



# Multithreading (deel 2)



**DE HOGESCHOOL  
MET HET NETWERK**

Hogeschool PXL – Elfde-Liniestraat 24 – B-3500 Hasselt  
[www.pxl.be](http://www.pxl.be) - [www.pxl.be/facebook](https://www.pxl.be/facebook)



# Multithreading – deel 2

1. Synchronisatie
2. TimerTask
3. Concurrency
4. Parallelisme met streams



# 1. Synchronisatie

Verschillende threads op asynchrone wijze toegang tot hetzelfde object

- Object locking
  - synchronized
  - thread state BLOCKED



# 1. Synchronisatie - Object locking

```
public class Koekjesdoos {  
    private int aantalKoekjes;  
  
    public Koekjesdoos(int aantalKoekjes) {  
        this.aantalKoekjes = aantalKoekjes;  
    }  
  
    public boolean neemKoekje() {  
        if (aantalKoekjes > 0) {  
            aantalKoekjes--;  
            return true;  
        }  
        return false;  
    }  
}
```



```

public class Kind extends Thread {
    private int aantalKoekjes;
    private Koekjesdoos koekjesdoos;
    private String naam;

    public Kind(String naam, Koekjesdoos koekjesdoos) {
        this.koekjesdoos = koekjesdoos;
        this.naam = naam;
    }

    @Override
    public void run() {
        while (koekjesdoos.neemKoekje()) {
            aantalKoekjes++;
            try {
                Thread.sleep(5);
            } catch (InterruptedException e) {
                e.printStackTrace();
            }
        }
        System.out.println(naam + " at " + aantalKoekjes + " koekjes" );
    }

    public int getAantalKoekjes() {
        return aantalKoekjes;
    }
}

```

```
public class KoekjesEten {
    public static void main(String[] args) {
        Koekjesdoos koekjesdoos = new Koekjesdoos(50);
        Kind[] kinderen = {
            new Kind("Bram", koekjesdoos),
            new Kind("Sophie", koekjesdoos),
            new Kind("Elke", koekjesdoos),
            new Kind("Robin", koekjesdoos),
            new Kind("Sammy", koekjesdoos),
            new Kind("Max", koekjesdoos)};
        for (int i = 0; i < kinderen.length; i++) {
            kinderen[i].start();
        }
        for (int i = 0; i < kinderen.length; i++) {
            try {
                kinderen[i].join();
            } catch (InterruptedException e) {
                e.printStackTrace();
            }
        }
        System.out.println("De kinderen aten: " +
            Arrays.stream(kinderen)
                .mapToInt(kind -> kind.getAantalKoekjes())
                .sum());
    }
}
```

```
KoekjesEten x
"C:\Program Files\Java\jdk-11\bin\java.exe" "-ja
Bram at 9 koekjes
Sophie at 9 koekjes
Max at 9 koekjes
Elke at 9 koekjes
Martien at 9 koekjes
Robin at 9 koekjes
De kinderen aten: 54

Process finished with exit code 0
```

aantalKoekjes = 1

neemKoekje()

if (aantalKoekjes > 0)

neemKoekje()

if (aantalKoekjes > 0)

aantalKoekjes--

[aantalKoekjes -> 0]

aantalKoekjes--

[aantalKoekjes -> -1]





```
KoekjesEten x
"C:\Program Files\Java\jdk-11\bin\java.exe" "-ja
Bram at 9 koekjes
Sophie at 9 koekjes
Max at 9 koekjes
Elke at 9 koekjes
Martien at 9 koekjes
Robin at 9 koekjes
De kinderen aten: 54

Process finished with exit code 0
```

```
KoekjesEten x
"C:\Program Files\Java\jd
Elke at 8 koekjes
Robin at 8 koekjes
Martien at 9 koekjes
Max at 9 koekjes
Sophie at 8 koekjes
Bram at 8 koekjes
De kinderen aten: 50
```

```
public class Koekjesdoos {
    private int aantalKoekjes;

    public Koekjesdoos(int aantalKoekjes) {
        this.aantalKoekjes = aantalKoekjes;
    }

    public synchronized boolean neemKoekje() {
        if (aantalKoekjes > 0) {
            aantalKoekjes--;
            return true;
        }
        return false;
    }
}
```

## 2. Timer en TimerTask

- Objecten van de `Timer`-klasse voeren een taak uit na een bepaalde tijd.
- De uit te voeren taak wordt omschreven door een object van de klasse `TimerTask`.
  - Leidt een nieuwe klasse af en vervang de methode `run()`



## 2. Timer en TimerTask

```
public class RepeatTask {  
    public static void main(String[] args) {  
        TimerTask repeatedTask = new TimerTask() {  
            public void run() {  
                System.out.println("Task performed on " +  
                                   LocalDateTime.now());  
            }  
        };  
        Timer timer = new Timer("Timer");  
        long delay = 5000L;  
        long period = 10000L;  
        timer.scheduleAtFixedRate(repeatedTask, delay, period);  
        System.out.println("Timer started " +  
                           LocalDateTime.now());  
    }  
}
```



# 3. Concurrency framework

- Dient om het ontwikkelen van *multithreaded* applicaties makkelijker te maken.
- `java.util.concurrent`
  - Concurrent collections
  - Atomaire objecten
  - Callable, ExecutorService and Future



# 3. Concurrent collections

`java.util.Collections`

<i><b>Methode</b></i>
<code>synchronizedCollection()</code>
<code>synchronizedList()</code>
<code>synchronizedNavigableMap()</code>
<code>synchronizedNavigableSet()</code>
<code>synchronizedSet()</code>
<code>synchronizedSortedMap()</code>
<code>synchronizedSortedSet()</code>



### 3. Concurrency – atomaire objecten

`java.util.concurrent.atomic`

Klasse	Omschrijving
<code>AtomicBoolean</code>	Atomaire boolean.
<code>AtomicInteger</code>	Atomaire integer.
<code>AtomicIntegerArray</code>	Atomarie reeks van integers.
<code>AtomicLong</code>	Atomaire long.
<code>AtomicLongArray</code>	Atomaire reeks van longs.
<code>AtomicReference</code>	Atomaire referentie.
<code>AtomicReferenceArray</code>	Atomare reeks van referenties.

- Voorbeeld Counter
- Opdracht 10 pagina 207



# 3. Concurrency

## Callable, ExecutorService and Future

- Klassen en interfaces die abstracFe maken van de *low level* details van het omgaan met *threads*



# 3. Concurrency

## Callable, ExecutorService and Future

`ExecutorService` voert een `Callable`-object uit en geeft een `Future`-object terug.





# 3. Concurrency

## Callable, ExecutorService and Future

- `Callable`: taak die uitgevoerd moet worden
  - Implementeer `call()` methode
- `ExecutorService`: voert taken uit.
  - instantie aanmaken met methoden van `Executors`
    - Vb: `newSingleThreadExecutor()`
  - kan taken uitvoeren via `submit()`



# 3. Concurrency

## Callable, ExecutorService and Future

- `Future`: resultaat ligt in de toekomst, berekening wordt uitgevoerd in een andere *thread*.
  - `isDone()`: nagaan of berekening uitgevoerd is.
  - `get()`: ophalen van het resultaat.
    - Return-type komt overeen met het generieke datatype van de interface `Callable`



# 3. Concurrency

## Callable, ExecutorService and Future

- Opdracht 11 pagine 210 : Faculteit berekenen



# 4. Parallel streams

```
public class ParallellStreams {
    public static void main(String[] args) {
        List<Employee> employees = new ArrayList<Employee>();
        for (int i = 0; i < 100; i++) {
            employees.add(new Employee("A", 20000));
            employees.add(new Employee("B", 3000));
            employees.add(new Employee("C", 15002));
            employees.add(new Employee("D", 7856));
            employees.add(new Employee("E", 200));
            employees.add(new Employee("F", 50000));
        }
        long t1 = System.currentTimeMillis();
        System.out.println("Sequential Stream Count?= " +
            employees.stream().filter(e -> e.getSalary() > 15000).count());
        long t2 = System.currentTimeMillis();
        System.out.println("Sequential Stream Time Taken?= " + (t2 - t1) + "\n");

        t1 = System.currentTimeMillis();
        System.out.println("Parallel Stream Count?= " +
            employees.parallelStream().filter(e -> e.getSalary() > 15000).count());
        t2 = System.currentTimeMillis();
        System.out.println("Parallel Stream Time Taken?= " + (t2 - t1));
    }
}
```

## 4. Parallel streams

- Ter herinnering: Een *stream* is een stroom van gegevens die vloeit uit een verzameling en waar bewerkingen op gedaan kunnen worden.
- 4 mogelijke bewerkingen:
  - Consumeren
  - Filteren
  - Reduceren
  - Collecteren



## 4. Parallel streams

- *Streams* worden door één enkele thread verwerkt.
  - Elementen worden één voor één behandeld
- Het is ook mogelijk om meerdere *threads* te gebruiken die elk een gedeelte van de *stream* voor hun rekening nemen.
  - `Collection.parallelStream()` (voor een nieuwe *stream*)
  - `parallel()` (om een bestaande *stream* om te zetten)



# 4. Parallel streams

```
public class ParallellStreams {
    public static void main(String[] args) {
        List<Employee> employees = new ArrayList<Employee>();
        for (int i = 0; i < 100; i++) {
            employees.add(new Employee("A", 20000));
            employees.add(new Employee("B", 3000));
            employees.add(new Employee("C", 15002));
            employees.add(new Employee("D", 7856));
            employees.add(new Employee("E", 200));
            employees.add(new Employee("F", 50000));
        }
        long t1 = System.currentTimeMillis();
        System.out.println("Sequential Stream Count?= " +
            employees.stream().filter(e -> e.getSalary() > 15000).count());
        long t2 = System.currentTimeMillis();
        System.out.println("Sequential Stream Time Taken?= " + (t2 - t1) + "\n");

        t1 = System.currentTimeMillis();
        System.out.println("Parallel Stream Count?= " +
            employees.parallelStream().filter(e -> e.getSalary() > 15000).count());
        t2 = System.currentTimeMillis();
        System.out.println("Parallel Stream Time Taken?= " + (t2 - t1));
    }
}
```