Core

De meeste Java-applicaties zijn multi-threaded. Servlets op webapplicaties kunnen door meer gebruikers tegelijkertijd aangeroepen worden. Java-applicaties met een grafische interface starten automatisch threads op om de gebruikersacties af te handelen. Elke programmeur moet daarom weten hoe je met concurrency moet omgaan. Helaas bestaan er veel misverstanden over concurrency. Boeken over concurrency leggen de problemen uit vanuit de taalspecificatie. Dat is echter niet het hele verhaal. De specificatie van de Java Virtual Machine bevat essentiële informatie die concurrency een heel stuk begrijpelijker maakt.

Concurrency in Java

Wat wel en niet kan

Peter van den Berkmortel
Senior Technologie Specialist, Sogeti INartkel

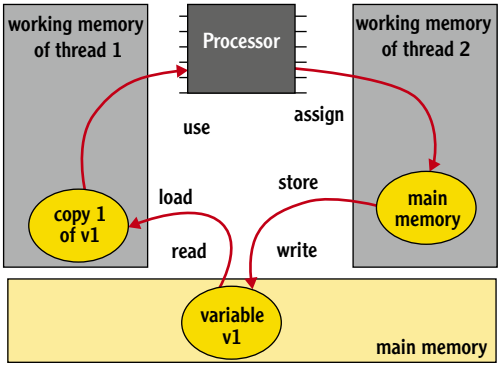
In Java 5 is de manier waarop met het gemeenschappelijk geheugen wordt omgegaan, helemaal opnieuw geschreven. Dit heeft onder andere geleid tot de toevoeging van het `java.util.concurrent` package. Hoewel dit een erg belangrijke toevoeging in Java 5 is, kan het de standaard concurrency-primitieven (`synchronized`, `wait()`, `notify()`, enzovoort) niet (volledig) vervangen. Het is dus zaak dat elke Java-programmeur weet hoe je met normale concurrency omgaat. Let op: dit artikel beschrijft hoe een en ander werkt in Java 5. In eerdere versies van Java gaat een aantal van de hier beschreven zaken niet op.

Java Memory Model

Het geheugen-model van Java (JMM) is de belangrijkste factor in het concurrency-vraagstuk. Een geheugenmodel beschrijft, gegeven een programma en een executievolgorde van dat programma, of die volgorde een mogelijke (correcte) executie is. Het geheugenmodel beschrijft dus het mogelijke gedrag van een programma. Het JMM wordt met name interessant als het om threads gaat. Een thread is in Java een object dat door de Java Virtual Machine (JVM) gescheduled wordt. Elke thread krijgt zo de tijd om een stukje processing te doen. Als jouw code met lokale variabelen werkt, dan heeft elke thread daar zijn eigen kopie van. Dat is ook precies wat een programmeur verwacht. Echter, attributen van een class moeten gedeeld kunnen worden tussen threads. Je verwacht dat een wijziging op een attribuut door de ene thread gezien wordt door een andere thread. Om dit mogelijk te maken heeft Java een gemeenschappelijk geheugen, het hoofdgeheugen, waarin attributen een plekje hebben. Wat veel programmeurs niet weten, is dat elke thread ook nog een eigen werkgeheugen heeft! In de virtual machine-specificatie[a] van Java 5 is dit te lezen in het hoofdstuk over threads – uiteraard het laatste hoofdstuk. In deze specificatie wordt uitgelegd dat elke variabele eerst van het hoofdgeheugen naar het werkgeheugen van de thread gekopieerd wordt, voordat deze gebruikt kan worden. Dit betekent dus dat er meerdere kopieën van de waarde van een variabele zijn – de kopie in het hoofdgeheugen en de kopieën in de werkgeheugens van threads!

De specificatie van de virtual machine legt uit hoe een implementatie van een JVM moet omgaan met het geheugen en met kopieën van variabelen. Dit werkt als volgt. Stel dat we een variabele (hoofdkopie) in het hoofdgeheugen hebben. Om deze te kunnen gebruiken, moet deze eerst naar het werkgeheugen gekopieerd worden. Dit gebeurt in twee stappen: eerst verstuurt het hoofdgeheugen de waarde naar de thread (dit is een zogenaamde '*read*'-actie, uitgevoerd door het hoofdgeheugen). Vervolgens ontvangt de thread de waarde en plaatst deze in het werkgeheugen (dit is een '*load*'-actie, uitgevoerd door de thread). Het is belangrijk om te constateren dat dit twee acties zijn: tussen twee acties in kunnen door andere threads andere acties uitgevoerd worden. Een andere thread zou bijvoorbeeld de waarde in het hoofdgeheugen kunnen wijzigen. Als je jezelf afvraagt waar de waarde van de variabele is tussen een *read*- en een *load*-actie in; dit is niet

beschreven. Blijkbaar wordt dat aan de implementatie van de JVM overgelaten. Als de thread nu een berekening wil uitvoeren, is nog een derde actie nodig om de waarde van de variabele in een register van de processor te laden (dit is een '*use*'-actie).



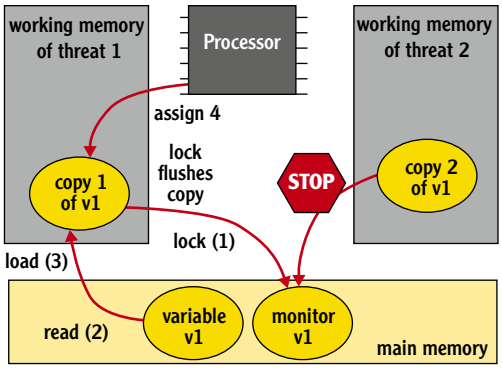
Afbeelding 1. Omgaan met het geheugen en met kopieën van variabelen

Als de waarde van een variabele gewijzigd wordt, dan moet de weg terug afgelegd worden: ook dit zijn weer drie acties. Eerst wordt de waarde van het register van de processor naar het werkgeheugen van de thread gekopieerd (in een '*assign*'-actie). Vervolgens wordt de waarde door de thread naar het hoofdgeheugen verstuurd (met een '*store*'-actie). Tot slot ontvangt het hoofdgeheugen de waarde en slaat 'm op (met een '*write*'-actie).

Twee threads kunnen afzonderlijk van elkaar *read/load/use-* en *assign/store/write*-acties uitvoeren. Het JMM garandeert dat er nooit gelijktijdig *read/write-acties* op de hoofdkopie van een variabele plaatsvinden. Het garandeert niet dat de ene thread een waarde wijzigt en dat die wijziging vervolgens door een andere thread wordt overschreven! Dit kan gebeuren als in slecht gesynchroniseerde programma's compileroptimalisaties worden uitgevoerd. Hoe de compileroptimalisaties werken, zien we straks. Laten we eerst kijken hoe synchronisatie werkt.

Elk Java-object heeft een (lock) monitor geassocieerd met het object. Als een thread een lock wil krijgen op het object, dan moet de thread nauw samenwerken met het hoofdgeheugen om de lock op de monitor te krijgen. Locken is dus één enkele actie die de thread en het hoofdgeheugen samen uitvoeren. Als een andere thread ook een lock op hetzelfde object probeert te krijgen, dan moet de thread wachten tot de lock is vrijgegeven. Een thread kan meerdere keren dezelfde monitor locken. De thread moet de monitor dan ook even zoveel keren vrijgeven, voordat de lock is opgeheven.

Tijdens een '*lock*'-actie gebeurt er nog iets extra's met het werkgeheugen van de thread. Alle werkkopieën van attributen worden als het ware geflusht uit het werkgeheugen, zodat deze opnieuw uit het hoofdgeheugen gelezen moeten worden, voordat ze gebruikt kunnen worden! Het JMM is wel wat vaag over dit flushen. In de specificatie staat dat het lijkt of dit gebeurt. Het is dus aan implementaties van de JVM om te bepalen hoe dit geïmplementeerd wordt.



Afbeelding 2.

Bij het vrijgeven van een lock, moeten gewijzigde kopieën van variabelen eerst naar het hoofdgeheugen verstuurd worden, voordat de unlock mag worden uitgevoerd. Een '*unlock*'-actie gebeurt dan weer in een nauwe samenwerking tussen de thread en het hoofdgeheugen.

Als een lock is vrijgegeven, kan een wachtende thread de lock verkrijgen. Werkkopieën van variabelen worden geflusht en dit zorgt er voor dat de wijzigingen die de eerste thread heeft uitgevoerd, zichtbaar zijn voor de tweede thread. Merk op dat als er meerdere variabelen zijn gewijzigd, het dus *niet uitmaakt in welke volgorde* dit is gebeurd. De wijzigingen zijn allemaal voor de unlock naar het hoofdgeheugen geschreven. Daarmee zijn ze allemaal zichtbaar voor de thread die vervolgens de lock krijgt.

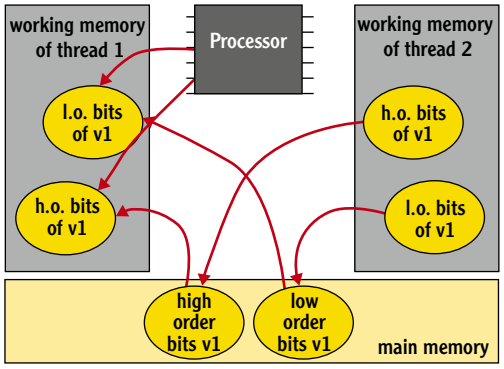
De specificaties van de JVM bevatten nog meer informatie. Het is niet nodig om een uitputtende lijst hiervan te geven. De belangrijkste onderdelen zijn de volgende.

Ten eerste het `final` sleutelwoord. Als een variabele `final` is, garandeert het JMM dat de variabele een waarde heeft als de constructor van het omvattende object eindigt. Met andere woorden, als je een referentie naar een object hebt (dan is de constructor dus afgelopen), dan hebben de `final`-attributen van dat object een waarde. Bovendien zal deze waarde ook niet meer wijzigen. Dit laat een aantal interessante openingen over voor compileroptimalisaties. Daarover straks meer.

Het tweede onderdeel is het `volatile` sleutelwoord. Dit sleutelwoord is niet nieuw in Java 5, maar het had in eerdere versies van Java nog geen

betekenis. Met het herschrijven van het JMM is dat veranderd. Als een variabele `volatile` is, dan zijn de *load*- en *use*-acties en de assign- en store-acties sterk aan elkaar gekoppeld. De twee acties moeten altijd direct na elkaar uitgevoerd worden, zonder andere tussenliggende acties in de thread en ook in die volgorde! Dit zijn dus restricties voor compiler optimalisaties.

Het laatste onderdeel is het niet atomair afhandelen van 64-bit variabelen op 32-bit hardware-architecturen. Normaal gesproken is het lezen en schrijven van en naar attributen atomair. Dit wil zeggen dat een *read*- of *write*-actie één actie is. Echter, 64-bit variabelen (`longs` en `doubles`) hebben in het fysieke geheugen in 32-bit hardwarearchitecturen twee acties nodig om ze te kunnen lezen. De ontwerpers van Java hadden er voor kunnen kiezen om dit probleem door de implementatie van de JVM te laten afhandelen. Er is echter voor gekozen om het probleem bij de programmeur neer te leggen: het is dus aan jullie om dit probleem op te lossen!



Afbeelding 3.

Wat er gebeurt, is dat een 64-bit variabele eigenlijk twee plekjes in het hoofdgeheugen inneemt. Er zijn dan dus twee acties nodig om zo'n variabele te lezen. Tussen deze acties in, kunnen in andere threads andere acties worden uitgevoerd. Het gevolg is dat tussen het lezen of schrijven van de hoge order bits en de lage order bits een andere thread de waarde kan aanpassen. De hoge order bits passen dan niet meer bij de lage order bits. Er ontstaan op die manier dus spookwaarden, die schijnbaar door geen enkele thread zijn weggeschreven. De oplossing voor dit probleem is het gebruik van normale synchronisatie, de variabele `volatile` maken (het JMM garandeert dat dit een oplossing is) of de nieuwe `AtomicLong` class gebruiken. Er is geen `AtomicDouble` class. De `Atomic*` classes hebben overigens nog meer interessante mogelijkheden, maar dat is huiswerk voor de lezer. Op 64-bit architecturen is het zeer waarschijnlijk dat 64-bit variabelen wel atomair zijn. Daar is het fysieke geheugen van de computer namelijk geen beperking.

Compileroptimalisaties

Laten we nu eens kijken welke compileroptimalisaties er allemaal automatisch kunnen worden uitgevoerd op Java-code. Compileroptimalisaties kunnen op verschillende niveaus worden uitgevoerd. Meteen bij het compileren van de broncode naar Java-bytecode, tijdens het uitvoeren van de bytecode door de virtual machine, maar ook tijdens het uitvoeren van de code door de processor. Moderne processors zijn namelijk heel goed in staat om code te optimaliseren. Al deze manieren van optimaliseren noemen we voor het gemak compileroptimalisaties.

Volgorde wijzigen

De eerste optimalisatie die we bekijken is het wijzigen van de volgorde waarin *assign*-acties (op verschillende variabelen binnen dezelfde thread) worden uitgevoerd. Dit mag alleen als de functionaliteit niet wijzigt – binnen de thread! Op andere threads kan dit vreemde gevolgen hebben – als ze niet op de juiste manier gesynchroniseerd zijn. Het wijzigen van de volgorde mag daarom niet ten opzichte van *lock*- en *unlock*-acties.

```
// Assumption: if y then z is valid.
void save(int value) {
   z = value;
   y = true;
}

int get() {
   if (y) {
          y = false;
          return z;
   }

   return -1;
}
```

Codevoorbeeld 1.

Als de compiler de volgorde van de assign-acties in de `save()` methode in het voorgaande voorbeeld omdraait, dan geldt de aanname niet meer. In slecht gesynchroniseerde threads kan dan een waarde door de `get()` methode worden teruggegeven die niet geldig is!

Prescient store

Een andere optimalisatie is de zogenaamde prescient store. Dit gebeurt als er een `store`-actie wordt uitgevoerd, voordat de *assign*-actie heeft plaatsgevonden! Dit klinkt vreemd (hoe kan de compiler weten welke waarde er uit gaat komen, als deze nog niet berekend is), maar er zijn veel loops te bedenken, waar vooraf voorspelbaar is, wat het resultaat zal zijn.

```
int index;

for (index = 0; index < 12; index++) {
   // Do something
```

Compiler-optimalisaties kunnen op verschillende niveaus worden uitgevoerd

```
}

// Guess the value of index
```

Codevoorbeeld 2.

Neem deze for-lus als voorbeeld. Het is duidelijk dat de lus zal eindigen met een waarde `12` voor `index`, mits er geen excepties optreden! Een prescient store mag daarom alleen als er zeker geen excepties zullen optreden – vals spelen is niet toegestaan!

Forward substitution

Een ander veelvoorkomende optimalisatie is de forward substitution. Als er twee opeenvolgende *use*-acties van dezelfde variabele zijn geprogrammeerd, worden in die gevallen de tussenliggende *read*- en *load*-acties overgeslagen. Dit kan alleen als er tussen de use-acties in geen *assign*-acties plaatsvinden – uiteraard binnen de thread.

```
void loadRegisters() {
   // r2 is eligible for substitution
   r1 = point.x; r2 = point.x;
}

void setX(int x) {
   point.x = x;
}
```

Codevoorbeeld 3.

In codevoorbeeld 3 wordt de assignment van `r2` gewijzigd in `r2 = r1`. Als `setX()` wordt uitgevoerd, parallel aan `loadRegisters()`, dan zou je mogelijk verwachten dat `r2` de nieuwe waarde van `point.x` heeft. Maar dat gebeurt na de optimalisatie niet meer! Denk eraan dat een *lock*-actie de werkkopieën van variabelen uit het werkgeheugen flusht. Dit wil echter nog niet zeggen dat er geen forward substitution op kan plaatsvinden: de variabele kan namelijk ook nog in een register van de processor staan. Variabelen die `volatile` zijn, komen niet in aanmerking voor forward substitution. Dit betekent dat `volatile` variabelen een slechtere performance hebben. Alle variabelen `volatile` maken is dus een slecht idee.

Final optimalisatie

Variabelen die `final` zijn, zijn perfecte kandidaten om forward substitution op uit te voeren. Aangezien de waarde nooit wijzigt, is het ook niet nodig om deze te flushen uit de werkkopie tijdens een lock-actie. Het `final` maken van een variabele is dus vooral een performanceoptimalisatie.

Consequenties

De consequenties van het JMM zijn dat als variabelen door meerdere threads worden gebruikt, je deze op drie manieren veilig kunt publiceren:

- Gebruik normale synchronisatie.
- Maak de variabele `volatile`.
- Maak de variabele `final`.

Als je de synchronisatie van jouw threads niet goed hebt gedaan, dan is het gedrag onvoorspelbaar, als gevolg van compileroptimalisaties.
Je kunt dus maar beter zorgen dat de synchronisatie klopt.

Valkuilen

Nu we gezien hebben hoe we onze variabelen veilig kunnen publiceren, is het tijd om te kijken wat we hier allemaal fout kunnen doen. Per geval wordt uitgelegd wat er fout gaat en waarom.

Eenzijdig locken

Dit is het geval als alleen schrijfacties op een variabele gelockt worden, maar de leesacties niet. In dat geval vindt er wel synchronisatie plaats tijdens het schrijven, maar niet tijdens het lezen. Als er bij het lezen dus *forward substitution* wordt uitgevoerd op de variabele, dan kan het lang duren voordat de lees-thread de wijziging van de schrijf-thread overneemt. De oplossing voor dit probleem is normale synchronisatie te gebruiken en zowel de lees- als de schrijfacties te synchroniseren.

Locken op een andere monitor

Als er gelockt wordt op verschillende monitoren, is de volgorde van acties niet gegarandeerd. Een voorbeeld:

```
Object mutex = new Object();
int count;
int index;

synchronized void setCount(int x) {
   System.out.println(index);
   count = x;
}

int setIndex(int x) {
   synchronized(mutex) {
          index = x;
          System.out.println(count);
   }
}
```

Codevoorbeeld 4.

Als `setIndex()` in een andere thread dan `setCount()` wordt uitgevoerd, dan kunnen we niet garanderen dat de eerste methode de waarde gezet in de tweede methode ziet. Bovendien kunnen we niet garanderen dat de volgorde waarin de acties geprogrammeerd staan, ook zo door de andere thread worden waargenomen (als gevolg van de 'volgorde wijzigen' compileroptimalisatie). We kunnen dan volgorden van output waarnemen die we niet kunnen verklaren.

Hoewel het JMM garandeert dat alle variabelen in het werkgeheugen bij een *lock*-actie verwijderd worden en alle gewijzigde waarden in het werkgeheugen naar het gemeenschappelijke geheugen geschreven worden, is er geen garantie over de waargenomen volgorde, aangezien het niet om dezelfde lock monitor gaat. Dit betekent dat de synchronisatie op de `setCount()` methode niet garandeert dat de tweede methode niet tegelijkertijd wordt uitgevoerd. De oplossing voor dit probleem is de synchronisatie op het mutex-object te verwijderen en de `setIndex()` methode ook `synchronised` te maken.

Niet-atomaire variabelen

Als een `long` of `double` variabele gebruikt wordt, dan zijn lees- en schrijfacties niet atomair op 32-bit hardware. Als er geen sprake is van locking, dan wordt het wegschrijven van de lage orde bytes van de variabele en de hoge orde bytes mogelijk onderbroken door een andere thread. Als die andere thread deze variabele dan benadert, kan een inconsistente waarde uit het niets vandaan getoverd worden. Dit probleem kan op drie manieren worden opgelost:

- Gebruik normale synchronisatie.
- Maak de variabele `volatile`.
- Of gebruik de `AtomicLong` class uit het `java.util.concurrent.atomic package`.

Attributen niet volatile

De waarde van een variabele die niet `volatile` is, en waarvan de lees- en schrijfacties niet gesynchroniseerd zijn, is mogelijk niet zichtbaar in een andere thread. Een bekend voorbeeld:

```
class Example implements Runnable {
   boolean keepRunning = true;

   public void run() {
          while (keepRunning) {
                    // Do stuff
          }
   }

   public void stopRunning() {
          keepRunning = false;
   }
}
```

Codevoorbeeld 5.

Na het aanroepen van de `stopRunning()` methode zou je mogelijk verwachten dat de `run()` methode (die in een andere thread wordt uitgevoerd) meteen eindigt. Het JMM geeft hier echter geen enkele garantie voor, aangezien de `keepRunning` variabele niet gesynchroniseerd wordt. De variabele is bijvoorbeeld een kandidaat voor forward substitution. De oplossing voor dit probleem is eenvoudig:

- Maak de variabele `volatile`.
- Of gebruik de `Atomic*` classes.

Attributen niet final

Als een variabele bij de constructie van het omvattende object gevuld wordt en later nooit meer wijzigt (en door meerdere threads tegelijkertijd benaderd kan worden), dan is het handig om de variabele `final` te maken. Het niet `final` maken zal op zichzelf geen fouten opleveren. Maar het betekent wel een factor drie slechtere performance. Zeker als je bedenkt dat de variabele, als deze wel wijzigt, `volatile` zou moeten zijn. Het `final` maken van de variabele is bovendien een trigger voor andere programmeurs. Als de variabele plotseling wel van waarde moet kunnen wijzigen, moet de programmeur weten dat de variabele door meerdere threads benaderd kan worden en dus `volatile` gemaakt moet worden.

ThreadGroup.activeCount()

Deze methode wordt soms gebruikt om te controleren of een vooraf bekend aantal threads klaar is met hun werk. Helaas werkt dit niet. Dat komt omdat threads op de achtergrond nieuwe threads starten, die ook in de `activeCount()` worden meegerekend. Grafische elementen gebruiken bijvoorbeeld aparte workthreads. De Javadoc van de methode maakt ook duidelijk dat de methode alleen een *schatting* teruggeeft. Dit probleem kan opgelost worden door gebruik te maken van `CountDownLatch` (een `java.util.concurrency` class). Dit werkt zo:

```
class Worker implements Runnable {
   WorkUnit workUnit;
   CountDownLatch latch;

   public Worker(WorkUnit u,
                    CountDownLatch l) {
          workUnit = u;
          latch = l;
   }

   public void run() {
          // Do the work
          workUnit.process();
          latch.countDown();
   }
}

class Driver {
   void execute(WorkUnit[] work) {
          CountDownLatch latch = new
                    CountDownLatch(work.length);

          for (WorkUnit unit : work) {
                    new Thread(new Worker(unit,
                              latch)).start();
          }

          latch.await();
   }
}
```

Codevoorbeeld 6.

java.util.concurrent

In Java 5 is een nieuwe package toegevoegd, te weten `java.util.concurrent`. In deze package staat een aantal interessante classes voor veelvoorkomende synchronisatieproblemen. Aangezien er

al eerder artikelen over deze nieuwe classes in het Java Magazine verschenen zijn, zullen we hier niet opnieuw een volledige opsomming geven. Wat wel belangrijk is om te weten, is dat normale synchronisatie net zo snel is als de concurrency classes als er geen lock contenders zijn (als er dus maar één thread is die een lock wil hebben). In Java 6 is de snelheid zelfs vergelijkbaar als er wel lock contenders zijn. Synchronisatie is over de jaren heen sterk in performance verbeterd. Het is nu zover geoptimaliseerd dat het bijna geen tijd meer kost om een lock te verkrijgen, zeker als er geen andere threads zijn die wachten op dezelfde lock. Uit performanceoogpunt maakt het daarom niet veel uit of je gebruikmaakt van normale synchronisatie of van de Java 5 concurrent classes. Overigens is performance vaak niet het probleem.

Een veelvoorkomende valkuil in het gebruik van de concurrent classes treedt op als er naast de geboden functionaliteit nog extra synchronisatie nodig is. De eerste oplossing voor dit probleem die bij de meeste programmeurs opkomt is om te synchroniseren op de concurrent class. Maar dat werkt niet. De concurrent classes gebruiken namelijk niet de class zelf om te synchroniseren, maar maken gebruik van mutex-objecten. Dit betekent dat bij het synchroniseren op de concurrent class locking op een andere monitor plaatsvindt dan degene die de concurrent class zelf gebruikt. Aangezien de mutex-objecten niet buiten de concurrent class benaderbaar zijn, is het niet mogelijk om dit op te lossen. De enige mogelijkheid die overblijft, is een en ander zelf te bouwen met normale synchronisatie.

Singleton

Als je een singleton pattern wil implementeren die door meerdere threads benaderd kan worden en die gebruik moet maken van lazy initialisatie, dan is het nog knap lastig om dit goed te krijgen.

```
final class Singleton {
  static Singleton instance = null;

  private Singleton() {
        super();
  }

  static Singleton instance() {
        if (instance == null) {
                instance = new Singleton();
        }
        return instance;
  }
}
```

Codevoorbeeld 7.

In bovenstaande implementatie kan de `instance()` methode door meerdere threads tegelijkertijd worden uitgevoerd. Dan zouden er meerdere instances van Singleton kunnen ontstaan (en dat mocht juist niet). Ook kan de `instance` variabele onderwerp zijn van forward substitution. De eerste reactie van programmeurs is vaak het aanmaken van de instance dan te synchroniseren.

```
static Singleton instance() {
  if (instance == null) {
        synchronized (Singleton.class) {
                instance = new Singleton();
        }
  }
  return instance;
}
```

Codevoorbeeld 8.

Dit is echter een typisch geval van éénzijdige locking. Degenen die dit realiseren zullen geneigd zijn, om het if-statement nog een keer te herhalen in het synchronized block.

```
static Singleton instance() {
  if (instance == null) {
        synchronized (Singleton.class) {
                if (instance == null)
                        instance    =    new
Singleton();
        }
  }
  return instance;
}
```

Codevoorbeeld 9.

Maar ook dit is een foutieve oplossing. Het tweede if-statement is namelijk een kandidaat voor forward substitution. Sterker nog, het hele if-statement kan weggeoptimaliseerd worden. Binnen één thread bekeken is het statement namelijk overbodig. We zouden de `instance` variabele `volatile` kunnen maken. Maar ook dat werkt niet. We hebben dan weliswaar geen last meer van forward substitution, maar twee threads kunnen nog steeds tegelijkertijd in het if-statement zitten. We kunnen de variabele wel `final` maken. Maar dan hebben we geen lazy initialisatie meer, dus dat kan alleen als dat geen requirement is.

Een goede oplossing is om de `instance()` methode `synchronised` te maken. Dit heeft echter wel een nadeel: als een thread eenmaal de juiste (enige) instantie van de Singleton heeft, dan heb je de synchronisatie niet meer nodig. Deze wordt vervolgens wel bij elke aanroep van de `instance()` methode uitgevoerd. Dit is qua performance dus een mindere oplossing.

De beste oplossing voor dit probleem is daarom een `Factory` class te gebruiken. De oplossing ziet er dan als volgt uit:

```
final class Singleton {
  static Singleton instance = null;

  private Singleton() {
        super();
  }
```

```
  static Singleton instance() {
        if (instance == null) {
                instance = Factory.get();
        }
        return instance;
  }

  private static class Factory {
        static Singleton s = null;

        static synchronized Singleton
                              get() {
                if (s == null) {
                        s = new Singleton();
                }
                return s;
        }
  }
}
```

Codevoorbeeld 10.

Aangezien `s` een andere variabele is dan `instance`, is er geen sprake van eenzijdig locken of forward substitution (op deze variabele). Hoewel elke thread mogelijk een keer de `get()` methode uitvoert, is dat geen probleem, aangezien de `Factory` er voor zorgt dat er daadwerkelijk altijd maar één instantie is. Als een thread eenmaal die instantie heeft, maakt het niet meer uit of er forward substitution wordt uitgevoerd.

Samenvatting

Samengevat hebben we het volgende geleerd:

- We kunnen onze attributen veilig publiceren door gebruik te maken van `final`, `volatile` en normale synchronisatie (locking).
- Bij locking moeten we er op letten dat we altijd zowel de lees- als de schrijfacties locken en dat deze gebruik maken van dezelfde lock monitor.
- Gebruik de concurrent classes niet als je extra synchronisatie nodig hebt. Gebruik in die gevallen normale synchronisatie. «

Referenties

[a] The Java Virtual Machine Specification – Second Edition, by Tim Lindholm and Frank Yellin.

[b] The Java Language Specification – Third Edition, by James Gosling, Bill Joy, Guy Steele, Gilad Bracha.

Patches Patches Patches Patches Patches Patches Patches P

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.

Preview van SOMA 3.0 met SOMA Game Wrapper voor J2ME

Smaato inc. biedt deze zomer een oplossing aan voor het plaatsen van advertenties in mobiele applicaties zonder de noodzaak code aan te passen, de SOMA wrapper.

De SOMA Wrapper is een service die het mogelijk maakt verschillende soorten advertentieformaten aan bestaande binary Java ME-applicaties toe te voegen, dus aan JAR-files. De advertenties kunnen bij start-up en bij exit, timer-gebaseerd en tussen levels (in games) getoond worden. De advertentieonderdelen worden als het ware om de JAR-file heen gewikkeld.

De gratis SOMA SDK is gebundeld met J2ME Polish van Enough Software. Deze toolkit is wijd verbreid onder meer door commeciële licneties van Nokia, Motorola, Mozilla en anderen. Het aanklikken van de advertentieoptie en recompilen schijnt genoeg te zijn om advertenties aan de applicaties toe te voegen.

Het SOMA-platform ondersteunt overigens niet alleen Java-smartphones, maar naar eigen zeggen alle besturingssystemen, waaronder Symbian, Windows Mobile, Palm en Blackberry en als een gesloten bèta de SOMA SDK voor de Apple iPhone.

JEXIN, Java foutsimulatie-platform

Versie 0.5,een voorlopige versie dus, van het Java foutsimulatie-platform Jexin is vrijgegeven.

Jexin maak gebruik van exception injectie voor error simulatie. Bij exception injectie wordt een method call afgevangen waarbij een gebruikergedefinieerde exceptie wordt gebruikt om de fout te simuleren.

Jexin maakt gebruik van Java annotations en aspects om methoden te identificeren. Of en wanneer een exceptie moet worden geïnjecteerd, wordt op runtime vastgesteld door de Jexin Web applicatie. De Jexin webapplicatie moet op en server worden geïnstalleerd die toegankelijk is voor de betreffende applicatie(s). Er is slechts één server instantie nodig om te werken met meerdere applicaties en/of omgevingen.

Sun kondigt Mural aan: Open Master Data Management

Sun Microsystems heeft een community gelanceerd onder de naam Mural die een MDM-oplossing moet bieden op basis van open standaarden.

Mural is een open source gemenschap rond het idee de datasilo's te beheren die grote ondernemingen iedere dag genereren. Mural behelst verschillende gemeenschapsprojecten die helpen om bedrijven gestructureerde en ongestructureerde data te beheren die over verschillenden systemen op verschillende platforms verspreid is. De gemeenschap bevat componenten die het (nu al) mogelijk maken een echte MDM-oplossing te bouwen en één weergave van gedistribueerde data te genereren.

Het is gebaseerd op toepassingen in verticale markten zoals de gezondheidsmarkt, de overheid en retail.
Projecten die deel uitmaken van Rural zijn: Master Index Studio, Data Integrator, Data Quality, Data Services, Data Migration. Mural is gebaseerd op de GlassFish, Open ESB, and NetBeans open-source communities. Mural-projecten kennen de CDDL-licentie. Mural is tevens in commerciële vorm verkrijgbaar met de Sun MDM Suite die in juni verkrijgbaar wordt.