MARTINELLI
Keep IT simple

jOOQ Workshop

Simon Martinelli
Martinelli GmbH

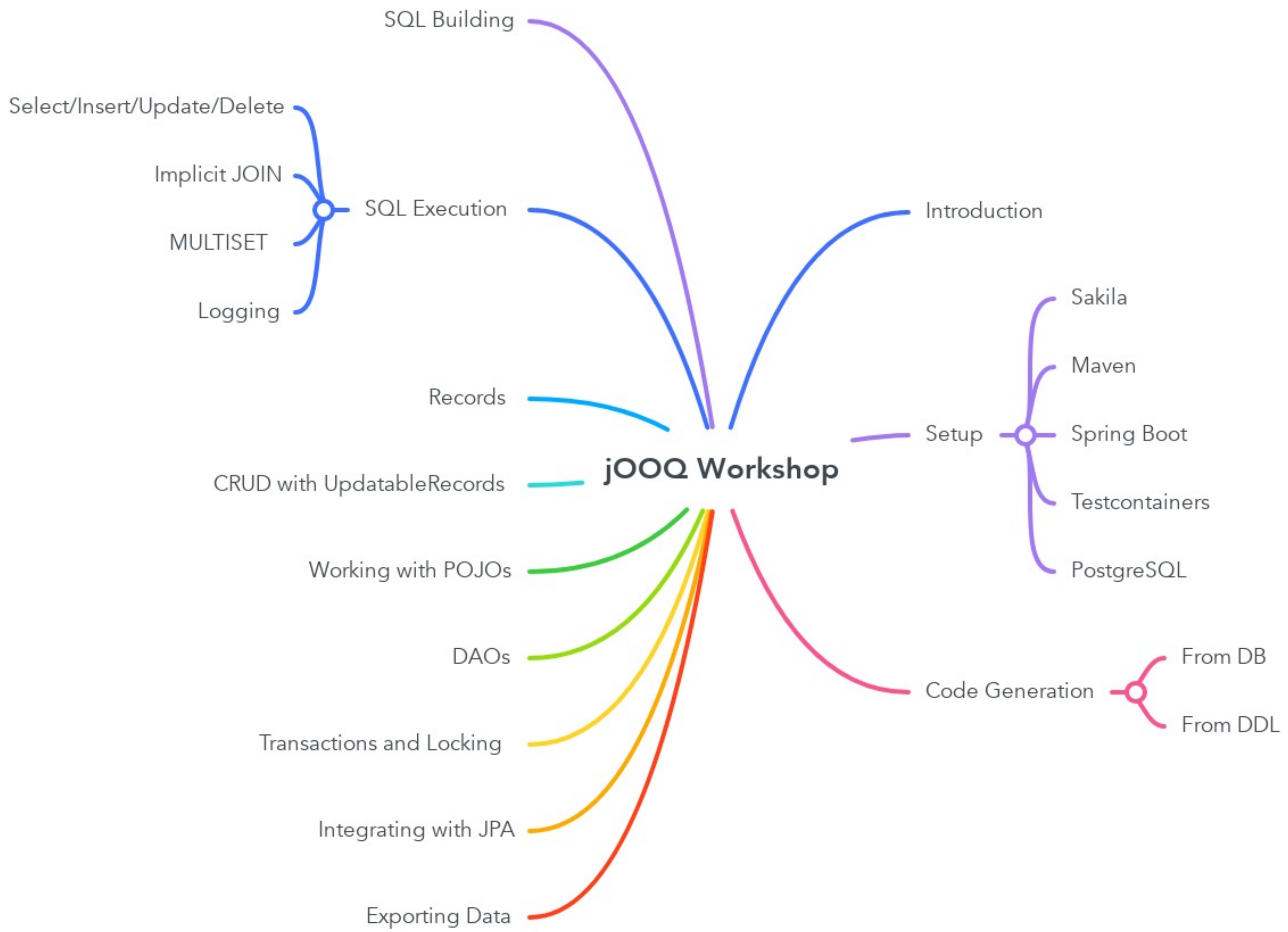
About Me

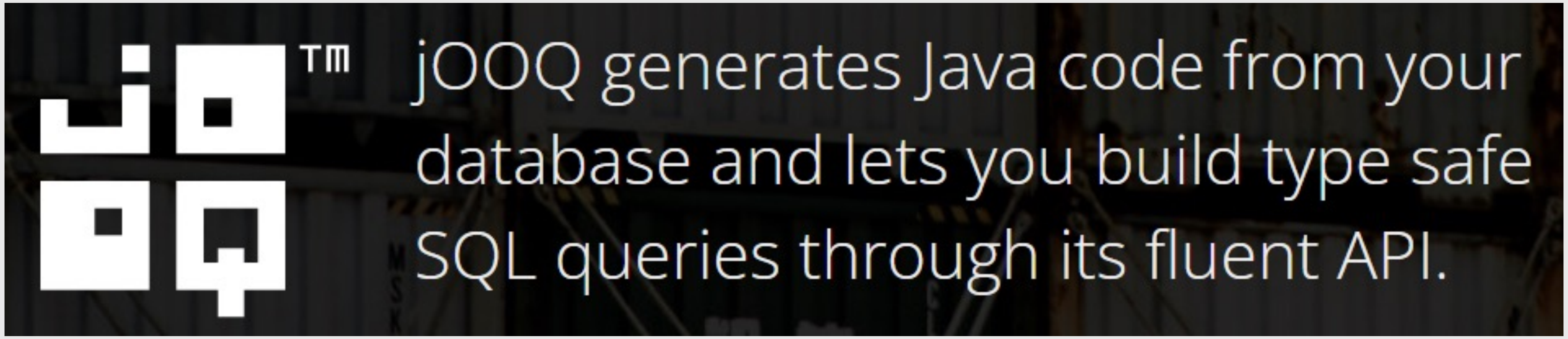
- 30 years of Software Engineering
- Java for 25 years
- Self-employed since 2009
- University teacher for 18 years
- JUG Switzerland, Location Bern

Java Champions

vaadin}> Champion

Oracle ACE Pro

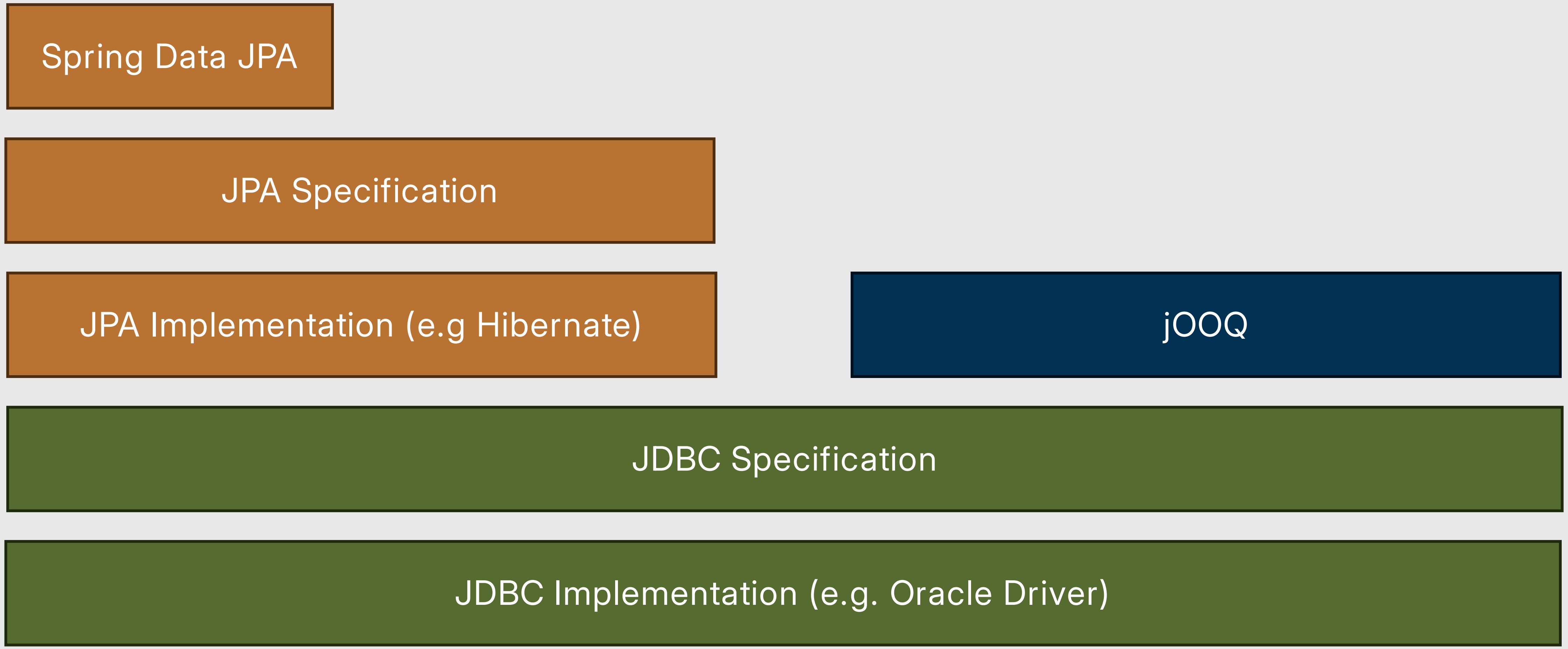

jOOQ Workshop
Introduction
Setup
Sakila
Maven
Spring Boot
Testcontainers
PostgreSQL
Code Generation
From DB
From DDL
SQL Building
SQL Execution
Select/Insert/Update/Delete
Implicit JOIN
MULTISET
Logging
Records
CRUD with UpdatableRecords
Working with POJOs
DAOs
Transactions and Locking
Integrating with JPA
Exporting Data



jOOQ generates Java code from your database and lets you build type safe SQL queries through its fluent API.

Introduction

Stack


Spring Data JPA
JPA Specification
JPA Implementation (e.g Hibernate)
jOOQ
JDBC Specification
JDBC Implementation (e.g. Oracle Driver)

Why jOOQ?

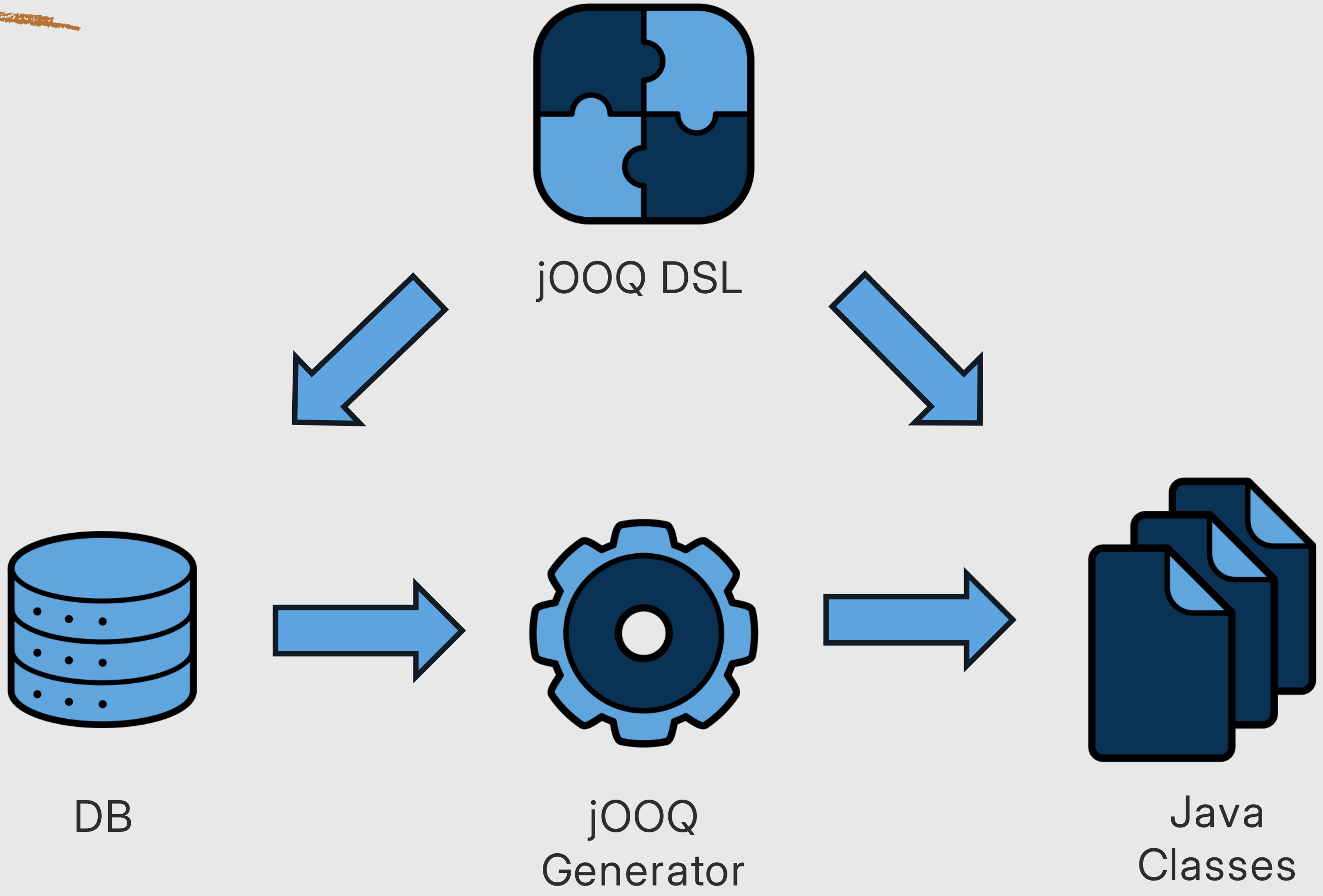
- **Type Safety**
  - Compile-time checking prevents errors
  - No String concatenation
  - DSL allows to re-use parts of the statements
  - Typed results, also nested
- **Database First**
  - Works with existing schemas
- **Full SQL Power**
  - No limitations like JPA

Set-based Thinking

- Most **conceptual differences** between JPA and jOOQ are not technology-specific, but a matter of how you think about your database interactions
- There are **two approaches**
  - Working with entity state transitions
  - Working with data set transformations

- **Neither approach is "the best" one**; both approaches are better suited to certain use cases

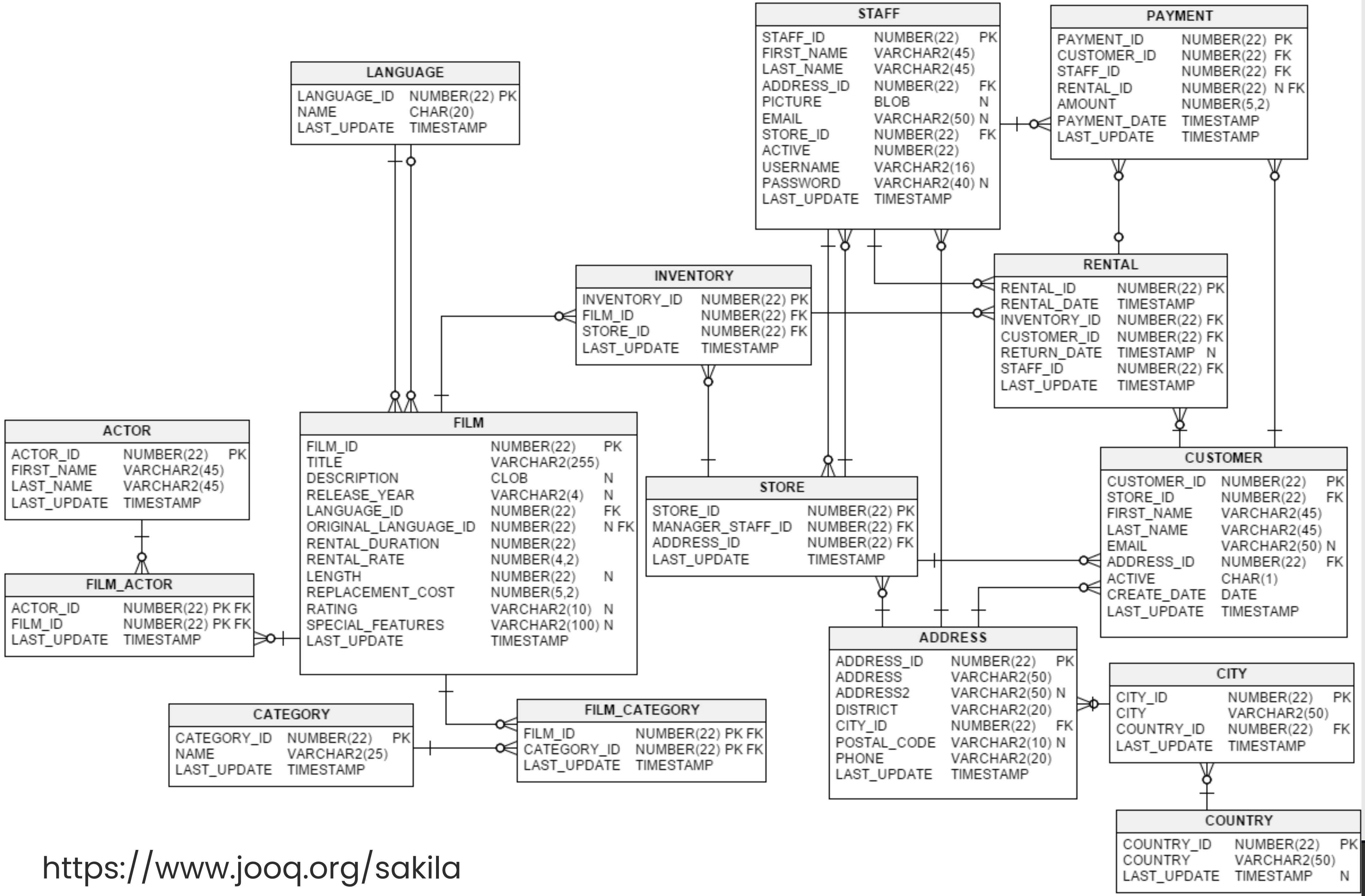
Architecture



Where to Start?

- **Exercises and Demo**
    - https://github.com/martinellich/jooq-workshop
- **Documentation**
    - https://jooq.org
- **Examples**
    - https://github.com/jOOQ/demo

Setup



https://www.jooq.org/sakila

Tools

- Maven
- Spring Boot
- Testcontainers
- PostgreSQL

Testcontainers

- https://testcontainers.com/
- Lifecycle management https://testcontainers.com/guides/testcontainers-container-lifecycle/

Code Generation

https://www.jooq.org/doc/latest/manual/code-generation/

Why?

- **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 also change, removing columns will lead to compilation errors, which you can detect early

What's the Source?

- **jOOQ assumes your database already exists**

- There are **two options**
  1. **Real Database**
     - Use all DB features
     - Use reference database or Testcontainers
  2. **DDL scripts**
     - Limited to "standard" SQL
     - Convenient because no running DB is needed

How?

- https://www.jooq.org/doc/latest/manual/code-generation/codegen-configuration/

Ex00: Get Familiar With the Project

- Inspect the project
  https://github.com/simasch/jooq-workshop

  1. pom.xml
  2. DDL Scripts

- Run
  `mvnw test`

SQL Building

https://www.jooq.org/doc/latest/manual/sql-building/

The Query DSL Type

- **Problem**
  - **SQL** is a **declarative** language that is hard to integrate into procedural, object-oriented, functional, or any other programming language
- **Solution**
  - jOOQ integrates SQL as an "**internal domain-specific language**" directly into Java
  - SQL building is the main feature of jOOQ
  - All other features (such as SQL execution and code generation) are mere conveniences built on top of jOOQ’s SQL-building capabilities

The Static Query DSL API

- **jOOQ exposes many 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 is a static factory for table expressions, column expressions (or “fields”), conditional expressions, and many other QueryParts.

The DSLContext API

- https://www.jooq.org/doc/latest/manual/sql-building/dsl-context/

Settings

- https://www.jooq.org/doc/current/manual/sql-building/dsl-context/custom-settings/

```
@Configuration
public class SakilaJooqConfiguration {

  @Bean
  public DefaultConfigurationCustomizer configurationCustomizer() {
     return c -> {
          c.set(new SetSessionUserExecuteListener());
          c.settings().withExecuteWithOptimisticLocking(true).withFetchSize(100);
     };
  }
}
```

DSLContext Example

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


Result<?> result = dsl
                   .select()
                   .from(BOOK)
                   .where(BOOK.TITLE.like("Animal%"))
                   .fetch();
```

JOIN

- **“Regular” JOINS**

```
join(FILM_ACTOR)
.on(FILM_ACTOR.ACTOR_ID.eq(ACTOR.ACTOR_ID))
```

- **Implicit JOINS**

```
select(FILM_ACTOR.actor().LAST_NAME)
```

Superpower MULTISET

- https://www.jooq.org/doc/current/manual/sql-building/column-expressions/multiset-value-constructor/

Ex01: Write Your First jOOQ Queries

- Create your very first jOOQ query that selects all Actors
  - Inspect the generated SQL statement
- Get all names of the Films, incl. the Language
  - Try the various JOINs incl. IMPLICIT JOIN
- List all Actors first and last name with their number of Films
- Evaluate MULTISET
  - Load the name of the Category and the name of the Film as nested element

SQL Execution

https://www.jooq.org/doc/latest/manual/sql-execution/

SQL Execution with JDBC

- JDBC calls executable objects "java.sql.Statement".
  It distinguishes between three types of statements:
  - `java.sql.Statement`, or "static statement"
  - `java.sql.PreparedStatement`
  - `java.sql.CallableStatement`

- These things are abstracted away by jOOQ, which exposes such concepts in a more object-oriented way

Comparing jOOQ and JDBC

- https://www.jooq.org/doc/latest/manual/sql-execution/comparison-with-jdbc/

Fetching

- The standard fetch
  - **`Result<R> fetch()`**
- When you know your query returns at most one record. This may return null.
  - **`R fetchOne()`**
- When you know your query returns exactly one record. This never returns null but will throw an exception if no value is present.
  - **`R fetchSingle()`**
- When you know your query returns at most one record.
  - **`Optional<R> fetchOptional()`**

Ex02: Use Different Return Types

- Select the first and last name of the first Actor
- Select a list of Films ordered by release year, ascending
- Create a query that returns a Customer as Optional

jOOQ Records

https://www.jooq.org/doc/latest/manual/sql-execution/fetching/record-vs-tablerecord/

Fetching Records

```
BookRecord book = dsl
                   .selectFrom(BOOK)
                   .where(BOOK.ID.eq(1))
                   .fetchOne();


// Typesafe field access
System.out.println("Title: " + book.getTitle());
```

Record1 to Record22

- Type-safety is also applied to records for degrees up to 22 (because of Scala)
- To express this fact, `org.jooq.Record` is extended by `org.jooq.Record1` to `org.jooq.Record22`

Example Record2

```
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();
}
```

Arrays, Maps and Lists

- https://www.jooq.org/doc/latest/manual/sql-execution/fetching/arrays-maps-and-lists/

Handling Records

- **RecordMapper**
  https://www.jooq.org/doc/latest/manual/sql-execution/fetching/recordmapper/

- **Example**

```
List<Book> books =
   create.select(BOOK.ID, BOOK.TITLE)
         .from(BOOK)
         .orderBy(BOOK.ID)
         .fetch(Records.mapping(Book::new));
```

Ex03: Records are your Friends

- Return the Film title and length ordered by length descending
  - Use a `RecordHandler` to log the result

- Select the Film but return only a List of IDs

CRUD with UpdatableRecord

https://www.jooq.org/doc/latest/manual/sql-execution/crud-with-updatablerecords/

CRUD

- Your database application probably consists of **50% - 80% CRUD**, but **only 20% - 50% of querying**
  - Create (INSERT)
  - Read (SELECT)
  - Update (UPDATE)
  - Delete (DELETE)
- **CRUD** always uses the same patterns and **leads to a lot of boilerplate code**
- jOOQ facilitates CRUD using a specific API involving **`org.jooq.UpdatableRecord`** types

Primary Keys and Updatability

```
-- Inserting uses a previously generated key value -
-- or generates it afresh
INSERT INTO BOOK (ID, TITLE)
       VALUES (5, 'Animal Farm');


-- Other operations can use the generated key value
SELECT * FROM BOOK WHERE ID = 5;
UPDATE BOOK SET TITLE = '1984' WHERE ID = 5;
DELETE FROM BOOK WHERE ID = 5;
```

Simple CRUD with UpdatableRecord

```
// Store (insert or update) a record to the DB
int store() throws DataAccessException;

// Delete a record from the database
int delete() throws DataAccessException;

// Refresh a record from the database
void refresh() throws DataAccessException;
```

Ex04: Fun with Updatable Records

- A Customer wants to rent a film

1. Find the Staff by e-mail
2. Find the Customer by first and last name
3. Find an Inventory by the Film name
4. Create and save the Rental

Working with POJOs

https://www.jooq.org/doc/latest/manual/sql-execution/fetching/pojos/

Usage

```
// A "mutable" POJO class
public class MyBook1 {
  public int id; public String title;
}

// Fetching records into your custom POJOs with reflection
MyBook1 myBook = dsl.select().from(BOOK).fetchAny().into(MyBook1.class);
List<MyBook1> myBooks = dsl.select().from(BOOK).fetch().into(MyBook1.class);
List<MyBook1> myBooks = dsl.select().from(BOOK).fetchInto(MyBook1.class);

// Best option use a Record Mapper
List<MyBook1> myBooks = dsl.select().from(BOOK).fetch(Records.mapping(MyBook1::new));
```

Storing POJOs

```java
// A "mutable" POJO class
public class MyBook {
  public int id;
  public String title;
}
```

```java
// Create a new POJO instance
MyBook myBook = new MyBook();
myBook.id = 10;
myBook.title = "Animal Farm";

// Populate a jOOQ-generated
// BookRecord from your POJO
BookRecord book =
dsl.newRecord(BOOK, myBook);

// Insert it (implicitly)
book.store();
```

Ex05: The Joy of POJOs

- Like in Ex03, return the Film title and length ordered by length descending like
  - But this time, create a Java Record FilmAndLength and use it as the result

- Use a nested Java Record to hold the result of the MULTISET exercise from Ex03

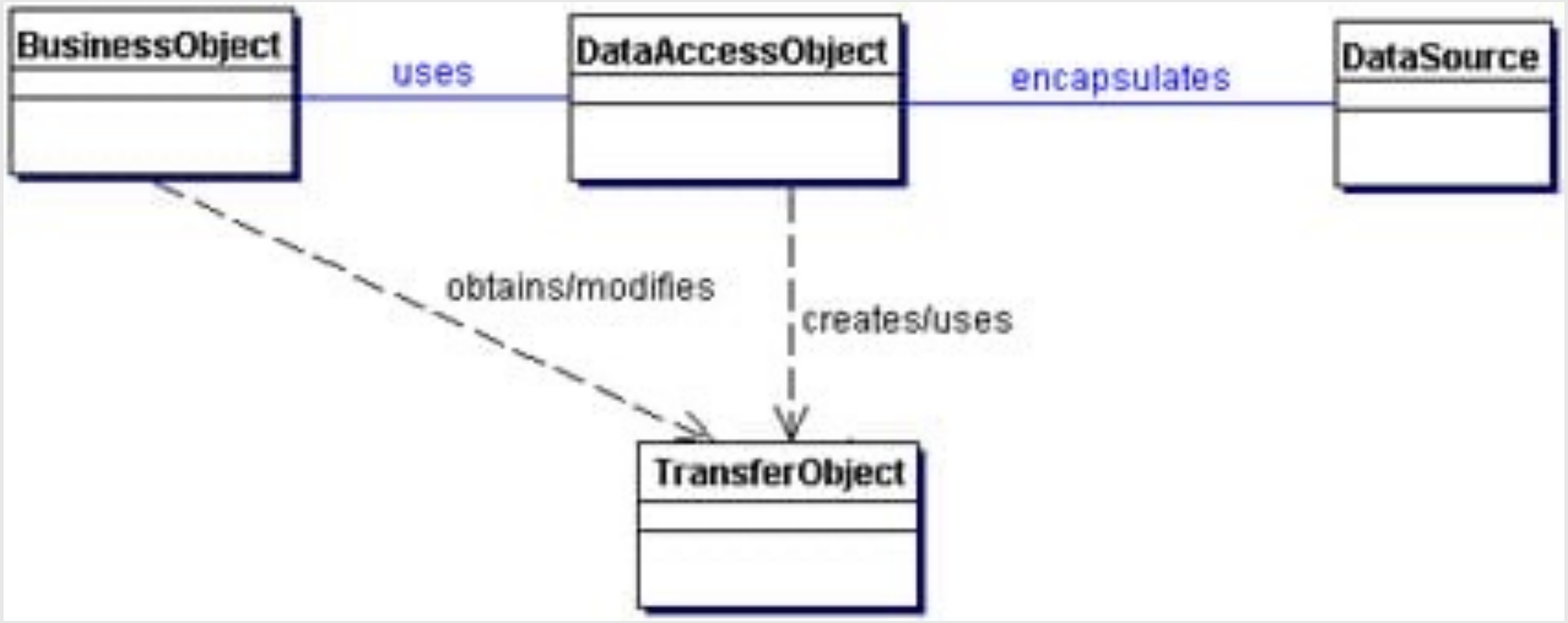
DAOs

https://www.jooq.org/doc/current/manual/sql-execution/daos/

Data Access Object (DAO)

- Access to data varies depending on the source of the data.
- Access to persistent storage, such as to a database, varies greatly depending on the type of storage (relational databases, object-oriented databases, flat files, and so forth) and the vendor implementation

https://www.oracle.com/java/technologies/dataaccessobject.html


BusinessObject
uses
DataAccessObject
encapsulates
DataSource
obtains/modifies
creates/uses
TransferObject

DAO per UpdatableRecord

- 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`

Example

```java
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 a list
    public List<Book> fetchByAuthorId(Integer... values) { ... }
    public List<Book> fetchByTitle(String... values) { ... }
}
```

Ex06: DAOs are Convenient

- Inspect the generated DAOs. Which methods are generated, and which are inherited?

- Use the RentalDao to
    - Read by rental date
    - Update the rental date
    - Delete by id

Transactions and Locking

https://www.jooq.org/doc/current/manual/sql-execution/transaction-management/

https://www.jooq.org/doc/current/manual/sql-execution/crud-with-updatablerecords/optimistic-locking/

jOOQ and Transactions

- Use third-party libraries like Spring Transactions
- Use a JTA-compliant Java EE transaction manager from your container
- Call JDBC's `Connection.commit()`, `Connection.rollback()` and other methods on your JDBC driver
- You can issue vendor-specific COMMIT, ROLLBACK and other statements directly in your database
- You use jOOQ's transaction API

jOOQ Transactions

```
dsl.transaction((Configuration trx) -> {
    // Important: Use the DSLContext of the transaction
    AuthorRecord author = trx.dsl()
        .insertInto(AUTHOR, AUTHOR.FIRST_NAME, AUTHOR.LAST_NAME)
        .values("George", "Orwell")
        .returning()
        .fetchOne();

    // Implicit commit executed here
});
```

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

Optimistic Locking: Processing

1. Before UPDATE or DELETE statements, jOOQ will run a SELECT .. FOR UPDATE statement, pessimistically locking the record for the subsequent UPDATE/DELETE
2. 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 has been modified in the meantime
    - The record is successfully stored/deleted, if the record had not been modified in the meantime

Optimistic Locking: Comparison

- By default, jOOQ compares all fields
- But you can also use a TIMESTAMP or VERSION field

- Reference: https://www.jooq.org/doc/current/manual/code-generation/codegen-advanced/codegen-config-database/codegen-database-record-version-timestamp-fields/

DataAccessException

- **`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 meantime
- **`DataTypeException`**
  Something went wrong during the 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

Ex07: Locking and transactions understood

- Write a test that updates a record in parallel and try to produce a `DataChangedException`
  - Hint: You will need to start two threads and add a wait time in one of the threads

- Read about nested transactions
  - https://blog.jooq.org/nested-transactions-in-jooq/

Integrating with JPA

https://www.jooq.org/doc/current/manual/getting-started/jooq-and-jpa/

Using jOOQ with JPA Native 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();
```

Ex08: Hello JPA

- Create JPA entities for Category and Film and map the OneToMany relationship from Category to Film
- Construct a query with jOOQ and use it in a JPA native Query
  - Check if the result are Entities

Exporting Data

https://www.jooq.org/doc/current/manual/sql-execution/exporting/

Exporting

- jOOQ can export `Result<Record>` to
  - XML
  - CSV
  - JSON
  - HTML
  - Text
  - Charts

Ex08: Export it!

- Try the various export formats

Thank you!

- **Web**
  https://martinelli.ch
- **E-Mail**
  simon@martinelli.ch
- **X/Twitter**
  simas_ch
- **Bluesky**
  martinelli.ch