JdbcTemplate

JdbcTemplate

- Die Klasse org.springframework.jdbc.core.JdbcTemplate ist zentral im JDBC-Paket von Spring.
- Die Klasse
  - führt einfach SQL-Anweisungen aus
  - liest Ergebnisse ein und transformiert sie in Objekte
  - ummantelt SQLException in DataAccessException.

- Ein Beispiel, wie man die Klasse prima für Tests nutzen kann:

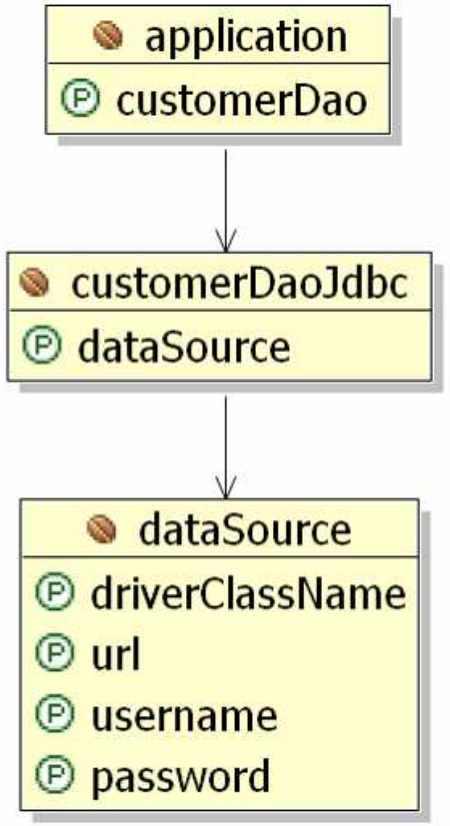
```
JdbcTemplate jt = new JdbcTemplate(ds);

jt.execute("delete from mytable");

jt.execute("insert into mytable (id, name) values(1, 'John')");

jt.execute("insert into mytable (id, name) values(2, 'Jane')");
```

Wie das JdbcTemplate nutzen?

- Für einen JDBC-DAO gibt es zwei Möglichkeiten, wie man mit dem JdbcTemplate weitermacht:
  - Die DAO-Implementierung lässt sich – etwa im Konstruktor – die DataSource übergeben und baut dann selbst das JdbcTemplate auf. Den Verweise auf das JdbcTemplate merkt man sich in einer Objekt-Variablen und bietet den anderen Methoden Zugriff darauf.
  - Die eigene DAO-Klasse erbt von JdbcTemplate. Man injiziert auch wieder DataSource, nur nimmt das schon setDataSource().
- Vorteil der Vererbungs-Lösung: Etwas kürzer. Ein Nachteil: Mit der Einfachvererbung unter Umständen eine Einschränkung.

JdbcTemplate als Basisklasse

- Mit dem JdbcTemplate als Basisklasse lässt sich der CustomerDaoJdbcImpl aufbauen.
  - Er empfängt die DataSource.
- Die Applikation wiederum bekommt diesen CustomerDaoJdbcImpl als einen CustomerDao geliefert.



JdbcTemplate: execute() und update()

- JdbcTemplate ist eine zustandslose und Thread-sichere Klasse, mit der im Allgemeinen jeder DAO verbunden ist.
- Zum Schicken einer SQL-Anweisung dient:
  - void execute(String sql)

  Die Anweisung ist oft eine DDL, etwa CREATE TABLE.
- Für SQL UPDATE-Anweisungen dienen die Methoden
  - int update(String sql)
  - int update(String sql, Object[] args)
  - int update(String sql, Object[] args, int[] argTypes)

  Die Rückgabe ist (wie üblich) die Anzahl modifizierter Sätze.

JdbcTemplate: queryForXXX() – 1

- Zum Anfragen definiert JdbcTemplate
  - int queryForInt(String sql)
  - long queryForLong(String sql)
  - Object queryForObject(String sql, Class requiredType)

- Eine TypeMismatchDataAccessException zeigt ein Konvertierungsfehler an.

```
JdbcTemplate jt = new JdbcTemplate( ds );
int nr = jt.queryForInt("SELECT COUNT(*) FROM Customer");
```

JdbcTemplate: queryForXXX() – 2

- Die Methode
  - List queryForList(String sql)

liefert eine Liste mit Map-Exemplare. Der Schlüssel der Map ist der Spaltenname.

Platzhalter definieren

- Die genannten Methoden führen ein SQL-Statement aus, aber kein PreparedStatement.
  - Das ist aber wichtig, wenn veränderbare Parameter eingesetzt werden.
- Die query()- und auch die update()-Methoden sind daher überladen, um ein Objektfeld anzunehmen, was die Parameter definiert.

```
JdbcTemplate jt = new JdbcTemplate( ds );
Object[] param = { "Chris" };
int count = jt.queryForInt(
 "SELECT COUNT(*) FROM Customer WHERE firstname LIKE ?",
 param );
```

RowMapper

- Die Rückgabe als Feld ist aus objektorientierter Sicht selten befriedigend.
  - Man möchte oft ein die Spalten auf ein Objekt mappen.
- Das Mapping von einem java.sql.ResultSet auf ein Objekt definiert eine org.springframework.jdbc.core.RowMapper.
- Die Schnittstelle RowMapper schreibt eine Methode vor:
  - Object mapRow(ResultSet rs, int rowNum)

  Die Rückgabe kann durch einen konvarianten Rückgabetyp in Java 5 natürlich auch etwas konkretes sein.

CustomerRowMapper

```
class CustomerRowMapper implements RowMapper
{
  public Customer mapRow( ResultSet rs, int rowNum )
                                        throws SQLException
  {
    Customer c = new Customer();
    c.setId( rs.getInt( "Id" ) );
    c.setName( rs.getString( "Name" ) );
    return c;
  }
}
```

getCustomers() aus dem DAO

- Aus der Schnittstelle CustomerDao implementieren wir getCustomers() wie folgt:

```
@SuppressWarnings("unchecked")
public Collection<Customer> getCustomers()
{
  return query( "SELECT Id, Name FROM Customers",
                new CustomerRowMapper() );
}
```

- Das @SuppressWarnings("unchecked") ist nötig, da query() nur eine untypisierte Collection liefert.

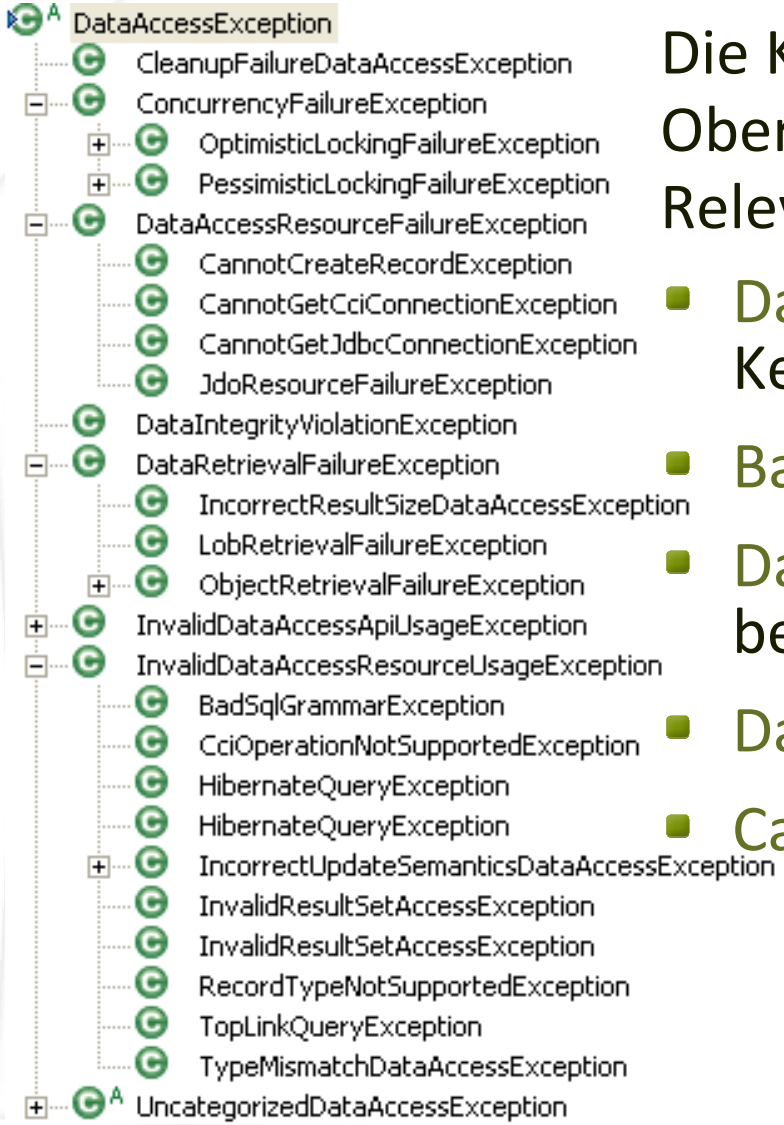
findCustomerById() aus dem DAO

```
public Customer findCustomerById( int id )
{
  String sql = "SELECT Id, Name FROM Customers WHERE Id=?";

  try
  {
    return (Customer) queryForObject( sql,
                                      new Object[] { id },
                                      new CustomerRowMapper() );
  }
  catch ( IncorrectResultSizeDataAccessException e ) { }

  return null;
}
```

save() aus dem DAO

```
public void save( Customer customer )
{
 if ( findCustomerById( customer.getId() ) == null )
 {
  Object[] args = { customer.getId(), customer.getName() };
  update("INSERT INTO Customers (Id, Name) VALUES (?,?)", args);
 }
 else
 {
  Object[] args = { customer.getName(), customer.getId() };
  update("UPDATE Customers SET Name = ? WHERE Id = ?", args);
 }
}
```

DataAccessException und Kinder



Die Klasse DataAccessException ist die Oberklasse für diverse Datenbank-Fehler. Relevant sind:

- DataAccessResourceFailureException. Kein Connect zur DB
- BadSqlGrammarException
- DataIntegrityViolationException. Etwa bei gleichen PK
- DataRetrievalFailureException
- CannotAcquireLockException

SimpleJdbcTemplate

SimpleJdbcTemplate

- Das org.springframework.jdbc.core.JdbcTemplate compiliert unter Java 1.4 und nutzt keine Eigenschaften von Java 5 wie Generics, Autoboxing und Varags.
- org.springframework.jdbc.core.simple.SimpleJdbcTemplate ist eine neue Klasse seit Spring 2.0 mit einer überschaubaren Anzahl Methoden für Java 5.
  - Es ist eine Java 5-Fassade vor dem JdbcTemplate.
  - Der Konstruktor von SimpleJdbcTemplate nimmt wie JdbcTemplate eine DataSource an.
  - SimpleJdbcTemplate ist eine Implementierung von SimpleJdbcOperations.

Methoden vom SimpleJdbcTemplate

- **int queryForInt(String sql, Object... args)**
- **long queryForLong(String sql, Object... args)**

  Erfragt ein int/long über einen SQL-String und optionalen Argumenten.
- **List<Map<String,Object>> queryForList(String sql, Object... args)**
- **Map<String,Object> queryForMap(String sql, Object... args)**

  Führt die SQL-Anfrage mit optionalen Argumenten aus.
- **<T> T queryForObject(String sql, ParameterizedRowMapper<T> rm, Object... args)**

  Die SQL-Anfrage liefert ein Ergebnis, das der Mapper auf ein Objekt überträgt.
- **<T>T queryForObject(String sql, Class<T> requiredType, Object... args)**

  Führt eine SQL-Anfrage aus und überträgt die Eigenschaften auf ein Objekt.
- **<T> List<T> query(String sql, ParameterizedRowMapper<T> rm, Object... args)**

  Der ParameterizedRowMapper überträgt Ergebnisse auf Objekte.
- **int update(String sql, Object... args)**

  Führt ein SQL-Update mit optionalen Argumenten durch.

SimpleJdbcTemplate und JdbcTemplate

- Neben dem Konstruktor SimpleJdbcTemplate(DataSource) gibt es einen zweiten:
  - SimpleJdbcTemplate(JdbcOperations classicJdbcTemplate)
- Als Übergabe ist ein Objekt vom Typ JdbcOperations nötig; JdbcTemplate implementiert diese Schnittstelle.
- Das SimpleJdbcTemplate hat 7 queryXXX()-Funktionen und eine update()-Funktion und bietet damit das, was man in der Praxis am häufigsten benötigt.
- Vom SimpleJdbcTemplate kann man auf die größere Schnittstelle JdbcOperations kommen.
  - Damit kann man etwa Batch-Updates ausführen.
  - JdbcOperations getJdbcOperations() liefert das Objekt.

SimpleJdbcDaoSupport

- Die DAO-Klassen von Spring erlauben Zugriff auf die Template-Objekte.
- Während JdbcDaoSupport ein JdbcTemplate liefert, gibt SimpleJdbcDaoSupport ein SimpleJdbcTemplate.
- SimpleJdbcDaoSupport ist eine Unterklasse von JdbcDaoSupport.

org.springframework.jdbc.core.support.JdbcDaoSupport
org.springframework.jdbc.core.simple.SimpleJdbcDaoSupport

- SimpleJdbcDaoSupport fügt eine öffentliche Methode hinzu:

SimpleJdbcTemplate getSimpleJdbcTemplate()

Transaktionen

DummyDao

```
public class DummyDao extends JdbcTemplate {
 public void initDatabase() {
  execute( "CREATE TABLE Customers( Id INTEGER, Lastname VARCHAR(128) )" );
 }
 public void save() {
  update("INSERT INTO Customers (Id, Lastname) VALUES (?,?)",new Object[]{1,"Chris"});
  update("INSERT INTO Customers (Id, Lastname) VALUES (?,?)",new Object[]{2,"Tina"});
 }
 public int noCustomers() {
  return queryForInt( "SELECT COUNT(*) FROM Customers" );
 }
}
```

Zugehörige Bean-Deklaration

```
<bean id="dummyDaoTarget"
        class="com.tutego.spring.dao.jdbc.DummyDao"
        init-method="initDatabase">
    <property name="dataSource">
      <ref local="dataSource" />
    </property>
 </bean>
```

Testprogramm

- Ein Testprogramm gibt Anzahl Kunden aus, fügt dann die zwei Kunden ein und gibt wiederum die Anzahl Kunden aus.

```
DummyDao bean =
  (DummyDao) context.getBean( "dummyDaoTarget" );
System.out.println( bean.noCustomers() );
try { bean.save(); }
catch ( Exception e ) { e.printStackTrace(); }
System.out.println( bean.noCustomers() );
```

- Die erwartete Ausgabe:

```
0
2
```

Gedankenspiel

- Was passiert bei folgendem?

```
public void save() {
 update("INSERT INTO Customers (Id, Lastname) VALUES (?,?)",new Object[]{1,"Chris"});

if ( 0 == 0 )
 throw new RuntimeException( "Keine Lust mehr" );

 update("INSERT INTO Customers (Id, Lastname) VALUES (?,?)",new Object[]{2,"Tina"});
}
```

Ausgabe mit Exception

- Wie erwartet fügt das Programm einen Datensatz ein.

  0

  java.lang.RuntimeException: Keine Lust mehr

  at com.tutego.spring.dao.jdbc.DummyDao.save(DummyDao.java:18)

  at com.tutego.spring.dao.jdbc.ApplicationTx.main(ApplicationTx.java:18)

  1

- Das Ziel ist es, dass die save() Funktion entweder alle Operationen durchführt, oder gar keine.
  - Die gewünschte Ausgabe ist 0 und 0.

Deklarative Transaktionsdefinition

- Die Java EE macht es vor, dass der Application Server über deklarative Transaktionen die Steuerung übernimmt.
- Auch in Spring ist das nicht anders. Dabei bietet Spring zwei Möglichkeiten:
  - Einen Proxy nutzen und dort die Methoden mit ihren Transaktionsattributen definieren.
  - Java 5 Annotationen verwenden.
- Der allgemeine Trick ist, dem Benutzer nicht mehr zum Beispiel ein Original-DAO zu geben, sondern ein Proxy, der die Methodenaufrufe abfängt und die transaktionale Steuerung übernimmt.
- Auf jeden Fall ist ein PlatformTransactionManager, der die Steuerung übernimmt.

PlatformTransactionManager

- Ein PlatformTransactionManager ist eine zentrale Schnittstelle in Springs Transaktionssteuerung. Ihn gibt es in verschiedenen Ausführungen:
  - DataSourceTransactionManager. Nutzt Transkations-Manager einer gegebenen DataSource. Auch **lokale Transaktion** genannt.
  - JtaTransactionManager. Bezieht den Transaktionsmanager vom Java EE Container, der unter dem Standard-Namen „java:comp/UserTransaction“ eingetragen ist. Auch **globale Transaktion** genannt.
  - HibernateTransactionManager. Nutzt die Datenquelle, die bei der Hiberante-Session eingetragen ist.

DataSourceTransactionManager

- Für normale JDBC-Transaktionen schreiben wir:

```
<bean id="transactionManager"
   class="org.springframework.jdbc.datasource.DataSourceTransactionManager">
 <property name="dataSource" ref="dataSource"/>
</bean>
```

TransactionProxyFactoryBean

- Als nächstes müssen wir einen Proxy bestimmen, durch den die Aufrufe gehen und der die transaktionale Steuerung übernimmt.
- Das ist bei Spring eine TransactionProxyFactoryBean. Sie wird konfiguriert mit:
  - transactionManager. Wer tatsächlich die Steuerung übernimmt.
  - target. Die Original-Bean, um die diese Proxy sich legt.
  - transactionAttributes. Deklarive Transaktionsattribute für aufgezählte Methoden.
- Die Namensgebung empfiehlt, dass das Original mit „Target“ endet. Der Proxy bekommt oft den Namen, den vorher das Target hatte.

TransactionProxyFactoryBean nutzen

```
<bean id="txDummyDao"
  class="org.springframework.transaction.interceptor.TransactionProxyFactoryBean">
   <property name="transactionManager" ref="transactionManager"/>
   <property name="target" ref="dummyDaoTarget"/>
   <property name="proxyTargetClass" value="true"/>
   <property name="transactionAttributes">
     <props>
       <prop key="*">PROPAGATION_REQUIRED,-Exception</prop>
       <prop key="get*">PROPAGATION_SUPPORTS</prop>
     </props>
   </property>
 </bean>
```

Zu den Attributen bei der Transaction-Propagation:
http://static.springframework.org/spring/docs/2.5.x/reference/transaction.html#tx-propagation

Neue Ausgabe mit Proxy

0

java.lang.RuntimeException: Keine Lust mehr

at com.tutego.spring.dao.jdbc.DummyDao.save(DummyDao.java:17)

at com.tutego.spring.dao.jdbc.DummyDao$$FastClassByCGLIB$$b676da67.invoke(<generated>)

at net.sf.cglib.proxy.MethodProxy.invoke(MethodProxy.java:149)

at org.springframework.aop.framework.Cglib2AopProxy$CglibMethodInvocation.invokeJoinpoint(Cglib2AopProxy.java:700)

at org.springframework.aop.framework.ReflectiveMethodInvocation.proceed(ReflectiveMethodInvocation.java:149)

at org.springframework.transaction.interceptor.TransactionInterceptor.invoke(TransactionInterceptor.java:106)

at org.springframework.aop.framework.ReflectiveMethodInvocation.proceed(ReflectiveMethodInvocation.java:171)

at org.springframework.aop.framework.Cglib2AopProxy$DynamicAdvisedInterceptor.intercept(Cglib2AopProxy.java:635)

at com.tutego.spring.dao.jdbc.DummyDao$$EnhancerByCGLIB$$af9177ef.save(<generated>)

at com.tutego.spring.dao.jdbc.ApplicationTx.main(ApplicationTx.java:18)

0

Neuerungen in Spring 2

- Seit Spring 2 gibt es eine Alternative zur TransactionProxyFactoryBean.
- Mit XML-Tags aus dem Namensraum „tx“ werden transaktionale Aspekte formuliert.
  - *http://static.springframework.org/spring/docs/2.5.x/reference/transaction.html#tx-resource-synchronization*
  - *http://jroller.com/raible/entry/migrating_to_spring_2_0*

tutego™
YOUR TRAINING COMPANY

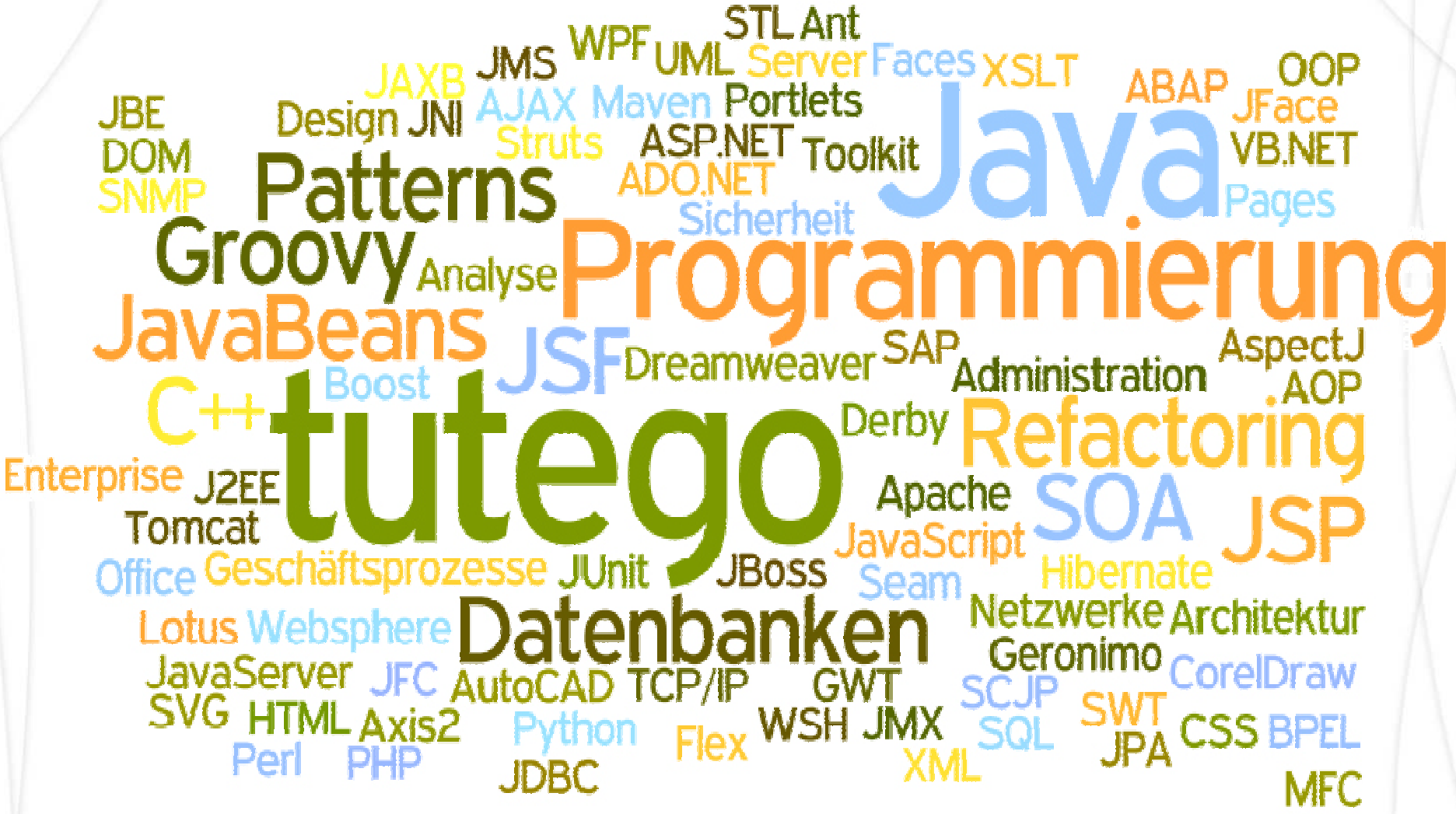
Professionelle IT-Qualifizierung

http://www.tutego.com/

tutego über tutego

- Inhouse-Schulungen mit individualisierten Inhalten und Terminauswahl
- 190 Seminare
- Kernkompetenz Java (60 Themen)
- Europaweit führende Position im Bereich Java-Schulungen
- Hochqualifizierte und zertifizierte Referenten
- Firmengründung 1997 durch Java Champion Christian Ullenboom

Unsere Themen



Unsere Themen