

Script generated by TTT

Title: Seidl: Informatik_1 (09.01.2013)

Date: Wed Jan 09 14:16:08 CET 2013

Duration: 64:07 min

Pages: 43

19.3 Dokumentation

Unterschiedliche Zielgruppen benötigen unterschiedliche **Informationen** über ein Software-System.

Benutzer: Bedienungsanleitung.

- Wie installiere ich das Programm?
- Wie rufe ich es auf? Wie beende ich es?
- Welche Menues gibt es?
- Wie hilft mir das Programm, meine Vorstellungen zu verwirklichen?
- Wie mache ich einen Fehler rückgängig?
- Was mache ich, wenn ich nicht mehr weiter weiß?

673

Entwickler: Beschreibung der Programmierschnittstelle (API).

- Welche Pakete gibt es?
- Welche Klassen gibt es, und wofür sind sie gut?
- Welche Methoden stellen die Klassen bereit ...
 - ... wie ruft man sie auf?
 - ... was bewirken sie?
 - ... welche Exceptions werfen sie?
- Welche Variablen gibt es?
- Welche Konstruktoren gibt es?

674

Programmierer: **Kommentierter** Programm-Code.

- Wozu sind Pakete/Klassen/Methoden gut und ...
- wie sind sie implementiert?
- Welche anderen Pakete/Klassen/Methoden benutzen sie?
- Welcher Algorithmus wurde zugrunde gelegt?
- Wie sieht der Kontroll-Fluss aus?
- Wozu werden die lokalen Variablen verwendet?
- Was bewirken einzelne Programm-Abschnitte?

675

- Zur Kommentierung von Programm-Code habt Ihr in den Übungen reichlich Gelegenheit.
- Zum professionellen Schreiben von Bedienungsanleitungen **sollten** Benutzer, Psychologen, Didaktiker hinzugezogen werden (schön wärs ...)
- Zur (halb-) automatischen Erstellung von API-Dokumentationen aus **Java**-Programm-Paketen gibt es das Hilfsprogramm javadoc.
- Der Aufruf:

```
> javadoc -d doc a a.b a.c
```

erzeugt im Verzeichnis doc die Unterverzeichnisse a, a/b und a/c für die Pakete a, a.b und a.c und legt darin HTML-Seiten für die Klassen an.

676

- Zusätzlich werden verschiedene Indices zum schnelleren Auffinden von Klassen oder Methoden angelegt.
- **Achtung:** Eine sinnvolle Beschreibung für Klassen, Variablen oder Methoden kann natürlich nicht automatisch generiert werden.
- Dazu dienen **spezielle Kommentare** im Programm-Text...

Ein Beispiel ...

677

```
package a;
/**
 * Die einzige Klasse des Pakets a.
 */
public class A {
    /**
     * Eine statische Beispiel-Methode. Sie legt ein A-Objekt
     * an, um es auf die Standard-Ausgabe zu schreiben.
     */
    public static void main(String[] args) {
        System.out.println(new A());
    } // end of main()
} // end of class A
```

678

- javadoc-Kommentare beginnen mit `/**` und enden mit `*/`
- einzelne `*` am Anfang einer Zeile werden überlesen;
- die Erklärung bezieht sich stets auf das Programm-Stück unmittelbar dahinter.
- Der erste Satz sollte eine knappe Zusammenfassung darstellen. Er endet beim ersten `.` bzw. am Beginn der ersten Zeile, die mit einem **Schlüsselwort** beginnt.
- Einige (wenige) Schlüsselworte gestatten es, besondere wichtige Informationen hervorzuheben.

679

```

package a;
/**
 * Die einzige Klasse des Pakets a.
 * @author Helmut Seidl
 * @see a
 */
public class A {
/**
 * Eine statische Beispiel-Methode. Sie legt ein A-Objekt
 * an, um es auf die Standard-Ausgabe zu schreiben.
 * @param args wird komplett ignoriert.
 */
public static void main(String[] args) {
    System.out.println(new A());
} // end of main()
} // end of class A

```

680

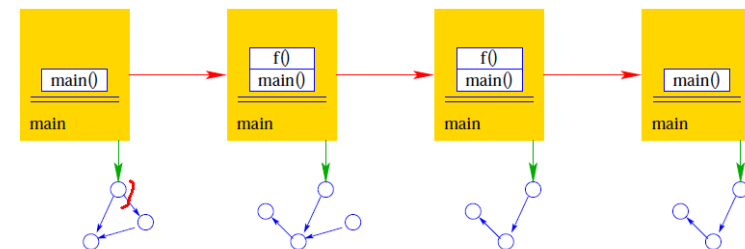
- Schlüsselworte beginnen mit “@”.
- Der zugehörige Textabschnitt geht bis zum nächsten Schlüsselwort bzw. bis zum Ende des Kommentars.
- @author kennzeichnet den Author.
- @see gibt eine Referenz an.
Ist die Referenz als Element der Doku bekannt, wird ein Link generiert ...
- @param erwartet einen Parameter-Namen, gefolgt von einer Erläuterung. Analog erwartet ...
- @return eine Erläuterung des Rückgabewertes;
- @exception eine möglicherweise geworfene Exception zusammen mit einer Erläuterung.

681

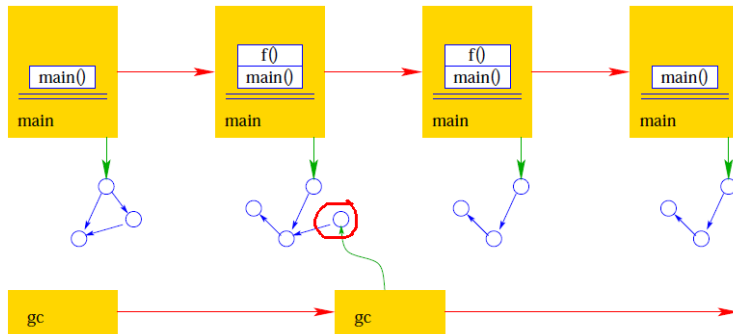
20 Threads

- Die Ausführung eines Java-Programms besteht in Wahrheit nicht aus einem, sondern mehreren parallel laufenden Threads.
- Ein Thread ist ein sequentieller Ausführungs-Strang.
- Der Aufruf eines Programms startet einen Thread main, der die Methode main() des Programms ausführt.
- Ein weiterer Thread, den das Laufzeitsystem parallel startet, ist die Garbage Collection.
- Die Garbage Collection soll mittlerweile nicht mehr erreichbare Objekte beseitigen und den von ihnen belegten Speicherplatz der weiteren Programm-Ausführung zur Verfügung stellen.

682



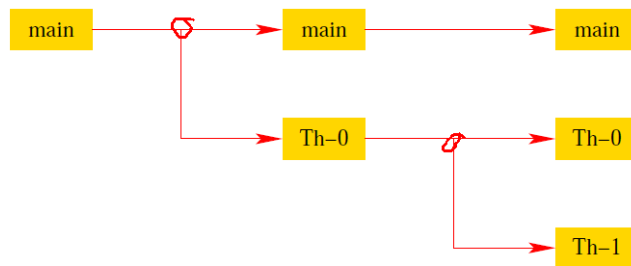
683



684

- Mehrere Threads sind auch nützlich, um
 - ... mehrere Eingabe-Quellen zu überwachen (z.B. Mouse-Klicks und Tastatur-Eingaben) ↑**Graphik**;
 - ... während der Blockierung einer Aufgabe etwas anderes Sinnvolles erledigen zu können;
 - ... die Rechenkraft mehrerer Prozessoren auszunutzen.
- Neue Threads können deshalb vom Programm selbst erzeugt und gestartet werden.
- Dazu stellt **Java** die Klasse **Thread** und das Interface **Runnable** bereit.

685



686

Beispiel:

```

public class MyThread extends Thread {
    public void hello(String s) {
        System.out.println(s);
    }
    public void run() {
        hello("I'm running ...");
    } // end of run()
    public static void main(String[] args) {
        MyThread t = new MyThread();
        t.start();
        System.out.println("Thread has been started ...");
    } // end of main()
} // end of class MyThread

```

687

- Neue Threads werden für Objekte aus (Unter-) Klassen der Klasse `Thread` angelegt.
- Jede Unterklasse von `Thread` sollte die Objekt-Methode `public void run();` implementieren.
- Ist `t` ein `Thread`-Objekt, dann bewirkt der Aufruf `t.start();` das folgende:
 1. ein neuer Thread wird initialisiert;
 2. die (parallele) Ausführung der Objekt-Methode `run()` für `t` wird angestoßen;
 3. die eigene Programm-Ausführung wird hinter dem Aufruf fortgesetzt.

688

Beispiel:

```
public class MyThread extends Thread {
    public void hello(String s) {
        System.out.println(s);
    }
    public void run() {
        hello("I'm running ...");
    } // end of run()
    public static void main(String[] args) {
        MyThread t = new MyThread();
        t.start();
        System.out.println("Thread has been started ...");
    } // end of main()
} // end of class MyThread
```

687

- Neue Threads werden für Objekte aus (Unter-) Klassen der Klasse `Thread` angelegt.
- Jede Unterklasse von `Thread` sollte die Objekt-Methode `public void run();` implementieren.
- Ist `t` ein `Thread`-Objekt, dann bewirkt der Aufruf `t.start();` das folgende:
 1. ein neuer Thread wird initialisiert;
 2. die (parallele) Ausführung der Objekt-Methode `run()` für `t` wird angestoßen;
 3. die eigene Programm-Ausführung wird hinter dem Aufruf fortgesetzt.

688

Beispiel:

```
public class MyRunnable implements Runnable {
    public void hello(String s) {
        System.out.println(s);
    }
    public void run() {
        hello("I'm running ...");
    } // end of run()
    public static void main(String[] args) {
        Thread t = new Thread(new MyRunnable());
        t.start();
        System.out.println("Thread has been started ...");
    } // end of main()
} // end of class MyRunnable
```

689

- Auch das Interface `Runnable` verlangt die Implementierung einer Objekt-Methode `public void run();`
- `public Thread(Runnable obj);` legt für ein `Runnable`-Objekt `obj` ein `Thread`-Objekt an.
- Ist `t` das `Thread`-Objekt für das `Runnable obj`, dann bewirkt der Aufruf `t.start();` das folgende:
 1. ein neuer `Thread` wird initialisiert;
 2. die (parallele) Ausführung der Objekt-Methode `run()` für `obj` wird angestoßen;
 3. die eigene Programm-Ausführung wird hinter dem Aufruf fortgesetzt.

690

Beispiel:

```
public class MyRunnable implements Runnable {
    public void hello(String s) {
        System.out.println(s);
    }
    public void run() {
        hello("I'm running ...");
    } // end of run()
    public static void main(String[] args) {
        Thread t = new Thread(new MyRunnable());
        t.start();
        System.out.println("Thread has been started ...");
    } // end of main()
} // end of class MyRunnable
```

689

Mögliche Ausführungen:

```
Thread has been started ...
I'm running ...
```

... oder:

```
I'm running ...
Thread has been started ...
```

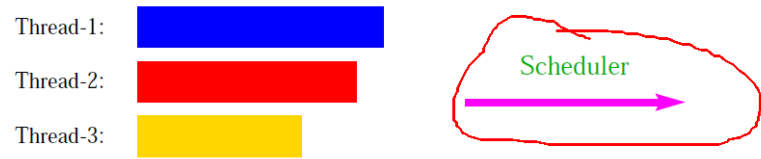
691

- Ein `Thread` kann nur eine Operation ausführen, wenn ihm ein Prozessor (CPU) zur Ausführung zugeteilt worden ist.
- Im Allgemeinen gibt es mehr `Threads` als CPUs.
- Der `Scheduler` verwaltet die verfügbaren CPUs und teilt sie den `Threads` zu.
- Bei verschiedenen Programm-Läufen kann diese Zuteilung verschieden aussehen!!!
- Es gibt verschiedene Politiken, nach denen sich `Scheduler` richten können **↑Betriebssysteme**.

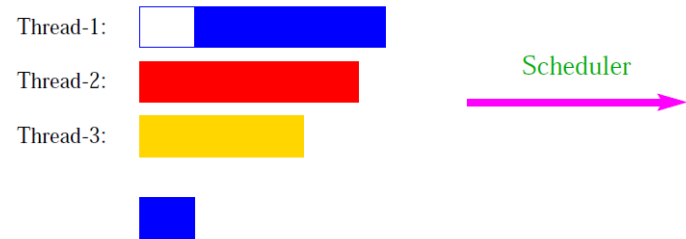
1. Zeitscheiben-Verfahren:

- Ein `Thread` erhält eine CPU nur für eine bestimmte Zeitspanne (`Time Slice`), in der er rechnen darf.
- Danach wird er unterbrochen. Dann darf ein anderer.

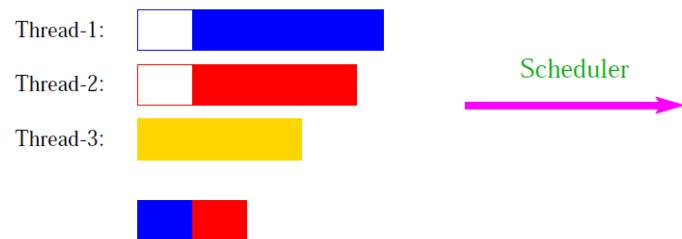
692



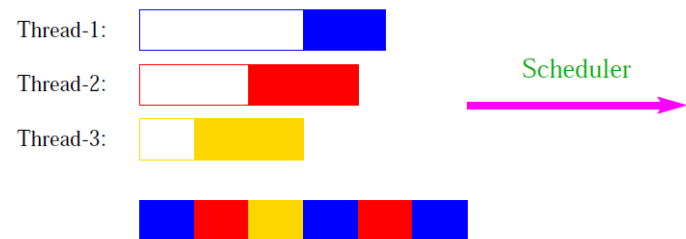
693



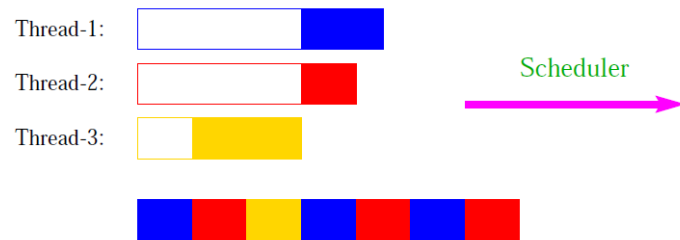
694



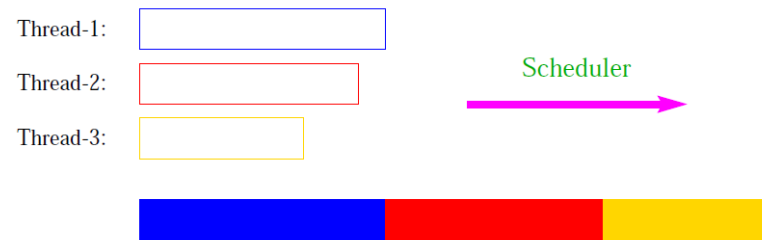
695



699



700



712

Beispiel:

```

public class Start extends Thread {
    public void run() {
        System.out.println("I'm running ...");
        while(true) ;
    }
    public static void main(String[] args) {
        (new Start()).start();
        (new Start()).start();
        (new Start()).start();
        System.out.println("main is running ...");
        while(true) ;
    }
} // end of class Start

```

713

... liefert als Ausgabe (bei naivem Scheduling und einer CPU) :

main is running ...

714

Beispiel:

```
public class Start extends Thread {
    public void run() {
        System.out.println("I'm running ...");
        while(true) ;
    }
    public static void main(String[] args) {
        (new Start()).start();
        (new Start()).start();
        (new Start()).start();
        System.out.println("main is running ...");
        while(true) ;
    }
} // end of class Start
```

713

... liefert als Ausgabe (bei naivem Scheduling und einer CPU) :

```
main is running ...
```

- Weil main nie fertig wird, erhalten die anderen Threads keine Chance, sie **verhungern**.
- Faires Scheduling mit einem Zeitscheiben-Verfahren würde z.B. **liefern**:

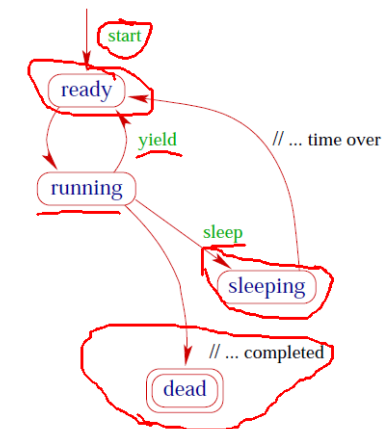
```
I'm running ...
main is running ...
I'm running ...
I'm running ...
```

715

- **Java** legt nicht fest, wie intelligent der Scheduler ist.
 - Die aktuelle Implementierung unterstützt **fares** Scheduling.
 - Programme sollten aber für jeden Scheduler das **gleiche Verhalten** zeigen. Das heißt:
 - ... Threads, die aktuell nichts sinnvolles zu tun haben, z.B. weil sie auf Verstreichen der Zeit oder besseres Wetter warten, sollten stets ihre CPU anderen Threads zur Verfügung stellen.
 - ... Selbst wenn Threads etwas Vernünftiges tun, sollten sie ab und zu andere Threads laufen lassen.
- (Achtung: Wechsel des Threads ist **teuer**!!!)
- Dazu verfügt jeder Thread über einen **Zustand**, der bei der Vergabe von Rechenzeit berücksichtigt wird.

716

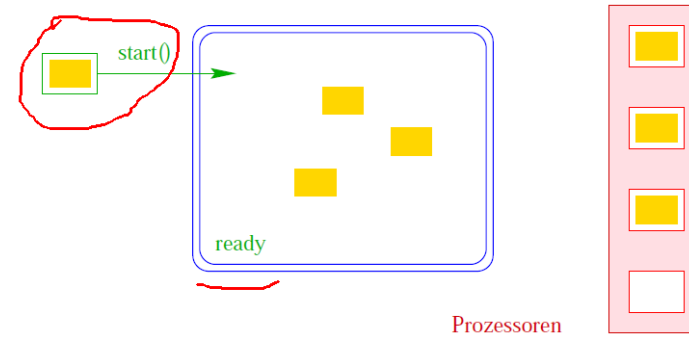
Einige Thread-Zustände:



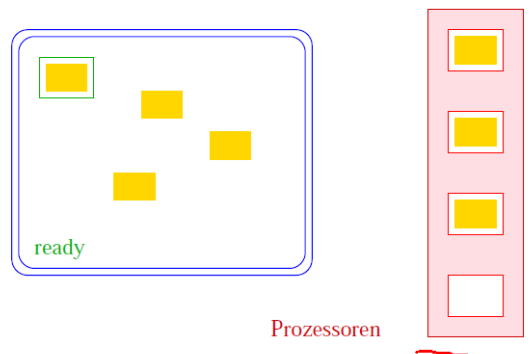
717

- `public void start();` legt einen neuen Thread an, setzt den Zustand auf `ready` und übergibt damit den Thread dem Scheduler zur Ausführung.
- Der Scheduler ordnet den Threads, die im Zustand `ready` sind, Prozessoren zu (“dispatching”). Aktuell laufende Threads haben den Zustand `running`.
- `public static void yield();` setzt den aktuellen Zustand zurück auf `ready` und unterbricht damit die aktuelle Programm-Ausführung. Andere ausführbare Threads erhalten die Gelegenheit zur Ausführung.
- `public static void sleep(int msec) throws InterruptedException;` legt den aktuellen Thread für msec Millisekunden schlafen, indem der Thread in den Zustand `sleeping` wechselt.

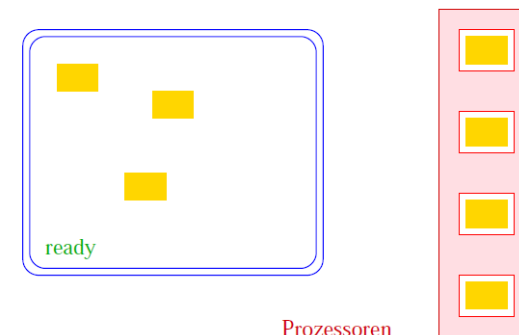
718



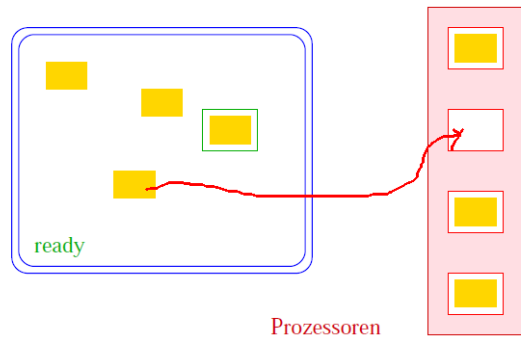
719



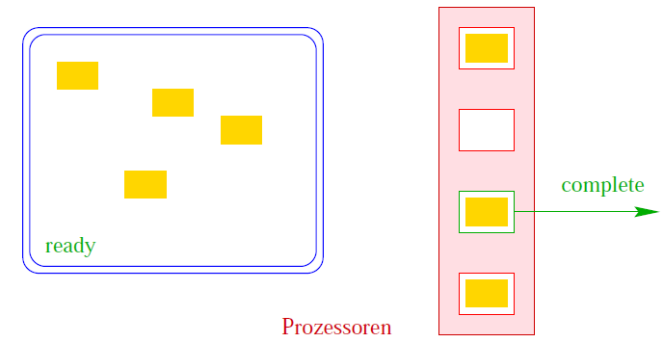
720



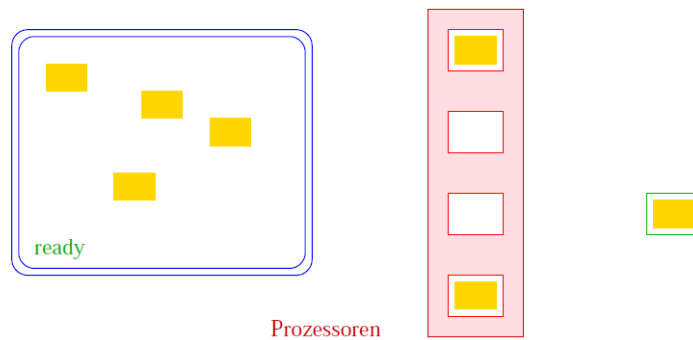
722



724



725



726

20.1 Futures

- Die Berechnung eines Zwischenergebnisses kann lange dauern.
- Während dieser Berechnung kann möglicherweise etwas anderes sinnvolles berechnet werden.

727