Lernzettel

Interoperabilität zwischen Java und Scala: JVM, Java-APIs, SAM-Interfaces.

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

Lernzettel: Interoperabilität zwischen Java und Scala: JVM, Java-APIs, SAM-Interfaces.

(1) Überblick: JVM und Interoperabilität

Die JVM ist die Laufzeitumgebung, in der sowohl Java- als auch Scala-Programme ausgeführt werden. Scala kompiliert zu Bytecode, der von der JVM ausgeführt wird, und Java-APIs sind standardmäßig nahtlos für Scala verfügbar. Dadurch können Scala-Programme Java-Bibliotheken direkt verwenden und Java-Anwendungen können Scala-Klassen verwenden.

Wichtige Punkte:

- JVM bietet plattformunabhängigen Bytecode, Garbage Collection und JIT-Optimierung.
- Scala- und Java-Code teilen sich denselben Bytecode-Standard (class-Dateien).
- Typ-System-Kompatibilität: Scala-Types werden auf der JVM als Java-Typen abgebildet; Nullbehandlung erfordert Vorsicht und oft Options statt Null.

(2) JVM-Architektur und Typ-System (Grundlagen)

- Bytecode-Dateien (`.class`) enthalten Klassen, Methoden und Felder.
- Primitive Typen (z. B. `int`, `long`) vs. Wrapper-Typen (`Integer`, `Long`).
- Scala fügt oft Optionen (`Option`) statt Null als Nullwert-Vertreter ein.
- Arrays und Sammlungen werden zwischen Java- und Scala-Welten konvertiert.

(3) Java-APIs in Scala verwenden

Scala kann Java-APIs direkt nutzen. Zur bequemen Handhabung von Java-Sammlungen und -Klassen empfiehlt sich die Konvertierung zwischen Java- und Scala-Collections.

Beispiel: Java-List in Scala verwenden und in eine Scala-Collection überführen

```
import java.util.{List, ArrayList}
import scala.jdk.CollectionConverters._
val javaList: List[String] = new ArrayList[String]()
javaList.add("Alpha")
javaList.add("Beta")
val scalaList = javaList.asScala.toList
scalaList.foreach(println)
```

Weitere Beispiele:

- Zugriff auf Java-Optionals und Java-Streams:

  ```
  import java.util.Optional
  val o: Optional[String] = Optional.of("Hallo")
  if (o.isPresent) println(o.get())
  ```

- Verwendung von Java-Streams aus Scala (minimal):

```
val nums = java.util.Arrays.asList(1,2,3,4)
val sum = nums.stream().mapToInt(Integer::intValue).sum()
```

(4) SAM-Interfaces und Lambda-Interoperabilität

SAM = Single Abstract Method. Java-Interfaces mit exakt einer abstrakten Methode können von Lambdas in Scala implementiert werden, was eine naturl lean-API-Bridge ermöglicht.

Beispiel (Java-Seite):

```
@FunctionalInterface
public interface Printer {
  void print(String s);
}
```

Beispiel (Scala-Seite) – lambda-fähige SAM-Implementierung:

```
val p: Printer = (s: String) => println(s)
p.print("Hallo aus Scala!")
```

Alternative (explicitere Implementierung):

```
val p: Printer = new Printer {
  def print(s: String): Unit = println(s)
}
p.print("Hallo nochmals!")
```

Hinweise:

- Scala kann Java-SAM-Typen oft direkt mit Lambdas belegen, wenn der Typ klar ist.
- Explizite Typannotationen helfen bei der Typinferenz, besonders bei komplexen Signaturen.

(5) Praktische Muster der Interoperabilität

- Konvertierung zwischen Java- und Scala-Collections:

  ```
  import scala.jdk.CollectionConverters._
  val javaList: java.util.List[String] = new java.util.ArrayList[String]()
  val scalaSeq: Seq[String] = javaList.asScala.toSeq
  ```

- Umgang mit Nullwerten:

  ```
  val optFromJava: Option[String] = Option(javaList.orNull)
  ```

- Nutzung von Java-APIs mit Scala-Funktionen:

  ```
  val rdd = scala.collection.JavaConverters.asScalaBuffer(javaList)
  rdd.foreach(println)
  ```

- Best-Practice: Bevorzugen Sie Scala-Kollektionen für Logik und verwenden Sie Java-APIs nur dort, wo sie eindeutig sinnvoll sind.

(6) Best Practices zur Interoperabilität

- Klare Trennung: Domänenlogik in Scala, Integrationscode dort, wo Java-APIs benötigt werden.
- Nutze Konvertierungen gezielt und minimal; zu viele Konvertierungen führen zu Performance-Overhead.
- Achte auf Nulls: Scala-Option statt `null` wo möglich.
- SAM-Interfaces effizient nutzen: Lambdas bevorzugen, wo sinnvoll; Typen klar annotieren.
- Fehler-Handling: Java-Ausnahmen in Scala behandeln (Try-Catch oder Optional/Either-Pattern).

(7) Zusammenfassung
- Die JVM ermöglicht eine nahtlose Ausführung von Java- und Scala-Code.
- Java-APIs lassen sich in Scala direkt verwenden; Sammlungen lassen sich zwischen Java und Scala konvertieren.
- SAM-Interfaces erlauben die Nutzung von Lambdas auch beim Austausch mit Java.
- Durch Clevere Konvertierung, Null-Sicherheit und gezieltes Pattern-Design erfolgt eine effiziente Interoperabilität zwischen Java und Scala.