Java Advanced

Multithreading

DE HOGESCHOOL
MET HET NETWERK

Hogeschool PXL – Elfde-Liniestraat 24 – B-3500 Hasselt
www.pxl.be - www.pxl.be/facebook

Multithreading

1. Wat is multithreading?
2. Toepassingen
3. Implementatie
4. Thread life cycle
5. Thread synchronisatie
6. Timer en TimerTask
7. Concurrency framework
8. Parallellisme met streams

Wat is multithreading?

Thread = sub-proces met taak

- 1 taak tegelijk
- “main thread”

Single-threaded applicaties

- Langdurige taak blokkeert main thread

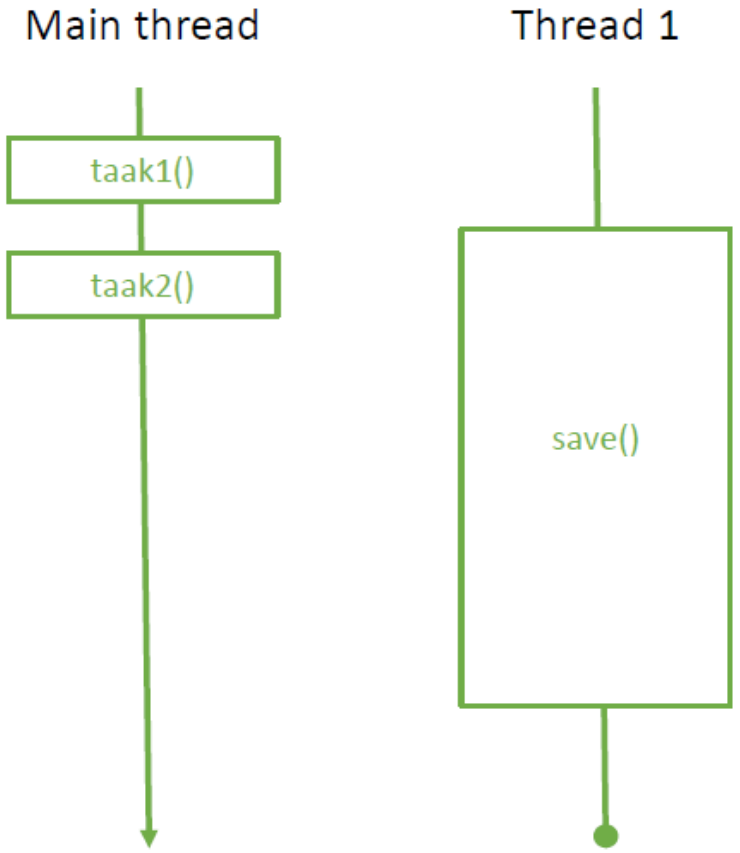
Multi-threaded:

- Parallelle threads voor deeltaken

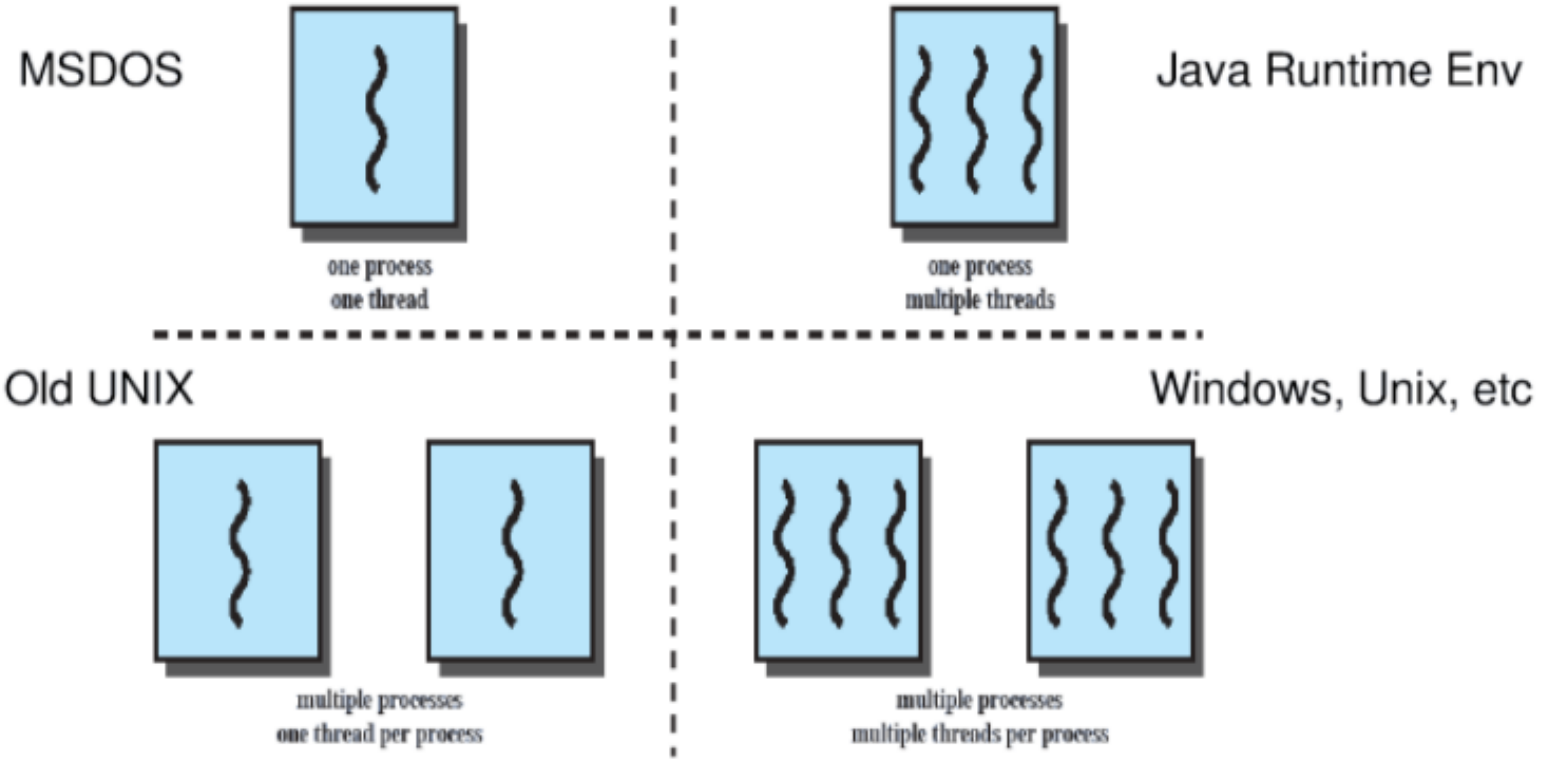
Wat is multithreading?

Multithreading:

- Threads met deeltaak
- Parallel uitgevoerd
- Main thread blijft responsief
- Thread afgesloten na uitvoering



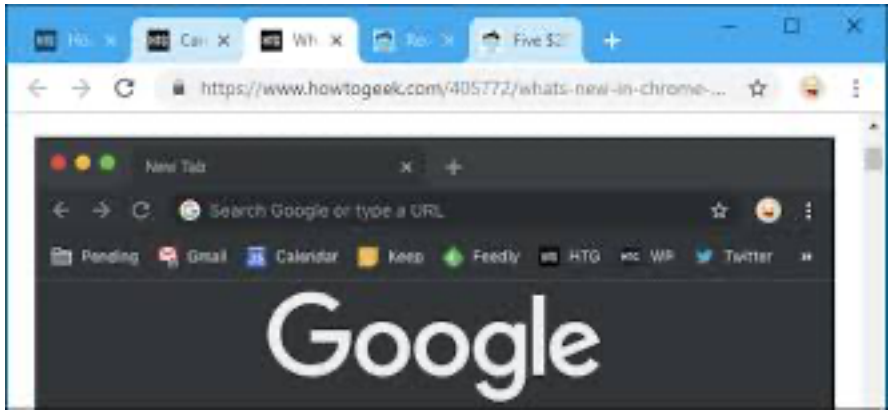
Processen en threads

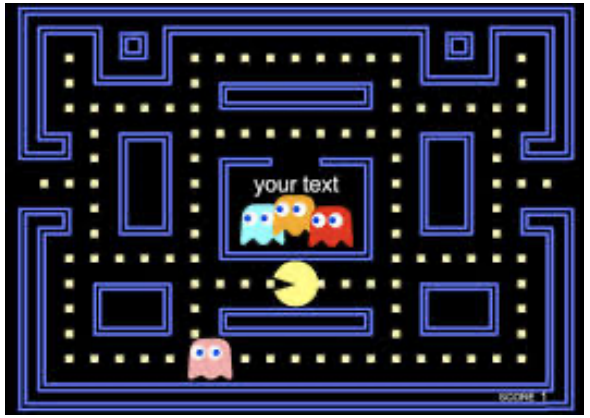


Toepassingen

Wanneer is multithreading nuttig?

Waar kan multithreading gebruikt worden?





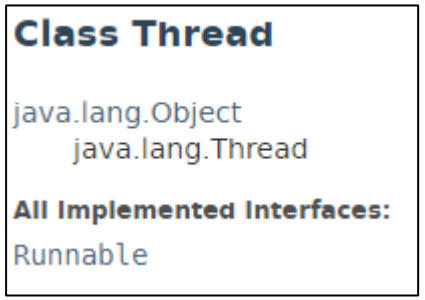
Implementatie

- *Runnable* interface implementeren
  - *run()* methode bevat code die door thread wordt uitgevoerd
  - Minimale vereiste om thread te maken

https://docs.oracle.com/en/java/javase/11/docs/api/java.base/java/lang/Runnable.html

- *Thread* klasse overerven
  - Implementeert zelf *Runnable*
  - Voegt extra functies toe

https://docs.oracle.com/en/java/javase/11/docs/api/java.base/java/lang/Thread.html



Thread

```
public class WorkerThread extends Thread {

    @Override
    public void run() {
        System.out.println("Executing thread");
    }

    public static void main(String[] args) {
        new WorkerThread().start();
    }
}
```

Runnable

```
public class WorkerThread implements Runnable {

    @Override
    public void run() {
        System.out.println("Executing thread");
    }

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

Runnable (with lambda)

```
public class WorkerThread {

    public static void main(String[] args) {
        new Thread(() -> System.out.println("Executing thread")).start();
    }
}
```

Runnable of Thread

- Thread lijkt ‘handiger’
- Extra methoden, makkelijk bruikbaar

- **Maar:**
  - Klasse kan maar overerven van 1 andere klasse
  - Runnable makkelijker toe te voegen aan bestaande klasse

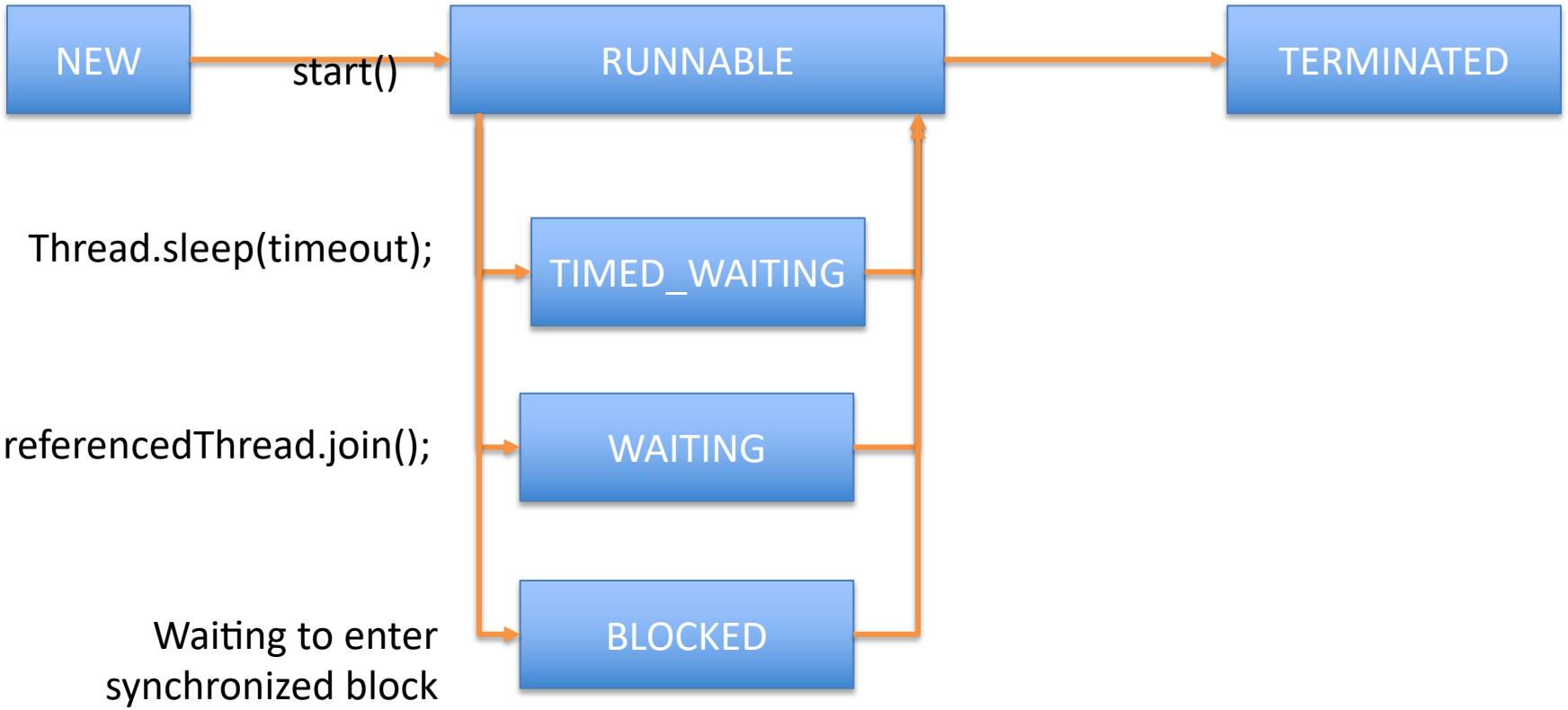
run() vs start()

- Thread implementatie
  → run()

- Thread opstarten/uitvoeren
  → start()

- start()
  - De thread opstarten
  - De code geïmplementeerd in de run() method wordt uitgevoerd

Thread life cycle



Thread life cycle

- Uitvoering
  - Opstarten
  - Uitvoeren taak (*run()*)
  - Beëindigen

- Statussen
  - NEW: aangemaakt, nog niet gestart
  - RUNNABLE: Gestart. (Ready-to-run of Running)
  - TIMED_WAITING, WAITING, BLOCKED: Uitvoering gepauzeerd
  - TERMINATED: Taak uitgevoerd

Thread scheduler

- 1 actieve thread per processor
- Veel threads, ‘weinig’ processoren
- Threads delen processor

Thread scheduler

- Bepaalt welke thread mag uitvoeren (en hoe lang)
- Verschillende mechanismes spelen rol
- Onderdeel van JVM
- In samenspraak met onderliggend OS

Thread.sleep(timeout)

- Tijdelijk in wachttoestand
  - Thread.sleep*(milliseconds)*
  - Altijd van toepassing op huidige Thread
- Running → Waiting
  - Uit wachttoestand halen
    - Door timeout
    - Aanroepen method interrupt()

referencedThread.join()

- Geef voorrang aan referencedThread en wacht tot deze beëindigd is.

Opgave 1a

- Maak een klasse ***Talker*** die overerft van *Thread*. Aan de constructor kan je een ID mee geven.
- Bij uitvoeren van de thread, moet 10x het ID afgeprint worden, met telkens een halve seconde er tussen. (*sleep(500);*)
- **Maak en start** in de main 4 instanties van *Talker.*

Opgave 1b

- Doe de nodige aanpassingen om *Talker* nu de *Runnable* interface te laten gebruiken.
- Wat moest er veranderen?

Opgave 1c

Welke statussen worden hier afgedrukt?

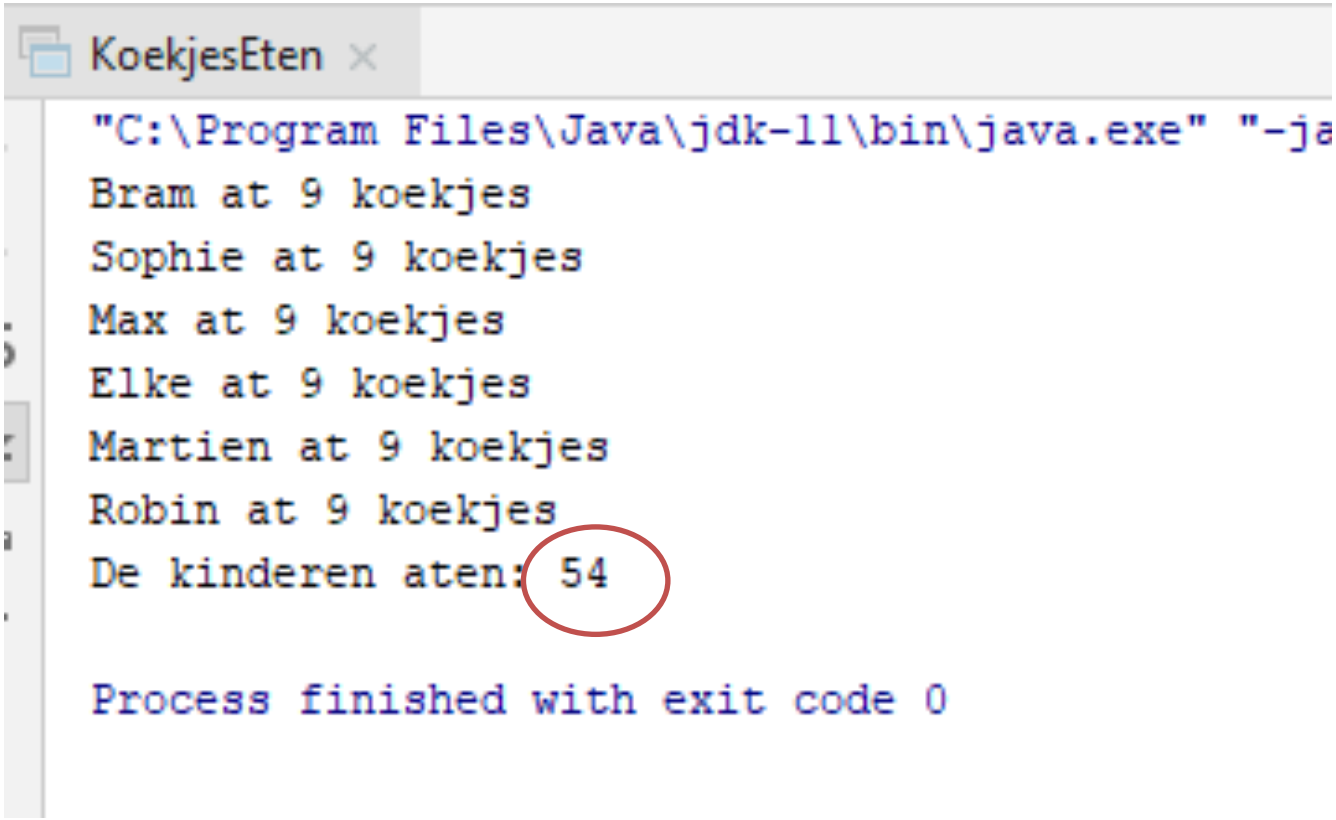
```
public static void main(String args[]) {
    Talker talker = new Talker();
    System.out.println(talker.getState());

    talker.start();
    System.out.println(talker.getState());

    try {
        talker.join();
    } catch (InterruptedException e) {
        e.printStackTrace();
    }
    System.out.println(talker.getState());
}
```

Thread synchronisatie

```
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
"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

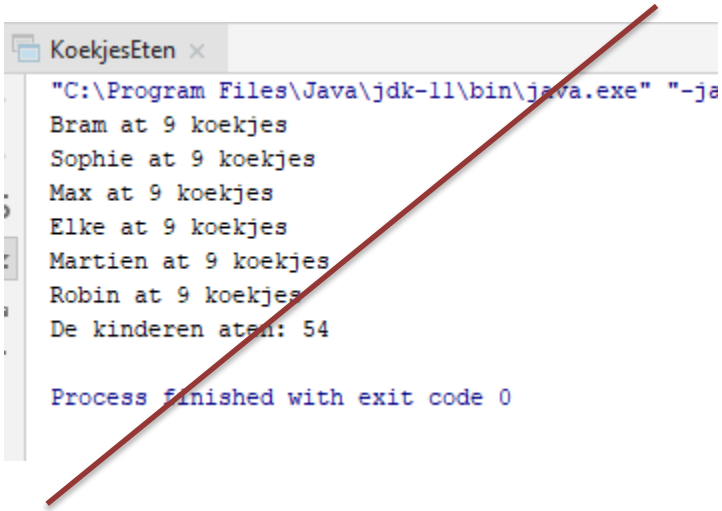
Thread Bram: neemKoekje()

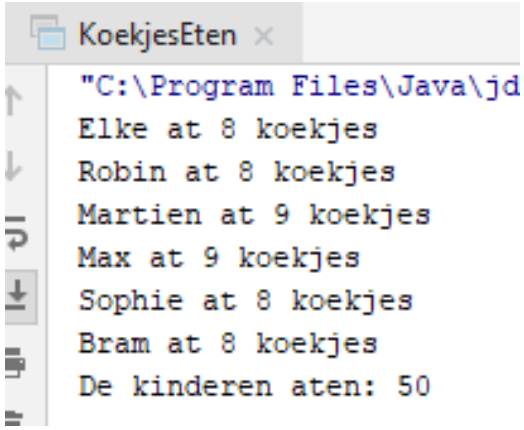
```
if (aantalKoekjes > 0)

aantalKoekjes--
[aantalKoekjes -> -1]
```

Thread Elke: neemKoekje()

```
if (aantalKoekjes > 0)
    aantalKoekjes--
    [aantalKoekjes -> 0]
```

aantalKoekjes = 1






```
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;
    }
}
```

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()`

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());
    }
}
```

Concurrency framework

- Dient om het ontwikkelen van *multithreaded* applicaties makkelijker te maken.

- `java.util.concurrent`
  - Concurrent collections
  - Atomaire objecten
  - Callable, ExecutorService and Future

Concurrent collections

`java.util.Collections`

Methode
`synchronizedCollection()`
`synchronizedList()`
`synchronizedNavigableMap()`
`synchronizedNavigalbeSet()`
`synchronizedSet()`
`synchronizedSortedMap()`
`synchronizedSortedSet()`

Concurrency – atomic variables

`java.util.concurrent.atomic`

Klasse	*Omschrijving*
`AtomicBoolean`	Atomaire boolean.
`AtomicInteger`	Atomaire integer.
`AtomicIntegerArray`	Atomarie reeks van integers.
`AtomicLong`	Atomaire long.
`AtomicLongArray`	Atomaire reeks van longs.
`AtomicReference`	Atomaire referentie.
`AtomicReferenceArray`	Atomare reeks van referenties.

Concurrency - atomic variables

```
public class Koekjesdoos {

   private AtomicInteger aantalKoekjes;

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

   public boolean neemKoekje() {
      int result = aantalKoekjes.getAndDecrement();
      return result > 0;
   }
}
```

Concurrency
Callable, ExecutorService and Future

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

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()`

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`

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));
    }
}
```

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

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.

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));
    }
}
```