List“. Dieses Konstrukt erlaubt es uns, zu jeder Zeit Einträge hinzuzufügen und welche zu entfernen.

Da eine ArrayList eben eine ArrayList und kein Array ist, sehen die Zugriffsformen entsprechend anders aus als bei einem Array. Hier müssen wir ein paar neue Methoden lernen, was aber recht intuitiv ist.

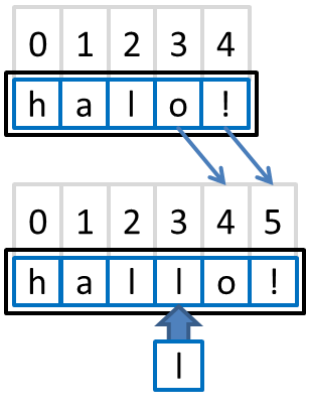
Folgender Code soll die Nutzung verdeutlichen:



Die ArraList Klasse versteckt sich in der Bibliothek „util“ und muss somit importiert werden. Danach erzeugen wir ein ArrayList Objekt. Hier kommt ein neues Syntaxelement auf uns zu – die Typisierung der „raw“ Klasse. Die ArrayList ist eigentlich so aufgebaut, dass sie jegliche Objekte aufnehmen kann und somit vom inneren Aufbau her alle Daten vom Typ „Object“ verarbeitet. Das ist aber vom Handling her recht umständlich, da die „get“ Methode demnach auch immer nur Elemente vom Typ „Object“ zurückgibt und wir für die Weiternutzung einen Typecast durchführen müssten. Dieser Typecast wird uns abgenommen, wenn wir bei der ArrayList in spitzen Klammern den eigentlichen Datentyp eintragen – in unserem Beispiel eben <String>.
Wer nun schon etwas in die objektorientierte Programmierung hineingeschnuppert hat weiß nun aber, dass ArrayList Objekte somit keine einfachen Datentypen aufnehmen können, sondern nur Objekte. Wer aber trotzdem bspw. ganze Zahlen in einer ArrayList ablegen möchte, muss mit den Wrapperklassen arbeiten:

```
ArrayList<Integer> myArrayList = new ArrayList<Integer>();
```

Nun können wir mit der Methode „**`add`“ die einzelnen Elemente einfügen. Der „**`get`“ Befehl liest sie wieder aus, wobei der Parameter die Position in der ArrayList festlegt.



Wenn wir nun an einer bestimmten Position etwas einfügen, so müssen wir eine andere „add“ Methode verwenden und zwar diejenige mit einem Indexparameter (bspw. „**`add(3,"h")`**“). Hier wird dann ein Element an die entsprechende Position gesetzt. Aber Achtung – alle nachfolgenden Elemente verschieben sich dann um eine Position. Gelöscht werden Elemente mit der „**`remove()`**“ Methode, welche ebenfalls die Indexposition als Parameter verlangt. Hier werden sich die nachfolgenden _Elemente entsprechend wieder nach links verschieben.

Wenn wir lediglich ein Element austauschen wollen, so müssen wir die Methode „**`set`**“ verwenden. Folgende Methode würde also aus einer ArrayList den ersten Eintrag durch das „H“ ersetzten:

```
myStringArrayList.set(0, "H");
```

~~Blöde~~ Frage:
Ich habe mal etwas von einem „Vector“ gehört. Ist das nicht das gleiche, wie eine ArrayList?
Antwort: Die wesentlichen Eigenschaften sind identisch. Der eigentliche Unterschied ist, dass der Vector „synchronized“ ist, was bedeutet, dass verschiedene Threads sicher auf das Objekt zugreifen können, ohne sich gegenseitig zu stören und somit Dateninkonsistenzen erzeugen. Für uns ist das jedoch (noch) nicht von Relevanz und demnach empfehle ich (und auch Oracle…) die Nutzung von ArrayLists.

Wir halten also fest:

- Eine ArrayList kann dynamisch erweitert oder verkürzt werden.
- ArrayLists sollten bei der Deklaration den Typ der aufzunehmenden Klasse angeben. Dies geschieht in spitzen Klammern: ArrayList<String>.
- Die Methode „add“ fügt ein Element in die ArrayList ein. Wird kein int-Parameter angegeben, so geschieht es am Ende. Existiert ein int-Parameter, so sorgt er dafür, dass das Element an der angegebenen Position eingefügt wird.
- Mit „remove“ wird das angegebene Element gelöscht.
- Mit „set“ kann ein Element ausgetauscht werden.
- Mit „get“ wird das angegebene Element ausgelesen. Wenn die Klasse angegeben wurde, so hat das Element den angegebenen Datentyp.

Zusatzinfo:
Intern nutzt die ArrayList ein Array – was für uns allerdings verborgen bleibt. Lediglich die „capacity“ gibt uns Rückschlüsse auf dieses Array. Es gibt bspw. einen Konstruktor, welcher die initiale „capacity“ vorgibt – also die Größe des intern genutzten Arrays. Wenn man vorher schon ungefähr weiß, wie groß die Datenmenge sein wird, kann das Performancevorteile bringen. Trotzdem kümmert sich Java automatisch um die Größenanpassung dieses Arrays, sollte der Platz nicht mehr ausreichend sein.

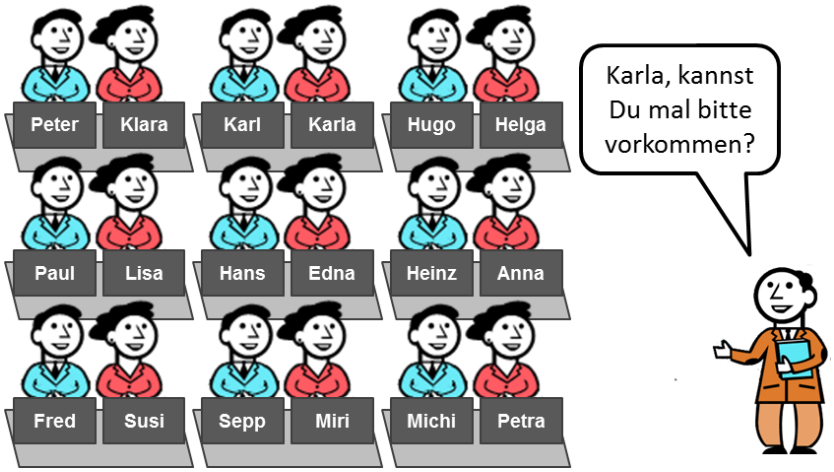
2 Assoziative Arrays – wer ist schon gerne nur eine Nummer?

Bis dato haben wir in Arrays stets die Indexposition benötigt, um an die einzelnen Elemente zu kommen. Dies ist durchaus praktisch, da wir hiermit eine eindeutige Adressierung jedes einzelnen Eintrages haben und somit unsere Elemente sicher identifizieren können. Es gibt aber Situationen, bei denen wir nicht über eine Indexposition zugreifen möchten, sondern bspw. über einen Namen oder ähnlichen Kennzeichnungen. Hier bieten sich die sogenannten „assoziativen Arrays“ an. Java bietet gleich eine ganze Fülle solcher Arrays an:

- HashMap
- Hashtable (wie Hashmap, nur „synchronized“)
- LinkedHashMap (merkt sich die Einfügereihenfolge)
- TreeMap (bildet die Keys in einem schnell durchsuchbaren Baum ab)

In diesem Kapitel werden wir uns „nur“ mit der HashMap und HashTable beschäftigen, wobei das wesentliche Handling der anderen beiden Klassen sich nicht wirklich unterscheidet – der Unterschied liegt eher in der inneren Umsetzung.

Nun, wie können wir uns nun diese HashMaps vorstellen? Prinzipiell funktioniert das wie in einer Schulklasse. Die Schüler sitzen mehr oder weniger geordnet im Klassenzimmer. Wenn der Lehrer jemanden ansprechen möchte, dann sagt er nicht „Die Schülerin in der hintersten Reihe, dritte von rechts solle sich melden“, sondern er spricht sie mit dem Namen an.



Genau so läuft es mit den assoziativen Arrays. Jeder Eintrag bekommt einen Namen – eigentlich einen „Key“. Dieser muss somit eindeutig innerhalb des Arrays sein, um den Eintrag eindeutig identifizieren zu können. Man spricht hier auch von „Schlüssel/Wert Paaren).
Das Besondere ist nun, dass wir jede Klasse, welche die Methoden „hashCode“ und „equals“ aufweist, als Key verwenden können. „hashCode“ liefert einen Code, welcher als Schlüssel zum Auffinden des Elements verwendet wird und „equals“ kennen wir ja bereits aus der String Methode – hier wird auf inhaltliche Gleichheit geprüft. Wir sehen hier wieder den Syntax mit den spitzen Klammern, welchen wir bei ArrayLists bereits gesehen haben. Auch hier gilt, wir können in die HashMap nur Objekte eintragen, keine einfachen Datentypen.

Auch die HashMap finden wir unter „utils“.

```
import java.util.HashMap;

public class HashMapExample {
    public static void main(String[] args) {
        HashMap<String, Integer> myHashMap = new HashMap<String, Integer>();
        myHashMap.put("Peter", 20);
        myHashMap.put("Frieda", 21);
        myHashMap.put("Karla", 19);
        String myKey = "Frieda";
        System.out.println(myKey + " ist " + myHashMap.get(myKey) + " Jahre alt.");
    }
}
```

Wir erzeugen eine Hashmap mit „String“ als Key und „Integer“ als Werte.

„put“ platziert die einzelnen Elemente samt Key.

Nun können wir mit „get“ über den Key an die Elemente kommen.

Der oben stehende Code erzeugt eine HashMap. In diesem Beispiel habe ich den Key als String vorgegeben und die Elemente (also die Werte, welche ich später suchen möchte) als „Integer“ – wir wollen somit zu jedem Namen eine Zahl, bspw. das Alter, einfügen. Wenn wir später objektorientiert arbeiten, können wir als Wertelement beliebige Klassen vorsehen. Die Methode „put“ fügt ein Schlüssel/Wert Paar ein. Wo in der HashMap dies geschieht ist für uns irrelevant. Wir haben ja den Key, der uns eindeutig zu unserem Wert führt.
Danach habe ich die Werte eingetragen, indem ich jedem Wert noch einen Schlüssel mitgegeben habe. Dieser muss eindeutig sein – taucht ein Schlüssel ein zweites mal auf, so wird der vorausgegangene Wert einfach überschrieben. Im Code habe ich nun eine Stringvariable „myKey“ deklariert und mit einem Schlüsselwert belegt – hier der Name „Frieda“. Nun fischen wir uns den Eintrag von Frieda mit folgendem Code aus unserer HashMap:

```
myHashMap.get(myKey);
```

Schon haben wir das Alter von Frieda herausgefunden.

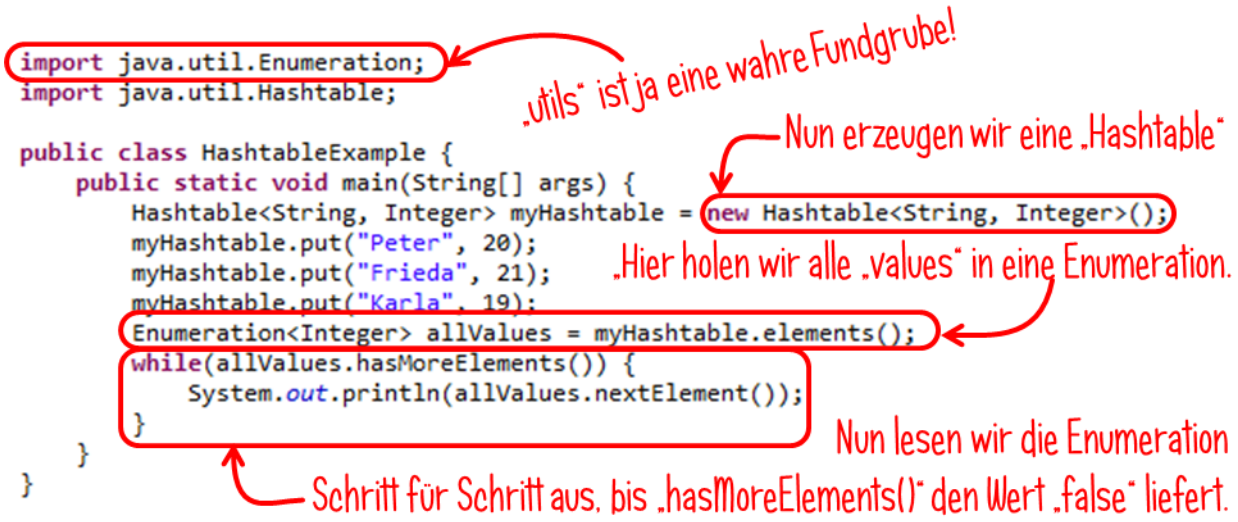
~~Blöde~~ Frage:
Was passiert eigentlich, wenn ich in dem oberen Beispiel einen Schlüssel suche, der gar nicht existiert, wie bspw. „Paul“?
Antwort: Die HashMap würde den Wert null liefern. Da wir ja eine Integerklasse als Wert (und nicht der einfache Datentyp „int“) vorgesehen haben, kann die HashMap auch null zurückliefern.

Nun wollen wir uns noch mal ein paar nützliche Methoden der HashMap ansehen, welche uns den Umgang mit den Hashmaps erleichtern werden. Beginnen wir mit den „contains“ Methoden. Diese zeigen uns an, ob ein Key oder ein Value existiert:

```
myHashMap.containsKey("Frieda");
myHashMap.containsValue(21);
```

Beide Methoden liefern einen boolean Wert zurück. „containsKey“ liefert „true“, wenn das angegebene Argument (in unserem Beispiel „Frieda“) als Schlüssel in der HashMap existiert. „containsValue“ liefert uns „true“ zurück, wenn der angegebene Wert (in unserem Beispiel „21) mindestens einmal als Wert existiert.

Was ebenfalls sehr von Nutzen sein kann, ist die Erzeugung einer „Enumeration“. Diese Funktion wird allerdings nicht von HashMaps unterstützt – hier muss man über Collections gehen, was aber umständlicher ist. Eine Enumeration ist ein Objekt, welches mir ermöglicht sämtliche Elemente einer **Hashtable** sequenziell abzuarbeiten. Sehen wir uns hierfür folgenden Code an, welcher im Wesentlichen dem oberen gleicht, es wird lediglich mit einer **Hashtable** gearbeitet und über eine Enumeration ausgelesen.



Der obere Teil ist also Identisch mit dem Code der HashMap, nur dass wir eine Hashtable nutzen. Nachdem wir unsere Hashtable mit Werten belegt haben, erzeugen wir ein Enumeration Objekt, welches im Prinzip eine Aneinanderreihung von Elementen ist, auf die ich sequenziell zugreifen kann. Zwei wesentliche Methoden sind zu beachten:
nextElement() liefert das aktuelle Element und schiebt den Zeiger auf das nächste Element weiter.
hasMoreElements() ist solange auf true, wie noch nicht alle Elemente ausgelesen wurden.

Wer alle Schlüssel aus einer Hashtable auslesen möchte, der kann auch diese in eine Enumeration laden:

```
myHashtable.keys()
```

Entfernt werden die Elemente wiederum über den Key:

```
myHashtable.remove(key);
```

Wir halten also fest:

- Assoziative Arrays bilden Schlüssel/Wertpaare ab.
- Sowohl die Schlüssel, als auch die Werte müssen „Objekte“ sein. Aus den Schlüsseln müssen sich „Hashes“ erzeugen lassen können. Sie müssen innerhalb des Arrays eindeutig sein.
- Mit „put“ werden die Schlüssel/Wertpaare abgelegt.
- Mit „get(key)“ werden die Werte zum vorgegebenen Schlüssel ausgelesen.
- Mit „remove(key)“ wird das Element zum angegebenen Schlüssel gelöscht.
- Man kann mit „containsKey“ bzw. „containsValue“ feststellen, ob ein gegebener Schlüssel bzw. ein gegebener Wert in dem Array vorhanden ist.
- Bei Hashtables können alle Werte, bzw. alle Schlüssel sequenziell ausgelesen werden, indem man mit „.elements()“ bzw. „.keys()“ eine Enumeration erzeugt.
- Diese Enumerations liest man sequenziell mit der Methode „nextElement()“ aus, bis die Methode „hasMoreElements()“ den Wert „false“ liefert.

Zusatzinfo:
Auch die Hashtable bietet einen Konstruktor mit einer initialen Kapazität an. Darüber hinaus gibt es noch den Wert des „loadFactors“, was im Wesentlichen angibt, bei welchem Füllgrad sich Java um die Erhöhung der Kapazität kümmert. Wer spezielles Verhalten seiner Hashtable bzw. Hashmap haben möchte, der sollte sich die Doku zum loadFactor ansehen – für die meisten Nutzer reicht der normale Konstruktor ohne Parameter völlig aus.