Lernzettel

Traits, abstrakte Klassen und Mixins in Scala.

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

Lernzettel: Traits, abstrakte Klassen und Mixins in Scala

(1) Traits – Grundidee.
Traits in Scala sind wie Interfaces in Java, aber mit der Möglichkeit, implementierten Code mitzuliefern. Man kann mehrere Traits mischen (mixin), um Verhalten zu kombinieren. Typischer Aufbau:

```
trait TimestampLogger {
  def log(msg: String): Unit =
    println(s"[${java.time.Instant.now()}] $msg")
}
trait Serializer[T] {
  def toJson(value: T): String
}
```

Beispiel für das Mischen von Traits:

```
class User(val name: String) {
  def greet(): Unit = println(s"Hi, $name")
}
trait Greeting {
  def sayHello(): Unit = println("Hello")
}
class FriendlyUser(name: String) extends User(name) with Greeting
```

(2) Abstrakte Klassen.
Abstrakte Klassen bieten einen gemeinsamen Bauplan mit teilweise konkreter Implementierung und können Konstruktorparameter enthalten. Eine Klasse kann nur von einer abstrakten Klasse erben, aber mehrere Traits hinzufügen.

```
abstract class Vehicle {
  val id: String
  def drive(): Unit
  def description(): String = s"Vehicle $id"
}
class Car(val id: String) extends Vehicle {
  def drive(): Unit = println(s"Driving car $id")
}
```

Wichtig: Traits dienen eher der modularen Zusammensetzung von Verhalten, abstrakte Klassen dienen dem gemeinsamen Basiskonstruktor und einer gemeinsamen Implementierung.

(3) Mixins – konkrete Nutzung.
Mixins ermöglichen das Hinzufügen von Verhalten zu bestehenden Klassen durch das Annotieren mit mehreren Traits.

```
trait Logging {
  def log(msg: String): Unit = println(s"[LOG] $msg")
}
trait Timestamp {
  def now(): String = java.time.Instant.now().toString
}
class Service {
  def run(): Unit = println("Service running")
}
class DataService extends Service with Logging {
  def start(): Unit = {
```

```
    log("Starting DataService")
    run()
  }
}
```

(4) Unterschiede zur Java-Programmierung.
- Traits ermöglichen Mehrfachvererbung von Verhalten, ohne die Probleme der traditionellen Mehrfachvererbung in Java.
- Abstracte Klassen in Scala können Konstruktorparameter haben; Traits können diese Parameter nicht direkt setzen.
- Bei der Reihenfolge der Mixins bestimmt die lineare Vererbung, welches Standardverhalten gilt (super-Aufruf chain).

(5) Java-Interop und praktische Hinweise.
- Scala-Traits werden zum Teil als Interfaces mit Implementierungs-Methoden auf Bytecode-Ebene umgesetzt.
- Java-Klassen können Scala-Typen und -Traits nutzen, sofern sie kompatibel modelliert sind.
- Verwende kleine, fokussierte Traits mit klarer Zuständigkeit; vermeide stateful Traits, wenn möglich.

(6) Kurze Anwendungsbeispiele in Wirtschaftsinformatik (BI).
- Modulare Logikbausteine für ETL-Pipelines: Traits für Logging, Fehlerbehandlung und Validierung können in Data-Transformationen gemischt werden.
- Beispiel-Design: Ein Trait TimestampLogger eignet sich, um Auditing-Informationen in BI-Prozessen zu hinterlegen, ohne die Kernlogik zu vermischen.

```
trait Auditable {
  def audit(action: String): Unit =
    println(s"[AUDIT] ${java.time.Instant.now()} - $action")
}
trait Validation {
  def validate(record: Map[String,String]): Boolean
}
class ETLJob extends Auditable with Validation {
  def process(record: Map[String,String]): Unit = {
    if (validate(record)) {
      audit("Processed record")
      // Transformationslogik hier
    } else {
      audit("Validation failed")
    }
  }
  def validate(record: Map[String,String]): Boolean = true
}
```

(7) Fazit.
Traits, abstrakte Klassen und Mixins ermöglichen in Scala eine flexible, modulare und gut testbare Gestaltung von Code. Durch richtige Grenzziehung und saubere Schnittstellen lassen sich Konzepte der objektorientierten und funktionalen Programmierung sinnvoll kombinieren.