

# PR1, Aufgabenblatt 7

Programmieren I – Wintersemester 2021/22

## Arrays – Klassen als Objekte – Listen und Mengen implementieren

Ausgabedatum: ..... 17. Januar 2022

### Lernziele

Java-Programme ohne BlueJ mit Hilfe der `main`-Methode von der Kommandozeile aufrufen können, Kommandozeilenparameter übergeben können; einfache und mehrdimensionale Arrays verstehen und anwenden können, Schleifen über Arrays verwenden können; die Implementationsformen Array-basierte Liste und verkettete Liste umsetzen können; Das Prinzip von Hash-Verfahren verstehen, noch sicherer mit Arrays umgehen, weitere Anwendungen von Interfaces kennen.

### Kernbegriffe

In Java kann auch eine Klasse als ein Objekt angesehen werden, das zur Laufzeit einen Zustand haben und Operationen anbieten kann. Die Operationen eines Klassenobjektes (seine *Klassenoperationen*) werden mit dem Schlüsselwort `static` deklariert, ebenso wie mögliche Zustandsfelder für den Zustand des Klassenobjektes (seine *Klassenvariablen*). Die meisten Klassen in der Java-API sind jedoch zustandslos, so dass das Ergebnis einer Klassenoperation üblicherweise nur von ihren Parametern abhängt.

*Arrays* sind spezielle Sammlungen gleichartiger Elemente. Sie werden klassisch von imperativen Sprachen angeboten, meist unterstützt durch eine spezielle Syntax. Der Zugriff auf ein Element erfolgt, wie bei einer Liste, über einen Index. Da Arrays jedoch direkt auf den zugrunde liegenden Speicher abgebildet werden, kann mit den Mitteln der unterliegenden Rechnerarchitektur (mit Indexregistern o.ä.) ein sehr schneller wahlfreier Zugriff gewährleistet werden. Dafür sind Arrays jedoch in den meisten Sprachen statisch in ihrer Größe festgelegt, entweder bereits in ihrer Deklaration (wie in der Sprache Pascal) oder spätestens bei ihrer Erzeugung zur Laufzeit (wie in Java).

In Java können die Elemente eines Arrays sowohl Werte der Basistypen als auch Referenzen auf Objekte sein. Der Index ist in Java eine natürliche Zahl von 0 bis (Größe des Arrays)-1. Er wird in eckigen Klammern direkt hinter dem Bezeichner des Arrays verwendet (`a[0]` beispielsweise bezeichnet das erste Element des Arrays `a`).

Ein Array ist in Java ein Objekt, eine Array-Variable ist daher immer eine Referenzvariable. Der Typ dieser Variablen wird als *Array von <Elementtyp>* festgelegt. Arrays müssen, genauso wie Exemplare von Klassen, mit einer *new-Anweisung* erzeugt werden. Erst bei der Erzeugung eines konkreten Array-Objekts wird festgelegt, wie viele *Zellen* es hat; jede Zelle eines Arrays kann, wie eine Variable, ein Element des Elementtyps aufnehmen. Die *Größe/Länge* eines Array-Objekts (die Anzahl seiner Zellen) ist mit der Erzeugung festgelegt und kann nicht mehr verändert werden. Eine Array-Variable kann jedoch zu verschiedenen Zeitpunkten auf Arrays unterschiedlicher Größe verweisen.

Das Konzept einer *Liste* (also einer unbeschränkten Sammlung mit einer klientendefinierbaren Reihenfolge, die auch Duplikate zulässt) lässt sich auf vielfältige Weise implementieren. Im JCF gibt es zum Interface `List` zwei klassische imperative Implementierungen: `ArrayList` und `LinkedList`.

`ArrayList` basiert auf dem Konzept der „wachsenden“ Arrays: Für eine neue Liste wird ein Array der Größe  $k$  angelegt, so dass  $k$  Elemente direkt gespeichert werden können. Sobald die Liste mehr als  $k$  Elemente aufnehmen muss, wird ein neues, größeres Array (beispielsweise der Größe  $2 \cdot k$ ) angelegt, und alle Elemente aus dem alten Array werden in das neue Array kopiert. Es wird deshalb zwischen der *Kapazität* und der *Kardinalität* einer `ArrayList` unterschieden: Die Kapazität ist die aktuelle Größe des Arrays, die Kardinalität hingegen ist die momentane Anzahl an Elementen in der Liste (oft als *Länge* oder *Größe* der Liste bezeichnet). Es gilt immer:  $\text{Kapazität} \geq \text{Kardinalität}$ . Die Kardinalität ist dabei für Klienten über die Methode `size()` zugreifbar. Die Kapazität ist dagegen (genau wie das zugrundeliegende Array) ein Implementationsdetail.

`LinkedList` basiert auf der Verkettung von Kettengliedern/Knoten über Referenzen. Jeder Knoten hält eine Referenz auf das eigentlich zu speichernde Element sowie eine Referenz auf den nächsten Knoten und eine Referenz auf den vorherigen Knoten (*doppelt verkettete Liste*). Gäbe es keine Referenz auf den vorigen Knoten, wäre die verkettete Liste nur *einfach verkettet*. Bei verketteten Listen

werden häufig so genannte *Wächterknoten* am Listenanfang und/oder -ende verwendet, um Sonderfälle beim Einfügen oder Entfernen zu vermeiden.

Eine Menge unterscheidet sich von anderen Sammlungstypen primär dadurch, dass sie keine Duplikate zulässt. Im Umgang mit einer Menge ist die zentrale Operation die Nachfrage, ob ein gegebenes Element bereits enthalten ist (*istEnthalten*). Bei einer Liste ist diese Operation nicht effizient realisierbar, da das gesuchte Element an jeder Position stehen kann. Eine Listen-Implementation muss somit im Schnitt die Hälfte aller Elemente überprüfen, bis das gewünschte Element gefunden ist (vorausgesetzt, es ist in der Liste enthalten), der Aufwand zur Laufzeit ist also proportional zur Länge der Liste (formaler: die Komplexität von *istEnthalten* auf einer Liste ist  $O(n)$ ).

Um festzustellen, ob ein Element in einer Sammlung enthalten ist, muss es ein Konzept von Gleichheit geben. Für Java wird hierfür die Operation `equals` verwendet, die jeder Objekttyp anbietet. Wenn ein Objekttyp keine eigene Definition von Gleichheit implementiert, erfolgt automatisch eine Überprüfung auf Referenzgleichheit. Für Java gilt außerdem: Sind zwei Objekte laut `equals` gleich, dann müssen beide als Ergebnis der Operation `hashCode` den gleichen Wert liefern, d.h. wenn `a.equals(b)`, dann `a.hashCode() == b.hashCode()`.

Die effizientesten Implementationen von Operationen wie *istEnthalten* basieren auf so genannten *Hash-Verfahren*. Die Elemente werden dabei in einem Array von *Überlaufbehältern* gespeichert. Dieses Array bezeichnet man auch als *Hash-Tabelle*. Jedes Element kann nur in einem dieser Behälter vorkommen. Eine *Hash-Funktion* bildet ein Element auf einen ganzzahligen Wert ab. Dieser Wert wird geeignet auf einen Index in der Hash-Tabelle abgebildet, so dass beim Einfügen, Löschen und Aufsuchen eines Elements der richtige Behälter verwendet wird.

Eine gute Hash-Funktion sollte die Elemente möglichst gleichmäßig über die Hash-Tabelle „verschmieren“ – Im Idealfall enthält jeder Überlaufbehälter maximal ein Element. Bei einer solchen Auslastung ist der Aufwand für das Auffinden eines Elements nicht mehr von der Kardinalität der Menge abhängig, sondern setzt sich konstant aus der Indexberechnung und dem indexbasierten Zugriff auf einen Überlaufbehälter zusammen.

Interfaces können als Spezifikationen angesehen werden, die das Verhalten einer Implementation stark festlegen (siehe `Set` und `List`). Pragmatisch werden sie häufig nur zur syntaktischen Festlegung einer Schnittstelle benutzt, die eine Dienstleistung mit relativ großen Freiheitsgraden bieten kann.

## Aufgabe 7.1 Strings in der Kommandozeile analysieren (Termin 1)

In dieser Aufgabe wollen wir eine Java-Methode einmal nicht innerhalb von BlueJ aufrufen, sondern von der Kommandozeile des jeweiligen Betriebssystems. In Java ist für diesen Zweck eine spezielle Operation definiert worden: `public static void main(String[] args)`. Wenn eine Klasse eine Methode mit genau dieser Signatur definiert, dann kann diese Methode aus der Laufzeitumgebung der plattformabhängigen Java Virtual Machine aufgerufen werden.


- 7.1.1 Erstellt eine Klasse, die eine solche `main`-Methode enthält. Im Rumpf der Methode soll vorläufig lediglich eine beliebige Meldung mit `System.out.println` auf der Konsole ausgegeben werden. Ruft diese Methode in BlueJ auf.
- 7.1.2 Versucht nun, diese Methode von der Kommandozeile aus aufzurufen. Dazu müsst ihr zuerst ein Fenster öffnen, in dem ihr Kommandos eingeben könnt. (*Beispiel Windows: Start → Ausführen → cmd*) Wechselt in das Projekt-Verzeichnis von BlueJ, in dem eure Klasse (*Klassenname.class*) liegt. Dann könnt ihr folgende Zeile eingeben:  
`java <Klassenname>`
- 7.1.3 In der Signatur der Methode `main` seht ihr, dass diese Parameter vom Typ `String` in Form eines String-Arrays entgegennimmt. Findet heraus, wie man aktuelle Parameter in der Konsole bei einem Aufruf der Methode übergeben kann. Ändert nun eure `main`-Methode so ab, dass die übergebenen Strings nacheinander mit `System.out.println` ausgegeben werden. Testet diese Änderung, indem ihr von der Kommandozeile aus eure `main`-Methode mit Parametern aufruft.
- 7.1.4 Schreibt in eurer Klasse eine Methode `analysiereText`, die für einen übergebenen String erfasst, wie häufig die 26 Buchstaben des Alphabets (ohne Umlaute, nur Kleinbuchstaben) darin vorkommen. Die Methode soll dazu auch ein `int`-Array der Länge 26 erhalten, das es entsprechend verändert:  
`void analysiereText(String text, int[] haeufigkeit)`  
Wendet eure Methode auf jeden Parameter der `main`-Methode an. Anschließend soll in der

main-Methode das *Gesamtergebnis* für alle Parameter mit `System.out.println` ausgegeben werden. Testet erneut, entweder von der Kommandozeile aus oder in BlueJ. Tipp: Für diese Aufgabe müssen Buchstaben auf Array-Positionen abgebildet werden. Dabei soll die Position 0 für den Buchstaben 'a' stehen, die Position 25 für den Buchstaben 'z'. Die korrekte Array-Position für einen Buchstaben erhaltet ihr, indem ihr 'a' subtrahiert.

- 7.1.5 *Zusatzaufgabe*: Was passiert, wenn ihr aus einer Klassenmethode (z.B. der `public static void main(String[] args)`) auf eine Exemplarvariable zugreifen wollt? Gebt euren Betreuern eine Erklärung für das auftretende Verhalten.

## Aufgabe 7.2 Titel-Listen mit „wachsenden“ Arrays implementieren (Termin 1)

Programme wie iTunes oder Spotify ermöglichen, Musiktitel in Form von Wiedergabelisten zu organisieren. Solche Listen (wir nennen sie kurz Titel-Listen) enthalten eine beliebige Anzahl von Titeln, in einer vom Benutzer festgelegten Reihenfolge. Es sind auch echte Listen in dem Sinne, dass derselbe Titel mehrfach vorkommen kann. Wir wollen in PR 1 solche Titel-Listen auch erstellen können und haben deshalb das Interface `TitelListe` definiert. Auch wenn eine Titel-Liste normalerweise mit der Position 1 beginnt, soll unsere Liste (der Einfachheit halber) mit dem Index 0 beginnen.

-  7.2.1 Kopiert das Projekt `PR1Tunes` in euer Arbeitsverzeichnis und öffnet es. Das Projekt enthält ein Interface `TitelListe` und zwei unfertige Implementierungen dazu (`ArrayTitelListe` und `LinkedTitelListe`) sowie passende JUnit-Testklassen. Führt die JUnit-Tests nacheinander aus. **Haltet schriftlich fest**: Was machen die Testklassen? Wie viele Testmethoden definieren sie? Wie viele Testmethoden schlagen fehl? Wie findet man jeweils heraus, wo der Fehler aufgetreten ist?

-  7.2.2 **Erstellt eine Skizze** einer Array-Titelliste mit der Kapazität 10 und der Kardinalität 4.

- 7.2.3 Vervollständigt die Implementation der `ArrayTitelListe` in zwei Schritten:


**Im ersten Schritt** solltet ihr euch nicht darum kümmern, dass die Kapazität des Arrays bei Bedarf erhöht werden muss. Nun laufen, bis auf `testeFuegeEinVergroessern`, `testeLoeschenAusGrosserListe` und `testeVergroessern`, alle Tests durch. **Tipp**: damit der Balken im ersten Schritt grün wird, könnt ihr die drei Tests auskommentieren und erst im zweiten Schritt wieder hinzunehmen.

**Im zweiten Schritt** wird die `fuegeEin`-Methode so ergänzt werden, dass das Array wächst. Entfernt die Kommentarzeichen bei den drei auskommentierten Tests. Am Ende sollten alle Tests durchlaufen.

Es ist hilfreich, während der schrittweisen Implementierung an geeigneten Stellen die Methode `schreibeAufKonsole` von Hand aufzurufen, um den Zustand des Arrays und damit die Korrektheit eines Zwischenschritts zu überprüfen. Auch der Debugger ist hilfreich, um zu verstehen, was eine augenblickliche Implementation genau macht.

- 7.2.4 Implementiert einen weiteren Konstruktor, der die Angabe einer Anfangskapazität für die Listenimplementation erlaubt.

## Aufgabe 7.3 Titel-Listen mit verketteten Listen implementieren (Termin 1)

-  7.3.1 Die Klasse `LinkedTitelListe` soll das Interface `TitelListe` mit einer doppelt verketteten Liste implementieren. Intern enthält sie zwei Wächterknoten, einen für den Listenanfang, einen für das Ende. Die Wächterknoten markieren technisch die Grenzen der Liste und enthalten keine Titel. Die Wächterknoten garantieren, dass jeder echte Knoten einen Vorgänger und einen Nachfolger hat, was die Implementation enorm vereinfacht. Schaut euch die Klasse `DoppelLinkKnoten` und die unfertige Implementierung der Klasse `LinkedTitelListe` an. **Erstellt zwei Zeichnungen**, die zeigen, wie ein neuer Eintrag in diese Liste eingefügt wird. In den Zeichnungen soll deutlich werden, wie die Referenzen der Knoten verändert werden. Die Zeichnungen sollen sowohl die Knoten als auch die Titelobjekte zeigen. **Diskutiert jetzt mit einem/r Betreuer/in eure Lösungen, bevor ihr euch an die Implementierung macht.**

- 7.3.2 Vervollständigt die Implementation der `LinkedTitelListe`, bis alle Tests durchlaufen.

- 7.3.3 *Zusatzaufgabe*: Erstellt eine alternative Implementation, die keine Wächterknoten verwendet. Vergleicht die Implementationen und erklärt eurem Betreuer die Unterschiede.

## Aufgabe 7.4 Bildbearbeitung (Termin 2)

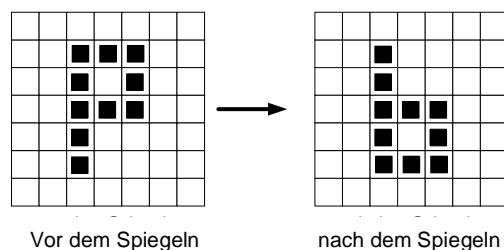
In dieser Aufgabe betrachten wir mehrdimensionale Arrays von elementaren Datentypen. Als Beispiel verwenden wir das Projekt *Bildbearbeitung*. Es enthält drei Klassen, von denen wir uns jedoch nur mit einer beschäftigen: `SWBild`. Die Klasse `BildEinleser` dient dazu, Bilder einzulesen, die Klasse `Leinwand` wird genutzt, um die Bilder anzuzeigen. Beide Klassen wollen wir nicht näher betrachten.

- 7.4.1 Öffnet das Projekt in BlueJ und erzeugt ein Exemplar von `SWBild`, eine Erläuterung findet sich in der Projektdokumentation. Wenn alles klappt, wird ein Bild in einem eigenen Fenster angezeigt.

Mit der Operation `dunkler(int delta)` könnt ihr dieses Bild abdunkeln. Probiert es aus. Schaut euch dann die Implementierung der Operation an. Wie wird das Array verwendet? Was befindet sich in dem Array? **Skizziert in einer Zeichnung** die implementierte Objektstruktur des Bilddaten-Arrays.

- 7.4.2 Implementiert die Operation `heller(int delta)` in der Klasse `SWBild`. Mit ihr soll das Bild um den Wert von `delta` aufgehellt werden.

- 7.4.3 Implementiert in `SWBild` die Operation `horizontalSpiegeln`, die das Bild an der horizontalen Achse spiegelt. Die folgende Darstellung soll die Aufgabe verdeutlichen:



Für die Lösung zu dieser Operation ist kein zusätzliches Array-Objekt zugelassen, sie muss also „in place“ erfolgen!

- 7.4.4 Implementiert die Operation `weichzeichnen`. Das Weichzeichnen wird erreicht, indem der Mittelwert der acht umgebenden Bildpunkte des jeweils betrachteten Bildpunktes errechnet wird und dieser Mittelwert dann den neuen Wert dieses Bildpunktes bildet. Warum wäre eine „in place“-Lösung hier problematisch?
- 7.4.5 Implementiert die Operation `punktSpiegeln`, die alle Punkte am Mittelpunkt des Bildes spiegelt.
- 7.4.6 *Zusatzaufgabe:* Implementiert die Operation `spot`, die ein Scheinwerferlicht projiziert.

## Aufgabe 7.5 Wortschatz und Hash-Tabelle (Termin 2)

- 7.5.1 Im vorgegebenen BlueJ-Projekt *Hashing* befindet sich unter anderem das Interface `Wortschatz`, das den möglichen Umgang mit einer Menge von Wörtern beschreibt. Schaut es euch in der Dokumentationsansicht gut an, denn in der nächsten Aufgabe sollt ihr es mit einer Klasse `HashWortschatz` implementieren. Es ist hilfreich, sich zuerst Gedanken über mögliche Testfälle zu machen und diese zu programmieren; dies steigert das Verständnis der Funktionalität eines Wortschatzes und erleichtert das Implementieren. In dieser Aufgabe ist ein JUnit-Testgerüst vorgegeben. Kommentiert die Testmethoden und füllt die Rümpfe.

Falls euch noch weitere Testfälle einfallen, solltet ihr diese ergänzen.

Hinweis: Wenn ihr in euren Testfällen Methoden benutzt, die selbst noch nicht durch Testfälle getestet sind, dann sollten die fehlenden Testfälle unbedingt ergänzt werden.

- 7.5.2 **Skizziert, wie eine Hash-Tabelle aufgebaut ist**, und erläutert bei der Abnahme anhand der Skizze, wie diese funktioniert und welche Vorteile diese Lösung gegenüber einer Listenimplementierung (ohne Hash-Verfahren) hat.

- 7.5.3 Vervollständigt nun die Klasse `HashWortschatz`, indem ihr ein Hash-Verfahren implementiert. Für die Hashwertberechnung steht (über einen Konstruktor-Parameter) eine Implementation des Interfaces `HashWertBerechner` bereit. Die Angabe eines Berechners erlaubt die Verwendung verschiedener Hash-Funktionen. Denkt daran, dass der von der Hash-Funktion gelieferte Wert noch auf die Größe der Tabelle angepasst werden muss, um einen gültigen

Index in die Tabelle zu erhalten. Das ist in mehreren Methoden nötig, beispielsweise in `fuegeWortHinzu`. Falls ihr ein und denselben Quelltext in mehreren Methoden benötigt, solltet ihr nicht copy/paste verwenden, sondern die Logik in eine Hilfsmethode auslagern.

Verwendet als (Überlauf-)Behälter Exemplare der mitgelieferten Klasse `WortListe`. Eure Hash-Tabelle soll also ein Array von `WortListen` sein. Die Größe dieser Hash-Tabelle bekommt eure Implementation über den Konstruktor von `HashWortschatz` als Parameter übergeben.

Tipp: Überlegt euch zu Beginn, welche Zustandsfelder die Klasse `HashWortschatz` benötigt, und beginnt mit der Implementierung des Konstruktors. Beachtet die Kommentare im Interface `Wortschatz`, wenn ihr danach die einzelnen Methoden implementiert.

- 7.5.4. Implementiert in der Klasse `HashWortschatz` die Operation `schreibeAufKonsole` so, dass sie den Inhalt aller Überlaufbehälter auf der Konsole ausgibt. In der Darstellung soll pro Überlaufbehälter eine Zeile verwendet werden, etwa folgendermaßen:

```
[0]: [Hund, Katze]
[1]:
[2]: [Maus]
```

Ruft nun (über einen Rechtsklick auf die Klasse `Startup`) die Methode `visualisiereHashtabelle` auf. Diese erzeugt ein Exemplar eurer `Wortschatz`-Implementation, fügt einige Wörter aus einer kurzen Textdatei in den Wortschatz ein und ruft anschließend `schreibeAufKonsole` auf.

- 7.5.5 Implementiert in `HashWortschatz` zusätzlich die Methode `fuellgrad`, die den *Füllgrad* in (ganzzahligen) Prozent angibt. Der Füllgrad sei über das Verhältnis zwischen der Anzahl der enthaltenen Wörter und der Größe der Hash-Tabelle definiert. Implementiert außerdem die Methode `laengsteKette`, die die Größe des vollsten Überlaufbehälters liefert.
- 7.5.6 Welche Komplexität hat die Operation `enthaeltWort(String)` im günstigsten Fall? Welche im ungünstigsten Fall? Wovon hängt das ab?
- 7.5.7 Erstellt eine weitere Klasse, die das Interface `HashWertBerechner` derart realisiert, dass `hashWert` einen konstanten Wert (unabhängig vom übergebenen Wort) zurückgibt. Was bewirkt das? Ergänzt die Testklasse um einen Test mit der neuen Klasse.
- 7.5.8 *Zusatzaufgabe:* Ergänzt die Methode `vergleichePerformance` der Klasse `Startup` so, dass sie beide Realisierungen des Interfaces `HashWertBerechner` testet. Um einen messbaren Unterschied feststellen zu können, wird ein sehr großer Wortschatz verwendet.
- 7.5.9 *Zusatzaufgabe:* Erstellt mindestens zwei weitere Implementationen von `HashWertBerechner`, welche eigene, sinnvolle Hash-Funktionen bereitstellen. Dazu müsst ihr lediglich „irgendwie“ die Buchstaben des eingegebenen Wortes verwenden, um daraus einen Integerwert zu berechnen. Bezieht diese Lösungen in eure Messungen mit ein!