Setup

Welche Schritte muss ich befolgen, um an einem neuen Gerät in Java zu programmieren?

> - Mein **Betriebssystem** ist **Windows** (10 oder höher), Linux oder MacOS
> - Mein **Broswer** ist **Firefox**, Safari, Chrome, oder Edge
> - Meine **IDE** ist der **Java-Editor**, Atom, VSCode oder Eclipse
>   - Den JDK 17 Windows Installer finde ich auf https://www.oracle.com
>   - Den Java-Editor finde ich auf https://javaeditor.org
> - Zur **Dokumentation** nutze ich **Markdown**, Word, Notion oder ein klassisches Notizbuch
>   - Die IDE Visual Studio Code finde ich auf https://code.visualstudio.com
>   - Markdown nutze ich in VSCode mithilfe der Extension "*Markdown All in One*"

Such- und Sortieralgorithmen

Lineare Suche

> Suche ein Element in einer Liste, indem von links nach rechts jedes Element auf Gleichheit geprüft wird. Brich ab, sobald das Element gefunden ist, und gib die Stelle aus. Laufzeit: O(n)

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

Binäre Suche

> Suche ein Element in einer **sortierten** Liste, indem das mittlere Element (falls gerade mit dem linken) verglichen wird. Ist es größer als das gesuchte, fahre mit der linken Hälfte fort, sonst mit der rechten. Brich ab, sobald das Element gefunden ist oder die Teilliste Länge 1 hat, und gib die Stelle aus. Laufzeit: O(logn)

```
public static int binarySearch(int[] list, int e) {
    int left = 0;
    int right = list.length - 1;
    while (left <= right) {
        int middle = left + (right - left) / 2;
        if (list[middle] == e)
            return middle;
```

```
            if (list[middle] < e)
                left = middle + 1;
            else
                right = middle - 1;
        }
        return -1;
    }
```

Insertionsort

> Beim Insertionsort wird eine List sortiert, indem die Elemente nacheinander in eine bereits sortierte Teilliste eingefügt werden. O(n^2)
>
> Beispiel:
>
> 5 2 4 6 1 3
> 5 2 4 6 1 3
> 2 5 4 6 1 3
> 2 4 5 6 1 3
> 2 4 5 6 1 3
> 1 2 4 5 6 3
> 1 2 3 4 5 6
> 1 2 3 4 5 6

```
public static int[] insertionSort(int[] list) {
    int tmp;
    for (int i = 1; i < list.length; i++) {
        tmp = list[i];
        int j = i;
        while (j > 0 && list[j-1] > tmp) {
            list[j] = list[j-1];
            j--;
        }
        list[j] = tmp;
    }
    return list;
}
```

Selectionsort

> Beim Selectionsort wird eine Liste sortiert, indem das kleinste Element gesucht und mit dem ersten Element getauscht wird. Anschließend werden das nächstkleinste und das zweite Element getauscht und so weiter, bis die gesamte Liste sortiert ist. O(n^2)
>
> Beispiel:
>
> 5 2 4 6 1 3
> 1 2 4 6 5 3

> 1 2 4 6 5 3
> 1 2 3 6 5 4
> 1 2 3 4 5 6
> 1 2 3 4 5 6
> 1 2 3 4 5 6
> 1 2 3 4 5 6

```
public static int[] selectionSort(int[] list) {
    for (int i = 0; i < list.length - 1; i++) {
        int minIndex = i;

        // Finde den Index des kleinsten Elements im unsortierten Teil
        for (int j = i + 1; j < list.length; j++) {
            if (list[j] < list[minIndex]) {
                minIndex = j;
            }
        }

        // Tausche das kleinste Element mit dem ersten Element im unsortierten
Teil
        int tmp = list[minIndex];
        list[minIndex] = list[i];
        list[i] = tmp;
    }
    return list;
}
```

Bubblesort

> Beim Bubblesort wird eine Liste sortiert, indem eine Art Blase von links nach rechts paarweise Elemente vergleicht und gegebenenfalls vertauscht. Dieser Prozess wird solange wiederholt, bis die gesamte Liste sortiert ist. O(n^2)
>
> Beispiel:
>
> 5 2 4 6 1 3
> 2 5 4 6 1 3
> 2 4 5 6 1 3
> 2 4 5 6 1 3
> 2 4 5 1 6 3
> 2 4 5 1 3 6
> 2 4 5 1 3 6
> 2 4 5 1 3 6
> 2 4 1 5 3 6
> 2 4 1 3 5 6
> 2 4 1 3 5 6
> 2 1 4 3 5 6
> 2 1 3 4 5 6

> 1 2 3 4 5 6
> 1 2 3 4 5 6
> 1 2 3 4 5 6
> 1 2 3 4 5 6

```
public static int[] bubbleSort(int[] list) {
    int tmp;
    for (int i = 1; i < list.length; i++) {
        for (int j = 0; j < list.length - i; j++) {
            if (list[j] > list[j+1]) {
                tmp = list[j];
                list[j] = list[j+1];
                list[j+1] = tmp;
            }
        }
    }
    return list;
}
```

Mergesort

> Mergesort ist ein effizienter Sortieralgorithmus, der eine Divide-and-Conquer-Strategie verwendet. Zuerst wird die Liste in zwei Hälften aufgeteilt, dann werden die Hälften separat sortiert. Anschließend werden die sortierten Hälften rekursiv zusammengeführt, um die endgültig sortierte Liste zu erhalten. O(nlogn)
>
> Beispiel:
>
> 5 2 4 7 6 1 3 8
> 5 2 4 7 | 6 1 3 8
> 5 2 | 4 7 | 6 1 | 3 8
> 2 5 | 4 7 | 1 6 | 3 8
> 2 4 5 7 | 1 3 6 8
> 1 2 3 4 5 6 7 8

Rekursion

Rekursion vs. Iteration

> Iteration und Rekursion sind in der Informatik unterschiedliche Herangehensweisen, Probleme zu lösen. Sie haben unterschiedliche Vor- und Nachteile und werden daher in bestimmten Szenarien bevorzugt. Grundsätzlich lässt sich jedoch jede Rekursion auch durch eine Iteration ersetzen.
>
> Bei der **Iteration** werden Anweisungen wiederholt ausgeführt, bis eine bestimmte Bedingung erfüllt ist.
>
> Vorteile:
>
> - Einfacher zu verstehen und implementieren

> - Oft effizienter bezüglich Laufzeit und Speicherbedarf
> - Passend für Algorithmen mit fester Anzahl an Wiederholungen
>
> Nachteile:
>
> - Manche Probleme sind von Natur aus rekursiv definiert, sodass auch die Lösung rekursiv effizienter sowie nachvollziehbarer ist
> - der Code kann in bestimmen Szenarien unübersichtlicher werden, wenn mehrer Schleifen verschachtelt sind
>
> Bei der **Rekursion** ruft eine Funktion sich selbst auf, um eine kleinere Version des gleichen Problems zu lösen. Es teilt daher ein großes Problem in viele kleine auf. Dieser Ansatz wird auch **"divide and conquer"** bzw. **"teile und herrsche"** genannt. Das kleinstmögliche Problem fangen wird mit der sog. **Abbruchbedingung** auf.
>
> Vorteile:
>
> - Es kann zu eleganterem und prägnanterem Code führen
> - Für bestimme Probleme (insb. mathmatischer Natur) ist es natürlicher
>
> Nachteile:
>
> - Es kann schwerer zu verstehen sein und bedarf ein tieferes Verständnis des Problems
> - Es kann in bestimmen Situationen weniger effizient sein

Beispiele

Summe

```
public static int sumIterative(int n) {
    int sum = 0;
    for (int i=1; i<=n; i++) {
        sum += i;
    }
    return sum;
}

public static int sumRekursive(int n) {
    if (n == 1) return 1;
    return n + sumRecursive(n-1);
}
```

Objektorientierte Modellierung

Klasse vs Objekt

> Klassen sind Vorlagen, aus denen Instanzen genannte Objekte zur Laufzeit erzeugt werden.

- Beispiel: Die Klasse `Auto` hat vier **Attribute** (Sichtbarkeit: `private`) samt **Gettern** und **Settern**, einen **Konstruktor** und eine weitere **Methode** `istAlt()`.

```
public class Auto {
    // Attribute
    private String marke;
    private int baujahr;
    private double verbrauch;
    private String farbe;

    // Konstruktor
    public Auto(String marke, int baujahr, double verbrauch, String farbe) {
        this.marke = marke;
        this.baujahr = baujahr;
        this.verbrauch = verbrauch;
        this.farbe = farbe;
    }

    // get-Methoden
    public String getMarke() {
        return this.marke;
    }

    public int getBaujahr() {
        return this.baujahr;
    }

    public double getVerbrauch() {
        return this.verbrauch;
    }

    public String getFarbe() {
        return this.farbe;
    }

    // set-Methoden
    public void setMarke(String marke) {
        this.marke = marke;
    }

    public void setBaujahr(int baujahr) {
        this.baujahr = baujahr;
    }

    public void setVerbrauch(double verbrauch) {
        this.verbrauch = verbrauch;
    }

    public void setFarbe(String farbe) {
        this.farbe = farbe;
    }
```

```
    // weitere Methoden
    public boolean istAlt() {
        return this.baujahr <= 1990;
    }
}
```

- Beispiel: Die Console `Autohaus` instanziiert (also erzeugt) drei Objekte der Klasse (also des Typs) `Auto`. Sie kann auf die Getter und Setter zugreifen, um Daten auszulesen oder zu ändern. Das Schlüsselwort `new` ruft dabei den Konstruktor auf, der entsprechende Paramter (hier: `String, int, double, String`) kennt.

```
public class Autohaus {
    public void main(String[] args) {
        Auto autoAbels = new Auto(“Toyota”, 2012, 6.7, “schwarz”);
        Auto autoBob = new Auto(“Audi”, 2009, 8.3, “blau”);
        Auto autoCorbi = new Auto(“VW”, 2005, 9.2, “grün”);

        System.out.println(autoAbels.getMarke());
        autoAbels.setMarke(“VW”);
        System.out.println(autoAbels.getMarke());
    }
}
```

Vererbung

> Wenn eine Unterklasse von einer Oberklasse erbt, erhält sie automatisch all ihre Attribute und Methoden.

- Beispiel: Die Klasse `Lehrer` dient als **Oberklasse**. Sie enthält vier Attribute, einen Konstruktor und eine Methode `gibLehrerAus()`.

```
public class Lehrer {
    protected int id;
    protected String vorname;
    protected String nachname;
    protected String fach;

    public Lehrer (int id, String vorname, String nachname, String fach) {
        this.id = id;
        this.vorname = vorname;
        this.nachname = nachname;
        this.fach = fach;
    }

    public void gibLehrerAus() {
        System.out.println(“Lehrer“);
        System.out.println(“ID: “ + id);
        System.out.println(“Name: “ + vorname + “ “ + nachname);
```

```
        System.out.println(“Fach: “ + fach);
    }
}
```

- Beispiel: Die Klasse `Fachleiter` dient als **Unterklasse** und erbt durch das Schlüsselwort `extends` von `Lehrer`. Damit hat sie automatisch die vier Attribute und die Methode `gibLehrerAus()`. Hier wird sie noch ergänzt um die Methode `gibFachleiterAus()`. Das Schlüsselwort `super()` ruft im Konstruktor der Unerklasse zunächst den Konstruktor der Oberklasse auf. Somit werden die Attribute, die von der Oberklasse vererbt wurden, zunächst initialisiert.

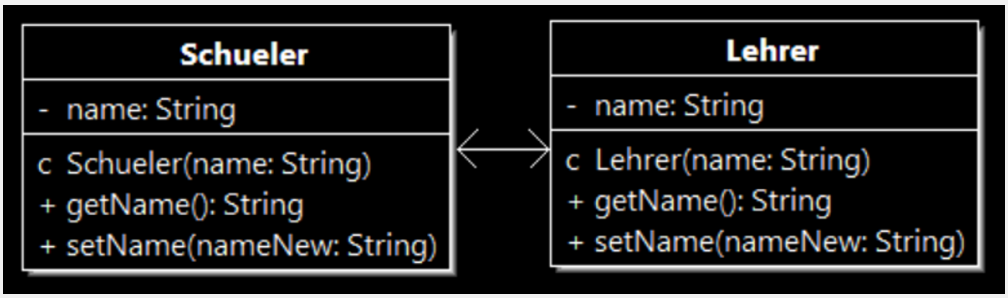
```
public class Fachleiter extends Lehrer {
    private int buero;

    public Fachleiter (int id, String vorname, String nachname, String fach, int
buero) {
        super(id, vorname, nachname);
        this.buero = buero;
    }

    public void gibFachleiterAus() {
        System.out.println(“Fachleiter“);
        System.out.println(“ID: “ + id);
        System.out.println(“Name: “ + vorname + “ “ + nachname);
        System.out.println(“Fach: “ + fach);
        System.out.println(“Büro: “ + buero);
    }
}
```

- Beispiel: Die Console `Main` instanziiert jeweilig ein Objekt vom Typ `Lehrer` sowie vom Typ `Fachleiter`. Dann ruft sie die Methoden `gibLehrerAus()` und `gibFachleiterAus()` auf. Bemerke, dass der Fachleiter nun auch die Methode `gibLehrerAus()` kennt, zumal ein Fachleiter ja auch ein Lehrer ist.

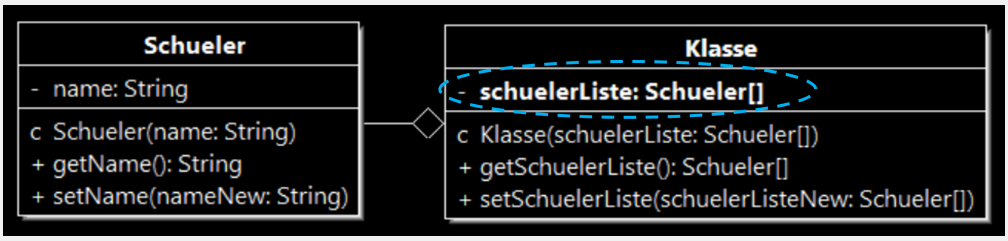
```
public class Main {
    public static void main (String[] args) {
        Lehrer abe = new Lehrer(1, "Patrick", "Abels", "Informatik");
            Fachleiter bru = new Fachleiter(2, "Kerstin", "Brunnermeier",
"Englisch", 112);

            abe.gibLehrerAus();
            bru.gibLehrerAus();
            bru.gibFachleiterAus();
    }
}
```

Beziehungen zwischen Klassen

- Eine **Assoziation** ("kennt"-Beziehung) beschreibt die Beziehung zwischen zwei Klassen, bei der jedes Objekt einer Klasse mit Objekten einer anderen Klasse in Verbindung steht.
    - **Beispiel**: Die Klassen `Schüler` und `Lehrer`können in einer Assoziation stehen, da Schüler Beziehungen zu Lehrern haben. Es gibt jedoch keine direkte Abhängigkeit, und beide Klassen *können unabhängig voneinander existieren.*



- Eine **Aggregation** ("ist-Teil-von"-Beziehung) ist eine spezielle Form der Assoziation, bei der eine Klasse (Container) eine Sammlung von Objekten einer anderen Klasse besitzt, und diese Objekte können *unabhängig voneinander existieren*.
    - **Beispiel**: Die Klasse `Klasse` kann eine Aggregation mit der Klasse `Schueler` haben, da eine Klasse eine Sammlung von Schülern enthält. Die Schüler können jedoch auch ohne die Klasse existieren.



- Eine **Komposition** ("ist-Teil-von"-Beziehung) ist eine strenge Form der Aggregation, bei der die Lebensdauer der enthaltenen Objekte von der Lebensdauer des übergeordneten Objekts abhängt. *Wenn das übergeordnete Objekt zerstört wird, werden auch die enthaltenen Objekte zerstört.*
    - **Beispiel**: Die Klasse `Raum` kann eine Komposition mit der Klasse `Schulgebaeude` haben, da der Raum nur im Kontext des Schulgebäudes existiert. Wird das Schulgebäude zerstört, endet auch die Existenz des Raums.