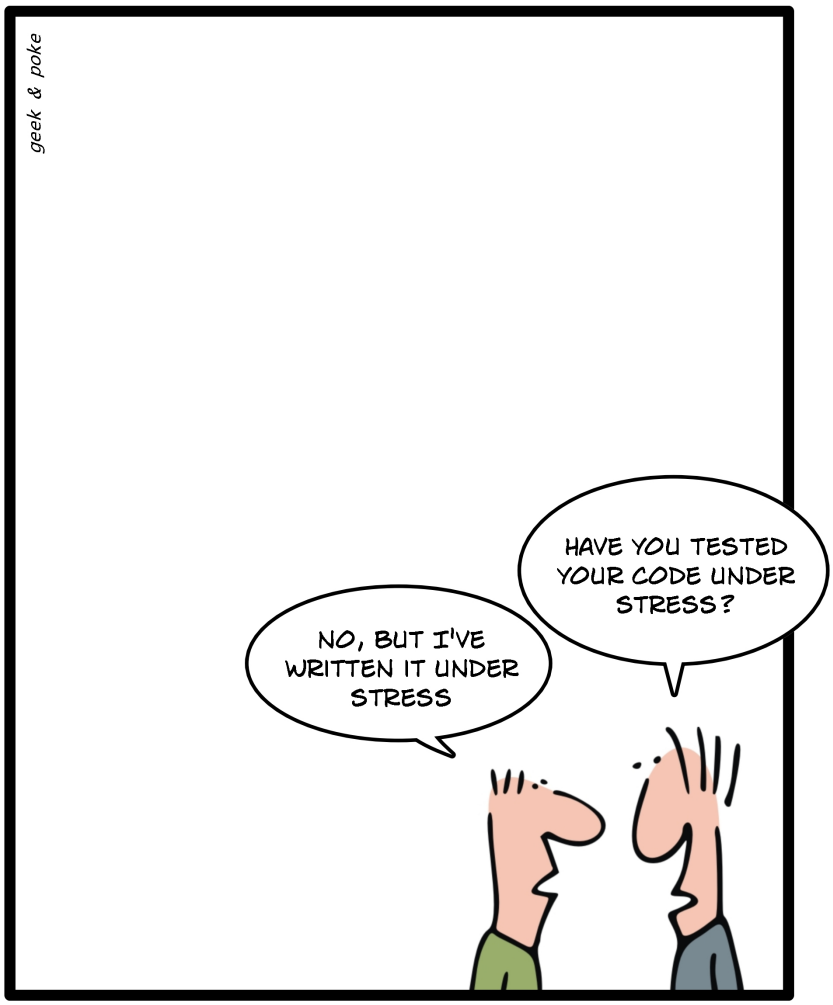
Universiteit van Amsterdam
Inleiding Programmeren

Opgave 3

José Lagerberg - Robin de Vries

Complexe Breuken



Deadline: zaterdag 21 september 2019, 18:00

1 Leerdoelen

Aan het einde van deze opdracht kan je:

- een eenvoudig, goed geconstrueerd, objectgeoriënteerd programma schrijven,
- eenvoudige standaard Java-methoden als **equals()** en **toString()** implementeren,
- (wiskundige) problemen omzetten naar code,
- een simpel testprogramma schrijven om je eigen code te testen.

2 Introductie

In deze opgave moet je twee klassen implementeren. De eerste klasse, de `Breuk`-klasse, moet het mogelijk maken om een rationaal getal (een breuk) te kunnen representeren en met deze representatie van breuken te kunnen rekenen. De tweede klasse doet hetzelfde met complexe getallen. Tot slot zullen deze klassen samengevoegd worden, om zo te kunnen rekenen met complexe breuken.

Om instanties van deze klassen te kunnen vergelijken met andere instanties van dezelfde klasse, moet de **equals()**-methode worden geïmplementeerd. Het moet ook mogelijk zijn om een instantie weer te geven door de **toString()**-methode te implementeren voor allebei de klassen.

Ten slotte moet er voor beide klassen een kort testprogramma geschreven worden, om zo te kunnen controleren of ze correct geïmplementeerd zijn. Dit testprogramma kun je het beste tijdens het programmeren van de klassen al schrijven, zodat je de geschreven code direct kan testen.

3 Opgave 3

3.1 Breuk

Rationale getallen hebben de vorm a/b met a de teller en b de noemer. Alhoewel een rationaal getal een quotiënt is van twee **integers** kan het niet altijd exact gerepresenteerd worden in een **floating-point**, zo is bijvoorbeeld $1/3 = 0.33333\ldots$.

Java levert primitieve datatypen voor gehele en floating-point getallen, maar niet voor rationale getallen. Daarom wordt er aan jullie gevraagd een nieuwe klasse Breuk te ontwerpen, waarmee rationale getallen als een Breuk object kunnen worden opgeslagen. Voor het representeren van deze rationale getallen gebruiken we een teller en een noemer. De teller kan alle gehele getallen bevatten, maar de noemer moet een geheel getal groter dan 0 zijn.

Begin met het schrijven van de klasse Breuk (**Breuk.java**). De volgende attributen moeten in deze klasse aanwezig zijn:

- Twee integer instantiatie-variabelen: *teller* en *noemer*.
- Vier verschillende constructoren, elk met verschillende parameters:
  - Een constructor met twee parameters: een **int** voor de teller en een **int** voor de noemer. Hier moet gecontroleerd worden of de noemer groter is dan nul. Als de noemer kleiner dan nul is moeten zowel de teller als de noemer met -1 vermenigvuldigd worden.
  - Een constructor met één parameter voor de teller; hier is de noemer gelijk aan één. Roep hier de bovengenoemde constructor aan om herhaling van code te voorkomen.

- Een constructor zonder parameters; hier is de teller gelijk aan nul. Roep hier weer de constructor met twee parameters aan.
- Een constructor met één parameter voor een bestaand Breuk object. Hier moet de nieuwe Breuk een kopie worden van de meegegeven (originele) breuk.

Verder moet de breuk opgeslagen worden in **vereenvoudigde vorm**: als je als teller 2 opgeeft en als noemer 12, moet de breuk 1/6 opgeslagen worden. Je kunt het algoritme van Euclides[1] gebruiken om de grootste gemene deler van de teller en de noemer uit te rekenen en zo de Breuk te vereenvoudigen. Indien je de implementatie van het algoritme van Euclides van internet kopieert, vergeet dan niet om netjes de bron te vermelden.

- Op breuken moeten de volgende operaties mogelijk zijn: *optellen*, *aftrekken*, *vermenigvuldigen* en *delen*.

  Bepaal voor iedere operatie hoe je dit kunt doen op basis van twee generieke breuken $\frac{a}{b}$ en $\frac{c}{d}$. Zo moet je voor het optellen van twee breuken ze eerst bijvoorbeeld eerst gelijknamig maken, waardoor er het volgende uitkomt:

$$\frac{a}{b} + \frac{c}{d} = \frac{a \cdot d}{b \cdot d} + \frac{b \cdot c}{b \cdot d} = \frac{a \cdot d + b \cdot c}{b \cdot d}$$

  En voor het vermenigvuldigen van twee generieke breuken geldt het volgende:

$$\frac{a}{b} \cdot \frac{c}{d} = \frac{a \cdot c}{b \cdot d}$$

  Als we dit implementeren resulteert dat in de code onderop deze pagina. Je hoeft de breuken niet te vereenvoudigen bij het doen van deze operaties omdat de constructor van het nieuwe Breuk object dat voor je doet (als je het handig hebt geïmplementeerd). Ook kun je slim gebruik maken van reeds geimplementeerde methoden (delen is bijvoorbeeld hetzelfde als vermenigvuldigen met het omgekeerde).

  Zorg ervoor dat je de volgende methoden implementeert:

  - `Breuk telOp(Breuk b)`
  - `Breuk trekAf(Breuk b)`
  - `Breuk vermenigvuldig(Breuk b)`
  - `Breuk deel(Breuk b)`
  - `Breuk omgekeerde()`

  De methoden moeten deze naam hebben, hier mag niet van worden afgeweken.

- Implementeer tot slot nog de methoden **toString()** en **equals()** van de Breuk klasse. Dit zijn standaard Java methoden die kunnen worden overschreven. De methode **toString()** geeft een breuk terug in (leesbaar) String formaat en de **equals()** methode bepaalt of twee breuken gelijk zijn.

Als voorbeeld geven we de implementatie van de methode *vermenigvuldig* die twee breuken vermenigvuldigt:

```
Breuk vermenigvuldig(Breuk breuk) {
    int nieuweTeller = this.teller * breuk.teller;
    int nieuweNoemer = this.noemer * breuk.noemer;
    return new Breuk(nieuweTeller, nieuweNoemer);
}
```

[1] https://nl.wikipedia.org/wiki/Algoritme_van_Euclides

3.2 ComplexGetal

Complexe getallen zijn uitdrukkingen van de vorm $a+bi$, waarin a en b reële getallen zijn, en i een nieuw getal voorstelt: de imaginaire eenheid. Deze imaginaire eenheid heeft als eigenschap: $i^2 = -1$.
Een complex getal $a + bi$ correspondeert met een punt met coördinaten (a, b) in een vlak: a is het reële deel van het complexe getal, en b het imaginaire deel. Je kunt hier meer informatie over vinden op Wikipedia of het eerste hoofdstuk lezen van Jan van de Craats [2]. Zorg dat je de basis snapt, voordat je begint met programmeren!

De volgende rekenregels gelden voor deze complexe getallen:

1. **som**

   $(a + bi) + (c + di) = (a + c) + (b + d)i$

2. **verschil**

   $(a + bi) - (c + di) = (a - c) + (b - d)i$

3. **product**

   $(a + bi)(c + di) = ac + adi + bci + bdi^2 = (ac - bd) + (ad + bc)i$

4. **quotiënt**

   $$\frac{a + bi}{c + di} = \frac{(a + bi)(c - di)}{(c + di)(c - di)} = \frac{ac + bd}{c^2 + d^2} + \frac{bc - ad}{c^2 + d^2}i$$

5. **omgekeerde**

   $$\frac{1}{a + bi} = \frac{a - bi}{(a + bi)(a - bi)} = \frac{a}{a^2 + b^2} + \frac{-b}{a^2 + b^2}i$$

Het is de bedoeling om twee instanties van de Breuk-klasse te gebruiken voor de twee reële delen van het complexe getal; het wordt echter sterk aangeraden om als tussenstap eerst de klasse zó te implementeren dat de twee reële delen van het complexe getal **double**-variabelen zijn. Deze tussenstap is in principe optioneel, de in te leveren klasse moet enkel werken met Breuk-objecten.

3.2.1 ComplexGetal met doubles (optioneel, als tussenstap)

Begin met het schrijven van de klasse ComplexGetal. De volgende attributen in deze te schrijven klasse aanwezig zijn:

- Twee **double** instantiatie-variabelen: *re*(het reële deel) en *im*(het imaginaire deel).
- Een constructor met twee parameters.
- Op complexe getallen moeten de volgende operaties mogelijk zijn: *optellen*, *aftrekken*, *vermenigvuldigen*, *delen* en omgekeerde:
  - `ComplexGetal telOp(ComplexGetal cg)`
  - `ComplexGetal trekAf(ComplexGetal cg)`
  - `ComplexGetal vermenigvuldig(ComplexGetal cg)`
  - `ComplexGetal deel(ComplexGetal cg)`
  - `ComplexGetal omgekeerde()`

  De methoden moeten deze naam hebben, hier mag niet van worden afgeweken.
- De methoden **toString()** om complexe getallen te weergeven en de methode **equals()** om te kijken of twee complexe getallen gelijk zijn.

[2] `https://staff.science.uva.nl/j.vandecraats/CGnieuw.pdf`

3.2.2 ComplexGetal met breuken

Als je de vorige stap met doubles hebt geïmplementeerd, zorg er dan voor dat je een backup maakt van **ComplexGetal.java**, voordat je begint met het aanpassen ervan. Doe dit bijvoorbeeld met het commando:

```
cp ComplexGetal.java ComplexGetalBackup.java
```

In deze stap gaan we complexe getallen modelleren met breuken.

Deze complexe getallen zien er als volgt uit: $a + bi = \frac{p}{q} + \frac{r}{s}i$.
Het reële deel van een zodanig complex getal is de breuk $\frac{p}{q}$, en het imaginaire deel is de breuk $\frac{r}{s}$.

De rekenregels voor deze speciale complexe getallen zijn als volgt:

1. **som**
   $\left(\frac{p_1}{q_1} + \frac{r_1}{s_1}i\right) + \left(\frac{p_2}{q_2} + \frac{r_2}{s_2}i\right) = \left(\frac{p_1}{q_1} + \frac{p_2}{q_2}\right) + \left(\frac{r_1}{s_1} + \frac{r_2}{s_2}\right)i$
2. **verschil**
   $\left(\frac{p_1}{q_1} + \frac{r_1}{s_1}i\right) - \left(\frac{p_2}{q_2} + \frac{r_2}{s_2}i\right) = \left(\frac{p_1}{q_1} - \frac{p_2}{q_2}\right) + \left(\frac{r_1}{s_1} - \frac{r_2}{s_2}\right)i$
3. **product**
   $\left(\frac{p_1}{q_1} + \frac{r_1}{s_1}i\right)\left(\frac{p_2}{q_2} + \frac{r_2}{s_2}i\right) = \left(\frac{p_1 p_2}{q_1 q_2} - \frac{r_1 r_2}{s_1 s_2}\right) + \left(\frac{p_1 r_2}{q_1 s_2} + \frac{r_1 p_2}{s_1 q_2}\right)i$
4. **omgekeerde**
   $$\frac{1}{\frac{p}{q} + \frac{r}{s}i} = \frac{\frac{p}{q}}{\frac{p^2}{q^2} + \frac{r^2}{s^2}} - \frac{\frac{r}{s}i}{\frac{p^2}{q^2} + \frac{r^2}{s^2}}$$
5. **quotiënt**

   Het quotiënt kun je berekenen door te vermenigvuldigen met het omgekeerde.

In de ComplexGetal-klasse met breuken moeten de volgende attributen aanwezig zijn:

- Twee Breuk instantiatie-variabelen: *re*(het reële deel) en *im*(het imaginaire deel).
- – Een constructor met twee parameters: een Breuk voor het reële deel, en een Breuk voor het imaginair deel.
  – Een constructor met vier **int** parameters: twee voor de tellers van de breuken in het complexe getal, en twee voor de noemers.
- Dezelfde operaties moeten mogelijk zijn als bij de ComplexGetal-klasse gemaakt met doubles. Ook moeten **toString()** en **equals()** geïmplementeerd worden.

4 toString-methoden

Het is de bedoeling dat je beide objecten kunt printen en dat ze een zinnige uitvoer op dat moment geven. Om je een beetje opweg te helpen volgen hieronder een paar voorbeelden:

```
Breuk b1 = new Breuk(1,2);
Breuk b2 = new Breuk();
Breuk b3 = new Breuk(4,2);

System.out.println(b1);
System.out.println(b2);
System.out.println(b3);

ComplexGetal cg1 = new ComplexGetal(1, 2, 4, 5);
ComplexGetal cg2 = new ComplexGetal(b1, b2);
ComplexGetal cg3 = new ComplexGetal(b2, b3);

System.out.println(cg1);
System.out.println(cg2);
System.out.println(cg3);
```

Met vervolgens als uitvoer:

```
1/2
0
2
1/2 + 4/5i
1/2 + 0i
0 + 2i
```

5 Testen

Bij het schrijven van code wordt regelmatig fouten gemaakt. Omdat mensen niet feiloos zijn is hun code dat ook niet. Toch kun je er wel voor zorgen dat relatief eenvoudige foutjes automatisch worden gedetecteerd door de computer.

In het geval van breuken bijvoorbeeld: je weet dat de breuk $\frac{1}{2}$ en $\frac{2}{4}$ identiek zijn. De laatste moet namelijk vereenvoudigd worden tot de eerste. Je kunt dus in een testprogrammaatje beide breuken instantiëren en vervolgens met equals controleren dat dit goed gaat.
Zo kun je bijvoorbeeld ook controleren dat als je een breuk hebt, en daar een andere breuk zowel bij optelt als aftrekt, dat je de oorspronkelijke breuk weer terug hebt: $a + b - b = a$.
En ook dat delen hetzelfde is als vermenigvuldigen met het omgekeerde, en dat een getal vermenigvuldigt met zijn omgekeerde altijd 1 oplevert.
Dit zijn allemaal dingen die je automatisch kunt testen met jouw Breuk klasse voor verschillende breuken, en dat is het doel van dit laatste stuk van de opgave.

Schrijf twee programmaatjes **TestBreuken.java** en **TestComplexeGetallen.java** die zoveel mogelijk de correctheid van je implementatie proberen te tonen. Je hoeft absoluut geen ingewikkelde JUnit testcases te schrijven (of te weten wat dat überhaupt zijn). Met de tips van de vorige alinea, wat met de hand uitgerekende voorbeeldjes en de equals methode moet je een heel eind komen. Probeer in ieder geval wat lastige gevallen te bedenken, waarna je met jouw tests kunt aantonen dat deze gevallen correct worden afgehandeld.

Het kan zeker geen kwaad om sommige tests al te schrijven terwijl je nog aan het programmeren bent, hier hebben we zelfs een naam voor: Test Driven Development.

6 Inleveren

Lever de files **Breuk.java**, **TestBreuken.java**, **ComplexGetal.java** en **TestComplexeGetallen.java** in op Canvas door ze in een archief (zip of tar.gz) te stoppen. Lever geen andere bestanden of .class files in!

Vergeet niet voor het inleveren nog een keer goed naar de rubric en de styleguide te kijken!