

11 Synchronization

Programmeren 2 – Java

2017 - 2018

KdG Karel de Grote
Hogeschool

Kris Behiels
Jan De Rijke
Mark Goovaerts

Programmeren 2 - Java

1. Herhaling en Collections
2. Generics en documenteren
3. Annotations en Reflection
4. Testen en logging
5. Design patterns (deel 1)
6. Design patterns (deel 2)
7. Lambda's en streams
8. Persistentie (JDBC)
9. XML en JSON
10. Threads
- 11. Synchronization**
12. Concurrency

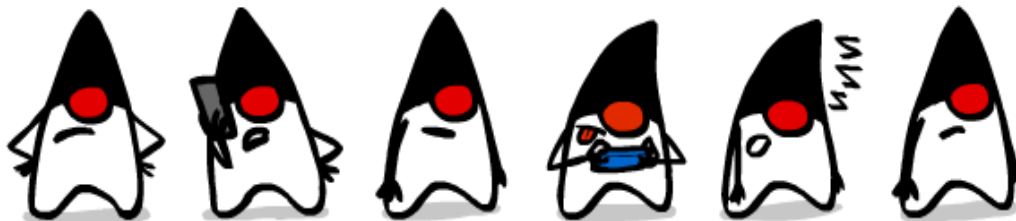


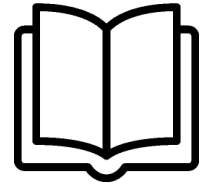
Agenda



De laatste 3 lesweken gaan over multi-threading en concurrent programming:

- W10:
 - Deel 1: **Threads**
- W11:
 - Deel 2: **Synchronization**
- W12:
 - Deel 3: **Concurrency**

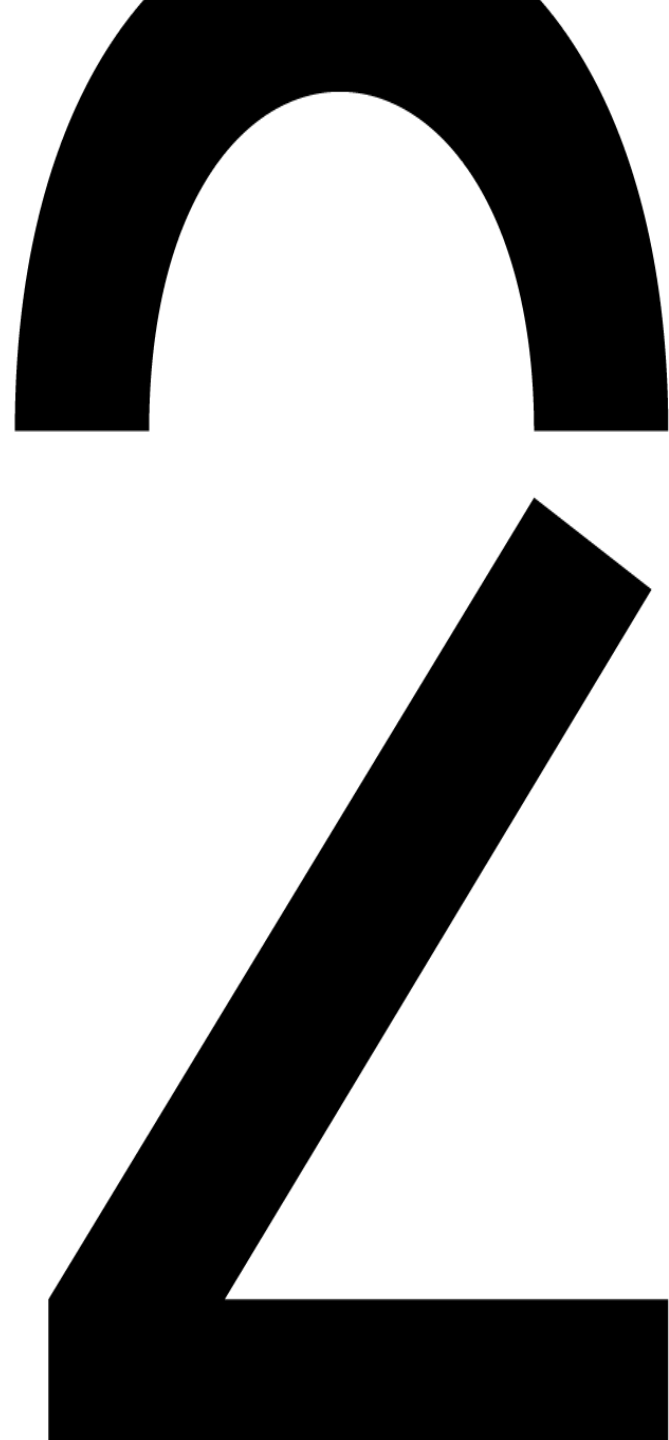




- E-book: "Concurrency" p.79 ev (Java How to Program, Tenth Edition)



Synchronization



Agenda

Deel 2: Synchronization

- Thread interferentie
- Geheugen consistentie fouten
- Synchronized
- Liveness
- Guarded Blocks
- Immutable Objects



```

public class Bankrekening {
    private String rekeningNummer;
    private int saldo;

    public Bankrekening(String reknr, int bedrag){
        rekeningNummer = reknr;
        saldo = bedrag;
    }

    public void geldOpname(int bedrag) throws InterruptedException {
        System.out.println("Geldopname te "
            + Thread.currentThread().getName()
            + " van rekening " + rekeningNummer);
        System.out.println("Bedrag: " + bedrag);

        if (bedrag <= saldo) {
            Thread.sleep(100); // tijd nodig voor de transactie
            saldo -= bedrag;
            System.out.println("Nieuw saldo: " + saldo);
        } else {
            System.out.println("Het saldo is te klein!");
        }
        System.out.println();
    }
}

```



Democode:
1_BankautomaatProbleem

Het bankautomaatprobleem

```
public class Automaat extends Thread {  
    private Bankrekening rekening;  
  
    public Automaat(String threadNaam, Bankrekening rekening) {  
        super(threadNaam);  
        this.rekening = rekening;  
    }  
  
    public void run() {  
        try {  
            rekening.geldOpname(500);  
            rekening.geldOpname(250);  
        } catch (InterruptedException e) {  
            // niet nodig  
        }  
    }  
}
```



Democode:
1_BankautomaatProbleem

Het bankautomaatprobleem

```
public class DemoBankrekening {  
    private static final int BEGINSALDO = 1325;  
  
    public static void main(String[] args) {  
        Bankrekening rekening =  
            new Bankrekening("BE26-3699-6941-1532", BEGINSALDO);  
  
        Automaat a1 = new Automaat("Antwerpen", rekening);  
        Automaat a2 = new Automaat("Gent", rekening);  
  
        System.out.println("Beginsaldo: " + BEGINSALDO + "\n");  
        a1.start();  
        a2.start();  
    }  
}
```



Het bankautomaatprobleem

- Output:

Beginsaldo: 1325

Geldopname te Antwerpen van rekening BE26-3699-6941-1532

Bedrag: 500

Geldopname te Gent van rekening BE26-3699-6941-1532

Bedrag: 500

Nieuw saldo: 825

Geldopname te Antwerpen van rekening BE26-3699-6941-1532

Bedrag: 250

Nieuw saldo: 325

Geldopname te Gent van rekening BE26-3699-6941-1532

Bedrag: 250

Nieuw saldo: 75

Nieuw saldo: -175

Saldo toch negatief
ondanks controle in
code? Verklaring?

Voorlopige vaststelling

- Naar aanleiding van het bankautomaat probleem:
- *OPGELET* als verschillende threads op één object inwerken
 - Ze sharen dezelfde resources
(dus ook dezelfde variabelen in het geheugen)
 - Ze hebben een apart ritme
- *DAAROM*: behoefte aan onderlinge afstemming, verkeersregels, communicatie
 - ***synchronization***



Problemen met threads

Op de volgende slides behandelen we 2 problemen:

1. Thread interferentie
2. Geheugen consistentie fouten



... en we trachten daarvoor een oplossing te vinden



1. Thread Interferentie



- **Interferentie** gebeurt als er 2 operaties in verschillende **threads** op dezelfde data worden uitgevoerd mekaar overlappen.
- Op het eerste gezicht kan dit zich bij de klasse Counter niet voordoen, maar ...

```
public class Counter {  
    private int c = 0;  
  
    public void increment() {  
        c++;  
    }  
  
    public void decrement() {  
        c--;  
    }  
  
    public int value() {  
        return c;  
    }  
}
```

1. Thread Interferentie



... als we de operatie **C++** nader bekijken bestaat die eigenlijk uit 3 stappen:

1. Haal de waarde van c op (in tijdelijke variabele)
2. Verhoog de waarde met 1
3. Plaats de waarde terug in c

Dus :

```
temp = c;  
temp = temp + 1;  
c = temp;
```

1. Thread Interferentie



Veronderstel dat Thread A en Thread B bijna op hetzelfde moment de increment en de decrement methode uitvoeren. We starten met $c = 0$.

- Thread A haalt de waarde van c op (temp_A = 0)
- Thread B haalt de waarde van c op (temp_B = 0)
- Thread A verhoogt de waarde met 1 (temp_A = 1)
- Thread B verlaagt de waarde met 1 (temp_B = -1)
- Thread A plaatst het resultaat in c (c = temp_A)
- Thread B plaatst het resultaat in c (c = temp_B)

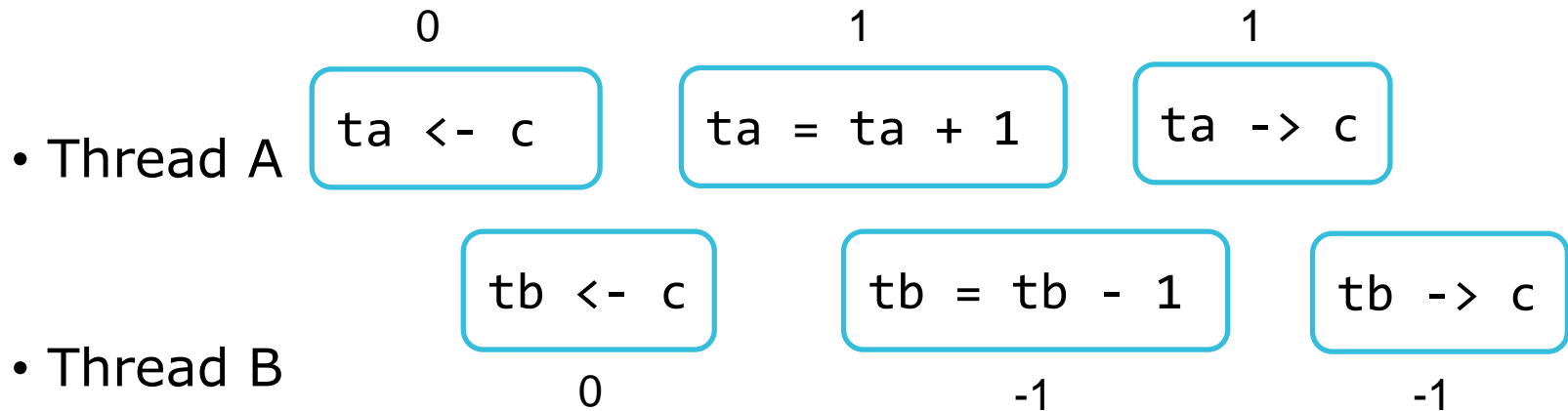
Het resultaat van Thread A gaat verloren, het wordt overschreven door Thread B.

We eindigen met een foutieve waarde: -1

1. Thread interferentie



- Visuele voorstelling:



Slides Opdracht 1



- Bestudeer de code van de module **2_DemoCounter** en voer ze verschillende keren uit. **Wat merk je op?**
- Hoe lossen we dit probleem op (m.a.w. hoe zorgen we ervoor dat de waarde van **value** op het einde altijd **0** is)?
 - Geen paniek als het niet lukt... zie verdere slides

2. Geheugen consistentie fouten



- Dit soort fouten treedt op als threads verschillende data zien waar de data dezelfde zou moeten zijn.

```
int counter = 0;  
counter++;  
System.out.println(counter);
```

- Als **counter** door 2 threads wordt gedeeld en de operaties in een afzonderlijke thread worden uitgevoerd is er geen garantie dat de afgedrukte waarde "1" is.
- Er treedt een zogenaamde **race condition** op, de print kan gebeuren op het moment dat de waarde van **counter** nog 0 is (of terug 0 is).

2. Geheugen consistentie fouten



```
public class Visibility {  
    private static boolean ready;  
    private static int number;  
  
    private static class ReaderThread extends Thread {  
        public void run() {  
            while (!ready) {  
                Thread.yield();  
            }  
            System.out.println(number);  
        }  
    }  
}
```

Hier wacht de **ReaderThread** tot de boolean **ready** **true** wordt.

2. Geheugen consistentie fouten



```
// in dezelfde klasse:  
public static void main(String[] args) {  
    new ReaderThread().start();  
    number = 42;  
    ready = true;  
}  
} // einde klasse Visibility
```

- Eerst start de **main**-thread, daarin start de **ReaderThread**, krijgt **number** z'n waarde en wordt de boolean **ready** op **true** gezet.
- De verwachte afdruk 42 zal echter niet altijd voorkomen, in sommige gevallen zal er 0 worden afgedrukt of zal het programma zelfs niet eindigen!

Synchronization

- Problemen:

- Thread interferentie
- Geheugen consistentie fouten



- **Oplossingen** die we deze week bespreken:

- synchronized
- wait, notify, notifyAll
- immutable



Synchronized Methods



```
public class Counter {  
    private int count = 0;  
  
    public synchronized void increment() { count++; }  
  
    public synchronized void decrement() { count--; }  
  
    public int value() { return count; }  
}
```

- Wanneer een thread een **synchronized** methode van een object uitvoert, worden alle andere threads die dit object willen benaderen in wacht gezet.
- Een **synchronized** methode kan dus maar door één thread tegelijk worden uitgevoerd.

Synchronized Methods



- Een constructor kan je niet **synchronized** maken, dit is ook niet nodig want alleen de thread die het object maakt mag toegang hebben tijdens de creatie ervan.
- **synchronized** zorgt alleen voor het uitsluiten van gelijktijdige toegang van verschillende threads **op één object**. Let dus goed op bij het gebruik ervan bij **static** methoden.

Slides Opdracht 2



- Bestudeer de code van de module **3_Share** en voer ze verschillende keren uit.

Wat merk je op? Altijd dezelfde output?

- Tracht ervoor te zorgen dat **ALTIJD** eerst alles van thread 1 en dan pas alles van thread 2 wordt uitgevoerd.
- Verwachte uitvoer:

```
Thread 1:  Dit
Thread 1:  is
Thread 1:  een
Thread 1:  demo
Thread 2:  Dit
Thread 2:  is
Thread 2:  een
Thread 2:  demo
```


Slides Opdracht 3



- Neem opnieuw de code van de module **2_DemoCounter** en los het probleem op met **synchronized**, de uitvoer moet nu altijd 0 zijn.
- Werk nu de module **2_DemoCounter_Lambda** uit, maak gebruik van lambda expressions (de klassen **DecrementRunnable** en **IncrementRunnable** heb je dan niet meer nodig). De klasse **Counter** moet/mag je niet wijzigen, vul alleen de klasse **TestCounter** aan.

Synchronized Statements



- In plaats van de volledige methode **synchronized** te maken kan je ook slechts een **deel ervan synchronized** maken. Zo kan je de performantie verbeteren.

```
public synchronized void increment() {  
    count++;  
}
```

is equivalent met:

```
public void increment() {  
    synchronized (this) {  
        count++;  
    }  
}
```

Als je een code blok **synchronized** wil maken, dan moet je een object opgeven dat tijdelijk *gelocked* zal worden. Hier: **this**

Synchronized Statements

SAFETY
FIRST

- In plaats van de volledige methode **synchronized** te maken kan je ook slechts een **deel ervan synchronized** maken. Zo kan je de prestatie verbeteren.

```
public synchronized void increment() {  
    count++;  
}
```

is equivalent met:

Als je **synchronized** in de header van de methode zet, dan gebeurt de lock ook op **this**

```
public void increment() {  
    synchronized (this) {  
        count++;  
    }  
}
```

Als je een code blok **synchronized** wil maken, dan moet je een object opgeven dat tijdelijk *geloocked* zal worden. Hier: **this**

Synchronized Statements



- Er zijn situaties waarbij het beter (of nodig) is om in een methode niet alles **synchronized** te maken.
- Als een **synchronized** methode een andere **synchronized** methode oproept kunnen **deadlocks** ontstaan (→ zie verder)

```
public void addName(String name) {  
    synchronized (this) {  
        lastName = name;  
        nameCount++;  
    }  
    nameList.add(name);  
}
```

Slides Opdracht 4



- Neem opnieuw de code van de module **1_BankautomatProbleem** en los het probleem op met **synchronized**
- Maak enkel het '*gevaarlijk*' gedeelte van de code **synchronized**
- Verwachte output:

```
...  
Geldopname te Gent van rekening BE26-3699-6941-1532  
Bedrag: 250  
Nieuw saldo: 75  
  
Het saldo is te klein!
```

Synchronized Statements

- By default wordt **synchronized** toegepast op de huidige instantie (**this**).
- Je mag **synchronized** ook toepassen op een ander object

```
public class Numbers {  
    private final List<Integer> myNumbers = new ArrayList<>();  
    public void voegToe(int nieuw) {  
        synchronized (myNumbers) {  
            myNumbers.add(nieuw);  
        }  
    }  
    public void verwijder(int oud) {  
        synchronized (myNumbers) {  
            myNumbers.remove(oud);  
        }  
    }  
}
```

GEVAARLIJK, want
Collections zijn niet
threadsafe!
→ zie volgende week...



Synchronized Statements


```
public class MyClass {  
    private int number;  
  
    public synchronized int getNumber() {  
        return number;  
    }  
  
    public synchronized void setNumber(int number) {  
        this.number = number;  
    }  
}
```

Is de bovenstaande klasse **thread-safe**?

Op het eerste gezicht wel, maar wat als we de klasse in het volgende programma gebruiken?

Synchronized Statements

```
public class MyRunnable implements Runnable {  
    private MyClass myclass;  
  
    public MyRunnable(MyClass myclass) {  
        this.myclass = myclass;  
    }  
  
    @Override  
    public void run() {  
        for (int i = 0; i < 10000; i++) {  
            myclass.setNumber(myclass.getNumber() + 1);  
        }  
    }  
}
```



In de **run**-methode
verhogen we de
getalwaarde 10000 keer
met 1.

Synchronized Statements

```
public class MyMain {  
    public static void main(String[] args) {  
        MyClass myClass = new MyClass();  
        Thread threadOne = new Thread(new MyRunnable(myClass));  
        Thread threadTwo = new Thread(new MyRunnable(myClass));  
  
        System.out.println("Starting threads...");  
        threadOne.start();  
        threadTwo.start();  
        try {  
            threadOne.join();  
            threadTwo.join();  
        } catch (InterruptedException e) {  
            // Leeg  
        }  
        System.out.println("Number: " + myClass.getNumber());  
    }  
}
```

Slides Opdracht 5



- Bestudeer de code van de module **4_SynchronizedWithObject** en voer ze verschillende keren uit.

Wat merk je op?

Hoe komt het dat de output niet altijd 20000 is?

Werkt **synchronized** hier dan niet correct?

Kan je het probleem oplossen?

Synchronized Statements

Indien we de voorgaande `main` uitvoeren verwachten we dat beide threads de waarde van getal elk voor zich 10000 keer met 1 verhoogd hebben. De afdruk moet dus 20000 zijn.

Toch zal dit eerder zelden het geval zijn. Hoe kan dit?

- We hebben alle methoden toch **synchronized** gemaakt?
- We hebben beide threads toch laten **joinen** zodat ze zeker gedaan zijn als de main thread het eindresultaat afdruckt?

Wat gaat er fout?

- We werken met 2 threads met 2 methoden tegelijk op hetzelfde attribuut!




Oplossing

```
public class MyRunnable implements Runnable {
    private static final Object LOCK = new Object();
    private MyClass myclass;

    public MyRunnable(MyClass myclass) {
        this.myclass = myclass;
    }

    @Override
    public void run() {
        for (int i = 0; i < 10000; i++) {
            synchronized (LOCK) {
                myclass.setNumber(myclass.getNumber() + 1);
            }
        }
    }
}
```

We doen de **lock** op een object dat gedeeld wordt door alle threads (want het is static)



Slides Opdracht 6



- Module **4_SynchronizedWithObject** : verwijder de **synchronized** instructies in **MyClass**
- Pas de code van de **Runnable** nu aan zoals in de voorgaande slide.
- Oplossing: Zorg ervoor dat er altijd op hetzelfde object gelockt wordt.
- Klaar? Maak een versie met een lambda expression, de klasse **MyRunnable** vervalt dan.

Agenda


Deel 2: Synchronization

- Thread interferentie
- Geheugen consistentie fouten
- Synchronized
- Liveness
- Guarded Blocks
- Immutable Objects



Liveness

- Tijdens de uitvoering van een concurrent applicatie kan er een ***liveness*** probleem optreden.
- Drie soorten:
 1. deadlock
 2. starvation
 3. livelock



A concurrent application's ability to execute in a timely manner is known as its ***liveness***.

1. Deadlock

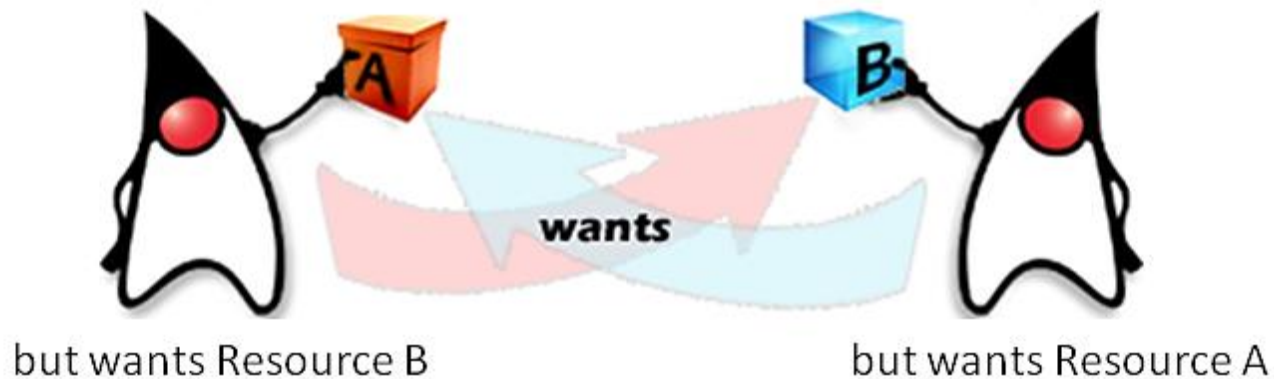


1. Deadlock

- Een situatie waarbij twee of meer threads geblokkeerd worden omdat ze op elkaar wachten.

Thread 1 is holding Resource A

Thread 2 is holding Resource B



Voorbeeld Deadlock

```
public class TestThread {  
    public static Object Lock1 = new Object();  
    public static Object Lock2 = new Object();  
  
    public static void main(String args[]) {  
        ThreadDemo1 T1 = new ThreadDemo1();  
        ThreadDemo2 T2 = new ThreadDemo2();  
        T1.start();  
        T2.start();  
    }  
}
```

// Vervolg op volgende slides...



Voorbeeld Deadlock

```
private static class ThreadDemo1 extends Thread {  
    public void run() {  
        synchronized (Lock1) {  
            System.out.println("Thread 1: Holding lock 1...");  
  
            try {  
                Thread.sleep(10);  
            } catch (InterruptedException e) {  
            }  
            System.out.println("Thread 1: Waiting for lock 2...");  
  
            synchronized (Lock2) {  
                System.out.println("Thread 1: Holding lock 1 & 2...");  
            }  
        }  
    }  
}
```

Thread1 doet lock op Lock1

Thread1 wil ook nog lock op Lock2



Voorbeeld Deadlock

```
private static class ThreadDemo2 extends Thread {  
    public void run() {  
        synchronized (Lock2) {  
            System.out.println("Thread 2: Holding lock 2...");  
  
            try {  
                Thread.sleep(10);  
            } catch (InterruptedException e) {  
            }  
  
            System.out.println("Thread 2: Waiting for lock 1...");  
  
            synchronized (Lock1) {  
                System.out.println("Thread 2: Holding lock 1 & 2...");  
            }  
        }  
    }  
}
```

Thread2 doet lock op Lock2

Thread2 wil ook nog lock op Lock1

```
Thread 1: Holding lock 1...  
Thread 2: Holding lock 2...  
Thread 2: Waiting for lock 1...  
Thread 1: Waiting for lock 2...
```



Slides Opdracht 7



- Een ander voorbeeld:

"Alphonse and Gaston are friends, and great believers in courtesy. A strict rule of courtesy is that when you bow to a friend, you must remain bowed until your friend has a chance to return the bow. Unfortunately, this rule does not account for the possibility that two friends might bow to each other at the same time."

- Bestudeer de code van de module **5_Deadlock_Friends** en voer verschillende keren uit.

Ben je absoluut zeker dat je altijd dezelfde uitvoer hebt?

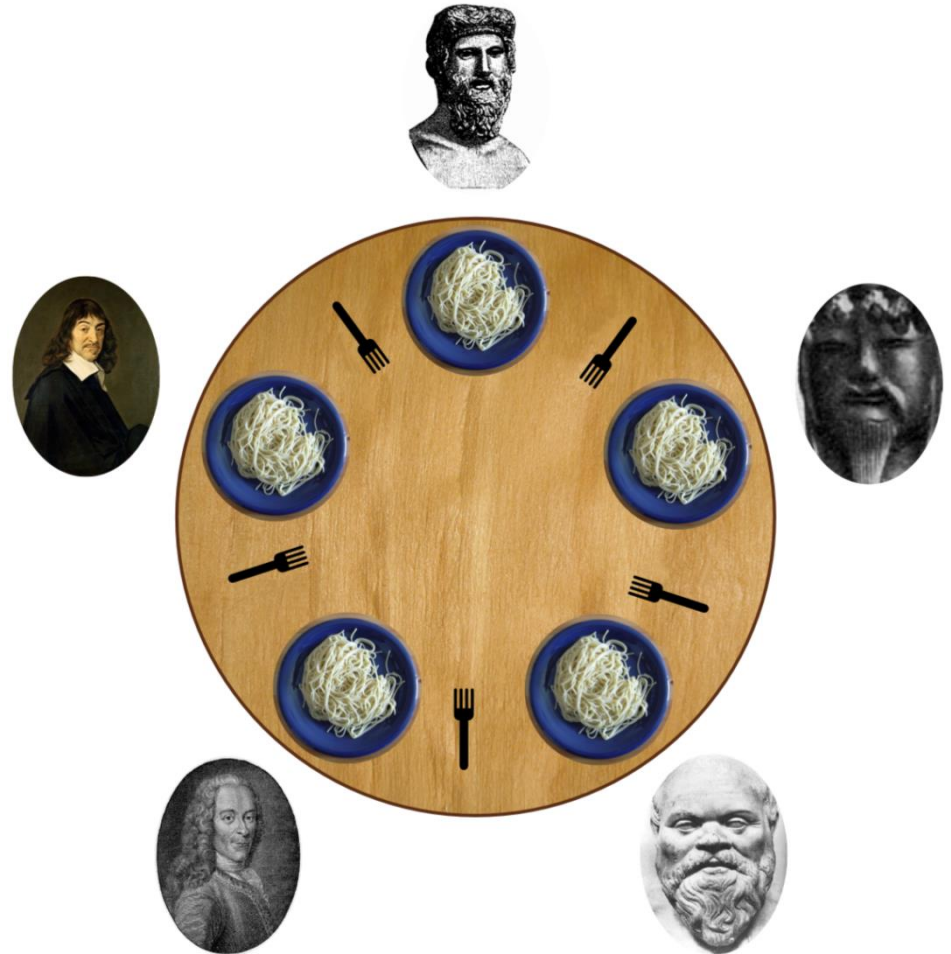
- Uitdagingen:

- **Minor** Kan je klasse BowRunnable wegwerken door met een lambda expression te werken?
- **Major** Kan je de deadlock op een of andere manier vermijden? (Zie volgende week)

2. Starvation

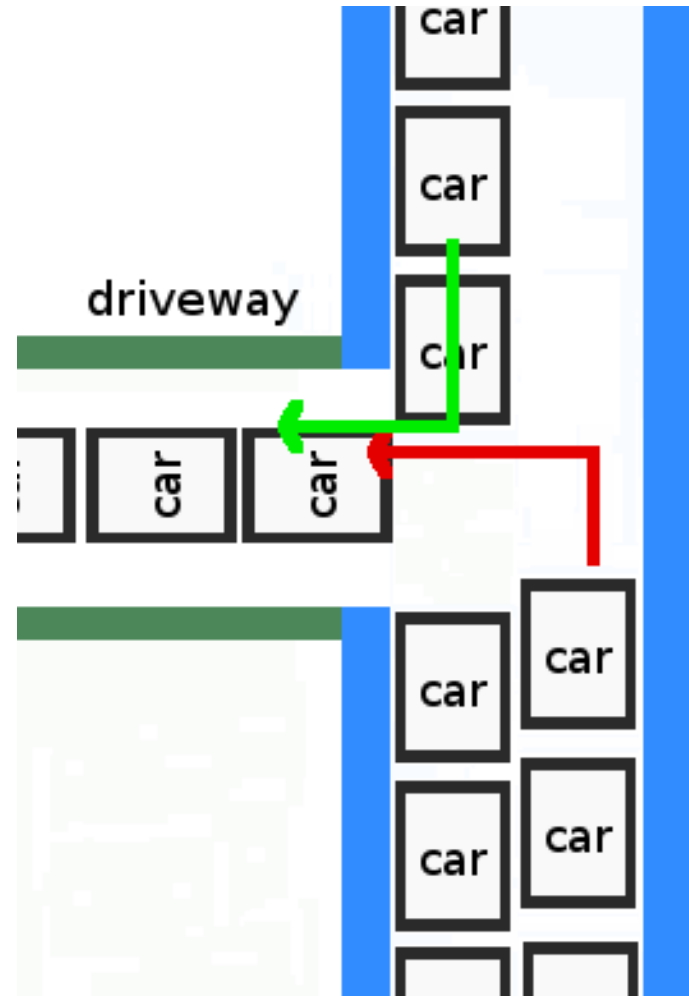
Het probleem van de "**dining philosophers**":

- Philosophers either eat or drink
- They must have 2 forks to eat
- Can only use forks on either side of their plate
- No talking!



2. Starvation

- Een situatie die optreedt wanneer een thread geen regelmatige toegang krijgt tot *gedeelde resources*, die door andere gulzige threads worden bezet gehouden.
- Dit gebeurt bijvoorbeeld als er threads lopen met *synchronized* methoden die lang duren, hierdoor worden de andere threads die ook toegang moeten krijgen geblokkeerd.



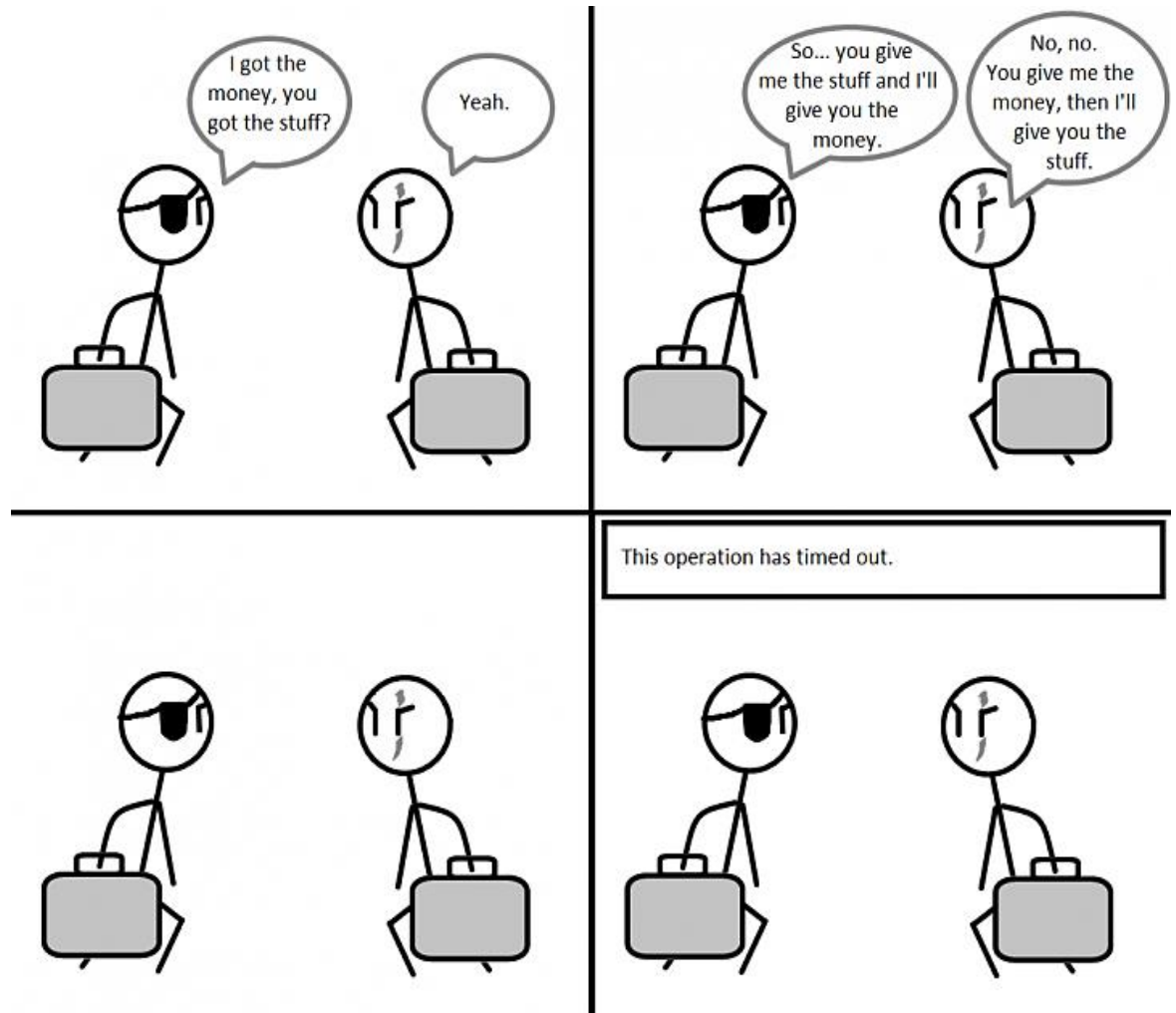
Slides Opdracht 8



- Bestudeer de code van de module **6_DiningPhilosophers** en voer ze verschillende keren uit.
Hoeveel filosofen kunnen eten?
Hoeveel filosofen sterven van honger?
- Uitdaging: Kan je de ***starvation*** op een of andere manier vermijden?

3. Livelock

- Is zoals een deadlock maar in dit geval geraken de threads zelf niet geblokkeerd.
- Bij een ***livelock*** reageren twee of meer threads zodanig op elkaar dat ze in een ***endless loop*** terecht komen.



Guarded Blocks

- Threads moeten dikwijls hun acties coördineren.
- Dit gebeurt meestal via een ***guarded block***.
- Zo'n guarded block wacht op een conditie die **true** moet zijn voordat het block verder kan gaan met de uitvoering ervan.
- Een typisch voorbeeld van het gebruik van een guarded block is een ***Producer-Consumer*** applicatie.

Guarded Blocks

- Een thread kan wachten op informatie van een andere thread. De klasse **Object** voorziet hiervoor de methoden **wait**, **notify** en **notifyAll**.

```
while (voorwaarde niet voldaan) {  
    try {  
        wait();  
    } catch (InterruptedException e) { }  
}
```

- De wachtlus kan beëindigd worden met behulp van de **notify** of de **notifyAll** methode.
 - notify** maakt één thread runnable die op dit object wacht
 - notifyAll** maakt alle threads runnable die op dit object wachten
- Je kan deze methoden enkel gebruiken in een **synchronized** blok!

Guarded Blocks – wait paradox



- Het gebruik van **wait** en **notify** is op het eerste zicht een contradictie. Want als wait wordt opgeroepen op een **synchronized** object, hoe kan een andere thread dan **notify** oproepen als die code **synchronized** is op hetzelfde object? Er kan toch maar één thread tegelijkertijd in een synchronized blok komen?
- De oplossing voor deze paradox: **de wait-methode heft deze blokkade tijdelijk op.**
Zo krijgen andere threads de mogelijkheid om **notify** uit te voeren. De thread in de **wait**-methode wordt terug runnable, maar moet wachten tot hij terug exclusieve toegang krijgt tot het synchronized blok (typisch als het **notify** blok gedaan is).

Slides Opdracht 9



- Bestudeer de code van de module **7_GuardedBlock**, let er vooral op hoe er vermeden wordt dat er threadproblemen kunnen optreden.
- Belangrijke opmerking: het is niet de bedoeling dat je later zo gaat programmeren, in de meeste gevallen zal je voor je probleem een bestaande klasse uit het Collections framework kunnen gebruiken die **thread-safe** is (zie volgende week).

Agenda

Deel 2: Synchronization

- Thread interferentie
- Geheugen consistentie fouten
- Synchronized
- Liveness
- Guarded Blocks
- Immutable Objects



Immutable Objects

- Een ***immutable*** object is een object dat eens gemaakt niet meer van toestand kan veranderen (bv **String**-object).
- Zinvol in concurrent applicaties → geen thread interferentie mogelijk.
- Performantie?
 - Synchroniseren vraagt extra code en tijd.
 - Wel meer objecten te maken in geheugen
 - Garbage collection geeft de vele niet-gebruikte objecten weer vrij.

Immutable Objects

- Regels om objecten **immutable** te maken:
 - Voorzie geen "setters"
 - Maak alle attributen **final** en **private**
 - Laat subklassen geen methoden overriden
 - Maak de klasse **final** of gebruik het factory method pattern (**private** constructor)
 - Als er mutable attributen zijn:
 - Voorzie dan geen methoden die ze wijzigen
 - Maak kopies van mutable objecten die je binnenkrijgt en sla referenties naar die kopies op
 - Geef in getters geen referentie naar interne mutable objecten terug, maar geef een kopie van het object terug

Voorbeeld: immutable klasse

```
public final class DwergPlaneet {  
    private final int diameter;  
    private final String naam;  
    private final Ontdekker ontdekker;  
  
    public DwergPlaneet(int diameter, String naam, Ontdekker ontdekker) {  
        this.diameter = diameter;  
        this.naam = naam;  
        this.ontdekker = new Ontdekker(ontdekker.getNaamOntdekker());  
    }  
  
    public int getDiameter() { return diameter; }  
  
    public String getNaam() { return naam; }  
  
    public Ontdekker getOntdekker() {  
        return new Ontdekker(ontdekker.getNaamOntdekker());  
    }  
}
```

final!

Een primitief type mag gewoon overgenomen worden, want: *call by value*

Strings zijn vanzelf al immutable, geen probleem

Elk ander object (hier van de klasse **Ontdekker**): opnieuw instantiëren!

Bij **return**: eerst een nieuwe instantie maken!

zeker **geen setters!**

Slides Opdracht 10



- Bestudeer de code van de module **8_Immutable**
- Doe vervolgens de nodige aanpassingen in de klasse `Person` zodat deze klasse *immutable* wordt.
- De klasse **Adress** mag je niet wijzigen.
- Run en controleer de uitvoer...

Na aanmaak:

Donald Trump (age 71)

Adress: Pennsylvania Avenue, 1600 Washington DC

Friends: [Melania Trump, Mike Pence, Jared Kushner]

Na poging tot wijziging:

Donald Trump (age 71)

Adress: Pennsylvania Avenue, 1600 Washington DC

Friends: [Melania Trump, Mike Pence, Jared Kushner]

Opdrachten



- Groeiproject
 - module 9
(deze week enkel deel 3: "Synchronization")
- Opdrachten op BB
 - Dubbeltellen
 - Carwash
 - CarwashDuo
 - Deadlock
 - Overdruk
- Zelftest op BB

