Übungsaufgabe

Nebenläufigkeit und Parallelität in Scala:
Futures, ExecutionContext, Parallel Collections.

Universität:	Technische Universität Berlin
Kurs/Modul:	Programmieren II für Wirtschaftsinformatiker
Erstellungsdatum:	September 6, 2025

Aufgabe 1: Grundlagen zu Futures, ExecutionContext und Parallel Collections in Scala

Beschreiben Sie die Kernkonzepte der Nebenläufigkeit und Parallelität in Scala mithilfe von Futures, ExecutionContext und Parallel Collections. Bearbeiten Sie die folgenden Unteraufgaben.

a) Definieren Sie die Begriffe Futures, ExecutionContext und Parallel Collections im Kontext von Scala. Erläutern Sie in kurzen Worten deren Rolle in der Neben- bzw. Parallelität.

b) Geben Sie ein kurzes Scala-Beispiel an, wie ein Future mit dem globalen ExecutionContext ausgeführt wird und wie das Ergebnis asynchron verarbeitet werden kann. Verwenden Sie eine onComplete- oder map-Seiteneffektierung, um das Ergebnis auszugeben.

```
import scala.concurrent.{Future, ExecutionContext}
import scala.util.{Success, Failure}
import scala.concurrent.ExecutionContext.Implicits.global

val f: Future[Int] = Future {
  Thread.sleep(100)
  21
}
f.onComplete {
  case Success(v) => println("Ergebnis: " + v)
  case Failure(ex) => println("Fehler: " + ex.getMessage)
}
```

c) Beschreiben Sie den Unterschied zwischen dem globalen ExecutionContext und einem benutzerdefinierten ExecutionContext, z. B. durch Erstellung eines ExecutorService. Geben Sie ein kurzes Beispiel, wie Sie einen custom ExecutionContext erstellen und ihn explizit verwenden.

```
import java.util.concurrent.{Executors, ExecutorService}
import scala.concurrent.{ExecutionContext,
   ExecutionContextExecutorService}

val es: ExecutorService = Executors.newFixedThreadPool(4)
implicit val ec: ExecutionContext = ExecutionContext.fromExecutorService
   (es)
```

d) Erläutern Sie, was Parallel Collections in Scala sind (z. B. ParVector, ParArray). Zeigen Sie ein kleines Beispiel, wie man eine normale Liste in eine Parallel-Collection konvertiert und eine map-Operation parallel ausführt.

```
val list = List.range(1, 10001)
val par = list.par
val squares = par.map(n => n * n).toList
```

Aufgabe 2: Praktische Nutzung von Futures, Fehlerbehandlung und ExecutionContext

Untersuchen Sie die praktische Nutzung von Futures, Fehlerbehandlung und benannten ExecutionContexts in Scala.

a) Entwerfen Sie eine einfache Futures-Kette, die eine Rechenaufgabe ausführt und dabei Fehler simuliert (z. B. Division durch Null). Zeigen Sie, wie Sie mit recover oder recoverWith eine Alternative liefern können.

```
import scala.concurrent.{Future, ExecutionContext}
import scala.util.{Success, Failure}
import java.util.concurrent.Executors

val es = Executors.newFixedThreadPool(2)
implicit val ec: ExecutionContext = ExecutionContext.fromExecutorService
   (es)

def safeDivide(a: Int, b: Int): Future[Int] = Future { a / b }

val f = safeDivide(10, 0).recover {
  case _: ArithmeticException => -1
}

f.onComplete {
  case Success(v) => println("Ergebnis␣(mit␣Recovery):␣" + v)
  case Failure(ex) => println("Fehler:␣" + ex.getMessage)
}
```

b) Implementieren Sie eine kleine Beispiel-Topologie mit einem custom ExecutionContext (aus Aufgabe 1c) und demonstrieren Sie, wie die Ausführung innerhalb eines begrenzten Thread-Pools erfolgt.

```
import scala.concurrent.{Future, ExecutionContext}
import java.util.concurrent.Executors

val es = Executors.newFixedThreadPool(2)
implicit val ec: ExecutionContext = ExecutionContext.fromExecutorService
   (es)

def compute(x: Int): Future[Int] = Future { x * x }

compute(7).foreach(println)
```

c) Welche Vorteile bieten Promises gegenüber Futures in Scala, und in welchen Fällen sind sie sinnvoll? Geben Sie eine knappe Begründung.

```
import scala.concurrent.{Promise, Future}
import scala.util.{Success, Failure}

val p = Promise[Int]()
```

```
val f: Future[Int] = p.future

f.onComplete {
  case Success(v) => println("Ergebnis: " + v)
  case Failure(ex) => println("Fehler: " + ex.getMessage)
}

p.success(42) // liefert das Ergebnis an das Future
```

Aufgabe 3: Parallel Collections, Nebenläufigkeit in der Praxis

Betrachten Sie typische Anwendungsfälle in der Wirtschaftsinformatik (z. B. BI) für Parallelität in Scala. Arbeiten Sie die folgenden Punkte aus.

a) Diskutieren Sie typische Anwendungsfälle, in denen Parallel Collections sinnvoll sind, und nennen Sie potenzielle Risiken (z. B. Nebenwirkungen, Reihenfolgen, Thread-Safety).

b) Geben Sie ein Beispiel, wie man eine große Liste von Zahlen mithilfe einer Parallel-Collection multipliziert und das Resultat wieder in eine Sequenz überführt. Verwenden Sie dabei eine unveränderliche Struktur.

```
val nums = (1 to 100000).toList
val result = nums.par.map(_ * 3).toList
```

c) Ergänzen Sie das Beispiel um eine sichere Aggregation, die das Maximum der Ergebnisse findet, ohne race conditions zu erzeugen. Nutzen Sie dafür geeignete Methoden der Parallel-Collections.

```
val maxVal = nums.par.map(_ * 3).max
println("Maximalwert:␣" + maxVal)
```

Lösungen

Aufgabe 1: Grundlagen zu Futures, ExecutionContext und Parallel Collections in Scala

Lösung zu a) Grundlagen

- Futures: Ein Future[T] repräsentiert eine asynchrone Berechnung, die zu irgendeinem späteren Zeitpunkt einen Wert vom Typ T liefert oder fehlschlägt. Futures ermöglichen die deklarative Modellierung von Nebenläufigkeit, ohne explizites Thread-Management im Anwendungscode.

- ExecutionContext: Der ExecutionContext definiert die Ausführungsumgebung für asynchrone Aufgaben (die Thread-Pools/Ereignisschleifen, auf denen Futures laufen). Er steuert, auf welchen Threads Tasks ausgeführt werden und wie Futures scheduliert werden.

- Parallel Collections: Parallele Sammlungen erhöhen die Parallelität durch außenliegende Threads, die Operationen auf Teilmengen der Daten ausführen (z. B. ParVector, ParArray). Sie bieten einfache Parallelisierungs-Strategien, erfordern jedoch Aufpassung bezüglich Nebenwirkungen und Thread-Safety.

Lösung zu b) Kurzes Beispiel mit globalem ExecutionContext

```
import scala.concurrent.{Future, ExecutionContext}
import scala.util.{Success, Failure}
import scala.concurrent.ExecutionContext.Implicits.global

val f: Future[Int] = Future {
  Thread.sleep(100)
  21
}
f.onComplete {
  case Success(v) => println("Ergebnis:␣" + v)
  case Failure(ex) => println("Fehler:␣" + ex.getMessage)
}
```

Lösung zu c) Unterschied global vs. benutzerdefinierter ExecutionContext; Beispiel

- Unterschied: Der globale ExecutionContext (z. B. ExecutionContext.global) verwendet typischerweise eine gemeinsam genutzte Thread-Pool-Konfiguration. Ein benutzerdefinierter ExecutionContext kapselt oft eine eigene Thread-Pool-Größe und Lebensdauer (z. B. via ExecutorService) und erlaubt so eine isolierte oder priorisierte Ausführung.

- Vorteil eines custom EC: besseres Capacity- und Ressourcen-Management, Vermeidung von contention mit anderen Aufgaben im globalen Pool.

```
import java.util.concurrent.{Executors, ExecutorService}
import scala.concurrent.{ExecutionContext,
   ExecutionContextExecutorService}

val es: ExecutorService = Executors.newFixedThreadPool(4)
implicit val ec: ExecutionContext = ExecutionContext.fromExecutorService
   (es)

// Beispiel-Future, die das ec-Implicit nutzt
def compute(x: Int): Future[Int] = Future { x * x }

compute(7).foreach(println)
```

Lösung zu d) Parallel Collections in Scala

- Parallele Collections ermöglichen Operationen wie map, reduce etc. auf Teilmengen der Daten parallel auszuführen. Typische Typen: ParVector, ParArray, ParList.

- Hinweis: Nebenwirkungen und Thread-Safety müssen beachtet werden; sortierte Ergebnisse oder deterministische Reihenfolge erfordern ggf. spezielle Reduktionsoperationen.

```
val list = List.range(1, 10001)
val par = list.par
val squares = par.map(n => n * n).toList
```

Aufgabe 2: Praktische Nutzung von Futures, Fehlerbehandlung und ExecutionContext

Lösung zu a) Futures-Kette mit Fehler-Simulation und Recovery

```
import scala.concurrent.{Future, ExecutionContext}
import scala.util.{Success, Failure}
import java.util.concurrent.Executors

val es = Executors.newFixedThreadPool(2)
implicit val ec: ExecutionContext = ExecutionContext.fromExecutorService
   (es)

def safeDivide(a: Int, b: Int): Future[Int] = Future { a / b }

val f = safeDivide(10, 0).recover {
  case _: ArithmeticException => -1
}

f.onComplete {
  case Success(v) => println("Ergebnis␣(mit␣Recovery):␣" + v)
  case Failure(ex) => println("Fehler:␣" + ex.getMessage)
}
```

Lösung zu b) Topologie mit benutzerdefiniertem ExecutionContext demonstrieren

```
import scala.concurrent.{Future, ExecutionContext}
import java.util.concurrent.Executors

val es = Executors.newFixedThreadPool(2)
implicit val ec: ExecutionContext = ExecutionContext.fromExecutorService
   (es)

def compute(x: Int): Future[Int] = Future { x * x }

compute(7).foreach(println)
```

Lösung zu c) Vorteile von Promises gegenüber Futures und Sinnhaftigkeit

- Promises ermöglichen explizites (spätes) Complete-Ereignis, während Futures read-only bleiben. Promises sind sinnvoll, wenn man z.B. Callback-basierte APIs integrieren oder komplexe asynchrone Workflows bauen möchte, bei denen die Fertigstellung von außen gesteuert wird.

```
import scala.concurrent.{Promise, Future}
import scala.util.{Success, Failure}
import scala.concurrent.ExecutionContext.Implicits.global

val p = Promise[Int]()
val f: Future[Int] = p.future

f.onComplete {
  case Success(v) => println("Ergebnis:␣" + v)
  case Failure(ex) => println("Fehler:␣" + ex.getMessage)
}
```

```
p.success(42) // liefert das Ergebnis an das Future
```

Aufgabe 3: Parallel Collections, Nebenläufigkeit in der Praxis

Lösung zu a) Typische Anwendungsfälle und Risiken

- Sinnvolle Einsatzgebiete: intensive numerische Berechnungen, große Datenmengen-Bearbeitung, BI-Analysen, ETL-Pipelines, Sampling, Monte-Carlo-Simulationen.

- Risiken: Nebenwirkungen in map/reduce-Operationen, Nicht-Determinismus von Ergebnissen bei unsachgemäßer Nutzung, Thread-Safety bei gemeinsam genutzten Mutablen, Overhead durch Thread-Wechsel, falsche Annahmen über Reihenfolge.

Lösung zu b) Beispiel: große Liste parallel multiplizieren

```
val nums = (1 to 100000).toList
val result = nums.par.map(_ * 3).toList
```

Lösung zu c) Sichere Aggregation mit paralleler Verarbeitung

```
val maxVal = nums.par.map(_ * 3).max
println("Maximalwert:␣" + maxVal)
```