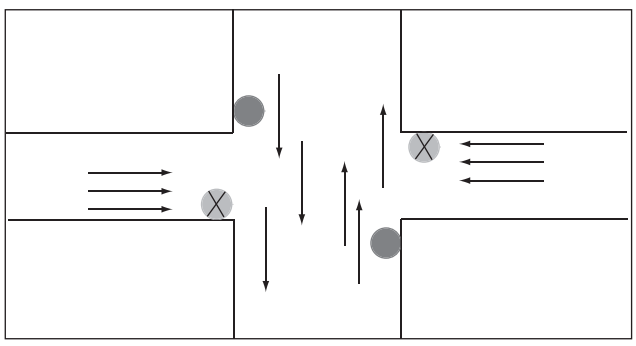
Met het verschijnen van package `java.util.concurrent` in J2SE 5.0 is een grote stap vooruit gezet om het Java-platform geschikt te maken voor het ontwikkelen van schaalbare en correcte concurrent applicaties. In dit artikel gaat Willem Koppenol in op de nieuwe synchronisatiemechanismen in J2SE 5.0. In een volgend artikel zal aandacht worden besteed aan de nieuwe thread safe collections en task management mogelijkheden.

Synchronisatiemechanismen in J2SE 5.0

Nieuwe high level bouwstenen toegevoegd

Op low level is de Java Virtual Machine (JVM) verrijkt doordat classes gebruik kunnen maken van concurrency-ondersteuning op hardware-niveau en zijn er verbeterde low level locking mogelijkheden bij gekomen. Anderzijds is een hele reeks high level bouwstenen aan de Java gereedschapskist toegevoegd die gericht zijn op het ontwikkelen van concurrent applicaties. Onder die bouwstenen bevinden zich constructies met soms exotische namen als semaphore, mutex, cyclicbarrier, executor, future, callback, countdownlatch en exchanger.

THREADS EN SYNCHRONISATIE Binnen processen van vrijwel ieder besturingssysteem kunnen tegenwoordig verschillende threads naast elkaar, *concurrent*, een taak uitvoeren en daarbij onafhankelijk van elkaar processortijd toebedeeld krijgen. Aangezien threads de adresruimte van het proces delen, kunnen ze dezelfde objecten benaderen en alloceren ze objecten vanuit hetzelfde dynamische geheugen. Dit betekent weliswaar dat threads op een eenvoudige manier informatie met elkaar kunnen delen, maar ook dat de toegang tot de gedeelde data moet worden gecoördineerd. Zoals we in het verkeer stoplichten gebruiken om de toegang tot een kruispunt te regelen, zo gebruiken we bij threads synchronisatiemechanismen om de toegang tot gedeelde data te regelen.

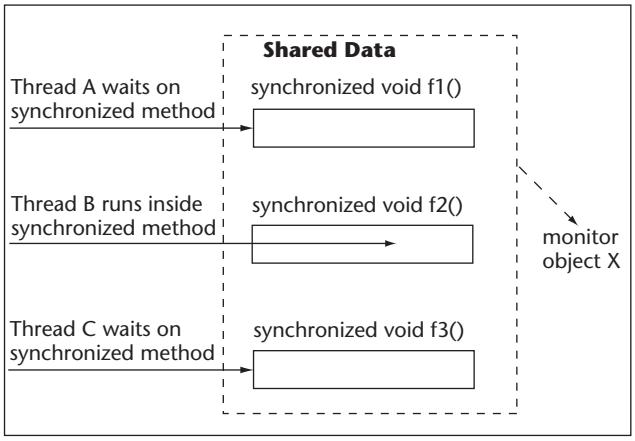


FIGUUR 1. Synchronisatie met stoplichten

REVIEW SYNCHRONIZED Eén van de innovaties van Java was, dat een cross-platform threading model direct in de taalspecificatie was opgenomen. Vanaf het begin kent Java de `Thread` class voor het creëren, starten en manipuleren van threads. Voor het coördineren van threads zijn sinds jaar en dag een aantal synchronisatieprimitieven aanwezig. Het `synchronized` keyword wordt gebruikt om methoden van een class of code blocks ertegen te beschermen dat verschillende threads zich tegelijkertijd in deze code bevinden. Het `synchronized` keyword levert een ingebouwde lockingfaciliteit. Als een thread het ingebouwde lock (de monitor) verkrijgt, zal een andere thread in een wachtstand komen als deze probeert hetzelfde lock te krijgen. De geblokkeerde thread wordt pas weer wakker als de oorspronkelijke thread de monitor vrijgeeft.

De oorspronkelijke Java-synchronisatieprimitieven missen de nodige functionaliteit. Zeker juniorontwikkelaars hebben er moeite mee om ze correct te gebruiken terwijl het gemakkelijk is om ze verkeerd te gebruiken. Een verkeerd gebruik kan bovendien de oorzaak zijn van een slechte performance. In feite zijn de constructies te low level voor de meeste applicaties. Veel ontwikkelaars bedachten daarom, bij gebrek aan een standaard, hun eigen high level constructies en vonden daarbij steeds weer het wiel opnieuw uit. Met de introductie van J2SE 5.0 hoeft dat niet meer.

LOCK Alternatieve low level synchronisatie mechanismen in J2SE 5.0 zijn implementaties van het Lock interface. Voor simpele bescherming tegen gelijktijdige



FIGUUR 2. Locking met synchronized

benadering van gedeelde data volstaat synchronisatie met het `synchronized` keyword. Deze vorm van synchronisatie heeft voor geavanceerde applicaties echter wat functionele beperkingen, zoals geen ondersteuning voor time-outs en interrupts. De implementaties van het Lock interface bieden deze mogelijkheden wel. Het Lock interface ziet er als volgt uit:

```
interface Lock {
void lock();
void lockInterruptibly() throws
InterruptedException;
boolean tryLock();
boolean tryLock(long time, TimeUnit unit)
throws InterruptedException;
void unlock();
Condition newCondition() throws UnsupportedOp
erationException;
}
```

Het standaard-synchronisatiemechanisme vereist dat locks worden vrijgegeven in hetzelfde stack frame als ze zijn verkregen. Meestal is dit goed en het komt ook mooi overeen met exception handling. In sommige gevallen heb je echter behoefte aan een lock mechanisme dat niet op code blocks is gebaseerd. Implementaties van de Lock interface voorzien hierin.

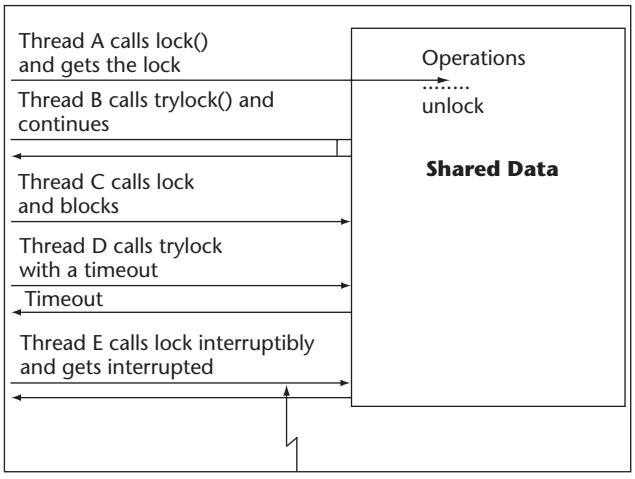
REENTRANTLOCK `ReentrantLock` is een implementatie van Lock met hetzelfde basisgedrag als het ingebouwde monitorlock. Met een `ReentrantLock` heb je de mogelijkheid te testen of het lock beschikbaar is (of om iets anders te doen, wanneer het lock niet beschikbaar is), een tijdslimiet te zetten op het wachten op een lock, en een thread die staat te wachten op een lock weer wakker te maken. Als een thread om een lock vraagt dat hij al heeft, mag hij doorgaan. De naam `ReentrantLock` is gekozen omdat er een teller met het lock is geassocieerd die bijhoudt hoe vaak een thread hetzelfde lock verkrijgt. Het lock wordt pas weer vrijgegeven als de teller nul wordt. Overigens komt dit gedrag overeen met het standaard-synchronisatiemechanisme.

De performance van `ReentrantLock` is onder zware belasting, wanneer veel threads om het lock vragen, beter dan `synchronized`. De JVM spendeert bij een ReentrantLock minder tijd aan het schedulen van threads en meer aan het uitvoeren ervan dan bij het gebruik van `synchronized`. Hoewel `ReentrantLock` dus veel voordelen heeft ten opzichte van `synchronized` is er één belangrijk nadeel: het is mogelijk om te vergeten het lock vrij te geven. Bij het gebruik van `synchronized` zorgt de JVM automatisch voor de vrijgave van het lock en heb je daar als ontwikkelaar geen omkijken naar. Een goed codepatroon voor het gebruik van het lock is het volgende:

```
Lock lock = new ReentrantLock();
...
lock.lock();
try {
// perform operations protected by lock
}
catch(Exception ex) {
      // restore invariants
}
finally {
     lock.unlock();
}
```

De gevolgen van het niet vrijgeven van het lock kunnen desastreus zijn, en daarom is het aan te bevelen `synchronized` te blijven gebruiken, tenzij je de extra functionaliteit en schaalbaarheid van `ReentrantLock` echt nodig hebt.

READWRITELOCK Het lockmechanisme van het `ReentrantLock` is vrij eenvoudig : slechts één thread kan het lock verkrijgen, andere threads moeten wachten tot het weer beschikbaar komt. Soms is het echter



FIGUUR 3. Synchronisatie met ReentrantLock

wenselijk een iets gecompliceerder lockmechanisme te gebruiken. Als datastructuren vaker worden gelezen dan gewijzigd, ligt een `ReadWriteLock` meer voor de hand. Dit lockingmechanisme staat meer gelijktijdige lezers toe, maar ook exclusieve locking door één schrijver. In het geval van verschillende parallelle lezers is er dan meer doorvoer, terwijl de veiligheid van exclusieve toegang door één schrijver ook mogelijk is. `ReadWriteLock` is een interface dat wordt geïmplementeerd door de `ReentrantReadWriteLock` class.

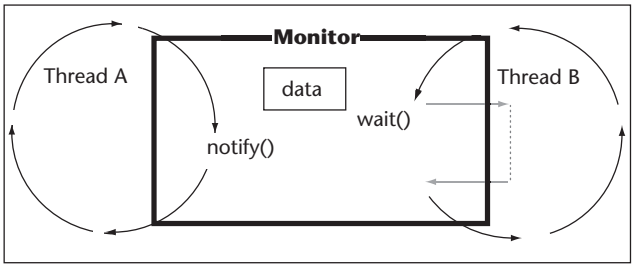
EERLIJKE LOCKS De constructors van implementaties van de `Lock` interface hebben een boolean parameter waarmee je kunt aangeven of je een eerlijk of oneerlijk `Lock` wilt:

```
ReentrantLock(boolean fair)
```

Bij een eerlijk lock krijgen de threads toegang in dezelfde volgorde waarin ze het lock hebben aangevraagd. Bij een oneerlijk lock kan een thread die later om het lock heeft gevraagd het mogelijk toch eerder krijgen. Je zou het misschien niet verwachten maar het standaard-synchronisatiemechanisme met `synchronized` is, in tegenstelling tot wat veel ontwikkelaars denken, niet eerlijk. Hier merk je overigens weinig van, want de JVM zorgt er wel voor dat alle wachters op een lock uiteindelijk aan de beurt komen, zodat van *starvation* geen sprake is. Oneerlijke locking is ook de default bij `Lock` implementaties. De reden dat eerlijke locks niet de standaard zijn, komt doordat het boekhoudwerk dat ermee verbonden is performance kost. Maar als je voor de correcte werking van een algoritme beslist wilt dat de threads een lock verkrijgen in dezelfde volgorde als ze het hebben aangevraagd, kun je dit nu in J2SE 5.0 door een eerlijk lock garanderen.

CONDITION VARIABELEN Van oudsher kent Java de `wait, notify` en `notifyAll` functies van de `Object` class die het mogelijk maken dat threads door interthread communicatie met behulp van conditiesynchronisatie hun werk met elkaar coördineren. Aan ieder monitor lock is een conditie variabele verbonden waarop je kunt wachten met `wait` en die wordt gezet door de `notify` methoden. De interactie tussen notificatie en locking is dat je het lock op het object moet bezitten om een `wait` of `notify` op dat object te kunnen doen.

Conditiesynchronisatie biedt geavanceerde mogelijkheden, maar is subtiel in het gebruik en wordt niet vaak door ontwikkelaars toegepast. Met name met `wait` en `notify` geven nogal eens aanleiding tot een deadlock in applicaties. Door de vele nieuwe synchronisatiemogelijkheden in `java.util.concurrent` wordt de



FIGUUR 4. Conditie synchronisatie met wait and notify

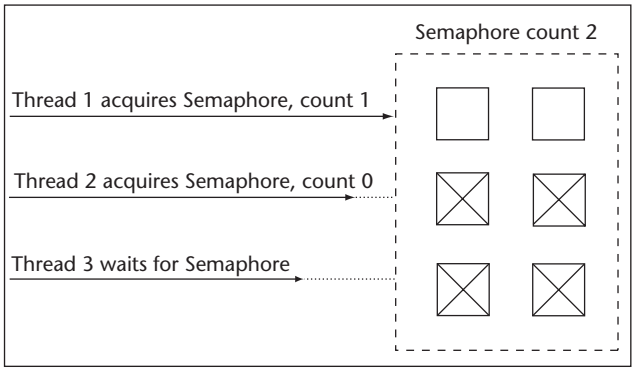
noodzaak om conditiesynchronisatie te gebruiken nog minder.

Nieuw in J2SE 5.0 is de toevoeging van een generalisatie op conditiesynchronisatie in de Condition interface. Een Lock object fungeert als factory voor conditievariabelen die horen bij dat lock en er kan door de aanroep van `newCondition` meer dan één conditievariabele aan een lock worden verbonden. De mogelijkheid van meer conditievariabelen versimpelt veel concurrent algoritmen. De methoden uit de Condition interface heten `await`, `signal` en `signalAll`. Ze kunnen de corresponderende methoden uit `Object` immers niet herdefiniëren. Functioneel verschillen ze niet van `wait` en `notify`.

ATOMIC VARIABLE De meest significante vernieuwingen op het gebied van concurrency, zijn mogelijk de Atomic Variable classes die in het package `java.util.concurrent.atomic` aan J2SE 5.0 zijn toegevoegd. Met deze classes wordt het voor het eerst mogelijk via hardware-ondersteuning, zonder native code schaalbare, wacht-vrije en lock-vrije algoritmen in Java te ontwikkelen. Dergelijke algoritmen worden veel gebruikt op het niveau van het besturingssysteem en de JVM en maken een grotere graad van parallellisme mogelijk dan hun op locking gebaseerde alternatieven. De meeste CPU's hebben tegenwoordig ondersteuning voor atomaire read-modify-write operaties, zoals de compare-and-swap (CAS) operatie en de load-linked/store-conditional (LL/SC) operatie. De snelste constructies die op de hardware aanwezig zijn, worden door de Atomic Variables gebruikt.

De meeste ontwikkelaars zullen deze classes zoals `AtomicInteger, AtomicLong`, et cetera. waarschijnlijk niet direct nodig hebben. Het directe gebruik is waarschijnlijk voorbehouden aan enkele concurrency experts. Toch staan ze aan de basis van *low level*-verbeteringen waardoor de classes uit `java.util.concurrent` veel beter schaalbaar zijn dan hun voorlopers. Bijna alle classes in `java.util.concurrent` maken immers intern gebruik van een `ReentrantLock`, dat op zichzelf bovenop de Atomic Variable classes is gebouwd.

Atomaire variabelen worden het meest gebruikt om velden die veel worden benaderd en gewijzigd door



FIGUUR 5. Semaphoor zet limiet op aantal gebruikers van een resource.

multiple threads, om een efficiënte manier atomair te wijzigen. In die zin zijn ze ook geschikt als tellers en voor de generatie van volgnummers.

SEMAPHOOR Een nieuw high level synchronisatie mechanisme in J2SE 5 is de semaphoor. Hiermee kan het aantal gebruikers van een resource worden gelimiteerd. Dit synchronisatiemechanisme werd ooit voorgesteld door de Nederlandse informaticus prof. Dijkstra en wordt ook wel counting semaphoor genoemd. Waar een lock of een `synchronized` block het aantal threads dat een gedeelde datastructuur benadert beperkt tot één, kan je met een semaphoor een maximaal aantal gebruikers - het aantal permits - specificeren. De semaphoor gebruikt daartoe een interne teller. Initieel staat deze teller op het maximaal aantal gebruikers van de resource. Bij iedere nieuwe gebruiker wordt de teller verlaagd. Als de teller nul wordt, krijgt een nieuwe gebruiker geen toegang. Een voorbeeld van een resource die beschermd wordt door een semaphoor is een parkeergarage. Het maximaal aantal gebruikers van deze resource, is het aantal plaatsen in de parkeergarage. Zodra er een auto parkeert wordt de teller verlaagd. Als alle vrije plaatsen bezet zijn, verschijnt het bordje vol aan de ingang en zul je moeten wachten tot een auto de parkeergarage verlaat en daarmee de semaphoor weer vrij geeft.

De `Semaphore` class in J2SE 5.0 implementeert een `semaphoor`. Aan de constructor van Semaphore geven we het maximaal aantal gebruikers van de resource mee, en of we een eerlijke of oneerlijke semaphoor willen. Een eerlijke semaphoor hanteert een FIFO (First In First Out) queue om de volgende gebruiker van een resource te selecteren. Met `acquire` verkrijg je toegang tot de resource die de semaphoor beschermt. Deze methode blokkeert tot een permit beschikbaar is, of totdat de wachtende thread wordt geïnterrumpeerd. `tryAcquire` blokkeert niet, maar retourneert 'true' wanneer het gevraagde aantal permits beschikbaar is, en in andere gevallen 'false'. Ook is het mogelijk een maximumwachttijd te specificeren. De semaphoor wordt weer vrijgegeven met `release()`. Voorbeeldcode waarin een Semaphore wordt gebruikt is:

```
if (semaphore.tryAcquire()) {
   try {
         ticket = Garage.park();
   } finally {
         semaphore.release();
   }
} else {
   Parking.search();
}
```

> Op low level is de JVM verrijkt, doordat classes gebruik kunnen maken van concurrency-ondersteuning

MUTEX Een speciale semaphoor is de mutex, de mutual exclusion semaphoor of ook wel genoemd binary semaphoor. Een mutex kan vergeleken worden met het stokje in een estafetteloop en is simpelweg een counting semaphoor met maximaal één gebruiker. Met een mutex kun je de exclusieve toegang tot een gedeelde resource regelen. Mutexes hebben veel gemeen met locks, maar hebben één eigenschap die locks over het algemeen niet hebben. Ze kunnen worden vrijgegeven door een andere thread dan de thread die de mutex bezit. Deze nuttige eigenschap komt van pas bij het oplossen van eventuele deadlock situaties.

CYCLICBARRIER Een ander high level-synchronisatiemechanisme dat is toegevoegd aan J2SE 5.0 is de barrier. Een barrier biedt een gemeenschappelijk wachtpunt waar een reeks threads op elkaar kan wachten. De `CyclicBarrier` class implementeert deze functionaliteit en wordt zo genoemd omdat - nadat een eerste reeks threads het wachtpunt is gepasseerd - een tweede reeks threads het wachtpunt kan gebruiken zonder eerst een nieuwe barrier te creëren. Je kunt een barrier creëren met alleen een aantal threads of met een aantal threads en een actie. Een actie is een `Runnable` taak die moet worden uitgevoerd als alle threads gearriveerd zijn. Nadat een thread zijn individuele taak heeft uitgevoerd, wacht de thread bij de barrier door de `await()` functie aan te roepen. Deze blokkeert de thread totdat alle threads bij het barrier punt zijn aangeland.

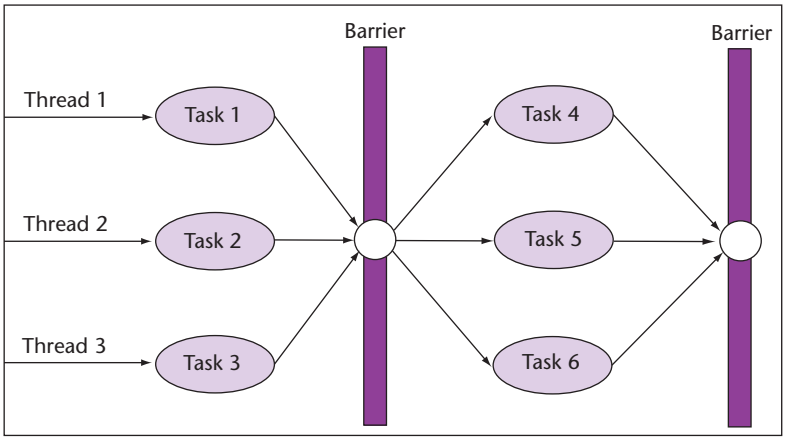
Een bekend voorbeeld om barriers te demonstreren is het opdelen van een grote taak in een aantal subtaken, zodanig dat iedere subtaak zelfstandig kan worden uitgevoerd. Een voorbeeld daarvan is het opdelen van een

grote sommatie van getallen in een aantal delen, die ieder in hun eigen thread op een afzonderlijke processor worden uitgevoerd. Als threads klaar zijn met hun deelberekening, stoppen ze bij het wachtpunt. Als alle threads klaar zijn, kan de totale som uit de som der delen worden uitgerekend. De barrier voorkomt de uiteindelijke sommatie tot alle delen bekend zijn. Voorbeeldcode waarin een `CyclicBarrier` wordt gebruikt is:

```
public class SumTotal{
   private static int data = {
       {1, 2, 3, 4, 5, 6, 7, 8, 9, 10}
   };
   ......
   private static class Summer extends Thread
{
     public void run() {
        ..... calculate a partial sum
        partialsums[i] = sum;
        barrier.await();    // wait for
others
     }
   }
   public static void main(String args[]) {
       ....... initialize
       Runnable joiner = new Runnable() {
       public void run() {
           for (int i=0; i<parts; i++) {
             totalsum += partialsums[i];
           }
         }
       };
       CyclicBarrier barrier = new
CyclicBarrier(parts, joiner);
       for (int i=0; i<parts; i++) {
         new Summer(barrier, i).start();
       }
    }
}
```

CountDownLatch Nog een high level-synchronisatiemechanisme dat is toegevoegd aan J2SE 5.0 is de latch. De latch wordt net als de barrier gebruikt om een groep threads te coördineren. Kenmerkend voor een latch is dat hij hoogstens één keer wordt gezet. Zolang de latch door een coördinerende thread nog niet is gezet, zullen de threads erop moeten wachten, maar nadat hij eenmaal is gezet kunnen de threads altijd door.

De `CountDownLatch` class implementeert een latch die wordt geïnitialiseerd met een teller. Deze teller representeert het aantal threads dat bij de latch betrokken is. De teller geeft aan hoe vaak `countDown` moet worden aangeroepen alvorens alle threads die in await op de latch stonden te wachten mogen doorgaan. In tegenstelling tot de `CyclicBarrier` is de `CountDownLatch` niet herbruikbaar. Terwijl bij een



Figuur 6. CyclicBarriers fungeren als wachtpunt voor een reeks threads.

`CyclicBarrier` alle threads aankomen en wachten bij een wachtpunt door een `await` aanroep, wordt bij een `CountDownLatch` de aankomst- en wachtfunctionaliteit gescheiden. Iedere thread kan `countDown` aanroepen en deze aanroep blokkeert niet. Threads die in `await` op de latch wachten, mogen door als de latchteller nul wordt en volgende aanroepen van await retourneren meteen.

Een `CountdownLatch` is nuttig als we een probleem kunnen opdelen in een aantal subtaken die door threads kunnen worden uitgevoerd. Als de worker threads klaar zijn met hun subtaak, verlagen ze de teller, terwijl een coördinerende thread op de latch wacht alvorens een volgende set subtaken te starten. Een `CountdownLatch` object met een tellerwaarde één is bij uitstek geschikt bij initialisaties, waarbij worker threads moeten wachten tot alle benodigde resources zijn geïnitialiseerd. Een voorbeeld is de implementatie van multi player games waarbij het spel pas mag beginnen als alle deelnemers zich hebben aangemeld. Het volgende codevoorbeeld gebruikt twee `CountdownLatches`. De eerste fungeert als startpunt na initialisatie. De tweede wacht op de beëindiging van de taken van de worker threads:

```
class Driver { // ...
   void main() throws InterruptedException {
     CountDownLatch startSignal = new
CountDownLatch(1);
     CountDownLatch doneSignal = new
CountDownLatch(N);
     for (int i = 0; i < N; ++i) // create
and start threads
        new Thread(new Worker(startSignal,
doneSignal)).start();
     doSomethingElse(); // don't let them run
yet
     startSignal.countDown(); // let all
threads proceed
     doSomethingElse();
```

```
      doneSignal.await(); // wait for all to
finish
    }
}
class Worker implements Runnable {
    private final CountDownLatch startSignal;
    private final CountDownLatch doneSignal;
    Worker(CountDownLatch startSignal,
CountDownLatch doneSignal) {
      this.startSignal = startSignal;
      this.doneSignal = doneSignal;
    }
    public void run() {
      try {
        startSignal.await();
        doWork();
        doneSignal.countDown();
      } catch (InterruptedException ex) {} //
return;
    }
}
```

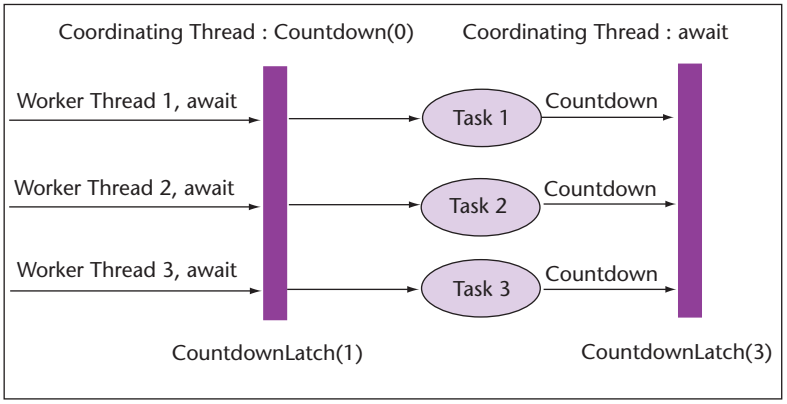
EXCHANGER<V> De laatste in de reeks van high level-synchronisatiemechanismen dat met J2SE 5.0 is geïntroduceerd is de `Exchanger`. Zoals de naam al aangeeft, biedt een Exchanger een manier op objecten tussen threads uit te wisselen. De `Exchanger` is een alternatief voor op IO streams gebaseerde inter-thread communicatie. De `Exchanger` gebruikt generics. In de constructor geef je het type object aan dat door de `Exchanger` wordt uitgewisseld:

```
Exchanger<List<Integer>> exchanger = new Exch
anger<List<Integer>>();
```

Met een aanroep van exchange wordt het object in beide richtingen uitgewisseld:

```
V exchange(V x)
```

Een `Exchanger` kan goed gebruikt worden bij de oplossing van een producer-consumer probleem waarbij één thread data produceert en een andere thread deze consumeert. Als de buffer die de producer gebruikt vol is zal de producer op de consumer wachten. Als de buffer die de consumer gebruikt leeg is, zal de consumer op de producer wachten. Als beide staan te wachten, wisselen de threads de buffer uit. In de onderstaande voorbeeldcode vindt de data uitwisseling tussen de producer thread (`FillingLoop class`) en de consumer thread (`EmptyingLoop class`) plaats zodra beide in `exchange` op elkaar staan te wachten.

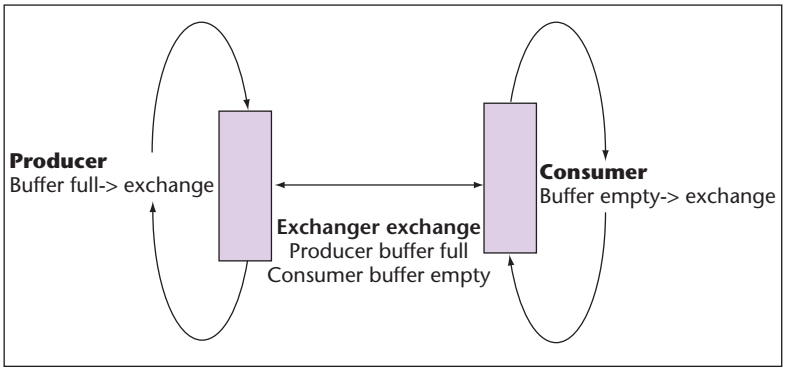


FIGUUR 7. CountDownLatches voor initialisatie en als coördinatiepunt.

```
class FillAndEmpty {
    Exchanger<DataBuffer> exchanger = new
Exchanger<DataBuffer>();
    DataBuffer initialEmptyBuffer = new
DataBuffer();
    DataBuffer initialFullBuffer = new
DataBuffer();

    class FillingLoop implements Runnable {
      public void run() {
        DataBuffer currentBuffer = initia-
lEmptyBuffer;
        try {
          while (currentBuffer != null) {
            addToBuffer(currentBuffer);
            if (currentBuffer.full())
              currentBuffer = exchanger.
exchange(currentBuffer);
          }
        } catch (InterruptedException ex) {
... handle ... }
      }
    }
    class EmptyingLoop implements Runnable {
      public void run() {
        DataBuffer currentBuffer = initial-
FullBuffer;
        try {
          while (currentBuffer != null) {
            takeFromBuffer(currentBuffer);
            if (currentBuffer.empty())
              currentBuffer = exchanger.
exchange(currentBuffer);
          }
        } catch (InterruptedException ex) {
... handle ...}
      }
    }
    void start() {
    new Thread(new FillingLoop()).start();
    new Thread(new EmptyingLoop()).start();
  }
}
```

SLOTWOORD De release van J2SE 5.0 is op het gebied van synchronisatiemogelijkheden een grote stap voorwaarts. In de JVM en de class library's zijn er belangrijke verbeteringen aangebracht, die gebruik maken van de door veel processoren geboden hardware-ondersteuning voor synchronisatie. De nieuwe low level locking classes maken intern van deze harware-ondersteuning gebruik en bieden behalve meer functionaliteit ook een betere performance dan eerder het geval was. Dit wil nog niet zeggen het aloude synchronized keyword daarmee een *deprecated* verschijnsel in Java is geworden. De nieuwe locking classes hebben weliswaar veel meer functionaliteit maar verleggen het lockmanagement naar de ontwikkelaar. Ze zijn daarom niet onder alle omstandigheden een beter alternatief dan het gebruik van het `synchronized` keyword. Daarnaast zijn er een reeks van high level-synchronisatiemechanismen zoals semaphoren, barriers en latches nu standaard in de Java-library's aanwezig. Ontwikkelaars hoeven deze wielen niet meer opnieuw uit te vinden en dit komt de productiviteit en kwaliteit van de code ten goede.



FIGUUR 8. Exchanger bij een producer-consumer probleem

drs. Willem Koppenol (e-mail: wkoppenol@twice.nl) is senior trainer en productspecialist software development bij Twice IT Training.

PATCHES Patches PATCHES Patches PATCHES Patches PATCHES

Artikelen over onderwerpen als software-ontwikkeling, Java, UML, eXteme Programming en nog veel meer vindt u in het Online Archief van Array Publications. Vaktijdschriften als Software Release, Java Magazine, Database Magazine en ons Oracle vakblad Optimize hebben hun artikelenarchief online gezet. Dankzij de heldere zoekstructuur vindt u snel wat u zoekt op www.release.nl.

Software AG introduceert EII 2.1

Vorige maand lanceerde Software AG haar Enterprise Information Integrator (EII) versie 2.1, een belangrijke bouwsteen voor het implementeren van een service oriented architecture. EII versie 2.1 is het eerste wereldwijde beschikbare informatie integratieproduct dat gebruik maakt van semantische webtechnologie. Door dynamisch combineren van betekenis en context van businessdata met de voorwaarden voor gebruik, stelt EII business managers in staat om sneller beslissingen te nemen gebaseerd op real-time informatie.

Enterprise Information Integrator versie 2.1 is ontworpen om organisaties te helpen de vraag naar een snelle en flexibele data integratie in een service oriented architecture te realiseren. Enterprise Information Integrator creëert real-time "views" van de gewenste business data die meestal verspreid over de organisatie in verschillende systemen zit.

Omdat deze views de actuele situatie weerspiegelen (inclusief context, betekenis en voorwaarden voor gebruik) gebruikt de beslisser altijd de meest recente en meest relevante informatie. Omdat Enterprise Information Integrator context toevoegt aan data en deze opneemt in een real-time informatiemodel, is het een aanvulling op de traditionele data-integratie methodes, die zich concentreren op analyse van historische data.

Versie 2.1 van de Enterprise Information Integrator ondersteunt W3C standaarden, inclusief de Web Ontology Language (OWL) en het Resource Description Framework (RDF), bevat een op Eclipse gebaseerde grafische design studio, waarmee gemakkelijker informatiemodellen en "single views" gemaakt kunnen worden, en heeft uitgebreide security ondersteuning voor authenticatie op alle standaard security systemen - inclusief die van mainframes.

Met Enterprise Information Integrator kunnen informatie-analisten de complexe relatie tussen data-elementen modelleren. Het inzetten van informatiemodellen is veel sneller dan het handmatig schrijven van code. Ook zijn "views" gemakkelijker te wijzigen en te hergebruiken als de marktomstandigheden wijzigen. Enterprise Information Integrator versie 2.1 is een onderdeel van Software AG's XML Business Integration portfolio en beschikbaar op Windows XP, Windows Server 2003 en Sun Solaris.