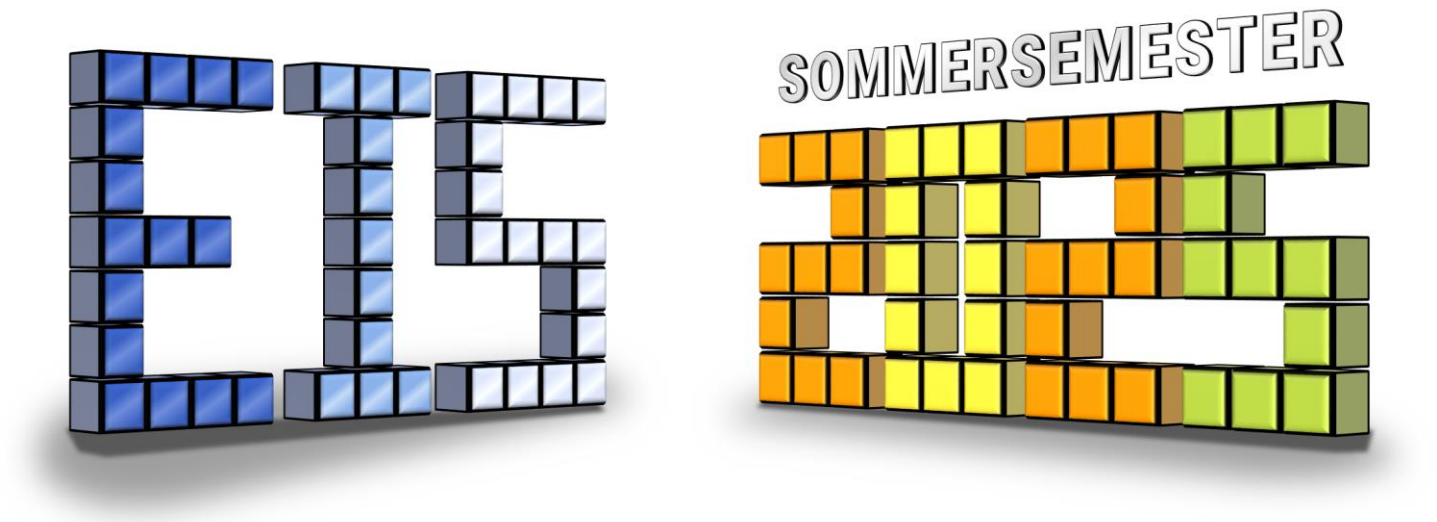
EINFÜHRUNG IN DIE SOFTWAREENTWICKLUNG

Sommersemester 2025



Foliensatz #2d

Programmiersprachen: Scala

Michael Wand
Institut für Informatik
Michael.Wand@uni-mainz.de

Übersicht

Inhalt heute

- Programmiersprachen
  - Python + MyPy
  - C/C++
  - Java/Scala

Programmiersprachen

Java + Scala

Programmiersprachen

(d) Scala

Diskussion: Programmiersprache

Vorlage zur Diskussion

- Daten + Variablen
- Ausdrücke und Berechnungen
- Befehle
- Abstraktionen
- Systemumgebung

Scala

Programmiersprache SCALA

- Ergebnis eines Forschungsprojektes an der EPFL
- Multi-Paradigmen-Sprache
  - OOP + FP (Funktionale Konzepte)
  - Stärkere Betonung von FP
  - Sehr viele „Features“ (daher beliebt)
- Baut auf der JVM auf
  - Nutzung von JAVA-Bibliotheken (sehr umfangreich)
  - Läuft auf vielen Plattformen
  - „Ersatz“ für JAVA
    - Wenn einem JAVA „zu langweilig“ ist :-)
    - Insbesondere FP macht mehr Spaß in SCALA

Scala Versionen

Ähnlich wie bei Python…

- …gibt es mehrere Scala Versionen
- Scala 2 sehr verbreitet (z.B. Ubuntu default)
- Scala 3 neuste Version
  - Mehr syntaktische Optionen
    (wie Python oder wie JAVA)
  - Renoviert in Bezug auf Konsistenz
    (was C++ nie gemacht hat)
- EIS nutzt Scala 3
  - Unterschiede wenig relevant
  - Wir schauen fast nur den „JAVA“-Teil von SCALA an

Diskussion: Programmiersprache

Vorlage zur Diskussion

- Daten + Variablen
- Ausdrücke und Berechnungen
- Befehle
- Abstraktionen
- Systemumgebung

Scala Beispielcode

```
// Variablen müssen deklariert werden
var a: Int = 23;          // var = normale, veränderliche Variable
val b: Double = 42.0;     // val = unveränderlich (ähnlich „const“ in C++)
val c = 1337;             // automatische Typinferenz: c ist Int (fest ab hier!)

//  Automatische Konvertierung nur „aufwärts“
var d: Double = a + b;    //  a wird in double umgewandelt
var e: Int = b;           //  Compiler-Fehler! Informationsverlust nicht erlaubt
var e: Int = b.asInstanceOf[Int];   //  Explizite Typumwandlung nötig

//  Berechnungen sonst wie in JAVA („boolean“ wird nun groß geschrieben)
val f: Boolean = (a != b) && (b-c > a);

//  Ausgabe auf der Konsole mit weniger Bürokratie als in JAVA
print("Hello "); println("World!"); //  (println mit Zeilenumbruch)

//  Strings sind (im fast genau) JAVA-Strings
val s: String = "Hello World!";
println(s.substring(0, 5).length); //  "Hello" hat Länge 5
```

(Un-) Veränderliche Variable

In funktionalen Sprachen

- Vermeidung, Variablen zu ändern
  - Erfordert Rekursion statt Schleifen
  - Später mehr dazu
- Daher unveränderliche Variablen besonders wichtig
- In Scala
  - **`val`** deklariert eine unveränderliche Variable
  - **`var`** deklariert eine veränderliche Variable
- Freie Auswahl
  - Sprachen wie Haskell sind hier restriktiver
  - Multi-paradigmen Ansatz

Scala Datentypen

Unterscheidung: Wert- und Referenztypen

- Alle Typen haben Oberklasse „`Any`“
  - Werttypen in Unterklasse „`AnyVal`“
    - Eigene Unterklassen **nur eingeschränkt** möglich
    - Dar nur eine Instanz eines primitiven Typs enthalten
  - Referenztypen in Unterklasse „`AnyRef`“
    - Entspricht JAVA „`Object`“ auf JVM-Ebene
- Wir sprechen noch über Vererbung im Detail

Eingeschränkte Auswahl

- Wert- und Referenzsemantik im Prinzip wie in JAVA

Scala Datentypen

Primitive Werttypen (fast wie in JAVA)

- `Boolean` (true oder false)
- `Byte` (8-bit signed)
- `Short` (16-bit signed)
- `Int` (32-bit signed)
- `Long` (64-bit signed)
- `Float`, `Double` (32/64 IEEE-Float, wie in JAVA)
- `BigInt`, `BitDecimal` (sehr große Zahlen)
- `Char` (16-bit unsigned UCS1)
- `Unit` (entspricht **`void`** in JAVA)

Scala Datentypen

„null“ Referenztypen

- **`null`** für leere Referenzen (JAVA-Kompatibilität)
  - Entspricht „**None**“ in Python
  - Empfehlung: Nicht direkt benutzen

None vs. Some (für Werte und Referenzen)

- Bessere Lösung: Explizite „Nullable Types“:

```
var s: Option[String] = None;
s = Some("Hello World!");
```

- Variablen, die None sein können, werden extra gekennzeichnet (vermeidet Laufzeitfehler)
  - Sehr ähnlich zu „**`Optional`**“ in MyPy

Zusammengesetzte Datentypen

Klassen (ähnlich Python und C++ structs & Klassen)

```
class Bruch extends AnyRef {
    var zaehler: Int = 0; // veränderliches Feld
    var nenner: Int = 1;  // veränderliches Feld
};
```

implizit, weglassen

```
var drei_viertel = new Bruch(); // veränderliche
drei_viertel.zaehler = 3;       // Variable
drei_viertel.nenner = 4;
```

implizit, kann man weglassen

*GC: **delete** gibt es nicht (genau wie in JAVA)*

Alternative

Klassen (immutable)

```
class Bruch(val zaehler: Int = 0, // unveränderlich
            val nenner: Int = 1) {// via Konstruktor
};
var drei_viertel = Bruch(3,4);
println(drei_viertel.zaehler); // 3
println(drei_viertel.nenner);  // 4
```

(Un)veränderlich ist einfach

- **val** / **var** reicht
- Normale Klassen via Referenzen (Zeiger implizit)

Standard Datentypen

Standardbibliothek

- Standardklassen `String`, `Array` (wie in Java)
  - Referenztypen
  - Es gibt auch „`List`“ für unveränderliche Arrays
- Boxing von primitiven Typen (AnyVal)
  - Wird implizit / automatisch erledigt
- Arrays: Gleiche Klasse wie in Java, andere Notation
  - `var myArray = new Array[Int](42);`
    - Leeres Array mit 42 Plätzen
  - `var myArray = Array("Ada", "Berta", "Frida");`
    - Initialisiertes Array
  - Beide Beispiele benutzen bereits Typinferenz

Diskussion: Programmiersprache

Vorlage zur Diskussion

- Daten + Variablen
- Ausdrücke und Berechnungen
- Befehle
- Abstraktionen
- Systemumgebung

The Basics

Arithmetische Ausdrücke, Operationen

- Fast alles aus Java möglich
  - `x++`, `x--`, etc. wurden wieder entfernt (wie in Python)

Syntax

- Syntax wie in JAVA immer möglich
  - Runde und geschweifte Klammern, Semikolons
- (Insbesondere in Scala 3) mehr Flexibilität
  - Auch Python Syntax möglich
    (kein **`{…}`**, kein **`;`**, statt dessen Einrückung)
- Ich verwende JAVA-Syntax auf den Folien
  - Um Vergleich einfach zu halten – keine Empfehlung!

Zusätzliche Features

Alles sind Ausdrücke

- **`if … then … else`** als Ausdruck

```
val y: Int = 42;
val x: String = if (y == 42) {
    "The meaning of life" // Ausdrücke statt Befehle
} else if (y == 23) {
    "The number from this book"
} else if (y == 1337) {
    "LEET"
} else {
    "Boring number"
}
println(x);
```

Zusätzliche Features

„Pattern Matching“-Ausdrücke

- Ausdrücke mit komplexen Fallunterscheidungen

```
val y: Int = 42;
val x: String = y match {
    case 42 => "The meaning of life"
    case 23 => "The number from this book"
    case 1337 => "LEET"
    case _ => "Boring number" // alle anderen Fälle
}
println(x);
```

- Match kann noch mehr (z.B. Typen matchen)
  - Mehr dazu noch später…

Diskussion: Programmiersprache

Vorlage zur Diskussion

- Daten + Variablen
- Ausdrücke und Berechnungen
- Befehle
- Abstraktionen
- Systemumgebung

Alles Ausdrücke

Es gibt nur Ausdrücke

- Seiteneffekte erlaubt
  - Ein bisschen wie in C++, nur extremer
- Beispiel

```
var a: Int = 42;
a match {
  case 42   => {println("The meaning of life");}
  case 23   => {println("Something about 5");}
  case 1337 => {println("LEET");}
  case _    => {println("Boring");}
}
```

- Das gleiche geht natürlich auch für **if**

Alles Ausdrücke

Was ist mit Schleifen?

- Beispiel „while“

```
var i: Int = 10; // Countdown!
while (i > 0) {
   println(i);
   i -= 1;
}
```

- Schleifen seltener genutzt als in C++/JAVA
  - Eher funktionale Datenflussgraphen (map, apply, etc.)
  - Mehr dazu später

Alles Ausdrücke

Was ist mit Schleifen?

- Beispiel „**for**“

```
for (i <- 10 to 1 by -1) do {
  println(i);
}
```

- Beispiel „**for**“ im Sinne von „for each“ (wie in Python)

```
val nums = List(10, 9, 8, 7, 6, 5, 4, 3, 2, 1);
for (i <- nums) do {
  println(i);
}
```

- List-Comprehesion (Ausdruck)

```
val nums_x_2 = for (i <- nums) yield {2*i;}
```

Reminder: Flexibler Syntax

Mann kann das auch kürzer schreiben

- ```
  for i <- 10 to 1 by -1 do
    println(i)
  ```
- ```
  val nums = List(10, 9, 8, 7, 6, 5, 4, 3, 2, 1)
  for i <- nums do
    println(i)
  ```
- ```
  val nums_x_2 = for i <- nums yield 2*i
  ```
- In dieser VL nicht vertieft (aber offiziell empfohlen)
  - Viele Klammern & Semikola optional
  - Einrückungssyntax wie in Python
  ```

Diskussion: Programmiersprache

Vorlage zur Diskussion

- Daten + Variablen
- Ausdrücke und Berechnungen
- Befehle
- Abstraktionen
- Systemumgebung

Scala Features

Scala Features

- Alles, was JAVA kann
- OOP zusätzlich z.B.
  - traits (JAVA 8) und mixins
- Functional Programming, z.B.
  - Currying, partial evaluation, lazy evaluation, closures (JAVA 8)
  - Lambda expressions (anonyme Funktionen, ab JAVA8)
  - Pattern matching & algebraic data types (JAVA17 preview)
- Coole Scala spezifische Features (jenseits dieser VL)
  - Macros auf AST-Ebene
  - „implicit“ Parameter

auch hier:
mehr dazu später!

Disclaimer

Vorlesung beschränkt sich auf Kernkonzepte

- Wir schauen uns die wichtigsten Techniken an
- Fast alles gleichermaßen umsetzbar in
  - Python
  - Scala
  - JAVA
  - C++
  - C/Pascal (mit erhöhtem Aufwand und ohne Typprüfung)
- Einige funktionale Techniken in Scala am besten
- Dynamische Konzepte schwer in C/C++ (z.B. Reflection)

Diskussion: Programmiersprache

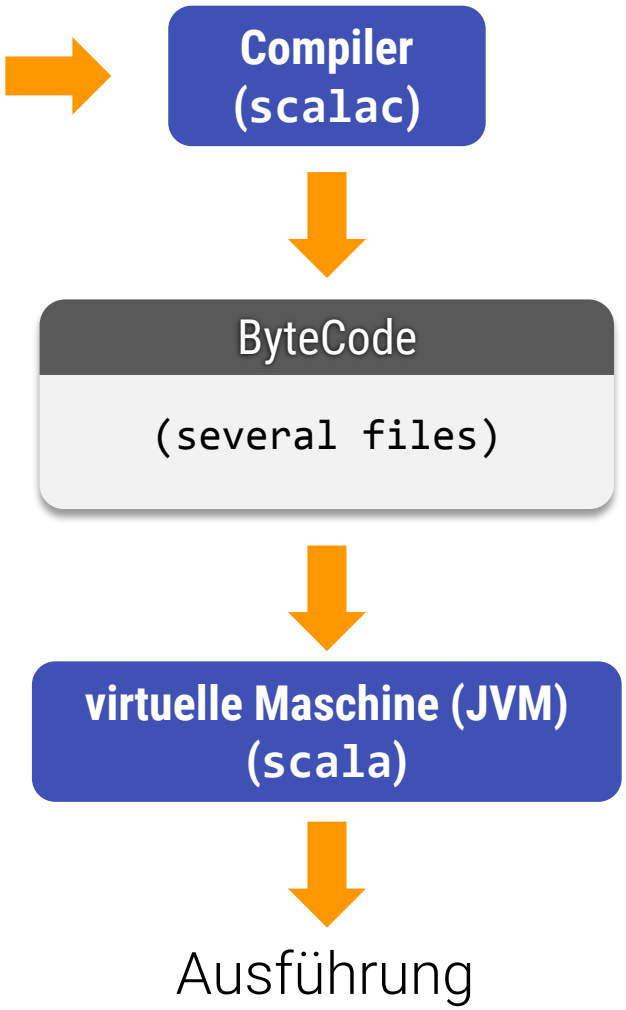
Vorlage zur Diskussion

- Daten + Variablen
- Ausdrücke und Berechnungen
- Befehle
- Abstraktionen
- Systemumgebung

Eine Klasse pro Datei

Datei: Test.scala

```
@main def hello() = {
    println("Hello World!");
}
```



Module wie in JAVA

Laufzeitumgebung von JAVA geerbt (JVM)

- ```
  import myMod.MyClass;
  import myMod._; // Underline „_“ statt Stern „*“ in Java
  ```
  - Modul / Bezeicher wird bereitgestellt
- Bibliotheken erzeugen via package:

  ```
  package myPack;
  class MyClass …
  ```
  - Wie in JAVA (Unterverzeichnisse als Konvention)
- Dynamisches Laden zur Laufzeit wie in JAVA (JVM)
  ```

Disclaimer (encore en fois)

Wichtig!

- Unsere Vorlesung ist konzeptionell orientiert!
- Dies ist kein klassischer „Programmierkurs“
  - Wir nutzen die Programmiersprachen nicht „voll aus“
  - Wir nutzen nicht immer die elegantesten / empfohlenen Konstrukte
  - Selbststudium + Übung für „schönen“ Code
- Ziel: Grundkonzepte verstehen
  - Sprachunabhängige Ideen
  - Guter Stil im Groben, aber nicht in syntaktischen Details
    - Daran scheitert kein Projekt :-)