Teil A

Welche Fehler begeht der Entwickler hier?

Ein Entwickler legt eine neue Java-Quelltextdatei mit dem Namen `MyApp.java` an und füllt sie mit folgendem Inhalt:

```
public class App {

}
```

Anschließend will der Entwickler seine Quelltextdatei mit dem Java Compiler `javac` übersetzen. Dazu gibt er folgendes Kommando ein:

```
javac MyApp.java
```

Der Compiler meldet einen Fehler. Woran kann das liegen? Welche Änderungen sind am Quelltext vorzunehmen, damit der Compiler ein Kompilat erzeugen kann?

Der Entwickler hat seinen Fehler im Quelltext behoben. Nun will er sein Programm mit Hilfe des Java Launchers `java` ausführen. Dazu tippt er folgendes Kommando ein:

```
java MyApp.class
```

Abermals erhält der Entwickler einen Fehler. Was hat er falsch gemacht? Wie müsste das Kommando stattdessen lauten?

Der Entwickler bemerkt seinen Fehler und ändert das Kommando entsprechend ab. Doch jetzt erhält er plötzlich einen weiteren Fehler. Angeblich findet der Java Launcher keine `main` Methode. Was hat es damit auf sich? Wie lässt sich das Problem beheben?

Welche Ergebnisse liefern die folgenden Zuweisungen?

Teste folgende Anweisungen in der JShell. Überlege dir vorher, welches Ergebnis zu erwarten ist und prüfe erst anschließend, ob deine Vermutung richtig war. Kannst du die Ergebnisse begründen?

Hinweis: Manche Anweisungen sind syntaktisch falsch. Welche sind es und warum schlagen sie fehl?

Bemerkung: Die JShell startest du mit dem Befehl `jshell`. Du kannst die JShell verlassen, indem du den Befehl `/exit` eintippst und mit Taste Enter bestätigst.

```
int a = 1703;
int b = 0703;
```

```
int c = 4 / 8;
int d = 0xCAFE;
int e = Integer.MIN_VALUE - 1;
int f = Integer.MAX_VALUE + 1;
int g = 07709;
int h = -2e3;
byte i = 120 + 9;
byte j = 0b1000_0000;
byte k = (byte)(0b1000_0000);
byte l = (byte)(0xFF);
byte m = 1;
byte n = 2;
byte o = m + n;
byte p = (byte)(m + n);
int q = 4 / 3;
int r = 5 / 2.5;
double s = 1 / 2;
double t = 1.0 / 2;
double u = (double)(1 / 2);
double v = 1 / 3.0;
float w = 1;
float x = 0.5;
float y = 0.5f;
float z = 3 * 5 / 2 + 2.5f;
float aa = 1.0 + 2.5f;
float ab = 1.0f + 2.5f;
double ac = 1.0 + 2.5f;
char ad = 'c';
char ae = "c";
char af = 0xff;
int ag = (int)('U');
char ah = 'ab';
char ai = '\n';
char aj = '"';
char ak = '\'';
boolean al = false;
boolean am = 1;
boolean an = 0;
boolean ao = (boolean)(1 + 1);
boolean ap = 4 > 3;
boolean aq = 4 == 4;
```

Wie lauten die Wrapper Klassen der primitiven Datentypen?

Finde heraus, wie die Wrapper-Klassen der primitiven Datentypen `byte`, `short`, `char`, `int`, `float`, `double` und `boolean` heißen. Nutze dazu die JDK API Documentation.

Teil B

Beachte folgende Hinweise:

- Falls dir für eine Aufgabe keine Lösung einfällt, widme dich einfach der nächsten Aufgabe. Vielleicht kommt dir später noch eine Idee.
- Ich rate dir dringend davon ab, eine KI einzusetzen, um diese Aufgaben zu lösen. Der Lerneffekt würde dadurch enorm sinken.
- Sofern nicht anders vorgegeben, ist pro Aufgabenstellung je ein Programm zu schreiben, das seine Eingabedaten über Kommandozeilenargumente erhält.
- Denke daran, dass Kommandozeilenargumente immer als Zeichenketten an das Java Programm übergeben werden. Argumente mit Leerzeichen sind auf der Kommandozeile in einfache oder doppelte Hochkommas zu setzen. Beispiel: `java MyApp argument1 argument2 "argument with spaces"`.
- In den Aufgabenstellungen heißt das Programm immer `MyApp`, aber du bist angehalten, passendere Namen zu wählen.
- Die in den Aufgabenstellungen genannten Tipps dienen nur als Denkanstoß und sind nicht verpflichtend. Meistens gibt es viele Lösungsmöglichkeiten für ein Problem.
- Wenn nicht anders vorgegeben, darfst du davon ausgehen, dass der Nutzer seine Argumente im korrekten Format angibt.
- Braucht ein Programm mindestens ein Argument, aber hat der Nutzer keines angegeben, soll das Programm den Nutzer über den korrekten Aufruf informieren.

Temperaturen in Grad Celsius und Fahrenheit umwandeln

Es ist ein Programm zu schreiben, das eine Temperatur in Grad Celsius (C) in Grad Fahrenheit (F) umrechnet. Das Programm soll außerdem in der Lage sein, eine Temperatur von Grad Fahrenheit (F) in Grad Celsius (C) umzuwandeln.

Programmaufruf: `java TemperatureConverter temperatur`. Beispiel: `java TemperatureConverter 20.5C` gibt `68.9F` aus. Beispiel 2: `java TemperatureConverter 68.9F` gibt `20.5C` aus.

Auf Teilbarkeit prüfen

Schreibe ein Programm, das prüft, ob eine ganze Zahl `z` durch eine andere ganze Zahl `t` teilbar ist. Es darf davon ausgegangen werden, dass der Nutzer nur positive Zahlen angibt.

Programmaufruf: `java MyApp z t`. Beispiel: `java MyApp 100 25` gibt `true` aus und `java MyApp 100 30` gibt `false` aus.

Im Anschluss ist das Program so zu erweitern, dass es eine Gleichung mit Teiler und ggf. auch mit Restwert (Modulo) ausgibt. Beispiel: `100 = 4 * 25` ist für `z = 100` und `t = 25` auszugeben. Bei `z = 100` und `t = 30` erscheint hingegen `100 = 3 * 30 + 10`.

Tipp: Verwende den Modulo-Operator `%` und die Integer-Division `/`.

Zeichenkette umkehren

Eine Zeichenkette `s` ist in umgekehrter Reihenfolge auszugeben.

Das Programm ist wie folgt aufzurufen: `java MyApp s`. Beispiel: `java MyApp "Hello World"` gibt `dlroW olleH` aus.

Tipp: Verwende eine herkömmliche for-Schleife.

Römische Ziffer in Dezimalzahl umwandeln

Römische Zahlen bestehen aus den Ziffern `i` (1), `v` (5), `x` (10), `l` (50), `c` (100), `d` (500) und `m` (1000). Es soll ein Programm geschrieben werden, das eine römische Ziffer in eine Dezimalzahl umwandelt.

Programmaufruf: `java MyApp ziffer`. Beispiel: `java MyApp v` gibt `5` aus und `java MyApp c` gibt 100 aus. Der Nutzer darf die Ziffer als Groß- oder Kleinbuchstabe angeben.

Das Programm soll keine vollständigen Römischen Zahlen wie `mcxx` in Dezimalzahlen umwandeln. Es geht hier nur um eine Ziffer wie `c` oder `m`. Siehe Zusatzaufgabe.

Vorgabe für Implementierung: Verwende eine moderne switch-Anweisung / switch-Expression.

Zusatzaufgabe 1

Erweitere das Program so, dass vollständige römische Zahlen übersetzt werden. Beispiel: `java MyApp mcxx` gibt `1120` aus. Hier zu sind folgende Regeln zu berücksichtigen:

- Steht eine kleinere Ziffer links von einer größeren Ziffer, so ist die kleinere Ziffer von der größeren abzuziehen. Andernfalls werden die Ziffern aufaddiert. Beispiel: `iv` ist 4 und `xc` ist 90. `cv` ist 105.
- Dieselbe Ziffer darf nicht mehr als dreimal nacheinander erscheinen. Beispiel: `iii` ist korrekt, aber `iiii` nicht, da die 4 als `iv` ausgedrückt wird.
- Folgende Ziffern dürfen grundsätzlich nicht wiederholt werden: `v`, `l` und `d`.
- Folgende Subtraktionen sind erlaubt: `iv` und `ix`, `xl` und `xc`, `cd` und `cm`.
- Die größtmögliche Zahl ist 3999, also `mmmcmxcix` (3000 + 900 + 90 + 9).

Zusatzaufgabe 2

Schreibe ein Programm, das eine Dezimalzahl in eine römische Zahl umwandelt. Es gelten dieselben Regeln wie in Zusatzaufgabe 1.

Schaltjahr bestimmen

Schreibe ein Programm, das bestimmt, ob ein eingegebenes Jahr ein Schaltjahr ist. Ein Jahr ist ein Schaltjahr, wenn es durch 4 teilbar ist. Falls das Jahr aber auch durch 100 teilbar ist, handelt es sich um kein Schaltjahr. Wenn das Jahr jedoch durch 400 teilbar ist, dann ist es trotzdem ein Schaltjahr.

Das Programm ist so aufzurufen: `java MyApp jahr`. Beispiel: `java MyApp 2024` gibt `true` aus. `java MyApp 1900` gibt `false` aus. `java MyApp 1600` gibt `true` aus.

Implementiere das Programm in zwei Varianten:

- Verwende `if` in Kombination mit `else if` und `else`.
- Verwende nur `if` und verknüpfe die Bedingungen mit `&&` und `||`.

Tipp: Benutze den Modulo-Operator `%` um die Teilbarkeit zu prüfen.

Schaltjahre in einem Bereich bestimmen

Es ist ein Programm zu schreiben, das alle Schaltjahre ausgibt, die sich in einem vorgegebenen Intervall befinden. Das Intervall wird durch Angabe eines Start- und eines Endjahres definiert.

Programmaufruf: `java MyApp start ende`. Beispiel: `java MyApp 1896 1915` gibt die Jahre 1896, 1904, 1908 und 1912 aus.

Tage eines Monats bestimmen

Es sind die Tage eines Monats auszugeben. Schaltjahre sind dabei zu berücksichtigen. Der Monat ist als Zahl anzugeben.

Programmaufruf: `java MyApp monat jahr`. Beispiel: `java MyApp 2 2024` gibt 29 aus und `java MyApp 2 2025` gibt 28 aus.

Vorgabe für Implementierung: Verwende ein modernes switch-Statement / switch-Expression.

Palindrom-Test

Es ist zu prüfen, ob eine Zeichenkette `s` ein Palindrom ist. Palindrome sind Wörter, die sowohl vorwärts als auch rückwärts gelesen dasselbe Wort ergeben. Bildlich gesprochen handelt es sich um Wörter, die gespiegelt sind. Beispiele: otto, anna, hannah, rentner, lagerregal.

Das Programm ist folgendermaßen aufzurufen: `java MyApp s`. Beispiel: `java MyApp "rentner"` gibt `true` und `java MyApp "java"` den Text `false` aus.

Vorgabe zur Implementierung: Verwende eine for-Schleife mit zwei Laufvariablen.

Zeichenketten vermischen

Zwei Zeichenketten `a` und `b` sollen nach dem Reißverschluss-Verfahren gemischt werden.

Das Programm ist wie folgt aufzurufen: `java MyApp a b`. Beispiel: `java MyApp "abc" "123"` gibt `a1b2c3` aus.

Falls sich die Längen von `a` und `b` unterscheiden, sind die verbleibenden Zeichen der längeren Zeichenkette am Ende auszugeben. Beispiel: `java MyApp "abcde" "123"` gibt `a1b2c3de` aus.

Tipp: Nutze eine herkömmliche for-Schleife.

Zusatzaufgabe

Erweitere das Programm so, dass es beliebig viele Zeichenketten miteinander mischt. Beispiel: `java MyApp "abc" "1234" "xy" "-+:"` gibt `a1x-b2y+c3:4` aus.

Zeichenketten auffüllen

Eine Zeichenkette `s`, die weniger als `n` Zeichen enthält, soll mit einem Füllzeichen `f` bis zur Länge `n` von rechts aufgefüllt werden.

Das Programm ist wie folgt aufzurufen: `java MyApp s n f`. Beispiel: `java MyApp "a b" 5 X` gibt `a bXX` aus.

Falls `s` schon lang genug ist, soll das Programm `s` unverändert ausgeben. Beispiel: `java MyApp "abcde" 3 X` gibt `abcde` aus.

Im Anschluss ist das obige Programm in zwei weiteren Varianten zu implementieren:

- Die Zeichenkette ist von links aufzufüllen.
- Die Zeichenkette ist beidseitig aufzufüllen. Beispiel: `java "abc" 7 X` gibt `XXabcXX` aus und `java MyApp "abc" 6 X` gibt `XXabcX` aus.

Tipp: Verwende `Integer.parseInt` zum Konvertieren von `String` nach `int`. Verwende den `+` Operator zum Verketten von Strings.

Zeichenketten abkürzen

Eine Zeichenkette `s`, die mehr als `n` Zeichen enthält, soll durch eine Ellipsis `...` (3 Punkte) so abgekürzt werden, dass sie genau die Länge `n` hat. Hat `s` höchstens `n` Zeichen, so ist `s` unverändert auszugeben.

Das Programm ist wie folgt aufzurufen: `java MyApp s n`. Beispiel: `java MyApp "Ein langer Text" 10` gibt `Ein lan...` aus.

Wenn `n <= 3` gilt und die Länge von s größer als `n` ist, sind `n` Punkte (`.`) auszugeben. Beispiel: `java MyApp "abcde" 3` gibt `...` aus und `java MyApp "abcde" 2` gibt `..` aus.

Im Anschluss ist das obige Programm in zwei weiteren Varianten zu implementieren:

- Die Zeichenkette ist am Beginn zu kürzen. Beispiel: `java MyApp "Ein langer Text" 10` gibt `...er Text` aus.
- Die Zeichenkette ist in der Mitte zu kürzen. Beispiel: `java MyApp "Ein langer Text" 10` gibt `Ein...Text` aus.

Tipp: Verwende die Instanzmethode `substring` der Klasse `String` und den `+` Operator zum Verketten. Du kannst auch einen `StringBuilder` verwenden.

Umwandlung von Tagen, Stunden, Minuten und Sekunden in Sekunden

Eine Dauer, die in Form von Tagen, Stunden, Minuten und Sekunden anzugeben ist, soll in Sekunden umgerechnet werden. Beispiel: 1 Tag, 2 Stunden, 5 Minuten und 20 Sekunden ergeben insgesamt 93920 Sekunden.

Das Programm ist wie folgt aufzurufen: `java MyApp 1d 2h 5m 20s`. Die Argumente dürfen vom Nutzer in *beliebiger* Reihenfolge angegeben werden! Außerdem soll es möglich sein, Argumente wegzulassen. Beispiel: `java MyApp 20s 1d`. Das Programm erkennt anhand der Suffixe `d` (Tage), `h` (Stunden), `m` (Minuten) und `s` (Sekunden), um welche Zeiteinheiten es sich handelt. Es darf davon ausgegangen werden, dass der Nutzer Argumente im korrekten Format angibt.

Taucht eine Zeiteinheit mehrfach auf, sind die Einheiten zu addieren. Beispiel: `java MyApp 20s 1d 60s` ist gleichbedeutend mit `java MyApp 80s 1d`. Steht vor der Zeiteinheit ein `-` (Minus), so sind die Zeiteinheiten zu subtrahieren. Beispiel: `java MyApp 1d 2h 5m 20s -2m` ergibt 93800 Sekunden.

Erhält das Programm gar keine Argumente, ist ein Text auszugeben, der den Nutzer darüber informiert, wie man das Programm korrekt aufruft - eine Synopsis.

Tipp: Du kannst hier Methoden der Klasse `String` verwenden, z.B. `endsWith`. Für die Umwandlung von Strings in Integer, steht dir die statische Methode `parseInt` der Klasse `Integer` zur Verfügung.

Umwandlung von Sekunden in Tage, Stunden, Minuten und Sekunden

Eine ganze Zahl soll in Tage, Stunden, Minuten und Sekunden zerlegt werden. Beispiel: 93920 Sekunden entsprechen 1 Tag, 2 Stunden, 5 Minuten und 20 Sekunden.

Tipp: Verwende den Modulo-Operator `%` und die Integer-Division `/`.

Arbeitsdauer aus Komm- und Gehzeit ermitteln

Anhand zweier Uhrzeiten im Format `hh:mm` soll die Arbeitszeit in Stunden und Minuten berechnet werden.

Das Programm ist wie folgt aufzurufen: `java MyApp hh:mm hh:mm`. Das erste Argument ist die Komm- und das zweite Argument die Gehzeit. Beispiel: `java MyApp 08:45 12:30` gibt `3h 45m` aus. Es darf davon ausgegangen werden, dass der Nutzer Argumente im korrekten Format angibt.

Das Program muss auch mit Tageswechseln umgehen können. Beispiel: `java MyApp 21:20 02:03` gibt `4h 43m` aus.

Wenn weniger als zwei Argumente angegeben werden, ist der Nutzer über die korrekte Verwendung des Programms zu informieren.

Tipp: Verwende `Integer.parseInt` zum Parsen. Die Uhrzeiten lassen sich zum Beispiel mit der Instanzmethode `split` der Klasse `String` zerlegen.

Teil C

Fakultät berechnen

Es ist ein Programm zu schreiben, dass die Fakultät einer positiven Zahl `n >= 0` berechnet. Die Fakultät einer Zahl `n` wird `n!` geschrieben und bedeutet `1 * 2 * ... * (n-1) * n`. Per Definition ist `0! = 1`.

Aufruf: `app n`. Beispiel: `app 4` gibt 24 aus.

Implementiere das Programm in zwei Varianten:

- Die erste Variante verwendet einen iterativen Algorithmus.
- Die zweite Variante verwendet einen rekursiven Algorithmus.

Fibonacci-Folge berechnen

Die Fibonacci-Folge `fib` ist eine rasant wachsende Zahlenfolge. Es gilt: `fib(0) = 0`, `fib(1) = 1` und `fib(2) = 1`. Für `n >= 3` gilt `fib(n) = fib(n-1) + fib(n-2)`.

Es ist ein Programm zu schreiben, dass die Zahlen `fib(0)`, `fib(1)` bis `fib(n)` auf der Kommandozeile ausgibt.

Aufruf: `Fibonacci n`. Beispiel: `Fibonacci 6` gibt `0, 1, 1, 2, 3, 5, 8` aus.

Implementiere das Programm in zwei Varianten:

- Die erste Variante soll einen iterativen Algorithmus verwenden.

- Die zweite Variante soll einen rekursiven Algorithmus benutzen.

Arrays konkatenieren (aneinanderhängen)

Schreibe ein Programm, das einen zusammengesetzten Array aus zwei vorgegebenen Arrays erzeugt und dann ausgibt.

Aufruf: `app array-b / array-c`. Beispiel: `app 1 2 3 / u v w x` erzeugt intern die beiden Arrays `["1","2","3"]` und `["u","v","w","x"]`. Das Programm fügt dann beide Arrays zu einem neuen Array mit den Elementen `["1","2","3","u","v","w","x"]` zusammen. Am Schluss erfolgt die Ausgabe.

Tipp: Erstelle dir ein neues Array mit passender Länge und kopiere die Elemente beider Arrays mit einer Schleife.

Arrays in Arrays einfügen

Schreibe ein Programm, das einen Array `a` in einen Array `b` an Position `index` "einfügt". Da Arrays eine feste Länge besitzten, ist ein neuer Array zu erstellen, der alle Elemente von `b` inklusive aller Elemente von `a` enthält.

Aufruf: `app array-a / array-b / index`. Beispiel: `app 1 2 3 / a b c d / 2` erzeugt intern den Array `a b 1 2 3 c d` und gibt ihn anschließend aus. Beispiel 2: `app 1 2 3 / a b c / 0` gibt `1 2 3 a b c` aus und `app 1 2 3 / a b c / 3` gibt `a b c 1 2 3` aus.

Tipp: Zum Einfügen sind herkömmliche Schleifen ausreichend. Du benötigst keine besonderen Array Funktionen.

Minimum, Maximum, Durchschnitt und Median ermitteln

Ein Programm soll aus einer Zahlenreihe das Minimum, das Maximum, den Durchschnitt sowie den Median ermitteln. Sofern die Zahlenreihe ungeräde Länge hat, ist der Median als Durchschnitt der beiden mittlersten Elemente der sortierten Zahlenreihe zu berechnen.

Aufruf: `app number...`. Beispiel: `app 2 5 1 2 8 9 9 9` gibt aus: `Min: 1 Max: 9 Avg: 5.625 Med: 6.5`. Der Median berechnet sich in diesem Fall aus dem arithmetischen Mittel der Zahlen 5 und 8, da diese in der sortierten Reihenfolge die beiden mittleren Elemente sind. Beispiel 2: `app 4 4 2` gibt aus: `Min: 2 Max: 4 Avg: 3.333333 Med: 4`. In diesem Fall ist der Median das mittlere Element der sortierten Zahlenfolge.

Unikate in einem Array finden

Schreibe ein Programm, dass die Unikate seiner Kommandozeilenargumente ermittelt. Die Unikate sind aufsteigend sortiert auszugeben.

Aufruf: `app arguments...`. Beispiel: `app 2 1 1 3 1 5 9 a B` gibt `1 2 3 5 9 B a` aus. Strings werden lexikographisch geordnet. Da die Code-Points der Dezimalziffern kleiner als die Code-Points der Buchstaben sind, erscheinen diese zuerst. Die Code-Points der Großbuchstaben sind wiederum kleiner als die der Kleinbuchstaben.

Das Programm ist in zwei Varianten zu implementieren:

- In der ersten Variante sollen die Kommandozeilenargumente aufsteigend sortiert werden. Die Suche nach Unikaten verwendet die sortierten Argumente.
- In der zweiten Variante sollen die Kommandozeilenargumente unberührt bleiben. Die Suche nach Unikaten verwendet die unsortierten Argumente.

Tipps:

- Verwende die Klasse `java.util.Arrays` zum Sortieren (`sort`) und Kürzen (`copyOf` und `copyOfRange`) von Arrays.
- Arrays haben eine feste Länge und können weder wachsen noch schrumpfen. Lege dir für die zu findenden Unikate ein Array an, dass theoretisch alle Elemente aufnehmen kann. Kürze am Ende der Suche, falls notwendig.

Wiederholungen eines Zeichens in einer Zeichenkette finden

Es ist ein Programm zu schreiben, dass in einer Zeichenkette `s` nach `n` aufeinanderfolgenden Zeichen `c` sucht. Das Programm beginnt die Suche von links und gibt den Index der ersten gefundenen Sequenz aus. Sofern das Programm keine Sequenz findet, informiert es den Nutzer darüber.

Aufruf: `app s c n`. Beispiel: `app abbcaaadee a 2` gibt `4` aus, da hier die erste Sequenz von `a` mit mindestens 2 Wiederholungen erscheint.

Vorgabe zur Implementierung:

- Verzichte auf die Verwendung von `substring` und nutze stattdessen zwei ineinander geschachtelte Schleifen.

Texte mit der Cäsar-Chiffre ver- und entschlüsseln

Funktionsweise der Chiffre

Die Cäsar-Chiffre ist ein sehr einfaches, symmetrisches Verschlüsselungsverfahren. Symmetrisch bedeutet, dass für die Ver- und Entschlüsselung derselbe Schlüssel genutzt wird. Die Cäsar-Chiffre verwendet zum Chiffrieren zwei Alphabete: das Klartextalphabet (plain text alphabet) und das Geheimtextalphabet (cipher text alphabet). Das Geheimalphabet wird gebildet, indem das Klartextalphabet um `shift` Positionen nach links bzw. rechts geschoben bzw. rotiert wird. Beispiel:

```
Plain-Text Alphabet  : a b c d e f g

Cipher-Text Alphabet : d e f g a b c  Shift: -3 (links)
Cipher-Text Alphabet : c b a g f e d  Shift: -3 (links) und reverse
Cipher-Text Alphabet : e f g a b c d  Shift: +3 (rechts)
Cipher-Text Alphabet : d c b a g f e  Shift: +3 (rechts) und reverse

Cipher-Text Alphabet : a b c d e f g  Shift: +7 (rechts)
Cipher-Text Alphabet : a b c d e f g  Shift: +0 (rechts)
Cipher-Text Alphabet : g a b c d e f  Shift: +8 (rechts)
Cipher-Text Alphabet : g a b c d e f  Shift: +1 (rechts)
```

Oben sind ein paar Verschiebungen des Klartextalphabets zu sehen. Ist der Shift negativ, wird das Klartextalphabet nach links verschoben bzw. rotiert. Ist der Shift hingegen positiv, wird entsprechend nach rechts verschoben bzw. rotiert. Bei der Cäsar-Chiffre kann das Geheimalphabet anschließend noch umgekehrt werden, sofern das erwünscht ist (reverse).

Das obige Klartextalphabet besteht aus 7 Buchstaben. Würde man um 7 Positionen nach links oder rechts verschieben, erhielte man wieder das Klartextalphabet. Eine Verschiebung um +8 ist identisch zu einer Verschiebung um +1. Analog ist eine Verschiebung um -8 Positionen identisch zu einer Verschiebung um -1.

Die Cäsar-Chiffre verschlüsselt einen Klartext, indem sie jedes Zeichen des Klartextes durch das korrespondierende Zeichen des Geheimalphabets ersetzt. Beispiel:

```
Plain-Text Alphabet  : a b c d e f g h i j k l m n o p q r s t u v w x y z
Cipher-Text Alphabet : x y z a b c d e f g h i j k l m n o p q r s t u v w

Plain-Text  : java ist toll
Cipher-Text : gxsx fpq qlii
```

Im obigen Beispiel wird das Klartextzeichen `j` durch das Geheimtextzeichen `g` ersetzt. Das Zeichen `a` durch `x` und `v` durch `s`. Hier erkennt man schon eine deutliche Schwachstelle des Verfahrens: gleiche Klartextzeichen werden immer auf gleiche Geheimtextzeichen abgebildet.

Die Entschlüsselung funktioniert genauso, nur dass man die Zeichen des Geheimtextes auf die korrespondierenden Klartextzeichen abbildet.

Für eine erfolgreiche Ver- und Entschlüsselung muss derselbe Schlüssel verwendet werden. Bei der Cäsar-Chiffre ist der Schlüssel der normalisierte Shift des Klartextalphabets. Mit normalisiert ist gemeint, dass der Shift-Wert auf den Bereich `[0, alphabet.length - 1]` abgebildet wird. Beispiel: Hat das Alphabet eine Länge von 26, dann wird der Shift-Wert `-1` auf den Wert `25` abgebildet, denn eine Verschiebung um eine Stelle nach links entspricht einer Verschiebung um 25 Stellen nach rechts. Analog würde der Wert `27` auf den Wert `1` abgebildet, denn eine Verschiebung um 27 Stellen nach rechts, entspricht effektiv einer Verschiebung um eine Stelle nach rechts.

Aufgabenstellung

Schreibe ein Programm, das Texte mit der Cäsar-Chiffre ver- und entschlüsselt. Der Anwender muss dazu das Kommando (encrypt, decrypt), den Text sowie den Schlüssel (Shift-Wert) angeben. Zusätzlich soll die Angabe des Arguments `--reverse` möglich sein. Ist `--reverse` angegeben, muss das gebildete Geheimtextalphabet vor der Ver- bzw. Entschlüsselung umgekehrt (reverse) werden.

Aufruf: `app <command> <text> <shift> [--reverse]`.

Beispielaufrufe: Hier gilt die Annahme, dass das Klartextalphabet aus den Buchstaben `a-z` besteht.

- `app encrypt "java is awesome" 3` liefert `gxsx fp xtbpljb`.
- `app encrypt "java is awesome" -5` liefert `ofaf nx fbjxtrj`.
- `app encrypt "java is awesome" -5 --reverse` liefert `veje wm eiamqsa`.
- `app decrypt "ofaf nx fbjxtrj" -5` liefert `java is awesome`.

Wenn zu wenig Argumente angegeben werden, ist folgender Text auszugeben:

```
Usage: CaesarCipher <command> <text> <shift> [--reverse]

Options:
  --reverse    Reverses the cipher alphabet.

Commands:
  encrypt      Encrypts the plain text with the given shift.
  decrypt      Decrypts the cipher text with the given shift.
```

Speichere den obigen Text händisch in einer Datei namens `help.txt` ab. Diese Datei ist in einem Unterordner namens `docs` abzulegen. Bei Programmbegin ist zu prüfen, ob genügend Argumente angegeben wurden. Ist das nicht der Fall, soll `help.txt` geladen und angezeigt werden. Die dafür notwendige Logik ist in einer Methode namens `printHelp` zu implementieren. Falls `help.txt` nicht lesbar ist, soll ein Hinweis auf den Error Stream geschrieben werden. Fange auf jeden Fall die Exception `IOException` ab.

Lege dir eine String-Konstante (statisches finales Klassenfeld) namens `ALPHABET` an, die alle Zeichen enthält, die von der Chiffre unterstützt werden. Hier sollten mindestens alle Groß- und Kleinbuchstaben enthalten sein, sowie alle Dezimalziffern `0-9`.

Schreibe nun eine Methode namens `isValidText(text)`, die prüft, ob ein Text nur aus Zeichen besteht, die in `ALPHABET` enthalten sind. Leerzeichen sind zu ignorieren. Beispiel: `isValidText("Java is cool")` liefert `true`, aber `isValidText("C# is godlike")` liefert `false`, da `#` nicht in `ALPHABET` enthalten ist.

Schreibe eine Methode namens `normalizeShift(value, length)`, die einen Shift-Wert normalisiert. Beispiel: `normalizeShift(-3, 26)` liefert 23, da eine Verschiebung um 3 Stellen nach links identisch zu einer Verschiebung um 23 Stellen nach rechts ist, wenn das Alphabet eine Länge von 26 hat. `normalizeShift(-29, 26)` liefert ebenfalls `23`. `normalizeShift(1, 26)` liefert `1` und `normalizeShift(27)` auch. Tipp: Verwende den Modulo Operator `%`.

Implementiere eine Methode `shift(s, n)`, die einen String um `n` Positionen verschiebt bzw. rotiert. Ist `n < 0` soll nach links verschoben werden, andernfalls nach rechts. Beispiel: `shift("abcd", -2)` liefert `cdab`. `shift("abcd", 3)` liefert `bcda`. Verwende die `normalizeShift` Methode, um den Shift-Wert zu normalisieren.

Implementiere eine Methode `map(s, source, target)`, die die Zeichen eines Strings `s` aus dem Alphabet `source` in korrespondierende Zeichen eines anderen Alphabets `target` transformiert. Die Alphabete `source` und `target` müssen dieselbe Länge besitzen, andernfalls soll eine `IllegalArgumentException` geworfen werden. Beispiel:

```
String source = "abcdefg";
String target = "+-*!XYZ";

map("cafebabe", source, target) liefert "*+YX-+-X"
```

Schreibe eine Methode namens `encrypt(text, alphabet, shift, reverse)`, die einen Text mit der Cäsar-Chiffre verschlüsselt. Beispiel: `encrypt("java is awesome", ALPHABET, -5, false)` liefert `ofaf nx fbjxtrj`, sofern `ALPHABET` aus den Zeichen `a-z` besteht. Der Aufruf `encrypt("java is awesome", ALPHABET, -5, true)` liefert hingegen `veje wm eiamqsa`. Verwende für die Implementierung die zuvor entwickelten Methoden `map` und `shift`.

Schreibe analog zu `encrypt` eine Methode namens `decrypt(text, alphabet, shift, reverse)`, die den `text` entschlüsselt. Auch hier soll für die Implementierung `map` und `shift` verwendet werden.

Schreibe das Hauptprogramm `main` unter Verwendung der bisher entwickelten Methoden.

Permutationen ohne Wiederholung

Eine Permutation ohne Wiederholung ist eine Anordnung von `n` unterschiedlichen Elementen. Beispiel: `4, 1, 3, 2` und `3, 4, 2, 1` sind Permutationen der Menge `{ 1, 2, 3, 4 }`. Jede Änderung der Reihenfolge ergibt eine neue Permutation.

Wenn eine Menge aus `n` Elementen besteht, sind insgesamt `n! = 1 * 2 * ... * (n-1) * n` Permutationen möglich.

Schreibe ein Programm, dass alle Permutationen der Zahlen `1, 2, ..., n` ermittelt. `n` ist von der Kommandozeile einzulesen.

Implementiere das Programm in zwei Varianten:

- Die erste Variante soll einen iterativen Algorithmus verwenden.
- Die zweite Variante soll einen rekursiven Algorithmus verwenden.