5.– 8. September 2011
in Nürnberg

Herbstcampus

Wissenstransfer
par excellence

Relationell auch ohne SQL

Relationale Datenbanken mit ScalaQuery nutzen

Stefan Zeiger
Commerzbank AG

Relationale Datenbanken

- Größere Anwendungen brauchen oft Datenbanken
- Relationales Modell verhindert Silobildung

Wozu? Wir haben doch JDBC

Wozu? Wir haben doch JDBC

```
def usersMatching(pattern: String)(conn: Connection) = {
  val st = conn.prepareStatement("select id, name from users where name like ?")
  try {
    st.setString(1, pattern)
    val rs = st.executeQuery()
    try {
      val b = new ListBuffer[(Int, String)]
      while(rs.next)
        b.append((rs.getInt(1), rs.getString(2)))
      b.toList
    } finally rs.close()
  } finally st.close()
}

Class.forName("org.h2.Driver")
val conn = DriverManager.getConnection("jdbc:h2:test1")
try {
  println(usersMatching("%zeiger%")(conn))
} finally conn.close()
```

JDBC

- Gute Grundlage für Frameworks
- Zu niedrige Abstraktionsebene für Anwendungen

ScalaQuery: Simple Queries

```
val usersMatching = query[String, (Int, String)]
  ("select id, name from users where name like ?")

Database.forURL("jdbc:h2:test1", driver = "org.h2.Driver") withSession {
  println(usersMatching("%zeiger%").list)
}
```

Wozu? Wir haben doch ORMs

- Object/Relational Mapping Tools
  - Hibernate, Toplink, JPA

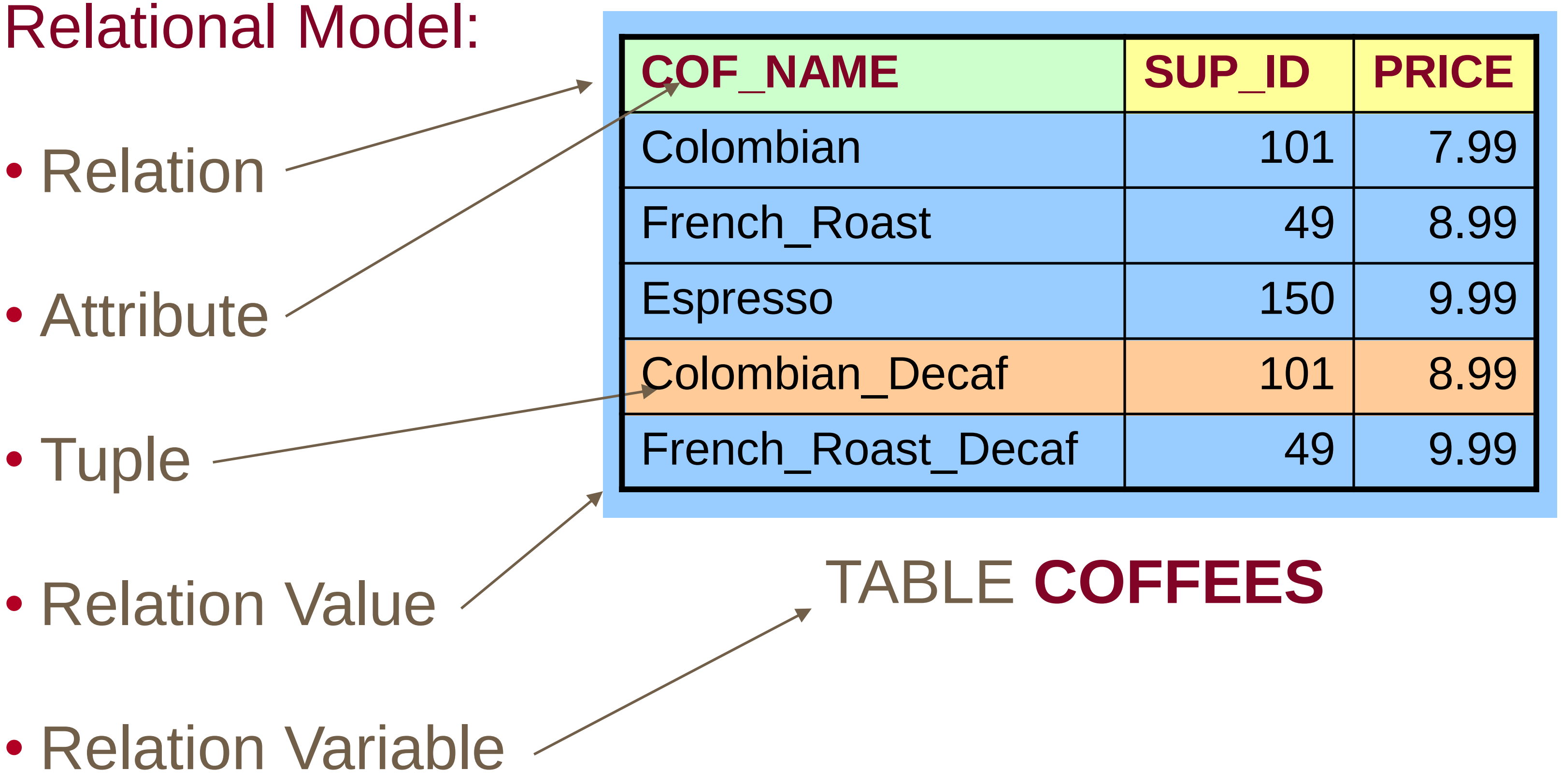
95%

- Lösen 80% des Problems

50%

Relationales Modell

Relational Model:



- Relation
- Attribute
- Tuple
- Relation Value
- Relation Variable

COF_NAME	SUP_ID	PRICE
Colombian	101	7.99
French_Roast	49	8.99
Espresso	150	9.99
Colombian_Decaf	101	8.99
French_Roast_Decaf	49	9.99

TABLE **COFFEES**

Beispiele aus: http://download.oracle.com/javase/tutorial/jdbc/basics/index.html

Impedance Mismatch: Konzepte

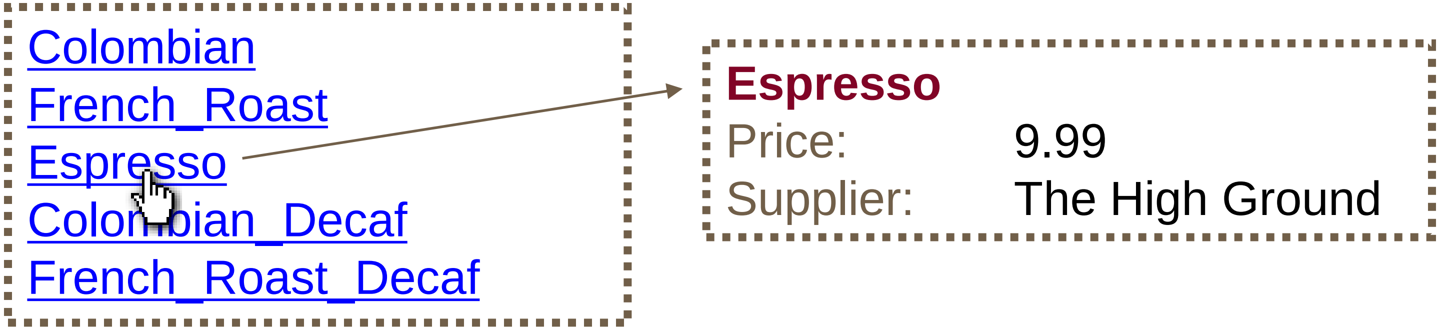
Object-Oriented:

- Identity
- State
- Behaviour
- Encapsulation

Relational:

- ~~Identity~~
- State: Transactional
- ~~Behaviour~~
- ~~Encapsulation~~

Impedance Mismatch: Retrieval Strategies



Colombian
French_Roast
Espresso
Colombian_Decaf
French_Roast_Decaf

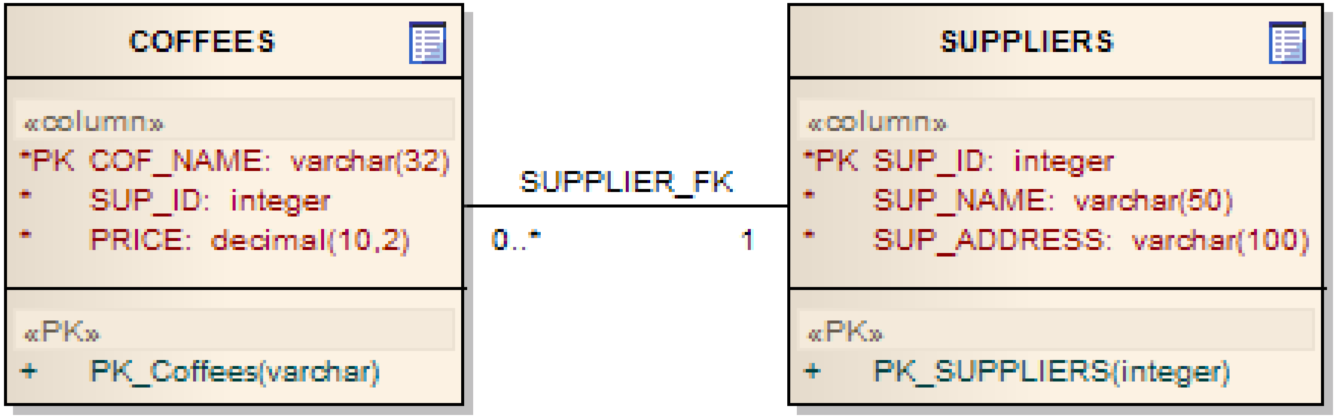
Espresso

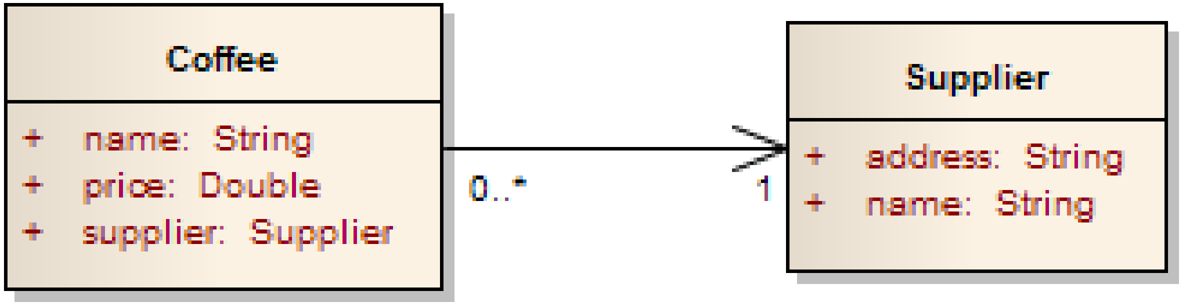
Price:	9.99
Supplier:	The High Ground

```
select COF_NAME
from COFFEES
```

```
select c.*, s.SUP_NAME
from COFFEES c, SUPPLIERS s
where c.COF_NAME = ?
and c.SUP_ID = s.SUP_ID
```

Impedance Mismatch: Retrieval Strategies



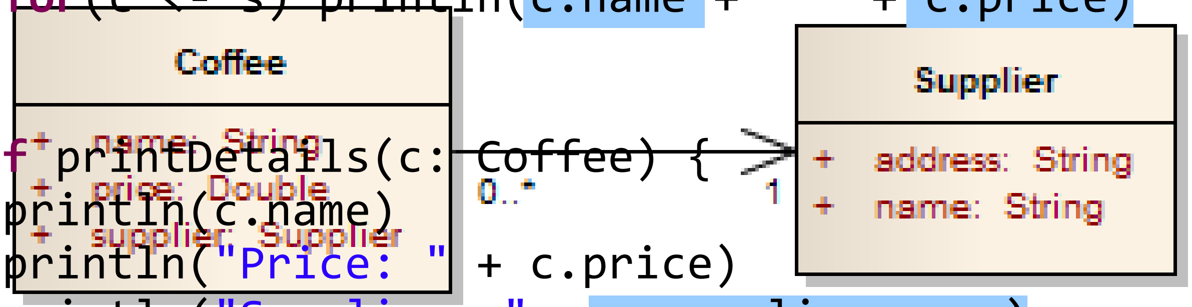


Impedance Mismatch: Retrieval Strategies

```
  def getAllCoffees(): Seq[Coffee] = …

  def printLinks(s: Seq[Coffee]) {
    for(c <- s) println(c.name + " " + c.price)
  }

  def printDetails(c: Coffee) {
    println(c.name)
    println("Price: " + c.price)
    println("Supplier: " + c.supplier.name)
}
```



O/R-Mapper

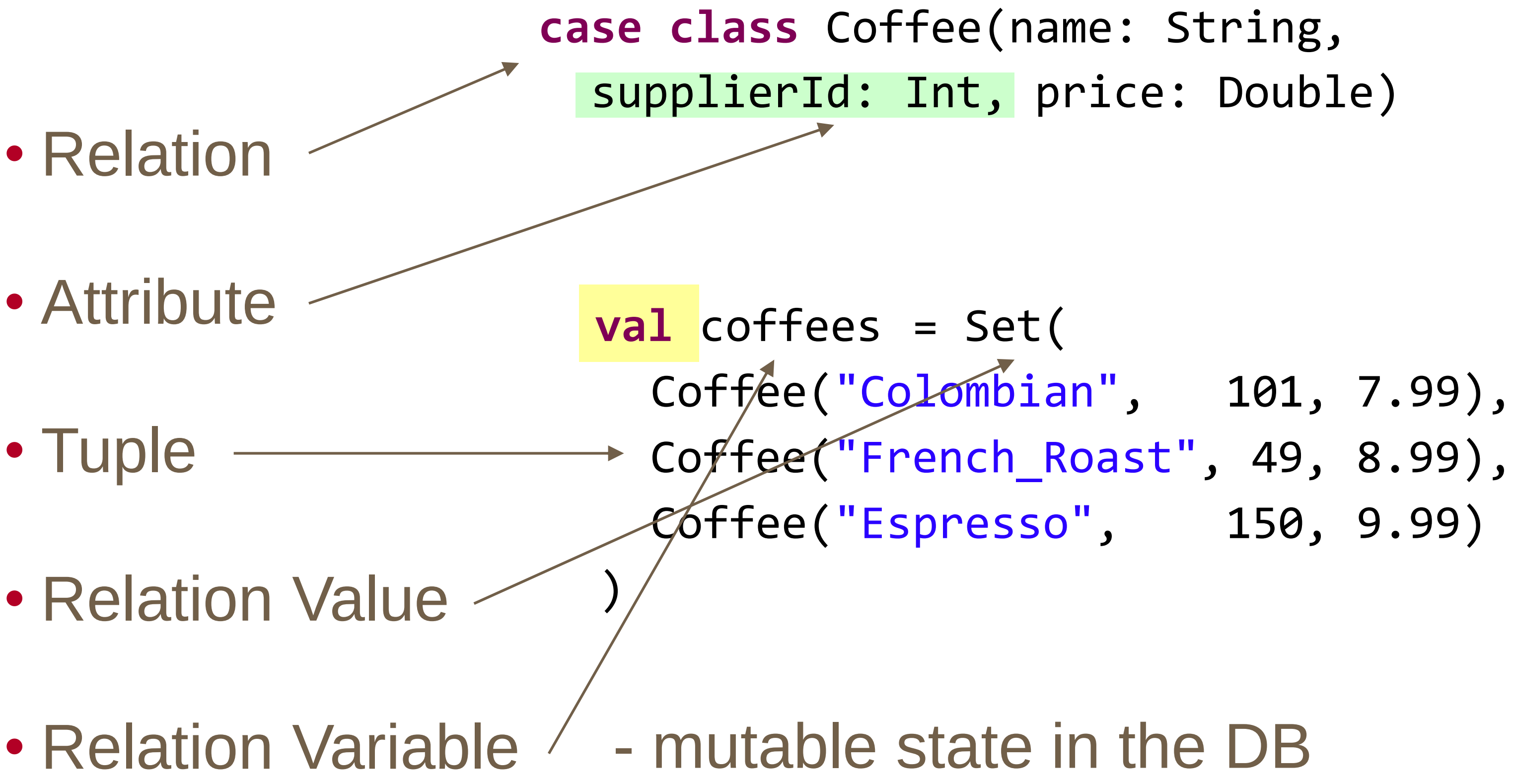
- Falsche Abstraktionsebene
- Nicht transparent

“Object/Relational Mapping is
The Vietnam of Computer Science”

(Ted Neward)

http://blogs.tedneward.com/2006/06/26/The+Vietnam+Of+Computer+Science.aspx

A Better Match: Functional Programming



```
case class Coffee(name: String,
  supplierId: Int, price: Double)

val coffees = Set(
    Coffee("Colombian",    101, 7.99),
    Coffee("French_Roast", 49, 8.99),
    Coffee("Espresso",     150, 9.99)
)
```

- Relation
- Attribute
- Tuple
- Relation Value
- Relation Variable - mutable state in the DB

- Session-Management `org.scalaquery.session`

- Gemeinsames API zur Ausführung beider Arten von Statements `org.scalaquery`

Session Management: Database

- JDBC kennt zwei Connection-Management-Modelle: DriverManager und DataSource
- Wie mit DriverManager Connections zu einer URL öffnen: `Database.forURL(…)`
- Ein DataSource-Objekt verwenden: `Database.forDataSource(…)`
- Ein DataSource-Objekt über einen JNDI-Namen holen: `Database.forName(…)`

Session Management: Session

- Alle Zugriffe auf die Datenbank erfolgen über ein `Session`-Objekt
- Wrapper für `java.sql.Connection`
- Oft als implizites Objekt verwendet: `Database.threadLocalSession`
- Kein Caching von Connections und PreparedStatements

Session Management

```
import org.scalaquery.session._
import org.scalaquery.session.Database.threadLocalSession

val db = Database.forURL("jdbc:h2:mem:test1",
  driver = "org.h2.Driver")

db withTransaction {

    doSomethingWithSession

}
```

Typsichere Queries: Scala-Collections

```
case class Coffee(
  name: String,
  supID: Int,
  price: Double
)


val coffees = List(
  Coffee("Colombian",         101, 7.99),
  Coffee("Colombian_Decaf",   101, 8.99),
  Coffee("French_Roast_Decaf", 49, 9.99)
)

val l = for {
  c <- coffees if c.supID == 101
} yield (c.name, c.price)

l.foreach { case (n, p) => println(n + ": " + p) }
```

Scala Collections

Typsichere Queries: Query Language

```
val Coffees = new Table[(String, Int, Double)]("COFFEES") {
  def name = column[String]("COF_NAME", O.PrimaryKey)
  def supID = column[Int]("SUP_ID")
  def price = column[Double]("PRICE")
  def * = name ~ supID ~ price
}

Coffees.insertAll(
  ("Colombian",         101, 7.99),
  ("Colombian_Decaf",   101, 8.99),
  ("French_Roast_Decaf", 49, 9.99)
)

val q = for {
  c <- Coffees if c.supID === 101
} yield c.name ~ c.price

q.foreach { case (n, p) => println(n + ": " + p) }
```

ScalaQuery

Tabellendefinitionen

```
val Suppliers = new Table[(Int, String, String,
    String, String, String)]("SUPPLIERS") {

   def id     = column[Int   ]("SUP_ID", O.PrimaryKey)
   def name   = column[String]("SUP_NAME")
   def street = column[String]("STREET")
   def city   = column[String]("CITY")
   def state  = column[String]("STATE")
   def zip    = column[String]("ZIP")

   def * = id ~ name ~ street ~ city ~ state ~ zip

   def nameConstraint = index("SUP_NAME_IDX", name, true)
}
```

Tabellendefinitionen

```
val Coffees = new Table[(String, Int, Double,
    Int, Int)]("COFFEES") {

  def name  = column[String]("COF_NAME")
  def supID = column[Int   ]("SUP_ID")
  def price = column[Double]("PRICE")
  def sales = column[Int   ]("SALES")
  def total = column[Int   ]("TOTAL")

  def * = name ~ supID ~ price ~ sales ~ total

  def supplier = foreignKey("SUP_FK", supID, Suppliers)(_.id)

  def pk = primaryKey("COF_NAME_PK", name)
}
```

Tabellen Erzeugen

```
val db = Database.forURL("jdbc:h2:mem:test1",
  driver = "org.h2.Driver")

val Suppliers = …
val Coffees = …

db withSession {

  (Suppliers.ddl ++ Coffees.ddl).create

}
```

Query Language Imports

```
import org.scalaquery.ql._
```

```
import org.scalaquery.ql.TypeMapper._
```

```
import org.scalaquery.ql.extended.H2Driver.Implicit._
impo                                 ended.{ExtendedTable => Table}
```

- basic.BasicDriver
- extended.AccessDriver
- extended.DerbyDriver
- extended.H2Driver
- extended.HsqldbDriver
- extended.MySQLDriver
- extended.PostgresDriver
- extended.SQLiteDriver
- extended.SQLServerDriver

```
pper](n: String,
on[C, ProfileType]*) = …
```

Ein DAO-Pattern

```
class DAO(driver: ExtendedProfile, db: Database) {
  import driver.Implicit._

  val Props = new Table[(String, String)]("properties") {
    def key = column[String]("key", O.PrimaryKey)
    def value = column[String]("value")
    def * = key ~ value
  }

  def insert(k: String, v: String) = db withSession
    Props.insert(k, v)

  def get(k: String) = db withSession
    ( for(p <- Props if p.key === k)
        yield p.value ).firstOption
}
```

Inner Joins & Abstraktionen

```
for {
  c <- Coffees if c.price < 9.0
  s <- Suppliers if s.id === c.supID
} yield (c.name, s.name)
```

ScalaQuery

```
for {
  c <- Coffees.cheaperThan(9.0)
  s <- c.supplier
} yield c.name ~ s.name
```

```
val Coffees = new Table … {
  def supplier = Suppliers.where(_.id === supID)
  def cheaperThan(d: Double) = this.where(_.price < d)
}
```

Datentypen

- Basistypen

`Byte, Int, Long`	`0`
`String`	`""`
`Boolean`	`false`
`Date, Time, Timestamp`	`1970-1-1 00:00:00`
`Float, Double`	`0.0`
`Blob, Clob, Array[Byte]`	`null, null, []`
`Option[`*`T`*`]` für alle Basistypen T	`None`

- Datenbank-`NULL` wird auf Default-Wert gemappt

NULL

- Three-Valued Logic (3VL) in SQL

```
a ⊕ b → NULL
wenn a = NULL oder b = NULL
```

- Gilt auch für „="

```
a = NULL     →  NULL
NULL = a     →  NULL
a IS NULL    →  TRUE oder FALSE
```

NULL

- In ScalaQuery über `OptionMapper` abgebildet
- Für Basistypen A, B, C:

```
Column[       A ] ⊕ Column[       B ] → Column[       C ]
Column[Option[A]] ⊕ Column[       B ] → Column[Option[C]]
Column[       A ] ⊕ Column[Option[B]] → Column[Option[C]]
Column[Option[A]] ⊕ Column[Option[B]] → Column[Option[C]]
```

Eigene Datentypen Verwenden

```
object Values extends Enumeration {
  val a, b, c = Value
}

implicit val valuesTypeMapper =
  MappedTypeMapper.base[Values.Value, Int](_.id, Values(_))

val MyTable = new Table[Values.Value]("MYTABLE") {
  def a = column[Values.Value]("A")
  def * = a
}

MyTable.ddl.create
MyTable.insertAll(Values.a, Values.c)

val q = MyTable.map(t => t.a ~ t.a.asColumnOf[Int])
q.foreach(println)
```

```
(a,0)
(c,2)
```

Aggregieren und Sortieren

```
val q = for {
  c <- Coffees
  s <- c.supplier
  _ <- Query groupBy s.id
  _ <- Query orderBy c.name.count
} yield s.id ~ s.name.min.get ~ c.name.count
```

- Aggregierungsmethoden: `.min`, `.max`, `.avg`, `.sum`, `.count`

Operatoren Für Columns

- **Allgemein:** .in(Query), .notIn(Query), .count, .countDistinct, .isNull, .isNotNull, .asColumnOf, .asColumnOfType
- **Vergleiche:** === (.is), =!= (.isNot), <, <=, >, >=, .inSet, .inSetBind, .between, .ifNull
- **Numerisch:** +, -, *, /, %, .abs, .ceil, .floor, .sign, .toDegrees, .toRadians
- **Boolean:** &&, ||, .unary_!
- **String:** .length, .like, ++, .startsWith, .endsWith, .toUpperCase, .toLowerCase, .ltrim, .rtrim, .trim

Invokers

- Alle Datenbankzugriffe erfolgen über `Invoker`
- Eine implizite Konvertierung von `Query` nach `Invoker` erlaubt das direkte Ausführen von Queries

Invoker-Methoden: Strict

- **`.to[C]()`** – erzeugt eine Collection `C` aller Ergebnisse

z.B. `myQuery.to[List]()`
`myQuery.to[Array]()`

- **`.list`** – Shortcut für `.to[List]()`
- **`.toMap`** – erzeugt eine `Map[K,V]` für einen `Query[(K,V)]`
- **`.first`**, **`.firstOption`**, **`.firstFlatten`** – geben das erste Ergebnis zurück

Invoker-Methoden: Lazy / Incremental

- **`.elements`** – erzeugt `CloseableIterator`, der alle Ergebnisse bei Bedarf liest
  - **`.elementsTo`** – nur bis zur angegebenen Maximalanzahl
- **`.foreach`** – führt die angegebene Funktion für jedes Ergebnis aus
  - Optional mit Maximalanzahl

```
for(r <- myQuery) ...
```

- **`.foldLeft`** – berechnet einen Wert aus allen Ergebnissen
- **`.execute`** – führt das Statement aus

Debugging

```
val q = for {
  c <- Coffees if c.supID === 101
} yield c.name ~ c.price
```

```
q.dump("q: ")
```

```
q: Query
 select: Projection2
  0: NamedColumn COF_NAME
   table: <t1> AbstractTable.Alias
    0: <t2> Table COFFEES
  1: NamedColumn PRICE
   table: <t1> ...
 where: Is(NamedColumn SUP_ID,ConstColumn[Int] 101)
  0: NamedColumn SUP_ID
   table: <t1> ...
  1: ConstColumn[Int] 101
```

```
SELECT "t1"."COF_NAME","t1"."PRICE"
FROM "COFFEES" "t1“
WHERE ("t1"."SUP_ID"=101)
```

```
println(q.selectStatement)
```

Explizite Inner Joins

name	supID
Colombian	101
Espresso	150
Colombian_Decaf	42

Coffees

id	name
101	Acme, Inc.
49	Superior Coffee
150	The High Ground

Suppliers

```
for (
  Join(c, s) <- Coffees innerJoin Suppliers
                        on (_.supID === _.id)
) yield c.name ~ s.name
```

```
(Colombian,Acme, Inc.)
(Espresso,The High Ground)
```

Left Outer Joins

name	supID
Colombian	101
Espresso	150
Colombian_Decaf	42

Coffees

id	name
101	Acme, Inc.
49	Superior Coffee
150	The High Ground

Suppliers

```
for (
  Join(c, s) <- Coffees leftJoin Suppliers
                        on (_.supID === _.id)
) yield c.name.? ~ s.name.?
```

```
(Some(Colombian),Some(Acme, Inc.))
(Some(Espresso),Some(The High Ground))
(Some(Colombian_Decaf),None)
```

Right Outer Joins

name	supID
Colombian	101
Espresso	150
Colombian_Decaf	42

Coffees

id	name
101	Acme, Inc.
49	Superior Coffee
150	The High Ground

Suppliers

```
for (
  Join(c, s) <- Coffees rightJoin Suppliers
                          on (_.supID === _.id)
) yield c.name.? ~ s.name.?
```

```
(Some(Colombian),Some(Acme, Inc.))
(None,Some(Superior Coffee))
(Some(Espresso),Some(The High Ground))
```

Full Outer Joins

name	supID
Colombian	101
Espresso	150
Colombian_Decaf	42

Coffees

id	name
101	Acme, Inc.
49	Superior Coffee
150	The High Ground

Suppliers

```
for (
  Join(c, s) <- Coffees outerJoin Suppliers
                        on (_.supID === _.id)
) yield c.name.? ~ s.name.?
```

```
(Some(Colombian),Some(Acme, Inc.))
(None,Some(Superior Coffee))
(Some(Espresso),Some(The High Ground))
(Some(Colombian_Decaf),None)
```

Case

```
for {
  c <- Coffees
} yield (Case when c.price < 8.0 then "cheap"
              when c.price < 9.0 then "medium"
              otherwise "expensive") ~ c.name
```

- If-then-else für Queries
- Rückgabetyp wird automatisch zu `Option`, wenn `otherwise` fehlt

Sub-Queries

```
for {
  c <- Coffees
  s <- c.supplier
  val lowestPriceForSupplier = (for {
    c2 <- Coffees
    s2 <- c2.supplier if s2.id === s.id
  } yield c2.price.min).asColumn
  _ <- Query if c.price === lowestPriceForSupplier
  _ <- Query orderBy s.id
} yield s.name ~ c.price
```

- Auch in **yield** verwendbar
- Direkt (ohne `.asColumn`) mit `.in` und `.notIn`
- `.exists`, `.count`

Unions

Scala Collections

```
val l1 = coffees.filter(_.supID == 101)
val l2 = coffees.filter(_.supID == 150)
val l3 = l1 ++ l2
```

ScalaQuery

```
val q1 = Coffees.filter(_.supID === 101)
val q2 = Coffees.filter(_.supID === 150)
val q3 = q1 unionAll q2
```

Paginierung

```
val l = for {
  c <- coffees if …
} yield …
val l2 = l.drop(20).take(10)
```

Scala Collections

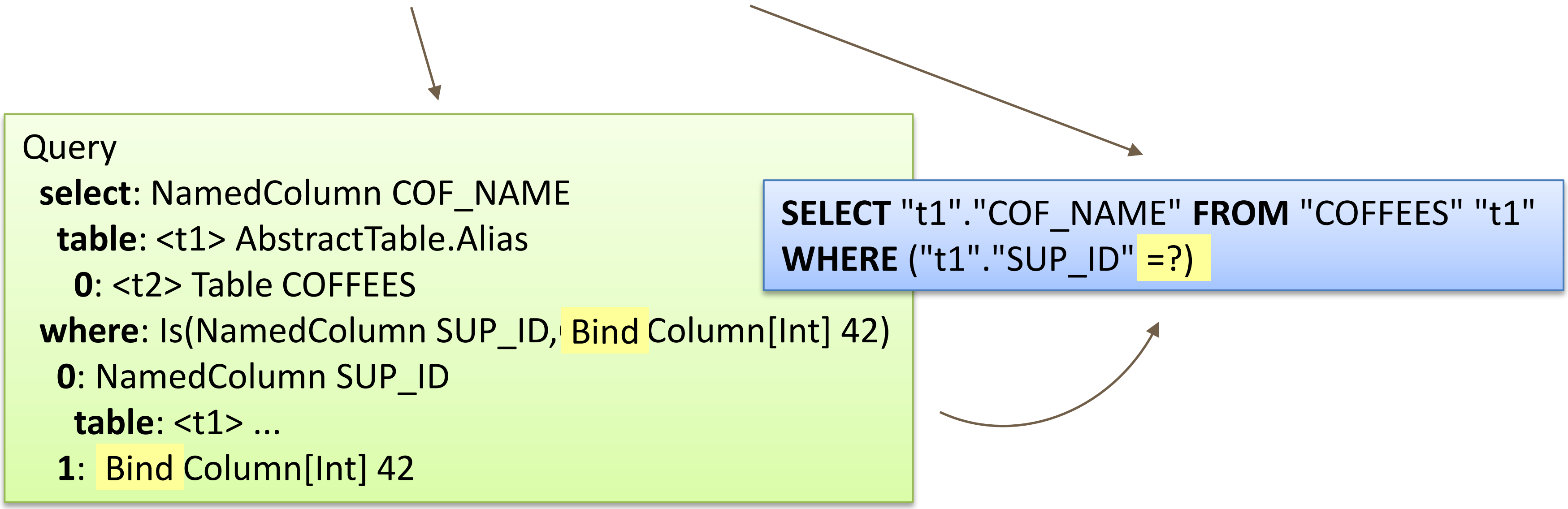
```
val q = for {
  c <- Coffees if …
  _ <- Query orderBy c.name
} yield …
val q2 = q.drop(20).take(10)
```

ScalaQuery

Bind-Variablen

```
def coffeesForSupplier(supID: Int) = for {
  c <- Coffees if c.supID === supID.bind
} yield c.name
```

```
coffeesForSupplier(42).list
```



```
Query
  select: NamedColumn COF_NAME
    table: <t1> AbstractTable.Alias
      0: <t2> Table COFFEES
  where: Is(NamedColumn SUP_ID, BindColumn[Int] 42)
    0: NamedColumn SUP_ID
      table: <t1> ...
    1: BindColumn[Int] 42
```

```
SELECT "t1"."COF_NAME" FROM "COFFEES" "t1"
WHERE ("t1"."SUP_ID" =?)
```

Query-Templates

```
val coffeesForSupplier = for {
  supID <- Parameters[Int]
  c <- Coffees if c.supID === supID
} yield c.name
```

```
coffeesForSupplier(42).list
```

```
Query
  select: NamedColumn COF_NAME
    table: <t1> AbstractTable.Alias
      0: <t2> Table COFFEES
  where: Is(NamedColumn SUP_ID,ParameterColumn[Int])
    0: NamedColumn SUP_ID
      table: <t1> ...
    1: ParameterColumn[Int]
```

```
SELECT "t1"."COF_NAME" FROM "COFFEES" "t1"
WHERE ("t1"."SUP_ID"=?)
```

Mapped Entities

```
case class Coffee(name: String, supID: Int, price: Double)

val Coffees = new Table[(String,Int, Double)]("COFFEES") {
  def name = column[String]("COF_NAME", O.PrimaryKey)
  def supID = column[Int]("SUP_ID")
  def price = column[Double]("PRICE")
  def * = name ~ supID ~ price <> (Coffee, Coffee.unapply _)
}

Coffees.insertAll(
  Coffee("Colombian",    101, 7.99),
  Coffee("French_Roast",  49, 8.99)
)

val q = for(c <- Coffees if c.supID === 101) yield c
q.foreach(println)
```

```
Coffee(Colombian,101,7.99)
```

Insert, Delete, Update

```
class Coffees(n: String)
    extends Table[(String, Int, Double)](n) {
  def name =  column[String]("COF_NAME")
  def supID = column[Int]("SUP_ID")
  def price = column[Double]("PRICE")
  def * = name ~ supID ~ price
}

val Coffees1 = new Coffees("COFFEES_1")
val Coffees2 = new Coffees("COFFEES_2")

(Coffees1

Coffees1.insertAll(
  ("Colombian",         101, 7.99),
  ("French_Roast",       49, 8.99),
  ("Espresso",          150, 9.99)
)
println(Coffees1.insertStatement)
```

INSERT INTO "COFFEES1" ("COF_NAME","SUP_ID","PRICE") **VALUES** (?,?,?)

Insert, Delete, Update

```
val q = Coffees1.where(_.supID === 101)

Coffees2.insert(q)

println(Coffees2.insertStatementFor(q))
```

INSERT INTO "COFFEES2" ("COF_NAME","SUP_ID","PRICE")
SELECT "t1"."COF_NAME","t1"."SUP_ID","t1"."PRICE" **FROM** "COFFEES1" "t1"
WHERE ("t1"."SUP_ID"=101)

```
q.delete

println(q.deleteStatement)
```

DELETE FROM "COFFEES1" **WHERE** ("COFFEES1"."SUP_ID"=101)

Insert, Delete, Update

```
val q2 = q.map(_.supID)

q2.update(49)

println(q2.updateStatement)
```

```
UPDATE "COFFEES1" SET "SUP_ID"=? WHERE ("COFFEES1"."SUP_ID"=101)
```

Static Queries

```
import org.scalaquery.simple._
import org.scalaquery.simple.StaticQuery._

def allCoffees = queryNA[String](
  "select cof_name from coffees").list

def supplierNameForCoffee(name: String) =
  query[String, String]("""
    select s.sup_name from suppliers s, coffees c
    where c.cof_name = ? and c.sup_id = s.sup_id
  """).firstOption(name)

def coffeesInPriceRange(min: Double, max: Double) =
  query[(Double, Double), (String, Int, Double)]("""
    select cof_name, sup_id, price from coffees
    where price >= ? and price <= ?
  """).list(min, max)
```

Static Queries

```
import org.scalaquery.simple._
import org.scalaquery.simple.StaticQuery._

case class Coffee(
  name: String, supID: Int, price: Double)

implicit val getCoffeeResult =
  GetResult(r => Coffee(r<<, r<<, r<<))
```

[P : SetParameter,

R : GetResult]

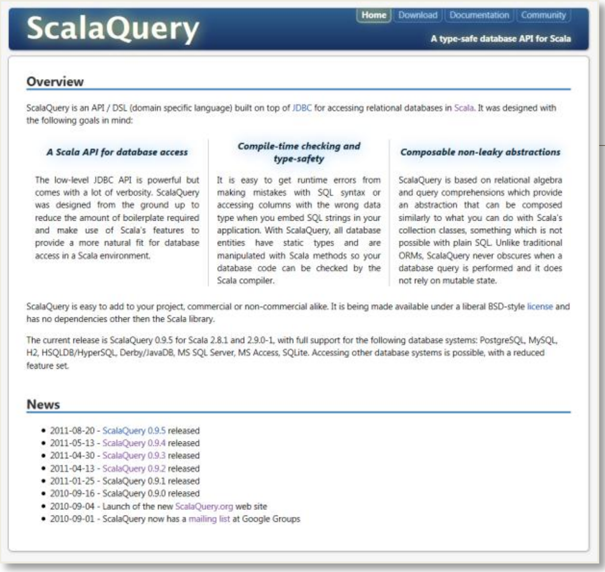
```
def coffeesInPriceRange(min: Double, max: Double) =
  query[(Double, Double),           Coffee           ]("""
    select cof_name, sup_id, price from coffees
    where price >= ? and price <= ?
  """).list(min, max)
```

Weitere Features

• Mutating Queries	`MutatingInvoker.mutate`
• JDBC-Metadaten	`org.scalaquery.meta`
• Iteratees	`org.scalaquery.iter`
• Sequences	
• Dynamic Queries	`org.scalaquery.simple`

Getting Started

- http://scalaquery.org



ScalaQuery

A type-safe database API for Scala

Overview

ScalaQuery is an API / DSL (domain specific language) built on top of JDBC for accessing relational databases in Scala. It was designed with the following goals in mind:

A Scala API for database access

The low-level JDBC API is powerful but comes with a lot of verbosity. ScalaQuery was designed from the ground up to reduce the amount of boilerplate required and make use of Scala's features to provide a more natural fit for database access in a Scala environment.

Compile-time checking and type-safety

It is easy to get runtime errors from making mistakes with SQL syntax or accessing columns with the wrong data type when you embed SQL strings in your application. With ScalaQuery, all database entities have static types and are manipulated with Scala methods so your database code can be checked by the Scala compiler.

Composable non-leaky abstractions

ScalaQuery is based on relational algebra and query comprehensions which provide an abstraction that can be composed similarly to what you can do with Scala's collection classes, something which is not possible with plain SQL. Unlike traditional ORMs, ScalaQuery never obscures when a database query is performed and it does not rely on mutable state.

ScalaQuery is easy to add to your project, commercial or non-commercial alike. It is being made available under a liberal BSD-style license and has no dependencies other then the Scala library.

The current release is ScalaQuery 0.9.5 for Scala 2.8.1 and 2.9.0-1, with full support for the following database systems: PostgreSQL, MySQL, H2, HSQLDB/HyperSQL, Derby/JavaDB, MS SQL Server, MS Access, SQLite. Accessing other database systems is possible, with a reduced feature set.

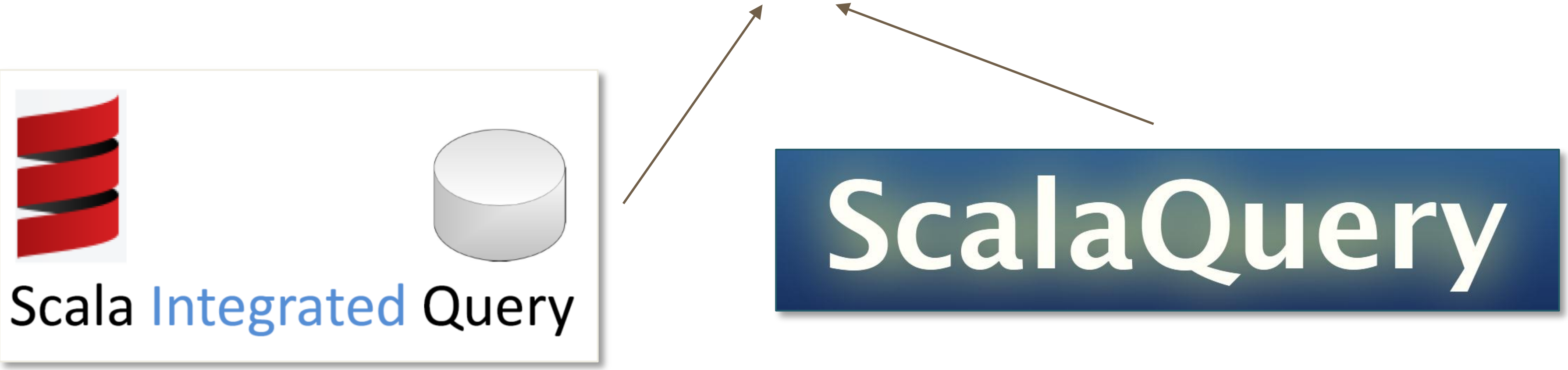
News

- 2011-08-20 - ScalaQuery 0.9.5 released
- 2011-05-13 - ScalaQuery 0.9.4 released
- 2011-04-30 - ScalaQuery 0.9.3 released
- 2011-04-13 - ScalaQuery 0.9.2 released
- 2011-01-25 - ScalaQuery 0.9.1 released
- 2010-09-16 - ScalaQuery 0.9.0 released
- 2010-09-04 - Launch of the new ScalaQuery.org web site
- 2010-09-01 - ScalaQuery now has a mailing list at Google Groups

- https://github.com/szeiger/scalaquery-examples
- https://github.com/szeiger/scala-query/tree/master/src/test/scala/org/scalaquery/test

Ausblick

- Typesafe
- **Slick** – A common framework for connecting with databases and distributed collections



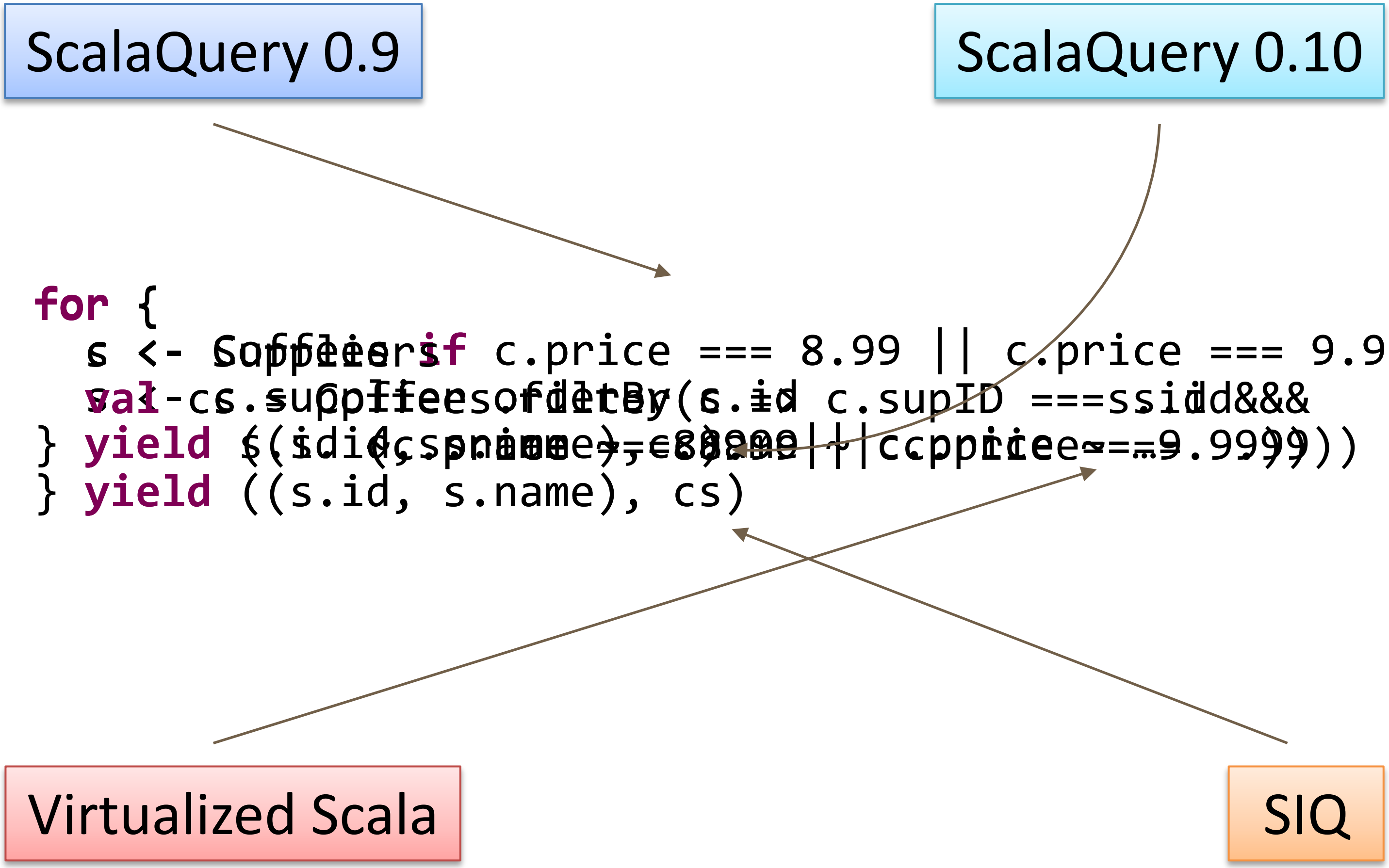
Scala Integrated Query

ScalaQuery

by Christopher Vogt

http://code.google.com/p/scala-integrated-query/

Ausblick



```
for {
  s <- Suppliers
  c <- Coffees if c.price === 8.99 || c.price === 9.99
  val cs = Coffees.filter(c => c.supID === s.id &&
    (c.price === 8.99 || c.price === 9.99))
} yield ((s.id, s.name), cs)
```

Vielen Dank!

Stefan Zeiger
Commerzbank AG

http://szeiger.de
Twitter: @StefanZeiger