

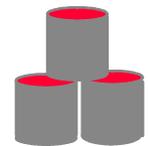
Grundlagen der Informatik

– Ausnahmebehandlung & Threads –

Prof. Dr. Bernhard Schiefer

(basierend auf Unterlagen von Prof. Dr. Duque-Antón)

bernhard.schiefer@fh-kl.de
<http://www.fh-kl.de/~schiefer>



Inhalt

- Ausnahmebehandlung
- Threads

Fehlersituationen

■ Wann treten Fehlersituationen auf?

- ⇒ Ungültige Eingabewerte
- ⇒ Programmierfehler
- ⇒ Fehler der Laufzeitumgebung

■ Mögliche Reaktionen auf Fehlersituationen

- ⇒ Rückgabe eines speziellen Rückgabewertes
- ⇒ Absturz der Anwendung
- ⇒ Werfen einer Ausnahme

Rückgabe eines speziellen Rückgabewertes

■ Beispiel:

- ⇒ Funktion zur Berechnung der Fakultät (n!)
 - ◆ `int fak (int n) { . . . }`
- ⇒ Wie soll die Funktion auf negative Werte reagieren?
- ⇒ Mögliche Designentscheidung: Rückgabe von -1
 - ◆ Im normalen Ablauf kann -1 niemals auftreten

■ Nachteile / Probleme

- ⇒ Fehleranfällig:
 - ◆ Falls der/die Entwickler/in des aufrufenden Programms nicht aufpasst wird mit dem zurückgelieferten Wert weitergearbeitet, ohne zu merken, dass dieser falsch ist.
- ⇒ Wie soll verfahren werden, wenn jeder Wert eine gültige Rückgabe darstellt? Beispiel:
 - ◆ `int div (int z, int n) { . . . }`

Absturz der Anwendung

- Jeder Aufruf mit ungültigen Werten führt zum Programmabbruch
- Vorteil:
 - ⇒ Es wird nie aus Versehen mit fehlerhaften Werten gearbeitet
- Nachteil:
 - ⇒ Ein kommentarloser Absturz ist für Anwender inakzeptabel
 - ⇒ Um Abbruch zu umgehen, müssen daher alle denkbaren Fehlerfälle vor jedem Funktionsaufruf abgefragt werden
 - ◆ Sehr aufwändig und fehleranfällig
 - ◆ Das führt zu komplexen, schwer wartbaren Programmen

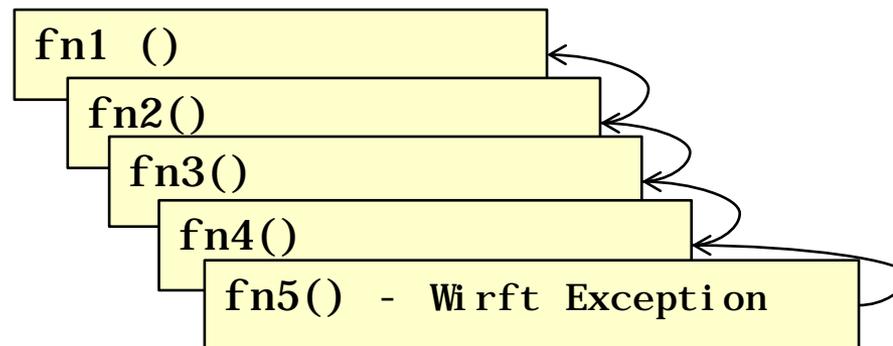
Ausnahmekonzepte

- Einige Programmiersprachen verfügen über ein spezielles Ausnahmekonzept:
 - ⇒ Jede Methode kann zusätzlich zum normalen Rückgabewert eine Ausnahme werfen
- Ausnahmen können entdeckt und signalisiert werden
 - ⇒ vom Betriebssystem
 - ⇒ vom Laufzeitsystem (z.B. Java VM)
 - ⇒ vom Entwickler selbst
- Ausnahmekonzepte findet man bei vielen Programmiersprachen:
 - ⇒ Java, C++, C#, Visual Basic.Net, PHP (ab Version 5), ABAP OO, . . .
 - ⇒ Die Konzepte unterscheiden sich in der Regel in den Details

Ausnahmebehandlung

■ Strukturierte Ausnahmebehandlung

- ⇒ Tritt eine Ausnahme auf, kann diese behandelt werden oder an die aufrufende Schicht weiter gegeben werden
- ⇒ Die Entscheidung kann auch - von der Art der Ausnahme abhängig - in jeder Schicht unterschiedlich ausfallen
- ⇒ Denkbar ist auch, dass zunächst eine eigene Behandlung erfolgt, die Ausnahme dann aber doch weiter gereicht wird.
- ⇒ Wird eine Ausnahme bis zur letzten Schicht nicht behandelt, führt dies in der Regel zum Programmabbruch.



Ausnahmebehandlung in Java

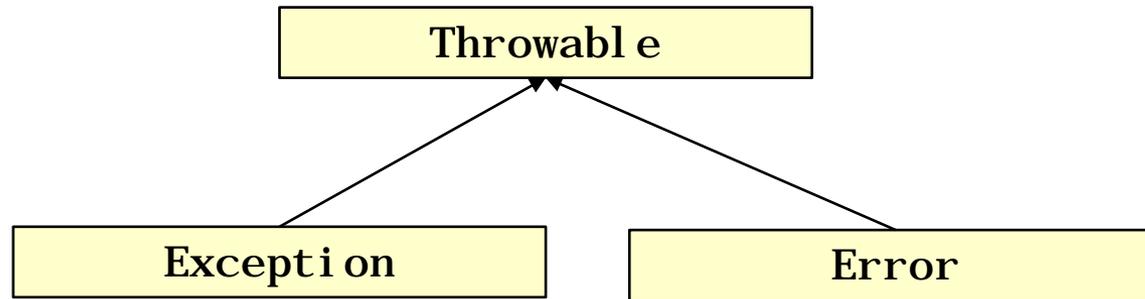
- Ausnahmen können mit Hilfe des try/catch-Konstruktes abgefangen werden.
- Ausnahmen werden als Instanzen von Unterklassen der Klasse *Exception* bzw. *Throwable* geliefert.
- Prinzip:
 - ⇒

```
try {  
    // Anweisungen  
} catch (Exception e) {  
    // Anweisungen zur Fehlerbehandlung  
} finally {  
    // für Dinge, die auch im Ausnahmefall ausgeführt werden  
}
```
- Es können mehrere catch-Anweisungen angegeben werden.

Die Klasse Throwable

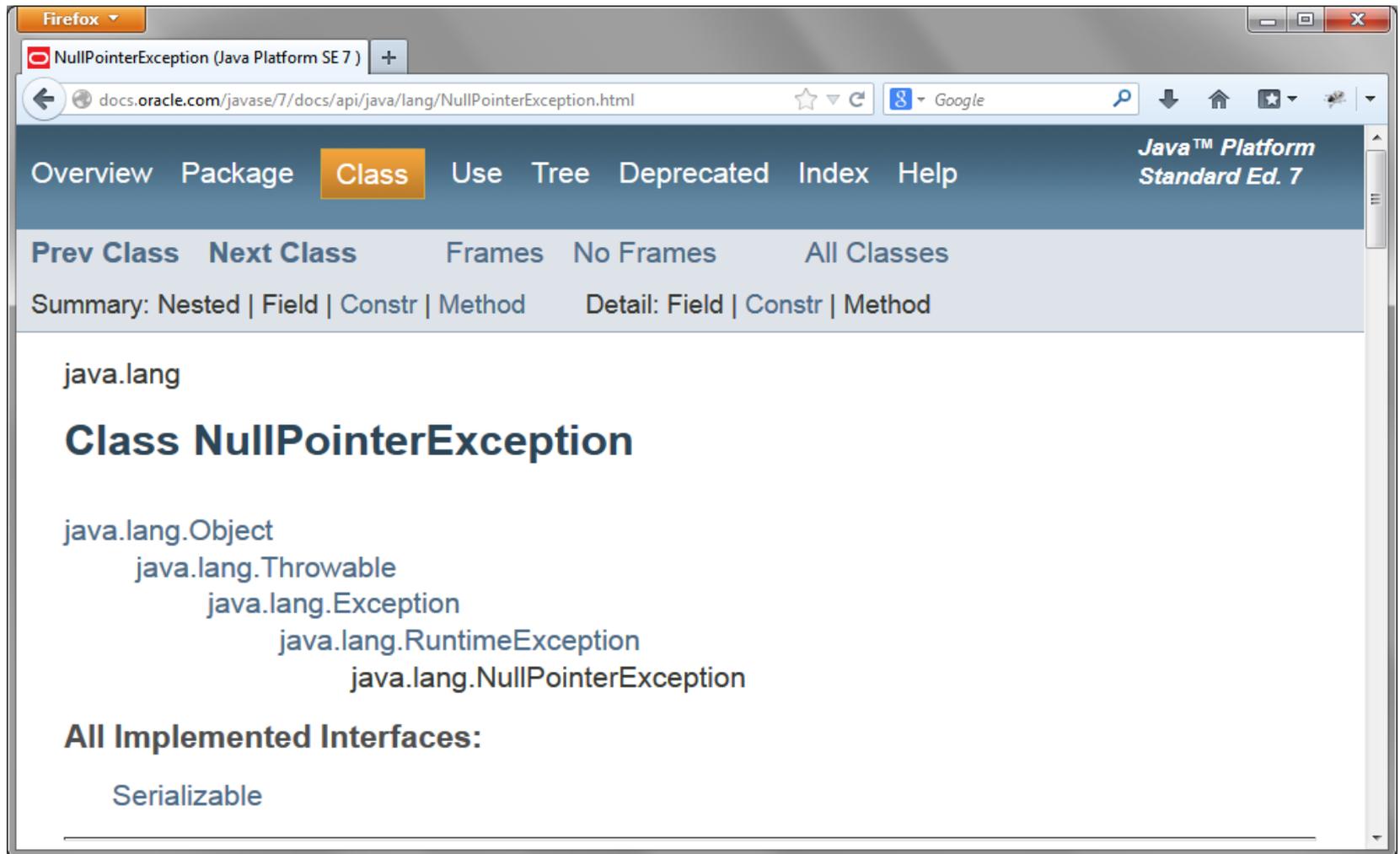
- Java unterscheidet verschiedene Formen von Ausnahmen

⇒ die relevanten Klassen sind: Throwable, Exception, Error



- Die für Anwendungsentwickler nutzbaren Ausnahmen sind in der Regel Unterklassen von Exception

Beispiel für Exception-Hierarchie



The screenshot shows the Oracle Java Platform SE 7 API documentation for the `NullPointerException` class. The browser window title is "NullPointerException (Java Platform SE 7)". The address bar shows the URL `docs.oracle.com/javase/7/docs/api/java/lang/NullPointerException.html`. The navigation menu includes "Overview", "Package", "Class" (highlighted), "Use Tree", "Deprecated", "Index", and "Help". The "Java™ Platform Standard Ed. 7" logo is visible in the top right. Below the navigation menu, there are links for "Prev Class", "Next Class", "Frames", "No Frames", and "All Classes". The "Summary" section shows "Nested | Field | Constr | Method" and "Detail: Field | Constr | Method". The class hierarchy is displayed as follows:

```
java.lang
  java.lang.Object
    java.lang.Throwable
      java.lang.Exception
        java.lang.RuntimeException
          java.lang.NullPointerException
```

The "All Implemented Interfaces:" section lists `Serializable`.

Deklaration von Ausnahmen

- Ausnahmen, die in einer Methode bewusst geworfen werden, müssen in der Signatur der Methode deklariert werden.
- Beispiel:
 - ⇒ `int demo (String arg) throws XYExcetion, YZException {`
`...`
`}`
- Nicht alle denkbaren Ausnahmen müssen deklariert werden. Keine Deklaration ist erforderlich bei fatalen Fehlern, deren Auftreten normalerweise kein sinnvolles Weiterarbeiten ermöglicht:
 - ⇒ Unterklassen von `Error`
 - ⇒ Unterklassen von `RuntimeException`

Ausnahmen selbst werfen

- Ausnahmen können auch von eigenen Programmen erzeugt und geworfen werden
- Dabei ist nur die Nutzung vorhandener Klassen möglich, sondern auch das Schreiben eigener, von Exception abgeleiteter, Klassen
 - ⇒ Instanzen vorhandener Exception-Klassen
 - ⇒ Instanzen eigener Exception-Klassen

- **Beispiel:**

```
⇒ public static int div (int z, int n) {  
    if (n==0) {  
        throw new IllegalArgumentException("Division durch 0!");  
    }  
    ...  
}
```

Eigene Ausnahmeklasse

- Große Freiheit besteht bei der Nutzung/Schaffung eigener Ausnahmeklassen.

- Beispiel:

```
⇒ public class ExceptionDemo extends Exception {  
    public ExceptionDemo(int z, int n) {  
        super("Div Operation gescheitert mit " + z + "/" + n);  
    }  
}
```

- Deklaration bei Nutzung erforderlich:

```
⇒ public static int div(int z, int n) throws ExceptionDemo {  
    if (n == 0) {  
        throw new ExceptionDemo(z,n);  
    }  
    ...  
}
```

Reaktion auf Ausnahmen - Java

- Auf einer Ausnahme werden verschiedene hilfreiche Methode angeboten, die ermöglichen:
 - ⇒ Analyse, Protokollierung und Behandlung der Ausnahme
- Hilfreiche Methoden von Throwable:
 - ⇒ public String getMessage()
 - ⇒ public String getLocalizedMessage()
 - ⇒ public StackTraceElement[] getStackTrace()
 - ⇒ public void printStackTrace()

Threads

- Bei vielen Anwendungen ist es wünschenswert, dass verschiedene Abläufe für einen Benutzer parallel ablaufen:
 - ⇒ So möchte z.B. ein Nutzer eine Datei aus dem Internet laden, während
 - ⇒ er gleichzeitig einen Text in einem Bildschirmfenster schreibt.
 - ⇒ Nutzer wäre nicht zufrieden, wenn er während des Ladevorgangs jegliche Aktivität einstellen und untätig warten müsste.

Threads

- Ein (entfernter) Server, soll gleichzeitig viele Nutzer bedienen:
 - ⇒ So soll ein Amazon-Server (scheinbar) gleichzeitig und nicht nacheinander viele Anfragen bedienen können,
 - ⇒ d.h. alle anfragenden Nutzer sollen den Eindruck gewinnen, sie würden sofort (ohne Wartezeit) vom Server bedient werden.
- Hätte man genug physikalische Prozessoren, so könnte man alle Programme, die nicht voneinander abhängig sind, tatsächlich unabhängig auf verschiedenen Prozessoren ablaufen lassen.
 - ⇒ Da Laden einer beliebigen Datei und das Schreiben eines Textes nichts miteinander zu tun hat, wäre eine parallele Abarbeitung des obigen Beispiels auf einem Mehrprozessorsystem tatsächlich hilfreich.

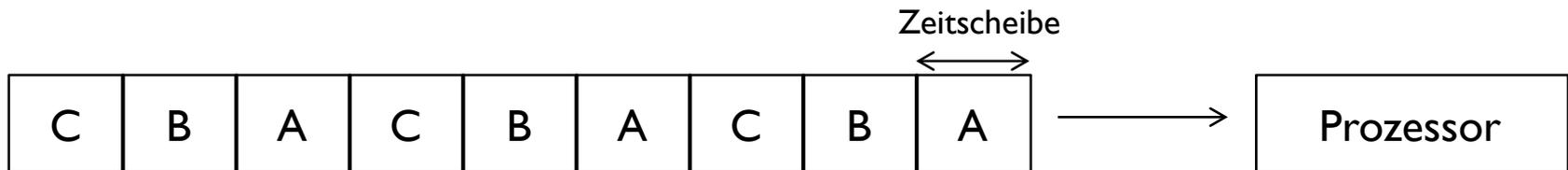
Prozesskonzept

- Aus Kostengründen wurden in der Praxis früher oft nur Rechner mit einem einzigen Prozessor verwendet,
 - ⇒ so dass zu einem Zeitpunkt tatsächlich nur ein Programm den Prozessor besitzen konnte,
 - ⇒ d.h. verschiedene Programme können nur nacheinander auf dem Prozessor ablaufen.
- Bis in die sechziger Jahre waren die Betriebssysteme von Rechnern sogenannte Batch- Betriebssysteme, d.h. sie arbeiteten nach dem Prinzip des Stapelbetriebs



Prozesskonzept

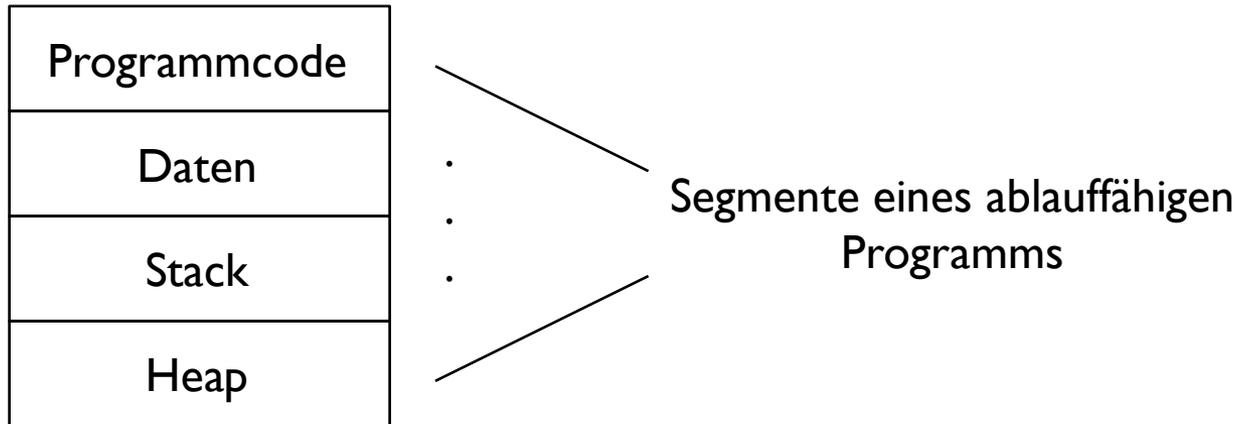
- Erste Betriebssysteme, die ein Prozesskonzept unterstützten, arbeiteten nach dem Prinzip der Zeitscheiben (Time Sharing Operation Systems).
 - ⇒ Jeder Prozess bekommt abwechselnd vom Scheduler eine bestimmte Zeit lang den Prozessor zur Verfügung gestellt.
- Statt einer realen parallelen Ausführung nun quasi-parallele Ausführung als Folge einer geeigneten sequentiellen Ausführung



Prozess-Kontext

- Jeder Prozess des Betriebssystems hat seinen eigenen Prozessekontext.
- Dieser besteht aus:
 - ⇒ Programmcode, Daten, Stack und Heap
 - ⇒ Prozessumgebung, mit Registerinhalten wie Stackpointer (Zeiger auf Stackanfang), Befehlszeiger (Zeiger auf nächste abzuarbeitende Anweisung)
 - ⇒ temporäre Daten und geöffnete Dateien, offene Datenbankverbindungen und ähnlichem

Prozess-Kontext

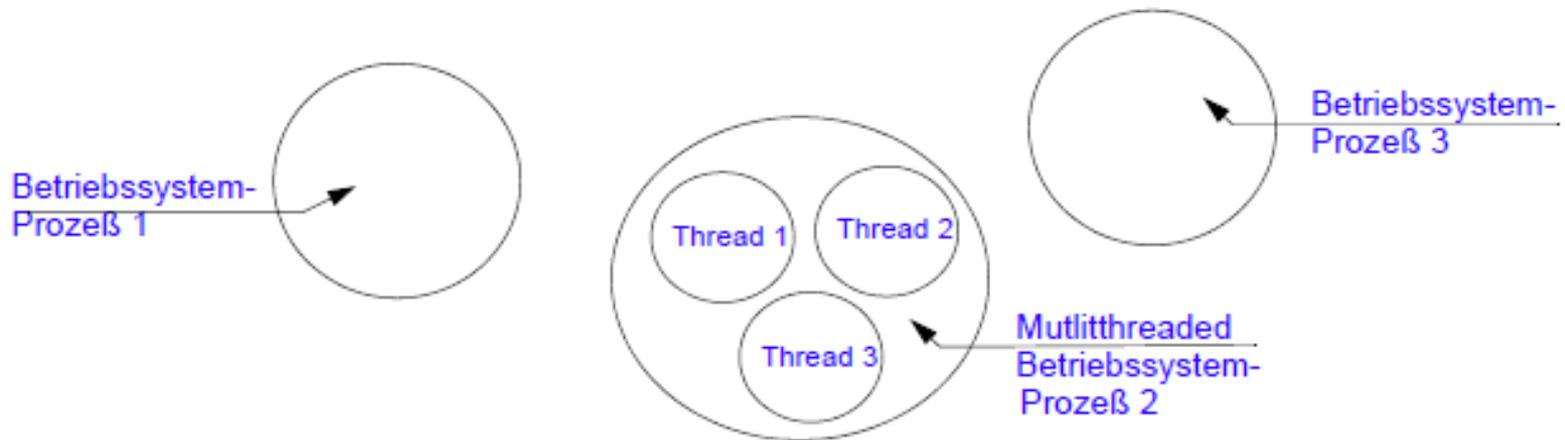


- Wenn der Scheduler einen anderen Prozess zur Ausführung bringen will, dann findet ein sogenannter Kontextwechsel statt.
- Wegen des hohen Aufwands für den Kontextwechsel wird ein (Betriebssystem-) Prozess auch als schwergewichtiger Prozess bezeichnet.

Leichtgewichtige Prozesse: Threads

- Ein klassischer Betriebssystem-Prozess stellt eine Einheit sowohl für die Speicherverwaltung (Memory Management) als auch für das Scheduling dar.
- Ein Thread stellt nur eine Einheit für das Scheduling dar, so dass nun innerhalb eines Betriebssystems-Prozesses mehrere Threads ablaufen können.

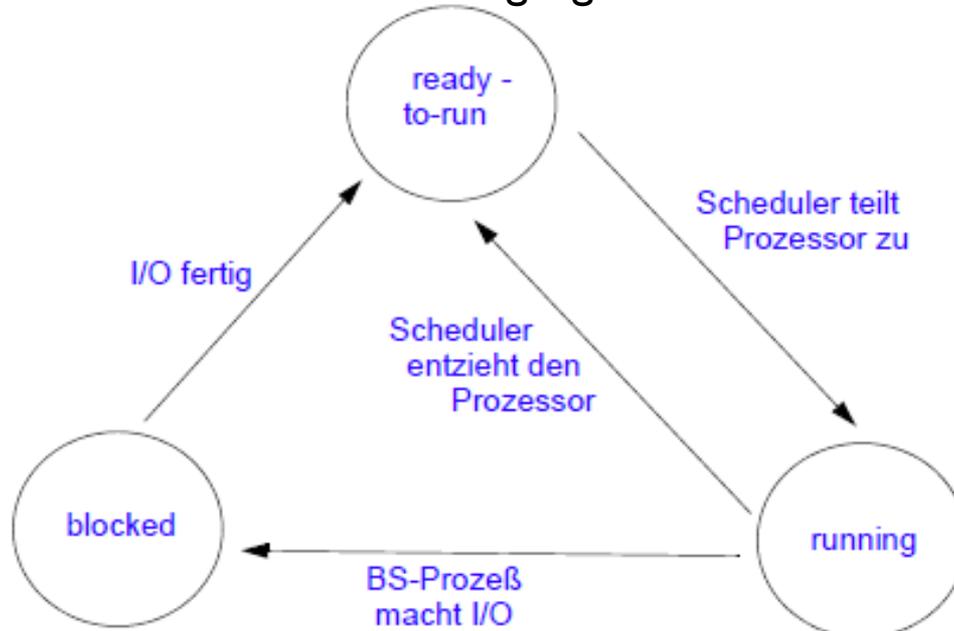
Leichtgewichtige Prozesse: Threads



- Während dem Betriebssystem-Prozeß der Speicher zugeordnet ist und ein Kontextwechsel mit Aufwand beim Memory Management verbunden ist, ist ein Thread-Wechsel auf der CPU nicht mit der Verwaltung des Speichers gekoppelt und daher viel schneller realisierbar.

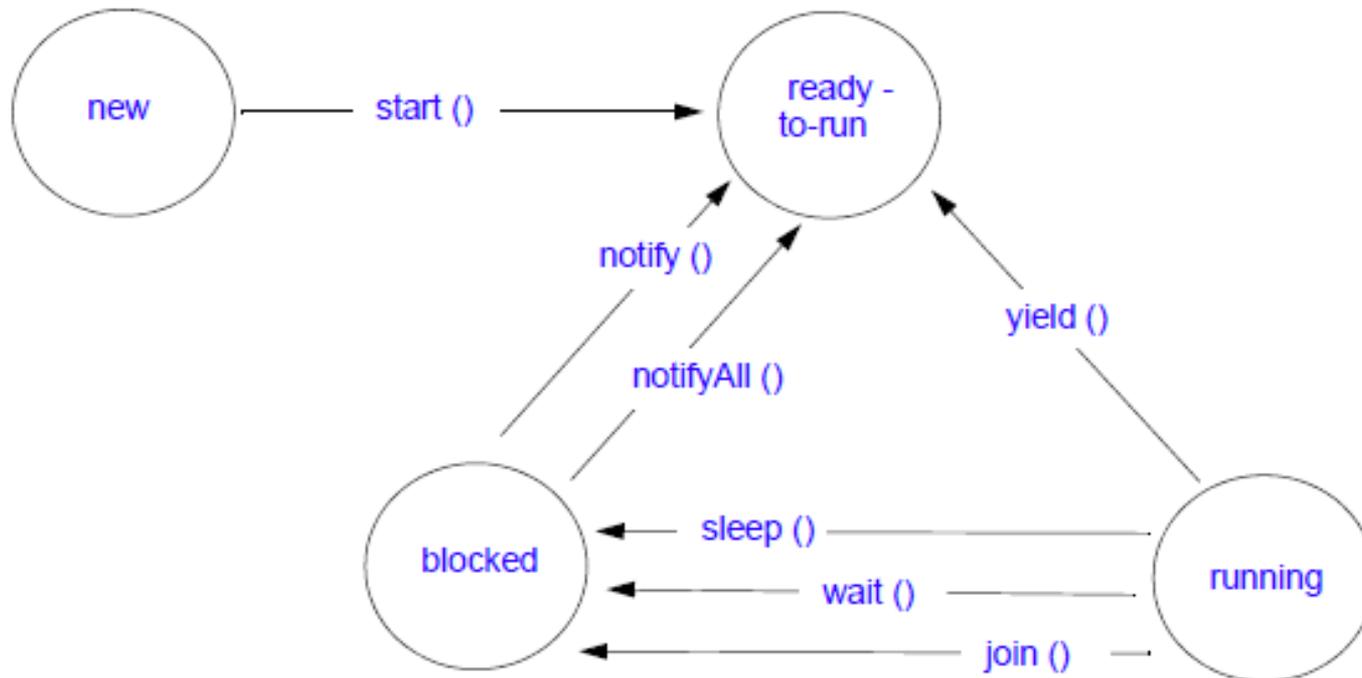
Zustandsübergabe von BS-Prozessen

- Prozesse haben Zustände, die von der aktuellen Zuordnung der Betriebsmittel abhängen.
- Im Bild wird ein vereinfachtes Diagramm dargestellt:
 - ⇒ Kreise stellen die Zustände eines Betriebssystem-Prozesses dar,
 - ⇒ Pfeile kennzeichnen die Übergänge zwischen den Zuständen



(User) Zustandsübergänge von Threads

- Threads haben ähnlich wie BS-Prozesse verschiedene Zustände. Die entsprechenden Übergänge erfolgen als Konsequenz von Methodenaufrufen wie z.B. `sleep ()` aber auch durch Aktionen des Betriebssystems wie z.B. die Zuteilung des Prozessors durch den Scheduler.



Programmierung von Threads

- Im Gegensatz zu den meisten anderen Programmiersprachen sind Threads bereits im Sprachumfang enthalten und lassen sich daher sehr einfach programmieren, erzeugen und starten.
- In Java gibt es zwei Möglichkeiten, einen Thread zu programmieren:
 - ⇒ Durch eine direkte Ableitung von der Klasse *Thread* oder
 - ⇒ durch die Übergabe eines Objekts, dessen Klasse die Schnittstelle *Runnable* implementiert, an ein Objekt der Klasse *Thread*.

Programmierung von Threads

- Im ersten Fall wird eine eigene (Thread-) Klasse geschrieben und von der Klasse *java.lang.Thread* abgeleitet.
 - ⇒ Dabei muss die Methode *run ()* der Klasse *java.lang.Thread* überschrieben werden,
 - ⇒ Der in der Methode *run ()* enthaltenen Code wird während des „running“-Zustandes ausgeführt.
- Im zweiten Fall wird die Schnittstelle *Runnable* in der Klasse implementiert, die zum Thread werden soll.
 - ⇒ Die Schnittstelle *Runnable* deklariert nur eine einzige Methode *run ()*.
 - ⇒ So schafft man sich die Möglichkeit, dass diese Klasse von einer anderen Klasse (außer *java.lang.Thread*) abgeleitet werden kann.

Ableiten der Klasse Thread: TestThread

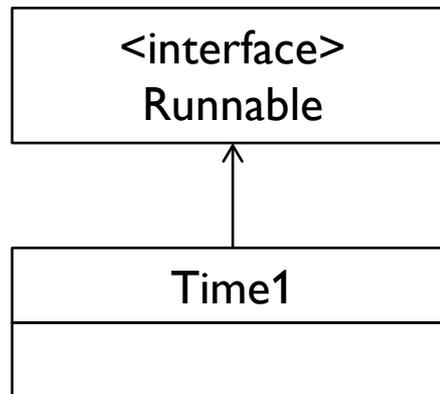
```
import java.util.Calendar;
import java.util.GregorianCalendar;
class Time extends Thread {
    public void run () {
        while (true) {
            GregorianCalendar d = new GregorianCalendar (); // aktuelle Zeit
            System.out.print (d.get(Calendar.HOUR_OF_DAY) + ":");
            System.out.print (d.get(Calendar.MINUTE) + ":");
            System.out.println (d.get(Calendar.SECOND));
            try { sleep(1000); // try
                } catch (InterruptedException e) { } // catch
        } // while
    } // run
} // class Time

public class TestThread {
    public static void main (String args []) {
        Time t = new Time ();
        t.start ();
        . . . // Hier könnten weitere Threads erzeugt und gestartet werden!!
    } // main
} // class TestThread
```

Implementieren der Schnittstelle Runnable

■ Thread-Erzeugung:

- ⇒ Mit dem new-Operator eine Instanz der Klasse `java.lang.Thread` generieren
- ⇒ Als Übergabeparameter beim Konstruktor eine Referenz auf ein Objekt mitgibt, dessen Klasse die Schnittstelle `Runnable` implementiert



Implementieren der Schnittstelle Runnable

- Innerhalb der Klasse Thread wird die übergebene Referenz in einem privaten Datenfeld vom Typ Runnable abgelegt, wie folgender Ausschnitt aus Thread-Implementierung zeigt:

```
public class Thread {  
  
    private Runnable target;  
    . . .  
    public Thread (Runnable target) {  
        . . .  
        this.target = target;  
        . . .  
    } // Konstruktor  
  
} // class Thread
```

Implementieren der Schnittstelle Runnable: Test1Thread

```
import java.util.Calendar;
import java.util.GregorianCalendar;
class Time1 implements Runnable {
    public void run () {
        while (true) {
            GregorianCalendar d = new GregorianCalendar ();
            System.out.print (d.get(Calendar.HOUR_OF_DAY) + ":");
            System.out.print (d.get(Calendar.MINUTE) + ":");
            System.out.println (d.get(Calendar.SECOND));
            try { Thread.sleep(1000); // Methode sleep ist eine Klassenmethode
                                     // der Klasse Thread
            } // try
            catch (InterruptedException e) { } // catch
        } // while
    } // run
} // Time1

public class Test1Thread {
    public static void main (String args []) {
        Thread t = new Thread (new Time1());
        t.start();
    } // main
} // class Test1Thread
```

System-Threads

- Besonders erwähnenswert erscheint noch die Tatsache, dass die Mehrzahl der Threads nicht explizit vom Anwender erzeugt und gestartet werden, sondern implizit vom System
- Beispielprogramm:

```
⇒ import ...  
public class TestThreadOutput  
    public static void main (String[] args) {  
        System.out.println ("Alle Threads beim Start von main");  
        ThreadTree.dump ( );  
        System.exit (0);  
    } // main  
} // TestThreadOutput
```

System-Threads

- Mit Beispielprogramm kann folgende Ausgabe produziert werden:
 - ⇒ Alle Threads beim Start von main
Thread[Reference Handler,10,system]
Thread[Finalizer,8,system]
Thread[Signal Dispatcher,9,system]
Thread[Attach Listener,5,system]
Thread[main,5,main]
- Es gibt also insgesamt 5 Threads, wobei nur der letzte Thread vom Anwender explizit generiert wurde. Alle anderen Threads wurden implizit vom System gestartet

Klasse ThreadTree

- Java-Programm, welches alle Threads ausgibt:

```
public class ThreadTree {  
  
    public static void dump ( )    {  
  
        ThreadGroup application =  
        Thread.currentThread().getThreadGroup ( );  
        ThreadGroup system = application.getParent ( );  
  
        int numberOfThreads = system.activeCount ( );  
        Thread[] threadList = new Thread [numberOfThreads];  
        int threadNumber = system.enumerate (threadList);  
  
        for (int i = 0; i < threadNumber; i++) {  
            System.out.println (threadList [i] );  
        } // for  
  
    } // dump  
} // ThreadTree
```