Testen nebenläufiger Objekte

Threads in Java

Julian Lambertz

Seminar „Tests in Informatik und Statistik“ im SS 2004
Universität Ulm

Themenüberblick

- Einleitung
  - Begriff der Nebenläufigkeit
  - Was sind Threads?

Themenüberblick

- Einleitung
  - ◇ Begriff der Nebenläufigkeit
  - ◇ Was sind Threads?
- Threads in Java

Themenüberblick

- Einleitung
  - ◇ Begriff der Nebenläufigkeit
  - ◇ Was sind Threads?
- Threads in Java
- Probleme bei der Verwendung von Threads
  - ◇ Gefahren + Lösungsmöglichkeiten
  - ◇ Hauptproblem beim Testen

Themenüberblick

- Einleitung
  - Begriff der Nebenläufigkeit
  - Was sind Threads?
- Threads in Java
- Probleme bei der Verwendung von Threads
  - Gefahren + Lösungsmöglichkeiten
  - Hauptproblem beim Testen
- Tests mit JUnit
  - mögliche Tests
  - Beispiel

Themenüberblick

- Einleitung
  - Begriff der Nebenläufigkeit
  - Was sind Threads?
- Threads in Java
- Probleme bei der Verwendung von Threads
  - Gefahren + Lösungsmöglichkeiten
  - Hauptproblem beim Testen
- Tests mit JUnit
  - mögliche Tests
  - Beispiel
- Fazit

Einleitung
Nebenläufigkeit

Das Prinzip der Nebenläufigkeit beruht auf der Vorstellung, dass Dinge gleichzeitig passieren.

Das kann bedeuten:

- **virtuell** auf einem Computer mit einem Prozessor

Einleitung
Nebenläufigkeit

Das Prinzip der Nebenläufigkeit beruht auf der Vorstellung, dass Dinge gleichzeitig passieren.

Das kann bedeuten:

- **virtuell** auf einem Computer mit einem Prozessor
- **real** auf Mehrprozessorsystemen oder bei verteilten Anwendungen

Einleitung

Was sind Threads?

Man unterscheidet zwischen Prozessen und Threads:

Einleitung
Was sind Threads?

Man unterscheidet zwischen Prozessen und Threads:

- Ein **Prozess** bezeichnet ein im Speicher befindliches ablauffähiges Programm mit seinen dafür notwendigen betriebsystemseitigen Datenstrukturen (z.B. Stack- und Programmzähler, Prozesszustand...).

Einleitung
Was sind Threads?

Man unterscheidet zwischen Prozessen und Threads:

- Ein **Prozess** bezeichnet ein im Speicher befindliches ablauffähiges Programm mit seinen dafür notwendigen betriebsystemseitigen Datenstrukturen (z.B. Stack- und Programmzähler, Prozesszustand...).
- Ein **Thread** (=Faden) ist eine nebenläufige Ausführungseinheit innerhalb eines Prozesses.

Eigenschaften von Threads:

Eigenschaften von Threads:

- ein gemeinsamer Adressraum innerhalb eines Prozesses
- jeweils eigener Programmzähler
- mehrere Threads können parallel ausgeführt werden
- ein Thread hat eine Priorität

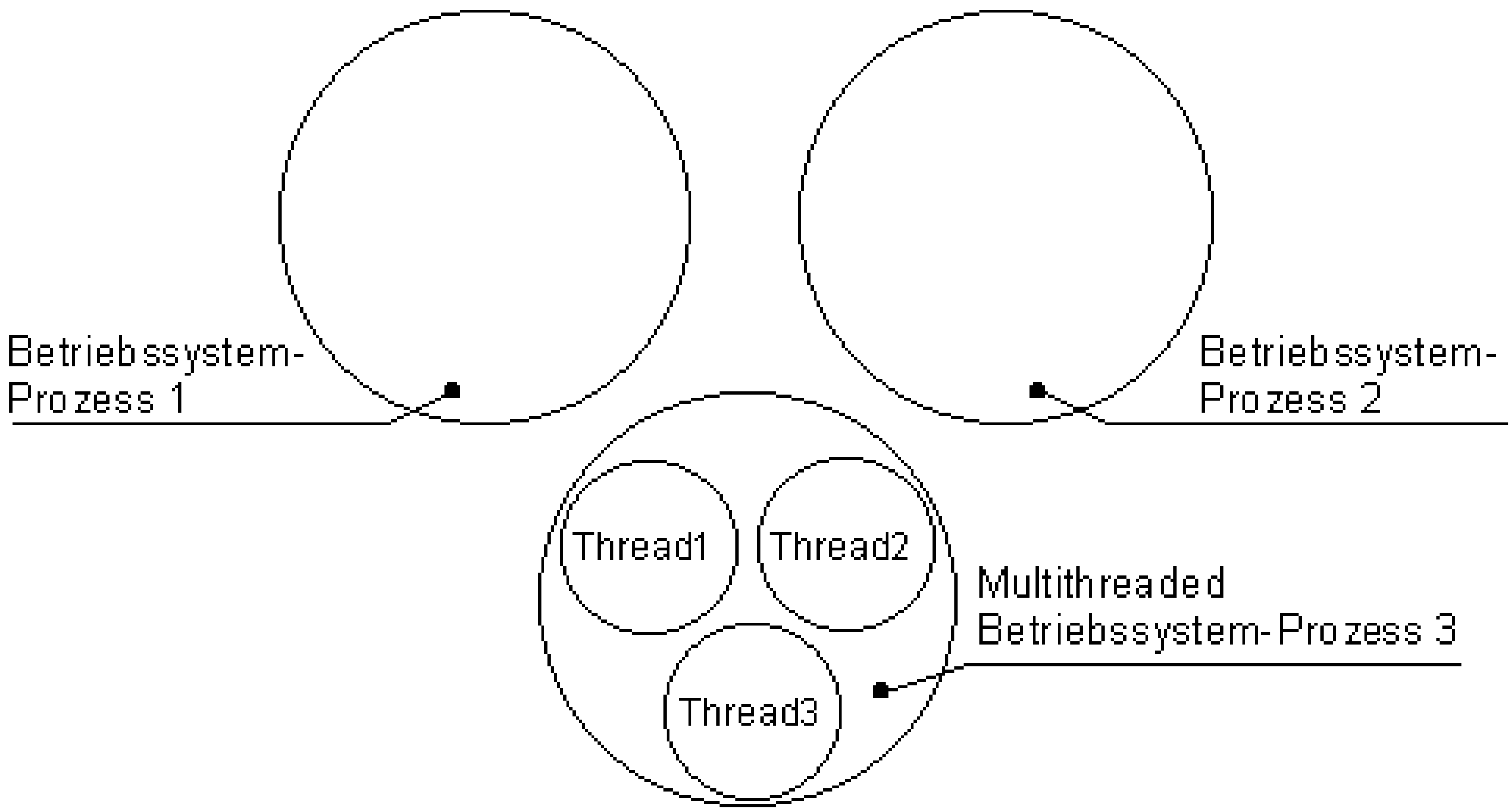
Einleitung
Was sind Threads?

Eigenschaften von Threads:

- ein gemeinsamer Adressraum innerhalb eines Prozesses
- jeweils eigener Programmzähler
- mehrere Threads können parallel ausgeführt werden
- ein Thread hat eine Priorität

$\longrightarrow$ ein Thread stellt die kleinste Einheit für die Zuteilung von Rechenzeit dar

Einleitung
Was sind Threads?



Beim Starten eines Java-Programms passiert folgendes:

Threads in Java
Ablauf beim Start eines Programms

Beim Starten eines Java-Programms passiert folgendes:

1. das Betriebssystem erzeugt einen Prozess: Java Virtual Machine (=JVM)
2. die JVM started den Thread *main*, der die *main*-Methode startet
3. ggf. werden im Programmverlauf weiter Threads gestartet

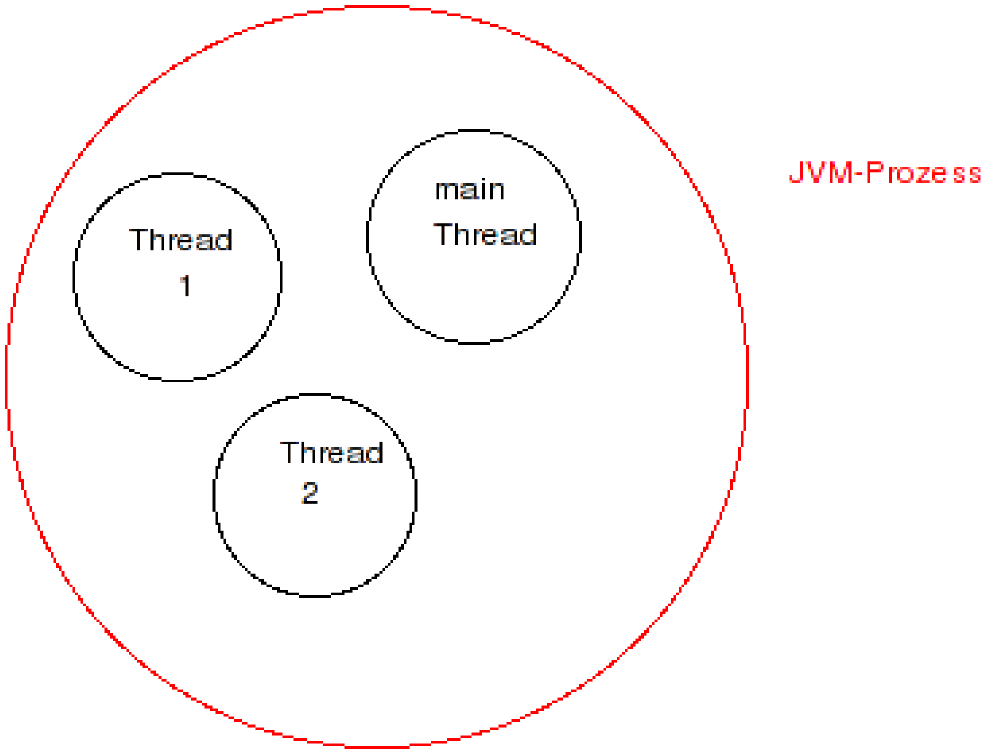
Threads in Java
Ablauf beim Start eines Programms

Beim Starten eines Java-Programms passiert folgendes:

1. das Betriebssystem erzeugt einen Prozess: Java Virtual Machine (=JVM)
2. die JVM started den Thread *main*, der die *main*-Methode startet
3. ggf. werden im Programmverlauf weiter Threads gestartet

Anmerkung: Durch Drücken von “Strg“+“\“ zur Laufzeit werden Informationen über die aktuell laufenden Threads der JVM angezeigt.

Threads in Java



Threads in Java
Scheduling

- eine JVM bildet die Threads normalerweise auf Betriebssystem-Threads ab
- hiervon hängt das Scheduling ab, d.h.:

Threads in Java
Scheduling

- eine JVM bildet die Threads normalerweise auf Betriebssystem-Threads ab
- hiervon hängt das Scheduling ab, d.h.:
  - welcher Thread wann und wieviel Rechenzeit bekommt

Threads in Java
Scheduling

- eine JVM bildet die Threads normalerweise auf Betriebssystem-Threads ab
- hiervon hängt das Scheduling ab, d.h.:
  - welcher Thread wann und wieviel Rechenzeit bekommt
  - wie (und ob überhaupt!) die Priorität berücksichtigt wird

Threads in Java
Scheduling

- eine JVM bildet die Threads normalerweise auf Betriebssystem-Threads ab
- hiervon hängt das Scheduling ab, d.h.:
  - welcher Thread wann und wieviel Rechenzeit bekommt
  - wie (und ob überhaupt!) die Priorität berücksichtigt wird

die Spezifikation der JVM ist in dieser Hinsicht ungenau
$\Rightarrow$ konkrete JVMs können sehr unterschiedlich implementiert sein

Eigene Threads erzeugen

In Java ist es leicht möglich, neue Threads zu erzeugen und zu starten:

Threads in Java

Eigene Threads erzeugen

In Java ist es leicht möglich, neue Threads zu erzeugen und zu starten:

1. - man erbt von der Klasse *Thread*
   - überschreibt die Methode *run()*
   - und startet die Instanz des Threads mit *start()*

Threads in Java
Eigene Threads erzeugen

In Java ist es leicht möglich, neue Threads zu erzeugen und zu starten:

1. - man erbt von der Klasse *Thread*
   - überschreibt die Methode *run()*
   - und startet die Instanz des Threads mit *start()*

   oder

2. - implementiert in der gewünschten Klasse XY das Interface *Runnable*
   - implementiert die Methode *run()*
   - erzeugt eine Thread-Instanz mit *new Thread(xyInstanz)*
   - und startet diese ebenfalls mit *start()*

Threads in Java

Beispiel 1: Erben von Thread

```
class MyThread extends Thread{

        public void run(){
            System.out.println(this);
        }
}
-----------------------------------------------
public static void main(String[] args){

        MyThread t1=new MyThread();
        MyThread t2=new MyThread();
        t1.start();
        t2.start();
}
```

Threads in Java

Beispiel 2: Runnable implementieren

```
class MyThread2 implements Runnable{

        public void run(){
            System.out.println(Thread.currentThread());
        }
}
------------------------------------------------
public static void main(String[] args){

        Thread t1=new Thread(new MyThread2());
        Thread t2=new Thread(new MyThread2());
        t1.start();
        t2.start();
}
```

Threads in Java
versteckte Threads

WICHTIG !
in Java können versteckt mehrere Threads gestartet werden:

- Eventbehandlung in AWT in eigenem Thread
- Java Servlets
- ...

Fehler bei der Verwendung von Threads

Die Verwendung von Threads bringt immer dann besondere Gefahren mit sich, wenn diese miteinander interagieren.

Fehler bei der Verwendung von Threads

Sie rufen wechselseitig Methoden auf oder schreiben/lesen wechselseitig Werte.

Beispiel:

int x=2;	
Thread1	**Thread2**
if(x<3)	if(x<3)
x++;	x++;

Fehler bei der Verwendung von Threads

Sie rufen wechselseitig Methoden auf oder schreiben/lesen wechselseitig Werte.

Beispiel:

int x=2;	
Thread1	**Thread2**
if(x<3)	if(x<3)
x++;	x++;

Fehler bei der Verwendung von Threads

Sie rufen wechselseitig Methoden auf oder schreiben/lesen wechselseitig Werte.

Beispiel:

int x=2;	
Thread1	**Thread2**
if(x<3)	if(x<3)
x++;	x++;

Fehler bei der Verwendung von Threads

Sie rufen wechselseitig Methoden auf oder schreiben/lesen wechselseitig Werte.

Beispiel:

int x=2;	
Thread1	**Thread2**
if(x<3)	if(x<3)
x++;	x++;

Fehler bei der Verwendung von Threads

Sie rufen wechselseitig Methoden auf oder schreiben/lesen wechselseitig Werte.

Beispiel:

int x=2;	
Thread1	**Thread2**
if(x<3)	if(x<3)
x++;	x++;

Fehler bei der Verwendung von Threads

Sie rufen wechselseitig Methoden auf oder schreiben/lesen wechselseitig Werte.

Beispiel:

int x=2;	
Thread1	**Thread2**
if(x<3)	if(x<3)
x++;	x++;

Fehler bei der Verwendung von Threads

Sie rufen wechselseitig Methoden auf oder schreiben/lesen wechselseitig Werte.

Beispiel:

int x=2;	
Thread1	**Thread2**
if(x<3)	if(x<3)
x++;	x++;

Somit ist x gleich 4 !

Fehler bei der Verwendung von Threads
synchronized 1

Lösung: Schlüsselwort ***synchronized***

```
class MyThread extends Thread{

         // ...

         public void foo(Ressource r){

              synchronized(r){
                   if(r.x<3)
                     r.x++;
              }
         }
}

class Ressource{ public int x; }
```

Fehler bei der Verwendung von Threads
synchronized 2

- solange ein Thread sich bei der Ausführung innerhalb eines *synchronized*-Blocks befindet, ist die betreffende Ressource für andere Threads gesperrt (gegenseitiger Aussschluss)

Fehler bei der Verwendung von Threads
synchronized 2

- solange ein Thread sich bei der Ausführung innerhalb eines *synchronized*-Blocks befindet, ist die betreffende Ressource für andere Threads gesperrt (gegenseitiger Aussschluss)
- ein *synchronized*-Block kann innerhalb eines anderen Code-Blocks verwendet werden
  $\longrightarrow$ die betreffende Ressource ist gesperrt

Fehler bei der Verwendung von Threads
synchronized 2

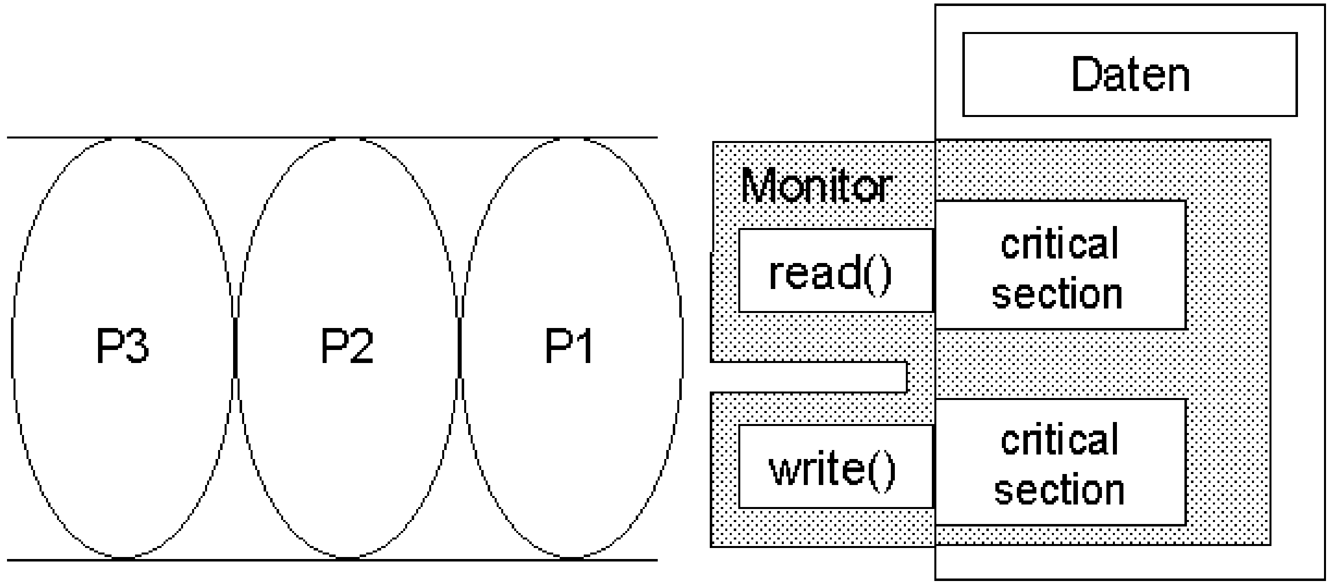
- solange ein Thread sich bei der Ausführung innerhalb eines *synchronized*-Blocks befindet, ist die betreffende Ressource für andere Threads gesperrt (gegenseitiger Aussschluss)
- ein *synchronized*-Block kann innerhalb eines anderen Code-Blocks verwendet werden
  $\longrightarrow$ die betreffende Ressource ist gesperrt
- Methoden können *synchronized* deklariert werden
  $\longrightarrow$ das ganze Objekt ist gesperrt

Fehler bei der Verwendung von Threads
synchronized 2

- solange ein Thread sich bei der Ausführung innerhalb eines *synchronized*-Blocks befindet, ist die betreffende Ressource für andere Threads gesperrt (gegenseitiger Aussschluss)
- ein *synchronized*-Block kann innerhalb eines anderen Code-Blocks verwendet werden
  $\longrightarrow$ die betreffende Ressource ist gesperrt
- Methoden können *synchronized* deklariert werden
  $\longrightarrow$ das ganze Objekt ist gesperrt
- *synchronized*-Methoden/-Blöcke können ineinander geschachtelt werden bzw. einander aufrufen

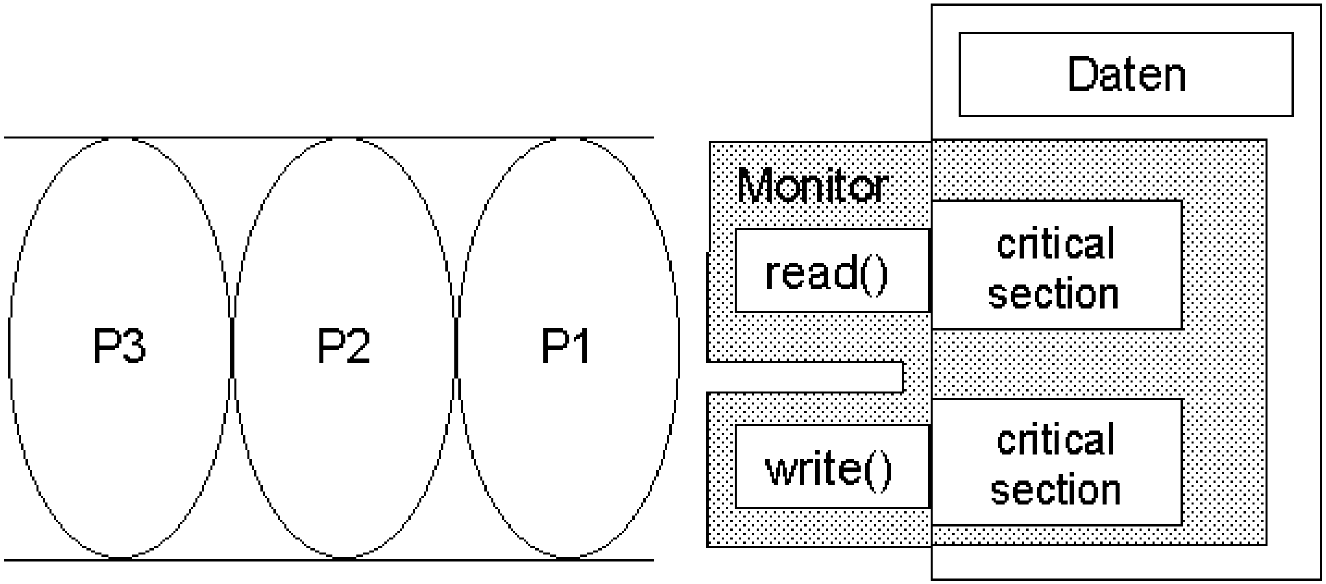
Fehler bei der Verwendung von Threads
synchronized 3

Graphisch dargestellt:



Fehler bei der Verwendung von Threads
synchronized 3

Graphisch dargestellt:



Anmerkung: Wenn mehrere Threads eine Ressource benötigen, ist die Verwendung der Methoden *wait()*, *notify()*, *notifyAll()* zu empfehlen um das Warten/die Benachrichtigung bzgl. gesperrten Ressourcen zu regeln (s. Dokumentation, Klasse Object).

Fehler bei der Verwendung von Threads
Deadlock

- aus den genannten Synchronisationsmechanismen resultiert ein weiterer möglicher Fehler, ein sog. Deadlock
- ein Deadlock tritt auf, wenn ein oder mehrere Threads unbeabsichtigt blockiert werden
  Ursachen sind:

Fehler bei der Verwendung von Threads
Deadlock

- aus den genannten Synchronisationsmechanismen resultiert ein weiterer möglicher Fehler, ein sog. Deadlock
- ein Deadlock tritt auf, wenn ein oder mehrere Threads unbeabsichtigt blockiert werden
  Ursachen sind:
  - mehrere Threads warten auf Ressourcen, die jeweils ein anderer gesperrt hat

Fehler bei der Verwendung von Threads
Deadlock

- aus den genannten Synchronisationsmechanismen resultiert ein weiterer möglicher Fehler, ein sog. Deadlock
- ein Deadlock tritt auf, wenn ein oder mehrere Threads unbeabsichtigt blockiert werden
  Ursachen sind:
  - mehrere Threads warten auf Ressourcen, die jeweils ein anderer gesperrt hat
  - irregulär beendete Threads

Testen nebenläufiger Anwendungen
Nichtdeterminismus

- das Hauptproblem beim Testen ist der **Nichtdeterminismus**
- das Verhalten eines Java-Programms mit mehreren Threads ist abhängig von:

Testen nebenläufiger Anwendungen
Nichtdeterminismus

- das Hauptproblem beim Testen ist der **Nichtdeterminismus**
- das Verhalten eines Java-Programms mit mehreren Threads ist abhängig von:
  - dem Scheduling (Betriebssystem/JVM)

Testen nebenläufiger Anwendungen

Nichtdeterminismus

- das Hauptproblem beim Testen ist der **Nichtdeterminismus**
- das Verhalten eines Java-Programms mit mehreren Threads ist abhängig von:
  - dem Scheduling (Betriebssystem/JVM)
  - von der Leistung des verwendeten Systems

Testen nebenläufiger Anwendungen
Nichtdeterminismus

- das Hauptproblem beim Testen ist der **Nichtdeterminismus**
- das Verhalten eines Java-Programms mit mehreren Threads ist abhängig von:
  - ◇ dem Scheduling (Betriebssystem/JVM)
  - ◇ von der Leistung des verwendeten Systems
  - ◇ u.U. auch von der momentanen Systemlast

Testen nebenläufiger Anwendungen
Nichtdeterminismus

- das Hauptproblem beim Testen ist der **Nichtdeterminismus**
- das Verhalten eines Java-Programms mit mehreren Threads ist abhängig von:
  - dem Scheduling (Betriebssystem/JVM)
  - von der Leistung des verwendeten Systems
  - u.U. auch von der momentanen Systemlast
  - dem verwendeten Compiler

Testen nebenläufiger Anwendungen
Nichtdeterminismus

- das Hauptproblem beim Testen ist der **Nichtdeterminismus**
- das Verhalten eines Java-Programms mit mehreren Threads ist abhängig von:
  - ◇ dem Scheduling (Betriebssystem/JVM)
  - ◇ von der Leistung des verwendeten Systems
  - ◇ u.U. auch von der momentanen Systemlast
  - ◇ dem verwendeten Compiler

→ Tests sind nicht einfach wiederholbar !

Testen nebenläufiger Anwendungen
Nichtdeterminismus

- das Hauptproblem beim Testen ist der **Nichtdeterminismus**
- das Verhalten eines Java-Programms mit mehreren Threads ist abhängig von:
  - dem Scheduling (Betriebssystem/JVM)
  - von der Leistung des verwendeten Systems
  - u.U. auch von der momentanen Systemlast
  - dem verwendeten Compiler

→ Tests sind nicht einfach wiederholbar !
→ ein Fehler tritt bei gleichen Eingaben vielleicht nur sehr selten auf !

JUnit

JUnit kann bedingt zum Testen nebenläufiger Anwendungen benutzt werden:

JUnit

JUnit kann bedingt zum Testen nebenläufiger Anwendungen benutzt werden:

- zum wiederholten Testen: die Klasse *RepeatedTest*

JUnit

JUnit kann bedingt zum Testen nebenläufiger Anwendungen benutzt werden:

- zum wiederholten Testen: die Klasse *RepeatedTest*
- zum Überprüfen bestimmter Annahmen über den Zustand von Threads

JUnit

JUnit kann bedingt zum Testen nebenläufiger Anwendungen benutzt werden:

- zum wiederholten Testen: die Klasse *RepeatedTest*
- zum Überprüfen bestimmter Annahmen über den Zustand von Threads

Anmerkung: Unter http://www.dpunkt.de/utmj können einige Erweiterungen und Beispiele zum Testen von Threads heruntergeladen werden.

JUnit

Beispiel:

```
public static Test suite(){

       TestSuite suite = new TestSuite(TestThreads.class);
       return new junit.extensions.RepeatedTest(suite, 300);
}
public void testDeadlock(){

        t.startT1andT2();
        Assert.assertFalse("deadlock ",
                ( (t.getT1Wants()==t.getT2Has())
                &&(t.getT2Wants()==t.getT1Has()) ));
 }
```

Fazit

Fazit

- die Programmierung mit mehreren Threads bringt neue Fehlerquellen mit sich

Fazit

- die Programmierung mit mehreren Threads bringt neue Fehlerquellen mit sich
- die Fehler sind oftmals schlecht zu finden und kaum reproduzierbar (Nichtdeterminismus)

Fazit

- die Programmierung mit mehreren Threads bringt neue Fehlerquellen mit sich
- die Fehler sind oftmals schlecht zu finden und kaum reproduzierbar (Nichtdeterminismus)
- deshalb kann man sich noch weniger auf die Ergebnisse von Tests verlassen als sonst

Fazit

- die Programmierung mit mehreren Threads bringt neue Fehlerquellen mit sich
- die Fehler sind oftmals schlecht zu finden und kaum reproduzierbar (Nichtdeterminismus)
- deshalb kann man sich noch weniger auf die Ergebnisse von Tests verlassen als sonst
- JUnit kann hier nur bedingt helfen

Fazit

- die Programmierung mit mehreren Threads bringt neue Fehlerquellen mit sich
- die Fehler sind oftmals schlecht zu finden und kaum reproduzierbar (Nichtdeterminismus)
- deshalb kann man sich noch weniger auf die Ergebnisse von Tests verlassen als sonst
- JUnit kann hier nur bedingt helfen

$\longrightarrow$ ein gutes Konzept und eine klare Trennng des nebenläufigen Codes vom Rest ist erforderlich!

Literatur

1. Maurice Schoenmakers, Nebenläufigkeit in Java, 2001 http://wwwspies.informatik.tu-muenchen.de/lehre/vorlesung/WS0001/info3/threads.pdf
2. J. Link, Unit Tests mit Java, dpunkt.verlag, 2002.
3. Goll, Weiß, Rothländer, http://www.it.fht-esslingen.de/∼heinisch/javabuch/, Teubner Verlag, 2001
4. Java 2 Platform, Standard Edition, v 1.4.1 API Specification, Sun Microsystems, Inc., 2002, http://java.sun.com
5. The Java Virtual Machine Specification, Sun Microsystems, Inc., 1999, http://java.sun.com