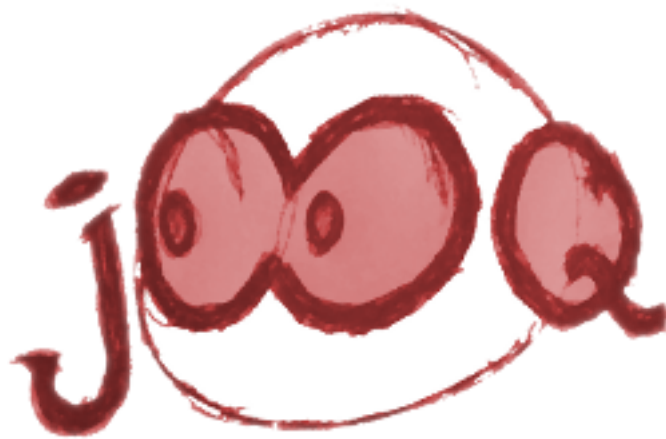# The jOOQ User Manual

SQL was never meant to be abstracted. To be confined in the narrow boundaries of heavy mappers, hiding the beauty and simplicity of relational data. SQL was never meant to be object-oriented. SQL was never meant to be anything other than... SQL!

Overview

This manual is divided into four main sections:

- **jOOQ classes and their usage**
  See these chapters for an overview of the jOOQ internal architecture and all types that are involved with jOOQ's query creation and execution. This is the important part for you, also, if you wish to extend jOOQ
- **Meta model code generation**
  See these chapters to understand how you can use jOOQ as a source code generator, and what type of artefacts are generated by jOOQ
- **DSL or fluent API. Where SQL meets Java**
  See these chapters to learn about how to use jOOQ in every day's work. The jOOQ DSL is the main way to create and execute jOOQ queries almost as if SQL was embedded in Java directly
- **Advanced topics**
  Some advanced topics including not-everyday functionality

Table of contents

1. jOOQ classes and their usage

In these sections, you will learn about how to use jOOQ object factories and the jOOQ query model, to express your SQL in jOOQ

Overview

jOOQ essentially has two packages:

- org.jooq: the jOOQ API. Here you will find interfaces for all SQL concepts
- org.jooq.impl: the jOOQ implementation and factories. Most implementation classes are package private, you can only access them using the **org.jooq.impl.Factory**

This section is about the main jOOQ classes and the global architecture. Most of the time, however, you will be using the **DSL or fluent API. Where SQL meets Java** in order to create queries the way you're used to in SQL

1.1. The example database

For the examples in this manual, the same database will always be referred to. It essentially consists of these entities created using the Oracle dialect

Example CREATE TABLE statements

```
CREATE TABLE t_language (
  id NUMBER(7) NOT NULL PRIMARY KEY,
  cd CHAR(2) NOT NULL,
  description VARCHAR2(50)
)

CREATE TABLE t_author (
  id NUMBER(7) NOT NULL PRIMARY KEY,
  first_name VARCHAR2(50),
  last_name VARCHAR2(50) NOT NULL,
  date_of_birth DATE,
  year_of_birth NUMBER(7)
)

CREATE TABLE t_book (
  id NUMBER(7) NOT NULL PRIMARY KEY,
  author_id NUMBER(7) NOT NULL,
  title VARCHAR2(400) NOT NULL,
  published_in NUMBER(7) NOT NULL,
  language_id NUMBER(7) NOT NULL,
  FOREIGN KEY (AUTHOR_ID) REFERENCES T_AUTHOR(ID),
  FOREIGN KEY (LANGUAGE_ID) REFERENCES T_LANGUAGE(ID)
)

CREATE TABLE t_book_store (
  name VARCHAR2(400) NOT NULL UNIQUE
)

CREATE TABLE t_book_to_book_store (
  book_store_name VARCHAR2(400) NOT NULL,
  book_id INTEGER NOT NULL,
  stock INTEGER,
  PRIMARY KEY(book_store_name, book_id),
  CONSTRAINT b2bs_book_store_id
    FOREIGN KEY (book_store_name)
    REFERENCES t_book_store (name)
    ON DELETE CASCADE,
  CONSTRAINT b2bs_book_id
    FOREIGN KEY (book_id)
    REFERENCES t_book (id)
    ON DELETE CASCADE
)
```

More entities, types (e.g. UDT's, ARRAY types, ENUM types, etc), stored procedures and packages are introduced for specific examples

1.2. The Factory class

jOOQ hides most implementation facts from you by letting you use the jOOQ Factory as a single entry point to all of the jOOQ API. This way, you can discover all of the API using syntax auto-completion, for instance.

The Factory and the jOOQ API

jOOQ exposes a lot of interfaces and hides most implementation facts from client code. The reasons for this are:

- Interface-driven design. This allows for modelling queries in a fluent API most efficiently
- Reduction of complexity for client code.
- API guarantee. You only depend on the exposed interfaces, not concrete (potentially dialect-specific) implementations.

The **org.jooq.impl.Factory** class is the main class from where you will create all jOOQ objects. The Factory implements **org.jooq.Configuration** and needs to be instanciated with the Configuration's properties:

- **org.jooq.SQLDialect** : The dialect of your database. This may be any of the currently supported database types
- **java.sql.Connection** : A JDBC Connection that will be re-used for the whole lifecycle of your Factory
- **org.jooq.conf.Settings** : An optional runtime configuration.

If you are planning on using several RDBMS (= SQLDialects) or several distinct JDBC Connections in your software, this will mean that you have to create a new Factory every time.

Factory settings

The jOOQ Factory allows for some optional configuration elements to be used by advanced users. The **Settings** class is a JAXB-annotated type. In future releases of jOOQ, these settings can be loaded from an XML file automatically. Subsequent sections of the manual contain some more in-depth explanations about these settings:

- **Runtime schema and table mapping**
- **Execute listeners and SQL tracing**

Please refer to the jOOQ runtime configuration XSD for more details:
http://www.jooq.org/xsd/jooq-runtime-2.0.5.xsd

Factory subclasses

There are a couple of subclasses for the general Factory. Each SQL dialect has its own dialect-specific factory. For instance, if you're only using the MySQL dialect, you can choose to create a new Factory using any of the following types:

```
// A general, dialect-unspecific factory
Factory create = new Factory(connection, SQLDialect.MYSQL);

// A MySQL-specific factory
MySQLFactory create = new MySQLFactory(connection);
```

jOOQ is brought to you by Lukas Eder. Distributed under the Apache 2 licence

Page 5 / 68

The advantage of using a dialect-specific Factory lies in the fact, that you have access to more proprietary RDMBS functionality. This may include:

- Oracle's **CONNECT BY** pseudo columns and functions
- MySQL's encryption functions
- PL/SQL constructs, pgplsql, or any other dialect's ROUTINE-language (maybe in the future)

Another type of Factory subclasses are each generated schema's factories. If you generate your schema TEST, then you will have access to a TestFactory. This will be useful in the future, when access to schema artefacts will be unified. Currently, this has no use.

Static Factory methods

With jOOQ 2.0, static factory methods have been introduced in order to make your code look more like SQL. Ideally, when working with jOOQ, you will simply static import all methods from the Factory class:

```
import static org.jooq.impl.Factory.*;
```

This will allow to access functions even more fluently:

```
concat(trim(FIRST_NAME), trim(LAST_NAME));
// ... which is in fact the same as:
Factory.concat(Factory.trim(FIRST_NAME), Factory.trim(LAST_NAME));
```

Objects created statically from the Factory do not need a reference to any factory, as they can be constructed independently from your Configuration (connection, dialect, schema mapping). They will access that information at render / bind time. See **more details on the QueryParts' internals**

Potential problems

The jOOQ Factory expects its underlying **java.sql.Connection** to be **open and ready** for **java.sql.PreparedStatement** creation. You are responsible yourself for the lifecycle dependency between Factory and Connection. This means:

- jOOQ will never close the Connection.
- jOOQ will never commit or rollback on the Connection (Except for CSV-imports, if explicitly configured in the **Import API**)
- jOOQ will never start any transactions.
- jOOQ does not know the concept of a session as for instance **Hibernate**
- jOOQ does not know the concept of a second-level cache. SQL is executed directly on the underlying RDBMS.
- jOOQ does not make assumptions about the origin of the Connection. If it is container managed, that is fine.

So if you want your queries to run in separate transactions, if you want to roll back a transaction, if you want to close a Connection and return it to your container, you will have to take care of that yourself. jOOQ's Factory will always expect its Connection to be in a ready state for creating new PreparedStatements. If it is not, you have to create a new Factory.
Please keep in mind that many jOOQ objects will reference your Factory for their whole lifecycle. This is especially interesting, when dealing with **Updatable Records**, that can perform CRUD operations on the Factory's underlying Connection.

1.3. Tables and Fields

Tables and their Fields are probably the most important objects in jOOQ. Tables represent any entity in your underlying RDBMS, that holds data for selection, insertion, updates, and deletion. In other words, views are also considered tables by jOOQ.

The Table

The formal definition of a **org.jooq.Table** starts with

```
public interface Table<R extends Record> // [...]
```

This means that every table is associated with a subtype of the **org.jooq.Record** class (see also **Results and Records**). For anonymous or ad-hoc tables, <R> will always bind to Record itself. Unlike in the **JPA CriteriaQuery API**, this generic type <R> is not given so much importance as far as type-safety is concerned. SQL itself is highly typesafe. You have incredible flexibility of creating anonymous or ad-hoc types and reusing them from **NESTED SELECT statements** or from many other use-cases. There is no way that this typesafety can be mapped to the Java world in a convenient way. If <R> would play a role as important as in JPA, jOOQ would suffer from the same verbosity, or inflexibility that JPA CriteriaQueries may have.

The Field

The formal definition of a Field starts with

```
public interface Field<T> // [...]
```

Fields are generically parameterised with a Java type <T> that reflects the closest match to the RDMBS's underlying datatype for that field. For instance, if you have a VARCHAR2 type Field in Oracle, <T> would bind to **java.lang.String** for that Field in jOOQ. Oracle's NUMBER(7) would let <T> bind to **java.lang.Integer**, etc. This generic type is useful for two purposes:

- It allows you to write type safe queries. For instance, you cannot compare Field <String> with Field <Integer>
- It allows you to fetch correctly cast and converted values from your database result set. This is especially useful when <T> binds to advanced data types, such as **UDT's, ARRAY or ENUM types** , where jOOQ does the difficult non-standardised JDBC data type conversions for you.

Fields and tables put into action

The Field itself is a very broad concept. Other tools, or databases refer to it as expression or column. When you just want to

```
SELECT 1 FROM DUAL
```

Then 1 is considered a Field or more explicitly, a **org.jooq.impl.Constant**, which implements Field, and DUAL is considered a Table or more explicitly **org.jooq.impl.Dual**, which implements Table More advanced uses become clear quickly, when you do things like

```
SELECT 1 + 1 FROM DUAL
```

jOOQ is brought to you by Lukas Eder. Distributed under the Apache 2 licence

Page 7 / 68

Where 1 + 1 itself is a Field or more explicitly, an **org.jooq.impl.Expression** joining two Constants together.
See some details about how to create these queries in the **Query section** of the manual

TableFields

A specific type of field is the **org.jooq.TableField**, which represents a physical Field in a physical Table. Both the TableField and its referenced Table know each other. The physical aspect of their nature is represented in jOOQ by **meta model code generation**, where every entity in your database schema will be generated into a corresponding Java class.
TableFields join both <R> and <T> generic parameters into their specification:

```
public interface TableField<R extends Record, T> // [...]
```

This can be used for additional type safety in the future, or by client code.

1.4. Results and Records

Results and their Records come into play, when SELECT statements are executed. There are various ways to fetch data from a jOOQ SELECT statement. Essentially, the query results are always provided in the Result API

The Result

The **Result**<R extends **Record**> is essentially a wrapper for a List<R extends Record> providing many convenience methods for accessing single elements in the result set. Depending on the type of SELECT statement, <R> can be bound to a sub-type of Record, for instance to an **org.jooq.UpdatableRecord**. See the section on **Updatable Records** for further details.

The Cursor

A similar object is the **Cursor**<R extends Record>. Unlike the Result, the cursor has not fetched all data from the database yet. This means, you save memory (and potentially speed), but you can only access data sequentially and you have to keep a JDBC ResultSet alive. Cursors behave very much like the **java.util.Iterator**, by providing a very simple API. Some sample methods are:

```
// Check whether there are any more records to be fetched
boolean hasNext() throws SQLException;

// Fetch the next record from the underlying JDBC ResultSet
R fetchOne() throws SQLException;

// Close the underlying JDBC ResultSet. Don't forget to call this, before disposing the Cursor.
void close() throws SQLException;
```

The Record

The Record itself holds all the data from your selected tuple. If it is a **org.jooq.TableRecord**, then it corresponds exactly to the type of one of your physical tables in your database. But any anonymous or ad-hoc tuple can be represented by the plain Record. A record mainly provides access to its data and adds convenience methods for data type conversion. These are the main access ways:

```
// If you can keep a reference of the selected field, then you can get the corresponding value type-safely
<T> T getValue(Field<T> field);

// If you know the name of the selected field within the tuple,
// then you can get its value without any type information
Object getValue(String fieldName);

// If you know the index of the selected field within the tuple,
// then you can get its value without any type information
Object getValue(int index);
```

In some cases, you will not be able to reference the selected Fields both when you create the SELECT statement and when you fetch data from Records. Then you might use field names or indexes, as with JDBC. However, of course, the type information will then be lost as well. If you know what type you want to get, you can always use the Record's convenience methods for type conversion, however. Some examples:

```
// These methods will try to convert a value to a BigDecimal.
// This will work for all numeric types and for CHAR/VARCHAR types, if they contain numeric values:
BigDecimal getValueAsBigDecimal(String fieldName);
BigDecimal getValueAsBigDecimal(int fieldIndex);

// This method can perform arbitrary conversions
<T> T getValue(String fieldName, Class<? extends T> type);
<T> T getValue(int fieldIndex, Class<? extends T> type);
```

For more information about the type conversions that are supported by jOOQ, read the Javadoc on **org.jooq.tools.Convert**

1.5. Updatable Records

UpdatableRecords are a specific subtype of TableRecord that have primary key information associated with them.

CRUD Operations

As of jOOQ 1.5, the UpdatableRecord essentially contains three additional methods **CRUD** (Create Read Update Delete) operations:

```
// Store any changes made to this record to the database.
// The record executes an INSERT if the PRIMARY KEY is NULL or has been changed. Otherwise, an UPDATE is performed.
int store();

// Deletes the record from the database.
int delete();

// Reflects changes made in the database to this Record
void refresh();
```

An example lifecycle of a book can be implemented as such:

```
// Create a new record and insert it into the database
TBookRecord book = create.newRecord(T_BOOK);
book.setTitle("My first book");
book.store();

// Update it with new values
book.setPublishedIn(2010);
book.store();

// Delete it
book.delete();
```

These operations are very simple utilities. They do not reflect the functionality offered by **Hibernate** or other persistence managers.

Performing CRUD on non-updatable records

If the jOOQ code-generator cannot detect any PRIMARY KEY, or UNIQUE KEY on your tables, then the generated artefacts implement TableRecord, instead of UpdatableRecord. A TableRecord can perform the same CRUD operations as we have seen before, if you provide it with the necessary key fields. The API looks like this:

```
// INSERT or UPDATE the record using the provided keys
int storeUsing(TableField<R, ?>... keys)

// DELETE a record using the provided keys
int deleteUsing(TableField<R, ?>... keys);

// Reflects changes made in the database to this Record
void refreshUsing(TableField<R, ?>... keys);
```

This is useful if your RDBMS does not support referential constraints (e.g. MySQL's **MyISAM**), or if you want to store records to an unconstrained view. An example lifecycle of a book without any keys can then be implemented as such:

```
// Create a new record and insert it into the database
TBookRecord book = create.newRecord(T_BOOK);
book.setTitle("My first book");
book.storeUsing(TBook.ID);

// Update it with new values
book.setPublishedIn(2010);
book.storeUsing(TBook.ID);

// Delete it
book.deleteUsing(TBook.ID);
```

1.6. The Query and its various subtypes

The Query type hierarchy is what you use to execute queries. It has the following subtypes for each kind of operation

SELECT statements

There are essentially two ways of creating SELECT statements in jOOQ. For historical reasons, you can create **org.jooq.SimpleSelectQuery** or **org.jooq.SelectQuery** objects and add additional query clauses, such as **Conditions** or **SortFields** to it. Since jOOQ 1.3, there is also the possibility to create SELECT statements using jOOQ's **DSL API** in a much more intuitive and SQL-like way.
Use the DSL API when:

- You want your code to look like SQL
- You want your IDE to help you with auto-completion (you will not be able to write select .. order by .. where .. join or any of that stuff)

Use the regular API when:

- You want to create your query step-by-step, creating query parts one-by-one
- You need to assemble your query from various places, passing the query around, adding new conditions and joins on the way

In any case, all API's will construct the same underlying implementation object, and in many cases, you can combine the two approaches. Let's check out the various SELECT statement types:

jOOQ is brought to you by Lukas Eder. Distributed under the Apache 2 licence

Page 10 / 68

- **org.jooq.Select**: This Query subtype stands for a general type of SELECT statement. It is also the main Select type for the **DSL API**. When executed, this object will hold a **Result containing the resulting Records**. This type is further subtyped for the various uses of a SELECT statement as such:
- **org.jooq.SimpleSelectQuery**: This Query will allow for selecting from single physical Tables only. It therefore has access to the Table's generic type parameter <R extends Record> and will provide a matching Result<R>. This is especially useful if <R> is a subtype of **UpdatableRecord**. Then you will be able to perform updates on your result set immediately.
- **org.jooq.SelectQuery**: This Query will allow for selecting a subset of Fields from several Tables. Because the results of such a query are considered of an anonymous or ad-hoc type, this Query will bind <R> to the general type Record itself. The purpose of this Query type is to allow for full SQL support, including SELECT, JOIN and GROUP BY clauses.

Example: SQL query and DSL query

```
-- Select all books by authors born after 1920, named
 "Paulo"
-- from a catalogue consisting of authors and books:

SELECT *
  FROM t_author
  JOIN t_book
    ON t_author.id = t_book.author_id
 WHERE t_author.year_of_birth > 1920
   AND t_author.first_name = 'Paulo'
 ORDER BY t_book.title
```

```
// Instanciate your factory using a JDBC connection.
Factory create = new Factory(connection,
 SQLDialect.ORACLE);

// Execute the query "on a single line"
Result<Record> result = create.select()
    .from(T_AUTHOR)
    .join(T_BOOK)
    .on(T_AUTHOR.ID.equal(T_BOOK.AUTHOR_ID))
    .where(T_AUTHOR.YEAR_OF_BIRTH.greaterThan(1920)
    .and(T_AUTHOR.FIRST_NAME.equal("Paulo")))
    .orderBy(T_BOOK.TITLE).fetch();
```

In the above example, some generated artefacts are used for querying. In this case, T_AUTHOR and T_BOOK are instances of types **TAuthor** and **TBook** respectively. Their full qualification would read TAuthor.T_AUTHOR and TBook.T_BOOK, but in many cases, it's useful to static import elements involved with queries, in order to decrease verbosity:

```
import static com.example.jooq.Tables.*;
```

Apart from the singleton Table instances TAuthor.T_AUTHOR and TBook.T_BOOK, these generated classes also contain one member for every physical field, such as TAuthor.ID or TBook.TAUTHOR_ID, etc. Depending on your configuration, those members can be static members (better for static imports) or instance members (better for aliasing)

- For more information about code generation, check out the manual's section about **Meta model source code generation**.
- For more DSL examples, please consider the manual's section about the **DSL API**.

Example: Non-DSL query

If you choose not to use the DSL API (for instance, because you don't want to add Query parts in the order SQL expects them), you can use this syntax:

jOOQ is brought to you by Lukas Eder. Distributed under the Apache 2 licence

Page 11 / 68

```
// Re-use the factory to create a SelectQuery. This example will not make use of static imports...
SelectQuery q = create.selectQuery();
q.addFrom(T_AUTHOR);

// This example shows some "mixed" API usage, where the JOIN is added with the standard API, and the
// Condition is created using the DSL API
q.addJoin(T_BOOK, T_AUTHOR.ID.equal(T_BOOK.AUTHOR_ID));

// The AND operator between Conditions is implicit here
q.addConditions(T_AUTHOR.YEAR_OF_BIRTH.greaterThan(1920));
q.addConditions(T_AUTHOR.FIRST_NAME.equal("Paulo"));
q.addOrderBy(T_BOOK.TITLE);
```

Fetching data

The **org.jooq.Select** interface extends **org.jooq.ResultQuery**, which provides a range of methods to fetch data from the database. Once you have constructed your SELECT query (see examples above), you may choose to either simply execute() it, or use a variety of convenience fetchXXX() methods. See the manual's **section on the ResultQuery** for more details.

INSERT Statements

jOOQ supports two modes for INSERT statements. The INSERT VALUES and the INSERT SELECT syntax

Example: SQL query and DSL query

```
INSERT INTO T_AUTHOR
    (ID, FIRST_NAME, LAST_NAME)
VALUES
    (100, 'Hermann', 'Hesse'),
    (101, 'Alfred', 'Döblin');
```

```
create.insertInto(T_AUTHOR,
        T_AUTHOR.ID, T_AUTHOR.FIRST_NAME,
    T_AUTHOR.LAST_NAME)
        .values(100, "Hermann", "Hesse")
        .values(101, "Alfred", "Döblin")
        .execute();
```

The DSL syntax tries to stay close to actual SQL. In detail, however, Java is limited in its possibilities. That's why the .values() clause is repeated for every record in multi-record inserts. Some RDBMS support inserting several records at the same time. This is also supported in jOOQ, and simulated using UNION clauses for those RDBMS that don't support this syntax.

```
INSERT INTO .. SELECT .. UNION ALL SELECT ..
```

Note: Just like in SQL itself, you can have syntax errors when you don't have matching numbers of fields/values. Also, you can run into runtime problems, if your field/value types don't match.

Example: DSL Query, alternative syntax

MySQL (and some other RDBMS) allow for using an UPDATE-like syntax for INSERT statements. This is also supported in jOOQ, should you prefer that syntax. The above INSERT statement can also be expressed as follows:

```
create.insertInto(T_AUTHOR)
      .set(T_AUTHOR.ID, 100)
      .set(T_AUTHOR.FIRST_NAME, "Hermann")
      .set(T_AUTHOR.LAST_NAME, "Hesse")
      .newRecord()
      .set(T_AUTHOR.ID, 101)
      .set(T_AUTHOR.FIRST_NAME, "Alfred")
      .set(T_AUTHOR.LAST_NAME, "Döblin")
      .execute();
```

As you can see, this syntax is a bit more verbose, but also more type-safe, as every field can be matched with its value.

jOOQ is brought to you by Lukas Eder. Distributed under the Apache 2 licence

Page 12 / 68

Example: ON DUPLICATE KEY UPDATE clause

The MySQL database supports a very convenient way to INSERT or UPDATE a record. This is a non-standard extension to the SQL syntax, which is supported by jOOQ and simulated in other RDBMS, where this is possible. Here is an example how to use the ON DUPLICATE KEY UPDATE clause:

```
// Add a new author called "Koontz" with ID 3.
// If that ID is already present, update the author's name
create.insertInto(T_AUTHOR, T_AUTHOR.ID, T_AUTHOR.LAST_NAME)
      .values(3, "Koontz")
      .onDuplicateKeyUpdate()
      .set(T_AUTHOR.LAST_NAME, "Koontz")
      .execute();
```

Example: INSERT .. RETURNING clause

The Postgres database has native support for an INSERT .. RETURNING clause. This is a very powerful concept that is simulated for all other dialects using JDBC's **getGeneratedKeys()** method. Take this example:

```
// Add another author, with a generated ID
Record<?> record =
create.insertInto(T_AUTHOR, T_AUTHOR.FIRST_NAME, T_AUTHOR.LAST_NAME)
      .values("Charlotte", "Roche")
      .returning(T_AUTHOR.ID)
      .fetchOne();

System.out.println(record.getValue(T_AUTHOR.ID));

// For some RDBMS, this also works when inserting several values
// The following should return a 2x2 table
Result<?> result =
create.insertInto(T_AUTHOR, T_AUTHOR.FIRST_NAME, T_AUTHOR.LAST_NAME)
      .values("Johann Wolfgang", "von Goethe")
      .values("Friedrich", "Schiller")
      // You can request any field. Also trigger-generated values
      .returning(T_AUTHOR.ID, T_AUTHOR.CREATION_DATE)
      .fetch();
```

Be aware though, that this can lead to race-conditions in those databases that cannot properly return generated ID values.

Example: Non-DSL Query

You can always use the more verbose regular syntax of the InsertQuery, if you need more control:

```
// Insert a new author into the T_AUTHOR table
InsertQuery<TAuthorRecord> i = create.insertQuery(T_AUTHOR);
i.addValue(T_AUTHOR.ID, 100);
i.addValue(T_AUTHOR.FIRST_NAME, "Hermann");
i.addValue(T_AUTHOR.LAST_NAME, "Hesse");

i.newRecord();
i.addValue(T_AUTHOR.ID, 101);
i.addValue(T_AUTHOR.FIRST_NAME, "Alfred");
i.addValue(T_AUTHOR.LAST_NAME, "Döblin");
i.execute();
```

Example: INSERT Query combined with SELECT statements

The InsertQuery.addValue() method is overloaded, such that you can also provide a Field, potentially containing an expression:

```
// Insert a new author into the T_AUTHOR table
InsertQuery<TAuthorRecord> i = create.insertQuery(T_AUTHOR);
i.addValue(T_AUTHOR.ID, create.select(max(T_AUTHOR.ID).add(1)).from(T_AUTHOR).asField())
i.addValue(T_AUTHOR.FIRST_NAME, "Hermann");
i.addValue(T_AUTHOR.LAST_NAME, "Hesse");
i.execute();
```

Note that especially MySQL (and some other RDBMS) has some limitations regarding that syntax. You may not be able to select from the same table you're inserting into

Example: INSERT SELECT syntax support

In some occasions, you may prefer the INSERT SELECT syntax, for instance, when you copy records from one table to another:

```
Insert i = create.insertInto(T_AUTHOR_ARCHIVE)
                 .select(create.selectFrom(T_AUTHOR).where(T_AUTHOR.DECEASED.isTrue()));
i.execute();
```

UPDATE Statements

UPDATE statements are only possible on single tables. Support for multi-table updates will be implemented in the near future.

Example: SQL query and DSL query

```
UPDATE T_AUTHOR
   SET FIRST_NAME = 'Hermann',
       LAST_NAME = 'Hesse'
 WHERE ID = 3;
```

```
create.update(T_AUTHOR)
      .set(T_AUTHOR.FIRST_NAME, "Hermann")
      .set(T_AUTHOR.LAST_NAME, "Hesse")
      .where(T_AUTHOR.ID.equal(3))
      .execute();
```

Example: Non-DSL Query

Using the **org.jooq.UpdateQuery** class, this is how you could express an UPDATE statement:

```
UpdateQuery<TAuthorRecord> u = create.updateQuery(T_AUTHOR);
u.addValue(T_AUTHOR.FIRST_NAME, "Hermann");
u.addValue(T_AUTHOR.FIRST_NAME, "Hesse");
u.addConditions(T_AUTHOR.ID.equal(3));
u.execute();
```

DELETE Statements

DELETE statements are only possible on single tables. Support for multi-table deletes will be implemented in the near future.

Example: SQL query and DSL query

```
DELETE T_AUTHOR
 WHERE ID = 100;
```

```
create.delete(T_AUTHOR)
      .where(T_AUTHOR.ID.equal(100))
      .execute();
```

Example: Non-DSL Query

Using the **org.jooq.DeleteQuery** class, this is how you could express a DELETE statement:

```
DeleteQuery<TAuthorRecord> d = create.deleteQuery(T_AUTHOR);
d.addConditions(T_AUTHOR.ID.equal(100));
d.execute();
```

MERGE Statement

The MERGE statement is one of the most advanced standardised SQL constructs, which is supported by DB2, HSQLDB, Oracle, SQL Server and Sybase (MySQL has the similar INSERT .. ON DUPLICATE KEY UPDATE construct. H2's MERGE variant is currently not supported.)
The point of the standard MERGE statement is to take a TARGET table, and merge (INSERT, UPDATE) data from a SOURCE table into it. DB2, Oracle, SQL Server and Sybase also allow for DELETING some data and for adding many additional clauses. With jOOQ 2.0.1, only Oracle's MERGE extensions are supported. Here is an example:

```
-- Check if there is already an author called 'Hitchcock'
-- If there is, rename him to John. If there isn't add him.

MERGE INTO T_AUTHOR
USING (SELECT 1 FROM DUAL)
ON (LAST_NAME = 'Hitchcock')
WHEN MATCHED THEN UPDATE SET FIRST_NAME = 'John'
WHEN NOT MATCHED THEN INSERT (LAST_NAME)
                      VALUES ('Hitchcock')
```

```
create.mergeInto(T_AUTHOR)
      .using(create().selectOne())
      .on(T_AUTHOR.LAST_NAME.equal("Hitchcock"))
      .whenMatchedThenUpdate()
      .set(T_AUTHOR.FIRST_NAME, "John")
      .whenNotMatchedThenInsert(T_AUTHOR.LAST_NAME)
      .values("Hitchcock")
      .execute();
```

TRUNCATE Statement

The syntax is trivial:

```
TRUNCATE TABLE T_AUTHOR;
```

```
create.truncate(T_AUTHOR).execute();
```

This is not supported by Ingres and SQLite. jOOQ will execute a DELETE FROM T_AUTHOR statement instead.

1.7. ResultQuery and various ways of fetching data

Various jOOQ query type extend the ResultQuery which provides many means of fetching data. In general, fetching means executing and returning some sort of result.

The ResultQuery provides many convenience methods

These methods allow for fetching a jOOQ Result or parts of it.

```
// Fetch the whole result
Result<R> fetch();

// Fetch a single field from the result
<T> List<T> fetch(Field<T> field);
    List<?> fetch(int fieldIndex);
<T> List<T> fetch(int fieldIndex, Class<? extends T> type);
    List<?> fetch(String fieldName);
<T> List<T> fetch(String fieldName, Class<? extends T> type);

// Fetch the first Record
R fetchAny();

// Fetch exactly one Record
R fetchOne();

// Fetch a single field of exactly one Record
<T> T  fetchOne(Field<T> field);
Object fetchOne(int fieldIndex);
<T> T  fetchOne(int fieldIndex, Class<? extends T> type);
Object fetchOne(String fieldName);
<T> T  fetchOne(String fieldName, Class<? extends T> type);
```

These methods transform the result into another form, if org.jooq.Result is not optimal

```
// Fetch the resulting records as Maps
List<Map<String, Object>> fetchMaps();
     Map<String, Object>  fetchOneMap();

// Fetch the result as a Map
<K>    Map<K, R> fetchMap(Field<K> key);
<K, V> Map<K, V> fetchMap(Field<K> key, Field<V> value);

// Fetch the resulting records as arrays
Object[][] fetchArrays();
Object[]   fetchOneArray();

// Fetch a single field as an array
<T>  T[] fetchArray(Field<T> field);
Object[] fetchArray(int fieldIndex);
<T>  T[] fetchArray(int fieldIndex, Class<? extends T> type);
Object[] fetchArray(String fieldName);
<T>  T[] fetchArray(String fieldName, Class<? extends T> type);
```

These methods transform the result into a user-defined form, if org.jooq.Result is not optimal

```
// Fetch the resulting records into a custom POJO
// type, which may or may not be JPA-annotated
// Use the generator's <pojos>true</pojos> and <jpaAnnotation>true</jpaAnnotation>
// configurations to generate such POJOs with jOOQ
<E> List<E> fetchInto(Class<? extends E> type);

// Fetch the resulting records into a custom
// record handler, similar to how Spring JdbcTemplate's
// RowMapper or the Ollin Framework works.
<H extends RecordHandler<R>> H fetchInto(H handler);

// These change the behaviour of fetching itself,
// especially, when not all data should be
// fetched at once
// --------------------------------------------

// Fetch a Cursor for lazy iteration
Cursor<R> fetchLazy();

// Or a JDBC ResultSet, if you prefer that
ResultSet fetchResultSet();

// Fetch data asynchronously and let client code
// decide, when the data must be available.
// This makes use of the java.util.concurrent API,
// Similar to how Avajé Ebean works.
FutureResult<R> fetchLater();
FutureResult<R> fetchLater(ExecutorService executor);
```

1.8. Bind values

Variable binding has a great impact on how you design your SQL queries. It will influence your SQL queries' security aspect as well as execution speed.

Bind values

Bind values are used in SQL / JDBC for various reasons. Among the most obvious ones are:

- Protection against SQL injection. Instead of inlining values possibly originating from user input, you bind those values to your prepared statement and let the JDBC driver / database take care of handling security aspects.
- Increased speed. Advanced databases such as Oracle can keep execution plans of similar queries in a dedicated cache to prevent hard-parsing your query again and again. In many cases, the actual value of a bind variable does not influence the execution plan, hence it can be reused. Preparing a statement will thus be faster
- On a JDBC level, you can also reuse the SQL string and prepared statement object instead of constructing it again, as you can bind new values to the prepared statement. This is currently not supported by jOOQ, though

Ways to introduce bind values with jOOQ

Bind values are omni-present in jOOQ. Whenever you create a condition, you're actually also adding a bind value:

```
// In jOOQ, "Poe" will be the bind value bound to the condition
LAST_NAME.equal("Poe");
```

The above notation is actually convenient way to explicitly create a bind value for "Poe". You could also write this, instead:

```
// The Factory allows for explicitly creating bind values
LAST_NAME.equal(Factory.val("Poe"));

// Or, when static importing Factory.val:
LAST_NAME.equal(val("Poe"))
```

Once created, bind values are part of the query's syntax tree (see **the manual's section about jOOQ's architecture** for more information about jOOQ's internals), and cannot be modified directly anymore. If you wish to reuse a query and modify bind values between subsequent query executions, you can access them again through the **org.jooq.Query** interface:

```
// Access the first bind value from a query. Indexes are counted from 1, just as with JDBC
Query query = create.select().from(T_AUTHOR).where(LAST_NAME.equal("Poe"));
Param<?> param = query.getParam("1");

// You could now modify the Query's underlying bind value:
if ("Poe".equal(param.getValue())) {
    param.setConverted("Orwell");
}
```

The **org.jooq.Param** type can also be named explicitly using the Factory's param() methods:

```
// Create a query with a named parameter. You can then use that name for accessing the parameter again
Query query1 = create.select().from(T_AUTHOR).where(LAST_NAME.equal(param("lastName", "Poe")));
Param<?> param1 = query.getParam("lastName");

// Or, keep a reference to the typed parameter in order not to lose the <T> type information:
Param<String> param2 = param("lastName", "Poe");
Query query2 = create.select().from(T_AUTHOR).where(LAST_NAME.equal(param2));

// You can now change the bind value directly on the Param reference:
param2.setValue("Orwell");
```

The **org.jooq.Query** interface also allows for setting new bind values directly, without accessing the Param type:

```
Query query1 = create.select().from(T_AUTHOR).where(LAST_NAME.equal("Poe"));
query1.bind(1, "Orwell");

// Or, with named parameters
Query query2 = create.select().from(T_AUTHOR).where(LAST_NAME.equal(param("lastName", "Poe")));
query2.bind("lastName", "Orwell");
```

NOTE: Should you wish to use jOOQ only as a query builder and execute queries with another tool, such as Spring Data instead, you can also use the Factory's renderNamedParams() method, to actually render named parameter names in generated SQL:

```
-- The named bind variable can be rendered

SELECT *
FROM T_AUTHOR
WHERE LAST_NAME = :lastName
```

```
create.renderNamedParams(
    create.select()
        .from(T_AUTHOR)
        .where(LAST_NAME.equal(
            param("lastName", "Poe"))));
```

1.9. QueryParts and the global architecture

When constructing Query objects in jOOQ, everything is considered a QueryPart. The purpose of this quickly becomes clear when checking out the QueryPart API essentials

Everything is a QueryPart

A **org.jooq.Query** and all its contained objects is a **org.jooq.QueryPart**. QueryParts essentially provide this functionality:

-       they can render SQL using the toSQL(RenderContext) method
-       they can bind variables using the bind(BindContext) method

Both of these methods are contained in jOOQ's internal API's **org.jooq.QueryPartInternal**, which is internally implemented by every QueryPart. QueryPart internals are best illustrated with an example.

Example: CompareCondition

A simple example can be provided by checking out jOOQ's internal representation of a **org.jooq.impl.CompareCondition**. It is used for any condition comparing two fields as for example the T_AUTHOR.ID = T_BOOK.AUTHOR_ID condition here:

```
-- [...]
FROM T_AUTHOR
JOIN T_BOOK ON T_AUTHOR.ID = T_BOOK.AUTHOR_ID
-- [...]
```

This is how jOOQ implements such a condition:

```
@Override
public final void bind(BindContext context) throws SQLException {
    // The CompareCondition itself does not bind any variables.
    // But the two fields involved in the condition might do so...
    context.bind(field1).bind(field2);
}

@Override
public final void toSQL(RenderContext context) {
    // The CompareCondition delegates rendering of the Fields to the Fields
    // themselves and connects them using the Condition's comparator operator:
    context.sql(field1)
           .sql(" ");

    // If the second field is null, some convenience behaviour can be
    // implemented here
    if (field2.isNullLiteral()) {
        switch (comparator) {
            case EQUALS:
                context.sql("is null");
                break;

            case NOT_EQUALS:
                context.sql("is not null");
                break;

            default:
                throw new IllegalStateException("Cannot compare null with " + comparator);
        }
    }

    // By default, also delegate the right hand side's SQL rendering to the
    // underlying field
    else {
        context.sql(comparator.toSQL())
               .sql(" ")
               .sql(field2);
    }
}
```

For more complex examples, please refer to the codebase, directly

1.10. Serializability of QueryParts and Results

Most of the jOOQ API implements the Serializable interface. This helps storing queries and partial queries in files, transferring queries or result data over TCP/IP, etc.

Attaching QueryParts

The only transient element in any jOOQ object is the **The Factory class**'s underlying **java.sql.Connection**. When you want to execute queries after de-serialisation, or when you want to store/refresh/delete **Updatable Records**, you will have to "import" or "re-attach" them to a Factory

```
// Deserialise a SELECT statement
ObjectInputStream in = new ObjectInputStream(...);
Select<?> select = (Select<?>) in.readObject();

// This will throw a DetachedException:
select.execute();

// In order to execute the above select, attach it first
Factory create = new Factory(connection, SQLDialect.ORACLE);
create.attach(select);
```

Automatically attaching QueryParts

In simple cases, you can register a ConfigurationProvider in jOOQ's ConfigurationRegistry

```
// Create your own custom ConfigurationProvider that will make
// your default Factory available to jOOQ
ConfigurationProvider provider = new CustomConfigurationProvider();

// Statically register the provider to jOOQ's ConfigurationRegistry
ConfigurationRegistry.setProvider(provider);
```

Once you have executed these steps, all subsequent deserialisations will try to access a Configuration (containing a JDBC Connection) from your ConfigurationProvider. This may be useful when

- transporting jOOQ QueryParts or Records via TCP/IP, RMI, etc (e.g. between client and server), before immediately executing queries, storing UpdatableRecords
- Using automatic mechanisms as known in **Wicket**

1.11. Extend jOOQ with custom types

Maybe jOOQ is missing functionality that you would like to see, or you can't wait for the next release... In this case, you can extend any of the following open jOOQ implementation classes

Write your own QueryPart implementations

If a SQL clause is too complex to express with jOOQ, you can extend either one of the following types for use directly in a jOOQ query:

```
public abstract class CustomField<T> extends AbstractField<T> {
  // [...]
}
public abstract class CustomCondition extends AbstractCondition {
  // [...]
}
```

These two classes are declared public and covered by integration tests. When you extend these classes, you will have to provide your own implementations for the **QueryParts**' bind(**BindContext**) and toSQL(**RenderContext**) methods:

```
// This method must produce valid SQL. If your QueryPart contains other QueryParts, you may delegate SQL code generation to them
// in the correct order, passing the render context.
//
// If context.inline() is true, you must inline all bind variables
// If context.inline() is false, you must generate ? for your bind variables
public void toSQL(RenderContext context);

// This method must bind all bind variables to a PreparedStatement. If your QueryPart contains other QueryParts, you may
 delegate
// variable binding to them in the correct order, passing the bind context.
//
// Every QueryPart must ensure, that it starts binding its variables at context.nextIndex().
public void bind(BindContext context) throws DataAccessException;
```

The above contract may be a bit tricky to understand at first. The best thing is to check out jOOQ source code and have a look at a couple of QueryParts, to see how it's done.

Plain SQL as an alternative

If you don't need integration of rather complex QueryParts into jOOQ, then you might be safer using simple **Plain SQL** functionality, where you can provide jOOQ with a simple String representation of your embedded SQL.

2. Meta model code generation

In these sections you will learn about how to configure and use jOOQ's source code generator

Overview

In the previous chapter, we have seen how to use the **Factory** in order to create **Queries** and fetch data in **Results**. The strength of jOOQ not only lies in its object-oriented **Query model**, but also in the fact that Java source code is generated from your database schema into the META data model. jOOQ follows the paradigm, that your database comes first (see also **home page**). This means that you should not modify generated source code, but only adapt entities in your database. Every change in your database is reflected in a corresponding change in your generated meta-model. Artefacts, such as tables, views, user defined types, sequences, stored procedures, packages have a corresponding artefact in Java.

2.1. Configuration and setup of the generator

jOOQ uses a simple configuration file to configure source code generation.

The deliverables

There are three binaries available with jOOQ, to be downloaded from **SourceForge** or from Maven central:

- **jOOQ.jar**
  The main library that you will include in your application to run jOOQ
- **jOOQ-meta.jar**
  The utility that you will include in your build to navigate your database schema for code generation. This can be used as a schema crawler as well.
- **jOOQ-codegen.jar**
  The utility that you will include in your build to generate your database schema

Dependencies

All of jOOQ's dependencies are "optional", i.e. you can run jOOQ without any of those libraries. For instance, jOOQ maintains an "optional" dependency on log4j and slf4j. This means, that jOOQ tries to find log4j (and /log4j.xml) or slf4j on the classpath. If they are not present, then java.util.logging.Logger is used instead.
Other optional dependencies are the JPA API, and the Oracle JDBC driver, which is needed for Oracle's advanced data types, only

Configure jOOQ's code generator

You need to tell jOOQ some things about your database connection. Here's an example of how to do it for an Oracle database

```xml
<?xml version="1.0" encoding="UTF-8" standalone="yes"?>
<configuration>
  <!-- Configure the database connection here -->
  <jdbc>
    <driver>oracle.jdbc.OracleDriver</driver>
    <url>jdbc:oracle:thin:@[your jdbc connection parameters]</url>
    <user>[your database user]</user>
    <password>[your database password]</password>
  </jdbc>

  <generator>
    <database>
      <!-- The database dialect from jooq-meta. Available dialects are
           named org.util.[database].[database]Database. Known values are:

           org.jooq.util.ase.ASEDatabase
           org.jooq.util.db2.DB2Database
           org.jooq.util.derby.DerbyDatabase
           org.jooq.util.h2.H2Database
           org.jooq.util.hsqldb.HSQLDBDatabase
           org.jooq.util.ingres.IngresDatabase
           org.jooq.util.mysql.MySQLDatabase
           org.jooq.util.oracle.OracleDatabase
           org.jooq.util.postgres.PostgresDatabase
           org.jooq.util.sqlite.SQLiteDatabaes
           org.jooq.util.sqlserver.SQLServerDatabase
           org.jooq.util.sybase.SybaseDatabase

           You can also provide your own org.jooq.util.Database implementation
           here, if your database is currently not supported -->
      <name>org.jooq.util.oracle.OracleDatabase</name>

      <!-- All elements that are generated from your schema (several Java
           regular expressions, separated by comma) Watch out for
           case-sensitivity. Depending on your database, this might be
           important! You can create case-insensitive regular expressions
           using this syntax: (?i:expr)A comma-separated list of regular
           expressions -->
      <includes>.*</includes>

      <!-- All elements that are excluded from your schema (several Java
           regular expressions, separated by comma). Excludes match before
           includes -->
      <excludes></excludes>

      <!-- The schema that is used locally as a source for meta information.
           This could be your development schema or the production schema, etc
           This cannot be combined with the schemata element. -->
      <inputSchema>[your database schema / owner / name]</inputSchema>
    </database>

    <generate>
      <!-- See advanced configuration properties -->
    </generate>

    <target>
      <!-- The destination package of your generated classes (within the
           destination directory) -->
      <packageName>[org.jooq.your.packagename]</packageName>

      <!-- The destination directory of your generated classes -->
      <directory>[/path/to/your/dir]</directory>
    </target>
  </generator>
</configuration>
```

There are also lots of advanced configuration parameters, which will be treated in the **manual's next section** Note, you can find the official XSD file at **http://www.jooq.org/xsd/jooq-codegen-2.0.4.xsd** for a formal specification

Run jOOQ code generation

Code generation works by calling this class with the above property file as argument.

```
org.jooq.util.GenerationTool /jooq-config.xml
```

Be sure that these elements are located on the classpath:

- The property file
- jooq.jar, jooq-meta.jar, jooq-codegen.jar
- The JDBC driver you configured

A command-line example (For Windows, unix/linux/etc will be similar)

- Put the property file, jooq*.jar and the JDBC driver into a directory, e.g. C:\temp\jooq
- Go to C:\temp\jooq
- Run java -cp jooq.jar;jooq-meta.jar;jooq-codegen.jar;[JDBC-driver].jar;. org.jooq.util.GenerationTool /[property file]

Note that the property file must be passed as a classpath resource

Run code generation from Eclipse

Of course, you can also run code generation from your IDE. In Eclipse, set up a project like this. Note that this example uses jOOQ's log4j support by adding log4j.xml and log4j.jar to the project classpath:
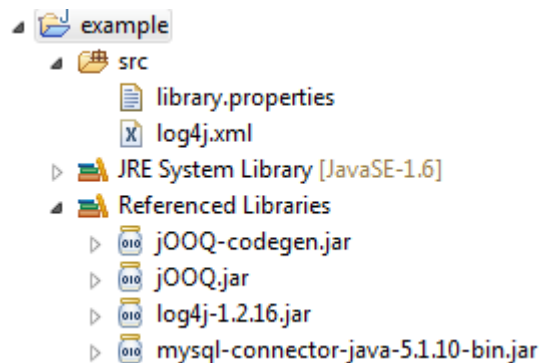
Once the project is set up correctly with all required artefacts on the classpath, you can configure an Eclipse Run Configuration for org.jooq.util.GenerationTool.
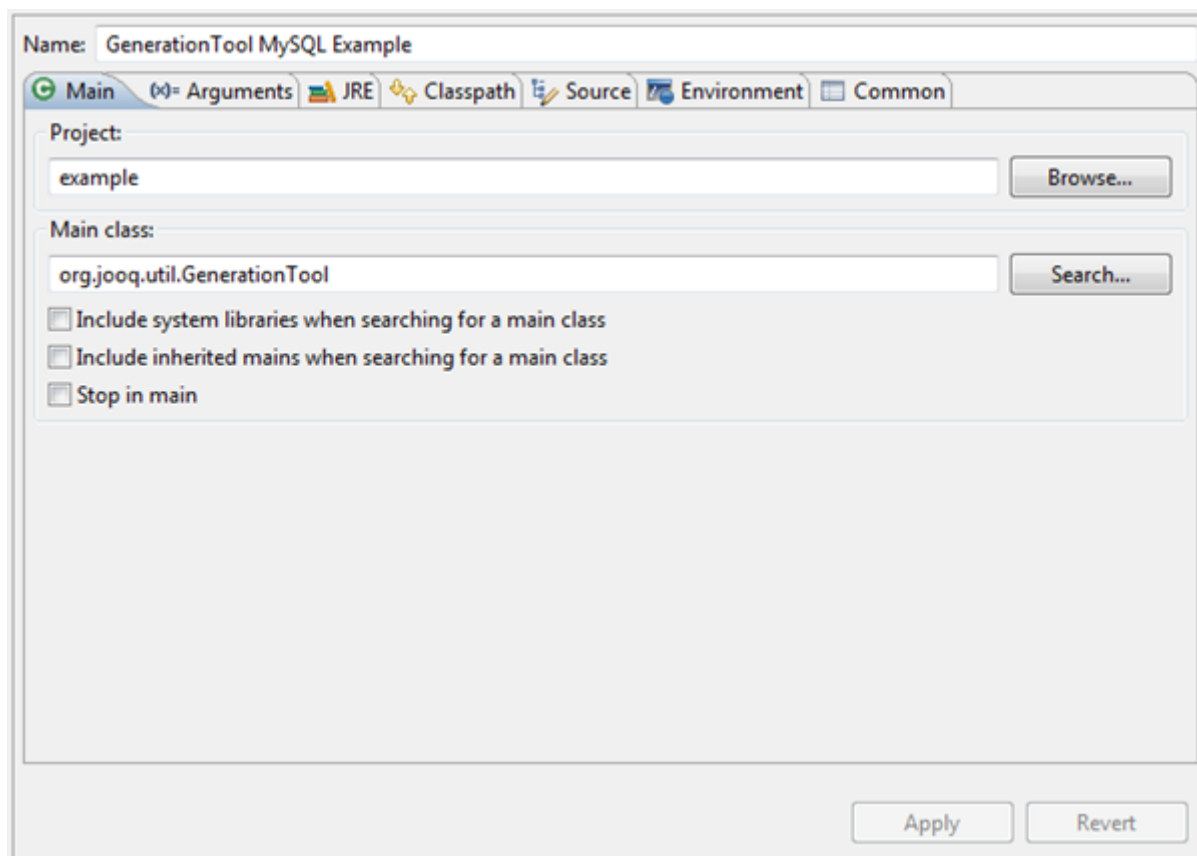
With the properties file as an argument
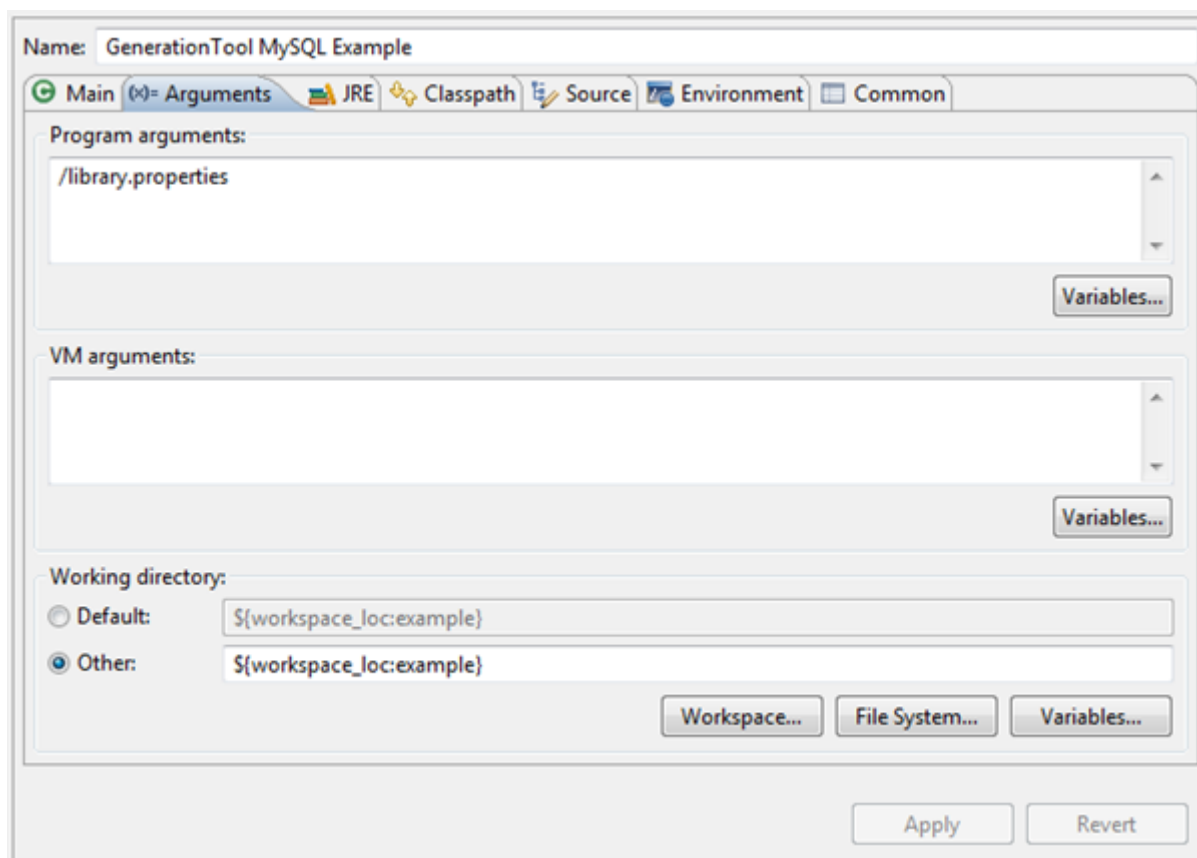
And the classpath set up correctly
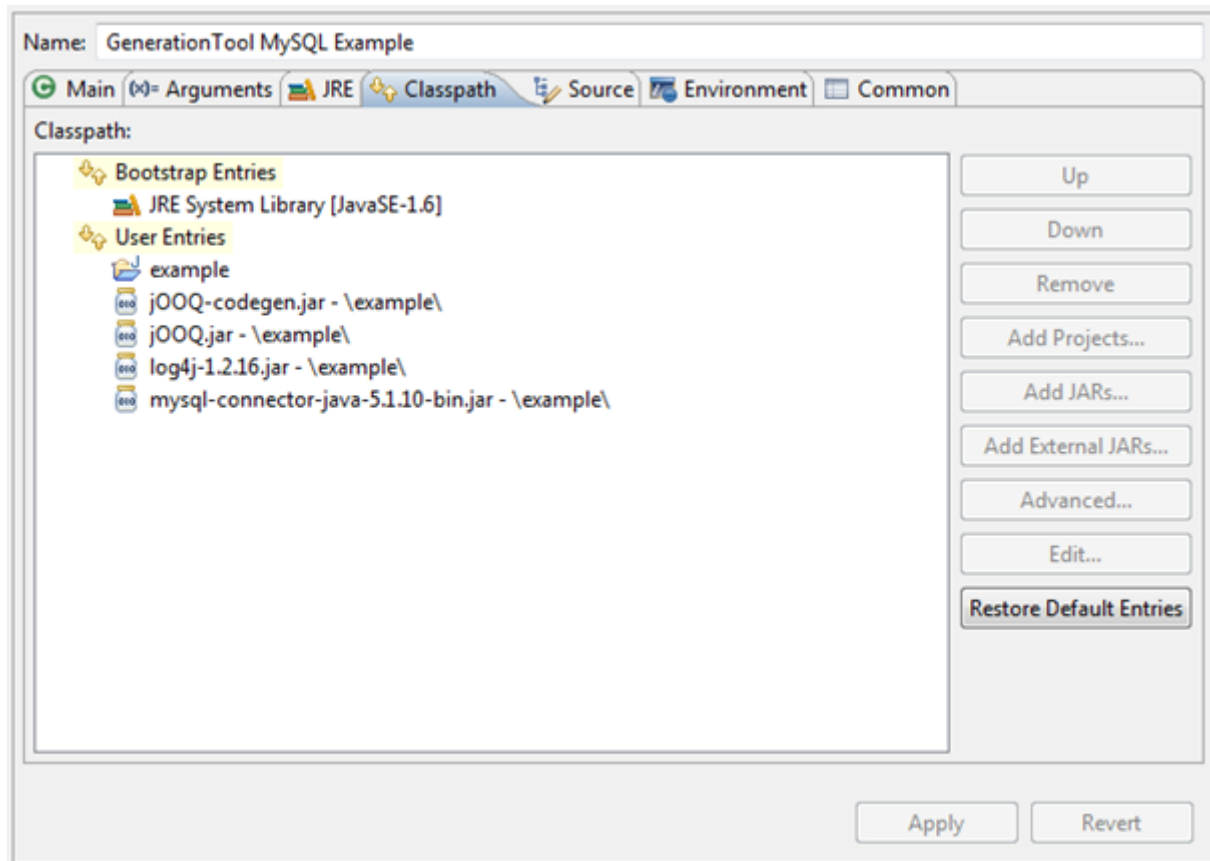
Finally, run the code generation and see your generated artefacts
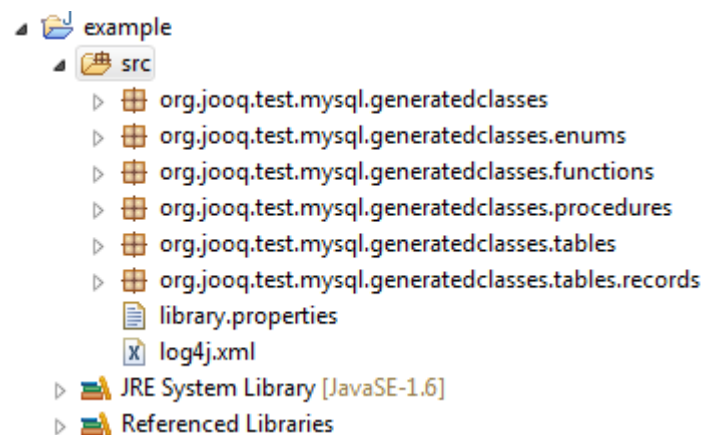
Run generation with ant

You can also use an ant task to generate your classes. As a rule of thumb, remove the dots "." and dashes "-" from the .properties file's property names to get the ant task's arguments:

```
<!-- Task definition -->
<taskdef name="generate-classes" classname="org.jooq.util.GenerationTask">
  <classpath>
    <fileset dir="${path.to.jooq.distribution}">
      <include name="jOOQ.jar"/>
      <include name="jOOQ-meta.jar"/>
      <include name="jOOQ-codegen.jar"/>
    </fileset>
    <fileset dir="${path.to.mysql.driver}">
      <include name="${mysql.driver}.jar"/>
    </fileset>
  </classpath>
</taskdef>

<!-- Run the code generation task -->
<target name="generate-test-classes">
  <generate-classes
      jdbcurl="jdbc:mysql://localhost/test"
      jdbcuser="root"
      jdbcpassword=""
      generatordatabaseinputschema="test"
      generatortargetpackage="org.jooq.test.generatedclasses"
      generatortargetdirectory="${basedir}/src"/>
</target>
```

Integrate generation with Maven

Using the official jOOQ-codegen-maven plugin, you can integrate source code generation in your Maven build process:

```
<plugin>

  <!-- Specify the maven code generator plugin -->
  <groupId>org.jooq</groupId>
  <artifactId>jooq-codegen-maven</artifactId>
  <version>1.6.7</version>

  <!-- The plugin should hook into the generate goal -->
  <executions>
    <execution>
      <goals>
        <goal>generate</goal>
      </goals>
    </execution>
  </executions>

  <!-- Manage the plugin's dependency. In this example, we'll use a Postgres database -->
  <dependencies>
    <dependency>
      <groupId>postgresql</groupId>
      <artifactId>postgresql</artifactId>
      <version>8.4-702.jdbc4</version>
    </dependency>
  </dependencies>

  <!-- Specify the plugin configuration -->
  <configuration>

    <!-- JDBC connection parameters -->
    <jdbc>
      <driver>org.postgresql.Driver</driver>
      <url>jdbc:postgresql:postgres</url>
      <user>postgres</user>
      <password>test</password>
    </jdbc>

    <!-- Generator parameters -->
    <generator>
      <name>org.jooq.util.DefaultGenerator</name>
      <database>
        <name>org.jooq.util.postgres.PostgresDatabase</name>
        <includes>.*</includes>
        <excludes></excludes>
        <inputSchema>public</inputSchema>
      </database>
      <generate>
        <relations>true</relations>
        <deprecated>false</deprecated>
      </generate>
      <target>
        <packageName>org.jooq.util.maven.example</packageName>
        <directory>target/generated-sources/jooq</directory>
      </target>
    </generator>
  </configuration>
</plugin>
```

See the full example of a pom.xml including the jOOQ-codegen artefact here: **https://github.com/lukaseder/jOOQ/blob/master/jOOQ-codegen-maven-example/pom.xml**

Migrate properties files from jOOQ 1.7, early versions of jOOQ 2.0.x:

Before jOOQ 2.0.4, the code generator was configured using properties files These files are still supported for source code generation, but their syntax won't be maintained any longer. If you wish to migrate to XML, you can migrate the file using this command on the command line

```
org.jooq.util.GenerationTool /jooq-config.properties migrate
```

Using the migrate flag, jOOQ will read the properties file and output a corresponding XML file on system out

Use jOOQ generated classes in your application

Be sure, both jOOQ.jar and your generated package (see configuration) are located on your classpath. Once this is done, you can execute SQL statements with your generated classes.

2.2. Advanced configuration of the generator

jOOQ power users may want to fine-tune their source code generation settings. Here's how to do this

Code generation

In the **previous section** we have seen how jOOQ's source code generator is configured and run within a few steps. In this chapter we'll treat some advanced settings

```xml
<!-- These properties can be added directly to the generator element: -->
<generator>
  <!-- The default code generator. You can override this one, to generate your own code style
       Defaults to org.jooq.util.DefaultGenerator -->
  <name>org.jooq.util.DefaultGenerator</name>

  <!-- The naming strategy used for class and field names.
       You may override this with your custom naming strategy. Some examples follow
       Defaults to org.jooq.util.DefaultGeneratorStrategy -->
  <strategy>
    <name>org.jooq.util.DefaultGeneratorStrategy</name>
  </strategy>
</generator>
```

The following example shows how you can override the DefaultGeneratorStrategy to render table and column names the way they are defined in the database, rather than switching them to camel case:

jOOQ is brought to you by Lukas Eder. Distributed under the Apache 2 licence

Page 27 / 68

```java
/**
 * It is recommended that you extend the DefaultGeneratorStrategy. Most of the
 * GeneratorStrategy API is already declared final. You only need to override any
 * of the following methods, for whatever generation behaviour you'd like to achieve
 *
 * Beware that most methods also receive a "Mode" object, to tell you whether a
 * TableDefinition is being rendered as a Table, Record, POJO, etc. Depending on
 * that information, you can add a suffix only for TableRecords, not for Tables
 */
public class AsInDatabaseStrategy extends DefaultGeneratorStrategy {

    /**
     * Override this to specifiy what identifiers in Java should look like.
     * This will just take the identifier as defined in the database.
     */
    @Override
    public String getJavaIdentifier(Definition definition) {
        return definition.getOutputName();
    }

    /**
     * Override these to specify what a setter in Java should look like. Setters
     * are used in TableRecords, UDTRecords, and POJOs. This example will name
     * setters "set[NAME_IN_DATABASE]"
     */
    @Override
    public String getJavaSetterName(Definition definition, Mode mode) {
        return "set" + definition.getOutputName();
    }

    /**
     * Just like setters...
     */
    @Override
    public String getJavaGetterName(Definition definition, Mode mode) {
        return "get" + definition.getOutputName();
    }

    /**
     * Override this method to define what a Java method generated from a database
     * Definition should look like. This is used mostly for convenience methods
     * when calling stored procedures and functions. This example shows how to
     * set a prefix to a CamelCase version of your procedure
     */
    @Override
    public String getJavaMethodName(Definition definition, Mode mode) {
        return "call" + org.jooq.tools.StringUtils.toCamelCase(definition.getOutputName());
    }

    /**
     * Override this method to define how your Java classes and Java files should
     * be named. This example applies no custom setting and uses CamelCase versions
     * instead
     */
    @Override
    public String getJavaClassName(Definition definition, Mode mode) {
        return super.getJavaClassName(definition, mode);
    }

    /**
     * Override this method to re-define the package names of your generated
     * artefacts.
     */
    @Override
    public String getJavaPackageName(Definition definition, Mode mode) {
        return super.getJavaPackageName(definition, mode);
    }

    /**
     * Override this method to define how Java members should be named. This is
     * used for POJOs and method arguments
     */
    @Override
    public String getJavaMemberName(Definition definition, Mode mode) {
        return definition.getOutputName();
    }
}
```

Within the <generator/> element, there are other configuration elements:

jOOQ is brought to you by Lukas Eder. Distributed under the Apache 2 licence

Page 28 / 68

```
<!-- These properties can be added to the database element: -->
<database>
  <!-- Generate java.sql.Timestamp fields for DATE columns. This is
       particularly useful for Oracle databases.
       Defaults to false -->
  <dateAsTimestamp>false</dateAsTimestamp>

  <!-- Generate jOOU data types for your unsigned data types, which are
       not natively supported in Java.
       Defaults to true -->
  <unsignedTypes>true</unsignedTypes>

  <!-- The schema that is used in generated source code. This will be the
       production schema. Use this to override your local development
       schema name for source code generation. If not specified, this
       will be the same as the input-schema. -->
  <outputSchema>[your database schema / owner / name]</outputSchema>

  <!-- A configuration element to configure several input and/or output
       schemata for jooq-meta, in case you're using jooq-meta in a multi-
       schema environment.
       This cannot be combined with the above inputSchema / outputSchema -->
  <schemata>
    <schema>
      <inputSchema>...</inputSchema>
      <outputSchema>...</outputSchema>
    </schema>
    [ <schema>...</schema> ... ]
  </schemata>

  <!-- A configuration element to configure master data table enum classes -->
  <masterDataTables>...</masterDataTables>

  <!-- A configuration element to configure synthetic enum types
       This is EXPERIMENTAL functionality. Use at your own risk -->
  <enumTypes>...</enumTypes>

  <!-- A configuration element to configure type overrides for generated
       artefacts (e.g. in combination with enumTypes)
       This is EXPERIMENTAL functionality. Use at your own risk -->
  <forcedTypes>...</forcedTypes>
</database>
```

Also, you can add some optional advanced configuration parameters for the generator:

```
<!-- These properties can be added to the generate element: -->
<generate>
  <!-- Primary key / foreign key relations should be generated and used.
       This is a prerequisite for various advanced features.
       Defaults to false -->
  <relations>false</relations>

  <!-- Generate navigation methods to navigate foreign key relationships
       directly from Record classes. This is only relevant if relations
       is set to true, too.
       Defaults to true -->
  <navigationMethods>true</navigationMethods>

  <!-- Generate deprecated code for backwards compatibility
       Defaults to true -->
  <deprecated>true</deprecated>

  <!-- Generate instance fields in your tables, as opposed to static
       fields. This simplifies aliasing.
       Defaults to true -->
  <instanceFields>true</instanceFields>

  <!-- Generate the javax.annotation.Generated annotation to indicate
       jOOQ version used for source code.
       Defaults to true -->
  <generatedAnnotation>true</generatedAnnotation>

  <!-- Generate POJOs in addition to Record classes for usage of the
       ResultQuery.fetchInto(Class) API
       Defaults to false -->
  <pojos>false</pojos>

  <!-- Annotate POJOs and Records with JPA annotations for increased
       compatibility and better integration with JPA/Hibernate, etc
       Defaults to false -->
  <jpaAnnotations>false</jpaAnnotations>
</generate>
```

Check out the manual's section about **master data** to find out more about those advanced configuration parameters.

2.3. The schema, top-level generated artefact

The schema is the top-level generated object in jOOQ. In many RDBMS, the schema coincides with the owner of tables and other objects

The Schema

As of jOOQ 1.5, the top-level generated object is the **org.jooq.Schema**. The Schema itself has no relevant functionality, except for holding the schema name for all dependent generated artefacts. jOOQ queries try to always fully qualify an entity within the database using that Schema Currently, it is not possible to link generated artefacts from various schemata. If you have a stored function from Schema A, which returns a UDT from Schema B, the types cannot be linked. This enhancement is on the roadmap, though: **#282**.
When you have several schemata that are logically equivalent (i.e. they contain identical entities, but the schemata stand for different users/customers/clients, etc), there is a solution for that. Check out the manual's section on support for **multiple equivalent schemata**

Schema contents

The schema can be used to dynamically discover generate database artefacts. Tables, sequences, and other items are accessible from the schema. For example:

```
public final java.util.List<org.jooq.Sequence<?>> getSequences();
public final java.util.List<org.jooq.Table<?>> getTables();
```

2.4. Tables, views and their corresponding records

The most important generated artefacts are Tables and TableRecords. Every Table has a Record type associated with it that models a single tuple of that entity: Table<R extends Record>.

Tables and TableRecords

The most important generated artefacts are **Tables** and **TableRecords**. As discussed in previous chapters about **Tables** and **Results**, jOOQ uses the Table class to model entities (both tables and views) in your database Schema. Every Table has a Record type associated with it that models a single tuple of that entity: Table<R extends Record>. This couple of Table<R> and R are generated as such:
Suppose we have the tables as defined in the **example database**. Then, using a default configuration, these (simplified for the example) classes will be generated:

The Table as an entity meta model

```
public class TAuthor extends UpdatableTableImpl<TAuthorRecord> {

    // The singleton instance of the Table
    public static final TAuthor T_AUTHOR = new TAuthor();

    // The Table's fields.
    // Depending on your jooq-codegen configuraiton, they can also be static
    public final TableField<TAuthorRecord, Integer> ID =         // [...]
    public final TableField<TAuthorRecord, String> FIRST_NAME =   // [...]
    public final TableField<TAuthorRecord, String> LAST_NAME =    // [...]
    public final TableField<TAuthorRecord, Date> DATE_OF_BIRTH =  // [...]
    public final TableField<TAuthorRecord, Integer> YEAR_OF_BIRTH = // [...]

    // When you don't choose the static meta model, you can typesafely alias your tables.
    // Aliased tables will then hold references to the above final fields, too
    public TAuthor as(String alias) {
      // [...]
    }
}
```

The Table's associated TableRecord

If you use the **SimpleSelectQuery** syntax (both in standard and DSL mode), then your SELECT statement will return the single Table<R extends Record>'s associated Record type <R>. In the case of the above TAuthor Table, this will be a TAuthorRecord.

```
public class TAuthorRecord extends UpdatableRecordImpl<TAuthorRecord> {

    // Getters and setters for the various fields
    public void setId(Integer value) {        // [...]
    public Integer getId() {                   // [...]
    public void setFirstName(String value) { // [...]
    public String getFirstName() {            // [...]
    public void setLastName(String value) {  // [...]
    public String getLastName() {             // [...]
    public void setDateOfBirth(Date value) { // [...]
    public Date getDateOfBirth() {            // [...]

    // Navigation methods for foreign keys
    public List<TBookRecord> fetchTBooks() { // [...]
}
```

2.5. Procedures and packages

Procedure support is one of the most important reasons why you should consider jOOQ. jOOQ heavily facilitates the use of stored procedures and functions via its source code generation.

Stored procedures in modern RDBMS

This is one of the most important reasons why you should consider jOOQ. Read also my **article on dzone** about why stored procedures become more and more important in future versions of RDMBS. In this section of the manual, we will learn how jOOQ handles stored procedures in code generation. Especially before **UDT and ARRAY support** was introduced to major RDBMS, these procedures tend to have dozens of parameters, with IN, OUT, IN OUT parameters mixed in all variations. JDBC only knows very basic, low-level support for those constructs. jOOQ heavily facilitates the use of stored procedures and functions via its source code generation. Essentially, it comes down to this:

"Standalone" stored procedures and functions

Let's say you have these stored procedures and functions in your Oracle database

```
-- Check whether there is an author in T_AUTHOR by that name
CREATE OR REPLACE FUNCTION f_author_exists (author_name VARCHAR2) RETURN NUMBER;

-- Check whether there is an author in T_AUTHOR by that name
CREATE OR REPLACE PROCEDURE p_author_exists (author_name VARCHAR2, result OUT NUMBER);

-- Check whether there is an author in T_AUTHOR by that name and get his ID
CREATE OR REPLACE PROCEDURE p_author_exists_2 (author_name VARCHAR2, result OUT NUMBER, id OUT NUMBER);
```

jOOQ will essentially generate two artefacts for every procedure/function:

- A class holding a formal Java representation of the procedure/function
- Some convenience methods to facilitate calling that procedure/function

Let's see what these things look like, in Java. The classes (simplified for the example):

```java
// The function has a generic type parameter <T> bound to its return value
public class FAuthorExists extends org.jooq.impl.AbstractRoutine<BigDecimal> {

    // Much like Tables, functions have static parameter definitions
    public static final Parameter<String> AUTHOR_NAME = // [...]

    // And much like TableRecords, they have setters for their parameters
    public void setAuthorName(String value) { // [...]
    public void setAuthorName(Field<String> value) { // [...]
}

public class PAuthorExists extends org.jooq.impl.AbstractRoutine<java.lang.Void> {

    // In procedures, IN, OUT, IN OUT parameters are all represented
    // as static parameter definitions as well
    public static final Parameter<String> AUTHOR_NAME = // [...]
    public static final Parameter<BigDecimal> RESULT = // [...]

    // IN and IN OUT parameters have generated setters
    public void setAuthorName(String value) { // [...]

    // OUT and IN OUT parameters have generated getters
    public BigDecimal getResult() { // [...]
}

public class PAuthorExists_2 extends org.jooq.impl.AbstractRoutine<java.lang.Void> {
    public static final Parameter<String> AUTHOR_NAME = // [...]
    public static final Parameter<BigDecimal> RESULT = // [...]
    public static final Parameter<BigDecimal> ID = // [...]

    // the setters...
    public void setAuthorName(String value) { // [...]

    // the getters...
    public BigDecimal getResult() { // [...]
    public BigDecimal getId() { // [...]
}
```

An example invocation of such a stored procedure might look like this:

```java
PAuthorExists p = new PAuthorExists();
p.setAuthorName("Paulo");
p.execute(configuration);
assertEquals(BigDecimal.ONE, p.getResult());
```

The above configuration is a **Factory**, holding a reference to a JDBC connection, as discussed in a previous section. If you use the generated convenience methods, however, things are much simpler, still:

jOOQ is brought to you by Lukas Eder. Distributed under the Apache 2 licence

Page 32 / 68

```
// Every schema has a single Routines class with convenience methods
public final class Routines {

    // Convenience method to directly call the stored function
    public static BigDecimal fAuthorExists(Configuration configuration, String authorName) { // [...]

    // Convenience methods to transform the stored function into a
    // Field<BigDecimal>, such that it can be used in SQL
    public static Field<BigDecimal> fAuthorExists(Field<String> authorName) { // [...]
    public static Field<BigDecimal> fAuthorExists(String authorName) { // [...]

    // Procedures with 0 OUT parameters create void methods
    // Procedures with 1 OUT parameter create methods as such:
    public static BigDecimal pAuthorExists(Configuration configuration, String authorName) { // [...]

    // Procedures with more than 1 OUT parameter return the procedure
    // object (see above example)
    public static PAuthorExists_2 pAuthorExists_2(Configuration configuration, String authorName) { // [...]
}
```

An sample invocation, equivalent to the previous example:

```
assertEquals(BigDecimal.ONE, Procedures.pAuthorExists(configuration, "Paulo"));
```

jOOQ's understanding of procedures vs functions

jOOQ does not formally distinguish procedures from functions. jOOQ only knows about routines, which can have return values and/or OUT parameters. This is the best option to handle the variety of stored procedure / function support across the various supported RDBMS. For more details, read on about this topic, here:
lukaseder.wordpress.com/2011/10/17/what-are-procedures-and-functions-after-all/

Packages in Oracle

Oracle uses the concept of a PACKAGE to group several procedures/functions into a sort of namespace. The **SQL standard** talks about "modules", to represent this concept, even if this is rarely implemented. This is reflected in jOOQ by the use of Java sub-packages in the source code generation destination package. Every Oracle package will be reflected by

- A Java package holding classes for formal Java representations of the procedure/function in that package
- A Java class holding convenience methods to facilitate calling those procedures/functions

Apart from this, the generated source code looks exactly like the one for standalone procedures/functions.

Member functions and procedures in Oracle

Oracle UDT's can have object-oriented structures including member functions and procedures. With Oracle, you can do things like this:

```
CREATE OR REPLACE TYPE u_author_type AS OBJECT (
  id NUMBER(7),
  first_name VARCHAR2(50),
  last_name VARCHAR2(50),

  MEMBER PROCEDURE LOAD,
  MEMBER FUNCTION count_books RETURN NUMBER
)

-- The type body is omitted for the example
```

These member functions and procedures can simply be mapped to Java methods:

```
// Create an empty, attached UDT record from the Factory
UAuthorType author = create.newRecord(U_AUTHOR_TYPE);

// Set the author ID and load the record using the LOAD procedure
author.setId(1);
author.load();

// The record is now updated with the LOAD implementation's content
assertNotNull(author.getFirstName());
assertNotNull(author.getLastName());
```

For more details about UDT's see the Manual's section on **User Defined Types**

2.6. UDT's including ARRAY and ENUM types

Databases become more powerful when you can structure your data in user defined types. It's time for Java developers to give some credit to that.

Increased RDBMS support for UDT's

In recent years, most RDBMS have started to implement some support for advanced data types. This support has not been adopted very well by database users in the Java world, for several reasons:

- They are usually orthogonal to relational concepts. It is not easy to modify a UDT once it is referenced by a table column.
- There is little standard support of accessing them from JDBC (and probably other database connectivity standards).

On the other hand, especially with stored procedures, these data types are likely to become more and more useful in the future. If you have a look at Postgres' capabilities of dealing with advanced data types (**ENUMs**, **ARRAYs**, **UDT's**), this becomes more and more obvious.
It is a central strategy for jOOQ, to standardise access to these kinds of types (as well as to **stored procedures**, of course) across all RDBMS, where these types are supported.

UDT types

User Defined Types (UDT) are helpful in major RDMBS with lots of proprietary functionality. The biggest player is clearly Oracle. Currently, jOOQ provides UDT support for only two databases:

- Oracle
- Postgres

Apart from that,

- DB2 UDT's are not supported as they are very tough to serialise/deserialise. We don't think that this is a big enough requirement to put more effort in those, right now (see also the developers' discussion on **#164**)

In Oracle, you would define UDTs like this:

```
CREATE TYPE u_street_type AS OBJECT (
  street VARCHAR2(100),
  no VARCHAR2(30)
)

CREATE TYPE u_address_type AS OBJECT (
  street u_street_type,
  zip VARCHAR2(50),
  city VARCHAR2(50),
  country VARCHAR2(50),
  since DATE,
  code NUMBER(7)
)
```

These types could then be used in tables and/or stored procedures like such:

```
CREATE TABLE t_author (
  id NUMBER(7) NOT NULL PRIMARY KEY,
  -- [...]
  address u_address_type
)

CREATE OR REPLACE PROCEDURE p_check_address (address IN OUT u_address_type);
```

Standard JDBC UDT support encourages JDBC-driver developers to implement interfaces such as **java.sql.SQLData**, **java.sql.SQLInput** and **java.sql.SQLOutput**. Those interfaces are non-trivial to implement, or to hook into. Also access to **java.sql.Struct** is not really simple. Due to the lack of a well-defined JDBC standard, Oracle's JDBC driver rolls their own proprietary methods of dealing with these types. jOOQ goes a different way, it hides those facts from you entirely. With jOOQ, the above UDT's will be generated in simple **UDT meta-model classes** and **UDT record classes** as such:

```
// There is an analogy between UDT/Table and UDTRecord/TableRecord...
public class UAddressType extends UDTImpl<UAddressTypeRecord> {

    // The UDT meta-model singleton instance
    public static final UAddressType U_ADDRESS_TYPE = new UAddressType();

    // UDT attributes are modeled as static members. Nested UDT's
    // behave similarly
    public static final UDTField<UAddressTypeRecord, UStreetTypeRecord> STREET = // [...]
    public static final UDTField<UAddressTypeRecord, String> ZIP =              // [...]
    public static final UDTField<UAddressTypeRecord, String> CITY =             // [...]
    public static final UDTField<UAddressTypeRecord, String> COUNTRY =          // [...]
    public static final UDTField<UAddressTypeRecord, Date> SINCE =              // [...]
    public static final UDTField<UAddressTypeRecord, Integer> CODE =            // [...]
}
```

Now, when you interact with entities or procedures that hold UDT's, that's very simple as well. Here is an example:

```
// Fetch any author from the T_AUTHOR table
TAuthorRecord author = create.selectFrom(T_AUTHOR).fetchAny();

// Print out the author's address's house number
System.out.println(author.getAddress().getStreet().getNo());
```

A similar thing can be achieved when interacting with the example stored procedure:

```
// Create a new UDTRecord of type U_ADDRESS_TYPE
UAddressTypeRecord address = new UAddressTypeRecord();
address.setCountry("Switzerland");

// Call the stored procedure with IN OUT parameter of type U_ADDRESS_TYPE
address = Procedures.pCheckAddress(connection, address);
```

ARRAY types

The notion of ARRAY types in RDBMS is not standardised at all. Very modern databases (especially the Java-based ones) have implemented ARRAY types exactly as what they are. "ARRAYs of something". In other words, an ARRAY OF VARCHAR would be something very similar to Java's notion of String[]. An ARRAY OF ARRAY OF VARCHAR would then be a String[][] in Java. Some

jOOQ is brought to you by Lukas Eder. Distributed under the Apache 2 licence

Page 35 / 68

RDMBS, however, enforce stronger typing and need the explicit creation of types for every ARRAY as well. These are example String[] ARRAY types in various SQL dialects supported by jOOQ 1.5.4:

- Oracle: VARRAY OF VARCHAR2. A strongly typed object encapsulating an ARRAY of a given type. See the **documentation.**
- Postgres: text[]. Any data type can be turned into an array by suffixing it with []. See the **documentation**
- HSQLDB: VARCHAR ARRAY. Any data type can be turned into an array by suffixing it with ARRAY. See the **documentation**
- H2: ARRAY. H2 does not know of typed arrays. All ARRAYs are mapped to Object[]. See the **documentation**

Soon to be supported:

- DB2: Knows a similar strongly-typed ARRAY type, like Oracle

From jOOQ's perspective, the ARRAY types fit in just like any other type wherever the <T> generic type parameter is existent. It integrates well with tables and stored procedures.

Example: General ARRAY types

An example usage of ARRAYs is given here for the Postgres dialect

```
CREATE TABLE t_arrays (
  id integer not null primary key,
  string_array VARCHAR(20)[],
  number_array INTEGER[]
)

CREATE FUNCTION f_arrays(in_array IN text[]) RETURNS text[]
```

When generating source code from the above entities, these artefacts will be created in Java:

```
public class TArrays extends UpdatableTableImpl<TArraysRecord> {

    // The generic type parameter <T> is bound to an array of a matching type
    public static final TableField<TArraysRecord, String[]> STRING_ARRAY =  // [...]
    public static final TableField<TArraysRecord, Integer[]> NUMBER_ARRAY = // [...]
}

// The convenience class is enhanced with these methods
public final class Functions {
    public static String[] fArrays(Connection connection, String[] inArray) { // [...]
    public static Field<String[]> fArrays(String[] inArray) {                         // [...]
    public static Field<String[]> fArrays(Field<String[]> inArray) {                  // [...]
}
```

Example: Oracle VARRAY types

In Oracle, a VARRAY type is something slightly different than in other RDMBS. It is a type that encapsules the actual ARRAY and creates a new type from it. While all text[] types are equal and thus compatible in Postgres, this does not apply for all VARRAY OF VARCHAR2 types. Hence, it is important to provide access to VARRAY types and generated objects from those types as well. The example above would read like this in Oracle:

```
CREATE TYPE u_string_array AS VARRAY(4) OF VARCHAR2(20)
CREATE TYPE u_number_array AS VARRAY(4) OF NUMBER(7)

CREATE TABLE t_arrays (
  id NUMBER(7) not null primary key,
  string_array u_string_array,
  number_array u_number_array
)

CREATE OR REPLACE FUNCTION f_arrays (in_array u_string_array)
RETURN u_string_array
```

Note that it becomes clear immediately, that a mapping from U_STRING_ARRAY to String[] is obvious. But a mapping from String[] to U_STRING_ARRAY is not. These are the generated **org.jooq.ArrayRecord** and other artefacts in Oracle:

```
public class UStringArrayRecord extends ArrayRecordImpl<String> {  // [...]
public class UNumberArrayRecord extends ArrayRecordImpl<Integer> { // [...]

public class TArrays extends UpdatableTableImpl<TArraysRecord> {
    public static final TableField<TArraysRecord, UStringArrayRecord> STRING_ARRAY = // [...]
    public static final TableField<TArraysRecord, UNumberArrayRecord> NUMBER_ARRAY = // [...]
}

public final class Functions {
    public static UStringArrayRecord fArrays3(Connection connection, UStringArrayRecord inArray) { // [...]
    public static Field<UStringArrayRecord> fArrays3(UStringArrayRecord inArray) {           // [...]
    public static Field<UStringArrayRecord> fArrays3(Field<UStringArrayRecord> inArray) {         // [...]
}
```

ENUM types

True ENUM types are a rare species in the RDBMS world. Currently, MySQL and Postgres are the only RDMBS supported by jOOQ, that provide ENUM types.

- In MySQL, an ENUM type is declared directly upon a column. It cannot be reused as a type. See the **documentation.**
- In Postgres, the ENUM type is declared independently and can be reused among tables, functions, etc. See the **documentation.**
- Other RDMBS know about "ENUM constraints", such as the Oracle CHECK constraint. These are not true ENUMS, however. jOOQ refrains from using their information for source code generation

Some examples:

```
-- An example enum type
CREATE TYPE u_book_status AS ENUM ('SOLD OUT', 'ON STOCK', 'ORDERED')

-- An example useage of that enum type
CREATE TABLE t_book (
  id INTEGER NOT NULL PRIMARY KEY,

  -- [...]
  status u_book_status
)
```

The above Postgres ENUM type will be generated as

```
public enum UBookStatus implements EnumType {
    ORDERED("ORDERED"),
    ON_STOCK("ON STOCK"),
    SOLD_OUT("SOLD OUT");

    // [...]
}
```

Intuitively, the generated classes for the T_BOOK table in Postgres would look like this:

```
// The meta-model class
public class TBook extends UpdatableTableImpl<TBookRecord> {

    // The TableField STATUS binds <T> to UBookStatus
    public static final TableField<TBookRecord, UBookStatus> STATUS = // [...]

    // [...]
}

// The record class
public class TBookRecord extends UpdatableRecordImpl<TBookRecord> {

    // Corresponding to the Table meta-model, also setters and getters
    // deal with the generated UBookStatus
    public void setStatus(UBookStatus value) { // [...]
    public UBookStatus getStatus() {           // [...]
}
```

Note that jOOQ allows you to simulate ENUM types where this makes sense in your data model. See the section on **master data** for more details.

2.7. Sequences

jOOQ also generates convenience artefacts for sequences, where this is supported: DB2, Derby, H2, HSQLDB, Oracle, Postgres, and more.

Sequences as a source for identity values

Sequences implement the **org.jooq.Sequence** interface, providing essentially this functionality:

```
// Get a field for the CURRVAL sequence property
Field<T> currval();

// Get a field for the NEXTVAL sequence property
Field<T> nextval();
```

So if you have a sequence like this in Oracle:

```
CREATE SEQUENCE s_author_id
```

This is what jOOQ will generate:

```
public final class Sequences {

    // A static sequence instance
    public static final Sequence<BigInteger> S_AUTHOR_ID = // [...]
}
```

Which you can use in a select statement as such:

```
Field<BigInteger> s = Sequences.S_AUTHOR_ID.nextval();
BigInteger nextID   = create.select(s).fetchOne(s);
```

Or directly fetch currval() and nextval() from the sequence using the Factory:

```
BigInteger currval = create.currval(Sequences.S_AUTHOR_ID);
BigInteger nextval = create.nextval(Sequences.S_AUTHOR_ID);
```

3. DSL or fluent API. Where SQL meets Java

In these sections you will learn about how jOOQ makes SQL available to Java as if Java natively supported SQL

Overview

jOOQ ships with its own DSL (or **Domain Specific Language**) that simulates SQL as good as possible in Java. This means, that you can write SQL statements almost as if Java natively supported that syntax just like .NET's C# does with **LINQ to SQL.**
Here is an example to show you what that means. When you want to write a query like this in SQL:

```
-- Select all books by authors born after 1920,
-- named "Paulo" from a catalogue:
SELECT *
  FROM t_author a
  JOIN t_book b ON a.id = b.author_id
 WHERE a.year_of_birth > 1920
   AND a.first_name = 'Paulo'
 ORDER BY b.title
```

```
Result<Record> result =
create.select()
      .from(T_AUTHOR.as("a"))
      .join(T_BOOK.as("b")).on(a.ID.equal(b.AUTHOR_ID))
      .where(a.YEAR_OF_BIRTH.greaterThan(1920))
      .and(a.FIRST_NAME.equal("Paulo")))
      .orderBy(b.TITLE)
      .fetch();
```

You couldn't come much closer to SQL itself in Java, without re-writing the compiler. We'll see how the aliasing works later in the section about **aliasing**

3.1. Complete SELECT syntax

A SELECT statement is more than just the R in CRUD. It allows for transforming your relational data into any other form using concepts such as equi-join, semi-join, anti-join, outer-join and much more. jOOQ helps you think in precisely those relational concepts.

SELECT from anonymous or ad-hoc types

When you don't just perform CRUD (i.e. SELECT * FROM your_table WHERE ID = ?), you're usually generating new types using custom projections. With jOOQ, this is as intuitive, as if using SQL directly. A more or less complete example of the "standard" SQL syntax, plus some extensions, is provided by a query like this:

```
-- get all authors' first and last names, and the number
-- of books they've written in German, if they have written
-- more than five books in German in the last three years
-- (from 2011), and sort those authors by last names
-- limiting results to the second and third row, locking
-- the rows for a subsequent update... whew!

  SELECT T_AUTHOR.FIRST_NAME, T_AUTHOR.LAST_NAME, COUNT(*)
    FROM T_AUTHOR
    JOIN T_BOOK ON T_AUTHOR.ID = T_BOOK.AUTHOR_ID
   WHERE T_BOOK.LANGUAGE = 'DE'
     AND T_BOOK.PUBLISHED > '2008-01-01'
GROUP BY T_AUTHOR.FIRST_NAME, T_AUTHOR.LAST_NAME
  HAVING COUNT(*) > 5
ORDER BY T_AUTHOR.LAST_NAME ASC NULLS FIRST
   LIMIT 2
  OFFSET 1
     FOR UPDATE
```

So that's daily business. How to do it with jOOQ: When you first create a SELECT statement using the Factory's select() methods

```
SelectFromStep select(Field<?>... fields);

// Example:
create.select(T_AUTHOR.FIRST_NAME, T_AUTHOR.LAST_NAME, count());
```

jOOQ will return an "intermediary" type to you, representing the SELECT statement about to be created (by the way, check out the section on **aggregate operators** to learn more about the

COUNT(*) function). This type is the **org.jooq.SelectFromStep**. When you have a reference to this type, you may add a FROM clause, although that clause is optional. This is reflected by the fact, that the SelectFromStep type extends **org.jooq.SelectJoinStep**, which allows for adding the subsequent clauses. Let's say you do decide to add a FROM clause, then you can use this method for instance:

```
SelectJoinStep from(TableLike<?>... table);

// The example, continued:
create.select(T_AUTHOR.FIRST_NAME, T_AUTHOR.LAST_NAME, count())
      .from(T_AUTHOR);
```

After adding the table-like structures (mostly just Tables) to select from, you may optionally choose to add a JOIN clause, as the type returned by jOOQ is the step where you can add JOINs. Again, adding these clauses is optional, as the **org.jooq.SelectJoinStep** extends **org.jooq.SelectWhereStep**. But let's say we add a JOIN:

```
// These join types are supported
SelectOnStep                 join(Table<?> table);
SelectOnStep        leftOuterJoin(Table<?> table);
SelectOnStep       rightOuterJoin(Table<?> table);
SelectOnStep        fullOuterJoin(Table<?> table);
SelectJoinStep          crossJoin(Table<?> table);
SelectJoinStep        naturalJoin(Table<?> table);
SelectJoinStep  naturalLeftOuterJoin(Table<?> table);
SelectJoinStep naturalRightOuterJoin(Table<?> table);

// The example, continued:
create.select(T_AUTHOR.FIRST_NAME, T_AUTHOR.LAST_NAME, count())
      .from(T_AUTHOR)
      .join(T_BOOK);
```

Now, if you do add a JOIN clause, you have to specify the JOIN .. ON condition before you can add more clauses. That's not an optional step for some JOIN types. This is reflected by the fact that **org.jooq.SelectOnStep** is a top-level interface.

```
// These join conditions are supported
SelectJoinStep   on(Condition... conditions);
SelectJoinStep onKey();
SelectJoinStep onKey(TableField<?, ?>... keyFields);
SelectJoinStep onKey(ForeignKey<?, ?> key);
SelectJoinStep using(Field<?>... fields);

// The example, continued:
create.select(T_AUTHOR.FIRST_NAME, T_AUTHOR.LAST_NAME, count())
      .from(T_AUTHOR)
      .join(T_BOOK).on(T_BOOK.AUTHOR_ID.equal(T_AUTHOR.ID));
```

See the section about **conditions** to learn more about the many ways to create Conditions in jOOQ. See also the section about **table sources** to learn more about the various ways of creating JOIN expressions

Now we're half way through. As you can see above, we're back to the SelectJoinStep. This means, we can re-iterate and add another JOIN clause, just like in SQL. Or we go on to the next step, adding conditions in the **org.jooq.SelectWhereStep**:

```
SelectConditionStep where(Condition... conditions);

// The example, continued:
create.select(TAuthor.FIRST_NAME, TAuthor.LAST_NAME, count())
      .from(T_AUTHOR)
      .join(T_BOOK).on(T_BOOK.AUTHOR_ID.equal(T_AUTHOR.ID))
      .where(T_BOOK.LANGUAGE.equal("DE"));
```

Now the returned type **org.jooq.SelectConditionStep** is a special one, where you can add more conditions to the already existing WHERE clause. Every time you add a condition, you will return to that SelectConditionStep, as the number of additional conditions is unlimited. Note that of course you can also just add a single combined condition, if that is more readable or suitable for your use-case. Here's how we add another condition:

```
SelectConditionStep and(Condition condition);

// The example, continued:
create.select(T_AUTHOR.FIRST_NAME, T_AUTHOR.LAST_NAME, count())
      .from(T_AUTHOR)
      .join(T_BOOK).on(T_BOOK.AUTHOR_ID.equal(T_AUTHOR.ID))
      .where(T_BOOK.LANGUAGE.equal("DE"))
      .and(T_BOOK.PUBLISHED.greaterThan(parseDate('2008-01-01')));
```

Let's assume we have that method parseDate() creating a **java.sql.Date** for us. Then we'll continue adding the GROUP BY clause

```
SelectHavingStep groupBy(Field<?>... fields);

// The example, continued:
create.select(T_AUTHOR.FIRST_NAME, T_AUTHOR.LAST_NAME, count())
      .from(T_AUTHOR)
      .join(T_BOOK).on(T_BOOK.AUTHOR_ID.equal(T_AUTHOR.ID))
      .where(T_BOOK.LANGUAGE.equal("DE"))
      .and(T_BOOK.PUBLISHED.greaterThan(parseDate('2008-01-01')))
      .groupBy(T_AUTHOR.FIRST_NAME, T_AUTHOR.LAST_NAME);
```

and the HAVING clause:

```
SelectOrderByStep having(Condition... conditions);

// The example, continued:
create.select(T_AUTHOR.FIRST_NAME, T_AUTHOR.LAST_NAME, count())
      .from(T_AUTHOR)
      .join(T_BOOK).on(T_BOOK.AUTHOR_ID.equal(T_AUTHOR.ID))
      .where(T_BOOK.LANGUAGE.equal("DE"))
      .and(T_BOOK.PUBLISHED.greaterThan(parseDate('2008-01-01')))
      .groupBy(T_AUTHOR.FIRST_NAME, T_AUTHOR.LAST_NAME)
      .having(count().greaterThan(5));
```

and the ORDER BY clause. Some RDBMS support NULLS FIRST and NULLS LAST extensions to the ORDER BY clause. If this is not supported by the RDBMS, then the behaviour is simulated with an additional CASE WHEN ... IS NULL THEN 1 ELSE 0 END clause.

```
SelectLimitStep orderBy(Field<?>... fields);

// The example, continued:
create.select(T_AUTHOR.FIRST_NAME, T_AUTHOR.LAST_NAME, count())
      .from(T_AUTHOR)
      .join(T_BOOK).on(T_BOOK.AUTHOR_ID.equal(T_AUTHOR.ID))
      .where(T_BOOK.LANGUAGE.equal("DE"))
      .and(T_BOOK.PUBLISHED.greaterThan(parseDate('2008-01-01')))
      .groupBy(T_AUTHOR.FIRST_NAME, T_AUTHOR.LAST_NAME)
      .having(count().greaterThan(5))
      .orderBy(T_AUTHOR.LAST_NAME.asc().nullsFirst());
```

and finally the LIMIT clause. Most dialects have a means of limiting the number of result records (except Oracle). Some even support having an OFFSET to the LIMIT clause. Even if your RDBMS does not support the full LIMIT ... OFFSET ... (or TOP ... START AT ..., or FETCH FIRST ... ROWS ONLY, etc) clause, jOOQ will simulate the LIMIT clause using nested selects and filtering on ROWNUM (for Oracle), or on ROW_NUMBER() (for DB2 and SQL Server):

```
SelectFinalStep limit(int offset, int numberOfRows);

// The example, continued:
create.select(T_AUTHOR.FIRST_NAME, T_AUTHOR.LAST_NAME, count())
      .from(T_AUTHOR)
      .join(T_BOOK).on(T_BOOK.AUTHOR_ID.equal(T_AUTHOR.ID))
      .where(T_BOOK.LANGUAGE.equal("DE"))
      .and(T_BOOK.PUBLISHED.greaterThan(parseDate('2008-01-01')))
      .groupBy(T_AUTHOR.FIRST_NAME, T_AUTHOR.LAST_NAME)
      .having(count().greaterThan(5))
      .orderBy(T_AUTHOR.LAST_NAME.asc().nullsFirst())
      .limit(1, 2);
```

In the final step, there are some proprietary extensions available only in some RDBMS. One of those extensions are the FOR UPDATE (supported in most RDBMS) and FOR SHARE clauses (supported only in MySQL and Postgres):

jOOQ is brought to you by Lukas Eder. Distributed under the Apache 2 licence

Page 41 / 68

```
SelectFinalStep forUpdate();

// The example, continued:
create.select(T_AUTHOR.FIRST_NAME, T_AUTHOR.LAST_NAME, count())
      .from(T_AUTHOR)
      .join(T_BOOK).on(T_BOOK.AUTHOR_ID.equal(T_AUTHOR.ID))
      .where(T_BOOK.LANGUAGE.equal("DE"))
      .and(T_BOOK.PUBLISHED.greaterThan(parseDate('2008-01-01')))
      .groupBy(T_AUTHOR.FIRST_NAME, T_AUTHOR.LAST_NAME)
      .having(count().greaterThan(5))
      .orderBy(T_AUTHOR.LAST_NAME.asc().nullsFirst())
      .limit(1, 2)
      .forUpdate();
```

Now the most relevant super-type of the object we have just created is **org.jooq.Select**. This type can be reused in various expressions such as in the **UNION and other set operations**, **Nested select statements using the EXISTS operator**, etc. If you just want to execute this select statement, you can choose any of these methods as discussed in the section about the **ResultQuery**:

```
// Just execute the query.
int execute();

// Execute the query and retrieve the results
Result<Record> fetch();

// Execute the query and retrieve the first Record
Record fetchAny();

// Execute the query and retrieve the single Record
// An Exception is thrown if more records were available
Record fetchOne();

// [...]
```

SELECT from single physical tables

A very similar, but limited API is available, if you want to select from single physical tables in order to retrieve TableRecords or even UpdatableRecords (see also the manual's section on **SelectQuery vs SimpleSelectQuery**). The decision, which type of select to create is already made at the very first step, when you create the SELECT statement with the Factory:

```
public <R extends Record> SimpleSelectWhereStep<R> selectFrom(Table<R> table);
```

As you can see, there is no way to further restrict/project the selected fields. This just selects all known TableFields in the supplied Table, and it also binds <R extends Record> to your Table's associated Record. An example of such a Query would then be:

```
TBook book = create.selectFrom(T_BOOK)
                   .where(TBook.LANGUAGE.equal("DE"))
                   .orderBy(TBook.TITLE)
                   .fetchAny();
```

3.2. Table sources

When OLTP selects/updates/inserts/deletes records in single tables, OLAP is all about creating custom table sources or ad-hoc row types

Create complex and nested table sources with jOOQ

In the **relational data model**, there are many operations performed on entities, i.e. tables in order to join them together before applying predicates, renaming operations and projections. Apart from the convenience methods for joining table sources in the **manual's section about the full SELECT syntax**, the **Table** type itself provides a rich API for creating joined table sources. See an extract of the Table API:

```
// These are the various supported JOIN clauses. These JOIN types
// are followed by an additional ON / ON KEY / USING clause
TableOnStep join(TableLike<?> table);
TableOnStep join(String sql);
TableOnStep join(String sql, Object... bindings);

// All other JOIN types are equally overloaded with "String sql" convenience methods...
TableOnStep  leftOuterJoin(TableLike<?> table);
TableOnStep rightOuterJoin(TableLike<?> table);
TableOnStep  fullOuterJoin(TableLike<?> table);

// These JOIN types don't take any additional clause
Table<Record>            crossJoin(TableLike<?> table);
Table<Record>          naturalJoin(TableLike<?> table);
Table<Record>  naturalLeftOuterJoin(TableLike<?> table);
Table<Record> naturalRightOuterJoin(TableLike<?> table);

// Oracle and SQL Server also know PIVOT / UNPIVOT clauses for transforming a
// table into another one using a list of PIVOT values
PivotForStep pivot(Field<?>... aggregateFunctions);
PivotForStep pivot(Collection<? extends Field<?>> aggregateFunctions);
```

The **TableOnStep** type contains methods for constructing the ON / ON KEY / USING clauses

```
// The ON clause is the most widely used JOIN condition. Provide arbitrary conditions as arguments
TableOnConditionStep on(Condition... conditions);
TableOnConditionStep on(String sql);
TableOnConditionStep on(String sql, Object... bindings);

// The USING clause is also part of the SQL standard. Use this if joined tables contain identical field names.
// The USING clause is simulated in databases that do not support it.
Table<Record> using(Field<?>... fields);
Table<Record> using(Collection<? extends Field<?>> fields);

// The ON KEY clause is a "synthetic" clause that does not exist in any SQL dialect. jOOQ usually has all
// foreign key relationship information to dynamically render "ON [ condition ... ]" clauses
TableOnConditionStep onKey() throws DataAccessException;
TableOnConditionStep onKey(TableField<?, ?>... keyFields) throws DataAccessException;
TableOnConditionStep onKey(ForeignKey<?, ?> key);
```

For more details about the PIVOT clause, see the **manual's section about the Oracle PIVOT syntax**

3.3. Conditions

The creation of conditions is the part of any database abstraction that attracts the most attention.

Conditions are the SELECT's core business

In your average application, you will typically have 3-4 SQL queries that have quite a long list of predicates (and possibly JOINs), such that you start to lose track over the overall boolean expression that you're trying to apply.
In jOOQ, most Conditions can be created and combined almost as easily as in SQL itself. The two main participants for creating Conditions are the **Field**, which is typically a participant of a condition, and the **Condition** itself:

```
public interface Condition {
    Condition and(Condition other);
    Condition and(String sql);
    Condition and(String sql, Object... bindings);
    Condition andNot(Condition other);
    Condition andExists(Select<?> select);
    Condition andNotExists(Select<?> select);
    Condition or(Condition other);
    Condition or(String sql);
    Condition or(String sql, Object... bindings);
    Condition orNot(Condition other);
    Condition orExists(Select<?> select);
    Condition orNotExists(Select<?> select);
    Condition not();
}
```

The above example describes the essence of boolean logic in jOOQ. As soon as you have a Condition object, you can connect that to other Conditions, which will then give you a combined condition with exactly the same properties. There are also convenience methods to create an EXISTS clause and connect it to an existing condition. In order to create a new Condition you are going to depart from a Field in most cases. Here are some important API elements in the Field interface:

```
public interface Field<T> {
    Condition isNull();
    Condition isNotNull();
    Condition like(T value);
    Condition notLike(T value);
    Condition in(T... values);
    Condition in(Select<?> query);
    Condition notIn(Collection<T> values);
    Condition notIn(T... values);
    Condition notIn(Select<?> query);
    Condition in(Collection<T> values);
    Condition between(T minValue, T maxValue);
    Condition contains(T value);
    Condition contains(Field<T> value);
    Condition equal(T value);
    Condition equal(Field<T> field);
    Condition equal(Select<?> query);
    Condition equalAny(Select<?> query);
    Condition equalAny(T... array);
    Condition equalAny(Field<T[]> array);
    Condition equalAll(Select<?> query);
    Condition equalAll(T... array);
    Condition equalAll(Field<T[]> array);
    Condition equalIgnoreCase(String value);
    Condition equalIgnoreCase(Field<String> value);
    Condition notEqual(T value);
    Condition notEqual(Field<T> field);
    Condition notEqual(Select<?> query);
    Condition notEqualAny(Select<?> query);
    Condition notEqualAny(T... array);
    Condition notEqualAny(Field<T[]> array);
    Condition notEqualAll(Select<?> query);
    Condition notEqualAll(T... array);
    Condition notEqualAll(Field<T[]> array);

    // Subselects, ANY and ALL quantifiers are also supported for these:
    Condition lessThan(T value);
    Condition lessOrEqual(T value);
    Condition greaterThan(T value);
    Condition greaterOrEqual(T value);
}
```

As you see in the partially displayed API above, you can compare a Field either with other Fields, with constant values (which is a shortcut for calling Factory.val(T value)), or with a nested SELECT statement. See some more **Examples of nested SELECTs**.
Combining the API of Field and Condition you can express complex predicates like this:

```
(T_BOOK.TYPE_CODE IN (1, 2, 5, 8, 13, 21)      AND T_BOOK.LANGUAGE = 'DE') OR
(T_BOOK.TYPE_CODE IN (2, 3, 5, 7, 11, 13)      AND T_BOOK.LANGUAGE = 'FR') OR
(T_BOOK.TYPE_CODE IN (SELECT CODE FROM T_TYPES) AND T_BOOK.LANGUAGE = 'EN')
```

Just write:

```
T_BOOK.TYPE_CODE.in(1, 2, 5, 8, 13, 21)                                .and(T_BOOK.LANGUAGE.equal("DE")).or(
T_BOOK.TYPE_CODE.in(2, 3, 5, 7, 11, 13)                                .and(T_BOOK.LANGUAGE.equal("FR")).or(
T_BOOK.TYPE_CODE.in(create.select(T_TYPES.CODE).from(T_TYPES)).and(T_BOOK.LANGUAGE.equal("EN"))));
```

3.4. Aliased tables and fields

Aliasing is at the core of SQL and relational algebra. When you join the same entity multiple times, you can rename it to distinguish the various meanings of the same entity

Aliasing Tables

A typical example of what you might want to do in SQL is this:

```
SELECT a.ID, b.ID
  FROM T_AUTHOR a
  JOIN T_BOOK b on a.ID = b.AUTHOR_ID
```

In this example, we are aliasing Tables, calling them a and b. The way aliasing works depends on how you generate your meta model using jooq-codegen (see the manual's section about **generating**

tables). Things become simpler when you choose the instance/dynamic model, instead of the static one. Here is how you can create Table aliases in jOOQ:

```
Table<TBookRecord> book = T_BOOK.as("b");
Table<TAuthorRecord> author = T_AUTHOR.as("a");

// If you choose not to generate a static meta model, this becomes even better
TBook book = T_BOOK.as("b");
TAuthor author = T_AUTHOR.as("a");
```

Now, if you want to reference any fields from those Tables, you may not use the original T_BOOK or T_AUTHOR meta-model objects anymore. Instead, you have to get the fields from the new book and author Table aliases:

```
Field<Integer> bookID = book.getField(TBook.ID);
Field<Integer> authorID = author.getField(TAuthor.ID);

// Or with the instance field model:
Field<Integer> bookID = book.ID;
Field<Integer> authorID = author.ID;
```

So this is how the above SQL statement would read in jOOQ:

```
create.select(authorID, bookID)
      .from(author)
      .join(book).on(authorID.equal(book.getField(T_BOOK.AUTHOR_ID)));

// Or with the instance field model:
create.select(author.ID, book.ID)
      .from(author)
      .join(book).on(author.ID.equal(book.AUTHOR_ID))
```

Aliasing nested selects as tables

There is an interesting, more advanced example of how you can select from an aliased nested select in the manual's section about **nested selects**

Aliasing fields

Fields can also be aliased independently from Tables. Most often, this is done when using functions or aggregate operators. Here is an example:

```
  SELECT FIRST_NAME || ' ' || LAST_NAME author, COUNT(*) books
    FROM T_AUTHOR
    JOIN T_BOOK ON T_AUTHOR.ID = AUTHOR_ID
GROUP BY FIRST_NAME, LAST_NAME;
```

Here is how it's done with jOOQ:

```
Record record = create.select(
        concat(T_AUTHOR.FIRST_NAME, " ", T_AUTHOR.LAST_NAME).as("author"),
        count().as("books"))
      .from(T_AUTHOR)
      .join(T_BOOK).on(T_AUTHOR.ID.equal(T_BOOK.AUTHOR_ID))
      .groupBy(T_AUTHOR.FIRST_NAME, T_AUTHOR.LAST_NAME).fetchAny();
```

When you alias Fields like above, you can access those Fields' values using the alias name:

```
System.out.println("Author : " + record.getValue("author"));
System.out.println("Books  : " + record.getValue("books"));
```

3.5. Nested SELECT using the IN operator

Set equal operations are very common when you want to select multiple records. The IN operator can also be used for semi-joins, though

The IN operator for use in semi-joins or anti-joins

In addition to a list of constant values, the IN operator in **org.jooq.Field** also supports a **org.jooq.Select** as an argument. This can be any type of select as discussed in the manual's section about **Query types**. However, you must ensure yourself, that the provided Select will only select a single Field.

Let's say you want to select books by authors born in 1920. Of course, this is possible with a plain JOIN as well, but let's say we want to use the IN operator. Then you have two possibilities:

```
SELECT *
  FROM T_BOOK
 WHERE T_BOOK.AUTHOR_ID IN (
         SELECT ID FROM T_AUTHOR
          WHERE T_AUTHOR.BORN = 1920)

-- OR:

SELECT T_BOOK.*
  FROM T_BOOK
  JOIN T_AUTHOR ON (T_BOOK.AUTHOR_ID = T_AUTHOR.ID
              AND T_AUTHOR.BORN    = 1920)
```

```
create.select()
      .from(T_BOOK)
      .where(T_BOOK.AUTHOR_ID.in(
          create.select(T_AUTHOR.ID).from(T_AUTHOR)
                .where(T_AUTHOR.BORN.equal(1920))));

// OR:

create.select(T_BOOK.getFields())
      .from(T_BOOK)
      .join(T_AUTHOR).on(T_BOOK.AUTHOR_ID.equal(TAuthor.ID)
                     .and(T_AUTHOR.BORN.equal(1920)));
```

3.6. Nested SELECT using the EXISTS operator

The EXISTS operator is a bit different from all other SQL elements, as it cannot really be applied to any object in a DSL.

The EXISTS operator for use in semi-joins or anti-joins

The EXISTS operator is rather independent and can stand any place where there may be a new condition:

- It may be placed right after a WHERE keyword
- It may be the right-hand-side of a boolean operator
- It may be placed right after a ON or HAVING keyword (although, this is less likely to be done...)

This is reflected by the fact that an EXISTS clause is usually created directly from the Factory:

```
Condition exists(Select<?> query);
Condition notExists(Select<?> query);
```

When you create such a Condition, it can then be connected to any other condition using AND, OR operators (see also the manual's section on **Conditions**). There are also quite a few convenience methods, where they might be useful. For instance in the **org.jooq.Condition** itself:

```
Condition andExists(Select<?> select);
Condition andNotExists(Select<?> select);
Condition orExists(Select<?> select);
Condition orNotExists(Select<?> select);
```

Or in the **org.jooq.SelectWhereStep**:

```
SelectConditionStep whereExists(Select<?> select);
SelectConditionStep whereNotExists(Select<?> select);
```

Or in the **org.jooq.SelectConditionStep**:

```
SelectConditionStep andExists(Select<?> select);
SelectConditionStep andNotExists(Select<?> select);
SelectConditionStep orExists(Select<?> select);
SelectConditionStep orNotExists(Select<?> select);
```

An example of how to use it is quickly given. Get all authors that haven't written any books:

```
SELECT *
  FROM T_AUTHOR
 WHERE NOT EXISTS (SELECT 1
                     FROM T_BOOK
                    WHERE T_BOOK.AUTHOR_ID = T_AUTHOR.ID)
```

```
create.select()
      .from(T_AUTHOR)
      .whereNotExists(create.selectOne()
          .from(T_BOOK)
          .where(T_BOOK.AUTHOR_ID.equal(T_AUTHOR.ID)));
```

3.7. Other types of nested SELECT

Apart from the most common IN and EXISTS clauses that encourage the use of nested selects, SQL knows a few more syntaxes to make use of such constructs.

Comparison with single-field SELECT clause

If you can ensure that a nested SELECT will only return one Record with one Field, then you can test for equality. This is how it is done in SQL:

```
SELECT *
  FROM T_BOOK
 WHERE T_BOOK.AUTHOR_ID = (
   SELECT ID
         FROM T_AUTHOR
        WHERE LAST_NAME = 'Orwell')
```

```
create.select()
      .from(T_BOOK)
      .where(T_BOOK.AUTHOR_ID.equal(create
          .select(T_AUTHOR.ID)
          .from(T_AUTHOR)
          .where(T_AUTHOR.LAST_NAME.equal("Orwell"))));
```

More examples like the above can be guessed from the **org.jooq.Field** API, as documented in the manual's section about **Conditions**. For the = operator, the available comparisons are these:

```
Condition equal(Select<?> query);
Condition equalAny(Select<?> query);
Condition equalAll(Select<?> query);
```

Selecting from a SELECT - SELECT acts as a Table

Often, you need to nest a SELECT statement simply because SQL is limited in power. For instance, if you want to find out which author has written the most books, then you cannot do this:

```
  SELECT AUTHOR_ID, count(*) books
     FROM T_BOOK
GROUP BY AUTHOR_ID
ORDER BY books DESC
```

Instead, you have to do this (or something similar). For jOOQ, this is an excellent example, combining various SQL features into a single statement. Here's how to do it:

```
SELECT nested.* FROM (
     SELECT AUTHOR_ID, count(*) books
       FROM T_BOOK
   GROUP BY AUTHOR_ID
) nested
ORDER BY nested.books DESC
```

```
Table<Record> nested =
    create.select(T_BOOK.AUTHOR_ID, count().as("books"))
          .from(T_BOOK)
          .groupBy(T_BOOK.AUTHOR_ID).asTable("nested");

create.select(nested.getFields())
      .from(nested)
      .orderBy(nested.getField("books"));
```

You'll notice how some verbosity seems inevitable when you combine nested SELECT statements with aliasing.

Selecting a SELECT - SELECT acts as a Field

Now SQL is even more powerful than that. You can also have SELECT statements, wherever you can have Fields. It get's harder and harder to find good examples, because there is always an easier way to express the same thing. But why not just count the number of books the really hard way? :-) But then again, maybe you want to take advantage of **Oracle Scalar Subquery Caching**

```
  SELECT LAST_NAME, (
     SELECT COUNT(*)
       FROM T_BOOK
      WHERE T_BOOK.AUTHOR_ID = T_AUTHOR.ID) books
    FROM T_AUTHOR
ORDER BY books DESC
```

```
// The type of books cannot be inferred from the Select<?>
Field<Object> books =
    create.selectCount()
          .from(T_BOOK)
          .where(T_BOOK.AUTHOR_ID.equal(T_AUTHOR.ID))
          .asField("books");
create.select(T_AUTHOR.ID, books)
      .from(T_AUTHOR)
      .orderBy(books, T_AUTHOR.ID));
```

3.8. UNION and other set operations

Unions, differences and intersections are vital set operations taken from set theory.

jOOQ's set operation API

The **org.jooq.Select** API directly supports the UNION syntax for all types of Select as discussed in the manual's section about **Queries and Query subtypes**. It consists of these methods:

```
public interface Select<R extends Record> {
    Select<R> union(Select<R> select);
    Select<R> unionAll(Select<R> select);
    Select<R> except(Select<R> select);
    Select<R> intersect(Select<R> select);
}
```

Hence, this is how you can write a simple UNION with jOOQ:

```
SELECT TITLE
  FROM T_BOOK
 WHERE PUBLISHED_IN > 1945
UNION
SELECT TITLE
  FROM T_BOOK
 WHERE AUTHOR_ID = 1
```

```
create.select(TBook.TITLE)
      .from(T_BOOK)
      .where(T_BOOK.PUBLISHED_IN.greaterThan(1945))
      .union(
create.select(T_BOOK.TITLE)
      .from(T_BOOK)
      .where(T_BOOK.AUTHOR_ID.equal(1)));
```

Nested UNIONs

In some SQL dialects, you can arbitrarily nest UNIONs to several levels. Be aware, though, that SQLite, Derby and MySQL have serious syntax limitations. jOOQ tries to render correct UNION SQL statements, but unfortunately, you can create situations that will cause syntax errors in the aforementioned dialects.
An example of advanced UNION usage is the following statement in jOOQ:

```
// Create a UNION of several types of books
Select<?> union =
    create.select(T_BOOK.TITLE, T_BOOK.AUTHOR_ID).from(T_BOOK).where(T_BOOK.PUBLISHED_IN.greaterThan(1945)).union(
    create.select(T_BOOK.TITLE, T_BOOK.AUTHOR_ID).from(T_BOOK).where(T_BOOK.AUTHOR_ID.equal(1)));

// Now, re-use the above UNION and order it by author
create.select(union.getField(T_BOOK.TITLE))
      .from(union)
      .orderBy(union.getField(T_BOOK.AUTHOR_ID).descending());
```

This example does not seem surprising, when you have read the previous chapters about **nested SELECT statements**. But when you check out the rendered SQL:

```
-- alias_38173 is an example of a generated alias,
-- generated by jOOQ for union queries
SELECT alias_38173.TITLE FROM (
  SELECT T_BOOK.TITLE, T_BOOK.AUTHOR_ID FROM T_BOOK WHERE T_BOOK.PUBLISHED_IN > 1945
  UNION
  SELECT T_BOOK.TITLE, T_BOOK.AUTHOR_ID FROM T_BOOK WHERE T_BOOK.AUTHOR_ID = 1
) alias_38173
ORDER BY alias_38173.AUTHOR_ID DESC
```

You can see that jOOQ takes care of many syntax pitfalls, when you're not used to the various dialects' unique requirements. The above automatic aliasing was added in order to be compliant with MySQL's requirements about aliasing nested selects.

Several UNIONs

It is no problem either for you to create SQL statements with several unions. Just write:

```
Select<?> part1;
Select<?> part2;
Select<?> part3;
Select<?> part4;

// [...]

part1.union(part2).union(part3).union(part4);
```

UNION and the ORDER BY clause

Strictly speaking, in SQL, you cannot order a subselect that is part of a UNION operation. You can only order the whole set. In set theory, or relational algebra, it wouldn't make sense to order subselects anyway, as a set operation cannot guarantee order correctness. Often, you still want to do it, because you apply a LIMIT to every subselect. Let's say, you want to find the employees with the highest salary in every department in Postgres syntax:

```
SELECT * FROM (
  SELECT * FROM emp WHERE dept = 'IT'
  ORDER BY salary LIMIT 1
) UNION (
  SELECT * FROM emp WHERE dept = 'Marketing'
  ORDER BY salary LIMIT 1
) UNION (
  SELECT * FROM emp WHERE dept = 'R&D'
  ORDER BY salary LIMIT 1
)
```

```
create.selectFrom(EMP).where(DEPT.equal("IT"))
      .orderBy(SALARY).limit(1)
      .union(
create.selectFrom(EMP).where(DEPT.equal("Marketing"))
      .orderBy(SALARY).limit(1))
      .union(
create.selectFrom(EMP).where(DEPT.equal("R&D"))
      .orderBy(SALARY).limit(1)))
```

There is a subtle difference between the above two queries. In SQL, every UNION subselect is in fact a **nested SELECT**, wrapped in parentheses. In this example, the notion of "nested SELECT" and "subselect" are slightly different.

3.9. Functions and aggregate operators

Highly effective SQL cannot do without functions. Operations on VARCHAR, DATE, and NUMERIC types in GROUP BY or ORDER BY clauses allow for very elegant queries.

jOOQ is brought to you by Lukas Eder. Distributed under the Apache 2 licence

Page 49 / 68

jOOQ's strategy for supporting vendor-specific functions

jOOQ allows you to access native functions from your RDBMS. jOOQ follows two strategies:

- Implement all SQL-92, SQL:1999, SQL:2003, and SQL:2008 standard functions, aggregate operators, and window functions. Standard functions could be **these functions as listed by O'Reilly**.
- Take the most useful of vendor-specific functions and simulate them for other RDBMS, where they may not be supported. An example for this are **Oracle Analytic Functions**

Functions

These are just a few functions in the Factory, so you get the idea:

```
Field<String> rpad(Field<String> field, Field<? extends Number> length);
Field<String> rpad(Field<String> field, int length);
Field<String> rpad(Field<String> field, Field<? extends Number> length, Field<String> c);
Field<String> rpad(Field<String> field, int length, char c);
Field<String> lpad(Field<String> field, Field<? extends Number> length);
Field<String> lpad(Field<String> field, int length);
Field<String> lpad(Field<String> field, Field<? extends Number> length, Field<String> c);
Field<String> lpad(Field<String> field, int length, char c);
Field<String> replace(Field<String> field, Field<String> search);
Field<String> replace(Field<String> field, String search);
Field<String> replace(Field<String> field, Field<String> search, Field<String> replace);
Field<String> replace(Field<String> field, String search, String replace);
Field<Integer> position(Field<String> field, String search);
Field<Integer> position(Field<String> field, Field<String> search);
```

Aggregate operators

Aggregate operators work just like functions, even if they have a slightly different semantics. Here are some examples from Factory:

```
// Every-day functions
AggregateFunction<Integer> count(Field<?> field);
AggregateFunction<T> max(Field<T> field);
AggregateFunction<T> min(Field<T> field);
AggregateFunction<BigDecimal> sum(Field<? extends Number> field);
AggregateFunction<BigDecimal> avg(Field<? extends Number> field);

// DISTINCT keyword in aggregate functions
AggregateFunction<Integer> countDistinct(Field<?> field);
AggregateFunction<T> maxDistinct(Field<T> field);
AggregateFunction<T> minDistinct(Field<T> field);
AggregateFunction<BigDecimal> sumDistinct(Field<? extends Number> field);
AggregateFunction<BigDecimal> avgDistinct(Field<? extends Number> field);

// Statistical functions
AggregateFunction<BigDecimal> median(Field<? extends Number> field);
AggregateFunction<BigDecimal> stddevPop(Field<? extends Number> field);
AggregateFunction<BigDecimal> stddevSamp(Field<? extends Number> field);
AggregateFunction<BigDecimal> varPop(Field<? extends Number> field);
AggregateFunction<BigDecimal> varSamp(Field<? extends Number> field);
```

A typical example of how to use an aggregate operator is when generating the next key on insertion of an ID. When you want to achieve something like this

```
SELECT MAX(ID) + 1 AS next_id
  FROM T_AUTHOR
```

```
create.select(max(ID).add(1).as("next_id"))
      .from(T_AUTHOR);
```

See also the section about **Arithmetic operations**

Window functions

Most major RDBMS support the concept of window functions. jOOQ knows of implementations in DB2, Oracle, Postgres, SQL Server, and Sybase SQL Anywhere, and supports most of their specific

syntaxes. Window functions can be used for things like calculating a "running total". The following example fetches transactions and the running total for every transaction going back to the beginning of the transaction table (ordered by booked_at). They are accessible from the previously seen AggregateFunction type using the over() method:

```
SELECT booked_at, amount,
   SUM(amount) OVER (PARTITION BY 1
                     ORDER BY booked_at
                     ROWS BETWEEN UNBOUNDED PRECEDING
                     AND CURRENT ROW) AS total
   FROM transactions
```

```
create.select(t.BOOKED_AT, t.AMOUNT,
        sum(t.AMOUNT).over().partitionByOne()
                     .orderBy(t.BOOKED_AT)
                     .rowsBetweenUnboundedPreceding()
                     .andCurrentRow().as("total")
      .from(TRANSACTIONS.as("t")));
```

3.10. Stored procedures and functions

The full power of your database's vendor-specific extensions can hardly be obtained outside of the database itself. Most modern RDBMS support their own procedural language. With jOOQ, stored procedures are integrated easily

Interaction with stored procedures

The main way to interact with your RDBMS's stored procedures and functions is by using the generated artefacts. See the manual's section about **generating procedures and packages** for more details about the source code generation for stored procedures and functions.

Stored functions

When it comes to DSL, stored functions can be very handy in SQL statements as well. Every stored function (this also applies to FUNCTIONS in Oracle PACKAGES) can generate a Field representing a call to that function. Typically, if you have this type of function in your database:

```
CREATE OR REPLACE FUNCTION f_author_exists (author_name VARCHAR2)
RETURN NUMBER;
```

Then convenience methods like these are generated:

```
// Create a field representing a function with another field as parameter
public static Field<BigDecimal> fAuthorExists(Field<String> authorName) { // [...]

// Create a field representing a function with a constant parameter
public static Field<BigDecimal> fAuthorExists(String authorName) { // [...]
```

Let's say, you have a T_PERSON table with persons' names in it, and you want to know whether there exists an author with precisely that name, you can reuse the above stored function in a SQL query:

```
SELECT T_PERSON.NAME, F_AUTHOR_EXISTS(T_PERSON.NAME)
   FROM T_PERSON

-- OR:

SELECT T_PERSON.NAME
   FROM T_PERSON
 WHERE F_AUTHOR_EXISTS(T_PERSON.NAME) = 1
```

```
create.select(T_PERSON.NAME,
 Functions.fAuthorExists(T_PERSON.NAME))
        .from(T_PERSON);

// OR: Note, the static import of Functions.*
create.select(T_PERSON.NAME)
        .from(T_PERSON)
        .where(fAuthorExists(T_PERSON.NAME));
```

Stored procedures

The notion of a stored procedure is implemented in most RDBMS by the fact, that the procedure has no RETURN VALUE (like void in Java), but it may well have OUT parameters. Since there is not a standard way how to embed stored procedures in SQL, they cannot be integrated in jOOQ's DSL either.

jOOQ is brought to you by Lukas Eder. Distributed under the Apache 2 licence

Page 51 / 68

3.11. Arithmetic operations and concatenation

Your database can do the math for you. Most arithmetic operations are supported, but also string concatenation can be very efficient if done already in the database.

Mathematical operations

Arithmetic operations are implemented just like **functions**, with similar limitations as far as type restrictions are concerned. You can use any of these operators:

```
+ - * / %
```

In order to express a SQL query like this one:

```
SELECT ((1 + 2) * (5 - 3) / 2) % 10 FROM DUAL
```

You can write something like this in jOOQ:

```
create.select(create.val(1).add(2).mul(create.val(5).sub(3)).div(2).mod(10));
```

Datetime arithmetic

jOOQ also supports the Oracle-style syntax for adding days to a Field<? extends java.util.Date>

```
SELECT SYSDATE + 3 FROM DUAL;
```

```
create.select(create.currentTimestamp().add(3));
```

String concatenation

This is not really an arithmetic expression, but it's still an expression with operators: The string concatenation. jOOQ provides you with the Field's concat() method:

```
SELECT 'A' || 'B' || 'C' FROM DUAL

-- Or in MySQL:
SELECT concat('A', 'B', 'C')
```

```
// For all RDBMS, including MySQL:
create.select(concat("A", "B", "C"));
```

3.12. The CASE clause

The SQL standard supports a CASE clause, which works very similar to Java's if-else statement. In complex SQL, this is very useful for value mapping

The two flavours of CASE

The CASE clause is part of the standard SQL syntax. While some RDBMS also offer an IF clause, or a DECODE function, you can always rely on the two types of CASE syntax:

```
CASE WHEN T_AUTHOR.FIRST_NAME = 'Paulo'  THEN 'brazilian'
     WHEN T_AUTHOR.FIRST_NAME = 'George' THEN 'english'
                                   ELSE 'unknown'
END

-- OR:

CASE T_AUTHOR.FIRST_NAME WHEN 'Paulo'  THEN 'brazilian'
                         WHEN 'George' THEN 'english'
                                  ELSE 'unknown'
END
```

```
create.decode()
      .when(T_AUTHOR.FIRST_NAME.equal("Paulo"),
 "brazilian")
      .when(T_AUTHOR.FIRST_NAME.equal("George"), "english")
      .otherwise("unknown");

// OR:

create.decode().value(T_AUTHOR.FIRST_NAME)
          .when("Paulo", "brazilian")
          .when("George", "english")
          .otherwise("unknown");
```

In jOOQ, both syntaxes are supported (although, Derby only knows the first one, which is more general). Unfortunately, both case and else are reserved words in Java. jOOQ chose to use decode() from the Oracle DECODE function, and otherwise(), which means the same as else. Please note that in the above examples, all values were always constants. You can of course also use Field instead of the various constants.

A CASE clause can be used anywhere where you can place a Field. For instance, you can SELECT the above expression, if you're selecting from T_AUTHOR:

```
SELECT T_AUTHOR.FIRST_NAME, [... CASE EXPR ...] AS nationality
   FROM T_AUTHOR
```

CASE clauses in an ORDER BY clause

Sort indirection is often implemented with a CASE clause of a SELECT's ORDER BY clause. In SQL, this reads:

```
SELECT *
FROM T_AUTHOR
ORDER BY CASE FIRST_NAME WHEN 'Paulo'  THEN 1
                         WHEN 'George' THEN 2
                                  ELSE null
         END
```

This will order your authors such that all 'Paulo' come first, then all 'George' and everyone else last (depending on your RDBMS' handling of NULL values in sorting). This is a very common task, such that jOOQ simplifies its use:

```
create.select()
      .from(T_AUTHOR)
      .orderBy(T_AUTHOR.FIRST_NAME.sortAsc("Paulo", "George"))
      .execute();
```

3.13. Type casting

Many RDBMS allow for implicit or explicit conversion between types. Apart from true type conversion, this is most often done with casting.

Enforcing a specific type when you need it

jOOQ's source code generator tries to find the most accurate type mapping between your vendor-specific data types and a matching Java type. For instance, most VARCHAR, CHAR, CLOB types will map to String. Most BINARY, BYTEA, BLOB types will map to byte[]. NUMERIC types will default to java.math.BigDecimal, but can also be any of java.math.BigInteger, Long, Integer, Short, Byte, Double, Float.

Sometimes, this automatic mapping might not be what you needed, or jOOQ cannot know the type of a field (because you created it from a **nested select**). In those cases you would write SQL type CASTs like this:

```
-- Let's say, your Postgres column LAST_NAME was VARCHAR(30)
-- Then you could do this:
SELECT CAST(T_AUTHOR.LAST_NAME AS TEXT) FROM DUAL
```

in jOOQ, you can write something like that:

```
create.select(TAuthor.LAST_NAME.cast(PostgresDataType.TEXT));
```

The same thing can be achieved by casting a Field directly to String.class, as TEXT is the default data type in Postgres to map to Java's String

```
create.select(TAuthor.LAST_NAME.cast(String.class));
```

The complete CAST API in Field consists of these three methods:

```
public interface Field<T> {
    <Z> Field<Z> cast(Field<Z> field);
    <Z> Field<Z> cast(DataType<Z> type);
    <Z> Field<Z> cast(Class<? extends Z> type);
}

// And additional convenience methods in the Factory:
public class Factory {
    <T> Field<T> cast(Object object, Field<T> field);
    <T> Field<T> cast(Object object, DataType<T> type);
    <T> Field<T> cast(Object object, Class<? extends T> type);
    <T> Field<T> castNull(Field<T> field);
    <T> Field<T> castNull(DataType<T> type);
    <T> Field<T> castNull(Class<? extends T> type);
}
```

3.14. When it's just easier: Plain SQL

jOOQ cannot foresee all possible vendor-specific SQL features for your database. And sometimes, even jOOQ code becomes too verbose. Then, you shouldn't hesitate to provide jOOQ with plain SQL, as you'd do with JDBC

Plain SQL in jOOQ

A DSL is a nice thing to have, it feels "fluent" and "natural", especially if it models a well-known language, such as SQL. But a DSL is always expressed in another language (Java in this case), which was not made for exactly that DSL. If it were, then jOOQ would be implemented on a compiler-level, similar to Linq in .NET. But it's not, and so, the DSL is limited. We have seen many functionalities where the DSL becomes verbose. This can be especially true for:

- **aliasing**
- **nested selects**
- **arithmetic expressions**
- **casting**

You'll probably find other examples. If verbosity scares you off, don't worry. The verbose use-cases for jOOQ are rather rare, and when they come up, you do have an option. Just write SQL the way you're used to!
jOOQ allows you to embed SQL as a String in these contexts:

- Plain SQL as a condition
- Plain SQL as a field
- Plain SQL as a function
- Plain SQL as a table
- Plain SQL as a query

To construct artefacts wrapping plain SQL, you should use any of these methods from the Factory class:

```
// A condition
Condition condition(String sql);
Condition condition(String sql, Object... bindings);

// A field with an unknown data type
Field<Object> field(String sql);
Field<Object> field(String sql, Object... bindings);

// A field with a known data type
<T> Field<T> field(String sql, Class<T> type);
<T> Field<T> field(String sql, Class<T> type, Object... bindings);
<T> Field<T> field(String sql, DataType<T> type);
<T> Field<T> field(String sql, DataType<T> type, Object... bindings);

// A function
<T> Field<T> function(String name, Class<T> type, Field<?>... arguments);
<T> Field<T> function(String name, DataType<T> type, Field<?>... arguments);

// A table
Table<?> table(String sql);
Table<?> table(String sql, Object... bindings);

// A query without results (update, insert, etc)
Query query(String sql);
Query query(String sql, Object... bindings);

// A query with results
ResultQuery<Record> resultQuery(String sql);
ResultQuery<Record> resultQuery(String sql, Object... bindings);

// A query with results. This is the same as resultQuery(...).fetch();
Result<Record> fetch(String sql);
Result<Record> fetch(String sql, Object... bindings);
```

Apart from the general factory methods, plain SQL is useful also in various other contexts. For instance, when adding a .where("a = b") clause to a query. Hence, there exist several convenience methods where plain SQL can be inserted usefully. This is an example displaying all various use-cases in one single query:

```
// You can use your table aliases in plain SQL fields
// As long as that will produce syntactically correct SQL
Field<?> LAST_NAME    = create.field("a.LAST_NAME");

// You can alias your plain SQL fields
Field<?> COUNT1       = create.field("count(*) x");

// If you know a reasonable Java type for your field, you
// can also provide jOOQ with that type
Field<Integer> COUNT2 = create.field("count(*) y", Integer.class);

        // Use plain SQL as select fields
create.select(LAST_NAME, COUNT1, COUNT2)

        // Use plain SQL as aliased tables (be aware of syntax!)
      .from("t_author a")
      .join("t_book b")

        // Use plain SQL for conditions both in JOIN and WHERE clauses
      .on("a.id = b.author_id")

        // Bind a variable in plain SQL
      .where("b.title != ?", "Brida")

        // Use plain SQL again as fields in GROUP BY and ORDER BY clauses
      .groupBy(LAST_NAME)
      .orderBy(LAST_NAME);
```

There are some important things to keep in mind when using plain SQL:

- jOOQ doesn't know what you're doing. You're on your own again!
- You have to provide something that will be syntactically correct. If it's not, then jOOQ won't know. Only your JDBC driver or your RDBMS will detect the syntax error.
- You have to provide consistency when you use variable binding. The number of ? must match the number of variables
- Your SQL is inserted into jOOQ queries without further checks. Hence, jOOQ can't prevent SQL injection.

4. Advanced topics

In these sections you will learn about advanced concepts that you might not use every day

Overview

This section covers some advanced topics and features that don't fit into any other section.

4.1. Master data generation. Enumeration tables

Enumerations are a powerful concept. Unfortunately, almost no RDBMS supports them, leaving you to create numerous tables for your enumerated values. But these values are still enumerations!

Enumeration tables

Only MySQL and Postgres databases support true ENUM types natively. Some other RDBMS allow you to map the concept of an ENUM data type to a CHECK constraint, but those constraints can contain arbitrary SQL. With jOOQ, you can "simulate" ENUM types by declaring a table as a "master data table" in the configuration. At code-generation time, this table will be treated specially, and a Java enum type is generated from its data.

Configure master data tables

As previously discussed in the **configuration and setup** section, you can configure master data tables as follows:

```xml
<!-- These properties can be added to the database element: -->
<database>
  <masterDataTables>
    <masterDataTable>
      <!-- The name of a master data table -->
      <name>[a table name]</name>

      <!-- The column used for enum literals -->
      <literal>[a column name]</literal>

      <!-- The column used for documentation -->
      <description>[a column name]</description>
    </masterDataTable>

    [ <masterDataTable>...</masterDataTable> ... ]
  </masterDataTables>
</database>
```

The results of this will be a Java enum that looks similar to this:

jOOQ is brought to you by Lukas Eder. Distributed under the Apache 2 licence

Page 57 / 68

```
public enum TLanguage implements MasterDataType<Integer> {

  /**
   * English
   */
  en(1, "en", "English"),

  /**
   * Deutsch
   */
  de(2, "de", "Deutsch"),

  /**
   * Français
   */
  fr(3, "fr", "Français"),

  /**
   * null
   */
  pt(4, "pt", null),
  ;

  private final Integer id;
  private final String cd;
  private final String description;

  // [ ... constructor and getters for the above properties ]
}
```

In the above example, you can see how the configured primary key is mapped to the id member, the configured literal column is mapped to the cd member and the configured description member is mapped to the description member and output as Javadoc. In other words, T_LANGUAGE is a table with 4 rows and at least three columns.
The general contract (with jOOQ 1.6.2+) is that there must be

-      A single-column primary key column of character or integer type
-      An optional unique literal column of character or integer type (otherwise, the primary key is used as enum literal)
-      An optional description column of any type

Using MasterDataTypes

The point of MasterDataTypes in jOOQ is that they behave exactly like true ENUM types. When the above T_LANGUAGE table is referenced by T_BOOK, instead of generating foreign key navigation methods and a LANGUAGE_ID Field<Integer>, a Field<TLanguage> is generated:

```
public class TBook extends UpdatableTableImpl<TBookRecord> {

  // [...]
  public static final TableField<TBookRecord, TLanguage> LANGUAGE_ID =
                new TableFieldImpl<TBookRecord, TLanguage>( /* ... */ );
}
```

Which can then be used in the TBookRecord directly:

```
public class TBookRecord extends UpdatableRecordImpl<TBookRecord> {

  // [...]
  public TLanguage getLanguageId() { // [...]
  public void setLanguageId(TLanguage value) { // [...]
}
```


When to use MasterDataTypes

You can use master data types when you're actually mapping master data to a Java enum. When the underlying table changes frequently, those updates will not be reflected by the statically generated code. Also, be aware that it will be difficult to perform actual JOIN operations on the underlying table with jOOQ, once the master data type is generated.

4.2. Mapping generated schemata and tables

Sometimes, you cannot control productive schema names, because your application is deployed on a shared host, and you only get one schema to work with.

Mapping your DEV schema to a productive environment

You may wish to design your database in a way that you have several instances of your schema. This is useful when you want to cleanly separate data belonging to several customers / organisation units / branches / users and put each of those entities' data in a separate database or schema.
In our T_AUTHOR example this would mean that you provide a book reference database to several companies, such as My Book World and Books R Us. In that case, you'll probably have a schema setup like this:

- DEV: Your development schema. This will be the schema that you base code generation upon, with jOOQ
- MY_BOOK_WORLD: The schema instance for My Book World
- BOOKS_R_US: The schema instance for Books R Us

Mapping DEV to MY_BOOK_WORLD with jOOQ

When a user from My Book World logs in, you want them to access the MY_BOOK_WORLD schema using classes generated from DEV. This can be achieved with the **org.jooq.conf.RenderMapping** class, that you can equip your Factory's settings with. Take the following example:

```
Settings settings = new Settings()
    .withRenderMapping(new RenderMapping()
    .withSchemata(
        new MappedSchema().withInput("DEV")
                          .withOutput("MY_BOOK_WORLD")));

// Add the settings to the factory
Factory create = new Factory(connection, SQLDialect.ORACLE, settings);

// Run queries with the "mapped" factory
create.selectFrom(T_AUTHOR).fetch();
```

The query executed with a Factory equipped with the above mapping will in fact produce this SQL statement:

```
SELECT * FROM MY_BOOK_WORLD.T_AUTHOR
```

Even if T_AUTHOR was generated from DEV.

Mapping several schemata

Your development database may not be restricted to hold only one DEV schema. You may also have a LOG schema and a MASTER schema. Let's say the MASTER schema is shared among all customers, but each customer has their own LOG schema instance. Then you can enhance your RenderMapping like this (e.g. using an XML configuration file):

jOOQ is brought to you by Lukas Eder. Distributed under the Apache 2 licence

Page 59 / 68

```
<settings xmlns="http://www.jooq.org/xsd/jooq-runtime-2.0.5.xsd">
  <renderMapping>
    <schemata>
      <schema>
        <input>DEV</input>
        <output>MY_BOOK_WORLD</output>
      </schema>
      <schema>
        <input>LOG</input>
        <output>MY_BOOK_WORLD_LOG</output>
      </schema>
    </schemata>
  </renderMapping>
</settings>
```

Note, you can load the above XML file like this:

```
Settings settings = JAXB.unmarshal(new File("jooq-runtime.xml"), Settings.class);
```

This will map generated classes from DEV to MY_BOOK_WORLD, from LOG to MY_BOOK_WORLD_LOG, but leave the MASTER schema alone. Whenever you want to change your mapping configuration, you will have to create a new Factory

Using a default schema

Another option to switch schema names is to use a default schema for the Factory's underlying Connection. Many RDBMS support a USE or SET SCHEMA command, which you can call like this:

```
// Set the default schema
Schema MY_BOOK_WORLD = ...
create.use(MY_BOOK_WORLD);

// Run queries with factory having a default schema
create.selectFrom(T_AUTHOR).fetch();
```

Queries generated from the above Factory will produce this kind of SQL statement:

```
-- the schema name is omitted from all SQL constructs.
SELECT * FROM T_AUTHOR
```

Mapping of tables

Not only schemata can be mapped, but also tables. If you are not the owner of the database your application connects to, you might need to install your schema with some sort of prefix to every table. In our examples, this might mean that you will have to map DEV.T_AUTHOR to something MY_BOOK_WORLD.MY_APP__T_AUTHOR, where MY_APP__ is a prefix applied to all of your tables. This can be achieved by creating the following mapping:

```
Settings settings = new Settings()
    .withRenderMapping(new RenderMapping()
    .withSchemata(
        new MappedSchema().withInput("DEV")
                          .withOutput("MY_BOOK_WORLD")
                          .withTables(
         new MappedTable().withInput("T_AUTHOR")
                          .withOutput("MY_APP__T_AUTHOR"))));

// Add the settings to the factory
Factory create = new Factory(connection, SQLDialect.ORACLE, settings);

// Run queries with the "mapped" factory
create.selectFrom(T_AUTHOR).fetch();
```

The query executed with a Factory equipped with the above mapping will in fact produce this SQL statement:

jOOQ is brought to you by Lukas Eder. Distributed under the Apache 2 licence

Page 60 / 68

```
SELECT * FROM MY_BOOK_WORLD.MY_APP__T_AUTHOR
```

Mapping at code generation time

Note that you can also hard-wire schema mapping in generated artefacts at code generation time, e.g. when you have 5 developers with their own dedicated developer databases, and a common integration database. In the code generation configuration, you would then write.

```
<schemata>
  <schema>
    <!-- Use this as the developer's schema: -->
    <inputSchema>LUKAS_DEV_SCHEMA</inputSchema>

    <!-- Use this as the integration / production database: -->
    <outputSchema>PROD</outputSchema>
  </schema>
</schemata>
```

See the manual's section about **jooq-codegen configuration** for more details

4.3. Execute listeners and SQL tracing

Feel the heart beat of your SQL statements at a very low level using listeners

ExecuteListener

The **jOOQ Factory Settings** let you specify a list of **org.jooq.ExecuteListener** classes. The ExecuteListener is essentially an event listener for Query, Routine, or ResultSet render, prepare, bind, execute, fetch steps. It is a base type for loggers, debuggers, profilers, data collectors. Advanced ExecuteListeners can also provide custom implementations of Connection, PreparedStatement and ResultSet to jOOQ in apropriate methods. For convenience, consider extending **org.jooq.impl.DefaultExecuteListener** instead of implementing this interface. Please read the **ExecuteListener Javadoc** for more details

jOOQ Console

The ExecuteListener API was driven by a feature request by Christopher Deckers, who has had the courtesy to contribute the jOOQ Console, a sample application interfacing with jOOQ's ExecuteListeners. Please note that the jOOQ Console is still experimental. Any feedback is very welcome on **the jooq-user group**
Here are the steps you need to do to run the console

```
// Create a new RemoteDebuggerServer in your application that listens to
// incoming connections on a given port
SERVER = new RemoteDebuggerServer(DEBUGGER_PORT);
```

And configure the **org.jooq.debug.DebugListener** in the Factory's settings:

```
<settings>
  <executeListeners>
    <executeListener>org.jooq.debug.DebugListener</executeListener>
  </executeListeners>
</settings>
```

Now start your application and the **org.jooq.debug.console.Console**, and start profiling! Use this command to start the console:

```
java -cp jooq-console-2.0.5.jar;rsyntaxtextarea-1.5.0.jar org.jooq.debug.console.Console [host] [port]
```

jOOQ is brought to you by Lukas Eder. Distributed under the Apache 2 licence

Page 61 / 68

The jOOQ Console also has other modes of execution, which will be documented here soon.

4.4. Adding Oracle hints to queries

Oracle has a powerful syntax to add hints as comments directly in your SQL

How to embed Oracle hints in SELECT

If you are closely coupling your application to an Oracle database, you might need to be able to pass hints of the form /*+HINT*/ with your SQL statements to the Oracle database. For example:

```
SELECT /*+ALL_ROWS*/ FIRST_NAME, LAST_NAME
  FROM T_AUTHOR
```

This can be done in jOOQ using the .hint() clause in your SELECT statement:

```
create.select(FIRST_NAME, LAST_NAME)
      .hint("/*+ALL_ROWS*/")
      .from(T_AUTHOR);
```

Note that you can pass any string in the .hint() clause. If you use that clause, the passed string will always be put in between the SELECT [DISTINCT] keywords and the actual projection list

4.5. The Oracle CONNECT BY clause

Hierarchical queries are supported by many RDBMS using the WITH clause. Oracle has a very neat and much less verbose syntax for hierarchical queries: CONNECT BY .. STARTS WITH

CONNECT BY .. STARTS WITH

If you are closely coupling your application to an Oracle database, you can take advantage of some Oracle-specific features, such as the CONNECT BY clause, used for hierarchical queries. The formal syntax definition is as follows:

```
--    SELECT ..
--      FROM ..
--     WHERE ..
 CONNECT BY [NOCYCLE] condition [AND condition, ...] [START WITH condition]
-- GROUP BY ..
```

This can be done in jOOQ using the .connectBy(Condition) clauses in your SELECT statement:

```
// Some Oracle-specific features are only available
// from the OracleFactory
OracleFactory create = new OracleFactory(connection);

// Get a table with elements 1, 2, 3, 4, 5
create.select(create.rownum())
      .connectBy(create.level().lessOrEqual(5))
      .fetch();
```

Here's a more complex example where you can recursively fetch directories in your database, and concatenate them to a path:

```
OracleFactory ora = new OracleFactory(connection);

List<?> paths =
ora.select(ora.sysConnectByPath(DIRECTORY.NAME, "/").substring(2))
   .from(DIRECTORY)
   .connectBy(ora.prior(DIRECTORY.ID).equal(DIRECTORY.PARENT_ID))
   .startWith(DIRECTORY.PARENT_ID.isNull())
   .orderBy(ora.literal(1))
   .fetch(0);
```

The output might then look like this

```
+---------------------------------------------+
|substring                                    |
+---------------------------------------------+
|C:                                           |
|C:/eclipse                                   |
|C:/eclipse/configuration                     |
|C:/eclipse/dropins                           |
|C:/eclipse/eclipse.exe                       |
+---------------------------------------------+
|...21 record(s) truncated...
```

4.6. The Oracle 11g PIVOT clause

Oracle 11g has formally introduced the very powerful PIVOT clause, which allows to specify a pivot column, expected grouping values for pivoting, as well as a set of aggregate functions

PIVOT (aggregate FOR column IN (columns))

If you are closely coupling your application to an Oracle database, you can take advantage of some Oracle-specific features, such as the PIVOT clause, used for statistical analyses. The formal syntax definition is as follows:

```
-- SELECT ..
     FROM table PIVOT (aggregateFunction [, aggregateFunction] FOR column IN (expression [, expression]))
--   WHERE ..
```

The PIVOT clause is available from the **org.jooq.Table** type, as pivoting is done directly on a table. Currently, only Oracle's PIVOT clause is supported. Support for SQL Server's PIVOT clause will be added later. Also, jOOQ may simulate PIVOT for other dialects in the future.

4.7. Exporting to XML, CSV, JSON, HTML, Text

Get your data out of the Java world. Stream your data using any of the supported, wide-spread formats

Exporting with jOOQ

If you are using jOOQ for scripting purposes or in a slim, unlayered application server, you might be interested in using jOOQ's exporting functionality (see also importing functionality). You can export any Result<Record> into any of these formats:

XML

Export your results as XML:

```
// Fetch books and format them as XML
String xml = create.selectFrom(T_BOOK).fetch().formatXML();
```

The above query will result in an XML document looking like the following one:

```
<!-- Find the XSD definition on www.jooq.org: -->
<jooq-export:result xmlns:jooq-export="http://www.jooq.org/xsd/jooq-export-1.6.2.xsd">
  <fields>
    <field name="ID"/>
    <field name="AUTHOR_ID"/>
    <field name="TITLE"/>
  </fields>
  <records>
    <record>
      <value field="ID">1</value>
      <value field="AUTHOR_ID">1</value>
      <value field="TITLE">1984</value>
    </record>
    <record>
      <value field="ID">2</value>
      <value field="AUTHOR_ID">1</value>
      <value field="TITLE">Animal Farm</value>
    </record>
  </records>
</jooq-export:result>
```

CSV

Export your results as CSV:

```
// Fetch books and format them as CSV
String csv = create.selectFrom(T_BOOK).fetch().formatCSV();
```

The above query will result in a CSV document looking like the following one:

jOOQ is brought to you by Lukas Eder. Distributed under the Apache 2 licence

Page 64 / 68

```
ID;AUTHOR_ID;TITLE
1;1;1984
2;1;Animal Farm
```

JSON

Export your results as JSON:

```
// Fetch books and format them as JSON
String json = create.selectFrom(T_BOOK).fetch().formatJSON();
```

The above query will result in a JSON document looking like the following one:

```
{fields:["ID","AUTHOR_ID","TITLE"],
 records:[[1,1,"1984"],[2,1,"Animal Farm"]]}
```

HTML

Export your results as HTML:

```
// Fetch books and format them as HTML
String html = create.selectFrom(T_BOOK).fetch().formatHTML();
```

The above query will result in an HTML document looking like the following one:

```
<table>
  <thead>
    <tr>
      <th>ID</th>
      <th>AUTHOR_ID</th>
      <th>TITLE</th>
    </tr>
  </thead>
  <tbody>
    <tr>
      <td>1</td>
      <td>1</td>
      <td>1984</td>
    </tr>
    <tr>
      <td>2</td>
      <td>1</td>
      <td>Animal Farm</td>
    </tr>
  </tbody>
</table>
```

Text

Export your results as text:

```
// Fetch books and format them as text
String text = create.selectFrom(T_BOOK).fetch().format();
```

The above query will result in a text document looking like the following one:

```
+---+---------+----------+
| ID|AUTHOR_ID|TITLE     |
+---+---------+----------+
|  1|        1|1984      |
|  2|        1|Animal Farm|
+---+---------+----------+
```

4.8. Importing data from XML, CSV

Use jOOQ to easily merge imported data into your database.

Importing with jOOQ

If you are using jOOQ for scripting purposes or in a slim, unlayered application server, you might be interested in using jOOQ's importing functionality (see also exporting functionality). You can import data directly into a table from any of these formats:

CSV

The below CSV data represents two author records that may have been exported previously, by jOOQ's exporting functionality, and then modified in Microsoft Excel or any other spreadsheet tool:

```
ID;AUTHOR_ID;TITLE
1;1;1984
2;1;Animal Farm
```

With jOOQ, you can load this data using various parameters from the loader API. A simple load may look like this:

```
Factory create = new Factory(connection, SQLDialect.ORACLE);

// Load data into the T_AUTHOR table from an input stream
// holding the CSV data.
create.loadInto(T_AUTHOR)
      .loadCSV(inputstream)
      .fields(ID, AUTHOR_ID, TITLE)
      .execute();
```

Here are various other examples:

```
// Ignore the AUTHOR_ID column from the CSV file when inserting
create.loadInto(T_AUTHOR)
      .loadCSV(inputstream)
      .fields(ID, null, TITLE)
      .execute();

// Specify behaviour for duplicate records.
create.loadInto(T_AUTHOR)

      // choose any of these methods
      .onDuplicateKeyUpdate()
      .onDuplicateKeyIgnore()
      .onDuplicateKeyError() // the default

      .loadCSV(inputstream)
      .fields(ID, null, TITLE)
      .execute();

// Specify behaviour when errors occur.
create.loadInto(T_AUTHOR)

      // choose any of these methods
      .onErrorIgnore()
      .onErrorAbort() // the default

      .loadCSV(inputstream)
      .fields(ID, null, TITLE)
      .execute();

// Specify transactional behaviour where this is possible
// (e.g. not in container-managed transactions)
create.loadInto(T_AUTHOR)

      // choose any of these methods
      .commitEach()
      .commitAfter(10)
      .commitAll()
      .commitNone() // the default

      .loadCSV(inputstream)
      .fields(ID, null, TITLE)
      .execute();
```

jOOQ is brought to you by Lukas Eder. Distributed under the Apache 2 licence

Page 66 / 68

Any of the above configuration methods can be combined to achieve the type of load you need. Please refer to the API's Javadoc to learn about more details. Errors that occur during the load are reported by the execute method's result:

```
Loader<TAuthor> loader = /* .. */ .execute();

// The number of processed rows
int processed = loader.processed();

// The number of stored rows (INSERT or UPDATE)
int stored = loader.stored();

// The number of ignored rows (due to errors, or duplicate rule)
int ignored = loader.ignored();

// The errors that may have occurred during loading
List<LoaderError> errors = loader.errors();
LoaderError error = errors.get(0);

// The exception that caused the error
SQLException exception = error.exception();

// The row that caused the error
int rowIndex = error.rowIndex();
String[] row = error.row();

// The query that caused the error
Query query = error.query();
```

XML

This will be implemented soon...

4.9. Using JDBC batch operations

Some JDBC drivers have highly optimised means of executing batch operations. The JDBC interface for those operations is a bit verbose. jOOQ abstracts that by re-using the existing query API's

JDBC batch operations

With JDBC, you can easily execute several statements at once using the addBatch() method. Essentially, there are two modes in JDBC

o   Execute several queries without bind values
o   Execute one query several times with bind values

In code, this looks like the following snippet:

```
// 1. several queries
// -----------------
Statement stmt = connection.createStatement();
stmt.addBatch("INSERT INTO author VALUES (1, 'Erich Gamma')");
stmt.addBatch("INSERT INTO author VALUES (2, 'Richard Helm')");
stmt.addBatch("INSERT INTO author VALUES (3, 'Ralph Johnson')");
stmt.addBatch("INSERT INTO author VALUES (4, 'John Vlissides')");
int[] result = stmt.executeBatch();

// 2. a single query
// ----------------
PreparedStatement stmt = connection.prepareStatement("INSERT INTO autho VALUES (?, ?)");
stmt.setInt(1, 1);
stmt.setString(2, "Erich Gamma");
stmt.addBatch();

stmt.setInt(1, 2);
stmt.setString(2, "Richard Helm");
stmt.addBatch();

stmt.setInt(1, 3);
stmt.setString(2, "Ralph Johnson");
stmt.addBatch();

stmt.setInt(1, 4);
stmt.setString(2, "John Vlissides");
stmt.addBatch();

int[] result = stmt.executeBatch();
```

This will also be supported by jOOQ

Version 1.6.9 of jOOQ now supports executing queries in batch mode as follows:

```
// 1. several queries
// -----------------
create.batch(
 create.insertInto(AUTHOR, ID, NAME).values(1, "Erich Gamma"),
 create.insertInto(AUTHOR, ID, NAME).values(2, "Richard Helm"),
 create.insertInto(AUTHOR, ID, NAME).values(3, "Ralph Johnson"),
 create.insertInto(AUTHOR, ID, NAME).values(4, "John Vlissides"))
.execute();

// 2. a single query
// ----------------
create.batch(create.insertInto(AUTHOR, ID, NAME).values("?", "?"))
    .bind(1, "Erich Gamma")
    .bind(2, "Richard Helm")
    .bind(3, "Ralph Johnson")
    .bind(4, "John Vlissides")
    .execute();
```