# Objektorientierte Programmierung — Blatt 3

Institut für Softwaretechnik und Programmiersprachen | Sommersemester 2024
Matthias Tichy, Raphael Straub und Florian Sihler

Abgabe (git) bis
19. Mai 2024

## Arrays Extended

3 ★ 3 🟦 4 🟢

- Ein tieferes Verständnis für Arrays in Java erlangen
- Pass-By-Reference und Pass-By-Value verstehen

### **Aufgabe 1:** The Code Is Missbehaving

🟦🟦🟦

Seit Wochen arbeiten wir an einer die Welt verändernden *Artificial Intelligence*, die uns in Tic-Tac-Toe unbesiegbar machen soll. Leider hat unser Code (völlig unerwartet) ein Problem und wir können nicht herausfinden, was es ist.

Aktuell testen wir die AI mit folgendem Beispiel:

```
System.out.println(new Ai().nextMove(
      new int[]{0, 0, 0, 2, 2, 0, 1, 0, 1}, 1
));
```

Wie man sieht, sollte ein Zug auf das vorletzte Feld gemacht werden, um als Spieler 1 das Spiel mit der Reihe `1 1 1` zu gewinnen. Dennoch schlägt unsere AI vor, dass wir das dritte Feld belegen sollen.

Verwenden Sie den Debugger Ihrer IDE, um den Fehler zu finden und zu beheben! Das eingebettete *To-Do* müssen Sie dabei natürlich nicht implementieren! Sie finden den Code auch im Repository unter `src/main/java/exercise1/Ai.java`.

```
3  public class Ai {
4
5      private static final int[][] WINNING_COMBINATIONS = {
6              {0, 1, 2}, {3, 4, 5}, {6, 7, 8}, {0, 3, 6},
7              {1, 4, 7}, {2, 5, 8}, {0, 4, 8}, {2, 4, 6}
8      };
9
10     public int nextMove(int[] board, int player) {
11        int[] possibleMoves = getPossibleMoves(board);
12        for(int move : possibleMoves) {
13           if(isWinningMove(board, move, player)) {
14              return move;
15           }
16        }
17        // TODO: if the next move does not win immediately
18        throw new UnsupportedOperationException("Not implemented yet");
19     }
20
21     private boolean isWinningMove(int[] board, int move, int player) {
```

```
22          board[move] = player;
23          return isOver(board);
24      }
25
26      private static boolean isOver(int[] board) {
27          for (int[] combination : WINNING_COMBINATIONS) {
28              if (board[combination[0]] != 0 &&
29                      board[combination[0]] == board[combination[1]] &&
30                      board[combination[0]] == board[combination[2]]) {
31                  return true;
32              }
33          }
34          return false;
35      }
36
37      private int[] getPossibleMoves(int[] board) {
38          int numberOfMoves = 0;
39          for(int i = 0; i < board.length; i++) {
40              if(board[i] == 0) {
41                  numberOfMoves++;
42              }
43          }
44          int[] possibleMoves = new int[numberOfMoves];
45          for (int i = 0, j = 0; i < board.length; i++) {
46              if(board[i] == 0) {
47                  possibleMoves[j] = i;
48                  j++;
49              }
50          }
51          return possibleMoves;
52      }
53  }
```

## **Aufgabe 2:** Let's Sort This Out

In dieser Aufgabe sollen Sie Funktionen implementieren, die ein Array von Zahlen sortieren.

Die Test-suite für diese Aufgabe ist dabei relativ minimal gestaltet, da Sie in Teilaufgabe 2d selbst einige Unit-Tests schreiben sollen.

### **a**) Swap

Die Basis vieler Sortieralgorithmen ist das Vertauschen von zwei Elementen. Schreiben Sie daher eine Funktion `void swap(int[] array, int index1, int index2)`, die die Elemente anhand den Indizes `index1` und `index2` vertauscht. Geben Sie zusätzlich Informationen über das Vertauschen im Format des Beispiels in der Konsole aus.

**Beispiel:**

```
int[] array = {10, 9, 8, 7, 6, 5, 4, 3, 2, 1}
swap(array, 4, 5)
```

Mit erwarteter Ausgabe:

```
Swapped value '6' (@index 4) with '5' (@index 5)
```

**b**) Bubblesort ★★●

Bubblesort ist ein einfacher Sortieralgorithmus, der durch wiederholtes Durchlaufen der Liste und Vertauschen benachbarter, noch nicht sortierter, Elemente funktioniert. Bei jedem Durchgang „schwimmt“ (oder „bubbelt“) das größte Element der verbleibenden unsortierten Abschnitte wie eine Blase nach oben zur sortierten Position (natürlich nur unter der Annahme, dass wir aufsteigend sortieren, was wir hier jetzt einmal stillschweigend annehmen). Aufgrund seiner einfachen Implementierung wird Bubblesort oft in Lehrkontexten verwendet, obwohl in seine average- und worst-case Laufzeitkomplexität von $\mathcal{O}(n^2)$ für große Datensätze ineffizient macht

Implementieren Sie den einfachen, aber ineffizienten Bubblesort Algorithmus. Verwenden Sie die bereits implementierte Funktion `swap` aus Teilaufgabe 2a, um die Elemente zu vertauschen.

**c**) Quicksort ★●●

QuickSort ist ein (im Vergleich zu Bubblesort) effizienterer Sortieralgorithmus, der auf dem Divide-and-Conquer Prinzip basiert. Er wählt ein sogenanntes Pivotelement aus der zu sortierenden Liste und ordnet dann alle anderen Elemente um, sodass Elemente kleiner als das Pivot links und solche größer als das Pivot rechts stehen (der „=“-Fall ist dabei der Implementation überlassen). Dies wird so lange für alle Teillisten wiederholt, bis die Liste vollständig sortiert ist (wir also bei Teillisten der Länge 1 angekommen sind). Durch seine durchschnittliche Laufzeitkomplexität von $\mathcal{O}(n \log n)$ ist QuickSort besonders geeignet für große Datensätze und wird häufig in der Praxis eingesetzt.

Implementieren Sie den Quicksort Algorithmus in Java. Verwenden Sie die bereits implementierte Funktion `swap` aus Teilaufgabe 2a, um die Elemente zu vertauschen.

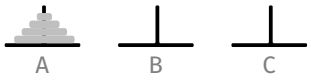
Zum Vergleich, die (relativ naive) Implementierung in Haskell, welche Ihnen vielleicht noch aus Grundlagen der praktischen Informatik bekannt ist, könnte wie folgt aussehen:

```haskell
qs :: Ord a => [a] -> [a]
qs []     = []
qs (p:xs) = qs lesser ++ [p] ++ qs greater
    where
        lesser  = filter (< p) xs
        greater = filter (>= p) xs
```

**d**) Did it sort? ★★■■●●

In dieser Aufgabe sollen Sie Unit-Tests schreiben um zu überprüfen, ob ihre Implementierung korrekt ist. Versuchen Sie zuerst die Eigenschaften und Fälle zu ermitteln, die Sie testen wollen und implementieren Sie dann eine (Ihrer Meinung nach) ausreichende Anzahl an Tests. Testen Sie beide Sortieralgorithmen. Fallen Ihnen hier Unterschiede, Gemeinsamkeiten und Synergien auf?

## **Aufgabe 3:** Towers of Hanoi




Das Problem der Türme von Hanoi besteht darin, eine Anzahl von unterschiedlich großen Scheiben, die anfangs auf einem von drei Stäben in absteigender Größe sortiert sind, vollständig auf einen anderen Stab zu übertragen (eine mögliche Startaufstellung für vier Scheiben ist rechts dargestellt). Dabei darf jeweils nur eine Scheibe bewegt werden, und eine größere Scheibe darf niemals auf einer kleineren Scheibe liegen. Der dritte Stab ist dabei als Hilfsstange zu verwenden, um die Scheiben temporär abzulegen. Die Herausforderung besteht darin, alle Scheiben mit der minimalen Anzahl von Zügen zu versetzen. Zur Übersicht benennen wir die Stäbe mit den Buchstaben A, B und C.

Implementieren Sie dafür eine Funktion `solveHanoi(int numDisks)`.

Die Funktion soll jeden notwendigen Zug in der Konsole ausgeben, formatiert als: „`Move disk [i] from rod [fromRod] to rod [toRod]`“ (bitte beachten Sie, dass die Tests exakt diese Ausgabe erwarten).

Wir empfehlen Ihnen, die Verwendung einer Hilfsfunktion mit folgenden Parametern:

- `int numDisks` — Die Gesamtzahl der Scheiben.
- `char fromRod` — Der Startstab, von dem die Scheiben zu Beginn bewegt werden.
- `char toRod` — Der Zielstab, zu dem alle Scheiben bewegt werden sollen.
- `char auxRod` — Der Hilfsstab, der während des Prozesses verwendet wird.

**Beispiel:**

```
solveHanoi(4);
```

Mit erwarteter Ausgabe:

```
Move disk 1 from rod A to rod B
Move disk 2 from rod A to rod C
Move disk 1 from rod B to rod C
Move disk 3 from rod A to rod B
Move disk 1 from rod C to rod A
Move disk 2 from rod C to rod B
Move disk 1 from rod A to rod B
Move disk 4 from rod A to rod C
Move disk 1 from rod B to rod C
Move disk 2 from rod B to rod A
Move disk 1 from rod C to rod A
Move disk 3 from rod B to rod C
Move disk 1 from rod A to rod B
Move disk 2 from rod A to rod C
Move disk 1 from rod B to rod C
```

## **Aufgabe 4:** Simple Turing Machine

Eine Turing-Maschine ist ein theoretisches Rechenmodell, das von Alan Turing etabliert wurde, um das Konzept der Berechnung und Ausführung von Algorithmen zu erklären. Die Maschine besteht aus einem unendlich langen Band, das in diskrete Zellen unterteilt ist, auf denen Symbole geschrieben werden können. Zudem hat sie einen Lese-/Schreibkopf, der Symbole lesen und

schreiben, sowie sich nach links oder rechts bewegen kann und einem Steuerwerk, das den Zustand der Maschine steuert. Die Turing-Maschine kann jedes Problem lösen, das durch algorithmische Mittel lösbar ist, und dient als Grundlage für das Verständnis, was Computer prinzipiell leisten können. Das Modell ist in der theoretischen Informatik besonders wichtig und hilft zu verstehen, wie Probleme in programmierbare Schritte zerlegt werden können, was auch für das Erlernen von Java nützlich sein kann.

In dieser Aufgabe bekommen Sie den sogenannten *Trace*, also die bereits ausgeführten Schritte einer Turing Maschine. Dieser Trace besteht aus einer Folge an Buchstaben und beschreibt die folgenden Operationen:

- >: Bewegung nach rechts.
- <: Bewegung nach links.

Alle anderen Symbole sollen direkt auf das Band der Turing Maschine geschrieben werden, welches für unsere Zwecke eine endliche Länge als Char-Array hat.

**Beispiel:**

```
char[] tape = {'0', '0', '1', '0', '+', '0', '0', '1', '1'}
turing(tape, ">>>>>>>0<1<0<0<0<0");
```

Mit erwarteter Ausgabe:

```
0 0 0 0 0 0 1 0 1
```

Gehen Sie davon aus, dass der Head/Kopf der Turing Maschine zu Beginn immer auf das erste Element des Arrays zeigt. Falls sich der Kopf vom Band bewegt, soll er auf das andere Ende des Arrays gesetzt werden. Bewegt sich der Kopf also beispielsweise zu weit nach rechts, so soll er wieder auf das erste Element des Arrays gesetzt werden.