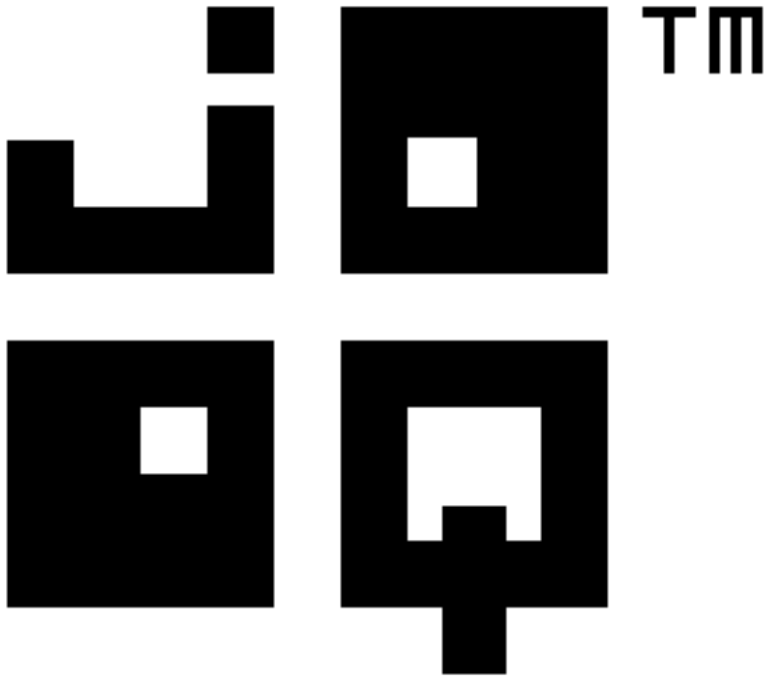
The jOOQ™ User Manual

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 six main sections:

- Getting started with jOOQ
  This section will get you started with jOOQ quickly. It contains simple explanations about what jOOQ is, what jOOQ isn't and how to set it up for the first time
- SQL building
  This section explains all about the jOOQ syntax used for building queries through the query DSL and the query model API. It explains the central factories, the supported SQL statements and various other syntax elements
- Code generation
  This section explains how to configure and use the built-in source code generator
- SQL execution
  This section will get you through the specifics of what can be done with jOOQ at runtime, in order to execute queries, perform CRUD operations, import and export data, and hook into the jOOQ execution lifecycle for debugging
- Tools
  This section is dedicated to tools that ship with jOOQ.
- Reference
  This section is a reference for elements in this manual

Table of contents

1. Preface

jOOQ's reason for being - compared to JPA

Java and SQL have come a long way. SQL is an "old", yet established and well-understood technology. Java is a legacy too, although its platform JVM allows for many new and contemporary languages built on top of it. Yet, after all these years, libraries dealing with the interface between SQL and Java have come and gone, leaving JPA to be a standard that is accepted only with doubts, short of any surviving options.

So far, there had been only few database abstraction frameworks or libraries, that truly respected SQL as a first class citizen among languages. Most frameworks, including the industry standards JPA, EJB, Hibernate, JDO, Criteria Query, and many others try to hide SQL itself, minimising its scope to things called JPQL, HQL, JDOQL and various other inferior query languages

jOOQ has come to fill this gap.

jOOQ's reason for being - compared to LINQ

Other platforms incorporate ideas such as LINQ (with LINQ-to-SQL), or Scala's SLICK, or also Java's QueryDSL to better integrate querying as a concept into their respective language. By querying, they understand querying of arbitrary targets, such as SQL, XML, Collections and other heterogeneous data stores. jOOQ claims that this is going the wrong way too.

In more advanced querying use-cases (more than simple CRUD and the occasional JOIN), people will want to profit from the expressivity of SQL. Due to the relational nature of SQL, this is quite different from what object-oriented and partially functional languages such as C#, Scala, or Java can offer.

It is very hard to formally express and validate joins and the ad-hoc table expression types they create. It gets even harder when you want support for more advanced table expressions, such as pivot tables, unnested cursors, or just arbitrary projections from derived tables. With a very strong object-oriented typing model, these features will probably stay out of scope.

In essence, the decision of creating an API that looks like SQL or one that looks like C#, Scala, Java is a definite decision in favour of one or the other platform. While it will be easier to evolve SLICK in similar ways as LINQ (or QueryDSL in the Java world), SQL feature scope that clearly communicates its underlying intent will be very hard to add, later on (e.g. how would you model Oracle's partitioned outer join syntax? How would you model ANSI/ISO SQL:1999 grouping sets? How can you support scalar subquery caching? etc...).

jOOQ has come to fill this gap.

jOOQ's reason for being - compared to SQL / JDBC

So why not just use SQL?

SQL can be written as plain text and passed through the JDBC API. Over the years, people have become wary of this approach for many reasons:

- No typesafety
- No syntax safety
- No bind value index safety
- Verbose SQL String concatenation
- Boring bind value indexing techniques
- Verbose resource and exception handling in JDBC
- A very "stateful", not very object-oriented JDBC API, which is hard to use

For these many reasons, other frameworks have tried to abstract JDBC away in the past in one way or another. Unfortunately, many have completely abstracted SQL away as well

jOOQ has come to fill this gap.

jOOQ is different

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!

2. Copyright, License, and Trademarks

This section lists the various licenses that apply to different versions of jOOQ. Prior to version 3.2, jOOQ was shipped for free under the terms of the Apache Software License 2.0. With jOOQ 3.2, jOOQ became dual-licensed: Apache Software License 2.0 (for use with Open Source databases) and commercial (for use with commercial databases).

This manual itself (as well as the www.jooq.org public website) is licensed to you under the terms of the CC BY-SA 4.0 license.

Please contact legal@datageekery.com, should you have any questions regarding licensing.

License for jOOQ 3.2 and later

```
This work is dual-licensed
- under the Apache Software License 2.0 (the "ASL")
- under the jOOQ License and Maintenance Agreement (the "jOOQ License")
==============================================================================
You may choose which license applies to you:

- If you're using this work with Open Source databases, you may choose
  either ASL or jOOQ License.
- If you're using this work with at least one commercial database, you must
  choose jOOQ License

For more information, please visit http://www.jooq.org/licenses

Apache Software License 2.0:
------------------------------------------------------------------------------
Licensed under the Apache License, Version 2.0 (the "License");
you may not use this file except in compliance with the License.
You may obtain a copy of the License at

 http://www.apache.org/licenses/LICENSE-2.0

Unless required by applicable law or agreed to in writing, software
distributed under the License is distributed on an "AS IS" BASIS,
WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either express or implied.
See the License for the specific language governing permissions and
limitations under the License.

jOOQ License and Maintenance Agreement:
------------------------------------------------------------------------------
Data Geekery grants the Customer the non-exclusive, timely limited and
non-transferable license to install and use the Software under the terms of
the jOOQ License and Maintenance Agreement.

This library is distributed with a LIMITED WARRANTY. See the jOOQ License
and Maintenance Agreement for more details: http://www.jooq.org/licensing
```

Historic license for jOOQ 1.x, 2.x, 3.0, 3.1

```
Licensed under the Apache License, Version 2.0 (the "License");
you may not use this file except in compliance with the License.
You may obtain a copy of the License at

    http://www.apache.org/licenses/LICENSE-2.0

Unless required by applicable law or agreed to in writing, software
distributed under the License is distributed on an "AS IS" BASIS,
WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either express or implied.
See the License for the specific language governing permissions and
limitations under the License.
```

Trademarks owned by Data Geekery™ GmbH

- jOOλ™ is a trademark by Data Geekery™ GmbH
- jOOQ™ is a trademark by Data Geekery™ GmbH
- jOOR™ is a trademark by Data Geekery™ GmbH
- jOOU™ is a trademark by Data Geekery™ GmbH
- jOOX™ is a trademark by Data Geekery™ GmbH

Trademarks owned by Data Geekery™ GmbH partners

- GSP and General SQL Parser are trademarks by Gudu Software Limited
- SQL 2 jOOQ is a trademark by Data Geekery™ GmbH and Gudu Software Limited
- Flyway is a trademark by Snow Mountain Labs UG (haftungsbeschränkt)

Trademarks owned by database vendors with no affiliation to Data Geekery™ GmbH

- Access® is a registered trademark of Microsoft® Inc.
- Adaptive Server® Enterprise is a registered trademark of Sybase®, Inc.
- CUBRID™ is a trademark of NHN® Corp.
- DB2® is a registered trademark of IBM® Corp.
- Derby is a trademark of the Apache™ Software Foundation
- H2 is a trademark of the H2 Group
- HANA is a trademark of SAP SE
- HSQLDB is a trademark of The hsql Development Group
- Ingres is a trademark of Actian™ Corp.
- MariaDB is a trademark of Monty Program Ab
- MySQL® is a registered trademark of Oracle® Corp.
- Firebird® is a registered trademark of Firebird Foundation Inc.
- Oracle® database is a registered trademark of Oracle® Corp.
- PostgreSQL® is a registered trademark of The PostgreSQL Global Development Group
- Postgres Plus® is a registered trademark of EnterpriseDB® software
- SQL Anywhere® is a registered trademark of Sybase®, Inc.
- SQL Server® is a registered trademark of Microsoft® Inc.
- SQLite is a trademark of Hipp, Wyrick & Company, Inc.

Other trademarks by vendors with no affiliation to Data Geekery™ GmbH

- Java® is a registered trademark by Oracle® Corp. and/or its affiliates
- Scala is a trademark of EPFL

Other trademark remarks

Other names may be trademarks of their respective owners.

Throughout the manual, the above trademarks are referenced without a formal ® (R) or ™ (TM) symbol. It is believed that referencing third-party trademarks in this manual or on the jOOQ website constitutes "fair use". Please contact us if you think that your trademark(s) are not properly attributed.

Contributions

The following are authors and contributors of jOOQ or parts of jOOQ in alphabetical order:

- Aaron Digulla
- Andreas Franzén
- Anuraag Agrawal
- Arnaud Roger
- Art O Cathain
- Artur Dryomov
- Ben Manes
- Brent Douglas
- Brett Meyer
- Christian Stein
- Christopher Deckers
- Ed Schaller
- Eric Peters
- Ernest Mishkin
- Espen Stromsnes
- Eugeny Karpov
- Fabrice Le Roy
- Gonzalo Ortiz Jaureguizar
- Gregory Hlavac
- Henrik Sjöstrand
- Ivan Dugic
- Javier Durante
- Johannes Bühler
- Joseph B Phillips
- Joseph Pachod
- Knut Wannheden
- Laurent Pireyn
- Luc Marchaud
- Lukas Eder
- Matti Tahvonen
- Michael Doberenz
- Michael Simons
- Michał Kołodziejski
- Miguel Gonzalez Sanchez
- Nathaniel Fischer
- Oliver Flege
- Peter Ertl
- Richard Bradley
- Robin Stocker
- Samy Deghou
- Sander Plas
- Sean Wellington
- Sergey Epik
- Sergey Zhuravlev
- Stanislas Nanchen
- Stephan Schroevers
- Sugiharto Lim
- Sven Jacobs
- Szymon Jachim
- Terence Zhang
- Timothy Wilson
- Timur Shaidullin
- Thomas Darimont
- Tsukasa Kitachi
- Victor Bronstein
- Victor Z. Peng
- Vladimir Kulev
- Vladimir Vinogradov
- Vojtech Polivka

See the following website for details about contributing to jOOQ:
http://www.jooq.org/legal/contributions

3. Getting started with jOOQ

These chapters contain a quick overview of how to get started with this manual and with jOOQ. While the subsequent chapters contain a lot of reference information, this chapter here just wraps up the essentials.

3.1. How to read this manual

This section helps you correctly interpret this manual in the context of jOOQ.

Code blocks

The following are code blocks:

```
-- A SQL code block
SELECT 1 FROM DUAL
```

```
// A Java code block
for (int i = 0; i < 10; i++);
```

```
<!-- An XML code block -->
<hello what="world"></hello>
```

```
# A config file code block
org.jooq.property=value
```

These are useful to provide examples in code. Often, with jOOQ, it is even more useful to compare SQL code with its corresponding Java/jOOQ code. When this is done, the blocks are aligned side-by-side, with SQL usually being on the left, and an equivalent jOOQ DSL query in Java usually being on the right:

```
-- In SQL:
SELECT 1 FROM DUAL
```

```
// Using jOOQ:
create.selectOne().fetch()
```

Code block contents

The contents of code blocks follow conventions, too. If nothing else is mentioned next to any given code block, then the following can be assumed:

```
-- SQL assumptions
------------------

-- If nothing else is specified, assume that the Oracle syntax is used
SELECT 1 FROM DUAL
```

```
// Java assumptions
// ----------------

// Whenever you see "standalone functions", assume they were static imported from org.jooq.impl.DSL
// "DSL" is the entry point of the static query DSL
exists(); max(); min(); val(); inline(); // correspond to DSL.exists(); DSL.max(); DSL.min(); etc...

// Whenever you see BOOK/Book, AUTHOR/Author and similar entities, assume they were (static) imported from the generated schema
BOOK.TITLE, AUTHOR.LAST_NAME // correspond to com.example.generated.Tables.BOOK.TITLE, com.example.generated.Tables.BOOK.TITLE
FK_BOOK_AUTHOR               // corresponds to com.example.generated.Keys.FK_BOOK_AUTHOR

// Whenever you see "create" being used in Java code, assume that this is an instance of org.jooq.DSLContext.
// The reason why it is called "create" is the fact, that a jOOQ QueryPart is being created from the DSL object.
// "create" is thus the entry point of the non-static query DSL
DSLContext create = DSL.using(connection, SQLDialect.ORACLE);
```

Your naming may differ, of course. For instance, you could name the "create" instance "db", instead.

Execution

When you're coding PL/SQL, T-SQL or some other procedural SQL language, SQL statements are always executed immediately at the semi-colon. This is not the case in jOOQ, because as an internal DSL, jOOQ can never be sure that your statement is complete until you call fetch() or execute(). The manual tries to apply fetch() and execute() as thoroughly as possible. If not, it is implied:

```
SELECT 1 FROM DUAL
UPDATE t SET v = 1
```

```
create.selectOne().fetch();
create.update(T).set(T.V, 1).execute();
```

Degree (arity)

jOOQ records (and many other API elements) have a degree N between 1 and 22. The variable degree of an API element is denoted as [N], e.g. Row[N] or Record[N]. The term "degree" is preferred over arity, as "degree" is the term used in the SQL standard, whereas "arity" is used more often in mathematics and relational theory.

Settings

jOOQ allows to override runtime behaviour using org.jooq.conf.Settings. If nothing is specified, the default runtime settings are assumed.

Sample database

jOOQ query examples run against the sample database. See the manual's section about the sample database used in this manual to learn more about the sample database.

3.2. The sample database used in this manual

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

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

CREATE TABLE 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),
  distinguished   NUMBER(1)
);

CREATE TABLE 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,

  CONSTRAINT fk_book_author     FOREIGN KEY (author_id)   REFERENCES author(id),
  CONSTRAINT fk_book_language   FOREIGN KEY (language_id) REFERENCES language(id)
);

CREATE TABLE book_store (
  name            VARCHAR2(400) NOT NULL UNIQUE
);

CREATE TABLE book_to_book_store (
  name            VARCHAR2(400) NOT NULL,
  book_id         INTEGER       NOT NULL,
  stock           INTEGER,

  PRIMARY KEY(name, book_id),
  CONSTRAINT fk_b2bs_book_store FOREIGN KEY (name)        REFERENCES book_store (name) ON DELETE CASCADE,
  CONSTRAINT fk_b2bs_book       FOREIGN KEY (book_id)     REFERENCES 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

In addition to the above, you may assume the following sample data:

```
INSERT INTO language (id, cd, description) VALUES (1, 'en', 'English');
INSERT INTO language (id, cd, description) VALUES (2, 'de', 'Deutsch');
INSERT INTO language (id, cd, description) VALUES (3, 'fr', 'Français');
INSERT INTO language (id, cd, description) VALUES (4, 'pt', 'Português');

INSERT INTO author (id, first_name, last_name, date_of_birth    , year_of_birth)
  VALUES           (1 , 'George'  , 'Orwell' , DATE '1903-06-26', 1903         );
INSERT INTO author (id, first_name, last_name, date_of_birth    , year_of_birth)
  VALUES           (2 , 'Paulo'   , 'Coelho' , DATE '1947-08-24', 1947         );

INSERT INTO book (id, author_id, title         , published_in, language_id)
  VALUES         (1 , 1        , '1984'        , 1948        , 1          );
INSERT INTO book (id, author_id, title         , published_in, language_id)
  VALUES         (2 , 1        , 'Animal Farm' , 1945        , 1          );
INSERT INTO book (id, author_id, title         , published_in, language_id)
  VALUES         (3 , 2        , 'O Alquimista', 1988        , 4          );
INSERT INTO book (id, author_id, title         , published_in, language_id)
  VALUES         (4 , 2        , 'Brida'       , 1990        , 2          );

INSERT INTO book_store VALUES ('Orell Füssli');
INSERT INTO book_store VALUES ('Ex Libris');
INSERT INTO book_store VALUES ('Buchhandlung im Volkshaus');

INSERT INTO book_to_book_store VALUES ('Orell Füssli'             , 1, 10);
INSERT INTO book_to_book_store VALUES ('Orell Füssli'             , 2, 10);
INSERT INTO book_to_book_store VALUES ('Orell Füssli'             , 3, 10);
INSERT INTO book_to_book_store VALUES ('Ex Libris'                , 1, 1 );
INSERT INTO book_to_book_store VALUES ('Ex Libris'                , 3, 2 );
INSERT INTO book_to_book_store VALUES ('Buchhandlung im Volkshaus', 3, 1 );
```

3.3. Different use cases for jOOQ

jOOQ has originally been created as a library for complete abstraction of JDBC and all database interaction. Various best practices that are frequently encountered in pre-existing software products are applied to this library. This includes:

- Typesafe database object referencing through generated schema, table, column, record, procedure, type, dao, pojo artefacts (see the chapter about code generation)
- Typesafe SQL construction / SQL building through a complete querying DSL API modelling SQL as a domain specific language in Java (see the chapter about the query DSL API)
- Convenient query execution through an improved API for result fetching (see the chapters about the various types of data fetching)
- SQL dialect abstraction and SQL clause emulation to improve cross-database compatibility and to enable missing features in simpler databases (see the chapter about SQL dialects)
- SQL logging and debugging using jOOQ as an integral part of your development process (see the chapters about logging)

Effectively, jOOQ was originally designed to replace any other database abstraction framework short of the ones handling connection pooling (and more sophisticated transaction management)

Use jOOQ the way you prefer

... but open source is community-driven. And the community has shown various ways of using jOOQ that diverge from its original intent. Some use cases encountered are:

- Using Hibernate for 70% of the queries (i.e. CRUD) and jOOQ for the remaining 30% where SQL is really needed
- Using jOOQ for SQL building and JDBC for SQL execution
- Using jOOQ for SQL building and Spring Data for SQL execution
- Using jOOQ without the source code generator to build the basis of a framework for dynamic SQL execution.

The following sections explain about various use cases for using jOOQ in your application.

3.3.1. jOOQ as a SQL builder

This is the most simple of all use cases, allowing for construction of valid SQL for any database. In this use case, you will not use jOOQ's code generator and probably not even jOOQ's query execution facilities. Instead, you'll use jOOQ's query DSL API to wrap strings, literals and other user-defined objects into an object-oriented, type-safe AST modelling your SQL statements. An example is given here:

```
// Fetch a SQL string from a jOOQ Query in order to manually execute it with another tool.
// For simplicity reasons, we're using the API to construct case-insensitive object references, here.
String sql = create.select(field("BOOK.TITLE"), field("AUTHOR.FIRST_NAME"), field("AUTHOR.LAST_NAME"))
                  .from(table("BOOK"))
                  .join(table("AUTHOR"))
                  .on(field("BOOK.AUTHOR_ID").eq(field("AUTHOR.ID")))
                  .where(field("BOOK.PUBLISHED_IN").eq(1948))
                  .getSQL();
```

The SQL string built with the jOOQ query DSL can then be executed using JDBC directly, using Spring's JdbcTemplate, using Apache DbUtils and many other tools (note that since jOOQ uses PreparedStatement by default, this will generate a bind variable for "1948". Read more about bind variables here).

If you wish to use jOOQ only as a SQL builder, the following sections of the manual will be of interest to you:

- SQL building: This section contains a lot of information about creating SQL statements using the jOOQ API
- Plain SQL: This section contains information useful in particular to those that want to supply table expressions, column expressions, etc. as plain SQL to jOOQ, rather than through generated artefacts

3.3.2. jOOQ as a SQL builder with code generation

In addition to using jOOQ as a standalone SQL builder, you can also use jOOQ's code generation features in order to compile your SQL statements using a Java compiler against an actual database schema. This adds a lot of power and expressiveness to just simply constructing SQL using the query DSL and custom strings and literals, as you can be sure that all database artefacts actually exist in the database, and that their type is correct. An example is given here:

```
// Fetch a SQL string from a jOOQ Query in order to manually execute it with another tool.
String sql = create.select(BOOK.TITLE, AUTHOR.FIRST_NAME, AUTHOR.LAST_NAME)
                  .from(BOOK)
                  .join(AUTHOR)
                  .on(BOOK.AUTHOR_ID.eq(AUTHOR.ID))
                  .where(BOOK.PUBLISHED_IN.eq(1948))
                  .getSQL();
```

The SQL string that you can generate as such can then be executed using JDBC directly, using Spring's JdbcTemplate, using Apache DbUtils and many other tools.

If you wish to use jOOQ only as a SQL builder with code generation, the following sections of the manual will be of interest to you:

- SQL building: This section contains a lot of information about creating SQL statements using the jOOQ API
- Code generation: This section contains the necessary information to run jOOQ's code generator against your developer database

3.3.3. jOOQ as a SQL executor

Instead of any tool mentioned in the previous chapters, you can also use jOOQ directly to execute your jOOQ-generated SQL statements. This will add a lot of convenience on top of the previously discussed API for typesafe SQL construction, when you can re-use the information from generated classes to fetch records and custom data types. An example is given here:

```
// Typesafely execute the SQL statement directly with jOOQ
Result<Record3<String, String, String>> result =
create.select(BOOK.TITLE, AUTHOR.FIRST_NAME, AUTHOR.LAST_NAME)
      .from(BOOK)
      .join(AUTHOR)
      .on(BOOK.AUTHOR_ID.eq(AUTHOR.ID))
      .where(BOOK.PUBLISHED_IN.eq(1948))
      .fetch();
```

By having jOOQ execute your SQL, the jOOQ query DSL becomes truly embedded SQL.

jOOQ doesn't stop here, though! You can execute any SQL with jOOQ. In other words, you can use any other SQL building tool and run the SQL statements with jOOQ. An example is given here:

```
// Use your favourite tool to construct SQL strings:
String sql = "SELECT title, first_name, last_name FROM book JOIN author ON book.author_id = author.id " +
             "WHERE book.published_in = 1984";

// Fetch results using jOOQ
Result<Record> result = create.fetch(sql);

// Or execute that SQL with JDBC, fetching the ResultSet with jOOQ:
ResultSet rs = connection.createStatement().executeQuery(sql);
Result<Record> result = create.fetch(rs);
```

If you wish to use jOOQ as a SQL executor with (or without) code generation, the following sections of the manual will be of interest to you:

- SQL building: This section contains a lot of information about creating SQL statements using the jOOQ API
- Code generation: This section contains the necessary information to run jOOQ's code generator against your developer database
- SQL execution: This section contains a lot of information about executing SQL statements using the jOOQ API
- Fetching: This section contains some useful information about the various ways of fetching data with jOOQ

3.3.4. jOOQ for CRUD

Apart from jOOQ's fluent API for query construction, jOOQ can also help you execute everyday CRUD operations. An example is given here:

```
// Fetch an author
AuthorRecord author : create.fetchOne(AUTHOR, AUTHOR.ID.eq(1));

// Create a new author, if it doesn't exist yet
if (author == null) {
    author = create.newRecord(AUTHOR);
    author.setId(1);
    author.setFirstName("Dan");
    author.setLastName("Brown");
}

// Mark the author as a "distinguished" author and store it
author.setDistinguished(1);

// Executes an update on existing authors, or insert on new ones
author.store();
```

If you wish to use all of jOOQ's features, the following sections of the manual will be of interest to you (including all sub-sections):

- SQL building: This section contains a lot of information about creating SQL statements using the jOOQ API
- Code generation: This section contains the necessary information to run jOOQ's code generator against your developer database
- SQL execution: This section contains a lot of information about executing SQL statements using the jOOQ API

3.3.5. jOOQ for PROs

jOOQ isn't just a library that helps you build and execute SQL against your generated, compilable schema. jOOQ ships with a lot of tools. Here are some of the most important tools shipped with jOOQ:

- jOOQ's Execute Listeners: jOOQ allows you to hook your custom execute listeners into jOOQ's SQL statement execution lifecycle in order to centrally coordinate any arbitrary operation performed on SQL being executed. Use this for logging, identity generation, SQL tracing, performance measurements, etc.
- Logging: jOOQ has a standard DEBUG logger built-in, for logging and tracing all your executed SQL statements and fetched result sets
- Stored Procedures: jOOQ supports stored procedures and functions of your favourite database. All routines and user-defined types are generated and can be included in jOOQ's SQL building API as function references.
- Batch execution: Batch execution is important when executing a big load of SQL statements. jOOQ simplifies these operations compared to JDBC
- Exporting and Importing: jOOQ ships with an API to easily export/import data in various formats

If you're a power user of your favourite, feature-rich database, jOOQ will help you access all of your database's vendor-specific features, such as OLAP features, stored procedures, user-defined types, vendor-specific SQL, functions, etc. Examples are given throughout this manual.

3.4. Tutorials

Don't have time to read the full manual? Here are a couple of tutorials that will get you into the most essential parts of jOOQ as quick as possible.

3.4.1. jOOQ in 7 easy steps

This manual section is intended for new users, to help them get a running application with jOOQ, quickly.

3.4.1.1. Step 1: Preparation

If you haven't already downloaded it, download jOOQ:
http://www.jooq.org/download

Alternatively, you can create a Maven dependency to download jOOQ artefacts:

Open Source Edition

```
<dependency>
  <groupId>org.jooq</groupId>
  <artifactId>jooq</artifactId>
  <version>3.11.11</version>
</dependency>
<dependency>
  <groupId>org.jooq</groupId>
  <artifactId>jooq-meta</artifactId>
  <version>3.11.11</version>
</dependency>
<dependency>
  <groupId>org.jooq</groupId>
  <artifactId>jooq-codegen</artifactId>
  <version>3.11.11</version>
</dependency>
```

Commercial Editions (Java 8+)

```
<!-- Note: These aren't hosted on Maven Central. Import them manually from your distribution -->
<dependency>
  <groupId>org.jooq.pro</groupId>
  <artifactId>jooq</artifactId>
  <version>3.11.11</version>
</dependency>
<dependency>
  <groupId>org.jooq.pro</groupId>
  <artifactId>jooq-meta</artifactId>
  <version>3.11.11</version>
</dependency>
<dependency>
  <groupId>org.jooq.pro</groupId>
  <artifactId>jooq-codegen</artifactId>
  <version>3.11.11</version>
</dependency>
```

Commercial Editions (Java 6+)

```
<!-- Note: These aren't hosted on Maven Central. Import them manually from your distribution -->
<dependency>
  <groupId>org.jooq.pro-java-6</groupId>
  <artifactId>jooq</artifactId>
  <version>3.11.11</version>
</dependency>
<dependency>
  <groupId>org.jooq.pro-java-6</groupId>
  <artifactId>jooq-meta</artifactId>
  <version>3.11.11</version>
</dependency>
<dependency>
  <groupId>org.jooq.pro-java-6</groupId>
  <artifactId>jooq-codegen</artifactId>
  <version>3.11.11</version>
</dependency>
```

Commercial Editions (Free Trial)

```
<!-- Note: These aren't hosted on Maven Central. Import them manually from your distribution -->
<dependency>
  <groupId>org.jooq.trial</groupId>
  <artifactId>jooq</artifactId>
  <version>3.11.11</version>
</dependency>
<dependency>
  <groupId>org.jooq.trial</groupId>
  <artifactId>jooq-meta</artifactId>
  <version>3.11.11</version>
</dependency>
<dependency>
  <groupId>org.jooq.trial</groupId>
  <artifactId>jooq-codegen</artifactId>
  <version>3.11.11</version>
</dependency>
```

Note that only the jOOQ Open Source Edition is available from Maven Central. If you're using the jOOQ Professional Edition or the jOOQ Enterprise Edition, you will have to manually install jOOQ in your local Nexus, or in your local Maven cache. For more information, please refer to the licensing pages.

Please refer to the manual's section about Code generation configuration to learn how to use jOOQ's code generator with Maven.

For this example, we'll be using MySQL. If you haven't already downloaded MySQL Connector/J, download it here:
http://dev.mysql.com/downloads/connector/j/

If you don't have a MySQL instance up and running yet, get XAMPP now! XAMPP is a simple installation bundle for Apache, MySQL, PHP and Perl

3.4.1.2. Step 2: Your database

We're going to create a database called "library" and a corresponding "author" table. Connect to MySQL via your command line client and type the following:

```
CREATE DATABASE `library`;

USE `library`;

CREATE TABLE `author` (
  `id` int NOT NULL,
  `first_name` varchar(255) DEFAULT NULL,
  `last_name` varchar(255) DEFAULT NULL,
  PRIMARY KEY (`id`)
);
```

3.4.1.3. Step 3: Code generation

In this step, we're going to use jOOQ's command line tools to generate classes that map to the Author table we just created. More detailed information about how to set up the jOOQ code generator can be found here:
jOOQ manual pages about setting up the code generator

The easiest way to generate a schema is to copy the jOOQ jar files (there should be 3) and the MySQL Connector jar file to a temporary directory. Then, create a library.xml that looks like this:

```
<?xml version="1.0" encoding="UTF-8" standalone="yes"?>
<configuration xmlns="http://www.jooq.org/xsd/jooq-codegen-3.11.0.xsd">
  <!-- Configure the database connection here -->
  <jdbc>
    <driver>com.mysql.jdbc.Driver</driver>
    <url>jdbc:mysql://localhost:3306/library</url>
    <user>root</user>
    <password></password>
  </jdbc>

  <generator>
    <!-- The default code generator. You can override this one, to generate your own code style.
         Supported generators:
         - org.jooq.codegen.JavaGenerator
         - org.jooq.codegen.ScalaGenerator
         Defaults to org.jooq.codegen.JavaGenerator -->
    <name>org.jooq.codegen.JavaGenerator</name>

    <database>
      <!-- The database type. The format here is:
           org.jooq.meta.[database].[database]Database -->
      <name>org.jooq.meta.mysql.MySQLDatabase</name>

      <!-- The database schema (or in the absence of schema support, in your RDBMS this
           can be the owner, user, database name) to be generated -->
      <inputSchema>library</inputSchema>

      <!-- All elements that are generated from your schema
           (A Java regular expression. Use the pipe to separate several expressions)
           Watch out for case-sensitivity. Depending on your database, this might be important! -->
      <includes>.*</includes>

      <!-- All elements that are excluded from your schema
           (A Java regular expression. Use the pipe to separate several expressions).
           Excludes match before includes, i.e. excludes have a higher priority -->
      <excludes></excludes>
    </database>

    <target>
      <!-- The destination package of your generated classes (within the destination directory) -->
      <packageName>test.generated</packageName>

      <!-- The destination directory of your generated classes. Using Maven directory layout here -->
      <directory>C:/workspace/MySQLTest/src/main/java</directory>
    </target>
  </generator>
</configuration>
```

Replace the username with whatever user has the appropriate privileges to query the database meta data. You'll also want to look at the other values and replace as necessary. Here are the two interesting properties:

generator.target.package - set this to the parent package you want to create for the generated classes. The setting of test.generated will cause the test.generated.Author and test.generated.AuthorRecord to be created

generator.target.directory - the directory to output to.

Once you have the JAR files and library.xml in your temp directory, type this on a Windows machine:

```
java -classpath jooq-3.11.11.jar;jooq-meta-3.11.11.jar;jooq-codegen-3.11.11.jar;mysql-connector-java-5.1.18-bin.jar;.
  org.jooq.codegen.GenerationTool library.xml
```

... or type this on a UNIX / Linux / Mac system (colons instead of semi-colons):

```
java -classpath jooq-3.11.11.jar:jooq-meta-3.11.11.jar:jooq-codegen-3.11.11.jar:mysql-connector-java-5.1.18-bin.jar:.
  org.jooq.codegen.GenerationTool library.xml
```

Note: jOOQ will try loading the library.xml from your classpath. This is also why there is a trailing period (.) on the classpath. If the file cannot be found on the classpath, jOOQ will look on the file system from the current working directory.

Replace the filenames with your actual filenames. In this example, jOOQ 3.11.11 is being used. If everything has worked, you should see this in your console output:

```
Nov 1, 2011 7:25:06 PM org.jooq.impl.JooqLogger info
INFO: Initialising properties  : /library.xml
Nov 1, 2011 7:25:07 PM org.jooq.impl.JooqLogger info
INFO: Database parameters
Nov 1, 2011 7:25:07 PM org.jooq.impl.JooqLogger info
INFO: ----------------------------------------------------------
Nov 1, 2011 7:25:07 PM org.jooq.impl.JooqLogger info
INFO:   dialect               : MYSQL
Nov 1, 2011 7:25:07 PM org.jooq.impl.JooqLogger info
INFO:   schema                : library
Nov 1, 2011 7:25:07 PM org.jooq.impl.JooqLogger info
INFO:   target dir            : C:/workspace/MySQLTest/src
Nov 1, 2011 7:25:07 PM org.jooq.impl.JooqLogger info
INFO:   target package        : test.generated
Nov 1, 2011 7:25:07 PM org.jooq.impl.JooqLogger info
INFO: ----------------------------------------------------------
Nov 1, 2011 7:25:07 PM org.jooq.impl.JooqLogger info
INFO: Emptying                : C:/workspace/MySQLTest/src/test/generated
Nov 1, 2011 7:25:07 PM org.jooq.impl.JooqLogger info
INFO: Generating classes in   : C:/workspace/MySQLTest/src/test/generated
Nov 1, 2011 7:25:07 PM org.jooq.impl.JooqLogger info
INFO: Generating schema       : Library.java
Nov 1, 2011 7:25:07 PM org.jooq.impl.JooqLogger info
INFO: Schema generated        : Total: 122.18ms
Nov 1, 2011 7:25:07 PM org.jooq.impl.JooqLogger info
INFO: Sequences fetched       : 0 (0 included, 0 excluded)
Nov 1, 2011 7:25:07 PM org.jooq.impl.JooqLogger info
INFO: Tables fetched          : 5 (5 included, 0 excluded)
Nov 1, 2011 7:25:07 PM org.jooq.impl.JooqLogger info
INFO: Generating tables       : C:/workspace/MySQLTest/src/test/generated/tables
Nov 1, 2011 7:25:07 PM org.jooq.impl.JooqLogger info
INFO: ARRAYs fetched          : 0 (0 included, 0 excluded)
Nov 1, 2011 7:25:07 PM org.jooq.impl.JooqLogger info
INFO: Enums fetched           : 0 (0 included, 0 excluded)
Nov 1, 2011 7:25:07 PM org.jooq.impl.JooqLogger info
INFO: UDTs fetched            : 0 (0 included, 0 excluded)
Nov 1, 2011 7:25:07 PM org.jooq.impl.JooqLogger info
INFO: Generating table        : Author.java
Nov 1, 2011 7:25:07 PM org.jooq.impl.JooqLogger info
INFO: Tables generated        : Total: 680.464ms, +558.284ms
Nov 1, 2011 7:25:07 PM org.jooq.impl.JooqLogger info
INFO: Generating Keys         : C:/workspace/MySQLTest/src/test/generated/tables
Nov 1, 2011 7:25:08 PM org.jooq.impl.JooqLogger info
INFO: Keys generated          : Total: 718.621ms, +38.157ms
Nov 1, 2011 7:25:08 PM org.jooq.impl.JooqLogger info
INFO: Generating records      : C:/workspace/MySQLTest/src/test/generated/tables/records
Nov 1, 2011 7:25:08 PM org.jooq.impl.JooqLogger info
INFO: Generating record       : AuthorRecord.java
Nov 1, 2011 7:25:08 PM org.jooq.impl.JooqLogger info
INFO: Table records generated  : Total: 782.545ms, +63.924ms
Nov 1, 2011 7:25:08 PM org.jooq.impl.JooqLogger info
INFO: Routines fetched        : 0 (0 included, 0 excluded)
Nov 1, 2011 7:25:08 PM org.jooq.impl.JooqLogger info
INFO: Packages fetched        : 0 (0 included, 0 excluded)
Nov 1, 2011 7:25:08 PM org.jooq.impl.JooqLogger info
INFO: GENERATION FINISHED!    : Total: 791.688ms, +9.143ms
```

3.4.1.4. Step 4: Connect to your database

Let's just write a vanilla main class in the project containing the generated classes:

```
// For convenience, always static import your generated tables and jOOQ functions to decrease verbosity:
import static test.generated.Tables.*;
import static org.jooq.impl.DSL.*;

import java.sql.*;

public class Main {
    public static void main(String[] args) {
        String userName = "root";
        String password = "";
        String url = "jdbc:mysql://localhost:3306/library";

        // Connection is the only JDBC resource that we need
        // PreparedStatement and ResultSet are handled by jOOQ, internally
        try (Connection conn = DriverManager.getConnection(url, userName, password)) {
            // ...
        }

        // For the sake of this tutorial, let's keep exception handling simple
        catch (Exception e) {
            e.printStackTrace();
        }
    }
}
```

This is pretty standard code for establishing a MySQL connection.

3.4.1.5. Step 5: Querying

Let's add a simple query constructed with jOOQ's query DSL:

```
DSLContext create = DSL.using(conn, SQLDialect.MYSQL);
Result<Record> result = create.select().from(AUTHOR).fetch();
```

First get an instance of DSLContext so we can write a simple SELECT query. We pass an instance of the MySQL connection to DSL. Note that the DSLContext doesn't close the connection. We'll have to do that ourselves.

We then use jOOQ's query DSL to return an instance of Result. We'll be using this result in the next step.

3.4.1.6. Step 6: Iterating

After the line where we retrieve the results, let's iterate over the results and print out the data:

```
for (Record r : result) {
    Integer id = r.getValue(AUTHOR.ID);
    String firstName = r.getValue(AUTHOR.FIRST_NAME);
    String lastName = r.getValue(AUTHOR.LAST_NAME);

    System.out.println("ID: " + id + " first name: " + firstName + " last name: " + lastName);
}
```

The full program should now look like this:

```
package test;

// For convenience, always static import your generated tables and
// jOOQ functions to decrease verbosity:
import static test.generated.Tables.*;
import static org.jooq.impl.DSL.*;

import java.sql.*;

import org.jooq.*;
import org.jooq.impl.*;

public class Main {

    /**
     * @param args
     */
    public static void main(String[] args) {
        String userName = "root";
        String password = "";
        String url = "jdbc:mysql://localhost:3306/library";

        // Connection is the only JDBC resource that we need
        // PreparedStatement and ResultSet are handled by jOOQ, internally
        try (Connection conn = DriverManager.getConnection(url, userName, password)) {
            DSLContext create = DSL.using(conn, SQLDialect.MYSQL);
            Result<Record> result = create.select().from(AUTHOR).fetch();

            for (Record r : result) {
                Integer id = r.getValue(AUTHOR.ID);
                String firstName = r.getValue(AUTHOR.FIRST_NAME);
                String lastName = r.getValue(AUTHOR.LAST_NAME);

                System.out.println("ID: " + id + " first name: " + firstName + " last name: " + lastName);
            }
        }

        // For the sake of this tutorial, let's keep exception handling simple
        catch (Exception e) {
            e.printStackTrace();
        }
    }
}
```

3.4.1.7. Step 7: Explore!

jOOQ has grown to be a comprehensive SQL library. For more information, please consider the documentation:
http://www.jooq.org/learn

... explore the Javadoc:
http://www.jooq.org/javadoc/latest/

... or join the news group:
https://groups.google.com/forum/#!forum/jooq-user

This tutorial is the courtesy of Ikai Lan. See the original source here:
http://ikaisays.com/2011/11/01/getting-started-with-jooq-a-tutorial/

3.4.2. Using jOOQ in modern IDEs

Feel free to contribute a tutorial!

3.4.3. Using jOOQ with Spring and Apache DBCP

jOOQ and Spring are easy to integrate. In this example, we shall integrate:

- Apache DBCP (but you may as well use some other connection pool, like BoneCP, C3P0, HikariCP, and various others).
- Spring TX as the transaction management library.
- jOOQ as the SQL building and execution library.

Before you copy the manual examples, consider also these further resources:

- The complete example can also be downloaded from GitHub.
- Another example using Spring and Guice for transaction management can be downloaded from GitHub.
- Another, excellent tutorial by Petri Kainulainen can be found here.

Add the required Maven dependencies

For this example, we'll create the following Maven dependencies

```
<!-- Use this or the latest Spring RELEASE version -->
<properties>
    <org.springframework.version>3.2.3.RELEASE</org.springframework.version>
</properties>

<dependencies>

    <!-- Database access -->
    <dependency>
        <!-- Use org.jooq            for the Open Source Edition
                 org.jooq.pro        for commercial editions,
                 org.jooq.pro-java-6 for commercial editions with Java 6 support,
                 org.jooq.trial      for the free trial edition

             Note: Only the Open Source Edition is hosted on Maven Central.
                   Import the others manually from your distribution -->
        <groupId>org.jooq</groupId>
        <artifactId>jooq</artifactId>
        <version>3.11.11</version>
    </dependency>
    <dependency>
        <groupId>org.apache.commons</groupId>
        <artifactId>commons-dbcp2</artifactId>
        <version>2.0</version>
    </dependency>
    <dependency>
        <groupId>com.h2database</groupId>
        <artifactId>h2</artifactId>
        <version>1.3.168</version>
    </dependency>

    <!-- Logging -->
    <dependency>
        <groupId>log4j</groupId>
        <artifactId>log4j</artifactId>
        <version>1.2.16</version>
    </dependency>
    <dependency>
        <groupId>org.slf4j</groupId>
        <artifactId>slf4j-log4j12</artifactId>
        <version>1.7.5</version>
    </dependency>

    <!-- Spring (transitive dependencies are not listed explicitly) -->
    <dependency>
        <groupId>org.springframework</groupId>
        <artifactId>spring-context</artifactId>
        <version>${org.springframework.version}</version>
    </dependency>
    <dependency>
        <groupId>org.springframework</groupId>
        <artifactId>spring-jdbc</artifactId>
        <version>${org.springframework.version}</version>
    </dependency>

    <!-- Testing -->
    <dependency>
        <groupId>junit</groupId>
        <artifactId>junit</artifactId>
        <version>4.11</version>
        <type>jar</type>
        <scope>test</scope>
    </dependency>
    <dependency>
        <groupId>org.springframework</groupId>
        <artifactId>spring-test</artifactId>
        <version>${org.springframework.version}</version>
        <scope>test</scope>
    </dependency>
</dependencies>
```

Note that only the jOOQ Open Source Edition is available from Maven Central. If you're using the jOOQ Professional Edition or the jOOQ Enterprise Edition, you will have to manually install jOOQ in your local Nexus, or in your local Maven cache. For more information, please refer to the licensing pages.

Create a minimal Spring configuration file

The above dependencies are configured together using a Spring Beans configuration:

```
<?xml version="1.0" encoding="UTF-8"?>

<beans xmlns="http://www.springframework.org/schema/beans"
       xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
       xmlns:tx="http://www.springframework.org/schema/tx"
       xsi:schemaLocation="
            http://www.springframework.org/schema/beans http://www.springframework.org/schema/beans/spring-beans.xsd
            http://www.springframework.org/schema/tx http://www.springframework.org/schema/tx/spring-tx-3.2.xsd">

    <!-- This is needed if you want to use the @Transactional annotation -->
    <tx:annotation-driven transaction-manager="transactionManager"/>

    <bean id="dataSource" class="org.apache.commons.dbcp2.BasicDataSource" destroy-method="close" >
        <!-- These properties are replaced by Maven "resources" -->
       <property name="url" value="${db.url}" />
       <property name="driverClassName" value="${db.driver}" />
       <property name="username" value="${db.username}" />
       <property name="password" value="${db.password}" />
    </bean>

    <!-- Configure Spring's transaction manager to use a DataSource -->
    <bean id="transactionManager"
        class="org.springframework.jdbc.datasource.DataSourceTransactionManager">
        <property name="dataSource" ref="dataSource" />
    </bean>

    <!-- Configure jOOQ's ConnectionProvider to use Spring's TransactionAwareDataSourceProxy,
         which can dynamically discover the transaction context -->
    <bean id="transactionAwareDataSource"
        class="org.springframework.jdbc.datasource.TransactionAwareDataSourceProxy">
        <constructor-arg ref="dataSource" />
    </bean>

    <bean class="org.jooq.impl.DataSourceConnectionProvider" name="connectionProvider">
        <constructor-arg ref="transactionAwareDataSource" />
    </bean>

    <!-- Configure the DSL object, optionally overriding jOOQ Exceptions with Spring Exceptions -->
    <bean id="dsl" class="org.jooq.impl.DefaultDSLContext">
        <constructor-arg ref="config" />
    </bean>

    <bean id="exceptionTranslator" class="org.jooq.example.spring.exception.ExceptionTranslator" />

    <!-- Invoking an internal, package-private constructor for the example
         Implement your own Configuration for more reliable behaviour -->
    <bean class="org.jooq.impl.DefaultConfiguration" name="config">
        <property name="SQLDialect"><value type="org.jooq.SQLDialect">H2</value></property>
        <property name="connectionProvider" ref="connectionProvider" />
        <property name="executeListenerProvider">
            <array>
                <bean class="org.jooq.impl.DefaultExecuteListenerProvider">
                    <constructor-arg index="0" ref="exceptionTranslator"/>
                </bean>
            </array>
        </property>
    </bean>

    <!-- This is the "business-logic" -->
    <bean id="books" class="org.jooq.example.spring.impl.DefaultBookService"/>
</beans>
```

Run a query using the above configuration:

With the above configuration, you should be ready to run queries pretty quickly. For instance, in an integration-test, you could use Spring to run JUnit:

```
@RunWith(SpringJUnit4ClassRunner.class)
@ContextConfiguration(locations = {"/jooq-spring.xml"})
public class QueryTest {

    @Autowired
    DSLContext create;

    @Test
    public void testJoin() throws Exception {
        // All of these tables were generated by jOOQ's Maven plugin
        Book b = BOOK.as("b");
        Author a = AUTHOR.as("a");
        BookStore s = BOOK_STORE.as("s");
        BookToBookStore t = BOOK_TO_BOOK_STORE.as("t");

        Result<Record3<String, String, Integer>> result =
        create.select(a.FIRST_NAME, a.LAST_NAME, countDistinct(s.NAME))
              .from(a)
              .join(b).on(b.AUTHOR_ID.eq(a.ID))
              .join(t).on(t.BOOK_ID.eq(b.ID))
              .join(s).on(t.BOOK_STORE_NAME.eq(s.NAME))
              .groupBy(a.FIRST_NAME, a.LAST_NAME)
              .orderBy(countDistinct(s.NAME).desc())
              .fetch();

        assertEquals(2, result.size());
        assertEquals("Paulo", result.getValue(0, a.FIRST_NAME));
        assertEquals("George", result.getValue(1, a.FIRST_NAME));

        assertEquals("Coelho", result.getValue(0, a.LAST_NAME));
        assertEquals("Orwell", result.getValue(1, a.LAST_NAME));

        assertEquals(Integer.valueOf(3), result.getValue(0, countDistinct(s.NAME)));
        assertEquals(Integer.valueOf(2), result.getValue(1, countDistinct(s.NAME)));
    }
}
```

Run a queries in an explicit transaction:

The following example shows how you can use Spring's TransactionManager to explicitly handle transactions:

```
@RunWith(SpringJUnit4ClassRunner.class)
@ContextConfiguration(locations = {"/jooq-spring.xml"})
@TransactionConfiguration(transactionManager="transactionManager")
public class TransactionTest {

    @Autowired DSLContext                  dsl;
    @Autowired DataSourceTransactionManager txMgr;
    @Autowired BookService                 books;

    @After
    public void teardown() {

        // Delete all books that were created in any test
        dsl.delete(BOOK).where(BOOK.ID.gt(4)).execute();
    }

    @Test
    public void testExplicitTransactions() {
        boolean rollback = false;

        TransactionStatus tx = txMgr.getTransaction(new DefaultTransactionDefinition());
        try {

            // This is a "bug". The same book is created twice, resulting in a
            // constraint violation exception
            for (int i = 0; i < 2; i++)
                dsl.insertInto(BOOK)
                   .set(BOOK.ID, 5)
                   .set(BOOK.AUTHOR_ID, 1)
                   .set(BOOK.TITLE, "Book 5")
                   .execute();

            Assert.fail();
        }

        // Upon the constraint violation, we explicitly roll back the transaction.
        catch (DataAccessException e) {
            txMgr.rollback(tx);
            rollback = true;
        }

        assertEquals(4, dsl.fetchCount(BOOK));
        assertTrue(rollback);
    }
}
```

Run queries using declarative transactions

Spring-TX has very powerful means to handle transactions declaratively, using the @Transactional annotation. The BookService that we had defined in the previous Spring configuration can be seen here:

```
public interface BookService {

    /**
     * Create a new book.
     * <p>
     * The implementation of this method has a bug, which causes this method to
     * fail and roll back the transaction.
     */
    @Transactional
    void create(int id, int authorId, String title);

}
```

And here is how we interact with it:

```
    @Test
    public void testDeclarativeTransactions() {
        boolean rollback = false;

        try {

            // The service has a "bug", resulting in a constraint violation exception
            books.create(5, 1, "Book 5");
            Assert.fail();
        }
        catch (DataAccessException ignore) {
            rollback = true;
        }

        assertEquals(4, dsl.fetchCount(BOOK));
        assertTrue(rollback);
    }
```

Run queries using jOOQ's transaction API

jOOQ has its own programmatic transaction API that can be used with Spring transactions by implementing the jOOQ org.jooq.TransactionProvider SPI and passing that to your jOOQ Configuration. More details about this transaction API can be found in the manual's section about transaction management.

You can try the above example yourself by downloading it from GitHub.

3.4.4. Using jOOQ with Flyway

When performing database migrations, we at Data Geekery recommend using jOOQ with Flyway - Database Migrations Made Easy. In this chapter, we're going to look into a simple way to get started with the two frameworks.

Philosophy

There are a variety of ways how jOOQ and Flyway could interact with each other in various development setups. In this tutorial we're going to show just one variant of such framework team play - a variant that we find particularly compelling for most use cases.

The general philosophy behind the following approach can be summarised as this:

- 1. Database increment
- 2. Database migration
- 3. Code re-generation
- 4. Development

The four steps above can be repeated time and again, every time you need to modify something in your database. More concretely, let's consider:

- 1. Database increment - You need a new column in your database, so you write the necessary DDL in a Flyway script
- 2. Database migration - This Flyway script is now part of your deliverable, which you can share with all developers who can migrate their databases with it, the next time they check out your change
- 3. Code re-generation - Once the database is migrated, you regenerate all jOOQ artefacts (see code generation), locally
- 4. Development - You continue developing your business logic, writing code against the udpated, generated database schema

Maven Project Configuration - Properties

The following properties are defined in our pom.xml, to be able to reuse them between plugin configurations:

```
<properties>
    <db.url>jdbc:h2:~/flyway-test</db.url>
    <db.username>sa</db.username>
</properties>
```

0. Maven Project Configuration - Dependencies

While jOOQ and Flyway could be used in standalone migration scripts, in this tutorial, we'll be using Maven for the standard project setup. You will also find the source code of this tutorial on GitHub at https://github.com/jOOQ/jOOQ/tree/master/jOOQ-examples/jOOQ-flyway-example, and the full pom.xml file here.

These are the dependencies that we're using in our Maven configuration:

```
<!-- We'll add the latest version of jOOQ and our JDBC driver - in this case H2 -->
<dependency>
    <!-- Use org.jooq            for the Open Source Edition
              org.jooq.pro        for commercial editions,
              org.jooq.pro-java-6 for commercial editions with Java 6 support,
              org.jooq.trial      for the free trial edition

         Note: Only the Open Source Edition is hosted on Maven Central.
               Import the others manually from your distribution -->
    <groupId>org.jooq</groupId>
    <artifactId>jooq</artifactId>
    <version>3.11.11</version>
</dependency>
<dependency>
    <groupId>com.h2database</groupId>
    <artifactId>h2</artifactId>
    <version>1.4.177</version>
</dependency>

<!-- For improved logging, we'll be using log4j via slf4j to see what's going on during migration and code generation -->
<dependency>
    <groupId>log4j</groupId>
    <artifactId>log4j</artifactId>
    <version>1.2.16</version>
</dependency>
<dependency>
    <groupId>org.slf4j</groupId>
    <artifactId>slf4j-log4j12</artifactId>
    <version>1.7.5</version>
</dependency>

<!-- To ensure our code is working, we're using JUnit -->
<dependency>
    <groupId>junit</groupId>
    <artifactId>junit</artifactId>
    <version>4.11</version>
    <scope>test</scope>
</dependency>
```

0. Maven Project Configuration - Plugins

After the dependencies, let's simply add the Flyway and jOOQ Maven plugins like so. The Flyway plugin:

```
<plugin>
    <groupId>org.flywaydb</groupId>
    <artifactId>flyway-maven-plugin</artifactId>
    <version>3.0</version>

    <!-- Note that we're executing the Flyway plugin in the "generate-sources" phase -->
    <executions>
        <execution>
            <phase>generate-sources</phase>
            <goals>
                <goal>migrate</goal>
            </goals>
        </execution>
    </executions>

    <!-- Note that we need to prefix the db/migration path with filesystem: to prevent Flyway
         from looking for our migration scripts only on the classpath -->
    <configuration>
        <url>${db.url}</url>
        <user>${db.username}</user>
        <locations>
            <location>filesystem:src/main/resources/db/migration</location>
        </locations>
    </configuration>
</plugin>
```

The above Flyway Maven plugin configuration will read and execute all database migration scripts from src/main/resources/db/migration prior to compiling Java source code. While the official Flyway documentation suggests that migrations be done in the compile phase, the jOOQ code generator relies on such migrations having been done ***prior*** to code generation.

After the Flyway plugin, we'll add the jOOQ Maven Plugin. For more details, please refer to the manual's section about the code generation configuration.

```
<plugin>
    <!-- Use org.jooq            for the Open Source Edition
              org.jooq.pro        for commercial editions,
              org.jooq.pro-java-6 for commercial editions with Java 6 support,
              org.jooq.trial      for the free trial edition

         Note: Only the Open Source Edition is hosted on Maven Central.
               Import the others manually from your distribution -->
    <groupId>org.jooq</groupId>
    <artifactId>jooq-codegen-maven</artifactId>
    <version>${org.jooq.version}</version>

    <!-- The jOOQ code generation plugin is also executed in the generate-sources phase, prior to compilation -->
    <executions>
        <execution>
            <phase>generate-sources</phase>
            <goals>
                <goal>generate</goal>
            </goals>
        </execution>
    </executions>

    <!-- This is a minimal working configuration. See the manual's section about the code generator for more details -->
    <configuration>
        <jdbc>
            <url>${db.url}</url>
            <user>${db.username}</user>
        </jdbc>
        <generator>
            <database>
                <includes>.*</includes>
                <inputSchema>FLYWAY_TEST</inputSchema>
            </database>
            <target>
                <packageName>org.jooq.example.flyway.db.h2</packageName>
                <directory>target/generated-sources/jooq-h2</directory>
            </target>
        </generator>
    </configuration>
</plugin>
```

This configuration will now read the FLYWAY_TEST schema and reverse-engineer it into the target/generated-sources/jooq-h2 directory, and within that, into the org.jooq.example.flyway.db.h2 package.

1. Database increments

Now, when we start developing our database. For that, we'll create database increment scripts, which we put into the src/main/resources/db/migration directory, as previously configured for the Flyway plugin. We'll add these files:

- V1__initialise_database.sql
- V2__create_author_table.sql
- V3__create_book_table_and_records.sql

These three scripts model our schema versions 1-3 (note the capital V!). Here are the scripts' contents

```
-- V1__initialise_database.sql
DROP SCHEMA flyway_test IF EXISTS;

CREATE SCHEMA flyway_test;
```

```
-- V2__create_author_table.sql
CREATE SEQUENCE flyway_test.s_author_id START WITH 1;

CREATE TABLE flyway_test.author (
  id INT NOT NULL,
  first_name VARCHAR(50),
  last_name VARCHAR(50) NOT NULL,
  date_of_birth DATE,
  year_of_birth INT,
  address VARCHAR(50),

  CONSTRAINT pk_author PRIMARY KEY (ID)
);
```

```
-- V3__create_book_table_and_records.sql
CREATE TABLE flyway_test.book (
  id INT NOT NULL,
  author_id INT NOT NULL,
  title VARCHAR(400) NOT NULL,

  CONSTRAINT pk_book PRIMARY KEY (id),
  CONSTRAINT fk_book_author_id FOREIGN KEY (author_id) REFERENCES flyway_test.author(id)
);


INSERT INTO flyway_test.author VALUES (next value for flyway_test.s_author_id, 'George', 'Orwell', '1903-06-25', 1903, null);
INSERT INTO flyway_test.author VALUES (next value for flyway_test.s_author_id, 'Paulo', 'Coelho', '1947-08-24', 1947, null);

INSERT INTO flyway_test.book VALUES (1, 1, '1984');
INSERT INTO flyway_test.book VALUES (2, 1, 'Animal Farm');
INSERT INTO flyway_test.book VALUES (3, 2, 'O Alquimista');
INSERT INTO flyway_test.book VALUES (4, 2, 'Brida');
```

2. Database migration and 3. Code regeneration

The above three scripts are picked up by Flyway and executed in the order of the versions. This can be seen very simply by executing:

```
mvn clean install
```

And then observing the log output from Flyway...

```
[INFO] --- flyway-maven-plugin:3.0:migrate (default) @ jooq-flyway-example ---
[INFO] Database: jdbc:h2:~/flyway-test (H2 1.4)
[INFO] Validated 3 migrations (execution time 00:00.004s)
[INFO] Creating Metadata table: "PUBLIC"."schema_version"
[INFO] Current version of schema "PUBLIC": << Empty Schema >>
[INFO] Migrating schema "PUBLIC" to version 1
[INFO] Migrating schema "PUBLIC" to version 2
[INFO] Migrating schema "PUBLIC" to version 3
[INFO] Successfully applied 3 migrations to schema "PUBLIC" (execution time 00:00.073s).
```

... and from jOOQ on the console:

```
[INFO] --- jooq-codegen-maven:3.11.11:generate (default) @ jooq-flyway-example ---
[INFO] --- jooq-codegen-maven:3.11.11:generate (default) @ jooq-flyway-example ---
[INFO] Using this configuration:
...
[INFO] Generating schemata      : Total: 1
[INFO] Generating schema        : FlywayTest.java
[INFO] ------------------------------------------------------------
[....]
[INFO] GENERATION FINISHED!     : Total: 337.576ms, +4.299ms
```

4. Development

Note that all of the previous steps are executed automatically, every time someone adds new migration scripts to the Maven module. For instance, a team member might have committed a new migration script, you check it out, rebuild and get the latest jOOQ-generated sources for your own development or integration-test database.

Now, that these steps are done, you can proceed writing your database queries. Imagine the following test case

```
import org.jooq.Result;
import org.jooq.impl.DSL;
import org.junit.Test;

import java.sql.DriverManager;

import static java.util.Arrays.asList;
import static org.jooq.example.flyway.db.h2.Tables.*;
import static org.junit.Assert.assertEquals;

public class AfterMigrationTest {

    @Test
    public void testQueryingAfterMigration() throws Exception {
        try (Connection c = DriverManager.getConnection("jdbc:h2:~/flyway-test", "sa", "")) {
            Result<?> result =
            DSL.using(c)
               .select(
                   AUTHOR.FIRST_NAME,
                   AUTHOR.LAST_NAME,
                   BOOK.ID,
                   BOOK.TITLE
               )
               .from(AUTHOR)
               .join(BOOK)
               .on(AUTHOR.ID.eq(BOOK.AUTHOR_ID))
               .orderBy(BOOK.ID.asc())
               .fetch();

            assertEquals(4, result.size());
            assertEquals(asList(1, 2, 3, 4), result.getValues(BOOK.ID));
        }
    }
}
```

Reiterate

The power of this approach becomes clear once you start performing database modifications this way. Let's assume that the French guy on our team prefers to have things his way:

```
-- V4__le_french.sql
ALTER TABLE flyway_test.book ALTER COLUMN title RENAME TO le_titre;
```

They check it in, you check out the new database migration script, run

```
mvn clean install
```

And then observing the log output:

```
[INFO] --- flyway-maven-plugin:3.0:migrate (default) @ jooq-flyway-example ---
[INFO] --- flyway-maven-plugin:3.0:migrate (default) @ jooq-flyway-example ---
[INFO] Database: jdbc:h2:~/flyway-test (H2 1.4)
[INFO] Validated 4 migrations (execution time 00:00.005s)
[INFO] Current version of schema "PUBLIC": 3
[INFO] Migrating schema "PUBLIC" to version 4
[INFO] Successfully applied 1 migration to schema "PUBLIC" (execution time 00:00.016s).
```

So far so good, but later on:

```
[ERROR] COMPILATION ERROR :
[INFO] -------------------------------------------------------------
[ERROR] C:\...\jOOQ-flyway-example\src\test\java\AfterMigrationTest.java:[24,19] error: cannot find symbol
[INFO] 1 error
```

When we go back to our Java integration test, we can immediately see that the TITLE column is still being referenced, but it no longer exists:

```
public class AfterMigrationTest {

    @Test
    public void testQueryingAfterMigration() throws Exception {
        try (Connection c = DriverManager.getConnection("jdbc:h2:~/flyway-test", "sa", "")) {
            Result<?> result =
            DSL.using(c)
                .select(
                    AUTHOR.FIRST_NAME,
                    AUTHOR.LAST_NAME,
                    BOOK.ID,
                    BOOK.TITLE
                    //   ^^^^^ This column no longer exists. We'll have to rename it to LE_TITRE
                )
                .from(AUTHOR)
                .join(BOOK)
                .on(AUTHOR.ID.eq(BOOK.AUTHOR_ID))
                .orderBy(BOOK.ID.asc())
                .fetch();

            assertEquals(4, result.size());
            assertEquals(asList(1, 2, 3, 4), result.getValues(BOOK.ID));
        }
    }
}
```

Conclusion

This tutorial shows very easily how you can build a rock-solid development process using Flyway and jOOQ to prevent SQL-related errors very early in your development lifecycle - immediately at compile time, rather than in production!

Please, visit the Flyway website for more information about Flyway.

3.4.5. Using jOOQ with JAX-RS

In some use-cases, having a lean, single-tier server-side architecture is desirable. Typically, such architectures expose a RESTful API implementing client code and the UI using something like AngularJS.

In Java, the standard API for RESTful applications is JAX-RS, which is part of JEE 7, along with a standard JSON implementation. But you can use JAX-RS also outside of a JEE container. The following example shows how to set up a simple license server using these technologies:

- Maven for building and running
- Jetty as a lightweight Servlet implementation
- Jersey, the JAX-RS (JSR 311 & JSR 339) reference implementation
- jOOQ as a data access layer

For the example, we'll use a PostgreSQL database.

Creating the license server database

We'll keep the example simple and use a LICENSE table to store all license keys and associated information, whereas a LOG_VERIFY table is used to log access to the license server. Here's the DDL:

```
CREATE TABLE LICENSE_SERVER.LICENSE (
  ID           SERIAL8      NOT NULL,

  LICENSE_DATE TIMESTAMP    NOT NULL,              -- The date when the license was issued
  LICENSEE     TEXT         NOT NULL,              -- The e-mail address of the licensee
  LICENSE      TEXT         NOT NULL,              -- The license key
  VERSION      VARCHAR(50)  NOT NULL DEFAULT '.*', -- The licensed version(s), a regular expression

  CONSTRAINT PK_LICENSE PRIMARY KEY (ID),
  CONSTRAINT UK_LICENSE UNIQUE (LICENSE)
);

CREATE TABLE LICENSE_SERVER.LOG_VERIFY (
  ID           SERIAL8      NOT NULL,

  LICENSEE     TEXT         NOT NULL,              -- The licensee whose license is being verified
  LICENSE      TEXT         NOT NULL,              -- The license key that is being verified
  REQUEST_IP   VARCHAR(50)  NOT NULL,              -- The request IP verifying the license
  VERSION      VARCHAR(50)  NOT NULL,              -- The version that is being verified
  MATCH        BOOLEAN      NOT NULL,              -- Whether the verification was successful

  CONSTRAINT PK_LOG_VERIFY PRIMARY KEY (ID)
);
```

To make things a bit more interesting (and secure), we'll also push license key generation into the database, by generating it from a stored function as such:

```
CREATE OR REPLACE FUNCTION LICENSE_SERVER.GENERATE_KEY(
    IN license_date TIMESTAMP WITH TIME ZONE,
    IN email TEXT
) RETURNS VARCHAR
AS $$
BEGIN
    RETURN 'license-key';
END;
$$ LANGUAGE PLPGSQL;
```

The actual algorithm might be using a secret salt to hash the function arguments. For the sake of a tutorial, a constant string will suffice.

Setting up the project

We're going to be setting up the jOOQ code generator using Maven

```
<?xml version="1.0" encoding="UTF-8"?>
<project xmlns="http://maven.apache.org/POM/4.0.0"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xsi:schemaLocation="http://maven.apache.org/POM/4.0.0 http://maven.apache.org/maven-v4_0_0.xsd">
    <modelVersion>4.0.0</modelVersion>

    <groupId>org.jooq</groupId>
    <artifactId>jooq-webservices</artifactId>
    <packaging>war</packaging>
    <version>1.0</version>

    <build>
        <plugins>
            <plugin>
                <groupId>org.apache.maven.plugins</groupId>
                <artifactId>maven-compiler-plugin</artifactId>
                <version>2.0.2</version>
                <configuration>
                    <source>1.7</source>
                    <target>1.7</target>
                </configuration>
            </plugin>

            <plugin>
                <groupId>org.mortbay.jetty</groupId>
                <artifactId>maven-jetty-plugin</artifactId>
                <version>6.1.26</version>
                <configuration>
                    <reload>manual</reload>
                    <stopKey>stop</stopKey>
                    <stopPort>9966</stopPort>
                </configuration>
            </plugin>

            <plugin>
                <!-- Use org.jooq            for the Open Source Edition
                          org.jooq.pro        for commercial editions,
                          org.jooq.pro-java-6 for commercial editions with Java 6 support,
                          org.jooq.trial      for the free trial edition

                     Note: Only the Open Source Edition is hosted on Maven Central.
                           Import the others manually from your distribution -->
                <groupId>org.jooq</groupId>
                <artifactId>jooq-codegen-maven</artifactId>
                <version>3.11.11</version>

                <!-- See GitHub for details -->
            </plugin>
        </plugins>
    </build>

    <dependencies>
        <dependency>
            <groupId>com.sun.jersey</groupId>
            <artifactId>jersey-server</artifactId>
            <version>1.0.2</version>
        </dependency>
        <dependency>
            <groupId>com.sun.jersey</groupId>
            <artifactId>jersey-json</artifactId>
            <version>1.0.2</version>
        </dependency>
        <dependency>
            <groupId>com.sun.jersey.contribs</groupId>
            <artifactId>jersey-spring</artifactId>
            <version>1.0.2</version>
        </dependency>
        <dependency>
            <groupId>javax.servlet</groupId>
            <artifactId>servlet-api</artifactId>
            <version>2.5</version>
        </dependency>

        <dependency>
            <!-- Use org.jooq            for the Open Source Edition
                      org.jooq.pro        for commercial editions,
                      org.jooq.pro-java-6 for commercial editions with Java 6 support,
                      org.jooq.trial      for the free trial edition

                 Note: Only the Open Source Edition is hosted on Maven Central.
                       Import the others manually from your distribution -->
            <groupId>org.jooq</groupId>
            <artifactId>jooq</artifactId>
            <version>3.11.11</version>
        </dependency>
        <dependency>
            <groupId>org.postgresql</groupId>
            <artifactId>postgresql</artifactId>
            <version>9.4.1212</version>
        </dependency>
        <dependency>
            <groupId>log4j</groupId>
            <artifactId>log4j</artifactId>
            <version>1.2.16</version>
        </dependency>
    </dependencies>
</project>
```

With the above setup, we're now pretty ready to start developing our license service as a JAX-RS service.

The license service class

Once we've run the jOOQ code generator using Maven, we can write the following service class:

```
/**
 * The license server.
 */
@Path("/license/")
@Component
@Scope("request")
public class LicenseService {

    /**
     * <code>/license/generate</code> generates and returns a new license key.
     *
     * @param mail The input email address of the licensee.
     */
    @GET
    @Produces("text/plain")
    @Path("/generate")
    public String generate(
        final @QueryParam("mail") String mail
    ) {
        return run(new CtxRunnable() {

            @Override
            public String run(DSLContext ctx) {
                Timestamp licenseDate = new Timestamp(System.currentTimeMillis());

                // Use the jOOQ query DSL API to generate a license key
                return
                ctx.insertInto(LICENSE)
                   .set(LICENSE.LICENSE_, generateKey(inline(licenseDate), inline(mail)))
                   .set(LICENSE.LICENSE_DATE, licenseDate)
                   .set(LICENSE.LICENSEE, mail)
                   .returning()
                   .fetchOne()
                   .getLicense();
            }
        });
    }

    /**
     * <code>/license/verify</code> checks if a given licensee has access to version using a license.
     *
     * @param request The servlet request from the JAX-RS context.
     * @param mail The input email address of the licensee.
     * @param license The license used by the licensee.
     * @param version The product version being accessed.
     */
    @GET
    @Produces("text/plain")
    @Path("/verify")
    public String verify(
        final @Context HttpServletRequest request,
        final @QueryParam("mail") String mail,
        final @QueryParam("license") String license,
        final @QueryParam("version") String version
    ) {
        return run(new CtxRunnable() {
            @Override
            public String run(DSLContext ctx) {
                String v = (version == null || version.equals("")) ? "" : version;

                // Use the jOOQ query DSL API to generate a log entry
                return
                ctx.insertInto(LOG_VERIFY)
                   .set(LOG_VERIFY.LICENSE, license)
                   .set(LOG_VERIFY.LICENSEE, mail)
                   .set(LOG_VERIFY.REQUEST_IP, request.getRemoteAddr())
                   .set(LOG_VERIFY.MATCH, field(
                           selectCount()
                          .from(LICENSE)
                          .where(LICENSE.LICENSEE.eq(mail))
                          .and(LICENSE.LICENSE_.eq(license))
                          .and(val(v).likeRegex(LICENSE.VERSION))
                          .asField().gt(0)))
                   .set(LOG_VERIFY.VERSION, v)
                   .returning(LOG_VERIFY.MATCH)
                   .fetchOne()
                   .getValue(LOG_VERIFY.MATCH, String.class);
            }
        });
    }

    // [...]
}
```

The INSERT INTO LOG_VERIFY query is actually rather interesting. In plain SQL, it would look like this:

```
INSERT INTO LOG_VERIFY (LICENSE, LICENSEE, REQUEST_IP, MATCH, VERSION)
VALUES (
  :license,
  :mail,
  :remoteAddr,
  (SELECT COUNT(*) FROM LICENSE WHERE LICENSEE = :mail AND LICENSE = :license AND :version ~ VERSION) > 0,
  :version
)
RETURNING MATCH;
```

Apart from the foregoing, the LicenseService also contains a couple of simple utilities:

```
    /**
     * This method encapsulates a transaction and initialises a jOOQ DSLcontext.
     * This could also be achieved with Spring and DBCP for connection pooling.
     */
    private String run(CtxRunnable runnable) {
        try (Connection c = getConnection("jdbc:postgresql:postgres", "postgres", System.getProperty("pw", "test"))) {
            DSLContext ctx = DSL.using(new DefaultConfiguration()
                    .set(new DefaultConnectionProvider(c))
                    .set(SQLDialect.POSTGRES)
                    .set(new Settings().withExecuteLogging(false)));

            return runnable.run(ctx);
        }
        catch (Exception e) {
            e.printStackTrace();
            Response.status(Status.SERVICE_UNAVAILABLE);
            return "Service Unavailable - Please contact support@datageekery.com for help";
        }
    }

    private interface CtxRunnable {
        String run(DSLContext ctx);
    }
```

Configuring Spring and Jetty

All we need now is to configure Spring...

```
<beans xmlns="http://www.springframework.org/schema/beans"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xmlns:context="http://www.springframework.org/schema/context"
    xsi:schemaLocation="
http://www.springframework.org/schema/beans http://www.springframework.org/schema/beans/spring-beans-2.5.xsd
http://www.springframework.org/schema/context http://www.springframework.org/schema/context/spring-context-2.5.xsd">

    <context:component-scan base-package="org.jooq.example.jaxrs" />

</beans>
```

... and Jetty ...

```
<?xml version="1.0" encoding="UTF-8"?>
<web-app version="2.4" xmlns="http://java.sun.com/xml/ns/j2ee"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xsi:schemaLocation="http://java.sun.com/xml/ns/j2ee http://java.sun.com/xml/ns/j2ee/web-app_2_4.xsd">

    <context-param>
        <param-name>contextConfigLocation</param-name>
        <param-value>classpath:applicationContext.xml</param-value>
    </context-param>
    <listener>
        <listener-class>org.springframework.web.context.ContextLoaderListener</listener-class>
    </listener>
    <listener>
        <listener-class>org.springframework.web.context.request.RequestContextListener</listener-class>
    </listener>
    <servlet>
        <servlet-name>Jersey Spring Web Application</servlet-name>
        <servlet-class>com.sun.jersey.spi.spring.container.servlet.SpringServlet</servlet-class>
    </servlet>
    <servlet-mapping>
        <servlet-name>Jersey Spring Web Application</servlet-name>
        <url-pattern>/*</url-pattern>
    </servlet-mapping>
</web-app>
```

... and we're done! We can now run the server with the following command:

```
mvn jetty:run
```

Or if you need a custom port:

```
mvn jetty:run -Djetty.port=8088
```

Using the license server

You can now use the license server at the following URLs

```
http://localhost:8088/jooq-jax-rs-example/license/generate?mail=test@example.com
-> license-key

http://localhost:8088/jooq-jax-rs-example/license/verify?mail=test@example.com&license=license-key&version=3.2.0
-> true

http://localhost:8088/jooq-jax-rs-example/license/verify?mail=test@example.com&license=wrong&version=3.2.0
-> false
```

Let's verify what happened, in the database:

```
select * from license_server.license
-- id | license_date            | licensee         | license     | version
-----------------------------------------------------------------------------
--  3 | 2013-11-22 14:26:07.768 | test@example.com | license-key | .*

select * from license_server.log_verify
-- id | licensee         | license     | request_ip      | version | match
-------------------------------------------------------------------------
--  2 | test@example.com | license-key | 0:0:0:0:0:0:0:1 | 3.2.0   | t
--  5 | test@example.com | wrong       | 0:0:0:0:0:0:0:1 | 3.2.0   | f
```

Downloading the complete example

The complete example can be downloaded for free and under the terms of the Apache Software License 2.0 from here:
https://github.com/jOOQ/jOOQ/tree/master/jOOQ-examples/jOOQ-jax-rs-example

3.4.6. A simple web application with jOOQ

Feel free to contribute a tutorial!

3.5. jOOQ and Java 8

Java 8 has introduced a great set of enhancements, among which lambda expressions and the new java.util.stream.Stream. These new constructs align very well with jOOQ's fluent API as can be seen in the following examples:

jOOQ and lambda expressions

jOOQ's RecordMapper API is fully Java-8-ready, which basically means that it is a SAM (Single Abstract Method) type, which can be instanciated using a lambda expression. Consider this example:

```
try (Connection c = getConnection()) {
    String sql = "select schema_name, is_default " +
                 "from information_schema.schemata " +
                 "order by schema_name";

    DSL.using(c)
       .fetch(sql)

       // We can use lambda expressions to map jOOQ Records
       .map(rs -> new Schema(
           rs.getValue("SCHEMA_NAME", String.class),
           rs.getValue("IS_DEFAULT", boolean.class)
       ))

       // ... and then profit from the new Collection methods
       .forEach(System.out::println);
}
```

The above example shows how jOOQ's Result.map() method can receive a lambda expression that implements RecordMapper to map from jOOQ Records to your custom types.

jOOQ and the Streams API

jOOQ's Result type extends java.util.List, which opens up access to a variety of new Java features in Java 8. The following example shows how easy it is to transform a jOOQ Result containing INFORMATION_SCHEMA meta data to produce DDL statements:

```
DSL.using(c)
   .select(
       COLUMNS.TABLE_NAME,
       COLUMNS.COLUMN_NAME,
       COLUMNS.TYPE_NAME
   )
   .from(COLUMNS)
   .orderBy(
       COLUMNS.TABLE_CATALOG,
       COLUMNS.TABLE_SCHEMA,
       COLUMNS.TABLE_NAME,
       COLUMNS.ORDINAL_POSITION
   )
   .fetch()  // jOOQ ends here
   .stream() // JDK 8 Streams start here
   .collect(groupingBy(
       r -> r.getValue(COLUMNS.TABLE_NAME),
       LinkedHashMap::new,
       mapping(
           r -> new Column(
               r.getValue(COLUMNS.COLUMN_NAME),
               r.getValue(COLUMNS.TYPE_NAME)
           ),
           toList()
       )
   ))
   .forEach(
       (table, columns) -> {
            // Just emit a CREATE TABLE statement
            System.out.println(
                "CREATE TABLE " + table + " (");

            // Map each "Column" type into a String
            // containing the column specification,
            // and join them using comma and
            // newline. Done!
            System.out.println(
                columns.stream()
                       .map(col -> "  " + col.name +
                                   " " + col.type)
                       .collect(Collectors.joining(",\n"))
            );

           System.out.println(");");
       }
   );
```

The above example is explained more in depth in this blog post: http://blog.jooq.org/2014/04/11/java-8-friday-no-more-need-for-orms/. For more information about Java 8, consider these resources:

- Our Java 8 Friday blog series
- A great Java 8 resources collection by the folks at Baeldung.com

3.6. jOOQ and JavaFX

One of the major improvements of Java 8 is the introduction of JavaFX into the JavaSE. With jOOQ and Java 8 Streams and lambdas, it is now very easy and idiomatic to transform SQL results into JavaFX XYChart.Series or other, related objects:

Creating a bar chart from a jOOQ Result

As we've seen in the previous section about jOOQ and Java 8, jOOQ integrates seamlessly with Java 8's Streams API. The fluent style can be maintained throughout the data transformation chain.

In this example, we're going to use Open Data from the world bank to show a comparison of countries GDP and debts:

```
DROP SCHEMA IF EXISTS world;

CREATE SCHEMA world;

CREATE TABLE world.countries (
  code CHAR(2) NOT NULL,
  year INT NOT NULL,
  gdp_per_capita DECIMAL(10, 2) NOT NULL,
  govt_debt DECIMAL(10, 2) NOT NULL
);

INSERT INTO world.countries
VALUES ('CA', 2009, 40764, 51.3),
       ('CA', 2010, 47465, 51.4),
       ('CA', 2011, 51791, 52.5),
       ('CA', 2012, 52409, 53.5),
       ('DE', 2009, 40270, 47.6),
       ('DE', 2010, 40408, 55.5),
       ('DE', 2011, 44355, 55.1),
       ('DE', 2012, 42598, 56.9),
       ('FR', 2009, 40488, 85.0),
       ('FR', 2010, 39448, 89.2),
       ('FR', 2011, 42578, 93.2),
       ('FR', 2012, 39759,103.8),
       ('GB', 2009, 35455,121.3),
       ('GB', 2010, 36573, 85.2),
       ('GB', 2011, 38927, 99.6),
       ('GB', 2012, 38649,103.2),
       ('IT', 2009, 35724,121.3),
       ('IT', 2010, 34673,119.9),
       ('IT', 2011, 36988,113.0),
       ('IT', 2012, 33814,131.1),
       ('JP', 2009, 39473,166.8),
       ('JP', 2010, 43118,174.8),
       ('JP', 2011, 46204,189.5),
       ('JP', 2012, 46548,196.5),
       ('RU', 2009,  8616,  8.7),
       ('RU', 2010, 10710,  9.1),
       ('RU', 2011, 13324,  9.3),
       ('RU', 2012, 14091,  9.4),
       ('US', 2009, 46999, 76.3),
       ('US', 2010, 48358, 85.6),
       ('US', 2011, 49855, 90.1),
       ('US', 2012, 51755, 93.8);
```

Once this data is set up (e.g. in an H2 or PostgreSQL database), we'll run jOOQ's code generator and implement the following code to display our chart:

```
CategoryAxis xAxis = new CategoryAxis();
NumberAxis yAxis = new NumberAxis();
xAxis.setLabel("Country");
yAxis.setLabel("% of GDP");

BarChart<String, Number> bc = new BarChart<String, Number>(xAxis, yAxis);
bc.setTitle("Government Debt");
bc.getData().addAll(

    // SQL data transformation, executed in the database
    // -------------------------------------------------
    DSL.using(connection)
       .select(
           COUNTRIES.YEAR,
           COUNTRIES.CODE,
           COUNTRIES.GOVT_DEBT)
       .from(COUNTRIES)
       .join(
           table(
               select(COUNTRIES.CODE, avg(COUNTRIES.GOVT_DEBT).as("avg"))
               .from(COUNTRIES)
               .groupBy(COUNTRIES.CODE)
           ).as("c1")
       )
       .on(COUNTRIES.CODE.eq(field(name("c1", COUNTRIES.CODE.getName()), String.class)))

       // order countries by their average projected value
       .orderBy(
           field(name("avg")),
           COUNTRIES.CODE,
           COUNTRIES.YEAR)

       // The result produced by the above statement looks like this:
       // +----+----+---------+
       // |year|code|govt_debt|
       // +----+----+---------+
       // |2009|RU  |     8.70|
       // |2010|RU  |     9.10|
       // |2011|RU  |     9.30|
       // |2012|RU  |     9.40|
       // |2009|CA  |    51.30|
       // +----+----+---------+

    // Java data transformation, executed in application memory
    // -------------------------------------------------------

       // Group results by year, keeping sort order in place
       .fetchGroups(COUNTRIES.YEAR)

       // Stream<Entry<Integer, Result<Record3<BigDecimal, String, Integer>>>>
       .entrySet()
       .stream()

       // Map each entry into a { Year -> Projected value } series
       .map(entry -> new XYChart.Series<>(
           entry.getKey().toString(),
           observableArrayList(

               // Map each country record into a chart Data object
               entry.getValue()
                    .map(country -> new XYChart.Data<String, Number>(
                         country.getValue(COUNTRIES.CODE),
                         country.getValue(COUNTRIES.GOVT_DEBT)
                    ))
           )
       ))
       .collect(toList())
);
```

The above example uses basic SQL-92 syntax where the countries are ordered using aggregate information from a nested SELECT, which is supported in all databases. If you're using a database that supports window functions, e.g. PostgreSQL or any commercial database, you could have also written a simpler query like this:00

```
DSL.using(connection)
   .select(
       COUNTRIES.YEAR,
       COUNTRIES.CODE,
       COUNTRIES.GOVT_DEBT)
   .from(COUNTRIES)

   // order countries by their average projected value
   .orderBy(
       DSL.avg(COUNTRIES.GOVT_DEBT).over(partitionBy(COUNTRIES.CODE)),
       COUNTRIES.CODE,
       COUNTRIES.YEAR)
   .fetch()
   ;

return bc;
```

When executed, we'll get nice-looking bar charts like these:

The complete example can be downloaded and run from GitHub:
https://github.com/jOOQ/jOOQ/tree/master/jOOQ-examples/jOOQ-javafx-example

3.7. jOOQ and Nashorn

With Java 8 and the new built-in JavaScript engine Nashorn, a whole new ecosystem of software can finally make easy use of jOOQ in server-side JavaScript. A very simple example can be seen here:

```
// Let's assume these objects were generated
// by the jOOQ source code generator
var Tables = Java.type("org.jooq.db.h2.information_schema.Tables");
var t = Tables.TABLES;
var c = Tables.COLUMNS;

// This is the equivalent of Java's static imports
var count = DSL.count;
var row = DSL.row;

// We can now execute the following query:
print(
    DSL.using(conn)
       .select(
           t.TABLE_SCHEMA,
           t.TABLE_NAME,
           c.COLUMN_NAME)
       .from(t)
       .join(c)
       .on(row(t.TABLE_SCHEMA, t.TABLE_NAME)
           .eq(c.TABLE_SCHEMA, c.TABLE_NAME))
       .orderBy(
           t.TABLE_SCHEMA.asc(),
           t.TABLE_NAME.asc(),
           c.ORDINAL_POSITION.asc())
       .fetch()
);
```

More details about how to use jOOQ, JDBC, and SQL with Nashorn can be seen here.

3.8. jOOQ and Scala

As any other library, jOOQ can be easily used in Scala, taking advantage of the many Scala language features such as for example:

- Optional "." to dereference methods from expressions
- Optional "(" and ")" to delimit method argument lists
- Optional ";" at the end of a Scala statement
- Type inference using "var" and "val" keywords
- Lambda expressions and for-comprehension syntax for record iteration and data type conversion

But jOOQ also leverages other useful Scala features, such as

- implicit defs for operator overloading
- Scala Macros (soon to come)

All of the above heavily improve jOOQ's querying DSL API experience for Scala developers.

A short example jOOQ application in Scala might look like this:

```
import collection.JavaConversions._                                    // Import implicit defs for iteration over org.jooq.Result
                                                                       //
import java.sql.DriverManager                                          //
                                                                       //
import org.jooq._                                                      //
import org.jooq.impl._                                                 //
import org.jooq.impl.DSL._                                             //
import org.jooq.examples.scala.h2.Tables._                             //
import org.jooq.scalaextensions.Conversions._                          // Import implicit defs for overloaded jOOQ/SQL operators
                                                                       //
object Test {                                                          //
  def main(args: Array[String]): Unit = {                              //
    val c = DriverManager.getConnection("jdbc:h2:~/test", "sa", ""); // Standard JDBC connection
    val e = DSL.using(c, SQLDialect.H2);                               //
    val x = AUTHOR as "x"                                              // SQL-esque table aliasing
                                                                       //
    for (r <- e                                                        // Iteration over Result. "r" is an org.jooq.Record3
        select (                                                       //
          BOOK.ID * BOOK.AUTHOR_ID,                                    // Using the overloaded "*" operator
          BOOK.ID + BOOK.AUTHOR_ID * 3 + 4,                            // Using the overloaded "+" operator
          BOOK.TITLE || " abc" || " xy"                                // Using the overloaded "||" operator
        )                                                              //
        from BOOK                                                      // No need to use parentheses or "." here
        leftOuterJoin (                                                //
          select (x.ID, x.YEAR_OF_BIRTH)                               // Dereference fields from aliased table
          from x                                                       //
          limit 1                                                      //
          asTable x.getName()                                          //
        )                                                              //
        on BOOK.AUTHOR_ID === x.ID                                     // Using the overloaded "===" operator
        where (BOOK.ID <> 2)                                           // Using the olerloaded "<>" operator
        or (BOOK.TITLE in ("O Alquimista", "Brida"))                   // Neat IN predicate expression
        fetch                                                          //
    ) {                                                                //
      println(r)                                                       //
    }                                                                  //
  }                                                                    //
}
```

For more details about jOOQ's Scala integration, please refer to the manual's section about SQL building with Scala.

3.9. jOOQ and Groovy

As any other library, jOOQ can be easily used in Groovy, taking advantage of the many Groovy language features such as for example:

- Optional ";" at the end of a Groovy statement
- Type inference for local variables

A short example jOOQ application in Groovy might look like this:
Note that while Groovy supports some means of operator overloading, we think that these means should be avoided in a jOOQ integration. For instance, a + b in Groovy maps to a formal a.plus(b) method invocation, and jOOQ provides the required synonyms in its API to help you write such expressions. Nonetheless, Groovy only offers little typesafety, and as such, operator overloading can lead to many runtime issues.

Another caveat of Groovy operator overloading is the fact that operators such as == or >= map to a.equals(b), a.compareTo(b) == 0, a.compareTo(b) >= 0 respectively. This behaviour does not make sense in a fluent API such as jOOQ.

3.10. jOOQ and Kotlin

As any other library, jOOQ can be easily used in Kotlin, taking advantage of the many Kotlin language features such as for example:

- Optional ";" at the end of a Kotlin statement
- Type inference for local variables

A short example jOOQ application in Kotlin might look like this:
Note that Kotlin supports some means of operator overloading. For instance, a + b in Kotlin maps to a formal a.plus(b) method invocation, and jOOQ provides the required synonyms in its API to help you write such expressions.

One particularly nice language feature is the fact that [square brackets] allow for accessing any object's contents via get() and set() methods. Instead of using the above value1(), value2(), and value3() methods, we could also iterate as such:
A caveat of Kotlin operator overloading is the fact that operators such as == or >= map to a.equals(b), a.compareTo(b) == 0, a.compareTo(b) >= 0 respectively. This behaviour does not make sense in a fluent API such as jOOQ.

3.11. jOOQ and NoSQL

jOOQ users often get excited about jOOQ's intuitive API and would then wish for NoSQL support.

There are a variety of NoSQL databases that implement some sort of proprietary query language. Some of these query languages even look like SQL. Examples are JCR-SQL2, CQL (Cassandra Query Language), Cypher (Neo4j's Query Language), SOQL (Salesforce Query Language) and many more.

Mapping the jOOQ API onto these alternative query languages would be a very poor fit and a leaky abstraction. We believe in the power and expressivity of the SQL standard and its various dialects. Databases that extend this standard too much, or implement it not thoroughly enough are often not suitable targets for jOOQ. It would be better to build a new, dedicated API for just that one particular query language.

jOOQ is about SQL, and about SQL alone. Read more about our visions in the manual's preface.

3.12. jOOQ and JPA

Just because you're using jOOQ doesn't mean you have to use it for everything!

When introducing jOOQ into an existing application that uses JPA, the common question is always: "Should we replace JPA by jOOQ?" and "How do we proceed doing that?"

Beware that jOOQ is not a replacement for JPA. Think of jOOQ as a complement. JPA (and ORMs in general) try to solve the ***object graph persistence*** problem. In short, this problem is about

- Loading an entity graph into client memory from a database
- Manipulating that graph in the client
- Storing the modification back to the database

As the above graph gets more complex, a lot of tricky questions arise like:

- What's the optimal order of SQL DML operations for loading and storing entities?
- How can we batch the commands more efficiently?
- How can we keep the transaction footprint as low as possible without compromising on ACID?
- How can we implement optimistic locking?

jOOQ only has *some* of the answers.

While jOOQ does offer updatable records that help running simple CRUD, a batch API, optimistic locking capabilities, jOOQ mainly focuses on executing actual SQL statements.

SQL is the preferred language of database interaction, when any of the following are given:

- You run reports and analytics on large data sets directly in the database
- You import / export data using ETL
- You run complex business logic as SQL queries

Whenever SQL is a good fit, jOOQ is a good fit. Whenever you're operating and persisting the ***object graph***, JPA is a good fit.

And sometimes, it's best to combine both

3.13. Dependencies

Dependencies are a big hassle in modern software. Many libraries depend on other, non-JDK library parts that come in different, incompatible versions, potentially causing trouble in your runtime environment. jOOQ has no external dependencies on any third-party libraries.

However, the above rule has some exceptions:

- logging APIs are referenced as "optional dependencies". jOOQ tries to find slf4j or log4j on the classpath. If it fails, it will use the java.util.logging.Logger
- Oracle ojdbc types used for array creation are loaded using reflection. The same applies to Postgres PG* types.
- Small libraries with compatible licenses are incorporated into jOOQ. These include jOOR, jOOU, parts of OpenCSV, json simple, parts of commons-lang
- javax.persistence and javax.validation will be needed if you activate the relevant code generation flags

3.14. Build your own

In order to build jOOQ (Open Source Edition) yourself, please download the sources from https://github.com/jOOQ/jOOQ and use Maven to build jOOQ, preferably in Eclipse. The jOOQ Open Source Edition requires Java 8+ to compile and run. The commercial jOOQ Editions require Java 8+ or Java 6+ to compile and run, depending on the distribution.

Some useful hints to build jOOQ yourself:

- Get the latest version of Git or EGit
- Get the latest version of Maven or M2E
- Check out the jOOQ sources from https://github.com/jOOQ/jOOQ
- Optionally, import Maven artefacts into an Eclipse workspace using the following command (see the maven-eclipse-plugin documentation for details):

    * mvn eclipse:eclipse

- Build the jooq-parent artefact by using any of these commands:

    * mvn clean package
      create .jar files in ${project.build.directory}
    * mvn clean install
      install the .jar files in your local repository (e.g. ~/.m2)
    * mvn clean {goal} -Dmaven.test.skip=true
      don't run unit tests when building artefacts

3.15. jOOQ and backwards-compatibility

Semantic versioning

jOOQ's understanding of backwards compatibility is inspired by the rules of semantic versioning according to http://semver.org. Those rules impose a versioning scheme [X].[Y].[Z] that can be summarised as follows:

- If a patch release includes bugfixes, performance improvements and API-irrelevant new features, [Z] is incremented by one.
- If a minor release includes backwards-compatible, API-relevant new features, [Y] is incremented by one and [Z] is reset to zero.
- If a major release includes backwards-incompatible, API-relevant new features, [X] is incremented by one and [Y], [Z] are reset to zero.

jOOQ's understanding of backwards-compatibility

Backwards-compatibility is important to jOOQ. You've chosen jOOQ as a strategic SQL engine and you don't want your SQL to break.

However, there are some elements of API evolution that would be considered backwards-incompatible in other APIs, but not in jOOQ. As discussed later on in the section about jOOQ's query DSL API, much of jOOQ's API is indeed an internal domain-specific language implemented mostly using Java interfaces. Adding language elements to these interfaces means any of these actions:

- Adding methods to the interface
- Overloading methods for convenience
- Changing the type hierarchy of interfaces

It becomes obvious that it would be impossible to add new language elements (e.g. new SQL functions, new SELECT clauses) to the API without breaking any client code that actually implements those interfaces. Hence, the following rules should be observed:

- jOOQ's DSL interfaces should not be implemented by client code! Extend only those extension points that are explicitly documented as "extendable" (e.g. custom QueryParts).
- Binary compatibility can be expected from patch releases, but not from minor releases as it is not practical to maintain binary compatibility in an internal DSL.
- Source compatibility can be expected from patch and minor releases.
- Behavioural compatibility can be expected from patch and minor releases.
- Any jOOQ SPI XYZ that is meant to be implemented ships with a DefaultXYZ or AbstractXYZ, which can be used safely as a default implementation.

jOOQ-codegen and jOOQ-meta

While a reasonable amount of care is spent to maintain these two modules under the rules of semantic versioning, it may well be that minor releases introduce backwards-incompatible changes. This will be announced in the respective release notes and should be the exception.

4. SQL building

SQL is a declarative language that is hard to integrate into procedural, object-oriented, functional or any other type of programming languages. jOOQ's philosophy is to give SQL the credit it deserves and integrate SQL itself as an "internal domain specific language" directly into Java.

With this philosophy in mind, SQL building is the main feature of jOOQ. All other features (such as SQL execution and code generation) are mere convenience built on top of jOOQ's SQL building capabilities.

This section explains all about the various syntax elements involved with jOOQ's SQL building capabilities. For a complete overview of all syntax elements, please refer to the manual's sections about SQL to DSL mapping rules.

4.1. The query DSL type

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.DSL class is the main class from where you will create all jOOQ objects. It serves as a static factory for table expressions, column expressions (or "fields"), conditional expressions and many other QueryParts.

The static query DSL API

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

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

Note, that when working with Eclipse, you could also add the DSL to your favourites. This will allow to access functions even more fluently:

```
concat(trim(FIRST_NAME), trim(LAST_NAME));

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

4.1.1. DSL subclasses

There are a couple of subclasses for the general query DSL. Each SQL dialect has its own dialect-specific DSL. For instance, if you're only using the MySQL dialect, you can choose to reference the MySQLDSL instead of the standard DSL:

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

- MySQL's encryption functions
- PL/SQL constructs, pgplsql, or any other dialect's ROUTINE-language (maybe in the future)

4.2. The DSLContext class

DSLContext references a org.jooq.Configuration, an object that configures jOOQ's behaviour when executing queries (see SQL execution for more details). Unlike the static DSL, the DSLContext allow for creating SQL statements that are already "configured" and ready for execution.

Fluent creation of a DSLContext object

The DSLContext object can be created fluently from the DSL type:

```
// Create it from a pre-existing configuration
DSLContext create = DSL.using(configuration);

// Create it from ad-hoc arguments
DSLContext create = DSL.using(connection, dialect);
```

If you do not have a reference to a pre-existing Configuration object (e.g. created from org.jooq.impl.DefaultConfiguration), the various overloaded DSL.using() methods will create one for you.

Contents of a Configuration object

A Configuration can be supplied with these objects:

- org.jooq.SQLDialect : The dialect of your database. This may be any of the currently supported database types (see SQL Dialect for more details)
- org.jooq.conf.Settings : An optional runtime configuration (see Custom Settings for more details)
- org.jooq.ExecuteListenerProvider : An optional reference to a provider class that can provide execute listeners to jOOQ (see ExecuteListeners for more details)
- org.jooq.RecordMapperProvider : An optional reference to a provider class that can provide record mappers to jOOQ (see POJOs with RecordMappers for more details)
- Any of these:

    * java.sql.Connection : An optional JDBC Connection that will be re-used for the whole lifecycle of your Configuration (see Connection vs. DataSource for more details). For simplicity, this is the use-case referenced from this manual, most of the time.
    * java.sql.DataSource : An optional JDBC DataSource that will be re-used for the whole lifecycle of your Configuration. If you prefer using DataSources over Connections, jOOQ will internally fetch new Connections from your DataSource, conveniently closing them again after query execution. This is particularly useful in J2EE or Spring contexts (see Connection vs. DataSource for more details)
    * org.jooq.ConnectionProvider : A custom abstraction that is used by jOOQ to "acquire" and "release" connections. jOOQ will internally "acquire" new Connections from your ConnectionProvider, conveniently "releasing" them again after query execution. (see Connection vs. DataSource for more details)

Wrapping a Configuration object, a DSLContext can construct statements, for later execution. An example is given here:

```
// The DSLContext is "configured" with a Connection and a SQLDialect
DSLContext create = DSL.using(connection, dialect);

// This select statement contains an internal reference to the DSLContext's Configuration:
Select<?> select = create.selectOne();

// Using the internally referenced Configuration, the select statement can now be executed:
Result<?> result = select.fetch();
```

Note that you do not need to keep a reference to a DSLContext. You may as well inline your local variable, and fluently execute a SQL statement as such:

```
// Execute a statement from a single execution chain:
Result<?> result =
DSL.using(connection, dialect)
   .select()
   .from(BOOK)
   .where(BOOK.TITLE.like("Animal%"))
   .fetch();
```

4.2.1. SQL Dialect

While jOOQ tries to represent the SQL standard as much as possible, many features are vendor-specific to a given database and to its "SQL dialect". jOOQ models this using the org.jooq.SQLDialect enum type.

The SQL dialect is one of the main attributes of a Configuration. Queries created from DSLContexts will assume dialect-specific behaviour when rendering SQL and binding bind values.

Some parts of the jOOQ API are officially supported only by a given subset of the supported SQL dialects. For instance, the Oracle CONNECT BY clause, which is supported by the Oracle and CUBRID databases, is annotated with a org.jooq.Support annotation, as such:

```
/**
 * Add an Oracle-specific <code>CONNECT BY</code> clause to the query
 */
@Support({ SQLDialect.CUBRID, SQLDialect.ORACLE })
SelectConnectByConditionStep<R> connectBy(Condition condition);
```

jOOQ API methods which are not annotated with the org.jooq.Support annotation, or which are annotated with the Support annotation, but without any SQL dialects can be safely used in all SQL dialects. An example for this is the SELECT statement factory method:

```
/**
 * Create a new DSL select statement.
 */
@Support
SelectSelectStep<R> select(Field<?>... fields);
```

jOOQ's SQL clause emulation capabilities

The aforementioned Support annotation does not only designate, which databases natively support a feature. It also indicates that a feature is emulated by jOOQ for some databases lacking this feature. An example of this is the DISTINCT predicate, a predicate syntax defined by SQL:1999 and implemented only by H2, HSQLDB, and Postgres:

```
A IS DISTINCT FROM B
```

Nevertheless, the IS DISTINCT FROM predicate is supported by jOOQ in all dialects, as its semantics can be expressed with an equivalent CASE expression. For more details, see the manual's section about the DISTINCT predicate.

jOOQ and the Oracle SQL dialect

Oracle SQL is much more expressive than many other SQL dialects. It features many unique keywords, clauses and functions that are out of scope for the SQL standard. Some examples for this are

- The CONNECT BY clause, for hierarchical queries
- The PIVOT keyword for creating PIVOT tables
- Packages, object-oriented user-defined types, member procedures as described in the section about stored procedures and functions
- Advanced analytical functions as described in the section about window functions

jOOQ has a historic affinity to Oracle's SQL extensions. If something is supported in Oracle SQL, it has a high probability of making it into the jOOQ API

4.2.2. SQL Dialect Family

In jOOQ 3.1, the notion of a SQLDialect.family() was introduced, in order to group several similar SQL dialects into a common family. An example for this is SQL Server, which is supported by jOOQ in various versions:

- SQL Server: The "version-less" SQL Server version. This always maps to the latest supported version of SQL Server
- SQL Server 2012: The SQL Server version 2012
- SQL Server 2008: The SQL Server version 2008

In the above list, SQLSERVER is both a dialect and a family of three dialects. This distinction is used internally by jOOQ to distinguish whether to use the OFFSET .. FETCH clause (SQL Server 2012), or whether to emulate it using ROW_NUMBER() OVER() (SQL Server 2008).

4.2.3. Connection vs. DataSource

Interact with JDBC Connections

While you can use jOOQ for SQL building only, you can also run queries against a JDBC java.sql.Connection. Internally, jOOQ creates java.sql.Statement or java.sql.PreparedStatement objects from such a Connection, in order to execute statements. The normal operation mode is to provide a Configuration with a JDBC Connection, whose lifecycle you will control yourself. This means that jOOQ will not actively close connections, rollback or commit transactions.

Note, in this case, jOOQ will internally use a org.jooq.impl.DefaultConnectionProvider, which you can reference directly if you prefer that. The DefaultConnectionProvider exposes various transaction-control methods, such as commit(), rollback(), etc.

Interact with JDBC DataSources

If you're in a J2EE or Spring context, however, you may wish to use a javax.sql.DataSource instead. Connections obtained from such a DataSource will be closed after query execution by jOOQ. The semantics of such a close operation should be the returning of the connection into a connection pool, not the actual closing of the underlying connection. Typically, this makes sense in an environment using distributed JTA transactions. An example of using DataSources with jOOQ can be seen in the tutorial section about using jOOQ with Spring.

Note, in this case, jOOQ will internally use a org.jooq.impl.DataSourceConnectionProvider, which you can reference directly if you prefer that.

Inject custom behaviour

If your specific environment works differently from any of the above approaches, you can inject your own custom implementation of a ConnectionProvider into jOOQ. This is the API contract you have to fulfil:

```
public interface ConnectionProvider {

    // Provide jOOQ with a connection
    Connection acquire() throws DataAccessException;

    // Get a connection back from jOOQ
    void release(Connection connection) throws DataAccessException;
}
```

4.2.4. Custom data

In advanced use cases of integrating your application with jOOQ, you may want to put custom data into your Configuration, which you can then access from your...

- Custom ExecuteListeners
- Custom QueryParts

Here is an example of how to use the custom data API. Let's assume that you have written an ExecuteListener, that prevents INSERT statements, when a given flag is set to true:

```
// Implement an ExecuteListener
public class NoInsertListener extends DefaultExecuteListener {

    @Override
    public void start(ExecuteContext ctx) {

        // This listener is active only, when your custom flag is set to true
        if (Boolean.TRUE.equals(ctx.configuration().data("com.example.my-namespace.no-inserts"))) {

            // If active, fail this execution, if an INSERT statement is being executed
            if (ctx.query() instanceof Insert) {
                throw new DataAccessException("No INSERT statements allowed");
            }
        }
    }
}
```

See the manual's section about ExecuteListeners to learn more about how to implement an ExecuteListener.

Now, the above listener can be added to your Configuration, but you will also need to pass the flag to the Configuration, in order for the listener to work:

```
// Create your Configuration
Configuration configuration = new DefaultConfiguration().set(connection).set(dialect);

// Set a new execute listener provider onto the configuration:
configuration.set(new DefaultExecuteListenerProvider(new NoInsertListener()));

// Use any String literal to identify your custom data
configuration.data("com.example.my-namespace.no-inserts", true);

// Try to execute an INSERT statement
try {
    DSL.using(configuration)
       .insertInto(AUTHOR, AUTHOR.ID, AUTHOR.LAST_NAME)
       .values(1, "Orwell")
       .execute();

    // You shouldn't get here
    Assert.fail();
}

// Your NoInsertListener should be throwing this exception here:
catch (DataAccessException expected) {
    Assert.assertEquals("No INSERT statements allowed", expected.getMessage());
}
```

Using the data() methods, you can store and retrieve custom data in your Configurations.

4.2.5. Custom ExecuteListeners

ExecuteListeners are a useful tool to...

- implement custom logging
- apply triggers written in Java
- collect query execution statistics

ExecuteListeners are hooked into your Configuration by returning them from an org.jooq.ExecuteListenerProvider:

```
// Create your Configuration
Configuration configuration = new DefaultConfiguration().set(connection).set(dialect);

// Hook your listener providers into the configuration:
configuration.set(
    new DefaultExecuteListenerProvider(new MyFirstListener()),
    new DefaultExecuteListenerProvider(new PerformanceLoggingListener()),
    new DefaultExecuteListenerProvider(new NoInsertListener())
);
```

See the manual's section about ExecuteListeners to see examples of such listener implementations.

4.2.6. Custom Settings

The jOOQ Configuration allows for some optional configuration elements to be used by advanced users. The org.jooq.conf.Settings class is a JAXB-annotated type, that can be provided to a Configuration in several ways:

- In the DSLContext constructor (DSL.using()). This will override default settings below
- in the org.jooq.impl.DefaultConfiguration constructor. This will override default settings below
- From a location specified by a JVM parameter: -Dorg.jooq.settings
- From the classpath at /jooq-settings.xml
- From the settings defaults, as specified in http://www.jooq.org/xsd/jooq-runtime-3.11.0.xsd

The most specific settings for a given context will apply.

If you wish to configure your settings through XML, but explicitly load them for a given Configuration, you can do so as well, using JAXB:

```
Settings settings = JAXB.unmarshal(new File("/path/to/settings.xml"), Settings.class);
```

Example

For example, if you want to indicate to jOOQ, that it should inline all bind variables, and execute static java.sql.Statement instead of binding its variables to java.sql.PreparedStatement, you can do so by creating the following DSLContext:

```
Settings settings = new Settings();
settings.setStatementType(StatementType.STATIC_STATEMENT);
DSLContext create = DSL.using(connection, dialect, settings);
```

More details

Please refer to the jOOQ runtime configuration XSD for more details:
http://www.jooq.org/xsd/jooq-runtime-3.11.0.xsd

4.2.6.1. Object qualification

By default, jOOQ fully qualifies all objects with their catalog and schema names, if such qualification is made available by the code generator. For instance, the following SQL statement containing full qualification may be produced by jOOQ code with seemingly no qualification:

```
-- Full qualification on columns and tables
SELECT catalog.schema.table.column
FROM catalog.schema.table
```

```
DSL.using(configuration)
   .select(TABLE.COLUMN) // Column only qualified with table
   .from(TABLE)          // No qualification on table
```

While the jOOQ code is also implicitly fully qualified (see implied imports), it may not be desireable to use fully qualified object names in SQL. The renderCatalog and renderSchema settings are used for this.

Programmatic configuration

```
new Settings()
  .withRenderCatalog(false) // Defaults to true
  .withRenderSchema(false)  // Defaults to true
```

XML configuration

```
<settings xmlns="http://www.jooq.org/xsd/jooq-runtime-3.11.0.xsd">
  <renderCatalog>false</renderCatalog>
  <renderSchema>false</renderSchema>
</settings>
```

By turning off the rendering of full qualification as can be seen above, it will be possible to use code generated from one schema on an entirely different schema of the same structure, e.g. for multitenancy purposes.

More sophisticated multitenancy approaches are available through the render mapping feature.

4.2.6.2. Runtime schema and table mapping

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 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 Configuration's settings with. Take the following example:

Programmatic configuration

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

XML configuration

```
<settings xmlns="http://www.jooq.org/xsd/jooq-runtime-3.11.0.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>
```

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

```
SELECT *
FROM MY_BOOK_WORLD.AUTHOR
```

```
DSL.using(connection, dialect, settings)
   .selectFrom(DEV.AUTHOR)
```

This works because AUTHOR was generated from the DEV schema, which is mapped to the MY_BOOK_WORLD schema by the above settings.

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.AUTHOR to something MY_BOOK_WORLD.MY_APP__AUTHOR, where MY_APP__ is a prefix applied to all of your tables. This can be achieved by creating the following mapping:

Programmatic configuration

```
Settings settings = new Settings()
    .withRenderMapping(new RenderMapping()
    .withSchemata(
        new MappedSchema().withInput("DEV")
                          .withTables(
         new MappedTable().withInput("AUTHOR")
                          .withOutput("MY_APP__AUTHOR"))));
```

XML configuration

```
<settings xmlns="http://www.jooq.org/xsd/jooq-runtime-3.11.0.xsd">
  <renderMapping>
    <schemata>
      <schema>
        <input>DEV</input>
        <tables>
          <table>
            <input>AUTHOR</input>
            <output>MY_APP__AUTHOR</output>
          </table>
        </tables>
      </schema>
    </schemata>
  </renderMapping>
</settings>
```

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

```
SELECT * FROM DEV.MY_APP__AUTHOR
```

Table mapping and schema mapping can be applied independently, by specifying several MappedSchema entries in the above configuration. jOOQ will process them in order of appearance and map at first match. Note that you can always omit a MappedSchema's output value, in case of which, only the table mapping is applied. If you omit a MappedSchema's input value, the table mapping is applied to all schemata!

Using regular expressions

All of the above examples were using 1:1 constant name mappings where the input and output schema or table names are fixed by the configuration. With jOOQ 3.8, regular expression can be used as well for mapping, for example:

Programmatic configuration

```
Settings settings = new Settings()
    .withRenderMapping(new RenderMapping()
    .withSchemata(
        new MappedSchema().withInputExpression(Pattern.compile("DEV_(.*)"))
                          .withOutput("PROD_$1")
                          .withTables(
         new MappedTable().withInputExpression(Pattern.compile("DEV_(.*)"))
                          .withOutput("PROD_$1"))));
```

XML configuration

```
<settings xmlns="http://www.jooq.org/xsd/jooq-runtime-3.11.0.xsd">
  <renderMapping>
    <schemata>
      <schema>
        <inputExpression>DEV_(.*)</inputExpression>
        <output>PROD_$1</output>
        <tables>
          <table>
            <inputExpression>DEV_(.*)</inputExpression>
            <output>PROD_$1</output>
          </table>
        </tables>
      </schema>
    </schemata>
  </renderMapping>
</settings>
```

The only difference to the constant version is that the input field is replaced by the inputExpression field of type java.util.regex.Pattern, in case of which the meaning of the output field is a pattern replacement, not a constant replacement.

Hard-wiring mappings at code-generation time

Note that the manual's section about code generation schema mapping explains how you can hard-wire your schema mappings at code generation time

4.2.6.3. Identifier style

By default, jOOQ will always generate quoted names for all identifiers (even if this manual omits this for readability). For instance:

```
SELECT "TABLE"."COLUMN" FROM "TABLE" -- SQL standard style
SELECT `TABLE`.`COLUMN` FROM `TABLE` -- MySQL style
SELECT [TABLE].[COLUMN] FROM [TABLE] -- SQL Server style
```

Quoting has the following effect on identifiers in most (but not all) databases:

- It allows for using reserved names as object names, e.g. a table called "FROM" is usually possible only when quoted.
- It allows for using special characters in object names, e.g. a column called "FIRST NAME" can be achieved only with quoting.
- It turns what are mostly case-insensitive identifiers into case-sensitive ones, e.g. "name" and "NAME" are different identifiers, whereas name and NAME are not. Please consider your database manual to learn what the proper default case and default case sensitivity is.

The renderNameStyle setting allows for overriding the name of all identifiers in jOOQ to a consistent style. Possible options are:

- QUOTED (the default): This will generate all names in their proper case with quotes around them.
- AS_IS: This will generate all names in their proper case without quotes.
- LOWER: This will transform all names to lower case.
- UPPER: This will transform all names to upper case.

Programmatic configuration

```
Settings settings = new Settings()
    .withRenderNameStyle(RenderNameStyle.AS_IS); // Defaults to QUOTED
```

XML configuration

```
<settings xmlns="http://www.jooq.org/xsd/jooq-runtime-3.11.0.xsd">
  <renderNameStyle>AS_IS</renderNameStyle>
</settings>
```

4.2.6.4. Keyword style

In all SQL dialects, keywords are case insensitive, and this is also the default in jOOQ, which mostly generates lower-case keywords.

Users may wish to adapt this and they have these options for the renderKeywordStyle setting:

- AS_IS (the default): Generate keywords as they are defined in the codebase (mostly lower case).
- LOWER: Generate keywords in lower case.
- UPPER: Generate keywords in upper case.
- PASCAL: Generate keywords in pascal case.

Programmatic configuration

```
Settings settings = new Settings()
    .withRenderKeywordStyle(RenderKeywordStyle.UPPER); // Defaults to AS_IS
```

XML configuration

```
<settings xmlns="http://www.jooq.org/xsd/jooq-runtime-3.11.0.xsd">
  <renderKeywordStyle>UPPER</renderKeywordStyle>
</settings>
```

4.2.6.5. Parameter types

Bind values or bind parameters come in different flavours in different SQL databases. JDBC standardises on their syntax by allowing only ? (question mark) characters as placeholders for bind variables. Thus, jOOQ, by default, generates ? placeholders for JDBC consumptions.

Users who wish to use jOOQ with a different backend than JDBC can specify that all jOOQ bind values, including indexed parameters and named parameters generate alternative strings, other than ?. These are the current options:

- INDEXED (the default): Generates indexed parameter placeholders using ?.
- NAMED: Generates named parameter placeholders, such as :param for parameters that are named explicitly or :1 for unnamed, indexed parameters.
- NAMED_OR_INLINED: Generates named parameter placeholders for parameters that are named explicitly and inlines all unnamed parameters.
- INLINED: Inlines all parameters.

An example:

```
-- INDEXED
SELECT FIRST_NAME || ? FROM AUTHOR WHERE ID = ?
-- NAMED
SELECT FIRST_NAME || :1 FROM AUTHOR WHERE ID = :x
-- NAMED_OR_INLINED
SELECT FIRST_NAME || 'x' FROM AUTHOR WHERE ID = :x
-- INLINED
SELECT FIRST_NAME || 'x' FROM AUTHOR WHERE ID = 42
```

```
Param<String> x = val("x");
Param<Integer> i = param("x", 42);

DSL.using(configuration)
   .select(FIRST_NAME.concat(x))
   .from(AUTHOR)
   .where(ID.eq(i))
   .fetch();
```

Programmatic configuration

```
Settings settings = new Settings()
    .withParamType(ParamType.NAMED); // Defaults to INDEXED
```

XML configuration

```
<settings xmlns="http://www.jooq.org/xsd/jooq-runtime-3.11.0.xsd">
  <paramType>NAMED</paramType>
</settings>
```

The following setting statementType may override this setting.

4.2.6.6. Statement Type

JDBC knows two types of statements:

- java.sql.PreparedStatement: This allows for sending bind variables to the server. jOOQ uses prepared statements by default.
- java.sql.Statement: Also "static statement". These do not support bind variables and may be useful for one-shot commands like DDL statements.

The statementType setting allows for overriding the default of using prepared statements internally. There are two possible options for this setting:

- PREPARED_STATEMENT (the default): Use prepared statements.
- STATIC_STATEMENT: Use static statements. This enforces the paramType == INLINED. See parameter types

Programmatic configuration

```
Settings settings = new Settings()
    .withStatementType(StatementType.STATIC_STATEMENT); // Defaults to PREPARED_STATEMENT
```

XML configuration

```
<settings xmlns="http://www.jooq.org/xsd/jooq-runtime-3.11.0.xsd">
  <statementType>STATIC_STATEMENT</statementType>
</settings>
```

4.2.6.7. Execute Logging

The executeLogging setting turns off the default logging implemented through org.jooq.tools.LoggerListener

Programmatic configuration

```
Settings settings = new Settings()
    .withExecuteLogging(false); // Defaults to true
```

XML configuration

```
<settings xmlns="http://www.jooq.org/xsd/jooq-runtime-3.11.0.xsd">
  <executeLogging>false</executeLogging>
</settings>
```

4.2.6.8. Optimistic Locking

There are two settings governing the behaviour of the jOOQ optimistic locking feature:

- executeWithOptimisticLocking: This allows for turning off the feature entirely.
- executeWithOptimisticLockingExcludeUnversioned: This allows for turning off the feature for updatable records who are not explicitly versioned.

Programmatic configuration

```
Settings settings = new Settings()
    .withExecuteWithOptimisticLocking(true)                    // Defaults to false
    .withExecuteWithOptimisticLockingExcludeUnversioned(false); // Defaults to false
```

XML configuration

```
<settings xmlns="http://www.jooq.org/xsd/jooq-runtime-3.11.0.xsd">
  <executeWithOptimisticLocking>true</executeWithOptimisticLocking>
  <executeWithOptimisticLockingExcludeUnversioned>false</executeWithOptimisticLockingExcludeUnversioned>
</settings>
```

For more details, please refer to the manual's section about the optimistic locking feature.

4.2.6.9. Auto-attach Records

By default, all records fetched through jOOQ are "attached" to the configuration that created them. This allows for features like updatable records as can be seen here:

```
AuthorRecord author =
DSL.using(configuration) // This configuration will be attached to any record produced by the below query.
   .selectFrom(AUTHOR)
   .where(AUTHOR.ID.eq(1))
   .fetchOne();

author.setLastName("Smith");
author.store(); // This store call operates on the "attached" configuration.
```

In some cases (e.g. when serialising records), it may be desirable not to attach the Configuration that created a record to the record. This can be achieved with the attachRecords setting:

Programmatic configuration

```
Settings settings = new Settings()
    .withAttachRecords(false); // Defaults to true
```

XML configuration

```
<settings xmlns="http://www.jooq.org/xsd/jooq-runtime-3.11.0.xsd">
  <attachRecords>false</attachRecords>
</settings>
```

4.2.6.10. Updatable Primary Keys

In most database design guidelines, primary key values are expected to never change - an assumption that is essential to a normalised database.

As always, there are exceptions to these rules, and users may wish to allow for updatable primary key values in the updatable records feature (note: any value can always be updated through ordinary update statements). An example:

```
AuthorRecord author =
DSL.using(configuration) // This configuration will be attached to any record produced by the below query.
   .selectFrom(AUTHOR)
   .where(AUTHOR.ID.eq(1))
   .fetchOne();

author.setId(2);
author.store(); // The behaviour of this store call is governed by the updatablePrimaryKeys flag
```

The above store call depends on the value of the updatablePrimaryKeys flag:

- false (the default): Since immutability of primary keys is assumed, the store call will create a new record (a copy) with the new primary key value.
- true: Since mutablity of primary keys is allowed, the store call will change the primary key value from 1 to 2.

Programmatic configuration

```
Settings settings = new Settings()
    .withUpdatablePrimaryKeys(true); // Defaults to false
```

XML configuration

```
<settings xmlns="http://www.jooq.org/xsd/jooq-runtime-3.11.0.xsd">
  <updatablePrimaryKeys>true</updatablePrimaryKeys>
</settings>
```

4.2.6.11. Reflection caching

All operations of the DefaultRecordMapper are cached in the Configuration by default for improved mapping and reflection speed. Users who prefer to override this cache, or work with their own custom record mapper provider may wish to turn off the out-of-the-box caching feature.

Programmatic configuration

```
Settings settings = new Settings()
    .withReflectionCaching(false); // Defaults to true
```

XML configuration

```
<settings xmlns="http://www.jooq.org/xsd/jooq-runtime-3.11.0.xsd">
  <reflectionCaching>false</reflectionCaching>
</settings>
```

4.2.6.12. Fetch Warnings

Apart from JDBC exceptions, there is also the possibility to handle java.sql.SQLWarning, which are made available to jOOQ users through the java.sql.ExecuteListener SPI and the log

Users who do not wish to get these notifications (e.g. for performance reasons), may turn off fetching of warnings through the fetchWarnings setting:

Programmatic configuration

```
Settings settings = new Settings()
    .withFetchWarnings(false); // Defaults to true
```

XML configuration

```
<settings xmlns="http://www.jooq.org/xsd/jooq-runtime-3.11.0.xsd">
  <fetchWarnings>false</fetchWarnings>
</settings>
```

4.2.6.13. Return All Columns On Store

When using the updatable records feature, jOOQ always fetches the generated identity value, if such a value is available.

The identity value is not the only value that is generated by default. Specifically, there may be triggers that are used for auditing or other reasons, which generate LAST_UPDATE or LAST_UPDATE_BY values in a record. Users who wish to also automatically fetch these values after all store(), insert(), or update() calls may do so by specifying the returnAllOnUpdatableRecord setting. This setting depends on the availability of INSERT .. RETURNING, UPDATE .. RETURNING, and DELETE .. RETURNING statements, which are not available from all databases.

Programmatic configuration

```
Settings settings = new Settings()
    .withReturnAllOnUpdatableRecord(true); // Defaults to false
```

XML configuration

```
<settings xmlns="http://www.jooq.org/xsd/jooq-runtime-3.11.0.xsd">
  <returnAllOnUpdatableRecord>true</returnAllOnUpdatableRecord>
</settings>
```

4.2.6.14. Map JPA Annotations

The org.jooq.impl.DefaultRecordMapper supports basic JPA mapping (mostly @Table and @Column annotations). Looking up these annotations costs a slight extra overhead (mostly taken care of through reflection caching). It can be turned off using the mapJPAAnnotations setting:

Programmatic configuration

```
Settings settings = new Settings()
    .withMapJPAAnnotations(false); // Defaults to true
```

XML configuration

```
<settings xmlns="http://www.jooq.org/xsd/jooq-runtime-3.11.0.xsd">
  <mapJPAAnnotations>false</mapJPAAnnotations>
</settings>
```

4.2.6.15. JDBC Flags

JDBC statements feature a couple of flags that influence the execution of such a statement. Each of these flags can be configured through jOOQ's org.jooq.Query and org.jooq.ResultQuery on a statement-per-statement basis, but there's also the possibility to centrally specify a value for these flags. These are the three flags:

- queryTimeout: Corresponds to Query.queryTimeout() or Statement.setQueryTimeout()
- maxRows: Corresponds to ResultQuery.maxRows() or Statement.setMaxRows()
- fetchSize: Corresponds to ResultQuery.fetchSize() or Statement.setFetchSize()

All of these flags are JDBC-only features with no direct effect on jOOQ. jOOQ only passes them through to the underlying statement.

Programmatic configuration

```
Settings settings = new Settings()
    .withQueryTimeout(5)
    .withMaxRows(1000)
    .withFetchSize(20);
```

XML configuration

```
<settings xmlns="http://www.jooq.org/xsd/jooq-runtime-3.11.0.xsd">
  <queryTimeout>5</queryTimeout>
  <maxRows>1000</maxRows>
  <fetchSize>20</fetchSize>
</settings>
```

4.2.6.16. IN-list Padding

Databases that feature a cursor cache / statement cache (e.g. Oracle, SQL Server, DB2, etc.) are highly optimised for prepared statement re-use. When a client sends a prepared statement to the server, the server will go to the cache and look up whether there already exists a previously calculated execution plan for the statement (i.e. the SQL string). This is called a "soft-parse" (in Oracle). If not, the execution plan is calculated on the fly. This is called a "hard-parse" (in Oracle).

Preventing hard-parses is extremely important in high throughput OLTP systems where queries are usually not very complex but are run millions of times in a short amount of time. Using bind variables, this is usually not a problem, with the exception of the IN predicate, which generates different SQL strings even when using bind variables:

```
-- All of these are different SQL statements:
SELECT * FROM AUTHOR WHERE ID IN (?)
SELECT * FROM AUTHOR WHERE ID IN (?, ?)
SELECT * FROM AUTHOR WHERE ID IN (?, ?, ?)
SELECT * FROM AUTHOR WHERE ID IN (?, ?, ?, ?)
SELECT * FROM AUTHOR WHERE ID IN (?, ?, ?, ?, ?)
```

This problem may not be obvious to Java / jOOQ developers, as they are always produced from the same jOOQ statement:

```
-- All of these are the same jOOQ statement
DSL.using(configuration)
   .select()
   .from(AUTHOR)
   .where(AUTHOR.ID.in(collection))
   .fetch();
```

Depending on the possible sizes of the collection, it may be worth exploring using arrays or temporary tables as a workaround, or to reuse the original query that produced the set of IDs in the first place (through a semi-join). But sometimes, this is not possible. In this case, users can opt in to a third workaround: enabling the inListPadding setting. If enabled, jOOQ will "pad" the IN list to a length that is a power of two. So, the original queries would look like this instead:

```
-- Original
SELECT * FROM AUTHOR WHERE ID IN (?)
SELECT * FROM AUTHOR WHERE ID IN (?, ?)
SELECT * FROM AUTHOR WHERE ID IN (?, ?, ?)
SELECT * FROM AUTHOR WHERE ID IN (?, ?, ?, ?)
SELECT * FROM AUTHOR WHERE ID IN (?, ?, ?, ?, ?)
SELECT * FROM AUTHOR WHERE ID IN (?, ?, ?, ?, ?, ?)
```

```
-- Padded
SELECT * FROM AUTHOR WHERE ID IN (?)
SELECT * FROM AUTHOR WHERE ID IN (?, ?)
SELECT * FROM AUTHOR WHERE ID IN (?, ?, ?, ?)
SELECT * FROM AUTHOR WHERE ID IN (?, ?, ?, ?)
SELECT * FROM AUTHOR WHERE ID IN (?, ?, ?, ?, ?, ?, ?, ?)
SELECT * FROM AUTHOR WHERE ID IN (?, ?, ?, ?, ?, ?, ?, ?)
```

This technique will drastically reduce the number of possible SQL strings without impairing too much the usual cases where the IN list is small. When padding, the last bind variable will simply be repeated many times.

Usually, there is a better way - use this as a last resort!

Programmatic configuration

```
Settings settings = new Settings()
    .withInListPadding(true); // Default to false
```

XML configuration

```
<settings xmlns="http://www.jooq.org/xsd/jooq-runtime-3.11.0.xsd">
  <inListPadding>true</inListPadding>
</settings>
```

4.2.6.17. Backslash Escaping

Some databases (mainly MySQL and MariaDB) unfortunately chose to go an alternative, non-SQL-standard route when escaping string literals. Here's an example of how to escape a string containing apostrophes in different dialects:

```
SELECT 'I''m sure this is OK' AS val            -- Standard SQL escaping of apostrophe by doubling it.
SELECT 'I\'m certain this causes trouble' AS val -- Vendor-specific escaping of apostrophe by using a backslash.
```

As most databases don't support backslash escaping (and MySQL also allows for turning it off!), jOOQ by default also doesn't support it when inlining bind variables. However, this can lead to SQL injection vulnerabilities and syntax errors when not dealing with it carefully!

This feature is turned on by default and for historic reasons for MySQL and MariaDB.

- DEFAULT (the - surprise! - default): Turns the feature ON for MySQL and MariaDB and OFF for all other dialects
- ON: Turn the feature on.
- OFF: Turn the feature off.

Programmatic configuration

```
Settings settings = new Settings()
    .withBackslashEscaping(BackslashEscaping.OFF); // Default to DEFAULT
```

XML configuration

```
<settings xmlns="http://www.jooq.org/xsd/jooq-runtime-3.11.0.xsd">
  <backslashEscaping>OFF</backslashEscaping>
</settings>
```

4.2.6.18. Scalar subqueries for stored functions

This setting is useful mostly for the Oracle database, which implements a feature called scalar subquery caching, which is a good tool to avoid the expensive PL/SQL-to-SQL context switch when predicates make use of stored function calls.

With this setting in place, all stored function calls embedded in SQL statements will be wrapped in a scalar subquery:

```
SELECT
  (SELECT my_package.format(LANGUAGE_ID) FROM dual)
FROM BOOK
```

```
DSL.using(configuration)
   .select(MyPackage.format(BOOK.LANGUAGE_ID))
   .from(BOOK)
```

If our table contains thousands of books, but only a dozen of LANGUAGE_ID values, then with scalar subquery caching, we can avoid most of the function calls and cache the result per LANGUAGE_ID.

Programmatic configuration

```
Settings settings = new Settings()
    .withRenderScalarSubqueriesForStoredFunctions(true);
```

XML configuration

```
<settings xmlns="http://www.jooq.org/xsd/jooq-runtime-3.11.0.xsd">
  <renderScalarSubqueriesForStoredFunctions>true</renderScalarSubqueriesForStoredFunctions>
</settings>
```

4.3. SQL Statements (DML)

jOOQ currently supports 5 types of SQL statements. All of these statements are constructed from a DSLContext instance with an optional JDBC Connection or DataSource. If supplied with a Connection or DataSource, they can be executed. Depending on the query type, executed queries can return results.

4.3.1. jOOQ's DSL and model API

jOOQ ships with its own DSL (or Domain Specific Language) that emulates SQL in Java. This means, that you can write SQL statements almost as if Java natively supported it, just like .NET's C# does with LINQ to SQL.

Here is an example to illustrate what that means:

```
-- Select all books by authors born after 1920,
-- named "Paulo" from a catalogue:
SELECT *
  FROM author a
  JOIN 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(AUTHOR.as("a"))
      .join(BOOK.as("b")).on(a.ID.eq(b.AUTHOR_ID))
      .where(a.YEAR_OF_BIRTH.gt(1920)
      .and(a.FIRST_NAME.eq("Paulo")))
      .orderBy(b.TITLE)
      .fetch();
```

We'll see how the aliasing works later in the section about aliased tables

jOOQ as an internal domain specific language in Java (a.k.a. the DSL API)

Many other frameworks have similar APIs with similar feature sets. Yet, what makes jOOQ special is its informal BNF notation modelling a unified SQL dialect suitable for many vendor-specific dialects, and implementing that BNF notation as a hierarchy of interfaces in Java. This concept is extremely powerful, when using jOOQ in modern IDEs with syntax completion. Not only can you code much faster, your SQL code will be compile-checked to a certain extent. An example of a DSL query equivalent to the previous one is given here:

```
DSLContext create = DSL.using(connection, dialect);
Result<?> result = create.select()
                         .from(AUTHOR)
                         .join(BOOK).on(BOOK.AUTHOR_ID.eq(AUTHOR.ID))
                         .fetch();
```

Unlike other, simpler frameworks that use "fluent APIs" or "method chaining", jOOQ's BNF-based interface hierarchy will not allow bad query syntax. The following will not compile, for instance:

```
DSLContext create = DSL.using(connection, dialect);
Result<?> result = create.select()
                         .join(BOOK).on(BOOK.AUTHOR_ID.eq(AUTHOR.ID))
                      //  ^^^^ "join" is not possible here
                         .from(AUTHOR)
                         .fetch();

Result<?> result = create.select()
                         .from(AUTHOR)
                         .join(BOOK)
                         .fetch();
                      //  ^^^^^ "on" is missing here

Result<?> result = create.select(rowNumber())
                      //         ^^^^^^^^^ "over()" is missing here
                         .from(AUTHOR)
                         .fetch();

Result<?> result = create.select()
                         .from(AUTHOR)
                         .where(AUTHOR.ID.in(select(BOOK.TITLE).from(BOOK)))
                      //                     ^^^^^^^^^^^^^^^^^^
                      // AUTHOR.ID is of type Field<Integer> but subselect returns Record1<String>
                         .fetch();

Result<?> result = create.select()
                         .from(AUTHOR)
                         .where(AUTHOR.ID.in(select(BOOK.AUTHOR_ID, BOOK.ID).from(BOOK)))
                      //                     ^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^
                      // AUTHOR.ID is of degree 1 but subselect returns Record2<Integer, Integer>
                         .fetch();
```

History of SQL building and incremental query building (a.k.a. the model API)

Historically, jOOQ started out as an object-oriented SQL builder library like any other. This meant that all queries and their syntactic components were modeled as so-called QueryParts, which delegate SQL rendering and variable binding to child components. This part of the API will be referred to as the model API (or non-DSL API), which is still maintained and used internally by jOOQ for incremental query building. An example of incremental query building is given here:

```
DSLContext create = DSL.using(connection, dialect);
SelectQuery<Record> query = create.selectQuery();
query.addFrom(AUTHOR);

// Join books only under certain circumstances
if (join) {
    query.addJoin(BOOK, BOOK.AUTHOR_ID.eq(AUTHOR.ID));
}

Result<?> result = query.fetch();
```

This query is equivalent to the one shown before using the DSL syntax. In fact, internally, the DSL API constructs precisely this SelectQuery object. Note, that you can always access the SelectQuery object to switch between DSL and model APIs:

```
DSLContext create = DSL.using(connection, dialect);
SelectFinalStep<?> select = create.select().from(AUTHOR);

// Add the JOIN clause on the internal QueryObject representation
SelectQuery<?> query = select.getQuery();
query.addJoin(BOOK, BOOK.AUTHOR_ID.eq(AUTHOR.ID));
```

Mutability

Note, that for historic reasons, the DSL API mixes mutable and immutable behaviour with respect to the internal representation of the QueryPart being constructed. While creating conditional expressions,

column expressions (such as functions) assumes immutable behaviour, creating SQL statements does not. In other words, the following can be said:

```
// Conditional expressions (immutable)
// -----------------------------------
Condition a = BOOK.TITLE.eq("1984");
Condition b = BOOK.TITLE.eq("Animal Farm");

// The following can be said
a       != a.or(b); // or() does not modify a
a.or(b) != a.or(b); // or() always creates new objects

// Statements (mutable)
// --------------------
SelectFromStep<?> s1 = select();
SelectJoinStep<?> s2 = s1.from(BOOK);
SelectJoinStep<?> s3 = s1.from(AUTHOR);

// The following can be said
s1 == s2; // The internal object is always the same
s2 == s3; // The internal object is always the same
```

On the other hand, beware that you can always extract and modify bind values from any QueryPart.

4.3.2. The WITH clause

The SQL:1999 standard specifies the WITH clause to be an optional clause for the SELECT statement, in order to specify common table expressions (also: CTE). Many other databases (such as PostgreSQL, SQL Server) also allow for using common table expressions also in other DML clauses, such as the INSERT statement, UPDATE statement, DELETE statement, or MERGE statement.

When using common table expressions with jOOQ, there are essentially two approaches:

- Declaring and assigning common table expressions explicitly to names
- Inlining common table expressions into a SELECT statement

Explicit common table expressions

The following example makes use of names to construct common table expressions, which can then be supplied to a WITH clause or a FROM clause of a SELECT statement:

```
-- Pseudo-SQL for a common table expression specification
"t1" ("f1", "f2") AS (SELECT 1, 'a')
```

```
// Code for creating a CommonTableExpression instance
name("t1").fields("f1", "f2").as(select(val(1), val("a")));
```

The above expression can be assigned to a variable in Java and then be used to create a full SELECT statement:

```
WITH "t1" ("f1", "f2") AS (SELECT 1, 'a'),
     "t2" ("f3", "f4") AS (SELECT 2, 'b')
SELECT
    "t1"."f1" + "t2"."f3" AS "add",
    "t1"."f2" || "t2"."f4" AS "concat"
FROM "t1", "t2"
;
```

```
CommonTableExpression<Record2<Integer, String>> t1 =
  name("t1").fields("f1", "f2").as(select(val(1), val("a")));
CommonTableExpression<Record2<Integer, String>> t2 =
  name("t2").fields("f3", "f4").as(select(val(2), val("b")));

Result<?> result2 =
create.with(t1)
      .with(t2)
      .select(
          t1.field("f1").add(t2.field("f3")).as("add"),
          t1.field("f2").concat(t2.field("f4")).as("concat"))
      .from(t1, t2)
      .fetch();
```

Note that the org.jooq.CommonTableExpression type extends the commonly used org.jooq.Table type, and can thus be used wherever a table can be used.

Inlined common table expressions

If you're just operating on plain SQL, you may not need to keep intermediate references to such common table expressions. An example of such usage would be this:

```
WITH "a" AS (SELECT
               1 AS "x",
               'a' AS "y"
             )
SELECT
FROM "a"
;
```

```
create.with("a").as(select(
                      val(1).as("x"),
                      val("a").as("y")
                    ))
      .select()
      .from(table(name("a")))
      .fetch();
```

Recursive common table expressions

The various SQL dialects do not agree on the use of RECURSIVE when writing recursive common table expressions. When using jOOQ, always use the DSLContext.withRecursive() or DSL.withRecursive() methods, and jOOQ will render the RECURSIVE keyword, if needed.

4.3.3. The SELECT statement

When you don't just perform CRUD (i.e. SELECT * FROM your_table WHERE ID = ?), you're usually generating new record 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:

SELECT from a complex table expression

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

```
// And with jOOQ...


DSLContext create = DSL.using(connection, dialect);

create.select(AUTHOR.FIRST_NAME, AUTHOR.LAST_NAME, count())
      .from(AUTHOR)
      .join(BOOK).on(BOOK.AUTHOR_ID.eq(AUTHOR.ID))
      .where(BOOK.LANGUAGE.eq("DE"))
      .and(BOOK.PUBLISHED.gt("2008-01-01"))
      .groupBy(AUTHOR.FIRST_NAME, AUTHOR.LAST_NAME)
      .having(count().gt(5))
      .orderBy(AUTHOR.LAST_NAME.asc().nullsFirst())
      .limit(2)
      .offset(1)
      .forUpdate()
      .fetch();
```

Details about the various clauses of this query will be provided in subsequent sections.

SELECT from single tables

A very similar, but limited API is available, if you want to select from single tables in order to retrieve TableRecords or even UpdatableRecords. The decision, which type of select to create is already made at the very first step, when you create the SELECT statement with the DSL or DSLContext types:

```
public <R extends Record> SelectWhereStep<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:

```
BookRecord book = create.selectFrom(BOOK)
                        .where(BOOK.LANGUAGE.eq("DE"))
                        .orderBy(BOOK.TITLE)
                        .fetchAny();
```

The "reduced" SELECT API is limited in the way that it skips DSL access to any of these clauses:

- The SELECT clause
- The JOIN clause

In most parts of this manual, it is assumed that you do not use the "reduced" SELECT API. For more information about the simple SELECT API, see the manual's section about fetching strongly or weakly typed records.

4.3.3.1. The SELECT clause

The SELECT clause lets you project your own record types, referencing table fields, functions, arithmetic expressions, etc. The DSL type provides several methods for expressing a SELECT clause:

```
-- The SELECT clause
SELECT BOOK.ID, BOOK.TITLE
SELECT BOOK.ID, TRIM(BOOK.TITLE)
```

```
// Provide a varargs Fields list to the SELECT clause:
Select<?> s1 = create.select(BOOK.ID, BOOK.TITLE);
Select<?> s2 = create.select(BOOK.ID, trim(BOOK.TITLE));
```

Some commonly used projections can be easily created using convenience methods:

```
-- Simple SELECTs
SELECT COUNT(*)
SELECT 0 -- Not a bind variable
SELECT 1 -- Not a bind variable
```

```
// Select commonly used values
Select<?> select1 = create.selectCount().fetch();
Select<?> select2 = create.selectZero().fetch();
Select<?> select2 = create.selectOne().fetch();
```

See more details about functions and expressions in the manual's section about Column expressions

The SELECT DISTINCT clause

The DISTINCT keyword can be included in the method name, constructing a SELECT clause

```
SELECT DISTINCT BOOK.TITLE
```

```
Select<?> select1 = create.selectDistinct(BOOK.TITLE).fetch();
```

SELECT *

jOOQ supports the asterisk operator in projections both as a qualified asterisk (through Table.asterisk()) and as an unqualified asterisk (through DSL.asterisk()). It is also possible to omit the projection entirely, in case of which an asterisk may appear in generated SQL, if not all column names are known to jOOQ.

```
// Explicitly selects all columns available from BOOK - No asterisk
create.select().from(BOOK).fetch();

// Explicitly selects all columns available from BOOK and AUTHOR - No asterisk
create.select().from(BOOK, AUTHOR).fetch();
create.select().from(BOOK).crossJoin(AUTHOR).fetch();

// Renders a SELECT * statement, as columns are unknown to jOOQ - Implicit unqualified asterisk
create.select().from(table(name("BOOK"))).fetch();

// Renders a SELECT * statement - Explicit unqualified asterisk
create.select(asterisk()).from(BOOK).fetch();

// Renders a SELECT BOOK.* statement - Explicit qualified asterisk
create.select(BOOK.asterisk()).from(BOOK).fetch();
create.select(BOOK.asterisk(), AUTHOR.asterisk()).from(BOOK, AUTHOR).fetch();
```

With all of the above syntaxes, the row type (as discussed below) is unknown to jOOQ and to the Java compiler.

It is worth mentioning that in many cases, using an asterisk is a sign of an inefficient query because if not all columns are needed, too much data is transferred between client and server, plus some joins that could be eliminated otherwise, cannot.

Typesafe projections with degree up to 22

Since jOOQ 3.0, records and row value expressions up to degree 22 are now generically typesafe. This is reflected by an overloaded SELECT (and SELECT DISTINCT) API in both DSL and DSLContext. An extract from the DSL type:

```
// Non-typesafe select methods:
public static SelectSelectStep<Record> select(Collection<? extends Field<?>> fields);
public static SelectSelectStep<Record> select(Field<?>... fields);

// Typesafe select methods:
public static <T1>         SelectSelectStep<Record1<T1>>         select(Field<T1> field1);
public static <T1, T2>     SelectSelectStep<Record2<T1, T2>>     select(Field<T1> field1, Field<T2> field2);
public static <T1, T2, T3> SelectSelectStep<Record3<T1, T2, T3>> select(Field<T1> field1, Field<T2> field2, Field<T3> field3);
// [...]
```

Since the generic R type is bound to some Record[N], the associated T type information can be used in various other contexts, e.g. the IN predicate. Such a SELECT statement can be assigned typesafely:

```
Select<Record2<Integer, String>> s1 = create.select(BOOK.ID, BOOK.TITLE);
Select<Record2<Integer, String>> s2 = create.select(BOOK.ID, trim(BOOK.TITLE));
```

For more information about typesafe record types with degree up to 22, see the manual's section about Record1 to Record22.

4.3.3.2. The FROM clause

The SQL FROM clause allows for specifying any number of table expressions to select data from. The following are examples of how to form normal FROM clauses:

```
SELECT 1 FROM BOOK
SELECT 1 FROM BOOK, AUTHOR
SELECT 1 FROM BOOK "b", AUTHOR "a"
```

```
create.selectOne().from(BOOK).fetch();
create.selectOne().from(BOOK, AUTHOR).fetch();
create.selectOne().from(BOOK.as("b"), AUTHOR.as("a")).fetch();
```

Read more about aliasing in the manual's section about aliased tables.

More advanced table expressions

Apart from simple tables, you can pass any arbitrary table expression to the jOOQ FROM clause. This may include unnested cursors in Oracle:

```
SELECT *
FROM TABLE(
    DBMS_XPLAN.DISPLAY_CURSOR(null, null, 'ALLSTATS')
);
```

```
create.select()
      .from(table(
          DbmsXplan.displayCursor(null, null, "ALLSTATS")
      ).fetch();
```

Note, in order to access the DbmsXplan package, you can use the code generator to generate Oracle's SYS schema.

Selecting FROM DUAL with jOOQ

In many SQL dialects, FROM is a mandatory clause, in some it isn't. jOOQ allows you to omit the FROM clause, returning just one record. An example:

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

```
DSL.using(SQLDialect.ORACLE).selectOne().fetch();
DSL.using(SQLDialect.POSTGRES).selectOne().fetch();
```

Read more about dual or dummy tables in the manual's section about the DUAL table. The following are examples of how to form normal FROM clauses:

4.3.3.3. The JOIN clause

jOOQ supports many different types of standard and non-standard SQL JOIN operations:

- [INNER] JOIN
- LEFT [OUTER] JOIN
- RIGHT [OUTER] JOIN
- FULL OUTER JOIN
- LEFT SEMI JOIN
- LEFT ANTI JOIN
- CROSS JOIN
- NATURAL JOIN
- NATURAL LEFT [OUTER] JOIN
- NATURAL RIGHT [OUTER] JOIN

Besides, jOOQ also supports

- CROSS APPLY (T-SQL and Oracle 12c specific)
- OUTER APPLY (T-SQL and Oracle 12c specific)
- LATERAL derived tables (PostgreSQL and Oracle 12c)
- partitioned outer join

All of these JOIN methods can be called on org.jooq.Table types, or directly after the FROM clause for convenience. The following example joins AUTHOR and BOOK

```
DSLContext create = DSL.using(connection, dialect);

// Call "join" directly on the AUTHOR table
Result<?> result = create.select()
                         .from(AUTHOR.join(BOOK)
                                     .on(BOOK.AUTHOR_ID.eq(AUTHOR.ID)))
                         .fetch();

// Call "join" on the type returned by "from"
Result<?> result = create.select()
                         .from(AUTHOR)
                         .join(BOOK)
                         .on(BOOK.AUTHOR_ID.eq(AUTHOR.ID))
                         .fetch();
```

The two syntaxes will produce the same SQL statement. However, calling "join" on org.jooq.Table objects allows for more powerful, nested JOIN expressions (if you can handle the parentheses):

```
SELECT *
FROM AUTHOR
LEFT OUTER JOIN (
  BOOK JOIN BOOK_TO_BOOK_STORE
       ON BOOK_TO_BOOK_STORE.BOOK_ID = BOOK.ID
)
ON BOOK.AUTHOR_ID = AUTHOR.ID
```

```
// Nest joins and provide JOIN conditions only at the end
create.select()
      .from(AUTHOR
      .leftOuterJoin(BOOK
        .join(BOOK_TO_BOOK_STORE)
        .on(BOOK_TO_BOOK_STORE.BOOK_ID.eq(BOOK.ID)))
      .on(BOOK.AUTHOR_ID.eq(AUTHOR.ID)))
      .fetch();
```

- See the section about conditional expressions to learn more about the many ways to create org.jooq.Condition objects in jOOQ.
- See the section about table expressions to learn about the various ways of referencing org.jooq.Table objects in jOOQ

JOIN ON KEY, convenience provided by jOOQ

Surprisingly, the SQL standard does not allow to formally JOIN on well-known foreign key relationship information. Naturally, when you join BOOK to AUTHOR, you will want to do that based on the BOOK.AUTHOR_ID foreign key to AUTHOR.ID primary key relation. Not being able to do this in SQL leads to a lot of repetitive code, re-writing the same JOIN predicate again and again - especially, when your foreign keys contain more than one column. With jOOQ, when you use code generation, you can use foreign key constraint information in JOIN expressions as such:

```
SELECT *
FROM AUTHOR
JOIN BOOK ON BOOK.AUTHOR_ID = AUTHOR.ID
```

```
create.select()
      .from(AUTHOR)
      .join(BOOK).onKey()
      .fetch();
```

In case of ambiguity, you can also supply field references for your foreign keys, or the generated foreign key reference to the onKey() method.

Note that formal support for the Sybase JOIN ON KEY syntax is on the roadmap.

The JOIN USING syntax

Most often, you will provide jOOQ with JOIN conditions in the JOIN .. ON clause. SQL supports a different means of specifying how two tables are to be joined. This is the JOIN .. USING clause. Instead of a condition, you supply a set of fields whose names are common to both tables to the left and right of a JOIN operation. This can be useful when your database schema has a high degree of relational normalisation. An example:

```
-- Assuming that both tables contain AUTHOR_ID columns
SELECT *
FROM AUTHOR
JOIN BOOK USING (AUTHOR_ID)
```

```
// join(...).using(...)
create.select()
      .from(AUTHOR)
      .join(BOOK).using(AUTHOR.AUTHOR_ID)
      .fetch();
```

In schemas with high degrees of normalisation, you may also choose to use NATURAL JOIN, which takes no JOIN arguments as it joins using all fields that are common to the table expressions to the left and to the right of the JOIN operator. An example:

```
-- Assuming that both tables contain AUTHOR_ID columns
SELECT *
FROM AUTHOR
NATURAL JOIN BOOK
```

```
// naturalJoin(...)
create.select()
      .from(AUTHOR)
      .naturalJoin(BOOK)
      .fetch();
```

Oracle's partitioned OUTER JOIN

Oracle SQL ships with a special syntax available for OUTER JOIN clauses. According to the Oracle documentation about partitioned outer joins this can be used to fill gaps for simplified analytical calculations. jOOQ only supports putting the PARTITION BY clause to the right of the OUTER JOIN clause. The following example will create at least one record per AUTHOR and per existing value in BOOK.PUBLISHED_IN, regardless if an AUTHOR has actually published a book in that year.

```
SELECT *
FROM AUTHOR
LEFT OUTER JOIN BOOK
PARTITION BY (PUBLISHED_IN)
ON BOOK.AUTHOR_ID = AUTHOR.ID
```

```
create.select()
      .from(AUTHOR)
      .leftOuterJoin(BOOK)
      .partitionBy(BOOK.PUBLISHED_IN)
      .on(BOOK.AUTHOR_ID.eq(AUTHOR.ID))
      .fetch();
```

SEMI JOIN and ANTI JOIN

Very few databases (e.g. Apache Impala) ship with a built-in syntax for { LEFT | RIGHT } SEMI JOIN and { LEFT | RIGHT } ANTI JOIN, which are much more concise versions of the SQL standard IN / EXISTS and NOT IN / NOT EXISTS predicates. The idea is that the JOIN syntax is expressed where it belongs, in the FROM clause, not in the WHERE clause.

Since jOOQ 3.7, these types of JOIN are also supported and they're emulated using EXISTS and NOT EXISTS respectively.

Here's how SEMI JOIN translates to EXISTS.

```
SELECT FIRST_NAME, LAST_NAME
FROM AUTHOR
WHERE EXISTS (
  SELECT 1 FROM BOOK WHERE AUTHOR.ID = BOOK.AUTHOR_ID
)
```

```
create.select(AUTHOR.FIRST_NAME, AUTHOR.LAST_NAME)
      .from(AUTHOR)
      .leftSemiJoin(BOOK)
      .on(BOOK.AUTHOR_ID.eq(AUTHOR.ID))
      .fetch();
```

And here's how ANTI JOIN translates to NOT EXISTS

```
SELECT FIRST_NAME, LAST_NAME
FROM AUTHOR
WHERE NOT EXISTS (
  SELECT 1 FROM BOOK WHERE AUTHOR.ID = BOOK.AUTHOR_ID
)
```

```
create.select(AUTHOR.FIRST_NAME, AUTHOR.LAST_NAME)
      .from(AUTHOR)
      .leftAntiJoin(BOOK)
      .on(BOOK.AUTHOR_ID.eq(AUTHOR.ID))
      .fetch();
```

T-SQL's CROSS APPLY and OUTER APPLY

T-SQL has long known what the SQL standard calls lateral derived tables, lateral joins using the APPLY keyword. To every row resulting from the table expression on the left, we apply the table expression on the right. This is extremely useful for table-valued functions, which are also supported by jOOQ. Some examples:

```
DSL.using(configuration)
   .select()
   .from(AUTHOR,
         lateral(select(count().as("c"))
                .from(BOOK)
                .where(BOOK.AUTHOR_ID.eq(AUTHOR.ID)))
   )
   .fetch("c", int.class);
```

The above example shows standard usage of the LATERAL keyword to connect a derived table to the previous table in the FROM clause. A similar statement can be written in T-SQL:

```
DSL.using(configuration)
   .from(AUTHOR)
   .crossApply(
       select(count().as("c"))
      .from(BOOK)
      .where(BOOK.AUTHOR_ID.eq(AUTHOR.ID))
   )
   .fetch("c", int.class)
```

While not all forms of LATERAL JOIN have an equivalent APPLY syntax, the inverse is true, and jOOQ can thus emulate OUTER APPLY and CROSS APPLY using LATERAL JOIN.

LATERAL JOIN or CROSS APPLY are particularly useful together with table valued functions, which are also supported by jOOQ.

4.3.3.4. Implicit JOIN

In SQL, a lot of explicit JOIN clauses are written simply to retrieve a parent table's column from a given child table. For example, we'll write:

```
-- Get all books, their authors, and their respective language
SELECT
  a.first_name,
  a.last_name,
  b.title,
  l.cd AS language
FROM book b
JOIN author a ON b.author_id = a.id
JOIN language l ON b.language_id = l.id;

-- Count the number of books by author and language
SELECT
  a.first_name,
  a.last_name,
  l.cd AS language,
  COUNT(*)
FROM book
JOIN author a ON b.author_id = a.id
JOIN language l ON b.language_id = l.id
GROUP BY a.id, a.first_name, a.last_name, l.cd
ORDER BY a.first_name, a.last_name, l.cd
```

There is quite a bit of syntactic ceremony (or we could even call it "noise") to get a relatively simple job done. A much simpler notation would be using implicit joins:

```
-- Get all books, their authors, and their respective language
SELECT
  b.author.first_name,
  b.author.last_name,
  b.title,
  b.language.cd AS language
FROM book b;

-- Count the number of books by author and language
SELECT
  b.author.first_name,
  b.author.last_name,
  b.language.cd AS language,
  COUNT(*)
FROM book
GROUP BY
  b.author_id,
  b.author.first_name,
  b.author.last_name,
  b.language.cd
ORDER BY
  b.author.first_name,
  b.author.last_name,
  b.language.cd
```

Notice how this alternative notation (depending on your taste) may look more tidy and straightforward, as the semantics of accessing a table's parent table (or an entity's parent entity) is straightforward.

From jOOQ 3.11 onwards, this syntax is supported for to-one relationship navigation. The code generator produces relevant navigation methods on generated tables, which can be used in a type safe way. The navigation method names are:

- The parent table name, if there is only one foreign key between child table and parent table
- The foreign key name, if there are more than one foreign keys between child table and parent table

This default behaviour can be overridden by using a Code Generator Strategy.

The jOOQ version of the previous queries looks like this:

```
// Get all books, their authors, and their respective language
create.select(
          BOOK.author().FIRST_NAME,
          BOOK.author().LAST_NAME,
          BOOK.TITLE,
          BOOK.language().CD.as("language"))
      .from(BOOK)
      .fetch();

// Count the number of books by author and language
create.select(
          BOOK.author().FIRST_NAME,
          BOOK.author().LAST_NAME,
          BOOK.language().CD.as("language"),
          count())
      .from(BOOK)
      .groupBy(
          BOOK.AUTHOR_ID,
          BOOK.author().FIRST_NAME,
          BOOK.author().LAST_NAME,
          BOOK.language().CD)
      .orderBy(
          BOOK.author().FIRST_NAME,
          BOOK.author().LAST_NAME,
          BOOK.language().CD)
      .fetch();
```

The generated SQL is almost identical to the original one - there is no performance penalty to this syntax.

How it works

During the SQL generation phase, implicit join paths are replaced by generated aliases for the path's last table. The paths are translated to a join graph, which is always LEFT JOINed to the path's "root table". If two paths share a common prefix, that prefix is also shared in the join graph.

Future versions of jOOQ may choose to generate correlated subqueries or inner joins where this may seem more appropriate, if the query semantics doesn't change through that.

Known limitations

- Implicit JOINs are currently only supported in SELECT statements (including any type of subquery), but not in the WHERE clause of UPDATE statements or DELETE statements, for instance.
- Implicit JOINs can currently only be used to access columns, not to produce joins. I.e. it is not possible to write things like FROM book IMPLICIT JOIN book.author
- Implicit JOINs are added to the SQL string after the entire SQL statement is available, for performance reasons. This means, that VisitListener SPI implementations cannot observe implicitly joined tables

4.3.3.5. The WHERE clause

The WHERE clause can be used for JOIN or filter predicates, in order to restrict the data returned by the table expressions supplied to the previously specified from clause and join clause. Here is an example:

```
SELECT *
FROM BOOK
WHERE AUTHOR_ID = 1
AND TITLE = '1984'
```

```
create.select()
      .from(BOOK)
      .where(BOOK.AUTHOR_ID.eq(1))
      .and(BOOK.TITLE.eq("1984"))
      .fetch();
```

The above syntax is convenience provided by jOOQ, allowing you to connect the org.jooq.Condition supplied in the WHERE clause with another condition using an AND operator. You can of course also create a more complex condition and supply that to the WHERE clause directly (observe the different placing of parentheses). The results will be the same:

```
SELECT *
FROM BOOK
WHERE AUTHOR_ID = 1
AND TITLE = '1984'
```

```
create.select()
      .from(BOOK)
      .where(BOOK.AUTHOR_ID.eq(1).and(
            BOOK.TITLE.eq("1984")))
      .fetch();
```

You will find more information about creating conditional expressions later in the manual.

4.3.3.6. The CONNECT BY clause

The Oracle database knows a very succinct syntax for creating hierarchical queries: the CONNECT BY clause, which is fully supported by jOOQ, including all related functions and pseudo-columns. A more or less formal definition of this clause is given here:

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

An example for an iterative query, iterating through values between 1 and 5 is this:


```
SELECT LEVEL
FROM DUAL
CONNECT BY LEVEL <= 5
```

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

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

```
SELECT
  SUBSTR(SYS_CONNECT_BY_PATH(DIRECTORY.NAME, '/'), 2)
FROM DIRECTORY
CONNECT BY
  PRIOR DIRECTORY.ID = DIRECTORY.PARENT_ID
START WITH DIRECTORY.PARENT_ID IS NULL
ORDER BY 1
```

```
.select(
   sysConnectByPath(DIRECTORY.NAME, "/").substring(2))
.from(DIRECTORY)
.connectBy(
   prior(DIRECTORY.ID).eq(DIRECTORY.PARENT_ID))
.startWith(DIRECTORY.PARENT_ID.isNull())
.orderBy(1)
.fetch();
```

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...
```

Some of the supported functions and pseudo-columns are these (available from the DSL):

- LEVEL
- CONNECT_BY_IS_CYCLE
- CONNECT_BY_IS_LEAF
- CONNECT_BY_ROOT
- SYS_CONNECT_BY_PATH
- PRIOR

Note that this syntax is also supported in the CUBRID database and might be emulated in other dialects supporting common table expressions in the future.

ORDER SIBLINGS

The Oracle database allows for specifying a SIBLINGS keyword in the ORDER BY clause. Instead of ordering the overall result, this will only order siblings among each other, keeping the hierarchy intact. An example is given here:

```
SELECT DIRECTORY.NAME
FROM DIRECTORY
CONNECT BY
  PRIOR DIRECTORY.ID = DIRECTORY.PARENT_ID
START WITH DIRECTORY.PARENT_ID IS NULL
ORDER SIBLINGS BY 1
```

```
.select(DIRECTORY.NAME)
.from(DIRECTORY)
.connectBy(
   prior(DIRECTORY.ID).eq(DIRECTORY.PARENT_ID))
.startWith(DIRECTORY.PARENT_ID.isNull())
.orderSiblingsBy(1)
.fetch();
```

4.3.3.7. The GROUP BY clause

GROUP BY can be used to create unique groups of data, to form aggregations, to remove duplicates and for other reasons. It will transform your previously defined set of table expressions, and return only one record per unique group as specified in this clause. For instance, you can group books by BOOK.AUTHOR_ID:

```
SELECT AUTHOR_ID, COUNT(*)
FROM BOOK
GROUP BY AUTHOR_ID
```

```
create.select(BOOK.AUTHOR_ID, count())
      .from(BOOK)
      .groupBy(BOOK.AUTHOR_ID)
      .fetch();
```

The above example counts all books per author.

Note, as defined in the SQL standard, when grouping, you may no longer project any columns that are not a formal part of the GROUP BY clause, or aggregate functions.

Empty GROUP BY clauses

jOOQ supports empty GROUP BY () clause as well. This will result in SELECT statements that return only one record.

```
SELECT COUNT(*)
FROM BOOK
GROUP BY ()
```

```
create.selectCount()
      .from(BOOK)
      .groupBy()
      .fetch();
```

ROLLUP(), CUBE() and GROUPING SETS()

Some databases support the SQL standard grouping functions and some extensions thereof. See the manual's section about grouping functions for more details.

4.3.3.8. The HAVING clause

The HAVING clause is commonly used to further restrict data resulting from a previously issued GROUP BY clause. An example, selecting only those authors that have written at least two books:

```
SELECT AUTHOR_ID, COUNT(*)
FROM BOOK
GROUP BY AUTHOR_ID
HAVING COUNT(*) >= 2
```

```
create.select(BOOK.AUTHOR_ID, count())
      .from(BOOK)
      .groupBy(AUTHOR_ID)
      .having(count().ge(2))
      .fetch();
```

According to the SQL standard, you may omit the GROUP BY clause and still issue a HAVING clause. This will implicitly GROUP BY (). jOOQ also supports this syntax. The following example selects one record, only if there are at least 4 books in the books table:

```
SELECT COUNT(*)
FROM BOOK
HAVING COUNT(*) >= 4
```

```
create.select(count(*))
      .from(BOOK)
      .having(count().ge(4))
      .fetch();
```

4.3.3.9. The WINDOW clause

The SQL:2003 standard supports a WINDOW clause that allows for specifying WINDOW frames for reuse in SELECT clauses and ORDER BY clauses. It is natively supported by:

- H2
- MySQL
- PostgreSQL
- SQLite
- Sybase SQL Anywhere

```
SELECT
  LAG(first_name, 1) OVER w "prev",
  first_name,
  LEAD(first_name, 1) OVER w "next"
FROM author
WINDOW w AS (ORDER first_name)
ORDER BY first_name DESC
```

```
WindowDefinition w = name("w").as(
  orderBy(PEOPLE.FIRST_NAME));

 select(
   lag(AUTHOR.FIRST_NAME, 1).over(w).as("prev"),
   AUTHOR.FIRST_NAME,
   lead(AUTHOR.FIRST_NAME, 1).over(w).as("next"))
.from(AUTHOR)
.window(w)
.orderBy(AUTHOR.FIRST_NAME.desc())
.fetch();
```

Note that in order to create such a window definition, we need to first create a name reference using DSL.name().

Even if only PostgreSQL and Sybase SQL Anywhere natively support this great feature, jOOQ can emulate it by expanding any org.jooq.WindowDefinition and org.jooq.WindowSpecification types that you pass to the window() method - if the database supports window functions at all.

Some more information about window functions and the WINDOW clause can be found on our blog: http://blog.jooq.org/2013/11/03/probably-the-coolest-sql-feature-window-functions/

4.3.3.10. The ORDER BY clause

Databases are allowed to return data in any arbitrary order, unless you explicitly declare that order in the ORDER BY clause. In jOOQ, this is straight-forward:

```
SELECT AUTHOR_ID, TITLE
FROM BOOK
ORDER BY AUTHOR_ID ASC, TITLE DESC
```

```
create.select(BOOK.AUTHOR_ID, BOOK.TITLE)
      .from(BOOK)
      .orderBy(BOOK.AUTHOR_ID.asc(), BOOK.TITLE.desc())
      .fetch();
```

Any jOOQ column expression (or field) can be transformed into an org.jooq.SortField by calling the asc() and desc() methods.

Ordering by field index

The SQL standard allows for specifying integer literals (literals, not bind values!) to reference column indexes from the projection (SELECT clause). This may be useful if you do not want to repeat a lengthy expression, by which you want to order - although most databases also allow for referencing aliased column references in the ORDER BY clause. An example of this is given here:

```
SELECT AUTHOR_ID, TITLE
FROM BOOK
ORDER BY 1 ASC, 2 DESC
```

```
create.select(BOOK.AUTHOR_ID, BOOK.TITLE)
      .from(BOOK)
      .orderBy(one().asc(), inline(2).desc())
      .fetch();
```

Note, how one() is used as a convenience short-cut for inline(1)

Ordering and NULLS

A few databases support the SQL standard "null ordering" clause in sort specification lists, to define whether NULL values should come first or last in an ordered result.

```
SELECT
  AUTHOR.FIRST_NAME,
  AUTHOR.LAST_NAME
FROM AUTHOR
ORDER BY LAST_NAME ASC,
         FIRST_NAME ASC NULLS LAST
```

```
create.select(
         AUTHOR.FIRST_NAME,
         AUTHOR.LAST_NAME)
      .from(AUTHOR)
      .orderBy(AUTHOR.LAST_NAME.asc(),
               AUTHOR.FIRST_NAME.asc().nullsLast())
      .fetch();
```

If your database doesn't support this syntax, jOOQ emulates it using a CASE expression as follows

```
SELECT
  AUTHOR.FIRST_NAME, AUTHOR.LAST_NAME
FROM AUTHOR
ORDER BY LAST_NAME ASC,
         CASE WHEN FIRST_NAME IS NULL
              THEN 1 ELSE 0 END ASC,
         FIRST_NAME ASC
```

Ordering using CASE expressions

Using CASE expressions in SQL ORDER BY clauses is a common pattern, if you want to introduce some sort indirection / sort mapping into your queries. As with SQL, you can add any type of column expression into your ORDER BY clause. For instance, if you have two favourite books that you always want to appear on top, you could write:

```
SELECT *
FROM BOOK
ORDER BY CASE TITLE
         WHEN '1984' THEN 0
         WHEN 'Animal Farm' THEN 1
         ELSE 2 END ASC
```

```
create.select()
      .from(BOOK)
      .orderBy(choose(BOOK.TITLE)
               .when("1984", 0)
               .when("Animal Farm", 1)
               .otherwise(2).asc())
      .fetch();
```

But writing these things can become quite verbose. jOOQ supports a convenient syntax for specifying sort mappings. The same query can be written in jOOQ as such:

```
create.select()
      .from(BOOK)
      .orderBy(BOOK.TITLE.sortAsc("1984", "Animal Farm"))
      .fetch();
```

More complex sort indirections can be provided using a Map:

```
create.select()
      .from(BOOK)
      .orderBy(BOOK.TITLE.sort(new HashMap<String, Integer>() {{
          put("1984", 1);
          put("Animal Farm", 13);
          put("The jOOQ book", 10);
      }}))
      .fetch();
```

Of course, you can combine this feature with the previously discussed NULLS FIRST / NULLS LAST feature. So, if in fact these two books are the ones you like least, you can put all NULLS FIRST (all the other books):

```
create.select()
      .from(BOOK)
      .orderBy(BOOK.TITLE.sortAsc("1984", "Animal Farm").nullsFirst())
      .fetch();
```

jOOQ's understanding of SELECT .. ORDER BY

The SQL standard defines that a "query expression" can be ordered, and that query expressions can contain UNION, INTERSECT and EXCEPT clauses, whose subqueries cannot be ordered. While this is defined as such in the SQL standard, many databases allowing for the LIMIT clause in one way or another, do not adhere to this part of the SQL standard. Hence, jOOQ allows for ordering all SELECT statements, regardless whether they are constructed as a part of a UNION or not. Corner-cases are handled internally by jOOQ, by introducing synthetic subselects to adhere to the correct syntax, where this is needed.

Oracle's ORDER SIBLINGS BY clause

jOOQ also supports Oracle's SIBLINGS keyword to be used with ORDER BY clauses for hierarchical queries using CONNECT BY

4.3.3.11. The LIMIT .. OFFSET clause

While being extremely useful for every application that does pagination, or just to limit result sets to reasonable sizes, this clause is not yet part of any SQL standard (up until SQL:2008). Hence, there exist a variety of possible implementations in various SQL dialects, concerning this limit clause. jOOQ chose to implement the LIMIT .. OFFSET clause as understood and supported by MySQL, H2, HSQLDB, Postgres, and SQLite. Here is an example of how to apply limits with jOOQ:

```
create.select().from(BOOK).limit(1).offset(2).fetch();
```

This will limit the result to 1 books starting with the 2nd book (starting at offset 0!). limit() is supported in all dialects, offset() in all but Sybase ASE, which has no reasonable means to emulate it. This is how jOOQ trivially emulates the above query in various SQL dialects with native OFFSET pagination support:

```
-- MySQL, H2, HSQLDB, Postgres, and SQLite
SELECT * FROM BOOK LIMIT 1 OFFSET 2

-- CUBRID supports a MySQL variant of the LIMIT .. OFFSET clause
SELECT * FROM BOOK LIMIT 2, 1

-- Derby, SQL Server 2012, Oracle 12c, the SQL:2008 standard
SELECT * FROM BOOK OFFSET 2 ROWS FETCH NEXT 1 ROWS ONLY

-- Informix has SKIP .. FIRST support
SELECT SKIP 2 FIRST 1 * FROM BOOK

-- Ingres (almost the SQL:2008 standard)
SELECT * FROM BOOK OFFSET 2 FETCH FIRST 1 ROWS ONLY

-- Firebird
SELECT * FROM BOOK ROWS 2 TO 3

-- Sybase SQL Anywhere
SELECT TOP 1 ROWS START AT 3 * FROM BOOK

-- DB2 (almost the SQL:2008 standard, without OFFSET)
SELECT * FROM BOOK FETCH FIRST 1 ROWS ONLY

-- Sybase ASE, SQL Server 2008 (without OFFSET)
SELECT TOP 1 * FROM BOOK
```

Things get a little more tricky in those databases that have no native idiom for OFFSET pagination (actual queries may vary):

```
-- DB2 (with OFFSET), SQL Server 2008 (with OFFSET)
SELECT * FROM (
  SELECT BOOK.*,
    ROW_NUMBER() OVER (ORDER BY ID ASC) AS RN
  FROM BOOK
) AS X
WHERE RN > 2
AND RN <= 3

-- DB2 (with OFFSET), SQL Server 2008 (with OFFSET)
SELECT * FROM (
  SELECT DISTINCT BOOK.ID, BOOK.TITLE
    DENSE_RANK() OVER (ORDER BY ID ASC, TITLE ASC) AS RN
  FROM BOOK
) AS X
WHERE RN > 2
AND RN <= 3

-- Oracle 11g and less
SELECT *
FROM (
  SELECT b.*, ROWNUM RN
  FROM (
    SELECT *
    FROM BOOK
    ORDER BY ID ASC
  ) b
  WHERE ROWNUM <= 3
)
WHERE RN > 2
```

As you can see, jOOQ will take care of the incredibly painful ROW_NUMBER() OVER() (or ROWNUM for Oracle) filtering in subselects for you, you'll just have to write limit(1).offset(2) in any dialect.

Side-note: If you're interested in understanding why we chose ROWNUM for Oracle, please refer to this very interesting benchmark, comparing the different approaches of doing pagination in Oracle: http://www.inf.unideb.hu/~gabora/pagination/results.html.

SQL Server's ORDER BY, TOP and subqueries

As can be seen in the above example, writing correct SQL can be quite tricky, depending on the SQL dialect. For instance, with SQL Server, you cannot have an ORDER BY clause in a subquery, unless you also have a TOP clause. This is illustrated by the fact that jOOQ renders a TOP 100 PERCENT clause for you. The same applies to the fact that ROW_NUMBER() OVER() needs an ORDER BY windowing clause, even if you don't provide one to the jOOQ query. By default, jOOQ adds ordering by the first column of your projection.

4.3.3.12. The WITH TIES clause

The previous chapter talked about the LIMIT clause, which limits the result set to a certain number of rows. The SQL standard specifies the following syntax:

```
OFFSET m { ROW | ROWS }
FETCH { FIRST | NEXT } n { ROW | ROWS } { ONLY | WITH TIES }
```

By default, most users will use the semantics of the ONLY keyword, meaning a LIMIT 5 expression (or FETCH NEXT 5 ROWS ONLY expression) will result in at most 5 rows. The alternative clause WITH TIES will return at most 5 rows, except if the 5th row and the 6th row (and so on) are "tied" according to the ORDER BY clause, meaning that the ORDER BY clause does not deterministically produce a 5th or 6th row. For example, let's look at our book table:

```
SELECT *
FROM book
ORDER BY actor_id
FETCH NEXT 1 ROWS WITH TIES
```

```
DSL.using(configuration)
   .selectFrom(BOOK)
   .orderBy(BOOK.ACTOR_ID)
   .limit(1).withTies()
   .fetch();
```

Resulting in:

```
id   actor_id   title
---------------------
1    1          1984
2    1          Animal Farm
```

We're now getting two rows because both rows "tied" when ordering them by ACTOR_ID. The database cannot really pick the next 1 row, so they're both returned. If we omit the WITH TIES clause, then only a random one of the rows would be returned.

Not all databases support WITH TIES. Oracle 12c supports the clause as specified in the SQL standard, and SQL Server knows TOP n WITH TIES without OFFSET support.

4.3.3.13. The SEEK clause

One of the previous chapters talked about OFFSET pagination using LIMIT .. OFFSET, or OFFSET .. FETCH or some other vendor-specific variant of the same. This can lead to significant performance issues when reaching a high page number, as all unneeded records need to be skipped by the database.

A much faster and more stable way to perform pagination is the so-called ***keyset pagination method*** also called ***seek method***. jOOQ supports a synthetic seek() clause, that can be used to perform keyset pagination. Imagine we have these data:

```
|   ID | VALUE | PAGE_BOUNDARY |
|------|-------|---------------|
|  ... |   ... |           ... |
|  474 |     2 |             0 |
|  533 |     2 |             1 | <-- Before page 6
|  640 |     2 |             0 |
|  776 |     2 |             0 |
|  815 |     2 |             0 |
|  947 |     2 |             0 |
|   37 |     3 |             1 | <-- Last on page 6
|  287 |     3 |             0 |
|  450 |     3 |             0 |
|  ... |   ... |           ... |
```

Now, if we want to display page 6 to the user, instead of going to page 6 by using a record OFFSET, we could just fetch the record strictly after the last record on page 5, which yields the values (533, 2). This is how you would do it with SQL or with jOOQ:

```
SELECT id, value
FROM t
WHERE (value, id) > (2, 533)
ORDER BY value, id
LIMIT 5
```

```
DSL.using(configuration)
   .select(T.ID, T.VALUE)
   .from(T)
   .orderBy(T.VALUE, T.ID)
   .seek(2, 533)
   .limit(5)
   .fetch();
```

As you can see, the jOOQ SEEK clause is a synthetic clause that does not really exist in SQL. However, the jOOQ syntax is far more intuitive for a variety of reasons:

- It replaces OFFSET where you would expect
- It doesn't force you to mix regular predicates with *"seek"* predicates
- It is typesafe
- It emulates row value expression predicates for you, in those databases that do not support them

This query now yields:

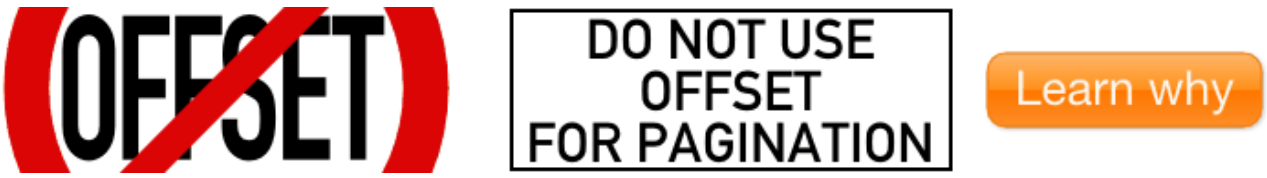
```
|  ID | VALUE |
|-----|-------|
| 640 |     2 |
| 776 |     2 |
| 815 |     2 |
| 947 |     2 |
|  37 |     3 |
```

Note that you cannot combine the SEEK clause with the OFFSET clause.

More information about this great feature can be found in the jOOQ blog:

- http://blog.jooq.org/2013/10/26/faster-sql-paging-with-jooq-using-the-seek-method/
- http://blog.jooq.org/2013/11/18/faster-sql-pagination-with-keysets-continued/

Further information about offset pagination vs. keyset pagination performance can be found on our partner page:



4.3.3.14. The FOR UPDATE clause

For inter-process synchronisation and other reasons, you may choose to use the SELECT .. FOR UPDATE clause to indicate to the database, that a set of cells or records should be locked by a given transaction for subsequent updates. With jOOQ, this can be achieved as such:

```
SELECT *
FROM BOOK
WHERE ID = 3
FOR UPDATE
```

```
create.select()
      .from(BOOK)
      .where(BOOK.ID.eq(3))
      .forUpdate()
      .fetch();
```

The above example will produce a record-lock, locking the whole record for updates. Some databases also support cell-locks using FOR UPDATE OF ..

```
SELECT *
FROM BOOK
WHERE ID = 3
FOR UPDATE OF TITLE
```

```
create.select()
      .from(BOOK)
      .where(BOOK.ID.eq(3))
      .forUpdate().of(BOOK.TITLE)
      .fetch();
```

Oracle goes a bit further and also allows to specify the actual locking behaviour. It features these additional clauses, which are all supported by jOOQ:

- FOR UPDATE NOWAIT: This is the default behaviour. If the lock cannot be acquired, the query fails immediately
- FOR UPDATE WAIT n: Try to wait for [n] seconds for the lock acquisition. The query will fail only afterwards
- FOR UPDATE SKIP LOCKED: This peculiar syntax will skip all locked records. This is particularly useful when implementing queue tables with multiple consumers

With jOOQ, you can use those Oracle extensions as such:

```
create.select().from(BOOK).where(BOOK.ID.eq(3)).forUpdate().nowait().fetch();
create.select().from(BOOK).where(BOOK.ID.eq(3)).forUpdate().wait(5).fetch();
create.select().from(BOOK).where(BOOK.ID.eq(3)).forUpdate().skipLocked().fetch();
```

FOR UPDATE in CUBRID and SQL Server

The SQL standard specifies a FOR UPDATE clause to be applicable for cursors. Most databases interpret this as being applicable for all SELECT statements. An exception to this rule are the CUBRID and SQL Server databases, that do not allow for any FOR UPDATE clause in a regular SQL SELECT statement. jOOQ emulates the FOR UPDATE behaviour, by locking record by record with JDBC. JDBC allows for specifying the flags TYPE_SCROLL_SENSITIVE, CONCUR_UPDATABLE for any statement, and then using ResultSet.updateXXX() methods to produce a cell-lock / row-lock. Here's a simplified example in JDBC:

```
try (
    PreparedStatement stmt = connection.prepareStatement(
        "SELECT * FROM author WHERE id IN (3, 4, 5)",
        ResultSet.TYPE_SCROLL_SENSITIVE,
        ResultSet.CONCUR_UPDATABLE);
    ResultSet rs = stmt.executeQuery()
) {
    while (rs.next()) {
        // UPDATE the primary key for row-locks, or any other columns for cell-locks
        rs.updateObject(1, rs.getObject(1));
        rs.updateRow();

        // Do more stuff with this record
    }
}
```

The main drawback of this approach is the fact that the database has to maintain a scrollable cursor, whose records are locked one by one. This can cause a major risk of deadlocks or race conditions if the JDBC driver can recover from the unsuccessful locking, if two Java threads execute the following statements:

```
-- thread 1
SELECT * FROM author ORDER BY id ASC;

-- thread 2
SELECT * FROM author ORDER BY id DESC;
```

So use this technique with care, possibly only ever locking single rows!

Pessimistic (shared) locking with the FOR SHARE clause

Some databases (MySQL, Postgres) also allow to issue a non-exclusive lock explicitly using a FOR SHARE clause. This is also supported by jOOQ

Optimistic locking in jOOQ

Note, that jOOQ also supports optimistic locking, if you're doing simple CRUD. This is documented in the section's manual about optimistic locking.

4.3.3.15. UNION, INTERSECTION and EXCEPT

SQL allows to perform set operations as understood in standard set theory on result sets. These operations include unions, intersections, subtractions. For two subselects to be combinable by such a set operator, each subselect must return a table expression of the same degree and type.

UNION and UNION ALL

These operators combine two results into one. While UNION removes all duplicate records resulting from this combination, UNION ALL leaves subselect results as they are. Typically, you should prefer UNION ALL over UNION, if you don't really need to remove duplicates. The following example shows how to use such a UNION operation in jOOQ.

```
SELECT * FROM BOOK WHERE ID = 3
UNION ALL
SELECT * FROM BOOK WHERE ID = 5
```

```
create.selectFrom(BOOK).where(BOOK.ID.eq(3))
      .unionAll(
create.selectFrom(BOOK).where(BOOK.ID.eq(5)))
      .fetch();
```

INTERSECT [ALL] and EXCEPT [ALL]

INTERSECT is the operation that produces only those values that are returned by both subselects. EXCEPT (or MINUS in Oracle) is the operation that returns only those values that are returned exclusively in the first subselect. Both operators will remove duplicates from their results.

Just like with UNION ALL, these operators have an optional ALL keyword that allows for keeping duplicate rows after intersection or subtraction, which is supported in jOOQ 3.7+.

jOOQ's set operators and how they're different from standard SQL

As previously mentioned in the manual's section about the ORDER BY clause, jOOQ has slightly changed the semantics of these set operators. While in SQL, a subselect may not contain any ORDER BY clause

or LIMIT clause (unless you wrap the subselect into a nested SELECT), jOOQ allows you to do so. In order to select both the youngest and the oldest author from the database, you can issue the following statement with jOOQ (rendered to the MySQL dialect):

```
  (SELECT * FROM AUTHOR
   ORDER BY DATE_OF_BIRTH ASC LIMIT 1)
UNION
  (SELECT * FROM AUTHOR
   ORDER BY DATE_OF_BIRTH DESC LIMIT 1)
ORDER BY 1
```

```
create.selectFrom(AUTHOR)
      .orderBy(AUTHOR.DATE_OF_BIRTH.asc()).limit(1)
      .union(
       selectFrom(AUTHOR)
      .orderBy(AUTHOR.DATE_OF_BIRTH.desc()).limit(1))
      .orderBy(1)
      .fetch();
```

In case your database doesn't support ordered UNION subselects, the subselects are nested in derived tables:

```
SELECT * FROM (
  SELECT * FROM AUTHOR
  ORDER BY DATE_OF_BIRTH ASC LIMIT 1
)
UNION
SELECT * FROM (
  SELECT * FROM AUTHOR
  ORDER BY DATE_OF_BIRTH DESC LIMIT 1
)
ORDER BY 1
```

Projection typesafety for degrees between 1 and 22

Two subselects that are combined by a set operator are required to be of the same degree and, in most databases, also of the same type. jOOQ 3.0's introduction of Typesafe Record[N] types helps compile-checking these constraints:

```
// Some sample SELECT statements
Select<Record2<Integer, String>>  s1 = select(BOOK.ID, BOOK.TITLE).from(BOOK);
Select<Record1<Integer>>          s2 = selectOne();
Select<Record2<Integer, Integer>> s3 = select(one(), zero());
Select<Record2<Integer, String>>  s4 = select(one(), inline("abc"));

// Let's try to combine them:
s1.union(s2); // Doesn't compile because of a degree mismatch. Expected: Record2<...>, got: Record1<...>
s1.union(s3); // Doesn't compile because of a type mismatch. Expected: <Integer, String>, got: <Integer, Integer>
s1.union(s4); // OK. The two Record[N] types match
```

4.3.3.16. Oracle-style hints

If you are closely coupling your application to an Oracle (or CUBRID) 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 AUTHOR
```

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

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

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. This can be useful in other databases too, such as MySQL, for instance:

```
SELECT SQL_CALC_FOUND_ROWS field1, field2
FROM table1
```

```
create.select(field1, field2)
      .hint("SQL_CALC_FOUND_ROWS")
      .from(table1)
      .fetch()
```

4.3.3.17. Lexical and logical SELECT clause order

SQL has a lexical and a logical order of SELECT clauses. The lexical order of SELECT clauses is inspired by the English language. As SQL statements are commands for the database, it is natural to express a statement in an imperative tense, such as "SELECT this and that!".

Logical SELECT clause order

The logical order of SELECT clauses, however, does not correspond to the syntax. In fact, the logical order is this:

- The FROM clause: First, all data sources are defined and joined
- The WHERE clause: Then, data is filtered as early as possible
- The CONNECT BY clause: Then, data is traversed iteratively or recursively, to produce new tuples
- The GROUP BY clause: Then, data is reduced to groups, possibly producing new tuples if grouping functions like ROLLUP(), CUBE(), GROUPING SETS() are used
- The HAVING clause: Then, data is filtered again
- The SELECT clause: Only now, the projection is evaluated. In case of a SELECT DISTINCT statement, data is further reduced to remove duplicates
- The UNION clause: Optionally, the above is repeated for several UNION-connected subqueries. Unless this is a UNION ALL clause, data is further reduced to remove duplicates
- The ORDER BY clause: Now, all remaining tuples are ordered
- The LIMIT clause: Then, a paginating view is created for the ordered tuples
- The FOR UPDATE clause: Finally, pessimistic locking is applied

The SQL Server documentation also explains this, with slightly different clauses:

- FROM
- ON
- JOIN
- WHERE
- GROUP BY
- WITH CUBE or WITH ROLLUP
- HAVING
- SELECT
- DISTINCT
- ORDER BY
- TOP

As can be seen, databases have to logically reorder a SQL statement in order to determine the best execution plan.

Alternative syntaxes: LINQ, SLICK

Some "higher-level" abstractions, such as C#'s LINQ or Scala's SLICK try to inverse the lexical order of SELECT clauses to what appears to be closer to the logical order. The obvious advantage of moving the SELECT clause to the end is the fact that the projection type, which is the record type returned by the SELECT statement can be re-used more easily in the target environment of the internal domain specific language.

A LINQ example:

```
// LINQ-to-SQL looks somewhat similar to SQL
// AS clause    // FROM clause
From p          In db.Products

// WHERE clause
Where p.UnitsInStock <= p.ReorderLevel AndAlso Not p.Discontinued

// SELECT clause
Select p
```

A SLICK example:

```
// "for" is the "entry-point" to the DSL
val q = for {

    // FROM clause   WHERE clause
    c <- Coffees     if c.supID === 101

// SELECT clause and projection to a tuple
} yield (c.name, c.price)
```

While this looks like a good idea at first, it only complicates translation to more advanced SQL statements while impairing readability for those users that are used to writing SQL. jOOQ is designed to look just like SQL. This is specifically true for SLICK, which not only changed the SELECT clause order, but also heavily "integrated" SQL clauses with the Scala language.

For these reasons, the jOOQ DSL API is modelled in SQL's lexical order.

4.3.4. The INSERT statement

The INSERT statement is used to insert new records into a database table. The following sections describe the various operation modes of the jOOQ INSERT statement.

4.3.4.1. INSERT .. VALUES

INSERT .. VALUES with a single row

Records can either be supplied using a VALUES() constructor, or a SELECT statement. jOOQ supports both types of INSERT statements. An example of an INSERT statement using a VALUES() constructor is given here:

```
INSERT INTO AUTHOR
       (ID, FIRST_NAME, LAST_NAME)
VALUES (100, 'Hermann', 'Hesse');
```

```
create.insertInto(AUTHOR,
        AUTHOR.ID, AUTHOR.FIRST_NAME, AUTHOR.LAST_NAME)
      .values(100, "Hermann", "Hesse")
      .execute();
```

Note that for explicit degrees up to 22, the VALUES() constructor provides additional typesafety. The following example illustrates this:

```
InsertValuesStep3<AuthorRecord, Integer, String, String> step =
  create.insertInto(AUTHOR, AUTHOR.ID, AUTHOR.FIRST_NAME, AUTHOR.LAST_NAME);
    step.values("A", "B", "C");
         // ^^^ Doesn't compile, the expected type is Integer
```

INSERT .. VALUES with multiple rows

The SQL standard specifies that multiple rows can be supplied to the VALUES() constructor in an INSERT statement. Here's an example of a multi-record INSERT

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

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

jOOQ tries to stay close to actual SQL. In detail, however, Java's expressiveness is limited. That's why the values() clause is repeated for every record in multi-record inserts.

Some RDBMS do not support inserting several records in a single statement. In those cases, jOOQ emulates multi-record INSERTs using the following SQL:

```
INSERT INTO AUTHOR
    (ID, FIRST_NAME, LAST_NAME)
SELECT 100, 'Hermann', 'Hesse' FROM DUAL UNION ALL
SELECT 101, 'Alfred', 'Döblin' FROM DUAL;
```

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

4.3.4.2. INSERT .. DEFAULT VALUES

A lesser-known syntactic feature of SQL is the INSERT .. DEFAULT VALUES statement, where a single record is inserted, containing only DEFAULT values for every row. It is written as such:

```
INSERT INTO AUTHOR
DEFAULT VALUES;
```

```
create.insertInto(AUTHOR)
      .defaultValues()
      .execute();
```

This can make a lot of sense in situations where you want to "reserve" a row in the database for an subsequent UPDATE statement within the same transaction. Or if you just want to send an event containing trigger-generated default values, such as IDs or timestamps.

The DEFAULT VALUES clause is not supported in all databases, but jOOQ can emulate it using the equivalent statement:

```
INSERT INTO AUTHOR
    (ID, FIRST_NAME, LAST_NAME, ...)
VALUES (
 DEFAULT,
 DEFAULT,
 DEFAULT, ...);
```

```
create.insertInto(
        AUTHOR, AUTHOR.ID, AUTHOR.FIRST_NAME,
 AUTHOR.LAST_NAME, ...)
      .values(
       defaultValue(AUTHOR.ID),
       defaultValue(AUTHOR.FIRST_NAME),
       defaultValue(AUTHOR.LAST_NAME), ...)
      .execute();
```

The DEFAULT keyword (or DSL#defaultValue() method) can also be used for individual columns only, although that will have the same effect as leaving the column away entirely.

4.3.4.3. INSERT .. SET

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

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

As you can see, this syntax is a bit more verbose, but also more readable, as every field can be matched with its value. Internally, the two syntaxes are strictly equivalent.

4.3.4.4. INSERT .. SELECT

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

```
create.insertInto(AUTHOR_ARCHIVE)
      .select(selectFrom(AUTHOR).where(AUTHOR.DECEASED.isTrue()))
      .execute();
```

4.3.4.5. INSERT .. ON DUPLICATE KEY

The synthetic 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 emulated in other RDBMS, where this is possible (i.e. if they support the SQL standard MERGE statement). 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(AUTHOR, AUTHOR.ID, AUTHOR.LAST_NAME)
      .values(3, "Koontz")
      .onDuplicateKeyUpdate()
      .set(AUTHOR.LAST_NAME, "Koontz")
      .execute();
```

The synthetic ON DUPLICATE KEY IGNORE clause

The MySQL database also supports an INSERT IGNORE INTO clause. This is supported by jOOQ using the more convenient SQL syntax variant of ON DUPLICATE KEY IGNORE:

```
// Add a new author called "Koontz" with ID 3.
// If that ID is already present, ignore the INSERT statement
create.insertInto(AUTHOR, AUTHOR.ID, AUTHOR.LAST_NAME)
      .values(3, "Koontz")
      .onDuplicateKeyIgnore()
      .execute();
```

If the underlying database doesn't have any way to "ignore" failing INSERT statements, (e.g. MySQL via INSERT IGNORE), jOOQ can emulate the statement using a MERGE statement, or using INSERT .. SELECT WHERE NOT EXISTS:

Emulating IGNORE with MERGE

The above jOOQ statement can be emulated with the following, equivalent SQL statement:

```
MERGE INTO AUTHOR
USING (SELECT 1 FROM DUAL)
ON (AUTHOR.ID = 3)
WHEN NOT MATCHED THEN INSERT (ID, LAST_NAME)
  VALUES (3, 'Koontz')
```

Emulating IGNORE with INSERT .. SELECT WHERE NOT EXISTS

The above jOOQ statement can be emulated with the following, equivalent SQL statement:

```
INSERT INTO AUTHOR (ID, LAST_NAME)
SELECT 3, 'Koontz'
WHERE NOT EXISTS (
  SELECT 1
  FROM AUTHOR
  WHERE AUTHOR.ID = 3
)
```

4.3.4.6. INSERT .. RETURNING

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

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

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

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

Some databases have poor support for returning generated keys after INSERTs. In those cases, jOOQ might need to issue another SELECT statement in order to fetch an @@identity value. Be aware, that this can lead to race-conditions in those databases that cannot properly return generated ID values. For more information, please consider the jOOQ Javadoc for the returning() clause.

4.3.5. The UPDATE statement

The UPDATE statement is used to modify one or several pre-existing records in a database table. UPDATE statements are only possible on single tables. Support for multi-table updates will be implemented in the near future. An example update query is given here:

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

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

Most databases allow for using scalar subselects in UPDATE statements in one way or another. jOOQ models this through a set(Field<T>, Select<? extends Record1<T>>) method in the UPDATE DSL API:

```
UPDATE AUTHOR
   SET FIRST_NAME = (
         SELECT FIRST_NAME
         FROM PERSON
         WHERE PERSON.ID = AUTHOR.ID
       ),
 WHERE ID = 3;
```

```
create.update(AUTHOR)
      .set(AUTHOR.FIRST_NAME,
         select(PERSON.FIRST_NAME)
        .from(PERSON)
        .where(PERSON.ID.eq(AUTHOR.ID))
      )
      .where(AUTHOR.ID.eq(3))
      .execute();
```

Using row value expressions in an UPDATE statement

jOOQ supports formal row value expressions in various contexts, among which the UPDATE statement. Only one row value expression can be updated at a time. Here's an example:

```
UPDATE AUTHOR
   SET (FIRST_NAME, LAST_NAME) =
       ('Hermann',  'Hesse')
 WHERE ID = 3;
```

```
create.update(AUTHOR)
      .set(row(AUTHOR.FIRST_NAME, AUTHOR.LAST_NAME),
           row("Herman",          "Hesse"))
      .where(AUTHOR.ID.eq(3))
      .execute();
```

This can be particularly useful when using subselects:

```
UPDATE AUTHOR
   SET (FIRST_NAME, LAST_NAME) = (
         SELECT PERSON.FIRST_NAME, PERSON.LAST_NAME
         FROM PERSON
         WHERE PERSON.ID = AUTHOR.ID
       )
 WHERE ID = 3;
```

```
create.update(AUTHOR)
      .set(row(AUTHOR.FIRST_NAME, AUTHOR.LAST_NAME),
           select(PERSON.FIRST_NAME, PERSON.LAST_NAME)
          .from(PERSON)
          .where(PERSON.ID.eq(AUTHOR.ID))
      )
      .where(AUTHOR.ID.eq(3))
      .execute();
```

The above row value expressions usages are completely typesafe.

UPDATE .. FROM

Some databases, including PostgreSQL and SQL Server, support joining additional tables to an UPDATE statement using a vendor-specific FROM clause. This is supported as well by jOOQ:

```
UPDATE BOOK_ARCHIVE
SET
  BOOK_ARCHIVE.TITLE = BOOK.TITLE
FROM BOOK
WHERE BOOK_ARCHIVE.ID = BOOK.ID
```

```
create.update(BOOK_ARCHIVE)
      .set(BOOK_ARCHIVE.TITLE, BOOK.TITLE)
      .from(BOOK)
      .where(BOOK_ARCHIVE.ID.eq(BOOK.ID))
      .execute();
```

In many cases, such a joined update statement can be emulated using a correlated subquery, or using updatable views.

UPDATE .. RETURNING

The Firebird and Postgres databases support a RETURNING clause on their UPDATE statements, similar as the RETURNING clause in INSERT statements. This is useful to fetch trigger-generated values in one go. An example is given here:

```
-- Fetch a trigger-generated value
UPDATE BOOK
SET TITLE = 'Animal Farm'
WHERE ID = 5
RETURNING TITLE
```

```
String title = create.update(BOOK)
                     .set(BOOK.TITLE, "Animal Farm")
                     .where(BOOK.ID.eq(5))
                     .returning(BOOK.TITLE)
                     .fetchOne().getValue(BOOK.TITLE);
```

The UPDATE .. RETURNING clause is emulated for DB2 using the SQL standard SELECT .. FROM FINAL TABLE(UPDATE ..) construct, and in Oracle, using the PL/SQL UPDATE .. RETURNING statement.

4.3.6. The DELETE statement

The DELETE statement removes records from a database table. DELETE statements are only possible on single tables. Support for multi-table deletes will be implemented in the near future. An example delete query is given here:

```
DELETE AUTHOR
 WHERE ID = 100;
```

```
create.delete(AUTHOR)
      .where(AUTHOR.ID.eq(100))
      .execute();
```

4.3.7. The 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)

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 3.11.11, 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 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(AUTHOR)
      .using(create.selectOne())
      .on(AUTHOR.LAST_NAME.eq("Hitchcock"))
      .whenMatchedThenUpdate()
      .set(AUTHOR.FIRST_NAME, "John")
      .whenNotMatchedThenInsert(AUTHOR.LAST_NAME)
      .values("Hitchcock")
      .execute();
```

MERGE Statement (H2-specific syntax)

The H2 database ships with a somewhat less powerful but a little more intuitive syntax for its own version of the MERGE statement. An example more or less equivalent to the previous one can be seen here:

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

MERGE INTO AUTHOR (FIRST_NAME, LAST_NAME)
KEY (LAST_NAME)
VALUES ('John', 'Hitchcock')
```

```
create.mergeInto(AUTHOR,
                 AUTHOR.FIRST_NAME,
                 AUTHOR.LAST_NAME)
      .key(AUTHOR.LAST_NAME)
      .values("John", "Hitchcock")
      .execute();
```

This syntax can be fully emulated by jOOQ for all other databases that support the SQL standard MERGE statement. For more information about the H2 MERGE syntax, see the documentation here: http://www.h2database.com/html/grammar.html#merge

Typesafety of VALUES() for degrees up to 22

Much like the INSERT statement, the MERGE statement's VALUES() clause provides typesafety for degrees up to 22, in both the standard syntax variant as well as the H2 variant.

4.4. SQL Statements (DDL)

jOOQ's DDL support is currently still very limited. In the long run, jOOQ will support the most important statement types for frequent informal database migrations, though. Note that jOOQ will not aim to replace existing database migration frameworks. At Data Geekery, we usually recommend using Flyway for migrations. See also the tutorial about using jOOQ with Flyway for more information.

4.4.1. The SET statement

Most databases support a variety of SET statements to set session specific environment variables. jOOQ supports two of these set statements that are particularly useful when running DDL scripts:

```
SET CATALOG catalogname;
SET SCHEMA schemaname;
```

```
create.setCatalog("catalogname").execute();
create.setSchema("schemaname").execute();
```

Depending on whether your database supports catalogs and schemas, the above SET statements may be supported in your database.

In MariaDB, MySQL, SQL Server, the SET CATALOG statement is emulated using:

```
USE catalogname;
```

In Oracle, the SET SCHEMA statement is emulated using:

```
ALTER SESSION SET CURRENT_SCHEMA = schemaname;
```

4.4.2. The ALTER statement

jOOQ currently supports the following ALTER statements (SQL examples in PostgreSQL syntax):

Indexes

```
// Renaming the index
create.alterIndex("old_index").renameTo("new_index").execute();

// Renaming the index only if it exists (not all databases support this)
create.alterIndexIfExists("old_index").renameTo("new_index").execute();
```

Schemas

```
// Renaming the schema
create.alterSchema("old_schema").renameTo("new_schema").execute();

// Renaming the schema only if it exists (not all databases support this)
create.alterSchemaIfExists("old_schema").renameTo("new_schema").execute();
```

Sequences

```
// Renaming the sequence
create.alterSequence("old_sequence").renameTo("new_sequence").execute();

// Renaming the sequence only if it exists (not all databases support this)
create.alterSequenceIfExists("old_sequence").renameTo("new_sequence").execute();

// Restarting the sequence
create.alterSequence(S_AUTHOR_ID).restart().execute();
create.alterSequence(S_AUTHOR_ID).restartWith(n).execute();
```

Tables

These statements alter the table itself:

```
// Renaming the table
create.alterTable("old_table").renameTo("new_table").execute();

// Renaming the table only if it exists (not all databases support this)
create.alterTableIfExists("old_table").renameTo("new_table").execute();
```

These statements alter / add / drop columns and their types:

```
// Adding columns
create.alterTable(AUTHOR).add(AUTHOR.TITLE, VARCHAR.length(5)).execute();
create.alterTable(AUTHOR).add(AUTHOR.TITLE, VARCHAR.length(5).nullable(false)).execute();

// Altering columns
create.alterTable(AUTHOR).alter(TITLE).defaultValue("no title").execute();
create.alterTable(AUTHOR).alter(TITLE).set(VARCHAR.length(5)).execute();
create.alterTable(AUTHOR).alter(TITLE).set(VARCHAR.length(5).nullable(false)).execute();
create.alterTable(AUTHOR).renameColumn("old_column").to("new_column").execute();

// Dropping columns
create.alterTable(AUTHOR).drop(TITLE).execute();
```

These statements alter / add / drop constraints:

```
// Adding constraints
create.alterTable(BOOK).add(constraint("PK_BOOK").primaryKey(BOOK.ID)).execute();
create.alterTable(BOOK).add(constraint("UK_TITLE").unique(BOOK.TITLE)).execute();
create.alterTable(BOOK).add(
   constraint("FK_AUTHOR_ID")
  .foreignKey(BOOK.AUTHOR_ID)
  .references(AUTHOR, AUTHOR.ID)).execute();
create.alterTable(BOOK).add(
   constraint("CHECK_PUBLISHED_IN")
  .check(BOOK.PUBLISHED_IN.between(1900).and(2000))).execute();

// Altering constraints
create.alterTable(BOOK).renameConstraint("old_constraint").to("new_constraint").execute();

// Dropping constraints
create.alterTable(AUTHOR).dropConstraint("UK_TITLE").execute();
```

Views

```
// Renaming the view
create.alterView("old_view").renameTo("new_view").execute();

// Renaming the view only if it exists (not all databases support this)
create.alterViewIfExists("old_view").renameTo("new_view").execute();
```

4.4.3. The CREATE statement

jOOQ currently supports the following CREATE statements (SQL examples in PostgreSQL syntax):

Indexes

```
// Create a non-unique index
create.createIndex("I_AUTHOR_LAST_NAME").on(AUTHOR, AUTHOR.LAST_NAME).execute();

// Create an index only if it doesn't exist (not all databases support this)
create.createIndexIfNotExists("I_AUTHOR_LAST_NAME").on(AUTHOR, AUTHOR.LAST_NAME).execute();

// Create a partial index (not all databases support this)
create.createIndex("I_AUTHOR_LAST_NAME").on(AUTHOR, AUTHOR.LAST_NAME).where(AUTHOR.LAST_NAME.like("A%")).execute();

// Create a unique index
create.createUniqueIndex("I_AUTHOR_LAST_NAME").on(AUTHOR, AUTHOR.LAST_NAME).execute();
```

Schemas

```
// Create a schema
create.createSchema("new_schema").execute();

// Create a schema only if it doesn't exists (not all databases support this)
create.createSchemaIfNotExists("new_schema").execute();
```

Sequences

```
// Create a sequence
create.createSequence(S_AUTHOR_ID).execute();

// Create a sequence only if it doesn't exists (not all databases support this)
create.createSequence(S_AUTHOR_ID).execute();
```

Tables

```
// Creating a table with columns and inline constraints
create.createTable(AUTHOR)
      .column(AUTHOR.ID, SQLDataType.INTEGER)
      .column(AUTHOR.FIRST_NAME, SQLDataType.VARCHAR.length(50).nullable(false))
      .column(AUTHOR.LAST_NAME, SQLDataType.VARCHAR.length(50))
      .constraints(
          constraint("PK_AUTHOR").primaryKey(AUTHOR.ID),
          constraint("UK_AUTHOR").unique(FIRST_NAME, LAST_NAME))
      .execute();

// Creating a table from a SELECT statement
create.createTable("TOP_AUTHORS").as(
       select(
         AUTHOR.ID,
         AUTHOR.FIRST_NAME,
         AUTHOR.LAST_NAME)
      .from(AUTHOR)
      .where(val(50).lt(
         selectCount().from(BOOK)
        .where(BOOK.AUTHOR_ID.eq(AUTHOR.ID))
      ))).execute();

// Create a table only if it doesn't exists (not all databases support this)
create.createTableIfNotExists("TOP_AUTHORS")
      ...
```

Views

```
// Create a view
create.createView("V_TOP_AUTHORS").as(
       select(
         AUTHOR.ID,
         AUTHOR.FIRST_NAME,
         AUTHOR.LAST_NAME)
      .from(AUTHOR)
      .where(val(50).lt(
         selectCount().from(BOOK)
        .where(BOOK.AUTHOR_ID.eq(AUTHOR.ID))
      ))).execute();

// Create a view only if it doesn't exists (not all databases support this)
create.createTableIfNotExists("TOP_AUTHORS")
      ...
```

4.4.4. The DROP statement

jOOQ currently supports the following DROP statements (SQL examples in PostgreSQL syntax):

Indexes

```
// Drop an index
create.dropIndex("I_AUTHOR_LAST_NAME").execute();

// Drop an index only if it exists (not all databases support this)
create.dropIndexIfExists("I_AUTHOR_LAST_NAME").execute();
```

Schemas

```
// Drop a schema
create.dropSchema("schema").execute();

// Drop a schema only if it exists (not all databases support this)
create.dropSchemaIfExists("schema").execute();
```

Sequences

```
// Drop a sequence
create.dropSequence(S_AUTHOR_ID).execute();

// Drop a sequence only if it exists (not all databases support this)
create.dropSequenceIfExists(S_AUTHOR_ID).execute();
```

Tables

```
// Drop a table
create.dropTable(AUTHOR).execute();

// Drop a table only if it exists (not all databases support this)
create.dropTableIfExists(AUTHOR).execute();
```

Views

```
// Drop a view
create.dropView(V_AUTHOR).execute();

[// Drop a view only if it exists (not all databases support this)
create.dropViewIfExists(V_AUTHOR).execute();
```

4.4.5. The TRUNCATE statement

Even if the TRUNCATE statement mainly modifies data, it is generally considered to be a DDL statement. It is popular in many databases when you want to bypass constraints for table truncation. Databases may behave differently, when a truncated table is referenced by other tables. For instance, they may fail if records from a truncated table are referenced, even with ON DELETE CASCADE clauses in place. Please, consider your database manual to learn more about its TRUNCATE implementation.

The TRUNCATE syntax is trivial:

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

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

4.4.6. Generating DDL from objects

When using jOOQ's code generator, a whole set of meta data is generated with the generated artefacts, such as schemas, tables, columns, data types, constraints, default values, etc.

This meta data can be used to generate DDL CREATE statements in any SQL dialect, in order to partially restore the original schema again on a new database instance. This is particularly useful, for instance, when working with an Oracle production database, and an H2 in-memory test database. The following code produces the DDL for a schema:

```
// SCHEMA is the generated schema that contains a reference to all generated tables
Queries ddl =
DSL.using(configuration)
   .ddl(SCHEMA);

for (Query query : ddl.queries()) {
    System.out.println(query);
}
```

When executing the above, you should see something like the following:

```
create table "PUBLIC"."T_AUTHOR"(
  "ID" int not null,
  "FIRST_NAME" varchar(50) null,
  "LAST_NAME" varchar(50) not null,
  ...
  constraint "PK_T_AUTHOR"
    primary key ("ID")
)
create table "PUBLIC"."T_BOOK"(
  "ID" int not null,
  "AUTHOR_ID" int not null,
  "TITLE" varchar(400) not null,
  ...
  constraint "PK_T_BOOK"
    primary key ("ID")
)
...
alter table "PUBLIC"."T_BOOK"
  add constraint "FK_T_BOOK_AUTHOR_ID"
    foreign key ("AUTHOR_ID")
    references "T_AUTHOR" ("ID")
```

Do note that these features only restore ***parts*** of the original schema. For instance, vendor-specific storage clauses that are not available to jOOQ's generated meta data cannot be reproduced this way.

4.5. Catalog and schema expressions

Most databases know some sort of namespace to group objects like tables, stored procedures, sequences and others into a common catalog or schema. jOOQ uses the types org.jooq.Catalog and org.jooq.Schema to model these groupings, following SQL standard naming.

The catalog

A catalog is a collection of schemas. In many databases, the catalog corresponds to the database, or the database instance. Most often, catalogs are completely independent and their tables cannot be joined or combined in any way in a single query. The exception here is SQL Server, which allows for fully referencing tables from multiple catalogs:

```
SELECT *
FROM [Catalog1].[Schema1].[Table1] AS [t1]
JOIN [Catalog2].[Schema2].[Table2] AS [t2] ON [t1].[ID] = [t2].[ID]
```

By default, the Settings.renderCatalog flag is turned on. In case a database supports querying multiple catalogs, jOOQ will generate fully qualified object names, including catalog name. For more information about this setting, see the manual's section about settings

jOOQ's code generator generates subpackages for each catalog.

The schema

A schema is a collection of objects, such as tables. Most databases support some sort of schema (except for some embedded databases like Access, Firebird, SQLite). In most databases, the schema is an independent structural entity. In Oracle, the schema and the user / owner is mostly treated as the same thing. An example of a query that uses fully qualified tables including schema names is:

```
SELECT *
FROM "Schema1"."Table1" AS "t1"
JOIN "Schema2"."Table2" AS "t2" ON "t1"."ID" = "t2"."ID"
```

By default, the Settings.renderSettings flag is turned on. jOOQ will thus generate fully qualified object names, including the setting name. For more information about this setting, see the manual's section about settings

4.6. Table expressions

The following sections explain the various types of table expressions supported by jOOQ

4.6.1. Generated Tables

Most of the times, when thinking about a table expression you're probably thinking about an actual table in your database schema. If you're using jOOQ's code generator, you will have all tables from your database schema available to you as type safe Java objects. You can then use these tables in SQL FROM clauses, JOIN clauses or in other SQL statements, just like any other table expression. An example is given here:

```
SELECT *
FROM AUTHOR -- Table expression AUTHOR
JOIN BOOK   -- Table expression BOOK
ON (AUTHOR.ID = BOOK.AUTHOR_ID)
```

```
create.select()
      .from(AUTHOR) // Table expression AUTHOR
      .join(BOOK)   // Table expression BOOK
      .on(AUTHOR.ID.eq(BOOK.AUTHOR_ID))
      .fetch();
```

The above example shows how AUTHOR and BOOK tables are joined in a SELECT statement. It also shows how you can access table columns by dereferencing the relevant Java attributes of their tables.

See the manual's section about generated tables for more information about what is really generated by the code generator

4.6.2. Aliased Tables

The strength of jOOQ's code generator becomes more obvious when you perform table aliasing and dereference fields from generated aliased tables. This can best be shown by example:

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



SELECT *
  FROM author a
  JOIN book b ON a.id = b.author_id
 WHERE a.year_of_birth > 1920
   AND a.first_name = 'Paulo'
 ORDER BY b.title
```

```
// Declare your aliases before using them in SQL:
Author a = AUTHOR.as("a");
Book b = BOOK.as("b");

// Use aliased tables in your statement
create.select()
      .from(a)
      .join(b).on(a.ID.eq(b.AUTHOR_ID))
      .where(a.YEAR_OF_BIRTH.gt(1920)
      .and(a.FIRST_NAME.eq("Paulo")))
      .orderBy(b.TITLE)
      .fetch();
```

As you can see in the above example, calling as() on generated tables returns an object of the same type as the table. This means that the resulting object can be used to dereference fields from the aliased table. This is quite powerful in terms of having your Java compiler check the syntax of your SQL statements. If you remove a column from a table, dereferencing that column from that table alias will cause compilation errors.

Dereferencing columns from other table expressions

Only few table expressions provide the SQL syntax typesafety as shown above, where generated tables are used. Most tables, however, expose their fields through field() methods:

```
// "Type-unsafe" aliased table:
Table<?> a = AUTHOR.as("a");

// Get fields from a:
Field<?> id = a.field("ID");
Field<?> firstName = a.field("FIRST_NAME");
```

Derived column lists

The SQL standard specifies how a table can be renamed / aliased in one go along with its columns. It references the term "derived column list" for the following syntax (as supported by Postgres, for instance):

```
SELECT t.a, t.b
FROM (
  SELECT 1, 2
) t(a, b)
```

This feature is useful in various use-cases where column names are not known in advance (but the table's degree is!). An example for this are unnested tables, or the VALUES() table constructor:

```
-- Unnested tables
SELECT t.a, t.b
FROM unnest(my_table_function()) t(a, b)

-- VALUES() constructor
SELECT t.a, t.b
FROM VALUES(1, 2),(3, 4) t(a, b)
```

Only few databases really support such a syntax, but fortunately, jOOQ can emulate it easily using UNION ALL and an empty dummy record specifying the new column names. The two statements are equivalent:

```
-- Using derived column lists
SELECT t.a, t.b
FROM (
  SELECT 1, 2
) t(a, b)

-- Using UNION ALL and a dummy record
SELECT t.a, t.b
FROM (
  SELECT null a, null b FROM DUAL WHERE 1 = 0
  UNION ALL
  SELECT 1, 2 FROM DUAL
) t
```

In jOOQ, you would simply specify a varargs list of column aliases as such:

```
// Unnested tables
create.select().from(unnest(myTableFunction()).as("t", "a", "b")).fetch();

// VALUES() constructor
create.select().from(values(
  row(1, 2),
  row(3, 4)
).as("t", "a", "b"))
.fetch();
```

Unnamed derived tables

The org.jooq.Table type can reference a derived table:

```
-- Derived table
(SELECT 1 AS a)
```

```
// Derived table
table(select(inline(1).as("a")));
```

Most databases do not support unnamed derived tables, they require an explicit alias. If you do not provide jOOQ with such an explicit alias, an alias will be generated based on the derived table's content, to make sure the generated SQL will be syntactically correct. The generated alias is not specified and should not be referenced explicitly.

4.6.3. Joined tables

The JOIN operators that can be used in SQL SELECT statements are the most powerful and best supported means of creating new table expressions in SQL. Informally, the following can be said:

```
A(colA1, ..., colAn) "join" B(colB1, ..., colBm) "produces" C(colA1, ..., colAn, colB1, ..., colBm)
```

SQL and relational algebra distinguish between at least the following JOIN types (upper-case: SQL, lower-case: relational algebra):

- CROSS JOIN or cartesian product: The basic JOIN in SQL, producing a relational cross product, combining every record of table A with every record of table B. Note that cartesian products can also be produced by listing comma-separated table expressions in the FROM clause of a SELECT statement
- NATURAL JOIN: The basic JOIN in relational algebra, yet a rarely used JOIN in databases with everyday degree of normalisation. This JOIN type unconditionally equi-joins two tables by all columns with the same name (requiring foreign keys and primary keys to share the same name). Note that the JOIN columns will only figure once in the resulting table expression.
- INNER JOIN or equi-join: This JOIN operation performs a cartesian product (CROSS JOIN) with a filtering predicate being applied to the resulting table expression. Most often, a equal comparison predicate comparing foreign keys and primary keys will be applied as a filter, but any other predicate will work, too.
- OUTER JOIN: This JOIN operation performs a cartesian product (CROSS JOIN) with a filtering predicate being applied to the resulting table expression. Most often, a equal comparison predicate comparing foreign keys and primary keys will be applied as a filter, but any other predicate will work, too. Unlike the INNER JOIN, an OUTER JOIN will add "empty records" to the left (table A) or right (table B) or both tables, in case the conditional expression fails to produce a .
- semi-join: In SQL, this JOIN operation can only be expressed implicitly using IN predicates or EXISTS predicates. The table expression resulting from a semi-join will only contain the left-hand side table A
- anti-join: In SQL, this JOIN operation can only be expressed implicitly using NOT IN predicates or NOT EXISTS predicates. The table expression resulting from a semi-join will only contain the left-hand side table A
- division: This JOIN operation is hard to express at all, in SQL. See the manual's chapter about relational division for details on how jOOQ emulates this operation.

jOOQ supports all of these JOIN types (including semi-join and anti-join) directly on any table expression:

```
// jOOQ's relational division convenience syntax
DivideByOnStep divideBy(Table<?> table)

// INNER JOIN
TableOnStep join(TableLike<?>)
TableOnStep innerJoin(TableLike<?>)

// OUTER JOIN (supporting Oracle's partitioned OUTER JOIN)
TablePartitionByStep leftJoin(TableLike<?>)
TablePartitionByStep leftOuterJoin(TableLike<?>)

TablePartitionByStep rightJoin(TableLike<?>)
TablePartitionByStep rightOuterJoin(TableLike<?>)

// FULL OUTER JOIN
TableOnStep fullOuterJoin(TableLike<?>)

// SEMI JOIN
TableOnStep<R> leftSemiJoin(TableLike<?>);

// ANTI JOIN
TableOnStep<R> leftAntiJoin(TableLike<?>);

// CROSS JOIN
Table<Record> crossJoin(TableLike<?>)

// NATURAL JOIN
Table<Record> naturalJoin(TableLike<?>)
Table<Record> naturalLeftOuterJoin(TableLike<?>)
Table<Record> naturalRightOuterJoin(TableLike<?>)
```

Most of the above JOIN types are overloaded also to accommodate plain SQL use-cases for convenience:

```
// Standard overload accepting a formal jOOQ table reference
TableOnStep join(TableLike<?>)

// Overloaded versions taking SQL template strings with bind variables, or other forms of
// "plain SQL" QueryParts:
TableOnStep join(String)
TableOnStep join(String, Object...)
TableOnStep join(String, QueryPart...)
TableOnStep join(SQL)
TableOnStep join(Name)
```

Note that most of jOOQ's JOIN operations give way to a similar DSL API hierarchy as previously seen in the manual's section about the JOIN clause

4.6.4. The VALUES() table constructor

Some databases allow for expressing in-memory temporary tables using a VALUES() constructor. This constructor usually works the same way as the VALUES() clause known from the INSERT statement or from the MERGE statement. With jOOQ, you can also use the VALUES() table constructor, to create tables that can be used in a SELECT statement's FROM clause:

```
SELECT a, b
FROM VALUES(1, 'a'),
           (2, 'b') t(a, b)
```

```
create.select()
      .from(values(row(1, "a"),
                   row(2, "b")).as("t", "a", "b"))
      .fetch();
```

Note, that it is usually quite useful to provide column aliases ("derived column lists") along with the table alias for the VALUES() constructor.

The above statement is emulated by jOOQ for those databases that do not support the VALUES() constructor, natively (actual emulations may vary):

```
-- If derived column expressions are supported:
SELECT a, b
FROM (
  SELECT 1, 'a' FROM DUAL UNION ALL
  SELECT 2, 'b' FROM DUAL
) t(a, b)

-- If derived column expressions are not supported:
SELECT a, b
FROM (

  -- An empty dummy record is added to provide column names for the emulated derived column expression
  SELECT NULL a, NULL b FROM DUAL WHERE 1 = 0 UNION ALL

  -- Then, the actual VALUES() constructor is emulated
  SELECT 1,      'a'    FROM DUAL             UNION ALL
  SELECT 2,      'b'    FROM DUAL
) t
```

4.6.5. Nested SELECTs

A SELECT statement can appear almost anywhere a table expression can. Such a "nested SELECT" is often called a "derived table". Apart from many convenience methods accepting org.jooq.Select objects directly, a SELECT statement can always be transformed into a org.jooq.Table object using the asTable() method.

Example: Scalar subquery

```
SELECT *
  FROM BOOK
 WHERE BOOK.AUTHOR_ID = (
   SELECT ID
          FROM AUTHOR
         WHERE LAST_NAME = 'Orwell')
```

```
create.select()
      .from(BOOK)
      .where(BOOK.AUTHOR_ID.eq(create
             .select(AUTHOR.ID)
             .from(AUTHOR)
             .where(AUTHOR.LAST_NAME.eq("Orwell"))))
      .fetch();
```

Example: Derived table

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

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

create.select(nested.fields())
      .from(nested)
      .orderBy(nested.field("books"))
      .fetch();
```

Example: Correlated subquery

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

```
// The type of books cannot be inferred from the Select<?>
Field<Object> books =
    create.selectCount()
          .from(BOOK)
          .where(BOOK.AUTHOR_ID.eq(AUTHOR.ID))
          .asField("books");

create.select(AUTHOR.ID, books)
      .from(AUTHOR)
      .orderBy(books, AUTHOR.ID))
      .fetch();
```

4.6.6. The Oracle 11g PIVOT clause

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 slightly different PIVOT clause will be added later. Also, jOOQ may emulate PIVOT for other dialects in the future.

4.6.7. jOOQ's relational division syntax

There is one operation in relational algebra that is not given a lot of attention, because it is rarely used in real-world applications. It is the relational division, the opposite operation of the cross product (or, relational multiplication). The following is an approximate definition of a relational division:

```
Assume the following cross join / cartesian product
C = A × B

Then it can be said that
A = C ÷ B
B = C ÷ A
```

With jOOQ, you can simplify using relational divisions by using the following syntax:

```
C.divideBy(B).on(C.ID.eq(B.C_ID)).returning(C.TEXT)
```

The above roughly translates to

```
SELECT DISTINCT C.TEXT FROM C "c1"
WHERE NOT EXISTS (
  SELECT 1 FROM B
  WHERE NOT EXISTS (
    SELECT 1 FROM C "c2"
    WHERE "c2".TEXT = "c1".TEXT
    AND "c2".ID = B.C_ID
  )
)
```

Or in plain text: Find those TEXT values in C whose ID's correspond to all ID's in B. Note that from the above SQL statement, it is immediately clear that proper indexing is of the essence. Be sure to have indexes on all columns referenced from the on(...) and returning(...) clauses.

For more information about relational division and some nice, real-life examples, see

- http://en.wikipedia.org/wiki/Relational_algebra#Division
- http://www.simple-talk.com/sql/t-sql-programming/divided-we-stand-the-sql-of-relational-division/

4.6.8. Array and cursor unnesting

The SQL standard specifies how SQL databases should implement ARRAY and TABLE types, as well as CURSOR types. Put simply, a CURSOR is a pointer to any materialised table expression. Depending on the cursor's features, this table expression can be scrolled through in both directions, records can be locked, updated, removed, inserted, etc. Often, CURSOR types contain s, whereas ARRAY and TABLE types contain simple scalar values, although that is not a requirement

ARRAY types in SQL are similar to Java's array types. They contain a "component type" or "element type" and a "dimension". This sort of ARRAY type is implemented in H2, HSQLDB and Postgres and supported by jOOQ as such. Oracle uses strongly-typed arrays, which means that an ARRAY type (VARRAY or TABLE type) has a name and possibly a maximum capacity associated with it.

Unnesting array and cursor types

The real power of these types become more obvious when you fetch them from stored procedures to unnest them as table expressions and use them in your FROM clause. An example is given here, where Oracle's DBMS_XPLAN package is used to fetch a cursor containing data about the most recent execution plan:

```
SELECT *
FROM TABLE(DBMS_XPLAN.DISPLAY_CURSOR(null, null, 'ALLSTATS'));
```

```
create.select()
      .from(table(DbmsXplan.displayCursor(null, null,
 "ALLSTATS"))
      .fetch();
```

Note, in order to access the DbmsXplan package, you can use the code generator to generate Oracle's SYS schema.

4.6.9. Table-valued functions

Some databases support functions that can produce tables for use in arbitrary SELECT statements. jOOQ supports these functions out-of-the-box for such databases. For instance, in SQL Server, the following function produces a table of (ID, TITLE) columns containing either all the books or just one book by ID:

```
CREATE FUNCTION f_books (@id INTEGER)
RETURNS @out_table TABLE (
    id INTEGER,
    title VARCHAR(400)
)
AS
BEGIN
    INSERT @out_table
    SELECT id, title
    FROM book
    WHERE @id IS NULL OR id = @id
    ORDER BY id
    RETURN
END
```

The jOOQ code generator will now produce a generated table from the above, which can be used as a SQL function:

```
// Fetching all books records
Result<FBooksRecord> r1 = create.selectFrom(fBooks(null)).fetch();

// Lateral joining the table-valued function to another table using CROSS APPLY:
create.select(BOOK.ID, F_BOOKS.TITLE)
      .from(BOOK.crossApply(fBooks(BOOK.ID)))
      .fetch();
```

4.6.10. The DUAL table

The SQL standard specifies that the FROM clause is optional in a SELECT statement. However, according to the standard, you may then no longer use some other clauses, such as the WHERE clause. In the real world, there exist three types of databases:

- The ones that always require a FROM clause (as required by the SQL standard)
- The ones that never require a FROM clause (and still allow a WHERE clause)
- The ones that require a FROM clause only with a WHERE clause, GROUP BY clause, or HAVING clause

With jOOQ, you don't have to worry about the above distinction of SQL dialects. jOOQ never requires a FROM clause, but renders the necessary "DUAL" table, if needed. The following program shows how jOOQ renders "DUAL" tables

```
SELECT 1 FROM (SELECT COUNT(*) FROM MSysResources) AS dual
SELECT 1
SELECT 1 FROM "db_root"
SELECT 1 FROM "SYSIBM"."DUAL"
SELECT 1 FROM "SYSIBM"."SYSDUMMY1"
SELECT 1 FROM "RDB$DATABASE"
SELECT 1 FROM dual
SELECT 1 FROM "SYS"."DUMMY"
SELECT 1 FROM "INFORMATION_SCHEMA"."SYSTEM_USERS"
SELECT 1 FROM (SELECT 1 AS dual FROM systables WHERE tabid = 1)
SELECT 1 FROM (SELECT 1 AS dual) AS dual
SELECT 1 FROM dual
SELECT 1 FROM dual
SELECT 1 FROM dual
SELECT 1
SELECT 1
SELECT 1
SELECT 1 FROM [SYS].[DUMMY]
```

```
DSL.using(SQLDialect.ACCESS   ).selectOne().getSQL();
DSL.using(SQLDialect.ASE      ).selectOne().getSQL();
DSL.using(SQLDialect.CUBRID   ).selectOne().getSQL();
DSL.using(SQLDialect.DB2      ).selectOne().getSQL();
DSL.using(SQLDialect.DERBY    ).selectOne().getSQL();
DSL.using(SQLDialect.FIREBIRD ).selectOne().getSQL();
DSL.using(SQLDialect.H2       ).selectOne().getSQL();
DSL.using(SQLDialect.HANA     ).selectOne().getSQL();
DSL.using(SQLDialect.HSQLDB   ).selectOne().getSQL();
DSL.using(SQLDialect.INFORMIX ).selectOne().getSQL();
DSL.using(SQLDialect.INGRES   ).selectOne().getSQL();
DSL.using(SQLDialect.MARIADB  ).selectOne().getSQL();
DSL.using(SQLDialect.MYSQL    ).selectOne().getSQL();
DSL.using(SQLDialect.ORACLE   ).selectOne().getSQL();
DSL.using(SQLDialect.POSTGRES ).selectOne().getSQL();
DSL.using(SQLDialect.SQLITE   ).selectOne().getSQL();
DSL.using(SQLDialect.SQLSERVER).selectOne().getSQL();
DSL.using(SQLDialect.SYBASE   ).selectOne().getSQL();
```

Note, that some databases (H2, MySQL) can normally do without "DUAL". However, there exist some corner-cases with complex nested SELECT statements, where this will cause syntax errors (or parser bugs). To stay on the safe side, jOOQ will always render "dual" in those dialects.

4.7. Column expressions

Column expressions can be used in various SQL clauses in order to refer to one or several columns. This chapter explains how to form various types of column expressions with jOOQ. A particular type of column expression is given in the section about tuples or row value expressions, where an expression may have a degree of more than one.

Using column expressions in jOOQ

jOOQ allows you to freely create arbitrary column expressions using a fluent expression construction API. Many expressions can be formed as functions from DSL methods, other expressions can be formed based on a pre-existing column expression. For example:

```
// A regular table column expression
Field<String> field1 = BOOK.TITLE;

// A function created from the DSL using "prefix" notation
Field<String> field2 = trim(BOOK.TITLE);

// The same function created from a pre-existing Field using "postfix" notation
Field<String> field3 = BOOK.TITLE.trim();

// More complex function with advanced DSL syntax
Field<String> field4 = listAgg(BOOK.TITLE)
                         .withinGroupOrderBy(BOOK.ID.asc())
                         .over().partitionBy(AUTHOR.ID);
```

In general, it is up to you whether you want to use the "prefix" notation or the "postfix" notation to create new column expressions based on existing ones. The "SQL way" would be to use the "prefix notation", with functions created from the DSL. The "Java way" or "object-oriented way" would be to use

the "postfix" notation with functions created from org.jooq.Field objects. Both ways ultimately create the same query part, though.

4.7.1. Table columns

Table columns are the most simple implementations of a column expression. They are mainly produced by jOOQ's code generator and can be dereferenced from the generated tables. This manual is full of examples involving table columns. Another example is given in this query:

```
SELECT BOOK.ID, BOOK.TITLE
FROM BOOK
WHERE BOOK.TITLE LIKE '%SQL%'
ORDER BY BOOK.TITLE
```

```
create.select(BOOK.ID, BOOK.TITLE)
   .from(BOOK)
   .where(BOOK.TITLE.like("%SQL%"))
   .orderBy(BOOK.TITLE)
   .fetch();
```

Table columns implement a more specific interface called org.jooq.TableField, which is parameterised with its associated <R extends Record> record type.

See the manual's section about generated tables for more information about what is really generated by the code generator

4.7.2. Aliased columns

Just like tables, columns can be renamed using aliases. Here is an example:

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

Here is how it's done with jOOQ:

```
Record record = create.select(
         concat(AUTHOR.FIRST_NAME, val(" "), AUTHOR.LAST_NAME).as("author"),
         count().as("books"))
      .from(AUTHOR)
      .join(BOOK).on(AUTHOR.ID.eq(BOOK.AUTHOR_ID))
      .groupBy(AUTHOR.FIRST_NAME, 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"));
```

Unnamed column expressions

In most SQL databases, aliasing of column expressions in top level selects is optional. The database will generate a column name that is roughly based on the expression for documentation purposes (e.g. when running the query in a tool like SQL Developer), but applications cannot rely on the name explicitly. This is not a problem as columns can still be referenced by index.

In a similar fashion, jOOQ will assume an unspecified, generated column name for column expressions, based on their content.

```
-- Arithmetic expression
1 + 2

-- Correlated subquery
(SELECT 1 AS a)
```

```
// Arithmetic expression
inline(1).plus(inline(2));

// Correlated subquery
field(select(inline(1).as("a")));
```

These unnamed expressions can be used both in SQL as well as with jOOQ. However, do note that jOOQ will use Field.getName() to extract this column name from the field, when referencing the field or when nesting it in derived tables. In order to stay in full control of any such column names, it is always a good idea to provide explicit aliasing for column expressions, both in SQL as well as in jOOQ.

4.7.3. Cast expressions

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, java.lang.Long, java.lang.Integer, java.lang.Short, java.lang.Byte, java.lang.Double, java.lang.Float.

Sometimes, this automatic mapping might not be what you needed, or jOOQ cannot know the type of a field. In those cases you would write SQL type CAST like this:

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

in jOOQ, you can write something like that:

```
create.select(AUTHOR.LAST_NAME.cast(SQLDataType.VARCHAR(100))).fetch();
```

The same thing can be achieved by casting a Field directly to String.class, as VARCHAR is the default SQLDataType to map to Java's String

```
create.select(AUTHOR.LAST_NAME.cast(String.class)).fetch();
```

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

```
public interface Field<T> {

    // Cast this field to the type of another field
    <Z> Field<Z> cast(Field<Z> field);

    // Cast this field to a given DataType
    <Z> Field<Z> cast(DataType<Z> type);

    // Cast this field to the default DataType for a given Class
    <Z> Field<Z> cast(Class<? extends Z> type);
}

// And additional convenience methods in the DSL:
public class DSL {
    <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);
}
```

4.7.4. Datatype coercions

A slightly different use case than CAST expressions are data type coercions, which are not rendered through to generated SQL. Sometimes, you may want to pretend that a numeric value is really treated as a string value, for instance when binding a numeric bind value:

```
Field<String>  field1 = val(1).coerce(String.class);
Field<Integer> field2 = val("1").coerce(Integer.class);
```

In the above example, field1 will be treated by jOOQ as a Field<String>, binding the numeric literal 1 as a VARCHAR value. The same applies to field2, whose string literal "1" will be bound as an INTEGER value.

This technique is better than performing unsafe or rawtype casting in Java, if you cannot access the "right" field type from any given expression.

4.7.5. Collations

Many databases support "collations", which defines the sort order on character data types, such as VARCHAR.

Such databases usually allow for specifying:

- System-wide default collations
- Session-wide default collations
- Per-table specific default collations
- Per-column specific default collations
- Per-usage specific collation

The actual implementation is vendor-specific, including the way the above defaults override each other.

To accommodate most use-cases jOOQ 3.11 introduced the org.jooq.Collation type, which can be attached to a org.jooq.DataType through DataType.collate(Collation), or to a org.jooq.Field through Field.collate(Collation), for example:

```
SELECT *
FROM book
ORDER BY title COLLATE utf8_bin
```

```
create.selectFrom(BOOK)
      .orderBy(BOOK.TITLE.collate("utf8_bin"))
      .fetch();
```

4.7.6. Arithmetic expressions

Numeric arithmetic expressions

Your database can do the math for you. Arithmetic operations are implemented just like numeric 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(val(1).add(2).mul(val(5).sub(3)).div(2).mod(10)).fetch();
```

Operator precedence

jOOQ does not know any operator precedence (see also boolean operator precedence). All operations are evaluated from left to right, as with any object-oriented API. The two following expressions are the same:

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

Datetime arithmetic expressions

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

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

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

For more advanced datetime arithmetic, use the DSL's timestampDiff() and dateDiff() functions, as well as jOOQ's built-in SQL standard INTERVAL data type support:

- INTERVAL YEAR TO MONTH: org.jooq.types.YearToMonth
- INTERVAL DAY TO SECOND: org.jooq.types.DayToSecond

4.7.7. String concatenation

The SQL standard defines the concatenation operator to be an infix operator, similar to the ones we've seen in the chapter about arithmetic expressions. This operator looks like this: | |. Some other dialects do not support this operator, but expect a concat() function, instead. jOOQ renders the right operator / function, depending on your SQL dialect:

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

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

4.7.8. General functions

There are a variety of general functions supported by jOOQ As discussed in the chapter about SQL dialects functions are mostly emulated in your database, in case they are not natively supported.

This is a list of general functions supported by jOOQ's DSL:

- COALESCE: Get the first non-null value in a list of arguments.
- NULLIF: Return NULL if both arguments are equal, or the first argument, otherwise.
- NVL: Get the first non-null value among two arguments.
- NVL2: Get the second argument if the first is null, or the third argument, otherwise.

Please refer to the DSL Javadoc for more details.

4.7.9. Numeric functions

Math can be done efficiently in the database before returning results to your Java application. In addition to the arithmetic expressions discussed previously, jOOQ also supports a variety of numeric functions. As discussed in the chapter about SQL dialects numeric functions (as any function type) are mostly emulated in your database, in case they are not natively supported.

This is a list of numeric functions supported by jOOQ's DSL:

- ABS: Get the absolute value of a value.
- ACOS: Get the arc cosine of a value.
- ASIN: Get the arc sine of a value.
- ATAN: Get the arc tangent of a value.
- ATAN2: Get the atan2 function of two values.
- CEIL: Get the smalles integer value larger than a given numeric value.
- COS: Get the cosine of a value.
- COSH: Get the hyperbolic cosine of a value.
- COT: Get the cotangent of a value.
- COTH: Get the hyperbolic cotangent of a value.
- DEG: Transform radians into degrees.
- EXP: Calculate e^value.
- FLOOR: Get the largest integer value smaller than a given numeric value.
- GREATEST: Finds the greatest among all argument values (can also be used with non-numeric values).
- LEAST: Finds the least among all argument values (can also be used with non-numeric values).
- LN: Get the natural logarithm of a value.
- LOG: Get the logarithm of a value given a base.
- POWER: Calculate value^exponent.
- RAD: Transform degrees into radians.
- RAND: Get a random number.
- ROUND: Rounds a value to the nearest integer.
- SIGN: Get the sign of a value (-1, 0, 1).
- SIN: Get the sine of a value.
- SINH: Get the hyperbolic sine of a value.
- SQRT: Calculate the square root of a value.
- TAN: Get the tangent of a value.
- TANH: Get the hyperbolic tangent of a value.
- TRUNC: Truncate the decimals off a given value.

Please refer to the DSL Javadoc for more details.

4.7.10. Bitwise functions

Interestingly, bitwise functions and bitwise arithmetic is not very popular among SQL databases. Most databases only support a few bitwise operations, while others ship with the full set of operators. jOOQ's API includes most bitwise operations as listed below. In order to avoid ambiguities with conditional operators, all bitwise functions are prefixed with "bit"

- BIT_COUNT: Count the number of bits set to 1 in a number
- BIT_AND: Set only those bits that are set in two numbers
- BIT_OR: Set all bits that are set in at least one number
- BIT_NAND: Set only those bits that are set in two numbers, and inverse the result
- BIT_NOR: Set all bits that are set in at least one number, and inverse the result
- BIT_NOT: Inverse the bits in a number
- BIT_XOR: Set all bits that are set in at exactly one number
- BIT_XNOR: Set all bits that are set in at exactly one number, and inverse the result
- SHL: Shift bits to the left
- SHR: Shift bits to the right

Some background about bitwise operation emulation

As stated before, not all databases support all of these bitwise operations. jOOQ emulates them wherever this is possible. More details can be seen in this blog post: http://blog.jooq.org/2011/10/30/the-comprehensive-sql-bitwise-operations-compatibility-list/

4.7.11. String functions

String formatting can be done efficiently in the database before returning results to your Java application. As discussed in the chapter about SQL dialects string functions (as any function type) are mostly emulated in your database, in case they are not natively supported.

This is a list of numeric functions supported by jOOQ's DSL:

- ASCII: Get the ASCII code of a character.
- BIT_LENGTH: Get the length of a string in bits.
- CHAR_LENGTH: Get the length of a string in characters.
- CONCAT: Concatenate several strings.
- ESCAPE: Escape a string for use with the LIKE predicate.
- LENGTH: Get the length of a string.
- LOWER: Get a string in lower case letters.
- LPAD: Pad a string on the left side.
- LTRIM: Trim a string on the left side.
- OCTET_LENGTH: Get the length of a string in octets.
- POSITION: Find a string within another string.
- REPEAT: Repeat a string a given number of times.
- REPLACE: Replace a string within another string.
- RPAD: Pad a string on the right side.
- RTRIM: Trim a string on the right side.
- SUBSTRING: Get a substring of a string.
- TRIM: Trim a string on both sides.
- UPPER: Get a string in upper case letters.

Please refer to the DSL Javadoc for more details.

Regular expressions, REGEXP, REGEXP_LIKE, etc.

Various databases have some means of searching through columns using regular expressions if the LIKE predicate does not provide sufficient pattern matching power. While there are many different functions and operators in the various databases, jOOQ settled for the SQL:2008 standard REGEX_LIKE operator. Being an operator (and not a function), you should use the corresponding method on org.jooq.Field:

```
create.selectFrom(BOOK).where(TITLE.likeRegex("^.*SQL.*$")).fetch();
```

Note that the SQL standard specifies that patterns should follow the XQuery standards. In the real world, the POSIX regular expression standard is the most used one, some use Java regular expressions, and only a few ones use Perl regular expressions. jOOQ does not make any assumptions about regular expression syntax. For cross-database compatibility, please read the relevant database manuals carefully, to learn about the appropriate syntax. Please refer to the DSL Javadoc for more details.

4.7.12. Case sensitivity with strings

Most databases allow for specifying a COLLATION which allows for re-defining the ordering of string values. By default, ASCII, ISO, or Unicode encodings are applied to character data, and ordering is applied according to the respective encoding.

Sometimes, however, certain queries like to ignore parts of the encoding by treating upper-case and lower-case characters alike, such that ABC = abc, or such that ABC, jkl, XyZ are an ordered list of strings (case-insensitively).

For these ad-hoc ordering use-cases, most people resort to using LOWER() or UPPER() as follows:

```
-- Case-insensitive filtering:
SELECT * FROM BOOK
WHERE upper(TITLE) = 'ANIMAL FARM'

-- Case-insensitive ordering:
SELECT *
FROM AUTHOR
ORDER BY upper(FIRST_NAME), upper(LAST_NAME)
```

```
// Case-insensitive filtering:
create.selectFrom(BOOK)
      .where(upper(BOOK.TITLE).eq("ANIMAL FARM")).fetch();

// Case-insensitive ordering:
create.selectFrom(AUTHOR)
      .orderBy(upper(AUTHOR.FIRST_NAME), upper(AUTHOR.LAST_NAME))
      .fetch();
```

4.7.13. Date and time functions

This is a list of date and time functions supported by jOOQ's DSL:

- CURRENT_DATE: Get current date as a DATE object.
- CURRENT_TIME: Get current time as a TIME object.
- CURRENT_TIMESTAMP: Get current date as a TIMESTAMP object.
- DATE_ADD: Add a number of days or an interval to a date.
- DATE_DIFF: Get the difference in days between two dates.
- TIMESTAMP_ADD: Add a number of days or an interval to a timestamp.
- TIMESTAMP_DIFF: Get the difference as an INTERVAL DAY TO SECOND between two dates.

Intervals in jOOQ

jOOQ fills a gap opened by JDBC, which neglects an important SQL data type as defined by the SQL standards: INTERVAL types. See the manual's section about INTERVAL data types for more details.

4.7.14. System functions

This is a list of system functions supported by jOOQ's DSL:

- CURRENT_USER: Get current user.

4.7.15. Aggregate functions

Aggregate functions work just like functions, even if they have a slightly different semantics. Here are some example aggregate functions from the DSL:

```
// Every-day, SQL standard aggregate functions
AggregateFunction<Integer>    count();
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);

// String aggregate functions
AggregateFunction<String> groupConcat        (Field<?> field);
AggregateFunction<String> groupConcatDistinct(Field<?> field);
OrderedAggregateFunction<String> listAgg(Field<?> field);
OrderedAggregateFunction<String> listAgg(Field<?> field, String separator);

// 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);

// Linear regression functions
AggregateFunction<BigDecimal> regrAvgX     (Field<? extends Number> y, Field<? extends Number> x);
AggregateFunction<BigDecimal> regrAvgY     (Field<? extends Number> y, Field<? extends Number> x);
AggregateFunction<BigDecimal> regrCount    (Field<? extends Number> y, Field<? extends Number> x);
AggregateFunction<BigDecimal> regrIntercept(Field<? extends Number> y, Field<? extends Number> x);
AggregateFunction<BigDecimal> regrR2       (Field<? extends Number> y, Field<? extends Number> x);
AggregateFunction<BigDecimal> regrSlope    (Field<? extends Number> y, Field<? extends Number> x);
AggregateFunction<BigDecimal> regrSXX      (Field<? extends Number> y, Field<? extends Number> x);
AggregateFunction<BigDecimal> regrSXY      (Field<? extends Number> y, Field<? extends Number> x);
AggregateFunction<BigDecimal> regrSYY      (Field<? extends Number> y, Field<? extends Number> x);
```

Here's an example, counting the number of books any author has written:

```
SELECT AUTHOR_ID, COUNT(*)
FROM BOOK
GROUP BY AUTHOR_ID
```

```
create.select(BOOK.AUTHOR_ID, count())
      .from(BOOK)
      .groupBy(BOOK.AUTHOR_ID)
      .fetch();
```

Aggregate functions have strong limitations about when they may be used and when not. For instance, you can use aggregate functions in scalar queries. Typically, this means you only select aggregate functions, no regular columns or other column expressions. Another use case is to use them along with a GROUP BY clause as seen in the previous example. Note, that jOOQ does not check whether your using of aggregate functions is correct according to the SQL standards, or according to your database's behaviour.

Filtered aggregate functions

The SQL standard specifies an optional FILTER clause, that can be appended to all aggregate functions. This is very useful to implement "pivot" tables, such as the following:

```
SELECT
  count(*),
  count(*) FILTER (WHERE TITLE LIKE 'A%')
FROM BOOK
```

```
create.select(
         count(),
         count().filterWhere(BOOK.TITLE.like("A%")))
      .from(BOOK)
```

It is usually a good idea to calculate multiple aggregate functions in a single query, if this is possible.

Only few databases (e.g. HSQLDB, PostgreSQL) implement native support for the FILTER clause. In all other databases, jOOQ emulates the clause using a CASE expression:

```
SELECT
  count(*),
  count(CASE WHEN TITLE LIKE 'A%' THEN 1 END)
FROM BOOK
```

Aggregate functions exclude NULL values from aggregation, so the above query is equivalent to the one using FILTER.

Ordered-set aggregate functions

Oracle and some other databases support "ordered-set aggregate functions". This means you can provide an ORDER BY clause to an aggregate function, which will be taken into consideration when aggregating. The best example for this is Oracle's LISTAGG() (also known as GROUP_CONCAT in other SQL dialects). The following query groups by authors and concatenates their books' titles

```
SELECT   LISTAGG(TITLE, ', ')
         WITHIN GROUP (ORDER BY TITLE)
FROM     BOOK
GROUP BY AUTHOR_ID
```

```
create.select(listAgg(BOOK.TITLE, ", ")
      .withinGroupOrderBy(BOOK.TITLE))
      .from(BOOK)
      .groupBy(BOOK.AUTHOR_ID)
      .fetch();
```

The above query might yield:

```
+--------------------+
| LISTAGG            |
+--------------------+
| 1984, Animal Farm  |
| O Alquimista, Brida |
+--------------------+
```

FIRST and LAST: Oracle's "ranked" aggregate functions

Oracle allows for restricting aggregate functions using the KEEP() clause, which is supported by jOOQ. In Oracle, some aggregate functions (MIN, MAX, SUM, AVG, COUNT, VARIANCE, or STDDEV) can be restricted by this clause, hence org.jooq.AggregateFunction also allows for specifying it. Here are a couple of examples using this clause:

```
SUM(BOOK.AMOUNT_SOLD)
  KEEP(DENSE_RANK FIRST ORDER BY BOOK.AUTHOR_ID)
```

```
sum(BOOK.AMOUNT_SOLD)
  .keepDenseRankFirstOrderBy(BOOK.AUTHOR_ID)
```

User-defined aggregate functions

jOOQ also supports using your own user-defined aggregate functions. See the manual's section about user-defined aggregate functions for more details.

Window functions / analytical functions

In those databases that support window functions, jOOQ's org.jooq.AggregateFunction can be transformed into a window function / analytical function by calling over() on it. See the manual's section about window functions for more details.

4.7.16. 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. Note, that H2 and HSQLDB have implemented ROW_NUMBER() functions, without true windowing support.

As previously discussed, any org.jooq.AggregateFunction can be transformed into a window function using the over() method. See the chapter about aggregate functions for details. In addition to those, there are also some more window functions supported by jOOQ, as declared in the DSL:

```
// Ranking functions
    WindowOverStep<Integer>    rowNumber();
    WindowOverStep<Integer>    rank();
    WindowOverStep<Integer>    denseRank();
    WindowOverStep<BigDecimal> percentRank();

// Windowing functions
<T> WindowIgnoreNullsStep<T>   firstValue(Field<T> field);
<T> WindowIgnoreNullsStep<T>   lastValue(Field<T> field);
<T> WindowIgnoreNullsStep<T>   nthValue(Field<T> field, int nth);
<T> WindowIgnoreNullsStep<T>   nthValue(Field<T> field, Field<Integer> nth);
<T> WindowIgnoreNullsStep<T>   lead(Field<T> field);
<T> WindowIgnoreNullsStep<T>   lead(Field<T> field, int offset);
<T> WindowIgnoreNullsStep<T>   lead(Field<T> field, int offset, T defaultValue);
<T> WindowIgnoreNullsStep<T>   lead(Field<T> field, int offset, Field<T> defaultValue);
<T> WindowIgnoreNullsStep<T>   lag(Field<T> field);
<T> WindowIgnoreNullsStep<T>   lag(Field<T> field, int offset);
<T> WindowIgnoreNullsStep<T>   lag(Field<T> field, int offset, T defaultValue);
<T> WindowIgnoreNullsStep<T>   lag(Field<T> field, int offset, Field<T> defaultValue);

// Statistical functions
    WindowOverStep<BigDecimal> cumeDist();
    WindowOverStep<Integer>    ntile(int number);

// Inverse distribution functions
    OrderedAggregateFunction<BigDecimal> precentileCont(Number number);
    OrderedAggregateFunction<BigDecimal> precentileCont(Field<? extends Number> number);
    OrderedAggregateFunction<BigDecimal> precentileDisc(Number number);
    OrderedAggregateFunction<BigDecimal> precentileDisc(Field<? extends Number> number);
```

SQL distinguishes between various window function types (e.g. "ranking functions"). Depending on the function, SQL expects mandatory PARTITION BY or ORDER BY clauses within the OVER() clause. jOOQ does not enforce those rules for two reasons:

- Your JDBC driver or database already checks SQL syntax semantics
- Not all databases behave correctly according to the SQL standard

If possible, however, jOOQ tries to render missing clauses for you, if a given SQL dialect is more restrictive.

Some examples

Here are some simple examples of window functions with jOOQ:

```
-- Sample uses of ROW_NUMBER()
ROW_NUMBER() OVER()
ROW_NUMBER() OVER(PARTITION BY 1)
ROW_NUMBER() OVER(ORDER BY BOOK.ID)
ROW_NUMBER() OVER(PARTITION BY BOOK.AUTHOR_ID ORDER BY BOOK.ID)

-- Sample uses of FIRST_VALUE
FIRST_VALUE(BOOK.ID) OVER()
FIRST_VALUE(BOOK.ID IGNORE NULLS) OVER()
FIRST_VALUE(BOOK.ID RESPECT NULLS) OVER()
```

```
// Sample uses of rowNumber()
rowNumber().over()
rowNumber().over().partitionByOne()
rowNumber().over().partitionBy(BOOK.AUTHOR_ID)
rowNumber().over().partitionBy(BOOK.AUTHOR_ID).orderBy(BOOK.ID)

// Sample uses of firstValue()
firstValue(BOOK.ID).over()
firstValue(BOOK.ID).ignoreNulls().over()
firstValue(BOOK.ID).respectNulls().over()
```

An advanced window function example

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). Window functions are accessible from the previously seen org.jooq.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"))
      .fetch();
```

Window functions created from ordered-set aggregate functions

In the previous chapter about aggregate functions, we have seen the concept of "ordered-set aggregate functions", such as Oracle's LISTAGG(). These functions have a window function / analytical function variant, as well. For example:

```
SELECT   LISTAGG(TITLE, ', ')
         WITHIN GROUP (ORDER BY TITLE)
         OVER (PARTITION BY BOOK.AUTHOR_ID)
FROM     BOOK
```

```
create.select(listAgg(BOOK.TITLE, ", ")
      .withinGroupOrderBy(BOOK.TITLE)
      .over().partitionBy(BOOK.AUTHOR_ID))
      .from(BOOK)
      .fetch();
```

Window functions created from Oracle's FIRST and LAST aggregate functions

In the previous chapter about aggregate functions, we have seen the concept of "FIRST and LAST aggregate functions". These functions have a window function / analytical function variant, as well. For example:

```
SUM(BOOK.AMOUNT_SOLD)
  KEEP(DENSE_RANK FIRST ORDER BY BOOK.AUTHOR_ID)
  OVER(PARTITION BY 1)
```

```
sum(BOOK.AMOUNT_SOLD)
  .keepDenseRankFirstOrderBy(BOOK.AUTHOR_ID)
  .over().partitionByOne();
```

Window functions created from user-defined aggregate functions

User-defined aggregate functions also implement org.jooq.AggregateFunction, hence they can also be transformed into window functions using over(). This is supported by Oracle in particular. See the manual's section about user-defined aggregate functions for more details.

4.7.17. Grouping functions

ROLLUP() explained in SQL

The SQL standard defines special functions that can be used in the GROUP BY clause: the grouping functions. These functions can be used to generate several groupings in a single clause. This can best be explained in SQL. Let's take ROLLUP() for instance:

```
-- ROLLUP() with one argument
SELECT AUTHOR_ID, COUNT(*)
FROM BOOK
GROUP BY ROLLUP(AUTHOR_ID)


-- ROLLUP() with two arguments
SELECT AUTHOR_ID, PUBLISHED_IN, COUNT(*)
FROM BOOK
GROUP BY ROLLUP(AUTHOR_ID, PUBLISHED_IN)
```

```
-- The same query using UNION ALL:
  SELECT AUTHOR_ID, COUNT(*) FROM BOOK GROUP BY (AUTHOR_ID)
UNION ALL
  SELECT NULL, COUNT(*) FROM BOOK GROUP BY ()
ORDER BY 1 NULLS LAST

-- The same query using UNION ALL:
  SELECT AUTHOR_ID, PUBLISHED_IN, COUNT(*)
  FROM BOOK GROUP BY (AUTHOR_ID, PUBLISHED_IN)
UNION ALL
  SELECT AUTHOR_ID, NULL, COUNT(*)
  FROM BOOK GROUP BY (AUTHOR_ID)
UNION ALL
  SELECT NULL, NULL, COUNT(*)
  FROM BOOK GROUP BY ()
ORDER BY 1 NULLS LAST, 2 NULLS LAST
```

In English, the ROLLUP() grouping function provides N+1 groupings, when N is the number of arguments to the ROLLUP() function. Each grouping has an additional group field from the ROLLUP() argument field list. The results of the second query might look something like this:

```
+-----------+--------------+----------+
| AUTHOR_ID | PUBLISHED_IN | COUNT(*) |
+-----------+--------------+----------+
|         1 |         1945 |        1 | <- GROUP BY (AUTHOR_ID, PUBLISHED_IN)
|         1 |         1948 |        1 | <- GROUP BY (AUTHOR_ID, PUBLISHED_IN)
|         1 |         NULL |        2 | <- GROUP BY (AUTHOR_ID)
|         2 |         1988 |        1 | <- GROUP BY (AUTHOR_ID, PUBLISHED_IN)
|         2 |         1990 |        1 | <- GROUP BY (AUTHOR_ID, PUBLISHED_IN)
|         2 |         NULL |        2 | <- GROUP BY (AUTHOR_ID)
|      NULL |         NULL |        4 | <- GROUP BY ()
+-----------+--------------+----------+
```

CUBE() explained in SQL

CUBE() is different from ROLLUP() in the way that it doesn't just create N+1 groupings, it creates all 2^N possible combinations between all group fields in the CUBE() function argument list. Let's re-consider our second query from before:

```
-- CUBE() with two arguments
SELECT AUTHOR_ID, PUBLISHED_IN, COUNT(*)
FROM BOOK
GROUP BY CUBE(AUTHOR_ID, PUBLISHED_IN)
```

```
-- The same query using UNION ALL:
  SELECT AUTHOR_ID, PUBLISHED_IN, COUNT(*)
  FROM BOOK GROUP BY (AUTHOR_ID, PUBLISHED_IN)
UNION ALL
  SELECT AUTHOR_ID, NULL, COUNT(*)
  FROM BOOK GROUP BY (AUTHOR_ID)
UNION ALL
  SELECT NULL, PUBLISHED_IN, COUNT(*)
  FROM BOOK GROUP BY (PUBLISHED_IN)
UNION ALL
  SELECT NULL, NULL, COUNT(*)
  FROM BOOK GROUP BY ()
ORDER BY 1 NULLS FIRST, 2 NULLS FIRST
```

The results would then hold:

```
+-----------+-------------+----------+
| AUTHOR_ID | PUBLISHED_IN | COUNT(*) |
+-----------+-------------+----------+
|      NULL |         NULL |        2 | <- GROUP BY ()
|      NULL |         1945 |        1 | <- GROUP BY (PUBLISHED_IN)
|      NULL |         1948 |        1 | <- GROUP BY (PUBLISHED_IN)
|      NULL |         1988 |        1 | <- GROUP BY (PUBLISHED_IN)
|      NULL |         1990 |        1 | <- GROUP BY (PUBLISHED_IN)
|         1 |         NULL |        2 | <- GROUP BY (AUTHOR_ID)
|         1 |         1945 |        1 | <- GROUP BY (AUTHOR_ID, PUBLISHED_IN)
|         1 |         1948 |        1 | <- GROUP BY (AUTHOR_ID, PUBLISHED_IN)
|         2 |         NULL |        2 | <- GROUP BY (AUTHOR_ID)
|         2 |         1988 |        1 | <- GROUP BY (AUTHOR_ID, PUBLISHED_IN)
|         2 |         1990 |        1 | <- GROUP BY (AUTHOR_ID, PUBLISHED_IN)
+-----------+-------------+----------+
```

GROUPING SETS()

GROUPING SETS() are the generalised way to create multiple groupings. From our previous examples

- ROLLUP(AUTHOR_ID, PUBLISHED_IN) corresponds to GROUPING SETS((AUTHOR_ID, PUBLISHED_IN), (AUTHOR_ID), ())
- CUBE(AUTHOR_ID, PUBLISHED_IN) corresponds to GROUPING SETS((AUTHOR_ID, PUBLISHED_IN), (AUTHOR_ID), (PUBLISHED_IN), ())

This is nicely explained in the SQL Server manual pages about GROUPING SETS() and other grouping functions:
http://msdn.microsoft.com/en-us/library/bb510427(v=sql.105)

jOOQ's support for ROLLUP(), CUBE(), GROUPING SETS()

jOOQ fully supports all of these functions, as well as the utility functions GROUPING() and GROUPING_ID(), used for identifying the grouping set ID of a record. The DSL API thus includes:

```
// The various grouping function constructors
GroupField rollup(Field<?>... fields);
GroupField cube(Field<?>... fields);
GroupField groupingSets(Field<?>... fields);
GroupField groupingSets(Field<?>[]... fields);
GroupField groupingSets(Collection<? extends Field<?>>... fields);

// The utility functions generating IDs per GROUPING SET
Field<Integer> grouping(Field<?>);
Field<Integer> groupingId(Field<?>...);
```

MySQL's and CUBRID's WITH ROLLUP syntax

MySQL and CUBRID don't know any grouping functions, but they support a WITH ROLLUP clause, that is equivalent to simple ROLLUP() grouping functions. jOOQ emulates ROLLUP() in MySQL and CUBRID, by rendering this WITH ROLLUP clause. The following two statements mean the same:

```
-- Statement 1: SQL standard
GROUP BY ROLLUP(A, B, C)

-- Statement 2: SQL standard
GROUP BY A, ROLLUP(B, C)
```

```
-- Statement 1: MySQL
GROUP BY A, B, C WITH ROLLUP

-- Statement 2: MySQL
-- This is not supported in MySQL
```

4.7.18. User-defined functions

Some databases support user-defined functions, which can be embedded in any SQL statement, if you're using jOOQ's code generator. Let's say you have the following simple function in Oracle SQL:

```
CREATE OR REPLACE FUNCTION echo (INPUT NUMBER)
RETURN NUMBER
IS
BEGIN
    RETURN INPUT;
END echo;
```

The above function will be made available from a generated Routines class. You can use it like any other column expression:

```
SELECT echo(1) FROM DUAL WHERE echo(2) = 2
```

```
create.select(echo(1)).where(echo(2).eq(2)).fetch();
```

Note that user-defined functions returning CURSOR or ARRAY data types can also be used wherever table expressions can be used, if they are unnested

4.7.19. User-defined aggregate functions

Some databases support user-defined aggregate functions, which can then be used along with GROUP BY clauses or as window functions. An example for such a database is Oracle. With Oracle, you can define the following OBJECT type (the example was taken from the Oracle 11g documentation):

```
CREATE TYPE U_SECOND_MAX AS OBJECT
(
  MAX NUMBER, -- highest value seen so far
  SECMAX NUMBER, -- second highest value seen so far
  STATIC FUNCTION ODCIAggregateInitialize(sctx IN OUT U_SECOND_MAX) RETURN NUMBER,
  MEMBER FUNCTION ODCIAggregateIterate(self IN OUT U_SECOND_MAX, value IN NUMBER) RETURN NUMBER,
  MEMBER FUNCTION ODCIAggregateTerminate(self IN U_SECOND_MAX, returnValue OUT NUMBER, flags IN NUMBER) RETURN NUMBER,
  MEMBER FUNCTION ODCIAggregateMerge(self IN OUT U_SECOND_MAX, ctx2 IN U_SECOND_MAX) RETURN NUMBER
);

CREATE OR REPLACE TYPE BODY U_SECOND_MAX IS
STATIC FUNCTION ODCIAggregateInitialize(sctx IN OUT U_SECOND_MAX)
RETURN NUMBER IS
BEGIN
  SCTX := U_SECOND_MAX(0, 0);
  RETURN ODCIConst.Success;
END;

MEMBER FUNCTION ODCIAggregateIterate(self IN OUT U_SECOND_MAX, value IN NUMBER) RETURN NUMBER IS
BEGIN
  IF VALUE > SELF.MAX THEN
    SELF.SECMAX := SELF.MAX;
    SELF.MAX := VALUE;
  ELSIF VALUE > SELF.SECMAX THEN
    SELF.SECMAX := VALUE;
  END IF;
  RETURN ODCIConst.Success;
END;

MEMBER FUNCTION ODCIAggregateTerminate(self IN U_SECOND_MAX, returnValue OUT NUMBER, flags IN NUMBER) RETURN NUMBER IS
BEGIN
  RETURNVALUE := SELF.SECMAX;
  RETURN ODCIConst.Success;
END;

MEMBER FUNCTION ODCIAggregateMerge(self IN OUT U_SECOND_MAX, ctx2 IN U_SECOND_MAX) RETURN NUMBER IS
BEGIN
  IF CTX2.MAX > SELF.MAX THEN
    IF CTX2.SECMAX > SELF.SECMAX THEN
      SELF.SECMAX := CTX2.SECMAX;
    ELSE
      SELF.SECMAX := SELF.MAX;
    END IF;
    SELF.MAX := CTX2.MAX;
  ELSIF CTX2.MAX > SELF.SECMAX THEN
    SELF.SECMAX := CTX2.MAX;
  END IF;
  RETURN ODCIConst.Success;
END;
END;
```

The above OBJECT type is then available to function declarations as such:

```
CREATE FUNCTION SECOND_MAX (input NUMBER) RETURN NUMBER
PARALLEL_ENABLE AGGREGATE USING U_SECOND_MAX;
```

Using the generated aggregate function

jOOQ's code generator will detect such aggregate functions and generate them differently from regular user-defined functions. They implement the org.jooq.AggregateFunction type, as mentioned in the manual's section about aggregate functions. Here's how you can use the SECOND_MAX() aggregate function with jOOQ:

```
-- Get the second-latest publishing date by author
SELECT SECOND_MAX(PUBLISHED_IN)
FROM BOOK
GROUP BY AUTHOR_ID
```

```
// Routines.secondMax() can be static-imported
create.select(secondMax(BOOK.PUBLISHED_IN))
      .from(BOOK)
      .groupBy(BOOK.AUTHOR_ID)
      .fetch();
```

4.7.20. The CASE expression

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

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

-- OR:

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

```
DSL
      .when(AUTHOR.FIRST_NAME.eq("Paulo"), "brazilian")
      .when(AUTHOR.FIRST_NAME.eq("George"), "english")
      .otherwise("unknown");

// OR:

DSL.choose(AUTHOR.FIRST_NAME)
   .when("Paulo", "brazilian")
   .when("George", "english")
   .otherwise("unknown");
```

In jOOQ, both syntaxes are supported (The second one is emulated in Derby, which only knows the first one). Unfortunately, both case and else are reserved words in Java. jOOQ chose to use decode() from the Oracle DECODE function, or choose(), and otherwise(), which means the same as else.

A CASE expression can be used anywhere where you can place a column expression (or Field). For instance, you can SELECT the above expression, if you're selecting from AUTHOR:

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

The Oracle DECODE() function

Oracle knows a more succinct, but maybe less readable DECODE() function with a variable number of arguments. This function roughly does the same as the second case expression syntax. jOOQ supports the DECODE() function and emulates it using CASE expressions in all dialects other than Oracle:

```
-- Oracle:
DECODE(FIRST_NAME, 'Paulo', 'brazilian',
                   'George', 'english',
                   'unknown');

-- Other SQL dialects
CASE AUTHOR.FIRST_NAME WHEN 'Paulo'  THEN 'brazilian'
                       WHEN 'George' THEN 'english'
                                     ELSE 'unknown'
END
```

```
// Use the Oracle-style DECODE() function with jOOQ.
// Note, that you will not be able to rely on type-safety
DSL.decode(AUTHOR.FIRST_NAME,
    "Paulo", "brazilian",
    "George", "english",
    "unknown");
```

CASE clauses in an ORDER BY clause

Sort indirection is often implemented with a CASE clause of a SELECT's ORDER BY clause. See the manual's section about the ORDER BY clause for more details.

4.7.21. Sequences and serials

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
```

You can then use your generated sequence object directly in a SQL statement as such:

```
// Reference the sequence in a SELECT statement:
Field<BigInteger> s = S_AUTHOR_ID.nextval();
BigInteger nextID = create.select(s).fetchOne(s);

// Reference the sequence in an INSERT statement:
create.insertInto(AUTHOR, AUTHOR.ID, AUTHOR.FIRST_NAME, AUTHOR.LAST_NAME)
      .values(S_AUTHOR_ID.nextval(), val("William"), val("Shakespeare"))
      .execute();
```

- For more information about generated sequences, refer to the manual's section about generated sequences
- For more information about executing standalone calls to sequences, refer to the manual's section about sequence execution

4.7.22. Tuples or row value expressions

According to the SQL standard, row value expressions can have a degree of more than one. This is commonly used in the INSERT statement, where the VALUES row value constructor allows for providing a row value expression as a source for INSERT data. Row value expressions can appear in various other places, though. They are supported by jOOQ as records / rows. jOOQ's DSL allows for the construction of type-safe records up to the degree of 22. Higher-degree Rows are supported as well, but without any type-safety. Row types are modelled as follows:

```
// The DSL provides overloaded row value expression constructor methods:
public static <T1>             Row1<T1>             row(T1 t1)                       { ... }
public static <T1, T2>         Row2<T1, T2>         row(T1 t1, T2 t2)                { ... }
public static <T1, T2, T3>     Row3<T1, T2, T3>     row(T1 t1, T2 t2, T3 t3)         { ... }
public static <T1, T2, T3, T4> Row4<T1, T2, T3, T4> row(T1 t1, T2 t2, T3 t3, T4 t4) { ... }

// [ ... idem for Row5, Row6, Row7, ..., Row22 ]

// Degrees of more than 22 are supported without type-safety
public static RowN row(Object... values) { ... }
```

Using row value expressions in predicates

Row value expressions are incompatible with most other QueryParts, but they can be used as a basis for constructing various conditional expressions, such as:

- comparison predicates
- NULL predicates
- BETWEEN predicates
- IN predicates
- OVERLAPS predicate (for degree 2 row value expressions only)

See the relevant sections for more details about how to use row value expressions in predicates.

Using row value expressions in UPDATE statements

The UPDATE statement also supports a variant where row value expressions are updated, rather than single columns. See the relevant section for more details

Higher-degree row value expressions

jOOQ chose to explicitly support degrees up to 22 to match Scala's typesafe tuple, function and product support. Unlike Scala, however, jOOQ also supports higher degrees without the additional typesafety.

4.8. Conditional expressions

Conditions or conditional expressions are widely used in SQL and in the jOOQ API. They can be used in

- The CASE expression
- The JOIN clause (or JOIN .. ON clause, to be precise) of a SELECT statement, UPDATE statement, DELETE statement
- The WHERE clause of a SELECT statement, UPDATE statement, DELETE statement
- The CONNECT BY clause of a SELECT statement
- The HAVING clause of a SELECT statement
- The MERGE statement's ON clause

Boolean types in SQL

Before SQL:1999, boolean types did not really exist in SQL. They were modelled by 0 and 1 numeric/char values. With SQL:1999, true booleans were introduced and are now supported by most databases. In short, these are possible boolean values:

- 1 or TRUE
- 0 or FALSE
- NULL or UNKNOWN

It is important to know that SQL differs from many other languages in the way it interprets the NULL boolean value. Most importantly, the following facts are to be remembered:

- [ANY] = NULL yields NULL (not FALSE)
- [ANY] != NULL yields NULL (not TRUE)
- NULL = NULL yields NULL (not TRUE)
- NULL != NULL yields NULL (not FALSE)

For simplified NULL handling, please refer to the section about the DISTINCT predicate.

Note that jOOQ does not model these values as actual column expression compatible.

4.8.1. Condition building

With jOOQ, most conditional expressions are built from column expressions, calling various methods on them. For instance, to build a comparison predicate, you can write the following expression:

```
TITLE  = 'Animal Farm'
TITLE != 'Animal Farm'
```

```
BOOK.TITLE.eq("Animal Farm")
BOOK.TITLE.ne("Animal Farm")
```

Create conditions from the DSL

There are a few types of conditions, that can be created statically from the DSL. These are:

- plain SQL conditions, that allow you to phrase your own SQL string conditional expression
- The EXISTS predicate, a standalone predicate that creates a conditional expression
- Constant TRUE and FALSE conditional expressions

Connect conditions using boolean operators

Conditions can also be connected using boolean operators as will be discussed in a subsequent chapter.

4.8.2. AND, OR, NOT boolean operators

In SQL, as in most other languages, conditional expressions can be connected using the AND and OR binary operators, as well as the NOT unary operator, to form new conditional expressions. In jOOQ, this is modelled as such:

```
-- A simple conditional expression
TITLE = 'Animal Farm' OR TITLE = '1984'

-- A more complex conditional expression
       (TITLE = 'Animal Farm' OR TITLE = '1984')
AND NOT (AUTHOR.LAST_NAME = 'Orwell')
```

```
// A simple boolean connection
BOOK.TITLE.eq("Animal Farm").or(BOOK.TITLE.eq("1984"))

// A more complex conditional expression
BOOK.TITLE.eq("Animal Farm").or(BOOK.TITLE.eq("1984"))
    .andNot(AUTHOR.LAST_NAME.eq("Orwell"))
```

The above example shows that the number of parentheses in Java can quickly explode. Proper indentation may become crucial in making such code readable. In order to understand how jOOQ composes combined conditional expressions, let's assign component expressions first:

```
Condition a = BOOK.TITLE.eq("Animal Farm");
Condition b = BOOK.TITLE.eq("1984");
Condition c = AUTHOR.LAST_NAME.eq("Orwell");

Condition combined1 = a.or(b);             // These OR-connected conditions form a new condition, wrapped in parentheses
Condition combined2 = combined1.andNot(c); // The left-hand side of the AND NOT () operator is already wrapped in parentheses
```

The Condition API

Here are all boolean operators on the org.jooq.Condition interface:

```
and(Condition)            // Combine conditions with AND
and(String)               // Combine conditions with AND. Convenience for adding plain SQL to the right-hand side
and(String, Object...)    // Combine conditions with AND. Convenience for adding plain SQL to the right-hand side
and(String, QueryPart...) // Combine conditions with AND. Convenience for adding plain SQL to the right-hand side
andExists(Select<?>)      // Combine conditions with AND. Convenience for adding an exists predicate to the rhs
andNot(Condition)         // Combine conditions with AND. Convenience for adding an inverted condition to the rhs
andNotExists(Select<?>)   // Combine conditions with AND. Convenience for adding an inverted exists predicate to the rhs

or(Condition)             // Combine conditions with OR
or(String)                // Combine conditions with OR. Convenience for adding plain SQL to the right-hand side
or(String, Object...)     // Combine conditions with OR. Convenience for adding plain SQL to the right-hand side
or(String, QueryPart...)  // Combine conditions with OR. Convenience for adding plain SQL to the right-hand side
orExists(Select<?>)       // Combine conditions with OR. Convenience for adding an exists predicate to the rhs
orNot(Condition)          // Combine conditions with OR. Convenience for adding an inverted condition to the rhs
orNotExists(Select<?>)    // Combine conditions with OR. Convenience for adding an inverted exists predicate to the rhs

not()                     // Invert a condition (synonym for DSL.not(Condition)
```

4.8.3. Comparison predicate

In SQL, comparison predicates are formed using common comparison operators:

- = to test for equality
- <> or != to test for non-equality
- > to test for being strictly greater
- >= to test for being greater or equal
- < to test for being strictly less
- <= to test for being less or equal

Unfortunately, Java does not support operator overloading, hence these operators are also implemented as methods in jOOQ, like any other SQL syntax elements. The relevant parts of the org.jooq.Field interface are these:

```
eq or equal(T);                                 // =  (some bind value)
eq or equal(Field<T>);                          // =  (some column expression)
eq or equal(Select<? extends Record1<T>>);      // =  (some scalar SELECT statement)
ne or notEqual(T);                              // <> (some bind value)
ne or notEqual(Field<T>);                       // <> (some column expression)
ne or notEqual(Select<? extends Record1<T>>);   // <> (some scalar SELECT statement)
lt or lessThan(T);                              // <  (some bind value)
lt or lessThan(Field<T>);                       // <  (some column expression)
lt or lessThan(Select<? extends Record1<T>>);   // <  (some scalar SELECT statement)
le or lessOrEqual(T);                           // <= (some bind value)
le or lessOrEqual(Field<T>);                    // <= (some column expression)
le or lessOrEqual(Select<? extends Record1<T>>); // <= (some scalar SELECT statement)
gt or greaterThan(T);                           // >  (some bind value)
gt or greaterThan(Field<T>);                    // >  (some column expression)
gt or greaterThan(Select<? extends Record1<T>>); // >  (some scalar SELECT statement)
ge or greaterOrEqual(T);                        // >= (some bind value)
ge or greaterOrEqual(Field<T>);                 // >= (some column expression)
ge or greaterOrEqual(Select<? extends Record1<T>>); // >= (some scalar SELECT statement)
```

Note that every operator is represented by two methods. A verbose one (such as equal()) and a two-character one (such as eq()). Both methods are the same. You may choose either one, depending on your taste. The manual will always use the more verbose one.

jOOQ's convenience methods using comparison operators

In addition to the above, jOOQ provides a few convenience methods for common operations performed on strings using comparison predicates:

```
-- case insensitivity
LOWER(TITLE)  = LOWER('animal farm')
LOWER(TITLE) <> LOWER('animal farm')
```

```
// case insensitivity
BOOK.TITLE.equalIgnoreCase("animal farm")
BOOK.TITLE.notEqualIgnoreCase("animal farm")
```

4.8.4. Boolean operator precedence

As previously mentioned in the manual's section about arithmetic expressions, jOOQ does not implement operator precedence. All operators are evaluated from left to right, as expected in an object-oriented API. This is important to understand when combining boolean operators, such as AND, OR, and NOT. The following expressions are equivalent:

```
   A.and(B) .or(C) .and(D) .or(E)
(((A.and(B)).or(C)).and(D)).or(E)
```

In SQL, the two expressions wouldn't be the same, as SQL natively knows operator precedence.

```
   A AND B  OR C  AND D  OR E -- Precedence is applied
(((A AND B) OR C) AND D) OR E -- Precedence is overridden
```

4.8.5. Comparison predicate (degree > 1)

All variants of the comparison predicate that we've seen in the previous chapter also work for row value expressions. If your database does not support row value expression comparison predicates, jOOQ emulates them the way they are defined in the SQL standard:

```
-- Row value expressions (equal)
(A, B)    =  (X, Y)
(A, B, C) =  (X, Y, Z)
-- greater than
(A, B)    >  (X, Y)

(A, B, C) >  (X, Y, Z)


-- greater or equal
(A, B)    >= (X, Y)


(A, B, C) >= (X, Y, Z)



-- Inverse comparisons

(A, B)    <> (X, Y)
(A, B)    <  (X, Y)
(A, B)    <= (X, Y)
```

```
-- Equivalent factored-out predicates (equal)
(A = X) AND (B = Y)
(A = X) AND (B = Y) AND (C = Z)
-- greater than
(A > X)
  OR ((A = X) AND (B > Y))
(A > X)
  OR ((A = X) AND (B > Y))
  OR ((A = X) AND (B = Y) AND (C > Z))
-- greater or equal
(A > X)
  OR ((A = X) AND (B > Y))
  OR ((A = X) AND (B = Y))
(A > X)
  OR ((A = X) AND (B > Y))
  OR ((A = X) AND (B = Y) AND (C > Z))
  OR ((A = X) AND (B = Y) AND (C = Z))
-- For simplicity, these predicates are shown in terms
-- of their negated counter parts
NOT((A, B) =  (X, Y))
NOT((A, B) >= (X, Y))
NOT((A, B) >  (X, Y))
```

jOOQ supports all of the above row value expression comparison predicates, both with column expression lists and scalar subselects at the right-hand side:

```
-- With regular column expressions
(BOOK.AUTHOR_ID, BOOK.TITLE) = (1, 'Animal Farm')

-- With scalar subselects
(BOOK.AUTHOR_ID, BOOK.TITLE) = (
  SELECT PERSON.ID, 'Animal Farm'
  FROM PERSON
  WHERE PERSON.ID = 1
)
```

```
// Column expressions
row(BOOK.AUTHOR_ID, BOOK.TITLE).eq(1, "Animal Farm");

// Subselects
row(BOOK.AUTHOR_ID, BOOK.TITLE).eq(
  select(PERSON.ID, val("Animal Farm"))
  .from(PERSON)
  .where(PERSON.ID.eq(1))
);
```

4.8.6. Quantified comparison predicate

If the right-hand side of a comparison predicate turns out to be a non-scalar table subquery, you can wrap that subquery in a quantifier, such as ALL, ANY, or SOME. Note that the SQL standard defines ANY and SOME to be equivalent. jOOQ settled for the more intuitive ANY and doesn't support SOME. Here are some examples, supported by jOOQ:

```
TITLE = ANY('Animal Farm', '1982')
PUBLISHED_IN > ALL(1920, 1940)
```

```
BOOK.TITLE.eq(any("Animal Farm", "1982"));
BOOK.PUBLISHED_IN.gt(all(1920, 1940));
```

For the example, the right-hand side of the quantified comparison predicates were filled with argument lists. But it is easy to imagine that the source of values results from a subselect.

ANY and the IN predicate

It is interesting to note that the SQL standard defines the IN predicate in terms of the ANY-quantified predicate. The following two expressions are equivalent:

```
[ROW VALUE EXPRESSION] IN [IN PREDICATE VALUE]
```

```
[ROW VALUE EXPRESSION] = ANY [IN PREDICATE VALUE]
```

Typically, the IN predicate is more readable than the quantified comparison predicate.

4.8.7. NULL predicate

In SQL, you cannot compare NULL with any value using comparison predicates, as the result would yield NULL again, which is neither TRUE nor FALSE (see also the manual's section about conditional expressions). In order to test a column expression for NULL, use the NULL predicate as such:

```
TITLE IS NULL
TITLE IS NOT NULL
```

```
BOOK.TITLE.isNull()
BOOK.TITLE.isNotNull()
```

4.8.8. NULL predicate (degree > 1)

The SQL NULL predicate also works well for row value expressions, although it has some subtle, counter-intuitive features when it comes to inversing predicates with the NOT() operator! Here are some examples:

```
-- Row value expressions
(A, B) IS     NULL
(A, B) IS NOT NULL

-- Inverse of the above
NOT((A, B) IS     NULL)
NOT((A, B) IS NOT NULL)
```

```
-- Equivalent factored-out predicates
(A IS     NULL) AND (B IS     NULL)
(A IS NOT NULL) AND (B IS NOT NULL)

-- Inverse
(A IS NOT NULL) OR  (B IS NOT NULL)
(A IS     NULL) OR  (B IS     NULL)
```

The SQL standard contains a nice truth table for the above rules:

```
+----------------------+-----------+---------------+---------------+-------------------+
| Expression           | R IS NULL | R IS NOT NULL | NOT R IS NULL | NOT R IS NOT NULL |
+----------------------+-----------+---------------+---------------+-------------------+
| degree 1: null       | true      | false         | false         |  true             |
| degree 1: not null   | false     | true          | true          |  false            |
| degree > 1: all null | true      | false         | false         |  true             |
| degree > 1: some null| false     | false         | true          |  true             |
| degree > 1: none null| false     | true          | true          |  false            |
+----------------------+-----------+---------------+---------------+-------------------+
```

In jOOQ, you would simply use the isNull() and isNotNull() methods on row value expressions. Again, as with the row value expression comparison predicate, the row value expression NULL predicate is emulated by jOOQ, if your database does not natively support it:

```
row(BOOK.ID, BOOK.TITLE).isNull();
row(BOOK.ID, BOOK.TITLE).isNotNull();
```

4.8.9. DISTINCT predicate

Some databases support the DISTINCT predicate, which serves as a convenient, NULL-safe comparison predicate. With the DISTINCT predicate, the following truth table can be assumed:

- [ANY] IS DISTINCT FROM NULL yields TRUE
- [ANY] IS NOT DISTINCT FROM NULL yields FALSE
- NULL IS DISTINCT FROM NULL yields FALSE
- NULL IS NOT DISTINCT FROM NULL yields TRUE

For instance, you can compare two fields for distinctness, ignoring the fact that any of the two could be NULL, which would lead to funny results. This is supported by jOOQ as such:

```
TITLE IS DISTINCT FROM SUB_TITLE
TITLE IS NOT DISTINCT FROM SUB_TITLE
```

```
BOOK.TITLE.isDistinctFrom(BOOK.SUB_TITLE)
BOOK.TITLE.isNotDistinctFrom(BOOK.SUB_TITLE)
```

If your database does not natively support the DISTINCT predicate, jOOQ emulates it with an equivalent CASE expression, modelling the above truth table:

```
-- [A] IS DISTINCT FROM [B]
CASE WHEN [A] IS     NULL AND [B] IS     NULL THEN FALSE
     WHEN [A] IS     NULL AND [B] IS NOT NULL THEN TRUE
     WHEN [A] IS NOT NULL AND [B] IS     NULL THEN TRUE
     WHEN [A] =              [B]             THEN FALSE
     ELSE                                         TRUE
END
```

```
-- [A] IS NOT DISTINCT FROM [B]
CASE WHEN [A] IS     NULL AND [B] IS     NULL THEN TRUE
     WHEN [A] IS     NULL AND [B] IS NOT NULL THEN FALSE
     WHEN [A] IS NOT NULL AND [B] IS     NULL THEN FALSE
     WHEN [A] =              [B]             THEN TRUE
     ELSE                                         FALSE
END
```

... or better, if the INTERSECT set operation is supported:

```
-- [A] IS DISTINCT FROM [B]
NOT EXISTS(SELECT A INTERSECT SELECT B)
```

```
-- [A] IS NOT DISTINCT FROM [B]
EXISTS(SELECT a INTERSECT SELECT b)
```

4.8.10. BETWEEN predicate

The BETWEEN predicate can be seen as syntactic sugar for a pair of comparison predicates. According to the SQL standard, the following two predicates are equivalent:

```
[A] BETWEEN [B] AND [C]
```

```
[A] >= [B] AND [A] <= [C]
```

Note the inclusiveness of range boundaries in the definition of the BETWEEN predicate. Intuitively, this is supported in jOOQ as such:

```
PUBLISHED_IN     BETWEEN 1920 AND 1940
PUBLISHED_IN NOT BETWEEN 1920 AND 1940
```

```
BOOK.PUBLISHED_IN.between(1920).and(1940)
BOOK.PUBLISHED_IN.notBetween(1920).and(1940)
```

BETWEEN SYMMETRIC

The SQL standard defines the SYMMETRIC keyword to be used along with BETWEEN to indicate that you do not care which bound of the range is larger than the other. A database system should simply swap range bounds, in case the first bound is greater than the second one. jOOQ supports this keyword as well, emulating it if necessary.

```
PUBLISHED_IN     BETWEEN SYMMETRIC 1940 AND 1920
PUBLISHED_IN NOT BETWEEN SYMMETRIC 1940 AND 1920
```

```
BOOK.PUBLISHED_IN.betweenSymmetric(1940).and(1920)
BOOK.PUBLISHED_IN.notBetweenSymmetric(1940).and(1920)
```

The emulation is done trivially:

```
[A] BETWEEN SYMMETRIC [B] AND [C]
```

```
([A] BETWEEN [B] AND [C]) OR ([A] BETWEEN [C] AND [B])
```

4.8.11. BETWEEN predicate (degree > 1)

The SQL BETWEEN predicate also works well for row value expressions. Much like the BETWEEN predicate for degree 1, it is defined in terms of a pair of regular comparison predicates:

```
[A] BETWEEN           [B] AND [C]
[A] BETWEEN SYMMETRIC [B] AND [C]
```

```
 [A] >= [B] AND [A] <= [C]
([A] >= [B] AND [A] <= [C]) OR ([A] >= [C] AND [A] <= [B])
```

The above can be factored out according to the rules listed in the manual's section about row value expression comparison predicates.

jOOQ supports the BETWEEN [SYMMETRIC] predicate and emulates it in all SQL dialects where necessary. An example is given here:

```
row(BOOK.ID, BOOK.TITLE).between(1, "A").and(10, "Z");
```

4.8.12. LIKE predicate

LIKE predicates are popular for simple wildcard-enabled pattern matching. Supported wildcards in all SQL databases are:

- _: (single-character wildcard)
- %: (multi-character wildcard)

With jOOQ, the LIKE predicate can be created from any column expression as such:

```
TITLE     LIKE '%abc%'
TITLE NOT LIKE '%abc%'
```

```
BOOK.TITLE.like("%abc%")
BOOK.TITLE.notLike("%abc%")
```

Escaping operands with the LIKE predicate

Often, your pattern may contain any of the wildcard characters "_" and "%", in case of which you may want to escape them. jOOQ does not automatically escape patterns in like() and notLike() methods. Instead, you can explicitly define an escape character as such:

```
TITLE     LIKE '%The !%-Sign Book%' ESCAPE '!'
TITLE NOT LIKE '%The !%-Sign Book%' ESCAPE '!'
```

```
BOOK.TITLE.like("%The !%-Sign Book%", '!')
BOOK.TITLE.notLike("%The !%-Sign Book%", '!')
```

In the above predicate expressions, the exclamation mark character is passed as the escape character to escape wildcard characters "!_" and "!%", as well as to escape the escape character itself: "!!"

Please refer to your database manual for more details about escaping patterns with the LIKE predicate.

jOOQ's convenience methods using the LIKE predicate

In addition to the above, jOOQ provides a few convenience methods for common operations performed on strings using the LIKE predicate. Typical operations are "contains predicates", "starts with predicates", "ends with predicates", etc. Here is the full convenience API wrapping LIKE predicates:

```
-- case insensitivity
LOWER(TITLE) LIKE LOWER('%abc%')
LOWER(TITLE) NOT LIKE LOWER('%abc%')

-- contains and similar methods
TITLE LIKE '%' || 'abc' || '%'
TITLE LIKE 'abc' || '%'
TITLE LIKE '%' || 'abc'
```

```
// case insensitivity
BOOK.TITLE.likeIgnoreCase("%abc%")
BOOK.TITLE.notLikeIgnoreCase("%abc%")

// contains and similar methods
BOOK.TITLE.contains("abc")
BOOK.TITLE.startsWith("abc")
BOOK.TITLE.endsWith("abc")
```

Note, that jOOQ escapes % and _ characters in value in some of the above predicate implementations. For simplicity, this has been omitted in this manual.

4.8.13. IN predicate

In SQL, apart from comparing a value against several values, the IN predicate can be used to create semi-joins or anti-joins. jOOQ knows the following methods on the org.jooq.Field interface, to construct such IN predicates:

```
in(Collection<T>)                  // Construct an IN predicate from a collection of bind values
in(T...)                           // Construct an IN predicate from bind values
in(Field<?>...)                    // Construct an IN predicate from column expressions
in(Select<? extends Record1<T>>)   // Construct an IN predicate from a subselect
notIn(Collection<T>)               // Construct a NOT IN predicate from a collection of bind values
notIn(T...)                        // Construct a NOT IN predicate from bind values
notIn(Field<?>...)                 // Construct a NOT IN predicate from column expressions
notIn(Select<? extends Record1<T>>) // Construct a NOT IN predicate from a subselect
```

A sample IN predicate might look like this:

```
TITLE     IN ('Animal Farm', '1984')
TITLE NOT IN ('Animal Farm', '1984')
```

```
BOOK.TITLE.in("Animal Farm", "1984")
BOOK.TITLE.notIn("Animal Farm", "1984")
```

NOT IN and NULL values

Beware that you should probably not have any NULL values in the right hand side of a NOT IN predicate, as the whole expression would evaluate to NULL, which is rarely desired. This can be shown informally using the following reasoning:

```
-- The following conditional expressions are formally or informally equivalent
A NOT IN (B, C)
A != ANY(B, C)
A != B AND A != C

-- Substitute C for NULL, you'll get
A NOT IN (B, NULL)   -- Substitute C for NULL
A != B AND A != NULL -- From the above rules
A != B AND NULL      -- [ANY] != NULL yields NULL
NULL                 -- [ANY] AND NULL yields NULL
```

A good way to prevent this from happening is to use the EXISTS predicate for anti-joins, which is NULL-value insensitive. See the manual's section about conditional expressions to see a boolean truth table.

4.8.14. IN predicate (degree > 1)

The SQL IN predicate also works well for row value expressions. Much like the IN predicate for degree 1, it is defined in terms of a quantified comparison predicate. The two expressions are equivalent:

```
R IN [IN predicate value]
```

```
R = ANY [IN predicate value]
```

jOOQ supports the IN predicate with row value expressions.An example is given here:

```
-- Using an IN list
(BOOK.ID, BOOK.TITLE) IN ((1, 'A'), (2, 'B'))

-- Using a subselect
(BOOK.ID, BOOK.TITLE) IN (
  SELECT T.ID, T.TITLE
  FROM T
)
```

```
// Using an IN list
row(BOOK.ID, BOOK.TITLE).in(row(1, "A"), row(2, "B"));

// Using a subselect
row(BOOK.ID, BOOK.TITLE).in(
  select(T.ID, T.TITLE)
  .from(T)
);
```

In both cases, i.e. when using an IN list or when using a subselect, the type of the predicate is checked. Both sides of the predicate must be of equal degree and row type.

Emulation of the IN predicate where row value expressions aren't well supported is currently only available for IN predicates that do not take a subselect as an IN predicate value.

4.8.15. EXISTS predicate

Slightly less intuitive, yet more powerful than the previously discussed IN predicate is the EXISTS predicate, that can be used to form semi-joins or anti-joins. With jOOQ, the EXISTS predicate can be formed in various ways:

- From the DSL, using static methods. This is probably the most used case
- From a conditional expression using convenience methods attached to boolean operators
- From a SELECT statement using convenience methods attached to the where clause, and from other clauses

An example of an EXISTS predicate can be seen here:

```
    EXISTS (SELECT 1 FROM BOOK
            WHERE AUTHOR_ID = 3)
NOT EXISTS (SELECT 1 FROM BOOK
            WHERE AUTHOR_ID = 3)
```

```
   exists(create.selectOne().from(BOOK)
                            .where(BOOK.AUTHOR_ID.eq(3)));
notExists(create.selectOne().from(BOOK)
                            .where(BOOK.AUTHOR_ID.eq(3)));
```

Note that in SQL, the projection of a subselect in an EXISTS predicate is irrelevant. To help you write queries like the above, you can use jOOQ's selectZero() or selectOne() DSL methods

Performance of IN vs. EXISTS

In theory, the two types of predicates can perform equally well. If your database system ships with a sophisticated cost-based optimiser, it will be able to transform one predicate into the other, if you have all necessary constraints set (e.g. referential constraints, not null constraints). However, in reality, performance between the two might differ substantially. An interesting blog post investigating this topic on the MySQL database can be seen here:

http://blog.jooq.org/2012/07/27/not-in-vs-not-exists-vs-left-join-is-null-mysql/

4.8.16. OVERLAPS predicate

When comparing dates, the SQL standard allows for using a special OVERLAPS predicate, which checks whether two date ranges overlap each other. The following can be said:

```
-- This yields true
(DATE '2010-01-01', DATE '2010-01-03') OVERLAPS (DATE '2010-01-02' DATE '2010-01-04')

-- INTERVAL data types are also supported. This is equivalent to the above
(DATE '2010-01-01', CAST('+2 00:00:00' AS INTERVAL DAY TO SECOND)) OVERLAPS
(DATE '2010-01-02', CAST('+2 00:00:00' AS INTERVAL DAY TO SECOND))
```

The OVERLAPS predicate in jOOQ

jOOQ supports the OVERLAPS predicate on row value expressions of degree 2. The following methods are contained in org.jooq.Row2:

```
Condition overlaps(T1 t1, T2 t2);
Condition overlaps(Field<T1> t1, Field<T2> t2);
Condition overlaps(Row2<T1, T2> row);
```

This allows for expressing the above predicates as such:

```
// The date range tuples version
row(Date.valueOf('2010-01-01'), Date.valueOf('2010-01-03')).overlaps(Date.valueOf('2010-01-02'), Date.valueOf('2010-01-04'))

// The INTERVAL tuples version
row(Date.valueOf('2010-01-01'), new DayToSecond(2)).overlaps(Date.valueOf('2010-01-02'), new DayToSecond(2))
```

jOOQ's extensions to the standard

Unlike the standard (or any database implementing the standard), jOOQ also supports the OVERLAPS predicate for comparing arbitrary row vlaue expressions of degree 2. For instance, (1, 3) OVERLAPS (2, 4) will yield true in jOOQ. This is emulated as such

```
-- This predicate
(A, B) OVERLAPS (C, D)

-- can be emulated as such
(C <= B) AND (A <= D)
```

4.8.17. Query By Example (QBE)

A popular approach to querying database tables is called Query by Example, meaning that an "example" of a result record is provided instead of a formal query:

```
-- example book record:
ID          :
AUTHOR_ID   : 1
TITLE       :
PUBLISHED_IN: 1970
LANGUAGE_ID : 1
```

```
-- Corresponding query
SELECT *
FROM book
WHERE author_id = 1
AND published_in = 1970
AND language_id = 1
```

The translation from an example record to a query is fairly straight-forward:

- If a record attribute is set to a value, then that value is used for an equality predicate
- If a record attribute is not set, then that attribute is not used for any predicates

jOOQ knows a simple API called DSL.condition(Record), which translates a org.jooq.Record to a org.jooq.Condition:

```
BookRecord book = new BookRecord();
book.setAuthorId(1);
book.setPublishedIn(1970);
book.setLanguageId(1);

// Using the explicit condition() API
Result<BookRecord> books1 =
DSL.using(configuration)
   .selectFrom(BOOK)
   .where(condition(book))
   .fetch();

// Using the convenience API on DSLContext
Result<BookRecord> books2 = DSL.using(configuration).fetchByExample(book);
```

The latter API call makes use of the convenience API DSLContext.fetchByExample(TableRecord).

4.9. Dynamic SQL

In most cases, table expressions, column expressions, and conditional expressions as introduced in the previous chapters will be embedded into different SQL statement clauses as if the statement were a static SQL statement (e.g. in a view or stored procedure):

```
create.select(
          AUTHOR.FIRST_NAME.concat(AUTHOR.LAST_NAME),
          count()
      .from(AUTHOR)
      .join(BOOK).on(AUTHOR.ID.eq(BOOK.AUTHOR_ID))
      .groupBy(AUTHOR.ID, AUTHOR.FIRST_NAME, AUTHOR.LAST_NAME)
      .orderBy(count().desc())
      .fetch();
```

It is, however, interesting to think of all of the above expressions as what they are: expressions. And as such, nothing keeps users from extracting expressions and referencing them from outside the statement. The following statement is exactly equivalent:

```
SelectField<?>[] select = {
    AUTHOR.FIRST_NAME.concat(AUTHOR.LAST_NAME),
    count()
};
Table<?> from = AUTHOR.join(BOOK).on(AUTHOR.ID.eq(BOOK.AUTHOR_ID));
GroupField[] groupBy = { AUTHOR.ID, AUTHOR.FIRST_NAME, AUTHOR.LAST_NAME };
SortField<?>[] orderBy = { count().desc() };

create.select(select)
      .from(from)
      .groupBy(groupBy)
      .orderBy(orderBy)
      .fetch();
```

Each individual expression, and collection of expressions can be seen as an independent entity that can be

- Constructed dynamically
- Reused across queries

Dynamic construction is particularly useful in the case of the WHERE clause, for dynamic predicate building. For instance:

```
public Condition condition(HttpServletRequest request) {
    Condition result = trueCondition();

    if (request.getParameter("title") != null)
        result = result.and(BOOK.TITLE.like("%" + request.getParameter("title") + "%"));

    if (request.getParameter("author") != null)
        result = result.and(BOOK.AUTHOR_ID.in(
            selectOne().from(AUTHOR).where(
                    AUTHOR.FIRST_NAME.like("%" + request.getParameter("author") + "%")
                .or(AUTHOR.LAST_NAME .like("%" + request.getParameter("author") + "%"))
            )
        ));

    return result;
}

// And then:
create.select()
      .from(BOOK)
      .where(condition(httpRequest))
      .fetch();
```

The dynamic SQL building power may be one of the biggest advantages of using a runtime query model like the one offered by jOOQ. Queries can be created dynamically, of arbitrary complexity. In the above example, we've just constructed a dynamic WHERE clause. The same can be done for any other clauses, including dynamic FROM clauses (dynamic JOINs), or adding additional WITH clauses as needed.

4.10. Plain SQL

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 a host language (Java in this case), which was not made for exactly the same purposes as its hosted 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 by language constraints of its host language. We have seen many functionalities where the DSL becomes a bit 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 into any supported statement in these contexts:

- Plain SQL as a conditional expression
- Plain SQL as a column expression
- Plain SQL as a function
- Plain SQL as a table expression
- Plain SQL as a query

The DSL plain SQL API

Plain SQL API methods are usually overloaded in three ways. Let's look at the condition query part constructor:

```
// Construct a condition without bind values
// Example: condition("a = b")
Condition condition(String sql);

// Construct a condition with bind values
// Example: condition("a = ?", 1);
Condition condition(String sql, Object... bindings);

// Construct a condition taking other jOOQ object arguments
// Example: condition("a = {0}", val(1));
Condition condition(String sql, QueryPart... parts);
```

Both the bind value and the query part argument overloads make use of jOOQ's plain SQL templating language.

Please refer to the org.jooq.impl.DSL Javadoc for more details. The following is a more complete listing of plain SQL construction methods from the DSL:

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

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

// 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, Class<T> type, QueryPart... parts);
<T> Field<T> field(String sql, DataType<T> type);
<T> Field<T> field(String sql, DataType<T> type, Object... bindings);
<T> Field<T> field(String sql, DataType<T> type, QueryPart... parts);

// A field with a known name (properly escaped)
Field<Object> field(Name name);
<T> Field<T>  field(Name name, Class<T> type);
<T> Field<T>  field(Name name, DataType<T> type);

// 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);
Table<?> table(String sql, QueryPart... parts);

// A table with a known name (properly escaped)
Table<Record> table(Name name);

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

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

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

Apart from the general factory methods, plain SQL is also available 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("author a")
      .join("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)
      .fetch();
```

Important things to note about plain SQL!

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.

4.11. Plain SQL Templating Language

The plain SQL API, as documented in the previous chapter, supports a string templating mini-language that allows for constructing complex SQL string content from smaller parts. A simple example can be seen below, e.g. when looking for support for one of PostgreSQL's various vendor-specific operator types:

```
ARRAY[1,4,3] && ARRAY[2,1]
```

```
condition("{0} && {1}", array1, array2);
```

Such a plain SQL template always consists of two things:

- The SQL string fragment
- A set of org.jooq.QueryPart arguments, which are expected to be embedded in the SQL string

The SQL string may reference the arguments by 0-based indexing. Each argument may be referenced several times. For instance, SQLite's emulation of the REPEAT(string, count) function may look like this:

```
Field<Integer> count = val(3);
Field<String> string = val("abc");
field("replace(substr(quote(zeroblob(({0} + 1) / 2)), 3, {0}), '0', {1})", String.class, count, string);
//                                    ^                 ^         ^                   ^^^^^  ^^^^^^
//                                    |                 |         |                     |      |
// argument "count" is repeated twice: \-----------------+---------|---------------------/      |
// argument "string" is used only once:                          \----------------------------/
```

For convenience, there is also a DSL.list(QueryPart...) API that allows for wrapping a comma-separated list of query parts in a single template argument:

```
Field<String> a = val("a");
Field<String> b = val("b");
Field<String> c = val("c");

// These two produce the same result:
condition("my_column IN ({0}, {1}, {2})", a, b, c); // Using distinct template arguments
condition("my_column IN ({0})", list(a, b, c));    // Using a single template argument
```

Parsing rules

When processing these plain SQL templates, a mini parser is run that handles things like

- String literals
- Quoted names
- Comments
- JDBC escape sequences

The above are recognised by the templating engine and contents inside of them are ignored when replacing numbered placeholders and/or bind variables. For instance:

```
query(
  "SELECT /* In a comment, this is not a placeholder: {0}. And this is not a bind variable: ? */ title AS `title {1} ?` " +
  "-- Another comment without placeholders: {2} nor bind variables: ?" +
  "FROM book " +
  "WHERE title = 'In a string literal, this is not a placeholder: {3}. And this is not a bind variable: ?'"
);
```

The above query does not contain any numbered placeholders nor bind variables, because the tokens that would otherwise be searched for are contained inside of comments, string literals, or quoted names.

4.12. SQL Parser

A full-fledged SQL parser is available from DSLContext.parser() and from DSLContext.parsingConnection() (see the manual's section about parsing connections for the latter).

4.12.1. SQL Parser API

Goal

Historically, jOOQ implements an internal domain-specific language in Java, which generates SQL (an external domain-specific language) for use with JDBC. The jOOQ API is built from two parts: The DSL and the model API where the DSL API adds lexical convenience for programmers on top of the model API, which is really just a SQL expression tree, similar to what a SQL parser does inside of any database.

With this parser, the whole set of jOOQ functionality will now also be made available to anyone who is not using jOOQ directly, including JDBC and/or JPA users, e.g. through the parsing connection, which proxies all JDBC Connection calls to the jOOQ parser before forwarding them to the database, or through the DSLContext.parser() API, which allows for a more low-level access to the parser directly, e.g. for tool building on top of jOOQ.

The possibilities are endless, including standardised, SQL string based database migrations that work on any SQLDialect that is supported by jOOQ.

Example

This parser API allows for parsing an arbitrary SQL string fragment into a variety of jOOQ API elements:

- Parser.parse(String): This produces the org.jooq.Queries type, containing a batch of queries.
- Parser.parseQuery(String): This produces the org.jooq.Query type, containing a single query.
- Parser.parseResultQuery(String): This produces the org.jooq.ResultQuery type, containing a single query.
- Parser.parseTable(String): This produces the org.jooq.Table type, containing a table expression.
- Parser.parseField(String): This produces the org.jooq.Field type, containing a field expression.
- Parser.parseRow(String): This produces the org.jooq.Row type, containing a row expression.
- Parser.parseCondition(String): This produces the org.jooq.Condition type, containing a condition expression.
- Parser.parseName(String): This produces the org.jooq.Name type, containing a name expression.

The parser is able to parse any unspecified dialect to produce a jOOQ representation of the SQL expression, for instance:

```
ResultQuery<?> query =
DSL.using(configuration)
   .parser()
   .parseResultQuery("SELECT * FROM (VALUES (1, 'a'), (2, 'b')) t(a, b)")
```

The above SQL query is valid standard SQL and runs out of the box on PostgreSQL and SQL Server, among others. The jOOQ ResultQuery that is generated from this SQL string, however, will also work on any other database, as jOOQ can emulate the two interesting SQL features being used here:

- The VALUES() constructor
- The derived column list syntax (aliasing table ***and*** columns in one go)

The query might be rendered as follows on the H2 database, which supports VALUES(), but not derived column lists:

```
select
  t.a,
  t.b
from (
  (
    select
      null a,
      null b
    where 1 = 0
  )
  union all (
    select *
    from (values
      (1, 'a'),
      (2, 'b')
    ) t
  )
) t;
```

Or like this on Oracle, which supports neither feature:

```
select
  t.a,
  t.b
from (
  (
    select
      null a,
      null b
    from dual
    where 1 = 0
  )
  union all (
    select *
    from (
      (
        select
          1,
          'a'
        from dual
      )
      union all (
        select
          2,
          'b'
        from dual
      )
    ) t
  )
) t;
```

4.12.2. SQL Parser CLI

The Parser API can be used as a translator between source and target dialects programmatically, as we've seen in the previous section about the parser API. This functionality can also usefully be accessed on the command line as shown below:

```
$ java -cp jooq-3.11.11.jar org.jooq.ParserCLI -h
Usage:
  -f / --formatted                      Format output SQL
  -h / --help                           Display this help
  -k / --keyword    <RenderKeywordStyle>  Specify the output keyword style (org.jooq.conf.RenderKeywordStyle)
  -i / --identifier <RenderNameStyle>     Specify the output identifier style (org.jooq.conf.RenderNameStyle)
  -t / --to-dialect <SQLDialect>          Specify the output dialect (org.jooq.SQLDialect)
  -s / --sql        <String>              Specify the input SQL string

$ java -cp jooq-3.11.11.jar org.jooq.ParserCLI -t ORACLE -s "SELECT substring('abcde', 2, 3)"
select substr('abcde', 2, 3) from dual;
```

Another way to use this API is the https://www.jooq.org/translate website.

4.12.3. SQL Parser Grammar

The existing implementation of the SQL parser is a hand-written, recursive descent parser. There are great advantages of this approach over formal grammar-based, generated parsers (e.g. by using ANTLR). These advantages are, among others:

- They can be tuned easily for performance
- They are very simple and easy to maintain
- It's easy to implement corner cases of the grammar, which might require context (that's a big plus with SQL)
- It's the easiest way to bind a grammar to an existing backing expression tree implementation (which is what jOOQ really is)

Nevertheless, there is a grammar available for documentation purposes and it is included in the manual here:

(The layout of the grammar and the grammar itself is still work in progress)

The diagrams have been created with the neat RRDiagram library by Christopher Deckers.

4.13. Names and identifiers

Various SQL objects columns or tables can be referenced using names (often also called identifiers). SQL dialects differ in the way they understand names, syntactically. The differences include:

- The permitted characters to be used in "unquoted" names
- The permitted characters to be used in "quoted" names
- The name quoting characters (e.g. "double quotes", `backticks`, or [brackets])
- The standard case for case-insensitive ("unquoted") names

For the above reasons, and also to prevent an additional SQL injection risk where names might contain SQL code, jOOQ by default quotes all names in generated SQL to be sure they match what is really contained in your database. This means that the following names will be rendered

```
-- Unquoted name
AUTHOR.TITLE

-- MariaDB, MySQL
`AUTHOR`.`TITLE`

-- MS Access, SQL Server, Sybase ASE, Sybase SQL Anywhere
[AUTHOR].[TITLE]

-- All the others, including the SQL standard
"AUTHOR"."TITLE"
```

Note that you can influence jOOQ's name rendering behaviour through custom settings, if you prefer another name style to be applied.

Creating custom names

Custom, qualified or unqualified names can be created very easily using the DSL.name() constructor:

```
// Unqualified name
Name name = name("TITLE");

// Qualified name
Name name = name("AUTHOR", "TITLE");
```

Such names can be used as standalone QueryParts, or as DSL entry point for SQL expressions, like

- Common table expressions to be used with the WITH clause
- Window specifications to be used with the WINDOW clause

More details about how to use names / identifiers to construct such expressions can be found in the relevant sections of the manual.

4.14. Bind values and parameters

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. jOOQ currently does not cache prepared statements, internally.

The following sections explain how you can introduce bind values in jOOQ, and how you can control the way they are rendered and bound to SQL.

4.14.1. Indexed parameters

JDBC only knows indexed bind values. A typical example for using bind values with JDBC is this:

```
try (PreparedStatement stmt = connection.prepareStatement("SELECT * FROM BOOK WHERE ID = ? AND TITLE = ?")) {

    // bind values to the above statement for appropriate indexes
    stmt.setInt(1, 5);
    stmt.setString(2, "Animal Farm");
    stmt.executeQuery();
}
```

With dynamic SQL, keeping track of the number of question marks and their corresponding index may turn out to be hard. jOOQ abstracts this and lets you provide the bind value right where it is needed. A trivial example is this:

```
create.select().from(BOOK).where(BOOK.ID.eq(5)).and(BOOK.TITLE.eq("Animal Farm")).fetch();

// This notation is in fact a short form for the equivalent:
create.select().from(BOOK).where(BOOK.ID.eq(val(5))).and(BOOK.TITLE.eq(val("Animal Farm"))).fetch();
```

Note the using of DSL.val() to explicitly create an indexed bind value. You don't have to worry about that index. When the query is rendered, each bind value will render a question mark. When the query binds its variables, each bind value will generate the appropriate bind value index.

Extract bind values from a query

Should you decide to run the above query outside of jOOQ, using your own java.sql.PreparedStatement, you can do so as follows:

```
Select<?> select = create.select().from(BOOK).where(BOOK.ID.eq(5)).and(BOOK.TITLE.eq("Animal Farm"));

// Render the SQL statement:
String sql = select.getSQL();
assertEquals("SELECT * FROM BOOK WHERE ID = ? AND TITLE = ?", sql);

// Get the bind values:
List<Object> values = select.getBindValues();
assertEquals(2, values.size());
assertEquals(5, values.get(0));
assertEquals("Animal Farm", values.get(1));
```

You can also extract specific bind values by index from a query, if you wish to modify their underlying value after creating a query. This can be achieved as such:

```
Select<?> select = create.select().from(BOOK).where(BOOK.ID.eq(5)).and(BOOK.TITLE.eq("Animal Farm"));
Param<?> param = select.getParam("2");

// You could now modify the Query's underlying bind value:
if ("Animal Farm".equals(param.getValue())) {
    param.setConverted("1984");
}
```

For more details about jOOQ's internals, see the manual's section about QueryParts.

4.14.2. Named parameters

Some SQL access abstractions that are built on top of JDBC, or some that bypass JDBC may support named parameters. jOOQ allows you to give names to your parameters as well, although those names are not rendered to SQL strings by default. Here is an example of how to create named parameters using the org.jooq.Param type:

```
// Create a query with a named parameter. You can then use that name for accessing the parameter again
Query query1 = create.select().from(AUTHOR).where(LAST_NAME.eq(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(AUTHOR).where(LAST_NAME.eq(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(AUTHOR).where(LAST_NAME.eq("Poe"));
query1.bind(1, "Orwell");

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

In order to actually render named parameter names in generated SQL, use the DSLContext.renderNamedParams() method:

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

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

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

4.14.3. Inlined parameters

Sometimes, you may wish to avoid rendering bind variables while still using custom values in SQL. jOOQ refers to that as "inlined" bind values. When bind values are inlined, they render the actual value in SQL rather than a JDBC question mark. Bind value inlining can be achieved in several ways:

- Globally, by using the Settings and setting the org.jooq.conf.StatementType to STATIC_STATEMENT. This will inline all bind values for SQL statements rendered from such a Configuration.
- Per query locally, by using the Query.getSQL(ParamType) method.
- Per value locally, by using DSL.inline() methods.

In all cases, your inlined bind values will be properly escaped to avoid SQL syntax errors and SQL injection. Some examples:

```
// Use dedicated calls to inline() in order to specify
// single bind values to be rendered as inline values
// ------------------------------------------------
create.select()
      .from(AUTHOR)
      .where(LAST_NAME.eq(inline("Poe")))
      .fetch();

// Or render the whole query with inlined values
// --------------------------------------------
Settings settings = new Settings()
    .withStatementType(StatementType.STATIC_STATEMENT);

// Add the settings to the Configuration
DSLContext create = DSL.using(connection, SQLDialect.ORACLE, settings);

// Run queries that omit rendering schema names
create.select()
      .from(AUTHOR)
      .where(LAST_NAME.eq("Poe"))
      .fetch();
```

4.14.4. SQL injection

SQL injection is serious

SQL injection is a serious problem that needs to be taken care of thoroughly. A single vulnerability can be enough for an attacker to dump your whole database, and potentially seize your database server. We've blogged about the severity of this threat on the jOOQ blog.

SQL injection happens because a programming language (SQL) is used to dynamically create arbitrary server-side statements based on user input. Programmers must take lots of care not to mix the language parts (SQL) with the user input (bind variables)

SQL injection in jOOQ

With jOOQ, SQL is usually created via a type safe, non-dynamic Java abstract syntax tree, where bind variables are a part of that abstract syntax tree. It is not possible to expose SQL injection vulnerabilities this way.

However, jOOQ offers convenient ways of introducing plain SQL strings in various places of the jOOQ API (which are annotated using org.jooq.PlainSQL since jOOQ 3.6). While jOOQ's API allows you to specify bind values for use with plain SQL, you're not forced to do that. For instance, both of the following queries will lead to the same, valid result:

```
// This query will use bind values, internally.
create.fetch("SELECT * FROM BOOK WHERE ID = ? AND TITLE = ?", 5, "Animal Farm");

// This query will not use bind values, internally.
create.fetch("SELECT * FROM BOOK WHERE ID = 5 AND TITLE = 'Animal Farm'");
```

All methods in the jOOQ API that allow for plain (unescaped, untreated) SQL contain a warning message in their relevant Javadoc, to remind you of the risk of SQL injection in what is otherwise a SQL-injection-safe API.

4.15. QueryParts

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 accept(Context) method
- they can bind variables using the accept(Context) method

Both of these methods are contained in jOOQ's internal API's org.jooq.QueryPartInternal, which is internally implemented by every QueryPart.

The following sections explain some more details about SQL rendering and variable binding, as well as other implementation details about QueryParts in general.

4.15.1. SQL rendering

Every org.jooq.QueryPart must implement the accept(Context) method to render its SQL string to a org.jooq.RenderContext. This RenderContext has two purposes:

- It provides some information about the "state" of SQL rendering.
- It provides a common API for constructing SQL strings on the context's internal java.lang.StringBuilder

An overview of the org.jooq.RenderContext API is given here:

```
// These methods are useful for generating unique aliases within a RenderContext (and thus within a Query)
String peekAlias();
String nextAlias();

// These methods return rendered SQL
String render();
String render(QueryPart part);

// These methods allow for fluent appending of SQL to the RenderContext's internal StringBuilder
RenderContext keyword(String keyword);
RenderContext literal(String literal);
RenderContext sql(String sql);
RenderContext sql(char sql);
RenderContext sql(int sql);
RenderContext sql(QueryPart part);

// These methods allow for controlling formatting of SQL, if the relevant Setting is active
RenderContext formatNewLine();
RenderContext formatSeparator();
RenderContext formatIndentStart();
RenderContext formatIndentStart(int indent);
RenderContext formatIndentLockStart();
RenderContext formatIndentEnd();
RenderContext formatIndentEnd(int indent);
RenderContext formatIndentLockEnd();

// These methods control the RenderContext's internal state
boolean       inline();
RenderContext inline(boolean inline);
boolean       qualify();
RenderContext qualify(boolean qualify);
boolean       namedParams();
RenderContext namedParams(boolean renderNamedParams);
CastMode      castMode();
RenderContext castMode(CastMode mode);
Boolean       cast();
RenderContext castModeSome(SQLDialect... dialects);
```

The following additional methods are inherited from a common org.jooq.Context, which is shared among org.jooq.RenderContext and org.jooq.BindContext:

```
// These methods indicate whether fields or tables are being declared (MY_TABLE AS MY_ALIAS) or referenced (MY_ALIAS)
boolean declareFields();
Context declareFields(boolean declareFields);
boolean declareTables();
Context declareTables(boolean declareTables);

// These methods indicate whether a top-level query is being rendered, or a subquery
boolean subquery();
Context subquery(boolean subquery);

// These methods provide the bind value indices within the scope of the whole Context (and thus of the whole Query)
int nextIndex();
int peekIndex();
```

An example of rendering SQL

A simple example can be provided by checking out jOOQ's internal representation of a (simplified) CompareCondition. It is used for any org.jooq.Condition comparing two fields as for example the AUTHOR.ID = BOOK.AUTHOR_ID condition here:

```
-- [...]
FROM AUTHOR
JOIN BOOK ON AUTHOR.ID = BOOK.AUTHOR_ID
-- [...]
```

This is how jOOQ renders such a condition (simplified example):

```
@Override
public final void accept(Context<?> context) {
    // The CompareCondition delegates rendering of the Fields to the Fields
    // themselves and connects them using the Condition's comparator operator:
    context.visit(field1)
           .sql(" ")
           .keyword(comparator.toSQL())
           .sql(" ")
           .visit(field2);
}
```

See the manual's sections about custom QueryParts and plain SQL QueryParts to learn about how to write your own query parts in order to extend jOOQ.

4.15.2. Pretty printing SQL

As mentioned in the previous chapter about SQL rendering, there are some elements in the org.jooq.RenderContext that are used for formatting / pretty-printing rendered SQL. In order to obtain pretty-printed SQL, just use the following custom settings:

```
// Create a DSLContext that will render "formatted" SQL
DSLContext pretty = DSL.using(dialect, new Settings().withRenderFormatted(true));
```

And then, use the above DSLContext to render pretty-printed SQL:

```
String sql = pretty.select(
                       AUTHOR.LAST_NAME, count().as("c"))
                   .from(BOOK)
                   .join(AUTHOR)
                   .on(BOOK.AUTHOR_ID.eq(AUTHOR.ID))
                   .where(BOOK.TITLE.ne("1984"))
                   .groupBy(AUTHOR.LAST_NAME)
                   .having(count().eq(2))
                   .getSQL();
```

```
select
  "TEST"."AUTHOR"."LAST_NAME",
  count(*) "c"
from "TEST"."BOOK"
  join "TEST"."AUTHOR"
  on "TEST"."BOOK"."AUTHOR_ID" = "TEST"."AUTHOR"."ID"
where "TEST"."BOOK"."TITLE" <> '1984'
group by "TEST"."AUTHOR"."LAST_NAME"
having count(*) = 2
```

The section about ExecuteListeners shows an example of how such pretty printing can be used to log readable SQL to the stdout.

4.15.3. Variable binding

Every org.jooq.QueryPart must implement the accept(Context<?>) method. This Context has two purposes (among many others):

- It provides some information about the "state" of the variable binding in process.
- It provides a common API for binding values to the context's internal java.sql.PreparedStatement

An overview of the org.jooq.BindContext API is given here:

```
// This method provides access to the PreparedStatement to which bind values are bound
PreparedStatement statement();

// These methods provide convenience to delegate variable binding
BindContext bind(QueryPart part) throws DataAccessException;
BindContext bind(Collection<? extends QueryPart> parts) throws DataAccessException;
BindContext bind(QueryPart[] parts) throws DataAccessException;

// These methods perform the actual variable binding
BindContext bindValue(Object value, Class<?> type) throws DataAccessException;
BindContext bindValues(Object... values) throws DataAccessException;
```

Some additional methods are inherited from a common org.jooq.Context, which is shared among org.jooq.RenderContext and org.jooq.BindContext. Details are documented in the previous chapter about SQL rendering

An example of binding values to SQL

A simple example can be provided by checking out jOOQ's internal representation of a (simplified) CompareCondition. It is used for any org.jooq.Condition comparing two fields as for example the AUTHOR.ID = BOOK.AUTHOR_ID condition here:

```
-- [...]
WHERE AUTHOR.ID = ?
-- [...]
```

This is how jOOQ binds values on such a condition:

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

See the manual's sections about custom QueryParts and plain SQL QueryParts to learn about how to write your own query parts in order to extend jOOQ.

4.15.4. Custom data type bindings

jOOQ supports all the standard SQL data types out of the box, i.e. the types contained in java.sql.Types. But your domain model might be more specific, or you might be using a vendor-specific data type, such as JSON, HSTORE, or some other data structure. If this is the case, this section will be right for you, we'll see how you can create org.jooq.Converter types and org.jooq.Binding types.

Converters

The simplest use-case of injecting custom data types is by using org.jooq.Converter. A Converter can convert from a database type <T> to a user-defined type <U> and vice versa. You'll be implementing this SPI:

```
public interface Converter<T, U> {

    // Your conversion logic goes into these two methods, that can convert
    // between the database type T and the user type U:
    U from(T databaseObject);
    T to(U userObject);

 // You need to provide Class instances for each type, too:
    Class<T> fromType();
    Class<U> toType();
}
```

If, for instance, you want to use Java 8's java.time.LocalDate for SQL DATE and java.time.LocalDateTime for SQL TIMESTAMP, you write a converter like this:

```
import java.sql.Date;
import java.time.LocalDate;

import org.jooq.Converter;

public class LocalDateConverter implements Converter<Date, LocalDate> {

    @Override
    public LocalDate from(Date t) {
        return t == null ? null : LocalDate.parse(t.toString());
    }

    @Override
    public Date to(LocalDate u) {
        return u == null ? null : Date.valueOf(u.toString());
    }

    @Override
    public Class<Date> fromType() {
        return Date.class;
    }

    @Override
    public Class<LocalDate> toType() {
        return LocalDate.class;
    }
}
```

This converter can now be used in a variety of jOOQ API, most importanly to create a new data type:

```
DataType<LocalDate> type = SQLDataType.DATE.asConvertedDataType(new LocalDateConverter());
```

And data types, in turn, can be used with any org.jooq.Field, i.e. with any column expression, including plain SQL or name based ones:

```
DataType<LocalDate> type = SQLDataType.DATE.asConvertedDataType(new LocalDateConverter());

// Plain SQL based
Field<LocalDate> date1 = DSL.field("my_table.my_column", type);

// Name based
Field<LocalDate> date2 = DSL.field(name("my_table", "my_column"), type);
```

Bindings

While converters are very useful for simple use-cases, org.jooq.Binding is useful when you need to customise data type interactions at a JDBC level, e.g. when you want to bind a PostgreSQL JSON data type. Custom bindings implement the following SPI:

```
public interface Binding<T, U> extends Serializable {

    // A converter that does the conversion between the database type T
    // and the user type U (see previous examples)
    Converter<T, U> converter();

    // A callback that generates the SQL string for bind values of this
    // binding type. Typically, just ?, but also ?::json, etc.
    void sql(BindingSQLContext<U> ctx) throws SQLException;

    // Callbacks that implement all interaction with JDBC types, such as
    // PreparedStatement, CallableStatement, SQLOutput, SQLinput, ResultSet
    void register(BindingRegisterContext<U> ctx) throws SQLException;
    void set(BindingSetStatementContext<U> ctx) throws SQLException;
    void set(BindingSetSQLOutputContext<U> ctx) throws SQLException;
    void get(BindingGetResultSetContext<U> ctx) throws SQLException;
    void get(BindingGetStatementContext<U> ctx) throws SQLException;
    void get(BindingGetSQLInputContext<U> ctx) throws SQLException;
}
```

Below is full fledged example implementation that uses Google Gson to model JSON documents in Java

```
import static org.jooq.tools.Convert.convert;
import java.sql.*;
import org.jooq.*;
import org.jooq.impl.DSL;
import com.google.gson.*;

// We're binding <T> = Object (unknown database type), and <U> = JsonElement (user type)
public class PostgresJSONGsonBinding implements Binding<Object, JsonElement> {

    // The converter does all the work
    @Override
    public Converter<Object, JsonElement> converter() {
        return new Converter<Object, JsonElement>() {
            @Override
            public JsonElement from(Object t) {
                return t == null ? JsonNull.INSTANCE : new Gson().fromJson("" + t, JsonElement.class);
            }

            @Override
            public Object to(JsonElement u) {
                return u == null || u == JsonNull.INSTANCE ? null : new Gson().toJson(u);
            }

            @Override
            public Class<Object> fromType() {
                return Object.class;
            }

            @Override
            public Class<JsonElement> toType() {
                return JsonElement.class;
            }
        };
    }

    // Rending a bind variable for the binding context's value and casting it to the json type
    @Override
    public void sql(BindingSQLContext<JsonElement> ctx) throws SQLException {
        // Depending on how you generate your SQL, you may need to explicitly distinguish
        // between jOOQ generating bind variables or inlined literals.
        if (ctx.render().paramType() == ParamType.INLINED)
            ctx.render().visit(DSL.inline(ctx.convert(converter()).value())).sql("::json");
        else
            ctx.render().sql("?::json");
    }

    // Registering VARCHAR types for JDBC CallableStatement OUT parameters
    @Override
    public void register(BindingRegisterContext<JsonElement> ctx) throws SQLException {
        ctx.statement().registerOutParameter(ctx.index(), Types.VARCHAR);
    }

    // Converting the JsonElement to a String value and setting that on a JDBC PreparedStatement
    @Override
    public void set(BindingSetStatementContext<JsonElement> ctx) throws SQLException {
        ctx.statement().setString(ctx.index(), Objects.toString(ctx.convert(converter()).value(), null));
    }

    // Getting a String value from a JDBC ResultSet and converting that to a JsonElement
    @Override
    public void get(BindingGetResultSetContext<JsonElement> ctx) throws SQLException {
        ctx.convert(converter()).value(ctx.resultSet().getString(ctx.index()));
    }

    // Getting a String value from a JDBC CallableStatement and converting that to a JsonElement
    @Override
    public void get(BindingGetStatementContext<JsonElement> ctx) throws SQLException {
        ctx.convert(converter()).value(ctx.statement().getString(ctx.index()));
    }

    // Setting a value on a JDBC SQLOutput (useful for Oracle OBJECT types)
    @Override
    public void set(BindingSetSQLOutputContext<JsonElement> ctx) throws SQLException {
        throw new SQLFeatureNotSupportedException();
    }

    // Getting a value from a JDBC SQLInput (useful for Oracle OBJECT types)
    @Override
    public void get(BindingGetSQLInputContext<JsonElement> ctx) throws SQLException {
        throw new SQLFeatureNotSupportedException();
    }
}
```

Code generation

There is a special section in the manual explaining how to automatically tie your Converters and Bindings to your generated code. The relevant sections are:

- Custom data types for org.jooq.Converter
- Custom data type binding for org.jooq.Binding

4.15.5. Custom syntax elements

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 void accept(Context<?> ctx);
```

An example for implementing custom multiplication.

Here's an example org.jooq.impl.CustomField showing how to create a field multiplying another field by 2

```
// Create an anonymous CustomField, initialised with BOOK.ID arguments
final Field<Integer> IDx2 = new CustomField<Integer>(BOOK.ID.getName(), BOOK.ID.getDataType()) {
    @Override
    public void accept(Context<?> context) {
        context.visit(BOOK.ID).sql(" * ").visit(DSL.val(2));
    }
};

// Use the above field in a SQL statement:
create.select(IDx2).from(BOOK);
```

An example for implementing vendor-specific functions.

Many vendor-specific functions are not officially supported by jOOQ, but you can implement such support yourself using CustomField, for instance. Here's an example showing how to implement Oracle's TO_CHAR() function, emulating it in SQL Server using CONVERT():

```
// Create a CustomField implementation taking two arguments in its constructor
class ToChar extends CustomField<String> {

    final Field<?> arg0;
    final Field<?> arg1;

    ToChar(Field<?> arg0, Field<?> arg1) {
        super("to_char", SQLDataType.VARCHAR);

        this.arg0 = arg0;
        this.arg1 = arg1;
    }

    @Override
    public void accept(RenderContext context) {
        context.visit(delegate(context.configuration()));
    }

    private QueryPart delegate(Configuration configuration) {
        switch (configuration.dialect().family()) {
            case ORACLE:
                return DSL.field("TO_CHAR({0}, {1})", String.class, arg0, arg1);

            case SQLSERVER:
                return DSL.field("CONVERT(VARCHAR(8), {0}, {1})", String.class, arg0, arg1);

            default:
                throw new UnsupportedOperationException("Dialect not supported");
        }
    }
}
```

The above CustomField implementation can be exposed from your own custom DSL class:

```
public class MyDSL {
    public static Field<String> toChar(Field<?> field, String format) {
        return new ToChar(field, DSL.inline(format));
    }
}
```

4.15.6. Plain SQL QueryParts

If you don't need the 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. Plain SQL methods in jOOQ's API come in two flavours.

- method(String, Object...): This is a method that accepts a SQL string and a list of bind values that are to be bound to the variables contained in the SQL string
- method(String, QueryPart...): This is a method that accepts a SQL string and a list of QueryParts that are "injected" at the position of their respective placeholders in the SQL string

The above distinction is best explained using an example:

```
// Plain SQL using bind values. The value 5 is bound to the first variable, "Animal Farm" to the second variable:
create.selectFrom(BOOK).where(
    "BOOK.ID = ? AND TITLE = ?",     // The SQL string containing bind value placeholders ("?")
    5,                               // The bind value at index 1
    "Animal Farm"                    // The bind value at index 2
).fetch();

// Plain SQL using embeddable QueryPart placeholders (counting from zero).
// The QueryPart "index" is substituted for the placeholder {0}, the QueryPart "title" for {1}
Field<Integer> id   = val(5);
Field<String> title = val("Animal Farm");
create.selectFrom(BOOK).where(
    "BOOK.ID = {0} AND TITLE = {1}", // The SQL string containing QueryPart placeholders ("{N}")
    id,                              // The QueryPart at index 0
    title                            // The QueryPart at index 1
).fetch();
```

Note that for historic reasons the two API usages can also be mixed, although this is not recommended and the exact behaviour is unspecified.

Plain SQL templating specification

Templating with QueryPart placeholders (or bind value placeholders) requires a simple parsing logic to be applied to SQL strings. The jOOQ template parser behaves according to the following rules:

- Single-line comments (starting with -- in all databases (or #) in MySQL) are rendered without modification. Any bind variable or QueryPart placeholders in such comments are ignored.
- Multi-line comments (starting with /* and ending with */ in all databases) are rendered without modification. Any bind variable or QueryPart placeholders in such comments are ignored.
- String literals (starting and ending with ' in all databases, where all databases support escaping of the quote character by duplication as such: '', or in MySQL by escaping as such: \' (if Settings.backslashEscaping is turned on)) are rendered without modification. Any bind variable or QueryPart placeholders in such comments are ignored.
- Quoted names (starting and ending with " in most databases, with ` in MySQL, or with [and] in T-SQL databases) are rendered without modification. Any bind variable or QueryPart placeholders in such comments are ignored.
- JDBC escape syntax ({fn ...}, {d ...}, {t ...}, {ts ...}) is rendered without modification. Any bind variable or QueryPart placeholders in such comments are ignored.
- Bind variable placeholders (? or :name for named bind variables) are replaced by the matching bind value in case inlining is activated, e.g. through Settings.statementType == STATIC_STATEMENT.
- QueryPart placeholders ({number}) are replaced by the matching QueryPart.
- Keywords ({identifier}) are treated like keywords and rendered in the correct case according to Settings.renderKeywordStyle.

Tools for templating

A variety of API is provided to create template elements that are intended for use with the above templating mechanism. These tools can be found in org.jooq.impl.DSL

```
// Keywords (which are rendered according to Settings.renderKeywordStyle) can be specified as such:
public static Keyword keyword(String keyword) { ... }

// Identifiers / names (which are rendered according to Settings.renderNameStyle) can be specified as such:
public static Name name(String... qualifiedName) { ... }

// QueryPart lists (e.g. IN-lists for the IN predicate) can be generated via these methods:
public static QueryPart list(QueryPart... parts) { ... }
public static QueryPart list(Collection<? extends QueryPart> parts) { ... }
```

4.15.7. Serializability

A lot of jOOQ types extend and implement the java.io.Serializable interface for your convenience. Beware, however, that jOOQ will make no guarantees related to the serialisation format, and its backwards compatible evolution. This means that while it is generally safe to rely on jOOQ types being serialisable when two processes using the exact same jOOQ version transfer jOOQ state over some network, it is not safe to rely on persisting serialised jOOQ state to be deserialised again at a later time - even after a patch release upgrade!

As always with Java's serialisation, if you want reliable serialisation of Java objects, please use your own serialisation protocol, or use one of the official export formats.

What types are serializable?

The only transient, non-serializable element in any jOOQ object is the Configuration'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 may have to "re-attach" them to a Configuration

```
// 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
DSLContext create = DSL.using(connection, SQLDialect.ORACLE);
create.attach(select);
```

Automatically attaching QueryParts

Another way of attaching QueryParts automatically, or rather providing them with a new java.sql.Connection at will, is to hook into the Execute Listener support. More details about this can be found in the manual's chapter about ExecuteListeners

4.15.8. Custom SQL transformation

With jOOQ 3.2's org.jooq.VisitListener SPI, it is possible to perform custom SQL transformation to implement things like shared-schema multi-tenancy, or a security layer centrally preventing access to certain data. This SPI is extremely powerful, as you can make ad-hoc decisions at runtime regarding local or global transformation of your SQL statement. The following sections show a couple of simple, yet real-world use-cases.

4.15.8.1. Logging abbreviated bind values

When implementing a logger, one needs to carefully assess how much information should really be disclosed on what logger level. In log4j and similar frameworks, we distinguish between FATAL, ERROR, WARN, INFO, DEBUG, and TRACE. In DEBUG level, jOOQ's internal default logger logs all executed statements including inlined bind values as such:

```
Executing query          : select * from "BOOK" where "BOOK"."TITLE" like ?
-> with bind values      : select * from "BOOK" where "BOOK"."TITLE" like 'How I stopped worrying%'
```

But textual or binary bind values can get quite long, quickly filling your log files with irrelevant information. It would be good to be able to abbreviate such long values (and possibly add a remark to the logged statement). Instead of patching jOOQ's internals, we can just transform the SQL statements in the logger implementation, cleanly separating concerns. This can be done with the following VisitListener:

```
// This listener is inserted into a Configuration through a VisitListenerProvider that creates a
// new listener instance for every rendering lifecycle
public class BindValueAbbreviator extends DefaultVisitListener {

    private boolean anyAbbreviations = false;

    @Override
    public void visitStart(VisitContext context) {

        // Transform only when rendering values
        if (context.renderContext() != null) {
            QueryPart part = context.queryPart();

            // Consider only bind variables, leave other QueryParts untouched
            if (part instanceof Param<?>) {
                Param<?> param = (Param<?>) part;
                Object value = param.getValue();

                // If the bind value is a String (or Clob) of a given length, abbreviate it
                // e.g. using commons-lang's StringUtils.abbreviate()
                if (value instanceof String && ((String) value).length() > maxLength) {
                    anyAbbreviations = true;

                    // ... and replace it in the current rendering context (not in the Query)
                    context.queryPart(val(abbreviate((String) value, maxLength)));
                }

                // If the bind value is a byte[] (or Blob) of a given length, abbreviate it
                // e.g. by removing bytes from the array
                else if (value instanceof byte[] && ((byte[]) value).length > maxLength) {
                    anyAbbreviations = true;

                    // ... and replace it in the current rendering context (not in the Query)
                    context.queryPart(val(Arrays.copyOf((byte[]) value, maxLength)));
                }
            }
        }
    }

    @Override
    public void visitEnd(VisitContext context) {

        // If any abbreviations were performed before...
        if (anyAbbreviations) {

            // ... and if this is the top-level QueryPart, then append a SQL comment to indicate the abbreviation
            if (context.queryPartsLength() == 1) {
                context.renderContext().sql(" -- Bind values may have been abbreviated");
            }
        }
    }
}
```

If maxLength were set to 5, the above listener would produce the following log output:

```
Executing query          : select * from "BOOK" where "BOOK"."TITLE" like ?
-> with bind values      : select * from "BOOK" where "BOOK"."TITLE" like 'Ho...' -- Bind values may have been abbreviated
```

The above VisitListener is in place since jOOQ 3.3 in the org.jooq.tools.LoggerListener.

4.16. Zero-based vs one-based APIs

Any API that bridges two languages / mind sets, such as Java / SQL will inevitably face the difficulty of finding a consistent strategy to solving the "based-ness" problem. Should arrays be one-based or zero-based?

Clearly, Java is zero-based and SQL is one-based, and the best strategy for jOOQ is to keep things this way. The following are a set of rules that you should remember if this ever confuses you:

All SQL API is one-based

When using SQL API, such as the index-based ORDER BY clause, or window functions such as in the example below, jOOQ will not interpret indexes but send them directly as-is to the SQL engine. For instance:

```
SELECT nth_value(title, 3) OVER (ORDER BY id)
FROM book
ORDER BY 1
```

```
create.select(nthValue(BOOK.TITLE, 3).over(orderBy(BOOK.ID)))
      .from(BOOK)
      .orderBy(1).fetch();
```

In the above example, we're looking for the 3rd value of X in T ordered by Y. Clearly, this window function uses one-based indexing. The same is true for the ORDER BY clause, which orders the result by the 1st column - again one-based counting. There is no column zero in SQL.

All jOOQ API is zero-based

jOOQ is a Java API and as such, one-basedness would be quite surprising despite the fact that JDBC is one-based (see below). For instance, when you access a record by index in a jOOQ org.jooq.Result, given that the result extends java.util.List, you will use zero-based index access:

```
Result<?> result = create.select(BOOK.ID, BOOK.TITLE)
      .from(BOOK)
      .orderBy(1)
      .fetch();

for (int i = 0; i < result.size(); i++)
    System.out.println(result.get(i));
```

Unlike in JDBC, where java.sql.ResultSet#absolute(int) positions the underlying cursor at the one-based index, we Java developers really don't like that way of thinking. As can be seen in the above loop, we iterate over this result as we do over any other Java collection.

All JDBC API is one-based

An exception to the above rule is, obviously, all jOOQ API that is JDBC-interfacing. E.g. when you implement a custom data type binding, you will work with JDBC API directly from within jOOQ, which is one-based.

4.17. SQL building in Scala

jOOQ-Scala is a maven module used for leveraging some advanced Scala features for those users that wish to use jOOQ with Scala.

Using Scala's implicit defs to allow for operator overloading

The most obvious Scala feature to use in jOOQ are implicit defs for implicit conversions in order to enhance the org.jooq.Field type with SQL-esque operators.

The following depicts a trait which wraps all fields:

```
/**
 * A Scala-esque representation of {@link org.jooq.Field}, adding overloaded
 * operators for common jOOQ operations to arbitrary fields
 */
trait SAnyField[T] extends Field[T] {

    // String operations
    // -----------------

    def ||(value : String)            : Field[String]
    def ||(value : Field[_])          : Field[String]

    // Comparison predicates
    // ---------------------

    def ===(value : T)                : Condition
    def ===(value : Field[T])         : Condition

    def !==(value : T)                : Condition
    def !==(value : Field[T])         : Condition

    def <>(value : T)                 : Condition
    def <>(value : Field[T])          : Condition

    def >(value : T)                  : Condition
    def >(value : Field[T])           : Condition

    def >=(value : T)                 : Condition
    def >=(value : Field[T])          : Condition

    def <(value : T)                  : Condition
    def <(value : Field[T])           : Condition

    def <=(value : T)                 : Condition
    def <=(value : Field[T])          : Condition

    def <=>(value : T)                : Condition
    def <=>(value : Field[T])         : Condition
}
```

The following depicts a trait which wraps numeric fields:

```
/**
 * A Scala-esque representation of {@link org.jooq.Field}, adding overloaded
 * operators for common jOOQ operations to numeric fields
 */
trait SNumberField[T <: Number] extends SAnyField[T] {

    // Arithmetic operations
    // ---------------------

    def unary_-                       : Field[T]

    def +(value : Number)             : Field[T]
    def +(value : Field[_ <: Number]) : Field[T]

    def -(value : Number)             : Field[T]
    def -(value : Field[_ <: Number]) : Field[T]

    def *(value : Number)             : Field[T]
    def *(value : Field[_ <: Number]) : Field[T]

    def /(value : Number)             : Field[T]
    def /(value : Field[_ <: Number]) : Field[T]

    def %(value : Number)             : Field[T]
    def %(value : Field[_ <: Number]) : Field[T]

    // Bitwise operations
    // ------------------

    def unary_~                       : Field[T]

    def &(value : T)                  : Field[T]
    def &(value : Field[T])           : Field[T]

    def |(value : T)                  : Field[T]
    def |(value : Field[T])           : Field[T]

    def ^(value : T)                  : Field[T]
    def ^(value : Field[T])           : Field[T]

    def <<(value : T)                 : Field[T]
    def <<(value : Field[T])          : Field[T]

    def >>(value : T)                 : Field[T]
    def >>(value : Field[T])          : Field[T]
}
```

An example query using such overloaded operators would then look like this:

```
select (
  BOOK.ID * BOOK.AUTHOR_ID,
  BOOK.ID + BOOK.AUTHOR_ID * 3 + 4,
  BOOK.TITLE || " abc" || " xy")
from BOOK
leftOuterJoin (
  select (x.ID, x.YEAR_OF_BIRTH)
  from x
  limit 1
  asTable x.getName()
)
on BOOK.AUTHOR_ID === x.ID
where (BOOK.ID <> 2)
or (BOOK.TITLE in ("O Alquimista", "Brida"))
fetch
```

Scala 2.10 Macros

This feature is still being experimented with. With Scala Macros, it might be possible to inline a true SQL dialect into the Scala syntax, backed by the jOOQ API. Stay tuned!

5. SQL execution

In a previous section of the manual, we've seen how jOOQ can be used to build SQL that can be executed with any API including JDBC or ... jOOQ. This section of the manual deals with various means of actually executing SQL with jOOQ.

SQL execution with JDBC

JDBC calls executable objects "java.sql.Statement". It distinguishes between three types of statements:

- java.sql.Statement, or "static statement": This statement type is used for any arbitrary type of SQL statement. It is particularly useful with inlined parameters
- java.sql.PreparedStatement: This statement type is used for any arbitrary type of SQL statement. It is particularly useful with indexed parameters (note that JDBC does not support named parameters)
- java.sql.CallableStatement: This statement type is used for SQL statements that are "called" rather than "executed". In particular, this includes calls to stored procedures. Callable statements can register OUT parameters

Today, the JDBC API may look weird to users being used to object-oriented design. While statements hide a lot of SQL dialect-specific implementation details quite well, they assume a lot of knowledge about the internal state of a statement. For instance, you can use the PreparedStatement.addBatch() method, to add a the prepared statement being created to an "internal list" of batch statements. Instead of returning a new type, this method forces user to reflect on the prepared statement's internal state or "mode".

jOOQ is wrapping JDBC

These things are abstracted away by jOOQ, which exposes such concepts in a more object-oriented way. For more details about jOOQ's batch query execution, see the manual's section about batch execution.

The following sections of this manual will show how jOOQ is wrapping JDBC for SQL execution

Alternative execution modes

Just because you can, doesn't mean you must. At the end of this chapter, we'll show how you can use jOOQ to generate SQL statements that are then executed with other APIs, such as Spring's JdbcTemplate, or Hibernate. For more information see the section about alternative execution models.

5.1. Comparison between jOOQ and JDBC

Similarities with JDBC

Even if there are two general types of Query, there are a lot of similarities between JDBC and jOOQ. Just to name a few:

- Both APIs return the number of affected records in non-result queries. JDBC: Statement.executeUpdate(), jOOQ: Query.execute()
- Both APIs return a scrollable result set type from result queries. JDBC: java.sql.ResultSet, jOOQ: org.jooq.Result

Differences to JDBC

Some of the most important differences between JDBC and jOOQ are listed here:

- Query vs. ResultQuery: JDBC does not formally distinguish between queries that can return results, and queries that cannot. The same API is used for both. This greatly reduces the possibility for fetching convenience methods
- Exception handling: While SQL uses the checked java.sql.SQLException, jOOQ wraps all exceptions in an unchecked org.jooq.exception.DataAccessException
- org.jooq.Result: Unlike its JDBC counter-part, this type implements java.util.List and is fully loaded into Java memory, freeing resources as early as possible. Just like statements, this means that users don't have to deal with a "weird" internal result set state.
- org.jooq.Cursor: If you want more fine-grained control over how many records are fetched into memory at once, you can still do that using jOOQ's lazy fetching feature
- Statement type: jOOQ does not formally distinguish between static statements and prepared statements. By default, all statements are prepared statements in jOOQ, internally. Executing a statement as a static statement can be done simply using a custom settings flag
- Closing Statements: JDBC keeps open resources even if they are already consumed. With JDBC, there is a lot of verbosity around safely closing resources. In jOOQ, resources are closed after consumption, by default. If you want to keep them open after consumption, you have to explicitly say so.
- JDBC flags: JDBC execution flags and modes are not modified. They can be set fluently on a Query
- Zero-based vs one-based APIs: JDBC is a one-based API, jOOQ is a zero-based API. While this makes sense intuitively (JDBC is the less intuitive API from a Java perspective), it can lead to confusion in certain cases.

5.2. Query vs. ResultQuery

Unlike JDBC, jOOQ has a lot of knowledge about a SQL query's structure and internals (see the manual's section about SQL building). Hence, jOOQ distinguishes between these two fundamental types of queries. While every org.jooq.Query can be executed, only org.jooq.ResultQuery can return results (see

the manual's section about fetching to learn more about fetching results). With plain SQL, the distinction can be made clear most easily:

```
// Create a Query object and execute it:
Query query = create.query("DELETE FROM BOOK");
query.execute();

// Create a ResultQuery object and execute it, fetching results:
ResultQuery<Record> resultQuery = create.resultQuery("SELECT * FROM BOOK");
Result<Record> result = resultQuery.fetch();
```

5.3. Fetching

Fetching is something that has been completely neglegted by JDBC and also by various other database abstraction libraries. Fetching is much more than just looping or listing records or mapped objects. There are so many ways you may want to fetch data from a database, it should be considered a first-class feature of any database abstraction API. Just to name a few, here are some of jOOQ's fetching modes:

- Untyped vs. typed fetching: Sometimes you care about the returned type of your records, sometimes (with arbitrary projections) you don't.
- Fetching arrays, maps, or lists: Instead of letting you transform your result sets into any more suitable data type, a library should do that work for you.
- Fetching through handler callbacks: This is an entirely different fetching paradigm. With Java 8's lambda expressions, this will become even more powerful.
- Fetching through mapper callbacks: This is an entirely different fetching paradigm. With Java 8's lambda expressions, this will become even more powerful.
- Fetching custom POJOs: This is what made Hibernate and JPA so strong. Automatic mapping of tables to custom POJOs.
- Lazy vs. eager fetching: It should be easy to distinguish these two fetch modes.
- Fetching many results: Some databases allow for returning many result sets from a single query. JDBC can handle this but it's very verbose. A list of results should be returned instead.
- Fetching data asynchronously: Some queries take too long to execute to wait for their results. You should be able to spawn query execution in a separate process.

Convenience and how ResultQuery, Result, and Record share API

The term "fetch" is always reused in jOOQ when you can fetch data from the database. An org.jooq.ResultQuery provides many overloaded means of fetching data:

Various modes of fetching

These modes of fetching are also documented in subsequent sections of the manual

```
// The "standard" fetch
Result<R> fetch();

// The "standard" fetch when you know your query returns only one record. This may return null.
R fetchOne();

// The "standard" fetch when you know your query returns only one record.
Optional<R> fetchOptional();

// The "standard" fetch when you only want to fetch the first record
R fetchAny();

// Create a "lazy" Cursor, that keeps an open underlying JDBC ResultSet
Cursor<R> fetchLazy();
Cursor<R> fetchLazy(int fetchSize);
Stream<R> stream();

// Fetch several results at once
List<Result<Record>> fetchMany();

// Fetch records into a custom callback
<H extends RecordHandler<R>> H fetchInto(H handler);

// Map records using a custom callback
<E> List<E> fetch(RecordMapper<? super R, E> mapper);

// Execute a ResultQuery with jOOQ, but return a JDBC ResultSet, not a jOOQ object
ResultSet fetchResultSet();
```

Fetch convenience

These means of fetching are also available from org.jooq.Result and org.jooq.Record APIs

```
// These methods are convenience for fetching only a single field,
// possibly converting results to another type
<T>    List<T> fetch(Field<T> field);
<T>    List<T> fetch(Field<?> field, Class<? extends T> type);
<T, U> List<U> fetch(Field<T> field, Converter<? super T, U> converter);
       List<?> fetch(int fieldIndex);
<T>    List<T> fetch(int fieldIndex, Class<? extends T> type);
<U>    List<U> fetch(int fieldIndex, Converter<?, U> converter);
       List<?> fetch(String fieldName);
<T>    List<T> fetch(String fieldName, Class<? extends T> type);
<U>    List<U> fetch(String fieldName, Converter<?, U> converter);

// These methods are convenience for fetching only a single field, possibly converting results to another type
// Instead of returning lists, these return arrays
<T>    T[]      fetchArray(Field<T> field);
<T>    T[]      fetchArray(Field<?> field, Class<? extends T> type);
<T, U> U[]      fetchArray(Field<T> field, Converter<? super T, U> converter);
       Object[] fetchArray(int fieldIndex);
<T>    T[]      fetchArray(int fieldIndex, Class<? extends T> type);
<U>    U[]      fetchArray(int fieldIndex, Converter<?, U> converter);
       Object[] fetchArray(String fieldName);
<T>    T[]      fetchArray(String fieldName, Class<? extends T> type);
<U>    U[]      fetchArray(String fieldName, Converter<?, U> converter);

// These methods are convenience for fetching only a single field from a single record,
// possibly converting results to another type
<T>    T      fetchOne(Field<T> field);
<T>    T      fetchOne(Field<?> field, Class<? extends T> type);
<T, U> U      fetchOne(Field<T> field, Converter<? super T, U> converter);
       Object fetchOne(int fieldIndex);
<T>    T      fetchOne(int fieldIndex, Class<? extends T> type);
<U>    U      fetchOne(int fieldIndex, Converter<?, U> converter);
       Object fetchOne(String fieldName);
<T>    T      fetchOne(String fieldName, Class<? extends T> type);
<U>    U      fetchOne(String fieldName, Converter<?, U> converter);
```

Fetch transformations

These means of fetching are also available from org.jooq.Result and org.jooq.Record APIs

```
// Transform your Records into arrays, Results into matrices
       Object[][] fetchArrays();
       Object[]   fetchOneArray();

// Reduce your Result object into maps
<K>    Map<K, R>      fetchMap(Field<K> key);
<K, V> Map<K, V>      fetchMap(Field<K> key, Field<V> value);
<K, E> Map<K, E>      fetchMap(Field<K> key, Class<E> value);
       Map<Record, R> fetchMap(Field<?>[] key);
<E>    Map<Record, E> fetchMap(Field<?>[] key, Class<E> value);

// Transform your Result object into maps
       List<Map<String, Object>> fetchMaps();
       Map<String, Object>       fetchOneMap();

// Transform your Result object into groups
<K>    Map<K, Result<R>>      fetchGroups(Field<K> key);
<K, V> Map<K, List<V>>        fetchGroups(Field<K> key, Field<V> value);
<K, E> Map<K, List<E>>        fetchGroups(Field<K> key, Class<E> value);
       Map<Record, Result<R>> fetchGroups(Field<?>[] key);
<E>    Map<Record, List<E>>   fetchGroups(Field<?>[] key, Class<E> value);

// Transform your Records into custom POJOs
<E>    List<E> fetchInto(Class<? extends E> type);

// Transform your records into another table type
<Z extends Record> Result<Z> fetchInto(Table<Z> table);
```

Note, that apart from the fetchLazy() methods, all fetch() methods will immediately close underlying JDBC result sets.

5.3.1. Record vs. TableRecord

jOOQ understands that SQL is much more expressive than Java, when it comes to the declarative typing of table expressions. As a declarative language, SQL allows for creating ad-hoc row value expressions (records with indexed columns, or tuples) and records (records with named columns). In Java, this is not possible to the same extent.

Yet, still, sometimes you wish to use strongly typed records, when you know that you're selecting only from a single table:

Fetching strongly or weakly typed records

When fetching data only from a single table, the table expression's type is known to jOOQ if you use jOOQ's code generator to generate TableRecords for your database tables. In order to fetch such strongly typed records, you will have to use the simple select API:

```
// Use the selectFrom() method:
BookRecord book = create.selectFrom(BOOK).where(BOOK.ID.eq(1)).fetchOne();

// Typesafe field access is now possible:
System.out.println("Title       : " + book.getTitle());
System.out.println("Published in: " + book.getPublishedIn());
```

When you use the DSLContext.selectFrom() method, jOOQ will return the record type supplied with the argument table. Beware though, that you will no longer be able to use any clause that modifies the type of your table expression. This includes:

- The SELECT clause
- The JOIN clause

Mapping custom row types to strongly typed records

Sometimes, you may want to explicitly select only a subset of your columns, but still use strongly typed records. Alternatively, you may want to join a one-to-one relationship and receive the two individual strongly typed records after the join.

In both of the above cases, you can map your org.jooq.Record "into" a org.jooq.TableRecord type by using Record.into(Table).

```
// Join two tables
Record record = create.select()
                      .from(BOOK)
                      .join(AUTHOR).on(BOOK.AUTHOR_ID.eq(AUTHOR.ID))
                      .where(BOOK.ID.eq(1))
                      .fetchOne();

// "extract" the two individual strongly typed TableRecord types from the denormalised Record:
BookRecord book = record.into(BOOK);
AuthorRecord author = record.into(AUTHOR);

// Typesafe field access is now possible:
System.out.println("Title       : " + book.getTitle());
System.out.println("Published in: " + book.getPublishedIn());
System.out.println("Author      : " + author.getFirstName() + " " + author.getLastName();
```

5.3.2. Record1 to Record22

jOOQ's row value expression (or tuple) support has been explained earlier in this manual. It is useful for constructing row value expressions where they can be used in SQL. The same typesafety is also applied to records for degrees up to 22. To express this fact, org.jooq.Record is extended by org.jooq.Record1 to org.jooq.Record22. Apart from the fact that these extensions of the R type can be used throughout the jOOQ DSL, they also provide a useful API. Here is org.jooq.Record2, for instance:

```
public interface Record2<T1, T2> extends Record {

    // Access fields and values as row value expressions
    Row2<T1, T2> fieldsRow();
    Row2<T1, T2> valuesRow();

    // Access fields by index
    Field<T1> field1();
    Field<T2> field2();

    // Access values by index
    T1 value1();
    T2 value2();
}
```

Higher-degree records

jOOQ chose to explicitly support degrees up to 22 to match Scala's typesafe tuple, function and product support. Unlike Scala, however, jOOQ also supports higher degrees without the additional typesafety.

5.3.3. Arrays, Maps and Lists

By default, jOOQ returns an org.jooq.Result object, which is essentially a java.util.List of org.jooq.Record. Often, you will find yourself wanting to transform this result object into a type that corresponds more to

your specific needs. Or you just want to list all values of one specific column. Here are some examples to illustrate those use cases:

```
// Fetching only book titles (the two calls are equivalent):
List<String> titles1 = create.select().from(BOOK).fetch().getValues(BOOK.TITLE);
List<String> titles2 = create.select().from(BOOK).fetch(BOOK.TITLE);
String[]     titles3 = create.select().from(BOOK).fetchArray(BOOK.TITLE);

// Fetching only book IDs, converted to Long
List<Long> ids1 = create.select().from(BOOK).fetch().getValues(BOOK.ID, Long.class);
List<Long> ids2 = create.select().from(BOOK).fetch(BOOK.ID, Long.class);
Long[]     ids3 = create.select().from(BOOK).fetchArray(BOOK.ID, Long.class);

// Fetching book IDs and mapping each ID to their records or titles
Map<Integer, BookRecord> map1 = create.selectFrom(BOOK).fetch().intoMap(BOOK.ID);
Map<Integer, BookRecord> map2 = create.selectFrom(BOOK).fetchMap(BOOK.ID);
Map<Integer, String>     map3 = create.selectFrom(BOOK).fetch().intoMap(BOOK.ID, BOOK.TITLE);
Map<Integer, String>     map4 = create.selectFrom(BOOK).fetchMap(BOOK.ID, BOOK.TITLE);

// Group by AUTHOR_ID and list all books written by any author:
Map<Integer, Result<BookRecord>> group1 = create.selectFrom(BOOK).fetch().intoGroups(BOOK.AUTHOR_ID);
Map<Integer, Result<BookRecord>> group2 = create.selectFrom(BOOK).fetchGroups(BOOK.AUTHOR_ID);
Map<Integer, List<String>>       group3 = create.selectFrom(BOOK).fetch().intoGroups(BOOK.AUTHOR_ID, BOOK.TITLE);
Map<Integer, List<String>>       group4 = create.selectFrom(BOOK).fetchGroups(BOOK.AUTHOR_ID, BOOK.TITLE);
```

Note that most of these convenience methods are available both through org.jooq.ResultQuery and org.jooq.Result, some are even available through org.jooq.Record as well.

5.3.4. RecordHandler

In a more functional operating mode, you might want to write callbacks that receive records from your select statement results in order to do some processing. This is a common data access pattern in Spring's JdbcTemplate, and it is also available in jOOQ. With jOOQ, you can implement your own org.jooq.RecordHandler classes and plug them into jOOQ's org.jooq.ResultQuery:

```
// Write callbacks to receive records from select statements
create.selectFrom(BOOK)
      .orderBy(BOOK.ID)
      .fetch()
      .into(new RecordHandler<BookRecord>() {
          @Override
          public void next(BookRecord book) {
              Util.doThingsWithBook(book);
          }
      });

// Or more concisely
create.selectFrom(BOOK)
      .orderBy(BOOK.ID)
      .fetchInto(new RecordHandler<BookRecord>() {...});

// Or even more concisely with Java 8's lambda expressions:
create.selectFrom(BOOK)
      .orderBy(BOOK.ID)
      .fetchInto(book -> { Util.doThingsWithBook(book); }; );
```

See also the manual's section about the RecordMapper, which provides similar features

5.3.5. RecordMapper

In a more functional operating mode, you might want to write callbacks that map records from your select statement results in order to do some processing. This is a common data access pattern in Spring's JdbcTemplate, and it is also available in jOOQ. With jOOQ, you can implement your own org.jooq.RecordMapper classes and plug them into jOOQ's org.jooq.ResultQuery:

```
// Write callbacks to receive records from select statements
List<Integer> ids =
create.selectFrom(BOOK)
      .orderBy(BOOK.ID)
      .fetch()
      .map(BookRecord::getId);

// Or more concisely, as fetch().map(mapper) can be written as fetch(mapper):
create.selectFrom(BOOK)
      .orderBy(BOOK.ID)
      .fetch(BookRecord::getId);

// Or using a lambda expression:
create.selectFrom(BOOK)
      .orderBy(BOOK.ID)
      .fetch(book -> book.getId());

// Of course, the lambda could be expanded into the following anonymous RecordMapper:
create.selectFrom(BOOK)
      .orderBy(BOOK.ID)
      .fetch(new RecordMapper<BookRecord, Integer>() {
          @Override
          public Integer map(BookRecord book) {
              return book.getId();
          }
      });
```

Your custom RecordMapper types can be used automatically through jOOQ's POJO mapping APIs, by injecting a RecordMapperProvider into your Configuration.

See also the manual's section about the RecordHandler, which provides similar features

5.3.6. POJOs

Fetching data in records is fine as long as your application is not really layered, or as long as you're still writing code in the DAO layer. But if you have a more advanced application architecture, you may not want to allow for jOOQ artefacts to leak into other layers. You may choose to write POJOs (Plain Old Java Objects) as your primary DTOs (Data Transfer Objects), without any dependencies on jOOQ's org.jooq.Record types, which may even potentially hold a reference to a Configuration, and thus a JDBC java.sql.Connection. Like Hibernate/JPA, jOOQ allows you to operate with POJOs. Unlike Hibernate/JPA, jOOQ does not "attach" those POJOs or create proxies with any magic in them.

If you're using jOOQ's code generator, you can configure it to generate POJOs for you, but you're not required to use those generated POJOs. You can use your own. See the manual's section about POJOs with custom RecordMappers to see how to modify jOOQ's standard POJO mapping behaviour.

Using JPA-annotated POJOs

jOOQ tries to find JPA annotations on your POJO types. If it finds any, they are used as the primary source for mapping meta-information. Only the javax.persistence.Column annotation is used and understood by jOOQ. An example:

```
// A JPA-annotated POJO class
public class MyBook {
  @Column(name = "ID")
  public int myId;

  @Column(name = "TITLE")
  public String myTitle;
}

// The various "into()" methods allow for fetching records into your custom POJOs:
MyBook myBook        = create.select().from(BOOK).fetchAny().into(MyBook.class);
List<MyBook> myBooks = create.select().from(BOOK).fetch().into(MyBook.class);
List<MyBook> myBooks = create.select().from(BOOK).fetchInto(MyBook.class);
```

Just as with any other JPA implementation, you can put the javax.persistence.Column annotation on any class member, including attributes, setters and getters. Please refer to the Record.into() Javadoc for more details.

Using simple POJOs

If jOOQ does not find any JPA-annotations, columns are mapped to the "best-matching" constructor, attribute or setter. An example illustrates this:

```
// A "mutable" POJO class
public class MyBook1 {
  public int id;
  public String title;
}

// The various "into()" methods allow for fetching records into your custom POJOs:
MyBook1 myBook        = create.select().from(BOOK).fetchAny().into(MyBook1.class);
List<MyBook1> myBooks = create.select().from(BOOK).fetch().into(MyBook1.class);
List<MyBook1> myBooks = create.select().from(BOOK).fetchInto(MyBook1.class);
```

Please refer to the Record.into() Javadoc for more details.

Using "immutable" POJOs

If jOOQ does not find any default constructor, columns are mapped to the "best-matching" constructor. This allows for using "immutable" POJOs with jOOQ. An example illustrates this:

```
// An "immutable" POJO class
public class MyBook2 {
  public final int id;
  public final String title;

  public MyBook2(int id, String title) {
    this.id = id;
    this.title = title;
  }
}

// With "immutable" POJO classes, there must be an exact match between projected fields and available constructors:
MyBook2 myBook        = create.select(BOOK.ID, BOOK.TITLE).from(BOOK).fetchAny().into(MyBook2.class);
List<MyBook2> myBooks = create.select(BOOK.ID, BOOK.TITLE).from(BOOK).fetch().into(MyBook2.class);
List<MyBook2> myBooks = create.select(BOOK.ID, BOOK.TITLE).from(BOOK).fetchInto(MyBook2.class);

// An "immutable" POJO class with a java.beans.ConstructorProperties annotation
public class MyBook3 {
  public final String title;
  public final int id;

  @ConstructorProperties({ "title", "id" })
  public MyBook3(String title, int id) {
    this.title = title;
    this.id = id;
  }
}

// With annotated "immutable" POJO classes, there doesn't need to be an exact match between fields and constructor arguments.
// In the below cases, only BOOK.ID is really set onto the POJO, BOOK.TITLE remains null and BOOK.AUTHOR_ID is ignored
MyBook3 myBook        = create.select(BOOK.ID, BOOK.AUTHOR_ID).from(BOOK).fetchAny().into(MyBook3.class);
List<MyBook3> myBooks = create.select(BOOK.ID, BOOK.AUTHOR_ID).from(BOOK).fetch().into(MyBook3.class);
List<MyBook3> myBooks = create.select(BOOK.ID, BOOK.AUTHOR_ID).from(BOOK).fetchInto(MyBook3.class);
```

Please refer to the Record.into() Javadoc for more details.

Using proxyable types

jOOQ also allows for fetching data into abstract classes or interfaces, or in other words, "proxyable" types. This means that jOOQ will return a java.util.HashMap wrapped in a java.lang.reflect.Proxy implementing your custom type. An example of this is given here:

```
// A "proxyable" type
public interface MyBook3 {
  int getId();
  void setId(int id);

  String getTitle();
  void setTitle(String title);
}

// The various "into()" methods allow for fetching records into your custom POJOs:
MyBook3 myBook        = create.select(BOOK.ID, BOOK.TITLE).from(BOOK).fetchAny().into(MyBook3.class);
List<MyBook3> myBooks = create.select(BOOK.ID, BOOK.TITLE).from(BOOK).fetch().into(MyBook3.class);
List<MyBook3> myBooks = create.select(BOOK.ID, BOOK.TITLE).from(BOOK).fetchInto(MyBook3.class);
```

Please refer to the Record.into() Javadoc for more details.

Loading POJOs back into Records to store them

The above examples show how to fetch data into your own custom POJOs / DTOs. When you have modified the data contained in POJOs, you probably want to store those modifications back to the database. An example of this is given here:

```
// A "mutable" POJO class
public class MyBook {
  public int id;
  public String title;
}

// Create a new POJO instance
MyBook myBook = new MyBook();
myBook.id = 10;
myBook.title = "Animal Farm";

// Load a jOOQ-generated BookRecord from your POJO
BookRecord book = create.newRecord(BOOK, myBook);

// Insert it (implicitly)
book.store();

// Insert it (explicitly)
create.executeInsert(book);

// or update it (ID = 10)
create.executeUpdate(book);
```

Note: Because of your manual setting of ID = 10, jOOQ's store() method will asume that you want to insert a new record. See the manual's section about CRUD with UpdatableRecords for more details on this.

Interaction with DAOs

If you're using jOOQ's code generator, you can configure it to generate DAOs for you. Those DAOs operate on generated POJOs. An example of using such a DAO is given here:

```
// Initialise a Configuration
Configuration configuration = new DefaultConfiguration().set(connection).set(SQLDialect.ORACLE);

// Initialise the DAO with the Configuration
BookDao bookDao = new BookDao(configuration);

// Start using the DAO
Book book = bookDao.findById(5);

// Modify and update the POJO
book.setTitle("1984");
book.setPublishedIn(1948);
bookDao.update(book);

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

More complex data structures

jOOQ currently doesn't support more complex data structures, the way Hibernate/JPA attempt to map relational data onto POJOs. While future developments in this direction are not excluded, jOOQ claims that generic mapping strategies lead to an enormous additional complexity that only serves very few use cases. You are likely to find a solution using any of jOOQ's various fetching modes, with only little boiler-plate code on the client side.

5.3.7. POJOs with RecordMappers

In the previous sections we have seen how to create RecordMapper types to map jOOQ records onto arbitrary objects. We have also seen how jOOQ provides default algorithms to map jOOQ records onto POJOs. Your own custom domain model might be much more complex, but you want to avoid looking up the most appropriate RecordMapper every time you need one. For this, you can provide jOOQ's Configuration with your own implementation of the org.jooq.RecordMapperProvider interface. An example is given here:

```
DSL.using(new DefaultConfiguration()
   .set(connection)
   .set(SQLDialect.ORACLE)
   .set(
       new RecordMapperProvider() {
           @Override
           public <R extends Record, E> RecordMapper<R, E> provide(RecordType<R> recordType, Class<? extends E> type) {

               // UUID mappers will always try to find the ID column
               if (type == UUID.class) {
                   return new RecordMapper<R, E>() {
                       @Override
                       public E map(R record) {
                           return (E) record.getValue("ID");
                       }
                   }
               }

               // Books might be joined with their authors, create a 1:1 mapping
               if (type == Book.class) {
                   return new BookMapper();
               }

               // Fall back to jOOQ's DefaultRecordMapper, which maps records onto
               // POJOs using reflection.
               return new DefaultRecordMapper(recordType, type);
           }
       }
   ))
   .selectFrom(BOOK)
   .orderBy(BOOK.ID)
   .fetchInto(UUID.class);
```

The above is a very simple example showing that you will have complete flexibility in how to override jOOQ's record to POJO mapping mechanisms.

Using third party libraries

A couple of useful libraries exist out there, which implement custom, more generic mapping algorithms. Some of them have been specifically made to work with jOOQ. Among them are:

- ModelMapper (with an explicit jOOQ integration)
- SimpleFlatMapper (with an explicit jOOQ integration)
- Orika Mapper (without explicit jOOQ integration)

5.3.8. Lazy fetching

Unlike JDBC's java.sql.ResultSet, jOOQ's org.jooq.Result does not represent an open database cursor with various fetch modes and scroll modes, that needs to be closed after usage. jOOQ's results are simple in-memory Java java.util.List objects, containing all of the result values. If your result sets are large, or if you have a lot of network latency, you may wish to fetch records one-by-one, or in small chunks. jOOQ supports a org.jooq.Cursor type for that purpose. In order to obtain such a reference, use the ResultQuery.fetchLazy() method. An example is given here:

```
// Obtain a Cursor reference:
try (Cursor<BookRecord> cursor = create.selectFrom(BOOK).fetchLazy()) {

    // Cursor has similar methods as Iterator<R>
    while (cursor.hasNext()) {
        BookRecord book = cursor.fetchOne();

        Util.doThingsWithBook(book);
    }
}
```

As a org.jooq.Cursor holds an internal reference to an open java.sql.ResultSet, it may need to be closed at the end of iteration. If a cursor is completely scrolled through, it will conveniently close the underlying ResultSet. However, you should not rely on that.

Cursors ship with all the other fetch features

Like org.jooq.ResultQuery or org.jooq.Result, org.jooq.Cursor gives access to all of the other fetch features that we've seen so far, i.e.

- Strongly or weakly typed records: Cursors are also typed with the <R> type, allowing to fetch custom, generated org.jooq.TableRecord or plain org.jooq.Record types.
- RecordHandler callbacks: You can use your own org.jooq.RecordHandler callbacks to receive lazily fetched records.
- RecordMapper callbacks: You can use your own org.jooq.RecordMapper callbacks to map lazily fetched records.
- POJOs: You can fetch data into your own custom POJO types.

5.3.9. Lazy fetching with Streams

jOOQ 3.7+ supports Java 8, and with Java 8, it supports java.util.stream.Stream. This opens up a range of possibilities of combining the declarative aspects of SQL with the functional aspects of the new Stream API. Much like the Cursors from the previous section, such a Stream keeps an internal reference to a JDBC java.sql.ResultSet, which means that the Stream has to be treated like a resource. Here's an example of using such a stream:

```
// Obtain a Stream reference:
try (Stream<BookRecord> stream = create.selectFrom(BOOK).stream()) {
    stream.forEach(Util::doThingsWithBook);
}
```

A more sophisticated example would be using streams to transform the results and add business logic to it. For instance, to generate a DDL script with CREATE TABLE statements from the INFORMATION_SCHEMA of an H2 database:

```
create.select(
          COLUMNS.TABLE_NAME,
          COLUMNS.COLUMN_NAME,
          COLUMNS.TYPE_NAME)
      .from(COLUMNS)
      .orderBy(
          COLUMNS.TABLE_CATALOG,
          COLUMNS.TABLE_SCHEMA,
          COLUMNS.TABLE_NAME,
          COLUMNS.ORDINAL_POSITION)
      .fetch() // Eagerly load the whole ResultSet into memory first
      .stream()
      .collect(groupingBy(
          r -> r.getValue(COLUMNS.TABLE_NAME),
          LinkedHashMap::new,
          mapping(
              r -> new SimpleEntry(
                  r.getValue(COLUMNS.COLUMN_NAME),
                  r.getValue(COLUMNS.TYPE_NAME)
              ),
              toList()
          )))
      .forEach(
          (table, columns) -> {
              // Just emit a CREATE TABLE statement
              System.out.println("CREATE TABLE " + table + " (");

              // Map each "Column" type into a String containing the column specification,
              // and join them using comma and newline. Done!
              System.out.println(
                  columns.stream()
                         .map(col -> "  " + col.getKey() +
                                     " " + col.getValue())
                         .collect(Collectors.joining(",\n"))
              );

              System.out.println(");");
          });
```

The above combination of SQL and functional programming will produce the following output:

```

CREATE TABLE CATALOGS(
  CATALOG_NAME VARCHAR
);
CREATE TABLE COLLATIONS(
  NAME VARCHAR,
  KEY VARCHAR
);
CREATE TABLE COLUMNS(
  TABLE_CATALOG VARCHAR,
  TABLE_SCHEMA VARCHAR,
  TABLE_NAME VARCHAR,
  COLUMN_NAME VARCHAR,
  ORDINAL_POSITION INTEGER,
  COLUMN_DEFAULT VARCHAR,
  IS_NULLABLE VARCHAR,
  DATA_TYPE INTEGER,
  CHARACTER_MAXIMUM_LENGTH INTEGER,
  CHARACTER_OCTET_LENGTH INTEGER,
  NUMERIC_PRECISION INTEGER,
  NUMERIC_PRECISION_RADIX INTEGER,
  NUMERIC_SCALE INTEGER,
  CHARACTER_SET_NAME VARCHAR,
  COLLATION_NAME VARCHAR,
  TYPE_NAME VARCHAR,
  NULLABLE INTEGER,
  IS_COMPUTED BOOLEAN,
  SELECTIVITY INTEGER,
  CHECK_CONSTRAINT VARCHAR,
  SEQUENCE_NAME VARCHAR,
  REMARKS VARCHAR,
  SOURCE_DATA_TYPE SMALLINT
);
```

5.3.10. Many fetching

Many databases support returning several result sets, or cursors, from single queries. An example for this is Sybase ASE's sp_help command:

```
> sp_help 'author'

+--------+-----+-----------+-------------+-------------------+
|Name    |Owner|Object_type|Object_status|Create_date        |
+--------+-----+-----------+-------------+-------------------+
|  author|dbo  |user table | -- none --  |Sep 22 2011 11:20PM|
+--------+-----+-----------+-------------+-------------------+

+-------------+-------+------+----+-----+-----+
|Column_name  |Type   |Length|Prec|Scale|...  |
+-------------+-------+------+----+-----+-----+
|id           |int    |     4|NULL| NULL|    0|
|first_name   |varchar|    50|NULL| NULL|    1|
|last_name    |varchar|    50|NULL| NULL|    0|
|date_of_birth|date   |     4|NULL| NULL|    1|
|year_of_birth|int    |     4|NULL| NULL|    1|
+-------------+-------+------+----+-----+-----+
```

The correct (and verbose) way to do this with JDBC is as follows:

```
ResultSet rs = statement.executeQuery();

// Repeat until there are no more result sets
for (;;) {

  // Empty the current result set
  while (rs.next()) {
    // [ .. do something with it .. ]
  }

  // Get the next result set, if available
  if (statement.getMoreResults()) {
    rs = statement.getResultSet();
  }
  else {
    break;
  }
}

// Be sure that all result sets are closed
statement.getMoreResults(Statement.CLOSE_ALL_RESULTS);
statement.close();
```

As previously discussed in the chapter about differences between jOOQ and JDBC, jOOQ does not rely on an internal state of any JDBC object, which is "externalised" by Javadoc. Instead, it has a straight-forward API allowing you to do the above in a one-liner:

```
// Get some information about the author table, its columns, keys, indexes, etc
Results results = create.fetchMany("sp_help 'author'");
```

The returned org.jooq.Results type extends the List<Result<Record>> type for backwards-compatibility reasons, but it also allows to access individual update counts that may have been returned by the database in between result sets.

5.3.11. Later fetching

Using Java 8 CompletableFutures

Java 8 has introduced the new java.util.concurrent.CompletableFuture type, which allows for functional composition of asynchronous execution units. When applying this to SQL and jOOQ, you might be writing code as follows:

```
// Initiate an asynchronous call chain
CompletableFuture

    // This lambda will supply an int value indicating the number of inserted rows
    .supplyAsync(() ->
        DSL.using(configuration)
           .insertInto(AUTHOR, AUTHOR.ID, AUTHOR.LAST_NAME)
           .values(3, "Hitchcock")
           .execute()
    )

    // This will supply an AuthorRecord value for the newly inserted author
    .handleAsync((rows, throwable) ->
        DSL.using(configuration)
           .fetchOne(AUTHOR, AUTHOR.ID.eq(3))
    )

    // This should supply an int value indicating the number of rows,
    // but in fact it'll throw a constraint violation exception
    .handleAsync((record, throwable) -> {
        record.changed(true);
        return record.insert();
    })

    // This will supply an int value indicating the number of deleted rows
    .handleAsync((rows, throwable) ->
        DSL.using(configuration)
           .delete(AUTHOR)
           .where(AUTHOR.ID.eq(3))
           .execute()
    )
    .join();
```

The above example will execute four actions one after the other, but asynchronously in the JDK's default or common java.util.concurrent.ForkJoinPool.

For more information, please refer to the java.util.concurrent.CompletableFuture Javadoc and official documentation.

Using deprecated API

Some queries take very long to execute, yet they are not crucial for the continuation of the main program. For instance, you could be generating a complicated report in a Swing application, and while this report is being calculated in your database, you want to display a background progress bar, allowing the user to pursue some other work. This can be achived simply with jOOQ, by creating a org.jooq.FutureResult, a type that extends java.util.concurrent.Future. An example is given here:

```
// Spawn off this query in a separate process:
FutureResult<BookRecord> future = create.selectFrom(BOOK).where(... complex predicates ...).fetchLater();

// This example actively waits for the result to be done
while (!future.isDone()) {
    progressBar.increment(1);
    Thread.sleep(50);
}

// The result should be ready, now
Result<BookRecord> result = future.get();
```

Note, that instead of letting jOOQ spawn a new thread, you can also provide jOOQ with your own java.util.concurrent.ExecutorService:

```
// Spawn off this query in a separate process:
ExecutorService service = // [...]
FutureResult<BookRecord> future = create.selectFrom(BOOK).where(... complex predicates ...).fetchLater(service);
```

5.3.12. ResultSet fetching

When interacting with legacy applications, you may prefer to have jOOQ return a java.sql.ResultSet, rather than jOOQ's own org.jooq.Result types. This can be done simply, in two ways:

```
try (

    // jOOQ's Cursor type exposes the underlying ResultSet:
    ResultSet rs1 = create.selectFrom(BOOK).fetchLazy().resultSet();

    // But you can also directly access that ResultSet from ResultQuery:
    ResultSet rs2 = create.selectFrom(BOOK).fetchResultSet()) {

// ...
}
```

Transform jOOQ's Result into a JDBC ResultSet

Instead of operating on a JDBC ResultSet holding an open resource from your database, you can also let jOOQ's org.jooq.Result wrap itself in a java.sql.ResultSet. The advantage of this is that the so-created ResultSet has no open connection to the database. It is a completely in-memory ResultSet:

```
// Transform a jOOQ Result into a ResultSet
Result<BookRecord> result = create.selectFrom(BOOK).fetch();
ResultSet rs = result.intoResultSet();
```

The inverse: Fetch data from a legacy ResultSet using jOOQ

The inverse of the above is possible too. Maybe, a legacy part of your application produces JDBC java.sql.ResultSet, and you want to turn them into a org.jooq.Result:

```
// Transform a JDBC ResultSet into a jOOQ Result
ResultSet rs = connection.createStatement().executeQuery("SELECT * FROM BOOK");

// As a Result:
Result<Record> result = create.fetch(rs);

// As a Cursor
Cursor<Record> cursor = create.fetchLazy(rs);
```

You can also tighten the interaction with jOOQ's data type system and data type conversion features, by passing the record type to the above fetch methods:

```
// Pass an array of types:
Result<Record> result = create.fetch    (rs, Integer.class, String.class);
Cursor<Record> result = create.fetchLazy(rs, Integer.class, String.class);

// Pass an array of data types:
Result<Record> result = create.fetch    (rs, SQLDataType.INTEGER, SQLDataType.VARCHAR);
Cursor<Record> result = create.fetchLazy(rs, SQLDataType.INTEGER, SQLDataType.VARCHAR);

// Pass an array of fields:
Result<Record> result = create.fetch    (rs, BOOK.ID, BOOK.TITLE);
Cursor<Record> result = create.fetchLazy(rs, BOOK.ID, BOOK.TITLE);
```

If supplied, the additional information is used to override the information obtained from the ResultSet's java.sql.ResultSetMetaData information.

5.3.13. Auto data type conversion

Many native SQL data types can be automatically converted from one another, such as VARCHAR to INTEGER and vice versa.

The jOOQ API also supports a variety of such auto conversions through the org.jooq.tools.Convert utility API, which implements the following rules:

- null is always converted to null, or the primitive default value, or Optional.empty(), regardless of the target type.
- Identity conversion (converting a value to its own type) is always possible.
- Primitive types can be converted to their wrapper types and vice versa
- All types can be converted to String
- All types can be converted to Object
- All Number types can be converted to other Number types
- All Number or String types can be converte to Boolean. Possible (case-insensitive) values for true:

  * 1
  * 1.0
  * y
  * yes
  * true
  * on
  * enabled

  Possible (case-insensitive) values for false:

  * 0
  * 0.0
  * n
  * no
  * false
  * off
  * disabled

  All other values evaluate to null
- All java.util.Date subtypes (java.sql.Date, java.sql.Time, java.sql.Timestamp), as well as most java.time.temporal.Temporal subtypes (java.time.LocalDate, java.time.LocalTime, java.time.LocalDateTime, java.time.OffsetTime, java.time.OffsetDateTime, as well as java.time.Instant) can be converted into each other.
- byte[] can be converted into String, using the platform's default charset
- Object[] can be converted into any other array type, if array elements can be converted, too

This auto conversion can be applied explicitly, but is also available through a variety of API, in particular anywhere a java.lang.Class reference can be provided, such as:

```
Record record = ...
int i = record.get(0, int.class);
String s = record.get(1, String.class);
```

5.3.14. Custom data type conversion

Apart from a few extra features (user-defined types), jOOQ only supports basic types as supported by the JDBC API. In your application, you may choose to transform these data types into your own ones, without writing too much boiler-plate code. This can be done using jOOQ's org.jooq.Converter types. A converter essentially allows for two-way conversion between two Java data types <T> and <U>. By convention, the <T> type corresponds to the type in your database whereas the <U> type corresponds to your own user type. The Converter API is given here:

```
public interface Converter<T, U> extends Serializable {

    /**
     * Convert a database object to a user object
     */
    U from(T databaseObject);

    /**
     * Convert a user object to a database object
     */
    T to(U userObject);

    /**
     * The database type
     */
    Class<T> fromType();

    /**
     * The user type
     */
    Class<U> toType();
}
```

Such a converter can be used in many parts of the jOOQ API. Some examples have been illustrated in the manual's section about fetching.

A Converter for GregorianCalendar

Here is a some more elaborate example involving a Converter for java.util.GregorianCalendar:

```
// You may prefer Java Calendars over JDBC Timestamps
public class CalendarConverter implements Converter<Timestamp, GregorianCalendar> {

    @Override
    public GregorianCalendar from(Timestamp databaseObject) {
        GregorianCalendar calendar = (GregorianCalendar) Calendar.getInstance();
        calendar.setTimeInMillis(databaseObject.getTime());
        return calendar;
    }

    @Override
    public Timestamp to(GregorianCalendar userObject) {
        return new Timestamp(userObject.getTime().getTime());
    }

    @Override
    public Class<Timestamp> fromType() {
        return Timestamp.class;
    }

    @Override
    public Class<GregorianCalendar> toType() {
        return GregorianCalendar.class;
    }
}

// Now you can fetch calendar values from jOOQ's API:
List<GregorianCalendar> dates1 = create.selectFrom(BOOK).fetch().getValues(BOOK.PUBLISHING_DATE, new CalendarConverter());
List<GregorianCalendar> dates2 = create.selectFrom(BOOK).fetch(BOOK.PUBLISHING_DATE, new CalendarConverter());
```

Enum Converters

jOOQ ships with a built-in default org.jooq.impl.EnumConverter, that you can use to map VARCHAR values to enum literals or NUMBER values to enum ordinals (both modes are supported). Let's say, you want to map a YES / NO / MAYBE column to a custom Enum:

```
// Define your Enum
public enum YNM {
    YES, NO, MAYBE
}

// Define your converter
public class YNMConverter extends EnumConverter<String, YNM> {
    public YNMConverter() {
        super(String.class, YNM.class);
    }
}

// And you're all set for converting records to your custom Enum:
for (BookRecord book : create.selectFrom(BOOK).fetch()) {
    switch (book.getValue(BOOK.I_LIKE, new YNMConverter())) {
        case YES:    System.out.println("I like this book             : " + book.getTitle()); break;
        case NO:     System.out.println("I didn't like this book      : " + book.getTitle()); break;
        case MAYBE:  System.out.println("I'm not sure about this book : " + book.getTitle()); break;
    }
}
```

If you're using forcedTypes in your code generation configuration, you can configure the application of an EnumConverter by adding <enumConverter>true</enumConverter> to your <forcedType/> configuration.

Using Converters in generated source code

jOOQ also allows for generated source code to reference your own custom converters, in order to permanently replace a table column's <T> type by your own, custom <U> type. See the manual's section about custom data types for details.

5.3.15. Interning data

SQL result tables are not optimal in terms of used memory as they are not designed to represent hierarchical data as produced by JOIN operations. Specifically, FOREIGN KEY values may repeat themselves unnecessarily:

```
+----+-----------+--------------+
| ID | AUTHOR_ID | TITLE        |
+----+-----------+--------------+
|  1 |         1 | 1984         |
|  2 |         1 | Animal Farm  |
|  3 |         2 | O Alquimista |
|  4 |         2 | Brida        |
+----+-----------+--------------+
```

Now, if you have millions of records with only few distinct values for AUTHOR_ID, you may not want to hold references to distinct (but equal) java.lang.Integer objects. This is specifically true for IDs of type java.util.UUID or string representations thereof. jOOQ allows you to "intern" those values:

```
// Interning data after fetching
Result<?> r1 = create.select(BOOK.ID, BOOK.AUTHOR_ID, BOOK.TITLE)
                     .from(BOOK)
                     .join(AUTHOR).on(BOOK.AUTHOR_ID.eq(AUTHOR.ID))
                     .fetch()
                     .intern(BOOK.AUTHOR_ID);

// Interning data while fetching
Result<?> r1 = create.select(BOOK.ID, BOOK.AUTHOR_ID, BOOK.TITLE)
                     .from(BOOK)
                     .join(AUTHOR).on(BOOK.AUTHOR_ID.eq(AUTHOR.ID))
                     .intern(BOOK.AUTHOR_ID)
                     .fetch();
```

You can specify as many fields as you want for interning. The above has the following effect:

- If the interned Field is of type java.lang.String, then String.intern() is called upon each string
- If the interned Field is of any other type, then the call is ignored

Future versions of jOOQ will implement interning of data for non-String data types by collecting values in java.util.Set, removing duplicate instances.

Note, that jOOQ will not use interned data for identity comparisons: string1 == string2. Interning is used only to reduce the memory footprint of org.jooq.Result objects.

5.4. Static statements vs. Prepared Statements

With JDBC, you have full control over your SQL statements. You can decide yourself, if you want to execute a static java.sql.Statement without bind values, or a java.sql.PreparedStatement with (or without) bind values. But you have to decide early, which way to go. And you'll have to prevent SQL injection and syntax errors manually, when inlining your bind variables.

With jOOQ, this is easier. As a matter of fact, it is plain simple. With jOOQ, you can just set a flag in your Configuration's Settings, and all queries produced by that configuration will be executed as static statements, with all bind values inlined. An example is given here:

```
-- These statements are rendered by the two factories:
SELECT ? FROM DUAL WHERE ? = ?
SELECT 1 FROM DUAL WHERE 1 = 1
```

```
// This DSLContext executes PreparedStatements
DSLContext prepare = DSL.using(connection, SQLDialect.ORACLE);

// This DSLContext executes static Statements
DSLContext inlined = DSL.using(connection, SQLDialect.ORACLE,
  new
 Settings().withStatementType(StatementType.STATIC_STATEMENT));

prepare.select(val(1)).where(val(1).eq(1)).fetch();
inlined.select(val(1)).where(val(1).eq(1)).fetch();
```

Reasons for choosing one or the other

Not all databases are equal. Some databases show improved performance if you use java.sql.PreparedStatement, as the database will then be able to re-use execution plans for identical SQL statements, regardless of actual bind values. This heavily improves the time it takes for soft-parsing a SQL statement. In other situations, assuming that bind values are irrelevant for SQL execution plans may be a bad idea, as you might run into "bind value peeking" issues. You may be better off spending the extra cost for a new hard-parse of your SQL statement and instead having the database fine-tune the new plan to the concrete bind values.

Whichever aproach is more optimal for you cannot be decided by jOOQ. In most cases, prepared statements are probably better. But you always have the option of forcing jOOQ to render inlined bind values.

Inlining bind values on a per-bind-value basis

Note that you don't have to inline all your bind values at once. If you know that a bind value is not really a variable and should be inlined explicitly, you can do so by using DSL.inline(), as documented in the manual's section about inlined parameters

5.5. Reusing a Query's PreparedStatement

As previously discussed in the chapter about differences between jOOQ and JDBC, reusing PreparedStatements is handled a bit differently in jOOQ from how it is handled in JDBC

Keeping open PreparedStatements with JDBC

With JDBC, you can easily reuse a java.sql.PreparedStatement by not closing it between subsequent executions. An example is given here:

```
// Execute the statement
try (PreparedStatement stmt = connection.prepareStatement("SELECT 1 FROM DUAL")) {

    // Fetch a first ResultSet
    try (ResultSet rs1 = stmt.executeQuery()) { ... }

    // Without closing the statement, execute it again to fetch another ResultSet
    try (ResultSet rs2 = stmt.executeQuery()) { ... }
}
```

The above technique can be quite useful when you want to reuse expensive database resources. This can be the case when your statement is executed very frequently and your database would take non-negligible time to soft-parse the prepared statement and generate a new statement / cursor resource.

Keeping open PreparedStatements with jOOQ

This is also modeled in jOOQ. However, the difference to JDBC is that closing a statement is the default action, whereas keeping it open has to be configured explicitly. This is better than JDBC, because the default action should be the one that is used most often. Keeping open statements is rarely done in average applications. Here's an example of how to keep open PreparedStatements with jOOQ:

```
// Create a query which is configured to keep its underlying PreparedStatement open
try (ResultQuery<Record> query = create.selectOne().keepStatement(true)) {
    Result<Record> result1 = query.fetch(); // This will lazily create a new PreparedStatement
    Result<Record> result2 = query.fetch(); // This will reuse the previous PreparedStatement
}
```

The above example shows how a query can be executed twice against the same underlying PreparedStatement. Notice how the Query must now be treated like a resource, i.e. it must be managed in a try-with-resources statement, or Query.close() must be called explicitly.

5.6. JDBC flags

JDBC knows a couple of execution flags and modes, which can be set through the jOOQ API as well. jOOQ essentially supports these flags and execution modes:

```
public interface Query extends QueryPart, Attachable {

    // [...]

    // The query execution timeout.
    // ------------------------------------------------------------
    Query queryTimeout(int timeout);

}
```

```
public interface ResultQuery<R extends Record> extends Query {

    // [...]

    // The query execution timeout.
    // ------------------------------------------------------------
    @Override
    ResultQuery<R> queryTimeout(int timeout);

    // Flags allowing to specify the resulting ResultSet modes
    // ------------------------------------------------------------
    ResultQuery<R> resultSetConcurrency(int resultSetConcurrency);
    ResultQuery<R> resultSetType(int resultSetType);
    ResultQuery<R> resultSetHoldability(int resultSetHoldability);

    // The buffer size for JDBC cursors
    // ------------------------------------------------------------
    ResultQuery<R> fetchSize(int size);

    // The maximum number of rows to be fetched by JDBC
    // ------------------------------------------------------------
    ResultQuery<R> maxRows(int rows);

}
```

Using ResultSet concurrency with ExecuteListeners

An example of why you might want to manually set a ResultSet's concurrency flag to something non-default is given here:

```
DSL.using(new DefaultConfiguration()
   .set(connection)
   .set(SQLDialect.ORACLE)
   .set(DefaultExecuteListenerProvider.providers(
           new DefaultExecuteListener() {

               @Override
               public void recordStart(ExecuteContext ctx) {
                   try {

                       // Change values in the cursor before reading a record
                       ctx.resultSet().updateString(BOOK.TITLE.getName(), "New Title");
                       ctx.resultSet().updateRow();
                   }
                   catch (SQLException e) {
                       throw new DataAccessException("Exception", e);
                   }
               }
           }
       )
   ))
   .select(BOOK.ID, BOOK.TITLE)
   .from(BOOK)
   .orderBy(BOOK.ID)
   .resultSetType(ResultSet.TYPE_SCROLL_INSENSITIVE)
   .resultSetConcurrency(ResultSet.CONCUR_UPDATABLE)
   .fetch(BOOK.TITLE);
```

In the above example, your custom ExecuteListener callback is triggered before jOOQ loads a new Record from the JDBC ResultSet. With the concurrency being set to ResultSet.CONCUR_UPDATABLE, you can now modify the database cursor through the standard JDBC ResultSet API.

5.7. Using JDBC batch operations

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

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

Using JDBC

In code, this looks like the following snippet:

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

// 2. a single query
// -----------------
try (PreparedStatement stmt = connection.prepareStatement("INSERT INTO author(id, first_name, last_name) VALUES (?, ?, ?)")) {
    stmt.setInt(1, 1);
    stmt.setString(2, "Erich");
    stmt.setString(3, "Gamma");
    stmt.addBatch();

    stmt.setInt(1, 2);
    stmt.setString(2, "Richard");
    stmt.setString(3, "Helm");
    stmt.addBatch();

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

    stmt.setInt(1, 4);
    stmt.setString(2, "John");
    stmt.setString(3, "Vlissides");
    stmt.addBatch();

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

Using jOOQ

jOOQ supports executing queries in batch mode as follows:

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

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

When creating a batch execution with a single query and multiple bind values, you will still have to provide jOOQ with dummy bind values for the original query. In the above example, these are set to null. For subsequent calls to bind(), there will be no type safety provided by jOOQ.

5.8. Sequence execution

Most databases support sequences of some sort, to provide you with unique values to be used for primary keys and other enumerations. If you're using jOOQ's code generator, it will generate a sequence object per sequence for you. There are two ways of using such a sequence object:

Standalone calls to sequences

Instead of actually phrasing a select statement, you can also use the DSLContext's convenience methods:

```
// Fetch the next value from a sequence
BigInteger nextID = create.nextval(S_AUTHOR_ID);

// Fetch the current value from a sequence
BigInteger currID = create.currval(S_AUTHOR_ID);
```

Inlining sequence references in SQL

You can inline sequence references in jOOQ SQL statements. The following are examples of how to do that:

```
// Reference the sequence in a SELECT statement:
Field<BigInteger> s = S_AUTHOR_ID.nextval();
BigInteger nextID = create.select(s).fetchOne(s);

// Reference the sequence in an INSERT statement:
create.insertInto(AUTHOR, AUTHOR.ID, AUTHOR.FIRST_NAME, AUTHOR.LAST_NAME)
      .values(S_AUTHOR_ID.nextval(), val("William"), val("Shakespeare"))
      .execute();
```

For more info about inlining sequence references in SQL statements, please refer to the manual's section about sequences and serials.

5.9. Stored procedures and functions

Many RDBMS support the concept of "routines", usually calling them procedures and/or functions. These concepts have been around in programming languages for a while, also outside of databases. Famous languages distinguishing procedures from functions are:

- Ada
- BASIC
- Pascal
- etc...

The general distinction between (stored) procedures and (stored) functions can be summarised like this:

Procedures

- Are called using JDBC CallableStatement
- Have no return value
- Usually support OUT parameters

Functions

- Can be used in SQL statements
- Have a return value
- Usually don't support OUT parameters

Exceptions to these rules

- DB2, H2, and HSQLDB don't allow for JDBC escape syntax when calling functions. Functions must be used in a SELECT statement
- H2 only knows functions (without OUT parameters)
- Oracle functions may have OUT parameters
- Oracle knows functions that must not be used in SQL statements for transactional reasons
- Postgres only knows functions (with all features combined). OUT parameters can also be interpreted as return values, which is quite elegant/surprising, depending on your taste
- The Sybase jconn3 JDBC driver doesn't handle null values correctly when using the JDBC escape syntax on functions

In general, it can be said that the field of routines (procedures / functions) is far from being standardised in modern RDBMS even if the SQL:2008 standard specifies things quite well. Every database has its ways and JDBC only provides little abstraction over the great variety of procedures / functions implementations, especially when advanced data types such as cursors / UDT's / arrays are involved.

To simplify things a little bit, jOOQ handles both procedures and functions the same way, using a more general org.jooq.Routine type.

Using jOOQ for standalone calls to stored procedures and functions

If you're using jOOQ's code generator, it will generate org.jooq.Routine objects for you. Let's consider the following example:

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

The generated artefacts can then be used as follows:

```
// Make an explicit call to the generated procedure object:
AuthorExists procedure = new AuthorExists();

// All IN and IN OUT parameters generate setters
procedure.setAuthorName("Paulo");
procedure.execute(configuration);

// All OUT and IN OUT parameters generate getters
assertEquals(new BigDecimal("1"), procedure.getResult());
assertEquals(new BigDecimal("2"), procedure.getId();
```

But you can also call the procedure using a generated convenience method in a global Routines class:

```
// The generated Routines class contains static methods for every procedure.
// Results are also returned in a generated object, holding getters for every OUT or IN OUT parameter.
AuthorExists procedure = Routines.authorExists(configuration, "Paulo");

// All OUT and IN OUT parameters generate getters
assertEquals(new BigDecimal("1"), procedure.getResult());
assertEquals(new BigDecimal("2"), procedure.getId();
```

For more details about code generation for procedures, see the manual's section about procedures and code generation.

Inlining stored function references in SQL

Unlike procedures, functions can be inlined in SQL statements to generate column expressions or table expressions, if you're using unnesting operators. Assume you have a function like this:

```
-- Check whether there is an author in AUTHOR by that name and get his ID
CREATE OR REPLACE FUNCTION author_exists (author_name VARCHAR2) RETURN NUMBER;
```

The generated artefacts can then be used as follows:

```
-- This is the rendered SQL

SELECT AUTHOR_EXISTS('Paulo') FROM DUAL
```

```
// Use the static-imported method from Routines:
boolean exists =
create.select(authorExists("Paulo")).fetchOne(0, boolean.class);
```

For more info about inlining stored function references in SQL statements, please refer to the manual's section about user-defined functions.

5.9.1. Oracle Packages

Oracle uses the concept of a PACKAGE to group several procedures/functions into a sort of namespace. The SQL 92 standard talks about "modules", to represent this concept, even if this is rarely implemented as such. 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.

For more details about code generation for procedures and packages see the manual's section about procedures and code generation.

5.9.2. Oracle member procedures

Oracle UDTs 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 counBOOKs 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 DSLContext
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 code generation for UDTs see the manual's section about user-defined types and code generation.

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

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 the importing functionality). You can export any Result<Record> into the formats discussed in the subsequent chapters of the manual

5.10.1. Exporting XML

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

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

```
<result xmlns="http://www.jooq.org/xsd/jooq-export-3.10.0.xsd">
  <fields>
    <field schema="TEST" table="BOOK" name="ID" type="INTEGER"/>
    <field schema="TEST" table="BOOK" name="AUTHOR_ID" type="INTEGER"/>
    <field schema="TEST" table="BOOK" name="TITLE" type="VARCHAR"/>
  </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>
</result>
```

The same result as an org.w3c.dom.Document can be obtained using the Result.intoXML() method:

```
// Fetch books and format them as XML
Document xml = create.selectFrom(BOOK).fetch().intoXML();
```

See the XSD schema definition here, for a formal definition of the XML export format:
http://www.jooq.org/xsd/jooq-export-3.10.0.xsd

5.10.2. Exporting CSV

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

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

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

In addition to the standard behaviour, you can also specify a separator character, as well as a special string to represent NULL values (which cannot be represented in standard CSV):

```
// Use ";" as the separator character
String csv = create.selectFrom(BOOK).fetch().formatCSV(';');

// Specify "{null}" as a representation for NULL values
String csv = create.selectFrom(BOOK).fetch().formatCSV(';', "{null}");
```

5.10.3. Exporting JSON

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

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

```
{"fields":[{"schema":"schema-1","table":"table-1","name":"field-1","type":"type-1"},
           {"schema":"schema-2","table":"table-2","name":"field-2","type":"type-2"},
           ...,
           {"schema":"schema-n","table":"table-n","name":"field-n","type":"type-n"}],
 "records":[[value-1-1,value-1-2,...,value-1-n],
            [value-2-1,value-2-2,...,value-2-n]]}
```

Note: This format has been modified in jOOQ 2.6.0 and 3.7.0

5.10.4. Exporting HTML

```
// Fetch books and format them as HTML
String html = create.selectFrom(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>
```

5.10.5. Exporting Text

```
// Fetch books and format them as text
String text = create.selectFrom(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|
+---+---------+-----------+
```

A simple text representation can also be obtained by calling toString() on a Result object. See also the manual's section about DEBUG logging

5.11. Importing data

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 the formats described in the subsequent sections of this manual.

5.11.1. Importing 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 <-- Note the CSV header. By default, the first line is ignored
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:

```
DSLContext create = DSL.using(connection, dialect);

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

Here are various other examples:

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

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

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

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

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

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

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

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

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

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

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<Author> 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
DataAccessException 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();
```

5.11.2. Importing JSON

The below JSON data represents two author records that may have been exported previously, by jOOQ's exporting functionality:

```
{"fields" :[{"name":"ID","type":"INTEGER"},
            {"name":"AUTHOR_ID","type":"INTEGER"},
            {"name":"TITLE","type":"VARCHAR"}],
 "records":[[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:

```
DSLContext create = DSL.using(connection, dialect);

// Load data into the BOOK table from an input stream
// holding the JSON data.
create.loadInto(BOOK)
      .loadJSON(inputstream, encoding)
      .fields(BOOK.ID, BOOK.AUTHOR_ID, BOOK.TITLE)
      .execute();
```

No other, JSON-specific options are currently available. For additional Loader API options, please refer to the manual's section about importing CSV.

5.11.3. Importing Records

A common use-case for importing records via jOOQ's Loader API is when data needs to be transferred between databases. For instance, when fetching the following data from database 1:

```
Result<Record3<Integer, Integer, String>> result =
DSL.using(configuration1)
   .select(BOOK.ID, BOOK.AUTHOR_ID, BOOK.TITLE)
   .from(BOOK)
   .fetch();
```

Now, this result should be imported back into a database 2:

```
DSL.using(configuration2)
   .loadInto(BOOK)
   .loadRecords(result)
   .fields(BOOK.ID, BOOK.AUTHOR_ID, BOOK.TITLE)
   .execute();
```

No other, Record-specific options are currently available. For additional Loader API options, please refer to the manual's section about importing CSV.

5.11.4. Importing Arrays

A common use-case for importing arrays via jOOQ's Loader API is when data is fetched into memory from some data source, or even ad-hoc data, which needs to be imported into a database.

```
DSL.using(configuration)
   .loadInto(BOOK)
   .loadArrays(
       new Object[] { 1, 1, "1984" },
       new Object[] { 2, 1, "Animal Farm" })
   .fields(BOOK.ID, BOOK.AUTHOR_ID, BOOK.TITLE)
   .execute();
```

No other, Array-specific options are currently available. For additional Loader API options, please refer to the manual's section about importing CSV.

5.11.5. Importing XML

This is not yet supported

5.12. CRUD with UpdatableRecords

Your database application probably consists of 50% - 80% CRUD, whereas only the remaining 20% - 50% of querying is actual querying. Most often, you will operate on records of tables without using any advanced relational concepts. This is called CRUD for

- Create (INSERT)
- Read (SELECT)
- Update (UPDATE)
- Delete (DELETE)

CRUD always uses the same patterns, regardless of the nature of underlying tables. This again, leads to a lot of boilerplate code, if you have to issue your statements yourself. Like Hibernate / JPA and other ORMs, jOOQ facilitates CRUD using a specific API involving org.jooq.UpdatableRecord types.

Primary keys and updatability

In normalised databases, every table has a primary key by which a tuple/record within that table can be uniquely identified. In simple cases, this is a (possibly auto-generated) number called ID. But in many cases, primary keys include several non-numeric columns. An important feature of such keys is the fact that in most databases, they are enforced using an index that allows for very fast random access to the table. A typical way to access / modify / delete a book is this:

```
-- Inserting uses a previously generated key value or generates it afresh
INSERT INTO BOOK (ID, TITLE) VALUES (5, 'Animal Farm');

-- Other operations can use a previously generated key value
SELECT * FROM BOOK WHERE ID = 5;
UPDATE BOOK SET TITLE = '1984' WHERE ID = 5;
DELETE FROM BOOK WHERE ID = 5;
```

Normalised databases assume that a primary key is unique "forever", i.e. that a key, once inserted into a table, will never be changed or re-inserted after deletion. In order to use jOOQ's CRUD operations correctly, you should design your database accordingly.

5.12.1. Simple CRUD

If you're using jOOQ's code generator, it will generate org.jooq.UpdatableRecord implementations for every table that has a primary key. When fetching such a record form the database, these records are "attached" to the Configuration that created them. This means that they hold an internal reference to the same database connection that was used to fetch them. This connection is used internally by any of the following methods of the UpdatableRecord:

```
// Refresh a record from the database.
void refresh() throws DataAccessException;

// Store (insert or update) a record to the database.
int store() throws DataAccessException;

// Delete a record from the database
int delete() throws DataAccessException;
```

See the manual's section about serializability for some more insight on "attached" objects.

Storing

Storing a record will perform an INSERT statement or an UPDATE statement. In general, new records are always inserted, whereas records loaded from the database are always updated. This is best visualised in code:

```
// Create a new record
BookRecord book1 = create.newRecord(BOOK);

// Insert the record: INSERT INTO BOOK (TITLE) VALUES ('1984');
book1.setTitle("1984");
book1.store();

// Update the record: UPDATE BOOK SET PUBLISHED_IN = 1984 WHERE ID = [id]
book1.setPublishedIn(1948);
book1.store();

// Get the (possibly) auto-generated ID from the record
Integer id = book1.getId();

// Get another instance of the same book
BookRecord book2 = create.fetchOne(BOOK, BOOK.ID.eq(id));

// Update the record: UPDATE BOOK SET TITLE = 'Animal Farm' WHERE ID = [id]
book2.setTitle("Animal Farm");
book2.store();
```

Some remarks about storing:

- jOOQ sets only modified values in INSERT statements or UPDATE statements. This allows for default values to be applied to inserted records, as specified in CREATE TABLE DDL statements.
- When store() performs an INSERT statement, jOOQ attempts to load any generated keys from the database back into the record. For more details, see the manual's section about IDENTITY values.
- In addition to loading identity values, store() can also be configured to refresh the entire record. See the returnAllOnUpdatableRecord setting for details
- When loading records from POJOs, jOOQ will assume the record is a new record. It will hence attempt to INSERT it.
- When you activate optimistic locking, storing a record may fail, if the underlying database record has been changed in the mean time.

Deleting

Deleting a record will remove it from the database. Here's how you delete records:

```
// Get a previously inserted book
BookRecord book = create.fetchOne(BOOK, BOOK.ID.eq(5));

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

Refreshing

Refreshing a record from the database means that jOOQ will issue a SELECT statement to refresh all record values that are not the primary key. This is particularly useful when you use jOOQ's optimistic locking feature, in case a modified record is "stale" and cannot be stored to the database, because the underlying database record has changed in the mean time.

In order to perform a refresh, use the following Java code:

```
// Fetch an updatable record from the database
BookRecord book = create.fetchOne(BOOK, BOOK.ID.eq(5));

// Refresh the record
book.refresh();
```

CRUD and SELECT statements

CRUD operations can be combined with regular querying, if you select records from single database tables, as explained in the manual's section about SELECT statements. For this, you will need to use the selectFrom() method from the DSLContext:

```
// Loop over records returned from a SELECT statement
for (BookRecord book : create.fetch(BOOK, BOOK.PUBLISHED_IN.eq(1948))) {

  // Perform actions on BookRecords depending on some conditions
  if ("Orwell".equals(book.fetchParent(Keys.FK_BOOK_AUTHOR).getLastName())) {
    book.delete();
  }
}
```

5.12.2. Records' internal flags

All of jOOQ's Record types and subtypes maintain an internal state for every column value. This state is composed of three elements:

- The value itself
- The "original" value, i.e. the value as it was originally fetched from the database or null, if the record was never in the database
- The "changed" flag, indicating if the value was ever changed through the Record API.

The purpose of the above information is for jOOQ's CRUD operations to know, which values need to be stored to the database, and which values have been left untouched.

5.12.3. IDENTITY values

Many databases support the concept of IDENTITY values, or SEQUENCE-generated key values. This is reflected by JDBC's getGeneratedKeys() method. jOOQ abstracts using this method as many databases and JDBC drivers behave differently with respect to generated keys. Let's assume the following SQL Server BOOK table:

```
CREATE TABLE book (
  ID INTEGER IDENTITY(1,1) NOT NULL,

  -- [...]

  CONSTRAINT pk_book PRIMARY KEY (id)
)
```

If you're using jOOQ's code generator, the above table will generate a org.jooq.UpdatableRecord with an IDENTITY column. This information is used by jOOQ internally, to update IDs after calling store():

```
BookRecord book = create.newRecord(BOOK);
book.setTitle("1984");
book.store();

// The generated ID value is fetched after the above INSERT statement
System.out.println(book.getId());
```

Database compatibility

DB2, Derby, HSQLDB, Ingres

These SQL dialects implement the standard very neatly.

```
id INTEGER GENERATED BY DEFAULT AS IDENTITY
id INTEGER GENERATED BY DEFAULT AS IDENTITY (START WITH 1)
```

H2, MySQL, Postgres, SQL Server, Sybase ASE, Sybase SQL Anywhere

These SQL dialects implement identites, but the DDL syntax doesn't follow the standard

```
-- H2 mimicks MySQL's and SQL Server's syntax
ID INTEGER IDENTITY(1,1)
ID INTEGER AUTO_INCREMENT
-- MySQL and SQLite
ID INTEGER NOT NULL AUTO_INCREMENT

-- Postgres serials implicitly create a sequence
-- Postgres also allows for selecting from custom sequences
-- That way, sequences can be shared among tables
id SERIAL NOT NULL

-- SQL Server
ID INTEGER IDENTITY(1,1) NOT NULL
-- Sybase ASE
id INTEGER IDENTITY NOT NULL
-- Sybase SQL Anywhere
id INTEGER NOT NULL IDENTITY
```

5.12.4. Navigation methods

org.jooq.TableRecord and org.jooq.UpdatableRecord contain foreign key navigation methods. These navigation methods allow for "navigating" inbound or outbound foreign key references by executing an appropriate query. An example is given here:

```
CREATE TABLE book (
  AUTHOR_ID NUMBER(7) NOT NULL,

  -- [...]

  FOREIGN KEY (AUTHOR_ID) REFERENCES author(ID)
)
```

```
BookRecord book = create.fetch(BOOK, BOOK.ID.eq(5));

// Find the author of a book (static imported from Keys)
AuthorRecord author = book.fetchParent(FK_BOOK_AUTHOR);

// Find other books by that author
Result<BookRecord> books = author.fetchChildren(FK_BOOK_AUTHOR);
```

Note that, unlike in Hibernate, jOOQ's navigation methods will always lazy-fetch relevant records, without caching any results. In other words, every time you run such a fetch method, a new query will be issued.

These fetch methods only work on "attached" records. See the manual's section about serializability for some more insight on "attached" objects.

5.12.5. Non-updatable records

Tables without a PRIMARY KEY are considered non-updatable by jOOQ, as jOOQ has no way of uniquely identifying such a record within the database. If you're using jOOQ's code generator, such tables will generate org.jooq.TableRecord classes, instead of org.jooq.UpdatableRecord classes. When you fetch typed records from such a table, the returned records will not allow for calling any of the store(), refresh(), delete() methods.

Note, that some databases use internal rowid or object-id values to identify such records. jOOQ does not support these vendor-specific record meta-data.

5.12.6. Optimistic locking

jOOQ allows you to perform CRUD operations using optimistic locking. You can immediately take advantage of this feature by activating the relevant executeWithOptimisticLocking Setting. Without any further knowledge of the underlying data semantics, this will have the following impact on store() and delete() methods:

- INSERT statements are not affected by this Setting flag
- Prior to UPDATE or DELETE statements, jOOQ will run a SELECT .. FOR UPDATE statement, pessimistically locking the record for the subsequent UPDATE / DELETE
- The data fetched with the previous SELECT will be compared against the data in the record being stored or deleted
- An org.jooq.exception.DataChangedException is thrown if the record had been modified in the mean time
- The record is successfully stored / deleted, if the record had not been modified in the mean time.

The above changes to jOOQ's behaviour are transparent to the API, the only thing you need to do for it to be activated is to set the Settings flag. Here is an example illustrating optimistic locking:

```
// Properly configure the DSLContext
DSLContext optimistic = DSLContext.using(connection, SQLDialect.ORACLE,
  new Settings().withExecuteWithOptimisticLocking(true));

// Fetch a book two times
BookRecord book1 = optimistic.fetchOne(BOOK, BOOK.ID.eq(5));
BookRecord book2 = optimistic.fetchOne(BOOK, BOOK.ID.eq(5));

// Change the title and store this book. The underlying database record has not been modified, it can be safely updated.
book1.setTitle("Animal Farm");
book1.store();

// Book2 still references the original TITLE value, but the database holds a new value from book1.store().
// This store() will thus fail:
book2.setTitle("1984");
book2.store();
```

Optimised optimistic locking using TIMESTAMP fields

If you're using jOOQ's code generator, you can take indicate TIMESTAMP or UPDATE COUNTER fields for every generated table in the code generation configuration. Let's say we have this table:

```
CREATE TABLE book (

  -- This column indicates when each book record was modified for the last time
  MODIFIED TIMESTAMP NOT NULL,
  -- [...]
)
```

The MODIFIED column will contain a timestamp indicating the last modification timestamp for any book in the BOOK table. If you're using jOOQ and it's store() methods on UpdatableRecords, jOOQ will then generate this TIMESTAMP value for you, automatically. However, instead of running an additional SELECT .. FOR UPDATE statement prior to an UPDATE or DELETE statement, jOOQ adds a WHERE-clause to the UPDATE or DELETE statement, checking for TIMESTAMP's integrity. This can be best illustrated with an example:

```
// Properly configure the DSLContext
DSLContext optimistic = DSL.using(connection, SQLDialect.ORACLE,
  new Settings().withExecuteWithOptimisticLocking(true));

// Fetch a book two times
BookRecord book1 = optimistic.fetchOne(BOOK, BOOK.ID.eq(5));
BookRecord book2 = optimistic.fetchOne(BOOK, BOOK.ID.eq(5));

// Change the title and store this book. The MODIFIED value has not been changed since the book was fetched.
// It can be safely updated
book1.setTitle("Animal Farm");
book1.store();

// Book2 still references the original MODIFIED value, but the database holds a new value from book1.store().
// This store() will thus fail:
book2.setTitle("1984");
book2.store();
```

As before, without the added TIMESTAMP column, optimistic locking is transparent to the API.

Optimised optimistic locking using VERSION fields

Instead of using TIMESTAMPs, you may also use numeric VERSION fields, containing version numbers that are incremented by jOOQ upon store() calls.

Note, for explicit pessimistic locking, please consider the manual's section about the FOR UPDATE clause. For more details about how to configure TIMESTAMP or VERSION fields, consider the manual's section about advanced code generator configuration.

5.12.7. Batch execution

When inserting, updating, deleting a lot of records, you may wish to profit from JDBC batch operations, which can be performed by jOOQ. These are available through jOOQ's DSLContext as shown in the following example:

```
// Fetch a bunch of books
Result<BookRecord> books = create.fetch(BOOK);

// Modify the above books, and add some new ones:
modify(books);
addMore(books);

// Batch-update and/or insert all of the above books
create.batchStore(books);
```

Internally, jOOQ will render all the required SQL statements and execute them as a regular JDBC batch execution.

5.12.8. CRUD SPI: RecordListener

When performing CRUD, you may want to be able to centrally register one or several listener objects that receive notification every time CRUD is performed on an UpdatableRecord. Example use cases of such a listener are:

- Adding a central ID generation algorithm, generating UUIDs for all of your records.
- Adding a central record initialisation mechanism, preparing the database prior to inserting a new record.

An example of such a RecordListener is given here:

```
// Extending DefaultRecordListener, which provides empty implementations for all methods...
public class InsertListener extends DefaultRecordListener {

    @Override
    public void insertStart(RecordContext ctx) {

        // Generate an ID for inserted BOOKs
        if (ctx.record() instanceof BookRecord) {
            BookRecord book = (BookRecord) ctx.record();
            book.setId(IDTools.generate());
        }
    }
}
```

Now, configure jOOQ's runtime to load your listener

```
// Create a configuration with an appropriate listener provider:
Configuration configuration = new DefaultConfiguration().set(connection).set(dialect);
configuration.set(new DefaultRecordListenerProvider(new InsertListener()));

// Create a DSLContext from the above configuration
DSLContext create = DSL.using(configuration);
```

For a full documentation of what RecordListener can do, please consider the RecordListener Javadoc. Note that RecordListener instances can be registered with a Configuration independently of ExecuteListeners.

5.13. DAOs

If you're using jOOQ's code generator, you can configure it to generate POJOs and DAOs for you. jOOQ then generates one DAO per UpdatableRecord, i.e. per table with a single-column primary key. Generated DAOs implement a common jOOQ type called org.jooq.DAO. This type contains the following methods:

```
// <R> corresponds to the DAO's related table
// <P> corresponds to the DAO's related generated POJO type
// <T> corresponds to the DAO's related table's primary key type.
// Note that multi-column primary keys are not yet supported by DAOs
public interface DAO<R extends TableRecord<R>, P, T> {

    // These methods allow for inserting POJOs
    void insert(P object) throws DataAccessException;
    void insert(P... objects) throws DataAccessException;
    void insert(Collection<P> objects) throws DataAccessException;

    // These methods allow for updating POJOs based on their primary key
    void update(P object) throws DataAccessException;
    void update(P... objects) throws DataAccessException;
    void update(Collection<P> objects) throws DataAccessException;

    // These methods allow for deleting POJOs based on their primary key
    void delete(P... objects) throws DataAccessException;
    void delete(Collection<P> objects) throws DataAccessException;
    void deleteById(T... ids) throws DataAccessException;
    void deleteById(Collection<T> ids) throws DataAccessException;

    // These methods allow for checking record existence
    boolean exists(P object) throws DataAccessException;
    boolean existsById(T id) throws DataAccessException;
    long count() throws DataAccessException;

    // These methods allow for retrieving POJOs by primary key or by some other field
    List<P> findAll() throws DataAccessException;
    P findById(T id) throws DataAccessException;
    <Z> List<P> fetch(Field<Z> field, Z... values) throws DataAccessException;
    <Z> P fetchOne(Field<Z> field, Z value) throws DataAccessException;

    // These methods provide DAO meta-information
    Table<R> getTable();
    Class<P> getType();
}
```

Besides these base methods, generated DAO classes implement various useful fetch methods. An incomplete example is given here, for the BOOK table:

```
// An example generated BookDao class
public class BookDao extends DAOImpl<BookRecord, Book, Integer> {

    // Columns with primary / unique keys produce fetchOne() methods
    public Book fetchOneById(Integer value) { ... }

    // Other columns produce fetch() methods, returning several records
    public List<Book> fetchByAuthorId(Integer... values) { ... }
    public List<Book> fetchByTitle(String... values) { ... }
}
```

Note that you can further subtype those pre-generated DAO classes, to add more useful DAO methods to them. Using such a DAO is simple:

```
// Initialise an Configuration
Configuration configuration = new DefaultConfiguration().set(connection).set(SQLDialect.ORACLE);

// Initialise the DAO with the Configuration
BookDao bookDao = new BookDao(configuration);

// Start using the DAO
Book book = bookDao.findById(5);

// Modify and update the POJO
book.setTitle("1984");
book.setPublishedIn(1948);
bookDao.update(book);

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

5.14. Transaction management

There are essentially four ways how you can handle transactions in Java / SQL:

- You can issue vendor-specific COMMIT, ROLLBACK and other statements directly in your database.
- You can call JDBC's Connection.commit(), Connection.rollback() and other methods on your JDBC driver.
- You can use third-party transaction management libraries like Spring TX. Examples shown in the jOOQ with Spring examples section.
- You can use a JTA-compliant Java EE transaction manager from your container.

While jOOQ does not aim to replace any of the above, it offers a simple API (and a corresponding SPI) to provide you with jOOQ-style programmatic fluency to express your transactions. Below are some Java examples showing how to implement (nested) transactions with jOOQ. For these examples, we're using Java 8 syntax. Java 8 is not a requirement, though.

```
create.transaction(configuration -> {
    AuthorRecord author =
    DSL.using(configuration)
       .insertInto(AUTHOR, AUTHOR.FIRST_NAME, AUTHOR.LAST_NAME)
       .values("George", "Orwell")
       .returning()
       .fetchOne();

    DSL.using(configuration)
       .insertInto(BOOK, BOOK.AUTHOR_ID, BOOK.TITLE)
       .values(author.getId(), "1984")
       .values(author.getId(), "Animal Farm")
       .execute();

    // Implicit commit executed here
});
```

Note how the lambda expression receives a new, ***derived*** configuration that should be used within the local scope:

```
create.transaction(configuration -> {

    // Wrap configuration in a new DSLContext:
    DSL.using(configuration).insertInto(...);
    DSL.using(configuration).insertInto(...);

    // Or, reuse the new DSLContext within the transaction scope:
    DSLContext ctx = DSL.using(configuration);
    ctx.insertInto(...);
    ctx.insertInto(...);

    // ... but avoid using the scope from outside the transaction:
    create.insertInto(...);
    create.insertInto(...);
});
```

While some org.jooq.TransactionProvider implementations (e.g. ones based on ThreadLocals, e.g. Spring or JTA) may allow you to reuse the globally scoped DSLContext reference, the jOOQ transaction API design allows for TransactionProvider implementations that require your transactional code to use the new, locally scoped Configuration, instead.

Transactional code is wrapped in jOOQ's org.jooq.TransactionalRunnable or org.jooq.TransactionalCallable types:

```
public interface TransactionalRunnable {
    void run(Configuration configuration) throws Exception;
}

public interface TransactionalCallable<T> {
    T run(Configuration configuration) throws Exception;
}
```

Such transactional code can be passed to transaction(TransactionRunnable) or transactionResult(TransactionCallable) methods.

Rollbacks

Any uncaught checked or unchecked exception thrown from your transactional code will rollback the transaction to the beginning of the block. This behaviour will allow for nesting transactions, if your configured org.jooq.TransactionProvider supports nesting of transactions. An example can be seen here:

```
create.transaction(outer -> {
    final AuthorRecord author =
    DSL.using(outer)
       .insertInto(AUTHOR, AUTHOR.FIRST_NAME, AUTHOR.LAST_NAME)
       .values("George", "Orwell")
       .returning()
       .fetchOne();

    // Implicit savepoint created here
    try {
        DSL.using(outer)
           .transaction(nested -> {
               DSL.using(nested)
                  .insertInto(BOOK, BOOK.AUTHOR_ID, BOOK.TITLE)
                  .values(author.getId(), "1984")
                  .values(author.getId(), "Animal Farm")
                  .execute();

            // Rolls back the nested transaction
            if (oops)
                throw new RuntimeException("Oops");

            // Implicit savepoint is discarded, but no commit is issued yet.
        });
    }
    catch (RuntimeException e) {

        // We can decide whether an exception is "fatal enough" to roll back also the outer transaction
        if (isFatal(e))

            // Rolls back the outer transaction
            throw e;
    }

    // Implicit commit executed here
});
```

TransactionProvider implementations

By default, jOOQ ships with the org.jooq.impl.DefaultTransactionProvider, which implements nested transactions using JDBC java.sql.Savepoint. You can, however, implement your own org.jooq.TransactionProvider and supply that to your Configuration to override jOOQ's default behaviour. A simple example implementation using Spring's DataSourceTransactionManager can be seen here:

```
import static org.springframework.transaction.TransactionDefinition.PROPAGATION_NESTED;

import org.jooq.Transaction;
import org.jooq.TransactionContext;
import org.jooq.TransactionProvider;
import org.jooq.tools.JooqLogger;

import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.jdbc.datasource.DataSourceTransactionManager;
import org.springframework.transaction.TransactionStatus;
import org.springframework.transaction.support.DefaultTransactionDefinition;

public class SpringTransactionProvider implements TransactionProvider {

    private static final JooqLogger log = JooqLogger.getLogger(SpringTransactionProvider.class);

    @Autowired
    DataSourceTransactionManager txMgr;

    @Override
    public void begin(TransactionContext ctx) {
        log.info("Begin transaction");

        // This TransactionProvider behaves like jOOQ's DefaultTransactionProvider,
        // which supports nested transactions using Savepoints
        TransactionStatus tx = txMgr.getTransaction(new DefaultTransactionDefinition(PROPAGATION_NESTED));
        ctx.transaction(new SpringTransaction(tx));
    }

    @Override
    public void commit(TransactionContext ctx) {
        log.info("commit transaction");

        txMgr.commit(((SpringTransaction) ctx.transaction()).tx);
    }

    @Override
    public void rollback(TransactionContext ctx) {
        log.info("rollback transaction");

        txMgr.rollback(((SpringTransaction) ctx.transaction()).tx);
    }
}

class SpringTransaction implements Transaction {
    final TransactionStatus tx;

    SpringTransaction(TransactionStatus tx) {
        this.tx = tx;
    }
}
```

More information about how to use jOOQ with Spring can be found in the tutorials about jOOQ and Spring

5.15. Exception handling

Checked vs. unchecked exceptions

This is an eternal and religious debate. Pros and cons have been discussed time and again, and it still is a matter of taste, today. In this case, jOOQ clearly takes a side. jOOQ's exception strategy is simple:

- All "system exceptions" are unchecked. If in the middle of a transaction involving business logic, there is no way that you can recover sensibly from a lost database connection, or a constraint violation that indicates a bug in your understanding of your database model.
- All "business exceptions" are checked. Business exceptions are true exceptions that you should handle (e.g. not enough funds to complete a transaction).

With jOOQ, it's simple. All of jOOQ's exceptions are "system exceptions", hence they are all unchecked.

jOOQ's DataAccessException

jOOQ uses its own org.jooq.exception.DataAccessException to wrap any underlying java.sql.SQLException that might have occurred. Note that all methods in jOOQ that may cause such a DataAccessException document this both in the Javadoc as well as in their method signature.

DataAccessException is subtyped several times as follows:

- DataAccessException: General exception usually originating from a java.sql.SQLException
- DataChangedException: An exception indicating that the database's underlying record has been changed in the mean time (see optimistic locking)
- DataTypeException: Something went wrong during type conversion
- DetachedException: A SQL statement was executed on a "detached" UpdatableRecord or a "detached" SQL statement.
- InvalidResultException: An operation was performed expecting only one result, but several results were returned.
- MappingException: Something went wrong when loading a record from a POJO or when mapping a record into a POJO

Override jOOQ's exception handling

The following section about execute listeners documents means of overriding jOOQ's exception handling, if you wish to deal separately with some types of constraint violations, or if you raise business errors from your database, etc.

5.16. ExecuteListeners

The Configuration lets you specify a list of org.jooq.ExecuteListener instances. 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, triggers, etc. Advanced ExecuteListeners can also provide custom implementations of Connection, PreparedStatement and ResultSet to jOOQ in apropriate methods.

For convenience and better backwards-compatibility, consider extending org.jooq.impl.DefaultExecuteListener instead of implementing this interface.

Example: Query statistics ExecuteListener

Here is a sample implementation of an ExecuteListener, that is simply counting the number of queries per type that are being executed using jOOQ:

```
package com.example;

// Extending DefaultExecuteListener, which provides empty implementations for all methods...
public class TestStatisticsListener extends DefaultExecuteListener {

    /**
     * Generated UID
     */
    private static final long                     serialVersionUID = 7399239846062763212L;

    public static final Map<ExecuteType, Integer> STATISTICS       = new ConcurrentHashMap<>();

    @Override
    public void start(ExecuteContext ctx) {
        STATISTICS.compute(ctx.type(), (k, v) -> v == null ? 1 : v + 1);
    }
}
```

Now, configure jOOQ's runtime to load your listener

```
// Create a configuration with an appropriate listener provider:
Configuration configuration = new DefaultConfiguration().set(connection).set(dialect);
configuration.set(new DefaultExecuteListenerProvider(new StatisticsListener()));

// Create a DSLContext from the above configuration
DSLContext create = DSL.using(configuration);
```

And log results any time with a snippet like this:

```
log.info("STATISTICS");
log.info("----------");

for (ExecuteType type : ExecuteType.values()) {
    log.info(type.name(), StatisticsListener.STATISTICS.get(type) + " executions");
}
```

This may result in the following log output:

```
15:16:52,982  INFO - TEST STATISTICS
15:16:52,982  INFO - ---------------
15:16:52,983  INFO - READ                    : 919 executions
15:16:52,983  INFO - WRITE                   : 117 executions
15:16:52,983  INFO - DDL                     : 2 executions
15:16:52,983  INFO - BATCH                   : 4 executions
15:16:52,983  INFO - ROUTINE                 : 21 executions
15:16:52,983  INFO - OTHER                   : 30 executions
```

Please read the ExecuteListener Javadoc for more details

Example: Custom Logging ExecuteListener

The following depicts an example of a custom ExecuteListener, which pretty-prints all queries being executed by jOOQ to stdout:

```
import org.jooq.DSLContext;
import org.jooq.ExecuteContext;
import org.jooq.conf.Settings;
import org.jooq.impl.DefaultExecuteListener;
import org.jooq.tools.StringUtils;

public class PrettyPrinter extends DefaultExecuteListener {

    /**
     * Hook into the query execution lifecycle before executing queries
     */
    @Override
    public void executeStart(ExecuteContext ctx) {

        // Create a new DSLContext for logging rendering purposes
        // This DSLContext doesn't need a connection, only the SQLDialect...
        DSLContext create = DSL.using(ctx.dialect(),

        // ... and the flag for pretty-printing
         new Settings().withRenderFormatted(true));

        // If we're executing a query
        if (ctx.query() != null) {
            System.out.println(create.renderInlined(ctx.query()));
        }

        // If we're executing a routine
        else if (ctx.routine() != null) {
            System.out.println(create.renderInlined(ctx.routine()));
        }
    }
}
```

See also the manual's sections about logging for more sample implementations of actual ExecuteListeners.

Example: Bad query execution ExecuteListener

You can also use ExecuteListeners to interact with your SQL statements, for instance when you want to check if executed UPDATE or DELETE statements contain a WHERE clause. This can be achieved trivially with the following sample ExecuteListener:

```
public class DeleteOrUpdateWithoutWhereListener extends DefaultExecuteListener {

    @Override
    public void renderEnd(ExecuteContext ctx) {
        if (ctx.sql().matches("^(?i:(UPDATE|DELETE)(?!.* WHERE ).*)$")) {
            throw new DeleteOrUpdateWithoutWhereException();
        }
    }
}

public class DeleteOrUpdateWithoutWhereException extends RuntimeException {}
```

You might want to replace the above implementation with a more efficient and more reliable one, of course.

5.17. Database meta data

Since jOOQ 3.0, a simple wrapping API has been added to wrap JDBC's rather awkward java.sql.DatabaseMetaData. This API is still experimental, as the calls to the underlying JDBC type are not always available for all SQL dialects.

5.18. Mocking Connection

When writing unit tests for your data access layer, you have probably used some generic mocking tool offered by popular providers like Mockito, jmock, mockrunner, or even DBUnit. With jOOQ, you can take advantage of the built-in JDBC mock API that allows you to emulate a simple database on the JDBC level for precisely those SQL/JDBC use cases supported by jOOQ.

Disclaimer: The general idea of mocking a JDBC connection with this jOOQ API is to provide quick workarounds, injection points, etc. using a very simple JDBC abstraction. It is ***NOT RECOMMENDED*** to emulate an entire database (including complex state transitions, transactions, locking, etc.) using this mock API. Once you have this requirement, please consider using an actual database instead for integration testing, rather than implementing your test database inside of a MockDataProvider.

Mocking the JDBC API

JDBC is a very complex API. It takes a lot of time to write a useful and correct mock implementation, implementing at least these interfaces:

- java.sql.Connection
- java.sql.Statement
- java.sql.PreparedStatement
- java.sql.CallableStatement
- java.sql.ResultSet
- java.sql.ResultSetMetaData

Optionally, you may even want to implement interfaces, such as java.sql.Array, java.sql.Blob, java.sql.Clob, and many others. In addition to the above, you might need to find a way to simultaneously support incompatible JDBC minor versions, such as 4.0, 4.1

Using jOOQ's own mock API

This work is greatly simplified, when using jOOQ's own mock API. The org.jooq.tools.jdbc package contains all the essential implementations for both JDBC 4.0 and 4.1, which are needed to mock JDBC for jOOQ. In order to write mock tests, provide the jOOQ Configuration with a MockConnection, and implement the MockDataProvider:

```
// Initialise your data provider (implementation further down):
MockDataProvider provider = new MyProvider();
MockConnection connection = new MockConnection(provider);

// Pass the mock connection to a jOOQ DSLContext:
DSLContext create = DSL.using(connection, SQLDialect.ORACLE);

// Execute queries transparently, with the above DSLContext:
Result<BookRecord> result = create.selectFrom(BOOK).where(BOOK.ID.eq(5)).fetch();
```

As you can see, the configuration setup is simple. Now, the MockDataProvider acts as your single point of contact with JDBC / jOOQ. It unifies any of these execution modes, transparently:

- Statements without results
- Statements without results but with generated keys
- Statements with results
- Statements with several results
- Batch statements with single queries and multiple bind value sets
- Batch statements with multiple queries and no bind values

The above are the execution modes supported by jOOQ. Whether you're using any of jOOQ's various fetching modes (e.g. pojo fetching, lazy fetching, many fetching, later fetching) is irrelevant, as those modes are all built on top of the standard JDBC API.

Implementing MockDataProvider

Now, here's how to implement MockDataProvider:

```
public class MyProvider implements MockDataProvider {

    @Override
    public MockResult[] execute(MockExecuteContext ctx) throws SQLException {

        // You might need a DSLContext to create org.jooq.Result and org.jooq.Record objects
        DSLContext create = DSL.using(SQLDialect.ORACLE);
        MockResult[] mock = new MockResult[1];

        // The execute context contains SQL string(s), bind values, and other meta-data
        String sql = ctx.sql();

        // Exceptions are propagated through the JDBC and jOOQ APIs
        if (sql.toUpperCase().startsWith("DROP")) {
            throw new SQLException("Statement not supported: " + sql);
        }

        // You decide, whether any given statement returns results, and how many
        else if (sql.toUpperCase().startsWith("SELECT")) {

            // Always return one record
            Result<Record2<Integer, String>> result = create.newResult(AUTHOR.ID, AUTHOR.LAST_NAME);
            result.add(create
                .newRecord(AUTHOR.ID, AUTHOR.LAST_NAME)
                .values(1, "Orwell"));
            mock[0] = new MockResult(1, result);
        }

        // You can detect batch statements easily
        else if (ctx.batch()) {
            // [...]
        }

        return mock;
    }
}
```

Essentially, the MockExecuteContext contains all the necessary information for you to decide, what kind of data you should return. The MockResult wraps up two pieces of information:

- Statement.getUpdateCount(): The number of affected rows
- Statement.getResultSet(): The result set

You should return as many MockResult objects as there were query executions (in batch mode) or results (in fetch-many mode). Instead of an awkward JDBC ResultSet, however, you can construct a "friendlier" org.jooq.Result with your own record types. The jOOQ mock API will use meta data provided with this Result in order to create the necessary JDBC java.sql.ResultSetMetaData

See the MockDataProvider Javadoc for a list of rules that you should follow.

5.19. Mock File Database

An alternative to the previous programmatic implementation of a mocking connection is the more declarative approach using a org.jooq.tools.jdbc.MockFileDatabase, which reads SQL statements and their corresponding result sets directly from a file. Assuming the following file content:

```
# All lines with a leading hash are ignored. This is the MockFileDatabase comment syntax
-- SQL comments are parsed and passed to the SQL statement
/* The same is true for multi-line SQL comments */
select 'A';
> A
> -
> A
@ rows: 1

select 'A', 'B' union all 'C', 'D';
> A B
> - -
> A B
> C D
@ rows: 2

# Statements without result sets just leave that section empty
update t set x = 1;
@ rows: 3
```

The above syntax consists of the following elements to define an individual statement:

- MockFileDatabase comments are any line with a leading hash ("#") symbol. They are ignored when reading the file
- SQL comments are part of the SQL statement
- A SQL statement always starts on a new line and ends with a semi colon (;), which is the last symbol on the line (apart from whitespace)
- If the statement has a result set, it immediately succeeds the SQL statement and is prefixed by angle brackets and a whitespace ("> "). Any format that is accepted by DSLContext.fetchFromTXT() is accepted.
- The statement is always terminated by the row count, which is prefixed by an at symbol, the "rows" keyword, and a double colon ("@ rows:").

The above database supports exactly two statements in total, and is completely stateless (e.g. an INSERT statement cannot be made to affect the results of a subsequent SELECT statement on the same table). It can be loaded through the MockFileDatabase can be used as follows:

```
// Initialise your data provider (implementation further down):
MockFileDatabase db = new MockFileDatabase(new File("/path/to/db.txt"));
MockConnection connection = new MockConnection(provider);

// Pass the mock connection to a jOOQ DSLContext:
DSLContext create = DSL.using(connection, SQLDialect.POSTGRES);

// Execute queries transparently, with the above DSLContext:
Result<?> result = create.select(inline("A")).fetch();
Result<?> result = create.select(inline("A"), inline("B")).fetch();

// Queries that are not listed in the MockFileDatabase will simply fail
Result<?> result = create.select(inline("C")).fetch();
```

In situations where the expected set of queries are well-defined, the MockFileDatabase can offer a very effective way of mocking parts of the database engine, without offering the complete functionality of the programmatic mocking connection.

5.20. Parsing Connection

As previously discussed in the manual's section about the SQL parser, jOOQ exposes a full-fledged SQL parser through DSLContext.parser(), and often more interestingly: Through DSLContext.parsingConnection().

A parsing connection is a JDBC java.sql.Connection, which proxies all commands through the jOOQ parser, transforming the inbound SQL statement (and bind variables) given the entirety of jOOQ's Configuration, including SQLDialect, Settings (e.g. formatting SQL or inlining bind variables) and much more. This allows for transparent SQL transformation of the SQL produced by any JDBC client (including JPA!). Here's a simple usage example:

```
// Configuration is configured with the target DataSource, SQLDialect, etc. for instance Oracle.
try (Connection c = DSL.using(configuration).parsingConnection();
     Statement s = c.createStatement();

     // This syntax is not supported in Oracle, but thanks to the parser and jOOQ,
     // it will run on Oracle and produce the expected result
     ResultSet rs = s.executeQuery("SELECT * FROM (VALUES (1, 'a'), (2, 'b')) t(x, y)")) {

    while (rs.next())
        System.out.println("x: " + rs.getInt(1) + ", y: " + rs.getString());
}
```

Running the above statement will yield:

```
x: 1, y: a
x: 2, y: b
```

5.21. Diagnostics

jOOQ includes a powerful diagnostics SPI, which can be used to detect problems and inefficiencies on different levels of your database interaction:

- On the jOOQ API level
- On the JDBC level
- On the SQL level

Just like the parsing connection, which was documented in the previuos section, this functionality does not depend on using the jOOQ API in a client application, but can expose itself through a JDBC java.sql.Connection that proxies your real database connection.

```
// A custom DiagnosticsListener SPI implementation
class MyDiagnosticsListener extends DefaultDiagnosticsListener {
    // Override methods here
}

// Configuration is configured with the target DataSource, SQLDialect, etc. for instance Oracle.
try (Connection c = DSL.using(configuration.derive(new MyDiagnosticsListener()))
                       .diagnosticsConnection();
     Statement s = c.createStatement()) {

    // The tooManyRowsFetched() event is triggered.
    // --------------------------------------------
    // This logic does not consume the entire ResultSet. There is more than one row
    // ready to be fetched into the client, but the client only fetches one row.
    try (ResultSet rs = s.executeQuery("SELECT id, title FROM book WHERE id > 1")) {
        if (rs.next())
            System.out.println("ID: " + rs.getInt(1) + ", title: " + rs.getInt(2));
    }

    // The duplicateStatements() event is triggered.
    // ---------------------------------------------
    // The statement is the same as the previous one, apart from a different "bind variable".
    // Unfortunately, no actual bind variables were used, which may
    // 1) hint at a SQL injection risk
    // 2) can cause a lot of pressure / contention on execution plan caches and SQL parsers
    //
    // The tooManyColumnsFetched() event is triggered.
    // -----------------------------------------------
    // When iterating the ResultSet, we're actually only ever reading the TITLE column, never
    // the ID column. This means we probably should not have projected it in the first place
    try (ResultSet rs = s.executeQuery("SELECT id, title FROM book WHERE id > 2")) {
        while (rs.next())
            System.out.println("Title: " + rs.getString(2));
    }
}
```

This feature incurs a certain overhead over normal operation as it requires:

- Parsing SQL statements and re-rendering them back to normalised SQL.
- Storing a limited size list of such normalised SQL in a cache to gather statistics on that cache.

The following sections describe each individual event, how it can happen, how and why it should be remedied.

5.21.1. Too Many Rows

The SPI method handling this event is tooManyRowsFetched()

If you're using jOOQ (or an ORM) to eagerly fetch your entire result set, then this will not be a problem in your code base, but when working with jOOQ's lazy fetching API or lazy streaming support, or as well as with JDBC java.sql.ResultSet directly, then it may certainly be possible that client code is aborting the iteration over the entire result set prematurely.

Why is it bad?

While it is definitely good not to fetch too many rows from a JDBC ResultSet, it would be even better to communicate to the database that only a limited number of rows are going to be needed in the client, by using the LIMIT clause. Not only will this prevent the pre-allocation of some resources both in the client and in the server, but it opens up the possibility of much better execution plans. For instance, the optimiser may prefer to chose nested loop joins over hash joins if it knows that the loops can be aborted early.

An example is given here:

```
// A custom DiagnosticsListener SPI implementation
class TooManyRows extends DefaultDiagnosticsListener {
    @Override
    public void tooManyRowsFetched(DiagnosticsContext ctx) {
        System.out.println("Consumed rows: " + ctx.resultSetConsumedRows());

        // This incurs overhead by consuming the ResultSet! Use only if needed.
        System.out.println("Fetched rows: " + ctx.resultSetFetchedRows());
    }
}

// Configuration is configured with the target DataSource, SQLDialect, etc. for instance Oracle.
try (Connection c = DSL.using(configuration.derive(new TooManyRows()))
                       .diagnosticsConnection();
     Statement s = c.createStatement()) {

    try (ResultSet rs = s.executeQuery("SELECT id FROM book")) {

        // Unlike "while", "if" only consumes the first row, here.
        if (rs.next())
            System.out.println("ID: " + rs.getInt(1));
    }
}
```

5.21.2. Too Many Columns

The SPI method handling this event is tooManyColumnsFetched()

A common problem with SQL is the high probability of having too many columns in the projection. This may be due to reckless usage of SELECT * or some refactoring which removes the need to select some of the projected columns, but the query was not adapted.

If the columns are consumed by some client (e.g. an ORM), then the jOOQ diagnostics have no way of knowing whether they were actually needed or not. But if they are never consumed from the JDBC java.sql.ResultSet, then we can say with certainty that too many columns have been projected.

Why is it bad?

The drawbacks of projecting too many columns are manifold:

- Too much data is loaded, cached, transferred between server and client. The overall resource consumption of a system is too high if too many columns are projected. This can cause orders of magnitude of overhead in extreme cases!
- Locking could occur in cases where it otherwise wouldn't happen, because two conflicting queries actually don't really need to touch the same columns.
- The probability of using "covering indexes" (or "index only scans") on some tables decreases because of the unnecessary projection. This can have drastic effects!
- The probability of applying JOIN elimination decreases, because of the unnecessary projection. This is particularly true if you're querying views.

An example is given here:

```
// A custom DiagnosticsListener SPI implementation
class TooManyColumns extends DefaultDiagnosticsListener {
    @Override
    public void tooManyColumnsFetched(DiagnosticsContext ctx) {
        System.out.println("Consumed columns: " + ctx.resultSetConsumedColumnCount());
        System.out.println("Fetched columns: " + ctx.resultSetFetchedColumnCount());
    }
}

// Configuration is configured with the target DataSource, SQLDialect, etc. for instance Oracle.
try (Connection c = DSL.using(configuration.derive(new TooManyColumns()))
                       .diagnosticsConnection();
     Statement s = c.createStatement()) {

    try (ResultSet rs = s.executeQuery("SELECT id, title FROM book")) {

        // On none of the rows, we retrieve the TITLE column, so selecting it would not have been necessary.
        while (rs.next())
            System.out.println("ID: " + rs.getInt(1));
    }
}
```

5.21.3. Duplicate Statements

The SPI method handling this event is duplicateStatements()

A common source of overhead in databases that have an execution plan cache (or "cursor cache", etc.) are static statements, or prepared statements that differ only by "irrelevant" things:

- They may differ by white space
- They may differ by irrelevant syntactic elements (e.g. excess parentheses, or object name qualification, table aliasing, etc.)
- They may differ by input values, which are inlined into the statement rather than parameterised as bind values
- They may differ by the length of their IN predicate's IN-list sizes

Why is it bad?

There are two main problems:

- If the duplicate SQL appears in dynamic SQL (i.e. in generated SQL), then there is an indication that the database's parser and optimiser may have to do too much work parsing the various similar (but not identical) SQL queries and finding an execution plan for them, each time. In fact, it will find the ***same*** execution plan most of the time, but with some significant overhead. Depending on the query complexity, this overhead can easily go from milliseconds into several seconds, blocking important resources in the database. If duplicate SQL happens at peak load times, this problem can have a significant impact in production. It never affects your (single user) development environments, where the overhead of parsing duplicate SQL is manageable.
- If the duplicate SQL appears in static SQL, this can simply indicate that the query was copy pasted, and you might be able to refactor it. There's probably not any performance issue arising from duplicate static SQL

An example is given here:

```
// A custom DiagnosticsListener SPI implementation
class DuplicateStatements extends DefaultDiagnosticsListener {
    @Override
    public void duplicateStatements(DiagnosticsContext ctx) {

        // The statement that is being executed and which has duplicates
        System.out.println("Actual statement: " + ctx.actualStatement());

        // A normalised version of the actual statement, which is shared by all duplicates
        // This statement has "normal" whitespace, bind variables, IN-lists, etc.
        System.out.println("Normalised statement: " + ctx.normalisedStatement());

        // All the duplicate actual statements that have produced the same normalised
        // statement in the recent past.
        System.out.println("Duplicate statements: " + ctx.duplicateStatements());
    }
}

// Utility to run SQL on a new JDBC Statement
void run(String sql) {

    // Configuration is configured with the target DataSource, SQLDialect, etc. for instance Oracle.
    try (Connection c = DSL.using(configuration.derive(new DuplicateStatements()))
                          .diagnosticsConnection();
         Statement s = c.createStatement();
         ResultSet rs = s.executeQuery(sql)) {

        while (rs.next()) {
            // Consume result set
        }
    }
}
    // Everything is fine with the first execution
    run("SELECT title FROM book WHERE id = 1");

    // This query is identical to the previous one, differing only in irrelevant white space
    run("SELECT title FROM book WHERE  id = 1");

    // This query is identical to the previous one, differing only in irrelevant additional parentheses
    run("SELECT title FROM book WHERE (id = 1)");

    // This query is identical to the previous one, differing only in what should be a bind variable
    run("SELECT title FROM book WHERE id = 2");

    // Everything is fine with the first execution of a new query that has never been seen
    run("SELECT title FROM book WHERE id IN (1, 2, 3, 4, 5)");

    // This query is identical to the previous one, differing only in what should be bind variables
    run("SELECT title FROM book WHERE id IN (1, 2, 3, 4, 5, 6)");
}
```

Unlike when detecting repeated statements, duplicate statement statistics are performed globally over all JDBC connections and data sources.

5.21.4. Repeated statements

The SPI method handling this event is repeatedStatements()

Sometimes, there is no other option than to repeat an identical (or similar, see duplicate statements) statement many times in a row, but often, it is a sign that your queries can be rewritten and your repeated statements should really be joined to a larger query.

Why is it bad?

This problem is usually referred to as the N+1 problem. A parent entity is loaded (often by an ORM), and its child entities are loaded lazily. Unfortunately, there were several parent instances, so for each parent instance, we're now loading a set of child instances, resulting in many many queries. This diagnostic detects if on the same connection, there is repeated execution of the same statement, even if it is not exactly identical.

An example is given here:

```
// A custom DiagnosticsListener SPI implementation
class RepeatedStatement extends DefaultDiagnosticsListener {
    @Override
    public void tooManyRowsFetched(DiagnosticsContext ctx) {

        // The statement that is being executed and which has duplicates
        System.out.println("Actual statement: " + ctx.actualStatement());

        // A normalised version of the actual statement, which is shared by all duplicates
        // This statement has "normal" whitespace, bind variables, IN-lists, etc.
        System.out.println("Normalised statement: " + ctx.normalisedStatement());

        // All the duplicate actual statements that have produced the same normalised
        // statement in the recent past.
        System.out.println("Repeated statements: " + ctx.repeatedStatements());
    }
}

// Configuration is configured with the target DataSource, SQLDialect, etc. for instance Oracle.
try (Connection c = DSL.using(configuration.derive(new RepeatedStatement()))
                       .diagnosticsConnection();
     Statement s1 = c.createStatement();
     ResultSet a = s1.executeQuery("SELECT id FROM author WHERE first_name LIKE 'A%'")) {

    while (a.next()) {
        int id = a.getInt(1);

        // This query is run once for every author, when we could have joined the author table
        try (PreparedStatement s2 = c.prepareStatement("SELECT title FROM book WHERE author_id = ?")) {
            s2.setInt(1, id);

            try (ResultSet b = s2.executeQuery()) {
                while (b.next())
                    System.out.println("ID: " + id + ", title: " + b.getString(1));
            }
        }
    }
}
```

Unlike when detecting repeated statements, repeated statement statistics are performed locally only, for a single JDBC Connection, or, if possible, for a transaction. Repeated statements in different transactions are usually not an indication of a problem.

5.21.5. WasNull calls

The SPI methods handling these events are unnecessaryWasNullCall()

This problem appears only when using the JDBC API directly, as both jOOQ and most ORMs abstract over this JDBC legacy correctly. In JDBC, when fetching a primitive types (and some types are available only in their primitive form), a subsequent call to ResultSet.wasNull() is required, to see if the previous primitive type was really a NULL value, not a 0 or false value.

Why is it bad?

There are two misuses that can arise in this area:

- The call to wasNull() wasn't made when it should have been (nullable type, fetched as a primitive type), possibly resulting in wrong results in the client.
- The call to wasNull() was made too often, or when it did not need to have been made (non-nullable type, or types fetched as reference types), possibly resulting in a very slight performance overhead, depending on the driver.

An example is given here:

```
// A custom DiagnosticsListener SPI implementation
class WasNull extends DefaultDiagnosticsListener {
    @Override
    public void tooManyRowsFetched(DiagnosticsContext ctx) {
        System.out.println("Unnecessary wasNull() call: " + ctx.resultSetUnnecessaryWasNullCall());
        System.out.println("Missing wasNull() call: " + ctx.resultSetMissingWasNullCall());
    }
}

// Configuration is configured with the target DataSource, SQLDialect, etc. for instance Oracle.
try (Connection c = DSL.using(configuration.derive(new WasNull()))
                      .diagnosticsConnection();
     Statement s = c.createStatement()) {

    try (ResultSet rs = s.executeQuery("SELECT year_of_birth FROM author")) {

        // The YEAR_OF_BIRTH column is nullable, so a 0 int could really mean null
        while (rs.next())
            System.out.println("Year of birth: " + rs.getInt(1));
    }
}
```

5.22. Logging

jOOQ logs all SQL queries and fetched result sets to its internal DEBUG logger, which is implemented as an execute listener. By default, execute logging is activated in the jOOQ Settings. In order to see any DEBUG log output, put either log4j or slf4j on jOOQ's classpath along with their respective configuration. A sample log4j configuration can be seen here:

```
<?xml version="1.0" encoding="UTF-8"?>
<log4j:configuration>
    <appender name="stdout" class="org.apache.log4j.ConsoleAppender">
        <layout class="org.apache.log4j.PatternLayout">
            <param name="ConversionPattern" value="%m%n" />
        </layout>
    </appender>

    <root>
        <priority value="debug" />
        <appender-ref ref="stdout" />
    </root>
</log4j:configuration>
```

With the above configuration, let's fetch some data with jOOQ

```
create.select(BOOK.ID, BOOK.TITLE).from(BOOK).orderBy(BOOK.ID).limit(1, 2).fetch();
```

The above query may result in the following log output:

```
Executing query          : select "BOOK"."ID", "BOOK"."TITLE" from "BOOK" order by "BOOK"."ID" asc limit ? offset ?
-> with bind values      : select "BOOK"."ID", "BOOK"."TITLE" from "BOOK" order by "BOOK"."ID" asc limit 2 offset 1
Query executed           : Total: 1.439ms
Fetched result           : +----+------------+
                         : |  ID|TITLE       |
                         : +----+------------+
                         : |   2|Animal Farm |
                         : |   3|O Alquimista|
                         : +----+------------+
Finishing                : Total: 4.814ms, +3.375ms
```

Essentially, jOOQ will log

- The SQL statement as rendered to the prepared statement
- The SQL statement with inlined bind values (for improved debugging)
- The query execution time
- The first 5 records of the result. This is formatted using jOOQ's text export
- The total execution + fetching time

If you wish to use your own logger (e.g. avoiding printing out sensitive data), you can deactivate jOOQ's logger using your custom settings and implement your own execute listener logger.

5.23. Performance considerations

Many users may have switched from higher-level abstractions such as Hibernate to jOOQ, because of Hibernate's difficult-to-manage performance, when it comes to large database schemas and complex second-level caching strategies. However, jOOQ itself is not a lightweight database abstraction framework, and it comes with its own overhead. Please be sure to consider the following points:

- It takes some time to construct jOOQ queries. If you can reuse the same queries, you might cache them. Beware of thread-safety issues, though, as jOOQ's Configuration is not necessarily threadsafe, and queries are "attached" to their creating DSLContext
- It takes some time to render SQL strings. Internally, jOOQ reuses the same java.lang.StringBuilder for the complete query, but some rendering elements may take their time. You could, of course, cache SQL generated by jOOQ and prepare your own java.sql.PreparedStatement objects
- It takes some time to bind values to prepared statements. jOOQ does not keep any open prepared statements, internally. Use a sophisticated connection pool, that will cache prepared statements and inject them into jOOQ through the standard JDBC API
- It takes some time to fetch results. By default, jOOQ will always fetch the complete java.sql.ResultSet into memory. Use lazy fetching to prevent that, and scroll over an open underlying database cursor

Optimise wisely

Don't be put off by the above paragraphs. You should optimise wisely, i.e. only in places where you really need very high throughput to your database. jOOQ's overhead compared to plain JDBC is typically less than 1ms per query.

5.24. Alternative execution models

Just because you can, doesn't mean you must. In this chapter, we'll show how you can use jOOQ to generate SQL statements that are then executed with other APIs, such as Spring's JdbcTemplate, or Hibernate.

5.24.1. Using jOOQ with Spring's JdbcTemplate

A lot of people are using Spring's useful org.springframework.jdbc.core.JdbcTemplate in their projects to simplify common JDBC interaction patterns, such as:

- Variable binding
- Result mapping
- Exception handling

When adding jOOQ to a project that is using JdbcTemplate extensively, a pragmatic first step is to use jOOQ as a SQL builder and pass the query string and bind variables to JdbcTemplate for execution. For instance, you may have the following class to store authors and their number of books in our stores:

```
public class AuthorAndBooks {
    public final String firstName;
    public final String lastName;
    public final int books;

    public AuthorAndBooks(String firstName, String lastName, int books) {
        this.firstName = firstName;
        this.lastName = lastName;
        this.books = books;
    }
}
```

You can then write the following code

```
// The jOOQ part stays the same as always:
Book b = BOOK.as("b");
Author a = AUTHOR.as("a");
BookStore s = BOOK_STORE.as("s");
BookToBookStore t = BOOK_TO_BOOK_STORE.as("t");

ResultQuery<Record3<String, String, Integer>> query =
create.select(a.FIRST_NAME, a.LAST_NAME, countDistinct(s.NAME))
      .from(a)
      .join(b).on(b.AUTHOR_ID.equal(a.ID))
      .join(t).on(t.BOOK_ID.equal(b.ID))
      .join(s).on(t.BOOK_STORE_NAME.equal(s.NAME))
      .groupBy(a.FIRST_NAME, a.LAST_NAME)
      .orderBy(countDistinct(s.NAME).desc());

// But instead of executing the above query, we'll send the SQL string and the bind values to JdbcTemplate:
JdbcTemplate template = new JdbcTemplate(dataSource);
List<AuthorAndBooks> result = template.query(
    query.getSQL(),
    query.getBindValues().toArray(),
    (r, i) -> new AuthorAndBooks(
        r.getString(1),
        r.getString(2),
        r.getInt(3)
    ));
```

This approach helps you gradually migrate from using JdbcTemplate to a jOOQ-only execution model.

5.24.2. Using jOOQ with JPA

These sections will show how to use jOOQ with JPA's native query API in order to fetch tuples or managed entities using the Java EE standards.

In all of the following sections, let's assume we have the following JPA entities to model our database:

```
@Entity
@Table(name = "book")
public class JPABook {

    @Id
    public int id;

    @Column(name = "title")
    public String title;

    @ManyToOne
    public JPAAuthor author;

    @Override
    public String toString() {
        return "JPABook [id=" + id + ", title=" + title + ", author=" + author + "]";
    }
}

@Entity
@Table(name = "author")
public class JPAAuthor {

    @Id
    public int id;

    @Column(name = "first_name")
    public String firstName;

    @Column(name = "last_name")
    public String lastName;

    @OneToMany(mappedBy = "author")
    public Set<JPABook> books;

    @Override
    public String toString() {
        return "JPAAuthor [id=" + id + ", firstName=" + firstName +
               ", lastName=" + lastName + ", book size=" + books.size() + "]";
    }
}
```

5.24.2.1. Using jOOQ with JPA Native Query

If your query doesn't really map to JPA entities, you can fetch ordinary, untyped Object[] representations for your database records by using the following utility method:

```
static List<Object[]> nativeQuery(EntityManager em, org.jooq.Query query) {

    // Extract the SQL statement from the jOOQ query:
    Query result = em.createNativeQuery(query.getSQL());

    // Extract the bind values from the jOOQ query:
    List<Object> values = query.getBindValues();
    for (int i = 0; i < values.size(); i++) {
        result.setParameter(i + 1, values.get(i));
    }

    return result.getResultList();
}
```

Note, if you're using custom data types or bindings, make sure to take those into account as well. E.g. as follows:

```
static List<Object[]> nativeQuery(EntityManager em, org.jooq.Query query) {

    // Extract the SQL statement from the jOOQ query:
    Query result = em.createNativeQuery(query.getSQL());

    // Extract the bind values from the jOOQ query:
    int i = 1;
    for (Param<?> param : query.getParams().values())
        if (!param.isInline())
            result.setParameter(i++, convertToDatabaseType(param));

    return result.getResultList();
}

static <T> Object convertToDatabaseType(Param<T> param) {
    return param.getBinding().converter().to(param.getValue());
}
```

This way, you can construct complex, type safe queries using the jOOQ API and have your javax.persistence.EntityManager execute it with all the transaction semantics attached:

```
List<Object[]> books =
nativeQuery(em, DSL.using(configuration)
    .select(AUTHOR.FIRST_NAME, AUTHOR.LAST_NAME, BOOK.TITLE)
    .from(AUTHOR)
    .join(BOOK).on(AUTHOR.ID.eq(BOOK.AUTHOR_ID))
    .orderBy(BOOK.ID));

books.forEach((Object[] book) -> System.out.println(book[0] + " " + book[1] + " wrote " + book[2]));
```

5.24.2.2. Using jOOQ with JPA entities

The simplest way to fetch entities via the native query API is by passing the entity class along to the native query method. The following example maps jOOQ query results to JPA entities (from the previous section). Just add the following utility method:

```
public static <E> List<E> nativeQuery(EntityManager em, org.jooq.Query query, Class<E> type) {

    // Extract the SQL statement from the jOOQ query:
    Query result = em.createNativeQuery(query.getSQL(), type);

    // Extract the bind values from the jOOQ query:
    List<Object> values = query.getBindValues();
    for (int i = 0; i < values.size(); i++) {
        result.setParameter(i + 1, values.get(i));
    }

    // There's an unsafe cast here, but we can be sure that we'll get the right type from JPA
    return result.getResultList();
}
```

Note, if you're using custom data types or bindings, make sure to take those into account as well. E.g. as follows:

```
static List<Object[]> nativeQuery(EntityManager em, org.jooq.Query query, Class<E> type) {

    // Extract the SQL statement from the jOOQ query:
    Query result = em.createNativeQuery(query.getSQL(), type);

    // Extract the bind values from the jOOQ query:
    int i = 1;
    for (Param<?> param : query.getParams().values())
        if (!param.isInline())
            result.setParameter(i++, convertToDatabaseType(param));

    return result.getResultList();
}

static <T> Object convertToDatabaseType(Param<T> param) {
    return param.getBinding().converter().to(param.getValue());
}
```

With the above simple API, we're ready to write complex jOOQ queries and map their results to JPA entities:

```
List<JPAAuthor> authors =
nativeQuery(em,
    DSL.using(configuration)
       .select()
       .from(AUTHOR)
       .orderBy(AUTHOR.ID)
, JPAAuthor.class);

authors.forEach(author -> {
    System.out.println(author.firstName + " " + author.lastName + " wrote");

    books.forEach(book -> {
        System.out.println("  " + book.title);
    });
});
```

5.24.2.3. Using jOOQ with JPA EntityResult

While JPA specifies how the mapping should be implemented (e.g. using javax.persistence.SqlResultSetMapping), there are no limitations regarding how you want to generate the SQL statement. The following, simple example shows how you can produce JPABook and JPAAuthor entities (from the previous section) from a jOOQ-generated SQL statement.

In order to do so, we'll need to specify the SqlResultSetMapping. This can be done on any entity, and in this case, we're using javax.persistence.EntityResult:

```
@SqlResultSetMapping(
    name = "bookmapping",
    entities = {
        @EntityResult(
            entityClass = JPABook.class,
            fields = {
                @FieldResult(name = "id", column = "b_id"),
                @FieldResult(name = "title", column = "b_title"),
                @FieldResult(name = "author", column = "b_author_id")
            }
        ),
        @EntityResult(
            entityClass = JPAAuthor.class,
            fields = {
                @FieldResult(name = "id", column = "a_id"),
                @FieldResult(name = "firstName", column = "a_first_name"),
                @FieldResult(name = "lastName", column = "a_last_name")
            }
        )
    }
)
```

Note how we need to map between:

- FieldResult.name(), which corresponds to the entity's attribute name
- FieldResult.column(), which corresponds to the SQL result's column name

With the above boilerplate in place, we can now fetch entities using jOOQ and JPA:

```
public static <E> List<E> nativeQuery(EntityManager em, org.jooq.Query query, String resultSetMapping) {

    // Extract the SQL statement from the jOOQ query:
    Query result = em.createNativeQuery(query.getSQL(), resultSetMapping);

    // Extract the bind values from the jOOQ query:
    List<Object> values = query.getBindValues();
    for (int i = 0; i < values.size(); i++) {
        result.setParameter(i + 1, values.get(i));
    }

    return result.getResultList();
}
```

Note, if you're using custom data types or bindings, make sure to take those into account as well. E.g. as follows:

```
public static <E> List<E> nativeQuery(EntityManager em, org.jooq.Query query, String resultSetMapping) {

    // Extract the SQL statement from the jOOQ query:
    Query result = em.createNativeQuery(query.getSQL(), resultSetMapping);

    // Extract the bind values from the jOOQ query:
    int i = 1;
    for (Param<?> param : query.getParams().values())
        if (!param.isInline())
            result.setParameter(i++, convertToDatabaseType(param));

    return result.getResultList();
}

static <T> Object convertToDatabaseType(Param<T> param) {
    return param.getBinding().converter().to(param.getValue());
}
```

Using the above API

Now that we have everything setup, we can use the above API to run a jOOQ query to fetch JPA entities like this:

```
List<Object[]> result =
nativeQuery(em,
    DSL.using(configuration
       .select(
           AUTHOR.ID.as("a_id"),
           AUTHOR.FIRST_NAME.as("a_first_name"),
           AUTHOR.LAST_NAME.as("a_last_name"),
           BOOK.ID.as("b_id"),
           BOOK.AUTHOR_ID.as("b_author_id"),
           BOOK.TITLE.as("b_title")
       )
       .from(AUTHOR)
       .join(BOOK).on(BOOK.AUTHOR_ID.eq(AUTHOR.ID))
       .orderBy(BOOK.ID)),
    "bookmapping" // The name of the SqlResultSetMapping
);

result.forEach((Object[] entities) -> {
    JPAAuthor author = (JPAAuthor) entities[1];
    JPABook book = (JPABook) entities[0];

    System.out.println(author.firstName + " " + author.lastName + " wrote " + book.title);
});
```

The entities are now ready to be modified and persisted again.

Caveats:

- We have to reference the result set mapping by name (a String) - there is no type safety involved here
- We don't know the type contained in the resulting List - there is a potential for ClassCastException
- The results are in fact a list of Object[], with the individual entities listed in the array, which need explicit casting

6. Code generation

While optional, source code generation is one of jOOQ's main assets if you wish to increase developer productivity. jOOQ's code generator takes your database schema and reverse-engineers it into a set of Java classes modelling tables, records, sequences, POJOs, DAOs, stored procedures, user-defined types and many more.

The essential ideas behind source code generation are these:

- Increased IDE support: Type your Java code directly against your database schema, with all type information available
- Type-safety: When your database schema changes, your generated code will change as well. Removing columns will lead to compilation errors, which you can detect early.

The following chapters will show how to configure the code generator and how to generate various artefacts.

6.1. Configuration and setup of the generator

There are three binaries available with jOOQ, to be downloaded from http://www.jooq.org/download or from Maven central:

- jooq-3.11.11.jar
  The main library that you will include in your application to run jOOQ
- jooq-meta-3.11.11.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-3.11.11.jar
  The utility that you will include in your build to generate your database schema

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 version="1.0" encoding="UTF-8" standalone="yes"?>
<configuration xmlns="http://www.jooq.org/xsd/jooq-codegen-3.11.0.xsd">
  <!-- 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>

    <!-- You can also pass user/password and other JDBC properties in the optional properties tag: -->
    <properties>
      <property><key>user</key><value>[db-user]</value></property>
      <property><key>password</key><value>[db-password]</value></property>
    </properties>
  </jdbc>

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

           Natively supported values are:

               org.jooq.meta.ase.ASEDatabase
               org.jooq.meta.auroramysql.AuroraMySQLDatabase
               org.jooq.meta.aurorapostgres.AuroraPostgresDatabase
               org.jooq.meta.cubrid.CUBRIDDatabase
               org.jooq.meta.db2.DB2Database
               org.jooq.meta.derby.DerbyDatabase
               org.jooq.meta.firebird.FirebirdDatabase
               org.jooq.meta.h2.H2Database
               org.jooq.meta.hana.HANADatabase
               org.jooq.meta.hsqldb.HSQLDBDatabase
               org.jooq.meta.informix.InformixDatabase
               org.jooq.meta.ingres.IngresDatabase
               org.jooq.meta.mariadb.MariaDBDatabase
               org.jooq.meta.mysql.MySQLDatabase
               org.jooq.meta.oracle.OracleDatabase
               org.jooq.meta.postgres.PostgresDatabase
               org.jooq.meta.redshift.RedshiftDatabase
               org.jooq.meta.sqldatawarehouse.SQLDataWarehouseDatabase
               org.jooq.meta.sqlite.SQLiteDatabase
               org.jooq.meta.sqlserver.SQLServerDatabase
               org.jooq.meta.sybase.SybaseDatabase
               org.jooq.meta.teradata.TeradataDatabase
               org.jooq.meta.vertica.VerticaDatabase

           This value can be used to reverse-engineer generic JDBC DatabaseMetaData (e.g. for MS Access)

               org.jooq.meta.jdbc.JDBCDatabase

           This value can be used to reverse-engineer standard jOOQ-meta XML formats

               org.jooq.meta.xml.XMLDatabase

           This value can be used to reverse-engineer schemas defined by SQL files (requires jooq-meta-extensions dependency)

               org.jooq.meta.extensions.ddl.DDLDatabase

           This value can be used to reverse-engineer schemas defined by JPA annotated entities (requires jooq-meta-extensions
 dependency)

               org.jooq.meta.extensions.jpa.JPADatabase

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

      <!-- All elements that are generated from your schema (A Java regular expression.
           Use the pipe to separate several expressions) 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)

           Whitespace is ignored and comments are possible.
           -->
      <includes>.*</includes>

      <!-- All elements that are excluded from your schema (A Java regular expression.
           Use the pipe to separate several expressions). Excludes match before
           includes, i.e. excludes have a higher priority -->
      <excludes>
           UNUSED_TABLE               # This table (unqualified name) should not be generated
         | PREFIX_.*                  # Objects with a given prefix should not be generated
         | SECRET_SCHEMA\.SECRET_TABLE # This table (qualified name) should not be generated
         | SECRET_ROUTINE             # This routine (unqualified name) ...
      </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.

           If left empty, jOOQ will generate all available schemata. See the
           manual's next section to learn how to generate several schemata -->
      <inputSchema>[your database schema / owner / name]</inputSchema>
    </database>

    <generate>
      <!-- Generation flags: See advanced configuration properties -->
    </generate>

    <target>
      <!-- The destination package of your generated classes (within the
           destination directory)

           jOOQ may append the schema name to this package if generating multiple schemas,
           e.g. org.jooq.your.packagename.schema1
```

There are also lots of advanced configuration parameters, which will be treated in the manual's section about advanced code generation features Note, you can find the official XSD file for a formal specification at:
http://www.jooq.org/xsd/jooq-codegen-3.11.0.xsd

Run jOOQ code generation

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

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

Be sure that these elements are located on the classpath:

- The XML configuration file
- jooq-3.11.11.jar, jooq-meta-3.11.11.jar, jooq-codegen-3.11.11.jar
- The JDBC driver you configured

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

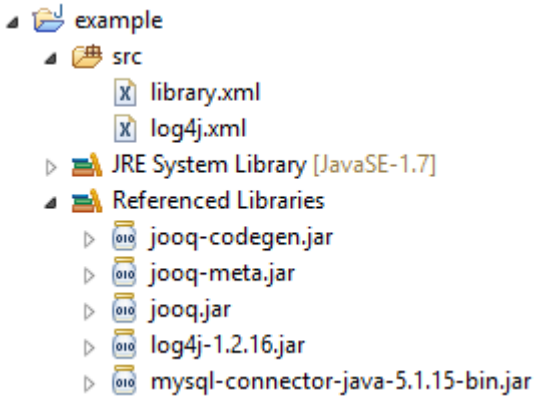
- Put the property file, jooq*.jar and the JDBC driver into a directory, e.g. C:\temp\jooq
- Go to C:\temp\jooq
- Run java -cp jooq-3.11.11.jar;jooq-meta-3.11.11.jar;jooq-codegen-3.11.11.jar;[JDBC-driver].jar;. org.jooq.codegen.GenerationTool /[XML file]

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

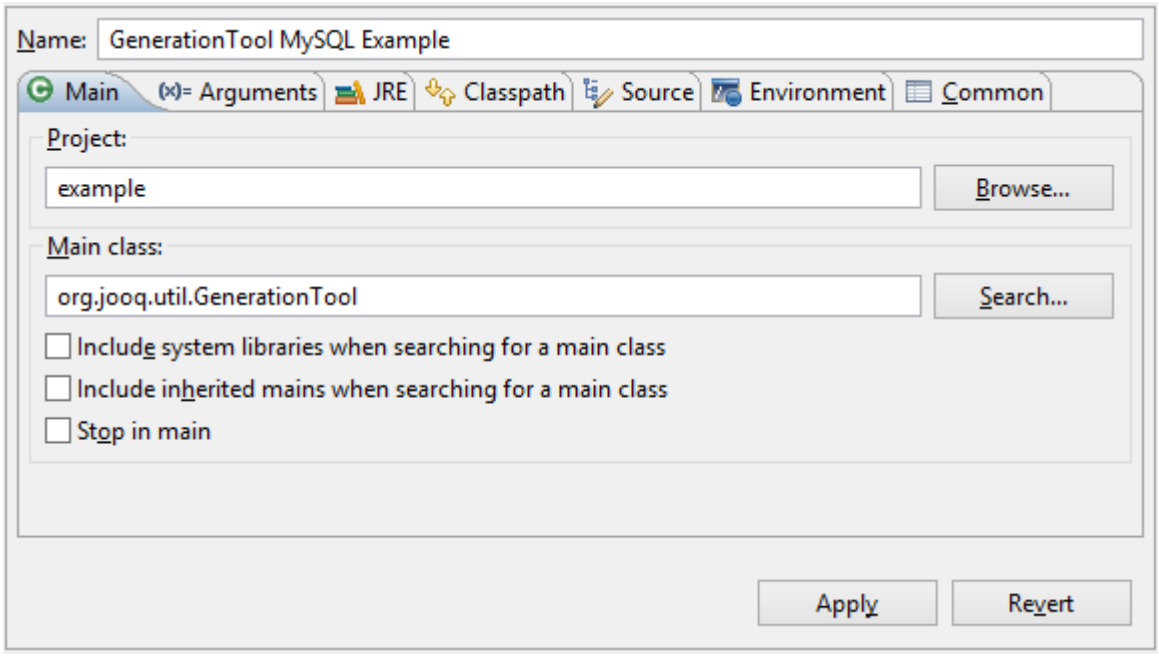
Run code generation from Eclipse

Of course, you can also run code generation from your IDE. In Eclipse, set up a project like this. Note that:

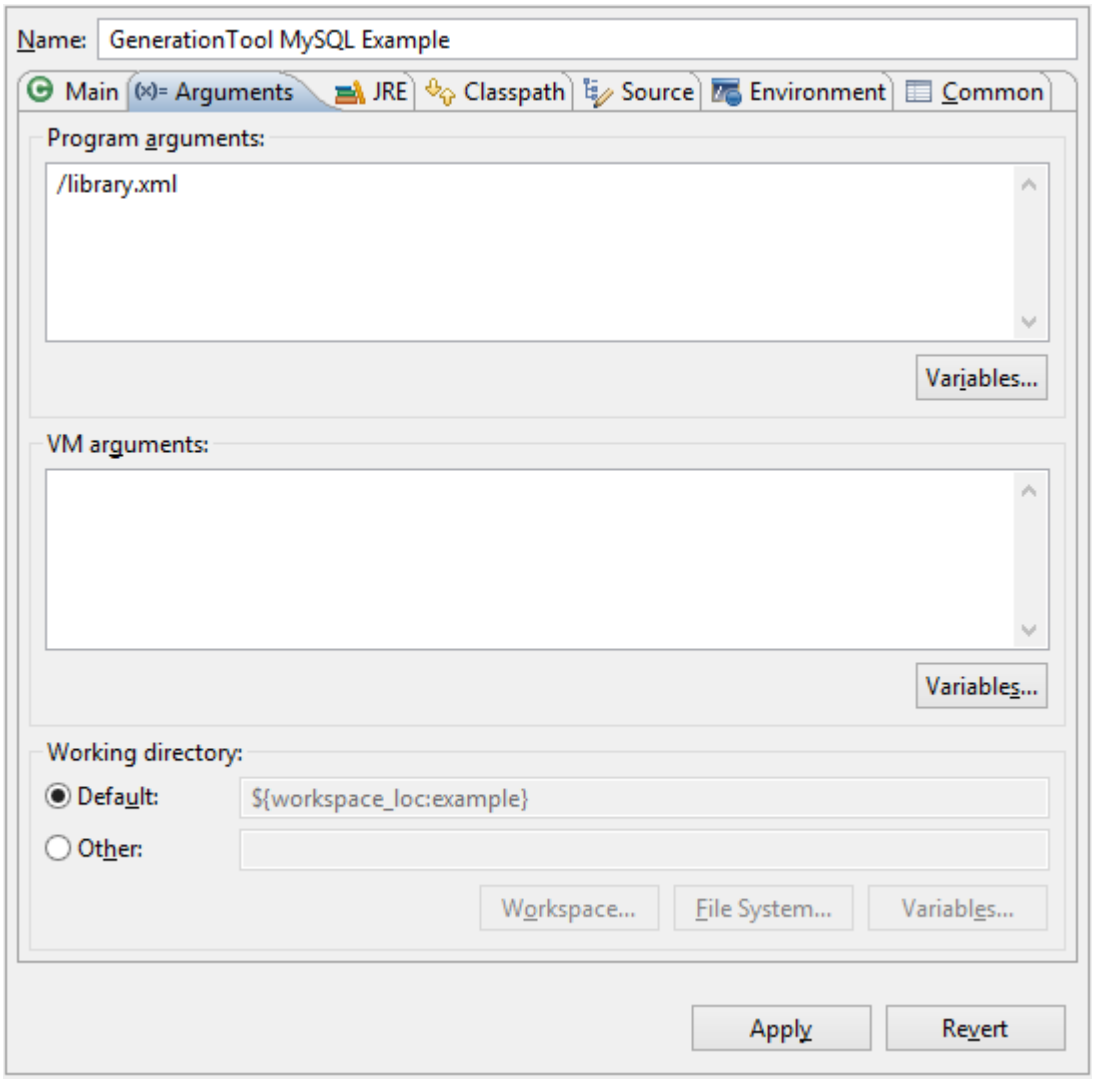
- this example uses jOOQ's log4j support by adding log4j.xml and log4j.jar to the project classpath.
- the actual jooq-3.11.11.jar, jooq-meta-3.11.11.jar, jooq-codegen-3.11.11.jar artefacts may contain version numbers in the file names.



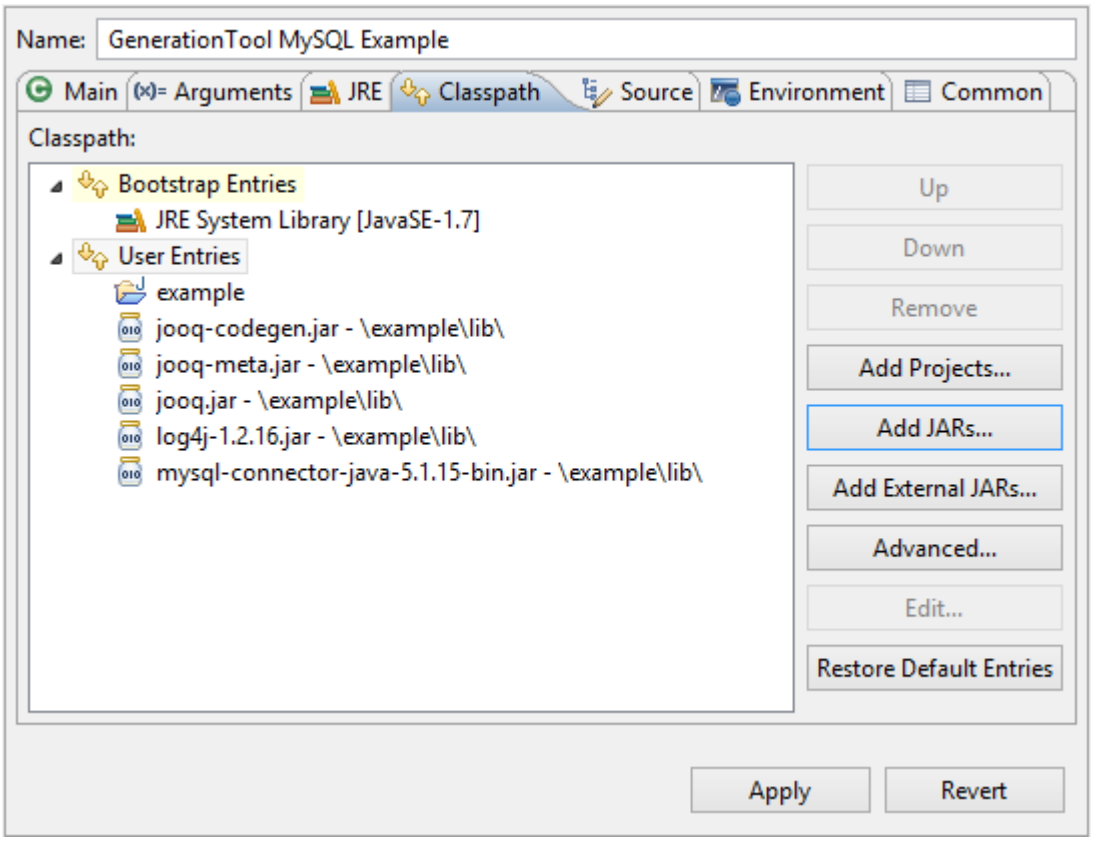
Once the project is set up correctly with all required artefacts on the classpath, you can configure an Eclipse Run Configuration for org.jooq.codegen.GenerationTool.



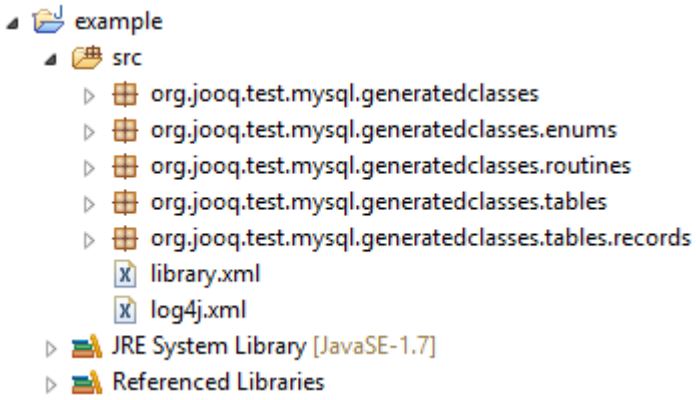
With the XML file as an argument



And the classpath set up correctly



Finally, run the code generation and see your generated artefacts



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 -->
  <!-- Use org.jooq            for the Open Source Edition
          org.jooq.pro        for commercial editions,
          org.jooq.pro-java-6 for commercial editions with Java 6 support,
          org.jooq.trial      for the free trial edition

       Note: Only the Open Source Edition is hosted on Maven Central.
             Import the others manually from your distribution -->
  <groupId>org.jooq</groupId>
  <artifactId>jooq-codegen-maven</artifactId>
  <version>3.11.11</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 PostgreSQL database -->
  <dependencies>
    <dependency>
      <groupId>org.postgresql</groupId>
      <artifactId>postgresql</artifactId>
      <version>9.4.1212</version>
    </dependency>
  </dependencies>

  <!-- Specify the plugin configuration.
       The configuration format is the same as for the standalone code generator -->
  <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>
      <database>
        <name>org.jooq.meta.postgres.PostgresDatabase</name>
        <includes>.*</includes>
        <excludes></excludes>
        <!-- In case your database supports catalogs, e.g. SQL Server:
        <inputCatalog>public</inputCatalog>
          -->
        <inputSchema>public</inputSchema>
      </database>
      <target>
        <packageName>org.jooq.codegen.maven.example</packageName>
        <directory>target/generated-sources/jooq</directory>
      </target>
    </generator>
  </configuration>
</plugin>
```

See a more complete example of a Maven pom.xml File in the jOOQ / Spring tutorial.

Use jOOQ generated classes in your application

Be sure, both jooq-3.11.11.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.

6.2. Advanced generator configuration

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 cover some advanced settings, individually.

6.2.1. Logging

This optional top level configuration element simply allows for overriding the log level of anything that has been specified by the runtime, e.g. in log4j or slf4j and is helpful for per-code-generation log configuration. For example, in order to mute everything that is less than WARN level, write:

XML configuration (standalone and Maven)

```
<configuration xmlns="http://www.jooq.org/xsd/jooq-codegen-3.11.0.xsd">
  <logging>WARN</logging>
  ...
</configuration>
```

Programmatic configuration

```
configuration.withLogging(Logging.WARN);
```

Gradle configuration

```
myConfigurationName(sourceSets.main) {
  logging = 'WARN'
}
```

Available log levels are

- TRACE
- DEBUG
- INFO
- WARN
- ERROR
- FATAL

6.2.2. Jdbc

This optional top level configuration element allows for configuring a JDBC connection. By default, the jOOQ code generator requires an active JDBC connection to reverse engineer your database schema. For example, if you want to connect to a MySQL database, write this:

XML configuration (standalone and Maven)

```
<configuration xmlns="http://www.jooq.org/xsd/jooq-codegen-3.11.0.xsd">
  <jdbc>
    <driver>com.mysql.jdbc.Driver</driver>
    <url>jdbc:mysql://localhost/testdb</url>

    <!--
    <username/> is a valid synonym for <user/>
    -->

    <user>root</user>
    <password>secret</password>
  </jdbc>
  ...
</configuration>
```

Programmatic configuration

```
configuration
  .withJdbc(new Jdbc()
    .withDriver("com.mysql.jdbc.Driver")
    .withUrl("jdbc:mysql://localhost/testdb")
    .withUser("root")
    .withPassword("secret"));
```

Note that when using the programmatic configuration API through the GenerationTool, you can also pass a pre-existing JDBC connection to the GenerationTool and leave this configuration element alone.

Gradle configuration

```
myConfigurationName(sourceSets.main) {
  jdbc {
    driver = 'com.mysql.jdbc.Driver'
    url = 'jdbc:mysql://localhost/testdb'
    user = 'root'
    password = 'secret'
  }
}
```

Optional JDBC properties

JDBC drivers allow for passing java.util.Properties to the JDBC driver when creating a connection. This is also supported in the code generator configuration with a list of key/value pairs as follows:

XML configuration (standalone and Maven)

```
<configuration xmlns="http://www.jooq.org/xsd/jooq-codegen-3.11.0.xsd">
  <jdbc>
    <driver>com.mysql.jdbc.Driver</driver>
    <url>jdbc:mysql://localhost/testdb</url>
    <properties>
      <property>
        <key>user</key>
        <value>root</value>
      </property>
      <property>
        <key>password</key>
        <value>secret</value>
      </property>
    </properties>
  </jdbc>
  ...
</configuration>
```

Programmatic configuration

```
configuration
  .withJdbc(new Jdbc()
    .withDriver("com.mysql.jdbc.Driver")
    .withUrl("jdbc:mysql://localhost/testdb")
    .withProperties(
      new Property().withKey("user").withValue("root"),
      new Property().withKey("password").withValue("secret")));
```

Gradle configuration

```
myConfigurationName(sourceSets.main) {
  jdbc {
    driver = 'com.mysql.jdbc.Driver'
    url = 'jdbc:mysql://localhost/testdb'
    properties {
      property {
        key = 'user'
        value = 'root'
      }
      property {
        key = 'password'
        value = 'secret'
      }
    }
  }
}
```

Using variables in Maven

Quite often, you'd like to keep passwords (or other elements) out of your configuration, or you'd like to repeat common values for different plugins, e.g. for both jOOQ and Flyway. In those cases, the programmatic configuration makes it easiest to quickly replace them, because those values are just ordinary Java local variables for that API. When you use the XML configuration with Maven, you can also very simply use Maven's property mechanism to achieve something like this:

```
<configuration xmlns="http://www.jooq.org/xsd/jooq-codegen-3.11.0.xsd">
  <jdbc>
    <driver>${db.driver}</driver>
    <url>${db.url}</url>
    <user>${db.user}</user>
    <password>${db.password}</password>
  </jdbc>
  ...
</configuration>
```

When the JDBC configuration is optional

There are some exceptions, where the JDBC connection does not need to be configured, for instance when using the JPADatabase (to reverse engineer JPA annotated entities) or when using the XMLDatabase (to reverse engineer an XML file). Please refer to the respective sections for more details.

6.2.3. Generator

This mandatory top level configuration element wraps all the remaining configuration elements related to code generation, including the overridable code generator class.

XML configuration (standalone and Maven)

```
<configuration xmlns="http://www.jooq.org/xsd/jooq-codegen-3.11.0.xsd">
  <generator>
    <!-- Optional: The fully qualified class name of the code generator. -->
    <name>...</name>

    <!-- Optional: The fully qualified class name of the generator strategy. -->
    <strategy>...</strategy>

    <!-- Optional: The jooq-meta configuration, configuring the information schema source. -->
    <database>...</database>

    <!-- Optional: The jooq-codegen configuration, configuring the generated output content. -->
    <generate>...</generate>

    <!-- Optional: The generation output target -->
    <target>...</target>
  </generator>
</configuration>
```

Programmatic configuration

```
configuration
  .withGenerator(new Generator(
    .withName("...")
    .withStrategy(new Strategy())
    .withDatabase(new Database())
    .withGenerate(new Generate())
    .withTarget(new Target())));
```

Gradle configuration

```
myConfigurationName(sourceSets.main) {
  generator {
    name = '...'
    strategy {
      ...
    }
    database {
      ...
    }
    generate {
      ...
    }
    target {
      ...
    }
  }
}
```

Specifying your own generator

The <name/> element allows for specifying a user-defined generator implementation. This is mostly useful when generating custom code sections, which can be added programmatically using the code generator's internal API. For more details, please refer to the relevant section of the manual.

Specifying a strategy

jOOQ by default applies standard Java naming schemes: PascalCase for classes, camelCase for members, methods, variables, parameters, UPPER_CASE_WITH_UNDERSCORES for constants and other literals. This may not be the desired default for your database, e.g. when you strongly rely on case-sensitive naming and if you wish to be able to search for names both in your Java code and in your database code (scripts, views, stored procedures) uniformly. For that purpose, you can override the <strategy/> element with your own implementation, either:

- programmatically
- configuratively

For more details, please refer to the relevant sections, above.

jooq-meta and jooq-codegen configuration

6.2.4. Database

This element wraps all the configuration elements that are used for the jooq-meta module, which reads the configured database meta data. In its simplest form, it can be left empty, when meaningful defaults will apply.

The two main elements in the <database/> element are <name/> and <properties>, which specify the class to implement the database meta data source, and an optional list of key/value parameters, which are described in the next chapter

Subsequent elements are:

6.2.4.1. Database name and properties

The two main elements in the <database/> element are <name/> and <properties>, which specify the class to implement the database meta data source, and an optional list of key/value parameters:

XML configuration (standalone and Maven)

```
<configuration xmlns="http://www.jooq.org/xsd/jooq-codegen-3.11.0.xsd">
  <generator>
    <database>
      <name>org.jooq.meta.xml.XMLDatabase</name>
      <properties>
        <property>
          <key>dialect</key>
          <value>MYSQL</value>
        </property>
        <property>
          <key>xml-file</key>
          <value>/path/to/database.xml</value>
        </property>
      </properties>
    </database>
  </generator>
</configuration>
```

Programmatic configuration

```
configuration
  .withGenerator(new Generator(
    .withDatabase(new Database()
      .withName("org.jooq.meta.xml.XMLDatabase")
      .withProperties(
        new Property().withKey("dialect").withValue("MYSQL"),
        new Property().withKey("xml-file").withValue("/path/to/database.xml")))));
```

Gradle configuration

```
myConfigurationName(sourceSets.main) {
  generator {
    database {
      name = 'org.jooq.meta.xml.XMLDatabase'
      properties {
        property {
          key = 'dialect'
          value = 'MYSQL'
        }
        property {
          key = 'xml-file'
          value = '/path/to/database.xml'
        }
      }
    }
  }
}
```

The default <name/> if no name is supplied will be derived from the JDBC connection. If you want to specifically specify your SQL dialect's database name, any of these values will be supported by jOOQ, out of the box:

- org.jooq.meta.ase.ASEDatabase
- org.jooq.meta.cubrid.CUBRIDDatabase
- org.jooq.meta.db2.DB2Database
- org.jooq.meta.derby.DerbyDatabase
- org.jooq.meta.firebird.FirebirdDatabase
- org.jooq.meta.h2.H2Database
- org.jooq.meta.hana.HanaDatabse
- org.jooq.meta.hsqldb.HSQLDBDatabase
- org.jooq.meta.informix.InformixDatabase
- org.jooq.meta.ingres.IngresDatabase
- org.jooq.meta.mariadb.MariaDBDatabase
- org.jooq.meta.mysql.MySQLDatabase
- org.jooq.meta.oracle.OracleDatabase
- org.jooq.meta.postgres.PostgresDatabase
- org.jooq.meta.redshift.RedshiftDatabase
- org.jooq.meta.sqlite.SQLiteDatabase
- org.jooq.meta.sqlserver.SQLServerDatabase
- org.jooq.meta.sybase.SybaseDatabase
- org.jooq.meta.vertica.VerticaDatabase

Alternatively, you can also specify the following database if you want to reverse-engineer a generic JDBC java.sql.DatabaseMetaData source for an unsupported database version / dialect / etc:

- org.jooq.meta.jdbc.JDBCDatabase

Furthermore, there are two out-of-the-box database meta data sources, that do not rely on a JDBC connection: the JPADatabase (to reverse engineer JPA annotated entities) and the XMLDatabase (to reverse engineer an XML file). Please refer to the respective sections for more details.

Last, but not least, you can of course implement your own by implementing org.jooq.meta.Database from the jooq-meta module.

6.2.4.2. RegexFlags

A lot of configuration elements rely on regular expressions. The most prominent examples are the useful includes and excludes elements. All of these regular expressions use the Java java.util.regex.Pattern API, with all of its features. The Pattern API allows for specifying flags and for your configuration convenience, the applied flags are, by default:

- COMMENTS: This allows for embedding comments (and, as a side-effect: meaningless whitespace) in regular expressions, which makes them much more readable.
- CASE_INSENSITIVE: Most schemas are case insensitive, so case-sensitive regular expressions are a bit of a pain, especially in multi-vendor setups, where databases like PostgreSQL (mostly lower case) and Oracle (mostly UPPER CASE) need to be supported simultaneously.

But of course, this default setting may get in your way, for instance if you rely on case sensitive identifiers and whitespace in identifiers a lot, it might be better for you to turn off the above defaults:

XML configuration (standalone and Maven)

```
<configuration xmlns="http://www.jooq.org/xsd/jooq-codegen-3.11.0.xsd">
  <generator>
    <database>
      <regexFlags>COMMENTS DOTALL</regexFlags>
    </database>
  </generator>
</configuration>
```

Programmatic configuration

```
configuration
  .withGenerator(new Generator(
    .withDatabase(new Database()
      .withRegexFlags("COMMENTS DOTALL"))));
```

Gradle configuration

```
myConfigurationName(sourceSets.main) {
  generator {
    database {
      regexFlags = 'COMMENTS DOTALL'
    }
  }
}
```

All the flags available from java.util.regex.Pattern are available as a whitespace-separated list.

6.2.4.3. Includes and Excludes

Perhaps the most important elements of the code generation configuration are used for the inclusion and exclusion of content as reported by your database meta data configuration

These expressions match any of the following object types, either by their fully qualified names (catalog.schema.object_name), or by their names only (object_name):

- Array types
- Domains
- Enums
- Links
- Packages
- Queues
- Routiens
- Sequences
- Tables
- UDTs

Excludes match ***before*** includes, meaning that something that has been excluded cannot be included again. Remember, these expressions are regular expressions with default flags, so multiple names need to be separated with the pipe symbol "|", not with commas, etc. For example:

XML configuration (standalone and Maven)

```
<configuration xmlns="http://www.jooq.org/xsd/jooq-codegen-3.11.0.xsd">
  <generator>
    <database>
      <includes>.*</includes>
      <excludes>
           UNUSED_TABLE               # This table (unqualified name) should not be generated
         | PREFIX_.*                  # Objects with a given prefix should not be generated
         | SECRET_SCHEMA\.SECRET_TABLE # This table (qualified name) should not be generated
         | SECRET_ROUTINE             # This routine (unqualified name) ...
      </excludes>
    </database>
  </generator>
</configuration>
```

Programmatic configuration

```
configuration
  .withGenerator(new Generator(
    .withDatabase(new Database()
      .withIncludes(".*")
      .withExcludes("UNUSED_TABLE|PREFIX_.*|SECRET_SCHEMA\\.SECRET_TABLE|SECRET_ROUTINE"))));
```

Gradle configuration

```
myConfigurationName(sourceSets.main) {
  generator {
    database {
      includes = '.*'
      excludes = 'UNUSED_TABLE|PREFIX_.*|SECRET_SCHEMA\\.SECRET_TABLE|SECRET_ROUTINE'
    }
  }
}
```

A special, additional option allows for specifying whether the above two regular expressions should also match table columns. The following example will hide an INVISIBLE_COL in any table (and also tables called this way, of course):

XML configuration (standalone and Maven)

```
<configuration xmlns="http://www.jooq.org/xsd/jooq-codegen-3.11.0.xsd">
  <generator>
    <database>
      <includes>.*</includes>
      <excludes>INVISIBLE_COL</excludes>
      <includeExcludeColumns>true</includeExcludeColumns>
    </database>
  </generator>
</configuration>
```

Programmatic configuration

```
configuration
  .withGenerator(new Generator(
    .withDatabase(new Database()
      .withIncludes(".*")
      .withExcludes("INVISIBLE_COL")
      .withIncludeExcludeColumns(true))));
```

Gradle configuration

```
myConfigurationName(sourceSets.main) {
  generator {
    database {
      includes = '.*'
      excludes = 'INVISIBLE_COL'
      includeExcludeColumns = true
    }
  }
}
```

6.2.4.4. Include object types

Sometimes, you want to generate only tables. Or only routines. Or you want to exclude them from being generated. Whatever the use-case, there's a way to do this with the following, additional includes flags:

XML configuration (standalone and Maven)

```
<configuration xmlns="http://www.jooq.org/xsd/jooq-codegen-3.11.0.xsd">
  <generator>
    <database>
      <includeTables>true</includeTables>
      <includeRoutines>true</includeRoutines>
      <includePackages>true</includePackages>
      <includePackageRoutines>true</includePackageRoutines>
      <includePackageUDTs>true</includePackageUDTs>
      <includePackageConstants>true</includePackageConstants>
      <includeUDTs>true</includeUDTs>
      <includeSequences>false</includeSequences>
      <includePrimaryKeys>false</includePrimaryKeys>
      <includeUniqueKeys>false</includeUniqueKeys>
      <includeForeignKeys>false</includeForeignKeys>
      <includeIndexes>false</includeIndexes>
    </database>
  </generator>
</configuration>
```

Programmatic configuration

```
configuration
  .withGenerator(new Generator(
    .withDatabase(new Database()
      .withIncludeTables(true)
      .withIncludeRoutines(true)
      .withIncludePackages(true)
      .withIncludePackageRoutines(true)
      .withIncludePackageUDTs(true)
      .withIncludePackageConstants(true)
      .withIncludeUDTs(true)
      .withIncludeSequences(false)
      .withIncludePrimaryKeys(false)
      .withIncludeUniqueKeys(false)
      .withIncludeForeignKeys(false)
      .withIncludeIndexes(false))));
```

Gradle configuration

```
myConfigurationName(sourceSets.main) {
  generator {
    database {
      includeTables = true
      includeRoutines = true
      includePackages = true
      includePackageRoutines = true
      includePackageUDTs = true
      includePackageConstants = true
      includeUDTs = true
      includeSequences = false
      includePrimaryKeys = false
      includeUniqueKeys = false
      includeForeignKeys = false
      includeIndexes = false
    }
  }
}
```

By default, all these flags are set to true.

6.2.4.5. Record Version and Timestamp Fields

jOOQ's org.jooq.UpdatableRecord supports an optimistic locking feature, which can be enabled in the code generator by specifying a regular expression that defines such a record's version and/or

timestamp fields. These regular expressions should match at most one column per table, again either by their fully qualified names (catalog.schema.table.column_name) or by their names only (column_name):

XML configuration (standalone and Maven)

```
<configuration xmlns="http://www.jooq.org/xsd/jooq-codegen-3.11.0.xsd">
  <generator>
    <database>
      <recordVersionFields>REC_VERSION</recordVersionFields>
      <recordTimestampFields>REC_TIMESTAMP</recordTimestampFields>
    </database>
  </generator>
</configuration>
```

Programmatic configuration

```
configuration
  .withGenerator(new Generator(
    .withDatabase(new Database()
      .withRecordVersionFields("REC_VERSION")
      .withRecordTimestampFields("REC_TIMESTAMP"))));
```

Gradle configuration

```
myConfigurationName(sourceSets.main) {
  generator {
    database {
      recordVersionFields = 'REC_VERSION'
      recordTimestampFields = 'REC_TIMESTAMP'
    }
  }
}
```

Note again that these expressions are regular expressions with default flags

6.2.4.6. Synthetic identities

jOOQ's code generator recognises identity columns if they are reported as such by the database. Some databases do not support "real" identity columns, but allow for emulating them, e.g. through triggers and sequences (e.g. Oracle prior to 12c). If a column is a known "identity" without formally being one, users can specify a regular expression that matches all columns (one column per table), which will be treated as if they were formal identities. For example:

XML configuration (standalone and Maven)

```
<configuration xmlns="http://www.jooq.org/xsd/jooq-codegen-3.11.0.xsd">
  <generator>
    <database>
      <syntheticIdentities>SCHEMA\.TABLE\.ID</syntheticIdentities>
    </database>
  </generator>
</configuration>
```

Programmatic configuration

```
configuration
  .withGenerator(new Generator(
    .withDatabase(new Database()
      .withSyntheticIdentities("SCHEMA\.TABLE\.ID"))));
```

Gradle configuration

```
myConfigurationName(sourceSets.main) {
  generator {
    database {
      syntheticIdentities = 'SCHEMA\.TABLE\.ID'
    }
  }
}
```

Note again that these expressions are regular expressions with default flags

6.2.4.7. Synthetic primary keys

jOOQ's code generator recognises primary keys that are declared and reported as such by the database. But some databases don't report all keys, or some tables don't have them enabled, or sometimes, a view is a 1:1 representation of an underlying table, but it doesn't expose the key information. In these cases, this regular expression can match all columns that users wish to "pretend" are part of such a primary key. If a composite synthetic primary key is desired, the regular expression should match all columns of that table that are part of the primary key. For example, a composite synthetic primary key consists of (COLUMN1, COLUMN2) in table SCHEMA.TABLE:

XML configuration (standalone and Maven)

```
<configuration xmlns="http://www.jooq.org/xsd/jooq-codegen-3.11.0.xsd">
  <generator>
    <database>
      <syntheticPrimaryKeys>SCHEMA\.TABLE\.COLUMN(1|2)</syntheticPrimaryKeys>
    </database>
  </generator>
</configuration>
```

Programmatic configuration

```
configuration
  .withGenerator(new Generator(
    .withDatabase(new Database()
      .withSyntheticPrimaryKeys("SCHEMA\.TABLE\.COLUMN(1|2)"))));
```

Gradle configuration

```
myConfigurationName(sourceSets.main) {
  generator {
    database {
      syntheticPrimaryKeys = 'SCHEMA\.TABLE\.COLUMN(1|2)'
    }
  }
}
```

If the regular expression matches column in a table that already has an existing primary key, that existing primary key will be replaced by the synthetic one. It will still be reported as a unique key, though.

Note again that these expressions are regular expressions with default flags

6.2.4.8. Override primary keys

In some legacy contexts, there might not be any primary key, only a bunch of unique keys, defined in a table. jOOQ cannot pick one of those as the "primary key", and the table won't be generated as a org.jooq.UpdatableRecord. Users can, however, use this regular expression to match all UNIQUE key names that should be used as a table's primary key.

XML configuration (standalone and Maven)

```
<configuration xmlns="http://www.jooq.org/xsd/jooq-codegen-3.11.0.xsd">
  <generator>
    <database>
      <overridePrimaryKeys>MY_UNIQUE_KEY_NAME</overridePrimaryKeys>
    </database>
  </generator>
</configuration>
```

Programmatic configuration

```
configuration
  .withGenerator(new Generator(
    .withDatabase(new Database()
      .withOverridePrimaryKeys("MY_UNIQUE_KEY_NAME"))));
```

Gradle configuration

```
myConfigurationName(sourceSets.main) {
  generator {
    database {
      overridePrimaryKeys = 'MY_UNIQUE_KEY_NAME'
    }
  }
}
```

If several keys match, a warning is emitted and the first one encountered will be used. This flag will also replace synthetic primary keys, if it matches.

Note again that these expressions are regular expressions with default flags

6.2.4.9. Date as timestamp

The Oracle database doesn't know a SQL standard DATE type (YYYY-MM-DD). It's vendor-specific DATE type is really a TIMESTAMP(0), i.e. a TIMESTAMP with zero fractional seconds precision (YYYY-MM-DD HH24:MI:SS). For historic reasons, many legacy Oracle databases do not use the TIMESTAMP data type, but the DATE data type for timestamps, in case of which client applications also need to treat these columns as timestamps.

If upgrading the schema to proper TIMESTAMP usage isn't an option, and neither is using data type rewrites on a per-column basis, then this flag is the right one to activate. It will remove the DATE type from the Oracle type system (at least as far as the jOOQ code generator is concerned), and pretend all such columns are really TIMESTAMP typed. This is how to activate the flag:

XML configuration (standalone and Maven)

```
<configuration xmlns="http://www.jooq.org/xsd/jooq-codegen-3.11.0.xsd">
  <generator>
    <database>
      <dateAsTimestamp>true</dateAsTimestamp>
    </database>
  </generator>
</configuration>
```

Programmatic configuration

```
configuration
  .withGenerator(new Generator(
    .withDatabase(new Database()
      .withDateAsTimestamp(true))));
```

Gradle configuration

```
myConfigurationName(sourceSets.main) {
  generator {
    database {
      dateAsTimestamp = true
    }
  }
}
```

This flag will apply before any other data type related flags are applied, including forced types.

6.2.4.10. Ignore procedure return values (deprecated)

In jOOQ 3.6.0, #4106 was implemented to support Transact-SQL's optional return values from stored procedures. This turns all procedures into Routine<Integer> (instead of Routine<Void>). For backwards-compatibility reasons, users can suppress this change in jOOQ 3.x

This feature is deprecated as of jOOQ 3.6.0 and will be removed again in jOOQ 4.0.

XML configuration (standalone and Maven)

```
<configuration xmlns="http://www.jooq.org/xsd/jooq-codegen-3.11.0.xsd">
  <generator>
    <database>
      <ignoreProcedureReturnValues>true</ignoreProcedureReturnValues>
    </database>
  </generator>
</configuration>
```

Programmatic configuration

```
configuration
  .withGenerator(new Generator(
    .withDatabase(new Database()
      .withIgnoreProcedureReturnValues(true))));
```

Gradle configuration

```
myConfigurationName(sourceSets.main) {
  generator {
    database {
      ignoreProcedureReturnValues = true
    }
  }
}
```

6.2.4.11. Unsigned types

The JDBC and Java type system don't know any unsigned integer data types, but some databases do, most importantly MySQL. This flag allows for overriding the default mapping from unsigned to signed integers and generates jOOU types instead:

- org.jooq.types.UByte
- org.jooq.types.UShort
- org.jooq.types.UInteger
- org.jooq.types.ULong

Those types work just like ordinary java.lang.Number wrapper types, except that there is no primitive version of them. The configuration looks like follows:

XML configuration (standalone and Maven)

```
<configuration xmlns="http://www.jooq.org/xsd/jooq-codegen-3.11.0.xsd">
  <generator>
    <database>
      <unsignedTypes>true</unsignedTypes>
    </database>
  </generator>
</configuration>
```

Programmatic configuration

```
configuration
  .withGenerator(new Generator(
    .withDatabase(new Database()
      .withUnsignedTypes(true))));
```

Gradle configuration

```
myConfigurationName(sourceSets.main) {
  generator {
    database {
      unsignedTypes = true
    }
  }
}
```

6.2.4.12. Catalog and schema mapping

These configuration elements combine two features in one:

- They allow for specifying one or more catalogs (default: all catalogs) as well as one or more schemas (default: all schemas) for inclusion in the code generator. This works in a similar fashion as the includes and excludes elements, but it is applied on an earlier stage.
- Once all "input" catalogs and schemas are specified, they can each be associated with a matching "output" catalog or schema, in case of which the "input" will be mapped to the "output" by the code generator. For more details about this, please refer to the manual section about schema mapping.

There are two ways to operate "input" and "output" catalogs and schemas configurations: "top level" and "nested". Note that catalogs are only supported in very few databases, so usually, users will only use the "input" and "output" schema feature.

Top level configurations

This mode is preferrable for small projects or quick tutorials, where only a single catalog and a/or a single schema need to be generated. In this case, the following "top level" configuration elements can be applied:

XML configuration (standalone and Maven)

```
<!-- Read only a single schema (from all catalogs, but in most databases, there is only one "default catalog") -->
<configuration xmlns="http://www.jooq.org/xsd/jooq-codegen-3.11.0.xsd">
  <generator>
    <database>
      <inputSchema>my_schema</inputSchema>
    </database>
  </generator>
</configuration>

<!-- Read only a single catalog and all its schemas -->
<configuration xmlns="http://www.jooq.org/xsd/jooq-codegen-3.11.0.xsd">
  <generator>
    <database>
      <inputCatalog>my_catalog</inputCatalog>
    </database>
  </generator>
</configuration>

<!-- Read only a single catalog and only a single schema -->
<configuration xmlns="http://www.jooq.org/xsd/jooq-codegen-3.11.0.xsd">
  <generator>
    <database>
      <inputCatalog>my_catalog</inputCatalog>
      <inputSchema>my_schema</inputSchema>
    </database>
  </generator>
</configuration>
```

Programmatic configuration

```
configuration
  .withGenerator(new Generator(
    .withDatabase(new Database()
      .withInputCatalog("my_catalog")
      .withInputSchema("my_schema"))));
```

Gradle configuration

```
myConfigurationName(sourceSets.main) {
  generator {
    database {
      inputCatalog = 'my_catalog'
      inputSchema = 'my_schema'
    }
  }
}
```

Nested configurations

This mode is preferrable for larger projects where several catalogs and/or schemas need to be included. The following examples show different possible configurations:

XML configuration (standalone and Maven)

```
<!-- Read two schemas (from all catalogs, but in most databases, there is only one "default catalog") -->
<configuration xmlns="http://www.jooq.org/xsd/jooq-codegen-3.11.0.xsd">
  <generator>
    <database>
      <schemata>
        <schema>
          <inputSchema>schema1</inputSchema>
        </schema>
        <schema>
          <inputSchema>schema2</inputSchema>
        </schema>
      </schemata>
    </database>
  </generator>
</configuration>

<!-- Read two catalogs and all their schemas -->
<configuration xmlns="http://www.jooq.org/xsd/jooq-codegen-3.11.0.xsd">
  <generator>
    <database>
      <catalogs>
        <catalog>
          <inputCatalog>catalog1</inputCatalog>
        </catalog>
        <catalog>
          <inputCatalog>catalog2</inputCatalog>
        </catalog>
      </catlogs>
    </database>
  </generator>
</configuration>

<!-- Read two schemas from one specific catalog -->
<configuration xmlns="http://www.jooq.org/xsd/jooq-codegen-3.11.0.xsd">
  <generator>
    <database>
      <catalogs>
        <catalog>
          <inputCatalog>catalog</inputCatalog>
          <schemata>
            <schema>
              <inputSchema>schema1</inputSchema>
            </schema>
            <schema>
              <inputSchema>schema2</inputSchema>
            </schema>
          </schemata>
        </catalog>
      </catlogs>
    </database>
  </generator>
</configuration>
```

Programmatic configuration

```
// Two schemas from any catalog
configuration
  .withGenerator(new Generator(
    .withDatabase(new Database()
      .withSchemata(
        new Schema().withInputSchema("schema1"),
        new Schema().withInputSchema("schema2")))));

// All schemas from two catalogs
configuration
  .withGenerator(new Generator(
    .withDatabase(new Database()
      .withCatalogs(
        new Catalog().withInputCatalog("catalog1"),
        new Catalog().withInputCatalog("catalog2")))));

// Two schemas from a specific catalog
configuration
  .withGenerator(new Generator(
    .withDatabase(new Database()
      .withCatalogs(new Catalog()
        .withInputCatalog("catalog")
        .withSchemata(
          new Schema().withInputSchema("schema1"),
          new Schema().withInputSchema("schema2"))))));
```

Gradle configuration

```
// Two schemas from any catalog
myConfigurationName(sourceSets.main) {
  generator {
    database {
      schemata {
        schema {
          inputSchema = 'schema1'
        }
        schema {
          inputSchema = 'schema2'
        }
      }
    }
  }
}

// All schemas from two catalogs
myConfigurationName(sourceSets.main) {
  generator {
    database {
      catalogs {
        catalog {
          inputCatalog = 'catalog1'
        }
        catalog {
          inputCatalog = 'catalog2'
        }
      }
    }
  }
}

// Two schemas from a specific catalog
myConfigurationName(sourceSets.main) {
  generator {
    database {
      catalogs {
        catalog {
          inputCatalog = 'catalog'
          schemata {
            schema {
              inputSchema = 'schema1'
            }
            schema {
              inputSchema = 'schema2'
            }
          }
        }
      }
    }
  }
}
```

Catalog and schema mapping

Wherever you can place an inputCatalog or inputSchema element (top level or nested), you can also put a matching mapping instruction, if you wish to profit from the catalog and schema mapping feature. The following configurations are possible:

XML configuration (standalone and Maven)

```
<!-- Map input names to a concrete output name: -->
<configuration xmlns="http://www.jooq.org/xsd/jooq-codegen-3.11.0.xsd">
  <generator>
    <database>
      <inputCatalog>my_input_catalog</inputCatalog>
      <outputCatalog>my_output_catalog</outputCatalog>
      <inputSchema>my_input_schema</inputSchema>
      <outputSchema>my_output_schema</outputSchema>
    </database>
  </generator>
</configuration>

<!-- Map input names to the "default" catalog or schema (i.e. no name): -->
<configuration xmlns="http://www.jooq.org/xsd/jooq-codegen-3.11.0.xsd">
  <generator>
    <database>
      <inputCatalog>my_input_catalog</inputCatalog>
      <outputCatalogToDefault>true</outputCatalogToDefault>
      <inputSchema>my_input_schema</inputSchema>
      <outputSchemaToDefault>true</outputSchemaToDefault>
    </database>
  </generator>
</configuration>
```

Programmatic configuration

```
// Map input names to a concrete output name:
configuration
  .withGenerator(new Generator(
    .withDatabase(new Database()
      .withInputCatalog("my_input_catalog")
      .withOutputCatalog("my_output_catalog")
      .withInputSchema("my_input_schema")
      .withOutputSchema("my_output_schema"))));

// Map input names to the "default" catalog or schema (i.e. no name):
configuration
  .withGenerator(new Generator(
    .withDatabase(new Database()
      .withInputCatalog("my_input_catalog")
      .withOutputCatalogToDefault(true)
      .withInputSchema("my_input_schema")
      .withOutputSchemaToDefault(true))));
```

Gradle configuration

```
// Map input names to a concrete output name:
myConfigurationName(sourceSets.main) {
  generator {
    database {
      inputCatalog = 'my_input_catalog'
      outputCatalog = 'my_output_catalog'
      inputSchema = 'my_input_schema'
      outputSchema = 'my_output_schema'
    }
  }
}

// Map input names to the "default" catalog or schema (i.e. no name):
myConfigurationName(sourceSets.main) {
  generator {
    database {
      inputCatalog = 'my_input_catalog'
      outputCatalogToDefault = true
      inputSchema = 'my_input_schema'
      outputSchemaToDefault = true
    }
  }
}
```

For more information about the catalog and schema mapping feature, please refer to the relevant section of the manual.

6.2.4.13. Catalog and schema version providers

For continuous integration reasons, users often like to version their database schemas, e.g. with tools like Flyway. In these cases, it is usually beneficial if the catalog and/or schema version can be placed with generated jOOQ code for documentation purposes and to prevent unnecessary re-generation of a catalog and/or schema.

For this reason, jOOQ allows for implementing a simple code generation SPI which tells jOOQ what the user-defined version of any given catalog or schema is.

There are three possible ways to implement this SPI:

- By providing a fully qualified class name that implements any of org.jooq.meta.CatalogVersionProvider or org.jooq.meta.SchemaVersionProvider respectively for programmatic version providing.
- By providing a SELECT statement returning one row with one column containing the version string. The SELECT statement may contain a named variable called :catalog_name or :schema_name respectively.
- By providing a constant, such as a Maven property, for instance.

These schema versions will be generated into the javax.annotation.Generated annotation on generated artefacts.

XML configuration (standalone and Maven)

```
<configuration xmlns="http://www.jooq.org/xsd/jooq-codegen-3.11.0.xsd">
  <generator>
    <database>
      <catalogVersionProvider>SELECT :catalog_name || '_' || MAX("version") FROM "schema_version"</catalogVersionProvider>
      <schemaVersionProvider>SELECT :schema_name || '_' || MAX("version") FROM "schema_version"</schemaVersionProvider>
    </database>
  </generator>
</configuration>
```

Programmatic configuration

```
configuration
  .withGenerator(new Generator(
    .withDatabase(new Database()
      .withCatalogVersionProvider("SELECT :catalog_name || '_' || MAX(\"version\") FROM \"schema_version\"")
      .withSchemaVersionProvider("SELECT :schema_name || '_' || MAX(\"version\") FROM \"schema_version\""))));
```

Gradle configuration

```
myConfigurationName(sourceSets.main) {
  generator {
    database {
      catalogVersionProvider = 'SELECT :catalog_name || '_' || MAX("version") FROM "schema_version"'
      schemaVersionProvider = 'SELECT :schema_name || '_' || MAX("version") FROM "schema_version"'
    }
  }
}
```

6.2.4.14. Custom ordering of generated code

By default, the jOOQ code generator maintains the following ordering of objects:

- Catalogs, schemas, tables, user-defined types, packages, routines, sequences, constraints are ordered alphabetically
- Table columns, user-defined type attributes, routine parameters are ordered in their order of definition

Sometimes, it may be desireable to override this default ordering to a custom ordering. In particular, the default ordering may be case-sensitive, when case-insensitive ordering is really more desireable at times. Users may define an order provider by specifying a fully qualified class on the code generator's class path, which must implement java.util.Comparator<org.jooq.meta.Definition> as follows:

XML configuration (standalone and Maven)

```
<configuration xmlns="http://www.jooq.org/xsd/jooq-codegen-3.11.0.xsd">
  <generator>
    <database>
      <orderProvider>com.example.CaseInsensitiveOrderProvider</orderProvider>
    </database>
  </generator>
</configuration>
```

Programmatic configuration

```
configuration
  .withGenerator(new Generator(
    .withDatabase(new Database()
      .withOrderProvider("com.example.CaseInsensitiveOrderProvider");
```

Gradle configuration

```
myConfigurationName(sourceSets.main) {
  generator {
    database {
      orderProvider = 'com.example.CaseInsensitiveOrderProvider'
    }
  }
}
```

This order provider may then look as follows:

```
package com.example;

import java.util.Comparator;

import org.jooq.meta.Definition;

public class CaseInsensitiveOrderProvider implements Comparator<Definition> {
    @Override
    public int compare(Definition o1, Definition o2) {
        return o1.getQualifiedInputName().compareToIgnoreCase(o2.getQualifiedInputName());
    }
}
```

While changing the order of "top level types" (like tables) is irrelevant to the jOOQ runtime, there may be some side-effects to changing the order of table columns, user-defined type attributes, routine parameters, as the database might expect the exact same order as is defined in the database. In order to only change the ordering for tables, the following order provider can be implemented instead:

```
package com.example;

import java.util.Comparator;

import org.jooq.meta.Definition;
import org.jooq.meta.TableDefinition;

public class CaseInsensitiveOrderProvider implements Comparator<Definition> {
    @Override
    public int compare(Definition o1, Definition o2) {
        if (o1 instanceof TableDefinition && o2 instanceof TableDefinition)
            return o1.getQualifiedInputName().compareToIgnoreCase(o2.getQualifiedInputName());
        else
            return 0; // Retain input ordering
    }
}
```

6.2.4.15. Forced types

The code generator allows users to override column data types in three different ways:

- By rewriting them to some other data type using the data type rewriting feature.
- By mapping them to some user type using the data type converter feature and a custom org.jooq.Converter.
- By mapping them to some user type using the data type binding feature and a custom org.jooq.Binding.

All of this can be done using forced types.

Data type rewriting

XML configuration (standalone and Maven)

```
<configuration xmlns="http://www.jooq.org/xsd/jooq-codegen-3.11.0.xsd">
  <generator>
    <database>
      <!-- The first matching forcedType will be applied to the data type definition. -->
      <forcedTypes>
        <forcedType>

          <!-- Specify any data type that is supported in your database, or if unsupported, a type from org.jooq.impl.SQLDataType -->
          <name>BOOLEAN</name>

          <!-- Add a Java regular expression matching fully-qualified columns. Use the pipe to separate several expressions.
               If provided, both "expressions" and "types" must match. -->
          <expression>.*\.IS_VALID</expression>

          <!-- Add a Java regular expression matching data types to be forced to have this type.

               Data types may be reported by your database as:
               - NUMBER              regexp suggestion: NUMBER
               - NUMBER(5)           regexp suggestion: NUMBER\(5\)
               - NUMBER(5, 2)        regexp suggestion: NUMBER\(5,\s*2\)
               - any other form.

               It is thus recommended to use defensive regexes for types.

               If provided, both "expressions" and "types" must match. -->
          <types>.*</types>

          <!-- Force a type depending on data type nullability. Default is ALL.

                - ALL - Force a type regardless of whether data type is nullable or not (default)
                - NULL - Force a type only when data type is nullable
                - NOT_NULL - Force a type only when data type is not null -->
          <nullability>ALL</nullability>
        </forcedType>
      </forcedTypes>
    </database>
  </generator>
</configuration>
```

Programmatic configuration

```
configuration
  .withGenerator(new Generator(
    .withDatabase(new Database()
      // The first matching forcedType will be applied to the data type definition.
      .withForcedTypes(new ForcedType()
        .withName("BOOLEAN")
        .withExpression(".*\.IS_VALID")
        .withTypes(".*")
        .withNullability(Nullability.ALL)))));
```

Gradle configuration

```
myConfigurationName(sourceSets.main) {
  generator {
    database {
      // The first matching forcedType will be applied to the data type definition.
      forcedTypes {
        forcedType {
          name = 'BOOLEAN'
          expression = '.*\.IS_VALID'
          types = '.*'
          nullability = ALL
        }
      }
    }
  }
}
```

Mapping to user type with a converter

Both user type mappings work the same way, regardless if you're using a org.jooq.Converter or a org.jooq.Binding.

XML configuration (standalone and Maven)

```
<configuration xmlns="http://www.jooq.org/xsd/jooq-codegen-3.11.0.xsd">
  <generator>
    <database>
      <!-- The first matching forcedType will be applied to the data type definition. -->
      <forcedTypes>
        <forcedType>

          <!-- Specify the Java type of your custom type. This corresponds to the Converter's <U> type. -->
          <userType>java.time.Instant</userType>

          <!-- Associate that custom type with your converter. -->
          <converter>com.example.LongToInstantConverter</converter>

          <!-- These are the same as for type rewriting -->
          <expression>.*\.DATE_OF_.*</expression>
          <types>.*</types>
        </forcedType>
      </forcedTypes>
    </database>
  </generator>
</configuration>
```

Programmatic configuration

```
configuration
  .withGenerator(new Generator(
    .withDatabase(new Database()
      // The first matching forcedType will be applied to the data type definition.
      .withForcedTypes(new ForcedType()
        .withUserType("java.time.Instant")
        .withConverter("com.example.LongToInstantConverter")
        .withExpression(".*\.DATE_OF_.*")
        .withTypes(".*")))));
```

Gradle configuration

```
myConfigurationName(sourceSets.main) {
  generator {
    database {
      // The first matching forcedType will be applied to the data type definition.
      forcedTypes {
        forcedType {
          userType = 'java.time.Instant'
          converter = 'com.example.LongToInstantConverter'
          expression = '.*\.DATE_OF_.*'
          types = '.*'
        }
      }
    }
  }
}
```

For more information about using converters, please refer to the manual's section about custom data type conversion.

Mapping to user type with an inline converter

For convenience, you can inline your converter code directly into the configuration instead of providing a class reference.

XML configuration (standalone and Maven)

```
<configuration xmlns="http://www.jooq.org/xsd/jooq-codegen-3.11.0.xsd">
  <generator>
    <database>
      <!-- The first matching forcedType will be applied to the data type definition. -->
      <forcedTypes>
        <forcedType>

          <!-- Specify the Java type of your custom type. This corresponds to the Converter's <U> type. -->
          <userType>com.example.MyEnum</userType>

          <!-- Associate that custom type with your inline converter. -->
          <converter>org.jooq.Converter.ofNullable(Integer.class, MyEnum.class, i -> MyEnum.values()[i], MyEnum::ordinal)</converter>

          <!-- These are the same as for type rewriting -->
          <expression>.*\.DATE_OF_.*</expression>
          <types>.*</types>
        </forcedType>
      </forcedTypes>
    </database>
  </generator>
</configuration>
```

Programmatic configuration

```
configuration
  .withGenerator(new Generator(
    .withDatabase(new Database()
      // The first matching forcedType will be applied to the data type definition.
      .withForcedTypes(new ForcedType()
        .withUserType("com.example.MyEnum")
        .withConverter("org.jooq.Converter.ofNullable(Integer.class, MyEnum.class, i -> MyEnum.values()[i], MyEnum::ordinal)")
        .withExpression(".*\.DATE_OF_.*")
        .withTypes(".*")))));
```

Gradle configuration

```
myConfigurationName(sourceSets.main) {
  generator {
    database {
      // The first matching forcedType will be applied to the data type definition.
      forcedTypes {
        forcedType {
          userType = 'com.example.MyEnum'
          converter = 'org.jooq.Converter.ofNullable(Integer.class, MyEnum.class, i -> MyEnum.values()[i], MyEnum::ordinal)'
          expression = '.*\.DATE_OF_.*'
          types = '.*'
        }
      }
    }
  }
}
```

Mapping to an enum user type with a converter

If your user type is a Java enum, you can use the <enumConverter/> convenience flag instead of an explicit converter per enum type. This will apply the built-in org.jooq.impl.EnumConverter.

XML configuration (standalone and Maven)

```
<configuration xmlns="http://www.jooq.org/xsd/jooq-codegen-3.11.0.xsd">
  <generator>
    <database>
      <!-- The first matching forcedType will be applied to the data type definition. -->
      <forcedTypes>
        <forcedType>

          <!-- Specify the Java type of your custom type. This corresponds to the Converter's <U> type. -->
          <userType>com.example.MyEnum</userType>

          <!-- Apply the built in org.jooq.impl.EnumConverter. -->
          <enumConverter>true</enumConverter>

          <!-- These are the same as for type rewriting -->
          <expression>.*\.MY_STATUS</expression>
          <types>.*</types>
        </forcedType>
      </forcedTypes>
    </database>
  </generator>
</configuration>
```

Programmatic configuration

```
configuration
  .withGenerator(new Generator(
    .withDatabase(new Database()
      // The first matching forcedType will be applied to the data type definition.
      .withForcedTypes(new ForcedType()
        .withUserType("com.example.MyEnum")
        .withEnumConverter(true)
        .withExpression(".*\.MY_STATUS")
        .withTypes(".*")))));
```

Gradle configuration

```
myConfigurationName(sourceSets.main) {
  generator {
    database {
      // The first matching forcedType will be applied to the data type definition.
      forcedTypes {
        forcedType {
          userType = 'com.example.MyEnum'
          enumConverter = true
          expression = '.*\.MY_STATUS'
          types = '.*'
        }
      }
    }
  }
}
```

Mapping to user type with a binding

Just switch the converter configuration to a binding configuration.

XML configuration (standalone and Maven)

```
<configuration xmlns="http://www.jooq.org/xsd/jooq-codegen-3.11.0.xsd">
  <generator>
    <database>
      <!-- The first matching forcedType will be applied to the data type definition. -->
      <forcedTypes>
        <forcedType>

          <!-- Specify the Java type of your custom type. This corresponds to the Binding's <U> type. -->
          <userType>java.time.Instant</userType>

          <!-- Associate that custom type with your converter. -->
          <binding>com.example.LongToInstantBinding</binding>

          <!-- These are the same as for type rewriting -->
          <expression>.*\.DATE_OF_.*</expression>
          <types>.*</types>
        </forcedType>
      </forcedTypes>
    </database>
  </generator>
</configuration>
```

Programmatic configuration

```
configuration
  .withGenerator(new Generator(
    .withDatabase(new Database()
      // The first matching forcedType will be applied to the data type definition.
      .withForcedTypes(new ForcedType()
        .withUserType("java.time.Instant")
        .withBinding("com.example.LongToInstantBinding")
        .withExpression(".*\.DATE_OF_.*")
        .withTypes(".*")))));
```

Gradle configuration

```
myConfigurationName(sourceSets.main) {
  generator {
    database {
      // The first matching forcedType will be applied to the data type definition.
      forcedTypes {
        forcedType {
          userType = 'java.time.Instant'
          binding = 'com.example.LongToInstantBinding'
          expression = '.*\.DATE_OF_.*'
          types = '.*'
        }
      }
    }
  }
}
```

For more information about using converters, please refer to the manual's section about custom data type bindings.

6.2.4.16. Table valued functions

jOOQ supports table valued functions in many databases, including Oracle, PostgreSQL, SQL Server. By default, table valued functions are treated as:

- ordinary tables in most databases including PostgreSQL, SQL Server - because that's what they are. They're intended for use in FROM clauses of SELECT statements, not as standalone routines.
- ordinary routines in some databases including Oracle - for historic reasons. While Oracle also allows for embedding (pipelined) table functions in FROM clauses of SELECT statements, it is not uncommon to call these as standalone routines in Oracle.

The <tableValuedFunctions/> flag is thus set to false by default on Oracle, and true otherwise. Here's how to explicitly change this behaviour:

XML configuration (standalone and Maven)

```
<configuration xmlns="http://www.jooq.org/xsd/jooq-codegen-3.11.0.xsd">
  <generator>
    <database>
      <tableValuedFunctions>true</tableValuedFunctions>
    </database>
  </generator>
</configuration>
```

Programmatic configuration

```
configuration
  .withGenerator(new Generator(
    .withDatabase(new Database()
      .withTableValuedFunctions(true))));
```

Gradle configuration

```
myConfigurationName(sourceSets.main) {
  generator {
    database {
      tableValuedFunctions = true
    }
  }
}
```

6.2.5. Generate

This element wraps all the configuration elements that are used for the jooq-codegen module, which generates Java or Scala code, or XML from your database.

Contained elements are:

6.2.5.1. Global Artefacts

For convenience, jOOQ generates a set of global artefacts, which group static constants of the same type in a well-known class. The following set of flags allows for turning off the generation of these artefacts, individually:

XML configuration (standalone and Maven)

```
<configuration xmlns="http://www.jooq.org/xsd/jooq-codegen-3.11.0.xsd">
  <generator>
    <generate>
      <!-- This overrides all the other individual flags -->
      <globalObjectReferences>true</globalObjectReferences>

      <!-- Individual flags for each object type -->
      <globalCatalogReferences>true</globalCatalogReferences>
      <globalSchemaReferences>true</globalSchemaReferences>
      <globalTableReferences>true</globalTableReferences>
      <globalSequenceReferences>true</globalSequenceReferences>
      <globalUDTReferences>true</globalUDTReferences>
      <globalRoutineReferences>true</globalRoutineReferences>
      <globalQueueReferences>true</globalQueueReferences>
      <globalLinkReferences>true</globalLinkReferences>
    </generate>
  </generator>
</configuration>
```

Programmatic configuration

```
configuration
  .withGenerator(new Generator(
    .withGenerate(new Generate()

      // This overrides all the other individual flags
      .withGlobalObjectReferences(true)

      // Individual flags for each object type
      .withGlobalCatalogReferences(true)
      .withGlobalSchemaReferences(true)
      .withGlobalTableReferences(true)
      .withGlobalSequenceReferences(true)
      .withGlobalUDTReferences(true)
      .withGlobalRoutineReferences(true)
      .withGlobalQueueReferences(true)
      .withGlobalLinkReferences(true))));
```

Gradle configuration

```
myConfigurationName(sourceSets.main) {
  generator {
    generate {

      // This overrides all the other individual flags
      globalObjectReferences = true

      // Individual flags for each object type
      globalCatalogReferences = true
      globalSchemaReferences = true
      globalTableReferences = true
      globalSequenceReferences = true
      globalUDTReferences = true
      globalRoutineReferences = true
      globalQueueReferences = true
      globalLinkReferences = true
    }
  }
}
```

6.2.5.2. Annotations

The code generator supports a set of annotations on generated code, which can be turned on using the following flags. These annotations include:

- JPA annotations: A minimal set of JPA annotations can be generated on POJOs and other artefacts to convey type and metadata information that is available to the code generator. These annotations include:

  * javax.persistence.Column
  * javax.persistence.Entity
  * javax.persistence.GeneratedValue
  * javax.persistence.GenerationType
  * javax.persistence.Id
  * javax.persistence.Index (JPA 2.1 and later)
  * javax.persistence.Table
  * javax.persistence.UniqueConstraint

  While jOOQ generated code cannot really be used as full-fledged entities (use e.g. Hibernate or EclipseLink to generate such entities), this meta information can still be useful as documentation on your generated code. Some of the annotations (e.g. @Column) can be used by the org.jooq.impl.DefaultRecordMapper for mapping records to POJOs.
- Validation annotations: A set of Bean Validation API annotations can be added to the generated code to convey type information. They include:

  * javax.validation.constraints.NotNull
  * javax.validation.constraints.Size

  jOOQ does implement the validation spec, nor does it validate your data, but you can use third-party tools to read the jOOQ-generated validation annotations.
- Spring annotations: Some useful Spring annotations can be generated on DAOs for better Spring integration. These include:

  * org.springframework.beans.factory.annotation.Autowired
  * org.springframework.stereotype.Repository

The flags governing the generation of these annotations are:

XML configuration (standalone and Maven)

```
<configuration xmlns="http://www.jooq.org/xsd/jooq-codegen-3.11.0.xsd">
  <generator>
    <generate>
      <jpaAnnotations>true</jpaAnnotations>
      <jpaVersion>2.2</jpaVersion>
      <validationAnnotations>true</validationAnnotations>
      <springAnnotations>true</springAnnotations>
    </generate>
  </generator>
</configuration>
```

Programmatic configuration

```
configuration
  .withGenerator(new Generator(
    .withGenerate(new Generate()
      .withJpaAnnotations(true)
      .withJpaVersion("2.2")
      .withValidationAnnotations(true)
      .withSpringAnnotations(true))));
```

Gradle configuration

```
myConfigurationName(sourceSets.main) {
  generator {
    generate {
      jpaAnnotations = true
      jpaVersion = '2.2'
      validationAnnotations = true
      springAnnotations = true
    }
  }
}
```

6.2.5.3. Java Time Types

With jOOQ 3.9, support for JSR-310 java.time types has been added to the jOOQ API and to the code generator. Users of Java 8 can now specify that the jOOQ code generator should prefer JSR 310 types over their equivalent JDBC types. This includes:

- java.time.LocalDate instead of java.sql.Date
- java.time.LocalTime instead of java.sql.Time
- java.time.LocalDateTime instead of java.sql.Timestamp

Semantically, the above types are exactly equivalent, although the new types do away with the many flaws of the JDBC types. If there is no JDBC type for an equivalent JSR 310 type, then the JSR 310 type is generated by default. This includes

- java.time.OffsetTime (for SQL TIME WITH TIME ZONE)
- java.time.OffsetDateTime (for SQL TIMESTAMP WITH TIME ZONE)

To get more fine-grained control of the above, you may wish to consider applying data type rewriting.

In order to activate the generation of these types, use:

XML configuration (standalone and Maven)

```
<configuration xmlns="http://www.jooq.org/xsd/jooq-codegen-3.11.0.xsd">
  <generator>
    <generate>
      <javaTimeTypes>true</javaTimeTypes>
    </generate>
  </generator>
</configuration>
```

Programmatic configuration

```
configuration
  .withGenerator(new Generator(
    .withGenerate(new Generate()
      .withJavaTimeTypes(true))));
```

Gradle configuration

```
myConfigurationName(sourceSets.main) {
  generator {
    generate {
      javaTimeTypes = true
    }
  }
}
```

6.2.5.4. Zero Scale Decimal Types

A zero-scale decimal, such as DECIMAL(10) or NUMBER(10, 0) is really an integer type with a decimal precision rather than a binary / bitwise precision. Some databases (e.g. Oracle) do not support actual integer types at all, only decimal types. Historically, jOOQ generates the most appropriate integer wrapper type instead of BigDecimal or BigInteger:

- NUMBER(2, 0) and less: java.lang.Byte
- NUMBER(4, 0) and less: java.lang.Short
- NUMBER(9, 0) and less: java.lang.Integer
- NUMBER(19, 0) and less: java.lang.Long

If this is not a desireable default, it can be deactivated either explicitly on a per-column basis using forced types, or globally using the following flag:

XML configuration (standalone and Maven)

```
<configuration xmlns="http://www.jooq.org/xsd/jooq-codegen-3.11.0.xsd">
  <generator>
    <database>
      <forceIntegerTypesOnZeroScaleDecimals>true</forceIntegerTypesOnZeroScaleDecimals>
    </database>
  </generator>
</configuration>
```

Programmatic configuration

```
configuration
  .withGenerator(new Generator(
    .withDatabase(new Database()
      .withForceIntegerTypesOnZeroScaleDecimals(true))));
```

Gradle configuration

```
myConfigurationName(sourceSets.main) {
  generator {
    database {
      forceIntegerTypesOnZeroScaleDecimals = true
    }
  }
}
```

6.2.5.5. Fully Qualified Types

By default, the jOOQ code generator references all types as unqualified types, generating the necessary import statement at the beginning of generated classes.

In rare cases, this can cause problems when two types conflict with each other, e.g. when there is both a TABLE and a TABLE_RECORD table (generating a TableRecord org.jooq.Record type for TABLE as well as a TableRecord org.jooq.Table type for TABLE_RECORD). In this case, users can specify a regular expression that matches all objects whose corresponding generated artefacts should never be imported, but always be fully qualified.

XML configuration (standalone and Maven)

```
<configuration xmlns="http://www.jooq.org/xsd/jooq-codegen-3.11.0.xsd">
  <generator>
    <generate>
      <fullyQualifiedTypes>.*\.MY_TABLE</fullyQualifiedTypes>
    </generate>
  </generator>
</configuration>
```

Programmatic configuration

```
configuration
  .withGenerator(new Generator(
    .withGenerate(new Generate()
      .withFullyQualifiedTypes(".*\.MY_TABLE"))));
```

Gradle configuration

```
myConfigurationName(sourceSets.main) {
  generator {
    generate {
      fullyQualifiedTypes = '.*\.MY_TABLE'
    }
  }
}
```

6.2.6. Output target configuration

In the previous sections, we've seen the <target/> element which configures the location of your generated output. The following XML snippet illustrates some additional flags that can be specified in that section:

- packageName: Specifies the root package name inside of which all generated code is located. This package is located inside of the <directory/>. The package name is part of the generator strategy and can be modified by a custom implementation, if so desired.
- directory: Specifies the root directoy inside of which all generated code is located.
- encoding: The encoding that should be used for generated classes.
- clean: Whether the target package (<packageName/>) should be cleaned to contain only generated code after a generation run. Defaults to true.

6.3. Programmatic generator configuration

Configuring your code generator with Java, Groovy, etc.

In the previous sections, we have covered how to set up jOOQ's code generator using XML, either by running a standalone Java application, or by using Maven. However, it is also possible to use jOOQ's GenerationTool programmatically. The XSD file used for the configuration (http://www.jooq.org/xsd/jooq-codegen-3.11.0.xsd) is processed using XJC to produce Java artefacts. The below example uses those artefacts to produce the equivalent configuration of the previous PostgreSQL / Maven example:

```
// Use the fluent-style API to construct the code generator configuration
import org.jooq.meta.jaxb.*;

// [...]

Configuration configuration = new Configuration()
    .withJdbc(new Jdbc()
        .withDriver("org.postgresql.Driver")
        .withUrl("jdbc:postgresql:postgres")
        .withUser("postgres")
        .withPassword("test"))
    .withGenerator(new Generator()
        .withDatabase(new Database()
            .withName("org.jooq.meta.postgres.PostgresDatabase")
            .withIncludes(".*")
            .withExcludes("")
            .withInputSchema("public"))
        .withTarget(new Target()
            .withPackageName("org.jooq.codegen.maven.example")
            .withDirectory("target/generated-sources/jooq")));

GenerationTool.generate(configuration);
```

For the above example, you will need all of jooq-3.11.11.jar, jooq-meta-3.11.11.jar, and jooq-codegen-3.11.11.jar, on your classpath.

Manually loading the XML file

Alternatively, you can also load parts of the configuration from an XML file using JAXB and programmatically modify other parts using the code generation API:

```
<?xml version="1.0" encoding="UTF-8" standalone="yes"?>
<configuration xmlns="http://www.jooq.org/xsd/jooq-codegen-3.11.0.xsd">
    <jdbc>
        <driver>org.h2.Driver</driver>
        <!-- ... -->
```

Load the above using standard JAXB API:

```
import java.io.File;
import javax.xml.bind.JAXB;
import org.jooq.meta.jaxb.Configuration;

// [...]
// and then

Configuration configuration = JAXB.unmarshal(new File("jooq.xml"), Configuration.class);
configuration.getJdbc()
             .withUser("username")
             .withPassword("password");

GenerationTool.generate(configuration);
```

... and then, modify parts of your configuration programmatically, for instance the JDBC user / password:

6.4. Custom generator strategies

Using custom generator strategies to override naming schemes

jOOQ allows you to override default implementations of the code generator or the generator strategy. Specifically, the latter can be very useful if you want to inject custom behaviour into jOOQ's code generator with respect to naming classes, members, methods, and other Java objects.

```
<!-- 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.codegen.JavaGenerator -->
  <name>org.jooq.codegen.JavaGenerator</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.codegen.DefaultGeneratorStrategy -->
  <strategy>
    <name>org.jooq.codegen.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:

```
/**
 * 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.
 *
 * Also, the DefaultGeneratorStrategy takes care of disambiguating quite a few object
 * names in case of conflict. For example, MySQL indexes do not really have a name, so
 * a synthetic, non-ambiguous name is generated based on the table. If you override
 * the default behaviour, you must ensure that this disambiguation still takes place
 * for generated code to be compilable.
 *
 * 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) {
        // The DefaultGeneratorStrategy disambiguates some synthetic object names,
        // such as the MySQL PRIMARY key names, which do not really have a name
        // Uncomment the below code if you want to reuse that logic.
        // if (definition instanceof IndexDefinition)
        //     return super.getJavaIdentifier(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();
    }

    /**
     * Override this method to define the base class for those artefacts that
     * allow for custom base classes
     */
    @Override
    public String getJavaClassExtends(Definition definition, Mode mode) {
        return Object.class.getName();
    }

    /**
     * Override this method to define the interfaces to be implemented by those
     * artefacts that allow for custom interface implementation
     */
    @Override
    public List<String> getJavaClassImplements(Definition definition, Mode mode) {
        return Arrays.asList(Serializable.class.getName(), Cloneable.class.getName());
    }

    /**
     * Override this method to define the suffix to apply to routines when
```

An org.jooq.Table example:

This is an example showing which generator strategy method will be called in what place when generating tables. For improved readability, full qualification is omitted:

```
package com.example.tables;
//   1: ^^^^^^^^^^^^^^^^^
public class Book extends TableImpl<com.example.tables.records.BookRecord> {
//        2: ^^^^                                          3: ^^^^^^^^^^
    public static final Book    BOOK = new Book();
//                   2: ^^^^ 4: ^^^^
    public final TableField<BookRecord, Integer> ID = /* ... */
//                       3: ^^^^^^^^^^        5: ^^
}

// 1: strategy.getJavaPackageName(table)
// 2: strategy.getJavaClassName(table)
// 3: strategy.getJavaClassName(table, Mode.RECORD)
// 4: strategy.getJavaIdentifier(table)
// 5: strategy.getJavaIdentifier(column)
```

An org.jooq.Record example:

This is an example showing which generator strategy method will be called in what place when generating records. For improved readability, full qualification is omitted:

```
package com.example.tables.records;
//   1: ^^^^^^^^^^^^^^^^^^^^^^^^^
public class BookRecord extends UpdatableRecordImpl<BookRecord> {
//        2: ^^^^^^^^^^                          2: ^^^^^^^^^^
    public void setId(Integer value) { /* ... */ }
//           3: ^^^^^
    public Integer getId() { /* ... */ }
//              4: ^^^^^
}

// 1: strategy.getJavaPackageName(table, Mode.RECORD)
// 2: strategy.getJavaClassName(table, Mode.RECORD)
// 3: strategy.getJavaSetterName(column, Mode.RECORD)
// 4: strategy.getJavaGetterName(column, Mode.RECORD)
```

A POJO example:

This is an example showing which generator strategy method will be called in what place when generating pojos. For improved readability, full qualification is omitted:

```
package com.example.tables.pojos;
//   1: ^^^^^^^^^^^^^^^^^^^^^^^^
public class Book implements java.io.Serializable {
//        2: ^^^^
    private Integer id;
//                3: ^^
    public void setId(Integer value) { /* ... */ }
//           4: ^^^^^
    public Integer getId() { /* ... */ }
//              5: ^^^^^

}

// 1: strategy.getJavaPackageName(table, Mode.POJO)
// 2: strategy.getJavaClassName(table, Mode.POJO)
// 3: strategy.getJavaMemberName(column, Mode.POJO)
// 4: strategy.getJavaSetterName(column, Mode.POJO)
// 5: strategy.getJavaGetterName(column, Mode.POJO)
```

An out-of-the-box strategy to keep names as they are

By default, jOOQ's generator strategy will convert your database's UNDER_SCORE_NAMES to PascalCaseNames as this is a more common idiom in the Java ecosystem. If, however, you want to retain the names and the casing exactly as it is defined in your database, you can use the org.jooq.codegen.KeepNamesGeneratorStrategy, which will retain all names exactly as they are.

More examples can be found here:

- org.jooq.codegen.example.JPrefixGeneratorStrategy
- org.jooq.codegen.example.JVMArgsGeneratorStrategy

6.5. Matcher strategies

Using custom matcher strategies

In the previous section, we have seen how to override generator strategies programmatically. In this chapter, we'll see how such strategies can be configured in the XML or Maven code generator configuration. Instead of specifying a strategy name, you can also specify a <matchers/> element as such:

- NOTE: All regular expressions that match object identifiers try to match identifiers first by unqualified name (org.jooq.meta.Definition.getName()), then by qualified name (org.jooq.meta.Definition.getQualifiedName()).
- NOTE: There had been an incompatible change between jOOQ 3.2 and jOOQ 3.3 in the configuration of these matcher strategies. See Issue #3217 for details.

```
<!-- These properties can be added directly to the generator element: -->
<generator>
  <strategy>
    <matchers>
      <!-- Specify 0..n schema matchers to provide a strategy for naming objects created from schemas. -->
      <schemas>
        <schema>

          <!-- Match unqualified or qualified schema names. If left empty, this matcher applies to all schemas. -->
          <expression>MY_SCHEMA</expression>

          <!-- These elements influence the naming of a generated org.jooq.Schema object. -->
          <schemaClass> see below MatcherRule specification </schemaClass>
          <schemaIdentifier> see below MatcherRule specification </schemaIdentifier>
          <schemaImplements>com.example.MyOptionalCustomInterface</schemaImplements>
        </schema>
      </schemas>

      <!-- Specify 0..n table matchers to provide a strategy for naming objects created from tables. -->
      <tables>
        <table>

          <!-- Match unqualified or qualified table names. If left empty, this matcher applies to all tables. -->
          <expression>MY_TABLE</expression>

          <!-- These elements influence the naming of a generated org.jooq.Table object. -->
          <tableClass> see below MatcherRule specification </tableClass>
          <tableIdentifier> see below MatcherRule specification </tableIdentifier>
          <tableImplements>com.example.MyOptionalCustomInterface</tableImplements>

          <!-- These elements influence the naming of a generated org.jooq.Record object. -->
          <recordClass> see below MatcherRule specification </recordClass>
          <recordImplements>com.example.MyOptionalCustomInterface</recordImplements>

          <!-- These elements influence the naming of a generated interface, implemented by
               generated org.jooq.Record objects and by generated POJOs. -->
          <interfaceClass> see below MatcherRule specification </interfaceClass>
          <interfaceImplements>com.example.MyOptionalCustomInterface</interfaceImplements>

          <!-- These elements influence the naming of a generated org.jooq.DAO object. -->
          <daoClass> see below MatcherRule specification </daoClass>
          <daoImplements>com.example.MyOptionalCustomInterface</daoImplements>

          <!-- These elements influence the naming of a generated POJO object.  -->
          <pojoClass> see below MatcherRule specification </pojoClass>
          <pojoExtends>com.example.MyOptionalCustomBaseClass</pojoExtends>
          <pojoImplements>com.example.MyOptionalCustomInterface</pojoImplements>
        </table>
      </tables>

      <!-- Specify 0..n field matchers to provide a strategy for naming objects created from fields. -->
      <fields>
        <field>

          <!-- Match unqualified or qualified field names. If left empty, this matcher applies to all fields. -->
          <expression>MY_FIELD</expression>

          <!-- These elements influence the naming of a generated org.jooq.Field object. -->
          <fieldIdentifier> see below MatcherRule specification </fieldIdentifier>
          <fieldMember> see below MatcherRule specification </fieldMember>
          <fieldSetter> see below MatcherRule specification </fieldSetter>
          <fieldGetter> see below MatcherRule specification </fieldGetter>
        </field>
      </fields>

      <!-- Specify 0..n routine matchers to provide a strategy for naming objects created from routines. -->
      <routines>
        <routine>

          <!-- Match unqualified or qualified routine names. If left empty, this matcher applies to all routines. -->
          <expression>MY_ROUTINE</expression>

          <!-- These elements influence the naming of a generated org.jooq.Routine object. -->
          <routineClass> see below MatcherRule specification </routineClass>
          <routineMethod> see below MatcherRule specification </routineMethod>
          <routineImplements>com.example.MyOptionalCustomInterface</routineImplements>
        </routine>
      </routines>

      <!-- Specify 0..n sequence matchers to provide a strategy for naming objects created from sequences. -->
      <sequences>
        <sequence>

          <!-- Match unqualified or qualified sequence names. If left empty, this matcher applies to all sequences. -->
          <expression>MY_SEQUENCE</expression>

          <!-- These elements influence the naming of the generated Sequences class. -->
          <sequenceIdentifier> see below MatcherRule specification </sequenceIdentifier>
        </sequence>
      </sequences>

      <!-- Specify 0..n enum matchers to provide a strategy for naming objects created from enums. -->
      <enums>
        <enum>

          <!-- Match unqualified or qualified enum names. If left empty, this matcher applies to all enums. -->
          <expression>MY_ENUM</expression>

          <!-- These elements influence the naming of a generated org.jooq.EnumType object. -->
          <enumClass> see below MatcherRule specification </enumClass>
          <enumImplements>com.example.MyOptionalCustomInterface</enumImplements>
        </enum>
      </enums>
    </matchers>
  </strategy>
</generator>
```

The above example used references to "MatcherRule", which is an XSD type that looks like this:

```
<schemaClass>
  <!-- The optional transform element lets you apply a name transformation algorithm
       to transform the actual database name into a more convenient form. Possible values are:

       - AS_IS  : Leave the database name as it is             : MY_name => MY_name
       - LOWER  : Transform the database name into lower case  : MY_name => my_name
       - UPPER  : Transform the database name into upper case  : MY_name => MY_NAME
       - CAMEL  : Transform the database name into camel case  : MY_name => myName
       - PASCAL : Transform the database name into pascal case : MY_name => MyName -->
  <transform>CAMEL</transform>

  <!-- The mandatory expression element lets you specify a replacement expression to be used when
       replacing the matcher's regular expression. You can use indexed variables $0, $1, $2. -->
  <expression>PREFIX_$0_SUFFIX</expression>
</schemaClass>
```

Some examples

The following example shows a matcher strategy that adds a "T_" prefix to all table classes and to table identifiers:

```
<generator>
  <strategy>
    <matchers>
      <tables>
        <table>

          <!-- Expression is omitted. This will make this rule apply to all tables -->
          <tableIdentifier>
            <transform>UPPER</transform>
            <expression>T_$0</expression>
          </tableIdentifier>
          <tableClass>
            <transform>PASCAL</transform>
            <expression>T_$0</expression>
          </tableClass>
        </table>
      </tables>
    </matchers>
  </strategy>
</generator>
```

The following example shows a matcher strategy that renames BOOK table identifiers (or table identifiers containing BOOK) into BROCHURE (or tables containing BROCHURE):

```
<generator>
  <strategy>
    <matchers>
      <tables>
        <table>
          <expression>^(.*?)_BOOK_(.*)$</expression>
          <tableIdentifier>
            <transform>UPPER</transform>
            <expression>$1_BROCHURE_$2</expression>
          </tableIdentifier>
        </table>
      </tables>
    </matchers>
  </strategy>
</generator>
```

For more information about each XML tag, please refer to the http://www.jooq.org/xsd/jooq-codegen-3.11.0.xsd XSD file.

6.6. Custom code sections

Power users might choose to re-implement large parts of the org.jooq.codegen.JavaGenerator class. If you only want to add some custom code sections, however, you can extend the JavaGenerator and override only parts of it.

An example for generating custom class footers

```
public class MyGenerator1 extends JavaGenerator {

    @Override
    protected void generateRecordClassFooter(TableDefinition table, JavaWriter out) {
        out.println();
        out.tab(1).println("public String toString() {");
        out.tab(2).println("return \"MyRecord[\" + valuesRow() + \"]\";");
        out.tab(1).println("}");
    }
}
```

The above example simply adds a class footer to generated records, in this case, overriding the default toString() implementation.

An example for generating custom class Javadoc

```
public class MyGenerator2 extends JavaGenerator {

    @Override
    protected void generateRecordClassJavadoc(TableDefinition table, JavaWriter out) {
        out.println("/**");
        out.println(" * This record belongs to table " + table.getOutputName() + ".");

        if (table.getComment() != null && !"".equals(table.getComment())) {
            out.println(" * <p>");
            out.println(" * Table comment: " + table.getComment());
        }

        out.println(" */");
    }
}
```

Any of the below methods can be overridden:

```
generateArray(SchemaDefinition, ArrayDefinition)          // Generates an Oracle array class
generateArrayClassFooter(ArrayDefinition, JavaWriter)     // Callback for an Oracle array class footer
generateArrayClassJavadoc(ArrayDefinition, JavaWriter)    // Callback for an Oracle array class Javadoc

generateDao(TableDefinition)                              // Generates a DAO class
generateDaoClassFooter(TableDefinition, JavaWriter)       // Callback for a DAO class footer
generateDaoClassJavadoc(TableDefinition, JavaWriter)      // Callback for a DAO class Javadoc

generateEnum(EnumDefinition)                              // Generates an enum
generateEnumClassFooter(EnumDefinition, JavaWriter)       // Callback for an enum footer
generateEnumClassJavadoc(EnumDefinition, JavaWriter)      // Callback for an enum Javadoc

generateInterface(TableDefinition)                        // Generates an interface
generateInterfaceClassFooter(TableDefinition, JavaWriter) // Callback for an interface footer
generateInterfaceClassJavadoc(TableDefinition, JavaWriter) // Callback for an interface Javadoc

generatePackage(SchemaDefinition, PackageDefinition)      // Generates an Oracle package class
generatePackageClassFooter(PackageDefinition, JavaWriter) // Callback for an Oracle package class footer
generatePackageClassJavadoc(PackageDefinition, JavaWriter) // Callback for an Oracle package class Javadoc

generatePojo(TableDefinition)                             // Generates a POJO class
generatePojoClassFooter(TableDefinition, JavaWriter)      // Callback for a POJO class footer
generatePojoClassJavadoc(TableDefinition, JavaWriter)     // Callback for a POJO class Javadoc

generateRecord(TableDefinition)                           // Generates a Record class
generateRecordClassFooter(TableDefinition, JavaWriter)    // Callback for a Record class footer
generateRecordClassJavadoc(TableDefinition, JavaWriter)   // Callback for a Record class Javadoc

generateRoutine(SchemaDefinition, RoutineDefinition)      // Generates a Routine class
generateRoutineClassFooter(RoutineDefinition, JavaWriter) // Callback for a Routine class footer
generateRoutineClassJavadoc(RoutineDefinition, JavaWriter) // Callback for a Routine class Javadoc

generateSchema(SchemaDefinition)                          // Generates a Schema class
generateSchemaClassFooter(SchemaDefinition, JavaWriter)   // Callback for a Schema class footer
generateSchemaClassJavadoc(SchemaDefinition, JavaWriter)  // Callback for a Schema class Javadoc

generateTable(SchemaDefinition, TableDefinition)          // Generates a Table class
generateTableClassFooter(TableDefinition, JavaWriter)     // Callback for a Table class footer
generateTableClassJavadoc(TableDefinition, JavaWriter)    // Callback for a Table class Javadoc

generateUDT(SchemaDefinition, UDTDefinition)              // Generates a UDT class
generateUDTClassFooter(UDTDefinition, JavaWriter)         // Callback for a UDT class footer
generateUDTClassJavadoc(UDTDefinition, JavaWriter)        // Callback for a UDT class Javadoc

generateUDTRecord(UDTDefinition)                          // Generates a UDT Record class
generateUDTRecordClassFooter(UDTDefinition, JavaWriter)   // Callback for a UDT Record class footer
generateUDTRecordClassJavadoc(UDTDefinition, JavaWriter)  // Callback for a UDT Record class Javadoc
```

When you override any of the above, do note that according to jOOQ's understanding of semantic versioning, incompatible changes may be introduced between minor releases, even if this should be the exception.

6.7. Generated global artefacts

For increased convenience at the use-site, jOOQ generates "global" artefacts at the code generation root location, referencing tables, routines, sequences, etc. In detail, these global artefacts include the following:

- Keys.java: This file contains all of the required primary key, unique key, foreign key and identity references in the form of static members of type org.jooq.Key.
- Routines.java: This file contains all standalone routines (not in packages) in the form of static factory methods for org.jooq.Routine types.
- Sequences.java: This file contains all sequence objects in the form of static members of type org.jooq.Sequence.
- Tables.java: This file contains all table objects in the form of static member references to the actual singleton org.jooq.Table object
- UDTs.java: This file contains all UDT objects in the form of static member references to the actual singleton org.jooq.UDT object

Referencing global artefacts

When referencing global artefacts from your client application, you would typically static import them as such:

```
// Static imports for all global artefacts (if they exist)
import static com.example.generated.Keys.*;
import static com.example.generated.Routines.*;
import static com.example.generated.Sequences.*;
import static com.example.generated.Tables.*;

// You could then reference your artefacts as follows:
create.insertInto(MY_TABLE)
      .values(MY_SEQUENCE.nextval(), myFunction())

// as a more concise form of this:
create.insertInto(com.example.generated.Tables.MY_TABLE)
      .values(com.example.generated.Sequences.MY_SEQUENCE.nextval(), com.example.generated.Routines.myFunction())
```

Configuring these artefacts

The generation of these artefacts can be turned off. For details, see the relevant section in the manual.

6.8. Generated tables

Every table in your database will generate a org.jooq.Table implementation that looks like this:

```
public class Book extends TableImpl<BookRecord> {

    // The singleton instance
    public static final Book BOOK = new Book();

    // Generated columns
    public final TableField<BookRecord, Integer> ID        = createField("ID",        SQLDataType.INTEGER, this);
    public final TableField<BookRecord, Integer> AUTHOR_ID = createField("AUTHOR_ID", SQLDataType.INTEGER, this);
    public final TableField<BookRecord, String>  ITLE      = createField("TITLE",     SQLDataType.VARCHAR, this);

    // Covariant aliasing method, returning a table of the same type as BOOK
    @Override
    public Book as(java.lang.String alias) {
        return new Book(alias);
    }

    // [...]
}
```

Flags influencing generated tables

These flags from the code generation configuration influence generated tables:

- recordVersionFields: Relevant methods from super classes are overridden to return the VERSION field
- recordTimestampFields: Relevant methods from super classes are overridden to return the TIMESTAMP field
- syntheticPrimaryKeys: This overrides existing primary key information to allow for "custom" primary key column sets
- overridePrimaryKeys: This overrides existing primary key information to allow for unique key to primary key promotion
- dateAsTimestamp: This influences all relevant columns
- unsignedTypes: This influences all relevant columns
- relations: Relevant methods from super classes are overridden to provide primary key, unique key, foreign key and identity information
- instanceFields: This flag controls the "static" keyword on table columns, as well as aliasing convenience
- records: The generated record type is referenced from tables allowing for type-safe single-table record fetching

Flags controlling table generation

Routine generation can be deactivated using the tables flag

6.9. Generated records

Every table in your database will generate an org.jooq.Record implementation that looks like this:

```
// JPA annotations can be generated, optionally
@Entity
@Table(name = "BOOK", schema = "TEST")
public class BookRecord extends UpdatableRecordImpl<BookRecord>

// An interface common to records and pojos can be generated, optionally
implements IBook {

    // Every column generates a setter and a getter
    @Override
    public void setId(Integer value) {
        setValue(BOOK.ID, value);
    }

    @Id
    @Column(name = "ID", unique = true, nullable = false, precision = 7)
    @Override
    public Integer getId() {
        return getValue(BOOK.ID);
    }

    // More setters and getters
    public void setAuthorId(Integer value) {...}
    public Integer getAuthorId() {...}

    // Convenience methods for foreign key methods
    public void setAuthorId(AuthorRecord value) {
        if (value == null) {
            setValue(BOOK.AUTHOR_ID, null);
        }
        else {
            setValue(BOOK.AUTHOR_ID, value.getValue(AUTHOR.ID));
        }
    }

    // Navigation methods
    public AuthorRecord fetchAuthor() {
        return create.selectFrom(AUTHOR).where(AUTHOR.ID.eq(getValue(BOOK.AUTHOR_ID))).fetchOne();
    }

    // [...]
}
```

Flags influencing generated records

These flags from the code generation configuration influence generated records:

- syntheticPrimaryKeys: This overrides existing primary key information to allow for "custom" primary key column sets, possibly promoting a TableRecord to an UpdatableRecord
- overridePrimaryKeys: This overrides existing primary key information to allow for unique key to primary key promotion, possibly promoting a TableRecord to an UpdatableRecord
- dateAsTimestamp: This influences all relevant getters and setters
- unsignedTypes: This influences all relevant getters and setters
- relations: This is needed as a prerequisite for navigation methods
- daos: Records are a pre-requisite for DAOs. If DAOs are generated, records are generated as well
- interfaces: If interfaces are generated, records will implement them
- jpaAnnotations: JPA annotations are used on generated records (details here)
- jpaVersion: Version of JPA specification is to be used to generate version-specific annotations. If it is omitted, the latest version is used by default. (details here)

Flags controlling record generation

Record generation can be deactivated using the records flag

6.10. Generated POJOs

Every table in your database will generate a POJO implementation that looks like this:

```
// JPA annotations can be generated, optionally
@javax.persistence.Entity
@javax.persistence.Table(name = "BOOK", schema = "TEST")
public class Book implements java.io.Serializable

// An interface common to records and pojos can be generated, optionally
, IBook {

    // JSR-303 annotations can be generated, optionally
    @NotNull
    private Integer id;

    @NotNull
    private Integer authorId;

    @NotNull
    @Size(max = 400)
    private String title;

    // Every column generates a getter and a setter
    @Id
    @Column(name = "ID", unique = true, nullable = false, precision = 7)
    @Override
    public Integer getId() {
        return this.id;
    }

    @Override
    public void setId(Integer id) {
        this.id = id;
    }

    // [...]
}
```

Flags influencing generated POJOs

These flags from the code generation configuration influence generated POJOs:

- dateAsTimestamp: This influences all relevant getters and setters
- unsignedTypes: This influences all relevant getters and setters
- interfaces: If interfaces are generated, POJOs will implement them
- immutablePojos: Immutable POJOs have final members and no setters. All members must be passed to the constructor
- daos: POJOs are a pre-requisite for DAOs. If DAOs are generated, POJOs are generated as well
- jpaAnnotations: JPA annotations are used on generated records (details here)
- jpaVersion: Version of JPA specification is to be used to generate version-specific annotations. If it is omitted, the latest version is used by default. (details here)
- validationAnnotations: JSR-303 validation annotations are used on generated records (details here)

Flags controlling POJO generation

POJO generation can be activated using the pojos flag

6.11. Generated Interfaces

Every table in your database will generate an interface that looks like this:

```
public interface IBook extends java.io.Serializable {

    // Every column generates a getter and a setter
    public void setId(Integer value);
    public Integer getId();

    // [...]
}
```

Flags influencing generated interfaces

These flags from the code generation configuration influence generated interfaces:

- dateAsTimestamp: This influences all relevant getters and setters
- unsignedTypes: This influences all relevant getters and setters

Flags controlling interface generation

Interface generation can be activated using the interfaces flag

6.12. Generated DAOs

Generated DAOs

Every table in your database will generate a org.jooq.DAO implementation that looks like this:

```
public class BookDao extends DAOImpl<BookRecord, Book, Integer> {

    // Generated constructors
    public BookDao() {
        super(BOOK, Book.class);
    }

    public BookDao(Configuration configuration) {
        super(BOOK, Book.class, configuration);
    }

    // Every column generates at least one fetch method
    public List<Book> fetchById(Integer... values) {
        return fetch(BOOK.ID, values);
    }

    public Book fetchOneById(Integer value) {
        return fetchOne(BOOK.ID, value);
    }

    public List<Book> fetchByAuthorId(Integer... values) {
        return fetch(BOOK.AUTHOR_ID, values);
    }

    // [...]
}
```

Flags controlling DAO generation

DAO generation can be activated using the daos flag

6.13. Generated sequences

Every sequence in your database will generate a org.jooq.Sequence implementation that looks like this:

```
public final class Sequences {

    // Every sequence generates a member
    public static final Sequence<Integer> S_AUTHOR_ID = new SequenceImpl<Integer>("S_AUTHOR_ID", TEST, SQLDataType.INTEGER);
}
```

Flags controlling sequence generation

Routine generation can be deactivated using the sequences flag

6.14. Generated procedures

Every procedure or function (routine) in your database will generate a org.jooq.Routine implementation that looks like this:

```
public class AuthorExists extends AbstractRoutine<java.lang.Void> {

    // All IN, IN OUT, OUT parameters and function return values generate a static member
    public static final Parameter<String>     AUTHOR_NAME = createParameter("AUTHOR_NAME", SQLDataType.VARCHAR);
    public static final Parameter<BigDecimal> RESULT      = createParameter("RESULT",      SQLDataType.NUMERIC);

    // A constructor for a new "empty" procedure call
    public AuthorExists() {
        super("AUTHOR_EXISTS", TEST);

        addInParameter(AUTHOR_NAME);
        addOutParameter(RESULT);
    }

    // Every IN and IN OUT parameter generates a setter
    public void setAuthorName(String value) {
        setValue(AUTHOR_NAME, value);
    }

    // Every IN OUT, OUT and RETURN_VALUE generates a getter
    public BigDecimal getResult() {
        return getValue(RESULT);
    }

    // [...]
}
```

Package and member procedures or functions

Procedures or functions contained in packages or UDTs are generated in a sub-package that corresponds to the package or UDT name.

Flags controlling routine generation

Routine generation can be deactivated using the routines flag

6.15. Generated UDTs

Every UDT in your database will generate a org.jooq.UDT implementation that looks like this:

```
public class AddressType extends UDTImpl<AddressTypeRecord> {

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

    // Every UDT attribute generates a static member
    public static final UDTField<AddressTypeRecord, String> ZIP     =
      createField("ZIP",     SQLDataType.VARCHAR, U_ADDRESS_TYPE);
    public static final UDTField<AddressTypeRecord, String> CITY    =
      createField("CITY",    SQLDataType.VARCHAR, U_ADDRESS_TYPE);
    public static final UDTField<AddressTypeRecord, String> COUNTRY =
      createField("COUNTRY", SQLDataType.VARCHAR, U_ADDRESS_TYPE);

    // [...]
}
```

Besides the org.jooq.UDT implementation, a org.jooq.UDTRecord implementation is also generated

```
public class AddressTypeRecord extends UDTRecordImpl<AddressTypeRecord> {

    // Every attribute generates a getter and a setter

    public void setZip(String value) {...}
    public String getZip() {...}
    public void setCity(String value) {...}
    public String getCity() {...}
    public void setCountry(String value) {...}
    public String getCountry() {...}

    // [...]
}
```

Flags controlling UDT generation

Routine generation can be deactivated using the udts flag

6.16. Data type rewrites

Sometimes, the actual database data type does not match the SQL data type that you would like to use in Java. This is often the case for ill-supported SQL data types, such as BOOLEAN or UUID. jOOQ's code generator allows you to apply simple data type rewriting. The following configuration will rewrite IS_VALID columns in all tables to be of type BOOLEAN.

```
<database>

  <!-- Associate data type rewrites with database columns -->
  <forcedTypes>
    <forcedType>
      <!-- Specify any data type that is supported in your database, or if unsupported, a type from org.jooq.impl.SQLDataType -->
      <name>BOOLEAN</name>

      <!-- Add a Java regular expression matching fully-qualified columns. Use the pipe to separate several expressions.

           If provided, both "expressions" and "types" must match. -->
      <expression>.*\.IS_VALID</expression>

      <!-- Add a Java regular expression matching data types to be forced to have this type.

           Data types may be reported by your database as:
           - NUMBER              regexp suggestion: NUMBER
           - NUMBER(5)           regexp suggestion: NUMBER\(5\)
           - NUMBER(5, 2)        regexp suggestion: NUMBER\(5,\s*2\)
           - any other form.

           It is thus recommended to use defensive regexes for types.

           If provided, both "expressions" and "types" must match. -->
      <types>.*</types>
    </forcedType>
  </forcedTypes>
</database>
```

You must provide at least either an <expressions/> or a <types/> element, or both.

See the section about Custom data types for rewriting columns to your own custom data types.

6.17. Custom data types and type conversion

When using a custom type in jOOQ, you need to let jOOQ know about its associated org.jooq.Converter. Ad-hoc usages of such converters has been discussed in the chapter about data type conversion. However, when mapping a custom type onto a standard JDBC type, a more common use-case is to let jOOQ know about custom types at code generation time (if you're using non-standard JDBC types, like for example JSON or HSTORE, see the manual's section about custom data type bindings). Use the

following configuration elements to specify, that you'd like to use GregorianCalendar for all database fields that start with DATE_OF_

```
<database>

  <!-- Then, associate custom types with database columns -->
  <forcedTypes>
    <forcedType>

      <!-- Specify the Java type of your custom type. This corresponds to the Converter's <U> type. -->
      <userType>java.util.GregorianCalendar</userType>

      <!-- Associate that custom type with your converter. -->
      <converter>com.example.CalendarConverter</converter>

      <!-- Add a Java regular expression matching fully-qualified columns. Use the pipe to separate several expressions.

           If provided, both "expressions" and "types" must match. -->
      <expression>.*\.DATE_OF_.*</expression>

      <!-- Add a Java regular expression matching data types to be forced to
           have this type.

           Data types may be reported by your database as:
           - NUMBER              regexp suggestion: NUMBER
           - NUMBER(5)           regexp suggestion: NUMBER\(5\)
           - NUMBER(5, 2)        regexp suggestion: NUMBER\(5,\s*2\)
           - any other form

           It is thus recommended to use defensive regexes for types.

           If provided, both "expressions" and "types" must match. -->
      <types>.*</types>
    </forcedType>
  </forcedTypes>
</database>
```

See also the section about data type rewrites to learn about an alternative use of <forcedTypes/>.

The above configuration will lead to AUTHOR.DATE_OF_BIRTH being generated like this:

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

    // [...]
    public final TableField<TAuthorRecord, GregorianCalendar> DATE_OF_BIRTH =    // [...]
    // [...]

}
```

This means that the bound type of <T> will be GregorianCalendar, wherever you reference DATE_OF_BIRTH. jOOQ will use your custom converter when binding variables and when fetching data from java.util.ResultSet:

```
// Get all date of births of authors born after 1980
List<GregorianCalendar> result =
create.selectFrom(AUTHOR)
      .where(AUTHOR.DATE_OF_BIRTH.gt(new GregorianCalendar(1980, 0, 1)))
      .fetch(AUTHOR.DATE_OF_BIRTH);
```

6.18. Custom data type binding

The previous section discussed the case where your custom data type is mapped onto a standard JDBC type as contained in org.jooq.impl.SQLDataType. In some cases, however, you want to map your own type onto a type that is not explicitly supported by JDBC, such as for instance, PostgreSQL's various advanced data types like JSON or HSTORE, or PostGIS types. For this, you can register an org.jooq.Binding for relevant columns in your code generator. Consider the following trivial implementation of a binding for PostgreSQL's JSON data type, which binds the JSON string in PostgreSQL to a Google GSON object:

```
import static org.jooq.tools.Convert.convert;
import java.sql.*;
import org.jooq.*;
import org.jooq.impl.DSL;
import com.google.gson.*;

// We're binding <T> = Object (unknown JDBC type), and <U> = JsonElement (user type)
public class PostgresJSONGsonBinding implements Binding<Object, JsonElement> {

    // The converter does all the work
    @Override
    public Converter<Object, JsonElement> converter() {
        return new Converter<Object, JsonElement>() {
            @Override
            public JsonElement from(Object t) {
                return t == null ? JsonNull.INSTANCE : new Gson().fromJson("" + t, JsonElement.class);
            }

            @Override
            public Object to(JsonElement u) {
                return u == null || u == JsonNull.INSTANCE ? null : new Gson().toJson(u);
            }

            @Override
            public Class<Object> fromType() {
                return Object.class;
            }

            @Override
            public Class<JsonElement> toType() {
                return JsonElement.class;
            }
        };
    }

    // Rending a bind variable for the binding context's value and casting it to the json type
    @Override
    public void sql(BindingSQLContext<JsonElement> ctx) throws SQLException {
        // Depending on how you generate your SQL, you may need to explicitly distinguish
        // between jOOQ generating bind variables or inlined literals.
        if (ctx.render().paramType() == ParamType.INLINED)
            ctx.render().visit(DSL.inline(ctx.convert(converter()).value())).sql("::json");
        else
            ctx.render().sql("?::json");
    }

    // Registering VARCHAR types for JDBC CallableStatement OUT parameters
    @Override
    public void register(BindingRegisterContext<JsonElement> ctx) throws SQLException {
        ctx.statement().registerOutParameter(ctx.index(), Types.VARCHAR);
    }

    // Converting the JsonElement to a String value and setting that on a JDBC PreparedStatement
    @Override
    public void set(BindingSetStatementContext<JsonElement> ctx) throws SQLException {
        ctx.statement().setString(ctx.index(), Objects.toString(ctx.convert(converter()).value(), null));
    }

    // Getting a String value from a JDBC ResultSet and converting that to a JsonElement
    @Override
    public void get(BindingGetResultSetContext<JsonElement> ctx) throws SQLException {
        ctx.convert(converter()).value(ctx.resultSet().getString(ctx.index()));
    }

    // Getting a String value from a JDBC CallableStatement and converting that to a JsonElement
    @Override
    public void get(BindingGetStatementContext<JsonElement> ctx) throws SQLException {
        ctx.convert(converter()).value(ctx.statement().getString(ctx.index()));
    }

    // Setting a value on a JDBC SQLOutput (useful for Oracle OBJECT types)
    @Override
    public void set(BindingSetSQLOutputContext<JsonElement> ctx) throws SQLException {
        throw new SQLFeatureNotSupportedException();
    }

    // Getting a value from a JDBC SQLInput (useful for Oracle OBJECT types)
    @Override
    public void get(BindingGetSQLInputContext<JsonElement> ctx) throws SQLException {
        throw new SQLFeatureNotSupportedException();
    }
}
```

Registering bindings to the code generator

The above org.jooq.Binding implementation intercepts all the interaction on a JDBC level, such that jOOQ will never need to know how to correctly serialise / deserialise your custom data type. Similar to what we've seen in the previous section about how to register Converters to the code generator, we can now register such a binding to the code generator. Note that you will reuse the same types of XML elements (<forcedType/>):

```
<database>
  <forcedTypes>
    <forcedType>

      <!-- Specify the Java type of your custom type. This corresponds to the Binding's <U> type. -->
      <userType>com.google.gson.JsonElement</userType>

      <!-- Associate that custom type with your binding. -->
      <binding>com.example.PostgresJSONGsonBinding</binding>

      <!-- Add a Java regular expression matching fully-qualified columns. Use the pipe to separate several expressions.

           If provided, both "expressions" and "types" must match. -->
      <expression>.*JSON.*</expression>

      <!-- Add a Java regular expression matching data types to be forced to
           have this type.

           Data types may be reported by your database as:
           - NUMBER              regexp suggestion: NUMBER
           - NUMBER(5)           regexp suggestion: NUMBER\(5\)
           - NUMBER(5, 2)        regexp suggestion: NUMBER\(5,\s*2\)
           - any other form

           It is thus recommended to use defensive regexes for types.

           If provided, both "expressions" and "types" must match. -->
      <types>.*</types>
    </forcedType>
  </forcedTypes>
</database>
```

See also the section about data type rewrites to learn about an alternative use of <forcedTypes/>.

The above configuration will lead to AUTHOR.CUSTOM_DATA_JSON being generated like this:

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

    // [...]
    public final TableField<TAuthorRecord, JsonElement> CUSTOM_DATA_JSON =    // [...]
    // [...]

}
```

6.19. Mapping generated catalogs and schemas

We've seen previously in the chapter about runtime schema mapping, that catalogs, schemata and tables can be mapped at runtime to other names. But you can also hard-wire catalog and 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.

Schema mapping

The following configuration applies mapping only for schemata, not for catalogs. The <schemata/> element is a standalone element that can be put in the code generator's <database/> configuration element:

```
<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>
```

The following configuration applies mapping for catalogs and their schemata. The <catalogs/> element is a standalone element that can be put in the code generator's <database/> configuration element:

```
<catalogs>
  <catalog>
    <!-- Use this as the developer's catalog: -->
    <inputCatalog>LUKAS_DEV_CATALOG</inputCatalog>

    <!-- Use this as the integration / production database: -->
    <outputCatalog>PROD</outputCatalog>

    <!-- Optionally, nest also schema mapping configurations: -->
    <schemata>
      ...
    </schemata>
  </catalog>
</catalogs>
```

6.20. Code generation for large schemas

Databases can become very large in real-world applications. This is not a problem for jOOQ's code generator, but it can be for the Java compiler. jOOQ generates some classes for global access. These classes can hit two sorts of limits of the compiler / JVM:

- Methods (including static / instance initialisers) are allowed to contain only 64kb of bytecode.
- Classes are allowed to contain at most 64k of constant literals

While there exist workarounds for the above two limitations (delegating initialisations to nested classes, inheriting constant literals from implemented interfaces), the preferred approach is either one of these:

- Distribute your database objects in several schemas. That is probably a good idea anyway for such large databases
- Configure jOOQ's code generator to exclude excess database objects
- Configure jOOQ's code generator to avoid generating global objects using <globalObjectReferences/>
- Remove uncompilable classes after code generation

6.21. Code generation and version control

When using jOOQ's code generation capabilities, you will need to make a strategic decision about whether you consider your generated code as

- Part of your code base
- Derived artefacts

In this section we'll see that both approaches have their merits and that none of them is clearly better.

Part of your code base

When you consider generated code as part of your code base, you will want to:

- Check in generated sources in your version control system
- Use manual source code generation
- Possibly use even partial source code generation

This approach is particularly useful when your Java developers are not in full control of or do not have full access to your database schema, or if you have many developers that work simultaneously on the same database schema, which changes all the time. It is also useful to be able to track side-effects of database changes, as your checked-in database schema can be considered when you want to analyse the history of your schema.

With this approach, you can also keep track of the change of behaviour in the jOOQ code generator, e.g. when upgrading jOOQ, or when modifying the code generation configuration.

The drawback of this approach is that it is more error-prone as the actual schema may go out of sync with the generated schema.

Derived artefacts

When you consider generated code to be derived artefacts, you will want to:

- Check in only the actual DDL (e.g. controlled via Flyway)
- Regenerate jOOQ code every time the schema changes
- Regenerate jOOQ code on every machine - including continuous integration

This approach is particularly useful when you have a smaller database schema that is under full control by your Java developers, who want to profit from the increased quality of being able to regenerate all derived artefacts in every step of your build.

The drawback of this approach is that the build may break in perfectly acceptable situations, when parts of your database are temporarily unavailable.

Pragmatic combination

In some situations, you may want to choose a pragmatic combination, where you put only some parts of the generated code under version control. For instance, jOOQ-meta's generated sources are put under version control as few contributors will be able to run the jOOQ-meta code generator against all supported databases.

6.22. JPADatabase: Code generation from entities

Many jOOQ users use jOOQ as a complementary SQL API in applications that mostly use JPA for their database interactions, e.g. to perform reporting, batch processing, analytics, etc.

In such a setup, you might have a pre-existing schema implemented using JPA-annotated entities. Your real database schema might not be accessible while developing, or it is not a first-class citizen in your application (i.e. you follow a Java-first approach). This section explains how you can generate jOOQ classes from such a JPA model. Consider this model:

```
@Entity
@Table(name = "author")
public class Author {

    @Id
    int       id;

    @Column(name = "first_name")
    String    firstName;

    @Column(name = "last_name")
    String    lastName;

    @OneToMany(mappedBy = "author")
    Set<Book> books;

    // Getters and setters...
}

@Entity
@Table(name = "book")
public class Book {

    @Id
    public int    id;

    @Column(name = "title")
    public String title;

    @ManyToOne
    public Author author;

    // Getters and setters...
}
```

Now, instead of connecting the jOOQ code generator to a database that holds a representation of the above schema, you can use jOOQ's JPADatabase and feed that to the code generator. The JPADatabase uses Hibernate internally, to generate an in-memory H2 database from your entities, and reverse-engineers that again back to jOOQ classes.

The easiest way forward is to use Maven in order to include the jooq-meta-extensions library (which then includes the H2 and Hibernate dependencies)

```
<dependency>
    <!-- Use org.jooq            for the Open Source Edition
             org.jooq.pro        for commercial editions,
             org.jooq.pro-java-6 for commercial editions with Java 6 support,
             org.jooq.trial      for the free trial edition

         Note: Only the Open Source Edition is hosted on Maven Central.
               Import the others manually from your distribution -->
    <groupId>org.jooq</groupId>
    <artifactId>jooq-meta-extensions</artifactId>
    <version>3.11.11</version>
</dependency>
```

With that dependency in place, you can now specify the JPADatabase in your code generator configuration:

```
<generator>
    <database>
        <name>org.jooq.meta.extensions.jpa.JPADatabase</name>
        <properties>
            <!-- A comma separated list of Java packages, that contain your entities -->
            <property>
                <key>packages</key>
                <value>com.example.entities</value>
            </property>

            <!-- Whether JPA 2.1 AttributeConverters should be auto-mapped to jOOQ Converters.
                 Custom <forcedType/> configurations will have a higher priority than these auto-mapped converters.
                 This defaults to true. -->
            <property>
                <key>use-attribute-converters</key>
                <value>true</value>
            </property>
        </properties>
    </database>
</generator>
```

The above will generate all jOOQ artefacts for your AUTHOR and BOOK tables.

How to organise your dependencies

The JPADatabase will use Spring to look up your annotated entities from the classpath. This means that you have to create several modules with a dependency graph that looks like this:

You cannot put your JPA entities in the same module as the one that runs the jOOQ code generator.

6.23. XMLDatabase: Code generation from XML files

By default, jOOQ's code generator takes live database connections as a database meta data source. In many project setups, this might not be optimal, as the live database is not always available.

One way to circumvent this issue is by providing jOOQ with a database meta definition file in XML format and by passing this XML file to jOOQ's XMLDatabase.

The XMLDatabase can read a standardised XML file that implements the http://www.jooq.org/xsd/jooq-meta-3.11.0.xsd schema. Essentially, this schema is an XML representation of the SQL standard INFORMATION_SCHEMA, as implemented by databases like H2, HSQLDB, MySQL, PostgreSQL, or SQL Server.

An example schema definition containing simple schema, table, column definitions can be seen below:

```
<?xml version="1.0"?>
<information_schema xmlns="http://www.jooq.org/xsd/jooq-meta-3.11.0.xsd">
    <schemata>
        <schema>
            <schema_name>TEST</schema_name>
        </schema>
    </schemata>

    <tables>
        <table>
            <table_schema>TEST</table_schema>
            <table_name>AUTHOR</table_name>
        </table>
        <table>
            <table_schema>TEST</table_schema>
            <table_name>BOOK</table_name>
        </table>
    </tables>

    <columns>
        <column>
            <table_schema>PUBLIC</table_schema>
            <table_name>AUTHOR</table_name>
            <column_name>ID</column_name>
            <data_type>NUMBER</data_type>
            <numeric_precision>7</numeric_precision>
            <ordinal_position>1</ordinal_position>
            <is_nullable>false</is_nullable>
        </column>
        ...
    </columns>
</information_schema>
```

Constraints can be defined with the following elements:

```
    <table_constraints>
        <table_constraint>
            <constraint_schema>TEST</constraint_schema>
            <constraint_name>PK_AUTHOR</constraint_name>
            <constraint_type>PRIMARY KEY</constraint_type>
            <table_schema>TEST</table_schema>
            <table_name>AUTHOR</table_name>
        </table_constraint>
        ...
    </table_constraints>

    <key_column_usages>
        <key_column_usage>
            <constraint_schema>TEST</constraint_schema>
            <constraint_name>PK_AUTHOR</constraint_name>
            <table_schema>TEST</table_schema>
            <table_name>AUTHOR</table_name>
            <column_name>ID</column_name>
            <ordinal_position>1</ordinal_position>
        </key_column_usage>
        ...
    </key_column_usage>

    <referential_constraints>
        <referential_constraint>
            <constraint_schema>TEST</constraint_schema>
            <constraint_name>FK_BOOK_AUTHOR_ID</constraint_name>
            <unique_constraint_schema>TEST</unique_constraint_schema>
            <unique_constraint_name>PK_AUTHOR</unique_constraint_name>
        </referential_constraint>
        ...
    </referential_constraints>
</information_schema>
```

The above file can be made available to the code generator configuration by using the XMLDatabase as follows:

```
<generator>
    <database>
        <name>org.jooq.meta.xml.XMLDatabase</name>
        <properties>

            <!-- Use any of the SQLDialect values here -->
            <property>
                <key>dialect</key>
                <value>ORACLE</value>
            </property>

            <!-- Specify the location of your database file -->
            <property>
                <key>xml-file</key>
                <value>src/main/resources/database.xml</value>
            </property>
        </properties>
    </database>
</generator>
```

If you already have a different XML format for your database, you can either XSL transform your own format into the one above via an additional Maven plugin, or pass the location of an XSL file to the XMLDatabase by providing an additional property:

```
<generator>
    <database>
        <name>org.jooq.meta.xml.XMLDatabase</name>
        <properties>

            ...

            <!-- Specify the location of your xsl file -->
            <property>
                <key>xsl-file</key>
                <value>src/main/resources/transform-to-jooq-format.xsl</value>
            </property>
        </properties>
    </database>
</generator>
```

This XML configuration can now be checked in and versioned, and modified independently from your live database schema.

6.24. DDLDatabase: Code generation from SQL files

In many cases, the schema is defined in the form of a SQL script, which can be used with Flyway, or some other database migration tool.

If you have a complete schema definition in a single file, or perhaps a set of incremental files that can reproduce your schema in any SQL dialect, then the DDLDatabase might be the right choice for you. It uses the SQL parser internally and applies all your DDL increments to an in-memory H2 database, in order to produce a replica of your schema prior to reverse engineering it again using ordinary code generation.

For example, the following database.sql script (the sample database from this manual) could be used:

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

CREATE TABLE 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),
  distinguished   NUMBER(1)
);

CREATE TABLE 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,

  CONSTRAINT fk_book_author     FOREIGN KEY (author_id)   REFERENCES author(id),
  CONSTRAINT fk_book_language   FOREIGN KEY (language_id) REFERENCES language(id)
);

CREATE TABLE book_store (
  name            VARCHAR2(400) NOT NULL UNIQUE
);

CREATE TABLE book_to_book_store (
  name            VARCHAR2(400) NOT NULL,
  book_id         INTEGER       NOT NULL,
  stock           INTEGER,

  PRIMARY KEY(name, book_id),
  CONSTRAINT fk_b2bs_book_store FOREIGN KEY (name)        REFERENCES book_store (name) ON DELETE CASCADE,
  CONSTRAINT fk_b2bs_book       FOREIGN KEY (book_id)     REFERENCES book (id)         ON DELETE CASCADE
);
```

While the script uses pretty standard SQL constructs, you may well use some vendor-specific extensions, and even DML statements in between to set up your schema - it doesn't matter. You will simply need to set up your code generation configuration as follows:

XML configuration (standalone and Maven)

```
<configuration xmlns="http://www.jooq.org/xsd/jooq-codegen-3.11.0.xsd">
  <generator>
    <database>
      <name>org.jooq.meta.extensions.ddl.DDLDatabase</name>
      <properties>

        <!-- Specify the location of your SQL script.
             You may use ant-style file matching, e.g. /path/**/to/*.sql

             Where:
             - ** matches any directory subtree
             - * matches any number of characters in a directory / file name
             - ? matches a single character in a directory / file name
          -->
        <property>
          <key>scripts</key>
          <value>src/main/resources/database.sql</value>
        </property>

        <!-- The sort order of the scripts within a directory, where:

             - semantic: sorts versions, e.g. v-3.10.0 is after v-3.9.0 (default)
             - alphanumeric: sorts strings, e.g. v-3.10.0 is before v-3.9.0
             - none: doesn't sort directory contents after fetching them from the directory
          -->
        <property>
          <key>sort</key>
          <value>semantic</value>
        </property>
      </properties>
    </database>
  </generator>
</configuration>
```

Programmatic configuration

```
configuration
  .withGenerator(new Generator(
    .withDatabase(new Database()
      .withName("org.jooq.meta.extensions.ddl.DDLDatabase")
      .withProperties(
        new Property()
          .withKey("scripts")
          .withValue("src/main/resources/database.sql"),
        new Property()
          .withKey("sort")
          .withValue("semantic")))));
```

Gradle configuration

```
myConfigurationName(sourceSets.main) {
  generator {
    database {
      name = 'org.jooq.meta.extensions.ddl.DDLDatabase'
      properties {
        property {
          key = 'scripts'
          value = 'src/main/resources/database.sql'
        }
        property {
          key = 'sort'
          value = 'semantic'
        }
      }
    }
  }
}
```

Dependencies

Note that the org.jooq.meta.extensions.ddl.DDLDatabase class is located in an external dependency, which needs to be placed on the classpath of the jOOQ code generator. E.g. using Maven:

```
<dependency>
  <!-- Use org.jooq            for the Open Source Edition
           org.jooq.pro        for commercial editions,
           org.jooq.pro-java-8 for commercial editions with Java 8 support,
           org.jooq.pro-java-6 for commercial editions with Java 6 support,
           org.jooq.trial      for the free trial edition

       Note: Only the Open Source Edition is hosted on Maven Central.
             Import the others manually from your distribution -->
  <groupId>org.jooq.trial</groupId>
  <artifactId>jooq-meta-extensions</artifactId>
  <version>3.11.11</version>
</dependency>
```

6.25. XMLGenerator: Generating XML

By default the code generator produces Java files for use with the jOOQ API as documented throughout this manual. In some cases, however, it may be desireable to generate other meta data formats, such as an XML document. This can be done with the XMLGenerator.

The format produced by the XMLGenerator is the same as the one consumed by the XMLDatabase, which can read such XML content to produce Java code. It is specified in the http://www.jooq.org/xsd/jooq-meta-3.11.0.xsd schema. Essentially, this schema is an XML representation of the SQL standard INFORMATION_SCHEMA, as implemented by databases like H2, HSQLDB, MySQL, PostgreSQL, or SQL Server.

In order to use the XMLGenerator, simply place the following class reference into your code generation configuration:

XML configuration (standalone and Maven)

```
<configuration xmlns="http://www.jooq.org/xsd/jooq-codegen-3.11.0.xsd">
  <generator>
    <name>org.jooq.codegen.XMLGenerator</name>
  </generator>
  ...
</configuration>
```

Programmatic configuration

```
configuration.withGenerator(new Generator()
  .withName("org.jooq.codegen.XMLGenerator"));
```

Gradle configuration

```
myConfigurationName(sourceSets.main) {
  generator {
    name = 'org.jooq.codegen.XMLGenerator'
  }
}
```

This configuration does not interfere with most of the remaining code generation configuration, e.g. you can still specify the JDBC connection or the generation output target as usual.

6.26. Running the code generator with Maven

There is no substantial difference between running the code generator with Maven or in standalone mode. Both modes use the exact same <configuration/> element. The Maven plugin configuration adds some additional boilerplate around that:

```
<plugin>
  <!-- Specify the maven code generator plugin -->
  <!-- Use org.jooq            for the Open Source Edition
          org.jooq.pro        for commercial editions,
          org.jooq.pro-java-6 for commercial editions with Java 6 support,
          org.jooq.trial      for the free trial edition

       Note: Only the Open Source Edition is hosted on Maven Central.
             Import the others manually from your distribution -->
  <groupId>org.jooq</groupId>
  <artifactId>jooq-codegen-maven</artifactId>
  <version>3.11.11</version>

  <executions>
    <execution>
      <id>jooq-codegen</id>
      <phase>generate-sources</phase>
      <goals>
        <goal>generate</goal>
      </goals>
      <configuration>
        ...
      </configuration>
    </execution>
  </executions>
</plugin>
```

Additional Maven-specific flags

There are, however, some additional, Maven-specific flags that can be specified with the jooq-codegen-maven plugin only:

```
<plugin>
  ...
      <configuration>

        <!-- A boolean property (or constant) can be specified here to tell the plugin not to do anything -->
        <skip>${skip.jooq.generation}</skip>

        <!-- Instead of providing an inline configuration here, you can specify an external XML configuration file here -->
        <configurationFile>${externalfile}</configurationFile>
      </configuration>
  ...
</plugin>
```

6.27. Running the code generator with Ant

Run generation with Ant

When running code generation with ant's <java/> task, you may have to set fork="true":

```
<!-- Run the code generation task -->
<target name="generate-test-classes">
  <java fork="true"
        classname="org.jooq.codegen.GenerationTool">
    <arg value="/path/to/configuration.xml"/>
    <classpath>
      <pathelement location="/path/to/jooq-3.11.11.jar"/>
      <pathelement location="/path/to/jooq-meta-3.11.11.jar"/>
      <pathelement location="/path/to/jooq-codegen-3.11.11.jar"/>
    </classpath>
  </java>
</target>
```

Using the Ant Maven plugin

Sometimes, ant can be useful to work around a limitation (misunderstanding?) of the Maven build. Just as with the above standalone ant usage example, the jOOQ code generator can be called from the maven-antrun-plugin:

```
<!-- Run the code generation task -->
<plugin>
  <artifactId>maven-antrun-plugin</artifactId>
  <version>1.8</version>
  <executions>
    <execution>
      <phase>generate-sources</phase>
      <configuration>
        <tasks>
          <java fork="true"
                classname="org.jooq.codegen.GenerationTool"
                classpathref="maven.compile.classpath">
              <arg value="/path/to/configuration.xml"/>
          </java>
       </tasks>
      </configuration>
      <goals>
        <goal>run</goal>
      </goals>
    </execution>
  </executions>

  <dependencies>
    <dependency>
      <!-- JDBC driver -->
    </dependency>
    <dependency>
      <!-- Use org.jooq            for the Open Source Edition
                org.jooq.pro        for commercial editions,
                org.jooq.pro-java-6 for commercial editions with Java 6 support,
                org.jooq.trial      for the free trial edition

           Note: Only the Open Source Edition is hosted on Maven Central.
                 Import the others manually from your distribution -->
      <groupId>org.jooq.trial</groupId>
      <artifactId>jooq-codegen</artifactId>
      <version>3.11.11</version>
    </dependency>
  </dependencies>
</plugin>
```

6.28. Running the code generator with Gradle

Run generation with the Gradle plugin

We recommend using the Gradle plugin by Etienne Studer (from Gradle Inc.). It provides a concise DSL that allows you to tune all configuration properties supported by each jOOQ version. Please direct any support questions or issues you may find directly to the third party plugin vendor.

Alternatively, the XML MarkupBuilder can be used

If you don't want to use the above third party plugin, there's also the possibility to use jOOQ's standalone code generator for simplicity. The following working example build.gradle script should work out of the box:

```
// Configure the Java plugin and the dependencies
// ----------------------------------------------
apply plugin: 'java'

repositories {
    mavenLocal()
    mavenCentral()
}

dependencies {
    compile 'org.jooq:jooq:3.11.11'

    runtime 'com.h2database:h2:1.4.177'
    testCompile 'junit:junit:4.11'
}

buildscript {
    repositories {
        mavenLocal()
        mavenCentral()
    }

    dependencies {
        classpath 'org.jooq:jooq-codegen:3.11.11'
        classpath 'com.h2database:h2:1.4.177'
    }
}

// Use your favourite XML builder to construct the code generation configuration file
// ----------------------------------------------------------------------------------
def writer = new StringWriter()
def xml = new groovy.xml.MarkupBuilder(writer)
.configuration('xmlns': 'http://www.jooq.org/xsd/jooq-codegen-3.11.0.xsd') {
    jdbc() {
        driver('org.h2.Driver')
        url('jdbc:h2:~/test-gradle')
        user('sa')
        password('')
    }
    generator() {
        database() {
        }

        // Watch out for this caveat when using MarkupBuilder with "reserved names"
        // - https://github.com/jOOQ/jOOQ/issues/4797
        // - http://stackoverflow.com/a/11389034/521799
        // - https://groups.google.com/forum/#!topic/jooq-user/wi4S9rRxk4A
        generate([:]) {
            pojos true
            daos true
        }
        target() {
            packageName('org.jooq.example.gradle.db')
            directory('src/main/java')
        }
    }
}

// Run the code generator
// ----------------------
org.jooq.codegen.GenerationTool.generate(writer.toString())
```

6.29. System properties governing code generation

Regardless if you're using a standalone code generation configuration, or if you're generating code with Maven, ant, or gradle, you can always provide default values for certain configuration elements through the following system properties:

- -Djooq.codegen.configurationFile (path): Specify an external configuration file, rather than using the inline configuration, e.g. in Maven
- -Djooq.codegen.jdbc.driver (class name): The JDBC driver to use for JDBC connection based code generation
- -Djooq.codegen.jdbc.url (url): The JDBC URL to use for JDBC connection based code generation
- -Djooq.codegen.jdbc.user (string): The JDBC user name to use for JDBC connection based code generation
- -Djooq.codegen.jdbc.username (string, same as user): The JDBC user name to use for JDBC connection based code generation
- -Djooq.codegen.jdbc.password (string): The JDBC password to use for JDBC connection based code generation
- -Djooq.codegen.logging (TRACE, DEBUG, INFO, WARN, ERROR, FATAL): The log level to use
- -Djooq.codegen.skip (boolean): Allows for skipping the execution of jOOQ code generation. Useful for larger builds, e.g. with Maven

In case of conflict between the above default value and a more concrete, local configuration, the latter prevails and the default is overridden.

7. Tools

These chapters hold some information about tools to be used with jOOQ

7.1. API validation using the Checker Framework

Java 8 introduced JSR 308 (type annotations) and with it, the Checker Framework was born. The Checker Framework allows for implementing compiler plugins that run sophisticated checks on your Java AST to introduce rich annotation based type semantics, e.g.

```
// This still compiles
@Positive int value1 = 1;

// This no longer compiles:
@Positive int value2 = -1;
```

jOOQ has two annotations that are very interesting for the Checker Framework to type check, namely:

- org.jooq.Support: This annotation documents jOOQ DSL API with valuable information about which database supports a given SQL clause or function, etc. For instance, only CUBRID, Informix, and Oracle currently support the CONNECT BY clause.
- org.jooq.PlainSQL: This annotation documents jOOQ DSL API which operates on plain SQL. Plain SQL being string-based SQL that is injected into a jOOQ expression tree, these API elements introduce a certain SQL injection risk (just like JDBC in general), if users are not careful.

Using the optional jooq-checker module (available only from Maven Central), users can now type-check their code to work only with a given set of dialects, or to forbid access to plain SQL.

Example:

A detailed blog post shows how this works in depth. By adding a simple dependency to your Maven build:

```
<dependency>
  <!-- Use org.jooq            for the Open Source edition
          org.jooq.pro        for commercial editions,
          org.jooq.pro-java-6 for commercial editions with Java 6 support,
          org.jooq.trial      for the free trial edition -->

  <groupId>org.jooq</groupId>
  <artifactId>jooq-checker</artifactId>
  <version>3.11.11</version>
</dependency>
```

... you can now include one of the two checkers:

SQLDialectChecker

The SQLDialect checker reads all of the org.jooq.Allow and org.jooq.Require annotations in your source code and checks if the jOOQ API you're using is allowed and/or required in a given context, where that context can be any scope, including:

- A package
- A class
- A method

Configure this compiler plugin:

```
<plugin>
  <artifactId>maven-compiler-plugin</artifactId>
  <version>3.3</version>
  <configuration>
    <source>1.8</source>
    <target>1.8</target>
    <fork>true</fork>
    <annotationProcessors>
      <annotationProcessor>org.jooq.checker.SQLDialectChecker</annotationProcessor>
    </annotationProcessors>
    <compilerArgs>
      <arg>-Xbootclasspath/p:1.8</arg>
    </compilerArgs>
  </configuration>
</plugin>
```

... annotate your packages, e.g.

```
// Scope: entire package (put in package-info.java)
@Allow(ORACLE)
package org.jooq.example.checker;
```

And now, you'll no longer be able to use any SQL Server specific functionality that is not available in Oracle, for instance. Perfect!

There are quite some delicate rules that play into this when you nest these annotations. Please refer to this blog post for details.

PlainSQLChecker

This checker is much simpler. Just add the following compiler plugin to deactivate plain SQL usage by default:

```
<plugin>
  <artifactId>maven-compiler-plugin</artifactId>
  <version>3.3</version>
  <configuration>
    <source>1.8</source>
    <target>1.8</target>
    <fork>true</fork>
    <annotationProcessors>
      <annotationProcessor>org.jooq.checker.PlainSQLChecker</annotationProcessor>
    </annotationProcessors>
    <compilerArgs>
      <arg>-Xbootclasspath/p:1.8</arg>
    </compilerArgs>
  </configuration>
</plugin>
```

From now on, you won't risk any SQL injection in your jOOQ code anymore, because your compiler will reject all such API usage. If, however, you need to place an exception on a given package / class / method, simply add the org.jooq.Allow.PlainSQL annotation, as such:

```
// Scope: Single method.
@Allow.PlainSQL
public List<Integer> iKnowWhatImDoing() {
    return DSL.using(configuration)
              .select(level())
              .connectBy("level < ?", bindValue)
              .fetch(0, int.class);
}
```

The Checker Framework does add some significant overhead in terms of compilation speed, and its IDE tooling is not yet at a level where such checks can be fed into IDEs for real user feedback, but the framework does work pretty well if you integrate it in your CI, nightly builds, etc.

7.2. SQL 2 jOOQ Parser

Together with Gudu Software, we have created an Open Source SQL 2 jOOQ parser that takes native SQL statements as input and generates jOOQ code as output.

Gudu Software Ltd has been selling enterprise quality SQL software to hundreds of customers to help them migrate from one database to another using the General SQL Parser. Now you can take advantage of their knowledge to parse your SQL statements and transform them directly into jOOQ statements using a free trial version of SQL 2 jOOQ!

It's as simple as this!

- Create a JDBC connection
- Create a new SQL2jOOQ converter object
- Convert your SQL code
- Get the result

See it in action:

```
package gudusoft.sql2jooq.readme;

import gudusoft.gsqlparser.EDbVendor;
import gudusoft.gsqlparser.sql2jooq.SQL2jOOQ;
import gudusoft.gsqlparser.sql2jooq.db.DatabaseMetaData;
import gudusoft.gsqlparser.sql2jooq.tool.DatabaseMetaUtil;

import java.sql.Connection;
import java.sql.DriverManager;

public class Test {
    public static void main(String[] args) throws Exception {

        // 1. Create a JDBC connection
        // ---------------------------
        String userName = "root";
        String password = "";
        String url = "jdbc:mysql://localhost:3306/guestbook";

        Class.forName("com.mysql.jdbc.Driver");
        Connection conn = DriverManager.getConnection(url, userName, password);

        // 2. Create a new SQL2jOOQ converter object
        // -----------------------------------------
        DatabaseMetaData metaData = DatabaseMetaUtil
            .getDataBaseMetaData(conn, "guestbook");

        SQL2jOOQ convert = new SQL2jOOQ(metaData,
            EDbVendor.dbvmysql,
            "select first_name, last_name from actor where actor_id = 1;");

        // 3. Convert your SQL code
        // ------------------------
        convert.convert();
        if (convert.getErrorMessage() != null) {
            System.err.println(convert.getErrorMessage());
            System.exit(-1);
        }

        // 4. Get the result
        // -----------------
        System.out.println(convert.getConvertResult());
    }
}
```

If all goes well, the above program yields:

```
DSLContext create = DSL.using(conn, SQLDialect.MYSQL);

Result<Record2<String, String>> result = create.select( Actor.ACTOR.FIRST_NAME, Actor.ACTOR.LAST_NAME )
    .from( Actor.ACTOR )
    .where( Actor.ACTOR.ACTOR_ID.eq( DSL.inline( UShort.valueOf( 1 ) ) ) ).fetch( );
```

SQL 2 jOOQ is a joint venture by Gudu Software Limited and Data Geekery GmbH. We will ship, test and maintain this awesome new addition with our own deliverables. So far, SQL 2 jOOQ supports the MySQL and PostgreSQL dialects and it is in an alpha stadium. Please, community, provide as much feedback as possible to make this great tool rock even more!

Please take note of the fact that the sql2jooq library is Open Source, but it depends on the commercial gsp.jar parser, whose trial licensing terms can be seen here:

https://github.com/sqlparser/sql2jooq/blob/master/sql2jooq/LICENSE-GSQLPARSER.txt

For more information about the General SQL Parser, please refer to the product blog.

Please report any issues, ideas, wishes to the jOOQ user group or the sql2jooq GitHub project.

7.3. jOOQ Console

The jOOQ Console is no longer supported or shipped with jOOQ 3.2+. You may still use the jOOQ 3.1 Console with new versions of jOOQ, at your own risk.

8. Reference

These chapters hold some general jOOQ reference information

8.1. Supported RDBMS

A list of supported databases

Every RDMBS out there has its own little specialties. jOOQ considers those specialties as much as possible, while trying to standardise the behaviour in jOOQ. In order to increase the quality of jOOQ, some 70 unit tests are run for syntax and variable binding verification, as well as some 400 integration tests with an overall of around 4000 queries for any of these databases:

- Aurora MySQL Edition
- Aurora PostgreSQL Edition
- Azure SQL Data Warehouse
- Azure SQL Database
- CUBRID
- DB2 LUW
- Derby
- Firebird
- H2
- HANA
- HSQLDB
- Informix
- Ingres
- MariaDB
- Microsoft Access
- MySQL
- Oracle
- PostgreSQL
- Redshift
- SQL Server
- SQLite
- Sybase Adaptive Server Enterprise
- Sybase SQL Anywhere
- Teradata
- Vertica

For an up-to-date list of currently supported RDBMS and minimal versions, please refer to http://www.jooq.org/legal/licensing/#databases.

8.2. Data types

There is always a small mismatch between SQL data types and Java data types. This is for two reasons:

- SQL data types are insufficiently covered by the JDBC API.
- Java data types are often less expressive than SQL data types

This chapter should document the most important notes about SQL, JDBC and jOOQ data types.

8.2.1. BLOBs and CLOBs

jOOQ aims for hiding all JDBC details from jOOQ client API. Specifically, java.sql.Clob and java.sql.Blob are quite "harsh" APIs with a few caveats that may even depend on JDBC driver specifics.

Clob and Blob are resources (but not java.lang.AutoCloseable!) with open connections to the database. This makes no sense in an ordinary jOOQ context, when eagerly fetching all the results through fetch() methods. fetchLazy() and fetchStream() might be candidates where Clob and Blob types could make sense as the underlying java.sql.ResultSet and java.sql.PreparedStatement are still open while consuming these resources.

ByteArrayInputStream and ByteArrayOutputStream on the other hand are two different types which cannot be represented as a single Field<T> type. If either would be chosen as the <T> type, we'd get read-only or write-only fields. So for full lazy streaming support, we'd need another 2-way wrapper type, similar to Clob and Blob.

In many cases, streaming binary data isn't really necessary as thebyte[] can be easily kept in memory (and it is done so for further processing anyway, e.g. when working with images), so the extra work might not really be needed. This is particularly true in Oracle, where BLOBs are the only binary types in the absences of a formal (VAR)BINARY type, and CLOBs start at 4000 bytes.

Hence, jOOQ currently doesn't explicitly support JDBC BLOB and CLOB data types. If you use any of these data types in your database, jOOQ will map them to byte[] and String instead. In simple cases (small data), this simplification is sufficient. In more sophisticated cases, you may have to bypass jOOQ, in order to deal with these data types and their respective resources.

8.2.2. Unsigned integer types

Some databases explicitly support unsigned integer data types. In most normal JDBC-based applications, they would just be mapped to their signed counterparts letting bit-wise shifting and tweaking to the user. jOOQ ships with a set of unsigned java.lang.Number implementations modelling the following types:

- org.jooq.types.UByte: Unsigned byte, an 8-bit unsigned integer
- org.jooq.types.UShort: Unsigned short, a 16-bit unsigned integer
- org.jooq.types.UInteger: Unsigned int, a 32-bit unsigned integer
- org.jooq.types.ULong: Unsigned long, a 64-bit unsigned integer

Each of these wrapper types extends java.lang.Number, wrapping a higher-level integer type, internally:

- UByte wraps java.lang.Short
- UShort wraps java.lang.Integer
- UInteger wraps java.lang.Long
- ULong wraps java.math.BigInteger

8.2.3. INTERVAL data types

jOOQ fills a gap opened by JDBC, which neglects an important SQL data type as defined by the SQL standards: INTERVAL types. SQL knows two different types of intervals:

- YEAR TO MONTH: This interval type models a number of months and years
- DAY TO SECOND: This interval type models a number of days, hours, minutes, seconds and milliseconds

Both interval types ship with a variant of subtypes, such as DAY TO HOUR, HOUR TO SECOND, etc. jOOQ models these types as Java objects extending java.lang.Number: org.jooq.types.YearToMonth (where Number.intValue() corresponds to the absolute number of months) and org.jooq.types.DayToSecond (where Number.intValue() corresponds to the absolute number of milliseconds)

Interval arithmetic

In addition to the arithmetic expressions documented previously, interval arithmetic is also supported by jOOQ. Essentially, the following operations are supported:

- DATETIME - DATETIME => INTERVAL
- DATETIME + or - INTERVAL => DATETIME
- INTERVAL + DATETIME => DATETIME
- INTERVAL + - INTERVAL => INTERVAL
- INTERVAL * or / NUMERIC => INTERVAL
- NUMERIC * INTERVAL => INTERVAL

8.2.4. XML data types

XML data types are currently not supported

8.2.5. Geospacial data types

Geospacial data types

Geospacial data types are currently not supported

8.2.6. CURSOR data types

Some databases support cursors returned from stored procedures. They are mapped to the following jOOQ data type:

```
Field<Result<Record>> cursor;
```

In fact, such a cursor will be fetched immediately by jOOQ and wrapped in an org.jooq.Result object.

8.2.7. ARRAY and TABLE data types

The SQL standard specifies ARRAY data types, that can be mapped to Java arrays as such:

```
Field<Integer[]> intArray;
```

The above array type is supported by these SQL dialects:

- H2
- HSQLDB
- Postgres

Oracle typed arrays

Oracle has strongly-typed arrays and table types (as opposed to the previously seen anonymously typed arrays). These arrays are wrapped by org.jooq.ArrayRecord types.

8.2.8. Oracle DATE data type

Oracle's DATE data type does not conform to the SQL standard. It is really a TIMESTAMP(0), i.e. a TIMESTAMP with a fractional second precision of zero. The most appropriate JDBC type for Oracle DATE types is java.sql.Timestamp.

Performance implications

When binding TIMESTAMP variables to SQL statements, instead of truncating such variables to DATE, the cost based optimiser may choose to widen the database column from DATE to TIMESTAMP using an Oracle INTERNAL_FUNCTION(), which prevents index usage. Details about this behaviour can be seen in this Stack Overflow question.

Use a data type binding to work around this issue

The best way to work around this issue is to implement a custom data type binding, which generates the CAST expression for every bind variable:

```
@Override
public final void sql(BindingSQLContext<Timestamp> ctx) throws SQLException {
        render.keyword("cast").sql('(')
              .visit(val(ctx.value()))
              .sql(' ').keyword("as date").sql(')');
}
```

Deprecated functionality

Historic versions of jOOQ used to support a <dateAsTimestamp/> flag, which can be used with the out-of-the-box org.jooq.impl.DateAsTimestampBinding as a custom data type binding:

```
<database>
  <!-- Use this flag to force DATE columns to be of type TIMESTAMP -->
  <dateAsTimestamp>true</dateAsTimestamp>

  <!-- Define a custom binding for such DATE as TIMESTAMP columns -->
  <forcedTypes>
    <forcedType>
      <userType>java.sql.Timestamp</userType>
      <binding>org.jooq.impl.DateAsTimestampBinding</binding>
      <types>DATE</types>
    </forcedType>
  </forcedTypes>
</database>
```

For more information, please refer to the manual's section about custom data type bindings.

8.3. SQL to DSL mapping rules

jOOQ takes SQL as an external domain-specific language and maps it onto Java, creating an internal domain-specific language. Internal DSLs cannot 100% implement their external language counter parts, as they have to adhere to the syntax rules of their host or target language (i.e. Java). This section explains the various problems and workarounds encountered and implemented in jOOQ.

SQL allows for "keywordless" syntax

SQL syntax does not always need keywords to form expressions. The UPDATE .. SET clause takes various argument assignments:

```
UPDATE t SET a = 1, b = 2
```

```
update(t).set(a, 1).set(b, 2)
```

The above example also shows missing operator overloading capabilities, where "=" is replaced by "," in jOOQ. Another example are row value expressions, which can be formed with parentheses only in SQL:

```
(a, b) IN ((1, 2), (3, 4))
```

```
row(a, b).in(row(1, 2), row(3, 4))
```

In this case, ROW is an actual (optional) SQL keyword, implemented by at least PostgreSQL.

SQL contains "composed" keywords

As most languages, SQL does not attribute any meaning to whitespace. However, whitespace is important when forming "composed" keywords, i.e. SQL clauses composed of several keywords. jOOQ follows standard Java method naming conventions to map SQL keywords (case-insensitive) to Java methods (case-sensitive, camel-cased). Some examples:

```
GROUP BY
ORDER BY
WHEN MATCHED THEN UPDATE
```

```
groupBy()
orderBy()
whenMatchedThenUpdate()
```

Future versions of jOOQ may use all-uppercased method names in addition to the camel-cased ones (to prevent collisions with Java keywords):

```
GROUP BY
ORDER BY
WHEN MATCHED THEN UPDATE
```

```
GROUP_BY()
ORDER_BY()
WHEN_MATCHED_THEN_UPDATE()
```

SQL contains "superfluous" keywords

Some SQL keywords aren't really necessary. They are just part of a keyword-rich language, the way Java developers aren't used to anymore. These keywords date from times when languages such as ADA, BASIC, COBOL, FORTRAN, PASCAL were more verbose:

- BEGIN .. END
- REPEAT .. UNTIL
- IF .. THEN .. ELSE .. END IF

jOOQ omits some of those keywords when it is too tedious to write them in Java.

```
CASE WHEN .. THEN .. END
```

```
decode().when(.., ..)
```

The above example omits THEN and END keywords in Java. Future versions of jOOQ may comprise a more complete DSL, including such keywords again though, to provide a more 1:1 match for the SQL language.

SQL contains "superfluous" syntactic elements

Some SQL constructs are hard to map to Java, but they are also not really necessary. SQL often expects syntactic parentheses where they wouldn't really be needed, or where they feel slightly inconsistent with the rest of the SQL language.

```
LISTAGG(a, b) WITHIN GROUP (ORDER BY c)
              OVER (PARTITION BY d)
```

```
listagg(a, b).withinGroupOrderBy(c)
             .over().partitionBy(d)
```

The parentheses used for the WITHIN GROUP (..) and OVER (..) clauses are required in SQL but do not seem to add any immediate value. In some cases, jOOQ omits them, although the above might be optionally re-phrased in the future to form a more SQLesque experience:

```
LISTAGG(a, b) WITHIN GROUP (ORDER BY c)
              OVER (PARTITION BY d)
```

```
listagg(a, b).withinGroup(orderBy(c))
             .over(partitionBy(d))
```

SQL uses some of Java's reserved words

Some SQL keywords map onto Java Language Keywords if they're mapped using camel-casing. These keywords currently include:

- CASE
- ELSE
- FOR

jOOQ replaces those keywords by "synonyms":

```
CASE .. ELSE
PIVOT .. FOR .. IN ..
```

```
decode() .. otherwise()
pivot(..).on(..).in(..)
```

There is more future collision potential with:

- BOOLEAN
- CHAR
- DEFAULT
- DOUBLE
- ENUM
- FLOAT
- IF
- INT
- LONG
- PACKAGE

SQL operators cannot be overloaded in Java

Most SQL operators have to be mapped to descriptive method names in Java, as Java does not allow operator overloading:

```
=
<>, !=
||
SET a = b
```

```
equal(), eq()
notEqual(), ne()
concat()
set(a, b)
```

For those users using jOOQ with Scala or Groovy, operator overloading and implicit conversion can be leveraged to enhance jOOQ:

```
=
<>, !=
||
```

```
===
<>, !==
||
```

SQL's reference before declaration capability

This is less of a syntactic SQL feature than a semantic one. In SQL, objects can be referenced before (i.e. "lexicographically before") they are declared. This is particularly true for aliasing

```
SELECT t.a
FROM my_table t
```

```
MyTable t = MY_TABLE.as("t");
select(t.a).from(t)
```

A more sophisticated example are common table expressions (CTE), which are currently not supported by jOOQ:

```
WITH t(a, b) AS (
  SELECT 1, 2 FROM DUAL
)
SELECT t.a, t.b
FROM t
```

Common table expressions define a "derived column list", just like table aliases can do. The formal record type thus created cannot be typesafely verified by the Java compiler, i.e. it is not possible to formally dereference t.a from t.

8.4. Quality Assurance

jOOQ is running some of your most mission-critical logic: the interface layer between your Java / Scala application and the database. You have probably chosen jOOQ for any of the following reasons:

- To evade JDBC's verbosity and error-proneness due to string concatenation and index-based variable binding
- To add lots of type-safety to your inline SQL
- To increase productivity when writing inline SQL using your favourite IDE's autocompletion capabilities

With jOOQ being in the core of your application, you want to be sure that you can trust jOOQ. That is why jOOQ is heavily unit and integration tested with a strong focus on integration tests:

Unit tests

Unit tests are performed against dummy JDBC interfaces using http://jmock.org/. These tests verify that various org.jooq.QueryPart implementations render correct SQL and bind variables correctly.

Integration tests

This is the most important part of the jOOQ test suites. Some 1500 queries are currently run against a standard integration test database. Both the test database and the queries are translated into every one of the 14 supported SQL dialects to ensure that regressions are unlikely to be introduced into the code base.

For libraries like jOOQ, integration tests are much more expressive than unit tests, as there are so many subtle differences in SQL dialects. Simple mocks just don't give as much feedback as an actual database instance.

jOOQ integration tests run the weirdest and most unrealistic queries. As a side-effect of these extensive integration test suites, many corner-case bugs for JDBC drivers and/or open source databases have been discovered, feature requests submitted through jOOQ and reported mainly to CUBRID, Derby, H2, HSQLDB.

Code generation tests

For every one of the 14 supported integration test databases, source code is generated and the tiniest differences in generated source code can be discovered. In case of compilation errors in generated source code, new test tables/views/columns are added to avoid regressions in this field.

API Usability tests and proofs of concept

jOOQ is used in jOOQ-meta as a proof of concept. This includes complex queries such as the following Postgres query

```
Routines r1 = ROUTINES.as("r1");
Routines r2 = ROUTINES.as("r2");

for (Record record : create.select(
        r1.ROUTINE_SCHEMA,
        r1.ROUTINE_NAME,
        r1.SPECIFIC_NAME,

        // Ignore the data type when there is at least one out parameter
        DSL.when(exists(
                selectOne()
                .from(PARAMETERS)
                .where(PARAMETERS.SPECIFIC_SCHEMA.eq(r1.SPECIFIC_SCHEMA))
                .and(PARAMETERS.SPECIFIC_NAME.eq(r1.SPECIFIC_NAME))
                .and(upper(PARAMETERS.PARAMETER_MODE).ne("IN"))),
                    val("void"))
           .otherwise(r1.DATA_TYPE).as("data_type"),
        r1.CHARACTER_MAXIMUM_LENGTH,
        r1.NUMERIC_PRECISION,
        r1.NUMERIC_SCALE,
        r1.TYPE_UDT_NAME,

        // Calculate overload index if applicable
        DSL.when(
            exists(
                selectOne()
                .from(r2)
                .where(r2.ROUTINE_SCHEMA.in(getInputSchemata()))
                .and(r2.ROUTINE_SCHEMA.eq(r1.ROUTINE_SCHEMA))
                .and(r2.ROUTINE_NAME.eq(r1.ROUTINE_NAME))
                .and(r2.SPECIFIC_NAME.ne(r1.SPECIFIC_NAME))),
            select(count())
                .from(r2)
                .where(r2.ROUTINE_SCHEMA.in(getInputSchemata()))
                .and(r2.ROUTINE_SCHEMA.eq(r1.ROUTINE_SCHEMA))
                .and(r2.ROUTINE_NAME.eq(r1.ROUTINE_NAME))
                .and(r2.SPECIFIC_NAME.le(r1.SPECIFIC_NAME)).asField())
        .as("overload"))
    .from(r1)
    .where(r1.ROUTINE_SCHEMA.in(getInputSchemata()))
    .orderBy(
        r1.ROUTINE_SCHEMA.asc(),
        r1.ROUTINE_NAME.asc())
    .fetch()) {

    result.add(new PostgresRoutineDefinition(this, record));
}
```

These rather complex queries show that the jOOQ API is fit for advanced SQL use-cases, compared to the rather simple, often unrealistic queries in the integration test suite.

Clean API and implementation. Code is kept DRY

As a general rule of thumb throughout the jOOQ code, everything is kept DRY. Some examples:

- There is only one place in the entire code base, which consumes values from a JDBC ResultSet
- There is only one place in the entire code base, which transforms jOOQ Records into custom POJOs

Keeping things DRY leads to longer stack traces, but in turn, also increases the relevance of highly reusable code-blocks. Chances that some parts of the jOOQ code base slips by integration test coverage decrease significantly.

8.5. Migrating to jOOQ 3.0

This section is for all users of jOOQ 2.x who wish to upgrade to the next major release. In the next sub-sections, the most important changes are explained. Some code hints are also added to help you fix compilation errors.

Type-safe row value expressions

Support for row value expressions has been added in jOOQ 2.6. In jOOQ 3.0, many API parts were thoroughly (but often incompatibly) changed, in order to provide you with even more type-safety.

Here are some affected API parts:

- [N] in Row[N] has been raised from 8 to 22. This means that existing row value expressions with degree >= 9 are now type-safe
- Subqueries returned from DSL.select(...) now implement Select<Record[N]>, not Select<Record>
- IN predicates and comparison predicates taking subselects changed incompatibly
- INSERT and MERGE statements now take typesafe VALUES() clauses

Some hints related to row value expressions:

```
// SELECT statements are now more typesafe:
Record2<String, Integer> record         = create.select(BOOK.TITLE, BOOK.ID).from(BOOK).where(ID.eq(1)).fetchOne();
Result<Record2<String, Integer>> result = create.select(BOOK.TITLE, BOOK.ID).from(BOOK).fetch();

// But Record2 extends Record. You don't have to use the additional typesafety:
Record record    = create.select(BOOK.TITLE, BOOK.ID).from(BOOK).where(ID.eq(1)).fetchOne();
Result<?> result = create.select(BOOK.TITLE, BOOK.ID).from(BOOK).fetch();
```

SelectQuery and SelectXXXStep are now generic

In order to support type-safe row value expressions and type-safe Record[N] types, SelectQuery is now generic: SelectQuery<R>

SimpleSelectQuery and SimpleSelectXXXStep API were removed

The duplication of the SELECT API is no longer useful, now that SelectQuery and SelectXXXStep are generic.

Factory was split into DSL (query building) and DSLContext (query execution)

The pre-existing Factory class has been split into two parts:

- The DSL: This class contains only static factory methods. All QueryParts constructed from this class are "unattached", i.e. queries that are constructed through DSL cannot be executed immediately. This is useful for subqueries.
  The DSL class corresponds to the static part of the jOOQ 2.x Factory type
- The DSLContext: This type holds a reference to a Configuration and can construct executable ("attached") QueryParts.
  The DSLContext type corresponds to the non-static part of the jOOQ 2.x Factory / FactoryOperations type.

The FactoryOperations interface has been renamed to DSLContext. An example:

```
// jOOQ 2.6, check if there are any books
Factory create = new Factory(connection, dialect);
create.selectOne()
      .whereExists(
        create.selectFrom(BOOK) // Reuse the factory to create subselects
      ).fetch();                // Execute the "attached" query

// jOOQ 3.0
DSLContext create = DSL.using(connection, dialect);
create.selectOne()
      .whereExists(
        selectFrom(BOOK)        // Create a static subselect from the DSL
      ).fetch();                // Execute the "attached" query
```

Quantified comparison predicates

Field.equalAny(...) and similar methods have been removed in favour of Field.eq(any(...)). This greatly simplified the Field API. An example:

```
// jOOQ 2.6
Condition condition = BOOK.ID.equalAny(create.select(BOOK.ID).from(BOOK));

// jOOQ 3.0 adds some typesafety to comparison predicates involving quantified selects
QuantifiedSelect<Record1<Integer>> subselect = any(select(BOOK.ID).from(BOOK));
Condition condition = BOOK.ID.eq(subselect);
```

FieldProvider

The FieldProvider marker interface was removed. Its methods still exist on FieldProvider subtypes. Note, they have changed names from getField() to field() and from getIndex() to indexOf()

GroupField

GroupField has been introduced as a DSL marker interface to denote fields that can be passed to GROUP BY clauses. This includes all org.jooq.Field types. However, fields obtained from ROLLUP(), CUBE(), and GROUPING SETS() functions no longer implement Field. Instead, they only implement GroupField. An example:

```
// jOOQ 2.6
Field<?>   field1a = Factory.rollup(...); // OK
Field<?>   field2a = Factory.one();       // OK

// jOOQ 3.0
GroupField field1b = DSL.rollup(...); // OK
Field<?>   field1c = DSL.rollup(...); // Compilation error
GroupField field2b = DSL.one();       // OK
Field<?>   field2c = DSL.one();       // OK
```

NULL predicate

Beware! Previously, Field.eq(null) was translated internally to an IS NULL predicate. This is no longer the case. Binding Java "null" to a comparison predicate will result in a regular comparison predicate (which never returns true). This was changed for several reasons:

- To most users, this was a surprising "feature".
- Other predicates didn't behave in such a way, e.g. the IN predicate, the BETWEEN predicate, or the LIKE predicate.
- Variable binding behaved unpredictably, as IS NULL predicates don't bind any variables.
- The generated SQL depended on the possible combinations of bind values, which creates unnecessary hard-parses every time a new unique SQL statement is rendered.

Here is an example how to check if a field has a given value, without applying SQL's ternary NULL logic:

```
String possiblyNull = null; // Or else...

// jOOQ 2.6
Condition condition1 = BOOK.TITLE.eq(possiblyNull);

// jOOQ 3.0
Condition condition2 = BOOK.TITLE.eq(possiblyNull).or(BOOK.TITLE.isNull().and(val(possiblyNull).isNull()));
Condition condition3 = BOOK.TITLE.isNotDistinctFrom(possiblyNull);
```

Configuration

DSLContext, ExecuteContext, RenderContext, BindContext no longer extend Configuration for "convenience". From jOOQ 3.0 onwards, composition is chosen over inheritance as these objects are not really configurations. Most importantly

- DSLContext is only a DSL entry point for constructing "attached" QueryParts
- ExecuteContext has a well-defined lifecycle, tied to that of a single query execution
- RenderContext has a well-defined lifecycle, tied to that of a single rendering operation
- BindContext has a well-defined lifecycle, tied to that of a single variable binding operation

In order to resolve confusion that used to arise because of different lifecycle durations, these types are now no longer formally connected through inheritance.

ConnectionProvider

In order to allow for simpler connection / data source management, jOOQ externalised connection handling in a new ConnectionProvider type. The previous two connection modes are maintained backwards-compatibly (JDBC standalone connection mode, pooled DataSource mode). Other connection modes can be injected using:

```
public interface ConnectionProvider {

    // Provide jOOQ with a connection
    Connection acquire() throws DataAccessException;

    // Get a connection back from jOOQ
    void release(Connection connection) throws DataAccessException;
}
```

These are some side-effects of the above change

- Connection-related JDBC wrapper utility methods (commit, rollback, etc) have been moved to the new DefaultConnectionProvider. They're no longer available from the DSLContext. This had been confusing to some users who called upon these methods while operating in pool DataSource mode.

ExecuteListeners

ExecuteListeners can no longer be configured via Settings. Instead they have to be injected into the Configuration. This resolves many class loader issues that were encountered before. It also helps listener implementations control their lifecycles themselves.

Data type API

The data type API has been changed drastically in order to enable some new DataType-related features. These changes include:

- [SQLDialect]DataType and SQLDataType no longer implement DataType. They're mere constant containers
- Various minor API changes have been done.

Object renames

These objects have been moved / renamed:

- jOOU: a library used to represent unsigned integer types was moved from org.jooq.util.unsigned to org.jooq.util.types (which already contained INTERVAL data types)

Feature removals

Here are some minor features that have been removed in jOOQ 3.0

- The ant task for code generation was removed, as it was not up to date at all. Code generation through ant can be performed easily by calling jOOQ's GenerationTool through a <java> target.
- The navigation methods and "foreign key setters" are no longer generated in Record classes, as they are useful only to few users and the generated code is very collision-prone.
- The code generation configuration no longer accepts comma-separated regular expressions. Use the regex pipe | instead.
- The code generation configuration can no longer be loaded from .properties files. Only XML configurations are supported.
- The master data type feature is no longer supported. This feature was unlikely to behave exactly as users expected. It is better if users write their own code generators to generate master enum data types from their database tables. jOOQ's enum mapping and converter features sufficiently cover interacting with such user-defined types.
- The DSL subtypes are no longer instanciable. As DSL now only contains static methods, subclassing is no longer useful. There are still dialect-specific DSL types providing static methods for dialect-specific functions. But the code-generator no longer generates a schema-specific DSL
- The concept of a "main key" is no longer supported. The code generator produces UpdatableRecords only if the underlying table has a PRIMARY KEY. The reason for this removal is the fact that "main keys" are not reliable enough. They were chosen arbitrarily among UNIQUE KEYs.
- The UpdatableTable type has been removed. While adding significant complexity to the type hierarchy, this type adds not much value over a simple Table.getPrimaryKey() != null check.
- The USE statement support has been removed from jOOQ. Its behaviour was ill-defined, while it didn't work the same way (or didn't work at all) in some databases.

8.6. Credits

jOOQ lives in a very challenging ecosystem. The Java to SQL interface is still one of the most important system interfaces. Yet there are still a lot of open questions, best practices and no "true" standard has been established. This situation gave way to a lot of tools, APIs, utilities which essentially tackle the same problem domain as jOOQ. jOOQ has gotten great inspiration from pre-existing tools and this section should give them some credit. Here is a list of inspirational tools in alphabetical order:

- Hibernate: The de-facto standard (JPA) with its useful table-to-POJO mapping features have influenced jOOQ's org.jooq.ResultQuery facilities
- JaQu: H2's own fluent API for querying databases
- JPA: The de-facto standard in the javax.persistence packages, supplied by Oracle. Its annotations are useful to jOOQ as well.
- OneWebSQL: A commercial SQL abstraction API with support for DAO source code generation, which was integrated also in jOOQ
- QueryDSL: A "LINQ-port" to Java. It has a similar fluent API, a similar code-generation facility, yet quite a different purpose. While jOOQ is all about SQL, QueryDSL (like LINQ) is mostly about querying.
- SLICK: A "LINQ-like" database abstraction layer for Scala. Unlike LINQ, its API doesn't really remind of SQL. Instead, it makes SQL look like Scala.
- Spring Data: Spring's JdbcTemplate knows RowMappers, which are reflected by jOOQ's RecordHandler or RecordMapper