Datenobjekte mit automatischer Persistenz

Sven Eric Panitz

25. Mai 2021

Inhaltsverzeichnis

1. Datenhaltungsklassen

Ziel dieses Lehrbrief soll es sein, zu zeigen, wie man in Java Bibliotheken und komplexe Frameworks umsetzen kann. Hierzu werden wir zwei noch nicht verwendete Techniken anwenden:

- das Java Reflection API
- Annotationen

Ziel soll es sein, dass für Javaklassen automatisch Datenbanktabellen einer relationalen Datenbank generiert werden und Objekte dieser Klassen dann in der Datenbank persistiert werden und aus der Datenbank wieder selektiert werden können. Man bezeichnet dieses als objektrelationale Abbildung (›*object-relational mapping*‹, kurz ORM).

Die genannten Funktionalitäten existieren nicht in den Standardklassen des JDK. Es gibt hingeben Frameworks, die genau diese Funktionalität anbieten. In diesem Lehrbrief zeigen wir, wie solche Meta-Bibliotheken in Java entwickelt werden können.

Prinzipiell sind wir damit in Bereich, einer Meta-Programmierung. Es soll für vom Anwendungsprogrammierer geschriebenen Datenhaltungsklassen automatisch zusätzliche Funktionalität erzeugt werden, so dass diese nicht mehr manuell codiert werden muss. Ziel ist es also, vielen Code, der für bestimmte Klassen immer wieder schematisch zu programmieren ist, automatisch bereitzustellen. In unserem Fall, Code zur Persistierung von Objekten der Klasse. Andere typische Beispiele wären, Code zum Serialisieren und Deserialisieren der Objekte als XML oder Json. Die XML Serialisierung ist im Aufgabenteil zu lösen.

Solch ein Code wird im englischen als *›boiler plate code‹* bezeichnet. Eine Terminologie, die sich aus der Druckereitechnik entlehnt. Dort waren *›boiler plates‹* fertig gesetzte Textfragmente, die immer wieder benötigte gleichartige Floskeln enthielten.

In Java gibt es zwei verschiedene Ansätze, wie man ein Framework entwickeln kann, das für bestimmte Klassen Funktionalitäten automatisch bereit stellt.

- Reflection: Das Reflection-API ermöglicht es in der Laufzeit über Objekte Meta-Informationen zu erfragen. Dieses betrifft die komplette Typinformation des Objektes, inklusive der Felder und Methoden und deren Typisierung, alle Attribute, und so weiter. Zusätzlich ermöglicht das Reflection API dann auch dynamisch, Methoden auf dem Objekt aufzurufen, oder einen Konstruktor für einen Typ aufzurufen. D.h. es ist möglich, in der Laufzeit Objekte von Klassen zu erzeugen, die während der Kompilierung noch nicht bekannt waren.
- Annotation Processing im Compiler: die zweite Möglichkeit ist, sich im Compiler einzuhängen und weiteren Java Code bei der Compilierung des Programmes zu generieren.

Wie man sieht, dient Reflection dazu, in der Laufzeit dynamisch Code in Abhängigkeit von den Meta-Informationen eines Objektes auszuführen, während der Annotation Processor in der Compilezeit einfach weiteren Code erzeugt und compiliert.

In diesem Lehrbrief zeigen wir den Weg über das Reflection API. Dabei werden wir aber auch zeigen, wie man im Reflection API eigenene Annotationen auswertet. Diese Annotationen werden dann aber nicht vom Compiler ausgewertet.

Der Weg über einen Annotation Processor in der Compilezeit wird in einem anderen Lehrbrief gezeigt.

Beginnen wir mit den benötigten Importanweisungen. Wir benötigen Klassen der SQL-Programmierung und der Ein- und Ausgabe:

Java: DataObject

```
package name.panitz.util;

import java.io.*;
import java.util.*;
import static java.util.Map.entry;

import java.sql.*;
```

Für den Aufgabenteil dieses Lehrbriefes benötigen wir noch allerhand Klassen aus dem XML-Umfeld.

Java: DataObject

```
import org.w3c.dom.*;
import javax.xml.parsers.*;
import javax.xml.transform.*;
import javax.xml.transform.dom.*;
import javax.xml.transform.stream.*;
```

Wir definieren eine zentrale Schnittstelle. Record-Klassen, die diese Schnittstelle implementieren werden automatisch allen nötigen Code erhalten, um die Objekte in eine Datenbank zu speichern.

Wir nennen die Schnittstelle `DataObject`, weil es sich um hircharchische aber nicht zyklische Daten handeln soll, also um beliebige Baumstrukturen.

Java: DataObject

```
public interface DataObject {
```

1.1. UID

Wenn unser Daten in einer Datenbank persistiert werden, so sollen sie in der Datenbank eine eindeutige ID erhalten. Für diese ID halten wir eine Klasse `UID` bereit, die die ID als `long`-Zahl speichert. Solange ein Objekt noch nicht in der Datenbank gespeichert ist, hat es die ID `-1`.

java: UID

```java
package name.panitz.util;
public class UID {
  long uid = -1;
  public UID() {this(-1);}
  public UID(long uid){this.uid = uid;}
  public String toString() {return uid + "";}
  public boolean equals(Object o){
    return (o instanceof UID that)&&uid==that.uid;
  }
}
```

Jede Record-Klasse, die diese Schnittstelle implementiert soll eine Methode zum Erfragen der uid haben.

Java: DataObject

```java
UID uid();
```

Damit lässt sich für jedes Objekt eine Methode schreiben, mit der die uid neu gesetzt werden kann.

Java: DataObject

```java
default void setUID(long id) {
  uid().uid = id;
}
```

1.2. Beispielklassen

Bevor wir uns der eigentlichen Implementierung der gewünschten Funktionalitäten widmen, seien zwei Beispielklassen angegeben.

Zunächst eine Klasse, die Werte für eine literarische Epoche speichern kann. Eine Epoche habe einen Namen und ein Jahrhundert, die die Epoche zeitlich verordnet:

java: Epoche

```java
package name.panitz.util;
public record Epoche
     (UID uid, String name, int jahrhundert) implements DataObject {
```

Zusätzlich zu den zwei Feldern für den Namen und des Jahrhunderts müssen wir in einer Record-Klasse, die `DataObject` implementiert, auch ein Feld für die uid vorsehen. Die uid soll allerdings erst von der Datenbank beim Persistieren vergeben werden. Wenn wir Objekte der Record-Klasse im Hauptspeicher anlegen, soll die uid keine Rolle spielen.

Hierfür sehen wir einen zweiten Konstruktor vor, der die uid auf den Wert `-1` setzt.

java: Epoche

```
  public Epoche(String name, int jahrhundert) {
    this(new UID(), name, jahrhundert);
  }
}
```

Somit lassen sich auf einfache und natürliche Weise die Objekte dieser Record-Klasse erzeugen, ohne dass wir uns Gedanken um die uid machen müssen.

Shell

```
jshell> var g = new Epoche("Romantik",19)
g ==> Epoche[uid=-1, name=Romantik, jahrhundert=19]
```

Wir definieren eine zweite Klasse. Diese soll Schriftsteller speichern. Ein Schriftsteller habe einen Namen, einen Vornamen, ein Geburtsjahr und eine Epoche, für die der Autor steht.

java: Person

```
package name.panitz.util;
public record Person
    (UID uid, @VARCHAR(200) String name, String vorname
    , int gebJahr, Epoche epoche)
    implements DataObject {
```

Wir verwenden hier eine neue Annotation, die wir noch definieren werden. Die Annotation heißt `VARCHAR` und sie hat genau ein Argument. Sie soll dazu dienen, um zu spezifiizieren, wie lang maximal ein String in der Datenbank werden darf.

Auch für diese Klasse sei ein zweiter Konstruktor definiert, der die uid auf den Standardwert von `-1` setzt.

java: Person

```
  public Person
      (String name, String vorname, int gebJahr, Epoche epoche) {
    this(new UID(), name, vorname, gebJahr, epoche);
  }
}
```

Das Spannende an dieser Klasse ist, dass sie Objekte einer zweiten `DataObject`-Klasse referenziert.

Auch hier ein Beispiel der Objekterzeugung.

Shell

```
jshell> var p = new Person("Hoffmann","ETA",1776,g)
p ==> Person[uid=-1, name=Hoffmann, vorname=ETA, gebJah ... Romantik,
↪  jahrhundert=19]]
```

1.2.1. Beispielaufrufe

Bevor es an eine allgemeine Umsetzung der Funktionalität für beliebige Record-Klassen geht, die `DataObject` implementieren, zeigen wir diese bereits in Aktion.

Tabellen erzeugen Für eine DataObject-Klasse soll es möglich sein, Datenbanktabellen zu erzeugen. Diese Erzeugung soll für jede korrekte `DataObject` Record-Klasse funktionieren. Dabei soll in einer Datenbank, die über eine `Connection` spezifiziert ist, die entsprechende Tabelle angelegt werden. Dabei sollen die Typen berücksichtigt werden und über id's auch auch referenzierte Typen zugegriffen werden. Desweiteren ist die Annotation `VARCHAR` für die Datenbank auszuwerten.

Shell

```
jshell> createTable(getConnection(),Epoche.class)

jshell> createTable(getConnection(),Person.class)
```

Objekte Speichern Wenn die Tabellen erstellt wurden, soll möglich sein, die Objekte in der Datenbank abzulegen.

Shell

```
jshell> p.save(getConnection())

jshell> p
p ==> Person[uid=1, name=Hoffmann, vorname=ETA, gebJahr=1776,
↪  epoche=Epoche[uid=1, name=Romantik, jahrhundert=19]]
```

Hierzu soll für jedes Objekt die Metode `save` zur Verfügung stehen. Ist das Objekt noch nicht in der Datenbank, so hat es die uid -1. Wenn das Objekt gespeichert wurde, erhält es die uid aus der Datenbank.

Ist ein Objekt mit der entsprechenden uid bereits in der Datenbank gespeichert, so bewirkt der Aufruf von `save` ein Update.

Objekte Selektieren Und schließlich sollen Objekte wieder aus er Datenbank selektiert werden können. Hierzu gibt es die Funktion `select`. Sie bekommt neben der Datenbankverbindung ein Class-Objekt, das angibt, aus welcher Datenbanktabelle Objekte selektiert werden sollen.

Shell

```
jshell> select(getConnection(),Person.class)
$8 ==> [Person[uid=1, name=Hoffmann, vorname=ETA, gebJahr=1776,
↪  epoche=Epoche[uid=1, name=Romantik, jahrhundert=19]]]
```

Das Speichern bewirkt, dass auch alle referenzierten Objekte gespeichert werden.

Als weiteres Beispiel wird ein neues Personenobjekt erzeugt, aber unter Verwendung desselben Epochenobjekts.

Shell

```
jshell> p = new Person("Hoffmann","ETA",1776,g)
p ==> Person[uid=-1, name=Hoffmann, vorname=ETA, gebJah ... Romantik,
↪  jahrhundert=19]]
```

Speichern wir nun dieses:

Shell

```
jshell> p.save(getConnection())
```

Haben wir weiterhin nur ein Epochenobjekt in der Datenbank:

Shell

```
jshell> select(getConnection(),Epoche.class)
SELECT * FROM Epoche;
$11 ==> [Epoche[uid=1, name=Romantik, jahrhundert=19]]
```

Aber zwei unterschiedliche Personenobjekte:

Shell

```
jshell> select(getConnection(),Person.class)
$12 ==> [Person[uid=1, name=Hoffmann, vorname=ETA, gebJahr=1776,
↪  epoche=Epoche[uid=1, name=Romantik, jahrhundert=19]], Person[uid=2,
↪  name=Hoffmann, vorname=ETA, gebJahr=1776, epoche=Epoche[uid=1,
↪  name=Romantik, jahrhundert=19]]]

jshell>
```

1.3. Implementierung

1.3.1. Annotationen

Wir beginnen mit der Annotation. Wir haben im Beispiel bereits exemplarisch ein Record-Feld mit einer Annotation versehen. Es war mit `VARCHAR(200)` annotiert. Hierzu müssen wir eine entsprechende Klasse implementieren. Es reicht hierbei, eine Schnittstelle zu definieren, die die entsprechende Annotation beschreibt.

Diese Annotationsschnittstelle ist zusätzlich mit einer `Retention` annotiert, die anzeigt, in welchen Phasen von Kompilierung und Ausführung in Java, die Annotation gespeichert sein und zur Verfügung stehen soll. Wir benötigen die Information während der Laufzeit und setzen somit den `Retention`-Wert auf `RUNTIME`.

java: VARCHAR

```java
package name.panitz.util;
import java.lang.annotation.*;

@Retention(value=RetentionPolicy.RUNTIME)
public @interface VARCHAR {
  int value();
}
```

Von nun an können wir die Annotation `VARCHAR` mit einem `int`-Wert in unserem Quelltext setzen. Zusätzlich können wir diese Information mit dem Reflection-API in der Laufzeit erfragen.

1.3.2. Tabellen erzeugen

Bevor wir die Objekte in einer relationalen Datenbank speichern, braucht die Datenbank eine entsprechende Tabelle, in der die Objekte gespeichert werden können.

Hierfür definieren wir zunächst eine Hilfsmethode, die eine Exception wirft, wenn für die übergebene Klasse nicht die automatische Persistierung unterstützt wird.

Der Parameter ist dabei ein Objekt der Klasse `Class` die interne Informationen für eine Klasse zur Laufzeit zur Verfügung stellt.

Zunächst testen wir, ob es sich bei der übergebenden Klasse um eine Record-Klasse handelt:

Java: DataObject

```java
static void checkPersistanceCapable(Class<?> klasse) {
  if (!klasse.isRecord())
    throw new RuntimeException("class is not a record: " + klasse);
```

Dann schauen wir, ob diese Klasse auch die Schnittstelle `DataObject` implementiert.

Java: DataObject

```
if (!DataObject.class.isAssignableFrom(klasse))
  throw new RuntimeException("class is not a DataObject: " + klasse);
```

Schließlich wird für generische Klasse die automatische Persistierung nicht unterstützt:

Java: DataObject

```
  if (klasse.getTypeParameters().length > 0)
    throw new RuntimeException
      ("Persistence of generic DataObject class is not supported: "
       + klasse);
}
```

Zunächst wird ein String erzeugt, der den SQL-Code zur Generierung der Tabellen enthält. Als Argument bekommt die Funktion die Klasse, für die die Create-Anweisung erzeugt werden soll.

Java: DataObject

```
static String createStatement(Class<?> klasse) {
  checkPersistanceCapable(klasse);
```

Nachdem sichergestellt ist, dass für die übergabene `klasse` eine Datenbanktabelle erzeugt werden kann, sammeln wir Stückweise das Ergebnis in einem `StringBuffer`. Die SQL-Anweisung beginnt mit einem `CREATE TABLE`:

Java: DataObject

```
var r = new StringBuffer("CREATE TABLE ");
r.append( klasse.getSimpleName() + " (");
```

Jetzt betrachten wir die Record-Felder. Hier kommt nun das Reflection-API ins Spiel. Für `Class`-Objekte lassen sich alle Informationen für die Klasse erfragen und für Record-Klassen sind das die einzelnen Record-Felder, die mit der Methode `getRecordComponents` erfragt werden können.

Java: DataObject

```
var first = true;
var rcs = klasse.getRecordComponents();
```

Jedes Record-Feld ergibt eine Datenbankspalte in der Tabelle. Wir iterieren dementsprechend über die Record-Felder:

Java: DataObject

```
for (var rc : rcs) {
```

Wenn wir nicht das erste Feld sind, trennen wir in der `CREATE`-Anweisung von SQL das Feld von dem vorherigen mit einem Komma:

Java: DataObject

```
if (first) first = !first;
else r.append(", ");
```

Dann folgt der Name des Feldes als Spaltenbezeichner der Datenbank. Auch den Namen des Feldes können wir mit dem Reflection-API erfragen.

Java: DataObject

```
r.append(rc.getName() + " ");
```

Nun geht es darum, für das Feld den richtigen Datenbanktyp zu finden.

Wir erfragen den Typ des Feldes:

Java: DataObject

```
var typ = rc.getType();
```

Handelt es sich hierbei um einen primitiven Typ, für den es direkt einen Datenbanktypen gibt, so lassen wir uns diesen direkt aus einer Abbildung geben:[1]

Java: DataObject

```
var dbt = dbTypes.get(typ.getName());
if (null!=dbt) {
  r.append(dbt);
```

Bei String-Feldern verwenden wir den Datenbanktyp `VARCHAR`.[2]

Hierbei ist jetzt von Interesse, ob das Feld eine Annotation des Typs `VARCHAR` hat. Dieses lässt sich wieder durch das Reflection-API erfragen. Gibt es eine solche Annotation, hat die einen Wert, der für die Länge des `VARCHAR` genommen werden soll. Ansonsten nehmen wir den Standardwert 100.

[1] Die Abbildung `dbTypes` ist weiter unten definiert.
[2] Durch weitere Annotation könnten wir auch den Typ `BLOB` unterstützen.

Java: DataObject

```
} else if (typ == String.class) {
  var an = rc.getAnnotation(VARCHAR.class);
  r.append("VARCHAR("+(null == an? 100: an.value()) + ")");
```

Der interessanteste Punkt sind die Record-Felder, die wieder ganze Objekte enthalten, die zu persistieren sind. In der Datenbanktabelle ist hierfür nur eine Zahl der uid des anderen Objektes hinterlegt. Dieses ist ein Schlüssel in einer anderen (oder auch derselben) Tabelle.

Java: DataObject

```
} else if (DataObject.class.isAssignableFrom(typ)) {
  r.append("BIGINT"); // TODO foreign key (...)
```

Schließlich haben wir noch das Feld für die uid.

Java: DataObject

```
} else if (typ == UID.class) {
  r.append("BIGINT NOT NULL AUTO_INCREMENT");
```

Andere Typen werden nicht unterstützt und es kommt zu einen Laufzeitfehler.

Java: DataObject

```
  } else {
    throw new RuntimeException("unsupported record field type: " + rc);
  }
}
```

Ganz zum Schluss bekommt die Tabelle noch die Information, dass die Spalte mit der uid der Primärschlüssel der Tabelle ist.

Java: DataObject

```
  r.append(", PRIMARY KEY (uid));");
  return r.toString();
}
```

Hier noch die verwendete Abbildung auf die Datenbanktypen.

Java: DataObject

```
static Map<String,String> dbTypes = Map.ofEntries
 (entry("byte", "TINYINT"),entry("Byte", "TINYINT")
 ,entry("short", "SMALLINT"),entry("Short", "SMALLINT")
 ,entry("int", "INT"),entry("Integer", "INT")
 ,entry("long", "BIGINT"),entry("Long", "BIGINT")
 ,entry("float", "REAL"),entry("Float", "REAL")
 ,entry("double", "DOUBLE"),entry("Double", "DOUBLE")
 ,entry("boolean", "BOOLEAN"),entry("Boolean", "BOOLEN")
 );
```

Für die beiden Beispielklassen lassen wir die entsprechende SQL-Anweisung einmal erzeugen:

Shell

```
jshell> createStatement(Person.class)
$6 ==> "CREATE TABLE Person (uid BIGINT NOT NULL AUTO_INCREMENT
, name VARCHAR(200), vorname VARCHAR(100), gebJahr INT, epoche BIGINT
, PRIMARY KEY (uid));"

jshell> createStatement(Epoche.class)
$7 ==> "CREATE TABLE Epoche (uid BIGINT NOT NULL AUTO_INCREMENT
, name VARCHAR(100), jahrhundert INT, PRIMARY KEY (uid));"
```

Eine weitere Funktion sieht vor, dass die CREATE-Anweisungen direkt auf einer Datenbank ausgeführt werden.

Java: DataObject

```
static void createTable(Connection con, Class<?> klasse) {
  try {
    con.createStatement().execute(createStatement(klasse));
  } catch (SQLException e) {
    e.printStackTrace();
  }
}
```

1.3.3. Objekte Speichern

Nachdem nun die Datenbanktabellen erzeugt werden können, sollen Objekte in die Datenbank gespeichert werden. Hierzu definieren wir die Methode `save` als default-Methode der Klasse `DataObject`, so dass sie allen implementierenden Klassen zur Verfügung steht.

Wir stellen zunächst erst einmal sicher, ob die Klasse nicht eine Eigenschaft verletzt, um persistiert zu werden:

Java: DataObject

```
default void save(Connection con) {
  try {
    var klasse = this.getClass();
    checkPersistanceCapable(klasse);
```

Es geht jetzt darum, eine SQL INSERT-Anweisung auszuführen. Wir sammeln dazu alle Feldnamen, das sind die Spalten der Datenbank und alle konkreten Werte für das konkrete Objekt zur Erzeugung des SQL-Statements in `StringBuffer`-Objekten:

Java: DataObject

```
var fieldNames = new StringBuffer(" (");
var fieldValues = new StringBuffer();
```

Wie bereits beim Erzeugen der Tabellen, iterieren wir über die Felder der Record-Klasse:

Java: DataObject

```
var rcs = klasse.getRecordComponents();
var first = true;
boolean isUpdate = true;
for (var rc : rcs) {
```

Da wir es jetzt mit einem konkreten Objekt zu tun haben, können wir das in dem Feld gespeicherte Objekt, mit dem Reflection-API erfragen. Hierzu lässt sich mit `getAccessor()` die Getter-Methode geben, auf die die Methode `invoke` für den Aufruf getätigt werden kann. Hier sieht man, wie im Reflection-API Methoden auf Objekten ausgeführt werden können, deren Klassen in der Compilierung noch gar nicht bekannt waren.

Java: DataObject

```
var recordData = rc.getAccessor().invoke(this);
```

Wir müssen unterscheiden, ob wir ein `INSERT` oder ein `UPDATE` auf der Datenbank machen. Das erkennen wir an den Wert des Feldes uid.

Java: DataObject

```
if (rc.getName().equals("uid") && ((UID) recordData).uid == -1) {
  isUpdate = false;
  continue;
}
```

Wie auch schon beim Erzeugen der Tabellen sind alle Felder nach dem ersten Feld mit einem Komma abzugrenzen:

Java: DataObject

```
if (first) first = !first;
else {
  fieldNames.append(", ");
  fieldValues.append(", ");
}
```

Der Feldnamen wird zu dem entsprechenden String-Buffer hinzugefügt:

Java: DataObject

```
fieldNames.append(rc.getName());
```

Bei einem Update sind Zuweisungen im SQL-Code zu erzeugen.

Java: DataObject

```
if (isUpdate) fieldValues.append(rc.getName() + " = ");
```

Jetzt geht es darum in den Feld-Werten, die Daten für SQL zu schreiben.

Dazu lassen wir uns den Typ der Daten, die in dem aktuellen Feld gespeichert sind, geben:

Java: DataObject

```
var typ = rc.getType(); //recordData.getClass();
```

Für die meisten Typen können wir die Daten direkt in ihrer String-Darstellung für die SQL-Query übernehmen:

Java: DataObject

```
if (dbTypes.get(typ.getName())!=null || typ == String.class) {
  fieldValues.append("\'" + recordData + "\'");
```

Für das Feld uid ist der eigentliche Wert der UID zu nehmen:

Java: DataObject

```
} else if (typ == UID.class) {
  fieldValues.append("\'" + ((UID) recordData).uid + "\'");
```

Am interessantesten sind Felder von einem Typ, der ebenfalls persistierbar ist. Es sind Felder von einem Typ, der die Schnittstelle `DataObject` implementiert.

Hier speichern wir zunächst das referenzierte Objekt. Dieses bekommt dann von der Datenbank die korrekte uid gesetzt. Diese verwenden wir dann als Fremdschlüssel in der entsprechenden Tabelle.

Auch null-Werte werden unterstützt und bekommen die uid -1 zugewiesen.

Java: DataObject

```
} else if (DataObject.class.isAssignableFrom(typ)) {
  if (null==recordData){
    fieldValues.append("\'-1\'");
  }else{
    var c = ((DataObject) recordData);
    c.save(con);
    fieldValues.append("\'" + c.uid().uid + "\'");
  }
```

Sollten weitere Typen für eine Feld aufgetreten sein, so hanelt es sich um einen Fehler.

Java: DataObject

```
  } else {
    throw new RuntimeException("unsupported record field type: " + rc);
  }
}
```

Die Schleife über alle Felder der Record-Klasse ist beendet. Den SQL-Code für die Feldnamen können wir mit einer schließenden Klammer beenden.

Java: DataObject

```
fieldNames.append(")");
```

Jetzt geht es darum, die eigentliche SQL-Anweisung zu schreiben.

Java: DataObject

```
var sql = new StringBuffer(isUpdate ? "UPDATE " : "INSERT INTO ");
sql.append(klasse.getSimpleName());
if (!isUpdate){
  sql.append(fieldNames);
  sql.append(" VALUES (");
}else
  sql.append(" SET ");
sql.append(fieldValues);
if (!isUpdate)
  sql.append(");");
else
  sql.append(" WHERE uid = " + uid().uid + ";");
```

Die so erhaltende Anweisung lässt sich auf der Datenbankverbindung ausführen. Wir sind an der uid interessiert.

Java: DataObject

```
var statement = con.prepareStatement
   (sql.toString(), Statement.RETURN_GENERATED_KEYS);
statement.executeUpdate();
var rs = statement.getGeneratedKeys();
```

Wurde eine neue uid erzeugt, weil es sich um ein INSERT gehandelt hat, so setzen wir diese für unser Objekt, das gerade gespeichert wurde.

Java: DataObject

```
    if (rs.next()) {
       setUID(rs.getLong(1));
    }
  } catch (Exception e) {
    throw new RuntimeException(e);
  }
}
```

1.3.4. Objekte Selektieren

Nun können wir unsere Objekte, ohne weiteren Code zu schreiben, in eine Datenbank abspeichern. Natürlich soll es auch möglich sein, Objekte wieder aus der Datenbank zu extrahieren, also eine klassische SELECT-Anweisung auf die Datenbank zu machen.

Selektoren Wir wollen nicht immer alle Objekte zu einer Klasse aus der Datenbank laden, sondern auch hier Selektionskriterien anwenden.

Zum Spezifizieren, dass wir nur Objekte haben wollen, in denen bestimmte Spalten feste Werte haben, dient die folgende Record-Klasse:

Java: DataObject

```
static record Selector(String column, String value) {  }
```

Es wird ausgedrückt, dass man nur Objekte selektieren will, die in einer bestimmten Spalte einen bestimmten Wert haben.

Selektion Die eigentliche Selektionsmethode, soll über eine Datenbankverbindung für eine Klasse Objekte selektieren, und dabei eine beliebige Anzahl von Selektoren berücksichtigen. Das Ergebnis ist die Liste der selektierten Objekte.

Java: DataObject

```
@SuppressWarnings("unchecked")
static <C> List<C> select(Connection con,Class<C> klasse,Selector...sls){
  checkPersistanceCapable(klasse);
```

Die Selektanweisung an sich ist einfach. Ein `SELECT *` in SQL für die Tabelle, die die spezifizierte Klasse repräsentiert.

Java: DataObject

```
try {
  var sql = new StringBuffer("SELECT * FROM " + klasse.getSimpleName());
  if (sls.length > 0)
```

Sollten wir Selektoren haben, dann bekommt die SQL-Anfrage noch eine `WHERE`-Klausel:

Java: DataObject

```
sql.append(" WHERE ");
```

Wir arbeiten nacheinander die Selektoren ab, um die komplette `WHERE`-Klausel zu generieren.

Java: DataObject

```
var first = true;
for (var sel : sls) {
  if (first)first = false;
  else  sql.append(", ");
  sql.append(sel.column);
  sql.append(" = '");
  sql.append(sel.value());
  sql.append("'");
}
sql.append(";");
```

Die erzeugte SQL-Anweisung wird nun über die Datenbankverbindung ausgeführt:

Java: DataObject

```
var stat = con.createStatement();
var rs = stat.executeQuery(sql.toString());
```

Nun geht es darum, die Ergebnisliste zu erzeugen, in der die Objekte eingefügt werden. Hierzu wird eine Liste für das Ergebnis erstellt.

Java: DataObject

```
var result = new ArrayList<C>();
```

Wir iterieren über alle Ergebnisse der SQL-Anfrage. Jedes gibt einen Eintrag in der Ergebnisliste.

Java: DataObject

```
while (rs.next()) {
```

Wir müssen die Objekte über das Reflection-API erzeugen. Hierzu brauchen wir die Argumente für den Konstruktor der Record-Klasse. Die Argumente werden in einem Array gesammelt.

Java: DataObject

```
var rcs = klasse.getRecordComponents();
var args = new Object[rcs.length];
int i = 0;
```

Nun iterieren wir über die Felder der Record-Klasse. Diese repräsentieren ja die Argumente des Konstruktors.

Die Indexvariable `i` führt Buch über den Index des bearbeiteten Feldes.

Java: DataObject

```
for (var rc : rcs) {
```

Um die Daten korrekt aus dem Datenbankobjekt zu extrahieren, betrachten wir den Typ des Feldes.

Java: DataObject

```
var typ1 = rc.getType();
```

Für die primitiven Typen und deren Wrapper-Klassen gibt es Zugriffsmethoden.

Java: DataObject

```
if (typ1 == boolean.class || typ1 == Boolean.class)
  args[i] = rs.getBoolean(rc.getName());
else if (typ1 == int.class || typ1 == Integer.class)
  args[i] = rs.getInt(rc.getName());
else if (typ1 == long.class || typ1 == Long.class)
  args[i] = rs.getLong(rc.getName());
else if (typ1 == float.class || typ1 == Float.class)
  args[i] = rs.getFloat(rc.getName());
else if (typ1 == double.class || typ1 == Double.class)
  args[i] = rs.getDouble(rc.getName());
else if (typ1 == byte.class || typ1 == Byte.class)
  args[i] = rs.getByte(rc.getName());
else if (typ1 == short.class || typ1 == Short.class)
  args[i] = rs.getShort(rc.getName());
```

Ebenso gibt es eine Methode, um ein String-Objekt aus dem Datenbankergebnis zu erhalten.

Java: DataObject

```
else if (typ1 == String.class)
  args[i] = rs.getString(rc.getName());
```

Spannend sind jetzt referenzierte Objekte. Hier haben wir bisher nur die uid. Mit dieser bauen wir einen neuen Selektor, um das referenzierte Objekt mit einer weiteren Anfrage aus der Datenbank zu extrahieren. Es ist ein rekursiver Aufruf auf die `select`-Methode, analog dazu, dass wir einen rekursiven Aufruf bei der Methode `save` brauchten, um referenzierte Objekte zu speichern.

Wie beim Speichern der Objekte werden auch hier als Sonderfall die Null-Werte berücksichtigt, die in der Datenbank durch den Wert -1 ausgedrückt sind.

Java: DataObject

```
else if (DataObject.class.isAssignableFrom(typ1)){
  var fid = rs.getLong(rc.getName());
  if (fid==-1){
    args[i] = null;
  }else{
    args[i] = select(con, typ1, new Selector("uid", fid + "")).get(0);
  }
}
```

Ein kleiner Sonderfall ist noch die uid. Diese ist als long-Wert in der Datenbank gespeichert:

Java: DataObject

```
else if (typ1 == UID.class)
  args[i] = new UID(rs.getLong(rc.getName()));
```

Alle weiteren Typen sollten nicht vorkommen. Dann handelt es sich um einen Fehler.

Java: DataObject

```
else
  throw new RuntimeException("unsupported type: " + typ1);
```

Zum Ende der foreach-Schleife ist der Index zu erhöhen

Java: DataObject

```
  i++;
}
```

Im Array `args` sind jetzt die Objekte der Argumente für den Konstruktoraufruf der Recordklasse gespeichert. Nun brauchen wir den entsprechenden Konstruktor, um diesen über Reflection aufzurufen. Unsere Record-Klassen haben mehrere Konstruktoren. Wir benötigen den, mit der höchsten Parameteranzahl.

Java: DataObject

```
var constr = Arrays.stream(klasse.getConstructors())
  .reduce((r, x)->r.getParameterCount() > x.getParameterCount() ? r : x);
```

Diesen Konstruktor können wir über die Reflection-Methode `newInstance` aufrufen. Wir erhalten ein Objekt des allgemeinen Typs `Object` und müssen dieses zu unseren Ergebnistypen casten.

Java: DataObject

```
      C elem = (C) constr.get().newInstance(args);
      result.add(elem);
    }
    return result;
  } catch (Exception e) {
    throw new RuntimeException(e);
  }
}
```

1.4. Rekursives Beispiel

Jetzt können wir für beliebige Record-Klassen, die `DataObject` implementieren direkt .

Das können auch rekursive Klassen sein. Hier ein Beispiel für eine Klasse von Binärbäumen.

java: Tree

```
package name.panitz.util;
public record Tree(UID uid,Tree left,int elem,Tree right)
                                          implements DataObject{
  public Tree(Tree left,int elem,Tree right){
    this(new UID(), left, elem, right);
  }
  public Tree(int elem){
    this(new UID(), null, elem, null);
  }
}
```

In einer interaktiven Session können wir das einmal austesten.

Wir lassen die Datenbanktabelle erzeugen:

Shell

```
jshell> createTable(getConnection(),Tree.class)
```

Erzeugen einen kleinen Binärbaum:

Shell

```
jshell> var t = new Tree(new Tree(new Tree(7),14,new Tree(18)),21,
new Tree(new Tree(31),42,null))
t ==> Tree[uid=-1, left=Tree[uid=-1, left=Tree[uid=-1,  ... ll],
elem=42, right=null]]
```

Speichern diesen

Shell

```
jshell> t.save(getConnection())
```

Dadurch erhalten die Baumknoten eine uid aus der Datenbank

Shell

```
jshell> t
t ==> Tree[uid=6, left=Tree[uid=3, left=Tree[uid=1, left=null, elem=7,
right=null], elem=14, right=Tree[uid=2, left=null, elem=18, right=null]],
elem=21, right=Tree[uid=5, left=Tree[uid=4, left=null, elem=31,
right=null], elem=42, right=null]]
```

Durch Selektion des Baumes über die uid des Wurzelknotens erlangen wir wieder den kompletten Baum aus der Datenbank.

Shell

```
jshell> select(getConnection(),Tree.class,new Selector("uid","6"))
$7 ==> [Tree[uid=6, left=Tree[uid=3, left=Tree[uid=1, left=null, elem=7,
right=null], elem=14, right=Tree[uid=2, left=null, elem=18, right=null]],
elem=21, right=Tree[uid=5, left=Tree[uid=4, left=null, elem=31,
right=null], elem=42, right=null]]]
```

1.5. Persistence Frameworks

Hier noch einen Überblick über Hibernate, Jakarta Persistence API, Scala Lift Framework, Spring und wer sonst noch so ORM macht.

2. Aufgaben

Aufgabe 1 Ergänzen Sie die Schnittstelle `DataObject` um eine Methode, mit deren Hilfe ein peristiertes Objekt wieder aus der Datenbank gelöscht werden kann. Es sollen referenzierte Objekte nicht gelöscht werden.

Java: DataObject

```java
default void delete(Connection con){
  //TODO
}
```

Aufgabe 2 Ergänzen Sie die Schnittstelle `DataObject` um eine Methode, mit deren Hilfe ein peristiertes Objekt wieder aus der Datenbank gelöscht werden kann. Es sollen jetzt auch alle referenzierte Objekte gelöscht werden.

Java: DataObject

```
default void deepDelete(Connection con){
  //TODO
}
```

Aufgabe 3 In den folgenden Aufgaben geht es jetzt darum mit dem Reflection-API eine Serialisierung und Deserialisierung in XML für über eine Record-Klasse definierte Objekte der Schnittstelle `DataObject` allgemein umzusetzen.

Zunächst sollen für die Objekte XML-Elemente eines XML-Dokuments im DOM-API erstellt werden.

Hierzu wird zunächst ein DOM Dokument erstellt:

Java: DataObject

```
default Element toXML() {
  try {
    var doc = DocumentBuilderFactory.newInstance()
      .newDocumentBuilder().newDocument();
    return toXML(doc);
  } catch (Exception e) {
    throw new RuntimeException(e);
  }
}
```

Vervollständigen Sie jetzt die Methode, die das this-Objekt als DOM-Element erzeugt.

Java: DataObject

```
default Element toXML(Document doc) {
  try {
    var r=doc.createElement(getClass().getName().replaceAll("\\$",""));
    r.setAttribute("type", this.getClass().getName());
    var klasse = this.getClass();
    var rcs = klasse.getRecordComponents();
    for (var rc : rcs) {
    /* TODO
       Hier sind jetzt rekursiv die Objekte des Record-Felder als
       Kinder des DataObjektes einzutragen.
       Die Kindelemente haben als Tagnamen den Namen des Record-Feldes.
       Jedes Kindelement soll auch das Attribute type haben, das den
       Wert des Laufzeittypen des Record-Feldes hat.
    */
    }
    return r;
  } catch (Exception e) {
    throw new RuntimeException(e);
  }
}
```

Folgende kleine Liste, die die Klassen für die primitiven Typen in Java bereithält kann bei der Umsetzung hilfreich sein.

Java: DataObject

```
List<Class<?>> primWrapper
  = List.of(Integer.class, Long.class, Boolean.class, Short.class
          ,Character.class, Byte.class, Float.class, Double.class);
```

Wenn Sie nicht an ein DOM-Element, sondern direkt an dem Dokument als String interessietr sind, dann kann man mit folgener kleinen Methoden, das DOM-Objekt als String serialisieren.

Java: DataObject

```
default String toXMLString() {
  var writer = new StringWriter();
  try {
    var transformer = TransformerFactory.newInstance().newTransformer();
    transformer.transform
        (new DOMSource(toXML()), new StreamResult(writer));
    return writer.toString();
  } catch (Exception e) {
    throw new RuntimeException(e);
  }
}
```

Hier ein kleiner Testaufruf für die umgesetzte Funktionalität:

Shell

```
jshell> var g = new Epoche("Romantik",19)
g ==> Epoche[uid=-1, name=Romantik, jahrhundert=19]

jshell> var p = new Person("Hoffmann","ETA",1776,g)
p ==> Person[uid=-1, name=Hoffmann, vorname=ETA, gebJah ... Romantik,
↪  jahrhundert=19]]

jshell> System.out.println(p.toXMLString())
<?xml version="1.0" encoding="UTF-8"?>
<name.panitz.util.Person type="name.panitz.util.Person"
  ><name type="java.lang.String">Hoffmann</name
  ><vorname type="java.lang.String">ETA</vorname
  ><gebJahr type="java.lang.Integer">1776</gebJahr
  ><epoche type="name.panitz.util.Epoche"
    ><name type="java.lang.String">Romantik</name
    ><jahrhundert type="java.lang.Integer">19</jahrhundert
  ></epoche
></name.panitz.util.Person>
```

Aufgabe 4 In dieser Aufgabe geht es jetzt um die inverse Funktionalität: aus einem XML-DOM Objekt soll wieder das ursprüngliech Objekt deserialisiert werden.

Vervollständigen Sie die Methode, die für einen Klasse aus einem in der ersten Aufgabe erzeugten DOM-Element wieder das ursprüngliche Objekt erzeugt.

Java: DataObject

```java
@SuppressWarnings("unchecked")
static <C> C fromXML(Class<C> klasse, Element node) {
  try {
    if (!DataObject.class.isAssignableFrom(klasse))
      throw new RuntimeException("class is not a DataObject: " + klasse);
    var rcs = klasse.getRecordComponents();
    var cs = node.getChildNodes();
    var args = new Object[rcs.length - 1]; //ohne uid, deshalb -1
    var i = 0;
    for (var rc : rcs) {
      if (rc.getName().equals("uid"))
        continue;
      Node child = cs.item(i);
      if (child.getNodeType() != Node.ELEMENT_NODE)
        throw new RuntimeException("Not an Element");

      /* TODO
         Lesen Sie die Kindknoten des Elements ein.
         Erzeugen Sie für die die entsprechenden Objekte,
         die sie in die Argumente einfügen.
         Das Vorgehen entspricht in Teilen den hinteren Teil der
         select Methode.
      */
      i++;
    }
    //der Konstruktor ohne uid, also der mit weniger Argumenten
    var constr = Arrays.stream(klasse.getConstructors())
      .reduce((r, x)->r.getParameterCount()==args.length?r:x);
    return (C) constr.get().newInstance(args);
  } catch (Exception e) {
    throw new RuntimeException(e);
  }
}
```

Wenn Sie ein XML-Dokument aus einen InputStream einlesen wollen, um es dann zu deserialisieren, können Sie nun folgende Methode verwenden.

Java: DataObject

```
static <C> C fromXML(Class<C> klasse, InputStream in) {
  try {
    return fromXML
             (klasse
             , DocumentBuilderFactory
                .newInstance()
                .newDocumentBuilder()
                .parse(in).getDocumentElement());
  } catch (Exception e) {
    e.printStackTrace();
    throw new RuntimeException();
  }
}
```

Ist das XML-Dokument diekt in einem String codiert, dann kann es mit folgender Methode deserialisiert werden.

Java: DataObject

```
static <C> C fromXML(Class<C> klasse, String in) {
  return fromXML(klasse, new ByteArrayInputStream(in.getBytes()));
}
```

Sie sollten jetzt in der Lage sein, dass die Objekte folgenden Rundtrip schaffen:

Shell

```
jshell> var g = new Epoche("Romantik",19)
g ==> Epoche[uid=-1, name=Romantik, jahrhundert=19]

jshell> var p = new Person("Hoffmann","ETA",1776,g)
p ==> Person[uid=-1, name=Hoffmann, vorname=ETA, gebJah ... Romantik,
↪  jahrhundert=19]]

jshell> var xml = p.toXMLString()
xml ==> "<?xml version=\"1.0\" encoding=\"UTF-8\"?><name. ...
/name.panitz.util.Person>"

jshell> DataObject.fromXML(Person.class,xml)
$5 ==> Person[uid=-1, name=Hoffmann, vorname=ETA, gebJahr=1776,
↪  epoche=Epoche[uid=-1, name=Romantik, jahrhundert=19]]
```

A. Datenbankverbindung für schnelle Tests

Für einfache schnelle Tests können wir das Datenbanksystem H2 verwenden. Dieses hat den Votreil, dass keine zusätzliche Administration oder Installation notwendig ist. Es muss lediglich die entsprechende Jar-Datei auf dem Klassenpfad liegen.

Die benötigte JAR-DAtei befindet sich im Archiv, dass von folgender URL geladen werden kann: `https://h2database.com/h2-2019-10-14.zip`

Java: DataObject

```
String JDBC_DRIVER = "org.h2.Driver";
String DB_URL = "jdbc:h2:./userData/test1";

// Database credentials
String USER = "sa";
String PASS = "";
```

Mit diesen Werten lässt sich eine Verbindung zur Datenbank erstellen:

Java: DataObject

```
static Connection getConnection() {
  try {
    Class.forName(JDBC_DRIVER);
    return DriverManager.getConnection(DB_URL, USER, PASS);
  } catch (Exception e) {
    e.printStackTrace();
    throw new RuntimeException(e);
  }
};
```

Java: Json

```
}
```