GHENT UNIVERSITY

FACULTY OF ENGINEERING AND ARCHITECTURE

Lab 2 : Multithreading

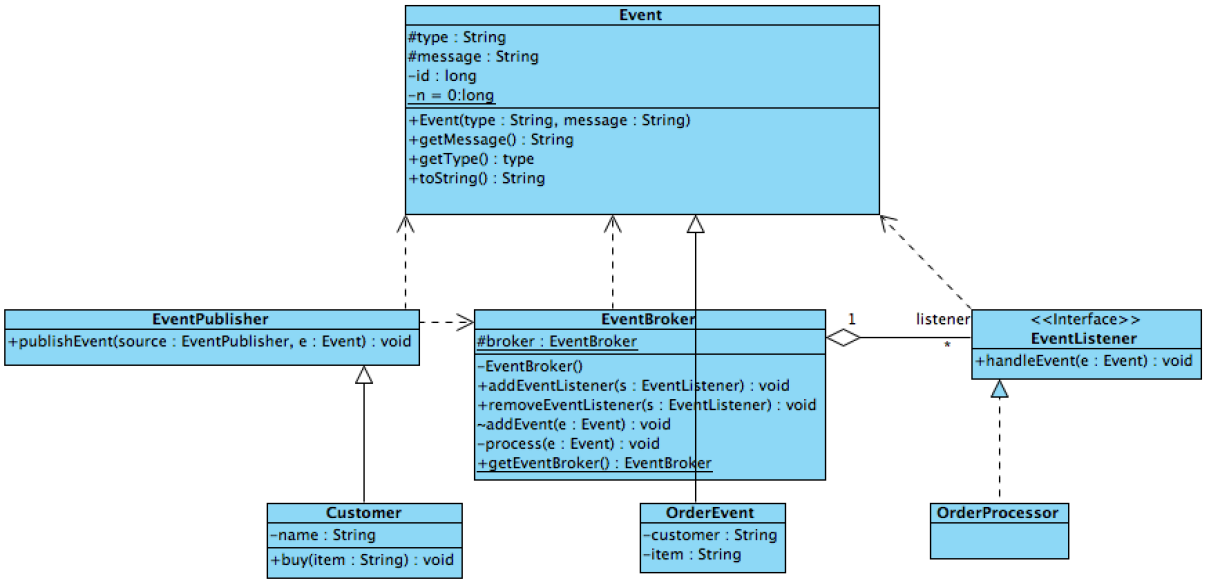
9 maart 2018

Cedric De Boom
(e-mail: {voornaam.familienaam}@ugent.be)

1 Inleiding

Multithreading is een zeer belangrijk concept in softwareontwikkeling. Parallellisme is o.a. nodig om bruikbare GUIs te bouwen, meer bepaald om een programma niet te laten blokkeren wanneer het wacht op input, of om de performantie te verhogen door verschillende taken in parallel uit te voeren. In Java wordt hiervoor de klasse `Thread` gebruikt, samen met vele laag-niveau en hoog-niveau synchronisatieprimitieven om communicatie tussen verschillende draden correct te laten verlopen.

In deze labsessie gaan we dieper in op het concept multithreading. Hierbij gebruiken we de code uit de package `eventbroker` van lab 1, met als doel de `EventBroker` multithreaded te maken. Op die manier kunnen verschillende `EventPublisher`s onafhankelijk van elkaar `Event`s genereren, en kunnen deze Events in parallel afgehandeld worden. In deze lab-sessie zal gebruik gemaakt worden van een eenvoudige applicatie die een webwinkel voorstelt, weergegeven in Figuur 1. Aan de ene kant heb je objecten van de klasse `Customer`, die klanten voorstellen en items kunnen kopen door de methode `buy(String item)` op te roepen. De bestellingen worden dan als `OrderEvent` doorgegeven aan de `EventBroker`, die ze verder stuurt naar een `OrderProcessor` object, die op zijn beurt de bestellingen verwerkt.



Figuur 1: UML klassendiagram van de orderverwerkingsapplicatie.

2 Multithreading basics

In Java is elke draad (Eng. thread) geassocieerd met een object van de klasse `Thread`. Een applicatie die een draad aanmaakt, dient zelf de code te voorzien die door die draad uitgevoerd dient te worden. Er zijn twee manieren om een draad aan te maken:

- Implementeer de `Runnable` interface. De `Runnable` interface definieert één enkele methode, `run()`, die de code bevat die de draad moet uitvoeren. Het `Runnable` object wordt meegegeven aan de `Thread` constructor, zoals in het `HelloRunnable` voorbeeld:

```
public class HelloRunnable implements Runnable {

  public void run() {
      System.out.println("Hello from a thread!");
  }

  public static void main(String[] args) {
    Thread t = new Thread(new HelloRunnable());
    t.start();
  }
}
```

- Maak een subklasse van `Thread`. De `Thread` klasse zelf implementeert de `Runnable` interface, maar met een lege `run()` methode. Door over te erven van `Thread`, kan je je eigen implementatie van `run` implementeren, zoals in het `HelloThread` voorbeeld:

```
public class HelloThread extends Thread {

    public void run() {
      System.out.println("Hello from a thread!");
    }

    public static void main(String[] args) {
      Thread t = new HelloThread();
      t.start();
    }
}
```

- Je kan ook gebruik maken van een *anonieme klasse* om snel een draad aan te maken en te starten:

```
public class Programma {
    public static void main(String[] args) {
      Runnable r = new Runnable() {
        public void run() {
          System.out.println("Hello from a thread!");
        }
      };
      Thread t = new Thread(r);
      t.start();
    }
}
```

  ... of met de nieuwe Lambda-notatie (vanaf Java 8):

```
public class Programma {
    public static void main(String[] args) {
        Thread t = new Thread(() -> {
            System.out.println("Hello from a thread!");
        });
        t.start();
    }
}
```

De `Thread` klasse voorziet ook een aantal methoden om het gedrag van de draad te controleren. De methode `sleep(long millis)` laat de draad slapen voor een opgegeven aantal milliseconden, zodat gedurende die tijd de processor vrijkomt voor andere draden. Deze tijd is echter niet gegarandeerd op de milliseconde nauwkeurig, aangezien dit afhangt van het onderliggende besturingssyteem.

```
try {
    Thread.sleep(4000);
} catch (InterruptedException e) {
    e.printStackTrace();
}
```

Merk op dat we de `InterruptedException` opvangen. Deze exception wordt opgegooid wanneer de methode `interrupt()` wordt opgeroepen tijdens het slapen. In de praktijk wordt echter niet vaak gebruik gemaakt van `interrupt()`.

Wanneer je de draad niet wil laten slapen voor een vastgelegde tijd, maar wel andere draden de kans wil geven om uit te voeren, kan je gebruik maken van de `yield()`-methode. Dit zal de huidige draad tijdelijk laten wachten, tot wanneer de scheduler hem opnieuw uitpikt om uit te voeren.

De `join()` methode laat toe om de huidige draad te laten wachten tot een andere draad volledig uitgevoerd is (m.a.w. tot de `run()`-methode van deze draad volledig uitgevoerd werd). Wanneer `t` een `Thread`-object is dat aan het uitvoeren is, dan laat je door `t.join()` op te roepen de huidige draad wachten tot `t` volledig uitgevoerd werd (m.a.w. tot haar `return`-statement bereikt is). Merk op dat `join()` net als `sleep()` ook de `InterruptedException` opgooit wanneer de huidige draad onderbroken wordt.

Opgave 1

Download de startcode naar een nieuw Java-project en kopieer hierin ook het package `eventbroker` uit practicum 1. Pas de `Main` klasse aan zodat voor elke `Customer` een nieuwe draad aangemaakt wordt die alle bestellingen van die klant creëert. Gebruik hiervoor een *anonieme klasse*. Pas wanneer alle orders verwerkt zijn, wordt het totaal aantal verwerkte orders uitgeprint met behulp van de methode `OrderProcessor.getNumberOfOrders()`. Merk op: het is de bedoeling dat alle customers in parallel verwerkt worden, niet dat ze sequentieel na elkaar in een aparte draad worden uitgevoerd (anders is het zinloos om draden te gebruiken).

Terzijde: multithreaded applicaties debuggen?

Het debuggen van meerdradige software is geen sinecure, vaak omdat het verloop ervan onderhevig is aan het toeval. Indien je programma in de loop van dit practicum niet het gewenste gedrag vertoont, plaats je best zo veel mogelijk controlepunten onder de vorm van `println`'s. Zo krijg je zicht op waar je code blokkeert of waar er racecondities optreden. In een professionele omgeving wordt er uiteraard gezorgd voor robuuste testen.

3 Synchronisatie

Wanneer meerdere draden in parallel dezelfde gegevens willen lezen en schrijven, kunnen er zogenaamde *races* optreden. Dit betekent dat het resultaat van het programma af kan hangen van de manier waarop de verschillende draden op de processorkernen gescheduled worden. Om dit te voorkomen kan in Java gebruik gemaakt worden van het `synchronized` sleutelwoord. Door een methode `synchronized` te declareren, zal steeds slechts één draad tegelijkertijd de methode kunnen uitvoeren. Als toch een andere draad tegelijk die methode wil oproepen, zal deze blokkeren tot de uitvoerende draad de methode returned.

```
public synchronized void doSomething(){
  // only one thread at a time can execute this method
}
```

Naast `synchronized` methoden kan je ook gebruik maken van `synchronized` statements. Hierbij declareer je als programmeur zelf het object dat de draden moeten gebruiken om een lock te nemen. Dit laat toe een fijnere granulariteit te gebruiken dan bij `synchronized` methoden.

```
Object lock = new Object();
synchronized(lock){
  // only one thread at a time can execute this block
}
```

In feite komen `synchronized` methoden op hetzelfde neer als `synchronized` statements, maar bij het eerste beslaat het synchronized block de hele methode, en wordt als lock object het `this` object genomen (het klasse object in geval van statische methoden).

Een ander nuttig sleutelwoord is `volatile`. Dit geeft de Java compiler aan dat een draad de waarde van de variabele steeds moet opvragen in het geheugen, en deze niet gecachet mag worden. Dit garandeert dat wijzigingen aan de variabele door de ene draad meteen "gezien" worden door de andere. Dit wordt vaak gebruikt wanneer je een draad continu een bewerking wil laten uitvoeren, tot een methode om te stoppen wordt opgeroepen.

```
class Test implements Runnable {
  private volatile boolean stop = false;

  public void run(){
    while(!stop){
      // do something
    }
  }

  public void stop(){
    stop = true;
  }
}
```

Opgave 2

Laat in de `Main` klasse het aantal geplaatste bestellingen per klant toenemen door de variabele `noOrders` te verhogen. (Tip: voor grote aantallen is het aangewezen de `println`-statements in `Customer` en `OrderProcessor` uit te commentariëren om de uitvoeringstijd beperkt te houden) Blijft de output nog steeds correct? Waarom (niet)? Los eventuele problemen op door gebruik te maken van synchronisatieprimitieven.

Opgave 3 - Busy wait

Laat de tijd nodig om een bestelling te verwerken toenemen door gebruik te maken van de methode `doWork()` in de klasse `OrderProcessor`. Vermits de event broker alle events afhandelt in dezelfde draad waar de events worden toegevoegd, zal de `buy()`-methode blokkeren totdat de bestelling verwerkt is. Dit is typisch geen gewenst gedrag, aangezien men doorgaans wil dat het programma responsief blijft m.a.w. dat klanten nieuwe bestellingen kunnen plaatsen terwijl vorige bestellingen afgehandeld worden. Maak volgende veranderingen aan de `EventBroker` om te zorgen voor asynchrone eventverwerking.

1. Voeg een attribuut `LinkedList<QueueItem> queue` toe, waarbij een `QueueItem` een `Event` en zijn bron bijhoudt dat is binnengekomen maar nog niet is verwerkt (Tip: gebruik een *inwendige* `private class` voor de klasse `QueueItem`).
2. In de methode `addEvent()` wordt nu niet meer de `process()` methode opgeroepen, maar in de plaats daarvan wordt een nieuw `QueueItem` gemaakt en aan de `queue` toegevoegd.
3. De events worden afgehandeld in een aparte draad. Laat de `EventBroker`-klasse de `Runnable`- interface implementeren waarbij in de `run()` methode in een lus de events van de `queue` gehaald en verwerkt worden (Tip: gebruik de methode `poll()` van `LinkedList`).
4. Implementeer een `start()` en `stop()` methode in de klasse `EventBroker`. De `start()` methode start een nieuwe draad om events van de `queue` af te halen en de correcte afhandelmethode op te roepen. De `stop()` methode zorgt ervoor dat geen nieuwe events meer aan de `queue` kunnen worden toegevoegd, en keert pas terug wanneer de `queue` opnieuw leeg is.
5. Zorg voor correcte synchronisatie bij lees- en of schrijfoperaties op het `queue` object.

Controleer de correcte werking van de parallelle `EventBroker` door de `Main`-klasse zo aan te passen dat:

- de `EventBroker`-thread opgestart wordt
- de `Customer`-objecten hun bestellingen plaatsen
- de `EventBroker`-thread gestopt wordt wanneer alle bestellingen aan de `OrderProcessor` doorgegeven zijn.

4 Escaping this reference

Een ander probleem dat kan optreden als gevolg van parallelisme is de zogenaamde *escaping this reference*. Dit probleem treedt op wanneer je in de constructor de `this`-referentie doorgeeft aan andere objecten. Beschouw het volgende voorbeeld.

```
public class EventListener {

  public EventListener(EventSource eventSource) {
    // do our initialization
    ...

    // register ourselves with the event source
    eventSource.registerListener(this);
  }

  public onEvent(Event e) {
    // handle the event
  }
}
```

Op het eerste gezicht is hier niets fout mee. Het probleem zit hem echter in het registreren van de `EventListener` in de constructor. Op het moment dat het `this`-object geregistreerd wordt als listener, is de constructor nog niet noodzakelijk afgelopen. Een andere draad kan reeds een `Event` gepubliceerd hebben, waardoor de `onEvent` methode wordt opgeroepen wanneer het object nog niet correct geïnitializeerd is. Het probleem wordt zichtbaar wanneer een andere klasse overerft van `EventListener`:

```
public class MyEventListener extends EventListener {

  public EventListener(EventSource eventSource) {
    super();
    // super registered this reference,
    // all that follows not yet initialized
    ...
  }

  public onEvent(Event e) {
    // this event handler can be called before constructor finished
  }
}
```

Daarom is het geen goed idee om de `this`-referentie reeds door te geven aan een ander object tijdens de constructor. Aan de andere kant dien je dan wel te vertrouwen op de programmeur om het object na constructie wel nog steeds te registreren. Om daaraan tegemoet te komen kan je gebruik maken van een zogenaamde factory-methode, die voor jou de constructor oproept, en nadien alle nodige registraties afhandelt.

Opgave 4

Bekijk de broncode van de `BlacklistOrderProcessor`. Wat zou kunnen mislopen bij constructie? (Tip: maak in de `main()`-methode een `BlacklistOrderProcessor` aan terwijl er tegelijkertijd bestellingen geplaatst worden). Los dit probleem op door gebruik te maken van een statische factory-methode in zowel `OrderProcessor` als `BlacklistOrderProcessor`.

5 Wait en notify

Vaak moeten de taken van verschillende draden ook gecoördineerd worden, waarbij de ene taak moet wachten op de andere. Een eenvoudig mechanisme om dit te doen is gebruik te maken van zogenaamde *guarded blocks*. Dit is een blok dat code slechts mag uitgevoerd worden op het moment dat een bepaalde beginconditie voldaan is. Om de ene draad te laten wachten tot de conditie voldaan is, terwijl de andere draad de conditie aanpast, wordt in Java gebruik gemaakt van de `wait()` en `notify()` methoden van de `Object` klasse.

```
synchronized(lock){
  while(!condition) {
    try {
        lock.wait();
    } catch (InterruptedException e) {}
  }
  // do something
}

synchronized(lock){
  condition = true;
  lock.notifyAll();
}
```

Wanneer een draad `lock.wait()` oproept, dient deze gesynchronizeerd te zijn op het lock object. De methode `wait()` zal de lock opgeven, en de draad laten wachten tot `notify()` wordt opgeroepen, die één van de wachtende draden opnieuw actief zal maken. De methode `notifyAll()` heeft dezelfde functionaliteit als `notify()`, alleen zullen bij `notifyAll()` alle wachtende draden actief gemaakt worden waarna ze opnieuw onderling strijden voor de lock, terwijl bij `notify()` slechts één draad actief zal worden. Om even bij stil te staan: waarom gebruiken we `while(!condition)` en niet `if(!condition)` ?

Opgave 5

Pas de `Main` klasse aan zodat na elke oproep van `buy()` eerst een tijd gewacht wordt alvorens de volgende bestelling te plaatsen met behulp van `Thread.sleep()`, zodanig dat de wachttijd tussen opeenvolgende bestellingen groter is dan de verwerkingstijd. Op momenten dat de `queue` leeg is, zal in de huidige implementatie de draad van de `EventBroker` de event queue continu pollen om te controleren of er nog geen event toegevoegd is, wat leidt tot 100% processorverbruik van één van de processor cores. Vermijd dit door gebruik te maken van het `wait()`/`notify()`-mechanisme. Zorg er ook voor dat wanneer alle customers hun orders geplaatst hebben en de EventBroker gestopt wordt, er geen onnodige rekentijd wordt besteed aan het actief wachten tot wanneer de queue leeg is.

6 Parallele orderverwerking

In grote applicaties zijn er vaak vele taken die je parallel wil uitvoeren om zo snelheidswinst te boeken. Het beheer van alle `Thread`s wordt dan complexer, en wanneer er teveel draden worden aangemaakt zullen deze allen strijden om processortijd te krijgen, waardoor de performantie zal dalen. Om dat probleem op te lossen, kan je gebruik maken van een threadpool. Dit is een collectie van draden die steeds hergebruikt kunnen worden en waarop je taken kan uitvoeren.

In Java kunnen threadpools eenvoudig geïmplementeerd worden met behulp van de `Executor` interface. Deze interface bevat één methode, namelijk `execute(Runnable r)`, die als argument een `Runnable` object meekrijgt en de `run()` methode zal oproepen om deze uit te voeren. De interface `ExecutorService` breidt `Executor` nog uit met extra methoden die ook toelaat taken uit te voeren die een resultaat terug geven met behulp van de `Future` interface.

Om gemakkelijk threadpools aan te maken voorziet Java ook de `Executors` klasse, die een aantal factory methoden bevat om verschillende types threadpools aan te maken.

```
ExecutorService threadpool = Executors.newCachedThreadPool();
for(int i=0;i<10;i++){
  Runnable task = new MyTask();
  threadpool.execute(task);
}
```

Opgave 6

In de huidige implementatie zullen de orders sequentieel verwerkt worden door de `OrderProcessor`. Breidt de implementatie van de klasse `OrderProcessor` uit, zodat de verwerking in parallel gebeurt met behulp van een threadpool. Gebruik hiervoor de `ExecutorService`. Wat is een goede grootte van de threadpool? Zorg dat deze threadpool ook correct wordt afgesloten wanneer alle orders verwerkt zijn.

7 Tot slot

De `EventBroker` uit opgave 5 dient als uitgangspunt voor de labsessie die handelt over GUI's (labsessie 4). Zorg dus dat je zeker een eigen werkende versie van de asynchrone `EventBroker` hebt, die conform de opgegeven UML-diagrammen en beschrijvingen functioneert.
Als in te dienen code verwachten we de (al dan niet aangepaste) startbestanden, samen met alle bestanden uit de package `eventbroker`, of meer concreet:

- package `main`
  - `Main.java`
- package `eventbroker`
  - `Event.java`
  - `EventBroker.java`
  - `EventListener.java`
  - `EventPublisher.java`
- package `order`
  - `BlacklistOrderProcessor.java`
  - `Customer.java`
  - `OrderEvent.java`
  - `OrderProcessor.java`

Om over na te denken:

- In opgave 5 wordt gebruik gemaakt van het `wait()`/`notify()`-mechanisme om de verwerkingsdraad te laten wachten wanneer er geen items in de queue zitten. Kan je een klasse vinden in de Java API die dit eenvoudiger maakt? (Tip: zoek naar implementaties van de `Queue` interface)