# Nebenläufige und verteilte Programme CS2301


Th. Letschert

TH Mittelhessen Gießen

*University of Applied Sciences*


## Threads – Leichtgewichtige Prozesse

- Lebenszyklus
- Unterbrechungen
- Thread-Management

## Threads – Einsatz

**Threads sind leichtgewichtige Prozesse**
**die in einer Anwendung eingesetzt werden :**

- **zum kooperativen Multitasking**

  **Multitasking**
  **Threads spiegeln die Nebenläufigkeit der Anwendung, dienen der Reaktivität des Systems**

  **Kooperativ**
  **Das Missverhalten eines Threads ist ein Programmierfehler,**
  **keine Sicherheitslücke**

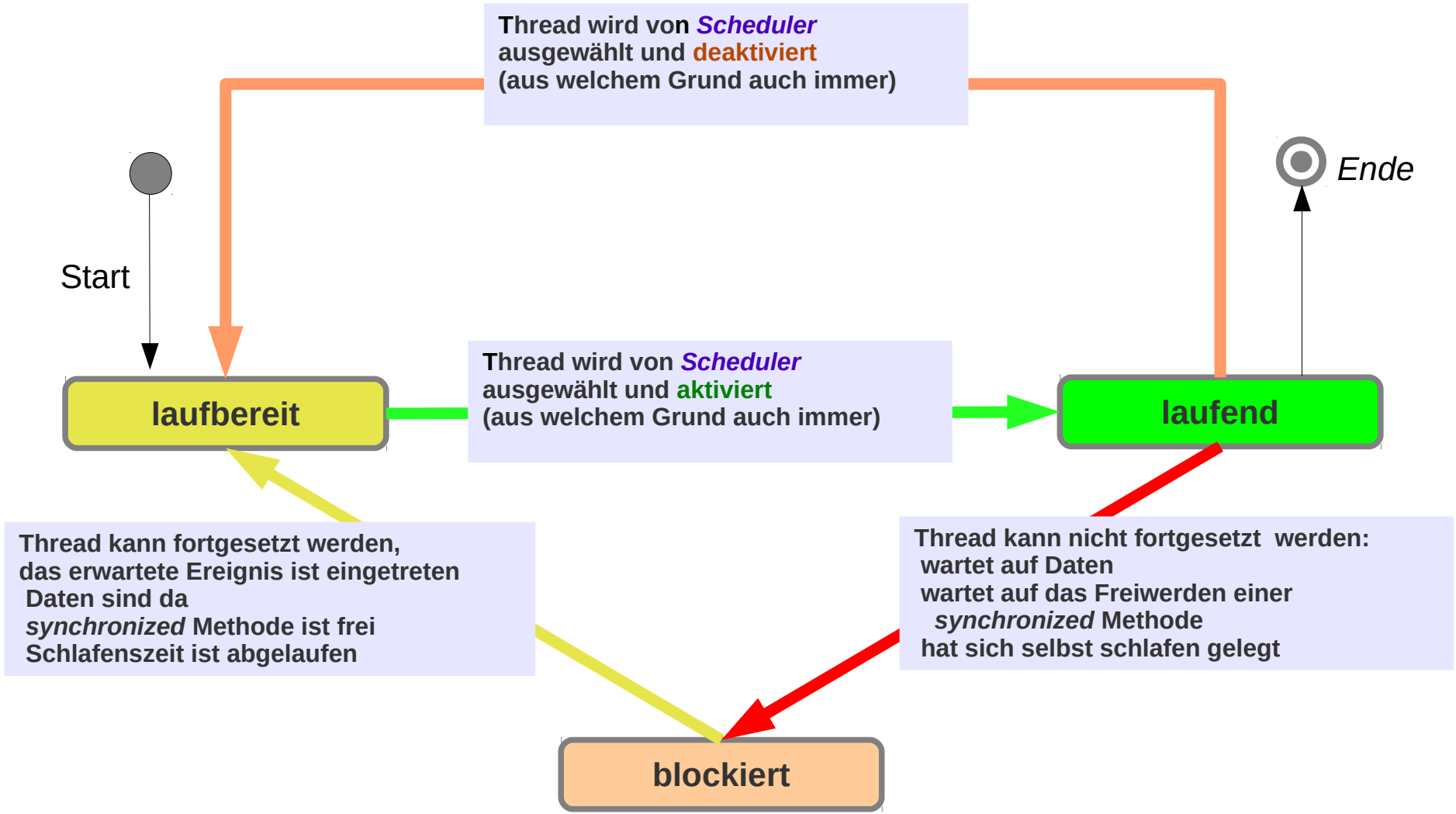
- **zur Parallelisierung**

  **Threads ermöglichen die effektive Nutzung der Hardware**

## Threads – Technisch

- Jeder Thread
  - hat seinen eigenen Laufzeitstack
  - teilt sich alle anderen Ressourcen (Heap, …) mit allen anderen Threads
  - hat einen Lebenszyklus
  - und einen aktuellen Zustand

## Threads – Zustände



## Scheduling

Der Scheduler verwaltet die Zustände der Threads

Der Scheduler

- ist Bestandteil der JVM
- bringt die Threads in ihre jeweiligen Zustände
  - laufend <~> laufbereit wird vom Scheduler entschieden
  - blockiert <~> laufbereit ergibt sich aus den Aktionen des Programms

## Scheduling

**Das Scheduling sollte vom Programmierer ignoriert werden.**

- **Das Programm sollte unabhängig vom Scheduling korrekt arbeiten**
- **Das System kann die Scheduling-Entscheidungen zur Laufzeit besser entscheiden, als der Programmierer im Voraus**

## Scheduling und Determinismus

**Programme mit Threads sind – mit Absicht – nichtdeterministisch**

- **Der genaue Ablauf wird nicht vom Programm (-ierer) bestimmt**
- **Die aktuelle Ausführungssituation kann bei der Ausführung berücksichtigt werden**
  - **Zahl und Auslastung der Prozessoren**
  - **Verfügbarkeit von Daten (von externem Speicher, Netzwerk, …)**
- **Der Programmierer muss irrelevante / unbekannte Details nicht berücksichtigen**

# Thread-Lebenszyklus

## Erzeugung und Start

**Jeder JVM-Prozess erzeugt diverse Threads**

- **Main-Thread:**
  **Der Thread der die main-Methode der Hauptklasse ausführt**
- **System-Threads der JVM**
  **Grabage-Collector-Threads, Compiler-Threads, Event-Dispatcher-Threads, ...**

**Beispiele Main-Thread:**

```
object Thread_Creation_App extends App {

  val mainThread: Thread = Thread.currentThread()

  println(s"mainThread name: ${mainThread.getName}, state: ${mainThread.getState}")
}
```

# Thread-Lebenszyklus

## Erzeugung und Start

**Threads in Scala (wie in Java)**

- **sind Instanzen der Klasse Thread**
- **werden mit der Methode *start* gestartet**

**Beispiele Erzeugung und Start weiterer Threads**

```
class MyThread extends Thread {

  override def run() {
    for (i <- 0 to 100) {
       println("hello No " + i);
       Thread.sleep(100)
    }
  }

}

object ThreadCreation_Main extends App {
    new MyThread start
}
```

***Unterklasse von Thread definieren***

```
new Thread(new Runnable{
   override def run() : Unit = {
      for (i <- 0 to 100) {
         println("hello No " + i);
         Thread.sleep(100)
      }
    }
}) start
```

***Thread mit Runnable als Konstruktor-Argument***

```
def thread(runCode: => Unit) : Thread =
    new Thread(new Runnable{
      override def run() : Unit = runCode
    }
)

thread( {
      for (i <- 0 to 100) {
           println("hello No " + i)
           Thread.sleep(100)
      }
  }) start
```

***Mit einer Hilfsfunktion für mehr Bequemlichkeit***

# Thread-Lebenszyklus

## Erzeugung und Start

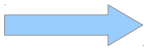
**Beispiele Erzeugung und Start weiterer Threads**

**Ab Scala 2.12 werden SAM-Interfaces in Scala unterstützt**

**SAM: Single Abstract Method, Lambda-Ausdrücke werden automatisch in Instanzen von Klassen / Interfaces umgewandelt, die eine einzige abstrakte Methode enthalten.**

**Beispiel: Runnable hat eine abstrakte Methode run.**

```
new Thread(new Runnable{
   override def run() : Unit = {
      for (i <- 0 to 100) {
         println("hello No " + i);
         Thread.sleep(100)
      }
    }
}) start
```



```
new Thread( () =>
      for (i <- 0 to 100) {
         println("hello No " + i);
         Thread.sleep(100)
      }
) start
```

# Thread-Lebenszyklus

## yield

**yield ist ein Hinweis an der Scheduler, dass der Thread bereit ist von *running* in den Zustand *ready to run* zu wechseln.**

```
thread({
    for (i <- 0 to 100) {
       println("hello No " + i);
       Thread.`yield`
    }
  }) start

thread({
    for (i <- 'a' to 'z') {
       println("hello No " + i);
       Thread.`yield`
    }
  }) start
```

***Da yield ein Schlüsselwort in Scala ist, muss es mit `…` in einen Bezeichner zurück verwandelt werden.***

***Achtung: yield ist ein Hinweis, der Scheduler kann, aber muss ihn nicht befolgen.***

## join

**mit join kann auf das Ende eines anderen Threads gewartet werden.**

```
val subThread1 = new Thread(new Runnable{
 override def run() : Unit = {
    for (i <- 0 to 100) {
       println("hello No " + i);
       Thread.sleep(100)
    }
  }
})

new Thread(new Runnable{
 override def run() : Unit = {
    println(s"${Thread.currentThread.getName} waits for ${subThread1.getName}" )
    subThread1.join()
    println(s"${subThread1.getName} is finished" )
  }
}).start()

subThread1.start()
```

```
Thread-1 waits for Thread-0
hello No 0
hello No 1
…
hello No 98
hello No 99
hello No 100
Thread-0 is finished
```

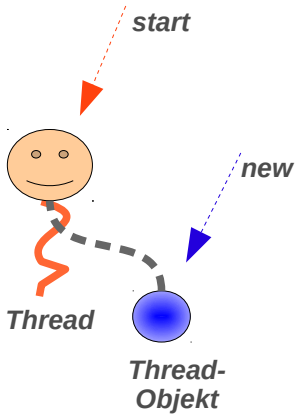
*Join-Beispiel: Ein Thread wartet auf das Ende eines anderen Threads*

# Thread-Lebenszyklus

## Threads, Thread-Klassen und Thread-Objekte

**Threads werden erzeugt als Instanzen der Klasse Thread**

- **Eine Instanz kann gestartet werden,**
- **durchläuft dann in unterschiedlichen Zustanden seinen Lebenszyklus**
- **und endet schließlich**
  **i.A. wenn die Kontrolle das Ende von *run* erreicht**
  **(Main-Thread wartet auf das Ende aller „*Nicht-Dämonen-Threads*“)**




***Threads sind immer mit einer Thread-Instanz verknüpft***

**Ein Thread pro Thread-Objekt**

- **Eine Thread-Instanz kann nur einen Lebenszyklus durchlaufen**
- **d.h. z.B.: Ein Thread-Objekt kann nicht mehrfach gestartet werden**

```
val t = new Thread(new Runnable{
 override def run() : Unit = {
    for (i <- 0 to 100) {
       println("hello No " + i);
       Thread.sleep(100)
    }
  }
})

t.start()

t.start()
```

```
Exception in thread "main" java.lang.IllegalThreadStateException
```

# Thread-Lebenszyklus

## Blockade

Im *blockierten* Zustand wartet der Thread auf das Eintreffen eines bestimmten Ereignisses. Nach dessen Eintreffen wird er in der Zustand *laufbereit* versetzt.

Der Zustand *blockiert* wird durch Aktionen des laufenden Threads erreicht:

- Thread.sleep
  Der Ablauf des Timers wird abgewartet.
- *someObject*.wait
  Die Freigabe einer mit einem Objekt assoziierten Bedingungsvariablen wird abgewartet
- *someThread*.join
  Das Ende eines Threads wird abgewartet.
- synchronized
  Java Schlüsselwort: synchronized (*someObject*)
  Scala Methode von *Object*: *someObject.synchronized*
  Die Freigabe des Mutex' der mit dem *this*-Objekt bzw. *anObject* assoziiert ist, wird abgewartet.
- I/O–Operationen
  I/O-Operationen blockieren bis die Operation ausgeführt ist.

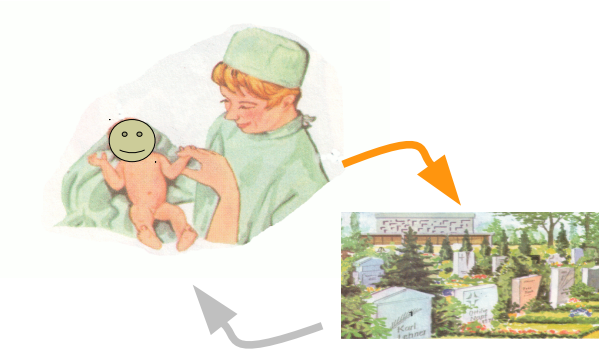
*Eine ausführliche Behandlung folgt später*

# Thread-Lebenszyklus

## Ende

**Ein Thread wird beendet, wenn**

- **die Kontrolle das Ende der *run*-Methode erreicht.**
- ***run* eine *Exception* wirft**
- ***run System.exit* aufruft**



## Test auf Lebendigkeit

**Mit isAlive kann getestet werden,**

**ob ein Thread noch aktiv ist.**

```
val myThread : Thread = thread({
  for (i <- 1 to 100) { Thread.sleep(100) }
})

myThread start

thread({
  while (myThread.isAlive) { println("myThread is still alive") }
  Thread.sleep(100)
  println("t is gone")
}) start
```

## Gruppen

**Threads sind in Thread-Gruppen. Ein Programm kann neue Thread-Gruppen erzeugen und Threads diesen Gruppen zuordnen.**

**Der Nutzen der Gruppen ist überschaubar:**

- **Man kann Informationen zu den Threads einer Gruppe abfragen**
- **alle Mitglieder einer Gruppe unterbrechen.**

## Gruppen

### Beispiel

```
object ThreadGroup_Main extends App {

  val threadGroup = new ThreadGroup("MyThreadgroup");
  new Thread(
       threadGroup,
       new Runnable {
         override def run() {
               for (i <- 0 to 100) {
               println("Hallo Nr " + i);
               try {
                  Thread.sleep(100);
               } catch  {
                  case e : InterruptedException => println("Ups, someone stopped me"); return
               }
             }
         }
       }) start()

  Thread.sleep(200);
  println(s"{threadGroup.getName()} has ${threadGroup.activeCount()} active members")

  threadGroup.interrupt();

  Thread.sleep(200);
  println(s"{threadGroup.getName()} has ${threadGroup.activeCount()} active members")

}
```

# Thread-Unterbrechungen

## Interrupt-Flag

**Jeder Thread besitzt ein Interrupt-Flag**

**Dies Flag kann mit folgenden Methoden abgefragt und manipuliert werden:**

- `void interrupt()`

  **Das Interrupt-Flag wird gesetzt, danach unterschiedliches Verhalten:**

  - **Der Thread ist laufend oder laufbereit: Nichts weiter, der Thread hat die Verantwortung angemessen mit dem Flag umzugehen.**
  - **Der Thread ist blockiert: Eine InterruptedException wird geworfen, das Interrupt-Flag wird zurück gesetzt**

- `boolean isInterrupted()`

  **Prüft das Interrupt-Flag und liefert seinen Wert, ohne es zu verändern**

- `static boolean interrupted()`

  **Prüft das Interrupt-Flag des aktuellen Threads, liefert seinen Wert und setzt es zurück**

# Thread-Unterbrechungen

## Interrupt-Methode

**Die Interrupt-Methode `void interrupt()`**

- **dient dazu, einen Thread über einen Unterbrechungswunsch zu informieren**
- **die Methode setzt dazu das Interrupt-Flag auf true**
- **Ist der Thread blockiert, dann kan er nicht selbst reagieren,**
  **Statt dessen wird die blockierende Aktionen (wait, sleep, join) von der JVM abgebrochen**
  - **das Flag zurück gesetzt und**
  - **die InterruptedException geworfen**
- **Achtung: befindet sich der Thread nicht in einer blockierenden Aktion, dann wird lediglich das Flag gesetzt. Der (Programmierer des) Thread(s) muss geeignet reagieren**

# Thread-Unterbrechungen

## Interrupt-Methode / Beispiel : Interrupt in „normalem Code“

```
val myThread = thread (
  {
    var l = 0.0
    var i = 0
    while (!Thread.currentThread().isInterrupted()) {
       for (j <- 0 to 1000)
         for (k <- 0 to 10000) { l = l + (k*0.5 / 100.0); }
       print(".");
       i = i+1
    }
    println(s"\ninterrupted with i = $i")
    println("Flag after loop: "
          + Thread.currentThread().isInterrupted());
});

myThread.start();
Thread.sleep(1000);

myThread.interrupt();
```

*interrupt setzt das Interrrupt-Flag*

```
..............
interrupted with i = 14
Flag after loop: true
```

*Mögliche Ausgabe*

*interrupt in einer nicht-blockierenden Aktion setzt nur das Flag. Es muss explizit abgefragt werden.*

## Interrupt-Methode / Beispiel : Interrupt in blockierender Aktion

```
val sleepingThread : Thread = thread({
     var i = 0
     while (!Thread.currentThread().isInterrupted()) {
       try {
         Thread.sleep(1000);
       } catch {
         case e: InterruptedException => {
           println("Flag in catchBlock: " + Thread.currentThread().isInterrupted()); // => false
           Thread.currentThread().interrupt()  // set flag to stop loop
         }
       }
    }
    println("Flag after loop "
            + Thread.currentThread().isInterrupted()); // => true
})

sleepingThread start

Thread.sleep(100);

sleepingThread.interrupt();
```

```
Flag in catchBlock: false
Flag after loop true
```

***interrupt in einer blockierenden Methode löst eine Exception aus und setzt das Flag zurück. Eventuell will man dann das Flag explizit setzen.***

# Thread-Unterbrechungen

## InterruptedException / Beispiel :

**Niemals so in Code, der in einem Thread eingesetzt werden könnte.**

```
try {
  wait();
   queue.put();
  .. etc ..
} catch (InterruptedException e) {
  e.printStackTrace();
}
```

***Der ausführende Thread kann nicht via interrupt gestoppt werden. Es stürzt statt dessen alles ab.***

```
try {
  wait();
   queue.put();
  .. etc ..
} catch (InterruptedException e) {
  /* ignore */
}
```

***Der ausführende Thread kann nicht via interrupt gestoppt werden.***

## Abbruch eines Threads: Thread von außen beenden

**Thread-Interrupt und das Abbrechen einer Aufgabe**

- **Interrupts auf Threads sind Techniken**
- **Das Abbrechen einer Aktivität (*Cancellation*) ist eine Aufgabe**

**Aufgabe und Thread: Die Aufgabe kann (muss aber nicht) einem Thread zugeordnet sein**

**Interrupt und Abbrechen : Das Abbrechen kann (muss aber nicht) durch Interrupts realisiert werden**

**Abbruch durch Interrupts zu implementieren ist eine sinnvolle Implementierung-Entscheidung**

## Thread-Abbruch: Thread von außen beenden

### Beispiel

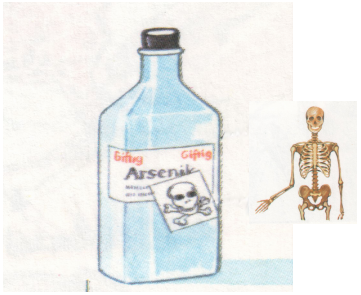
```
class MayBeCanceled extends Thread { // thread calling long running function

  override def run(): Unit = {
    var n : Long = 38921
    while ( n < 38921*38921 && !Thread.currentThread().isInterrupted()) {
      try {
          print(s"Factorizing $n to ")
          println(Factorization.factors(n)) // may throw InterruptedException, flag is cleared
          n = n*100
       } catch {
         case e: InterruptedException => {
           interrupt(); // break loop, set flag
         }
       }
    }
    // check flag: flag is set <=> loop was interrupted
    if (isInterrupted()) {
        Thread.interrupted(); // clear flag
        println("Thread was canceled") // some final action in case thread was canceled
    } else {
        println("Thead ended without being canceled") // some final action
    }
  }

  def cancel(): Unit = interrupt()

}
```

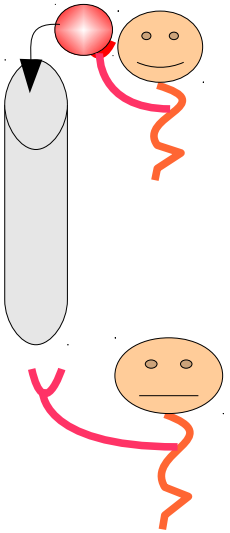
## Abbruch: Thread von außen beenden




### Giftpille

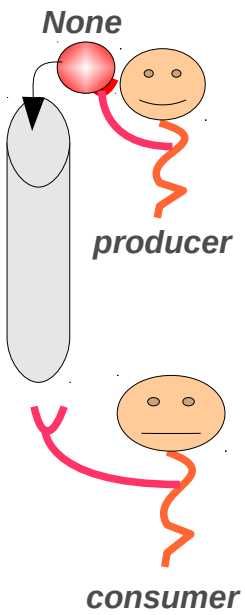
**Abbruch-Technik:**

**Senden einer besonderen („vergifteten“) Nachricht**



# Thread-Abbruch

## Giftpille




```
// producer may be canceled
class Producer(queue: BlockingQueue[Option[List[Long]]]) extends Thread {
  override def run(): Unit = {
      var n : Long = 38921
      while(n < 38921*38921 && !Thread.currentThread().isInterrupted()) {
        try {
          queue.put(Some(Factorization.factors(n)))
          n = n*10
        } catch {
          case e: InterruptedException => {
            interrupt(); // break loop
          }
       }
    }
    if (isInterrupted()) {
      Thread.interrupted()
      println("Producer was canceled")
      queue.put(None)
    } else {
      println("Producer ended normally")
      queue.put(None)
    }
  }

  def cancel: Unit = interrupt()
}
```

```
// consumer may be stopped only by poison pill
class Consumer(queue: BlockingQueue[Option[List[Long]]]) extends Thread {
  override def run() : Unit =
    while (true) {
      queue.take() match {
        case None    => {println("Consumer is stopped"); return }  // None is poison
        case Some(l) => println(l)
      }
    }
}
```

## Abbruch in I/O-Operation

### Blockierende I/O

**Abbruch mit interrupt nicht (unbedingt) möglich: Verwende andere Operation, die zum Abbruch führt. Z.B. *close* zum Abbruch einer Lese-Operation**

# Thread-Abbruch

## Abbruch in I/O-Operation Beispiel

```
import java.io.{InputStream, IOException}
import java.net.Socket

class SocketReader(val socket: Socket) extends Thread {
  private val in : InputStream = socket.getInputStream()

  def cancel(): Unit = {
       try {
          socket.close();
       } catch {
         case e: IOException =>  /* ignore */
       } finally {
         interrupt(); // if thread is blocked but not in IO.op
     }
    }

    override def run(): Unit = {
        try {
            val buf: Array[Byte]  = new Array[Byte](256)
            var break: Boolean = false
            while (!break) {
                val count = in.read(buf)
                if (count < 0) {
                    break = true
                } else if (count > 0)
                    processBuffer(buf, count);
            }
        } catch {
          case e: IOException => /* finished */
        }
    }
                                          def processBuffer(buf: Array[Byte], count: Int) : Unit = { … }
}
```

# Thread-Abbruch

## Abbruch eines Threads

### Schnittstelle

- Cancel-Methode
- interrupt
- ...

### Technik

- Boolesches Flag
- Interrupt
- Giftpille
- andere Methode
- …

### Semantik

- Was ist die Wirkung eines Abbruchs

## Unterbrechung von Threads und Methoden

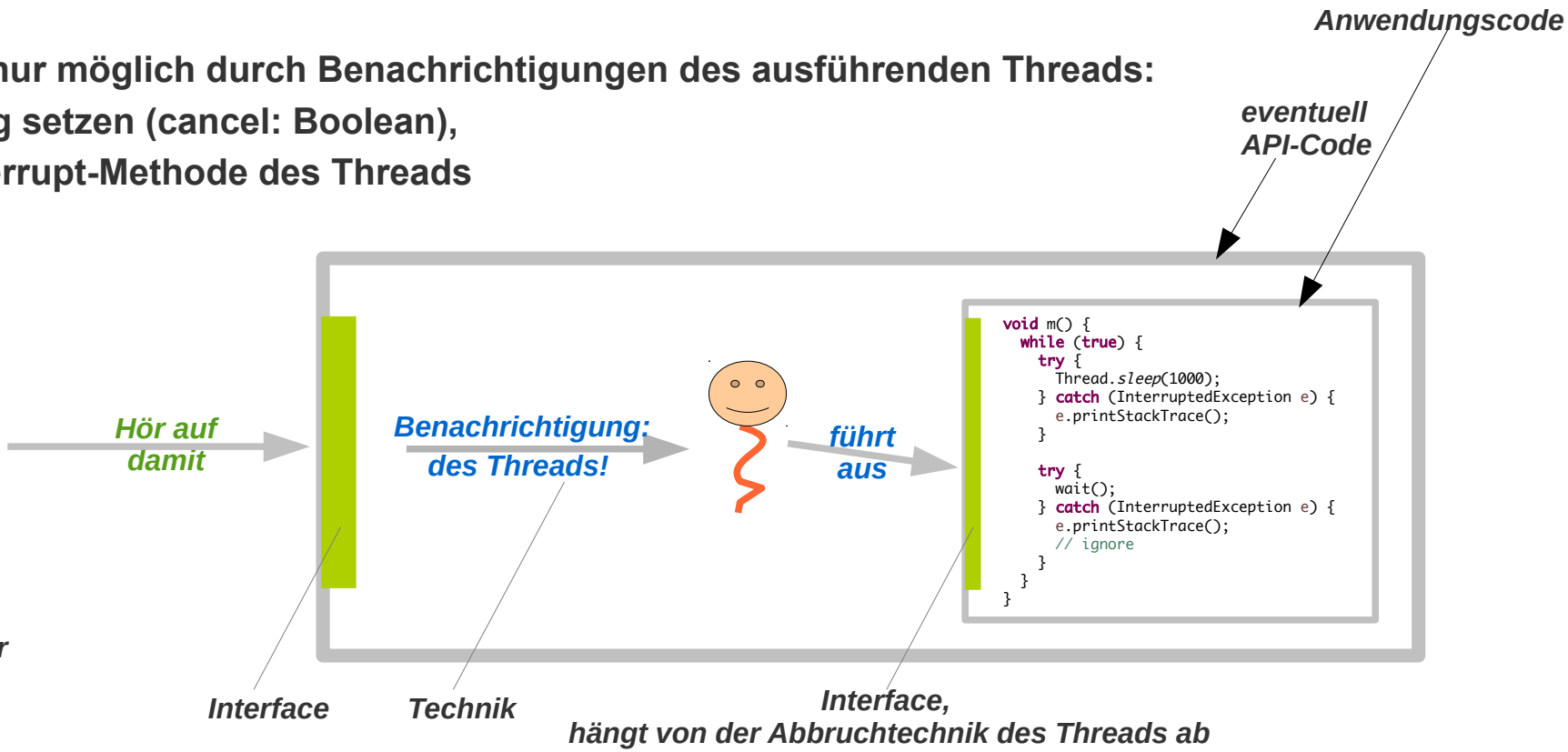
**Threads führen Methoden aus**

**Die Ausführung einer Methode soll gelegentlich unterbrochen werden:**

- **Dauert zu lange (Benutzer an GUI verliert Geduld / Timeout)**
- **Etwas geht schief**
- **Voraussetzungen zur Ausführung sind nicht gegeben**
- **…**

**Abbruch nur möglich durch Benachrichtigungen des ausführenden Threads:**

- **Flag setzen (cancel: Boolean),**
- **interrupt-Methode des Threads**
- **…**



# Abbruch von Methoden

## Interrupt-Strategie von Methoden

**Mache Threads in konsistenter Weise unterbrechbar!**

**Für jede Methode die (eventuell) in einem Thread verwendet wird:**

- **Langlaufende Aktionen sollten das Interrupt-Flag testen**
- **Die InterruptedException und das Flag sollten systematisch / konsistent eingesetzt werden:**
  **beispielsweise:**
  - **das Flag wird zurückgesetzt**
  - **eine unterbrochene Methode wirft / propagiert die InterruptedException**

  **oder:**
  - **das Flag wird nicht zurückgesetzt**
  - **wird ein blockierenden Aufruf unterbrochen – der dann die InterruptedException wirft und das Flag zurück setzt – dann wird es durch `interrupt` – neu gesetzt.**
- **Hilfs- und Bibliotheksfunktionen sollten ein `interrupt` immer in irgendeiner Form respektieren und propagieren (Flag / Execption / ...) und niemals „weg-schlucken“**
  **Die endgültige Verarbeitung der Aufforderung zum Interrupt erfolgt im benutzenden Code spätestens in der run-Methode.**
- **jede (!) ernsthafte Funktion sollte eine klare (und eventuell dokumentierte) Interrupt-Behandlung bieten.**

# Abbruch von Methoden

## Interrupt-Strategie einer Methode Beispiel 1:

*Strategie:*
*interrupt =bewirkt=>*
*- Exception,*
*- und Flag ist gesetzt.*

```
def m(...): ... = {
  while (! Thread.currentThread().isInterrupted()) {
    ...
    try {
       wait() / ...
    } catch {
       case e: InterruptedException =>
            Thread.currentThread().interrupt()
            throw new InterruptedException()
    }
    ...
  }
  ...
  if (Thread.currentThread().isInterrupted()) {
    throw new InterruptedException();
  } else {
   // some „normal“ final action
  }
}
```

*Interrupt in blockierender Aktion: bemerke Exception, setze Flag*

*Interrupt in normaler Aktion: bemerke Flag, werfe Exception*

# Abbruch von Methoden

## Interrupt-Strategie einer Methode Beispiel 2:

```
def m(...): ... = {
  while (! Thread.currentThread().isInterrupted()) {
    ...
    try {
       wait() / ...
    } catch {
       case e: InterruptedException =>
            Thread.currentThread().interrupt()
    }
    ...
  }
  ...
  if (isInterrupted()) {
   Thread.interrupted(); // clear flag
   // some final action in case thread was canceled
  } else {
   // some „normal“ final action
  }
}
```

***Strategie:***
***interrupt =bewirkt=>***
***- Ende der Methode ohne Exception, ohne gesetztes Flag.***
***- aber mit speziellem Ergebnis***

***Interrupt in blockierender Aktion:setze Flag***

***setze Flag zurück***
***werfe Exception***

**Mach das nicht!**
***Wenn die Methode in einem Thread ausgeführt werden könnte, der nicht von dir kontrolliert wird.***
***Die Unterbrechung könnte für den fremden Thread relevant sein, wird aber von Deinem Code geschluckt.***

# Abbruch von Methoden

## Interrupt-Strategie einer Methode Beispiel 3:

*Strategie:*
*interrupt =bewirkt=>*
*– Ende der Methode mit spezieller Exception und*
*– mit gesetztem Flag.*

```
def m(...): ... = {
  while (! Thread.currentThread().isInterrupted()) {
    ...
    try {
       wait() / ...
    } catch {
       case e: InterruptedException =>
           throw CanceledException
    }
    ...
  }
  ...
  if (isInterrupted()) {
    throw CanceledException
  } else {
   // some „normal“ final action
  }
}
```

```
object CanceledException extends Exception
```

# Abbruch von Methoden

## Interrupt-Strategie einer Methode Beispiel 4:

```
import scala.util.{Try, Success, Failure}

private def isPrime(n: Long): Boolean =
  Range.Long(2L, n/2+1, 1).count(n % _ == 0) == 0

case object CanceledException extends Exception

def factors(n: Long): Try[List[Long]] =
  if (n < 2) throw new IllegalArgumentException()
  else {
    try {
      Success(
        Range.Long(2L, n/2+1, 1)
        .filter( (i: Long) => {
          if (Thread.currentThread().isInterrupted()) {
            throw CanceledException
          }
          n%i == 0 && isPrime(i);
        }).toList)
    } catch {
      case t: Throwable => Failure(t)
    }
  }
```

***Strategie:***
***interrupt =bewirkt=>***
- ***Ende der Methode mit Failure-Ergebnis und***
- ***mit gesetztem Flag.***

*Die Methode liefert ein Try-Objekt mit dem Ergebnis oder eine Failure-Objekt mit dem Grund warum kein Ergebnis berechnet werden konnte.*

# Abbruch von Methoden

## Abgebrochene Methoden

**Interface einer Methode die abbrechen / abgebrochen werden kann:**

**Eingabe: Wie wird die Abbruchwunsch mitgeteilt / angenommen**

**Dringende Empfehlung:**

**Abbruchwünsche sollten in einer Methode über *interrupt* angenommen werden**

- **Das Interrupt-Flag sollte gesetzt bleiben**

**Ausgabe: Wie reagiert die Methode auf einen Wunsch zum Abbruch**

- **Welches Ergebnis hat die abgebrochene Methode**
  - **null**
  - **besonderer Wert**
  - **Exception werfen**
  - **Option / None**
  - **Try / Failure**
  - **Left / Either**
  - **….**
- **Welchen Status hat das Interrupt-Flag**

  **Dringende Empfehlung:**

  **Das Interrupt-Flag sollte gesetzt bleiben**

# Abbruch von Methoden

## Exkurs: Eventuell fehlschlagende Methode in Scala

**Scala bietet einige Ausdrucksmöglichkeiten**

- **um dem Aufrufer einer Methode**
- **deren „unvorhergesehenes“ Ergebnis**

**zu signalisieren:**

- **Exception**
  **Exceptions sind in Scala nicht als reguläres Mittel zur Steuerung des Kontrollflusses vorgesehen (es gibt keine *checked Exceptions* in Scala)!**
- **Option**
  **Option: None oder ein richtiger Wert**
- **Either**
  **Either: Etwas von entweder diesem oder einem anderen Typ**
- **Try**
  **Ein Wert oder eine Exception**

## Exkurs: Option / Either / Try in Scala Beispiel

```
object MayFail_App extends App {

  def mayFail_Option(s: String) : Option[Int] =
    try {
      Some(s.toInt)
    } catch {
      case e: NumberFormatException => None
    }

  def mayFail_Either(s: String) : Either[String, Int] =
    try {
      Right(s.toInt) // usually Right is considered to be the right value
    } catch {
      case e: NumberFormatException => Left(s"NumberFormatException ${e.getMessage}")
    }

  import scala.util.{Try, Success, Failure}

  def mayFail_Try(s: String) : Try[Int] =
    Try(s.toInt)

  mayFail_Option(scala.io.StdIn.readLine("Give me a number ")) match {
    case None    => println("I said: a number")
    case Some(i) => println(s"OK, you typed $i")
  }

  mayFail_Either(scala.io.StdIn.readLine("Give me a number ")) match {
    case Left(msg) => println(msg)
    case Right(i)  => println(s"OK, you typed $i")
  }

  mayFail_Try(scala.io.StdIn.readLine("Give me a number ")) match {
    case Success(i) => println(s"OK, you typed $i")
    case Failure(t) => println(s"failed with $t")
  }
}
```

*... oder auch so ...*

```
import scala.util.control.Exception._

def mayFail_Option(s: String) : Option[Int] =
  catching(classOf[NumberFormatException]).opt(s.toInt)

def mayFail_Either(s: String) : Either[Throwable,Int] =
  catching(classOf[NumberFormatException]).either(s.toInt)
```

# Abbruch von Methoden

## Exkurs: Option / Either / Try in Scala

**Achtung: InterruptedException ist eine „fatale“ Exception, sie wird nicht von Try gefangen**

```
def m_Try () : Try[Int] = Try( throw new InterruptedException )

m_Try() match {
  case Success(s) => println(s"OK: $s")
  case Failure(t) => println(s"Failed: $t")
}
```

```
Exception in thread "main" java.lang.InterruptedException
```

## Exkurs: ExecutionException

**ExecutionException bei Methoden, die mit fatalen Execptions abgebrochen werden**

```
def m_Try () : Try[Int] = Try(
      try {
        throw new InterruptedException
      } catch {
        case e: InterruptedException => throw new ExecutionException(e)
    })

m_Try() match {
  case Success(s) => println(s"OK: $s")
  case Failure(t) => println(s"Failed: $t")
}
```

*So kann eine fatale Exception weiter gegeben werden.*

*Das ist keine Empfehlung zum generellen Umgang mit der InterruptedException.*

```
Failed: java.util.concurrent.ExecutionException: java.lang.InterruptedException
```

## Shutdown Hook

**Ein shutdown hook ist ein Thread der beim Herunterfahren der JVM aktiviert wird.**

**Er kann genutzt werden, um die „Aufräumarbeiten“ auszuführen, die beim Ende einer Anwendung anfallen**

**Ein shutdown hook**

- **wird progammatisch registriert**
- **automatisch aktiviert wenn**
  - **alle nicht-Dämon-Threads beendet wurden, oder wenn**
  - **System.exit, oder wenn**
  - **die JVM von außen beendet wird**

```
object ShutDownHook_Main extends App {

  Runtime.getRuntime().addShutdownHook(
      new Thread(new Runnable{
                override def run(): Unit = {
                    println("It's all over now!");
        }}));

  for(i <- 0 to 10) System.out.println("Still working..");
  println("good bye")
}
```

```
Still working..
Still working..
Still working..
Still working..
Still working..
Still working..
Still working..
Still working..
Still working..
Still working..
Still working..
good bye
It's all over now!
```

## Dämonen-Thread

**Ein Dämonen-Thread *daemon thread* ist ein Thread der die JVM nicht am Herunterfahren hindert.**

**Er dient dazu „Hintergrund-Aufgaben“ auszuführen.**

**Ein Thread wird zum Dämon-Thread, wenn ein entsprechendes Flag vor der dem Start (!) gesetzt wird.**

```
class DaemonThread extends Thread {

  override def run(): Unit = {
    while(true){
      println("Daemon is alive");
      Thread.sleep(1000)
    }
  }
}

object Daemon_Main extends App {

  val t : Thread = new DaemonThread
  //t.setDaemon(true)
  t start

  Thread.sleep(1000)

  println("Main stops")
}
```

```
Daemon is alive
Main stops
Daemon is alive
Daemon is alive
...
```

```
class DaemonThread extends Thread {

  override def run(): Unit = {
    while(true){
      println("Daemon is alive");
      Thread.sleep(1000)
    }
  }
}

object Daemon_Main extends App {

  val t : Thread = new DaemonThread
  t.setDaemon(true)
  t start

  Thread.sleep(1000)

  println("Main stops")
}
```

```
Daemon is alive
Main stops
```

## Prioritäten

Threads kann eine Priorität zugewiesen werden.

Die Priorität kann das Scheduling beeinflussen: Laufbereite Threads mit einer höheren Priorität werden von der JVM bevorzugt.

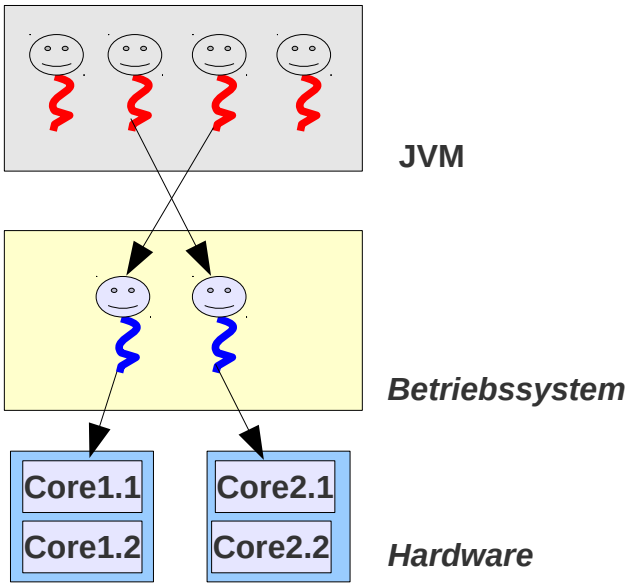
Moderen JVMs bilden Threads auf System-Threads des Betriebssystems ab, die diese wiederum auf CPUs und CPU-Kerne abbilden. Die Prioritäten der JVM haben dabei einen eher geringen Einfluss.

### Prioritäten: 10 Stufen

- 1 `java.lang.Thread.MIN_PRIORITY`
- ...
- 5 `java.lang.Thread.NORM_PRIORITY`
- ...
- 10 `java.lang.Thread.MAX_PRIORITY`

### Prioritäten setzen / abfragen

- `int getPriortity()`
- `void setPriority(int newPriority)`
- default-Wert: Priorität des startenden Threads




*Thread-Prioritäten sind, genau wie yield, Hinweise an die JVM. Das korrekte Verhalten einer Applikation sollte niemals darauf basieren, dass diese Hinweise befolgt werden! Bestenfalls sollten sie eingesetzt werden um letzte Optimierungen im Verhalten zu erreichen.*