Reaktive Programmierung

Wie verwendet man Threads in Scala?

Erstellen eines Threads:

```
class MyThread extends Thread:
    override def run(): Unit =
        println("Executed in new thread.")

val t = new MyThread
t.start()
```

Methode in Thread ausführen:

```
def doInThread(body: => Unit): Thread = // call-by-name
    val t = new Thread:
        override def run() = body

    t.start()
    t


val t = doInThread { println("Executed in new thread.") }
```

Wie kann man eine Race Condition in folgendem Code verhindern?

```
var currId = 0L
def generateUniqueId() =
    val newId = currId + 1 // (1)
    currId = newId // (2)
    newId

val t1 = doInThread { println(generateUniqueId()) }
val t2 = doInThread { println(generateUniqueId()) }
// Output: 1 2 or 2 1 or 1 1
```

Mit `synchronized`:

```
var currId = 0L
def generateUniqueId() =
    this.synchronized:
```

```
        val newId = currId + 1 // (1)
        currId = newId // (2)
        newId
```

Was macht `@volatile`?

damit definiert man atomic Variablen, die von mehreren Threads gleichzeitig gelesen und geschrieben werden können. Es wird sichergestellt, dass alle Threads immer den aktuellen Wert der Variablen sehen.

```
class Worker extends Thread:
    @volatile
    private var stopped = false
    override def run() =
        while !stopped do { /* do some work */ }
    def shutdown() = stopped = true
```

Was ist ein Executor?

- Executor entscheidet auf welchem Thread und wann ein Task ausgeführt wird
- Tasks müssen `Runnable` oder `Callable<T>` implementieren

Wie implementiert man einen Executor in Scala?

Es kann der Java-Executor direkt in Scala verwendet werden:

```
val executor = new java.util.concurrent.ForkJoinPool
executor.execute(
  () => println("Diese Aufgabe wird asynchron ausgeführt.")
)
```

Andere Möglickeit mit Scala spezifischen ExecutionContext:

```
val execCtx = scala.concurrent.ExecutionContext.global
execCtx.execute(
  () => println("Diese Aufgabe wird asynchron ausgeführt.")
)
```

Benutzerdefinierter ForkJoinPool:

```
val execCtx = scala.concurrent.ExecutionContext.fromExecutorService(
  new java.util.concurrent.ForkJoinPool(2)
)
```

Wie kann man atomare Variablen in Scala verwenden?

- Atomic-Variablen sind in `java.util.concurrent.atomic` definiert.

```
private val uid = new AtomicLong(0L)
def getUniqueId(): Long = uid.incrementAndGet()
```

- Operationen sind lock-free und schnell
- keine deadlocks
- basieren auf einer grundlegenden atomaren Operation: `compareAndSet`, gibt true zurück, wenn der alte Wert mit dem aktuellen Wert übereinstimmt und der neue Wert gesetzt werden kann. Andernfalls false.

```
private val currId = new AtomicLong(0L)
@tailrec def generateUniqueId(): Long =
  val oldId = currId.get
  val newId = oldId + 1
  if currId.compareAndSet(oldId, newId) then newId
  else generateUniqueId()
```

Was ist das Producer-Consumer-Problem, und wie hilft eine `BlockingQueue`, es zu lösen?

= klassisches Problem der Nebenläufigkeit

- Produzenten erzeugen Daten und legen sie in eine Warteschlange
- Konsumenten lesen Daten aus der Warteschlange

Herausforderung:

- Synchronisation zwischen Produzenten und Konsumenten
- Datenverlust oder Blockierung
- Effizienz

BlockingQueue:

- Thread-sichere Warteschlange
- mit automatischer Synchronisation - mehrere Threads können sicher auf die Warteschlange zugreifen
- Blockierende Operationen: `put` und `take`
- Verhindert Busy-Waiting

```
import java.util.concurrent._

object ProducerConsumerExample extends App {
```

```
  val queue = new LinkedBlockingQueue  // Begrenzte Kapazität von 5

  // Producer Thread
  val producer = new Thread(() => {
    for (i <- 1 to 10) {
      queue.put(i) // Blockiert, falls die Queue voll ist
      println(s"Produced: $i")
      Thread.sleep(500) // Simulierte Wartezeit
    }
  })

  // Consumer Thread
  val consumer = new Thread(() => {
    for (_ <- 1 to 10) {
      val item = queue.take() // Blockiert, falls die Queue leer ist
      println(s"Consumed: $item")
      Thread.sleep(1000) // Simulierte Verarbeitung
    }
  })

  producer.start()
  consumer.start()

  producer.join()
  consumer.join()
}
```

Was sind High-Order Functions?

= Funktionen, die andere Funktionen als Argumente akzeptieren

`sum` ist eine High-Order Function, die eine Funktion `f` als Argument akzeptiert und die Summe der Werte von `f` im Bereich `[a, b]` berechnet.

```
def sum(f: Int => Int, a: Int, b: Int): Int =
    if (a > b) 0 else f(a) + sum(f, a + 1, b)

def sumSquares(a: Int, b: Int): Int = sum(x => x * x, a, b)
def sumPowerOfTwo(a: Int, b: Int): Int = sum(Math.pow(2, _).toInt, a, b)
```

Was ist eine Monade?

Ein Typ M, der die drei Operationen `unit`, `flatMap` und `map` unterstützt.

Was ist eine `Future` und Future Callbacks?

Future = eine Abstraktion für asynchrone Berechnungen in Scala. Sie repräsentiert einen Wert, der möglicherweise noch nicht verfügbar ist, aber irgendwann in der Zukunft bereitgestellt wird.

Future Callbacks = Funktionen, die aufgerufen werden, wenn das Ergebnis einer Future verfügbar ist. Sie ermöglichen es, auf den Abschluss einer asynchronen Berechnung zu reagieren. zb. `onComplete`, `onSuccess`, `onFailure`.

```
  // Simuliert eine asynchrone Berechnung
  val futureResult: Future[Int] = Future {
    println("Berechnung startet...")
    Thread.sleep(2000) // Simulierte Verzögerung
    42 // Rückgabewert der Berechnung
  }

  // Reaktion auf das Ergebnis des Futures
  futureResult.onComplete {
    case Success(value) => println(s"Berechnung abgeschlossen: Ergebnis =
$value")
    case Failure(exception) => println(s"Fehler: ${exception.getMessage}")
  }

  println("Das Programm läuft weiter...")

  Thread.sleep(3000) // Warten, damit der Future abgeschlossen wird
```