

BTI1-PTP, Aufgabenblatt 7

Sommersemester 2018

Arrays – Vererbung – Listen und Mengen implementieren

Ausgabedatum: 08. Juni 2017

Lernziele

Einfache und mehrdimensionale Arrays verstehen und anwenden können, Schleifen über Arrays verwenden können; Die Implementationsform verkettete Liste umsetzen können; weitere Anwendungen von Interfaces kennen.

Kernbegriffe

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.

In objektorientierten Programmiersprachen können Typen hierarchisch angeordnet werden, es entstehen *Typhierarchien*. Jede Klasse und jedes Interface in Java definiert einen Typ. Subtypen werden gebildet, indem ein Interface ein anderes erweitert oder indem eine Klasse eine andere Klasse erweitert bzw. ein Interface implementiert. *Subtypen* (speziellere Typen) bieten immer mindestens die Schnittstelle ihrer *Supertypen* (allgemeinere Typen), darüber hinaus können sie diese Schnittstelle um eigene Operationen erweitern.

Will man ein Exemplar eines spezielleren Typs, dass an eine Variable mit einem allgemeineren statischen Typ gebunden ist, wieder unter seinem speziellen Typ ansprechen (um auch Operationen aufrufen zu können, die nur der speziellere Typ besitzt), ist eine *Typzusicherung* notwendig. In Java geschieht dies durch das Schreiben des spezielleren Typs in runden Klammern vor einem Ausdruck. *Vorsicht:* Eine Typzusicherung ohne vorherigen *Typstest* (in Java mit `instanceof`) kann zur Laufzeit zu einer `ClassCastException` führen, wenn das Exemplar nicht dem spezielleren Typ entspricht.

Eine Java-Klasse kann mit dem Schlüsselwort **extends** von genau einer anderen *Klasse* erben (engl. *inherit, inheritance*). Die *Unterklasse* (engl. *subclass, derived class*) erbt von der *Oberklasse* (auch *Basisklasse*, engl.: *super class, base class*) zusätzlich zum Typ auch deren Implementation, also ihre Methoden und Exemplarvariablen. Aus diesem Grund spricht man hier von *Implementationsvererbung*. Wenn man als Autor einer Klasse keine Oberklasse angibt, wird von der Klasse `java.lang.Object` geerbt, in der u.a. `boolean equals(Object o)`, `int hashCode()` und `String toString()` definiert sind.

Geerbte Methoden können in Unterklassen *redefiniert* werden (engl.: *override*). Beim Redefinieren wird eine bestehende Implementation einer Operation durch eine andere ersetzt. Dies ist abzugrenzen vom Überladen, bei dem eine neue Operation eingeführt wird, die sich in ihrer Signatur von einer bereits bestehenden, gleichnamigen Operation unterscheidet.

Innerhalb einer redefinierenden Methode kann man die jeweilige redefinierte Methode mit Hilfe des Schlüsselworts **super** aufrufen:

```
super.methodenname(aktuelleParameter); // Ruft die redefinierte Methode auf
```

Normalerweise können von einer Klasse mittels `new Klassenname` Exemplare erzeugt werden. Solche Klassen werden *konkrete Klassen* genannt. Durch die Einführung von Implementationsvererbung wird es jedoch sinnvoll, auch solche Klassen zu schreiben, deren Zweck ausschließlich in der Bereitstellung einer Implementationsbasis für Unterklassen besteht. Solche Oberklassen heißen *abstrakte Klassen* (engl. *abstract class*). Sie werden durch das Schlüsselwort **abstract** im Klassenkopf gekennzeichnet. Mit abstrakten Klassen sollen üblicherweise Redundanzen in der Implementierung mehrerer Klassen vermieden werden. Abstrakte Klassen haben Eigenschaften von Interfaces und konkreten Klassen: Wie bei Interfaces können keine Exemplare von ihnen erzeugt

werden, abstrakte Klassen können aber Exemplarvariablen festlegen und Operationen durch Methoden implementieren.

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.

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()`.

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.

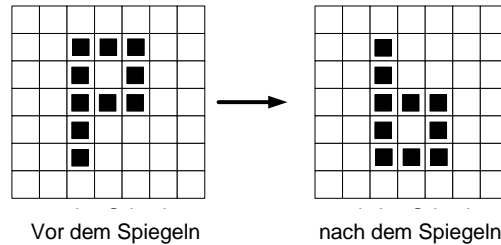
Aufgabe 7.1 Bildbearbeitung

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.

-  7.1.1 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.

- 7.1.2 Implementiert die Operation `heller(int delta)` in der Klasse `SWBild`. Mit ihr soll das Bild um den Wert von `delta` aufgehellt werden.
- 7.1.3 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!

- 7.1.4 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?
- 7.1.5 *Zusatzaufgabe:* Implementiert die Operation `punktSpiegeln`, die alle Punkte am Mittelpunkt des Bildes spiegelt.
- 7.1.6 *Zusatzaufgabe:* Implementiert die Operation `spot`, die ein Scheinwerferlicht projiziert.

Aufgabe 7.2 Nutzung von Vererbung

7.2.1 Ein produzierender Betrieb verwaltet seine hergestellten Produkte zurzeit mit folgenden drei Klassen:

```
public class Membranpumpe
{
    private String _bezeichnung;
    private int _tiefe;
    private float _maximalerBetriebsdruck;
    private int _hoehe;
    private String _membranmaterial;
    private int _gewicht;
    private int _maximaleFoerdermenge;
    private int _breite;
}
```

```
public class Kreislpumpe
{
    private int _breite;
    private int _hoehe;
    private int _gewicht;
    private int _anzahlSchaufeln;
    private int _maximaleFoerdermenge;
    private int _maximaleDrehzahl;
    private String _bezeichnung;
    private int _tiefe;
    private float _maximalerBetriebsdruck;
}
```

```
public class Auffangbecken
{
    private int _tiefe;
    private int _volumen;
    private int _breite;
    private int _gewicht;
    private String _bezeichnung;
    private int _hoehe;
}
```

Entwickelt hieraus eine Vererbungshierarchie, welche die gemeinsamen Attribute in Basisklassen zusammenfasst. Verwendet hierbei abstrakte und konkreten Klassen und begründet eure Auswahl.

7.2.2 Erweitert alle Klassen der Vererbungshierarchie mit Konstruktoren, um eine einfache Initialisierung der Klassen zu ermöglichen.

7.2.3 Erweitert alle Klassen der Vererbungshierarchie um eine Methode `infoAusgeben()`, um den (kompletten) Inhalt einer Klasse auf dem Bildschirm auszugeben. Schreibt zudem für die Abnahme eine Methode die mehrere Produkte anlegt und deren Inhalt auf dem Bildschirm ausgibt.

7.2.4 Macht euch mit der Räuber-Beute-Simulation aus der Vorlesung vertraut. Erzeugt hierfür ein Exemplar der Klasse `Simulator` und führt Simulationsschritte aus. Mittels der Methode `starteLangeSimulation` kann die Simulation über einen längeren Zeitraum ausgeführt werden. Welche natürlichen Prozesse werden Ihrer Meinung nach modelliert, die die Anzahl der Füchse erhöhen oder reduzieren.

7.2.5 Im Folgenden wollen wir dir Nutzung der Abstrakten Klasse `Tier` verbessern. Verschiebt das Datenfeld `alter` aus den Klassen `Fuchs` und `Hase` in die Klasse `Tier`. Initialisiert es mit 0 und stell sicher, dass das Programm sich kompilieren lässt.

7.2.6 Verschiebt die Methode `kannGebaeren` aus den Klassen `Fuchs` und `Hase` in die Klasse `Tier`. Und schreibt die Methode folgendermaßen um:

```
public boolean kannGebaeren()
{
    return alter >= gibGebaerAlter();
}
```

```
abstract protected int gibGebaerAlter();
```

Stellt Versionen von `gibGebaerAlter` in den Klassen `Fuchs` und `Hase` bereit, die die verschiedenen Werte für das Gebäralter zurückgeben.


7.2.7 Verschiebt die Methode `alterErhoehen` aus `Fuchs` und `Hase` nach `Tier`, indem Ihr eine abstrakte Methode `gibHoechstalter` in `Tier` und konkrete Versionen in `Fuchs` und `Hase` definiert.


7.2.8 Kann die Methode `traechtig` in die Klasse `Tier` verschoben werden? Wenn ja, dann führt diese Änderung durch.

7.2.9 **Zusatzaufgabe:** Definiert eine vollständig neue Tierart als Subklasse von `Tier`. Ihr müsst hierfür entscheiden, welche Auswirkungen die neue Spezies auf die bestehenden Tiere haben soll. Ihr werdet die Methode `bevoelkern` modifizieren müssen, damit einige Exemplare der neuen Tierart beim Start der Simulation erzeugt werden.

Aufgabe 7.3 Titel-Listen mit verketteten Listen implementieren

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.

 7.3.1 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 (`ArrayTitelListenTest` und `LinkedTitelListenTest`). Führt die JUnit-Tests in `ArrayTitelListenTest` und `LinkedTitelListenTest` nacheinander aus. Dazu könnt ihr jeweils in BlueJ mit rechts auf eine Testklasse klicken und den Menüpunkt *Alles testen* auswählen. **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?

 7.3.2 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.

7.3.3 Vervollständigt die Implementation der `LinkedTitelListe` bis alle Tests durchlaufen.

Zusatzaufgabe 7.4 Titel-Listen mit „wachsenden“ Arrays implementieren



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

7.4.2 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.

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

Danksagung

Die Aufgaben wurden freundlicher Weise von Prof. Dr. Schmoltzky aus früheren Veranstaltungen zur Verfügung gestellt. Zudem wurden Aufgaben aus den Büchern „Java als Erste Programmiersprache“ (J.Goll und C. Heinisch) sowie „Java lernen mit BlueJ“ (D.J. Barnes und M. Kölling) verwendet bzw. ergänzt.