Bereich: Methoden-Spezial

Potenzen	Schwierigkeit: ★☆☆☆☆
Package: `de.dhbwka.java.exercise.methods`	**Klasse:** `Exponentiation`

Aufgabenstellung:

Die *n*-te Potenz einer reellen Zahl *x* kann auch rekursiv über folgende Rechenvorschrift berechnet werden (*n*: ganze Zahl, $n \geq 0$):

$$x^n = \begin{cases} 1 & \text{für } n = 0 \\ x * x^{n-1} & \text{für } n > 0 \end{cases}$$

Schreiben Sie eine Klasse `Exponentiation` mit einer (statischen) rekursiven Methode
`public static double` **`xPowerN`**`(double x,int n),`
welche x^n entsprechend dieser Rechenvorschrift berechnet. Verwenden Sie diese Methode in einer `main`-Methode (oder in einer anderen Klasse), welche die Basis x und den Exponenten n einliest sowie x^n mit Hilfe der Methode `xPowerN` bestimmt und ausgibt.

Beispieldialog:

```
Geben Sie bitte die Basis ein: 3,0
Geben Sie bitte den positiven ganzzahligen Exponenten ein: 3
3.0^ 3 = 27.0
```

Hinweis:
Schauen Sie sich die Parameter- und Rückgabewerte mit Hilfe des Debuggers oder mit Hilfe von Kontrollausgaben an.

Bereich: Methoden-Spezial

Fibonacci-Zahlen rekursiv	Schwierigkeit: ★☆☆☆☆
Package: `de.dhbwka.java.exercise.methods`	**Klasse:** `Fibonacci`

Aufgabenstellung:

Die Fibonacci-Zahlen sind eine Folge von Zahlen, in der jede Zahl gleich der Summe der beiden vorhergehenden Zahlen ist, d.h. $F_i = F_{i-1} + F_{i-2}$ für i = 3, 4, 5, ... Die beiden ersten Zahlen sind 1, d.h. $F_1 = 1$ und $F_2 = 1$.

Die ersten Fibonacci-Zahlen lauten: 1, 1, 2, 3, 5, 8, 13, 21, ...

Definieren Sie eine rekursive Methode, welche F_i entsprechend der o.g. Rechenvorschrift berechnet!
Verwenden Sie diese Methode in einer `main`-Methode, welche eine positive ganze Zahl *n* einliest und auf Plausibilität prüft (mit Eingabewiederholung, falls *n* kleiner als 1 ist) sowie die ersten *n* Fibonacci-Zahlen mit Hilfe dieser Methode bestimmt und ausgibt.

Beispieldialog:

```
Wie viele Fibonacci-Zahlen berechnen? 6

F(1) =   1
F(2) =   1
F(3) =   2
F(4) =   3
F(5) =   5
F(6) =   8
```

Hinweis:
Experimentieren Sie zur Überprüfung des Programmablaufs und von Zwischenergebnissen mit verschiedenen Optionen des Debuggers oder machen Sie Kontrollausgaben.

Bereich: Methoden-Spezial

Quicksort*	Schwierigkeit: ★★★☆☆
Package: `de.dhbwka.java.exercise.methods`	**Klasse:** `Quicksort`

Aufgabenstellung:

Quicksort ist ein schneller, rekursiver, nicht-stabiler Sortieralgorithmus, der nach dem Prinzip „teile und herrsche“ arbeitet. Er wurde ca. 1960 von C. Antony R. Hoare in seiner Grundform entwickelt und seitdem von vielen Forschern verbessert. Der Algorithmus hat den Vorteil, dass er über eine sehr kurze innere Schleife verfügt (was die Ausführungsgeschwindigkeit stark erhöht) und ohne zusätzlichen Speicherplatz auskommt (abgesehen von dem für die Rekursion zusätzlichen benötigten Platz auf dem Aufruf-Stack).

[*Quelle und weitere Info inkl. Beispiel: Wikipedia*]

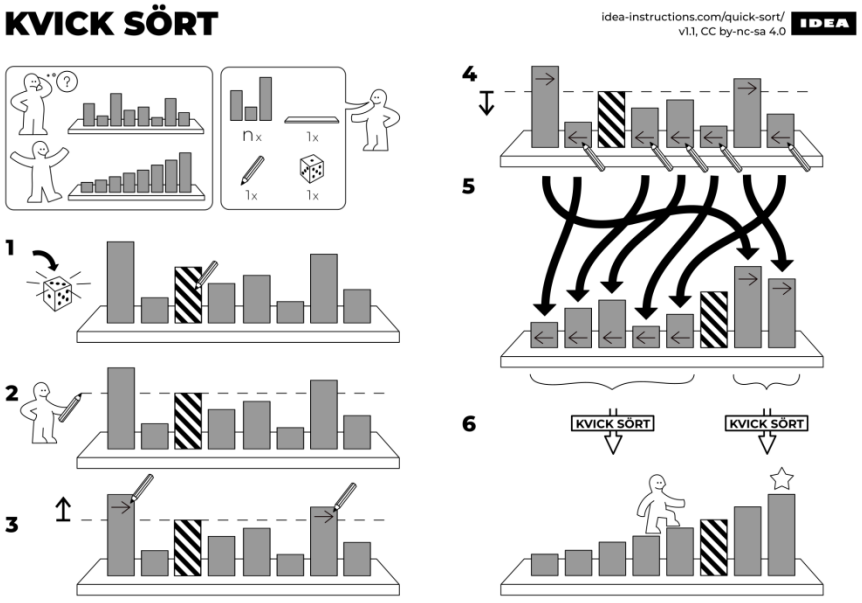
Implementieren Sie “Quicksort” in Java für Arrays von `int`-Werten! Die Methoden sollen neben der linken und rechten Grenze jeweils auch eine Referenz auf ein Array bekommen.
Implementieren Sie auch eine (statische) Methode
`public static void` **`sort`**`(int[] array),`
die ein beliebig großes Array (ohne die Angabe von Grenzen) sortieren kann!

Beispielausgabe:

Unsortierte Zahlenfolge:
9 2 5 7 8 3 4 10 1 6

Sortierte Zahlenfolge:
1 2 3 4 5 6 7 8 9 10

Algorithmus „Quicksort“ s. nächste Seite.



Der Quicksort-Algorithmus

```
daten[] = { …… } // Aufzählung von Elementen, hierauf operiert Quicksort
quicksort(0, daten.length-1)

funktion quicksort(links, rechts)
    falls links < rechts dann
        teiler := teile(links, rechts)
        quicksort(links, teiler-1)
        quicksort(teiler+1, rechts)
    ende
ende

funktion teile(links, rechts)
    // Ziel ist es hier, das Array in zwei Teile zu teilen.
    // Dazu wählen wir ein Pivotelement, mit dem wir alle Elemente des
    // Arrays vergleichen. Wir legen die beiden Teile des Arrays so an,
    // dass der linke alle kleineren Elemente enthält, der rechte Teil
    // alle größeren.
    // Wir tauschen hierzu, falls notwendig nicht passende Elemente.

    // wähle rechtes Element als Pivotelement aus
    // und starte mit j links vom Pivotelement
    i := links
    j := rechts - 1
    pivot := daten[rechts]

    wiederhole
        // Suche von links ein Element, das größer als das Pivot ist
        wiederhole solange daten[i] ≤ pivot und i < rechts
            i := i + 1
        ende
        // Suche von rechts ein Element, das kleiner als das Pivot ist
        wiederhole solange daten[j] ≥ pivot und j > links
             j := j - 1
        ende
        falls i < j dann
            tausche daten[i] mit daten[j]
        ende
    solange i < j // solange i an j nicht vorbeigelaufen ist

    // Tausche das Pivotelement mit seiner endgültigen Position i
    falls daten[i] > pivot dann
        tausche daten[i] mit daten[rechts]
    ende
    // gib die Position des Pivotelements zurück
    antworte i
ende
```