Scala

Kai Brünnler

Innovation Process Technology AG

Executive Summary

Scala = Java + Funktionale Programmierung

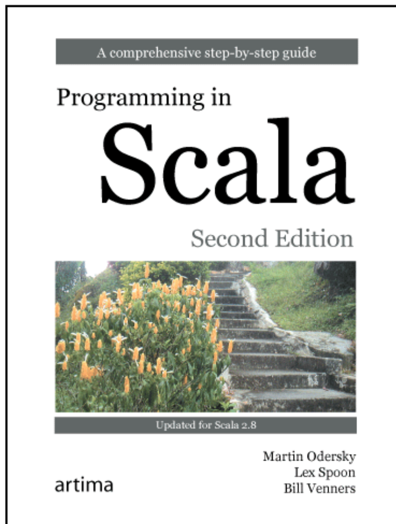
Executive Summary

Scala = Java + Funktionale Programmierung

Extended Executive Summary

- Martin Odersky, EPFL, 2004
- keine Erweiterung von Java
- kompiliert auf JVM, interoperabel mit Java
- statisch getypt – aber Typinferenz
- rein objektorientiert (wie Smalltalk)
- funktional: immutability, functions are values
- Ziel: Skalierbarkeit als Sprache
- real world software: Twitter backend, Etherpad

Buch



Inhalt

1. Objektmodell von Scala
   - Hello world
   - Eine Klasse
   - Nutzen des Objektmodells
2. Funktionales Programmieren
   - Mutability considered harmful
   - Functions are Values
   - Pattern Matching
3. A Domain Specific Language

Inhalt

1 Objektmodell von Scala
Hello world
Eine Klasse
Nutzen des Objektmodells

2 Funktionales Programmieren
Mutability considered harmful
Functions are Values
Pattern Matching

3 A Domain Specific Language

Section 1

Objektmodell von Scala

Objektmodell von Scala

- Jeder Wert ist ein Objekt, jedes Objekt Instanz einer Klasse.

Objektmodell von Scala

- Jeder Wert ist ein Objekt, jedes Objekt Instanz einer Klasse.
  - primitive Typen wie `int`, `long`, `float`, ...
  - Funktionen
  - Arrays und Enumerations

Objektmodell von Scala

- Jeder Wert ist ein Objekt, jedes Objekt Instanz einer Klasse.
  - primitive Typen wie `int`, `long`, `float`, ...
  - Funktionen
  - Arrays und Enumerations
- Jede Operation ist ein Methodenaufruf.

Objektmodell von Scala

- Jeder Wert ist ein Objekt, jedes Objekt Instanz einer Klasse.
  - primitive Typen wie `int`, `long`, `float`, ...
  - Funktionen
  - Arrays und Enumerations
- Jede Operation ist ein Methodenaufruf.

  `a op b`   ist syntactic sugar für   `a.op(b)`

Objektmodell von Scala

- Jeder Wert ist ein Objekt, jedes Objekt Instanz einer Klasse.
  - primitive Typen wie `int`, `long`, `float`, ...
  - Funktionen
  - Arrays und Enumerations
- Jede Operation ist ein Methodenaufruf.

  `a op b` ist syntactic sugar für `a.op(b)`

  `a + b` ist syntactic sugar für `a.+(b)`

Subsection 1

Hello world

Hello World in Scala

```scala
object HelloWorld
{
  def main(args: Array[String]): Unit
  {
    println("Hello World")
  }
}
```

Hello World in Scala

```scala
object HelloWorld
{
  def main(args: Array[String])
  {
    println("Hello World")
  }
}
```

Subsection 2

Eine Klasse

Class Complex in Java

```
public class Complex
{
  private double real, imaginary;

  public Complex(double real, double imaginary)
  {
      this.real=real;
      this.imaginary=imaginary;
  }
}
```

Class Complex in Scala

```
class Complex(real: Double, imaginary: Double)
```

Adding Getter Functions in Java

```
public class Complex
{
  private double real, imaginary;

  public Complex(double real, double imaginary)
  {
      this.real=real;
      this.imaginary=imaginary;
  }
}
```

Adding Getter Functions in Java

```
public class Complex
{
  private double real, imaginary;

  public Complex(double real, double imaginary)
  {
      this.real=real;
      this.imaginary=imaginary;
  }

  public double re() {return real;}
  public double im() {return imaginary;}
}
```

Adding Getter Functions in Scala

```
class Complex(real: Double, imaginary: Double)
{
  def re = real
  def im = imaginary
}
```

Overriding toString in Java

```
public class Complex
{
  private double real, imaginary;

  public Complex(double real, double imaginary)
  {
      this.real=real;
      this.imaginary=imaginary;
  }

  public double re() {return real;}
  public double im() {return imaginary;}
}
```

Overriding toString in Java

```java
public class Complex
{
  private double real, imaginary;

  public Complex(double real, double imaginary)
  {
      this.real=real;
      this.imaginary=imaginary;
  }

  public double re() {return real;}
  public double im() {return imaginary;}

  public String toString() {
     return re() + "+" + im() + "i";
  }
}
```

Overriding toString in Scala

```
class Complex(real: Double, imaginary: Double)
{
  def re = real
  def im = imaginary
  override def toString =
    "" + re + "+" + im + "i"
}
```

Overriding toString in Scala

```
class Complex(real: Double, imaginary: Double)
{
  def re = real
  def im = imaginary
  override def toString =
    "" + re + "+" + im + "i" // why ""?
}
```

Overriding toString in Scala

```
class Complex(real: Double, imaginary: Double)
{
  def re = real
  def im = imaginary
  override def toString =
  (  (  (  "".+(re) ).+("+") ).+(im) ).+("i")
}
```

Scala Interpreter, `var` und `val`

```
scala>
```

Scala Interpreter, `var` und `val`

```
scala> class Complex(real: Double, ...
```

Scala Interpreter, `var` und `val`

```
scala> class Complex(real: Double, ...
defined class Complex

scala>
```

Scala Interpreter, var und val

```
scala> class Complex(real: Double, ...
defined class Complex

scala> var c = new Complex(1.0,2.0)
```

Scala Interpreter, var und val

```
scala> class Complex(real: Double, ...
defined class Complex

scala> var c = new Complex(1.0,2.0)
c: Complex = 1.0+2.0i

scala>
```

Scala Interpreter, `var` und `val`

```
scala> class Complex(real: Double, ...
defined class Complex

scala> var c = new Complex(1.0,2.0)
c: Complex = 1.0+2.0i

scala> c = new Complex(3.0,4.0)
```

Scala Interpreter, var und val

```
scala> class Complex(real: Double, ...
defined class Complex

scala> var c = new Complex(1.0,2.0)
c: Complex = 1.0+2.0i

scala> c = new Complex(3.0,4.0)
c: Complex = 3.0+4.0i

scala>
```

Scala Interpreter, `var` und `val`

```
scala> class Complex(real: Double, ...
defined class Complex

scala> var c = new Complex(1.0,2.0)
c: Complex = 1.0+2.0i

scala> c = new Complex(3.0,4.0)
c: Complex = 3.0+4.0i

scala> val c = new Complex(1.0,2.0)
```

Scala Interpreter, var und val

```
scala> class Complex(real: Double, ...
defined class Complex

scala> var c = new Complex(1.0,2.0)
c: Complex = 1.0+2.0i

scala> c = new Complex(3.0,4.0)
c: Complex = 3.0+4.0i

scala> val c = new Complex(1.0,2.0)
c: Complex = 1.0+2.0i

scala>
```

Scala Interpreter, var und val

```
scala> class Complex(real: Double, ...
defined class Complex

scala> var c = new Complex(1.0,2.0)
c: Complex = 1.0+2.0i

scala> c = new Complex(3.0,4.0)
c: Complex = 3.0+4.0i

scala> val c = new Complex(1.0,2.0)
c: Complex = 1.0+2.0i

scala> c = new Complex(3.0,4.0)
```

Scala Interpreter, var und val

```
scala> class Complex(real: Double, ...
defined class Complex

scala> var c = new Complex(1.0,2.0)
c: Complex = 1.0+2.0i

scala> c = new Complex(3.0,4.0)
c: Complex = 3.0+4.0i

scala> val c = new Complex(1.0,2.0)
c: Complex = 1.0+2.0i

scala> c = new Complex(3.0,4.0)
error: reassignment to val
       c = new Complex(3.0,4.0)
         ^
scala>
```

Subsection 3

Nutzen des Objektmodells

Die factorial Funktion

Java

```
int factorial(int n) {
  if (n==0) return 1;
  else return n*factorial(n-1);
}
```

Die `factorial` Funktion

Java

```
int factorial(int n) {
  if (n==0) return 1;
  else return n*factorial(n-1);
}
```

Scala

```
def factorial(n: Int): Int =
   if (n==0) 1 else n*factorial(n-1)
```

Die `factorial` Funktion auf BigInts

Java

```
BigInteger factorial(BigInteger n) {
  if ( n == BigInteger.ZERO ) return BigInteger.ONE;
  else return
      n.multiply(factorial(n.subtract(BigInteger.ONE)));
}
```

Die `factorial` Funktion auf BigInts

Java

```
BigInteger factorial(BigInteger n) {
  if ( n == BigInteger.ZERO ) return BigInteger.ONE;
  else return
      n.multiply(factorial(n.subtract(BigInteger.ONE)));
}
```

Scala

```
def factorial(n: BigInt): BigInt =
   if (n==0) 1 else n*factorial(n-1)
```

Nutzen des Objektmodells

- Datentyp ohne Sprachunterstützung: unbequem
  (z.B. BigInteger in Java)

Nutzen des Objektmodells

- Datentyp ohne Sprachunterstützung: unbequem
  (z.B. BigInteger in Java)
- Datentyp mit Sprachunterstützung: bequem
  (z.B. large integers in Python), aber skaliert nicht

Nutzen des Objektmodells

- Datentyp ohne Sprachunterstützung: unbequem (z.B. BigInteger in Java)
- Datentyp mit Sprachunterstützung: bequem (z.B. large integers in Python), aber skaliert nicht
- Scala: bequem und skaliert

Features des Typsystems die wir nicht behandeln

- generics – durch type erasure wie in Java, aber besser
- traits – "interfaces with code"
- abstract attributes and types – enums, structural typing

Inhalt

1 Objektmodell von Scala
- Hello world
- Eine Klasse
- Nutzen des Objektmodells

2 Funktionales Programmieren
- Mutability considered harmful
- Functions are Values
- Pattern Matching

3 A Domain Specific Language

Section 2

Funktionales Programmieren

Subsection 1

Mutability considered harmful

Was ist “Mutability”?

Mutability

Mutability = Zustandsänderbarkeit = non-final Variable
Mutation = Zustandsänderung = Wertänderung einer Variable

Was ist “Mutability”?

Mutability

Mutability = Zustandsänderbarkeit = non-final Variable
Mutation = Zustandsänderung = Wertänderung einer Variable

Beispiele

- `i = i+1;`
- `vector.addElement(someElement);`
- `calendar.setTime(someTime);`

Was ist “Mutability”?

Mutability

Mutability = Zustandsänderbarkeit = non-final Variable
Mutation = Zustandsänderung = Wertänderung einer Variable

Beispiele

- `i = i+1;`
- `vector.addElement(someElement);`
- `calendar.setTime(someTime);`

Gegenbeispiele

- `int i = j+1;`
- `String s2 = s1.replace('a','b');`

Was ist "harmful"?

A Case against the GO TO Statement.

by Edsger W.Dijkstra

Technological University

Eindhoven, The Netherlands

Since a number of years I am familiar with the observation that the quality of programmers is a decreasing function of the density of go to statements in the programs they produce. Later I discovered why the use of the go to statement has such disastrous effects and did I become convinced that the go to statement should be abolished from all "higher level" programming languages (i.e. everything except -perhaps- plain machine code).

Slogan :)

Scala verhält sich zu mutability wie C zu `goto`.

Mutability vs. Immutability

Java

Mutable:	StringBuffer
Immutable:	String

Mutability vs. Immutability

Java

Mutable:	StringBuffer
Immutable:	String

Immutability Vorteile

- leichter zu verstehen
- Sharing ist einfacher
- Parallelisierung ist einfacher

Mutability vs. Immutability

Java

Mutable:	StringBuffer
Immutable:	String

Immutability Vorteile

- leichter zu verstehen
- Sharing ist einfacher
- Parallelisierung ist einfacher

Mutability Vorteil

- unter Umständen viel effizienter

Subsection 2

Functions are Values

Functions are Values

```
scala>
```

Functions are Values

```
scala> (x:Int) => x+1
```

Functions are Values

```
scala> (x:Int) => x+1
res0: (Int) => Int = <function1>

scala>
```

Functions are Values

```
scala> (x:Int) => x+1
res0: (Int) => Int = <function1>

scala> var plusone = (x:Int) => x+1
```

Functions are Values

```
scala> (x:Int) => x+1
res0: (Int) => Int = <function1>

scala> var plusone = (x:Int) => x+1
plusone: (Int) => Int = <function1>

scala>
```

Functions are Values

```
scala> (x:Int) => x+1
res0: (Int) => Int = <function1>

scala> var plusone = (x:Int) => x+1
plusone: (Int) => Int = <function1>

scala> var square = (x:Int) => x*x
```

Functions are Values

```
scala> (x:Int) => x+1
res0: (Int) => Int = <function1>

scala> var plusone = (x:Int) => x+1
plusone: (Int) => Int = <function1>

scala> var square = (x:Int) => x*x
square: (Int) => Int = <function1>

scala>
```

Functions are Values

```
scala> (x:Int) => x+1
res0: (Int) => Int = <function1>

scala> var plusone = (x:Int) => x+1
plusone: (Int) => Int = <function1>

scala> var square = (x:Int) => x*x
square: (Int) => Int = <function1>

scala> square(3)
```

Functions are Values

```
scala> (x:Int) => x+1
res0: (Int) => Int = <function1>

scala> var plusone = (x:Int) => x+1
plusone: (Int) => Int = <function1>

scala> var square = (x:Int) => x*x
square: (Int) => Int = <function1>

scala> square(3)
res1: Int = 9

scala>
```

A Higher-Order Function

Sum

$$\sum_{k=a}^{b} f(k)$$

A Higher-Order Function

Sum

$$\sum_{k=a}^{b} f(k)$$

In Scala

```
def sum(f: Int => Int, a: Int, b: Int): Int =
    if (a>b) 0 else f(a) + sum(f,a+1,b)
```

A Higher-Order Function

```
scala>
```

A Higher-Order Function

```
scala> def sum...
```

A Higher-Order Function

```
scala> def sum...
sum: (f: (Int) => Int, a: Int, b: Int) => Int

scala>
```

A Higher-Order Function

```
scala> def sum...
sum: (f: (Int) => Int, a: Int, b: Int) => Int

scala> sum(plusone,1,5)
```

A Higher-Order Function

```
scala> def sum...
sum: (f: (Int) => Int, a: Int, b: Int) => Int

scala> sum(plusone,1,5)
res1: Int = 20        //  (1+1)+(2+1)+(3+1)+(4+1)+(5+1)

scala>
```

A Higher-Order Function

```
scala> def sum...
sum: (f: (Int) => Int, a: Int, b: Int) => Int

scala> sum(plusone,1,5)
res1: Int = 20        //  (1+1)+(2+1)+(3+1)+(4+1)+(5+1)

scala> sum(square,1,5)
```

A Higher-Order Function

```
scala> def sum...
sum: (f: (Int) => Int, a: Int, b: Int) => Int

scala> sum(plusone,1,5)
res1: Int = 20        //  (1+1)+(2+1)+(3+1)+(4+1)+(5+1)

scala> sum(square,1,5)
res2: Int = 55        // (1*1)+(2*2)+(3*3)+(4*4)+(5*5)

scala>
```

Higher-Order Functions

```
class List[A] ...
  def foreach(f: A => Unit) : Unit
```

Higher-Order Functions

```
class List[A] ...
  def foreach(f: A => Unit) : Unit
```

```
scala>
```

Higher-Order Functions

```
class List[A] ...
  def foreach(f: A => Unit) : Unit
```

```
scala> List(-1,-2,1,2,3).foreach(println)
```

Higher-Order Functions

```
class List[A] ...
  def foreach(f: A => Unit) : Unit
```

```
scala> List(-1,-2,1,2,3).foreach(println)
-1
-2
1
2
3

scala>
```

Higher-Order Functions

```
class List[A] ...
  def filter (p: (A) => Boolean) : List[A]
```

Higher-Order Functions

```
class List[A] ...
  def filter (p: (A) => Boolean) : List[A]
```

```
scala>
```

Higher-Order Functions

```
class List[A] ...
  def filter (p: (A) => Boolean) : List[A]
```

```
scala> List(-1,-2,1,2,3).filter(x => 0<x)
```

Higher-Order Functions

```
class List[A] ...
  def filter (p: (A) => Boolean) : List[A]
```

```
scala> List(-1,-2,1,2,3).filter(x => 0<x)
res0: List[Int] = List(1,2,3)

scala>
```

Higher-Order Functions

```
class List[A] ...
  def filter (p: (A) => Boolean) : List[A]
```

```
scala> List(-1,-2,1,2,3).filter(x => 0<x)
res0: List[Int] = List(1,2,3)

scala> List(-1,-2,1,2,3).filter(0<_)
```

Higher-Order Functions

```
class List[A] ...
  def filter (p: (A) => Boolean) : List[A]
```

```
scala> List(-1,-2,1,2,3).filter(x => 0<x)
res0: List[Int] = List(1,2,3)

scala> List(-1,-2,1,2,3).filter(0<_)
res1: List[Int] = List(1,2,3)

scala>
```

Higher-Order Functions

```
class List[A] ...
  def exists (p: (A) => Boolean) : Boolean
  def forall (p: (A) => Boolean) : Boolean
```

Higher-Order Functions

```
class List[A] ...
  def exists (p: (A) => Boolean) : Boolean
  def forall (p: (A) => Boolean) : Boolean
```

```
scala>
```

Higher-Order Functions

```
class List[A] ...
  def exists (p: (A) => Boolean) : Boolean
  def forall (p: (A) => Boolean) : Boolean
```

```
scala> List(-1,-2,1,2,3).exists(0<_)
```

Higher-Order Functions

```
class List[A] ...
  def exists (p: (A) => Boolean) : Boolean
  def forall (p: (A) => Boolean) : Boolean
```

```
scala> List(-1,-2,1,2,3).exists(0<_)
res0: Boolean = true

scala>
```

Higher-Order Functions

```
class List[A] ...
  def exists (p: (A) => Boolean) : Boolean
  def forall (p: (A) => Boolean) : Boolean
```

```
scala> List(-1,-2,1,2,3).exists(0<_)
res0: Boolean = true

scala> List(-1,-2,1,2,3).forall(0<_)
```

Higher-Order Functions

```
class List[A] ...
  def exists (p: (A) => Boolean) : Boolean
  def forall (p: (A) => Boolean) : Boolean
```

```
scala> List(-1,-2,1,2,3).exists(0<_)
res0: Boolean = true

scala> List(-1,-2,1,2,3).forall(0<_)
res1: Boolean = false

scala>
```

Java vs. Scala

Problem: Hat der String `name` Großbuchstaben?

Java vs. Scala

Problem: Hat der String `name` Großbuchstaben?

Java

```
boolean nameHasUpperCase = false;
for (int i=0;i<name.length();++i) {
  if (Character.isUpperCase(name.charAt(i))) {
    nameHasUpperCase = true;
    break;
  }
}
```

Java vs. Scala

Problem: Hat der String `name` Großbuchstaben?

Java

```
boolean nameHasUpperCase = false;
for (int i=0;i<name.length();++i) {
  if (Character.isUpperCase(name.charAt(i))) {
    nameHasUpperCase = true;
    break;
  }
}
```

Scala

```
val nameHasUpperCase = name.exists(_.isUpper)
```

Higher-Order Functions

```
class List[A] ...
  def map [B] (f: (A) => B) : List[B]
```

Higher-Order Functions

```
class List[A] ...
  def map [B] (f: (A) => B) : List[B]
```

```
scala>
```

Higher-Order Functions

```
class List[A] ...
  def map [B] (f: (A) => B) : List[B]
```

```
scala> List(-1,-2,1,2,3).map(plusone)
```

Higher-Order Functions

```
class List[A] ...
  def map [B] (f: (A) => B) : List[B]
```

```
scala> List(-1,-2,1,2,3).map(plusone)
res0: List[Int] = List(0, -1, 2, 3, 4)

scala>
```

Higher-Order Functions

```
class List[A] ...
  def map [B] (f: (A) => B) : List[B]
```

```
scala> List(-1,-2,1,2,3).map(plusone)
res0: List[Int] = List(0, -1, 2, 3, 4)

scala> List(-1,-2,1,2,3).map(_*2)
```

Higher-Order Functions

```
class List[A] ...
  def map [B] (f: (A) => B) : List[B]
```

```
scala> List(-1,-2,1,2,3).map(plusone)
res0: List[Int] = List(0, -1, 2, 3, 4)

scala> List(-1,-2,1,2,3).map(_*2)
res1: List[Int] = List(-2, -4, 2, 4, 6)

scala>
```

Higher-Order Functions

```
class List[A] ...
  def partition (p: (A) => Boolean) : (List[A], List[A])
```

Higher-Order Functions

```
class List[A] ...
  def partition (p: (A) => Boolean) : (List[A], List[A])
```

```
scala>
```

Higher-Order Functions

```
class List[A] ...
  def partition (p: (A) => Boolean) : (List[A], List[A])
```

```
scala> List(-1,-2,1,2,3).partition(_<0)
```

Higher-Order Functions

```
class List[A] ...
  def partition (p: (A) => Boolean) : (List[A], List[A])
```

```
scala> List(-1,-2,1,2,3).partition(_<0)
res0: (List[Int], List[Int]) =
(List(-1, -2),List(1, 2, 3))

scala>
```

Higher-Order Functions

```
class List[A] ...
  def partition (p: (A) => Boolean) : (List[A], List[A])
```

```
scala> List(-1,-2,1,2,3).partition(_<0)
res0: (List[Int], List[Int]) =
(List(-1, -2),List(1, 2, 3))

scala> val (neg, pos) = List(-1,-2,1,2,3).partition(_<0)
```

Higher-Order Functions

```
class List[A] ...
  def partition (p: (A) => Boolean) : (List[A], List[A])
```

```
scala> List(-1,-2,1,2,3).partition(_<0)
res0: (List[Int], List[Int]) =
(List(-1, -2),List(1, 2, 3))

scala> val (neg, pos) = List(-1,-2,1,2,3).partition(_<0)
neg: List[Int] = List(-1, -2)
pos: List[Int] = List(1, 2, 3)


scala>
```

Subsection 3

Pattern Matching

Aufgabe

Gesucht:

Interpreter für arithmetische Ausdrücke bestehend aus Zahlen und Addition, wie z.B. $1 + (3 + 7)$

Aufgabe

Gesucht:

Interpreter für arithmetische Ausdrücke bestehend aus Zahlen und Addition, wie z.B. $1 + (3 + 7)$

Mögliche Lösung

Abstrakte Klasse `Expr` mit zwei Subklassen `Num` und `Sum`.

Klassisches OO-Design

```
abstract class Expr
class Num(n: Int) extends Expr
class Sum(e1: Expr, e2: Expr) extends Expr
```

Klassisches OO-Design

```
abstract class Expr
class Num(n: Int) extends Expr
class Sum(e1: Expr, e2: Expr) extends Expr
```

$(1 + (3 + 7))$ ist:

```
new Sum(new Num(1), new Sum(new Num(3), new Num(7)))
```

Klassisches OO-Design: eval

```
abstract class Expr
{
  def eval:Int
}
class Num(n: Int) extends Expr
{
  def eval: Int = n
}
class Sum(e1: Expr, e2: Expr) extends Expr
{
  def eval: Int = e1.eval + e2.eval
}
```

Klassisches OO-Design

Vorteil: Einfach um neue Arten von Daten zu erweitern.

Klassisches OO-Design

Vorteil: Einfach um neue Arten von Daten zu erweitern.

```
class Prod(e1: Expr, e2: Expr) extends Expr
{
  def eval: Int = e1.eval * e2.eval
}
```

Klassisches OO-Design

Nachteil: Umständlich um neue Operationen zu erweitern

Klassisches OO-Design

Nachteil: Umständlich um neue Operationen zu erweitern

```
abstract class Expr
{
  def eval:Int
  def print
}
class Num(n: Int) extends Expr
{
  def eval: Int = n
  def print {print(n)}
}
class Sum(e1: Expr, e2: Expr) extends Expr
{
  def eval: Int = e1.eval + e2.eval
  def print {print(e1) print("+") print(e2)}
}
```

Klassisches OO-Design

Nachteil: Umständlich um neue Operationen zu erweitern

```
abstract class Expr
{
  def eval:Int
  def print
}
class Num(n: Int) extends Expr
{
  def eval: Int = n
  def print {print(n)}
}
class Sum(e1: Expr, e2: Expr) extends Expr
{
  def eval: Int = e1.eval + e2.eval
  def print {print(e1) print("+") print(e2)}
}
```

Nachteil: Komplizierte Operationen schwer zu verstehen

Klassisches OO-Design

Nachteil: Umständlich um neue Operationen zu erweitern

```
abstract class Expr
{
  def eval:Int
  def print
}
class Num(n: Int) extends Expr
{
  def eval: Int = n
  def print {print(n)}
}
class Sum(e1: Expr, e2: Expr) extends Expr
{
  def eval: Int = e1.eval + e2.eval
  def print {print(e1) print("+") print(e2)}
}
```

Nachteil: Komplizierte Operationen schwer zu verstehen

Nachteil: Operationen, die tiefer in die Daten hineinschauen

Funktionales Design

Algebraische Datentypen, in Scala genannt: case classes.

```
abstract class Expr
case class Num(n: Int) extends Expr
case class Sum(e1: Expr, e2: Expr) extends Expr
```

Funktionales Design

Algebraische Datentypen, in Scala genannt: case classes.

```
abstract class Expr
case class Num(n: Int) extends Expr
case class Sum(e1: Expr, e2: Expr) extends Expr
```

case classes haben:

- Verschiedene Nettigkeiten
  - Konstruktorfunktion mit demselben Namen:
    $(1 + (3 + 7))$ ist `Sum(Num(1), Sum(Num(3), Num(7)))`
  - Accessorfunktionen mit demselben Namen wie die Parameter
  - `toString`, `equals` etc. sind sinnvoll überschrieben
- Pattern Matching!

Pattern Matching

```
abstract class Expr
case class Num(n: Int) extends Expr
case class Sum(e1: Expr, e2: Expr) extends Expr

def eval(e: Expr): Int = e match
{
  case Num(n) => n
  case Sum(l,r) => eval(l) + eval(r)
}
```

Pattern Matching

```
abstract class Expr
case class Num(n: Int) extends Expr
case class Sum(e1: Expr, e2: Expr) extends Expr
case class Prod(e1: Expr, e2: Expr) extends Expr

def eval(e: Expr): Int = e match
{
  case Num(n) => n
  case Sum(l,r) => eval(l) + eval(r)
}
```

Pattern Matching

```
abstract class Expr
case class Num(n: Int) extends Expr
case class Sum(e1: Expr, e2: Expr) extends Expr
case class Prod(e1: Expr, e2: Expr) extends Expr

def eval(e: Expr): Int = e match
{
  case Num(n) => n
  case Sum(l,r) => eval(l) + eval(r)
  case Prod(l,r) => eval(l) * eval(r)
}
```

Sealed Class

```
sealed abstract class Expr
case class Num(n: Int) extends Expr
case class Sum(e1: Expr, e2: Expr) extends Expr
case class Prod(e1: Expr, e2: Expr) extends Expr

def eval(e: Expr): Int = e match
{
  case Num(n) => n
  case Sum(l,r) => eval(l) + eval(r)
  case Prod(l,r) => eval(l) * eval(r)
}
```

Sealed Class

```scala
sealed abstract class Expr
case class Num(n: Int) extends Expr
case class Sum(e1: Expr, e2: Expr) extends Expr
case class Prod(e1: Expr, e2: Expr) extends Expr

def eval(e: Expr): Int = e match
{
  case Num(n) => n
  case Sum(l,r) => eval(l) + eval(r)
  // case Prod missing
}
```

Sealed Class

```
sealed abstract class Expr
case class Num(n: Int) extends Expr
case class Sum(e1: Expr, e2: Expr) extends Expr
case class Prod(e1: Expr, e2: Expr) extends Expr

def eval(e: Expr): Int = e match
{
  case Num(n) => n
  case Sum(l,r) => eval(l) + eval(r)
  // case Prod missing
}
```

scala compiler:

```
warning: match is not exhaustive!
missing combination: Prod
```

Inhalt

1. Objektmodell von Scala
   - Hello world
   - Eine Klasse
   - Nutzen des Objektmodells
2. Funktionales Programmieren
   - Mutability considered harmful
   - Functions are Values
   - Pattern Matching
3. A Domain Specific Language

Section 3

A Domain Specific Language

Domain Specific Languages

Beispiele

- Abfragesprachen wie SQL
- Parsergeneratorformat wie lex, yacc

Domain Specific Languages

Beispiele

- Abfragesprachen wie SQL
- Parsergeneratorformat wie lex, yacc

Ziel von Scala

DSLs innerhalb von Scala durch Bibliotheken nachbilden

Aufgabe: Parsen unserer Ausdrücke

Grammatik

$\texttt{expr} ::= \texttt{number | "(" expr "+" expr ")"}$

Aufgabe: Parsen unserer Ausdrücke

Grammatik

```
expr ::=  number | "(" expr "+" expr ")"
```

Aufgabe: Parsen unserer Ausdrücke

Grammatik

```
expr ::= number | "(" expr "+" expr ")"
```

Aufgabe: Parsen unserer Ausdrücke

Grammatik

```
expr ::=  number | "(" expr "+" expr ")"
```

In Scala

```
def expr = number | "(" ~expr ~"+" ~expr ~ ")"
```

Drumherum 1

```
import scala.util.parsing.combinator._

class Arith extends JavaTokenParsers {
  def expr: Parser[Any] =
    number | "(" ~expr ~"+" ~ expr ~ ")"
  def number: ...
}
```

Drumherum 2

```
import scala.util.parsing.combinator._

class Arith extends JavaTokenParsers {
  def expr: Parser[Any] =
    number ^^
      {x => Num(x.toInt)}
    |
    "(" ~> expr ~"+" ~ expr <~ ")" ^^
      {case p ~"+" ~ q => Sum(p,q)}
  def number: ...
}
```

Drumherum 2

```
import scala.util.parsing.combinator._

class Arith extends JavaTokenParsers {
  def expr: Parser[Any] =
    number ^^
      {x => Num(x.toInt)}
    |
    "(" ~> expr ~"+" ~ expr <~ ")" ^^
      {case p ~"+" ~ q => Sum(p,q)}
  def number: ...
}


scala>
```

Drumherum 2

```
import scala.util.parsing.combinator._

class Arith extends JavaTokenParsers {
  def expr: Parser[Any] =
    number ^^
      {x => Num(x.toInt)}
    |
    "(" ~> expr ~"+" ~ expr <~ ")" ^^
      {case p ~"+" ~ q => Sum(p,q)}
  def number: ...
}
```

```
scala> parseAll(expr,"(1 + (3 + 7))")
```

Drumherum 2

```
import scala.util.parsing.combinator._

class Arith extends JavaTokenParsers {
  def expr: Parser[Any] =
    number ^^
      {x => Num(x.toInt)}
    |
    "(" ~> expr ~"+" ~ expr <~ ")" ^^
      {case p ~"+" ~ q => Sum(p,q)}
  def number: ...
}
```

```
scala> parseAll(expr,"(1 + (3 + 7))")
res0: Expr = Sum(Num(1), Sum(Num(3), Num(7)))

scala>
```

Drumherum 2

```
import scala.util.parsing.combinator._

class Arith extends JavaTokenParsers {
  def expr: Parser[Any] =
    number ^^
      {x => Num(x.toInt)}
    |
    "(" ~> expr ~"+" ~ expr <~ ")" ^^
      {case p ~"+" ~ q => Sum(p,q)}
  def number: ...
}
```

```
scala> parseAll(expr,"(1 + (3 + 7))")
res0: Expr = Sum(Num(1), Sum(Num(3), Num(7)))

scala> eval(res0)
```

Drumherum 2

```
import scala.util.parsing.combinator._

class Arith extends JavaTokenParsers {
  def expr: Parser[Any] =
    number ^^
      {x => Num(x.toInt)}
    |
    "(" ~> expr ~"+" ~ expr <~ ")" ^^
      {case p ~"+" ~ q => Sum(p,q)}
  def number: ...
}
```

```
scala> parseAll(expr,"(1 + (3 + 7))")
res0: Expr = Sum(Num(1), Sum(Num(3), Num(7)))

scala> eval(res0)
res1: Int = 11
```

Rückblick

1. Objektmodell von Scala
   - Hello world
   - Eine Klasse
   - Nutzen des Objektmodells
2. Funktionales Programmieren
   - Mutability considered harmful
   - Functions are Values
   - Pattern Matching
3. A Domain Specific Language