Oefeningen week 12: Multithreading

Github: https://github.com/PXLJavaAdvanced1819/week12-multithreading

Oefening 1: Simulatie van het gebruik van een bankrekening

Volgende programma-instellingen worden via properties (programma attributen) beheerd:

- Startbalans voor een bankrekening
- Aantal gebruikers die toegang hebben tot de bankrekening
- Aantal transacties dat iedere gebruiker zal uitvoeren (iedere gebruiker zal evenveel transacties uitvoeren)
- Limietwaarde voor een transactie (zowel voor geldopneming als storting)

Maak nu een bank account aan met de opgegeven startbalans.

Maak voor iedere gebruiker een thread aan die het opgegeven aantal transacties zal uitvoeren, dit kunnen zowel geldopnemingen als stortingen zijn. Hou hierbij rekening met de opgegeven limietwaarde. Let wel op dat de balans van bankrekening nooit negatief mag worden.

Na iedere transactie worden de gegevens en de balans van de rekening in een transaction log bestand.

Extra oefening: Simulatie van de wachtrij bij een bank

We maken een simulatie van een wachtrij probleem. Wanneer klanten binnenkomen in de bank, zullen ze moeten wachten tot een bankbediende (teller) beschikbaar is. We willen met deze simulatie te weten komen hoe lang de klanten gemiddeld in de lijn moeten wachten. Dit is dan ook het doel van deze oefeningen.

Requirements van de toepassing.

Klanten komen aan in de bank en worden in de wachtrij geplaatst. Als bankbedienden vrij zijn, zullen ze de eerste persoon uit de wachtrij nemen en deze

verder helpen. Op regelmatige tijdstippen zal het programma een status-rapport tonen met het aantal klanten in de wachtrij, het aantal klanten dat reeds werd geholpen en tenslotte de gemiddelde wachttijd in de rij. Het programma eindigt pas wanneer je het zelf stopt.

Objecten

We identificeren dus de volgende objecten:

- klanten
- bank (met wachtrij)
- bankbedienden
- een object om regelmatig rapporten te genereren
- een object om klanten aan te maken

Nu zullen we beslissen welke objecten threads zullen worden.

Het object om rapporten te genereren is een thread. Op regelmatige tijdstippen wordt het rapport afgedrukt. Deze thread heeft dus het volgende algoritme:

```
0. Loop
   a. Wait N seconds.
   b. Print the report.
```

Het object om klant-objecten te genereren is ook een thread. De thread wacht steeds een willekeurige periode vooraleer hij een nieuwe klant aanmaakt. Zodra een klant wordt gegenereerd, wordt deze toegevoegd aan de wachtrij.

```
0. Loop
   a. Create a new customer.
   b. Put the customer in line.
   c. Wait a random number of seconds.
```

Ook de bankbedienden zijn threads in de simulatie. Als een bankbediende beschikbaar is, wordt de eerste klant uit de wachtrij genomen. Indien er geen klanten staan te wachten, zal de bediende even wachten en daarna opnieuw de wachtrij controleren. Bankbedienden gebruiken het volgende algoritme:

```
0. Loop
   a. Check the line.
   b. If there is a customer get it.
      A. Determine the transaction time.
      B. sleep for the transaction time.
   c. otherwise
      A. Sleep for a fixed amount of time.
```

Je eerste gedachte zal zijn om ook van de klant een thread te maken. In deze simulatie moet de klant echter GEEN actief object zijn. De klant wordt door de klant-object generator in de wachtrij geplaatst en door de bankbediende er weer uitgehaald. Indien we willen dat klanten ook verveeld geraken in de wachtrij en opstappen voor ze aan de beurt zijn geweest, zouden we wel een thread gebruiken. Dit valt buiten de scope van deze opgave, maar je mag het natuurlijk altijd proberen 😊.

De bank (met de wachtrij) en klanten worden dus voorgesteld door gewone klassen.

Start eerst met het vervolledigen van de **BankLine** klasse die de bank met de wachtrij voorstelt. Alle threads die we gaan aanmaken zullen gebruik maken van een object van de BankLine klasse.

De verantwoordelijkheden van de klasse BankLine zijn:

- De wachtrij met wachtende Customer objecten beheren
- Weten hoeveel klanten er in de wachtrij staan
- Weten hoeveel klanten er reeds werden geholpen

- De totale tijd kennen dat alle klanten samen moesten wachten
- De gemiddelde wachttijd kunnen berekenen
- Een Customer object toevoegen aan de wachtrij
- De volgende klant naar een bankbediende doorverwijzen

Implementeer nu de klasse **Customer**. Een object van deze klasse moet:

- Het tijdstip onthouden wanneer hij in de wachtrij werd toegevoegd
- Het tijdstip onthouden wanneer hij aan de beurt was
- De tijd dat hij moest wachten kunnen berekenen
- Zijn of haar naam kennen
- Weten hoeveel tijd er nodig is bij de bankbediende

De transactietijd, de tijd dat een klant nodig heeft bij de bankbediende, is een random waarde tussen 1000 en 10000. Deze wordt bepaald bij de aanmaak van de klant. Deze transactietijd kan je opvragen aan het object.

De methode *inLine()* voorzie je om de tijd te registreren dat de klant binnenkomt in de wachtrij.

De methode *long outOfLine()* gebruik je om te registreren dat de klant aan de beurt is. De tijd dat hij heeft moeten wachten geef je terug als resultaat.

De **CustomerGenerator** thread krijg je reeds cadeau. Het algoritme van de CustomerGenerator ziet er als volgt uit:

```
0. Loop
   A. Try
      a. Create a new customer.
      b. Put the customer in line.
      c. Generate a random integer.
      d. Sleep the given time.
   B. Catch
      a. Do nothing.
```

Maak nu de thread **Teller** aan. De bankbediende heeft een naam en werkt volgens dit algoritme:

```
0. Loop
   A. Try
      a. Get a customer from the line.
      b. If no customer sleep for some time.
      c. If there is a customer
         i. print out helping customer.
         ii. Sleep for the transaction time.
         iii. print out finished helping customer
   B. Catch
      a. Do nothing.
```

Als je de thread **ReportGenerator** aanmaakt gebruik je een constructor met 2 parameters:

```
public ReportGenerator(BankLine line, int reportWait)
```

De tweede parameter vertelt ons hoelang we moeten wachten tussen ieder rapport. Dit "rapport" is gewoon een afdruk in de console van de status van de wachtrij.

De **BankSimulation** klasse is reeds voorzien, maar een deel van de code staat nog in commentaar.

Hier volgt nog het algoritme van de ThreadDemo:

```
0. Print out a greeting.
1. Create the bank line.
2. Create a teller named Anna.
3. Create a teller named Betty.
4. Create a customer generator.
5. Create a report generator.
6. Start the customer generator.
7. Start the report generator.
8. Start the first teller.
9. Start the second teller.
```