Bereich: Klassen (2)

Polynom 2. Grades	Schwierigkeit: ★★☆☆☆
Package: `de.dhbwka.java.exercise.classes`	**Klasse:** `Polynomial`

Aufgabenstellung:

Schreiben Sie eine Klasse `Polynomial`, die Polynome 2. Grades repräsentiert!
Ein Polynom dieser Klasse hat allgemein das folgende Aussehen:

$$ax^2 + bx + c$$

Erstellen Sie einen oder mehrere geeignete Konstruktoren!

Erstellen Sie Instanzenmethoden für folgende Operationen:

- `String`-Darstellung eines Polynoms (z.B. "`1.0x^2 -2.0x +3.0`")
- Berechnen des Funktionswertes `f(x)` eines Polynoms.
- Addition/Subtraktion zweier Polynome. Das Ergebnis ist ein neues Polynom.
- Multiplikation mit einem Skalar. Das Ergebnis ist ein neues Polynom.
- Berechnung der reellen Nullstellen des Polynoms.

Testen Sie Ihre Klasse mit einer `main()`-Methode oder mit einer anderen Klasse.

Hinweis:
Als Rückgabewert der Methode für die Berechnung der Nullstellen kann man ein Array mit geeigneter Länge verwenden (Länge 0 wenn es keine reelle Nullstelle gibt, 1 bei einer und 2 bei zwei reellen Nullstellen).

Beispielausgabe:

```
P1: 2.0x^2 +0.0x +0.0
P2: 0.0x^2 -4.0x +1.0
P3 = P1 + P2: 2.0x^2 -4.0x +1.0
P3 = 2.0 * P3: 4.0x^2 -8.0x +2.0
Nullstellen von P3 (4.0x^2 -8.0x +2.0):
1.7071067811865475 0.2928932188134524
```

Bereich: Klassen (2)

Komplexe Zahlen	Schwierigkeit: ★★★☆☆
Package: `de.dhbwka.java.exercise.classes`	**Klasse:** `Complex`

Aufgabenstellung:

Eine komplexe Zahl ist durch einen Realteil *a* und Imaginärteil *b* gegeben. Komplexe Zahlen werden meist in der Form *a+bi* angegeben. *i* ist eine imaginäre Zahl mit $i^2 = -1$.

Mit komplexen Zahlen lassen sich zum Beispiel für alle reellwertigen Polynome Nullstellen finden. So sind *i* (also 0+1*i*) und *-i* (also 0+(–1)*i*) Nullstellen von x^2+1.

Das Symbol „+“ verhält sich wie eine „normale“ Addition reeller Zahlen.
Zwei komplexe Zahlen *a+bi* und *c+di* werden wie folgt addiert, subtrahiert, multipliziert und dividiert:

- Addition: $(a + bi) + (c + di) = a + c + (b+d)i$
- Subtraktion: $(a + bi) - (c + di) = a - c + (b-d)i$
- Multiplikation: $(a + bi) * (c + di) = ac - bd + (ad + bc)i$
- Division: $\frac{a+bi}{c+di} = \frac{(a+bi)(c-di)}{(c+di)(c-di)} = \frac{ac+bd}{c^2+d^2} + \frac{bc-ad}{c^2+d^2}i$

Implementieren Sie eine Klasse `Complex` für komplexe Zahlen mit den Instanzenmethoden

- `getReal()`
- `getImag()`
- `add(Complex comp)`
- `sub(Complex comp)`
- `mult(Complex comp)`
- `div(Complex comp)`
- `toString()`

sowie dem Konstruktor

- `Complex(double real, double imag)`

Wählen Sie jeweils geeignete Rückgabewerte!

(Fortsetzung auf der nächsten Seite)

Die komplexen Zahlen ***C*** sind im Gegensatz zu den reellen Zahlen kein geordneter Körper, d. h. es gibt keine mit der Körperstruktur verträgliche Ordnungsrelation „≤“ auf ***C***.
Von zwei unterschiedlichen komplexen Zahlen kann man daher nicht sagen, welche von beiden „die größere“ bzw. „die kleinere“ Zahl ist.

Dennoch gibt es für die komplexen Zahlen eine Quasi-Ordnung, die über ihren (reellen) Betrag

$$|a+bi| = \sqrt{(a+bi)(a-bi)} = \sqrt{a^2+b^2}$$

definiert ist.

Implementieren Sie zunächst eine Methode `getMagnitude()`, welche den Betrag einer komplexen Zahl liefert und anschließend eine Methode `isLess(Complex comp)`, die als Ergebnis einen booleschen Wert liefert, der genau dann `true` ist, wenn der Betrag der komplexen Zahl, für welche die Methode aufgerufen wurde, kleiner ist als der Betrag der übergebenen Zahl `comp`.

Erzeugen Sie 10 zufällige komplexe Zahlen und geben Sie diese aufsteigend nach ihrem Betrag sortiert aus. Passen Sie zum Beispiel den Bubblesort-Algorithmus (vgl. Übungsblatt „Arrays (1)“) entsprechend an. Sehen die Zahlen nun „sortiert“ aus?

Beispielausgabe:

```
C1:     Complex 1.0 2.0i
C2:     Complex 2.0 1.0i
C1+C2:  Complex 3.0 3.0i
C1-C2:  Complex -1.0 1.0i
C1*C2:  Complex 0.0 5.0i
C1/C2:  Complex 0.8 0.6i
C1<C2?: false
Unsortiert:
5,037 + 8,190i  Betrag: 9,615
5,832 + 0,398i  Betrag: 5,846
6,726 + 3,479i  Betrag: 7,573
5,501 + 4,718i  Betrag: 7,247
5,010 + 6,658i  Betrag: 8,332
8,535 + 3,448i  Betrag: 9,205
0,609 + 0,278i  Betrag: 0,669
0,910 + 5,862i  Betrag: 5,933
8,571 + 9,513i  Betrag: 12,805
6,567 + 7,344i  Betrag: 9,852
Sortiert:
0,609 + 0,278i  Betrag: 0,669
5,832 + 0,398i  Betrag: 5,846
0,910 + 5,862i  Betrag: 5,933
5,501 + 4,718i  Betrag: 7,247
6,726 + 3,479i  Betrag: 7,573
5,010 + 6,658i  Betrag: 8,332
8,535 + 3,448i  Betrag: 9,205
5,037 + 8,190i  Betrag: 9,615
6,567 + 7,344i  Betrag: 9,852
8,571 + 9,513i  Betrag: 12,805
```

Bereich: Klassen (2)

Polynome und Horner-Schema	Schwierigkeit: ★★★☆☆
Package: `de.dhbwka.java.exercise.classes`	**Klasse:** `Horner`

Aufgabenstellung:

Das Horner-Schema (nach William George Horner) ist ein Umformungsverfahren für Polynome, um die Berechnung von Funktionswerten so einfach wie möglich zu machen.

Bei Polynomen in einer Variablen *x* erfordert das Auswerten an einer Stelle $x = a$ in der klassischen Form die Berechnung der Potenzen a^k. Die Umformung des Polynoms nach dem Horner-Schema beschleunigt die Berechnung, indem überflüssige Berechnungen bei den Potenzen vermieden werden. Erfolgt die Auswertung durch Gleitkommaoperationen, so wird darüber hinaus auch der Rechenfehler geringer gehalten.
Durch fortgesetztes Ausklammern der freien Polynomvariablen *x* wird das Polynom als Schachtelung von Produkten und Summen dargestellt. In der umgewandelten Darstellung kommt keine Potenzfunktion, sondern nur noch Multiplikation und Addition vor.

Beispiel:
Zur Illustration an einem Beispiel sortieren wir die Terme des Polynoms nach aufsteigenden Potenzen; das Ausklammern zur Überführung in das Horner-Schema arbeitet die Terme dann von rechts nach links ab:

$$11 + 7x - 5x^2 - 4x^3 + 2x^4 \;=\; 11 + x \cdot (7 + x \cdot (-5 + x \cdot (-4 + x \cdot 2)))$$

Die Beschleunigung der Auswertung an einer Stelle *x* kann man an diesem Beispiel wie folgt quantifizieren: In der klassischen Darstellung (linke Seite) werden 7 Multiplikationen benötigt, davon 3 zur Bildung der Potenzen x^2, x^3, x^4. Im Horner-Schema (rechte Seite) kommt man dagegen mit 4 Multiplikationen aus. Die Zahl der – rechnerisch weniger aufwendigen – Additionen ist in beiden Fällen gleich, nämlich 4.
Generell wird der Aufwand für Multiplikationen durch die Anwendung des Horner-Schemas auf fast die Hälfte reduziert.

a) Schreiben Sie eine Klasse `Horner`, welche Polynome beliebigen Grades darstellen kann!
b) Erstellen Sie Konstruktor(en) zum Erzeugen eines beliebigen Polynoms!
c) Implementieren Sie eine Methode `getValue(double x)`, welche für eine beliebige Gleitkommazahl `x` den Funktionswert des Polynoms gemäß dem Hornerschema berechnet!
d) Implementieren Sie eine Methode `toString()` zur Darstellung als Zeichenkette
e) Testen Sie Ihre Klasse mit Polynomen 10. Grades!

Beispielausgabe:

```
Polynomial f: 0.5*x^10 -3.0*x^9 +2.0*x^8 +4.0*x^7 +3.0*x^6 -10.0*x^5
+8.0*x^4 +4.5*x^3 +3.0*x^2 -2.0*x^1 +1.0*x^0
f(1.5) = 51.77587890625
```