Skript zu Listen

Lars Wechsler

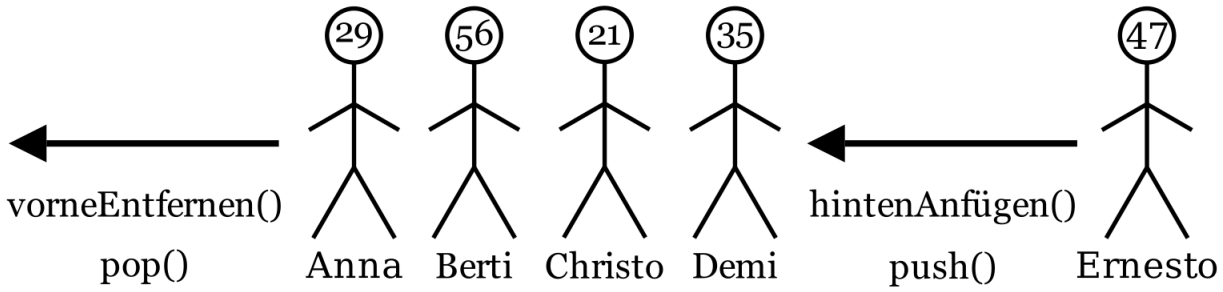
16. August 2022

1 Einleitung

Gemeinhin ist eine Liste eine Aufzählung von Dingen, seien es Zahlen (Altersangaben einer Klasse), Worte (Einkaufszettel) oder eine Liste von Personen.

- Eier
- Mehl
- Salz
- Karrotten

Listen sind so alltäglich, dass wir nicht aktiv über sie nachdenken.
Interessant wird die Übertragung des Konzepts der Liste in die Informatik. Hier benötigen wir noch viel öfter „Zusammenstellungen“von Dingen. Das Zusammenfassen von Objekten zu Listen ist sogar so zentral, dass es etliche Programmiersprachen gibt, die sich auf die Liste als Basiselement zurückgreift, z.B. Lisp.
Traditionell ist die Liste ein umso wichtigeres Element der Sprache, desto funktionaler diese angehaucht ist. In objektorientierten Sprachen sind Listen in ihrer Funktionalität zwar ebenfalls vorhanden, allerdings nicht immer ohne einen gewissen Aufwand zu verwenden.
In den folgenden Kapitel werden wir selbst Implementierungen für Listen in Java entwickeln. Wie oft in der Programmierung gibt es dabei nicht einen Königsweg, sondern verschiedene Ansätze, die unterschiedliche Vor- und Nachteile bringen.
Als Prototyp einer Liste wird uns dabei eine Warteschlange dienen (ganz bildlich gedacht), z.B. die Schlange vor einer Kasse.



Hinweis: Um sprachlich stringenter zu bleiben, werden die Methoden im Fließtext auf Deutsch stehen, mit der üblichen englischen Übersetzung bei der ersten Verwendung in Klammern. Auch die Abbildungen werden beide Schreibweisen enthalten, der Quellcode nur englische Bezeichungen (Im Anhang finden sich zu den entsprechenden Kapiteln auch voll deutsche Code-Schnipsel). Für Prüfungen sind sowohl deutscher als auch englischer Code in Ordnung. Begründungen müssen aber auf Deutsch geschrieben werden. Im Abitur ist auch der Quelltext i.d.R. auf Deutsch.

Zurück zum obigen Bild. Wir werden Anna, Berti, Christo, Demi und Ernesto noch öfter sehen. Sie stehen in einer Warteschlange (Queue). Auch eine Warteschlange ist eine Form von Liste, allerdings mit speziellen Eigenschaften. Halten sich alle an die Regeln, so ist es nur möglich, dass sich jemand hinten anstellt oder vorne aus der Warteschlange entfernt wird, weil er dran ist. In der Modellierung entspricht dies den Methoden vorneEntfernen() (pop()) bzw. hintenAnfügen() (push()). Die Warteschlange gehorcht damit dem sogenannten FIFO-Prinzip (First In, First Out).

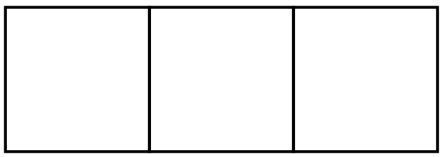
Es bleibt die Frage, wie diese Ideen objektorientiert umgesetzt werden können.

2 Implementierungen von Listen

Man unterscheidet im Wesentlichen zwei große Untertypen von Listenimplementierungen. Die, die sich auf Felder (arrays) stützen und die, die eine „verzeigerte“Struktur aufweisen (linked lists). Der zweite Typ kann noch weiter unterteilt werden, aber dazu später mehr.

2.1 Die Array-Liste

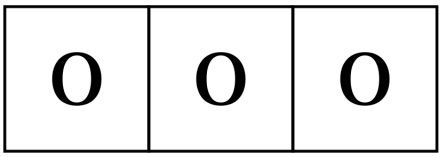
Für den Moment wollen wir uns ein Feld als reservierten Speicherplatz vorstellen, im Folgenden symbolisiert durch eine Aneinanderreihung an Quadraten. Aktuell ist das Feld noch vollständig leer.



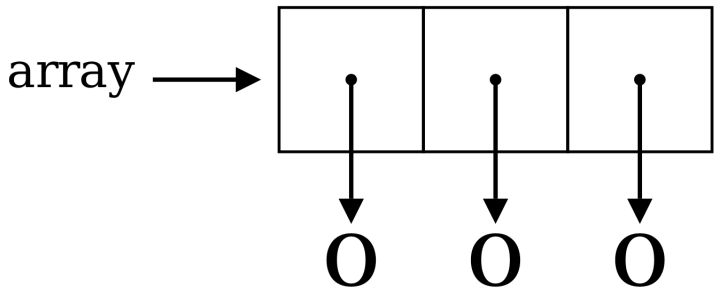
In Java werden Felder standardmäßig mit 0 bzw. null intialisiert, wenn keine weiteren Vorgaben gemacht werden. **Beispiel**:

```
int[] array = new int[3];
```

Die Variable array zeigt nun zu einem Feld, dass 3 mal den Eintrag 0 enthält, in Java-Schreibweise: {0,0,0}



Die Darstellung ist so noch nicht ganz akurat. Das Feld enthält eigentlich nur weitere Zeiger, die zu den entsprechenden Positionen im Arbeitsspeicher zeigen, an denen die Daten liegen. Insbesondere, wenn man keine primitiven Datentypen verwendet, müssten ansonsten häufig Daten verschoben werden, wenn neue Elemente hinzugefügt oder geändert werden.
Eine bessere Darstellung wäre also:



Neben primitiven Datentypen können auch Listen von beliebigen Objekten gebildet werden. Nehmen wir an, wir haben eine Klasse „Mensch “bereits erzeugt, dann wird

```
Human[] humans = new Human[5];
```

ein Feld mit 5 Einträgen null erzeugen, auf das durch die Variable humans zugegriffen wird.
Wir können auf beliebige Einträge eines zuvor erzeugten Feldes zugreifen, z.B.:

```
int[] numbers = {5,3,42,17,-2};
int thirdEntry = numbers[2];
System.out.println(thirdEntry);
```

Der obige Code wird die Zahl 42 auf die Konsole schreiben. Dieses Beispiel soll daran erinnern, dass in der Informatik bei 0 zu zählen begonnen wird. Auf den für uns dritten Eintrag wird also mit der „2“zugegriffen.

Grundsätzlich bietet das Feld an sich in seiner Implementierung bereits alle Eigenschaften, die wir von einer Liste verlangen würden. Wenn unsere Warteschlange benutzt wird, soll aber nur am Anfang entnommen und am Ende angefügt werden können. Es ist also mehr Kontrolle nötig.
Beginnen wir damit unsere Warteschlange als neue Klasse zu definieren. Da es unser Ziel ist eine Schlange von Menschen zu modellieren, brauchen wir aber zunächst eine Klasse, die einen Menschen beschreibt.

```
public class Human() {
    private String name;
    private int age;

    public Human(String name, int age){
        this.name = name;
        this.age = age;
    }

}
```

Unsere Menschen haben aktuell nur die Attribute Name und Alter und außer dem Konstruktor sind keine weiteren Methoden vorhanden. Das reicht aber bereits, um unsere Warteschlange zu bauen:

```
public class MyListArray(){

    private Human[] queue;

    public MyListArray() {
        queue = new Human[5];
    }

}
```

Um die Warteschlange auch sinnvoll verwenden zu können müssen Methoden ergänzt werden. Wir beginnen mit der hintenAnfügen() Methode. Da wir spezifizieren müssen, was wir anfügen wollen sieht die vollständige Signatur der Methode so aus:

```
public void push(Human humanToAdd)
```

Ein erster - naiver - Ansatz zur Befüllung wäre der folgende:

```
public void push(Human humanToAdd) {
    for(int i = 0; i < queue.length; i++) {
        if(queue[i] == null) {
            queue[i] = humanToAdd;
        }
    }
}
```

Aufgabe 1: Begründe Sie kurz, warum diese Implementierung ineffizient ist.

Eine effizientere Lösung besteht darin, die aktuelle Anzahl an Menschen in der Warteschlange in einem Attribut mitzuzählen, dadurch vereinfacht sich die Implementierung zu:

```
public class MyListArray(){

    private Human[] queue;
    private int count;

    public MyListArray() {
        queue = new Human[5];
    }

    public void push(Human humanToAdd) {
        queue[count] = humanToAdd;
        count++;
    }

}
```

Aufgabe 2: Begründen Sie, warum auch diese Implementierung noch problematisch ist. Lösen Sie das Problem!

Aktuell kann unsere Liste nur fünf Elemente enthalten. Versucht man ein sechstes Element anzuhängen, so erhält man als Nachricht „java.lang.ArrayIndexOutOfBoundsException“.
Ist die Liste voll, so muss das Feld vergrößert werden, da dies eine Funktionalität ist, die wir eventuell noch an anderer Stelle brauchen könnten, wird eine eigene Methode angelegt.

```
public void push(Human humanToAdd) {
    if(count == queue.length) {
        enlargeArray();
    }
    queue[count] = humanToAdd;
    count++;
}

private void enlargeArray() {
    Human[] newQueue = new Human[queue.length + 10];
    for(int i = 0; i < queue.length; i++){
        newQueue[i] = queue[i];
    }
    queue = newQueue;
}
```

Um die Methode effizienter zu gestalten, wird das Feld um 10 Plätze vergrößert. Man möchte zu große Felder vermeiden, um nicht unnötig Speicherplatz zu verschwenden (die Implementierung der ArrayList, die von Java direkt angeboten wird verwendet z.B. eine Vergrößerung um 5 Plätze).
Als nächstes muss noch das vorderste Element der Warteschlange entfernt werden können. Spätestens hier zeigt sich, dass das Feld für diese Anwendungsform der Liste nur schlecht geeignet ist. Wenn das vorderste Element entfernt wird, müssen alle folgenden Elemente einen Platz nach vorne geschoben werden. Dieser Kopiervorgang nimmt vergleichsweise viel Zeit und Leistung in Anspruch. Eine mögliche Implementierung der Methode sieht folgendermaßen aus:

```
public Human pop() {
    if(queue[0] == null) {
        return null;
    }
    Human toReturn = queue[0];
    for(int i = 0; i < queue.length -1; i++){
        if (queue[i+1] == null){
            break;
        }
        queue[i] = queue[i+1];
    }
    queue[queue.length-1] = null;
    count--
    return toReturn;
}
```

Die Methode gibt eine Referenz auf das entfernte Objekt zurück, sofern es existiert. Wird diese Referenz nicht in einer neuen Variable gespeichert bei Anwendung der Methode, so wird sie vom garbage collector aufgeräumt und ist vollständig verschwunden.

Die folgenden Aufgaben steigen grob im Schwierigkeitsgrad an und erweitern die Funktionalität der Warteschlange sukzessive.

Aufgabe 3: Schreiben Sie eine Methode schreibeListe (printList), die die Warteschlange in einer sinnvollen Weise auf der Konsole sichtbar macht.
Hinweis: Ergänzen Sie eine passende Methode in der Klasse Mensch (Human)!

Aufgabe 4: Schreiben Sie eine Methode menschAnPosition(int position) (humanAtPosition(int position)), die eine Referenz zum Menschen an der angegebenen Position zurückgibt. Beachten Sie Grenz- bzw. Spezialfälle!

Aufgabe 5: Schreiben Sie eine Methode sucheMenschInSchlange(Mensch m) (searchHumanInQueue(Human h)), die die Position des Menschen in der Schlange zurückgibt.
Hinweis: Überschreiben Sie die allgemeine equals() Methode in der Klasse Mensch.

Aufgabe 6: Schreiben Sie eine Methode enthält(Mensch m) (contains(Human h)), die wahr zurückgibt, wenn der der entsprechende Mensch in der Schlange ist, anderfalls falsch.
Hinweis: Verwenden Sie die equals-Methode aus Aufgabe 5

Aufgabe 7: Schreiben Sie eine Methode entferneAnPosition(int position) (removeAtPosition(int position)), die den Menschen an der gegebenen Stelle entfernt und die übrigen nach vorne rutschen lässt.
Schreiben Sie zwei Versionen dieser Methode, eine, die den Menschen nur entfernt und eine, die eine Referenz zu diesem Menschen zurückgibt.

Aufgabe 8 - für Experten: Schreiben Sie eine Methode zusammenfügen(MeineListeFeld zweiteWarteschlange) (concatenate(MyListArray secondQueue)), die eine zusammengefügte Liste zurückgibt.

Aufgabe 9 - für Experten: Schreiben Sie eine Methode fügeSortiertHinzu(Mensch m) (appendSorted(Human h)), die die Warteschlange nicht von hinten auffüllt, sondern die Menschen an einer bestimmten Stelle einsortiert.
Hinweis: Dazu muss eine Vergleichsmethode in der Klasse Mensch definiert werden. Z.B. könnte nach Alter, oder auch nach Namen oder nach einer Kombination von beidem sortiert werden.

Aufgabe 10 - frei: Überlegen Sie sich eigene, sinnvolle weitere Methoden für die Warteschlangen-Implementierung.

3 Anhang

3.1 Zu Kapitel 2.1

Code-Fragment 1:

```
    int[] feld = new int[5];
```

Code-Fragment 2:

```
    Mensch[] menschen = new Mensch[5]:
```

Code-Fragment 3:

```
    int[] zahlen = {5,3,42,17,-2};
    int dritterEintrag = numbers[2];
    System.out.println(dritterEintrag);
```

Code-Fragment 4 - erste Definition der Klasse Mensch:

```
public class Mensch() {
    private String name;
    private int alter;

    public Human(String name, int alter){
        this.name = name;
        this.alter = alter;
    }

}
```

Code-Fragment 5 - erste Definition der Klasse MeineListeFeld:

```
public class MeineListeFeld(){

    private Mensch[] warteschlange;

    public MeineListeFeld() {
        warteschlange = new Mensch[5];
    }

}
```

Code-Fragment 6:

```
public void hintenAnfügen(Mensch mensch)
```

Code-Fragment 7 - erste Implementierung von hintenAnfügen:

```
public void hintenAnfügen(Mensch mensch) {
    for(int i = 0; i < warteschlange.length; i++) {
        if(warteschlange[i] == null) {
            warteschlange[i] = mensch;
        }
    }
}
```

Code-Fragment 8 - zweite Version von hintenAnfügen:

```
public class MeineListeFeld(){

    private Mensch[] warteschlange;
    private int anzahl;

    public MeineListeArray() {
        warteschlange = new Mensch[5];
    }

    public void hintenAnfügen(Mensch mensch) {
        warteschlange[anzahl] = mensch;
        anzahl++;
    }

}
```

Code-Fragment 9 - finale Version von hintenAnfügen:

```
    public void hintenAnfügen(Mensch mensch) {
        if(anzahl == warteschlange.length) {
            warteschlangeVergroessern();
        }
        warteschlange[anzahl] = mensch;
        anzahl++;
    }

    private void warteschlangeVergroessern() {
        Mensch[] neueWarteschlange = new Mensch[warteschlange.length + 10];
        for(int i = 0; i < warteschlange.length; i++){
            neueWarteschlange[i] = warteschlange[i];
        }
        warteschlange = neueWarteschlange;
    }
```

Code-Fragment 10 - Implementierung von vorneEntfernen:

```
    public Mensch vorneEntfernen() {
        if(warteschlange[0] == null) {
            return null;
        }
        Mensch zuEntfernen = warteschlange[0];
        for(int i = 0; i < warteschlange.length -1; i++){
            if (warteschlange[i+1] == null){
                break;
            }
            warteschlange[i] = warteschlange[i+1];
        }
        warteschlange[warteschlange.length-1] = null;
        anzahl--;
        return zuEntfernen;
    }
```