Java Threads

- Wat zijn threads?
- Hoe werken threads?
- Hoe werk je met threads in Java?
- Scheduling
- Synchronisatie
  - lock-synchronisatie (met objecten als locks)
  - conditie-synchronisatie

Concurrency in Java met threads

In Java programma's is *concurrency* (aka *parallellisme*) mogelijk.

Concurrency in Java gaat dmv *threads*.

Definitie: *een thread is een executie van een sequentiëel programma*

In een multi-threaded Java programma zijn er meerdere threads, die "tegelijk" aan verschillende taken werken (die parallel uitgevoerd worden)

Voorbeelden concurrency in applicaties

- desktop, waar gebruiker tegelijkertijd meerdere applicaties draait
- webbrowser, die animatie met geluid laat zien & horen, terwijl de gebruiker kan scrollen, forms kan invullen, links kan saven, etc.
- vrijwel elke applicatie met een gecompliceerde gui
- sterker nog: vrijwel elke interactieve applicatie

Vrijwel elke realistische applicatie is multi-threaded.

Waarom concurrency in Java?

Aangezien zoveel applicaties multi-threaded zijn: Java concurrency is essentiële bijdrage aan 'portabiliteit' (de 'write once, run anywhere' filosofie)

Concurrency in bijv. C programma's moet mbv. mechanismen van het onderliggende operating system (eg. POSIX threads op UNIX, Win32API op Windows).

Maar: concurrency in Java wel degelijk beinvloed door onderliggende OS.

Er zijn meer talen met ingebouwde concurrency, bijv. Concurrent Pascal, Ada, C#, ...

Eerste OO taal (Simula) had al concurrency! Dit is geen toeval.

Single-threaded Java

Executie van `o.m(20)` is één thread:

```
m(int i){
 byte x;
 ...
 a.p(i+1);
 ...
 b.q(i*20);
 ...
}
```

Er is een **stack** voor *administratie van uitstaande method calls*, bijv. return adres, waarden van parameters en andere lokale variabelen (eg. `i` en `x`), waarde van `this` oftewel het object waarvan de methode aangeroepen is.

Wortel van de stack: aanroep van `main(..)`

(Terzijde: recursie

```
m(int i){
 ...
 if (i>0) m(i-1);
 ...
}
```

leidt tot meerdere invocaties van dezelfde methode, elke met eigen entry op de stack.)

Multi-threaded Java - het idee

Het idee: parallelle executie van method calls.
Bijv. executie van `o.m(20)` gaat verder terwijl `a.p(21)` en `b.q(60)` worden uitgevoerd

```
m(int i){
 ...
 "nieuwe thread" a.p(i+1);
 ...
 "nieuwe thread" b.q(i*20);
 ...
}
```

Executie van `o.m(20)` resulteert dus in 3 threads.

Elke thread heeft zijn eigen stack, maar de threads delen alle objecten (die op de heap staan)

Dit kan alleen als er niet teveel afhankelijkheden tussen `a.p`, `b.q` en `o.m` zijn; bijv, de methodes `a.p`, `b.q` moeten tenminste `void` zijn.

Echte vs Nep Concurrency

Parallele executie van threads kan nep of echte concurrency zijn:

- **true concurrency**:
  threads draaien op verschillende processoren op multi-processor machine
- **interleaving** aka nep-concurrency:
  threads draaien om de beurt op dezelfde processor

  Altijd als #threads $>$ #processoren

  VM en/of onderliggende OS bepalen welke thread aan de beurt komt: **scheduling**

Nep-concurrency is verreweg het gebruikelijkst (en dit zal voorlopig wel zo blijven, zolang Moore's law prima op blijft gaan voor uni-processor machines).

Elke thread heeft zijn eigen stack, maar alle threads werken met dezelfde heap.

Dwz er is interactie tussen threads door

- competitie voor processor(en)
- toegang tot shared objecten

Dit tweede punt is de wortel van al het kwaad. Meer hierover later.

Waarom zou je meerdere threads willen?

- *Functionaliteit*.

  Handig (of essentieel) voor de applicatie, voor

  – *concurrente taken*
  (bijv. animatie terwijl gebruiker kan scrollen), of
  – *periodieke taken*
  (bijv. backup, of verversen grafische display bij animaties) zijn.

- *Performance*:

  – **snellere executie** door interleaven van trage I/O met rekenwerk
  (Maar: ook overhead voor context switching en synchronisatie)
  (Zelden of nooit: snellere executie dankzij benutten multi-processor)
  – **betere responsetijd**: bijv. door aparte thread voor gebruikersinvoer, of timeout voor onbetrouwbare I/O.

Snellere executie dankzij threads

Threads, en programma's in het algemeen, zijn vaak bezig met wachten op I/O operaties!

Speedup door interleaven van trage I/O operaties met rekenwerk.

Bijvoorbeeld: stel we hebben threads die het volgende doen

```
while(true) {
  // haal webpagina op:
  contents = new URL(...);
  ...
  // analyseer contents
  ...
}
```

waar ophalen webpagina 100 msec duurt, en analyse ervan 10msec.

Welke speedup halen we met 2 ipv 1 thread?

Dit is vrijwel altijd een belangrijkere tijdwinst dan benutten van eventuele multi-processor capaciteit.

Threads in Java

Een thread is een object.

Zo'n object wordt gecreerd met `new` en een constructoraanroep van een klasse. Deze klasse moet

- de klasse Thread extenden, of
- de interface Runnable implementeren.

Een thread-object heeft velden en methodes, kan als parameter meegegeven worden, kan in een array gestopt worden, etc.

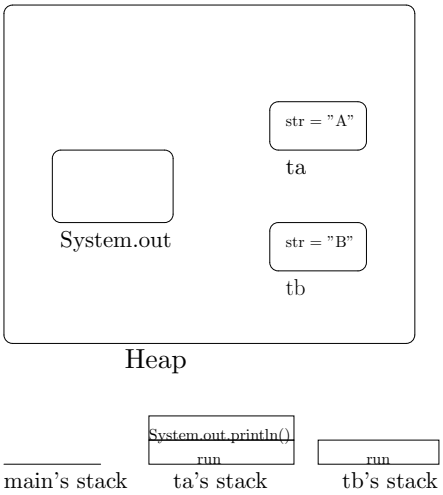
Eeen thread-object heeft een speciale void methode `run()`, die de thread iets te doen geeft (vgl. `main`, maar dan zonder argumenten)

De `run` methode van een thread `t` laten we uitvoeren met een aanroep van `t.start()`.

Threads in Java - subclassing Thread

```
public class PrintThread extends Thread{

  private String str;

  public PrintThread(String str){
    this.str = str;
  }

  public void run(){
    while(true) System.out.print(str);
  }

  public static void main (String[] args){
    PrintThread ta = new PrintThread("A");
    PrintThread tb = new PrintThread("B");
    ta.start();
    tb.start();
  }
}
```


Threads in Java - subclassing Thread



Threads in Java - subclassing Thread

Je kunt `start()` ook in de constructor aanroepen.
(Typisch als laatste statement. Waarom?)

```
public class PrintThread extends Thread{

  private String str;

  public PrintThread(String str){
    this.str = str;
    this.start();
  }

  public void run(){
    while(true) System.out.print(str);
  }

  public static void main (String[] args){
    PrintThread ta = new PrintThread("A");
    PrintThread tb = new PrintThread("B");
  }
}
```


Threads en Processen

Een executie van een Java programma wordt een **proces** genoemd. Een proces bestaat uit één of meer threads.

Een thread eindigt (sterft) als het klaar is met z'n `run` methode.

Een single-threaded Java programma eindigt als het klaar is met z'n `main` methode.

Een multi-threaded Java programma eindigt als

- al z'n threads klaar zijn. (excl. threads die als *deamon threads* aangewezen zijn, met `setDaemon(true)`), of
- als één van z'n threads `System.exit(int)` doet.

Implementing de interface `Runnable`

`Runnable` implementen ipv `Thread` extenden:

```
public class PrintRun implements Runnable{

 private String str;

 public PrintRun(String str){
   this.str = str; }

 public void run(){
   while(true) System.out.print(str); }

 public static void main (String[] args){
   PrintRun runa = new PrintRun("A");
   PrintRun runb = new PrintRun("B");
   Thread ta = new Thread(runa); ta.start();
   Thread tb = new Thread(runb); ta.start();
 }
}
```

NB twee new's nodig voor elke thread.

Implementing de interface `Runnable`

Implementen van de interface `Runnable` is iets ingewikkelder dan extenden van de klasse `Thread`:

2 `new` operaties nodig voor het starten van een thread:
één om een `Runnable` object te maken,
een tweede om hier een `Thread` van te maken.

De enige reden om `Runnable` te implementeren ipv `Thread` te extenden: als je al een andere klasse wilt extenden, bijv.

```
 public MyThread extends SomeSuperClass
                 implements Runnable{
  ....
}
```

Als Java multiple inheritance had, was Runnable niet nodig.

Scheduling

Er is veel vrijheid met scheduling van threads!

- Er wordt een willekeurige thread gekozen die 'runnable' is;
- als deze thread niet verder wil of kan, wordt er een andere 'runnable' thread gekozen,
- én op een willekeurig moment kan de thread onderbroken worden om een andere runnable thread de beurt te geven. Dit heet **preemption**.

  Maar: er is geen garantie dat dit ooit gebeurt.

  Bijv, geen garantie dat thread `tb` ooit aan de beurt kunnen komen, zolang als thread `ta` verder wil.

Scheduling van de threads bepaald door de VM implementatie en het onderliggende OS. Sommige OS'en gaan eerder tot preemption over dan andere.

Scheduling: yield

Met `yield` geeft een runnable thread voorrang aan andere runnable threads.

Bijv. met

```
public class PrintThread extends Thread{

 ...

  public void run(){
    while(true) {
       System.out.println(str);
       yield();  // geef andere thread de beurt
    }
 }

 ...
}
```

zullen beide threads, `ta` en `tb`, aan de beurt komen, en de verdeling van tijd eerlijker zijn.

Scheduling: priorities

Je kunt threads verschillende prioriteiten geven

```
public static void main (String[] args){
  PrintThread ta = new PrintThread("A");
  PrintThread tb = new PrintThread("B");
  ta.setPriority(Thread.MAX_PRIORITY);
  tb.setPriority(ta.getPriority()-10);
  ta.start();
  tb.start();
}
```

Maar verwacht hier niet te veel van! Het is maar de vraag hoeveel effect dit heeft op de scheduling.

Wanneer is thread niet runnable?

Een thread kan in 4 toestanden verkeren

- new thread

  kan enkel gestart worden

- runnable (aka ready)

  de thread kan iets doen

- not runnable (aka blocked)

  de thread kan (tijdelijk) niets doen, want hij

  – slaapt
  – wacht tot een andere thread klaar is
  – wacht op uitvoering van I/O operatie
  – wacht op een lock

- dead

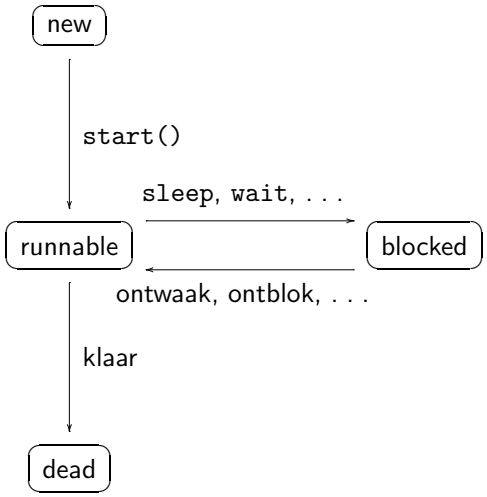
sleep

Met `sleep` laat je een thread slapen; wakker worden gaat gepaard met een `InterruptedException`.

```
public class Clock extends Thread{

 private long time;

 public Clock(){
  this.start();
 }

 public void run(){
  while(true) {
    System.out.println(time);
    try { sleep(100); } // 100 millisec
    catch(InterruptedException e){}; //negeer
    time += period;
  }
 }
}
```

Of gebruik `java.util.Timer(Task)`

Thread toestanden



Ontwaken, ontblokken, . . . gaat met een `InterruptedException`.

join

Met `join` laat je threads op elkaar wachten;

```
// start nieuwe thread t
MyThread t = new MyThread(....);
t.start();

// ga iets anders doen
...

// wacht tot thread t helemaal klaar is
try { t.join();}
catch (InterruptedException e) { }
...
```

Net als bij `sleep`, gaat wakker worden uit `join` gepaard met een `InterruptedException`.

join

Typisch gebruik je `join` voor iets als

```
byte[] buffer = new b[1024]
// start nieuwe thread t, om buffer te vullen
MyThread t = new BufferVuller(b);
t.start();

// ga iets anders doen
...

// wacht tot thread bv helemaal klaar is
try { t.join();}
catch (InterruptedException e) { }

// ga nu data in buffer verwerken
...
```


Synchronisatie

`join` is een voorbeeld van synchronisatie tussen threads.

De interactie van threads door *access aan shared objects* vereist *synchronisatie*, om ongewenste interacties uit te sluiten cq. om gewenste interacties te garanderen.

Het regelen van de synchronisatie is *het* probleem bij concurrency:

- Er kan van alles missgaan als verschillende threads tegelijkertijd aan dezelfde objecten, datastructuren, of I/O devices zitten .
- Dit voorkomen we door synchronisatie tussen de threads toe te voegen
- *Maar* door het toevoegen van synchronisatie kan er ook weer van alles misgaan . . .

Synchronisatie: voorbeeld

```
public class PrintThread extends Thread{

  private String str;

  public PrintThread(String str){
    this.str = str;
  }

  public void run(){
    while(true) System.out.println(str);
  }

  public static void main (String[] args){
    PrintThread ta = new PrintThread("abcd");
    PrintThread tb = new PrintThread("1234");
    ta.start();
    tb.start();
  }
}
```

Kan dit ooit de uitvoer `....ab123cd4...` produceren?

Synchronisatie: voorbeeld

Als `System.out.println` geimplementeerd is als

```
public void println(String str){

 for (int i=0; i<str.length();i++)
      System.out.print(str.charAt(i));
 System.out.println();

}
```

dan kan de uitvoer `....ab123cd4...` voorkomen

namelijk als thread ta halverwege zijn `System.out.println(str)` ge-preempt wordt.

Synchronisatie: voorbeeld

Als `System.out.println` geimplementeerd is als

```
public void synchronized println(String str){

 for (int i=0; i<str.length();i++)
      System.out.print(str.charAt(i));
 System.out.println();
}
```

dan kan de uitvoer `....ab123cd4...` *niet* voorkomen.

Een **synchronized** methode `m` van een object `o` kan niet onderbroken worden door een andere `synchronized` methode op hetzelfde object.

Dit wordt ook wel *mutual exclusion* genoemd.

Synchronized methodes

- Ieder Java object heeft een synchronisatie-lock.
- Methodes kunnen als `synchronized` gedeclareerd worden.
- Een thread kan alleen een `synchronized` methode uitvoeren als het de lock voor het betreffende object heeft. Zoniet, dan *blockt* de thread tot dat lock vrijkomt.

Dwz voor `synchronized` method calls op hetzelfde object hebben we *mutual exclusion*.

NB andere threads kunnen wel niet-`synchronized` methodes uitvoeren op `o` terwijl een thread bezig is met een `synchronized` methode op `o`.

Synchronized methodes

Een thread houdt het lock als in een `synchronized` methode willekeurige andere methodes worden aangeroepen.

Ihb. een `synchronized` methode kan dus andere `synchronized` methodes op hetzelfde object aanroepen.

Elke thread houdt een *verzameling* locks, die je kunt bepalen gegeven de stack van die thread. (Hoe?)

Hierdoor wordt deadlock mogelijk! (Waarom?)

Synchronized statische methodes

Synchronized voor statische methodes werkt ongeveer hetzelfde als voor instance methodes:

- Net als elk object, heeft ook elke *klasse* een synchronisatie lock.
- Een thread kan alleen een `synchronized` static methode uitvoeren als het het de lock voor betreffende klasse heeft.

Dus voor de `synchronized` statische methodes van een klasse hebben we mutual exclusion.

Terminologie

Nog wat terminologie:

- Een object met `synchronized` methodes ook wel een **monitor** genoemd.
- Een methode – of, algemener, een code fragment – waarvoor we mutual exclusion hebben wordt wel een **kritische sectie** (critical section) genoemd.
- Als een method call – of, algemener, een code fragment – nooit onderbroken (pre-empted) kan worden, noemen we het een **atomaire** operatie.

  Atomiciteit is een sterkere eigenschap dan mutual exclusion: een `synchronized` methode kan wel degelijk onderbroken worden, alleen niet door aanroepen van andere `synchronized` methodes op hetzelfde object.

Synchronized methodes: voorbeeld

Welke methodes moeten hier `synchronized` zijn?

```
public class Log{

  private Vector log;
  private int size = 0;

  int getSize() { return size; }

  String getLastEntry() {
    return (String)log.elementAt(size);
  }

  void addEntry(String s){
    log.addElement(s);
    size++;
  }
}
```


Synchronized methodes: voorbeeld

Vaak – maar niet altijd – moeten set-methodes `synchronized` zijn, en hoeven get-methodes dat niet te zijn.

```
public class Log{

  private Vector log;
  private int size = 0;

  int getSize() { return size };

  String getLastEntry() {
    return (String)log.elementAt(size);
  };

  void synchronized addEntry(String s){
    log.addElement(s);
    size++;
  }
}
```

Subtiel punt: volgorde van operaties in `addEntry`!

Gevaren van synchronisatie

Het gebruik van `synchronized` is niet zonder gevaren:

- **Deadlock**
  Een programma kan vastlopen doordat threads om elkaar wachten

- **Starvation**
  Een enkele thread kan nooit meer aan de beurt komen omdat de vereiste locks nooit kan bemachtigen.

Alternatieven voor synchronisatie

Er zijn meerdere manieren om ongewenste interacties van threads uit te sluiten:

1. Eén manier is het gebruik van `synchronized`.

2. Een andere, betere, manier is ervoor te zorgen dat threads geen objecten delen, zodat synchronisatie onnodig wordt.

   Bijv. laat elke `PrintThread` *z'n uitvoer naar een ander window sturen, of geef elke thread z'n eigen* `Log`.

   Indien mogelijk, heeft dit de voorkeur. Voorkomen is beter dan genezen!

3. De twee technieken hierboven kunnen niet voldoende zijn: naast het gebruik van `synchronized`, moeten moeten er dan extra, fijnere synchronisaties toegevoegd worden

Fijnere synchronisatie

Fijnere synchronisatie dan `synchronized` van methodes is mogelijk door een blok code te synchronizen met de lock van een willekeurig object Bijv.

```
class PrintThread extends Thread{

  private String str;

  public void print(){
    synchronized(System.out){
      System.out.print("De uitvoer van thread");
      System.out.print(this.getName());
      System.out.print(" : ")
      System.out.println(str);
  } }
```

voorkomt dat uitvoer van verschillende PrintThread's door elkaar komt.

(Mooi is anders: liever laat je alle synchronisatie door het object `System.out` zelf regelen.)

Fijnere synchronisatie

Je kunt ook dummy objecten gebruiken als locks. Deze dummy objecten representeren het recht om een bepaalde resource (bijv. een I/O device) te gebruiken. Bijv.

```
final static Object resource1 = new Object();
final static Object resource2 = new Object();

class PrintThread extends Thread{

 public void a(){
  synchronized(resource1){
    synchronized(resource2){
    // beide resources geclaimd
    ...
 } } }
```

In een mooi OO ontwerp wil je zulke dummy objecten vermijden, en synchronisatie liever regelen dmv. synchronized methodes op objecten!

Fijnere synchronisatie : conditie-synchronisatie

Soms wil je een thread laten wachten tot een bepaalde conditie geldt, bijvoorbeeld tot een queue niet meer leeg is:

```
public class Queue {
  Linkedlist q = new LinkedList();

  public synchronized? Object pop(){
    while (q.size() == 0) {
       "wacht tot de q niet leeg is";
    }
    return q.remove(0);
  }

  public synchronized? push (Object x){
    q.add(x);
  }
}
```

Hoe doe je dit? Dit is meer dan mutual exclusion, en met een simpele lock lukt 't niet.

Conditie-synchronisatie

Bij elk object hoort behalve een lock ook een **wachtrij met threads**.

Met `o.wait()` gaat een thread in de wachtrij van object o staan, en geeft het lock van het object vrij (als de thread dat block had).

Met `o.notifyAll()` maak je alle threads in de wachtrij van o wakker. Deze threads gaan verder, maar moeten wel het het lock van o weer krijgen als ze met een `synchronized` methode verder willen.

Wakker worden gaat weer gepaard met een `InterruptedException`.

Conditie-synchronisatie: `wait` en `notifyAll`

```
public class Queue {
  Linkedlist q = new LinkedList();

  public synchronized Object pop(){
    while (q.size() == 0) {
       try {this.wait();}
       catch (InterruptedException e) {}
    }
    return q.remove(0);
  }

  public synchronized push (Object x){
    q.add(x);
    this.notifyAll();
  }
}
```


Conditie-synchronisatie: `while` ipv `if`!

De standaard-manier om op een conditie te synchroniseren is met

```
public synchronized Object pop(){
  while ( conditie ) {
     try {this.wait();}
     catch (InterruptedException e) {}
  ...}
```

en *niet* met

```
public synchronized Object pop(){
  if ( conditie ) {
     try {this.wait();}
     catch (InterruptedException e) {}
  ...}
```

Gebruik van `if` is meestal incorrect!

Conditie-synchronisatie: `while` ipv `if`!

Waarom gaat dit programma mis?

```
public class QueueFout {
  Linkedlist q = new LinkedList();

  public synchronized Object pop(){
    if (q.size() == 0) {
       try {this.wait();}
       catch (InterruptedException e) {}
    }
    return q.remove(0);
  }

  public synchronized push (Object x){
    q.add(x);
    this.notifyAll();
  }
}
```

Merk op: programma's met `wait` en `notifyAll` zijn lastig.

`notifyAll()` vs `notify()`

Met `o.notifyAll()` maak je alle threads in de wachtrij van object o wakker.

Met `o.notify()` maak je één thread in de wachtrij van object o wakker.

Het gebruik van `notify` is link omdat het risico van starvation van threads erg is.

`QueueFout` is ook incorrect als je de `notifyAll` door `notify` vervangt.

`wait(timeout)`

Ipv op `notify(All)` te vertrouwen om threads te wekken, kan `wait` ook een time-out meekrijgen:

```
public synchronized Object pop(){
  while ( conditie ) {
     try {this.wait( timeoutInMillisecs );}
     catch (InterruptedException e) {}
  ...}
```

Dit is inefficienter maar robuuster dan op `notify(All)` te vertrouwen.

Concurrency is moeilijk!

Concurrency handig/essentiëel, maar moeilijk!

Mogelijke bugs door concurrency:

- rare interacties tussen threads door onvoldoende synchronisatie (aka. data races)
- deadlock
- starvation

Minimaliseer bij je ontwerp het gebruik van shared objects, en minimaliseer dus de noodzaak voor synchronisatie.

Multi-threaded programma's gedragen zich *non-deterministisch*, want gedrag afhankelijk van allerlei omgevingsfactoren (scheduling, snelheid I/O, user-interacties)

Dit non-determinisme maakt debuggen moeilijk!

Merk op: bij C(++) zijn single-threaded programma's al non-determistisch, bijv. door pointer-manipulaties en ontbreken default-initialisatie!

Shared variables

Interactie tussen threads is niet alleen mogelijk door – al dan niet `synchronized` – methodes van shared objecten aan te roepen.

Threads kunnen ook direct aan shared velden komen.

Normaliter wil je dit zoveel mogelijk vermijden!

Shared variables

```
public class Weird{

 int x,y,i,j;

 class Thread1 extends Thread {
  public void run(){ x=1; j=y;}
 }

 class Thread2 extends Thread {
  public void run(){ y=1; i=x;}
 }

 public void main (String[] args){
  new Thread1().start();
  new Thread2().start();
 }
}
```

Kan na afloop `i=0` én `j=0` zijn?

Shared variables

Antwoord: JA!

Want

- compiler kan volgorde van statements veranderen
- compiler kan waarden in registers opslaan
- op multiprocessorsysteem kunnen threads tijdelijk een lokale kopie van globale variabelen bijhouden. Deze lokale kopieën kunnen tijdelijk verschillen van de waarden in het globale geheugen; het globale geheugen loopt soms achter, omdat writes gebufferd worden.

`volatile` velden

De modifier `volatile` voor veld waarschuwt dat verschillende threads er tegelijkertijd aan kunnen zitten:

```
public class Weird{

 volatile int x,y;
 int i,j;

 class Thread1 extends Thread {
  public void run(){ x=1; j=y;}
 }

 class Thread2 extends Thread {
  public void run(){ y=1; i=x;}
 }

 ...
}
```

maar *vermijden van dit soort shared velden is veel beter!*

Java Memory Model

Exacte semantiek van threads, `synchronized`, `volatile` etc. is erg ingewikkeld (vooral voor slecht-gesynchroniseerde code).

Hoofdstuk 17 van de Java Language Spec (H 8 van de Java Virtual Machine Spec) is *onbegrijpelijk* en *incorrect*. Geen enkele implementatie houdt zich aan deze spec.

Actief onderwerp van onderzoek & discussie over de afgelopen jaren.

Factoren van invloed: niet alleen implementatie van de VM en onderliggende hardware, maar ook compiler-optimalisaties.

Verbeterde definitie, JSR-133, 'Java Memory Model and Thread Specification Revision', goedgekeurd 15 september 2003.

Real-time Java

Java biedt weliswaar concurrency, maar is ongeschikt voor echte real-time applicaties:

- scheduling erg vrijblijvend
- wachtende threads niet noodzakelijk FIFO
- hierdoor: timing erg onvoorspelbaar

Er is een standaard voor real-time Java ontwikkeld, JSR-1, zie `www.rtj.org`.

Trend: hogere programmeertalen (met bijv. garbage collection) in steeds meer devices & toepassingen.