

# 12 Concurrency

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

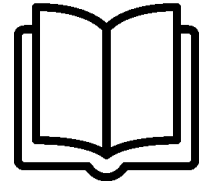
---



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)



---

Concurrency

# Agenda Deel 3: Concurrency

---

- Lock Objects
- Thread interactie en management
  - Executors en Thread pools
  - Callable
  - CompletableFuture
- Fork/Join
- Consistente data
  - Atomic Variables
  - Concurrent Collections
- Parallelle Streams
- Conclusies




# Inleiding

---

- Het aansturen van threads (via **synchronized**, **wait**, **notify**, **join**, **yield**) is relatief eenvoudig maar beperkt en primitief.
- Sinds JDK 5.0: high-level concurrency features:
  - meer geavanceerd en veiliger
  - Betere tools om multi-core processoren optimaal te benutten
- Overzicht:
  - **Lock** objecten voor synchronizeren van threads
  - **Executors** voor thread pool management
  - **Atomic** variabelen voorkomen memory consistency errors.
  - **Concurrent collections** om veilig met grote hoeveelheden data te werken

# Lock objecten

---

- **synchronized** = eenvoudig lock-systeem
- Geavanceerdere lock-systemen in **java.util.concurrent.locks**
- De belangrijkste interface is de **Lock** interface
  - slechts één thread tegelijk toegestaan op een object (zoals **synchronized**)
  - ondersteunt een **wait/notify** mechanisme met **Condition** objecten (Lock wacht tot een Condition vervuld is)
  - **tryLock** lockt alleen als dat mogelijk is (anders: keert terug of wacht een opgegeven tijdsinterval)

Eerste hulp bij  
bestrijding van  
deadlocks!



# Locks package

---

- Interfaces:

- Condition

- Lock**

- ReadWriteLock

- Klassen (belangrijkste):

- ReentrantLock**

- ReentrantReadWriteLock.ReadLock

- ReentrantReadWriteLock.WriteLock

**Lock tov synchronized:**

- kan onderbroken worden
- mogelijkheid timed waiting
- "locking without blocking"
- expliciete unlock
- flexibeler, extra features...

- Methoden en meer info: zie Java documentatie

# Blokkeren met synchronized

```
public class SynchronizedLockExample implements Runnable {  
    private final Resource resource;  
  
    public SynchronizedLockExample(Resource resource) {  
        this.resource = resource;  
    }  
  
    @Override  
    public void run() {  
        synchronized (resource) {  
            resource.doSomething();  
        }  
        resource.doLogging();  
    }  
}
```

Resource-object wordt geblokkeerd.  
Kans op deadlock!

# Blokkeren met ReentrantLock

```
public class ConcurrencyLockExample implements Runnable {  
    private final Resource resource;  
    private Lock lock;  
  
    public ConcurrencyLockExample(Resource resource){  
        this.resource = resource;  
        this.lock = new ReentrantLock();  
    }  
  
    public void run() {  
        try {  
            if(lock.tryLock(10, TimeUnit.SECONDS)){  
                resource.doSomething();  
            }  
        } catch (InterruptedException e) {  
            e.printStackTrace();  
        } finally{  
            lock.unlock();  
        }  
        resource.doLogging();  
    }  
}
```

Probeer gedurende 10 seconden lock in te stellen.  
Zonder parameters: return onmiddellijk true/false

Expliciet lock opheffen

# Slides Opdracht 1



- Bestudeer de code van de module **1\_ReentrantLock** en voer ze verschillende keren uit.

Wat merk je op? Altijd hetzelfde verloop?

Hoe wordt deadlock vermeden?

- In de klasse **Friend** staat een try zonder catch maar wel finally.

Is dit zinvol?

Wat is de bedoeling?

# Agenda Deel 3: Concurrency

---



- Lock Objects
- Thread interactie en management
  - Executors en Thread pools
  - Callable
  - CompletableFuture
- Fork/Join
- Consistente data
  - Atomic Variables
  - Concurrent Collections
- Parallelle Streams
- Conclusies

# Executors

---

- Tot nu toe:
  - Nauwe 1-1 band tussen taak (Runnable) en uitvoering (Thread).
  - OK bij eenvoudige toepassingen
- Bij grotere applicaties:
  - Scheiding threadmanagement en creatie
    - creatie is duur
  - Objecten die hiervoor zorgen noemen we **Executors**

# Executor Interfaces

---

`java.util.concurrent` bevat 3 **Executor interfaces**:

–**Executor**

- eenvoudige interface met één methode: `execute`

–**ExecutorService** extends `Executor`

- extra methoden om de lifecycle te managen:  
`submit`, `invokeAny`, `invokeAll`, `shutdown`, ...
- werkt met `Runnable` of `Callable` objecten

–**ScheduledExecutorService** extends `ExecutorService`

- idem, maar start pas na een opgegeven tijdsinterval, of herhalend met periodieke intervals

# Executor Interface

---

- Slechts één methode: **execute**.
  - Te vergelijken met **start** methode
  - Voorbeeld waarbij **r** een **Runnable** object is en **e** een **Executor** object:

**e.execute(r);**

Zelfde als:  
**new Thread(r).start();**

- Voordeel van **Executor**:
  - Je kan een **Executor** *hergebruiken* om een volgende **Runnable** uit te voeren, of de **Runnable** kan in een *wachtrij* geplaatst worden tot er een **Executor** beschikbaar is.



# Thread Pools

- De meeste **Executor** implementaties maken gebruik van **thread pools**, die bestaan uit *worker threads*
- Door **worker threads** te hergebruiken is er minder geheugenruimte nodig (er worden minder threads gecreëerd)
- Het meest gebruikte systeem is een **fixed thread pool**. Hierbij zijn er altijd een vast aantal threads in uitvoering.  
Een interne queue zorgt voor het management van binnenkomende taken.



# Thread Pools

---

De klasse **Executors** voorziet onder andere volgende *static factory methods* voor de creatie van verschillende soorten thread pools:

## –newFixedThreadPool

- een pool met een vast aantal threads

## –newCachedThreadPool

- een uitbreidbare pool (voor toepassingen met veel korte threads)

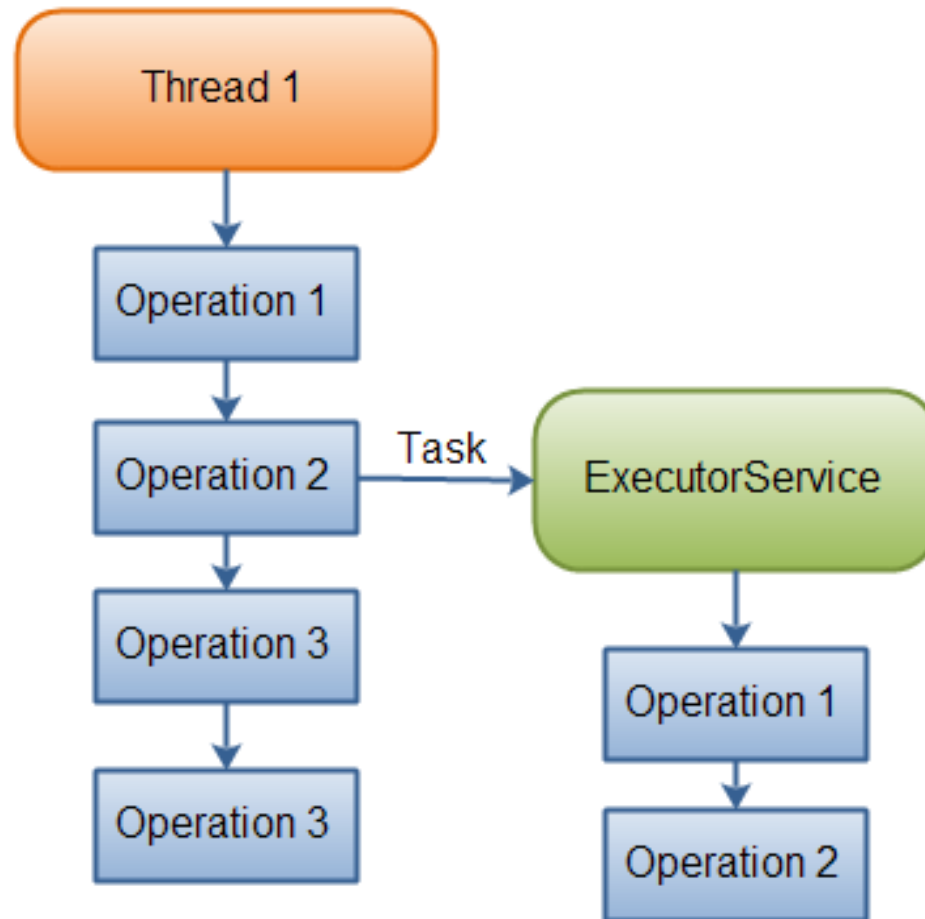
## –newSingleThreadExecutor

- een executor die slechts één taak tegelijk uitvoert

## –newScheduledThreadPool

- een fixed size thread pool met mogelijkheden voor uitgestelde start of periodieke uitvoering (vergelijkbaar met Timer)

# Asynchrone taak delegatie



**A thread delegating a task to an ExecutorService for asynchronous execution.**

# Voorbeeld FixedThreadPool

```
public class SimpleThreadPool {  
    public static void main(String[] args) {  
        ExecutorService executor =  
            Executors.newFixedThreadPool(5);  
        for (int i = 0; i < 10; i++) {  
            Runnable worker = new WorkerThread("" + i);  
            executor.execute(worker);  
        }  
  
        executor.shutdown();  
        try {  
            executor.awaitTermination(5, TimeUnit.MINUTES);  
        } catch (InterruptedException e) {  
            System.err.println("Executor onderbroken: " + e);  
        }  
        System.out.println("Finished all threads");  
    }  
}
```

Een threadpool met 5 worker threads

Worker threads worden opgestart

*awaitTermination* een beetje zoals *join*...  
wacht tot alle threads gedaan zijn om "Finished..." te printen, normaal is *shutdown()* voldoende!

## Slides Opdracht 2



- Bestudeer de code van de module **2\_Executors** en voer ze verschillende keren uit.  
Merk op dat er 5 worker threads simultaan lopen om de 10 jobs uit te voeren.
- Verander de code, zodat er in de pool slechts 3 worker threads zijn.
- Raadpleeg de documentatie: wat doet de methode shutdown?
- Zet `executor.shutdown()` en de `try/catch` in commentaar; wat is daarvan het gevolg?

# Callable

---

- Nadeel **Runnable**:
  - kan geen return doen
  - kan geen exception gooien
- **Callable** is een alternatief voor **Runnable**:

```
public interface Callable<V> {  
    V call() throws Exception;  
}
```

← <V> staat voor Value

← call i.p.v. run

- Uit te voeren met **ExecutorService** → **submit**:

```
<V> Future<V> submit(Callable<V> task);
```

← **submit** i.p.v.  
execute

- De **get()** methode van het **Future** object geeft de returnwaarde van de **Callable** en doet tegelijk hetzelfde als een **join()**

## Callable (voorbeeld 1)

In dit voorbeeld berekenen we de individuele lengte van alle woorden en tellen ze samen.

```
public class WordLengthCallable implements Callable<Integer> {  
    private String word;  
  
    public WordLengthCallable(String word) {  
        this.word = word;  
    }  
  
    public Integer call() throws Exception {  
        return word.length();  
    }  
}
```

zoals methode **run** bij **Runnable**

# Callable (voorbeeld 1)

```
public class CallableDemo {  
    private static final String[] words = {"red", "green", "blue"};  
  
    public static void main(String[] args) throws Exception {  
        ExecutorService pool = Executors.newFixedThreadPool(3);  
        List<Future<Integer>> list = new ArrayList<>();  
        for (String word : words) {  
            Callable<Integer> callable = new WordLengthCallable(word);  
            Future<Integer> future = pool.submit(callable);  
            list.add(future);  
        }  
        int sum = 0;  
        for (Future<Integer> future : list) {  
            sum += future.get();  
        }  
        System.out.printf("Het aantal letters is %s%n", sum);  
        pool.shutdown();  
    }  
}
```

join en opvragen  
returnwaarden bij  
beëindigen threads

*The sum of lengths is 12*



## Callable (voorbeeld 2)

```
public class MyCallable {  
    public static void main(String[] args) {  
        List<Integer> integers = Arrays.asList(1, 2, 3, 4, 5, 6, 7, 8, 9);  
        Callable<Integer> callable =  
            () -> integers.stream().mapToInt(i -> i).sum();  
        ExecutorService service = Executors.newSingleThreadExecutor();  
        Future<Integer> future = service.submit(callable);  
        Integer result = 0;  
        try {  
            result = future.get();  
        } catch (InterruptedException | ExecutionException e) {  
            e.printStackTrace();  
        }  
        System.out.println("Sum = " + result);  
        service.shutdown();  
    }  
}
```

**Sum = 45**

In dit voorbeeld maken we gebruik van een lambda expression om de som van alle getallen te berekenen.



## Slides Opdracht 3



- Bestudeer de code van de module **3\_Callable** en voer ze uit.
- Pas aan zodat ook het aantal woorden geteld wordt. Output:

```
The sum of lengths is 36  
The number of words is 9
```

- Maak een lambda **Callable** die de omgekeerde string teruggeeft. Probeer ze uit vanuit de main:

```
eird nevez sez negen fjiv nee eewt reiv thca
```

# CompletableFuture

---

- Voor het resultaat van een Future moet je blokkeren (`get()`) of pollen (steeds opnieuw checken met `isDone()`)
- Onhandig, zeker als je het resultaat wil gebruiken voor verdere (asynchrone) acties
  - Voorbeeld: opeenvolgende requests naar een cloud server
- **CompletableFuture** heeft extra methoden om, wanneer het resultaat beschikbaar is, een nieuwe taak te starten.
  - De methoden die op Async eindigen runnen de taak in (weer) een nieuwe thread
  - De andere methoden runnen de taak in dezelfde thread (synchroon)

# CompletableFuture

---

- De nieuwe taak die op de CompletableFuture start, kan als (asynchroon) resultaat een nieuwe CompletableFuture geven
- Hierop kan je wéér een nieuwe taak starten, enz.
- Zo kan je een ketting van opeenvolgende asynchrone taken vormen

**=> Functionele (vloeiende) stijl met asynchrone taken (elk in een nieuwe thread)!**

# Voorbeeld 1: Wachten op resultaat

---

- In een eerste voorbeeld gebruiken we de `CompletableFuture` als een gewone `Future`: we halen het resultaat op met `get()`
- De thread wordt gestart via  
`CompletableFuture::supplyAsync`
  - De taak die meegegeven wordt is een `Supplier`
    - In het voorbeeld is de `Supplier` een lambda een `List<String>` omzet in hoofdletters en als een `Stream<String>` teruggeeft

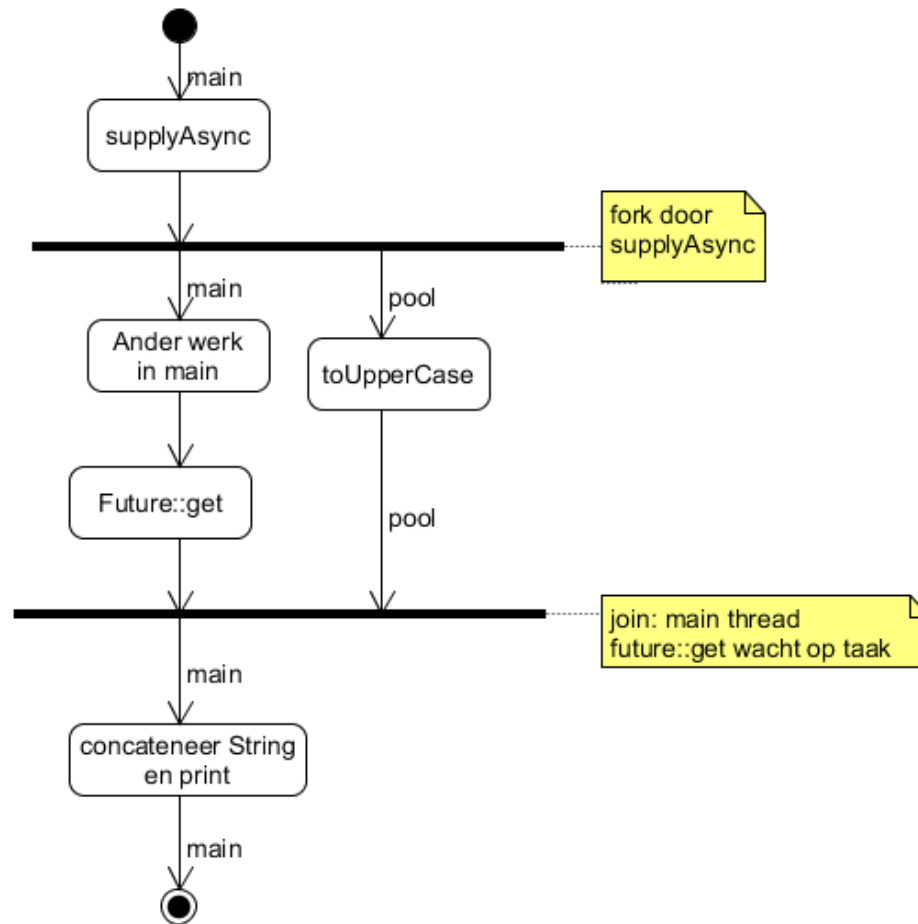


## Voorbeeld 1: Wachten op resultaat

```
public class OneStepCompletable {  
    private static final String[] words = {"red", "green", "blue"};  
  
    public static void main(String[] args) throws Exception {  
        // zet words om naar een woorden stream in hoofdletters  
        // in een nieuwe thread (Async)  
        CompletableFuture<Stream<String>> upperFuture  
            = CompletableFuture.supplyAsync(  
                () -> Arrays.stream(words).map(word -> word.toUpperCase());  
            );  
        System.out.println("Thread main werkt verder tijdens omzetten.");  
        // de get() methode blokkeert main tot de async thread gedaan is  
        System.out.println("Omzetten words in hoofdletters is gedaan: " +  
            upperFuture.get().collect(Collectors.joining(" ")));  
        System.out.println("Einde main thread.");  
    }  
}
```

```
Thread main werkt verder tijdens omzetten.  
Omzetten words in hoofdletters is gedaan: RED GREEN BLUE  
Einde main thread..
```

# Voorbeeld 1: Wachten op resultaat



## Voorbeeld 2: Actie op resultaat

---

- We passen nu het voorbeeld aan om op het resultaat van de Thread (`CompletableFuture` met `Stream<String>`) een nieuwe taak uit te voeren (Strings joinen en afdrukken).
- Aanpassingen in vorig voorbeeld
  - Geen `Future::get` (en geen throws `Exception`)
  - We roepen op de `CompletableFuture` `thenAccept` aan en werken daar verder met het resultaat
  - We geven aan `CompletableFuture::supplyAsync` onze eigen `ExecutorService` mee
    - default runt de `CompletableFuture` op daemon threads uit de fork/join pool (zie verder)





## Voorbeeld 2: Actie op resultaat

```

public class ThenApplyCompletable {
    private static final String[] words = {"red", "green", "blue"};

    public static void main(String[] args) {
        ExecutorService pool = Executors.newFixedThreadPool(10);
        CompletableFuture.supplyAsync(
            () -> Arrays.stream(words).map(word -> word.toUpperCase()),
            pool)
            .thenAccept(result -> System.out.println(
                "Omzetten words in hoofdletters is gedaan: "
                + result.collect(Collectors.joining(" ")))));
        System.out.println("Thread main werkt verder tijdens omzetten.");
        System.out.println("Einde main thread.");
        pool.shutdown();
    }
}

```

Geen throws Exception

Eindigt direct

Wacht op einde supplyAsync

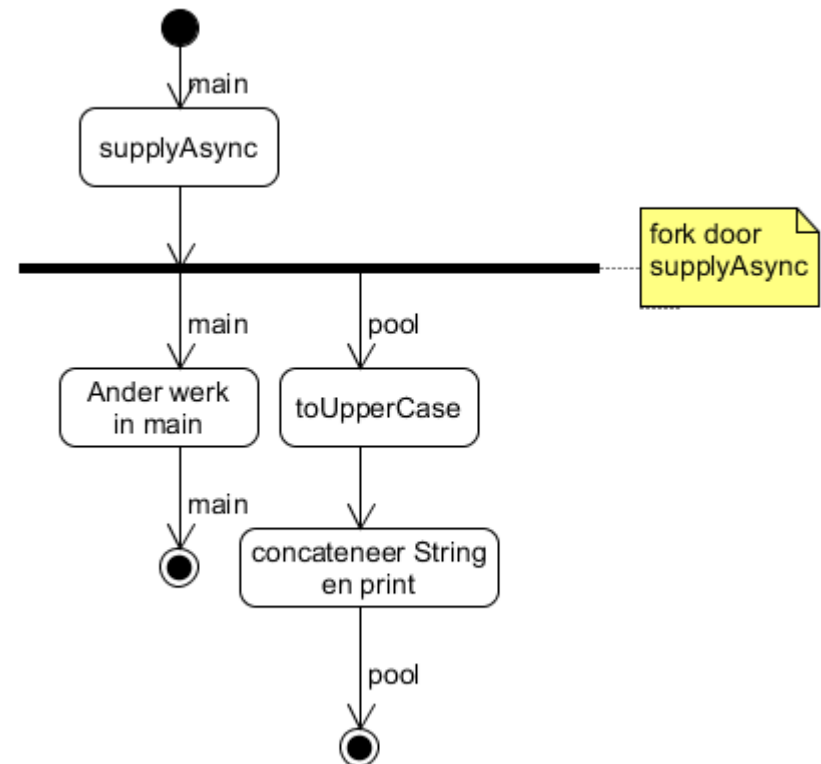
Thread main werkt verder tijdens omzetten.

Einde main thread..

Omzetten words in hoofdletters is gedaan: RED GREEN BLUE

## Voorbeeld 2: Actie op resultaat

- Het hele `supplyAsync` blok geeft onmiddellijk controle terug aan de main thread
  - Maar het `thenAccept` stukje wacht tot de `supplyAsync` taak gedaan is
  - Als input parameter krijgt `thenAccept` het resultaat van `supplyAsync`



## Voorbeeld 3: Extra actie en foutafhandeling

---

- Aanpassingen in vorige voorbeeld
  1. We doen nog een extra actie op het resultaat: nadat de strings in hoofdletters gezet zijn, starten we een taak om die de strings achterstevoren zet
  2. `thenAccept` wordt `thenApply`: deze geeft een resultaat terug waar we verder op kunnen werken
  3. Error handling
    - De `handle` methode heeft als 1e argument het resultaat van de vorige stap en als 2<sup>e</sup> argument een eventuele `Exception`
    - De `handle` methode wordt altijd aangeroepen (ook als de `Exception` in de eerste stap van de ketting gebeurt)
    - Door een methode te gebruiken ipv `try/catch` heb je de mogelijkheid de `Exception` af te handelen zonder de ketting te breken



## Voorbeeld 3: Extra actie en foutafhandeling

```

public class ExceptionCompletable {
    private static String[] words = {"red", "green", "blue"};

    public static void main(String[] args) {
        //words=null;
        ExecutorService pool = Executors.newFixedThreadPool(5);
        CompletableFuture.supplyAsync(
            () -> Arrays.stream(words).map(word -> word.toUpperCase()),
            pool)
            .thenApply(s -> s.map(word -> new StringBuffer(word).reverse()))
            .handle((result, x) -> x == null
                ? result.collect(Collectors.joining(" "))
                : "Foutje: " + x.toString())
            .thenAccept(print -> System.out.println("=> "+print));
        System.out.println("Thread main werkt verder tijdens omzetten.");
        System.out.println("Einde main thread.");
        pool.shutdown();
    }
}

```

Uncomment: Forceer een Exception

Extra stap: keer de woorden om

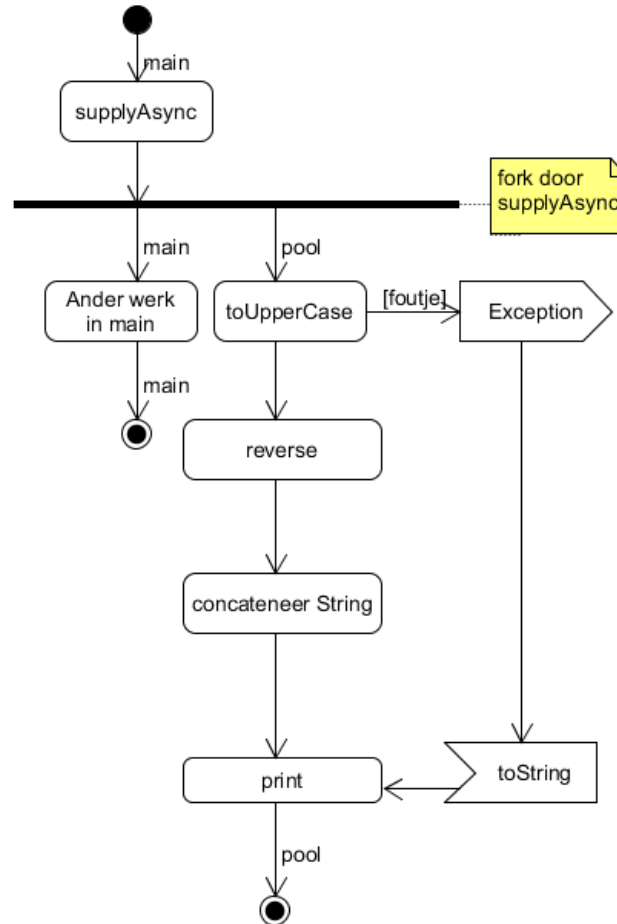
IF x == null ( alles OK)

ELSE Exception

=> Foutje: java.util.concurrent.CompletionException: java.lang.NullPointerException  
Thread main werkt verder tijdens omzetten.  
Einde main thread..

Thread main werkt verder tijdens omzetten.  
Einde main thread.  
=> DER NEERG EULB

# Voorbeeld 3: Extra actie en foutafhandeling





# Verder met CompletableFuture

- CompletableFuture heeft nog meer mogelijkheden om taken te beheren
  - `allOf(CompletableFuture... list)`: Wacht tot alle Futures klaar zijn
  - `anyOf(CompletableFuture... list)`: Wacht tot één van de Futures klaar is
  - `thenCombine(CompletableFuture andereFuture, Bifunction taak)`: Combineer twee CompletableFuture's. Start een BiFunction taak met twee parameters
    1. resultaat van de CompletableFuture waarop `thenCombine` opgeroepen wordt
    2. Resultaat van `andereFuture` (eerste parameter van `thenCombine`)
- Java 9
  - `completeOnTimeout(T value, long timeout, TimeUnit unit)`: als de asynchrone actie niet gedaan is voor de timeout, geef dan waarde T terug

# Opdracht



- Pas de code in het tweede voorbeeld (thenApply) aan zodat je de eigen Executor service niet meer gebruikt (zoals in voorbeeld 1).
- Thread main werkt verder tijdens omzetten.  
Einde main thread..
- Kan je verklaren waarom je de output van de Stringomzetting niet meer ziet?

# Agenda Deel 3: Concurrency

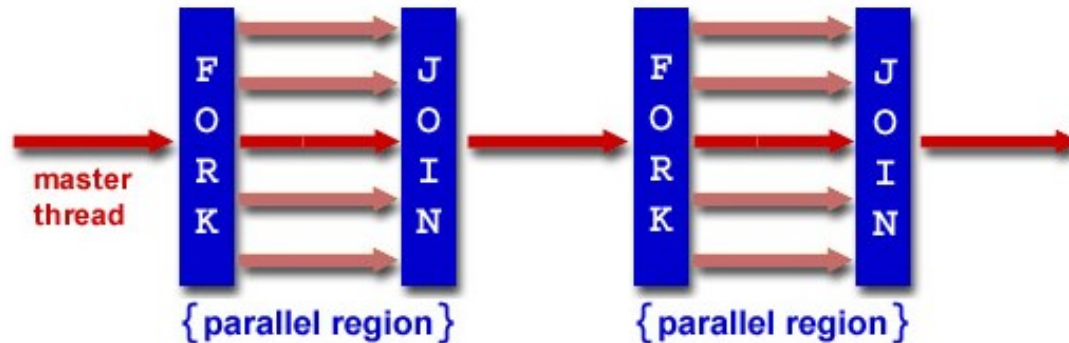
---



- Lock Objects
- Thread interactie en management
  - Executors en Thread pools
  - Callable
  - CompletableFuture
  - Fork/Join
- Consistente data
  - Atomic Variables
  - Concurrent Collections
- Parallelle Streams
- Conclusies



# Fork/Join



Waarom **Fork/Join** framework?

- Taken die een groot aantal threads nodig hebben (negatieve invloed op de performantie)
- Optimaal gebruik van multiple processors
- Ideaal voor grote berekeningen die kunnen worden opgedeeld in kleine stukken
- werkt volgens *work-stealing* algoritme
- Sinds Java 7

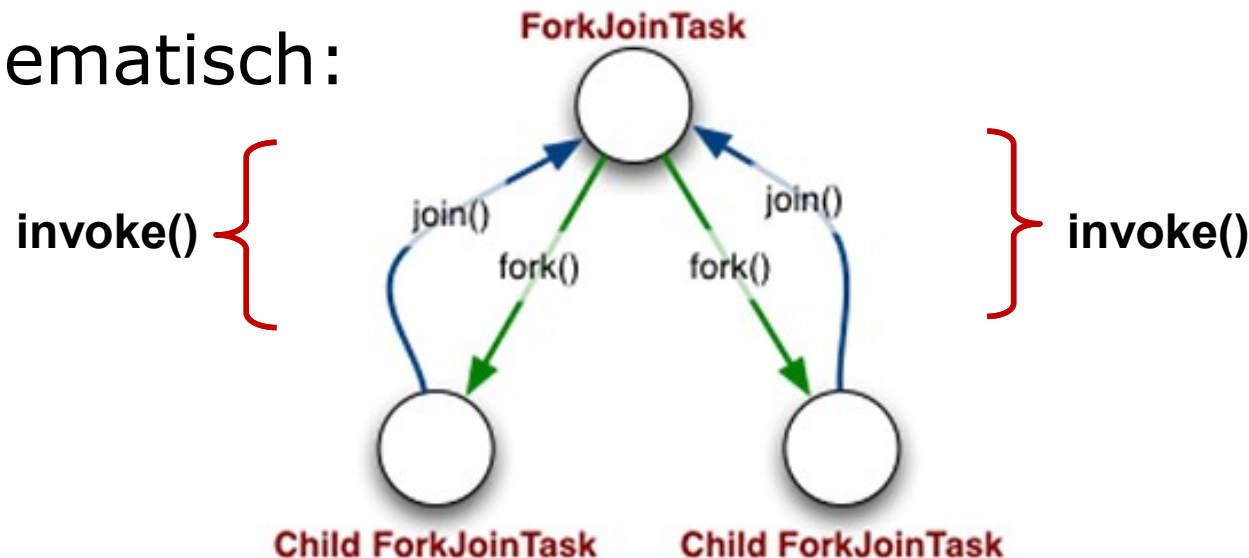
Worker threads die niets meer te doen hebben kunnen taken **stelen** van andere threads

# Fork/Join

- Concept (in pseudocode):

```
if (my portion of the work is small enough)
    do the work directly
else
    split problem into independent parts
    fork new subtasks to solve each part
    join all subtasks
    compose result from subresults
```

- Schematisch:



# ForkJoinTask

---

- Afgeleide abstracte klassen van **ForkJoinTask**:
  - **RecursiveAction**
    - indien geen returnwaarde nodig
  - **RecursiveTask**
    - speciale vorm van een ForkJoinTask die een returnwaarde geeft.
- Belangrijkste (abstracte) methode:
  - **compute()** = *verplichte run-methode*

# ForkJoinTask

---

- **compute()** bekijkt hoeveel werk er is en voert dit zelf uit of delegeert dit via:
  - **invoke** (= fork + join):
    - start nieuwe **ForkJoinTask** vanuit bestaande task
    - laat toe dat een **ForkJoinTask** wacht op het voltooien van een child-**ForkJoinTask**
  - **invokeAll** (= fork + join):
    - parallelle verwerking van een reeks taken

# Voorbeeld RecursiveAction

```
public class Transform extends RecursiveAction {  
    double[] data;  
    int start, end, threshold;  
    // Constructor weggelaten  
    protected void compute() {  
        if ((end - start) < threshold) {  
            for (int i = start; i < end; i++) {  
                if ((data[i] % 2) == 0)  
                    data[i] = Math.sqrt(data[i]);  
                else // derdemachtswortel  
                    data[i] = Math.cbrt(data[i]);  
            }  
        } else {int middle = (start + end) / 2;  
            invokeAll(new Transform(data, start, middle, threshold),  
                    new Transform(data, middle, end, threshold));  
        }  
    }  
}
```

**compute** is zoals de **run** bij een **Runnable**, **call** bij een **Callable**

Als te doorzoeken gebied kleiner is dan drempelwaarde

Anders: gebied opsplitsen in 2 helften en 2 nieuwe **ForkJoinTasks** opstarten



Democode: 4\_Forkjoin > RecursiveAction

# Voorbeeld RecursiveAction (main)

```
public class ForkJoinTest {  
    public static void main(String[] args) {  
        int cores = 4, size = 1000;  
  
        ForkJoinPool pool = new ForkJoinPool(cores);  
  
        double[] numbers = new double[1_000_000];  
        for (int i = 0; i < numbers.length; i++) {  
            numbers[i] = (double) i;  
        }  
        Transform task = new Transform(numbers, 0, numbers.length, size);  
  
        long start = System.nanoTime();  
        pool.invoke(task);  
        long end = System.nanoTime();  
        System.out.printf("Elapsed time: %.3f ms%n",  
                           (end - start)/1_000_000.0);  
    }  
}
```

Implements **ExecutorService**

Wacht op einde uitvoering



Democode: 4\_Forkjoin > RecursiveAction

# Voorbeeld RecursiveTask

```
public class Accumulate extends RecursiveTask<Long> {
    double[] data;
    int start, end, threshold;
    // Constructor weggelaten
    protected Long compute() {
        if ((end - start) < threshold) { // bereken in één blok
            return sequentialCompute(); // zie volgende slide
        } else { // Uitsplisen in 2 nieuwe taken
            int middle = (start + end) / 2;
            Accumulate t1 = new Accumulate(data, start, middle, threshold);
            Accumulate t2 = new Accumulate(data, middle, end, threshold);
            invokeAll(task1, task2);
            try {
                return task1.get() + task2.get();
            } catch (InterruptedException | ExecutionException e) {
                e.printStackTrace();
                return 0L;
            }
        }
    }
} // einde compute methode
// wordt vervolgd...
```

Resultaat taak van type **Long**

Haal intermediair resultaat uit Future



Democode: 4\_Forkjoin > RecursiveTask

# Voorbeeld RecursiveTask (vervolg)

```
// bereken in één blok van start tot end  
protected Long sequentialCompute() {  
    long sum = 0;  
    for (int i = start; i < end; i++) {  
        sum += data[i];  
    }  
    return sum;  
} // einde methode sequentialCompute  
} // einde klasse Accumulate
```






## Voorbeeld RecursiveTask (main)

```
public class ForkJoinTest {
    public static void main(String[] args) {
        int cores = 4, size = 1000;
        ForkJoinPool pool = new ForkJoinPool(cores);
        double[] numbers = new double[1_000_000];

        for (int i = 0; i < numbers.length; i++) {
            numbers[i] = (double) i;
        }
        Accumulate task= new Accumulate(numbers,0, numbers.length, size);
        long start = System.nanoTime();
        pool.invoke(task);
        long end = System.nanoTime();
        try {
            System.out.println(String.format("Som: %,d", task.get()));
        } catch (InterruptedException | ExecutionException e) {
            e.printStackTrace();
        }
    }
}
```

Haal resultaat uit Future



Democode: 4\_Forkjoin > RecursiveTask

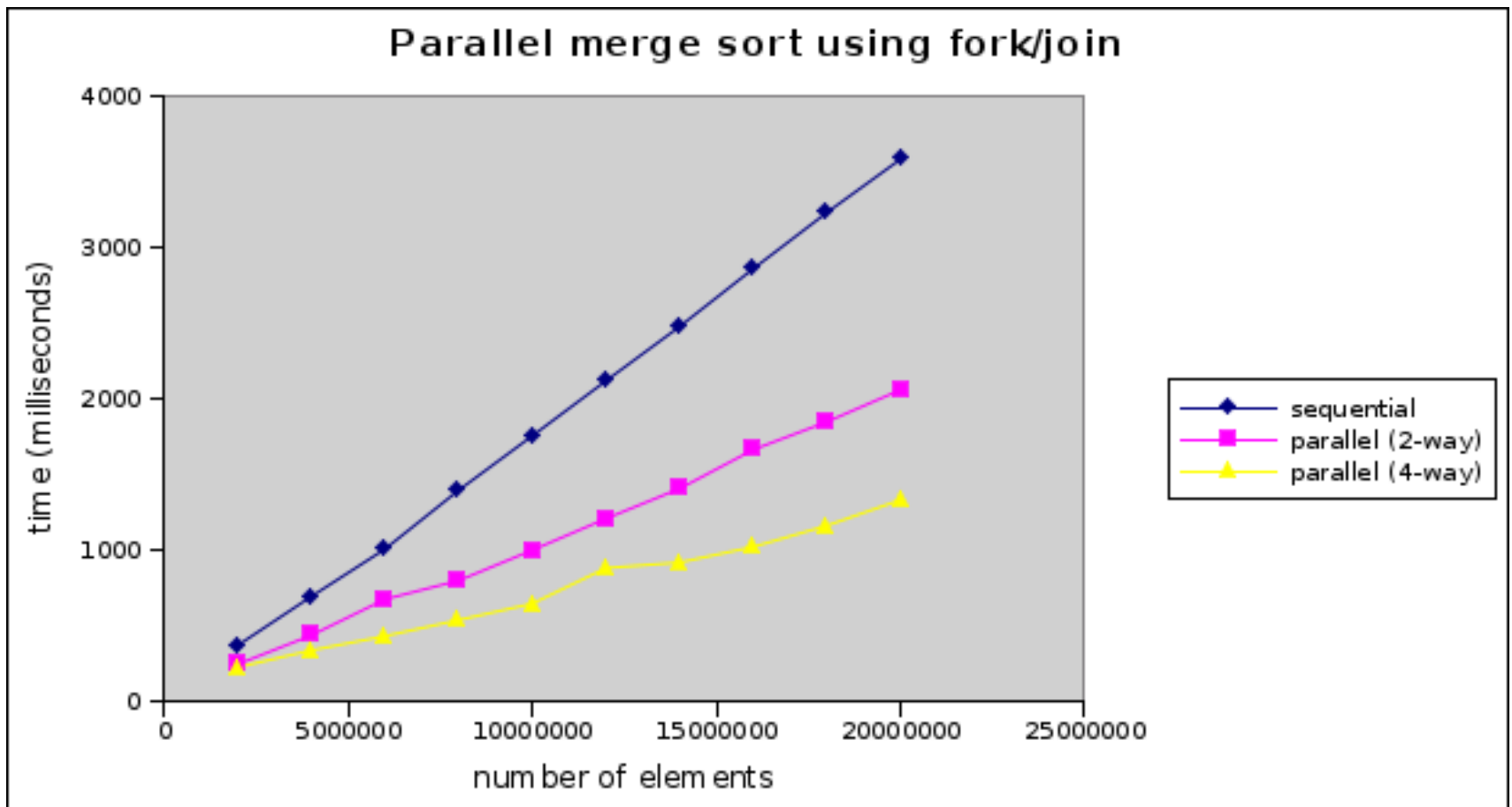


# Opdracht

- Bestudeer de code van het recursiveAction pakket in module **4\_ForkJoin** en voer ze een aantal keer uit.
- Krijg je altijd hetzelfde resultaat?
- Wijzig de waarde van het attribuut cores eens in 1, in 2 en in 8. Wat is de impact op de gemeten tijden? Wat leert dat over jouw processor?
- Vraag het aantal cores eens op via:  
`Runtime.getRuntime().availableProcessors()`
- Voeg code toe onderaan de main. Initialiseer de tabel opnieuw met dezelfde beginwaarden. Doe nu de berekeningen zonder fork/join, dus in een gewone sequentiële for-lus.  
Gaat dit sneller/trager?  
→Stemt dit overeen met cores = 1 of 2 of 4 of 8?

# Performatievoorbeld

In de klasse `java.util.Arrays` wordt in de `parallelSort` methode gebruik gemaakt van **fork/join**:



# Agenda Deel 3: Concurrency

---



- Lock Objects
- Thread interactie en management
  - Executors en Thread pools
  - Callable
  - CompletableFuture
- Fork/Join
- Consistente data
  - Atomic Variables
  - Concurrent Collections
- Parallelle Streams
- Conclusies

# Atomic Variables

---


- Een eenvoudige klasse als deze kan via **synchronized** threadsafe gemaakt worden
- Synchronisatie geeft ook overhead. Java zelf bevat dikwijls **sync**- en **non-sync** versies van een klasse – bv. **StringBuffer** en **StringBuilder**

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

# Atomic Variables

- De [java.util.concurrent.atomic](#) package bevat klassen die atomic operaties ondersteunen. Beperkt aantal, veel sneller dan synchronized (assembler niveau).

```
public class AtomicCounter {  
    private AtomicInteger counter = new AtomicInteger(0);  
  
    public void increment() {  
        counter.incrementAndGet();  
    }  
    public void decrement() {  
        counter.decrementAndGet();  
    }  
    public int value() {  
        return counter.get();  
    }  
}
```



counter.incrementAndGet()  
zoals ++i  
counter.getAndIncrement()  
zoals i++

- Andere atomic klassen: **AtomicBoolean**, **AtomicLong**, ...

# Concurrent Collections

---

Het `java.util.concurrent` package bevat een aantal aanvullingen op het Java Collections Framework:

- **ConcurrentMap**

- Is 'concurrency proof' en voorziet atomic operaties voor toevoegen (`put`, `putIfAbsent`), verwijderen (`remove`) of vervangen (`replace`) van een entry met gegeven key of key/value
- `ConcurrentHashMap` kan `HashMap` vervangen

- **ConcurrentNavigableMap**

- subinterface van `sortedMap` met zoekoperaties
- `ConcurrentSkipListMap` kan `TreeMap` vervangen

- **BlockingQueue**

- FIFO datastructuur waarbij er controle is voor het ophalen uit een lege queue en voor het toevoegen aan een volle queue.

# ConcurrentHashMap

---

- Een iterator op een ConcurrentHashMap kan verder gebruikt worden als de oorspronkelijke map verandert (maar ziet die veranderingen niet)
  - De ConcurrentHashMap wordt niet volledig gelockt, verwar niet met synchronized Collections
  - **ConcurrentHashMap<V, K>** bevat een aantal methoden voor lambda's:
    - forEach
    - forEachKey
    - forEachValue
    - forEachEntry
- Zie Javadoc van de klasse ConcurrentHashMap.



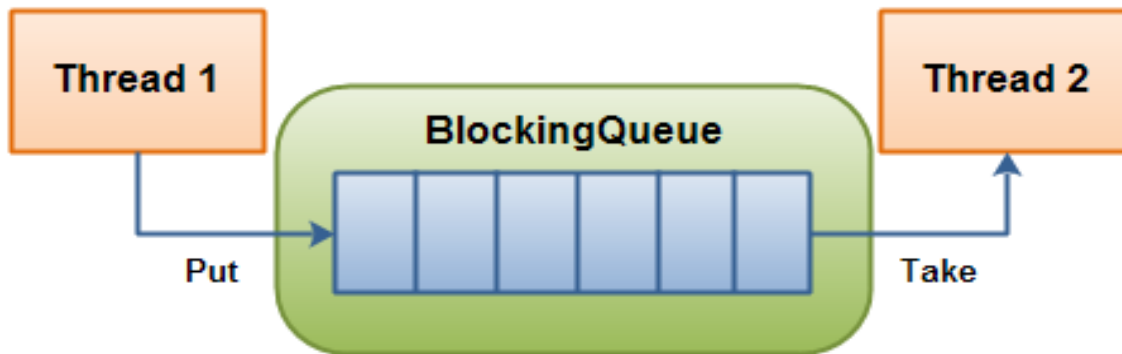


- Bestudeer de code van de module **6\_ConcurrentHashMap** en voer ze uit.
  - Loopt alles zoals verwacht? Verklaring?
- Vervang de `HashMap` door een `ConcurrentHashMap` en voer opnieuw uit.
  - Conclusie?
- Vervang de Entry `{"Louis", 60}` door `{"Louis", 80}` met zo weinig mogelijk code.

# Interface BlockingQueue

---

- Typisch voor producer-consumer situaties:



- Implementaties van deze interface:
  - ArrayBlockingQueue
  - DelayQueue
  - LinkedBlockingQueue
  - PriorityBlockingQueue
  - SynchronousQueue

# Interface BlockingQueue

---

- Implementaties: **ArrayBlockingQueue**, ...
- Methoden die wachten indien nodig
- Vier soorten methoden:
  1. Throws exception
  2. Geeft speciale waarde (`null` of `false`) terug
  3. Blokkeert de thread tot de methode kan uitgevoerd worden
  4. Blokkeert gedurende een opgegeven tijd (times out)

	<i>Throws exception</i>	<i>Special value</i>	<i>Blocks</i>	<i>Times out</i>
<b>Insert</b>	<code>add(e)</code>	<code>offer(e)</code>	<code>put(e)</code>	<code>offer(e, time, unit)</code>
<b>Remove</b>	<code>remove()</code>	<code>poll()</code>	<code>take()</code>	<code>poll(time, unit)</code>
<b>Examine</b>	<code>element()</code>	<code>peek()</code>	<i>not applicable</i>	<i>not applicable</i>

# Voorbeeld producer-consumer

```
public static void main(String[] args) throws Exception {
    BlockingQueue<String> queue = new ArrayBlockingQueue<>(1024);

    Runnable producer = () -> {
        try {
            queue.put("Tesla"); Thread.sleep(1000);
            queue.put("Ferrari"); Thread.sleep(1000);
            queue.put("Porsche");
        } catch (InterruptedException e) { }
    };

    Runnable consumer = () -> {
        try {
            System.out.println(queue.take());
            System.out.println(queue.take());
            System.out.println(queue.take());
        } catch (InterruptedException e) { }
    };

    new Thread(producer).start();
    new Thread(consumer).start();
}
```

# Collectors

---

De klasse **Collectors** voorziet ook in de volgende methodes (JDK 8):

- **groupingByConcurrent**
- **toConcurrentMap**

→ Zie Javadoc van de klasse `Collectors`



- Bestudeer de code van de module **5\_BlockingQueue** en voer ze uit.  
Loopt alles zoals verwacht?
- Pas de code aan zodat er gewacht wordt:
  - om een element in de queue te plaatsen tot er plaats is
  - om een element te verwijderen tot er een aanwezig is
- Voeg bij de producer een `sleep` van 150ms toe.  
Alles nog OK?

# Agenda Deel 3: Concurrency

---



- Lock Objects
- Thread interactie en management
  - Executors en Thread pools
  - Callable
  - CompletableFuture
- Fork/Join
- Consistente data
  - Atomic Variables
  - Concurrent Collections
- Parallele Streams
- Conclusies

# Parallele Streams

---

- Een parallele stream werkt net als een gewone stream, maar verwerkt gegevens parallel in meerdere threads
  - Gebruikt intern een fork/join pool



# Parallele Streams

- Creatie methode **parallelStream** of intermediaire methode **parallel()**.

```
List<String> list = Arrays.asList("Alfa", "Beta", "Gamma", "Delta");  
Stream<String> rechtstreeks = list.parallelStream();  
rechtstreeks.forEach(s -> System.out.print(s + " "));  
System.out.println();
```

```
Stream<String> intermediair = list.stream().parallel();  
intermediair.forEach(s -> System.out.print(s + " "));
```

- Mogelijke afdruk:

```
Gamma Beta Delta Alfa  
Gamma Alfa Delta Beta
```

# ForkJoinTest omgezet naar parallele streams

```
public class FJ2ParallelTest {  
    public static void main(String[] args) {  
        int cores = Runtime.getRuntime().availableProcessors() - 1;  
  
        double[] numbers = new double[10_000_000];  
        for (int i = 0; i < numbers.length; i++) {  
            numbers[i] = (double) i;  
        }  
  
        long start = System.nanoTime();  
        Arrays.stream(numbers).parallel()  
            .map(nr -> nr % 2 == 0 ? Math.sqrt(nr) : Math.cbrt(nr))  
            .toArray();  
        long end = System.nanoTime();  
        System.out.printf("Elapsed time: %.3f ms",  
            (end - start) / 1_000_000.0);  
    }  
}
```

Doet hetzelfde als code voorbeeld ForkJoinTask

Parallel gebruikt intern **ForkJoinPool**

Logica uit **RecursiveAction**



Democode: 7\_FJ2ParallelStream

# Parallele Array Operaties

---

De klasse **Arrays** bevat een aantal nieuwe methoden voor parallele bewerkingen op arrays (verkorte vorm parallele streams)

Er zijn 3 (overloaded) methoden:

- **parallelSort()** → parallel sorteren
- **parallelPrefix()** → cumulatieve parallele toepassing van BinaryOperator functionele interface op opeenvolgende elementen.
- **parallelSetAll()** → parallel alle elementen een waarde geven

→ Zie Javadoc van de klasse **Arrays**.



# Parallele Streams

Let op! Parallele streams kunnen ook veel ***trager*** zijn dan sequentiële streams. Onder andere bij:

- Kleine tot middelgrote datasets (overhead opzetten fork/join pools en threads)
- data structuren die niet eenvoudig opgesplitst kunnen worden (bv. **LinkedList**);
- taken die niet onafhankelijk van elkaar zijn;
- taken met veel in- en uitvoeroperaties of synchronization;



# Parallele Streams

- Let op! Benchmarking is niet eenvoudig.
  - `System.nanoTime()` is niet zo nauwkeurig
  - Je programma gebruikt ook tijd om te compileren naar het lokale platform
  - VM Optimisaties kunnen de uit te voeren job veranderen
- Professionals maken gebruik van libraries zoals **JMH** of **Caliper**.

# Opdracht

---



- Bestudeer de code van de module **8\_ParallelBench** en voer ze uit.  
Wat zijn je conclusies?
- Wat doet de stream-methode **boxed()**?
- Vrijblijvend: Bestudeer de klasse `IntSummaryStatistics` en pas ze toe om de goede werking van de klasse `RandomNumbersGenerator` te testen.

# Agenda Deel 3: Concurrency

---



- Lock Objects
- Thread interactie en management
  - Executors en Thread pools
  - Callable
  - CompletableFuture
- Fork/Join
- Consistente data
  - Atomic Variables
  - Concurrent Collections
- Parallelle Streams
- Conclusies

## Let's jump to conclusions...

---

- Gevaren bij threads
- Thread safety
- Thread designtips
- Parallel Streams tips







- Critical Section Race Condition
  - Gedeelde gegevens → Vermijden: niet meer dan één thread tegelijk!
- Starvation
  - Gedeelde resources en threads die niet meer aan bod komen...
- Deadlock
  - Twee of meer threads willen toegang tot hetzelfde object, elk van hen heeft een "lock" op de andere.
  - Opgelet met synchronized methoden die andere synchronized methoden oproepen



- Immutable classes
  - zijn impliciet thread safe.
- Zorg dat, als je een framework/bibliotheek gebruikt, je goed weet welke klassen thread safe zijn
  - Veiligheid  $\Leftrightarrow$  snelheid: thread-safe varianten zijn trager



- Collections
  - De 'gewone' collections zijn niet thread safe
  - Gebruik **Collections.synchronizedXxx()**
  - Of gebruik de nieuwe concurrent collections
- JavaFX
  - **Scene** (= GUI van JavaFX) is niet *thread safe*!
  - Alle acties worden uitgevoerd door de JavaFX Application thread (= UI thread)
  - Let op voor starvation
  - Maak gebruik van de klassen in de **javafx.concurrent** package.
  - meer info: <http://docs.oracle.com/javafx/2/threads/jfxpub-threads.htm>

# Threads Designtips

---



- Splits je code in blokken die onafhankelijk van elkaar zijn
- Zet code die samen uitgevoerd moet worden samen:  
Voorbeeld: groepeer getAantal() en setAantal samen in verhoogAantal
- Maak alle kwetsbare code synchronized
- Gebruik liever synchronized(this) { } voor een beperkt blok ipv synchronized voor een hele methode
- Beperk de scope van variabelen zo veel mogelijk (gebruik liefst lokale variabelen)

## Parallel Streams Tips

---



- Voor low-level klassen: maak unsynchronized en synchronized versies
- Maak gebruik van snelle functies en predicates
- Let op voor "side effects"
- Gebruik niet overal parallel -> test eerst de performantie met een goede externe tool
- Ga er niet van uit dat parallel altijd de originele volgorde bewaart!

# Asynchrone taken

Te kort overzicht



# Java 1 concurrency

---

- Definieer taak met Runnable functionele interface

```
void run()
```

- Run taak met Thread

- Voorbeeld:

```
Runnable r = () -> {doIets();}
```

```
Thread t = new Thread(r);
```

```
t.start();
```

# Java 5 concurrency

---

- Probleem: voor elke run moet je een nieuwe Thread maken. Niet efficiënt.
- Oplossing: herbruikbare run met Executor
- Voorbeeld

```
Runnable r1 = () -> {doeIets();};
```

```
Runnable r2 = () -> {doeNogIets():};
```

```
ExecutorService e = ExecutorService.newFixedThreadPool(3);
```

```
e.execute(r1);
```

```
e.execute(r2);
```



# Java 5 concurrency

---

- Probleem: een Runnable taak heeft geen resultaat
- Oplossing: Definieer taak met Callable<T> functionele interface

`T call()`

- run taak met Executor
- Voorbeeld

```
Callable<String> producer = () -> geefString(); //return String
ExecutorService e = ExecutorService.newFixedThreadPool(3);
Future<String> uitgesteldResult = e.submit(producer);
String result = uitgesteldResult.get(); // wacht
```

- Probleem: call() geeft een resultaat maar heeft geen parameter?
- Oplossing: geef parameters aan de constructor van je Callable implementatie. Je kan dan geen lambda's gebruiken.

## Java 8: concurrency met functionele stijl

---

- Probleem: de `get()` methode laat de thread wachten op het Future resultaat
- Oplossing: Vloeiende programmeerstijl met `CompletableFuture`
- Voorbeeld

```
Callable<String> producer = () -> geefString(); //return
String

ExecutorService e = ExecutorService.newFixedThreadPool(3);

CompletableFuture.supplyAsync(producer,e) // e is optioneel
    .thenApply(result -> result.doeIets()); // wacht niet
```

– De `ExecutorService` is optioneel. Indien niet meegegeven, gebruikt `supplyAsync(producer)` de fork/join common pool.

# Java 8: concurrency met functionele stijl

---

- Met parallel streams is concurrency eenvoudig
  - parallel streams gebruiken de fork/join common pool

# Opdrachten

---

- Groeiproject

- module 9  
(laatste deel 4: "Concurrency")



- Opdrachten op BB

- Perfecte getallen
- Carwash met ArrayBlockingQueue
- ParallelStreams



- Zelftest op BB

- Herhalingsopdracht!!!

- Zie BB onder het menu-item  
"Voorbereiding examen".

# Veel succes met je examen!

