# Abschlussprojekt SS22

**Zusammenfassung**

Bearbeiten Sie folgende Aufgaben und Beachtung der im Studium gelernten Programmierkonzepte. Testen Sie vor Abgabe, ob alle Ihre Lösungen funktionstüchtig, wieso dokumentiert, sauber, ordentlich, sinnvoll und vollständig sind. Beachten Sie, dass keine Ihrer umgesetzten Lösungen in der Main-Klasse oder Main-Methode steht!

## 1 Utils

Implementieren Sie folgende Funktionen / Methoden.

- `min(int ... numbers)` Nimmt eine beliebige Anzahl an Zahlen an und gibt die kleinste dieser Zahlen zurück.
- `max(int ... numbers)` Nimmt eine beliebige Anzahl an Zahlen an und gibt die größte dieser Zahlen zurück.
- `abs(int number)` Nimmt eine beliebige Zahl and und gibt diese positiv zurück. z.B. `abs(-4) = 4`, `abs(4) = 4`.
- `swap(T[] array, int a, int b)` Nimmt ein Array vom beliebigen Typ an und vertauscht die Elemente im Array an Position $a$ und $b$.
- `reverse(T[] a)` Nimm ein Array $a = (e_1, e_2, \ldots, e_n)$ mit $n$ Elementen an (wo $e_i$ ($i \in \{1, \ldots, n\}$) die Elemente in $a$ sind) und gibt ein neues Array $b = (e_n, e_{n-1}, \ldots, e_1)$ zurück, das alle Elemente von $a$ in umgekehrter Reihenfolge enthält.
- `contains(T[] a, T e)` Nimmt ein Array $a$ und ein Element $e$ an und gibt `true` zurück, wenn und nur wenn $e$ in $a$ auftaucht.

## 2 Stack

Implementieren Sie einen neuen Datentyp “Stack” (de: Stapel). Ein Stack ist eine Sammlung von Elementen das nach dem “lifo” (Last-In-First-Out) funktioniert: Man kann hinzugefügte Elemente nur in der umgekehrten Reihenfolge ihrer Hinzufügung vom Stack entfernen.

1. Implementieren Sie einen Stack, der das vorgegebene Interface aus Bild 1 verwendet.
2. Passen Sie das Interface und Ihre Implementierung so an, dass der Stack einen generischen Typ verwendet, d.h. einen Typen der bei der Erzeugung eines Stacks angegeben werden kann.

```java
/**
 * The stack maintains a collection of objects based on Last-In-First-Out (LIFO).
 */
public interface Stack {
    /**
     * @return Returns the last added element.
     * @throws EmptyStackException if stack is empty.
     */
    int top() throws EmptyStackException;
    /**
     * @return Returns and removes the last added element.
     * @throws EmptyStackException if stack is empty.
     */
    int pop() throws EmptyStackException;
    /**
     * Puts the given element on top of the stack.
     */
    void push(int e);
    /**
     * @return The current number of elements of the stack.
     */
    int size();
    /**
     * @return True if and only if the stack contains no elements.
     */
    boolean isEmpty();
}
```

Listing 1: Interface Stack

# 3 Textähnlichkeit

Die Textähnlichkeit zwischen zwei Texten kann als Dice-Koeffizient angegeben werden. Umgangssprachlich kann er wie folgt berechnet werden: Als Eingabe bekommt der Algorithmus $A$ zwei Texte $t_1 \in \Sigma^*$ und $t_2 \in \Sigma^*$ (wo $\Sigma^*$ für uns dem Datentyp String entspricht).
Im ersten Schritt berechnet dieser die n-Gramm-Mengen $N(t_1)$ und $N(t_2)$ wobei eine 2-Gramm eines Textes $t =$ `Hallo Welt` die Menge $N(t) =$ {Ha, al, ll, lo, o␣, ␣W, We, el, lt} ist (der Unterstrich ␣ stellt das Leerzeichen dar).
Die $n$-Gramm Menge eines Strings $t$ berechnet man typischerweise mit einem sliding-window Algorithmus. Das window, folgend Fenster genannt, umfasst $n$ Buchstaben und man verschiebt dieses in einer Schleife über die Länge des Textes. In jedem Schleifendurchlauf von $i = 0, \ldots, i-(n-1)$ werden alle $n$ Buchstaben des Fensters als String $a =$ t.substring(i, i+n) in eine initial leere Menge geschrieben und nach dem Schleifendurchlauf wird die entstandene Menge als Ergebnis zurückgegeben.
Die Textähnlichkeit wird anschließend mit dem Dice-Koeffizienten bestimmt, der sich aus folgender Formel ergibt.

$$A(t_1, t_2) = \frac{2|N(t_1) \cap N(t_2)|}{|N(t_1)| + |N(t_2)|} \tag{1}$$

**Beachte:** Den Schnitt $\cap$ zweier Mengen kann man mit der Operation `retainAll` berechnen lassen und die Mächtigkeit $|M|$ einer Menge $M$ kann man mit der Operation `size` bekommen. Achten Sie darauf, dass die Java Mengen mutable (de. veränderbar) sind.

# 4 Model-Klassen

Eine Hochschulsoftware muss mit verschiedenen Arten von Benutzern umgehen, Studenten, Mitarbeiter und Professoren. Alle drei Arten von Benutzern haben einen Vor- / und Nachnamen sowie Geburtsdatum. Ein Student hat zusätzlich eine Matrikelnummer und ein Studienfach. Mitarbeiter und Professoren sind Angestellte der Hochschule und verfügen zusätzlich über ein Gehalt und einen Vorgesetzten, der ein Angestellter ist. Mitarbeiter haben eine Tätigkeitsbeschreibung und Professoren haben einen Forschungsschwerpunkt.
Erstellen Sie Klassen, um Studenten, Mitarbeiter und Professoren zu erstellen. (Falls Sie zusätzliche Klassen benötigen, dann erstellen Sie diese.)