HoGent
BEDRIJF
EN
ORGANISATIE

Hoofdstuk 23 :

Concurrency

H 23: MULTITHREADING

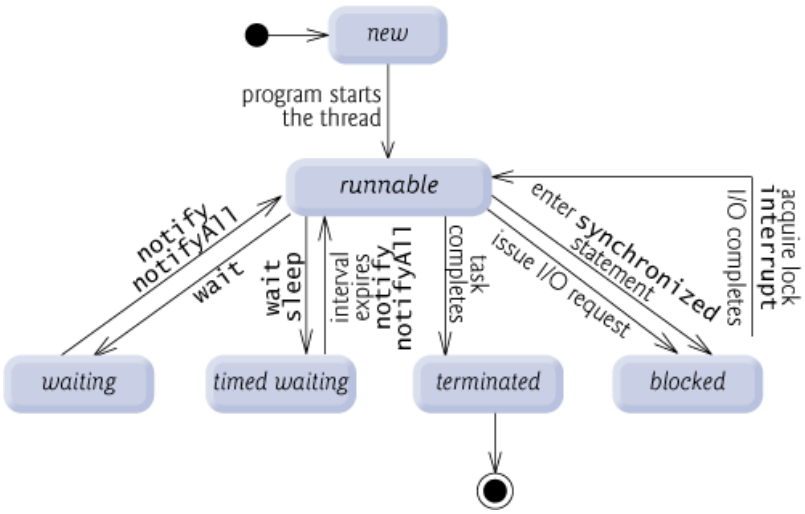
1. INLEIDING

- Threads: delen van het programma die in concurrentie met elkaar gelijktijdig in executie gaan.
  - Thread is een sequentiële besturingsstroom. Het zijn ‘lichtgewicht’ processen.

1. INLEIDING

- Java voorziet primitieven voor multithreading.
  - Meeste programmeertalen moeten gebruik maken van OS-primitieven voor multithreading en dus platformspecifieke code gebruiken.
  - De werking van Java's thread scheduling is wel platform afhankelijk.
  - Een voorbeeld van multithreading is Java's garbage collector.

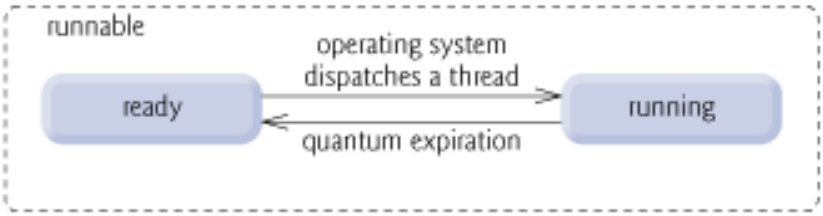
2. THREAD TOESTANDEN: LEVENSCYCLUS



2. THREAD TOESTANDEN: LEVENSCYCLUS

- Thread toestanden.
  - **new state**
    - Thread is juist gecreëerd.
  - **runnable state**
    - **start** method van thread geactiveerd.
    - Thread wordt beschouwd als in executie zijnde.
  - **waiting state**
    - Een deel code nog niet kan uitgevoerd worden (bepaalde vereisten moeten voldaan zijn)
      - De thread activeert Object's **wait** methode. Keert terug naar runnable state doordat een andere thread de **notify** methode activeert.
      - De thread wordt geblokkeerd (lock), door bv io verzoek totdat die gedeblokkeerd wordt (unlock).
  - **timed waiting state**
    - Wacht op andere thread of het einde van opgegeven tijdsinterval.
    - Wacht tot opgegeven "sleep" tijdsinterval is verstreken.
  - **terminated state**
    - Thread is beëindigd (taak volbracht of exit).
    - Garbage collector kan geheugen terug vrijgeven als er geen referentie meer is naar het thread object.
  - **blocked state**
    - Als een taak niet onmiddellijk kan volbracht worden (vb. i/o verzoek).

2. THREAD TOESTANDEN: LEVENSCYCLUS



- **runnable state (← JVM)**
  - **Ready state (← OS)**
    - Initiële state, timeslice/quantum vervallen.
  - **Running state (← OS)**
    - Thread is toegekend aan een processor en in executie (**dispatching**).

→ Thread scheduling is functie van OS en thread priorities.

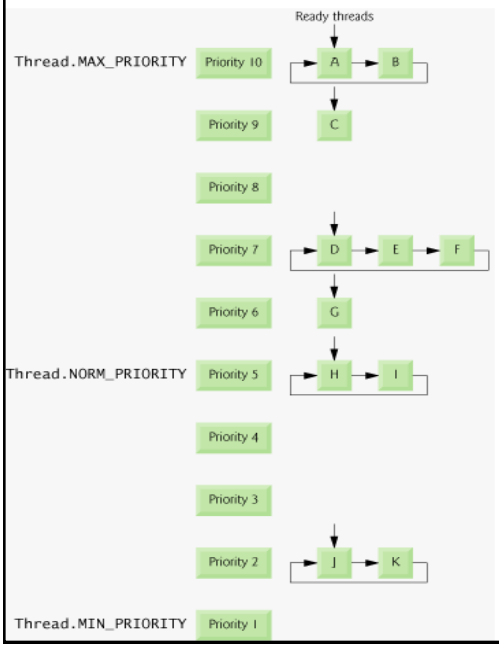
THREAD PRIORITIES EN SCHEDULING

- Priority: indicatie van belangrijkheid (→voorrang voor processortijd).
  - Thread.MIN_PRIORITY (1) .. Thread.NORM_PRIORITY (5) .. Thread.MAX_PRIORITY (10).
- Thread scheduler: zorgt ervoor dat de thread met hoogste prioriteit in **running** state verkeert. Bij meerdere threads met gelijke prioriteit wordt timeslicing gebruikt. Elke thread krijgt een 'quantum' processortijd toegewezen.
  - Niet elk Java platform ondersteunt timeslicing. De Thread methode **yield** kan er dan voor zorgen dat threads van gelijke prioriteit kunnen concurreren.

HoGent

7

THREAD PRIORITIES EN SCHEDULING



8

Nieuw vanaf Java7:

Klasse: **ThreadLocalRandom**

Binnen threads is het efficiënter om gebruik te maken van de nieuwe klasse **ThreadLocalRandom**, om aan randomwaarden te geraken.

Deze klasse heeft een static methode **current**().
Vb: int x = ThreadLocalRandom. current(). nextInt(2000));
x zal een waarde tussen 0 en 1999 toegekend krijgen.

Deze slides zijn niet aangepast om met deze nieuwe klasse te werken. Random wordt hier enkel gebruikt om een random vertraging (met **Thread.sleep**(randomwaarde)) te voorzien voor de simulaties.

3. CREATIE EN EXECUTIE VAN THREADS

- In J2SE 5.0 maken we een multithreaded applicatie door de Runnable interface te gebruiken. Tevens maken we gebruik van een “thread pool”. Een thread pool bevat een aantal threads; elke thread voert een runnable-object uit.
- Demonstratie van **sleep** methode. Creatie van 3 runnables met default prioriteit die op willekeurig tijdsinterval een bericht tonen.

```
// class PrintTask bestuurt de thread executie
public class PrintTask implements Runnable
{
    private final int sleepTime; // willekeurige tijdsinterval
    private final String threadName; // naam van de thread
    private static final SecureRandom generator = new SecureRandom();
```

3. CREATIE EN EXECUTIE VAN THREADS

```
public PrintTask( String name )
{
   // geef de thread een naam
  threadName = name;

   // kies een willekeurige slaaptijd tussen 0 en 5 seconden
   sleepTime = generator.nextInt( 5000 );

}
```


3. CREATIE EN EXECUTIE VAN THREADS

```
// method run wordt automatisch geactiveerd bij een nieuwe thread
// in de runnable state
public void run() {
    try // breng thread in sleep toestand voor sleepTime seconde
    {
      System.out.printf( "%s going to sleep for %d %n",
        threadName, sleepTime );

      Thread.sleep( sleepTime );
    }
    // als de thread gedurende de sleep toestand wordt onderbroken,
    //print stack trace
    catch ( InterruptedException exception )
    {
      exception.printStackTrace();
      Thread.currentThread().interrupt(); //re-interrupt thread
    }
    // print thread name
    System.out.printf( "%s done sleeping%n", threadName );
 }
}
```

3.1 CREATIE EN EXECUTIE VAN THREADS: KLASSE THREAD

```
// Meerdere threads die op verschillende tijdsintervallen printen.
public class ThreadTester
{   public static void main( String [] args )
  { // creatie van de drie threads in Born toestand
        Thread thread1 = new Thread (new PrintTask( "thread1" ));
        Thread thread2 = new Thread ( new PrintTask( "thread2" ));
        Thread thread3 = new Thread ( new PrintTask( "thread3" ));
        System.err.println( "Starting threads" );
        thread1.start(); // Plaats thread1 in Ready toestand
        thread2.start(); // Plaats thread2 in Ready toestand
        thread3.start(); // Plaats thread3 in Ready toestand
        System.err.println( "Threads started, main ends\n" );
  }
}
```

3.2 CREATIE EN EXECUTIE VAN THREADS: EXECUTOR FRAMEWORK (J2SE 5.0)

```
// Meerdere threads die op verschillende tijdsintervallen printen.
import java.util.concurrent.Executors;
import java.util.concurrent.ExecutorService;

public class RunnableTester
{
   public static void main( String[] args )
   {
      // creatie van de drie runnables
      PrintTask task1 = new PrintTask( "thread1" );
      PrintTask task2 = new PrintTask( "thread2" );
      PrintTask task3 = new PrintTask( "thread3" );
```

```
System.out.println( "Starting threads" );
// een “thread pool” creëren met drie threads
ExecutorService threadExecutor = Executors. newFixedThreadPool(3);

// thread1 van de thread-pool wordt gekoppeld aan de Runnable “task1”.
//thread1 is nu in de Ready toestand.
threadExecutor.execute( task1 );
threadExecutor.execute( task2 ); // Plaats thread2 in Ready toestand
threadExecutor.execute( task3 ); // Plaats thread3 in Ready toestand

threadExecutor.shutdown(); // de “thread pool” wordt afgesloten
wanneer de drie threads van de pool beëindigd zijn (m.a.w. wanneer
thread1, thread2 en thread3 in terminated state zijn)
System.out.println( "Threads started, main ends\n" );
}                    // end main   } // end class RunnableTester
```

3. CREATIE EN EXECUTIE VAN THREADS

- Als een thread voor de eerste keer in de **running** toestand komt wordt de methode run geactiveerd.
- Ook al is de maincode beëindigd, het programma eindigt pas als alle threads de **terminated** toestand bereikt hebben.

```
Starting threads
Threads started, main ends

thread1 going to sleep for 1217
thread2 going to sleep for 3989
thread3 going to sleep for 662
thread3 done sleeping
thread1 done sleeping
thread2 done sleeping
```

```
Starting threads
thread1 going to sleep for 314
thread2 going to sleep for 1990
Threads started, main ends

thread3 going to sleep for 3016
thread1 done sleeping
thread2 done sleeping
thread3 done sleeping
```

4. THREAD SYNCHRONISATIE

- Meerdere threads kunnen een object delen, '**shared object**'.
- Wanneer meerdere threads het shared object kunnen wijzigen kunnen er problemen ontstaan.
  - → mutual exclusion of thread **synchronisatie**.
  - → in Java gebruik van **locks** voor synchronisatie.
  - → **Lock** interface (java.util.concurrent.locks).
  - → Klasse **ReentrantLock** implementatie van Lock.

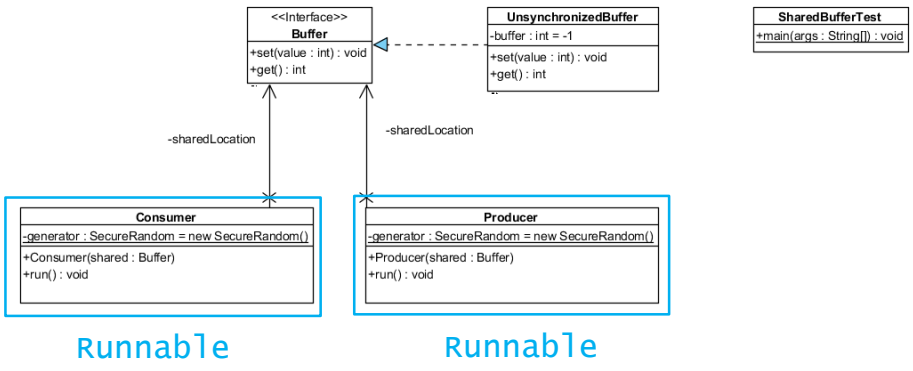
4. THREAD SYNCHRONISATIE

- **Elk object heeft een monitor:** geeft maar één thread tegelijkertijd executierecht bij een synchronized statement op het object, 'obtaining the lock'. Andere threads komen in **waiting** toestand (**lock**). Als de lock wordt vrijgegeven, 'released', zal de monitor de thread met hoogste priority laten voortgaan in **runnable** state (**unlock**).
- Een volledige methode kan **synchronized** zijn of je kan een synchronized block maken.
- **Deadlock preventie:** als een thread de **wait** methode activeert, zorg dan dat een afzonderlijke thread de **notify** methode activeert voor terugkeer naar de **runnable** state.

5. PRODUCER/CONSUMER RELATIE ZONDER SYNCHRONISATIE

- Mogelijke logische fouten als we **geen synchronisatie** gebruiken: VOORBEELD SharedBufferTest.
  - Producer genereert een reeks getallen van 1 tot 10. Consumer telt het aantal getallen en sommeert die getallen. Bij beide threads is een random vertraging ingebouwd (0 tot 3 seconden sleep) om een reëel programma te simuleren. Bij multithreaded applicaties is het ongekend wanneer een thread zijn taak uitvoert en hoelang dat duurt.
  - Beide threads drukken ook de getallen af.
  - Componenten: Interface Buffer en Klassen Producer, Consumer, UnsynchronizedBuffer en SharedBufferTest.

5. PRODUCER/CONSUMER RELATIE ZONDER SYNCHRONISATIE



5. PRODUCER/CONSUMER RELATIE ZONDER SYNCHRONISATIE

```
public interface Buffer
{  public void set( int value );  // plaats een waarde in de buffer
   public int get();              // haal een waarde uit de buffer
}

import java.security.SecureRandom;
public class Producer implements Runnable
{
   private static final SecureRandom generator = new SecureRandom();
   private final Buffer sharedLocation;// reference naar shared object

   public Producer( Buffer shared )
   {   sharedLocation = shared;
   }
```

5. PRODUCER/CONSUMER RELATIE ZONDER SYNCHRONISATIE

```
 public void run() // berg waarden 1 tot 10 op in sharedLocation
 {     int sum = 0;
       for ( int count = 1; count <= 10; count++ )
       {   try  {  // sleep 0 tot 3 seconden en plaats waarde in buffer
                   Thread.sleep( generator.nextInt( 3000 ) );
                   sharedLocation.set( count );
                   sum += count; // increment sum of values
                   System.out.printf( "\t%2d%n", sum );
               }
          catch ( InterruptedException exception )  {   exception.printStackTrace();
                                                        Thread.currentThread().interrupt();   }
       }
       System.out.printf( “%n%s%n%s%n", "Producer done producing.",
       "Terminating Producer." );
  }
} // einde klasse Producer
```

5. PRODUCER/CONSUMER RELATIE ZONDER SYNCHRONISATIE

```
public class Consumer implements Runnable
{
    private static final SecureRandom generator = new SecureRandom();
    private final Buffer sharedLocation;// referentie naar shared object

    public Consumer( Buffer shared )
    {
        sharedLocation = shared;
    }
```


5. PRODUCER/CONSUMER RELATIE ZONDER SYNCHRONISATIE

```
public void run()  // lees sharedLocation's waarde 10 keer en sommeer de waarden
{   int sum = 0;
    for ( int count = 1; count <= 10; count++ )
    {   try    // sleep 0 to 3 seconden en lees waarde uit de buffer en tel bij som
        {    Thread.sleep( generator.nextInt( 3000 ) );
             sum += sharedLocation.get();
             System.out.printf( "\t\t\t%2d%n", sum );
        }
        catch ( InterruptedException exception )
        {   exception.printStackTrace();
            Thread.currentThread().interrupt();
        }
    }
    System.out.printf( "%n%s %d.%n%s%n",
      "Consumer read values totaling", sum, "Terminating Consumer." );
} } // einde klasse Consumer
```

5. PRODUCER/CONSUMER RELATIE ZONDER SYNCHRONISATIE

```
public class UnsynchronizedBuffer implements Buffer
{   private int buffer = -1; // gedeeld door producer en consumer threads
    // plaats waarde in buffer
    public void set( int value )
    {    System.out.printf( "Producer writes\t%2d", value );
         buffer = value;
    }
    // return waarde uit buffer
    public int get()
    {    System.out.printf( "Consumer reads\t%2d", buffer );
        return buffer;
    }
}
```

5. PRODUCER/CONSUMER RELATIE ZONDER SYNCHRONISATIE

```
public class SharedBufferTest
{
   public static void main( String [] args )
     // een “thread pool” creëren met twee threads
     ExecutorService application = Executors.newFixedThreadPool( 2 );

    // instantiatie van het shared object gebruikt door de runnables
     Buffer sharedLocation = new UnsynchronizedBuffer();

     System.out.println( "Action\t\tValue\tProduced\tConsumed" );
     System.out.println( "------\t\t-----\t--------\t--------\n" );
```

5. PRODUCER/CONSUMER RELATIE ZONDER SYNCHRONISATIE

```
try
 { // instantiatie van producer and consumer objecten
   Producer producer =  new Producer( sharedLocation );
   Consumer consumer = new Consumer(sharedLocation);
   // de threads van  de “thread pool” koppelen met de runnables:
   application.execute(producer); // start thread1 (thread1 is
                          //gekoppeld aan de runnable “producer”
   application.execute(consumer); // start thread2
 }
  catch ( Exception exception ) { exception.printStackTrace();
  }
  application.shutdown();
}
} // end class SharedBufferTest
```

```
Action          Value   Produced        Consumed
------          -----   --------        --------

Consumer reads  -1                      -1
Consumer reads  -1                      -2
Producer writes  1       1
Consumer reads   1                      -1
Producer writes  2       3
Producer writes  3       6
Consumer reads   3                       2
Producer writes  4      10
Producer writes  5      15
Consumer reads   5                       7
Consumer reads   5                      12
Consumer reads   5                      17
Producer writes  6      21
Producer writes  7      28
Consumer reads   7                      24
Producer writes  8      36
Consumer reads   8                      32
Producer writes  9      45
Producer writes 10      55

Producer done producing.
Terminating Producer.
Consumer reads  10                      42

Consumer read values totaling 42.
Terminating Consumer.
```

HoGent

28

```
Action          Value   Produced        Consumed
------          -----   --------        --------

Producer writes  1       1
Producer writes  2       3
Consumer reads   2                       2
Producer writes  3       6
Consumer reads   3                       5
Consumer reads   3                       8
Producer writes  4      10
Consumer reads   4                      12
Producer writes  5      15
Producer writes  6      21
Consumer reads   6                      18
Producer writes  7      28
Consumer reads   7                      25
Consumer reads   7                      32
Producer writes  8      36
Producer writes  9      45
Consumer reads   9                      41
Consumer reads   9                      50
Producer writes 10      55

Producer done producing.
Terminating Producer.
Consumer reads  10                      60

Consumer read values totaling 60.
Terminating Consumer.
```


6. PRODUCER/CONSUMER RELATIE MET SYNCHRONISATIE

Uit vorige voorbeeld blijkt synchronisatie noodzakelijk.

FUNDAMENTELE WERKING: methoden van klasse Object

- **In multithreaded applicatie:**
  - Componenten: producerthread, consumerthread en buffer.

  - **Producerthread:**
    Activeer **wait**: als vorige data nog niet verwerkt is, consumer kan data ophalen.
    Activeer **notify**: als nieuwe data in buffer geplaatst is.

  - **Consumerthread:**
    Activeer **notify**: als de data opgehaald is, producer kan nieuwe data in buffer plaatsen.
    Activeer **wait**: als buffer leeg is of vorige data nog aanwezig is.

6. PRODUCER/CONSUMER RELATIE MET SYNCHRONISATIE

PRAKTISCH SYNCHRONISATIE TOEPASSEN :

→ **methoden await en signal**

→ **Lock en Condition objecten**

- Threads die toegang nemen tot een shared object merken zelf niets van de synchronisatie. Synchronisatiecode komt in de **set** en **get** methoden van **SynchronizedBuffer**.
- Er wordt een extra attribuut gebruikt: **boolean occupied**. Hiermee wordt de communicatie geregeld, mag de buffer gevuld/geledigd worden?
- Het activeren van de **await** methode brengt de thread in **waiting state** waarbij de blokkering voor het gesynchroniseerd object beëindigd wordt (release lock).
- Het activeren van de **signal** methode brengt de thread terug in **runnable state**, die dan terug de lock kan proberen te verkrijgen.

6. PRODUCER/CONSUMER RELATIE MET SYNCHRONISATIE

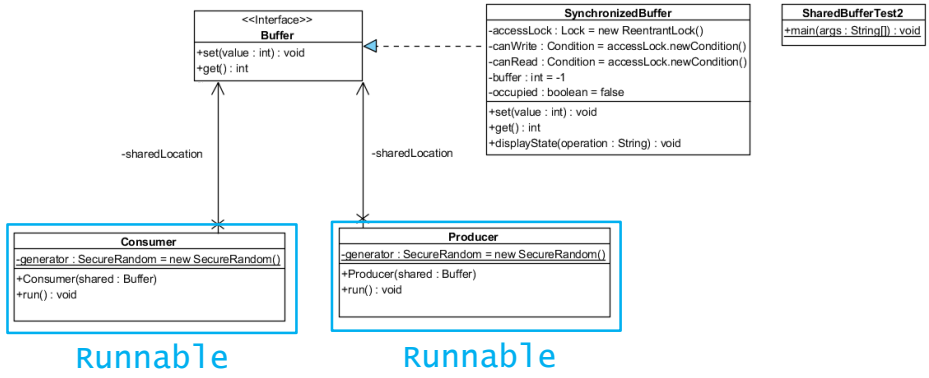
- Producer en Consumer gebruiken de methoden set en get van de klasse SynchronizedBuffer. In de klasse “SharedBufferTest2” wordt er een object van de klasse gecreëerd en doorgegeven aan Producer en Consumer:

```
Buffer sharedLocation = new SynchronizedBuffer();
Producer producer =  new Producer( sharedLocation );
    Consumer consumer = new Consumer(sharedLocation);
```

“sharedLocation” is het gesynchroniseerd object.

- Het gesynchroniseerd object wordt geblokkeerd door de methode **lock**.
- De blokkering van het gesynchroniseerd object wordt beëindigd door de methode **unlock** (release lock).

6. PRODUCER/CONSUMER RELATIE MET SYNCHRONISATIE



6. PRODUCER/CONSUMER RELATIE MET SYNCHRONISATIE

```
public class SynchronizedBuffer implements Buffer
{  // een object van ReentrantLock creëren zodat synchronisatie met deze
   // buffer mogelijk wordt.
   private Lock accessLock = new ReentrantLock();
   /* conditie "canWrite": kan de producer al dan niet schrijven in de
buffer. Indien de buffer vol is dan wordt de instructie "canWrite.await()"
opgeroepen; de producer moet wachten totdat de buffer leeg wordt.
Indien de buffer leeg wordt, dan wordt de instructie "canWrite.signal()"
opgeroepen; de producer kan dan een waarde plaatsen in de buffer.*/
   private Condition canWrite = accessLock.newCondition();

   private Condition canRead = accessLock.newCondition();
   private int buffer = -1; // gedeeld door producer en consumer threads
   private boolean occupied = false;
```

6. PRODUCER/CONSUMER RELATIE MET SYNCHRONISATIE

```
public void set( int value ) // plaats een waarde in de buffer
{
   accessLock.lock();
   // Het gesynchroniseerd object "sharedLocation "
   // wordt geblokkeerd (zie klasse SharedBufferTest2).
   try
   {                    // zolang buffer niet leeg,
      while ( occupied )      //plaats thread in waiting state
     {  // afdrukken buffer status
         System.out.println( "Producer tries to write." );
        displayState( "Buffer full. Producer waits." );
        canWrite.await(); // wachten totdat de buffer leeg is
     }
```

6. PRODUCER/CONSUMER RELATIE MET SYNCHRONISATIE

```
   buffer = value; // plaats nieuwe waarde in buffer

   // registreer dat buffer gevuld is… consumer moet buffer ledigen
   occupied = true;

   displayState( "Producer writes " + buffer );

   // verwittig de consumer dat hij de waarde in de buffer mag lezen.
   canRead.signal();
} // end try
```

HoGent

6. PRODUCER/CONSUMER RELATIE MET SYNCHRONISATIE

```
        catch ( InterruptedException exception )
        {     exception.printStackTrace();
              Thread.currentThread().interrupt();
         }
        finally
        {
              // beëindig de blokkering op het gesynchroniseerd
              // object “sharedLocation ” (release lock)
         accessLock.unlock();
        }
    } // end method set
```

6. PRODUCER/CONSUMER RELATIE MET SYNCHRONISATIE

```
  public int get() // levert waarde uit buffer
  {
     int readValue = 0;
     accessLock.lock();
    // Het gesynchroniseerd object “sharedLocation ”
   // wordt geblokkeerd (zie klasse SharedBufferTest2).
    try
    {
      while ( !occupied )
    // zolang buffer leeg, plaats thread in waiting state
      {  System.out.println( "Consumer tries to read." );
         displayState( "Buffer empty. Consumer waits." );
         canRead.await(); // wachten totdat de buffer vol is.
      }
```

```
        // registreer dat buffer leeg is… producer moet buffer vulllen:
        occupied = false;

        readValue = buffer; // waarde uit de buffer halen
        displayState( "Consumer reads " + readValue );
      // verwittig de producer dat hij een waarde in de buffer mag plaatsen.
        canWrite.signal();
    }
    catch ( InterruptedException exception ) { exception.printStackTrace();
                                Thread.currentThread().interrupt();}
    finally {  accessLock.unlock();  } // end finally
    return readValue;
} // end method get
```

6. PRODUCER/CONSUMER RELATIE MET SYNCHRONISATIE.

```
public void displayState( String operation )
{
    System.out.printf( "%-40s%d\t\t%b%n%n",
        operation, buffer, occupied );
  }

} // end class SynchronizedBuffer
```

6. PRODUCER/CONSUMER RELATIE MET SYNCHRONISATIE.

```
public class SharedBufferTest2
{  public static void main( String[] args )
   {  // een "thread pool" creëren met twee threads:
      ExecutorService application = Executors.newFixedThreadPool( 2 );

      // instantiatie van het shared object gebruikt door de runnables:
      Buffer sharedLocation = new SynchronizedBuffer();


      System.out.printf( "%-40s%s\t\t%s%n%-40s%s%n%n", "Operation",
        "Buffer", "Occupied", "---------", "------\t\t--------" );
      try  // threads v/d "thread pool" koppelen met de runnables Producer en Consumer:
      {    application.execute( new Producer( sharedLocation ) );
           application.execute( new Consumer( sharedLocation ) );
      } // end try
      catch ( Exception exception ) { exception.printStackTrace();
      } // end catch
      application.shutdown();
   } // end main      } // end class SharedBufferTest2
```

```
Operation                               Buffer          Occupied
---------                               ------          --------

Producer writes 1                       1               true

Consumer reads 1                        1               false

Consumer tries to read.
Buffer empty. Consumer waits.           1               false

Producer writes 2                       2               true

Consumer reads 2                        2               false

Consumer tries to read.
Buffer empty. Consumer waits.           2               false

Producer writes 3                       3               true

Consumer reads 3                        3               false

Producer writes 4                       4               true

Producer tries to write.
Buffer full. Producer waits.            4               true

Consumer reads 4                        4               false

Producer writes 5                       5               true

Producer tries to write.
Buffer full. Producer waits.            5               true
```

HoGent

```
Consumer reads 5                    5           false

Producer writes 6                   6           true

Consumer reads 6                    6           false

Consumer tries to read.
Buffer empty. Consumer waits.       6           false

Producer writes 7                   7           true

Consumer reads 7                    7           false

Consumer tries to read.
Buffer empty. Consumer waits.       7           false

Producer writes 8                   8           true

Consumer reads 8                    8           false

Consumer tries to read.
Buffer empty. Consumer waits.       8           false

Producer writes 9                   9           true

Consumer reads 9                    9           false
Producer writes 10                  10          true


Producer done producing.
Terminating Producer.
Consumer reads 10                   10          false


Consumer read values totaling 55.
Terminating Consumer.
```

OEFENING SYNCHRONISATIE: RESTAURANT

Simuleer de werking in een restaurant: er is één kok die orders klaarmaakt en er zijn twee kelners (Sofie en Hendrik) die de orders naar de klant brengen. Stel een random vertraging in van 0 tot 2 seconden bij de kok en de kelners.

Na 10 orders sluit het restaurant (voedsel is op).

```
Kelner Hendrik krijgt Order 1
Kelner Sofie krijgt Order 2
Kelner Hendrik krijgt Order 3
Kelner Sofie krijgt Order 4
Kelner Hendrik krijgt Order 5
Kelner Sofie krijgt Order 6
Kelner Hendrik krijgt Order 7
Kelner Sofie krijgt Order 8
Kelner Hendrik krijgt Order 9
Kelner Sofie krijgt Order 10
Voedsel is op, sluiten!
```

OEFENING SYNCHRONISATIE: RESTAURANT

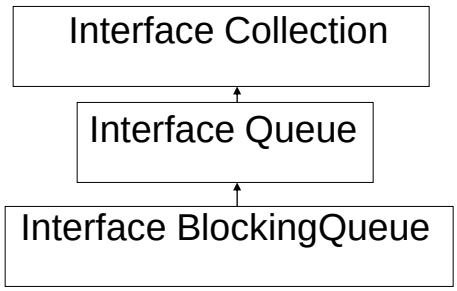
Gebruik onderstaande klasse voor de orders:

```
class Order
{   private static int i = 1;
    private int count = i++;
    public Order()
   {  if (count > 10)
      {   System.out.println("Voedsel is op, sluiten!");
          System.exit(0);
      }
   }
   public String toString()
   {   return String.format("Order %d“, count);
   }
}
```

HoGent

7. PRODUCER/CONSUMER RELATIE: ArrayBlockingQueue

- Nieuw sedert J2SE 5.0 is de klasse ArrayBlockingQueue.



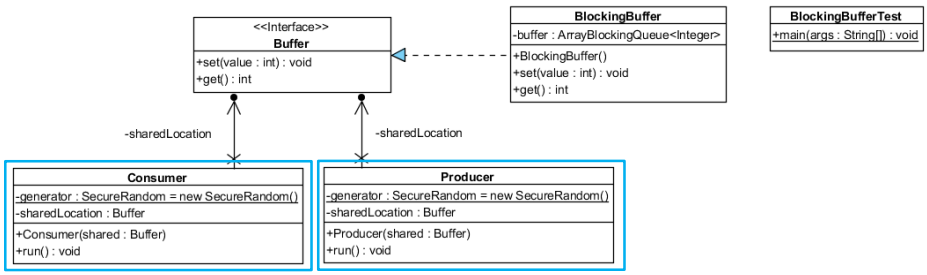
Klasse **ArrayBlockingQueue** is een implementatieklasse van interface **BlockingQueue**.

46

7. PRODUCER/CONSUMER RELATIE: ArrayBlockingQueue

- De **interface BlockingQueue** bevat de methoden put en take, welke equivalent zijn met de methoden offer en poll van de interface Queue.
  - Methode “put” zal een element in de BlockingQueue plaatsen (m.a.w. het element wordt achteraan in de wachtrij geplaatst.)
  - Methode “take” zal een element in de BlockingQueue ophalen (m.a.w. het eerste element in de wachtrij wordt opgehaald).
- De **klasse ArrayBlockingQueue** maakt gebruik van een array. De array wordt hier als een statische circulaire buffer gebruikt.

7. PRODUCER/CONSUMER RELATIE: ArrayBlockingQueue



```
import java.util.concurrent.ArrayBlockingQueue;

public class BlockingBuffer implements Buffer
{
  // declaratie van een circulaire buffer, m.a.w. een circulaire queue. De elementen van
  // de queue zullen van type Integer zijn.
   private ArrayBlockingQueue<Integer> buffer;

   public BlockingBuffer()
   { // creatie van de circulaire queue. De queue heeft als lengte 3.
     buffer = new ArrayBlockingQueue<Integer>( 3 );
   }
```

HoGent

```
  public void set( int value ) // plaats een waarde in de buffer
  {  try
     {  buffer.put( value ); // plaats de waarde "value" in de circulaire
        /* queue. Indien er geen plaats is in de queue, dan wordt er
        gewacht, totdat er terug een plaats vrijkomt. We hebben noch een
         Lock-, noch een Condition-object nodig vermits hier de
         synchronisatie automatisch gebeurt*/
        System.out.printf( "%s%2d\t%s%d%n", "Producer writes ", value,
         "Buffers occupied: ", buffer.size() );
      }
      catch ( Exception ex) {   ex.printStackTrace(); Thread.currentThread().interrupt();
      }
  }
```

```
public int get() // haal een waarde uit de buffer
{  int readValue = 0;
   try
   {  readValue = buffer.take(); /* haal de waarde uit de queue. Indien er
geen waarden zijn in de queue, dan wordt er gewacht, totdat er een
waarde in de queue werd geplaatst. Synchronisatie gebeurt hier ook
automatisch*/
      System.out.printf( "%s %2d\t%s%d%n", "Consumer reads ", readValue,
      "Buffers occupied: ", buffer.size() );
    }
    catch ( Exception ex )
    { ex.printStackTrace(); Thread.currentThread().interrupt(); } // end catch
   return readValue;
 } // einde methode get
} // einde klasse BlockingBuffer
```

```
import java.util.concurrent.ExecutorService;   import java.util.concurrent.Executors;
public class BlockingBufferTest
{   public static void main( String[] args )
    {   // een "thread pool" creëren met twee threads:
        ExecutorService application = Executors.newFixedThreadPool( 2 );
        // creatie van de gemeenschappelijke  buffer:
        Buffer sharedLocation = new BlockingBuffer();
        try // start de producer- en consumer-thread
        {   application.execute( new Producer( sharedLocation ) );
            application.execute( new Consumer( sharedLocation ) );
         }
        catch ( Exception ex) { ex.printStackTrace(); }
          application.shutdown();
     } // end main
} // end class BlockingBufferTest
```

```
Producer writes  1      Buffers occupied: 1
Consumer reads   1      Buffers occupied: 0
Producer writes  2      Buffers occupied: 1
Consumer reads   2      Buffers occupied: 0
Producer writes  3      Buffers occupied: 1
Consumer reads   3      Buffers occupied: 0
Producer writes  4      Buffers occupied: 1
Producer writes  5      Buffers occupied: 2
Consumer reads   4      Buffers occupied: 1
Producer writes  6      Buffers occupied: 2
Consumer reads   5      Buffers occupied: 1
Consumer reads   6      Buffers occupied: 0
Producer writes  7      Buffers occupied: 1
Consumer reads   7      Buffers occupied: 0
Producer writes  8      Buffers occupied: 1
Consumer reads   8      Buffers occupied: 0
Producer writes  9      Buffers occupied: 1
Consumer reads   9      Buffers occupied: 0
Producer writes 10      Buffers occupied: 1

Producer done producing.
Terminating Producer.
Consumer reads  10      Buffers occupied: 0

Consumer read values totaling 55.
Terminating Consumer.
```

HoGent

OEFENING ArrayBlockingQueue: SPEL VAT/ZWEMBAD

Er zijn een aantal kinderen en evenveel lege zwembadjes.
Er staat een groot vat met water gevuld, met een tafel ervoor die gebruikt wordt om een emmer met water te vullen.
De kinderen komen de emmer ophalen zodat ieder zijn eigen zwembad met water vult. Telkens een gevulde emmer van tafel gehaald wordt, zal er een nieuwe emmer met water gevuld worden, totdat het vat leeg is.
Als een kind zijn zwembad vol is stopt het met emmers halen, maar het stopt ook als het vat leeg is.

Als klassen hebben we **Zwembad**, **Kind**, **Tafel**, **Vat** en **Applicatie**. Als voorbeeld doe je een simulatie voor drie kinderen. Het vat bevat 9 emmers, een zwembad is vol bij 4 emmers. De tafel kan 2 emmers bevatten.

Het vullen van een emmer duurt tussen 1 à 2 seconden (telkens random kiezen). Een emmer naar het zwembad dragen en terug duurt 2 à 3 seconden (telkens random kiezen).

HoGent

OEFENING ArrayBlockingQueue: SPEL VAT/ZWEMBAD

```
emmer is gevuld
emmer is gevuld
emmer is gevuld
Kind 2 heeft een emmer genomen
Kind 3 heeft een emmer genomen
Kind 1 heeft een emmer genomen
emmer is gevuld
emmer is gevuld
emmer is gevuld
Kind 2 heeft een emmer genomen
emmer is gevuld
emmer is gevuld
Kind 1 heeft een emmer genomen
Kind 1 heeft een emmer genomen
emmer is gevuld
vat is leeg
Kind 2 heeft een emmer genomen
Kind 2 : reeds 3 emmers
Kind 3 heeft een emmer genomen
Kind 3 : reeds 2 emmers
Kind 1 heeft een emmer genomen
Kind 1 : zwembad vol
```

```
emmer is gevuld
emmer is gevuld
emmer is gevuld
emmer is gevuld
emmer is gevuld
Kind 3 heeft een emmer genomen
emmer is gevuld
Kind 1 heeft een emmer genomen
emmer is gevuld
Kind 2 heeft een emmer genomen
emmer is gevuld
Kind 3 heeft een emmer genomen
emmer is gevuld
vat is leeg
Kind 1 heeft een emmer genomen
Kind 2 heeft een emmer genomen
Kind 3 heeft een emmer genomen
Kind 3 : reeds 3 emmers
Kind 1 heeft een emmer genomen
Kind 1 : reeds 3 emmers
Kind 2 heeft een emmer genomen
Kind 2 : reeds 3 emmers
```

HoGent

GEGEVEN: Klasse Zwembad

```
package domein;

public class Zwembad
{
   private final int CAPACITEIT;
   private int inhoud;

   public Zwembad(int cap) { CAPACITEIT = cap; }

   public void gietEmmer() { inhoud++; }
   public void haalEmmer() { inhoud--; }

   public boolean vol() { return inhoud == CAPACITEIT; }
   public boolean leeg() { return inhoud == 0; }
   public int getInhoud() { return inhoud; }
}
```

VUL AAN: Klasse Applicatie

```
public class Applicatie
{  private Vat vat;
   private Kind kind[];

   public static void main(String args[]) { new Applicatie(); }

   public Applicatie()
   {
      Tafel tafel = new Tafel(2);
            vat = new Vat(9,tafel);
      kind = new Kind[3];
            for (int i = 0; i < kind.length; i++)
                  kind[i] = new Kind(tafel, new Zwembad(4), "Kind " + (i+1));

      . . .
```

8. MULTITHREADING WITH GUI

- Gui-componenten zijn niet thread-safe.
  De interactie met gui-componenten mag niet door meerdere threads 'tegelijkertijd' gebeuren, anders lopen we het risico dat de resultaten verkeerd zijn.

  Daarom gebruiken we één thread die toegang geeft tot de gui-componenten.

- Voor **JavaFX** is dat de **JavaFX Application thread**.

- Alle taken, waarvoor gui-activiteit nodig is, worden in een queue geplaatst. Deze voert de taken sequentieel uit. Deze werkwijze heet "thread confinement".

8. MULTITHREADING WITH GUI

- Wanneer we een multithreaded programma, dat gebruik zal maken van JavaFX, wensen te implementeren, dan dienen we twee regels in acht te nemen:
  - Tijdrovende taken mogen niet door de JavaFX Application thread uitgevoerd worden. Zolang de taak niet verwerkt is, dienen de overige taken in de queue te wachten. Hierdoor reageren de gui-componenten niet meer.
  - JavaFX-componenten mogen enkel door de JavaFX Application thread toegankelijk zijn.

HoGent

8. MULTITHREADING WITH GUI

- Hierdoor moet een GUI applicatie met verwerkingen tenminste twee threads bevatten:

  1) een thread om de verwerking uit te voeren.
  2) de JavaFX Application thread.

  De moeilijkheid ligt in de communicatie tussen de twee threads.

HoGent

8. MULTITHREADING WITH GUI

- Om de gui responsive te houden dienen we tijdsconsumerende taken uit te besteden aan backgroundthreads.
- Voor JavaFX is er het javafx.concurrent package voorzien.
  →**Worker** interface met twee basisklassen: **Task** and **Service**.

HoGent

8. MULTITHREADING WITH GUI - JavaFX →kleine taken.

- Voor kleine gui-opdrachten hoeven we geen afzonderlijke thread op te starten. De klasse **Platform** voorziet een methode:
  public static void **runLater**(Runnable runnable)
- runLater zal de GUI-opdrachten laten uitvoeren als deel van de JavaFX Application thread .
- Vanuit eender welke thread kan je een Runnable opdracht in de eventqueue plaatsen. De opdrachten worden in de volgorde van plaatsing in de queue afgewerkt.

HoGent 62

8. MULTITHREADING WITH GUI
→kleine taken.

```
class UpdateButtonText implements Runnable {
    private String txt;
    private Button btn;

    public UpdateButtonText(String tekst, Button button) {
        txt=tekst; btn = button;
    }
    public void run() {
        btn.setText(txt);
    }
}

// vanuit meerdere threads kan het button label gewijzigd worden
…
Platform.runLater(
    new UpdateButtonText("Buy now", btnRedButton);


…
```

9. DAEMON THREADS

- Een thread die als ondersteuning van andere threads meedraait. De daemon thread verhindert niet dat een programma beëindigd wordt.
  - Java's garbage collector is een voorbeeld van een daemon thread.
  - We markeren een thread als daemon met de methode setDaemon(true). Dit moet wel gebeuren voordat de start methode van de thread geactiveerd wordt.
  - Methode isDaemon(): geeft true of false terug naargelang de thread al dan niet een daemon thread is.

10. Stop, suspend en resume THREADS

stop , suspend en resume zijn deprecated.

➔ Hoe stoppen : Voorzie een lus in de run methode, stop door running op false te zetten

```
while (running) {
    // do job
}
```

➔ Hoe pauzeren :

```
while (suspended)
    wait();
```


11. Nieuw Java 8 : Parallel Streams

```
// StreamStatisticsComparison.java
// Comparing performance of sequential and parallel stream operations.
import java.time.Duration; //NIEUW JAVA8
import java.time.Instant; //NIEUW JAVA8
import java.util.Arrays;
import java.util.LongSummaryStatistics;
import java.util.stream.LongStream;
import java.security.SecureRandom;

public class StreamStatisticsComparison
{
  public static void main(String[] args)
  {
    SecureRandom random = new SecureRandom();

    // create array of random long values
    long[] values = random.longs(100_000_000, 1, 1001).toArray();
```

11. Parallel Streams

```
// perform calculcations separately
Instant separateStart = Instant.now();
long count = Arrays.stream(values).count();
long sum = Arrays.stream(values).sum();
long min = Arrays.stream(values).min().getAsLong();
long max = Arrays.stream(values).max().getAsLong();
double average = Arrays.stream(values).average().getAsDouble();
Instant separateEnd = Instant.now();

// display results
System.out.println("Calculations performed separately");
System.out.printf("    count: %,d%n", count);
System.out.printf("      sum: %,d%n", sum);
System.out.printf("      min: %,d%n", min);
System.out.printf("      max: %,d%n", max);
System.out.printf("  average: %f%n", average);
System.out.printf("Total time in milliseconds: %d%n%n",
  Duration.between(separateStart, separateEnd).toMillis());
```

67

11. Parallel Streams

```
// time sum operation with sequential stream
LongStream stream1 = Arrays.stream(values);
System.out.println("Calculating statistics on sequential stream");
Instant sequentialStart = Instant.now();
LongSummaryStatistics results1 = stream1.summaryStatistics();
Instant sequentialEnd = Instant.now();

// display results
displayStatistics(results1);
System.out.printf("Total time in milliseconds: %d%n%n",
  Duration.between(sequentialStart, sequentialEnd).toMillis());
```

HoGent

68

11. Parallel Streams

```
    // time sum operation with parallel stream
    LongStream stream2 = Arrays.stream(values).parallel();
    System.out.println("Calculating statistics on parallel stream");
    Instant parallelStart = Instant.now();
    LongSummaryStatistics results2 = stream2.summaryStatistics();
    Instant parallelEnd = Instant.now();

    // display results
    displayStatistics(results1);
    System.out.printf("Total time in milliseconds: %d%n%n",
      Duration.between(parallelStart, parallelEnd).toMillis());
}
```

HoGent

11. Parallel Streams

```
// display's LongSummaryStatistics values
private static void displayStatistics(LongSummaryStatistics stats)
{
  System.out.println("Statistics");
  System.out.printf("    count: %,d%n", stats.getCount());
  System.out.printf("      sum: %,d%n", stats.getSum());
  System.out.printf("      min: %,d%n", stats.getMin());
  System.out.printf("      max: %,d%n", stats.getMax());
  System.out.printf("  average: %f%n", stats.getAverage());
}
```

HoGent

11. Parallel Streams

```
run:
Calculations performed separately
    count: 100.000.000
      sum: 50.051.544.642
      min: 1
      max: 1.000
  average: 500,515446
Total time in milliseconds: 1637

Calculating statistics on sequential stream
Statistics
    count: 100.000.000
      sum: 50.051.544.642
      min: 1
      max: 1.000
  average: 500,515446
Total time in milliseconds: 697

Calculating statistics on parallel stream
Statistics
    count: 100.000.000
      sum: 50.051.544.642
      min: 1
      max: 1.000
  average: 500,515446
Total time in milliseconds: 226
```

HoGent

71