# Suchen, Sortieren, Aufwand

Authors of slides:
Partly extracted from script of Prof. Kai-Uwe Sattler

# Lernziele

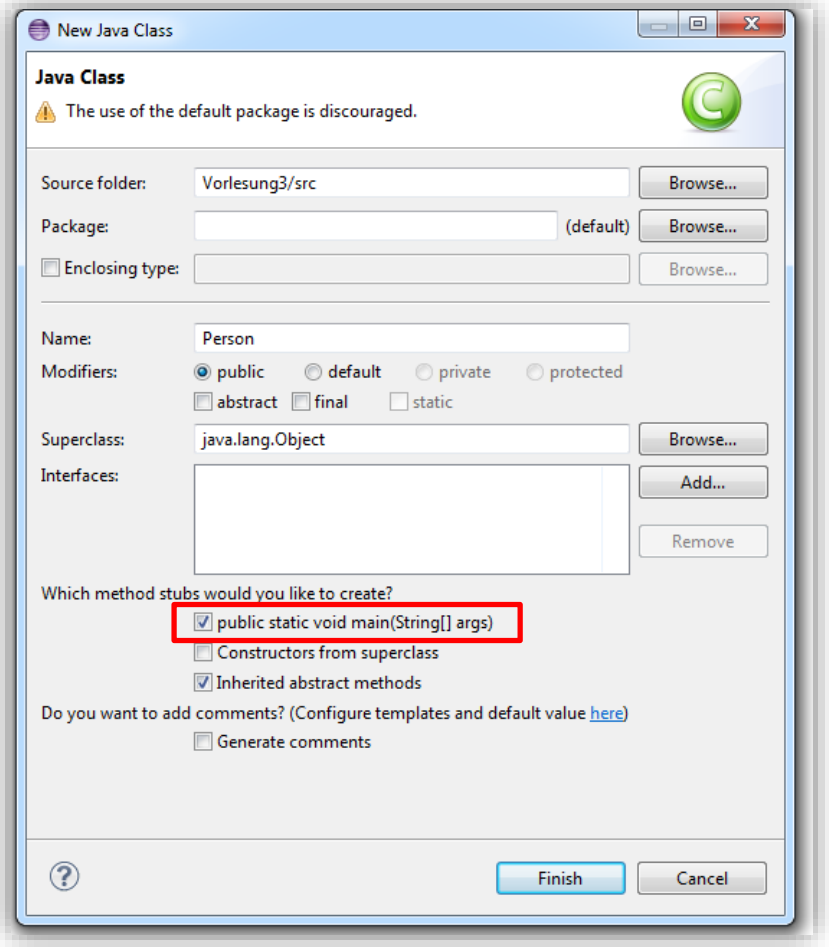
- BubbleSort, sequentielle Suche und binäre Suche implementieren können
- Aufwandabschätzungen

# Die „Start“-Methode: main

- Problem: Woher soll Java wissen, an welcher Stelle des Programms gestartet werden soll?
- Lösung: Spezielle Methode, die als Einsprungpunkt ins Programm dient

```
public static void main(String[] args) {
// TODO Auto-generated method stub

}
```



# Arrays

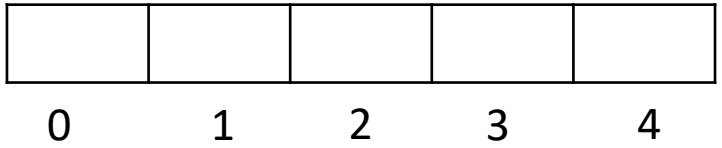
# Arrays I

- **Array** kann mehrere Werte/Objekte des **gleichen Typs** zusammenfassen

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

Typ Array Name Platz im Speicher reservieren! für 5 int-Werte

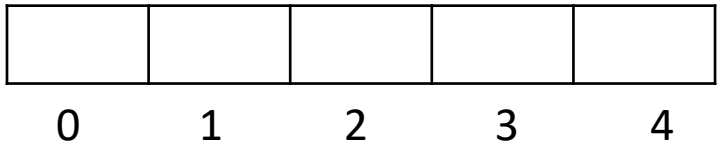
- Deklaration mittels „[ ]“ sowie unter Angabe der Größe
- Die Indizierung (d.h., der Zugriff) auf Elemente im Array fängt bei der Stelle **0** an!



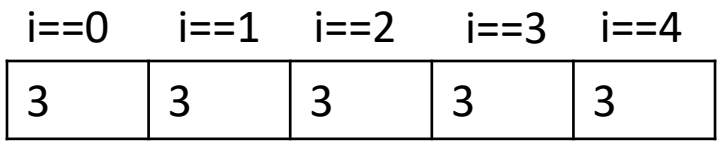
# Arrays II

- Achtung: Beim Anlegen von arrays wird **kein Objekt initialisiert**
- Es wird lediglich Speicherplatz **reserviert**!

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




```
for (int i = 0; i < fiveInts.length; ++i) {
   fiveInts[i] = 3;
}
```



# Arrays III

- Initialisierung

```
int[] fiveInts = new int[5];
int[] fourInts;
fourInts = new int[4];
```

Deklaration

Initialisierung (Speicher wird reserviert)

```
char[] c = new char[3];
c[0]= 'a'; c[1] = 'b'; c[2] = 'c';
char[] c = new char[] {'a', 'b', 'c'};
double[] d = {1.2, 3.5, 2.1};
```

- Größe eines Arrays mittels Variablenname**.length**

```
int[] fiveInts = new int[5];
int size = fiveInts.length; //ergibt 5
```

- Nochmal: Zugriff von 0 bis n–1, wobei n = length

# Sequentielle Suche

# Sequentielle Suche

- Das komplette Array wird von vorn nach hinten durchlaufen
- An jeder Stelle verglichen, ob der gesuchte Wert an dieser Stelle steht
- Wenn der Wert nicht gefunden wird, wird typischerweise -1 ausgegeben

# Sequentielle Suche: Implementierung in Java I

- Pausieren Sie das Video und implementieren Sie die sequentielle Suche in Java
- Die Methodensignatur sieht wie folgt aus:

```
public static int searchNum(int[] nums, int toFind) {
      for (int i = 0; i < nums.length; i++) {
            if (nums[i] == toFind)
                  return i;
      }
      return -1;
}
```

# Aufwand: Wiederholung

- Sie sollten bereits mit Komplexitätsklassen vertraut sein
- Wichtige Klassen nochmal in der Übersicht:

| | |
|---|---|
| $O(1)$ | Konstant |
| $O(\log n)$ | Logarithmisch |
| $O(n)$ | Linear |
| $O(n*\log n)$ | |
| $O(n^2)$ | Quadratisch |
| $O(n^k); k > 0$ | Polynomial |
| $O(2^n)$ | Exponentiell |

# Aufwand: Sequentielle Suche

- Bester Fall:
  - O(1)
  - Der erste Eintrag ist der gesuchte
- Schlechtester Fall:
  - O(n):
  - Der Eintrag ist nicht vorhanden und das gesamte Array muss durchlaufen werden
- Durchschnittlich:
  - O(n)
  - Bei Gleichverteilung der Daten und vielen Suchen würde man n/2 schätzen, was in der Klasse O(n) liegt

# Wie können wir schneller suchen?

- Auf einem sortierten Array könnten wir schneller suchen
- Also sortieren wir das Array erstmal
- Eine einfache Methode ist der BubbleSort-Algorithmus

# Bubble Sort

## 1. Durchlauf

| 0 | 1 | 2 | 3 | 4 | 5 |
|---|---|---|---|---|---|
| 23 | 42 | 5 | 7 | 37 | 99 |
| 23 | 5 | 42 | 7 | 37 | 99 |
| 23 | 5 | 7 | 42 | 37 | 99 |
| 23 | 5 | 7 | 37 | 42 | 99 |

## 2. Durchlauf

| 0 | 1 | 2 | 3 | 4 | 5 |
|---|---|---|---|---|---|
| 23 | 5 | 7 | 37 | 42 | 99 |
| 5 | 23 | 7 | 37 | 42 | 99 |
| 5 | 7 | 23 | 37 | 42 | 99 |

# Bubble Sort: Implementierung in Python

- Pausieren Sie das Video und implementieren Sie den bubbleSort-Algorithmus in Python

```
import array as arr
a = arr.array('i', [23, 42, 5, 7, 37, 99])

def bubbleSort(a):
    hasSwapped = True
    while hasSwapped:
        hasSwapped = False;
        for i in range (1,len(a)):
            if a[i-1] > a[i]:
                temp = a[i-1]
                a[i-1] = a[i]
                a[i] = temp
                hasSwapped = True
    return a

print(bubbleSort(a))
```

# Aufwand: BubbleSort

- Bester Fall:
  - O(n)
  - Das Array ist bereits sortiert
- Schlechtester Fall:
  - $O(n^2)$:
  - Das Array ist falsch herum sortiert und muss komplett umgedreht werden:
  - Innere Schleife hat n Vergleiche
  - Äußere Schleife hat n/2 Vergleiche (weil nach jedem Schleifendurchlauf ein Element weniger zu vergleichen ist, da das größte Element bereits an der richtigen Stelle steht)
- Durchschnittlich:
  - $O(n^2)$
  - Ähnlich wie beim schlechtesten Fall

# Binäre Suche: Konzept

- Auch Telefonbuchsuche
- Wir suchen die Zahl 42
- Das "Telefonbuch" wird in der Mitte aufgeschlagen
  - Index 9 (19/2 –> 9,5; Java ignoriert den Rest der ganzzahligen Division)
  - Index 14 ((9 + 19)/2 -> 14)
  - Index 11 ((9 + 14)/2 -> 11,5 -> also 11)

| 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 | 13 | 14 | 15 | 16 | 17 | 18 | 19 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 1 | 5 | 6 | 7 | 13 | 15 | 17 | 23 | 33 | 37 | 37 | 42 | 43 | 52 | 56 | 73 | 78 | 89 | 90 | 99 |

# Binäre Suche: Implementierung in Python

- Pausieren Sie das Video und implementieren Sie die binäre Suche in Python

```
def binarySearch (a, key):
    low = 0
    high = len(a) - 1
    while (low < high):
        index = low + math.floor((high - low) / 2)
        print(index)
        if a[index] == key:
            return index
        if a[index] > key: #links
            high = index
        else: low = index + 1 #rechts
    return -1
```

# Aufwand: Binäre Suche

- Bester Fall:
  - O(1)
  - Der erste Eintrag ist der gesuchte
- Schlechtester Fall:
  - O(log *n*)
  - Der Eintrag ist nicht vorhanden
  - Die Anzahl der zu durchsuchenden Elemente halbiert sich mit jedem Schritt
- Durchschnittlich:
  - O(log *n*)
  - Begründung wie beim schlechtesten Fall

# Sequentielle Suche oder Suchen und Sortieren? I

- Sie haben ein unsortiertes Array mit Nachnamen (keine doppelten Einträge)
- Sie wollen wissen, ob ein bestimmter Nachname vorkommt
- Was ist die schnellste Strategie?
  - Erst sortieren und dann binär suchen?
  - Sequentielle Suche?
- Diese Entscheidung hängt ab von zwei Faktoren:
  - Länge des Arrays
  - Anzahl der Suchen

# Sequentielle Suche oder Suchen und Sortieren? II

- Angenommen: Länge des Arrays ist 10, und wir suchen einmal
  - O(n) vs. O (log n) + einmalig $O(n^2)$ für Sortieren
  - 10 vs. log 10 + $10^2$
  - 10 vs. 1 + 100
- Wie sieht es aus bei 10 mal etwas suchen?
  - 10 * 10 vs. 10*1 + 100
  - 100 vs. 110
  - Sortieren zahlt sich relativ schnell aus!

# Lernziele

- BubbleSort, sequentielle Suche und binäre Suche implementieren können
- Aufwandabschätzungen