5.– 8. September 2011
in Nürnberg

Herbstcampus

Wissenstransfer
par excellence

Nur echt mit 52 Zähnen

Programmieren mit Monaden in Scala

Thomas Much

www.muchsoft.com

Keine Scala-Vorkenntnisse und kein Mathematik-Studium erforderlich!

Was sind Monaden?

- Die Geheimwaffe der funktionalen Programmierung!

- Mächtig.
- Sagenumwoben.
- Furchteinflößend.

Ziel dieses Vortrags + Voraussetzungen

- Dieser Vortrag soll zeigen
  - warum Monaden nötig bzw. sinnvoll sind.
  - welche Operationen in einer Monade vorhanden sein müssen (und wie sie in Scala funktionieren).
  - dass wichtige Scala-Standard-Datentypen monadisch sind (und wie man das praktisch nutzen kann).
- Scala-Vorkenntnisse sind *nicht* zwingend notwendig, aber man sollte zumindest Spaß an Java (oder einer vergleichbaren Sprache) haben.
- Die mathematische Theorie wird auf ein Mindestmaß reduziert, lässt sich bei diesem Thema aber nicht ganz vermeiden.

Was sind Monaden?

- Monade (gr. μονάς, lat. monas, die Einzelheit, die Einheit)
- Gottfried Wilhelm Leibniz (1646-1716):
  - Vergleichbares Konzept zu Atomen in der Physik für die Philosophie
  - Kleinste Einheiten (einfache, unteilbare Substanzen, d.h. wirkende „Wesen“ bzw. Seelen), die die Welt zum Leben erwecken (so ungefähr...)
  - „Monaden sind überall“
  - Ein bisschen schwammig... ;-)

Was sind Monaden?

- Ein Konzept aus der Kategorientheorie der Mathematik.
- Verkettbare Funktionen mit einem Zustand, die den Zustand kapseln (auch beim Überführen in einen anderen Zustand).
- Damit kann man sehr formell, mathematisch programmieren.

- Alternativ können wir Monaden als Art der Strukturierung von – vor allem funktionalen – Programmen sehen.
- Also als eine Art **funktionales Entwurfsmuster**.

Vor allem dieser Aspekt interessiert uns in diesem Vortrag!

Warum wurden Monaden erfunden?

- In *rein funktionalen Sprachen* sind (Neben-)Wirkungen („Seiteneffekte“) nicht erlaubt, sondern nur Funktionsergebnisse!
  - Man möchte unabhängig vom Systemzustand/Zeitpunkt des Aufrufs sein – Funktionen sollen bei gleicher Eingabe immer das gleiche Ergebnis liefern.
- Ein-/Ausgabe hat aber eigentlich nur Wirkung (und kein Ergebnis)
  - Keine I/O in funktionalen Sprachen?
  - Doch – selbst die Puristen haben eingesehen, dass jedes halbwegs nützliche Programm I/O braucht...

Warum heißen Monaden „Monaden“?

- Für die Ein-/Ausgabe in rein funktionalen Sprachen wurde eine passende Programmiertechnik entwickelt, die den Systemzustand (die „Welt“) kapselt und die Sprache frei von Nebenwirkungen lässt.
- Irgendwann wurde entdeckt, dass diese Programmiertechnik zufällig die Monadenaxiome erfüllt.
  - Und nicht nur in Haskell klingt *IO-Monade* viel schicker als IO-Datentyp.
  - *"Warm fuzzy thing"*

- Scala ist keine rein funktionale Sprache, die IO-Monade ist hier nicht wirklich sinnvoll.

Aber von Anfang an...

- Im Folgenden sehen wir einige Scala-Grundlagen zu Funktionen und Listen.
- Nebenbei sehen wir dabei diverse monadische Operationen (Funktionen) in Aktion.

- Wenn wir alles beisammen haben, schauen wir uns die theoretischen Grundlagen an (nicht im Detail!).

- Schließlich schreiben wir selber eine Monade und sehen uns diverse Monaden aus der Scala-Standard-Bibliothek an.

Listen & die Grundlagen von Scala

```
scala> 1
res0: Int = 1
```

Wer kennt den REPL *nicht*?

```
scala> List(1)
res1: List[Int] = List(1)
```

Konstruktor bzw. Fabrik

```
scala> List(1,2,3)
res2: List[Int] = List(1, 2, 3)
```

Listen aufbauen

```
scala> 1 :: 2 :: List(3)
res3: List[Int] = List(1, 2, 3)

scala> 1 :: 2 :: 3 :: List()
res4: List[Int] = List(1, 2, 3)

scala> 1 :: 2 :: 3 :: Nil
res5: List[Int] = List(1, 2, 3)
```

Leere Listen

```
scala> List()
res6: List[Nothing] = List()


scala> Nil
res7: scala.collection.immutable.Nil.type = List()


scala> res6 == res7
res8: Boolean = true
```

Listen verketten (1)

```
scala> List(1,2) :: 3 :: Nil
res9: List[Any] = List(List(1, 2), 3)
```


```
scala> List(1,2) :: List(3) :: Nil
res10: List[List[Int]] = List(List(1, 2), List(3))
```


```
scala> List(1,2) ::: 3 :: Nil
res11: List[Int] = List(1, 2, 3)
```

Typinferenz

```
scala> val lst1 = List(1,2,3)
lst1: List[Int] = List(1, 2, 3)
```

Implizite Typisierung durch Typinferenz

```
scala> val lst2: List[Int] = List(1,2,3)
lst2: List[Int] = List(1, 2, 3)
```

Explizite Typisierung

```
scala> val lst3: List[Int] = List(1,2) :: 3 :: Nil
<console>:5: error: type mismatch;
...
```

Listen verketten (2)

```
scala> List(1,2,3) ::: Nil
res12: List[Int] = List(1, 2, 3)

scala> Nil ::: List(1,2,3)
res13: List[Int] = List(1, 2, 3)

scala> Nil ::: List(1,2,3) ::: Nil ::: Nil
res14: List[Int] = List(1, 2, 3)
```

Listen ausgeben bzw. verarbeiten

```
scala> for (i <- List(1,2,3)) print(i*4 + " ")
4 8 12
```

Kein Ergebnis, nur (Neben-)Wirkung

```
scala> for (i <- List(1,2,3)) yield i*4
res15: List[Int] = List(4, 8, 12)
```

„for-Comprehension“

```
scala> List(1,2,3) map { x => x*4 }
res16: List[Int] = List(4, 8, 12)
```

Wow! Oder? ☺

Listenwerte mit Funktionen abbilden (1)

```
scala> List(1,2,3)
res17: List[Int] = List(1, 2, 3)
```

```
scala> { x: Int => x*4 }
scala> (x: Int) => x*4
res18: (Int) => Int = <function1>
```

Funktionsliteral (anonyme Funktion)

```
scala> res17 map res18
res19: List[Int] = List(4, 8, 12)
```

res18(7) => 28

Listenwerte mit Funktionen abbilden (2)

```
scala> def malVier(x:Int) = x*4
malVier: (x: Int)Int


scala> List(1,2,3) map malVier
res20: List[Int] = List(4, 8, 12)


scala> List(1,2,3) map { _*4 }
res21: List[Int] = List(4, 8, 12)
```

Funktionsabschlüsse (Closures)

```
scala> for (start <- 20 to 30 by 5) {
     |   println( res17 map { x => start + x } )
     | }
List(21, 22, 23)
List(26, 27, 28)
List(31, 32, 33)
```

- Können wir das Funktionsliteral außerhalb der Schleife definieren?

Teilweise (partiell) angewendete Funktionen (1)

```
scala> (start: Int, x: Int) => start+x
res22: (Int, Int) => Int = <function2>


scala> res22( 20, _:Int )
res23: (Int) => Int = <function1>


scala> res23(3)
res24: Int = 23
```

Teilweise (partiell) angewendete Funktionen (2)

```
scala> for (start <- 20 to 30 by 5) {
     |   val add = res22(start,_:Int)
     |   println( res17 map add )
     | }
Ausgabe…
```


```
scala> for (start <- 20 to 30 by 5) {
     |   println( res17 map { res22(start,_:Int) } )
     | }
Ausgabe…
```

Schönfinkeln (Currying)

```
scala> def adder(start: Int)(x: Int) = start + x
adder: (start: Int)(x: Int)Int


scala> adder(20)_
res25: (Int) => Int = <function1>


scala> for (start <- 20 to 30 by 5) {
     |   println( res17 map adder(start)_ )
     | }
Ausgabe…
```

... Scala-Grundlagen }

„Normales“ Scala ENDE.

(Oder doch nicht? ☺)

Wir schauen uns nochmal an, wie wir Listenwerte abbilden.

Verschachtelte Listen

```
scala> List(1,2,3) map { x => println(x);x }
1
2
3
res26: List[Int] = List(1, 2, 3)

scala> List(List(1,2),List(3)) map { x => println(x);x }
List(1, 2)
List(3)
res27: List[List[Int]] = List(List(1, 2), List(3))
```

Einfach ☺

Aber: Wie bekommen wir diese Liste flach?

Listen flachklopfen

```
scala> List(List(1,2),List(3)).flatten
res28: List[Int] = List(1, 2, 3)
```

```
scala> List(1,2,3).flatten
<console>:6: error: could not find implicit value
for parameter asTraversable: (Int) => Traversable[B]
```

- Wie können wir „flatten“ allgemeiner formulieren?

flatEric – äh – flatMap

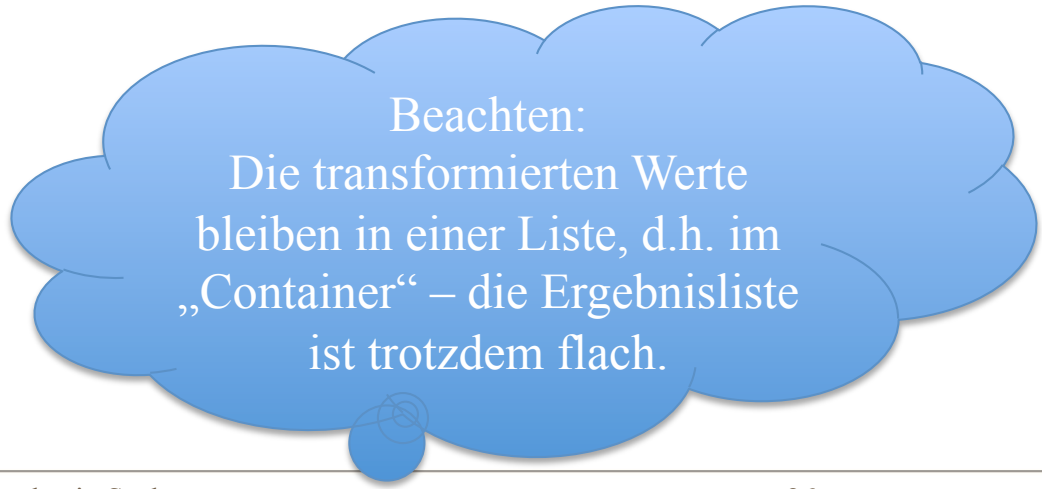
Scala verwendet hier Traversable[B]

```
class List[A] {
  def flatMap[B](f: (A)=>List[B]): List[B]
}
```

```
scala> List(1,2,3) flatMap {x => println(x);x }
<console>:6: error: type mismatch;
 found   : Int
 required: Traversable[?]
       List(1,2,3) flatMap {x => x }
                              ^
```

flatMap korrekt eingesetzt

```
scala> List(1,2,3) flatMap {x => println(x);List(x) }
1
2
3
res29: List[Int] = List(1, 2, 3)
```



flatMap und flatten

```
scala> List(List(1,2),List(3)) flatMap {x => println(x);x}
List(1, 2)
List(3)
res30: List[Int] = List(1, 2, 3)
```

Das ist exakt „flatten“!

```
scala> List(List(1,2),List(3)) flatMap {x =>
     |   println(x);List(x) }
List(1, 2)
List(3)
res31: List[List[Int]] = List(List(1, 2), List(3))
```

So erwartet?

Was können flatten und flatMap noch?

- Leere Elemente entfernen:

```
scala> List(List(1),Nil,List(2))
res32: List[List[Int]] = List(List(1),List(),List(2))


scala> res32.flatten
res33: List[Int] = List(1, 2)
```

Was kann flatMap noch? (1)

- Leere Elemente programmatisch erzeugen (= filtern!):

```
scala> List(1,2,3) flatMap { x =>
     |   if (x%2==0) Nil else List(x) }
res34: List[Int] = List(1, 3)
```

```
scala> List(1,2,3) flatMap { x =>
     |   if (x%2==0) None else Some(x) }
res35: List[Int] = List(1, 3)
```

Was kann flatMap noch? (2)

- Werte und Typen transformieren:

```
scala> List(1,2,3) flatMap { x => List("#" + x*4) }
res36: List[java.lang.String] = List(#4, #8, #12)
```

flatMap vs. Filter

```
scala> List(1,"zwei",3)
res37: List[Any] = List(1, zwei, 3)


scala> res37 filter { _.isInstanceOf[Int] }
res38: List[Any] = List(1, 3)


scala> res37 flatMap {
     |   case i: Int => Some(i)
     |   case _ => None
     | }
res39: List[Int] = List(1, 3)
```

Listen – was haben wir gesehen? (1)

- Typkonstruktor
  - `List[A]` mit z.B. `List[Int]` als Ausprägung/Instanziierung
- Funktion, um *einen* Wert in einer Liste zu „verpacken“
  - z.B. List(1)
- Funktion, die Werte der Liste transformiert, aber in einer Liste belässt
  - `flatMap`
- Das sind bereits alle Voraussetzungen für eine Monade!

Listen – was haben wir gesehen? (2)

- Zusätzlich haben wir noch folgende Eigenschaften untersucht:
  - Null-Wert: `List()` bzw. `Nil`
  - Funktionen `map` und `flatten`
  - Verkettung / Komposition von Listen mit `:::`

- Scalas List-Klasse ist also eine Monade!

Was sind Monaden?

- In der Kategorientheorie ist eine Monaden ein Tripel (T,η,μ):
  - T ist ein *Funktor*, der eine *Kategorie* (einen generischen/polymorphen Typ) in sich selbst abbildet.
  - η (eta, i) ist eine natürliche Transformation für die Identität, die die Existenz eines neutralen Elements garantiert.
  - μ ist eine natürliche Transformation für die Verkettung, die das Assoziativgesetz einhält.

- Daraus ergeben sich die Monadengesetze (monadischen Axiome), vergleichbar mit denen von *Monoiden*.

Im Detail später am praktischen Beispiel

Was sind Monaden?

- In der funktionalen Programmierung sind Monaden abstrakte Datentypen mit folgenden Komponenten:
  - *Typkonstruktor* (oft synonym mit der ganzen Monade verwendet): Ist M der Name der Monade und t der Datentyp, so ist M t der korrespondierende monadische Typ.
  - *Einheitsfunktion*: Bildet einen Wert des zugrunde liegenden Typs auf den monadischen Typ ab.
  - *Bindeoperation*: Bildet den zugrunde liegenden Wert eines monadischen Typs auf einen anderen monadischen Typ ab.

Was sind Monaden?

- In der funktionalen Programmierung sind Monaden abstrakte Datentypen mit folgenden Komponenten:
  - *Typkonstruktor* (oft synonym mit der ganzen Monade): Ist M der Name der Monade und t der Datentyp, so ist M t der korrespondierende monadische Typ.
  - *Einheitsfunktion*: Bildet einen Wert des zugr… monadischen Typ ab.
  - *Bindeoperation*: Bildet den zugrunde liegenden Wert eines monadischen Typs auf einen anderen monadischen Typ ab
  - Optional flatten (join) und die monadische Null (z.B. Nil)

List[A] bzw. List[Int]

return, unit, lift, pure... List(1)
„einwickeln“
Neutrales Element!

bind, flatMap
„ausrollen“

Was sind Monaden?

- In der Informatik sind Monaden *ein* Modell für Berechnungen (andere sind *Arrows* und *applikative Funktoren*).
- Monaden kombinieren diverse Techniken:
  - Continuations
  - Single-Threadedness
    (kein Welt-Ausgangszustand darf zweimal verarbeitet werden)
  - Kapselung
- Monaden sind also gutes Marketing für Altbekanntes.

Monaden sind Kollektionen (1)

- Durch das „Einwickeln“ von Werten in der Monade werden Monaden häufig mit Containern gleichgesetzt.
- Monaden sind aber nicht nur das, sondern allgemeiner „Transformationen“ (bzw. Berechnungen).
- Besser betrachtet man Monaden daher als Kollektionen:
  - Container speichern bestimmte, hineingepackte Werte.
  - Berechnungen können beliebige (unendliche) Folgen liefern.

Alle geraden Zahlen, die Fibonacci-Folge...
Aber auch Programme (DSLs!), XML etc.

Monaden sind Kollektionen (2)

- Die Entscheidung, ob man Werte fest im Container speichert oder berechnen lässt, ist eine Abwägung zwischen Rechenaufwand und Speicherbedarf.
  - Bei unendlichen Folgen wird man die Werte berechnen (Lazy Collections).

- Durch berechnete Kollektionen verschwimmt die Grenze zwischen Berechnung und Ein-/Ausgabe – beides sieht für das nutzende Programm gleich aus.

Wir bauen uns eine Monade (1)

In der Praxis vermutlich mit (Co-)Varianz

```
case class MiniMonade[A](wert:A) {
   def bind[B](f: A => MiniMonade[B]) = f(wert)
}
```

Durch case-Klasse kürzere Schreibweise möglich (ohne new)

```
def plusEins(i: Int) = new MiniMonade( i+1 )
def alsString(a: Any) = new MiniMonade( a.toString )
```

Wir bauen uns eine Monade (2)

```
val m1 = new MiniMonade(1)

val m2 = m1 bind plusEins
val m3 = m1 bind alsString
assert(m2 != m3)


val kette = m1 bind plusEins bind plusEins
println(kette.wert)
```

Wie bekommt man Werte aus der Monade? Darum kümmern sich *Co-Monaden* (List etc. sind beides in einem)

Monaden-Axiome in der Praxis

Links-/Rechts-Identität

```
assert((new MiniMonade(7) bind plusEins) == plusEins(7))
assert((m1 bind {x => new MiniMonade(x)}) == m1)
```

Assoziativität

```
val m4 = (m1 bind plusEins) bind alsString
val m5 = m1 bind {m => plusEins(m) bind alsString}
assert(m4 == m5)
```

Und was bringt uns das?

Verkettbarkeit von Monaden (1)

```
val zeilen = List(10,20)
val spalten = List(1,2)
val farben = List("rot","blau")


val erg1 = zeilen flatMap {
  z => spalten flatMap {
   s => farben map {
     f => f+(z+s)
}}}


println(erg1)
```

Jede Menge Closures

List(rot11, blau11,
rot12, blau12,
rot21, blau21,
rot22, blau22)

Verkettbarkeit von Monaden (2)

- Mit for-Comprehensions sieht das etwas übersichtlicher aus:

```
val erg2 = for {
  z <- zeilen
  s <- spalten
  f <- farben
}
yield f+(z+s)
```

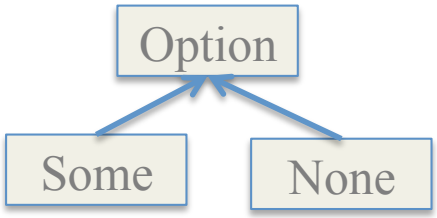
Option – eine ganz gewöhnliche Monade

```
val vorname: Option[String] = Some("Thomas")
val nachname: Option[String] = Some("Much") // oder None
```

z.B. aus Formulareingaben

```
val name = for {
  vn <- vorname
  nn <- nachname
}
yield vn + " " + nn
```

flatMap-“Magie“ – erspart viele null-Prüfungen!




```
println(name)  // Some("Thomas Much") bzw. None
```

Verkettung unterschiedlicher Monaden-Typen

```
val plz = List(22589,91052,12345)
val orte = Map(22589 -> "Hamburg", 91052 -> "Erlangen")


val erg = for {
  p <- plz
  o <- orte get(p) orElse(Some("kein Ort gefunden"))
}
yield o


println( erg )
```

List(Hamburg, Erlangen, kein Ort gefunden)
bzw. ohne orElse: List(Hamburg, Erlangen)

Überall Monaden... (1)

```
val xml =
  <adressen>
    <adresse><plz>22589</plz><ort>Hamburg</ort></adresse>
    <adresse><plz>91052</plz><ort>Erlangen</ort></adresse>
    <adresse><plz></plz><ort>Entenhausen</ort></adresse>
    <adresse><ort>Posemuckel</ort></adresse>
  </adressen>

val plzElemente = xml \\ "adresse" \ "plz"

println( plzElemente )
// <plz>22589</plz><plz>91052</plz><plz></plz>
```

Überall Monaden... (2)

Sequenzen sind auch Monaden

```
import scala.xml._

def extrahiereZahlen(pzn: NodeSeq): Option[String] = {
   val plzStr = pzn.text
   if (plzStr != "") Some(plzStr) else None
}
```

Da ist es wieder

```
val zahlen = plzElemente flatMap extrahiereZahlen

println( zahlen )
// List(22589, 91052)
```

Monaden in Scala (1)

- Fast alles, was in der Scala-Standard-Bibliothek „flatMap“ implementiert, ist ein monadischer Typ.
- Nett, weil man dann ungefähr weiß, wie man diese Typen verarbeiten kann (Entwurfsmuster!).

- Aber... Wann wären Monaden noch nützlicher?
  - Wenn die monadischen Typen einen Standard-Monaden-Typ erweiterten, der alle Funktionsnamen (Schnittstellen) festlegt.
  - Genau das bietet Scala *nicht*.

Monaden in Scala (2)

- Es gibt zwar Typen bzw. Traits für bestimmte Anwendungsfälle (FilterMonadic), die sind aber unvollständig.

Was fehlt hier?

```
package scala.collection.generic

/** A template trait that contains just the `map`, `flatMap`, `foreach` and `withFilter` methods
 *  of trait `TraversableLike`.
 */
trait FilterMonadic[+A, +Repr] {
  def map[B, That](f: A => B)(implicit bf: CanBuildFrom[Repr, B, That]): That
  def flatMap[B, That](f: A => Traversable[B])(implicit bf: CanBuildFrom[Repr, B, That]): That
  def foreach[U](f: A => U): Unit
  def withFilter(p: A => Boolean): FilterMonadic[A, Repr]
}
```

Monaden in scalaz

- Einen vollständigen Monaden-Typ bietet die Bibliothek „scalaz“ an (*http://code.google.com/p/scalaz/*):

```
trait Monad[M[_]] extends Applicative[M] with Bind[M] with Pointed[M] {
  override def fmap[A, B](fa: M[A], f: A => B) = bind(fa, (a: A) => pure(f(a)))
  override def apply[A, B](f: M[A => B], a: M[A]): M[B] = bind(f, (k: A => B) => fmap(a, k(_: A)))
}

object Monad {
  implicit def monad[M[_]](implicit b: Bind[M], p: Pure[M]): Monad[M] = new Monad[M] {
    override def pure[A](a: => A) = p.pure(a)
    override def bind[A, B](a: M[A], f: A => M[B]) = b.bind(a, f)
  }
  …
}
```

- Thema eines anderen Vortrags!

Scala-Compiler erkennt monadische Typen

```
case class MiniMonade[A](wert:A) {
  def flatMap[B](f: A => MiniMonade[B]) = f(wert)
  def map[B](f: A => B): MiniMonade[B]
                              = MiniMonade(f(wert))
}

val m1 = MiniMonade(1)

val erg = for (m <- m1) yield m+1

println( erg )  // MiniMonade(2)
```

"extends FilterMonadic" nicht notwendig!

Wird abgebildet auf
m1.map(m => m+1)
Entspricht
m1.flatMap(m => unit(m+1))

yield ist quasi Scalas *unit* bzw. *pure*

Scala ist objekt-funktional

Jedes Monaden-Objekt hat automatisch einen Zustand

```
case class CounterMonade(zaehlerstand : Int) {
  def flatMap(f: Int => CounterMonade) = f(zaehlerstand)

  def increment =  this flatMap plusEins
  private def plusEins(i: Int) = CounterMonade( i+1 )
}

val m1 = new CounterMonade(0)
val m2 = m1.increment
println(m2.zaehlerstand)
```

Bei rein funktionalen Ansätzen würde man hier zusätzlich die Monade (ggfs. mit aktuellem Zustand) übergeben

Fazit

- Monaden sind
  - abstrakte Datentypen für verkettbare Berechnungen.
  - wie Kollektionen (Container, Berechnungen).

  - nicht perfekt, aber zur Zeit die beste Lösung, um nützliche (interagierende) Programme zu entwerfen, ohne funktionalen Sprachen (Neben-)Wirkungen aufzuzwängen.

  - „das“ funktionale Entwurfsmuster
    - um Code verkettbar zu strukturieren (Komposition)
    - damit Komplexität beherrschbar skaliert.

  - überall ☺ – und nicht wirklich schwierig *zu nutzen*.

Literatur & Quellen

- Pepper/Hofstedt, „Funktionale Programmierung“, Springer 2006
- Greg Meredith, „Monadic Design Patterns for the Web“, C9 Lectures 2010/2011, *http://channel9.msdn.com/Tags/greg+meredith*
- Bernie Pope, „Monads in Scala“, *http://www.berniepope.id.au/docs/scala_monads.pdf*

- Debasish Ghosh, *http://debasishg.blogspot.com/*
- Daniel Spiewak, *http://www.codecommit.com/*
- James Iry, *http://james-iry.blogspot.com/*
- Tony Morris, *http://blog.tmorris.net/*

5.– 8. September 2011
in Nürnberg

Herbstcampus

Wissenstransfer
par excellence

Vielen Dank!

Thomas Much

thomas@muchsoft.com

www.muchsoft.com