Übungsaufgabe

Traits, abstrakte Klassen und Mixins in Scala.

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

Aufgabe 1: Traits, abstrakte Klassen und Mixins in Scala

In dieser Aufgabe befassen Sie sich mit den Konzepten Traits, abstrakte Klassen und Mixins in Scala. Bearbeiten Sie die folgenden Unteraufgaben. Setzen Sie kurze Textantworten mit ein, ergänzt durch einfache Code-Skelette, die Sie ausformulieren bzw. ergänzen.

a) Definieren Sie in eigenen Worten, was ein Trait, eine abstrakte Klasse und ein Mixin in Scala ist, und nennen Sie jeweils einen typischen Einsatz.

b) Betrachten Sie die folgende Skizze von Bausteinen. Ergänzen Sie das Scala-Code-Skelett so, dass es die Idee von Mixins widerspiegelt, ohne eine fertige Lösung vorzugeben.

```
trait Logger {
  def log(message: String): Unit
}
abstract class DataSource {
  def fetch(): Seq[String]
}
class ReportService extends DataSource with Logger {
  def fetch(): Seq[String] = Seq.empty // TODO: implementieren
  def log(message: String): Unit = ??? // TODO: implementieren
}
```

c) Diskutieren Sie kurz, welche Auswirkungen das Mischen mehrerer Traits auf die Reihenfolge der Ausführung von Methoden hat (linearization), und welche Vorteile der Einsatz von Mixins in der Praxis mit sich bringt. Geben Sie keine fertige Lösung an.

Aufgabe 2: Praktische Anwendung – Traits, abstrakte Klassen und Mixins in einer Beispielarchitektur

In dieser Aufgabe modellieren Sie eine kleine Beispielarchitektur, in der Traits und abstrakte Klassen sinnvoll kombiniert werden. Arbeiten Sie wieder nur mit Text und Code-Skeletten; keine endgültigen Implementierungen vorgeben.

a) Entwerfen Sie eine einfache Basisklasse `Account` mit dem Feld `balance: Double`. Definieren Sie zwei Traits `Feeable` und `Auditable`, welche ergänzende Funktionalitäten bereitstellen sollen. Formulieren Sie die Schnittstellen so, dass sie sich sinnvoll zu einem konkreten Konto kombinieren lassen.

b) Erstellen Sie ein Code-Skelett, das eine konkrete Klasse `CustomerAccount` zeigt, die `Account` mit `Feeable` und `Auditable` mischt. Implizieren Sie dabei nur Platzhalter (z. B. `???` oder leere Implementierungen) und kennzeichnen Sie TODO-Einträge.

```
abstract class Account(val balance: Double)

trait Feeable {
  def fee(): Double = 0.0 // TODO: implementieren
  def balanceAfterFee: Double = balance - fee()
}

trait Auditable {
  def logChange(change: String): Unit = () // TODO: implementieren
}

class CustomerAccount(bal: Double) extends Account(bal) with
   Feeable with Auditable {
  // TODO: weitere Implementierung
}
```

c) Diskutieren Sie in kurzen Stichpunkten: - Welche Vor- und Nachteile ergeben sich aus der Kombination von abstrakter Klasse und mehreren Traits? - Wie helfen Ihnen Traits beim Separation of Concerns und bei der Wiederverwendung von Verhalten?

Aufgabe 3: Stackable Traits und Abstrakte Klassen – Beispielaufgabe

Betrachten Sie das Muster der stackable traits (mehrfaches Überschreiben mit `super`). Arbeiten Sie ohne fertige Lösung; liefern Sie nur Aufgabenstellungen und minimale Code-Skelettes.

a) Skizzieren Sie drei Traits `Base`, `FeatureA`, `FeatureB`, die jeweils eine Methode `describe(): String` erweitern, so dass am Ende eine aussagekräftige Beschreibung entsteht. Verwenden Sie `abstract override` bzw. `super.describe()` entsprechend, legen Sie aber der Vollständigkeit halber keine konkrete Reihenfolge fest.

```
trait Base {
  def describe(): String = "Base"
}

trait FeatureA extends Base {
  abstract override def describe(): String = super.describe() + "
   + A"
}

trait FeatureB extends Base {
  abstract override def describe(): String = super.describe() + "
   + B"
}
```

b) Definieren Sie eine konkrete Klasse `SystemComponent` mit einer einfachen Implementierung von `describe()`. Erzeugen Sie eine Instanz von `SystemComponent` mit `FeatureA` und `FeatureB` und skizzieren Sie, wie die Endbeschreibung zustande kommt.

c) Reflektieren Sie kurz: Welche Herausforderungen können beim Einsatz stackabler Traits auftreten (z. B. Kompilierzeiten, Semantik, Friendly-API)? Welche Best Practices würden Sie empfehlen, um Missverständnisse zu vermeiden?

Lösungen

Lösung zu Aufgabe 1: Traits, abstrakte Klassen und Mixins in Scala

a) Lösung: Kurzdefinitionen und typische Einsätze

- Trait: - Eine Scala-Trait ist ein Typ, der sowohl unverankerte Interfaces (Methodenangaben) als auch konkrete Implementierungen liefern kann. Traits eignen sich hervorragend zum wiederholten Mischen von Verhalten über mehrere Klassen hinweg. - Typischer Einsatz: gemeinsames Verhalten (z. B. Logging, Validierung) über unterschiedliche Klassen hinweg, ohne eine gemeinsame Oberklasse zu erzwingen.

- Abstrakte Klasse: - Eine abstrakte Klasse kann nicht instanziiert werden und dient als Teil-Implementierung mit möglichen Feldern/Konstruktorparametern. Sie kann sowohl abstrakte als auch konkrete Members enthalten. - Typischer Einsatz: Bereitstellung einer gemeinsamen Basisklasse mit teils implementiertem Verhalten, das von konkreten Unterklassen genutzt wird.

- Mixin: - Mischformen (Mixins) werden über das Schlüsselwort `with` realisiert. Sie ermöglichen das Zusammensetzen von Verhalten, wobei die lineare Reihenfolge der gemischten Bausteine die Ausführung beeinflusst. - Typischer Einsatz: schrittweise Erweiterung von Verhalten (z. B. Logging, Auditing, Validierung) durch mehrere Traits in einer kontrollierten Reihenfolge.

b) Lösung: Fertige Implementierung des Code-Skeletts mit sinnvoller Ausführung

```
trait Logger {
  def log(message: String): Unit = println(s"[ReportService]
   $message")
}
abstract class DataSource {
  def fetch(): Seq[String]
}
class ReportService extends DataSource with Logger {
  def fetch(): Seq[String] = Seq("Report-Teil1", "Report-Teil2", "
   Report-Teil3")
  def log(message: String): Unit = println(s"[ReportService]
   $message")
}
```

c) Lösung: kurzer Diskurs zur Linearization und Nutzen von Mixins

- Die Reihenfolge der Mixins bestimmt die sogenannte Linearization der Klasse. Für Methoden, die über `super` weitergereicht werden, ergibt sich ein eindeutiger Ausführungspfad entsprechend der Linearization. - Typischer Vorteil: sehr feine, wiederverwendbare Bausteine für Verhalten, die unabhängig von der konkreten Klasse getestet und eingesetzt werden können. Gleichzeitig lässt sich Verhalten durch das Hinzufügen weiterer Traits flexibel erweitern, ohne die Basisklasse zu verändern. - Hinweis: Zu beachten ist, dass bei mehrfacher Vererbung (mehrere Traits) die Reihenfolge der Traits Auswirkungen darauf hat, welcher Trait zuerst seine Implementierung ausführt, welche wiederum `super` aufruft und so die Kette weiterführt.

Lösung zu Aufgabe 2: Praktische Anwendung – Traits, abstrakte Klassen und Mixins in einer Beispielarchitektur

a) Lösung: Basisklasse `Account` und zwei Traits `Feeable`, `Auditable`

```
abstract class Account(val balance: Double)

trait Feeable {
  def fee(): Double = 0.0
  def balanceAfterFee: Double = balance - fee()
}

trait Auditable {
  def logChange(change: String): Unit = println(s"Audit: $change")
}
```

b) Lösung: Code-Skelett mit konkreter Klasse `CustomerAccount`

```
class CustomerAccount(bal: Double) extends Account(bal) with
   Feeable with Auditable {
  // einfache Beispielimplementierung eines Geb hrenverhaltens
  override def fee(): Double = balance * 0.02  // z. B. 2% Geb hr

  // Beispielhafte Nutzung des Auditings
  def applyCharge(): Unit = {
    val newBal = balanceAfterFee
    logChange(s"Applied fee ${fee()} on balance ${balance}, new
   balance ${newBal}")
  }
}
```

c) Lösung: kurze Stichpunkte

- Vorteile: - Flexible Komposition von Verhalten durch Kombination abstrakter Klassen und mehrerer Traits. - Separation of Concerns: Gebührenlogik, Auditierung und Kontenlogik sind sauber getrennt. - Wiederverwendbarkeit: Traits können in mehreren Kontotypen genutzt werden.

- Nachteile / Risiken: - Konflikte oder unerwartete Überschreibungen durch mehrere Traits können auftreten (speziell bei `super`-Aufrufen). - Komplexität der Konstruktion bei vielen Mixins: die Linarization muss nachvollzogen werden. - Änderungen in einem Trait können ungewollt Auswirkungen auf alle Klassen haben, die ihn verwenden.

Lösung zu Aufgabe 3: Stackable Traits und Abstrakte Klassen – Beispielaufgabe

a) Lösung: Drei Traits, die `describe()` erweitern

```
trait Base {
  def describe(): String = "Base"
}
```

```
trait FeatureA extends Base {
  abstract override def describe(): String = super.describe() + "
   + A"
}

trait FeatureB extends Base {
  abstract override def describe(): String = super.describe() + "
   + B"
}
```

b) Lösung: Konkrete Klasse `SystemComponent` und Instanziierung mit Mixins

```
class SystemComponent extends Base

// Instanz mit Stackable Traits
val component = new SystemComponent with FeatureA with FeatureB
val result = component.describe()
// Erwartung: "Base + A + B" (abh ngig von der Linearization,
   hier ergibt sich gem    der Reihenfolge SystemComponent ->
   FeatureB -> FeatureA -> Base)
```

c) Lösung: Reflexion zu Herausforderungen und Best Practices

- Herausforderungen: - Komplexe Kompilierzeiten und schwer nachvollziehbare Fehler bei umfangreicher Mix-in-Ketten. - Semantische Verwirrung durch unklare Reihenfolge der Ausführung von Methoden (insbesondere bei `abstract override` und `super`). - API-Verwaltung: Missverständnisse über die Reihenfolge der Ausführung und die benötigten Super-Implementierungen.

- Best Practices: - Begrenze die Anzahl der Mischungen pro Klasse und dokumentiere die erwartete Linearization (z. B. durch klare Sequenz der Traits). - Nutze `abstract override` nur dort, wo wirklich ein Chain-of-Responsibility-Pfad sinnvoll ist; halte die Chain eindeutig. - Schreibe Tests, die verschiedene Mischungsreihen testen, um Regressionen der Linearization zu vermeiden. - Halte Traits klein, fokussiert und gut benannt (Single Responsibility Principle auch für Traits).