innFactory

# Anwendung der funktionalen Programmierung mit Scala

TH Rosenheim - SoSe 2025

# Intro & Organisatorisches
## Wer sind wir?

**Nicolas Schlecker** Bereichsleiter | Full-Stack Engineer
Bachelor Informatik TH Rosenheim
E-Mail: n.schlecker@innfactory.de

**Tobias Jonas** CEO | Full-Stack Engineer
Master Informatik TH Rosenheim
E-Mail: t.jonas@innfactory.de

# Intro & Organisatorisches
## Inhalt der Vorlesung

**1. Einführung und Überblick**

- Funktionsbegriff & Prinzipien
- Lambda-Kalkül, Kategorientheorie (historische Grundlagen)
- Funktionale Sprachen & Anwendungsszenarien

**2. Grundlagen funktionaler Programmierung mit Scala**

- Sprachelemente: def, trait, val/var, Klassen, Pattern Matching, Implicits/Givens
- Rekursion & rekursive Datenstrukturen
- Funktionen höherer Ordnung, Lambdas, Currying, Laziness
- Monaden: Option, Either, Try, map/flatMap, for-Comprehensions
- Nebenläufigkeit & Futures

**3. Fortgeschrittene Scala-Konzepte**

- Typsystem, Generics, Type Classes
- Einführung Cats: Semigroups, Monoids, Functors
- Effektvolle und parallele Programmierung (Cats Effect, ZIO, Pekko)
- Scala 2 vs. Scala 3
- **4. Anwendungen im Data Engineering & parallelen Systemen**
- Apache Spark: Funktionale Verarbeitung großer Datenmengen
- Apache Pekko (akka): Reaktive & parallele Systeme
- Real-World Beispielarchitekturen im IoT-Bereich

# Intro & Organisatorisches
## Lernziele

- **Konzepte** und **typische Merkmale** des funktionalen Programmierens kennen, verstehen und anwenden können:
  - Modellierung mit **algebraischen Datentypen**
  - **Rekursion**
  - Starke **Typisierung**
  - **Funktionen höherer Ordnung** (map, filter, fold)
- Datenstrukturen und Algorithmen in Scala **umsetzen** und auf einfachere praktische Probleme **anwenden** können

# Vortrag & Mündliche Prüfung
## Allgemein

- **Vorträge**
  - Teams aus je 2 Personen
  - Themenwahl wie FWPM-Wahl
  - 25-30 Minuten
  - Teil der Gesamtnote

- **Mündliche Prüfung**
  - Am 10.07.2025
  - 15 Minuten
  - Sowohl Vorlesungen als auch Übungen sind Inhalt der Prüfung

# Vortrag & Mündliche Prüfung
## Themen

1. Lisp & Scheme
2. Haskell
3. Funktionale Programmierung mit Python
4. Clojure: Funktionale Programmierung auf der JVM
5. Funktionale Konzepte in JavaScript (fp-ts / effect-ts)
6. Erlang und Elixir
7. Funktionale Konzepte in Rust
8. F-Sharp
9. Scala Framework: Spark Streaming
10. Scala Framework: ZIO
11. Scala Framework: Spotify SCIO
12. Scala Typelevel Ökosystem

Link im
Learning Campus

innFactory

# Funktionale Programmierung

# Funktionale Programmierung
## Was versteht man unter funktionaler Programmierung?

- Programmierparadigma (Imperativ *C*, Objektorientiert *Java*, Funktional *Scala*l)
- Funktionen im Zentrum
  - Übergabe von Funktionen als Parameter
  - Funktionen als Rückgabewert anderer Funktionen
  - Kombination mit anderen Funktionen
- Funktionale Programme beschreiben die Lösung eines Problems durch verschachtelte Funktionsaufrufe und Ausdrücke

# Funktionale Programmierung
## Paradigmenvergleich: Imperative Programmierung

- Beschreibt **wie** ein Problem gelöst wird, Schritt für Schritt
- Abfolge von Zustandsänderungen (Zuweisungen, Schleifen, etc.)
- Programmierer gibt genaue Anweisungen, etwa durch Schleifenvariablen oder modulübergreifende globale Zustände

# Funktionale Programmierung
## Paradigmenvergleich: Objektorientierte Programmierung (OOP)

- Organisiert Code um **Objekte**, welche Zustand (**Attribute**) und Verhalten (**Methoden**) kapseln
- Programme bestehen aus kooperierenden Objekten, die einander Nachrichten senden
- Zustände sind in Objekten verborgen (**Kapselung**) und Interaktion erfolgt über definierte Schnittstellen (**Methoden**)
- **OOP** ist letztlich auch imperativ, jedoch mit zusätzlicher Struktur
- Fokus liegt auf Datenstrukturen (**Klassen**) und deren Beziehungen

# Funktionale Programmierung
## Paradigmenvergleich: Funktionale Programmierung (FP)

- Beschreibt **was** berechnet werden soll
- Vermeidet veränderbare Zustände und Seiteneffekte, strebt **immutability** an
- Programme bestehen aus Ausdrücken (Funktionen und deren Verkettung)
- Ausführung ist eher **deklarativ**: Man formuliert Eigenschaften des Ergebnisses, und das **Wie** ergibt sich aus der Funktionskomposition

⇒ Funktionale Programmierung ist ein deklaratives Paradigma, das Funktionen in den Mittelpunkt stellt und Programmabläufe über Funktionsausdrücke definiert, anstatt über veränderliche Zustände und Kontrollstrukturen

innFactory

# Code Beispiele

## Imperativ vs Deklarativ

## Statements vs Expressions

# Funktionale Programmierung
## Warum funktionale Programmierung lernen?

### Einfachheit und Klarheit

- Keine versteckten Zustände und Seiteneffekte
- Funktionen verhalten sich wie mathematische Funktionen (gleicher Input => gleicher Output)
- Leichter verständlicher und kürzerer Code
- Einfacheres Debugging und bessere Nachvollziehbarkeit

# Funktionale Programmierung
## Warum funktionale Programmierung lernen?

### Modularität und Wiederverwendbarkeit

- Kleine, universelle Funktionsbausteine (vgl. Lego)
- Keine versteckten Nebeneffekte erleichtern Kombination
- Unabhängigere Entwicklung, Testbarkeit und vielseitige Wiederverwendung
- Höhere Modularität und bessere Code-Struktur

# Funktionale Programmierung
## Warum funktionale Programmierung lernen?

### Nebenläufigkeit und Parallelität

- Einfachere parallele Ausführung durch **immutability** (Unveränderbarkeit)
- Reduzierung von Race Conditions und Deadlocks
- Weniger Synchronisation nötig, dadurch bessere Performance

# Funktionale Programmierung
## Nachteile

- Höhere Einstiegshürde: Erfordert Umstellung auf „funktionales Denken“
- Potenziell höherer Speicherverbrauch durch Immutability
- Kleinere Community: Weniger Beispiele und Ressourcen (2,6% der Entwickler nutzen Scala[1])
- Komplexe Abstraktionen (Monaden, Typklassen) können anspruchsvoll sein



[1]StackOverflow Developer Survey 2024

# Funktionale Programmierung
## Zentrale Begriffe

- Funktionen als „**First-Class Citizens**“ (Bürger erster Klasse)
  - Gleicher Status wie etwa Zahlen oder Strings
  - Zuweisung von Funktionen an Variablen oder Übergabe als Parameter ermöglicht **Higher-Order Functions** (höherwertige Funktionen)
- **Pure Functions** (reine Funktionen)
  - Keine Seiteneffekte
  - Keine Veränderung von Daten außerhalb des lokalen Kontextes (kein I/O)
  - Gleiche Eingabe => Gleiches Ergebnis

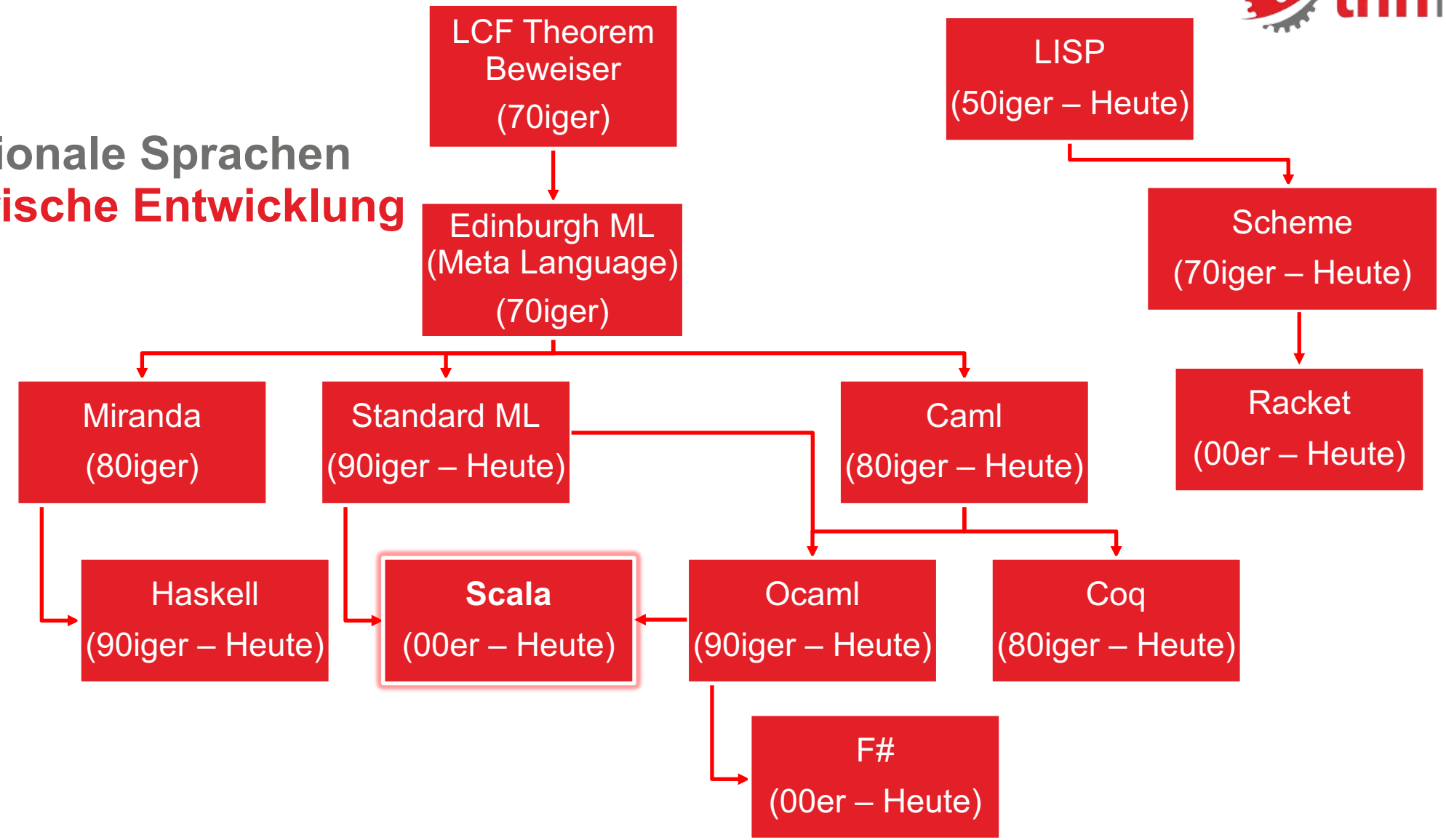
# Funktionale Programmierung
## Eigenschaften funktionaler Sprachen

- **Seiteneffektfreiheit (Purity)**
  - Seiteneffekt ist jede beobachtbare Auswirkung einer Funktion, die über ihre Berechnung des Rückgabewerts hinausgeht
  - I/O trotzdem möglich aber kontrolliert, z.B. über spezielle Typen **Monaden**
- **Unveränderliche Datenstrukturen**
  - In den meisten funktionalen Sprachen sind Datenstrukturen wie List, Map, Set standardmäßig **immutable**
  - Daten werden nicht verändert, sondern bei Bedarf neue Daten aus alten abgeleitet
  - Verhindert Fehler wie die unbedachte gemeinsame Nutzung von veränderbaren Objekten (Erleichtert Parallelisierung, da keine Synchronisation nötig)

innFactory

# Funktionale Sprachen

# Funktionale Sprachen
## Historische Entwicklung



# Funktionale Sprachen
## Wer benutzt Sie?


Integration von **smithy4s (Disney)**
in das Play Framework

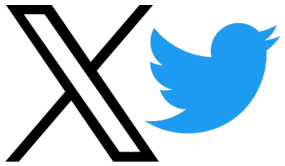
NETFLIX

Recommendation Engine

Spotify

SCIO Framework is a Scala API for Apache Beam and Google Cloud Dataflow.
**Spotify Wrapped**

Scalatra-Mikroframework
Echtzeit-Datenverarbeitung
Signal-API



Finagle für high concurrency RPC-Systeme

Scala

airbnb

Entwickelt “Aerosolve” Framework für ML

NuBank

Bank in Brasilien entwickelt Microservices mit Clojure

elixir

Pinterest

Echtzeit Datenverarbeitung

Disney streaming services

**Disney+**, Hulu, ESPN+ und Star+

Haskell

GitHub / Semantic

Parsing, analyzing, and comparing source code across many languages

# Funktionale Sprachen
## Ursprung

**Alonzo Church**
1934 Lambda-Kalkül
=> Programmiersprachen

**Alan Turing**
1936 Turing Maschine
=> Computer

# Funktionale Sprachen
## Lambda Kalkül

- Entwickelt um logische Probleme (wie das Entscheidungsproblem) zu untersuchen
- Alles ist eine Funktion, Anwendung von Funktion auf ein Argument oder eine abstrakte Variable
- Besteht aus
  - **Variablen**: Platzhalter für Werte/Eingaben („x“)
  - **Abstraktionen**: Funktionsdefinition („λx. x + 1” beschreibt eine Funktion, die 1 zu „x” addiert)
  - **Applikationen**: Funktionsaufrufe („(λx. x + 1) 2” ergibt 3, 2 ist hier der Parameter x)
- Ermöglicht die Beschreibung sämtlicher berechenbarer Funktionen
- „Programmiersprache auf Papier“

# Funktionale Sprachen
## Lambda Kalkül Beispiele

- **Identitätsfunktion**
  „λx. x“ – „(λx. x) 5“ => 5
- **Konstante Funktion**
  „λx. 42“ – „(λx. 42) 7“ => 42
- **Funktion als Argumente**
  „(λf. f 10) (λx. x * 2)“ => „(λx. x * 2) 10“ => 20
- **Zusammensetzung**
  „compose = λf. λg. λx. f (g x)“
  z.B. „(λx. (λy. y*y) ((λz. z+1) x)) 3“ => Innerer Teil g(x) „(λz. z+1) 3 = 4“ und dann „(λy. y*y) 4 = 16“

- Das Rechnen im Lambda-Kalkül besteht aus dem schrittweisen Ersetzen (Auswerten) von Lambda-Ausdrücken => Spiegelt Auswertung funktionaler Programme wieder

# Funktionale Sprachen
## Verbindung zur Kategorientheorie

- Abstraktes mathematisches Gerüst, beschreibt Strukturen und deren Beziehungen zueinander
- **Kategorien** bestehen aus
  - **Objekten:** z.B. Datentypen (Int, String, List[Int])
  - **Morphismen:** Funktionen zwischen diesen Typen
- Zentrale Operationen sind
  - **Komposition** (Hintereinanderausführung von Morphismen)
  - **Identität** (Jedes Objekt besitzt eine Abbildung auf sich selbst)
- **FP** basiert auf dem **Komponieren von Funktionen**
  - Kategorientheorie ist theoretisches Fundament für Konzepte wie **Funktoren, Monaden, Monoide**
  - „FP ist angewandte Kategorientheorie“
  - In der Praxis ist nur die Anwendung der Muster notwendig

# Funktionale Programmierung
## Anwendungsgebiete

- **Datenverarbeitung und Analyse (Apache Spark, Apache Flink, Kafka Streams)**
  Big-Data-Verarbeitung, Data Engineering und Streaming-Anwendungen
- **Webentwicklung (Play Framework, Akka (Pekko) HTTP, ZIO HTTP, http4s)**
  Backend-Systeme, APIs und Webservices mit starkem Fokus auf Wartbarkeit und Skalierbarkeit
- **Verteilte Systeme und Microservices (Akka (Pekko), Cloudstate, ZIO)**
  Entwicklung reaktiver, fehlertoleranter und skalierbarer Architekturen
- **Komplexe und nebenläufige Anwendungen (Cats Effect, ZIO, Monix)**
  Anwendungen mit hohem Parallelisierungsgrad und sicherer Nebenläufigkeit durch immutability und deklarative Programmierung
- **Wissenschaftliche und mathematische Berechnungen (Breeze, Spire, Saddle, ScalaLab)**
  Funktionale Ansätze zur präzisen Modellierung komplexer Berechnungen

innFactory

# Einführung in Scala

# Scala
## Überblick

- **Scala** ist eine hybride Sprache, unterstützt **FP** und **OOP** (Multiparadigma)
- Entwickelt von Martin Odersky, erstmals erschienen 2004
- Läuft auf der Java Virtual Machine (JVM), sowie in JavaScript (Scala.js) und Native (Scala Native)
- Interop mit Java, dadurch ist gesamtes Java-Bibliothek Ökosystem integrierbar
- Statisches Typsystem mit Typ-Inferenz durch den Compiler (Scala 3 erzwingt explizitere Typisierung)

# Scala
## Variablen und Funktionen

- Variablen Deklaration
  - **val** für unveränderliche Variablen („val x = 6“; „x = 5“ => Compiler Error)
  - **var** für veränderliche Variablen („var x = 6; „x = 5“ => Reassignment)
- Funktionsdefinition

  **def** name(parameter1: Typ1, parameter2: Typ2, …): Rückgabetyp = { Rumpf }
  - „def square(x: Double): Double = x * x
    => Kein **return** wie in anderen Sprachen
  - „val fun = (x: Int) => x * 2“
    => Funktionen als Werte („First-Class Citizen“)

# Scala
## Objekte und Klassen

- **Klassen** definieren einen neuen Objekttyp. Man verwendet „class“ gefolgt vom Namen und Konstruktorparametern
  class Person(val name: String, val age: Int)
- **Objekte (Singletons)** Keyword „object“ gefolgt vom Namen
  - Scala kennt kein separates Konzept von „statischen“ Methoden oder Feldern wie Java
  - Im JVM-Kontext umgesetzt als einzige Instanz einer anonymen Klasse
  - Einstiegspunkt für Programme („object Main“ mit „def main“ Methode) oder auch Sammlung von Hilfsfunktionen
  - Ein **Companion Objects** ist ein Objekt mit demselben Namen wie eine Klasse, erlaubt Zugriff auf private Elemente. Nützlich für Factories
    class Circle(val radius: Double)
    object Circle { def apply(r: Double): Double = new Circle (r) }

innFactory

# Code Beispiele

innFactory

# Übung

# Übung
## Setup & Erster Code

- Software
  - IntelliJ IDEA - https://www.jetbrains.com/idea/ (GitHub Student Developer Pack - https://education.github.com/pack)
  - Docker - https://docs.docker.com/desktop/
- Übung
  - Projekt erstellen
  - **ScalaTest** und **Iron (2.6.0)** integrieren (https://www.scalatest.org/ und https://github.com/Iltotore/iron)
  - Validierungsfunktionen erstellen für
    - Emails
    - Passwort (>=6 Zeichen, >=1 Zahl, >=1 Buchstabe, Groß und Klein)
  - Tests für runtime validation, optional compile time validation