Java Easy Persistent Layer (JEPLayer) Reference Manual v1.1

Doc. version 1.0

Oct. 24, 2012

© 2011,2012 Jose María Arranz Santamaría

TABLE OF CONTENTS

1. INTRODUCTION

1.1 WHY ANOTHER JDBC BASED TOOL

JEPLayer was born to provide

1) A simple API to avoid the tedious tasks of JDBC
2) Several optional listeners to fully customize the lifecycle of JDBC persistence
3) Methods to build simple and complex DAOs
4) An extremely simple, automatic, configurable and error-free way to demarcate transactions
5) Does not replace JDBC, instead of this, JDBC objects are exposed when required
6) Ever using `PreparedStatement`, ever secure
7) `PreparedStatement` objects are automatically cached and reused
8) Fluid query API very similar to JPA 2 Query
9) Pure JDBC and JTA based transaction support built-in similar to method based demarcation in Spring or JavaEE
10) JTA simulation ("fake JTA") to code the same in JTA and non-JTA environments

1.2 WHAT IS DIFFERENT IN JEPLAYER

JEPLayer is simpler than Spring's JdbcTemplate[1] and has similar power, the persistent lifecycle can be fully configurable providing more interception points, it does not try to replace JDBC and JDBC and JTA transactions utilities are built-in, transaction management and demarcation are much simpler than Spring's Transactions[2].

JEPLayer is programmatic instead of the declarative path of iBatis/MyBatis[3] and the learning curve is flatter because JDBC API is not replaced.

JEPLayer provides transaction demarcation based on methods in a similar way as Spring or JavaEE, but no intrusive IoC container is required and there is no loss of control of the lifecycle of user defined classes.

JEPLayer allows getting the most of the database with no performance penalty and no waste of control typical of transparent persistence ORMs like Hibernate or JPA.

[1] http://static.springsource.org/spring/docs/2.5.x/reference/jdbc.html

[2] http://static.springsource.org/spring/docs/2.0.8/reference/transaction.html

[3] http://www.mybatis.org/

2. CONSIDERATIONS

2.1 DOCUMENT SCOPE

This manual makes an extensive documentation of JEPLayer features but must be complemented with the API documentation in javadoc format.

2.2 DOCUMENT CONVENTIONS

A Verdana font is used to describe JEPLayer.

```
A Courier New font is used for Java source code.
```

2.3 LICENSE

JEPLayer is open source and Apache License Version 2 licensed[4].

2.4 COPYRIGHTS

Jose María Arranz Santamaría is the author and intellectual property owner of JEPLayer source code, documentation and examples.

2.5 REQUIRED DEVELOPER TECHNICAL SKILLS

Java 1.5 and JDBC knowledge are required.

[4] http://www.apache.org/licenses/LICENSE-2.0.html

3. REQUIREMENTS AND INSTALLATION

3.1 TECHNICAL REQUIREMENTS, LIMITATIONS AND DEPENDENCIES

JEPLayer is compiled with Oracle's Java Standard Edition (JavaSE) SDK 1.6 configured with 1.5 source and binary modes.

3.2 JEPLAYER DISTRIBUTION

JEPLayer is distributed on two forms:

3.2.1 Binaries version

`JEPLayer_bin_X.zip` contains the required binaries to include in Java applications using JEPLayer, where X is the version.

Decompress this distribution ZIP file. The content of this file is two folders:

doc/ => Contains this manual and javadoc archives

bin/ => Contains the `jeplayer.jar` containing the compiled Java classes.

3.2.2 Development version

`JEPLayer_src_X.zip` is a NetBeans project (created with NetBeans v6.9) used to build, test and deploy the project. This project needs a MySQL database server (tested MySQL 5) installed in the same computer to run the tests.

The code of examples and tests is not included on "binaries" distribution and is also Apache v2.0 licensed.

The code of this distribution is used in this manual to show with examples how JEPLayer works.

3.3 JEPLAYER SETUP

All of you need is to add `jeplayer.jar` to the classpath of your project and all is done, JEPLayer is fully configured programmatically, you just need to setup a database and obtain in Java the corresponding `javax.sql.DataSource`.

In Maven you can add to your `pom.xml` something like (names can change):

```
<dependency>
   <groupId>jeplayer</groupId>
   <artifactId>jeplayer-jar</artifactId>
   <version>1.0</version>
   <scope>system</scope>
   <systemPath>JEPLayer dist folder/bin/jeplayer.jar</systemPath>
</dependency>
```

Or install in your local repository (names can change):

```
mvn install:install-file -Dfile=<path-jeplayer.jar> -DgroupId=jeplayer \
    -DartifactId=jeplayer-jar -Dversion=1.0 -Dpackaging=jar
```

Use `${basedir}` to avoid absolute paths.

4. JEPLAYER ARCHITECTURE

4.1 PURE INTERFACE/IMPLEMENTATION PATTERN

JEPLayer is almost fully based on interfaces, only very few classes are public, the root class `JEPLBootRoot` is the factory for all utility objects of the framework, exposed to developers through interfaces (excluding `JEPLException`).

Interface based architecture avoids class inheritance as a way of extension of the framework for end users. Instead of class inheritance, JEPLayer provides many hooks (listeners) as the standard way to extend and use the framework. Class inheritance is absolutely fine when you have complete control of the source code of the tree including base classes and is extensively used inside JEPLayer, but is not a good idea in a public API to final users.

4.2 LAYERED ARCHITECTURE

JEPLayer is constructed on top of the JDBC interfaces using a layer & composition approach. For instance a `JEPLDataSource` object wraps and contains an attribute containing the standard `javax.sql.DataSource` object. You can ever obtain the wrapped `DataSource` calling `JEPLDataSource.getDataSource()`. The same pattern is applied to any other JDBC interface, that is most of JDBC interfaces have a corresponding JEPLayer wrapping interface with the pattern `JEPL`*`NameOfJDBCInterface`* and the wrapped object can be obtained calling `JEPL`*`NameOfJDBCInterface`*`.get`*`NameOfJDBCInterface()`* when required; for instance `Connection` is wrapped by `JEPLConnection` which contains a method `JEPLConnection.getConnection()` and so on.

All of JEPLayer interfaces and classes start with `JEPL` prefix.

4.3 DATA ACCESS LAYERS (DAL) AND DATA ACCESS OBJECTS (DAO)

JEPLayer has two levels of database access, Data Access Layer (DAL) and Data Access Object (DAO).

4.3.1 Data Access Layer (DAL)

In this lower level, represented by `JEPLDAL` interface, you can access your database in a generic way, that is, DAL APIs are not designed to provide utilities to create and fill POJOs of your object data model.

You can see DAL API as a higher level JDBC API.

4.3.2 Data Access Object (DAO)

This level, represented by `JEPLDAO` interface and extending `JEPLDAL`, can be used to create and fill POJOs of an object data model, that is, DAO APIs are the Object Relational Mapping (ORM) level of JEPLayer.

You can see JEPLayer DAO API as a lower level API of your favorite ORM providing transparent persistence (Hibernate, JPA, JDO…).

4.4 PACKAGES

All public classes and interfaces are inside the package: `jepl`

Implementation of non-public classes are located in: `jepl.impl`

4.5 PERFORMANCE AND SECURITY

4.5.1 Performance

JEPLayer is a thin layer on top of JDBC and the performance penalty due to the library is minimum or none compared with custom made equivalent JDBC code. Furthermore, JEPLayer ever uses `PreparedStatement` these objects are automatically cached and reused when the SQL sentence being used is the same, this caching can save around 15% of processing time in tests performed against MySQL when the same SQL statements are executed frequently.

JEPLayer is designed to be used in a multithread environment with almost no synchronized code. In fact synchronized code is only defined in `JEPLUserData` methods of ready to be singleton objects (`JEPLBoot`, `JEPLDataSource`, `JEPLDAL` and `JEPLDAO` objects), if you are not using these methods in these objects, there are no synchronized methods. The only significative shared registries managed by JEPLayer are some `ThreadLocal` objects for instance containing the current `Connection` associated to the thread. Of course JEPLayer uses the connection pooling provided by your `DataSource` (JEPLayer supposes your `DataSource` is a connection pool), the performance of this pool depends on the pool framework or database driver used, JEPLayer does not provide a `DataSource` pool.

4.5.2 Security

Regarding to security, JEPLayer ever uses `PreparedStatement` objects to avoid malicious SQL injection through parameters provided by end users[5].

[5] http://www.owasp.org/index.php/Preventing_SQL_Injection_in_Java

5. DEVELOPMENT WITH JEPLAYER

5.1 CREATING A DATABASE MODEL

In the following examples we are going to use a simple (well not so simple) database example to show JEPLayer capabilities, first of all we are going to create our DB schema programmatically with JEPLayer, this code is the first example of DAL-like access to the DB.

Previously we have set up a MySQL 5 database and added the JDBC driver to the classpath, JEPLayer is database agnostic but in these examples MySQL and MySQL JDBC driver is supposed. We have previously developed a custom class, `DataSourceFactory`, just to get a `javax.sql.DataSource` of our MySQL database, details of this factory class are not important, in your application use the kind of configuration technique you like more to get the `DataSource`.

```
public class CreateDBModel
{
    public static void main(String[] args)
    {
        DataSourceFactory dsFactory =
                        DataSourceFactoryImpl.createDataSourceFactoryJDBC();
        DataSource ds = dsFactory.getDataSource();
        JEPLBootNonJTA boot = JEPLBootRoot.get().createJEPLBootNonJTA();
        JEPLNonJTADataSource jds = boot.createJEPLNonJTADataSource(ds);

        try
        {
            JEPLDAL dal = jds.createJEPLDAL();

            dal.createJEPLDALQuery("DROP TABLE IF EXISTS PERSON")
                        .executeUpdate();
            dal.createJEPLDALQuery("DROP TABLE IF EXISTS COMPANY")
                        .executeUpdate();
            dal.createJEPLDALQuery("DROP TABLE IF EXISTS CONTACT")
                        .executeUpdate();

            dal.createJEPLDALQuery(
                "CREATE TABLE  CONTACT (" +
                "  ID INT NOT NULL AUTO_INCREMENT," +
                "  EMAIL VARCHAR(255) NOT NULL," +
                "  NAME VARCHAR(255) NOT NULL," +
                "  PHONE VARCHAR(255) NOT NULL," +
                "  PRIMARY KEY (ID)" +
                ")" +
                "ENGINE=InnoDB"
                ).executeUpdate();

            dal.createJEPLDALQuery(
                "CREATE TABLE  PERSON (" +
                "  ID INT NOT NULL," +
                "  AGE SMALLINT," +
                "  CONSTRAINT PERSON_FK_ID FOREIGN KEY (ID) REFERENCES
CONTACT(ID) " +
                "     ON DELETE CASCADE" +
                ")" +
                "ENGINE=InnoDB"
                ).executeUpdate();
```

```
            dal.createJEPLDALQuery(
                "CREATE TABLE  COMPANY (" +
                "  ID INT NOT NULL," +
                "  ADDRESS VARCHAR(255) NOT NULL," +
                "  CONSTRAINT COMPANY_FK_ID FOREIGN KEY (ID) REFERENCES
CONTACT(ID) " +
                "     ON DELETE CASCADE" +
                ")" +
                "ENGINE=InnoDB"
                ).executeUpdate();
        }
        finally
        {
            dsFactory.destroy();
        }
    }
}
```

This code creates three tables, CONTACT, PERSON, COMPANY, the relationship is 1-1 of CONTACT with PERSON or COMPANY, in fact, later we are going to model in Java this relationship as inheritance of classes `Person` and `Company` from `Contact`. Note the `ON DELETE CASCADE`, this constraint will be important (in spite of we may not rely on this constraint and work with un-constrained tables).

Let's see in more detail…

```
JEPLBootNonJTA boot = JEPLBootRoot.get().createJEPLBootNonJTA();
```

Creates the root factory object of non-JTA utilities in JEPLayer, this object can be a singleton and is the source of `JEPLDataSource` objects wrapping `DataSource` objects.

The method `JEPLBootNonJTA.createJEPLNonJTADataSource(DataSource)` creates a `JEPLNonJTADataSource` object, this interface inherits from `JEPLDataSource` and contains a `DataSource` to be used in a pure JDBC transactional environment, that is, auto-commit modes and JDBC transaction methods `commit()` and `rollback()` of `java.sql.Connection` are explicitly used to control transaction behavior instead of the Java Transaction API (JTA) approach.

In spite of JEPLayer provides non-JTA and JTA APIs, most of methods are shared between both approaches, furthermore, JEPLayer provides a "fake" JTA `UserTransaction` using underlying pure JDBC (non-JTA).

The same as `DataSource`, a `JEPLNonJTADataSource` object can also be a singleton.

Through `JEPLNonJTADataSource`, `JEPLDAL` objects are created calling `JEPLNonJTADataSource.createJEPLDAL()`. As you can see a `JEPLDAL` object is bound indirectly to a `DataSource` and any SQL action is performed through the internal `DataSource`.

Take a look to the sentence:

```
dal.createJEPLDALQuery("DROP TABLE IF EXISTS PERSON").executeUpdate();
```

The lifecycle is getting a `javax.sql.Connection` from the `DataSource` pool, executing the specified SQL sentence calling `PreparedStatement.executeUpdate(String)` and closing the connection (returning to the pool). In this simple example, the complete lifecycle is executed on every call, we will see later how to reuse the same `Connection` for several statements before returning to the pool and optionally group them into a transaction.

The method `JEPLDAL.createJEPLDALQuery()` creates a `JEPLDALQuery` object (a DAL level query), this object provides a "fluid API" very similar to `Query` of JPA, one method is

`executeUpdate()` which execute the SQL sentence calling `PreparedStatement.executeUpdate(String)`.

If a `JEPLDAL` object can be a singleton, why are not `JEPLDAL` methods defined in `JEPLDataSource`?

Because typical use of `JEPLDAL` is to be used alongside the DAO level API, `JEPLDAO`, a `JEPLDAO` object is also a `JEPLDAL` (because `JEPLDAO` inherits from `JEPLDAL`), and usually you create a `JEPLDAO` instance (and a class implementing this interface) per table and POJO class, the typical pattern of DAO programming. Further we will see how we will need to configure every JEPLDAO instance to perform queries to the same table/POJO.

Note that `JEPLDataSource` and `JEPLDAL` inherit both from `JEPLUserData`, this interface represents a simple `Map` for auxiliary data, most of JEPLayer objects implement this interface. If an object can be a singleton, `JEPLUserData` methods are synchronized (thread safe).

5.2 CREATING A DATA ACCESS OBJECT (DAO)

The class `Contact` is the obvious POJO class representing the table `CONTACT`:

```
public class Contact
{
    protected int id;
    protected String name;
    protected String phone;
    protected String email;

    public Contact(int id, String name, String phone, String email)
    {
        this.id = id;
        this.name = name;
        this.phone = phone;
        this.email = email;
    }

    public Contact()
    {
    }
  /* ... gets and sets ... */
}
```

The `ContactDAO` based on JEPLayer would be something like (imports are omitted):

```
public class ContactDAO implements JEPLResultSetDAOListener<Contact>
{
    protected JEPLDAO<Contact> dao;

    public ContactDAO(JEPLDataSource ds)
    {
        this.dao = ds.createJEPLDAO(Contact.class);
        dao.addJEPLListener(this);
    }

    public JEPLDAO<Contact> getJEPLDAO()
    {
        return dao;
    }
```

```
@Override
public void setupJEPLResultSet(JEPLResultSet jrs,JEPLTask<?> task)
    throws Exception
{
}

@Override
public Contact createObject(JEPLResultSet jrs) throws Exception
{
    return new Contact();
}

@Override
public void fillObject(Contact obj,JEPLResultSet jrs) throws Exception
{
    ResultSet rs = jrs.getResultSet();

    obj.setId(rs.getInt("ID"));
    obj.setName(rs.getString("NAME"));
    obj.setPhone(rs.getString("PHONE"));
    obj.setEmail(rs.getString("EMAIL"));
}

public void insert(Contact contact)
{
    int key = dao.createJEPLDALQuery(
            "INSERT INTO CONTACT (EMAIL, NAME, PHONE) VALUES (?, ?, ?)")
             .addParameters(contact.getEmail(),
                    contact.getName(),contact.getPhone())
             .getGeneratedKey(int.class);
    contact.setId(key);
}

public void update(Contact contact)
{
    dao.createJEPLDALQuery(
            "UPDATE CONTACT SET EMAIL = ?, NAME = ?, PHONE = ? WHERE ID = ?")
             .addParameters(contact.getEmail(),contact.getName(),
                    contact.getPhone(),contact.getId())
             .setStrictMinRows(1).setStrictMaxRows(1)
             .executeUpdate();
}

public boolean delete(Contact obj)
{
    return deleteById(obj.getId());
}

public boolean deleteById(int id)
{
    // Only if there is no "inherited" rows or declared ON DELETE CASCADE
    return dao.createJEPLDALQuery("DELETE FROM CONTACT WHERE ID = ?")
                .setStrictMinRows(0).setStrictMaxRows(1)
                .addParameter(id)
                .executeUpdate() > 0;
}

public int deleteAll()
{
    // Only if "inherited" tables are empty or declared ON DELETE CASCADE
    return dao.createJEPLDALQuery("DELETE FROM CONTACT").executeUpdate();
}
```

```
    public List<Contact> selectAll()
    {
        return dao.createJEPLDAOQuery("SELECT * FROM CONTACT").getResultList();
    }

    public JEPLResultSetDAO<Contact> selectAllResultSetDAO()
    {
        return dao.createJEPLDAOQuery(
                        "SELECT * FROM CONTACT").getJEPLResultSetDAO();
    }

    public List<Contact> selectJEPLDAOQueryRange(int from,int to)
    {
        return dao.createJEPLDAOQuery("SELECT * FROM CONTACT")
                .setFirstResult(from)
                .setMaxResults(to - from)
                .getResultList();
    }

    public Contact selectById(int id)
    {
        return dao.createJEPLDAOQuery("SELECT * FROM CONTACT WHERE ID = ?")
                .addParameter(id)
                .getSingleResult();
    }

    public List<Contact> selectByNameAndEMail(String name,String email)
    {
        return dao.createJEPLDAOQuery(
               "SELECT * FROM CONTACT WHERE NAME = ? AND EMAIL = ?")
                .addParameters(name,email)
                .getResultList();
    }

    public int selectCount()
    {
        return dao.createJEPLDALQuery("SELECT COUNT(*) FROM CONTACT")
                    .getOneRowFromSingleField(int.class);
    }
}
```

The first we need to know is how we create a `JEPLDAO<Contact>` object calling `JEPLDataSource.createJEPLDAO(Class<T>)`:

```
ds.createJEPLDAO(Contact.class);
```

`JEPLDAO` objects will be used to perform persistent operations over user defined persistent objects/classes. As you can see we are usually going to need a `JEPLDAO` object per persistent class (usually associated to a single database table). Because `JEPLDAO` inherits from `JEPLDAL`, all methods of "DAL level" are also available.

5.2.1 Insertion with automatic generation of primary key

When the table `CONTACT` was defined the key column (ID) was declared as `AUTO_INCREMENT` (this syntax is only valid in MySQL, other databases have a similar syntax do not be fooled by this minor SQL incompatibility). The method `JEPLDALQuery.getGeneratedKey(Class<U>)` is designed to execute `INSERT` SQL sentences in tables with automatic key generation, in this case the generated key is the ID attribute:

```
    public void insert(Contact contact)
    {
```

```
        int key = dao.createJEPLDALQuery(
                "INSERT INTO CONTACT (EMAIL, NAME, PHONE) VALUES (?, ?, ?)")
                .addParameters(contact.getEmail(),
                        contact.getName(),contact.getPhone())
                .getGeneratedKey(int.class);
        contact.setId(key);
    }
```

The parameter `int.class` is used just to cast the returned JDBC value (the generated key) to the required type, internally to perform this conversion the following `JEPLDAL` method is called:

```
public <U> U cast(Object obj,Class<U> returnType);
```

As we can see, parameters are provided using the standard JDBC ? notation of `PreparedStatement` and calling the "fluid" method `addParameters(Object…)`, parameters are sequentially associated to each ? in SQL sentence.

5.2.2 Options to declare and provide parameters in SQL sentences

In the previous example we have seen the standard JDBC ? notation to declare parameters and the method `addParameters(Object…)`, JEPLayer provides more options:

- JDBC standard `?` notation
- `?number` notation (ex. `?1 ?2 …`) similar to JPA
- `:name` notation (ex. `:email`) similar to JPA
- Mixed previous options

If you use the `?number` notation, you can use the fluid method:

```
JEPLDALQuery.setParameter(int position,Object value)
```

to set a parameter in the specified position (starting in 1).

The previous query could be rewritten as:

```
    int key = dao.createJEPLDALQuery(
                "INSERT INTO CONTACT (EMAIL, NAME, PHONE) VALUES (?1, ?2, ?3)")
            .setParameter(1,contact.getEmail())
            .setParameter(2,contact.getName())
            .setParameter(3,contact.getPhone())
            .getGeneratedKey(int.class);
```

If you use the `:name` notation, you can use the fluid method:

```
JEPLDALQuery.setParameter(String name,Object value)
```

to bind a parameter to the specified name, the same :name declaration can be repeated into the SQL sentence.

The previous query could be rewritten as:

```
    int key = dao.createJEPLDALQuery(
        "INSERT INTO CONTACT (EMAIL, NAME, PHONE) VALUES (:email,:name,:phone)")
            .setParameter("email",contact.getEmail())
            .setParameter("name",contact.getName())
            .setParameter("phone",contact.getPhone())
            .getGeneratedKey(int.class);
```

Finally you can mix several notations in the same sentence:

```
    int key = dao.createJEPLDALQuery(
        "INSERT INTO CONTACT (EMAIL, NAME, PHONE) VALUES (?, ?2, :phone)")
            .setParameter(1,contact.getEmail())
            .setParameter(2,contact.getName())
            .setParameter("phone",contact.getPhone())
            .getGeneratedKey(int.class);
```

Previous code is ugly and confusing and is not recommended, is just useful to explain how parameter declaration works internally, every parameter either `?` or `?num` or `:name` has a position starting in 1, this is the reason why in previous example `?` is position 1, `?2` is position 2 (any different integer is wrong) and `:phone` is internally 3.

Now you can understand why

```
.addParameters(contact.getEmail(),
       contact.getName(),contact.getPhone())
```

is the same as:

```
.setParameter(1,contact.getEmail())
.setParameter(2,contact.getName())
.setParameter(3,contact.getPhone())
```

Later we will see SQL sentences returning user data objects executed by `JEPLDAOQuery` objects, because this interface inherits from `JEPLDALQuery` fluid methods are repeated in `JEPLDAOQuery`, in this case all of them return `JEPLDAOQuery` instead of `JEPLDALQuery` but behavior is the same.

5.2.3 Specifying listeners to control the persistent lifecycle

Previously we have seen this code:

```
public class ContactDAO implements JEPLResultSetDAOListener<Contact>
{
    protected JEPLDAO<Contact> dao;

    public ContactDAO(JEPLDataSource ds)
    {
        this.dao = ds.createJEPLDAO(Contact.class);
        dao.addJEPLListener(this);
    }

...
```

The method `JEPLDAL.addJEPLListener(JEPLListener listener)` is called to register "this", this is possible because `ContactDAO` is a `JEPLListener` because it implements the interface `JEPLResultSetDAOListener` and this interface inherits from `JEPLListener`.

What is the `JEPLListener` parameter?

`JEPLListener` classes are designed to provide "persistent lifecycle listeners", if an object implementing one or several `JEPListener` based interfaces, is provided as parameter, this listener is called in the middle of the persistent lifecycle.

There are several `JEPLListener` based interfaces, all of them inherit from `JEPListener` (`JEPListener` does nothing by itself).

- **`JEPLConnectionListener<T>`**

  Its method:

  ```
  public void setupJEPLConnection(JEPLConnection con,JEPLTask<T> task)
                  throws Exception;
  ```

  is called before executing the SQL statement.

- **`JEPLPreparedStatementListener<T>`**

  Its method:

  ```
  public void setupJEPLPreparedStatement(JEPLPreparedStatement stmt,
                  JEPLTask<T> task) throws Exception;
  ```

  is called before executing the SQL statement.

- **`JEPLResultSetDALListener`**

  Its methods:

```
public void setupJEPLResultSet(JEPLResultSet jrs,JEPLTask<?> task)
                throws Exception;
public <U> U getValue(int columnIndex,Class<U> returnType,
                JEPLResultSet jrs) throws Exception;
```

  are called to process the result of the "DAL" SQL statement (executing a sentence with a `JEPLDALQuery`), when no data model object is returned.

- **`JEPLResultSetDAOListener<T>`**

  Its methods:

```
public void setupJEPLResultSet(JEPLResultSet jrs,JEPLTask<?> task)
                throws Exception;
public T createObject(JEPLResultSet jrs) throws Exception;
public void fillObject(T obj,JEPLResultSet jrs) throws Exception;
```

  are called to process the result of the "DAO" SQL statement (executing a sentence with a `JEPLDAOQuery`), when expected data model objects to be returned.

Interfaces `JEPLConnection`, `JEPLPreparedStatement`, `JEPLResultSet` are just JEPLayer wrappers of standard `Connection`, `PreparedStatement` and `ResultSet` objects. `JEPLTask` requires more attention (see later).

There are several points where we can inject a `JEPLListener` listener, because only a listener of the same type is picked per SQL sentence, there is a priority list:

1) The listener of the required type was registered calling `JEPLDALQuery.addJEPLListener(JEPLListener)`.
2) Registered into the DAO/DAL object calling `JEPLDAL.addJEPLListener(JEPLListener)`
3) Registered into the `JEPLDataSource` calling `JEPLDataSource.addJEPLListener(JEPLListener)`

Following with our example:

```
public void insert(Contact contact)
{
    int key = dao.createJEPLDALQuery(
           "INSERT INTO CONTACT (EMAIL, NAME, PHONE) VALUES (?, ?, ?)")
            .addParameters(contact.getEmail(),
                   contact.getName(),contact.getPhone())
            .getGeneratedKey(int.class);
    contact.setId(key);
}
```

This is a DAL level SQL sentence, no data model object is returned, in this example no appropriated `JEPLListener` seems to be registered and used, cast of the returned generated key is performed calling the `JEPLDAL` method:

```
public <U> U cast(Object obj,Class<U> returnType);
```

We can change the behavior of this sentence registering a `JEPLResultSetDALListener` listener, in this example on `JEPLDALQuery` level if we add a new method to our DAO like this:

```
public void insertExplicitResultSetListener(Contact contact)
{
    JEPLResultSetDALListener listener = new JEPLResultSetDALListener()
    {
        @Override
        public void setupJEPLResultSet(JEPLResultSet jrs,JEPLTask<?> task)
```

```
                                throws Exception
            {
            }
            @Override
            public <U> U getValue(int columnIndex, Class<U> returnType,
                                JEPLResultSet jrs) throws Exception
            {
                if (!returnType.equals(int.class))
                        throw new RuntimeException("UNEXPECTED");
                // Expected columnIndex = 1
                int resInt = jrs.getResultSet().getInt(columnIndex);
                Object resObj = jrs.getResultSet().getObject(columnIndex);
                Integer resIntObj = (Integer)jrs.getJEPLStatement()
                        .getJEPLDAL().cast(resObj, returnType);
                if (resInt != resIntObj)
                        throw new RuntimeException("UNEXPECTED");
                return (U)resIntObj;
            }
        };

        int key = dao.createJEPLDALQuery(
                "INSERT INTO CONTACT (EMAIL, NAME, PHONE) VALUES (?, ?, ?)")
                .addParameters(contact.getEmail(),contact.getName(),
                        contact.getPhone())
                .addJEPLListener(listener)
                .getGeneratedKey(int.class);
         contact.setId(key);
    }
```

This example plays with types and finally returns just the same as the default behaviour (calling `JEPLDAL.cast(…)` method), but it was useful to show how the specified `JEPLResultSetDALListener` is called in this context (expected a single row with a single column containing the generated key).

5.2.4 Executing non-SELECT SQL sentences

Non-SELECT sentences are typically executed calling `JEPLDALQuery.`**`executeUpdate`**`()`

```
        public void update(Contact contact)
        {
            dao.createJEPLDALQuery("UPDATE CONTACT SET EMAIL = ?,
    NAME = ?, PHONE = ? WHERE ID = ?")
                    .addParameters(contact.getEmail(),contact.getName(),
                            contact.getPhone(),contact.getId())
                    .setStrictMinRows(1).setStrictMaxRows(1)
                    .executeUpdate();
        }
```

In this example the method `executeUpdate()` executes the SQL sentence to update the row. Take a look to the optional `setStrictMinRows(1)` and `setStrictMaxRows(1)` calls, they are fluid methods to specify the required number of returned rows, minimum and maximum, in this example if no row has been updated (0) or more than one (2 or more) rows has been updated a `JEPLException` is thrown.

The following example shows how we can specify a range or affected rows:

```
    public boolean deleteById(int id)
    {
        // Only if there is no "inherited" rows or declared ON DELETE CASCADE
        return dao.createJEPLDALQuery("DELETE FROM CONTACT WHERE ID = ?")
                    .setStrictMinRows(0).setStrictMaxRows(1)
```

```
                    .addParameter(id)
                    .executeUpdate() > 0;
    }
```

This code tolerates the specified row does not exist (no deletion).

5.2.5 Select returning POJOs of your user data object model

Just with this simple call we get all rows of the `CONTACT` table and converted to `Contact` objects:

```
    public List<Contact> selectAll()
    {
        return dao.createJEPLDAOQuery("SELECT * FROM CONTACT").getResultList();
    }
```

Where is the magic? It was already shown before:

```
public class ContactDAO implements JEPLResultSetDAOListener<Contact>
{
    protected JEPLDAO<Contact> dao;

    public ContactDAO(JEPLDataSource ds)
    {
        this.dao = ds.createJEPLDAO(Contact.class);
        dao.addJEPLListener(this);
    }

    public JEPLDAO<Contact> getJEPLDAO()
    {
        return dao;
    }

    @Override
    public void setupJEPLResultSet(JEPLResultSet jrs,JEPLTask<?> task)
                throws Exception
    {
    }

    @Override
    public Contact createObject(JEPLResultSet jrs) throws Exception
    {
        return new Contact();
    }

    @Override
    public void fillObject(Contact obj,JEPLResultSet jrs) throws Exception
    {
        ResultSet rs = jrs.getResultSet();

        obj.setId(rs.getInt("ID"));
        obj.setName(rs.getString("NAME"));
        obj.setPhone(rs.getString("PHONE"));
        obj.setEmail(rs.getString("EMAIL"));
    }
```

...

Because `ContactDAO` object implements `JEPLResultSetDAOListener<T>` the method `getResultList()` use this implementation of this kind of listener according to the priority list explained before.

When a SELECT clause is executed the method `setupJEPLResultSet` is called immediately *after* the SQL sentence is executed, to setup the `ResultSet` before iterating. The framework iterates through the `ResultSet` calling `next()` and delegating all row reading to `createObject` and `fillObject`, called per row. If `createObject` returns `null` the method `fillObject` is not called, and no element (nor null) is added to the `List` result of the call to `getResultList()` (equivalent to skip/filter this row-Object).

Of course you can provide this listener in different ways, for instance as a parameter, in this case the parameter has priority, the following method can be added to our `ContactDAO` class:

```
public List<Contact> selectAllExplicitResultSetListener()
{
    JEPLResultSetDAOListener<Contact> listener =
           new JEPLResultSetDAOListener<Contact>()
    {
        @Override
        public void setupJEPLResultSet(JEPLResultSet jrs,JEPLTask<?> task)
            throws Exception
        {
        }

        @Override
        public Contact createObject(JEPLResultSet jrs) throws Exception
        {
            return new Contact();
        }

        @Override
        public void fillObject(Contact obj,JEPLResultSet jrs)
                           throws Exception
        {
            ResultSet rs = jrs.getResultSet();

            obj.setId(rs.getInt("ID"));
            obj.setName(rs.getString("NAME"));
            obj.setPhone(rs.getString("PHONE"));
            obj.setEmail(rs.getString("EMAIL"));
        }
    };
    return dao.createJEPLDAOQuery("SELECT * FROM CONTACT")
            .addJEPLListener(listener)
            .getResultList();
}
```

A registered `JEPLResultSetDAOListener` is required when you query the database using SELECT to get user defined POJOs calling methods like `JEPLDAOQuery.getResultList()`, `JEPLDAOQuery.getSingleResult()` or `JEPLDAOQuery.getJEPLResultSetDAO()`, if no one is provided a `JEPLException` is thrown.

5.2.6 Select returning POJOs of data model using an automatic bean mapper

Alternatively you can use a `JEPLResultSetDAOListenerDefault` instance instead of your custom `JEPLResultSetDAOListener`, JEPLayer provides this ready to use implementation calling `JEPLDataSource.createJEPLResultSetDAOListenerDefault(Class<T> clasz)`. It automatically maps every result row to an object of the specified class, every column is mapped to the corresponding property calling the corresponding set method by name following the Java Beans convention.

The following method does the same as the previous hand-made mapping:

```
public List<Contact> selectAllExplicitResultSetDAOListenerBean()
{
    JEPLResultSetDAOListenerDefault<Contact> listener =
            dao.getJEPLDataSource()
           .createJEPLResultSetDAOListenerDefault(Contact.class);

    return dao.createJEPLDAOQuery("SELECT * FROM CONTACT")
            .addJEPLListener(listener)
            .getResultList();
}
```

In current implementation of JEPLayer, a `JEPLResultSetDAOListenerDefault` instance is thread safe and can be used as a singleton the same as the associated `JEPLDataSource` can be[6].

You can optionally use a `JEPLRowBeanMapper` to modify the default behavior of a `JEPLResultSetDAOListenerDefault`. An object of this interface associated to a `JEPLResultSetDAOListenerDefault` intercepts all column-field bindings and when the method `JEPLRowBeanMapper.setColumnInBean(…)` returns true, JEPLayer understands this column-field binding has already been "manually" done by user code and no action is done by JEPLayer. The method `setColumnInBean(T obj,JEPLResultSet jrs,int col,String columnName,Object value,Method setter)` provides all necessary data to set the property (usually attribute) in the bean, the `setter` parameter is the proposed `java.reflect.Method` of the bean, it is null when there is no set method with the specified column name or is static.

For example:

```
public List<Contact> selectAllExplicitResultSetDAOListenerBeanWithMapper()
{
    JEPLRowBeanMapper<Contact> rowMapper = new JEPLRowBeanMapper<Contact>()
    {
        public boolean setColumnInBean(Contact obj,JEPLResultSet jrs,
                    int col, String columnName, Object value, Method setter)
        {
            if (columnName.equalsIgnoreCase("email"))
            {
                obj.setEmail((String)value);
                return true;
            }
            return false;
        }
    };
    JEPLResultSetDAOListenerDefault<Contact> listener =
            dao.getJEPLDataSource()
           .createJEPLResultSetDAOListenerDefault(Contact.class,rowMapper);

    return dao.createJEPLDAOQuery("SELECT * FROM CONTACT")
            .addJEPLListener(listener)
            .getResultList();
}
```

5.2.7 Select a range of returning POJOs with JEPLDAOQuery

[6] Current implementation in no way bounds `JEPLResultSetDAOListenerDefault` to `JEPLDataSource`, anyway to prevent future changes you should create almost one `JEPLResultSetDAOListenerDefault` per `JEPLDataSource` and avoid reusing the same `JEPLResultSetDAOListenerDefault` instance with several `JEPLDataSource`.

`JEPLDAOQuery` provides two configuration methods before executing a SELECT sentence:

- `setFirstResult(int)` : sets the `ResultSet` initially to the specified position calling `ResultSet.absolute(int)`.
- `setMaxResults(int)` : ends the `ResultSet` iteration when the number of specified results is achieved.

The following example shows how we can get a subset of queried rows avoiding getting all rows from database (`from` and `to` are row indexes and row `to` is not included):

```
public List<Contact> selectJEPLDAOQueryRange(int from,int to)
{
    return dao.createJEPLDAOQuery("SELECT * FROM CONTACT")
            .setFirstResult(from)
            .setMaxResults(to - from)
            .getResultList();
}
```

5.3 ADVANCED USES OF DAO AND LISTENERS

5.3.1 Select query with a max number of rows using a listener and the JEPLTask

We have explained that inherited interfaces of `JEPLListener` are used to modify the default lifecycle, the `setupJEPLX` methods are called *before* processing the JDBC object passed as parameter. What if we need some processing *after* using the JDBC object?

For instance, we want to query only the first results of a large number of rows/objects using pure JDBC, we could code something like:

```
public List<Contact> selectAllStatementListenerMaxRows(final int maxRows)
{
    JEPLPreparedStatementListener<List<Contact>> listener =
            new JEPLPreparedStatementListener<List<Contact>>()
    {
        public void setupJEPLPreparedStatement(JEPLPreparedStatement jstmt,
                JEPLTask<List<Contact>> task) throws Exception
        {
            PreparedStatement stmt = jstmt.getPreparedStatement();
            stmt.setMaxRows(maxRows);
        }
    };

    return dao.createJEPLDAOQuery("SELECT * FROM CONTACT")
            .addJEPLListener(listener)
            .getResultList();
}
```

This method works as expected and just returns the required max number of rows but it has a problem, JEPLayer caches `PreparedStatement` objects with the same SQL code[7], the next query with the same SQL statement, if the `PreparedStatement` is cached, keeps the same number of rows declared in a different DAO method call. Yes so far this is not actually a problem because every SQL sentence gets a `Connection` and recycles it returning to the pool in the end of the call, there is no option for caching, but later we will see how automatically several sentences like previously shown can reuse the same `Connection` and optionally

[7] Of course meanwhile the `Connection` is not returned to the pool, only `PreparedStatement` of the same Connection are recycled/reused

participate in a transaction. Hence we must solve this problem, we must restore the original old "max rows" value (usually "no limit") and setting the default value before any SQL statement is tedious, the obvious solution is to restore the old "max rows" value *after* the SQL statement is fully processed.

This kind of problems is the reason of the `JEPLTask` parameter, this object represents the processing going to be executed *`after`* the call to `setupJEPLX` methods. Because this object is like a "closure" of this consecutive task, we can *optionally* call `JEPLTask.exec()` when this call returns we know the consecutive processing has finished. The `JEPLTask.exec()` returns the value going to be returned by the next step of the persistence lifecycle, the return value may be ignored.

For instance:

- `void setupJEPLConnection(JEPLConnection con,JEPLTask<T> task)`

  When `JEPLTask.exec()` is executed the SQL statement has been executed and results processed, it returns the expected result value of the statement.

- `void setupJEPLPreparedStatement(JEPLPreparedStatement stmt,JEPLTask<T> task)`

  When `JEPLTask.exec()` is executed the SQL statement has been executed and results processed (`ResultSet` is closed), it returns the expected result value of the statement.

- `void setupJEPLResultSet(JEPLResultSet jrs,JEPLTask<?> task)`

  When `JEPLTask.exec()` is executed the `ResultSet` has been processed collecting rows but it is still alive.

Now we can solve our "configuration memory" problem easily:

```
public List<Contact> selectAllStatementListenerMaxRows(final int maxRows)
{
    JEPLPreparedStatementListener<List<Contact>> listener =
            new JEPLPreparedStatementListener<List<Contact>>()
    {
        public void setupJEPLPreparedStatement(JEPLPreparedStatement jstmt,
                JEPLTask<List<Contact>> task) throws Exception
        {
            PreparedStatement stmt = jstmt.getPreparedStatement();
            int old = stmt.getMaxRows();
            stmt.setMaxRows(maxRows);
            try
            {
                List<Contact> res = task.exec();
            }
            finally
            {
                stmt.setMaxRows(old); // Restore
            }
        }
    };

    return dao.createJEPLDAOQuery("SELECT * FROM CONTACT")
            .addJEPLListener(listener)
            .getResultList();
}
```

Anyway if you are not comfortable with this kind of stuff you can disable statement caching calling `JEPLDataSource.setPreparedStatementCached(false)` and now the `PreparedStatement` object is ever new (not reused).

```
JEPLDataSource jds = ...;
jds.setPreparedStatementCached(false);

...
JEPLPreparedStatementListener<List<Contact>> listener =
        new JEPLPreparedStatementListener<List<Contact>>()
{
    public void setupJEPLPreparedStatement(JEPLPreparedStatement jstmt,
            JEPLTask<List<Contact>> task) throws Exception
    {
        PreparedStatement stmt = jstmt.getPreparedStatement();
        stmt.setMaxRows(maxRows);  // Now is not reused
    }
};
```

5.3.2 Select query paged specified by index range

In this case we want only the results in a range specified by indexes, in this example `from` and `to` are row indexes and `row` to is not included.

```
public List<Contact> selectAllExplicitResultSetListenerRange(final int from,
            final int to)
{
    JEPLResultSetDAOListener<Contact> listener =
            new JEPLResultSetDAOListener<Contact>()
    {
        @Override
        public  void setupJEPLResultSet(JEPLResultSet jrs,JEPLTask<?> task)
                        throws Exception
        {
            ResultSet rs = jrs.getResultSet();
            rs.absolute(from);
        }

        @Override
        public Contact createObject(JEPLResultSet jrs) throws Exception
        {
            return new Contact();
        }

        @Override
        public void fillObject(Contact obj,JEPLResultSet jrs)
                        throws Exception
        {
            ResultSet rs = jrs.getResultSet();

            obj.setId(rs.getInt("ID"));
            obj.setName(rs.getString("NAME"));
            obj.setPhone(rs.getString("PHONE"));
            obj.setEmail(rs.getString("EMAIL"));
            int row = rs.getRow();
            if (row + 1 == to)
                jrs.stop();
        }
    };
```

```
        return dao.createJEPLDAOQuery("SELECT * FROM CONTACT")
                .addJEPLListener(listener)
                .getResultList();
    }
```

Note the call to the method `JEPLResultSet.stop()`, this call instructs JEPLayer to stop iterating results, in some way is like the "break" keyword in a Java loop.

5.4 FREE-HAND QUERIES

One of the most interesting features of RDBMSs is the flexibility of queries, we can get the exact information we want. Sometimes we need a minimal subset of columns, or several columns of disparate tables or aggregated results. In these cases we do not need DAOs.

In JEPLayer we have the `JEPLDALQuery.getJEPLCachedResultSet()` method returning `JEPLCachedResultSet` objects; `JEPLCachedResultSet` is similar to `javax.sql.ResultSet`, the main difference is data returned by iterating the `ResultSet` is saved into a `JEPLCachedResultSet` instance and returned, hence you don't need a database connection open to make use of this object. `JEPLCachedResultSet` is similar to `javax.sql.rowset.CachedRowSet` but a lot of simpler. `JEPLDALQuery.getJEPLCachedResultSet()` is a DAL-level method.

An example (two rows are supposed):

```
JEPLDataSource jds = ...;
JEPLDAL dal = jds.createJEPLDAL();
JEPLCachedResultSet resSet = dal.createJEPLDALQuery(
            "SELECT COUNT(*) AS CO,AVG(ID) AS AV FROM CONTACT")
            .getJEPLCachedResultSet();

String[] colNames = resSet.getColumnLabels();
if (colNames.length != 2) throw new RuntimeException("UNEXPECTED");
if (!colNames[0].equals("CO")) throw new RuntimeException("UNEXPECTED");
if (!colNames[1].equals("AV")) throw new RuntimeException("UNEXPECTED");
if (resSet.size() != 2) throw new RuntimeException("UNEXPECTED");

int count = resSet.getValue(1, 1, int.class); // Row 1, column 1
if (count != 2) throw new RuntimeException("UNEXPECTED");
count = resSet.getValue(1, "CO", int.class);
if (count != 2) throw new RuntimeException("UNEXPECTED");

float avg = resSet.getValue(1, 2, float.class); // Row 1, column 2
if (avg <= 0) throw new RuntimeException("UNEXPECTED");
avg = resSet.getValue(1, "AV", float.class);
if (avg <= 0) throw new RuntimeException("UNEXPECTED");
```

5.5 GROUPING PERSISTENT ACTIONS: JEPLTASKS

So far every SQL sentence completes a persistent JDBC lifecycle getting a `Connection` from the pool and releasing it before the method call ends.

To reuse the same `Connection` we can automatically group several calls executing them into a `JEPLTask`:

```
final JEPLDataSource jds = ...;
final ContactDAO dao = new ContactDAO(jds);
```

```
JEPLTask<Contact> task = new JEPLTask<Contact>()
{
    @Override
    public Contact exec() throws Exception
    {
        Contact contact = new Contact();
        contact.setName("A Contact object");
        contact.setPhone("9999999");
        contact.setEmail("contact@world.com");
        dao.insert(contact);

        Contact contact2 = dao.selectById(contact.getId());
        return contact2;
    }
};
Contact contact = jds.exec(task);
...
```

Before executing the `exec()` method a `Connection` is got from the `DataSource` and in the end the `Connection` is released returning to the pool. Because the `Connection` object is now shared by several SQL statements, `PreparedStatement` objects are cached and reused when the SQL string is the same, this usually happens when the same persistence method is called several times inside the same task execution.

JEPLayer automatically detects whether a `JEPLDAL` or `JEPLDAO` object is being used into a `JEPLTask` executed by using the same `JEPLDataSource` (the `JEPLDataSource` object must be the same as the object used to create `JEPLDAL` or `JEPLDAO` objects) so the same `Connection` object is ever user.

A `JEPLTask` can return any object or none (null).

We can provide a listener `JEPLListener` calling the `exec` method of `JEPLDataSource` with this signature:

```
public <T> T exec(JEPLTask<T> task,JEPLListener listener);
```

This listener takes precedence of any other level of listener registration (unless two or more `JEPLTask` are nested).

5.6 NON-JTA TRANSACTIONS

A non-JTA transaction is a transaction only based on JDBC, that is auto-commit mode set to false, with no other artefact like those defined in Java Transaction API (JTA). JEPLayer also supports JTA based transactions.

Once we have defined how to group persistent actions we are ready to execute transactions, transaction demarcation is automatic, in JEPLayer you do not need to explicitly call to `Connection.commit()` and `Connection.rollback()` methods in spite of manually demarcation is also possible.

Transaction demarcation are based on `JEPLTask`, the objective is mimic the declarative transactional demarcation of Inversion of Control containers like Spring[8] or EJB[9] (JavaEE), in JEPLayer is amazingly simple and does not require the intrusiveness of a IoC container and you

[8] http://static.springsource.org/spring/docs/3.0.x/reference/transaction.html

[9] http://download.oracle.com/docs/cd/B32110_01/web.1013/b28221/servtran001.htm

do not need to lose control of the lifecycle of your objects avoiding the viral nature of current IoC approaches.

There are several ways of declaring transactional behavior:

5.6.1 Transactional behavior configured in JEPLDataSource level

```
DataSource ds = ...;
JEPLBootNonJTA boot = JEPLBootRoot.get().createJEPLBootNonJTA();
JEPLNonJTADataSource jds = boot.createJEPLNonJTADataSource(ds);
jds.setDefaultAutoCommit(false);
```

This code creates and sets up a `JEPLDataSource` as transactional based on pure JDBC transactions (by default auto-commit is enabled that is not transactional), concretely a `JEPLNonJTADataSource`. In this case is important to use this inherited interface because JTA and non-JTA transactions are different, in JTA based transactions `Connection.setAutoCommit(boolean)` method must not be called because transaction boundaries are defined in a different way, this is why `setDefaultAutoCommit(boolean)` method is only defined in `JEPLNonJTADataSource` level.

Following this example when a `Connection` is got from `DataSource`, `Connection.setAutoCommit(false)` is set before executing the SQL sentence(s), and `Connection.commit()` is called when SQL sentence(s) ends, on error (any kind of `Exception`) `Connection.rollback()` is called. Before returning the `Connection` to the pool, auto-commit is ever set to true by JEPLayer regardless the configuration because this is a good practice.

We have seen how we can execute SQL sentences with no `JEPLTask`, in this case the auto-commit configuration rule is applied to every SQL sentence, every SQL sentence is separately (atomically) executed, no `Connection` is shared and hence there is no shared transaction, obviously using a `JEPLTask` is the interesting use case when it comes to transactions.

The following task will be executed into a transaction:

```
JEPLNonJTADataSource jds = boot.createJEPLNonJTADataSource(ds);
jds.setDefaultAutoCommit(false);
...
JEPLTask<Contact> task = new JEPLTask<Contact>()
{
    @Override
    public Contact exec() throws Exception
    {
        ...
    }
};
Contact contact = jds.exec(task);
```

Configuration of transactional mode calling `JEPLNonJTADataSource.setDefaultAutoCommit(boolean)` must be done before executing any SQL sentence or `JEPLTask`.

5.6.2 Transactional behavior configured with an annotation in JEPLTask level

If we change the previous code to:

```
JEPLNonJTADataSource jds = ...;
jds.setDefaultAutoCommit(true);
...
JEPLTask<Contact> task = new JEPLTask<Contact>()
{
    @Override
    @JEPLTransactionalNonJTA
```

```
        public Contact exec() throws Exception
        {
            ...
        }
    };
    Contact contact = jds.exec(task);
```

The annotation `@JEPLTransactionalNonJTA` specifies that this task is transactional. This is the same as this verbose alternative:

```
@JEPLTransactionalNonJTA(autoCommit = false)
```

The annotation sets the auto-commit mode to false of the task, the annotation takes precedence to default configuration calling `setDefaultAutoCommit(boolean)` in `JEPLNonJTADataSource`, hence in this example this task is transactional (auto-commit is set to true).

To declare the task as not transactional use:

```
@JEPLTransactionalNonJTA(autoCommit = true)
```

5.6.3 Transactional behavior configured with a parameter in JEPLTask level

If we change the previous code to:

```
    JEPLNonJTADataSource jds = ...;
    jds.setDefaultAutoCommit(false);
    ...
    JEPLTask<Contact> task = new JEPLTask<Contact>()
    {
        @Override
        public Contact exec() throws Exception
        {
            ...
        }
    };
    Contact contact = jds.exec(task,true);
```

The signature of this `JEPLNonJTADataSource exec` method is:

```
        public <T> T exec(JEPLTask<T> task,boolean autoCommit);
```

The auto-commit parameter of `exec` method takes precedence to default configuration calling `setDefaultAutoCommit(boolean)` in `JEPLNonJTADataSource` and transaction demarcation by `@JEPLTransactionalNonJTA` annotation, hence in this example this task is not transactional (auto-commit is set to true).

5.6.4 Transactional behavior configured with JEPLConnectionListener

We can register a `JEPLConnectionListener` like this one:

```
    JEPLDataSource jds = ...;
    jds.addJEPLListener(new JEPLConnectionListener()
        {
            public void setupJEPLConnection(JEPLConnection con,JEPLTask<?> task)
                        throws Exception
            {
                 con.getConnection().setAutoCommit(true);
            }
        }
    );
```

This configuration approach is very similar to calling `JEPLNonJTADataSource.setDefaultAutoCommit(boolean)`.

This method must be called *before* executing any SQL sentence (otherwise an exception is thrown), the `Connection` is configured as non-transactional because this listener is called *after* any kind of `Connection` preparation by JEPLayer, hence this listener dictates the final transactional mode (if this listener is finally applied because they could be more listeners of this type with more priority). As `JEPLConnectionListener` registered in `JEPLDataSource` level has priority over `JEPLNonJTADataSource.setDefaultAutoCommit(boolean)`. JEPLayer only calls `Connection.commit()` or `Connection.rollback()` whether `Connection.getAutoCommit()` returns false when the SQL sentence or `JEPLTask` ends.

As you already know, you can register this kind of listener in several locations/levels, for instance in `JEPLTask`:

```
JEPLDataSource jds = ... ;
JEPLTask<Contact> task = ...;
jds.exec(task,new JEPLConnectionListener()
    {
        public void setupJEPLConnection(JEPLConnection con,JEPLTask<?> task)
                    throws Exception
        {
            con.getConnection().setAutoCommit(false);
        }
    }

 );
```

This code executes the task inside a JDBC transaction (note the listener is called once), in this case this listener takes precedence of another `JEPLConnectionListener` registered in `JEPLDataSource` level and the later is not executed. JEPLayer understands that your provided a `JEPLConnectionListener` as a parameter in `JEPLDataSource.exec()` method, is your opportunity to define the auto-commit mode in the highest level, so no other auto-commit rule is applied like `@JEPLTransactionalNonJTA` annotation if present or default auto-commit configuration or default `JEPLConnectionListener` registered in `JEPLDataSource` level.

5.6.5 Manual demarcation of a transaction with JEPLConnectionListener

The signature of the method:

```
void setupJEPLConnection(JEPLConnection con,JEPLTask<?> task)
```

Includes a `JEPLTask` parameter, if we execute this task accordingly with previously documented behavior, the SQL sentences will be executed and then the method returns… what about transactional behavior?

If you do not call `JEPLTask.exec()` JEPLayer automatically calls `commit()` and `rollback()` if `Connection` is not in auto-commit mode but if `JEPLTask.exec()` is called JEPLayer behavior IS DIFFERENT, no `commit()` or `rollback()` are called because the developer has a chance to manually control the transaction.

Example:

```
JEPLDataSource jds = ...;
JEPLTask<Contact> task = ...;
jds.exec(task,new JEPLConnectionListener()
    {
        public void setupJEPLConnection(JEPLConnection con,JEPLTask<?> task)
           throws Exception
        {
            con.getConnection().setAutoCommit(false); // transaction
            try
            {
                task.exec();
                con.getConnection().commit();
```

```
                }
                catch(Exception ex)
                {
                    con.getConnection().rollback();
                    throw ex;
                }
            }
        }
    );
```

There is no need to restore `setAutoCommit(true)`, JEPLayer restores the default state before returning to the `DataSource` pool the active `Connection`.

5.6.6 Manual demarcation of a transaction using JEPLTransaction

`JEPLTransaction` is a fancy interface to hide direct transaction management with `Connection`.

The previous example can be coded with exactly the same behavior using `JEPLTransaction`:

```
JEPLDataSource jds = ...;
JEPLTask<Contact> task = ...;
jds.exec(task,new JEPLConnectionListener()
    {
        public void setupJEPLConnection(JEPLConnection con,JEPLTask<?> task)
                    throws Exception
        {
            JEPLTransaction txn = con.getJEPLTransaction();
            txn.begin(); // Executes setAutoCommit(false);
            try
            {
                task.exec();
                txn.commit();
            }
            catch(Exception ex)
            {
                txn.rollback();
                throw ex;
            }
        }
    }
);
```

5.6.7 Manual demarcation of a transaction inside JEPLTask

Yes you can, you can directly demarcate you transaction inside your `JEPLTask`, it may be a bit confusing for JEPLayer but it works as expected. Take a look to this example:

```
final JEPLNonJTADataSource jds = ...;

JEPLTask<Contact> task = new JEPLTask<Contact>()
{
    @Override
    public Contact exec() throws Exception
    {
        Connection con = jds.getCurrentJEPLConnection().getConnection();
        con.setAutoCommit(false); // transaction

        Contact contact = ...; // Some persistent statements

        con.commit();
```

```
            con.setAutoCommit(true);

            return contact;
        }
    };
    jds.exec(task,false);
```

This example is intentionally confusing because the `JEPLTask` is executed in transactional mode (auto-commit set to false), JEPLayer sets auto-commit to false before executing `JEPLTask` custom code, a new call to `setAutoCommit(false)` is not important, the interesting part is when the task method ends, auto-commit mode was set to true, that is, no transaction, but JEPLayer thinks it must be transactional… not a problem JEPLayer only calls commit (or rollback in exception) when the `Connection` is not in auto-commit mode. If the last call, `setAutoCommit(true)` is removed JEPLayer detects the connection with auto-commit disabled and calls `commit()` again, this call does nothing because there is no pending persistent statement.

If you want to code this way of demarcation, disabling transactions is recommended, for instance calling:

```
...
jds.exec(task,true);
```

5.6.8 Nested transactional tasks

You can nest transactional tasks, when nested one `Connection` is opened by the outermost and is shared by all nested tasks, in the end of the outermost task the `Connection` is closed.

For instance:

```
final JEPLNonJTADataSource jds = ...;
...
JEPLTask<Contact> taskOutside = new JEPLTask<Contact>()
{
    public Contact exec() throws Exception
    {
        JEPLTask<Contact> taskInside = new JEPLTask<Contact>()
        {
            public Contact exec() throws Exception
            {
                return ...; // Database operations
            }
        };
        return jds.exec(taskInside,false);
    }
};
jds.exec(taskOutside,false);
```

In spite of previous example may seem weird, nested tasks may be very common, for instance when you have several methods in a DAO class and every method executes a `JEPLTask`, and inside a `JEPLTask` you need to call other method (containing a `JEPLTask`).

5.7 JTA TRANSACTIONS

JEPLayer also supports transactions based on Java Transaction API (JTA) by using the provided `javax.transaction.UserTransaction` and optionally a `javax.transaction.TransactionManager` object.

As you know in JTA `DataSource` objects must be JTA/XA capable and bound to the JTA infrastructure of your application server or JTA provider, otherwise JTA transactions have no effect into a conventional pure JDBC `DataSource`.

5.7.1 Tested JTA environments

JTA support in JEPLayer has been tested with success in the following environments:

- JOTM 2.1.9 with XAPool 1.6 beta (XAPool already included in JOTM 2.1.9 is buggy)[10] and MySQL 5.1.15 JDBC driver
- Atomikos 3.70[11]
- GlassFish 3.0.1 and built-in XA capable MySQL `DataSource`

In JOTM environment you must wrap your JDBC driver with XAPool `StandardXADataSource` to convert it into a `XADataSource` and then wrap it with `StandardXAPoolDataSource`, do not forget registering the JOTM `UserTransaction` singleton (this object is also a `TransactionManager`) on both objects. The source code distribution of JEPLayer contains a complete example. JEPLayer automatically recognizes a XAPool `XADataSource` and applies some workaround avoiding one important bug related to `PreparedStatement`[12]. Furthermore in JOTM example code of JEPLayer you will see a call to `StandardXADataSource.setPreparedStmtCacheSize(int)` with parameter 0 (cache disabled) because `PreparedStatement` internal cache of XAPool is buggy (or ruse of `PreparedStatement` objects is not standard), do not worry JEPLayer provides automatic reuse (that is caching) of `PreparedStatement` objects for you. In spite of JOTM/XAPool are not longer supported by their authors and very far of a perfect JTA implementation, they provide a good enough JTA environment and performance is perhaps unmatched (maybe because reliability against system failures is not the best).

In Atomikos you must wrap the `XADataSource` driver of your database with `AtomikosDataSourceBean`[13], JEPLayer distribution includes an example of configuring with Atomikos.

In an application server like GlassFish you usually configure the XA capable `DataSource` using the administrative tools or configuration files of the server obtaining it from the JNDI registry, the same for `UserTransaction` and `TransactionManager` singletons[14].

5.7.2 How JEPLayer works in JTA mode

Definition of JTA transactions in JEPLayer is very similar to non-JTA (pure JDBC) transactions, the main difference is, in non-JTA you configure transaction behavior with the auto-commit flag and a transaction is demarcated by JEPLayer calling `Connection.setAutoCommit(false)`, `Connection.commit()` or

---

[10] http://jotm.ow2.org JOTM provides JTA for any JDBC driver

[11] http://www.atomikos.com/Main/InstallingTransactionsEssentials

[12] http://stackoverflow.com/questions/6927561/standardxaconnectionhandlepreparestatement-should-not-be-used-outside-an-ejbser

[13] http://www.atomikos.com/Documentation/ConfiguringJdbc

[14] For instance, in GlassFish 3 the `UserTransaction` singleton is registered in JNDI with "`java:comp/UserTransaction`" and `TransactionManager` with "`java:appserver/TransactionManager`"

`Connection.rollback()`. In JTA and JEPLayer, transactions are configured using very similar conventions to Java EE[15] and transaction demarcation is done by JEPLayer calling `UserTransaction` methods (`begin()`, `commit()` and `rollback()`) and in certain cases calling `TransactionManager` methods when provided.

The same as non-JTA you do not need to explicitly call to `UserTransaction` methods JEPLayer calls them for you because transaction demarcation is also based on `JEPLTask`, in spite of manually demarcation is also possible. Again JEPLayer mimics the declarative transactional demarcation of Inversion of Control containers like Spring[16] or EJB[17] (JavaEE) with minimal infrastructure, in this case JEPLayer is even more similar.

When transactional behavior is declared JEPLayer begins a new transaction (if there is no one already active in the current thread) and then gets a new `Connection` from the pool, this `Connection` must be automatically enrolled by your JTA provider (otherwise JTA transactions have no effect in your `DataSource`), database actions are executed and the `Connection` is returned to the pool, finally the transaction is committed (or rolled back if some exception was thrown).

There are several ways of declaring JTA based transactional behavior:

5.7.3 Transactional behavior configured in JEPLDataSource level

In JTA, JEPLayer bootstrap is a bit more complicated:

```
UserTransaction txn = ...;
TransactionManager txnMgr = ...;
DataSource ds = ...;

JEPLBootJTA boot = JEPLBootRoot.get().createJEPLBootJTA();
boot.setUserTransaction(txn);
boot.setTransactionManager(txnMgr);

JEPLJTADataSource jds = boot.createJEPLJTADataSource(ds);
jds.setDefaultJEPLTransactionPropagation(
            JEPLTransactionPropagation.REQUIRED);
...

JEPLTask<Contact> task = new JEPLTask<Contact>()
{
    @Override
    public Contact exec() throws Exception
    {
        return ...; // Database actions
    }
};
Contact contact = jds.exec(task);
```

This code creates and sets up a JTA based `JEPLDataSource`, concretely a `JEPLJTADataSource` and configures the default transaction mode (propagation) as `JEPLTransactionPropagation.REQUIRED` , this configuration call is not necessary because this is the default value.

The enumeration `JEPLTransactionPropagation` defines the same values and semantics as `javax.ejb.TransactionAttributeType` of Java EE 5 EJB. According to the propagation

[15] http://java.sun.com/javaee/6/docs/api/javax/ejb/TransactionAttributeType.html

[16] http://static.springsource.org/spring/docs/3.0.x/reference/transaction.html

[17] http://download.oracle.com/docs/cd/B32110_01/web.1013/b28221/servtran001.htm

value defined the transaction is demarcated following the same rules defined in `javax.ejb.TransactionAttributeType` attributes, read the javadoc[18] of this enumeration to understand how JEPLayer works.

In previous example `REQUIRED` is the default mode and no other value is provided in a different step, before calling `JEPLTask.exec()` method JEPLayer checks whether a JTA transaction is active, if there is no one `UserTransaction.begin()` is called (because a transaction is "REQUIRED"), after the end of `JEPLTask.exec()` method JEPLayer calls `UserTransaction.commit()` or `UserTransaction.rollback()` if some exception has been thrown. If a JTA transaction is active then JEPLayer does not demarcate again the transaction calling `UserTransaction` methods and no new transaction is created. As you can see the semantic of `JEPLTransactionPropagation.REQUIRED` is the same as `TransactionAttributeType.REQUIRED`.

Basic support of JTA only needs the `UserTransaction` object registered into the `JEPLBootJTA` used to create the `JEPLJTADataSource`. `TransactionManager` is optional and only is required if propagation mode is `NOT_SUPPORTED` or `REQUIRES_NEW` and there is a JTA transaction active before calling the persistent task, in this case the transaction is suspended/resumed calling `suspend()` and `resume()` methods of `TransactionManager`.

5.7.4 Transactional behavior configured with an annotation in JEPLTask level

Another way to define the transactional behavior is through the annotation `@JEPLTransactionalJTA`.

Adding this annotation to the previous example:

```
JEPLJTADataSource jds = ...;
jds.setDefaultJEPLTransactionPropagation(
            JEPLTransactionPropagation.NOT_SUPPORTED);
...

JEPLTask<Contact> task = new JEPLTask<Contact>()
{
    @Override
    @JEPLTransactionalJTA
    public Contact exec() throws Exception
    {
        return ...; // Database actions
    }
};
Contact contact = jds.exec(task);
```

The annotation `@JEPLTransactionalJTA` specifies that this task is transactional and the propagation mode is `REQUIRED` (the default mode). This is the verbose alternative:

```
@JEPLTransactionalJTA(propagation = JEPLTransactionPropagation.REQUIRED)
```

The annotation in `JEPLTask` takes precedence to default configuration calling `setDefaultJEPLTransactionPropagation()` in `JEPLJTADataSource`, hence in spite of value `NOT_SUPPORTED` this task is transactional.

For instance to declare the task as not transactional use:

```
@JEPLTransactionalJTA(propagation = JEPLTransactionPropagation.NEVER)
```

Or:

[18] http://java.sun.com/javaee/6/docs/api/javax/ejb/TransactionAttributeType.html

```
@JEPLTransactionalJTA(propagation = JEPLTransactionPropagation.NOT_SUPPORTED)
```

5.7.5 Transactional behavior configured with a parameter in JEPLTask level

If we change the previous code to:

```
JEPLJTADataSource jds = ...;
jds.setDefaultJEPLTransactionPropagation(
            JEPLTransactionPropagation.NOT_SUPPORTED);
...
JEPLTask<Contact> task = new JEPLTask<Contact>()
{
  @Override
  @JEPLTransactionalJTA(propagation=JEPLTransactionPropagation.NOT_SUPPORTED)
  public Contact exec() throws Exception
  {
      return ...; // Database actions
  }
};
Contact contact = jds.exec(task,JEPLTransactionPropagation.REQUIRED);
```

The signature of this `JEPLJTADataSource exec` method is:

```
public <T> T exec(JEPLTask<T> task,JEPLTransactionPropagation prop);
```

The propagation parameter of `exec` method takes precedence to default configuration calling `setDefaultJEPLTransactionPropagation()` in `JEPLJTADataSource` and transaction demarcation by `@JEPLTransactionalJTA` annotation, hence in this example in spite of `NOT_SUPPORTED` configurations, this task is transactional (`REQUIRED`).

5.7.6 Manual demarcation of a transaction with JEPLConnectionListener

In non-JTA we could ignore the automatic transaction demarcation of JEPLayer using a `JEPLConnectionListener` and implementing the method:

```
void setupJEPLConnection(JEPLConnection con,JEPLTask<T> task)
```

and executing the task in this method (calling `JEPLTask.exec()`), JEPLayer understood you wanted bypass built-in transactional demarcation.

In JTA this is not longer true because transaction demarcation takes place *before* getting and *after* releasing the `Connection` that is the transaction begins `setupJEPLConnection` before is executed. Sure you have thought "ok disabling transactions in JEPLayer I can manually begin and commit a transaction in `setupJEPLConnection`" and yes this an option but it is not going to work in any JTA environment, usually JTA providers enrol database actions into the current active transaction when the `Connection` is got from `DataSource` otherwise the `Connection` is used outside the transaction if it was got before the transaction begun[19].

In spite of this limitation you can manually commit or rollback a transaction created by JEPLayer into the `setupJEPLConnection` method. This works in JEPLayer because JEPLayer do not commit or rollback a transaction if the transaction is finished.

The following example shows how we can execute some persistent code in a transaction and ever rollback all persistent actions.

```
final JEPLJTADataSource jds = ...;
```

[19] In some stand alone JTA environment like JOTM Connections are ever bound to the transaction manager and order is not important.

```
// ...
JEPLTask<Contact> task = ...;
Contact contact = jds.exec(task,new JEPLConnectionListener<Contact>()
    {
      public void setupJEPLConnection(JEPLConnection con,
                          JEPLTask<Contact> task) throws Exception
      {
        UserTransaction txn = jds.getJEPLBootJTA().getUserTransaction();
        // DO NOT execute txn.begin() is already opened by JEPLayer
        try
        {
          Contact contact = task.exec();
        }
        finally
        {
          txn.rollback();
        }
      }
    },JEPLTransactionPropagation.REQUIRED);
```

5.7.7 Manual demarcation of a transaction using JEPLTransaction

`JEPLTransaction` is a wrapper hiding the underlying transaction mechanism, pure JDBC or JTA, because it was invented to make non-JTA and JTA compatible, the example seen in non-JTA is also valid in a JTA environment. In this case the underlying transaction manager is `UserTransaction` (see `JEPLTransaction.getUnderlyingTransaction(Class<T>)`), note the call to `isActive()` to detect if the transaction is already opened (in this example `begin` is not executed), by this way code of task is portable between JTA and non-JTA.

```
JEPLJTADataSource jds = ...;
JEPLTask<Contact> task = ...;
jds.exec(task,new JEPLConnectionListener()
    {
        public void setupJEPLConnection(JEPLConnection con,JEPLTask task)
                    throws Exception
        {
            JEPLTransaction txn = con.getJEPLTransaction();
            if (!txn.isActive()) txn.begin();
            try
            {
                task.exec();
                txn.commit();
            }
            catch(Exception ex)
            {
                txn.rollback();
                throw ex;
            }
        }
    },JEPLTransactionPropagation.REQUIRED);
```

5.7.8 Manual demarcation of a transaction outside JEPLTask

You can directly demarcate your transaction outside a `JEPLTask`, JEPLayer is still useful to automatically get and release database connections. To avoid built-in transaction management use `JEPLTransactionPropagation.REQUIRED` or `SUPPORTS`.

```
JEPLJTADataSource jds = ...;

UserTransaction txn = jds.getJEPLBootJTA().getUserTransaction();
```

```
txn.begin();
JEPLTask<Contact> task = new JEPLTask<Contact>()
{
    @Override
    public Contact exec() throws Exception
    {
        Contact contact = ...;
        return contact;
    }
};
jds.exec(task,JEPLTransactionPropagation.REQUIRED);
txn.commit();
```

5.7.9 Nested transactional tasks/regions

The same as non-JTA you can nest transactional regions and tasks, in nested tasks the same `Connection` is shared. For instance:

```
final JEPLJTADataSource jds = ...;
...
JEPLTask<Contact> taskOutside = new JEPLTask<Contact>()
{
    public Contact exec() throws Exception
    {
        JEPLTask<Contact> taskInside = new JEPLTask<Contact>()
        {
            public Contact exec() throws Exception
            {
                return ...;
            }
        };
        return jds.exec(taskInside,JEPLTransactionPropagation.SUPPORTS);
                                                        // OR REQUIRED
    }
};
jds.exec(taskOutside,JEPLTransactionPropagation.REQUIRED);
```

5.8 FAKE JTA TRANSACTIONS

JEPLayer has a "fake" mode for JTA, this mode is just a simple and basic but effective simulation of JTA based on pure JDBC transactions.

The interface `JEPLBootJTA` has a method `JEPLBootJTA.createJDBCUserTransaction()`, this method creates a "false" `UserTransaction` object simulating a true JTA compliant `UserTransaction`. You can use this object to register it into the `JEPLBootJTA`.

```
JEPLBootJTA boot = JEPLBootRoot.get().createJEPLBootJTA();
boot.setUserTransaction(boot.createJDBCUserTransaction());
```

This `UserTransaction` object is not JTA compliant, for instance:

1. Transactions are local (not distributed).
2. There is no two-phase commit: all connections are committed and rollbacked in the same time using basic JDBC transactions, for instance if one commit fails all pending transactions are rolled back but the already committed transactions cannot be rolled back (partial commit). Fortunately failing in `commit()` or `rollback()` is not usual.
3. Cannot be used outside of JEPLayer.

In spite of this fake `UserTransaction` object is not JTA compliant the JTA semantics and code are the same, the fake `UserTransaction` is multithread and knows all current

Connections opened by JEPLJTADataSource objects in the same JEPLBootJTA. You can use the fake mode and change easily to the true JTA environment, just changing the JEPLBootJTA.setUserTransaction(UserTransaction) call.

You just need the standard JTA API to compile and execute fake JTA transactions, for instance you can use the jar file ow2-jta-1.1-spec.jar included in latest JOTM distribution.

There is no "fake" TransactionManager, hence transactional behavior requiring a TransactionManager to suspend an alive transaction (NOT_SUPPORTED and REQUIRES_NEW modes) will fail.

5.9 SUPPORT OF MULTIPLE DATASOURCES IN JTA

The Java Transaction API standard has two main features: distributed transactions and coordination of several data sources (usually different databases) like two-phase commit transactions.

Regarding distributed transactions JEPLayer does not provide something special, if you call a remote method, for instance using EJB remote beans, your JTA transaction will be propagated to the remote end as usual.

However JEPLayer provides some utilities in case of several DataSource objects involved in the same (shared) JTA transaction. JEPLayer gives you a similar framework based on transactional tasks coordinating several JEPLJTADataSource, in this case the central framework object implements the public interface JEPLJTAMultipleDataSource, this object obtained from JEPLBootJTA calling JEPLBootJTA.getJEPLJTAMultipleDataSource() (ever the same object given the same JEPLBootJTA instance) manages all JEPLJTADataSource objects created by the same JEPLBootJTA object.

You only need the JEPLJTAMultipleDataSource object when you want to execute JTA transactions shared by two or more JTA data sources.

The approach is very similar to transactions managed by a JEPLJTADataSource, by using JEPLTask, JEPLTransactionPropagation and/or JEPLTransactionalJTA annotations. JEPLJTAMultipleDataSource provides the same semantic and behavior than JEPLJTADataSource based transactions.

The following example executes a transactional task shared by two data sources and default propagation JEPLTransactionPropagation.REQUIRED.

```
UserTransaction txn = ...;
TransactionManager txnMgr = ...;
DataSource ds1 = ...;
DataSource ds2 = ...;

JEPLBootJTA boot = JEPLBootRoot.get().createJEPLBootJTA();
boot.setUserTransaction(txn);
boot.setTransactionManager(txnMgr);

JEPLJTAMultipleDataSource jdsMgr = boot.getJEPLJTAMultipleDataSource();
final JEPLJTADataSource jds1 = boot.createJEPLJTADataSource(ds1);
final JEPLJTADataSource jds2 = boot.createJEPLJTADataSource(ds2);

JEPLTask<Contact> task = new JEPLTask<Contact>()
{
    @Override
    public Contact exec() throws Exception
    {
        // Database actions using jds1 and jds2
        return ...;
```

```
        }
    };
    jdsMgr.exec(task);
```

Before creating the JEPLJTADataSource objects you can change the default transaction propagation (must be called before otherwise an exception is thrown):

```
JEPLJTAMultipleDataSource jdsMgr = boot.get JEPLJTAMultipleDataSource();
jdsMgr.setDefaultJEPLTransactionPropagation(
        JEPLTransactionPropagation.REQUIRED);

final JEPLJTADataSource jds1 = boot.createJEPLJTADataSource(ds1);
final JEPLJTADataSource jds2 = boot.createJEPLJTADataSource(ds2);
...
```

And the same way JEPLJTADataSource as you can define the transaction propagation as a JEPLTransactionalJTA annotation

```
    JEPLJTAMultipleDataSource jdsMgr = ...;
    // ...
    JEPLTask<Contact> task = new JEPLTask<Contact>()
    {
        @Override
        @JEPLTransactionalJTA
        public Contact exec() throws Exception
        {
            // Persistent actions using jds1 and jds2
            return ...;
        }
    };
    jdsMgr.exec(task);
```

Or providing a second parameter of type JEPLTransactionPropagation to the JEPLJTAMultipleDataSource.exec() method.

```
    JEPLJTAMultipleDataSource jdsMgr = ...;
    // ...
    JEPLTask<Contact> task = ...;
    jdsMgr.exec(task,JEPLTransactionPropagation.MANDATORY);
```

When executing a transaction JEPLJTAMultipleDataSource ensures that the same Connection object is used in every JEPLJTADataSource associated and is automatically released (returning to the pool) in the end of the task execution, transaction demarcation happens before getting and after releasing connections from DataSource pools.

JEPLJTAMultipleDataSource also supports nested transactions and internal tasks executed by JEPLJTADataSource objects do not get/release a new Connection, the Connection automatically obtained by JEPLJTAMultipleDataSource is used instead.

5.10 UNCACHED RESULTSET

JEPLDAOQuery<T> has this method:

```
public JEPLResultSetDAO<T> getJEPLResultSetDAO()
```

returning a JEPLResultSetDAO<T> object, because we are into the DAO level of JEPLayer JEPLResultSetDAO<T> sounds like a ResultSet of developer defined objects, and this is right, then what is the difference with getResultList() methods returning a List of objects?

The returned `JEPLResultSetDAO<T>` keeps alive a `ResultSet` and is designed to get results on demand, the same way as `ResultSet` (in fact this object is used under the hood) but in this case we get developer defined objects. This is the only case the `ResultSet` used is not closed before the method returns. This approach allows querying millions of rows and in the same time only to load the required number or rows while iterating.

Use of `JEPLResultSetDAO` is very simple because it mimics `ResultSet` but a lot of easier because every row is converted to the required user object using the registered `JEPLResultSetDAOListener<T>` listener, so we just need two simple methods in `JEPLResultSetDAO<T>`:

```
public boolean next();
public T getObject();
```

Because `JEPLResultSetDAO<T>` keeps internally a `ResultSet` object we need to close this object in some way. The `ResultSet` is closed in three cases:

1. The first time calling `next()` method it returned false. In this case the `ResultSet` is automatically closed.
2. Explicit call to `JEPLResultSetDAO.close()`. This method is defined in `JEPLResultSet`, the base interface.
3. Explicit call to `ResultSet.close()` on the `ResultSet` obtained calling `JEPLResultSetDAO.getResultSet()`. This way is not recommended, use ever JEPLayer methods when provided.

Because `JEPLResultSetDAO<T>` needs an "alive" `Connection` for a while, `getJEPLResultSetDAO()` method *only can be called inside a* `JEPLTask`, in a different scenario the `Connection` object is closed before the method returns (in this case JEPLayer throws an exception of missing `JEPLTask`).

For instance, adding this method to `ContactDAO`:

```
public JEPLResultSetDAO<Contact> selectAllResultSetDAO()
{
    return dao.createJEPLDAOQuery(
           "SELECT * FROM CONTACT").getJEPLResultSetDAO();
}
```

And a use example (remember, this code must be executed inside a `JEPLTask`):

```
ContactDAO dao = ...;
JEPLResultSetDAO<Contact> resSetDAO = dao.selectAllResultSetDAO();
if (resSetDAO.isClosed()) throw new RuntimeException("UNEXPECTED");
while(resSetDAO.next())
{
    Contact contact = resSetDAO.getObject();
    System.out.println("Contact: " + contact.getName());
}
// Now we know is closed
if (!resSetDAO.isClosed()) throw new RuntimeException("UNEXPECTED");
```

`JEPLResultSetDAO<T>` is even more interesting because it inherits from `List<T>`, and internally defines an on demand `Iterator` and `ListIterator`. For instance, the previous code could be written this way:

```
ContactDAO dao = ...;
List<Contact> resSetDAO = dao.selectAllResultSetDAO();
if (((JEPLResultSetDAO)resSetDAO).isClosed())
    throw new RuntimeException("UNEXPECTED");
for(Contact contact : resSetDAO) // Uses Iterator<Contact>
{
```

```
            System.out.println("Contact: " + contact.getName());
        }
        // Now we know is closed
        if (!((JEPLResultSetDAO)resSetDAO).isClosed())
            throw new RuntimeException("UNEXPECTED");
```

What about other `List` methods?

The implementation of `JEPLResultSetDAO<T>` is backed by a conventional `LinkedList` collection working as a cache holding all already loaded objects, if you call a `List` method and this call tries to access an object not in the list because the required index is after the last obtained index from the database, then the `JEPLResultSetDAO<T>` object is automatically advanced to the required index, otherwise the internal `LinkedList` is used to return the required object. Many List methods forward `JEPLResultSetDAO<T>` to the end of the internal `ResultSet`, automatically closing the internal `ResultSet`, when the `ResultSet` is closed only the internal `LinkedList` collection is used and the `JEPLResultSetDAO<T>` works the same as a conventional `LinkedList`. For instance a call to `List.size()` reads all objects from database and automatically closes the `JEPLResultSetDAO` closing the internal `ResultSet` too.

For instance:

```
ContactDAO dao = ...;
List<Contact> resSetDAO = dao.selectAllResultSetDAO();
Contact contact1 = resSetDAO.get(0); // Got from DB
Contact contact2 = resSetDAO.get(0); // Got from internal list (same obj)
if (contact1 != contact2) throw new RuntimeException("UNEXPECTED");
```

Finally the interface `JEPLResultSetDAO` has this method:

```
int count()
```

This method works the same as `List.size()`, it was added to avoid the problem of `size()` with debuggers like NetBeans debugger. When debugging on NetBeans `List.size()` is automatically called to show the current size of the `List`, because this method is "database-sensible" this typical behavior of debuggers is problematic because it implies a collateral effect. Fortunately debugging is detected and `size()` method is not allowed in this scenario but JEPLayer cannot distinguish when `size()` is called by debugger or by end user code and throws an exception when called in debugging mode. The alternative problem-free to `size()` method is `JEPLResultSetDAO.count()`.

5.11 INHERITANCE EXAMPLE

This chapter does not add new JEPLayer concepts and API, is just an example of how we can use JEPLayer to resolve relatively complex problems. For instance inheritance table-class without a column discriminator, most of ORMs need this "disturbing" column, is not the case of JEPLayer.

In the beginning of this document we created three tables with 1-1 relationships, we are going to represent these relationships as inheritance of the classes `Person` and `Company` from `Contact`.

The `Person` class:

```
public class Person extends Contact
{
    protected int age;

    public Person(int id, String name, String phone, String email,int age)
    {
```

```
        super(id,name,phone,email);
        this.age = age;
    }

    public Person() {}

    public int getAge()
    {
        return age;
    }

    public void setAge(int age)
    {
        this.age = age;
    }
}
```

The Company class:

```
public class Company extends Contact
{
    protected String address;

    public Company(int id, String name, String phone, String email,
        String address)
    {
        super(id,name,phone,email);
        this.address = address;
    }

    public Company() {}

    public String getAddress()
    {
        return address;
    }

    public void setAddress(String address)
    {
        this.address = address;
    }
}
```

The `PersonDAO`:

```
public class PersonDAO implements JEPLResultSetDAOListener<Person>
{
    protected ContactDAO contactDAO;
    protected JEPLDAO<Person> dao;

    public PersonDAO(JEPLDataSource ds)
    {
        this.dao = ds.createJEPLDAO(Person.class);
        dao.addJEPLListener(this);
        this.contactDAO = new ContactDAO(ds);
    }

    public JEPLDAO<Person> getJEPLDAO()
    {
        return dao;
    }
```

```
@Override
public void setupJEPLResultSet(JEPLResultSet jrs,JEPLTask<?> task)
    throws Exception
{
}

@Override
public Person createObject(JEPLResultSet jrs) throws Exception
{
    return new Person();
}

@Override
public void fillObject(Person obj,JEPLResultSet jrs) throws Exception
{
    contactDAO.fillObject(obj, jrs);

    ResultSet rs = jrs.getResultSet();
    obj.setAge(rs.getInt("AGE"));
}

public void insert(Person obj)
{
    contactDAO.insert(obj);
    dao.createJEPLDALQuery("INSERT INTO PERSON (ID, AGE) VALUES (?, ?)")
            .addParameters(obj.getId(),obj.getAge())
            .setStrictMinRows(1).setStrictMaxRows(1)
            .executeUpdate();
}

public void update(Person obj)
{
    contactDAO.update(obj);
    dao.createJEPLDALQuery("UPDATE PERSON SET AGE = ? WHERE ID = ?")
            .addParameters(obj.getAge(),obj.getId())
            .setStrictMinRows(1).setStrictMaxRows(1)
            .executeUpdate();
}

public boolean deleteByIdCascade(int id)
{
    // Only use when ON DELETE CASCADE is defined in foreign keys
    return contactDAO.deleteById(id);
}

public boolean deleteByIdNotCascade1(int id)
{
    boolean res = dao.createJEPLDALQuery("DELETE FROM PERSON WHERE ID = ?")
            .setStrictMinRows(0).setStrictMaxRows(1)
            .addParameter(id)
            .executeUpdate() > 0;
    if (res) contactDAO.deleteById(id);
    return res;
}

public boolean deleteByIdNotCascade2(int id)
{
    // Only MySQL
    return dao.createJEPLDALQuery("DELETE C,P FROM CONTACT C " +
                "LEFT JOIN PERSON P ON C.ID = P.ID " +
                "WHERE P.ID = ?")
            .setStrictMinRows(0).setStrictMaxRows(1)
```

```
                .addParameter(id)
                .executeUpdate() > 0;
    }

    public boolean delete(Person person)
    {
        return deleteByIdCascade(person.getId());
    }

    public int deleteAll()
    {
        return deleteAllCascade();
    }

    public int deleteAllCascade()
    {
        // Only use when ON DELETE CASCADE is defined in foreign key
        return dao.createJEPLDALQuery(
              "DELETE FROM CONTACT WHERE CONTACT.ID IN (SELECT ID FROM PERSON)")
                .executeUpdate();
    }

    public int deleteAllNotCascade()
    {
        // MySQL Only
        return dao.createJEPLDALQuery(
"DELETE CONTACT,PERSON FROM CONTACT INNER JOIN PERSON WHERE CONTACT.ID =
PERSON.ID").executeUpdate();
    }

    public List<Person> selectAll()
    {
        return dao.createJEPLDAOQuery(
                "SELECT * FROM PERSON P,CONTACT C WHERE P.ID = C.ID")
                .getResultList();
    }

    public Person selectById(int id)
    {
        return dao.createJEPLDAOQuery(
              "SELECT * FROM PERSON P,CONTACT C WHERE P.ID = C.ID AND P.ID = ?")
                .addParameter(id)
                .getSingleResult();
    }

    public List<Person> selectByNameAndEMail(String name,String email)
    {
        return dao.createJEPLDAOQuery(
"SELECT * FROM PERSON P,CONTACT C WHERE P.ID = C.ID AND C.NAME = ? AND C.EMAIL =
?")
                .addParameters(name,email)
                .getResultList();
    }

    public int selectCount()
    {
        return dao.createJEPLDALQuery("SELECT COUNT(*) FROM PERSON")
                    .getOneRowFromSingleField(int.class);
    }
}
```

If you take a look to the code you will see how a ContactDAO is used as an auxiliary DAO to perform actions related to the CONTACT table and Contact base class.

The class CompanyDAO would be very similar.

With some more similar work we could change ContactDAO as the DAO only for pure Contact objects (not Company or Person) in this case we would check ID in PERSON and COMPANY to be NULL.

5.11.1 Queries mixing several tables/object types in the same hierarchy

What if we need queries returning mixed objects all of them inherited from Contact?

In this case we could create a mixed DAO to manage the hierarchy of derived classes from Contact and Contact objects.

```
public class ContactTreeDAO implements JEPLResultSetDAOListener<Contact>
{
    protected JEPLDAO<Contact> dao;
    protected ContactDAO contactDAO;
    protected PersonDAO personDAO;
    protected CompanyDAO companyDAO;

    public ContactTreeDAO(JEPLDataSource ds)
    {
        this.dao = ds.createJEPLDAO(Contact.class);
        dao.addJEPLListener(this);
        this.contactDAO = new ContactDAO(ds);
        this.personDAO = new PersonDAO(ds);
        this.companyDAO = new CompanyDAO(ds);
    }

    public JEPLDAO<Contact> getJEPLDAO()
    {
        return dao;
    }

    @Override
    public void setupJEPLResultSet(JEPLResultSet jrs,JEPLTask<?> task)
        throws Exception
    {
    }

    @Override
    public Contact createObject(JEPLResultSet jrs) throws Exception
    {
        ResultSet rs = jrs.getResultSet();
        if (rs.getObject("P_ID") != null)
            return new Person();
        else if(rs.getObject("CP_ID") != null)
            return new Company();
        return new Contact();
    }

    @Override
    public void fillObject(Contact obj,JEPLResultSet jrs) throws Exception
    {
        if (obj instanceof Person)
            personDAO.fillObject((Person)obj, jrs);
        else if(obj instanceof Company)
            companyDAO.fillObject((Company)obj, jrs);
        else // Contact
```

```
            contactDAO.fillObject(obj, jrs);
    }

    public int deleteAll()
    {
        return deleteAllCascade();
    }

    public int deleteAllCascade()
    {
        // Only use when ON DELETE CASCADE is defined in foreign keys
        return dao.createJEPLDALQuery("DELETE FROM CONTACT").executeUpdate();
    }

    public int deleteAllNotCascade()
    {
        // MySQL Only
        // http://www.haughin.com/2007/11/01/mysql-delete-across-multiple-
tables-using-join/
        return dao.createJEPLDALQuery("DELETE C,P,CP FROM CONTACT C " +
                "LEFT JOIN PERSON P ON C.ID = P.ID " +
                "LEFT JOIN COMPANY CP ON C.ID = CP.ID ").executeUpdate();
    }

    public boolean deleteByIdNotCascade(int id)
    {
        // MySQL Only
        // http://www.haughin.com/2007/11/01/mysql-delete-across-multiple-
tables-using-join/
        return dao.createJEPLDALQuery("DELETE C,P,CP FROM CONTACT C " +
                "LEFT JOIN PERSON P ON C.ID = P.ID " +
                "LEFT JOIN COMPANY CP ON C.ID = CP.ID " +
                "WHERE C.ID = ?")
                .setStrictMinRows(0).setStrictMaxRows(1)
                .addParameter(id)
                .executeUpdate() > 0;
    }

    public List<Contact> selectAll()
    {
        return dao.createJEPLDAOQuery("SELECT C.ID,C.EMAIL,C.NAME,C.PHONE,P.ID
AS P_ID,P.AGE,CP.ID AS CP_ID,CP.ADDRESS " +
                "FROM CONTACT C " +
                "LEFT JOIN PERSON P ON C.ID = P.ID " +
                "LEFT JOIN COMPANY CP ON C.ID = CP.ID")
                .getResultList();
    }

    public Contact selectById(int id)
    {
        return dao.createJEPLDAOQuery("SELECT C.ID,C.EMAIL,C.NAME,C.PHONE,P.ID
AS P_ID,P.AGE,CP.ID AS CP_ID,CP.ADDRESS " +
                "FROM CONTACT C " +
                "LEFT JOIN PERSON P ON C.ID = P.ID " +
                "LEFT JOIN COMPANY CP ON C.ID = CP.ID " +
                "WHERE C.ID = ?")
                .addParameter(id)
                .getSingleResult();
    }
}
```

As you can see only one SQL sentence is needed to load the entire hierarchy and no column discriminator has been needed.