Einführung in die Programmierung: Enums und Listen

Enums

```java
public class BuildingWithoutEnum {

    public static final BuildingWithoutEnum HG;
    public static final BuildingWithoutEnum CAB;
    public static final BuildingWithoutEnum CHN;
    public static final BuildingWithoutEnum LFW;

    static {
        HG = new BuildingWithoutEnum(90, "Hauptgebäude");
        CAB = new BuildingWithoutEnum(100, "CAB");
        CHN = new BuildingWithoutEnum(100, "CHN");
        LFW = new BuildingWithoutEnum(50, "LFW");
    }

    public int coolness;
    public String name;

    private BuildingWithoutEnum(int coolness, String name) {
        this.coolness = coolness;
        this.name = name;
    }
}
```

```java
public enum Building {

    HG(90, "Hauptgebäude"),
    CAB(100, "CAB"),
    CHN(100, "CHN"),
    LFW(50, "LFW");

    public int coolness;
    public String name;

    private Building(int coolness, String name) {
        this.coolness = coolness;
        this.name = name;
    }
}
```

Enums sind spezielle Klassen, mit denen ihr eine Liste von Konstanten repräsentieren könnt. Die Konstanten können zusätzlich eine beliebige Anzahl an Attributen speichern. Das ist unter anderem ein grosser Vorteil gegenüber anderen Konstanten (wie z.B. String Konstanten)

Da es sich bei Enums eigentlich um normale Java Klassen handelt, könnt ihr die Attribute, die die Konstanten haben sollen, über den Konstruktor des Enums angeben. Die Felder, die im Enum deklariert sind, werden nur benötigt, um die Attribute einer Konstante zu speichern und auf die Attribute einer Konstante zugreifen zu können.

Die Enumeration "Building" zeigt die grundlegende Struktur eines Enums: Konstanten und dazugehörige Attribute

Listen: ArrayList

In Java gibt es verschiedene Arten von Listen. Listen mit einer statischen (gleichbleibender) Grösse werden meistens von Arrays repräsentiert. Ein Nachteil von Arrays ist aber, dass die Interaktion mit Arrays mühsam ist. Wenn man ein neues Element hinzufügen will, muss man wissen, an welcher Stelle im Array das neue Element hinzugefügt werden soll. Um dieses Problem zu bewältigen, muss man irgendwie den aktuellen Zustand des Arrays speichern. Das ist aber meistens nicht so einfach. Zumal man die Korrektheit der Implementierung sicherstellen muss. Noch komplizierter wird es, wenn man Elemente aus dem Array löschen will. In diesem Fall, kann man alle Elemente, die nach dem gelöschten Element im Array vorkommen, nach links verschieben. Für diese Element wird also der Index kleiner.

Doch für diese Kalamität gibt es eine einfache Lösung in Java: ArrayList
Die ArrayList verwendet im Hintergrund einen Array zur Speicherung der Elemente. Falls man nun ein Element hinzufügen oder löschen will, kann man die eingebauten Methoden add() bzw. remove() verwenden. Diese Methoden implementieren ungefähr das Verhalten, das oben beschrieben wurde.
Es ist also möglich, mit einer ArrayList das gleiche Verhalten zu erzielen wie mit einem Array ohne grosse Performance Einbussen.

```java
Object[] staticList = new Object[100];
staticList[0] = new Object();
staticList[0] = null;
System.out.println(staticList[0]);
```

```java
ArrayList<Object> staticList = new ArrayList<Object>(100);
Object o = new Object();
staticList.add(o);
staticList.remove(o); // entferne Object o
staticList.remove(0); // entferne Element an Position 0
System.out.println(staticList.get(0));
```

In beiden Fällen wird ein Array mit der Kapazität 100 erstellt. add() fügt das Objekt immer an der nächsten freien Position im Array hinzu. Somit können wir folgern, dass die Einfüge-Reihefolge gleich ist wie die Speicher-Reihenfolge. Wenn also mit einer Enhanced For Loop über die Elemente in einer ArrayList iteriert wird, dann wird über die Elemente in der Einfüge-Reihenfolge iteriert.

Listen: ListIterator

```java
ArrayList<Object> list = new ArrayList<Object>(100);
ListIterator<Object> it = list.listIterator();
while (it.hasNext()) {
    Object o = it.next();
    o.myCoolField = "Hello World!";
    o.myCoolIntegerField = 10;
    if (deleteCondition) {
        it.remove();
    } else if (modifyCondition) {
        it.set(new Object());
    }
}
```

Wenn man Elemente in einer ArrayList verändern will, kann man das mit einem ListIterator machen. In so einem Fall ist es oft hilfreich, wenn man über alle Elemente in der Liste iteriert und gewisse Bedingungen überprüft, um zu entscheiden, ob ein Element entfernt oder ersetzt werden soll. Das Element kann aber problemlos auch ohne ListIterator bearbeitet werden. Wenn man also die Instanz einer Klasse verändern will, kann man dies auch ohne ListIterator machen. (bei der Enhanced For Loop wird im Hintergrund auch ein Iterator verwendet, aber kein ListIterator) Wenn man jedoch die Position des Elements in der ArrayList anpassen möchte, muss man unbedingt einen ListIterator verwenden. (es gäbe auch andere Möglichkeiten, die Probleme zu umgehen, die entstehen, wenn kein ListIterator verwendet wird. Z.B. Nutzung von CopyOnWriteArrayList. Diese Methoden sind aber nicht empfehlenswert)

Hinweise

Ich habe am Ende noch eine konkrete Implementierung einer Liste mit einer konstanten Grösse hinzugefügt. Diese Implementierung soll nochmals zeigen, wie man das beschriebene Verhalten von add() und remove() implementieren kann. Die Implementierung der Java ArrayList ist auch ähnlich. Die Grösser einer Java ArrayList ist aber nicht konstant.

https://docs.oracle.com/en/java/javase/21/docs/api/java.base/java/util/ArrayList.html
https://docs.oracle.com/en/java/javase/21/docs/api/java.base/java/util/ListIterator.html

```java
public class DynamicArrayWithFixedSize {

    private static final int DEFAULT_CAPACITY = 1000;

    private final Object[] data;

    private int cursor;

    public DynamicArrayWithFixedSize(int initialCapacity) {
        data = new Object[initialCapacity];
        cursor = 0;
    }

    public DynamicArrayWithFixedSize() {
        this(DEFAULT_CAPACITY);
    }

    public void add(Object objectToBeAdded) {
        if (cursor == data.length) {
            throw new IllegalStateException();
        }
        data[cursor] = objectToBeAdded;
        cursor++;
    }

    public Object[] toArray() {
        return Arrays.copyOf(data, cursor);
    }

    public Object get(int index) {
        if (index < 0 || index > data.length) {
            throw new IndexOutOfBoundsException();
        }
        return data[index];
    }

    public Object remove(int index) {
        if (index < 0 || index > data.length) {
            throw new IndexOutOfBoundsException();
        }
        if (index >= cursor) {
            return null;
        }
        Object value = data[index];
        data[index] = null; // remove the element at the specified index
        // if the element we want to remove is the last element in the array we don't have to do anything
        cursor--;
        if (index == cursor) {
            return value;
        }
        // if the element is not the last element in the array, we have to shift every element, that comes after the
        // element we removed, to the left. Since index != cursor and index < cursor+1, we know that index <= cursor-1
        // Thus cursor - index >= cursor - (cursor - 1) = 1. Therefore, at least one element will be shifted to the left.
        System.arraycopy(data, index+1, data, index, cursor-index);
        return value;
    }

    /**
     * This method returns the current size of the array which usually is not equal to data.length (size() only returns
     * array.length if the array has been completely filled with data)
     * @return the current size of the array
     */
    public int size() {
        return cursor;
    }
}
```