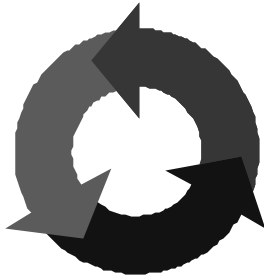


# Sprachkonstrukte in Java



1

## Übersicht Threads in Java

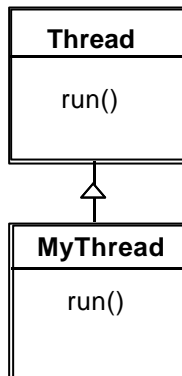
---

- ◆ Threads
  - durch Vererbung aus der Thread Class
  - durch ein Runnable Interface
- ◆ Lebenszyklus:
  - Start: Initiieren
  - Run: Ablaufen lassen
  - Stop: Terminieren -> durch Terminierung von Run()
- ◆ Thread Scheduling:
  - yield: freiwillige Aufgabe des Prozessors
  - getPriority, setPriority: Prioritäten zur programmgesteuerten Auswahl eines Threads
- ◆ Wechselseitiger Ausschluß:
  - Synchronized: Lock Mechanismus je Objekt
  - Wait: Warten auf Benachrichtigung
  - Notify, NotifyAll: Versenden einer Benachrichtigung

2

## Threads in Java

Die Thread Class implementiert einen einzelnen sequentiellen Thread. Thread Objekte können zur Laufzeit dynamisch instantiiert und beendet werden.



Die Thread Class führt Instruktionen der Methode `run()` aus. Eine Möglichkeit zur Implementierung eines Threads besteht durch Spezialisierung der Thread Class und Überlagerung der `run()` Methode.

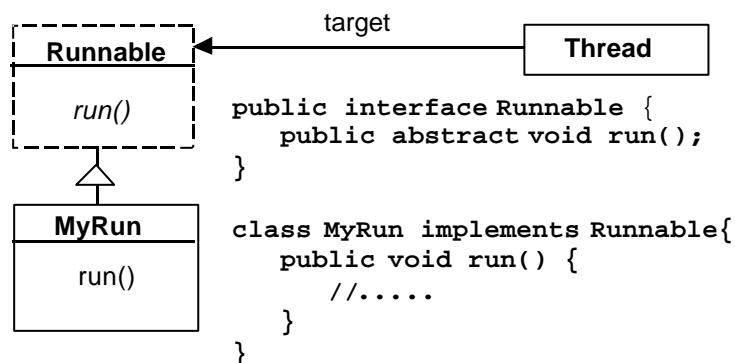
```
class MyThread extends Thread {
    public void run() {
        //.....
    }
}
```

Diese Variante ist nicht sinnvoll: warum ?

3

## Threads in Java

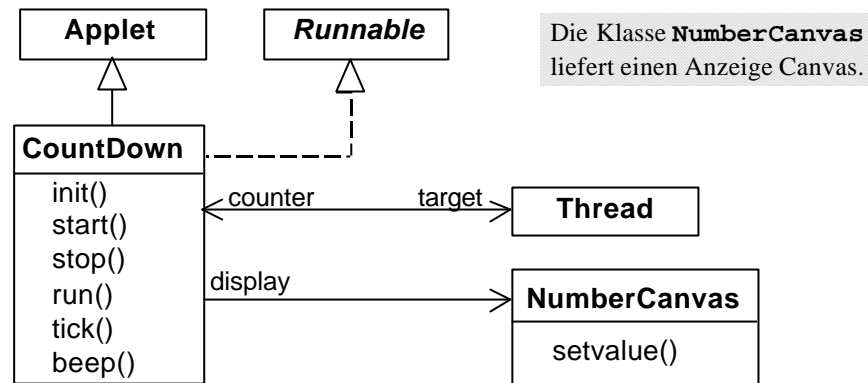
Java erlaubt keine mehrfache Vererbung, daher sinnvollere Alternative: Implementierung eines Interfaces "Runnable".



Wie erzeugt MyRun Methode Thread mit eigener `run()` Methode?  
`Thread lokalMyRun = new Thread(this);`

4

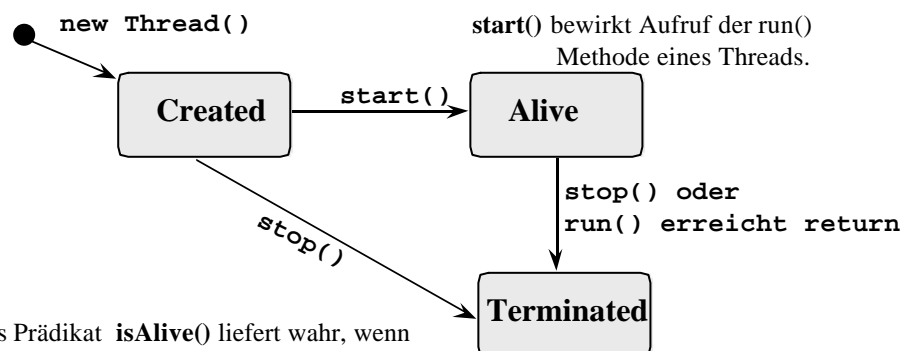
## Beispiel: Countdown timer - Klassendiagramm



5

## (Einfacher) Lebenszyklus eines Threads in Java

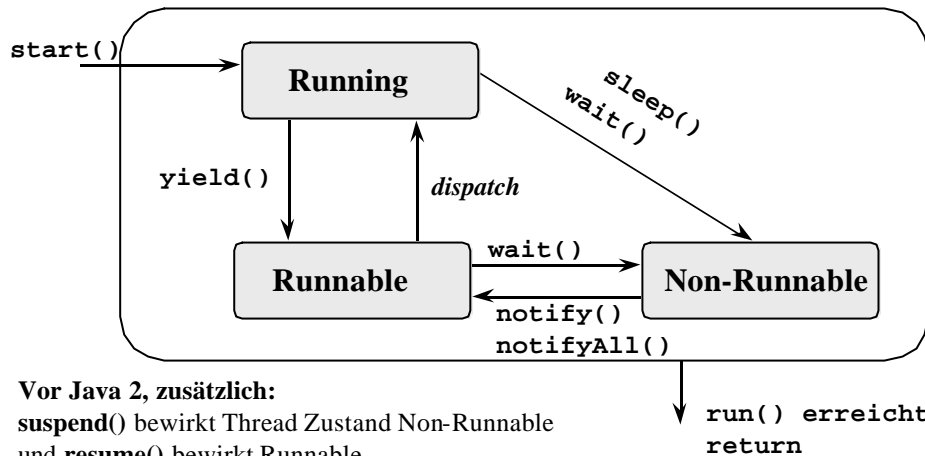
Zustandsübergangssystem zur Darstellung des Lebenszyklusses



6

## Thread Zustände eines “lebenden” Java Threads

Nach Start und vor Terminierung ex. unterschiedliche Thread Zustände :



7

## Initiierung eines Threads

### Methode start()

- ◆ Achtung nicht mit Applet.start() verwechseln !!!
- ◆ MyThread.start() bewirkt das Starten des Threads MyThread, d.h. die run() Methode von MyThread wird aufgerufen.
- ◆ Warum daher nicht einfach MyThread.run() ???  
 Weitere Schritte erforderlich:
  - 1) Initialisierung des Threads innerhalb der VM, Integration in Verwaltungsdatenstrukturen für Scheduling, Accounting etc.
  - 2) Aufruf von run()
  - 3) Behandlung von Exceptions durch Exception Handler
    - ◆ default exception handler, i.e. Methode uncaughtException() der ThreadGroup Class, bewirkt i.w. Ausgabe Stack Trace des throwable objects.
    - ◆ durch eigenen exception handler überladbar
- ◆ Start() darf je Thread nur einmal aufgerufen werden,
  2. Aufruf liefert IllegalThreadStateException falls für Thread isAlive() gilt.

8

## Bearbeitung eines Threads

### Methode run()

---

- ◆ Run() ist Hauptanforderung zur Implementierung von Runnable
- ◆ Run() ist Hauptfunktion eines Threads
- ◆ wenn Run() terminiert (return), terminiert der Thread
- ◆ ab Java 2 erfolgt Threadterminierung im wesentlichen auf freiwilliger Basis  
d.h. Thread terminiert
  - wenn Thread seine Berechnungen/Aufgaben abgeschlossen hat
  - wenn anderer Thread Aufforderung für Terminierung mitteilt, dies geschieht durch Setzen von eigenen, lokalen Daten oder geteilten Objekten.Voraussetzung: Thread kontrolliert Bedingungsflags selbst

9

## Die Klasse Countdown

```
public class Countdown extends Applet
                                implements Runnable {
    Thread counter; int i;
    final static int N = 10;
    AudioClip beepSound, tickSound;
    NumberCanvas display;

    public void init()    {...}
    public void start()   {...}
    public void stop()    {...}
    public void run()     {...}
    private void tick()   {...}
    private void beep()   {...}
}
```

## Die Klasse CountDown - start(), stop() and run()

```
public void start() {
    counter = new Thread(this);
    i = N; counter.start();
}

public void stop() {
    counter = null;
}

public void run() {
    while(true) {
        if (counter == null) return;
        if (i>0) { tick(); --i; }
        if (i==0) { beep(); return;}
    }
}
```

Achtung: Namensgleichheit  
Thread - Applet  
bzgl  
start, stop

Hier: Applet erzeugt bei  
start() einen Thread und  
startet diesen.

Thread Konstruktor mit  
Parameter (this)

bewirkt das  
counter.start()  
die Methode this.run()  
aufruft.

Terminierungserkennung  
über Bedingungen

counter == NULL !!!

11

## Terminierung eines Threads

### Methode stop(). deprecated in Java 2 !!!

- ◆ Achtung: nicht mit Applet.stop() verwechseln !!!
- ◆ Bewirkt Terminierung der run() Methode.
  - Stop() bewirkt Erzeugung eines Objektes der ThreadDeath Class und das run() dieses Objekt als Exception wirft
- ◆ Methode ist nicht empfehlenswert !!!
- ◆ Sinnvoller (wie im vorherigen Beispiel):
  - über Flagabfrage in run() jeweils Terminierungsbedingung testen und ggfs terminieren
  - statt stop(), Flag auf passenden Wert zur Terminierung setzen
- ◆ Bei Beendigung von run() wird
  - Thread deaktiviert
  - Aufräumarbeiten durchgeführt
  - isAlive() wird False
  - Reaktivierung ist nicht mehr möglich

12

## Einfluß auf Thread Scheduling

### Methode yield()

- ◆ MyThread.yield() hält den aktuellen Thread an und erlaubt Bearbeitung eines anderen Threads gleicher Priorität.
  - Yield() ist eine statische Methode, d.h. der aktuell laufende Thread muß nicht zwangsläufig MyThread sein.
  - Thread höherer Priorität kann nicht lauffähig sein, weil dann bereits vorher Unterbrechung erfolgt wäre.
  - Falls kein Thread gleicher Priorität lauffähig ist Wirkung von yield() gleich sleep(0)
  - Falls mehrere Threads gleicher Priorität lauffähig, so ist Auswahl des nächsten bearbeiteten Threads nicht vorherbestimmt, sondern hängt vom Thread-Scheduler der VM ab.
- ◆ Yield() taugt i.w. als Hinweis an VM Thread Scheduler potentiell Wechsel vorzunehmen.

13

## Einfluß auf Thread Scheduling

### Methoden setPriority(int prio), getPriority()

- ◆ Lese- und Schreibmethoden für die Priorität eines Threads
- ◆ Der Scheduler wählt aus den bearbeitbaren Threads denjenigen mit der höchsten Priorität für die weitere Bearbeitung aus.
- ◆ Scheduling ist unterbrechend (preemptive), berücksichtigt Prioritäten.
- ◆ Prioritäten werden nur explizit im Programm (also durch den Programmierer) gesetzt, die VM führt keine eigenmächtigen Änderungen zur Laufzeit durch.
- ◆ Achtung es gelten Wertebereichsbeschränkungen, auch Thread Group spezifisch.
- ◆ Genauere Betrachtung des Scheduling zu einem späteren Zeitpunkt in der Vorlesung.

14

## Warten auf Timeout

### Methode sleep()

---

- ◆ Static void sleep(long milliseconds)
- ◆ statische Methode der Thread Class
- ◆ unterbricht die Bearbeitung des Threads für den angegebenen Zeitraum
  - der Thread wird vom Scheduler in Zustand Non-Runnable versetzt
  - bei Ablauf des Zeitintervalls erfolgt Threadzustandswechsel von Non-Runnable zu Runnable
  - wg Thread Verwaltung und Scheduling ist realer Zeitraum der Unterbrechung größer
- ◆ eine feinere Unterscheidung in Nanoseconds wird typischerweise vom Laufzeitsystem, bzw vom unterliegenden Betriebssystem nicht in der geforderten Genauigkeit unterstützt

15

## Beispiel für sleep()

---

- ◆ Warten auf Eintreten einer Bedingung

```
K = 100 ;
...
while (tryCondition() == false) {
    try {
        Thread.sleep( K ) ;
    } catch (Exception e) { }
```
- ◆ Lösung reduziert Effekt von Busy Waiting, aber wie soll Konstante sinnvoll gewählt werden ?
  - K zu groß, dann wartet Thread unnötig lange
  - K zu klein, dann wird Thread unnötig oft geweckt und bearbeitet
- ◆ bessere Lösung durch explizite Benachrichtigung

16



## Lock Mechanismus: synchronized

---

- ◆ Jedes Objekt in Java hat einen „Lock“, eine Semaphore
- ◆ Methoden, die durch synchronized gekennzeichnet werden, müssen den Lock vor der Ausführung erhalten, geben nach Ausführung den Lock frei.
- ◆ Dadurch wird wechselseitiger Ausschluß bei der Durchführung von Methoden ermöglicht.

Beispiel:

```
public synchronized boolean tryCondition() {  
    ...  
}
```

- ◆ Besonderheiten:
  - nicht alle Methoden einer Klasse müssen synchronized sein
  - auch Blöcke können als synchronized gekennzeichnet werden
  - Allokation von Locks kann bei mehreren Threads und mehreren Objekten leicht zu Verklemmungen führen (Deadlockgefahr)

17

## Warten auf eine Bedingung, bzw Benachrichtigung Methode wait()

---

- ◆ Bewirkt Warten auf das Eintreten einer Bedingung
- ◆ Die Bedingung wird NICHT in wait() geprüft, kein Busy Waiting.
- ◆ Die Bedingung muß in einer umgebenden Schleife geprüft werden!
- ◆ Thread wird durch notify() oder notifyAll() benachrichtigt, aber Achtung, Bedingung muß anschließend neu geprüft werden !!!
- ◆ Untervarianten: wait(long timeout) mit zeitlicher Obergrenze
- ◆ Nur innerhalb einer synchronized Methode verwenden, weil wait-notify eine Race-Condition beinhalten
- ◆ Synchronized sichert exklusiven Zugriff auf Objekt (Lock-Mechanismus)
- ◆ Wait() gibt Lock temporär frei, realloziert Lock bevor es nach notify() zurückkehrt.
- ◆ Wait() ist „native method“, gehört zur Klasse Object

18

### **Gegenpart zu wait(), Benachrichtigung**

#### **Methode notify(), notifyAll()**

- ◆ Benachrichtigt einen/alle wartenden Threads
- ◆ nur innerhalb einer synchronized Methode verwenden
- ◆ Gegenstück zu wait()
- ◆ Typisches Anwendungsmuster:

```
public synchronized void getBusyFlag() {  
    while (tryCondition()==false) {  
        try {  
            wait() ;  
        } catch (Exception e) {}  
    }  
}
```

```
public synchronized void freeBusyFlag() {  
    ...  
    notify() ;  
}
```

### **Gegenpart zu wait(), Benachrichtigung**

#### **Methode notify(), notifyAll()**

- ◆ Beispiel mit blockweiser Belegung des Locks:
- ◆ Randbedingung: wait und notify müssen zum synchronized Objekt gehören
- ◆ Sei Variable StringBuffer sb innerhalb der Klasse definiert

```
Public void getLock() {  
    ...  
    synchronized (sb) {  
        try {  
            sb.wait() ;  
        } catch (Exception e) {}  
    }  
}
```

```
Public void freeLock() {  
    ...  
    synchronized (sb) {  
        sb.notify() ;  
    }  
}
```

## Race Condition bei wait-notify ohne synchronized

---

- ◆ Race Condition = Ausgang einer Berechnung abhängig von Interleaving in der Bearbeitung von Threads (UNERWÜNSCHT)
- ◆ Hier
  - 1. Thread testet Bedingung und entscheidet sich für Wartesituation
  - 2. Thread ändert Bedingungsvariable, Wartesituation entfällt eigentlich
  - 2. Thread ruft notify auf, Signal geht mangels wartenden Threads verloren.
  - 1. Thread ruft die wait() Methode auf
  - Resultat des Szenarios unbefriedigend, weil 1. Thread nicht mehr geweckt wird
- ◆ Wenn Testen der Bedingung und Modifikation der Bedingungsvariable, wait, und notify mit synchronized gekapselt werden, sichert wechselseitiger Ausschluß, daß Race Condition nicht auftreten kann.

21

## Methode join(), join(long timeout)

---

- ◆ Bewirkt Warten auf die Terminierung eines Threads, d.h. bis das Attribut isAlive() nicht erfüllt ist.
  - Dies schließt auch die Situation von Threads ein, die noch kein start() erfahren haben!
- ◆ Parametrisierte Variante wartet nicht länger als die angegebene Zeitspanne.
- ◆ Join kann zur Realisierung einer Barrier Synchronisation genutzt werden, typisches Muster:
  - Master-Thread startet N Worker-Threads zur Durchführung parallel bearbeitbarer Teilaufgaben
  - Master-Thread wartet mit N Join Aufrufen (je Worker 1 Aufruf) auf Terminierung der Worker bevor er selbst mit weiteren Aufgaben fortfährt.

22

## **Methode interrupt(), nur Java 2 und folgende**

---

- ◆ Statische Methode der Thread und ThreadGroup Class
- ◆ sendet Interrupt Signal an den angegebenen Thread
- ◆ ein blockierter Thread (join(), sleep(), wait()) ist dadurch nicht mehr blockiert, ggfs wird ein Interrupt Flag gesetzt
- ◆ falls der Ziel-Thread eine Methode ausführt, die eine Exception werfen kann, so wird durch den Interrupt Aufruf eine entsprechende Exception verursacht
- ◆ anderenfalls kann der Ziel-Thread einen Interrupt anhand interrupted() und isInterrupted() feststellen
- ◆ Anwendung z.B. bei Producer/Consumer Pattern, wenn Konsument bei leerem Puffer ein wait() ausführt, aus dem er mittels Interrupt und Exception Handling bei Eintreffen von Nachrichten geweckt wird.
- ◆ Achtung: NICHT zur Aufhebung von Blockierungen bei I/O verwenden !!!

23

## **Nur wg Vollständigkeit:**

### **Methoden suspend(), resume(). deprecated in Java 2!!!**

- ◆ Methoden der Thread und ThreadGroup Class
- ◆ suspend() bewirkt, dass der Thread vom jeweiligen Zustand zum Suspend Zustand versetzt wird und nicht bearbeitet wird
- ◆ resume() bewirkt, dass der Thread vom Suspend Zustand in den vorherigen Zustand (running, ready etc) zurückversetzt wird und ggfs weiterbearbeitet werden kann.
- ◆ Methoden sind ebenso wie stop() problematisch, weil dadurch Lock Starvation erreicht werden kann, im Extremfall bis zum Stillstand der VM.
- ◆ Suspend() bei Thread, der Lock besitzt, führt dies zwar zur Suspendierung, aber nicht zur Freigabe des Locks !!!
- ◆ Resume als Gegenpart ist harmlos, aber ohne Suspend zweckfrei.

24

## Zusammenfassung

---

- ◆ Überblick über Java Sprachunterstützung zur Threadprogrammierung
- ◆ Threads durch Vererbung oder Implementierung des Runnable Interfaces
- ◆ Methoden zum Initialisieren (start) und Betreiben (run)
- ◆ Kommunikation/Synchronisation:
  - wait-notify
  - interrupt- wait,sleep,join
- ◆ Scheduling
  - yield
  - getPriority, setPriority
- ◆ Im Folgenden
  - Besonderheiten beim Entwurf und Erstellung korrekter Java Programm mit mehreren Threads (Sicherheit, Lebendigkeit, ... )