Lecture Note

Persistence

An Example Architecture for Encapsulation of Database Access in Java Systems

2nd Edition

Istvan Knoll
knol@ucn.dk

A reworked edition of Kis Boisen Hansen: “An Example Architecture for Encapsulation of Database Access in Java Systems”. Thanks to Mogens Holm Iversen for reviewing the kode and note. Istvan Knoll, 2018-09-16

Table of Contents

Persistence – 2^{nd} semester

This lecture note demonstrates an example of an architecture for building a stand-alone program, using object-oriented programming techniques to access a relational database system. This lecture note is accompanied by an example system which you can execute and modify.

JDBC – Java Database Connectivity

JDBC is the standard method of accessing a database from Java. The JDBC framework is part of the Java specification, and database vendors implement JDBC drivers for their products. Alternative JDBC drivers provided by third parties may also exist, e.g. open source and commercial projects. All the discussed methods for accessing a database utilize JDBC drivers in different ways.

Business Applications Today

Most modern business applications use an object-oriented technology for software development, but the underlying database systems are most often relational databases. Such applications must handle the transformation between data in an object-oriented model and data in a relational model.

When the transformation between the object-oriented and the relational models is defined, it is up to the application to implement the conversion. We discuss three different approaches:

- Brute force
- Data Access Objects (the DAO-pattern)
- Persistence frameworks / Object-relational mapping frameworks (ORMs)

Brute force

The strategy here is, that the business objects accesses the data source directly – typically through using the JDBC-drivers and providing the Structured Query Language (SQL) code that accesses the database to the model classes.

The Brute force approached is not a database encapsulation strategy. It is only used, when you do not have a database encapsulation layer. This approach is commonly used because it is simple. It requires that the application programmer has full knowledge of how domain objects interact with the database. This approached may be applied when the access to the database is simple and straight forward.

A more complex use of the underlying data calls for a better solution, which is the DAO-pattern as discussed below.

Data Access Objects (DAO-pattern)

Data access objects (DAO) de-couple the business objects from the database access code – the database access is encapsulated. Typically, there is one DAO-class for each business object. This class contains all the SQL-code for the access to the database for that business object. DAO-classes may interact to retrieve and store associations to other classes in the business layer.

The main advantage of using the DAO pattern is that there is no longer a direct connection to the database from the business object classes.

Persistence frameworks / Object-relational mapping frameworks (ORMs)

A persistence framework (PF) or object-relational mapping framework (ORM) encapsulates the database access completely. Instead of writing code which implements the logic for the database access, you define the meta data that represents the transformation from the object model to the relational model. We shall not investigate this on the second semester. Examples of such frameworks are Hibernate and other implementations of the Java Persistence API (JPA). Meta data is historically provided though XML-files, now mostly replaced by in-code annotations.

Choosing an Implementation Strategy for OO – RDB mapping

Since the business objects will have to know all the details of how the database is constructed, the Brute Force strategy yields code which has low cohesion and high coupling.

The Data Access Objects approach removes this need for knowledge of database details from the business objects. The database code is encapsulated in the DAOs, so business objects are de-coupled from database access. Implementing the DAO pattern requires you to write data access classes for each business object.

A persistence framework will handle the implementation of the mapping without extra code, but you will have to acquire and install an implementation of the chosen framework and provide meta data to the framework.

Brute Force would be an undesirable choice, since it would yield very badly designed code with low cohesion and high coupling and therefore a system that is hard to modify, extend and maintain.

We choose to implement a simple and specific example that shows how a mapping layer between the object model and the relational model can be implemented. This solution follows the Data Access Object approach and represents a good design that is easy to understand.

The Case: The Company Database

The example is the well-known Company example from the textbook [Elmasri].

The relational model

The relational model is given in [Elmasri] fig. 3.7 (6th edition) which is known from the classes on databases.

The Layered Architecture

We choose a classic layered architecture:

<table>
<tr><td colspan="2">User Interface Layer (“gui”)</td></tr>
<tr><td colspan="2">Control Layer (“ctrl”)</td></tr>
<tr><td>Model Layer (“model”)</td><td>Database Transformation Layer (“db”)</td></tr>
</table>

The control layer contains the classes that control the use cases; that is controlling business logic. The user interface layer classes are responsible for input/output from and to the user and simple input validations.

We choose an open architecture, so that model layer classes are used as data transfer objects (DTO) between the control layer and the user interface layer. (In a closed architecture, DTO’s would be used to move information between the control layer and the user interface layer, and a conversion between model layer objects and DTO’s would need to be implemented.)

The database transformation layer contains the DB-classes (Data Access Objects), which are responsible for communication with the database and for building model layer objects from data retrieved from the database. It also supports CRUD-operations.

In this example we only implement one user interface class and some of the classes and methods needed to implement the related functionality.

The role of the Database Transformation Layer in this architecture is to hide database-specific implementation details; this makes it possible to make changes in the database design and even shift to another DBMS without having to make changes in the rest of the application.

Data is entered to the system through the User Interface Layer. Model Layer objects are created in the User Interface Layer, which calls the Control Layer and passes the model objects to the Control Layer. Then the Control Layer calls the Database Transformation Layer and passes the Model Layer objects to the relevant database classes. The Database Transformation Layer then saves the information in the database.

To retrieve data from the database, a request is passed from the User Interface Layer to the Control Layer. The control layer passes the request to the relevant Database Access classes, which handle the database queries and build the model objects. These objects are eventually returned from the Database Transformation Layer via the Control Layer back to the User Interface Layer.

The layered architecture is explicit in the implementation (traceability), because each layer is implemented as a Java package.

The User Interface Layer

The responsibility of this layer is to communicate with the user. The user interface is implemented in the *gui* package. In this example only one class is implemented.

The Model Layer

The Model Layer is implemented using object-oriented principles. This implies that the associations and aggregations are implemented using object references. This layer is implemented in the *model* package.

java.util.ArrayList is used when there are multiple object references. Other solutions could be considered.

The Database Transformation Layer

The primary object of the example-application. Everything necessary for accessing the database is implemented in this layer. This encapsulation makes it easier to make changes to the database.

The following design decisions have been made for the database transformation layer:

- One database class for each model class (EmployeeDB, WorksOnDB, etc.)
- Each database class is responsible for handling the persistence of the corresponding model class
- Database classes are responsible for finding, updating, deleting and inserting to the database
- Database classes are responsible for handling associations and aggregations between objects in the model layer

Java Interfaces

The behavior of each database class is specified by an interface. The naming convention used for interfaces is XxxxxxDBIF.

```java
public interface EmployeeDBIF {
    List<Employee> findByName(String name, boolean fullAssociation)
        throws DataAccessException;

    Employee findBySSN(String ssn, boolean fullAssociation)
        throws DataAccessException;

    List<Employee> findAll(boolean fullAssociation)
        throws DataAccessException;

    Employee insert(Employee employee) throws DataAccessException;
}
```

These methods make it possible to search for all information about employees, search on primary key (SSN in the case of employees), on name and to insert new employees. You may want to add additional methods for searching by other parameters, (*findBy…*), updating employee information or deleting employees from the database. The insert method returns an Employee object, with attributes set to reflect what eventually was stored in the database. In all cases, if the operation failed, an exception is thrown, which is expected to wrap the original exception that was thrown by e.g. the JDBC-driver.

As the persistence layer is expected to abstract away the implementation details of *how* persistence is obtained, we also hide the exceptions, this implementation may throw. *SQLException* is thrown by the JDBC driver, but if the persistence layer uses comma separated files (CSV), serialization, JSON objects, temporary in-memory storage (e.g. container classes), or a web service, the exception from this layer would be different. Therefore, we introduce the *DataAccessException* class, which wraps the original exception object from the persistence implementation, making the persistence layer implementation transparent also in this regard to the rest of the application.

The Connection to the Database

The DBConnection class establishes the connection to the database via an JDBC-driver. The class is implemented using the Singleton pattern, since we only need one connection to the database and to avoid locking up resources on the database. With a central database that services multiple clients, resource hogging can become a serious issue, which is the main reason for restricting the per-application connection to a single instance.

Database Classes

The database classes retrieve the information required to build corresponding objects and their associations. If it is an insert or update the information in the object is inserted into the right table in the database. Depending on which direction associations go from a given object, retrieving data can cause a minor chain reaction. If multiple associations can be found, the database class may have to retrieve and build the associated objects as well. If circular references or bi-directional references exist, it is up to the programmer to break the chain.

In the example code, the problem is solved by adding a boolean parameter (*fullAssociation*) to the search methods in the Database Layer. If the parameter is *true,* the associated objects are fully retrieved and built. If it is *false*, an associated object is created, but only the field, corresponding to its primary key (i.e. unique identifier) is set.

A simple solution could be to take the foreign key up in the model layer instead of the references to the object. Instead, as discussed above, a “dummy” or placeholder object is created, which contains the foreign key, thereby preserving the object-oriented nature of the model.

The Database Transformation Layer doesn’t handle *object caching*. The objects are read from the database every time they are needed. Therefore, the same (database) record may be present in multiple copies as model layer objects in the application. It is important to remember, that even if two objects represent the same entity (e.g. *employee1* contains information about Joe, and so does *employee2*), object equality does not apply to them (*employee1 != employee2*), as both variables point to different objects. To implement meaningful comparison between objects, we override the ***public boolean equals(Object o)*** method, and return *true*, if the primary key (unique identifier field) are the same in both objects. As an example, the *equals* method on Employee should return *true*, if *employee1.ssn == employee2.ssn*.

The Database Transformation Layer is not supposed to handle statistics queries, where the answer may be summing, counting or averaging attributes. Neither is it a good fit for handling queries that join multiple tables. If such needs arise, a solution would be to create a ***view*** in the database, and to create a database class for this view in the application.

The implementation

The relational model is implemented as a MS-SQL database. You can run the accompanying SQL-script to create and populate the test database. Once the database has been created, you can reset the database from inside the example implementation.

Implementation of the Model Layer

The model layer is implemented with the visibility shown in the design class diagram. There is a bi-directional association between Employee and Department.

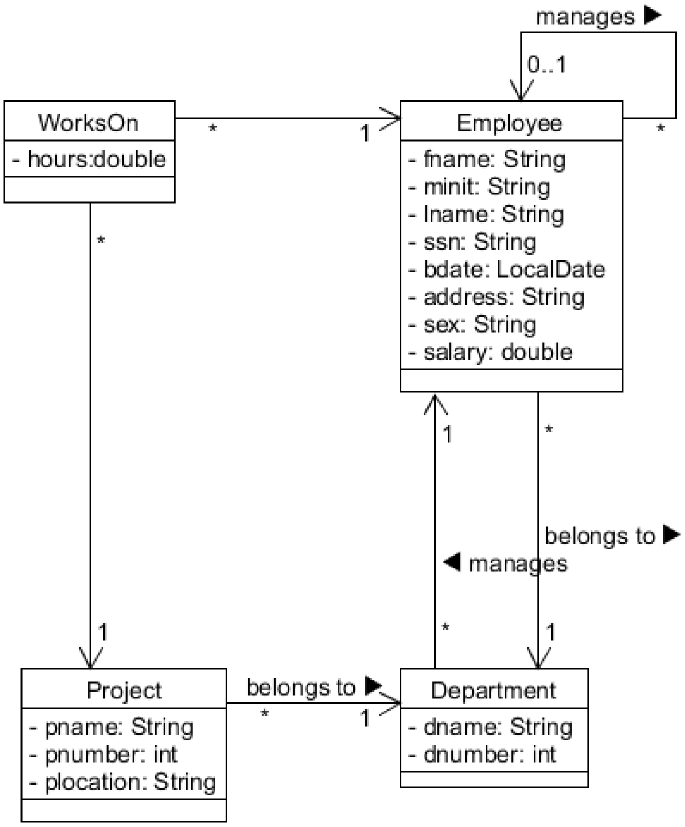


Figure 1: Class diagram Company model layer.

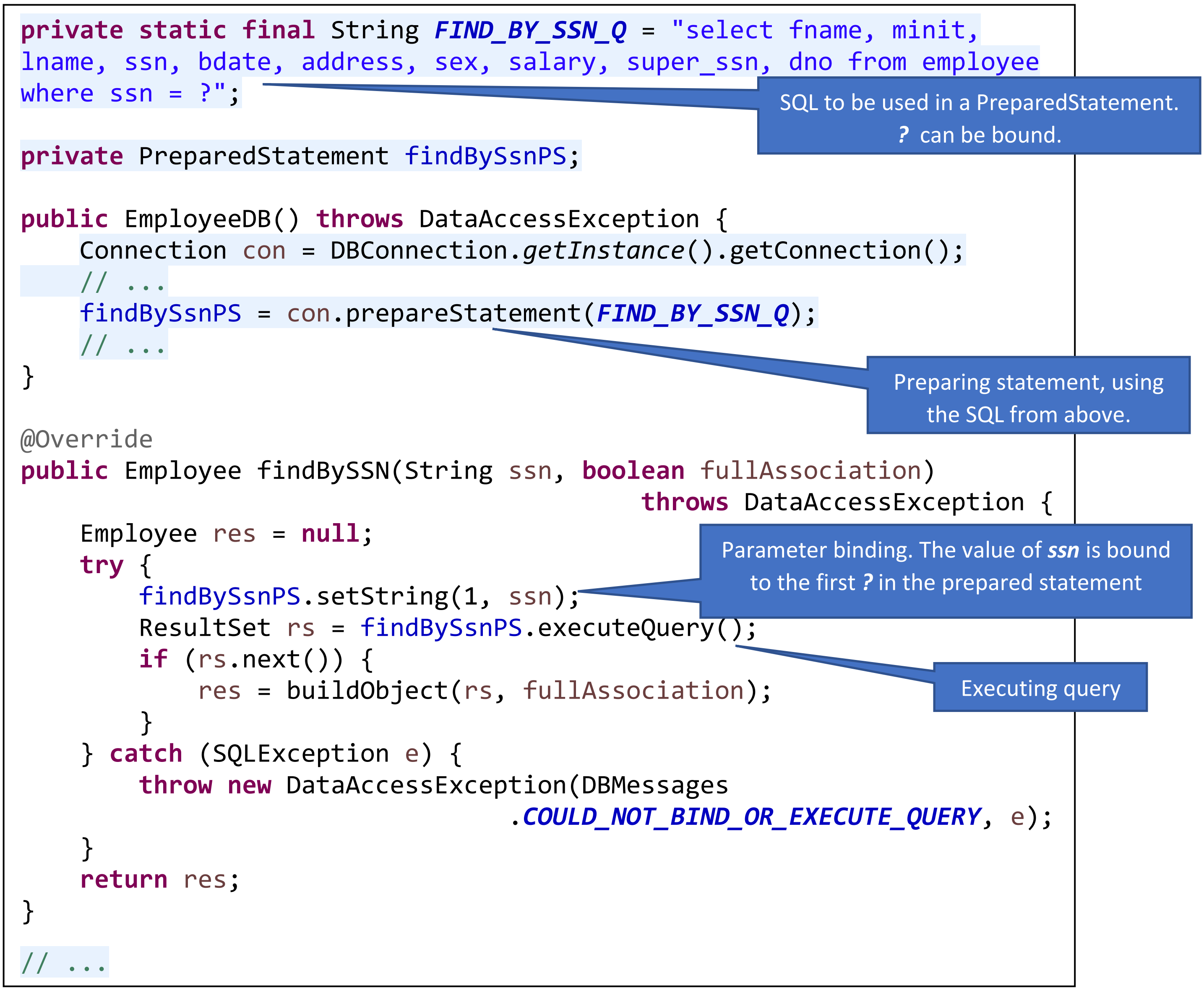
Implementation of the Database Transformation Layer

The database classes (EmployeeDB, WorksOnDB...) each implement its own interface. SQL and JDBC were used to implement the functionality. The classes also have private methods that are used to build objects.

The embedded SQL-statements are stored as String objects. The String representation of the queries and update statements are converted into executable *PreparedStatements* in the database class constructor. If parameters must be passed to the statement (e.g. values to insert, search parameters to findBy… methods), the ***parameter binding*** happens *each time* before the PreparedStatements are used. Execution is triggered by calling the *executeQuery()* or *executeUpdate()* method on the PreparedStatement.

Two private methods are used for the search in the database those are:

SQL-statements are best developed and tested in SSMS, then copied to the corresponding database class and stored as a constant (*static final*) String variable. Variables are replaced by question marks (?). Before use, the question marks are assigned values through the above discussed parameter binding. You specify the value for each question mark by numbering them. ***Numbering starts from 1*** (as opposed to indexing of arrays or lists, which starts from 0). An example is shown below.



```java
private static final String FIND_BY_SSN_Q = "select fname, minit, 
lname, ssn, bdate, address, sex, salary, super_ssn, dno from employee 
where ssn = ?";

private PreparedStatement findBySsnPS;

public EmployeeDB() throws DataAccessException {
    Connection con = DBConnection.getInstance().getConnection();
    // ...
    findBySsnPS = con.prepareStatement(FIND_BY_SSN_Q);
    // ...
}

@Override
public Employee findBySSN(String ssn, boolean fullAssociation)
                                        throws DataAccessException {
    Employee res = null;
    try {
        findBySsnPS.setString(1, ssn);
        ResultSet rs = findBySsnPS.executeQuery();
        if (rs.next()) {
            res = buildObject(rs, fullAssociation);
        }
    } catch (SQLException e) {
        throw new DataAccessException(DBMessages
                              .COULD_NOT_BIND_OR_EXECUTE_QUERY, e);
    }
    return res;
}

// ...
```

SQL to be used in a PreparedStatement. *?* can be bound.

Preparing statement, using the SQL from above.

Parameter binding. The value of ***ssn*** is bound to the first ***?*** in the prepared statement

Executing query

When converting a ResultSet to objects, you can imagine a table, where rows represent an entity each (which would correspond to objects in the application), and where columns correspond to attributes (fields in the application). You can access the value of a column by its name (in the database), and you can move on to the next row in the ResultSet object by calling the ***next()*** method. The *next()* method returns ***true*** if there is a row to move on to, *and move its internal cursor* to the next row. It returns *false* if there are no more rows. This also means that *before you access any attributes*, you must have called *next()* on the ResultSet.

It is good practice to explicitly name all the columns you want to retrieve instead of using a *.

An example of the conversion between a ResultSet and objects is shown below.

```java
private List<Employee> buildObjects(ResultSet rs, boolean fullAssociation)
                                                throws DataAccessException {
    List<Employee> res = new ArrayList<>();
    try {
        while (rs.next()) {
            Employee currEmployee = buildObject(rs, fullAssociation);
            res.add(currEmployee);
        }
    } catch (SQLException e) {
        throw new DataAccessException(
                                DBMessages.COULD_NOT_READ_RESULTSET, e);
    }
    return res;
}
```

Keep moving the RecordSet cursor while more unread rows

The Case as Inspiration

The implemented example can be used as inspiration for the second semester project architecture. Note that using the superior ***auto generated primary keys*** (*IDENTITY(1,1)* in MS-SQL) are not discussed in the company example, as the book does use natural primary keys. Study the ***int executeInsertWithIdentity(PreparedStatement ps)*** method in the DBConnection class, which with conjunction `PreparedStatement.RETURN_GENERATED_KEYS` will insert an entity that corresponds to a model layer object *and* return the auto-generated primary key from the database, which may be important, if it is to be used as a foreign key in another insert or update.

```java
private Employee buildObject(ResultSet rs, boolean fullAssociation)
                                                    throws DataAccessException {
    Employee currEmployee = new Employee();
    try {
        currEmployee.setFname(rs.getString("fname"));
        currEmployee.setMinit(rs.getString("minit"));
        currEmployee.setLname(rs.getString("lname"));
        currEmployee.setSsn(rs.getString("ssn"));
        currEmployee.setBdate(rs.getDate("bdate").toLocalDate());
        currEmployee.setAddress(rs.getString("address"));
        currEmployee.setSex(rs.getString("sex"));
        currEmployee.setSalary(rs.getDouble("salary"));
        currEmployee.setSupervisor(new Employee(rs.getString("super_ssn")));
        currEmployee.setDepartment(new Department(rs.getInt("dno")));
            if (fullAssociation) {
                Employee supervisor =
                    findBySSN(currEmployee.getSupervisor().getSsn(), false);
                currEmployee.setSupervisor(supervisor);
                Department department = this.departmentDB.findByDnumber
                    (currEmployee.getDept().getDnumber(), false);
                currEmployee.setDepartment(department);
            }
        } catch (SQLException e) {
            throw new DataAccessException(
                                            DBMessages.COULD_NOT_READ_RESULTSET, e);
        }
        return currEmployee;
}
```

Accessing columns by name (as specified in the database)

Wrapping foreign key in a “dummy” object

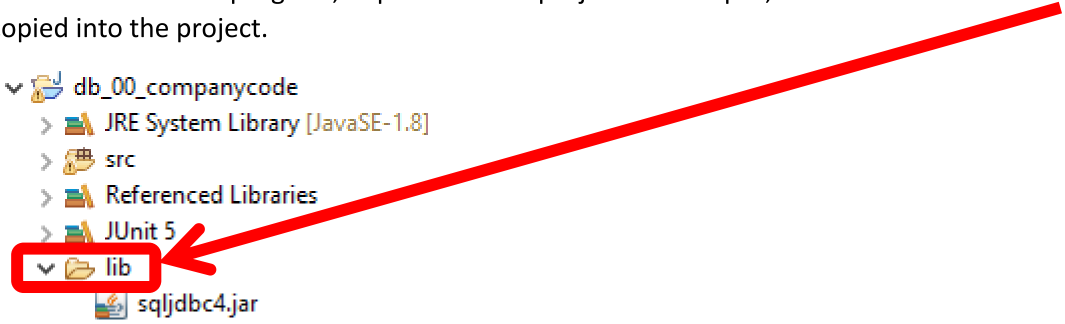
Collaboration with another database class to retrieve full association.

Executing the Company Example

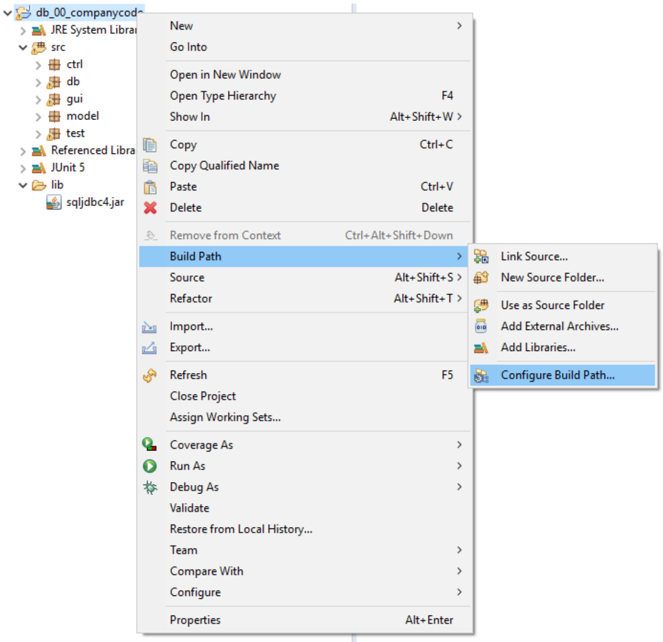
In order to run the described programs, MS-SQL Express must be installed on your computer, set to run in mix-mode authorization, and the *company* database must be created.

Set up MS SQL Express by following the accompanying document “Setting up MSSQL”.

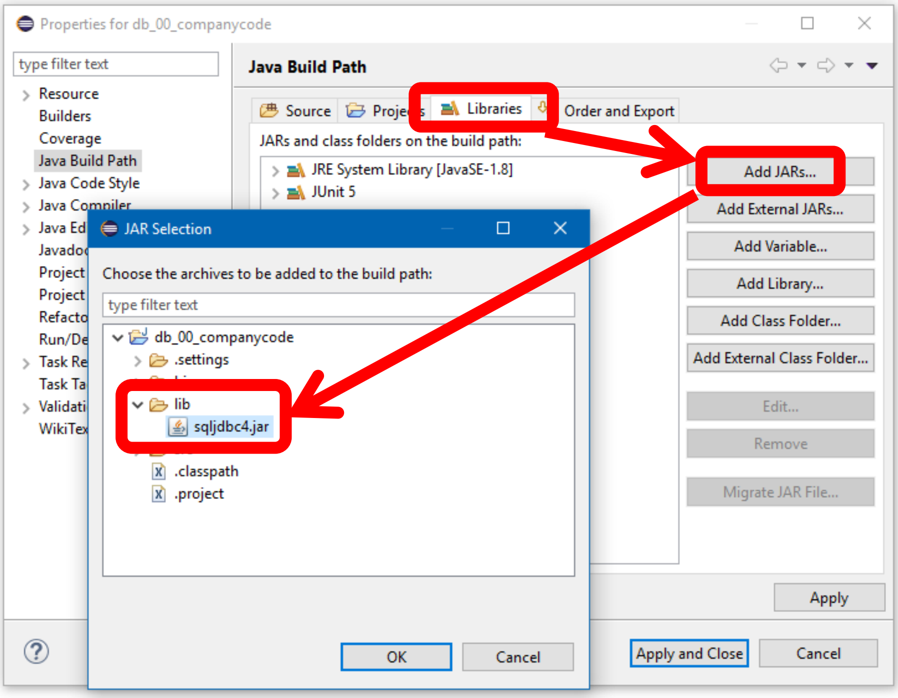
To execute the Java program, import the Java project into Eclipse, and make sure that the ***lib*** folder is copied into the project.



Add the JDBC driver to the Eclipse Build Path (right-click the project → Build Path → Configure Build Path…)



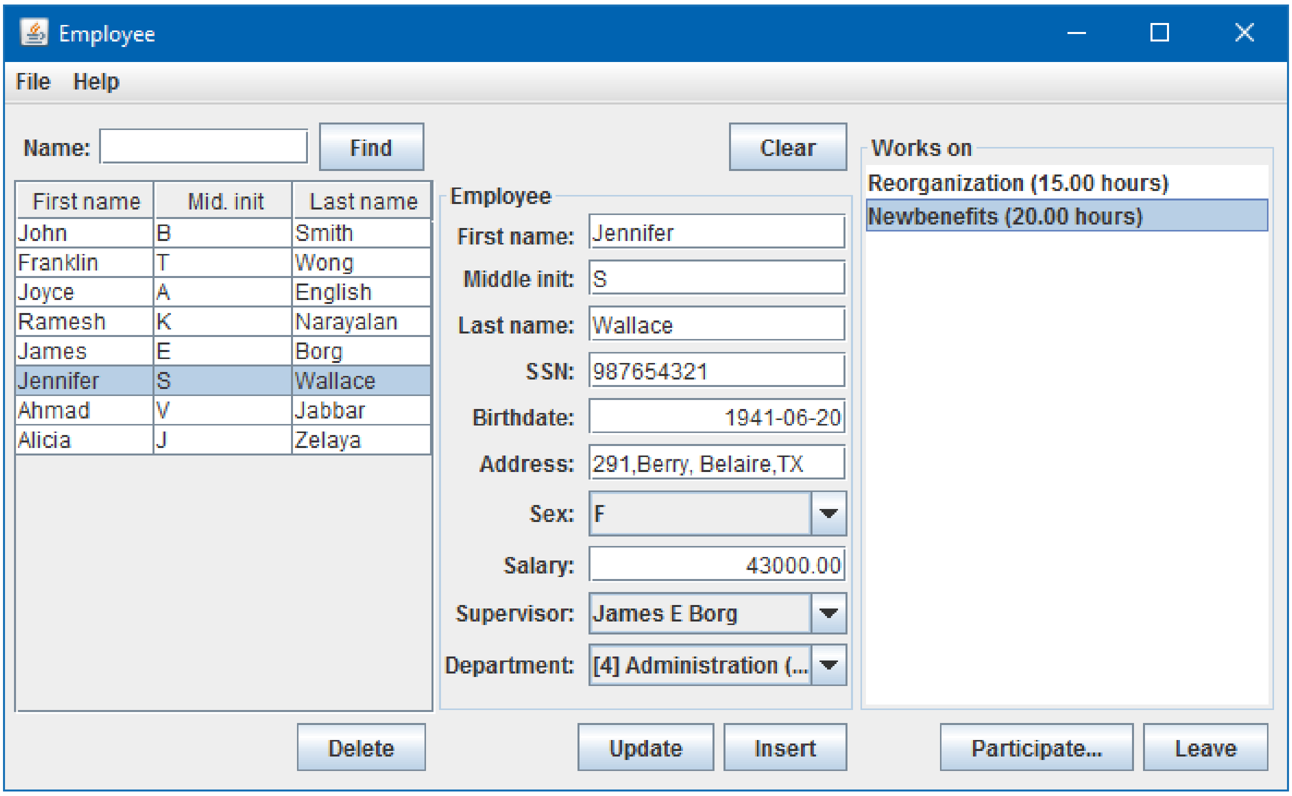
Complete the addition of the JDBC driver to the Eclipse Build Path (Pick the Libraries tab → Add JARs… → Open the lib folder, pick the JDBC-driver → OK → Apply and Close



JARs and class folders on the build path:

Choose the archives to be added to the build path:

To run the program, right-click the **gui.Gui** class and pick **Run As → Java Application**.



First name	Mid. init	Last name
John	B	Smith
Franklin	T	Wong
Joyce	A	English
Ramesh	K	Narayalan
James	E	Borg
Jennifer	S	Wallace
Ahmad	V	Jabbar
Alicia	J	Zelaya

Employee

First name: Jennifer

Middle init: S

Last name: Wallace

SSN: 987654321

Birthdate: 1941-06-20

Address: 291,Berry, Belaire,TX

Sex: F

Salary: 43000.00

Supervisor: James E Borg

Department: [4] Administration (...

Works on

Reorganization (15.00 hours)

Newbenefits (20.00 hours)