# Übungsaufgabe

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

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

# Aufgabe 1: Interoperabilität zwischen Java und Scala: JVM, Java-APIs, SAM-Interfaces

Der Fokus dieser Aufgabe liegt auf der Interoperabilität zwischen Java und Scala in der JVM-Umgebung. Untersuchen Sie zentrale Konzepte der gemeinsamen Ausführung, der Unterschiede zwischen Java-APIs und Scala-APIs sowie die Rolle von SAM-Interfaces in hybriden Projekten.

a) Beschreiben Sie, wie die JVM als gemeinsame Ausführungsplattform fungiert und welche Rolle Bytecode-Äquivalenz, Klassenladeverhalten sowie Binary- und Typ-Informationen zwischen Java- und Scala-Bibliotheken spielen. Diskutieren Sie, wie die JVM-Spezifikation eine plattformunabhängige Nutzung von Java- und Scala-Komponenten ermöglicht.

b) Diskutieren Sie zwei Mechanismen, über die Java-APIs und Scala-APIs in einem gemeinsamen Build koexistieren oder gemeinsam genutzt werden können. Gehen Sie dabei auf

*Typ-System-Differenzen* (Generics, Constraints) und

*Behandlung von Funktionen bzw. Lambda-Ausdrücken* (SAM-Konversion, Funktionsliterale)

ein. Erläutern Sie, wie SAM-Interfaces als Brücke zwischen Java- und Scala-Funktionen dienen und welche Auswirkungen dies auf API-Design-Entscheidungen hat.

c) Gegeben eine Java-Schnittstelle mit einer einzigen abstrakten Methode (SAM),

```
public interface Processor {
    void process(String input);
}
```

beschreiben Sie, wie eine Implementierung in Scala mithilfe der SAM-Konversion grundsätzlich erfolgen könnte und welche Vor- bzw. Nachteile dabei entstehen. Beziehen Sie sich dabei auf typische Interoperabilitäts-Szenarien in typisierten JVM-Projekten.

d) Welche Auswirkungen haben SAM-Interfaces auf die Binary-Kompatibilität zwischen Java- und Scala-Bibliotheken hinsichtlich der API-Änderungsverwaltung? Skizzieren Sie drei mögliche Design-Entscheidungen, die Interop-Features unterstützen oder behindern.

# Aufgabe 2: Praktische Interoperabilität – Java-APIs, SAM-Interfaces und Scala

In dieser Aufgabe betrachten Sie konkrete Interoperabilitäts-Szenarien und skizzieren jeweils eine Herangehensweise, wie Scala- und Java-Komponenten zusammenarbeiten können. Die Aufgaben richten sich an Normalfall- und Praxis-Szenarien.

a) Erklären Sie, wie eine Java-API, die ein Function- bzw. Consumer-Interface erwartet, in Scala mit Hilfe von SAM-Konversion genutzt werden kann. Beschreiben Sie die Schritte, die nötig sind, damit ein in Scala definiertes Funktionsliteral als Java-Funktions-Objekt übergeben wird, und welche Typ-Konversionen durchzuführen sind.

b) Diskutieren Sie, wie man SAM-Interfaces aus Java in eine saubere Scala-Bibliothek kapseln kann, damit Scala-Entwickler eine natürliche API nutzen können. Gehen Sie auf mögliche Designentscheidungen ein (z. B. Wrapper-Typen, Adapter-Klassen) und welche Auswirkungen dies auf die Binary-Kompatibilität hat.

c) Skizzieren Sie ein kleines Interop-Szenario (ohne ausführlichen Code) zwischen einer Java-API, die ein SAM-Interface erwartet, und einer Scala-Anwendung, die eine Scala-Funktion bereitstellt. Beschreiben Sie, wie die Übergabe eines Funktionswerts aus Scala an Java funktioniert und welche Schritte der Typ-Konversion erforderlich sind.

# Aufgabe 3: Design-Entscheidungen und Best Practices in der JVM-Interop

Hinweis: In dieser Aufgabe reflektieren Sie typische Muster zur Umsetzung von Interoperabilität in echten Projekten.

a) Skizzieren Sie drei gängige Architekturen, um Java-APIs in Scala-Anwendungen zu kapseln (z. B. direkte Nutzung, Wrapper-Typen, Adapter-Klassen) und diskutieren Sie Vor- und Nachteile hinsichtlich Typ-Tiefe, Binary-Kompatibilität und Wartbarkeit.

b) Erläutern Sie, wie Exceptions in der JVM bei der Interoperabilität zwischen Java und Scala behandelt werden sollten. Welche Muster (propagierte Ausnahmen, geprüfte Ausnahmen) sind in gemischten Codebasen sinnvoll, und wie beeinflussen sie API-Design und Fehlerbehandlung?

c) Diskutieren Sie, wie sich SAM-Interfaces auf die API-Design-Entscheidungen auswirken, insbesondere bei der Gestaltung von Funktions-APIs in Scala-Bibliotheken, die mit Java-Client-Code interoperieren sollen.

# Lösungen

# Aufgabe 1: Interoperabilität zwischen Java und Scala: JVM, Java-APIs, SAM-Interfaces

a) **Lösung:**

- Die JVM dient als gemeinsame Ausführungsplattform für Java- und Scala-Code, weil beide Sprachen in JVM-Bytecode kompiliert werden. Folgende Aspekte betreffen die Interoperabilität:
  - Bytecode-Äquivalenz: Java-Bytecode und Scala-Bytecode müssen auf derselben JVM lauffähig sein. Die JVM definiert das Klassenformat (Class-Dateien, Konstanten-Pool, Methoden-Deskriptoren) unabhängig von der Sprache, mit der der Code geschrieben wurde. Scala erzeugt zusätzlich synthetische Member (z. B. für implizite Parameter, Default-Parameter) und Hilfsklassen, aber die effektiv ausgeführte Logik sitzt in JVM-Bytecode, der mit Java-Bibliotheken kompatibel ist.
  - Klassenladeverhalten: Alle Klassen, Java- wie Scala-Bibliotheken, werden durch den gleichen oder kompatiblen ClassLoader geladen. Für reibungslose Interoperabilität müssen Typinformationen, Signaturen und ggf. lagerspezifische Namensmuster konsistent bleiben. Die JVM-Spezifikation ermöglicht so das plattformunabhängige Zusammenwirken von Java- und Scala-Komponenten.
  - Binary- und Typ-Informationen: Generische Typen in Java (Type Erasure) und lesbarer Typ-Informationen in Scala unterscheiden sich zwar; dennoch bleiben die Signaturen der Methoden sichtbar, und Aufrufer- bzw. Implementierungsverträge müssen kompatibel bleiben. Für Funktionalität in der Praxis ist entscheidend, dass öffentliche Schnittstellen mit derselben abstrakten Signatur existieren.
- Die JVM-Spezifikation sorgt dafür, dass plattformunabhängige Ausführung und Interoperabilität zwischen Java- und Scala-Bibliotheken möglich bleiben. Wichtige Punkte sind dabei stabile Signaturen, korrekte Erzeugung und Verwendung von Bytecode sowie konsistente Laufzeitverhalten von generischen Typen (mit der notwendigen Berücksichtigung von Type Erasure).

b) **Lösung:** Zwei Mechanismen, über die Java-APIs und Scala-APIs in einem gemeinsamen Build koexistieren oder gemeinsam genutzt werden können.

- Mechanismus 1: SAM-Konversion direkt im gemeinsamen Build nutzen
  - Beschreibung: Eine Java-SAM-Schnittstelle (z. B. ein funktionales Interface) wird von Scala aus mit einem Funktionsliteralen genutzt. Der Scala-Compiler erzeugt automatisch eine anonyme Implementierung des SAM, sodass scala-Funktionen direkt als Java-Funktionsobjekte übergeben werden können.
  - Typ-System-Differenzen: Java-Generics (invariant) vs. Scala-Generics (variance) müssen beachtet werden. Wenn eine Java-Methode einen Parameter wie Function<T,R> erwartet, kann in Scala eine Funktion T => R übergeben werden; der Compiler erzeugt die notwendige Adapter-Implementierung.
  - Behandlung von Funktionen/Lambda-Ausdrücken: SAM-Konversion (Scala → Java) wird genutzt; Funktionsliterale in Scala fungieren als schlanke Brücke zu Java-Funktionsinterfaces.
- Mechanismus 2: Wrapper/Adapter-Ansatz zur Abstraktion
  - Beschreibung: Eine Scala-spezifische API kapselt die Java-API in idiomatische Scala-Typen (z. B. Scala-Funktionen statt Java-SAM). Es existiert eine Adapter-

Schicht, die bei Bedarf die Scala-Funktionen in Java-SAM-Objekte übersetzt, bevor die Java-API aufgerufen wird.
  - Typ-System-Differenzen: Der Wrapper übersetzt Scala-Typen (z. B. Funktionen, Variance) in die Java-Typen. Dadurch bleiben Scala-Entwickler auf der Scala-API, während Java-Anwendungen die originalen Java-Schnittstellen nutzen können.
  - Behandlung von Funktionen/Lambda-Ausdrücken: Der Adapter nimmt Scala-Funktionen entgegen (z. B. A => B) und erzeugt daraus eine Java-SAM-Implementierung, die an die Java-Bibliothek weitergereicht wird.

c) **Lösung:** SAM-Konversion in Scala – Beispiel und Vor-/Nachteile

```
Public interface Processor { void process(String input); }
```

In Scala könnte eine Implementierung grundsätzlich auf zwei Arten erfolgen:

- Mit SAM-Konversion (empfohlen für prägnante Lösungen):

  ```
  let proc: Processor = (input: String) => println(input)
  ```

  Der Compiler erzeugt automatisch eine anonyme Klasse, die das SAM-Interface implementiert und die Methode `process` entsprechend aufruft.

- Mit expliziter Implementierung (manuell sichtbar):

  ```
  val proc = new Processor {
    def process(input: String): Unit = println(input)
  }
  ```

**Vor- und Nachteile:** - Vorteile der SAM-Konversion - Kürze, Klarheit, idiomatischer Scala-Stil. - Direkte Nutzung von Scala-Funktionen in Java-SAM-Kontexten. - Nachteile der SAM-Konversion - Man erhält eine implizite Brücke, deren Debugging-Informationen ggf. schwieriger nachzuzeichnen sind. - Bei komplexeren Abhängigkeiten kann die Übersetzung zu zusätzlicher Heap-Nutzung (kleine, invisible Klassen) führen. - Anwendungsfälle - Gemeinsamer Build mit Java-API, die eine SAM-Schnittstelle erwartet; Scala-Funktionen eignen sich hier hervorragend. - Falls eine klare Trennung von Scala- und Java-APIs gewünscht ist, können Wrapper-Adapter sinnvoll sein.

d) **Lösung:** Auswirkungen von SAM-Interfaces auf Binary-Kompatibilität und drei Design-Entscheidungen

- Design-Entscheidung 1: Verlässliche Stabilität der SAM-Interfaces
  - Vermeide das Hinzufügen oder Entfernen weiterer abstrakter Methoden in bestehenden SAM-Interfaces, da dies Binary-Kompatibilität bricht.
  - Nutze stattdessen Default- bzw. statische Methoden, um Erweiterungen einzuführen.
- Design-Entscheidung 2: Einführung neuer Interfaces statt Änderung bestehender
  - Wenn neue Funktionalitäten benötigt werden, erstelle neue SAM-Schnittstellen (z. B. ProcessorV2) statt das bestehende Interface zu erweitern.
  - Dadurch bleibt die Kompatibilität zu älteren Clients erhalten.
- Design-Entscheidung 3: Wrapper-Adapter als stabiler Schnittstellenlayer
  - Halte eine klare Trennung zwischen Java-API (unverändert) und einer Scala-API mit idiomatischen Typen. Wrapper-Adapter ermöglichen es, Scala-APIs unabhängig von Java-Änderungen weiterzuentwickeln.

  - Dokumentiere die Adapter-Ebene als stabilen API-Schutzlayer.
- Ergänzend: Nutze @FunctionalInterface in Java, setze auf Default-Methoden, verwende klare Versionsstrategie für API-Rollouts.

# Aufgabe 2: Praktische Interoperabilität – Java-APIs, SAM-Interfaces und Scala

a) **Lösung:**

- Java-API-Setup: Angenommen, eine Java-API erwartet ein Function- oder Consumer-Interface (Java 8+).
- Scala-Seite: Scala-Funktionsliterale können direkt als Implementierungen eines Java-SAM-Interfaces dienen, dank SAM-Konversion.
- Typkonversionen:
  - Java `java.util.function.Function[T,R]` entspricht in Scala dem Typ `T => R`.
  - Java `java.util.function.Consumer[T]` entspricht in Scala dem Typ `T => Unit`.
- Beispielablauf (High-Level):
  - Die Java-API definiert eine Methode `callWithFunction(Function<String, String> f)`.
  - In Scala ruft man diese Methode mit einem Funktionsliteral auf, z. B.:

```
val api: JavaApi = new JavaApi(...)
val result = api.callWithFunction((s: String) => s.trim + "!")
```

- Warum funktioniert das?
  - Der Scala-Compiler generiert zur Laufzeit eine SAM-Implementierung, die das Funktionsliteral in das Java-Interface übersetzt.
  - Typkonversionen laufen gemäß der JVM-Sprach-Interoperabilität ab; der generische Typ ist zur Compile-Zeit bekannt.

b) **Lösung:** SAM-Interfaces aus Java in eine saubere Scala-Bibliothek kapseln

- Design-Idee: Eine Scala-API, die idiomatische Scala-Typen (z. B. `String => Unit`) verwendet, aber intern Java-SAM-Interfaces nutzt, um Java-Client-Code zu unterstützen.
- Mögliche Design-Entscheidungen:
  - Wrapper-Typen: Definiere in Scala eine Wrapper-Klasse/ -Trait, die die Java-SAM-Funktionalität kapselt und eine scala-typisierte API bietet.
  - Adapter-Klassen: Implementiere Adapter, die zwischen der Scala-API und der Java-SAM-Schicht vermitteln.
  - Trennung von Modulen: Halte eine separate „Interop"-Schicht, die die Java-SAM-Interfaces kapselt, während die Haupt-Scala-Bibliothek eine idiomatische API präsentiert.
- Auswirkungen auf Binary-Kompatibilität:
  - Die Java-Interfaces bleiben unverändert; die Wrapper/Adapter-Schicht ist pure Scala und kann unabhängig von Java-Versionen weiterentwickelt werden.
  - Änderungen in der Java-API würden primär die Adapter-Schicht betreffen; die Scala-API kann stabil bleiben, solange der Adapter die gleichen Signaturen bietet.

c) **Lösung:** Kleines Interop-Szenario (ohne ausführlichen Code)

- Ausgangslage: Eine Java-API erwartet ein SAM-Interface `Processor` mit `process(String)`.

- Scala-Anwendung: Definiert eine Scala-Funktion `val f: String => Unit = s => println(s)`.
- Übergabe eines Funktionswerts: Die Scala-Funktion wird per SAM-Konversion in eine Java-SAM-Implementierung umgewandelt und an die Java-API übergeben.
- Typ-Konversion: Der Scala-Typ `String` wird zu `java.lang.String`, die Rückgabe `Unit` entspricht void.
- Laufzeitfluss: Java-Code ruft `process(input)`; die implementierte Methode ruft die Scala-Funktion auf.

# Aufgabe 3: Design-Entscheidungen und Best Practices in der JVM-Interop

a) **Lösung:** Drei gängige Architekturen, um Java-APIs in Scala-Anwendungen zu kapseln

- Architektur 1: Direkte Nutzung (no wrappers)
  - Vorteile: Minimaler Overhead, maximale Transparenz, direkte Nutzung der Java-API.
  - Nachteile: Weniger idiomatische Scala-API, potenzielle Diskrepanz bei Typ-System-Differenzen (Variance, Boxings).
- Architektur 2: Wrapper-Typen (Scala-API umschließt Java-APIs)
  - Vorteile: Saubere Scala-API, bessere Typabstraktionen, leichterer Wechsel der Java-Implementierung.
  - Nachteile: Zusätzlicher Wartungsaufwand für Wrapper, potenzielle Inkonsistenzen bei API-Änderungen.
- Architektur 3: Adapter-Klassen (klare Trennung Schwarz/Weiß)
  - Vorteile: Trennung von Idiomatik (Scala) und Persistenz (Java), klare Binary-Kompatibilitätslinien.
  - Nachteile: Komplexere Struktur, zusätzlicher Boilerplate-Code.

**Vergleich (Vorteile/Nachteile)**: - Typ-Tiefe: Wrapper und Adapter erlauben tiefere Typ-Sicherheit in der Scala-API; direkte Nutzung minimaler Abstraktion. - Binary-Kompatibilität: Wrapper/Adapter erleichtern Migration, da Java-APIs unverändert bleiben können. - Wartbarkeit: Wrapper/Adapter erhöhen die Wartbarkeit in Scala, direkte Nutzung reduziert Codevolumen, kann aber zu uneinheitlichem API-Stil führen.

b) **Lösung:** Exceptions in der JVM bei Interoperabilität

- Java unterstützt geprüfte (checked) Exceptions; Scala kennt keine Checked Exceptions.
- Empfohlene Muster:
  - Propagierte Ausnahmen: Verwende nur geprüfte Ausnahmen dort, wo die Java-API es zwingend erfordert; in Scala müssen diese entweder abgefangen oder als unchecked weitergereicht werden.
  - Verwendung von Wrappern: Fasse Java-Ausnahmen in eigene Scala-Ausnahmen (Unchecked) zusammen, um die Sprachgrenzen zu überbrücken.
  - Fehlerbehandlung konsolidieren: Nutze Try, Either oder andere Scala-Konstrukte in der Scala-Bibliothek, um Fehler sauber zu propagieren, auch wenn sie Java-Exzeptionen sind.
- API-Design und Fehlerbehandlung: Bevorzugt API-Designs, die keine sensiblen Abhängigkeiten von geprüften Ausnahmen in der Java-API benötigen; dokumentiere klare Grenzflächen für Fehlerszenarien; wenn nötig, biete adapterbasierte Fehlerbehandlung an.

c) **Lösung:** SAM-Interfaces und ihre Auswirkungen auf API-Design-Entscheidungen

- Auswirkungen auf API-Design in Scala-Bibliotheken, die mit Java-Client-Code interoperieren sollen:

  - Fördere funktionale APIs: SAM-Interfaces ermöglichen funktionsbasierte Nutzung, was idiomatisch zu Scala-Funktionen passt.
  - Verwende klare Abstraktionen: Bevorzuge wrappers/Adapters, um Scala-Lieferung von Funktionen in Java-SAM-Interfaces sauber zu kapseln.
  - Vermeide häufige API-Änderungen an bestehenden SAM-Interfaces: Nutze neue Interfaces für Erweiterungen, um Binary-Interoperabilität zu sichern.
  - Dokumentiere die Grenzen der Interoperabilität: Welche Funktionen als SAM implementiert werden können, welche Konversionen nötig sind, etc.
- Praktische Designempfehlung: Wenn Java-Client-Code SAM-Interfaces erwartet, bietet sich eine Scala-API an, die Scala-Funktionen als first-class citizens unterstützt, während eine Adapter-Schicht die Brücke zur Java-API bildet.