10 Threads

Programmeren 2 – Java

2017 - 2018

KdG Karel de Grote Hogeschool

Kris Behiels

Jan De Rijke

Mark Goovaerts

Programmeren 2 - Java

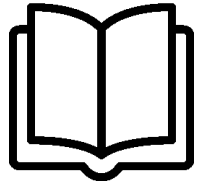
1. Herhaling en Collections
2. Generics en documenteren
3. Annotations en Reflection
4. Testen en logging
5. Design patterns (deel 1)
6. Design patterns (deel 2)
7. Lambda's en streams
8. Persistentie (JDBC)
9. XML en JSON
10. **Threads**
11. Synchronization
12. Concurrency

Agenda voor volgende weken

We zullen het de komende 3 lesweken hebben over multi-threading en concurrent programming:

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

Syllabus



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

Agenda

1. Deel 1: threads
   - Inleiding
   - Thread objecten
   - Creatie, opstarten en pauzeren
   - Runnable
   - De klasse Thread
   - Thread Life Cycle
   - Thread priority
   - Daemon thread
   - yield en join

Threads

Inleiding

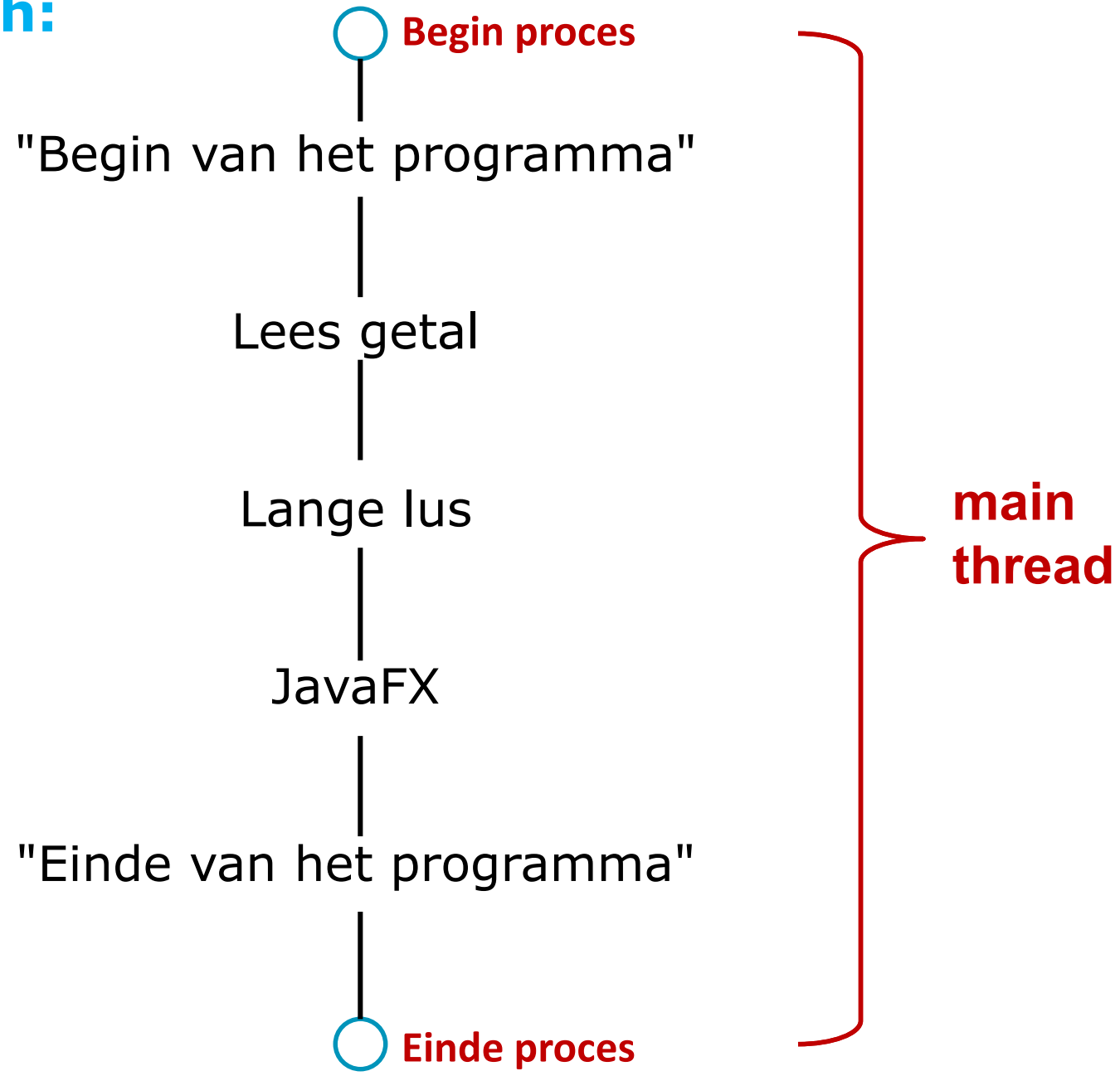
- Tot nu toe:
  - Java-programma = 1 thread (*main thread*)
    → wordt **sequentieel** uitgevoerd
- Vanaf vandaag:
  - Meerdere threads opstarten
    → worden **parallel** uitgevoerd

Eenvoudig voorbeeld: main thread

```
public class Demo extends Application {
    public static void main(String[] args) {
        System.out.println("BEGIN van het programma");

        double getal = UserIO.leesGetal();
        System.out.println("Het getal dat werd ingetikt: " + getal);
        new LangeLus();
        Application.launch(args); //JavaFX opstarten (zie "start")

        System.out.println("EINDE van het programma");
    }

    @Override
    public void start(Stage primaryStage) throws Exception {
        Button button = new Button("Klik hier!");
        button.setOnAction(event -> System.out.println("KLIK"));

        primaryStage.setScene(new Scene(new BorderPane(button)));
        primaryStage.setHeight(80);
        primaryStage.show();
    }
}
```

Schematische voorstelling op volgende slide

Schematisch:



Concurrent programming

Concurrency: verschillende delen van programma worden asynchroon / parallel uitgevoerd

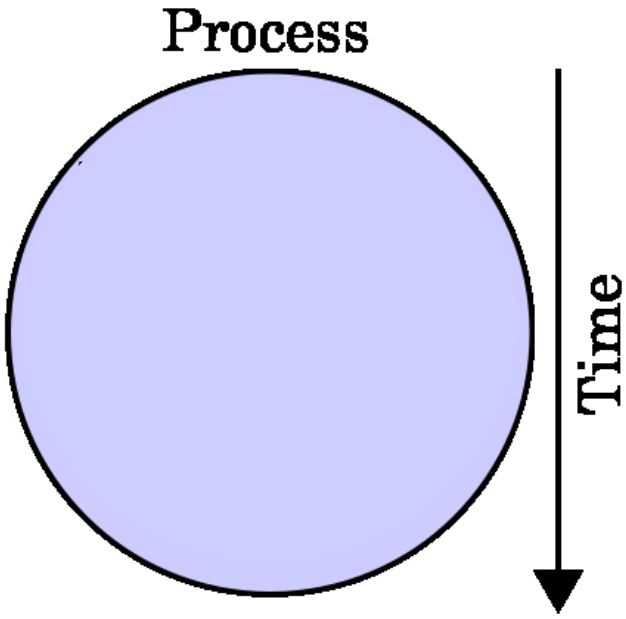
- Voordelen:
  - snelheid en performantie!
  - optimaal gebruik van multi-core processor

- Nadelen:
  - tricky!
  - synchronization!
  - sharing data in memory!

KdG Karel de Grote Hogeschool

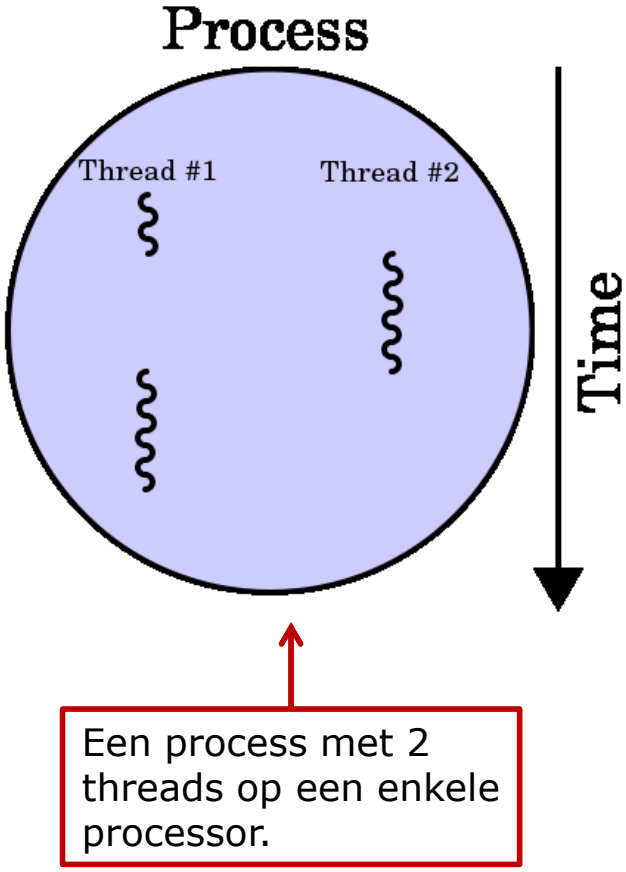
Process versus thread

- **Process**:
  - Onafhankelijk van andere processen
  - Eigen set van resources / data
  - Elk Java programma runt normaal als één proces



Process versus thread

- **Thread**:
  - Is een *'lightweight process'*
  - Toegang tot shared data / resources
  - Bestaat binnen een process (minstens één thread)
  - Elk programma start met de **main** thread: deze thread kan andere threads creëren
  - Definitie: "*A thread is a single sequential flow of control within a process*"



Thread Objecten

- Elke thread is gekoppeld aan een object van de klasse **`Thread`**
- Je kan de objecten op 2 verschillende manieren creëren:
  - Maak voor iedere ***asynchrone*** taak een nieuw **`Thread`** object
  - Laat de creatie over aan een ***Executor***

    (zie deel 3: "Concurrency")

Creatie van een thread

- Twee mogelijkheden om een nieuwe thread te creëren:

Kortst, maar subklasse is eerder bedoeld voor een nieuw soort thread

1. Maak een subklasse van de klasse **`Thread`** en override de **`run`** methode.

Aanbevolen

2. Implementeer de **`Runnable`** interface en implementeer de **`run`** methode.

   (**`Runnable`** is een Functional Interface, dus Lambda Expressions zijn mogelijk)

Een thread opstarten

- De "main" methode van een thread is de methode **`run`**
- Een thread **opstarten** gaat met behulp van de methode **`start`**
  - De JVM start dan een nieuwe uitvoeringscontext en roept daarin de **`run`** methode van het thread-object aan.
- Een thread **eindigt** wanneer de **`run`** methode gedaan is.
  - Gebruik **nooit** de *deprecated* methode **`stop`**!

Pauzeren met sleep

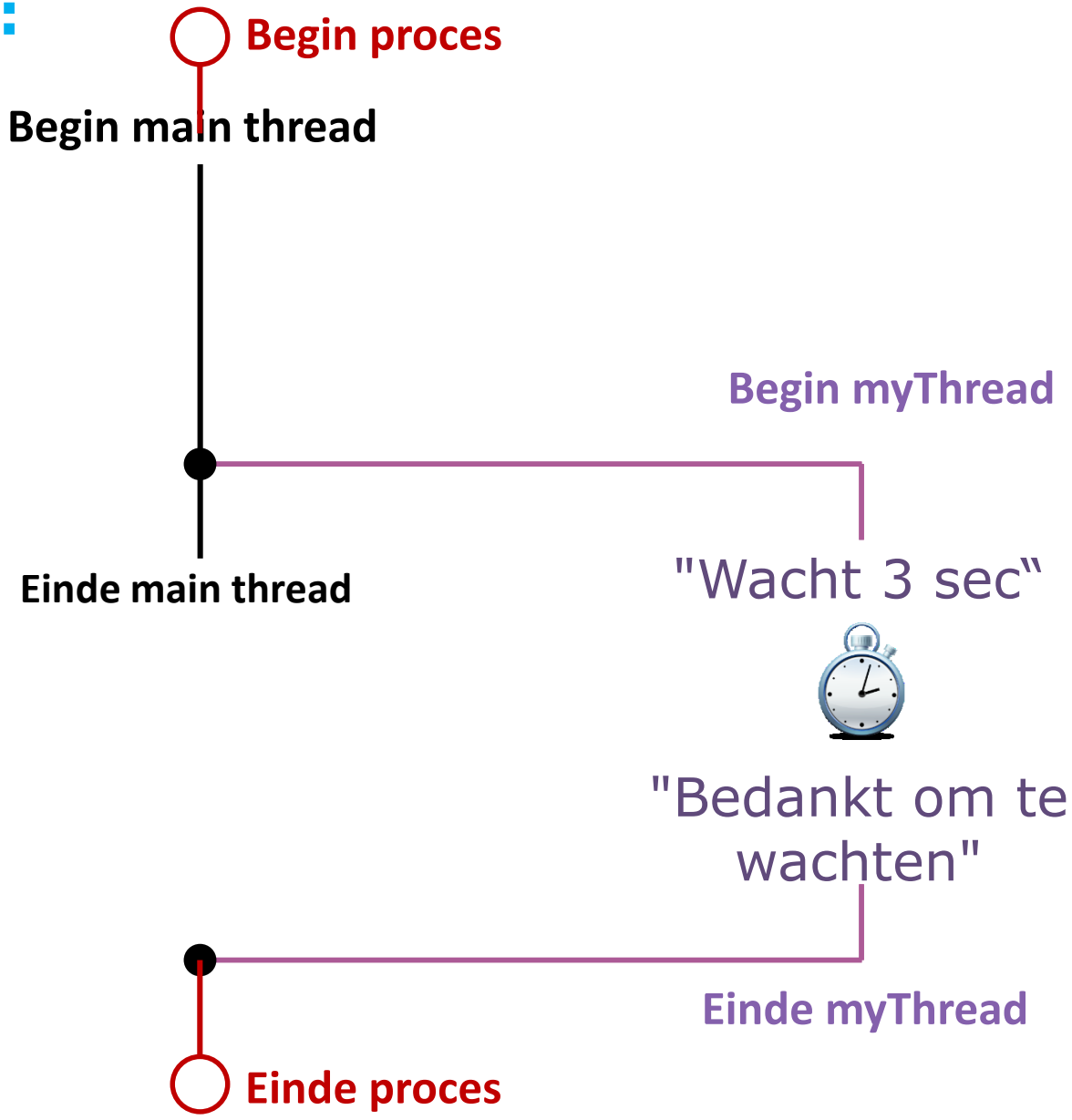
```
public class SleepDemo implements Runnable {
    @Override
    public void run() {
        System.out.println("Wacht 3 seconden...");
        try {
            Thread.sleep(3000);
        } catch (InterruptedException ex) {
            // negeer exception
        }
        System.out.println("Bedankt om te wachten!");
    }
}
```

sleep pauzeert de thread gedurende 3000 milliseconden.

```
public class RunSleep {
    public static void main(String[] args) {
        System.out.println("Begin van de main thread");
        Thread myThread = new Thread(new SleepDemo());
        myThread.start();
        System.out.println("Einde main thread!");
    }
}
```

Schematische voorstelling op volgende slide

Schematisch:



KdG
Karel de Grote Hogeschool

Pauzeren met sleep (lambda)

```
import java.util.concurrent.TimeUnit;

public class RunSleep {
    public static void main(String[] args) {
        Thread thread = new Thread(
                () -> {
                    System.out.println("Wacht 3 seconden...");
                    try {
                        TimeUnit.SECONDS.sleep(3);
                    } catch (InterruptedException e) {
                        // try/catch Noodzakelijk
                    }
                    System.out.println("Bedankt om te wachten! ");
                }
        );
        thread.start();
    }
}
```

sleep
pauzeert de thread 3 seconden.

Lambda expression vervangt:

```
new Runnable() {
  @Override
  public void run()
```

Runnable voorbeeld

```
public class Racer implements Runnable {

    public void run() {
        String naam = Thread.currentThread().getName();
        ThreadLocalRandom random = ThreadLocalRandom.current();
        System.out.println(naam + " START");
        for (int i = 0; i < 10; i++) {
            System.out.println(naam  + " ronde " + (i + 1));
            try {
                Thread.sleep(random.nextInt(1000));
            } catch (InterruptedException e) {
                // negeer
            }
        }
        System.out.println(naam + " AANGEKOMEN");
    }
}
```

currentThread vraagt de huidige thread op

Thread friendly random number

Democode: 1_RaceRunable

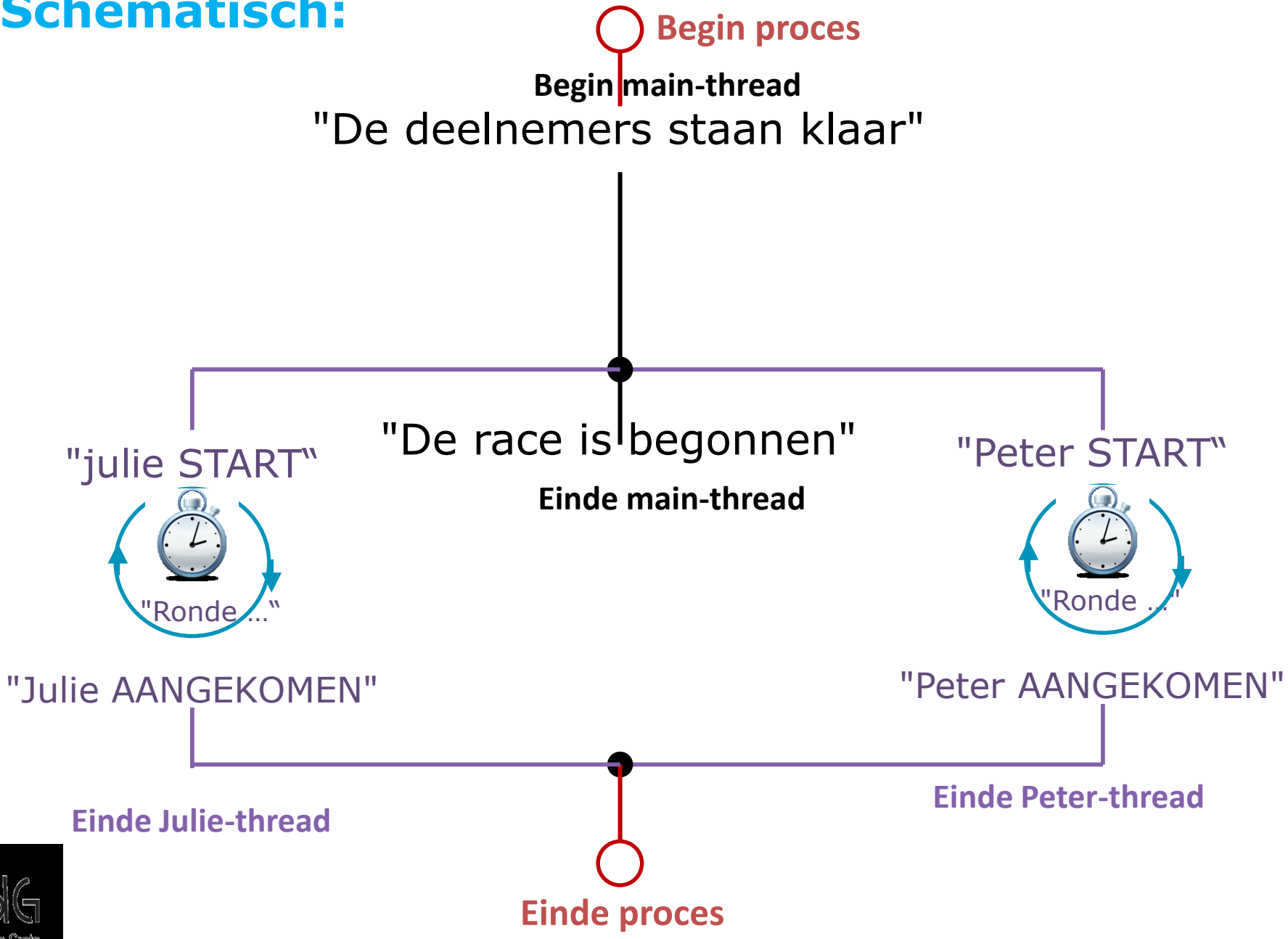
Runnable voorbeeld (vervolg)

```
public class StartRace {
    public static void main(String[] args) {
        Thread racerEen = new Thread(new Racer(), "Peter");
        Thread racerTwee = new Thread(new Racer(), "Julie");

        System.out.println("De deelnemers staan klaar");
        racerEen.start();
        racerTwee.start();
        System.out.println("De race is begonnen");
    }
}
```

Voer het programma enkele malen uit.
Wie wint er?
Wat merk je op?

Schematische voorstelling op volgende slide

Schematisch:



Agenda

1. Deel 1: threads

- Inleiding
- Thread objecten
- Creatie, opstarten en pauzeren
- Runnable
- De klasse Thread
- Thread Life Cycle
- Thread priority
- Daemon thread
- yield en join

Thread Constructors

- **`public Thread()`**
  - Naamloze thread (eerste thread krijgt de default-naam "Thread-0" enz…)
- **`public Thread(Runnable thread)`**
  - Naamloze thread, via Runnable geïmplementeerd
- **`public Thread(String naam)`**
  - Thread met naam
- **`public Thread(Runnable thread, String naam)`**
  - Thread met naam, via Runnable geïmplementeerd

Thread Methoden (1)

- **`public void run()`**
  - De "main" van de thread; geschreven door ontwikkelaar, opgeroepen door JVM
- **`public synchronized void start()`**
  - Start de thread: opgeroepen door ontwikkelaar, JVM start nieuwe uitvoeringscontext en roept run aan
- **`public static native void sleep(long msec)`**
  - Pauzeert de uitvoering van de thread (mogelijk `throw` van `InterruptedException`)
- **`public final void join()`**
  - De lopende thread (thread 1) wacht tot de thread waarop de methode opgeroepen wordt (thread 2) beëindigd is

Thread Methoden (2)

- **`public static native void yield()`**
  - De thread die momenteel runt verleent voorrang aan eventuele andere threads die klaar zijn om te lopen
- **`public static native Thread currentThread()`**
  - Geeft de thread in uitvoering
- **`public final String getName()`**
  - Geeft de naam van de thread (default is "`Thread-n`" met `n` een `int` waarde beginnend bij 0)
- **`public final void setDaemon(boolean on)`**
  - Maakt van een thread een **daemon thread** (zie verder). Doe dit voor je de thread start.

Thread Methoden (3)

- **`public final native boolean isAlive()`**
  - Geeft `true` als de thread in de toestand runnable is en false bij de toestand not runnable

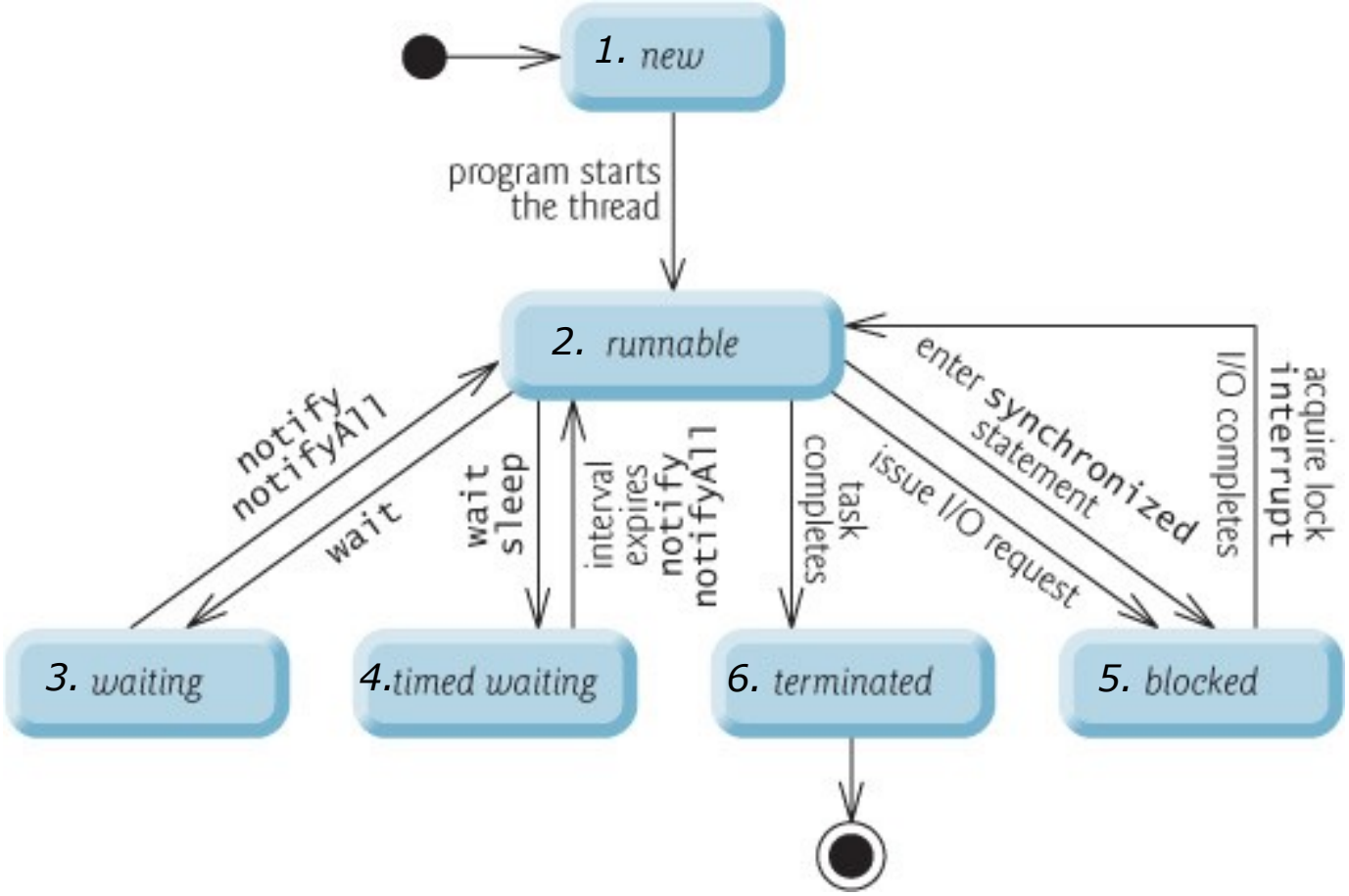
- **`Public void interrupt()`**
  - Genereert InterruptException in methode die door thread uitgevoerd wordt
  - Het is aangeraden InterruptedException af te handelen in de run methode

Slides Opdracht 1

- Download het project **Voorbeelden_Opdrachten_10_Threads** (BB)
- Bestudeer de code van de module **1_Race_Runnable** en voer dan het programma een aantal keer uit. Wat merk je op?
- Doe het opstarten van de threads eens met **run** ipv **start**. Wat merk je op?
- Vul vervolgens de code (TODO) in de module **1_Race_Thread** waar nodig aan (versie **`extends Thread`**) tot je bij uitvoering hetzelfde resultaat bekomt.
- Vul de lambda expression (TODO) in **1_Race_Lambda** aan

Thread levencyclus

- Een thread bevindt zich altijd in een bepaalde toestand (thread state)
- Er zijn 6 mogelijke toestanden:



Thread Toestand 1: *new*

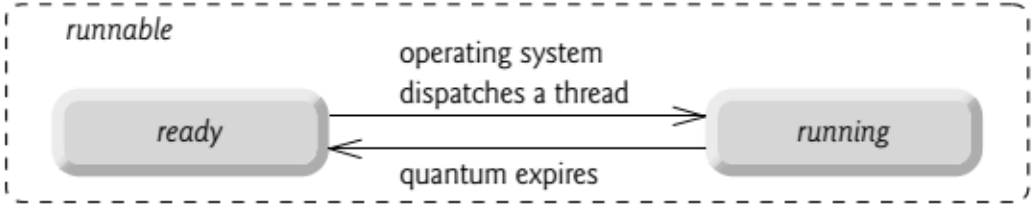
```
public class StartRace {
    public static void main(String[] args) {
        Racer racerEen = new Racer("Peter");
        Racer racerTwee = new Racer("Julie");

        // ...
    }
}
```

- na uitvoering van `new` zijn beide threads in de toestand ***new***
- In deze toestand kan je alleen de **`start`** levenscyclus methode toepassen. Het oproepen van elke andere methode geeft een **`IllegalThreadStateException`**.
- De methode **`isAlive`** geeft *false*.

Thread Toestand 2: *runnable*

```
public class StartRace {
    public static void main(String[] args) {
        Racer racerEen = new Racer("Peter");
        Racer racerTwee = new Racer("Julie");

        racerEen.start();
        racerTwee.start();
    }
}
```

- De **`start`**-methode geeft de JVM een signaal dat de thread kan uitgevoerd worden. Op dat moment is de thread in de toestand ***runnable***
  Vanaf dan is het **mogelijk** dat de thread wordt uitgevoerd door de JVM.
- De methode **`isAlive`** geeft nu *true*.



Thread Toestand 3: *waiting*

- Soms kan een thread zich bevinden in toestand ***waiting***

  (= een tijdelijke pauze)
- Er wordt gewacht op een andere thread die een taak uitvoert
- Een thread komt in deze toestand door:
  - de methode **`wait`** (uit klasse `Object`)
  - de methode **`join`** (uit klasse `Thread`)
  - de methode **`yield`** (uit klasse `Thread`)
- De methode **`isAlive`** retourneert nu *true*.
- Wanneer er een bericht komt van een andere thread, kan de thread terug ***runnable*** worden:
  - de methode **`notify`** en **`notifyAll`** (uit klasse `Object`)

wait, **notify** en **notifyAll** worden volgende week behandeld in deel 2 "*Synchronization*"

Thread Toestand 4: *timed waiting*

```
try {
    sleep(random.nextInt(1000));
} catch (InterruptedException e) {
    // ignore
}
```

- Je kan een thread tijdelijk pauzeren. Hij bevindt zich dan in toestand ***timed waiting***
- Een thread komt in deze toestand door:
  - de methode **`wait(timeout)`** (uit klasse `Object`)
  - de methode **`join(timeout)`** (uit klasse `Thread`)
  - de methode **`sleep(timeout)`** (uit klasse `Thread`)
- De methode **`isAlive`** retourneert nu *true*.
- Als het tijdsinterval verstreken is, dan komt de Thread terug in ***runnable*** toestand.

Thread Toestand 5: *blocked*

- Een runnable thread kan in toestand ***blocked*** terecht komen als:
  - een `synchronized` stuk code door een andere thread bezet wordt
  - als hij onverwacht onderbroken wordt (methode `interrupt()` van Thread)
  - als hij wacht op een I/O device

- De methode `isAlive` retourneert nu *true*.
- Als de blokkade opgeheven is, dan komt de Thread terug in ***runnable*** toestand.

synchronized en **blokkades** worden volgende week behandeld in deel 2 "*Synchronization*"

Thread Toestand 6: *terminated*

```
public void run() {
    // ...

}
```

- Een thread komt in de toestand ***terminated*** (of: ***dead Thread***) als de **`run`** methode van de thread beëindigd is.
- De methode **`isAlive`** retourneert *false*.
- In deze toestand kan je niets meer met het Thread-object doen.

Opvragen Thread Toestand

- Met de methode **getState** kan je de toestand in de life cycle van een thread opvragen.
- De klasse Thread bevat intern een enum **State** die in volgorde de volgende 6 constanten bevat:

  NEW, **RUNNABLE**, **BLOCKED**, **WAITING**, **TIMED_WAITING** en **TERMINATED**.

- Alle mogelijke toestanden afdrukken kan als volgt:

```
public class ShowStates {
    public static void main(String[] args) {
        for (Thread.State state : Thread.State.values()) {
            System.out.println(state);
        }
    }
}
```

Opvragen Thread Toestand

```
public class TreadStateDemo {
  public static void main(String[] as) throws InterruptedException {
    Runnable runnableJob =
              () -> System.out.println("Job is running");

    Thread thread = new Thread(runnableJob);
    System.out.println(thread.getState() + " " + thread.isAlive());

    thread.start();
    System.out.println(thread.getState() + " " + thread.isAlive());

    Thread.sleep(1000);
    System.out.println(thread.getState() + " " + thread.isAlive());
  }
}
```

Wat is de afdruk van dit programma?

Agenda

1. Deel 1: threads

- Inleiding
- Thread objecten
- Creatie, opstarten en pauzeren
- Runnable
- De klasse Thread
- Thread Life Cycle
- Thread priority
- Daemon thread
- yield en join

Thread priority

- De **priority** is bepalend voor de volgorde / frequentie waarmee de JVM de threads afhandelt (*thread scheduling*)
- Via `setPriority` kan een waarde van 1..10 gegeven worden:
  - `NORM_PRIORITY` = 5 (de defaultwaarde)
  - `MIN_PRIORITY` = 1
  - `MAX_PRIORITY` = 10

- Opgelet: thread priority garandeert geen bepaalde uitvoeringsvolgorde!
  (is enkel hint aan JVM)

Daemon Thread

- Elk programma bevat tenminste één thread (de `main` thread)
- Elke thread is standaard een ***User Thread*** (een *onafhankelijke* thread). De JVM exit pas als alle user threads gedaan zjin.
- Via de methode **`setDaemon(true)`** maak je een thread *afhankelijk* van een andere thread (de parent-thread). De JVM wacht niet op daemon threads. Ook als deze thread nog loopt zal de JVM eindigen.
- Een ***Daemon Thread*** eindigt:
  - ofwel als z'n eigen `run` methode eindigt
  - ofwel als alle user threads eindigen

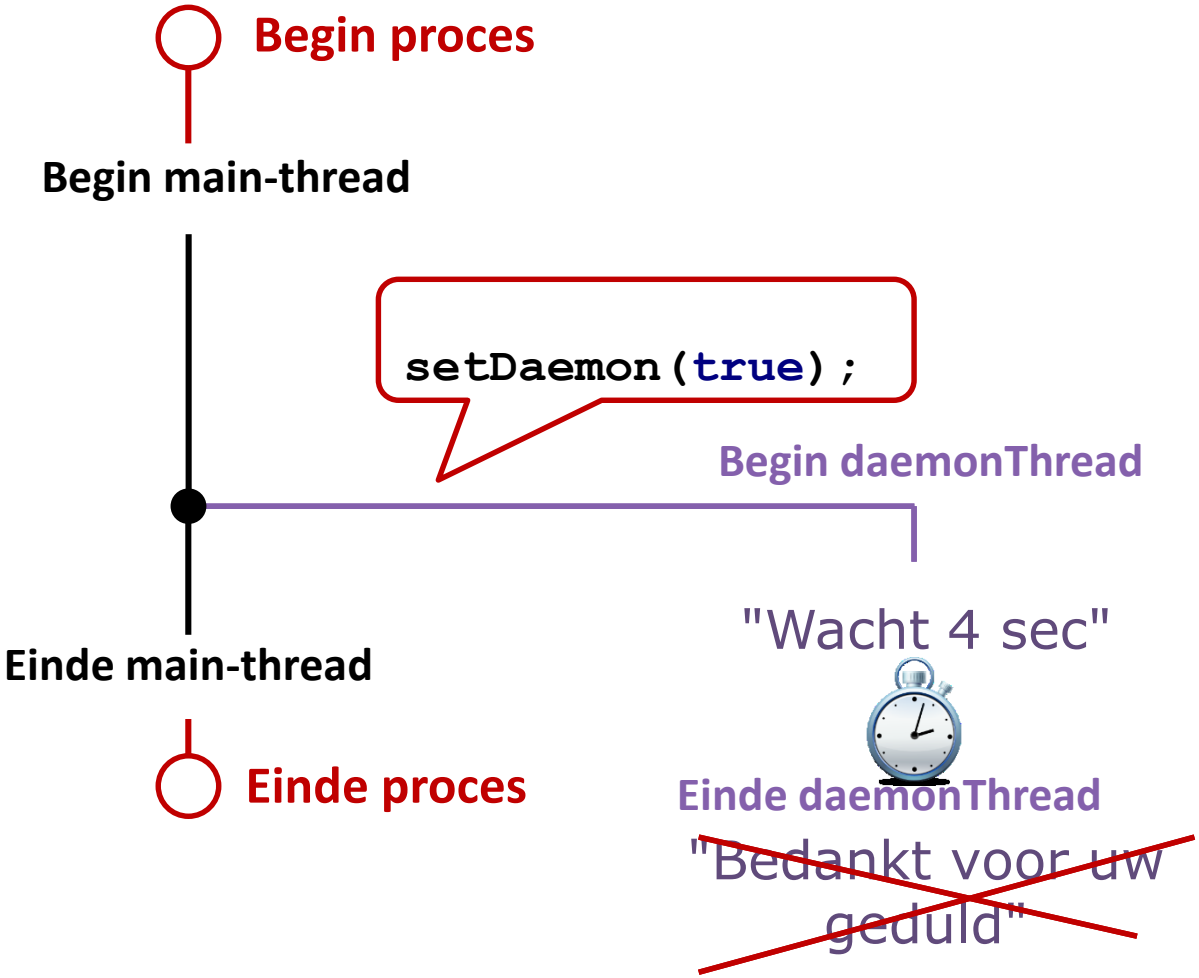
Voorbeeld Daemon (1)

```
public class DaemonTest {
    public static void main(String[] args) {
        System.out.println("Begin van de main thread");
        Thread daemonThread = new Thread() -> {
            System.out.println("Wacht 4 sec");
            try {
                Thread.sleep(4000);
            } catch (InterruptedException e) {
                e.printStackTrace();
            }
            System.out.println("Bedankt voor uw geduld");
        });
        daemonThread.setDaemon(true);
        daemonThread.start();
        System.out.println("Einde van de main thread");
    }
}
```

daemonThread is nu afhankelijk van de **main-thread**.

De **main-thread** eindigt. Dit is de enige user thread, dus wordt ook **daemonThread** gestopt.

Schematische voorstelling op volgende slide


Begin proces
Begin main-thread
setDaemon(true);
Begin daemonThread
"Wacht 4 sec"
Einde main-thread
Einde proces
Einde daemonThread
"Bedankt voor uw geduld"

Voorbeeld Daemon (2)

```
public class MyLambda {
    public static void main(String[] args) {
        Thread myThread = new Thread( () -> {
            for (int i = 0; i < 100000; i++) {
                System.out.println("Child Thread step " + (i + 1));
                try {
                    Thread.sleep(100);
                } catch (InterruptedException e) { }
            }
        });
        myThread.setDaemon(true);
        myThread.start();

        for (int i = 0; i < 5; i++) {
            System.out.println("Main Thread step " + (i + 1));
            try {
                Thread.sleep(200);
            } catch (InterruptedException e) { }
        }
    }
}
```

Wat is het gevolg als je hier **false** zet?

Output op volgende slide

Voorbeeld Daemon

Mogelijke uitvoer
bij setDaemon(true)

```
Main Thread step 1
Child Thread step 1
Child Thread step 2
Child Thread step 3
Main Thread step 2
Child Thread step 4
Child Thread step 5
Main Thread step 3
Child Thread step 6
Child Thread step 7
Main Thread step 4
Child Thread step 8
Child Thread step 9
Main Thread step 5
Child Thread step 10
Child Thread step 11
Process finished with
exit code 0
```

```
Main Thread step 1
Child Thread step 1
Child Thread step 2
Main Thread step 2
Child Thread step 3
Child Thread step 4
Main Thread step 3
Child Thread step 5
Child Thread step 6
Main Thread step 4
Child Thread step 7
Child Thread step 8
Main Thread step 5
Child Thread step 9
Child Thread step 10
Process finished with
exit code 0
```

Slides Opdracht 2

- Bestudeer de code van de module **2_Daemon_Thread** en voer dan het programma een aantal keer uit.
  - Wat merk je op?
  - Wat gebeurt als je van de 2 threads gewone user threads maakt?
- Vul vervolgens de `main` methode in de module **2_Daemon_Runnable** aan (versie `implements Runnable`) tot je bij uitvoering hetzelfde resultaat bekomt (enkel de main mag je aanpassen).
- Vul de lambda expression in **2_Daemon_Lambda** aan.

Slides Opdracht 3

- Bestudeer de code van de module **3_ThreadPriority** en voer dan het programma een aantal keer uit.
  - Wie komt het meest aan bod: de hello-thread of de bye-thread? Verander de priority en test opnieuw.
  - Welke van de 4 threads zijn daemon threads? Welke zijn user threads?
  - Zoek in de afdruk wanneer "MAIN IS ENDING" wordt afgedrukt: verklaar?
  - Maak er allemaal user threads van en run opnieuw: wat gebeurt er?

Verkeersregels

- Via 2 methoden kunnen we op een eenvoudige manier het verkeer tussen threads regelen:
- **`yield`**:
  - De thread die momenteel wordt uitgevoerd verleent voorrang aan alle andere threads (met dezelfde prioriteit)
- **`join`**:
  - De huidige thread roept deze methode op bij een andere thread; daardoor blokkeert de huidige thread tot de andere thread ten einde is

- **Opgelet**: **`yield`** en **`setPriority`** garanderen geen bepaalde uitvoeringsvolgorde! (enkel: hint)

KdG
Karel de Grote Hogeschool

Yield: voorbeeld (1)

```
public class YieldingThread extends Thread {
    private static int threadCount = 0;
    private int countDown = 5;

    public YieldingThread() {
        super("" + ++threadCount);
        start();
    }
    public String toString() {
        return "#" + getName() + ": " + countDown;
    }
    public void run() {
        while (true) {
            System.out.println(this);
            if (--countDown == 0) return;
            Thread.yield();
        }
    }
}
```

yield verleent voorrang aan andere threads.

Yield: voorbeeld (2)

```
public class StartYielding {
    public static void main(String[] args) {
        for (int i = 0; i < 3; i++)
            new YieldingThread();
    }
}
```

- Met behulp van de `yield` methode kan je ervoor zorgen dat alle Runnabe threads evenveel kans hebben om te runnen
- Kan je de uitvoer van dit programma voorspellen?
- Wat als je `yield()` weglaat?

```
#1: 5
#2: 5
#2: 4
#3: 5
#3: 4
#1: 4
#1: 3
#3: 3
#3: 2
#2: 3
#2: 2
#3: 1
#1: 2
#1: 1
#2: 1
```

```
#1: 5
#1: 4
#1: 3
#1: 2
#2: 5
#1: 1
#3: 5
#3: 4
#2: 4
#3: 3
#2: 3
#3: 2
#2: 2
#3: 1
#2: 1
```

Slides Opdracht 4

- Bestudeer de code van de module **4_Yielding_Thread** en voer dan het programma een aantal keer uit.
  Wat merk je op?

- Verwijder *Thread.yield()* uit het programma en kijk wat er gebeurt als je het een aantal keren uitvoert.
  Heeft dit effect?

Voorlopige conclusie:
yield en **setPriority** geven hints, die kunnen genegeerd worden. Afhankelijk van implementatie, OS, aantal processoren…
Er zijn betere systemen om het "verkeer te regelen" tussen threads zoals: **join, synchronized, wait, notify** (zie volgende week)

Join: voorbeeld (1)

```
public class DemoJoin {
    private List<String> threadNamen = new ArrayList<>();
    private Random random = new Random();

    private void startThreads(int aantal) {
        Thread[] threads = new Thread[aantal];
        for (int i = 0; i < threads.length; i++) {
            threads[i] = new DemoJoin.MyThread();
            threads[i].start();
        }
        for (Thread thread : threads) {
            try {
                thread.join();
            } catch (InterruptedException e) {
                // Leeg
            }
        }
    }
```

join zorgt ervoor dat de **main**-thread niet stopt vooraleer deze thread ten einde is.

Join (vervolg zelfde klasse)

```
    public static void main(String[] args) {
        DemoJoin test = new DemoJoin();
        test.startThreads(10);
        System.out.println(test.threadNamen);
    }

    private class MyThread extends Thread {
        public void run() {
        try {
          sleep(random.nextInt(1000));
        } catch (InterruptedException e) {// ignore}
            threadNamen.add(getName());
        }
    }
}
```

- Dit programma start 10 threads en plaatst op einde van de run methode telkens de naam van de thread in de ArrayList.

Slides Opdracht 5

- Bekijk de code van de module **5_Joining_Threads**. Run het programma en verklaar de output:
- Mogelijke uitvoer:

```
[Thread-2, Thread-0, Thread-1, Thread-5, Thread-4,
Thread-3, Thread-7, Thread-6, Thread-8, Thread-9]
```

- Zonder het join gedeelte:

```
[] (meestal)
```

KdG
Karel de Grote Hogeschool

Verklaring?

Slides Opdracht 6

- Bestudeer de code van de module **6_Joining_Threads** en voer het programma een aantal keer uit.
- Zet het try-catch gedeelte in de `main` in commentaar en voer opnieuw uit.
  - Wat valt er op?
  - Maak nu van de 10 threads Daemon-threads. Wat verandert er?

Opdrachten

- Groeiproject
  - module 9
    (deze week enkel deel 1 en 2: “Threads”)

- Opdrachten op BB
  - Thread LifeCycle
  - Streams
  - Factoren Priemgetallen
  - Bouncing balls
- Zelftest op BB