Het `java.util.concurrent` package in Java 2 Standard Edition versie 5 (J2SE 5) levert ons een reeks herbruikbare bouwstenen waarmee het eenvoudiger wordt multithreaded applicaties te ontwikkelen. In een vorig artikel werden al de nieuwe classes besproken die gebruikt worden voor de synchronisatie van multiple threads zoals semaphores, mutexes, latches en barriers. In dit artikel gaat Willem Koppenol in op het Executor Framework en de uitbreidingen van het Collections Framework.

Concurrency in J2SE 5

Een belangrijk onderdeel van het package is een nieuw thread Framework, het Executor Framework, waarmee het uitvoeren en managen van asynchrone taken wordt vergemakkelijkt. Ook het tegelijkertijd benaderen van collections vanuit verschillende threads is in J2SE 5 vereenvoudigd door de toevoeging van een aantal concurrent collection-classes en de introductie van de `Queue` interface.

JSR 166 De set concurrency-primitieven die sinds het begin in Java zijn ingebouwd zoals `synchronized`, `wait()` , `notify()` en `notifyAll()`, zijn lastig om correct te gebruiken. Ze zijn inderdaad nogal primitief. Ontwikkelaars denken net zo min in termen van `wait()` en `notify()` als termen van byte-codes. Ontwikkelaars hebben voor concurrent-applicaties sterke behoefte aan een bibliotheek van meer geavanceerde bouwstenen, zoals semaphoren, locks, thread pools en thread-safe collections. Een dergelijke bibliotheek van scheduling- en concurrency-classes kan bovenop de bestaande Java concurrency-primitieven worden gebouwd. Maar het is zelfs voor experts nog niet eenvoudig dit goed te krijgen, want er zijn vele deadlock-, race conditions-, performance-, thread-safeness- en resource management-gevaren die op de loer liggen. Toch hebben veel ontwikkelteams in de loop der jaren een dergelijke bibliotheek ontwikkeld, met alle problemen van dien. Een bibliotheek die reeds geruime tijd bestond en langzamerhand met de hulp van het gerelateerde boek “Concurrent Java Programming” van Addison-Wesley aan populariteit won, is de concurrency-bibliotheek van Professor Doug Lea van State University New York. Deze bibliotheek is goed getest in tal van praktijkprojecten en wordt bijvoorbeeld ook gebruikt in de JBoss J2EE-applicatieserver. Ze is nu aangepast en gestandaardiseerd via Java Specification Request (JSR) 166 en vormt de basis van het `java.util.concurrent` package in J2SE 5.

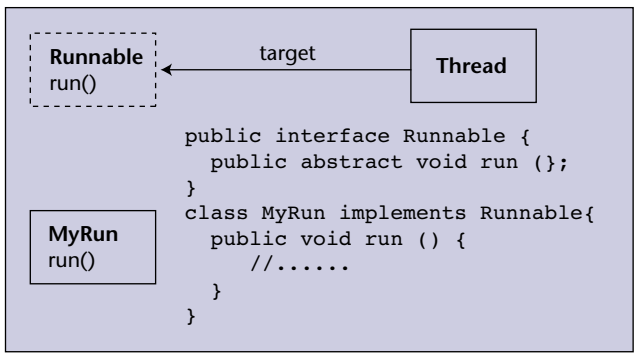
Executor Framework Threads worden in Java gecreëerd door een object dat van de Thread class is afgeleid te instantiëren en vervolgens `Thread.start()` aan te roepen. Van oudsher hebben ontwikkelaars daarbij een tweetal opties. Ofwel ze maken een afgeleide Thread class en implementeren daarvan de run() methode:

```
class MyThread extends Thread {
   public void run() { /* do work */ }
}
Thread t = new MyThread();
t.start();
```

Ofwel ze maken een class die de Runnable interface met de `run()` methode implementeert en gebruiken de `Thread(Runnable)` constructor:

```
Thread t = new Thread(new Runnable() {
     public void run() { /* do work */ }
}
t.start();
```

In beide gevallen wordt `run()` in de context van een nieuwe thread uitgevoerd.



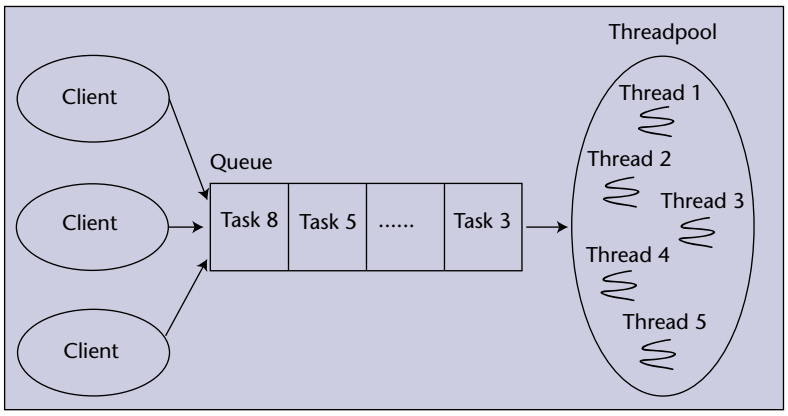
Figuur 1. Een Runnable taak

Een in het oog springend kenmerk van het nieuwe `java.util.concurrent` package is dat het starten van threads voortaan via een `Executor` kan en ook zou moeten verlopen. Een `Executor` is een object dat Runnable taken uitvoert. Na het creëren van een `Executor` met een factory methode, spreek je een `Executor` als volgt aan:

```
Executor executor = Executor factory method;
exector.execute (aRunnable);
```

Bij het starten van een enkele nieuwe thread is er mogelijk niet zoveel reden een `Executor` te gebruiken. De meeste multithreaded applicaties werken echter met meer threads. Threads hebben een stack en nemen geheugenruimte in beslag. Thread-creatie kan duur zijn - afhankelijk van het platform - en het stoppen van threads kan problemen geven. Het opstarten van steeds nieuwe threads voor een kortdurende simpele taak zoals het afhandelen van een event is inefficiënt. Een dergelijke oplossing is ook niet schaalbaar. Beter is het threads te hergebruiken en een thread-pool te gebruiken. Maar een goed ontworpen thread-pool mechanisme is niet eenvoudig.

Het nieuwe Executor Framework lost dit soort problemen op doordat het de opdracht de taak uit te voeren loskoppelt van de details van de uitvoering van de taak, zoals of en hoe threads gebruik worden, of hoe de scheduling verloopt. Het Executor Framework biedt met een aantal uitbreidbare classes en interfaces, waaronder een flexibele thread-pool implementatie, een standaardoplossing voor het creëren, schedulen en uitvoeren van taken. De ontwikkelaar wordt daarbij afgeschermd van de laag-bij-de-grondse details.



FIGUUR 2. Thread Pool en Queue met taken.

EXECUTOR INTERFACE Centraal in het Framework staat de `Executor` interface. Dit bestaat uit een simpele functie om een opdracht tot het uitvoeren van een taak in ontvangst te nemen:

```
interface Executor {
void execute(Runnable command)
}
```

De implementatie van de `Executor`-interface bepaalt hoe de taak wordt uitgevoerd, de execution policy. Zo kan een `Executor` een taak zowel synchroon als asynchroon uit laten voeren. Een synchrone `Executor` draait de aangeboden taken direct in de thread van de aanroeper en ziet er als volgt uit:

```
class DirectExecutor implements Executor {
      public void execute(Runnable r) {
          r.run();
      }
}
```

Een asynchrone `Executor` start voor iedere aangeboden taak een aparte thread en ziet er als volgt uit:

```
class ThreadPerTaskExecutor implements
Executor {
      public void execute(Runnable r) {
          new Thread(r).start();
      }
}
```

Als er verschillende `Runnable` taken moeten worden uitgevoerd en de volgorde van uitvoering is niet van belang, kun je een thread-pool aanmaken met `newFixedThreadPool(..)` van de `Executors` class en de taken door de thread-pool laten uitvoeren.

```
Executor executor = Executors.newFixedThread-
Pool(5);
executor.execute (new RunnableTask1 ());
executor.execute (new RunnableTask2 ());
executor.execute (new RunnableTask3 ());
....
```

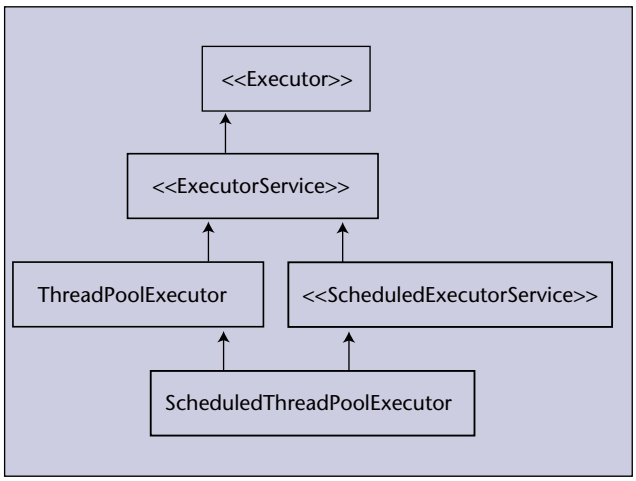
De taken worden onder volledige controle van de `Executor` uitgevoerd, die de threads uit de thread-pool naar believen hergebruikt zonder de overhead van thread-creatie.

EXECUTORS Het `java.util.concurrent` package kent verschillende `Executor` implementaties met een wisselende execution-policy. Ze worden gemaakt met static factory-methoden van de `Executors` class:

- `Executors.newCachedThreadPool()` : creëert een ongelimiteerde thread pool, die eerder aangemaakte threads hergebruikt. Als er geen thread beschikbaar is, wordt er een nieuwe thread aangemaakt en aan de pool toegevoegd. Threads die gedurende zestig seconden niet zijn gebruikt, worden beëindigd.
- `Executors.newFixedThreadPool(int n)`: creëert een thread-pool met een vast aantal threads. Als één van de threads crasht, zal een nieuwe thread automatisch zijn plaats innemen.
- `Executors.newSingleThreadExecutor()`: creëert een `Executor` die één thread gebruikt om taken uit te voeren. De aangeboden taken worden gegarandeerd sequentieel uitgevoerd en nooit is er meer dan één taak tegelijkertijd actief.

ExecutorService De meeste implementaties van `Executor` implementeren ook de `ExecutorService` interface. `ExecutorService` is een uitbreiding op `Executor`, waaraan methoden zijn toegevoegd voor het managen van de levenscyclus van de taken. Zo kan een `ExecutorService` gesloten worden met `shutdown()`. Nieuwe taken worden dan niet meer geaccepteerd en lopende taken worden op een ordelijke manier afgehandeld. Verder voorziet een `ExecutorService` in methoden die wachten op de beëindiging van een taak zoals `awaitTermination(..)`. De methoden `invokeAny()` en `invokeAll()` zijn nuttig om bulk-executie van een reeks taken te starten en dan te wachten tot er één of meer klaar zijn. De `submit()` methode is een uitbreiding op `execute()`. Deze geeft een `Future` interface terug waarmee een taak kan worden afgebroken of waarmee gewacht kan worden tot een taak klaar is.

Een specialisatie van `ExecutorService` is de `ScheduledExecutorService` interface met de methoden `scheduleAtFixedRate()` en `scheduleWithFixedDelay()` om taken met een zekere vertraging of periodiek uit te voeren.



Figuur 3. Deel van de class en interface hierarchy van het Executor Framework.

ThreadPoolExecutor Een in het `java.util.concurrent` package meegeleverde standaard-implementatie van de `ExecutorService` interface is de `ThreadPoolExecutor`. In feite retourneren de factory-methoden uit de `Executors` class-instanties van een `ThreadPoolExecutor`. Deze class is een uitbreidbare implementatie van een thread-pool en kent vele configuratie-opties.

De `ThreadPoolExecutor` kent de `corePoolSize` en de `maximumPoolSize` eigenschappen. De grootte van de thread-pool kun je beïnvloeden door `setCorePoolSize(int)` en de `setMaximumPoolSize(int)`. Een thread pool met een vaste grootte maak je door aan deze methoden hetzelfde mee te geven. Met `setMaximumPoolSize(Integer.MAX_VALUE)` krijg je een pool die een willekeurig groot aantal threads kan bevatten. Default zal een `ThreadPoolExecutor` zijn core-threads nog niet starten als er nog geen taken zijn om uit te voeren. Dit gedrag verander je met de `prestartCoreThread(s)` functies.

Een verzoek aan een `ThreadPoolExecutor` om een taak uit te voeren leidt tot het opstarten van een nieuwe thread als er minder dan `corePoolSize` threads draaien. Als er al `corePoolSize` of meer threads draaien wordt de taak in een queue gezet. Als dit echter niet kan, wordt toch een nieuwe thread gestart, tenzij dit zou leiden tot overschrijding van de `maximumPoolSize`. In dit laatste geval wordt de taak geweigerd.

De `ThreadPoolExecutor` kan ook worden geconfigureerd door aan de constructor een `ThreadFactory` mee te geven. Een `ThreadFactory` maakt de nieuwe threads aan die een executor gebruikt. Door een aangepaste `ThreadFactory` te gebruiken kun je bepalen dat de threads specifieke eigenschappen krijgen zoals een bepaalde prioriteit of naam. Een voorbeeld van een `ThreadFactory` die daemon threads (houden de JVM niet in leven) aanmaakt is:

```
public class DaemonThreadFactory
implements ThreadFactory {
      public Thread newThread(Runnable r)
{
         Thread thread = new Thread(r);
         thread.setDaemon(true);
         return thread;
     }
}
```

De `ThreadPoolExecutor` heeft verder `beforeExecute()` en `afterExecute()` methoden die voor en na de uitvoering van iedere taak worden uitgevoerd. Deze methoden zijn protected en als je ze wilt gebruiken, is het nodig de class hiervoor eerst te subclassen. Een subclass van `ThreadPoolExecutor` is `ScheduledThread`

PoolExecutor, met extra functionaliteit voor het periodiek uitvoeren van taken.

Callable interface Nieuw in het java.util.concurrent package is ook de Callable interface, die een alternatief vormt voor Runnable. Net als Runnable is Callable bedoeld om geïmplementeerd te worden door classes, waarvan instanties door andere threads worden uitgevoerd. Callable heft echter twee nadelen op, die wel bij Runnable optreden. De run() methode in Runnable kan geen resultaat teruggegeven (i.e. retourneert void). Als je een resultaat van een Runnable taak wilt krijgen, is het noodzakelijk dit extern te programmeren. Een veelgebruikte techniek is een instancevariabele in het Runnable object te zetten en een functie te maken die deze variabele teruggeeft:

```
<< begin kader met computercode >>
public MyRunnable implements Runnable
{
  private int fResult = 0;
  public void run () {
    ...
    fResult = 1;
  } // run

  // A getter method to provide the result of
the thread.
  public int getResult () { return fResult; }
} // class MyRunnable
<< einde met computercode >>
```

De run() methode in Runnable kan ook geen exceptie opleveren. Als je probeert een exceptie te genereren in de run()methode moet je een throws clause in de methode-signature opnemen. Dit kan echter niet omdat run()van de superclass deze exceptie niet declareert.

De Callable interface lost deze problemen op. In plaats van run()definieert het Callable de call() methode die geen parameters verwacht, een Integer retourneert en die een exceptie mag genereren. Een eenvoudig voorbeeld is:

```
  import java.util.concurrent.*;
  public  class  MyCallable  implements
Callable
  {
     public  Integer  call  ()  throws  java.
io.IOException {
       return 1;
    }
  } // MyCallable
```

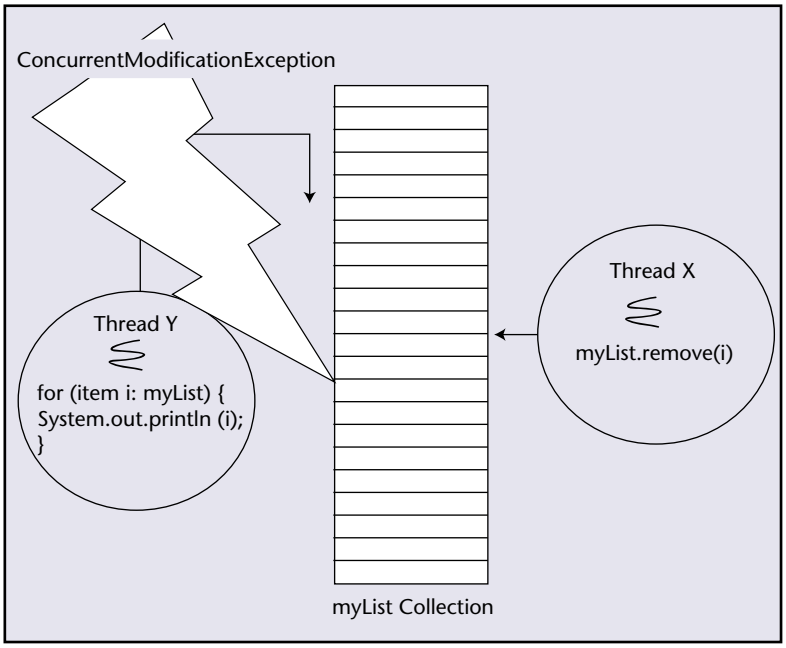
Future en FutureTask Een Callable wordt vaak in combinatie met Future en FutureTask gebruikt. Voor J2SE 5 was het lastig om de status van een draaiende taak te bepalen en een taak na beëindiging een waarde te laten teruggeven. Met het Future interface en de FutureTask class is dit nu wel mogelijk. De Future interface heeft methoden om te checken of een taak klaar is, de resultaten van een taak op te halen dan wel erop te wachten of de uitvoering van een taak te stoppen. FutureTask heeft een basisimplementatie van de methoden uit Future. Een FutureTask kan via de constructor een Callable meekrijgen. De resultaten van de uitvoering van de taak worden opgehaald met de get() methode van FutureTask en als de taak nog niet klaar is zal get()blokkeren. In onderstaande voorbeeldcode gebruiken we een FutureTask class die een Integer return waarde ondersteunt. De taak wordt aan de submit() methode van ExecutorService meegegeven. Het resultaat wordt opgehaald met get().

```
FutureTask task = new FutureTask (new
MyCallable ());
ExecutorService es = Executors.newSingle-
ThreadExecutor ();
es.submit (task);
try {
  int result = task.get ();
  System.out.println ("Result from task.get
() = " + result);
}
catch (Exception e) {
  System.err.println (e);
}
es.shutdown ();
```

Thread Safe Collections Het Collections Framework in het java.util package zoals dat werd geïntroduceerd in JDK 1.2 is prima geschikt voor het representeren van objectcollecties. Wat betreft het benaderen vanuit verschillende threads hebben deze collecties echter enige nadelen. Hashtable en Vector zijn volledig gesynchroniseerd en tot op zekere hoogte veilig te benaderen vanuit verschillende threads (thread-safe). Andere collecties, zoals ArrayList of HashMap, zijn geheel ongesynchroniseerd en worden thread-safe gemaakt door er synchronization-wrappers omheen te leggen. Deze thread-safeness is echter niet volledig. De collecties retourneren namelijk fail-fast iteratoren die ervan uitgaan dat de inhoud van een collectie niet verandert terwijl een thread door de objecten van een collectie itereert. Gebeurt dat toch dan zal een fail-fast iterator dat detecteren en zal er een ConcurrentModificationException optreden. Om dit te voorkomen is het nodig de collecties nog extra te locken tijdens itera-

ties, of ze eerst te klonen alvorens te itereren. Beide opties zijn slecht voor de performance. Deze karakteristiek wordt daarom ook wel *conditional thread safety* genoemd. Verder is de performance van de collecties matig als ze vaak vanuit verschillende threads worden benaderd.

De eis dat een collectie niet verandert tijdens iteraties is ongemakkelijk voor vele concurrent-applicaties. Het `java.util.concurrent` package komt daarom met een aantal nieuwe thread-safe collection classes zoals `ConcurrentHashMap`, `CopyOnWriteArrayList` en `CopyOnWriteArraySet`. Deze classes zijn echte thread-safe versies van de basis collection types en hebben een geoptimaliseerde performance in een multithreaded omgeving. De iteratoren die de `java.util.concurrent` collections retourneren worden zwak consistent genoemd. Ze krijgen snapshot-style data van de concurrent collection-classes. Als de onderliggende data veranderen treden geen excepties op, maar worden de veranderingen niet gereflecteerd door de iteratoren.



FIGUUR 4. ConcurrentModificationException in het Collection Framework

CopyOnWriteArrayList `CopyOnWriteArraySet` is een thread-safe variant van set en `CopyOnWriteArrayList` is een thread-safe variant van `ArrayList`. Beiden maken ze als een object wordt toegevoegd, gewijzigd of verwijderd een kopie van de onderliggende array of set, en brengen daar de verandering in aan. Iteraties die al aan de gang waren, blijven werken met de kopie die actueel was op het moment dat de iteratie begon. Als gevolg hiervan zijn leesacties snel maar updates langzaam. Hoewel aan het kopiëren natuurlijk ook de nodige kosten zijn verbonden, zullen iteraties over het algemeen vaker voorkomen dan wijzigingen. In deze gevallen biedt copy-on-write een betere performance en concurrency dan de alternatieven.

ConcurrentHashMap In een hash-based `Map` class worden objecten opgeslagen onder een key. De key is gebaseerd op een hash-codefunctie en door het uitvoeren van deze functie kan heel snel worden bepaald waar in de collectie een object moet worden opgeslagen of opgehaald. Een thread-safe `Map` krijg je door een `Hashtable` of de synchronized wrapper om een `Hashmap` te gebruiken. Deze `Map` implementaties bereiken thread-safety door iedere methode te synchroniseren. Het nadeel daarvan is dat het niet schaalbaar is, omdat maar één thread de `Hashtable` tegelijkertijd kan benaderen. Bovendien is het toch niet volledig thread-safe omdat samengestelde operaties zoals iteraties of `put-if-absent` toch nog extra synchronisatie nodig hebben.

In J2SE 5 is daarom de `ConcurrentHashMap` implementatie toegevoegd die bij uitstek bedoeld is voor het gebruik door multiple threads. De `ConcurrentHashMap` implementatie is veel schaalbaarder dan `HashTable` en biedt een betere performance. `ConcurrentHashMap` geeft volledige thread-safe concurrency voor leesoperaties, staat bijna altijd concurrent lees- en schrijfoperaties toe en laat ook meerdere gelijktijdige schrijfoperaties vaak toe.

Queue Interface J2SE 5 biedt naast de reeds bestaande `List`, `Set` en `Map` interfaces ook een tweetal nieuwe collection interfaces: `Queue` en `BlockingQueue`. De `Queue` interface is vergelijkbaar met `List`, maar is veel simpeler omdat elementen achteraan worden toegevoegd en vooraan worden opgehaald. Met het `List` interface kunnen elementen op een willekeurige plek worden benaderd.

```
interface Queue<E> extends Collection<E> {
boolean offer(E x);
E poll();
E remove() throws NoSuchElementException;
E peek();
E element() throws NoSuchElementException;
}
```

Een `LinkedList` wordt vaak gebruikt om een lijst of rij van items, bijvoorbeeld uit te voeren taken, op te slaan. Voor veel applicaties is de mogelijkheid om items achteraan toe te voegen en vooraan op te halen echter afdoende. Aangezien een `LinkedList` het volledige `List` interface ondersteunt is deze implementatie voor een dergelijke toepassing niet zo efficiënt. Een `Queue` implementatie geeft dan een betere performance.

Het `Queue` interface laat het aan de implementatie

Advertentie Transfer Solution

over om de volgorde te bepalen waarin elementen worden opgeslagen. De `ConcurrentLinkedQueue` class implementeert een thread-safe, non-blocking, first-in-first-out (FIFO) queue. Een `PriorityQueue` class daarentegen implementeert een `priority-queue`, en is bijvoorbeeld geschikt om schedulers te bouwen die taken op basis van priority moeten uitvoeren.

BlockingQueue Queues kunnen begrensd of onbegrensd zijn en een begrensde queue kan vol raken. Een poging iets toe te voegen aan een volle queue zal mislukken, evenals een poging iets te lezen uit een lege queue. Een blocking-queue voorkomt dit door de toevoegactie te blokkeren zolang de queue vol is en de lees actie te blokkeren zolang de queue leeg is. Blocking-queues zijn in J2SE 5 aanwezig als implementatie van het `BlockingQueue` interface. De onderstaande voorbeeldcode demonstreert het typische gebruik van een blocking-queue. De `put()` operatie in de producer zal blokkeren als er geen ruimte meer is en de `take()` operatie in de consumer zal blokkeren als er niets meer in de queue zit.

```
class Producer implements Runnable {
    private final BlockingQueue queue;
    Producer(BlockingQueue q) { queue = q; }
    public void run() {
        try {
           while(true) { queue.
put(produce()); }
        } catch (InterruptedException ex) {
... handle ...}
    }
    Object produce() { ... }
    }

class Consumer implements Runnable {
    private final BlockingQueue queue;
    Consumer(BlockingQueue q) { queue = q; }
    public void run() {
    try {
            while(true) { consume(queue.
take()); }
         } catch (InterruptedException ex) {
... handle ...}
    }
    void consume(Object x) { ... }
}
```

Slotwoord Het Executor Framework doet voor concurrency wat het Collection Framework deed voor datastructuren. Veel ontwikkelteams van met name serverapplicaties hebben in het verleden, bij gebrek aan een standaardoplossing, steeds hetzelfde wiel uitgevonden en hebben een eigen Thread Framework geschreven. Met het verschijnen van het `java.util.concurrent package` in Java 5 is dit niet meer nodig. Het Executor Framework heeft zijn sporen al in de praktijk bewezen, en is bovendien flexibel te configureren. Natuurlijk moeten ontwikkelaars nog steeds begrijpen wat overwegingen en consequenties zijn bij het schrijven van multithreaded-applicaties. Door het `java.util.concurrent` package worden ze voortaan gevrijwaard zich bezig te houden met de complexe details van de laag-bij-de-grondse Java concurrency-primitieven.

Implementatie	Beschrijving
ArrayBlockingQueue	Een begrensde FIFO blocking-queue geïmplementeerd als array
LinkedBlockingQueue	Een optioneel begrensde FIFO blocking-queue geïmplementeerd als linked list
PriorityBlockingQueue	Een onbegrensde blocking priority queue
SynchronousQueue	Geen echte queue, maar een simpel rendez vous-mechanisme
DelayQueue	Een op tijd gebaseerde scheduling-queue

Tabel 1. Het BlockingQueue interface kent in J2SE 5 een vijftal concrete implementatie-classes

drs. Willem Koppenol is Senior Trainer en Product Specialist Software Development bij Twice IT Training (e-mail: wkoppenol@twice.nl).