ÜBUNG 3

24WS-UE03-A01 Taschenrechner mit Methoden

Verbessern Sie mit Methoden die Struktur des Taschenrechner-Programms aus der letzten Übung.

Beispieldialog:

```
Easy Calculator

Number: 4
Operator: #
Operator ( + - * / ): +
Number: 6
4.0 + 6.0 = 10.0

Continue (y/n)? y

Number: 7
Operator: *
Number: 2.5
7.0 * 2.5 = 17.5

Continue (y/n)? n

bye bye!
```

Verwenden Sie folgende Struktur für die Methode main und implementieren Sie die Methoden readNumber, readOperator, calculate und userWantsToQuit mit den Keywords public static:

```
public static void main(String[] args) {
    Out.println("Easy Calculator");
    boolean isQuit = false;
    while (!isQuit) {
        Out.println();
        double n1 = readNumber();
        char operator = readOperator();
        double n2 = readNumber();
        double result = calculate(n1,n2,operator);
        Out.println(n1 + " " + operator + " " + n2 + " = " + result);
        Out.println();
        isQuit = userWantsToQuit();
    }
    Out.println();
    Out.println("bye bye!");
}
```

Achten Sie auf die jeweiligen Rückgabewerte und stellen Sie sicher, dass readOperator nur gültige Operatoren liefert.

24WS-UE03-A02 Baeckerei

Schreiben Sie ein Java-Programm, das den Preis von Produkten einer Bäckerei berechnen soll. Dazu soll der Verkäufer eingeben, um welches Produkt es sich handelt, wieviel davon gekauft wird und wieviel es kostet.
Ab 5 Stück gibt es 10% Rabatt, ab 10 Stück 15% Rabatt und ab 15 Stück 20% Rabatt. Gibt es einen Rabatt, so soll dieser extra ausgewiesen werden, ebenso der reguläre sowie der reduzierte Preis, siehe Beispieldialog.
Es können mehrere Produkte eingegeben werden. Am Ende soll auch der Gesamtpreis ausgewiesen werden.

Schreiben Sie dazu die main-Methode und die darin aufgerufenen static-Methoden readProduct, readAmount, readPricePerUnit, printReceipt, calculatePrice sowie userWantsToQuit (vgl. Aufgabe 23WS-UE03-A01):

```
public static void main(String[] args) {
   Out.println("Bakery");
   Out.println(" (discounts: 5..9: 10%, 10..14: 15%, 15..: 20%");
   boolean isQuit = false;
   double totalPrice = 0.0;
   while (!isQuit) {
      String product= readProduct();
      int amount = readAmount();
      double pricePerUnit = readPricePerUnit();
      printReceipt(amount, product, pricePerUnit);
      totalPrice+= calculatePrice(amount,pricePerUnit);
      isQuit= userWantsToQuit();
   }
   Out.println("Total price: " + String.format("%.2f",totalPrice) + " Euro");
}
```

Beispieldialog

```
Bakery
 (discounts: 5..9: 10%, 10..14: 15%, 15..: 20%)
Product: Bread
Amount: 0
Amount > 0: 2
Price per unit: 4.6
Bread: 2 x 4.60 = 9.20 Euro
Continue (y/n)? y
Product: Milk
Amount: 5
Price per unit: 1.59
Milk: 5 x 1.59 = 7.95 Euro
 Discount 10% = 0.80
 Total: 7.16 Euro
Continue (y/n)? y
Product: Pretzel stick
Amount: 12
Price per unit: 1.1
pretzel stick: 12 x 1.10 = 13.20 Euro
 Discount 15% = 1.98
 Total: 11.22 Euro
Continue (y/n)? y
Product: Roll
Amount: 20
```

```
Price per unit: 0.45
roll: 20 x 0.45 = 9.00 Euro
 Discount 20% = 1.80
 Total: 7.20 Euro
Continue (y/n)? n
Total price: 34.78 Euro
```

24WS-UE03-A03 Zahlen mit geraden oder ungeraden Ziffern

Schreiben Sie zwei Funktionen `isFullyOdd` und `isFullyEven`, die feststellen, ob eine ganze Zahl ausschließlich aus ungeraden bzw. geraden Ziffern besteht. Zahlen, die nur aus einer Ziffer bestehen, sollen dabei nicht berücksichtigt werden, d.h. weder als `isFullyOdd` noch als `isFullyEven` gelten.

Lesen Sie zwei Werte `low` und `high` vom Benutzer ein. Wenn der Wert von `low` größer ist als der von `high`, soll eine Fehlermeldung ausgegeben werden. Testen Sie alle ganzen Zahlen zwischen den Werten `low` und `high` (jeweils inklusive) und geben Sie jene aus, bei denen jede Ziffer ungerade bzw. gerade ist. Rufen Sie dazu die realisierten Funktionen `isFullyOdd` und `isFullEven` auf. Geben Sie weiters aus, für wie viele Zahlen jeweils die Eigenschaft `isFullyOdd` und `isFullEven` zutrifft, siehe Beispieldialog.

Beispiel:

`333` ist eine ungerade Zahl und sie besteht ausschließlich aus ungeraden Ziffern. Bei der Zahl `383` hingegen sind nicht alle Ziffern ungerade. Die Zahl `3` ist ungerade und besteht aus einer ungeraden Ziffer. Laut unserer Definition gilt sie aber nicht als fully odd.

Beispieldialog:

```
Even/odd digits of numbers!
Lower bound: -20
Upper bound: 20
Fully odd numbers:
-19, -17, -15, -13, -11, 11, 13, 15, 17, 19.
A total of 10 numbers is fully odd.
Fully even numbers:
-20, 20.
A total of 2 numbers is fully even.
Even/odd digits of numbers!
Lower bound: -100
Upper bound: -50
Fully odd numbers:
-99, -97, -95, -93, -91, -79, -77, -75, -73, -71, -59, -57, -55, -53, -51.
A total of 15 numbers is fully odd.
Fully even numbers:
-88, -86, -84, -82, -80, -68, -66, -64, -62, -60.
A total of 10 numbers is fully even.
Even/odd digits of numbers!
Lower bound: 100
Upper bound: 50
Lower bound is bigger than upper bound!
Even/odd digits of numbers!
Lower bound: 0
Upper bound: 0
Fully odd numbers:
.
A total of 0 numbers is fully odd.
Fully even numbers:
.
A total of 0 numbers is fully even.
```

24WS-UE03-A04 Zahlen mit aufsteigenden absteigenden gleichen und alternierenden

Schreiben Sie eine Funktion `static boolean hasIncreasingDigits(int number)`, die feststellt, ob die Ziffern einer positiven oder negativen Zahl `number` aufsteigend sind.
Sind die Ziffern der Zahl aufsteigend, soll die Funktion `true` als Ergebnis liefern, ansonsten `false`. Eine Funktion `hasDecreasingDigits` stellt fest, ob die Ziffern absteigend sind.

Beispiel: Die Zahlen 124458 und -124458 bestehen aus aufsteigenden Ziffern, nicht jedoch die Zahl 354. Sind alle Ziffern einer Zahl gleich, z.B. 99 oder 888, dann gilt sie sowohl als aufsteigend als auch als absteigend.

Über eine Funktion `hasSameDigits` soll eruiert werden, ob eine Zahl nur aus gleichen, aber zumindest zwei Ziffern besteht. (Hinweis: Diese Zahlen gelten sowohl als absteigend als auch als aufsteigend sortiert.)

Eine Funktion `hasAlternatingDigits` soll schließlich feststellen, ob eine Zahl alternierend auf-/absteigende Ziffern hat. Sind zwei benachbarte Ziffern gleich, dann soll eine Zahl nicht als alternierend gelten, siehe Beispieldialog. Eine alternierende Zahl muss aus mindestens drei Ziffern bestehen, auf-/absteigende sowie gleiche Zahlen aus mindestens zwei Ziffern.

Überprüfen Sie die Zahlen in einem bestimmten Zahlenbereich, z.B. -100 .. 100, und geben sie das Ergebnis gemäß dem angegebenen Beispiel aus.

Hinweis: Mit dem bereits bekannten `String.format("%,5d", number)` können die Zahlen untereinander ausgegeben werden. Die Zahl 5 gibt die Anzahl der Stellen für die Ausgabe an. Beginnen Sie jeweils nach 10 Zahlen eine neue Zeile.

Beispieldialog:

```
Numbers with ascending/descending/same/alternating digits!
Range: 1000..1200
Ascending:
1111, 1112, 1113, 1114, 1115, 1116, 1117, 1118, 1119, 1122,
1123, 1124, 1125, 1126, 1127, 1128, 1129, 1133, 1134, 1135,
1136, 1137, 1138, 1139, 1144, 1145, 1146, 1147, 1148, 1149,
1155, 1156, 1157, 1158, 1159, 1166, 1167, 1168, 1169, 1177,
1178, 1179, 1188, 1189, 1199.
Descending:
1000, 1100, 1110, 1111.
Same digits:
1111.
Alternating:
1010, 1020, 1021, 1030, 1031, 1032, 1040, 1041, 1042, 1043,
1050, 1051, 1052, 1053, 1054, 1060, 1061, 1062, 1063, 1064,
1065, 1070, 1071, 1072, 1073, 1074, 1075, 1076, 1080, 1081,
1082, 1083, 1084, 1085, 1086, 1087, 1090, 1091, 1092, 1093,
1094, 1095, 1096, 1097, 1098.
Numbers with ascending/descending/same/alternating digits!
Range: -20..20
Ascending:
 -19, -18, -17, -16, -15, -14, -13, -12, -11, 11,
 12, 13, 14, 15, 16, 17, 18, 19.
Descending:
 -20, -11, -10, 10, 11, 20.
Same digits:
 -11, 11.
```

```
Alternating:
.
```

24WS-UE03-A05 Einfaches Banking

Implementieren Sie ein Java-Programm, das einen Kontostand verwaltet. Es können Beträge eingezahlt und abgehoben werden. Erstellen Sie ein Menü, bei dem der Benutzer aus mehreren Operationen auswählen kann (+ … einzahlen, - … abheben, b … Kontostand abfragen). Stellen Sie jeweils sicher, dass der Benutzer einen gültigen Wert eingibt. Negative Zahlen sind bei der Eingabe nicht erlaubt, siehe Beispieldialog. Verwenden Sie eine globale Variable für den Kontostand und verwalten Sie diesen mit den Methoden deposit, withdraw und getBalance.

Verwenden Sie folgende Methoden:

- public static void deposit(int n): Der Betrag n wird auf das Konto eingezahlt.
- public static void withdraw(int n): Der Betrag n wird vom Konto abgehoben.
- public static int getBalance(): liefert den aktuellen Kontostand.
- public static int readNumber(): liest eine Zahl vom Benutzer ein.
- public static char readSelection(): liest eine gültige Operation vom Benutzer ein (+ - b q).
- public static void printBalance(): gibt den Kontostand auf der Konsole aus.

Beispieldialog

```
Simple Banking
+ .... Deposit
- .... Withdrawal
b .... Balance
q .... Quit

Action: +
Amount: 5000
Balance: € 5,000

Action: +
Amount: -50
Amount > 0: 50
Balance: € 5,050

Action: *
Action (+ - b q): -
Amount: 2000
Balance: € 3,050

Action: +
Amount: 999999
Balance: € 1,003,049

Action: q
Bye bye!
```

24WS-UE03-A06 TicTacToe mit Methoden

Lesen Sie den gesamten nachfolgenden Text bevor Sie mit Ihrer Lösung starten!

Schreiben sie ein Java-Programm, welches ein TicTacToe-Spiel repräsentiert.

Es soll ausgegeben werden, welcher Spieler*in an der Reihe ist ("Player X" oder "Player O"). Anschließend sollen die Koordinaten von der Spieler*in eingegeben werden. Diese geben an, wo der Kreis oder das Kreuz am Spielfeld platziert werden sollen.

Die erste Ziffer gibt die Reihe an und die zweite Ziffer die Spalte. Beispielsweise fürt die Konsoleneingabe `32` dazu, dass das Spielfeld in der dritten Reihe in der zweiten Spalte befüllt wird.

Weiterhin soll die Konsoleneingabe der Koordinaten einer Validierung unterzogen werden. Dabei sollen die folgenden Fälle überprüft werden:

- Die angegebenen Koordinaten dürfen nicht außerhalb des Spielfeldes liegen.
- Bereits belegte Koordinaten dürfen nicht erneut belegt werden.

Das Spiel soll mit einer Erfolgsmeldung beendet werden, wenn ein Spieler*in drei gleiche Zeichen vertikal, horizontal oder diagonal platzieren konnte. Im Falle von Unentschieden (gesamtes Spielfeld befüllt, allerdings kein Gewinner*in) soll ebenfalls eine entsprechend Ausgabe auf der Konsole ausgegeben werden.

Anstatt die gesamte Funktionalität in der `main`-Funktion zu deklarieren, sollen separate Funktionen erstellt werden. Nutzen sie dabei die bereits vordefinierten Funktionenköpfe. Sie können natürlich auch noch weitere Funktionen erstellen. Die angegebenen Funktionen stellen das Mindestmaß an ausgelagerter Funktionalität dar. Beachten Sie das Kommentarfeld (Text zwischen `/*` und `*/`) oberhalb der jeweiligen Funktionen, um die gewünschte Funktionalität korrekt zu implementieren. Sie finden diese Methodenköpfe auch nochmals unter dem Beispiel hier in der Angabe, sollten diese ausversehen gelöscht werden.

Im Template zur Bearbeitung sind auch bereits globale Variablen der Main-Klasse für die Koordinaten und dem aktuellen Spieler*in definiert. Auf diese Variablen kann entsprechend von allen Methoden aus zugegriffen werden und müssen nicht als Parameter übergeben werden.

Ein typischer Spielverlauf könnte entsprechend wie folgt aussehen:

```
+---+---+---+
|   |   |   |
+---+---+---+
|   |   |   |
+---+---+---+
|   |   |   |
+---+---+---+

Player X: 32
+---+---+---+
|   |   |   |
```

```
+---+---+---+
|   |   |   |
+---+---+---+
|   | X |   |
+---+---+---+

Player O: 32
invalid move (32 is occupied)
Player O: 14
invalid move (14 is outside the board)
Player O: 31
+---+---+---+
|   |   |   |
+---+---+---+
|   |   |   |
+---+---+---+
| O | X |   |
+---+---+---+

...


Player X: 22
+---+---+---+
| X | O | O |
+---+---+---+
| O | X | X |
+---+---+---+
| O | X | X |
+---+---+---+

Congratulations, X won!
```

Methodenköpfe und Variablen

```
public static final String DELIMITER_LINE = " +---+---+---+";
public static final String DELIMITER_SYMBOL = " | ";
public static final char EMPTY_CHAR = ' ';
public static final char PLAYER_X = 'X';
public static final char PLAYER_O = 'O';

// variables of the coordinates
public static char row1col1 = EMPTY_CHAR;
public static char row1col2 = EMPTY_CHAR;
public static char row1col3 = EMPTY_CHAR;
public static char row2col1 = EMPTY_CHAR;
public static char row2col2 = EMPTY_CHAR;
public static char row2col3 = EMPTY_CHAR;
public char row3col1 = EMPTY_CHAR;
public static char row3col2 = EMPTY_CHAR;
public static char row3col3 = EMPTY_CHAR;

public static char currentPlayer = PLAYER_X;

public static void main(String[] args) {
  "Hier Code einfügen"
}

/**
 * Places the current player character on the given coordinate.
 */
public static void makeMove(int userCoordinates) {
```

```
  "Hier Code einfügen"
}

/**
 * Determines if the given coordinate is already set on the game grid
 */
public static boolean isOccupied(int userCoordinates) {
  "Hier Code einfügen"
}

/**
 * Determines whether the board grid is completely filled
 */
public static boolean boardIsFull() {
  "Hier Code einfügen"
}

/**
 * Determines whether there is a winner on the board and returns the character of the winner, PLAYER_X
or PLAYER_O.
 * In case of no winner, function returns EMPTY_CHAR.
 */
public static char getWinner() {
  "Hier Code einfügen"
}

/**
 * Validates the input coordinates to be within the TicTacToe board range.
 */
public static boolean isOutsideBoard(int inputCoordinate) {
  "Hier Code einfügen"
}

/**
 * Prints out the TicTacToe board with all the values.
 */
public static void printBoard() {
  "Hier Code einfügen"
}
```