JDBaccess Version 1.0

for MySql and Oracle

User manual

Printed in Germany

First Publication: May 2006

Produced by: G. Christ

Contents

1 JDBaccess

1.1 What is JDBaccess

JDBaccess is a basic persistence library for the Java platform, which defines major database access operations in an easy usable API. The main interface classes are Transaction, Insert, Update, Delete, Select, Result, Procedure, Function, and CallResult.

JDBaccess is for programmers of the Java platform which do not sacrifice scalability but develop solutions that are simple and flexible.

JDBaccess is completely written in the Java programming language and only has a small size of about 170 KB. It fully supports the major two database systems MySql and Oracle with their Type 4 JDBC drivers (Oracle thin driver and Connector/J).

JDBaccess applications are portable and independent from the underlying database. You can focus on your object model and leave the details of persistence to the JDBaccess implementation. JDBC is completely hidden from the programmer. A JDBaccess application for example does not care about connection and statement handling. JDBaccess can help you to remove or encapsulate vendor-specific SQL code. JDBaccess goal is to relieve the developer from 90 % of common data persistence related programming tasks.

JDBaccess can’t be faster than JDBC itself, but it performs important operations by better usage of the JDBC methods. The average performance and throughput of applications with JDBaccess is much better than applications working with JDBC directly. We did a performance test with a typical application (see chapter 9 for details). You could see that these methods improve the overall performance and throughput.

You should use JDBaccess in a typical two-layer-architecture with a rich client and a database server. But within your client JDBaccess allows you to develop your application in a typical three-layer-architecture with presentation, application and persistence layer. JDBaccess is the software between your persistence layer in the client and the JDBC driver for your database server. JDBaccess supports the data access object and transfer object pattern (see java.sun.com/blueprints/corej2eepatterns/Patterns).

JDBaccess is an additional persistence framework to JDO (see java.sun.com/products/jdo) and Hibernate (see hibernate.org) which are more useful for handling complex objects and in application server environments.

1.2 Main features

1.2.1 Support of MySql and Oracle

JDBaccess supports the major two database systems MySql and Oracle with their type 4 JDBC drivers:

MySql: JDBaccess supports the MySql Connector/J driver versions 5 (5.0.3) and 3.1 (3.1.13) so that MySql 4 and MySql 5 database systems can be connected.

Oracle: JDBaccess supports the Oracle thin driver version 10 (10.1.0.4 and 10.2.0.1) and 9 (9.2.0.5) so that Oracle 8i, 9i and 10g database systems can be connected.

1.2.2 Simple

Easy interfaces
Database access operations are defined in easy usable but powerful interface classes (Transaction, Insert, Update, Delete, Select, Function, Procedure, Result, CallResult). These classes are all used in the same manner. You start a transaction, create your database access object and execute it. Results of selects and procedures can easily be used. On results you have typical fetch operations such as get next elements or get element at position.

Simple installation and usage
A programmer of the Java programming language is able to start his first JDBaccess application within one hour. Application development is fast because complex JDBC handling is completely hidden to the application programmer so that he can focus his attention on his main work - his application. JDaccess goal is to relieve the developer from 90 % of common data persistence related programming tasks.

1.2.3 Fast

Overall performance
We did a performance test with a typical application and get an improvement in overall performance and throughput (see chapter 9 for details).

Pooling of database connections
Connections which are used in write transactions are cached in the write connection pool. Connections which are used in read transactions such as for selections are cached in the read connection pool. For each pool you can define how many connections should be opened initially and how many connections should remain open.

Pooling of statements and their results
Statements of database access operations (insert, update, delete, select, function or procedure) in one transaction are cached in the statement pool. Executing a database access operation and getting meta data about the result such as the number, the SQL type or the length of the fields is done in one roundtrip. The state of the statement in the statement pool changed to used. All further result operations such as getting result values and result paging is done on this open statement. It remains in state used until the application explicitely ends the corresponding database access operation so that the statement can be released back to the cache of the statement pool. If the same database access operation is executed more than once, the corresponding open statement is fetched from statement pool without the need to prepare it again.

Batch inserts

The insert of objects is batched. Sequence numbers for id fields are fetched in one roundtrip.

Update only the modified fields
JDBaccess knows which fields are modified and will update only those fields.

Size of selection results
Determining the size of the select results is done with a special sql count statement. A SQL count statement can be set additionally by an application for further performance improvement.

Large object data (BLOB, CLOB)
Reading and writing large object data (clob, blob) is done by the fastest JDBC streaming methods.

1.2.4 Cheap

JDBaccess has a low fixed one-of price. The price is all-inclusive. No additional fees such as runtime fees are charged. All upgrades between major versions are for free. Since JDBacces is simple, no additional training and consulting is needed.

Following prices are valid:

	Single developer license	**Site license**
MySql	75 US$	750 US$
Oracle	150 US$	1500 US$

1.2.5 Hidden SQL

Data access files
JDBaccess loosely couples objects with SQL by using XML. Database access operations (Insert, Update, Delete, Select, Function and Procedure) can be defined in XML-files (da.xml). They have a type, a name, and the SQL code. Dependent of the type of the operation they have further fields such as a SQL count select string or output parameter types. The operational details are hidden from the programmer. He gets his operation by name and type. Also he no longer needs to compile his source code every time he changes some of his application SQL-code. For database specific performance improvements sql hints can be used in the SQL code.

The database access files can be placed anywhere in the java application package hierarchy. Inheritance is supported. A select “s” defined in “da.xml” in package “p1.p11” overwrites the same select “s” defined in “da.xml” in package “p1”. A select “s” only defined in package “p” can also be used in all subpackages of “p”.

1.2.6 Easy use of functions and procedures

Functions and procedures can be used easily. You start a transaction, set the package, the function or procedure name. If output parameters exist you set the appropriate output parameter types. Then you execute the function or procedure.

That’s all. After execution you can get the output parameters. Or if the function or procedure delivers one or more result sets you can access them by position and use them in the same manner as normal select results.

1.2.7 Additional features

Stable
JDBaccess applications are more stable than applications which use JDBC directly. You do not care about connection and statement handling which is done in central JDBaccess classes. Connections are automatically reconnected if network problems arise. If an application error appears a detailed ApplicationException - with message, cause and action instructions - is thrown.

Mapping of SQL and Java types
JDBaccess maps SQL types to the types of the Java programming language and vice versa automatically and in conformance to the Sun JDBC standard (see chapter 6 for details).

Unicode support
JDBaccess supports Unicode for writing and reading data in all char, varchar, text, clob, nchar, nvarchar and nclob columns. If you do so you have to set the character set or the national character set in your database management system to Unicode (e.g. in Oracle to AL32UTF8 or AL16UTF16).

2 System requirements

2.1 JDK

Before you can use JDBaccess you must first have the J2SE 1.4.2 or higher installed.

2.2 JDBC

JDBaccess works with the JDBC drivers of MySql and Oracle:

- MySql Connector/J: we recommend to use the version 5 (5.0.3) or version 3.1 (3.1.13) which support MySql 5 and MySql 4. Functions and procedures are supported by MySql since MySql 5.
- Oracle thin driver: we recommend to use the newest version 10 (10.1.0.4 or 10.2.0.1) which is faster than version 9 (9.2.0.5).

3 Installation

3.1 Unzipping JDBaccess

Unzip your jdbaccess zip file (for example jdbaccess-single-oracle.zip) into your project directory. You should see the following directories and files:

- desc: manual.pdf (this file), license.txt, oracleJDBCLicense.txt
- doc: Java documentation files of JDBaccess
- lib: jdbaccess.jar (and mysql-connector-java-5.0.3-bin.jar or ojdbc14.jar)
- src: Employee example files
- .project, .classpath, EmployeeExampleDA.launch, EmployeeExampleGUI.launch: Eclipse project files for the example applications

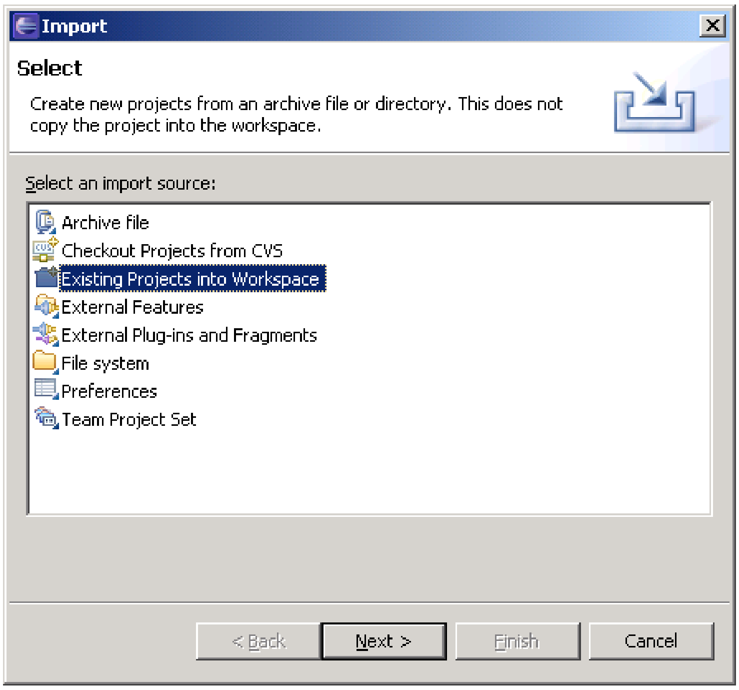
For MySql you should download the JDBC driver MySql Connector/J from www.mysql.com (mysql-connector-java-5.0.3-bin.jar) and copy it to the directory lib.

If you need another version of the JDBC driver copy it to the directory lib and change your .classpath-settings to the new entry, for example:

```
<classpathentry kind="lib" path="lib/mysql-connector-java-3.1.13-bin.jar"/>
```

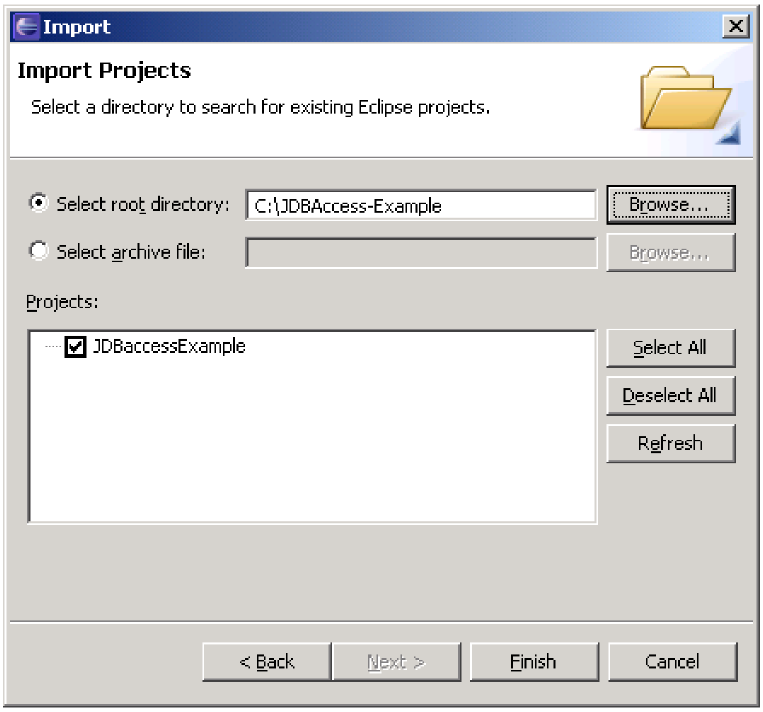
3.2 Starting the examples

With Eclipse you can directly import the example project by “File/Import/Existing projects into workspace”:



Import
Select
Create new projects from an archive file or directory. This does not copy the project into the workspace.
Select an import source:
Archive file
Checkout Projects from CVS
Existing Projects into Workspace
External Features
External Plug-ins and Fragments
File system
Preferences
Team Project Set
< Back
Next >
Finish
Cancel

Click on “Next” to get the next dialog for selecting your project directory:

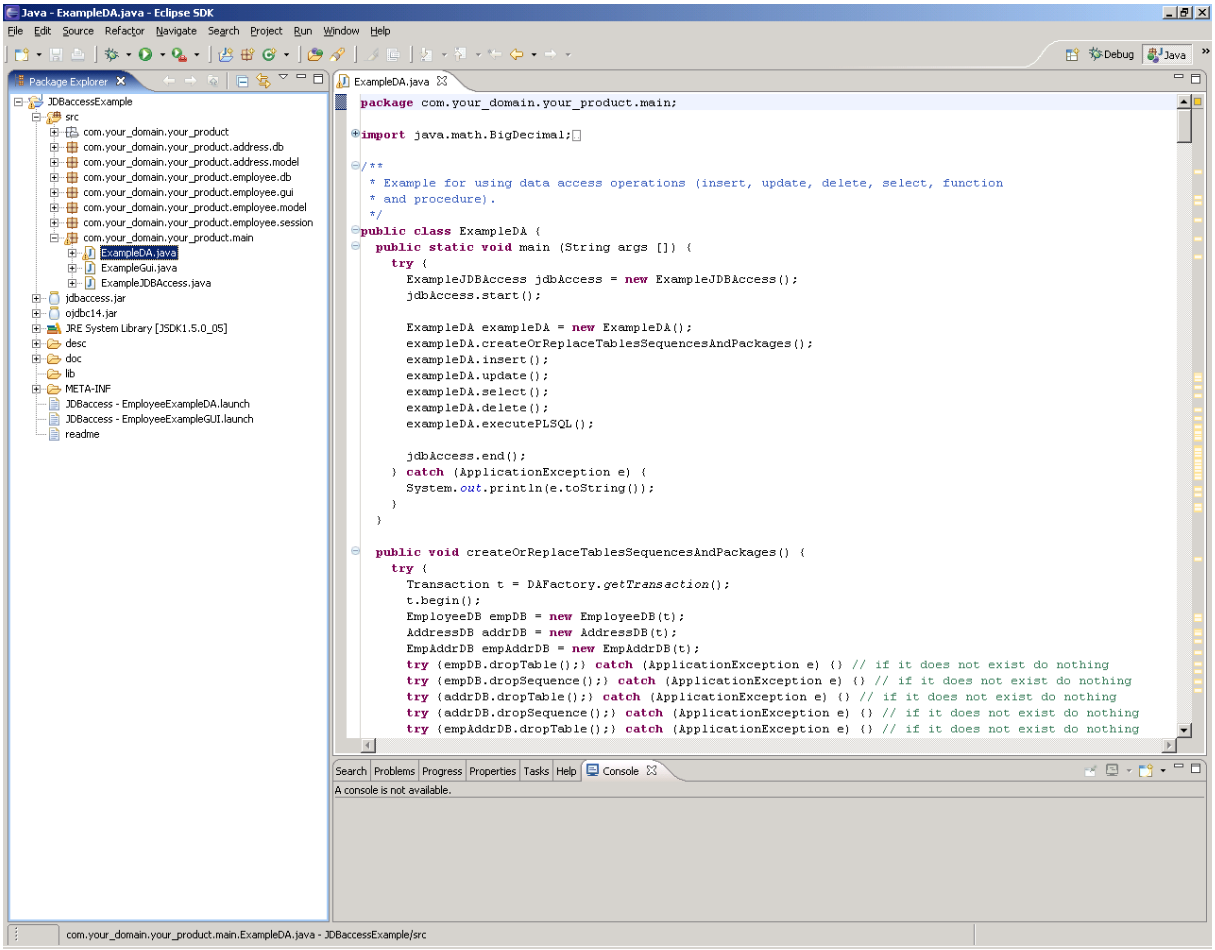


● Select root directory: C:\JDBAccess-Example
○ Select archive file:

Projects:
☑ JDBaccessExample

Click on “Finish” to finalize your import.

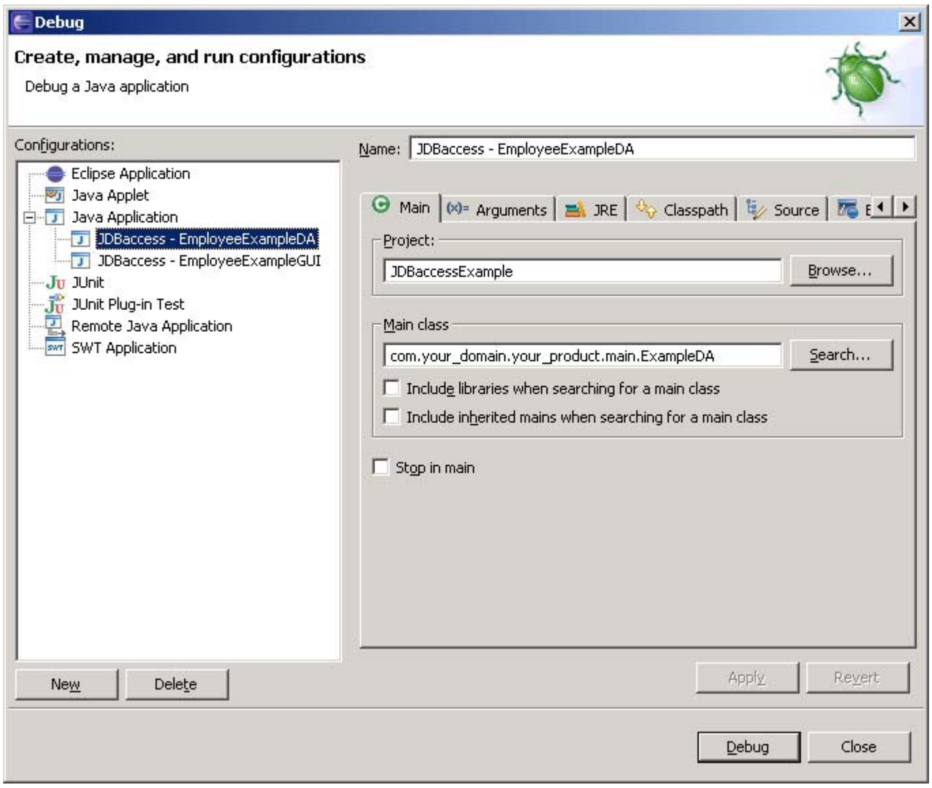
After importing your project in Eclipse you see the JDBaccessExample as a new Eclipse project:


Java - ExampleDA.java - Eclipse SDK
File
Edit
Source
Refactor
Navigate
Search
Project
Run
Window
Help
Debug
Java
Package Explorer
JDBaccessExample
src
com.your_domain.your_product
com.your_domain.your_product.address.db
com.your_domain.your_product.address.model
com.your_domain.your_product.employee.db
com.your_domain.your_product.employee.gui
com.your_domain.your_product.employee.model
com.your_domain.your_product.employee.session
com.your_domain.your_product.main
ExampleDA.java
ExampleGui.java
ExampleJDBAccess.java
jdbaccess.jar
ojdbc14.jar
JRE System Library [JSDK1.5.0_05]
desc
doc
lib
META-INF
JDBaccess - EmployeeExampleDA.launch
JDBaccess - EmployeeExampleGUI.launch
readme
ExampleDA.java
Search
Problems
Progress
Properties
Tasks
Help
Console
com.your_domain.your_product.main.ExampleDA.java - JDBaccessExample/src

```java
package com.your_domain.your_product.main;

import java.math.BigDecimal;

/**
 * Example for using data access operations (insert, update, delete, select, function
 * and procedure).
 */
public class ExampleDA {
  public static void main (String args []) {
    try {
      ExampleJDBAccess jdbAccess = new ExampleJDBAccess();
      jdbAccess.start();

      ExampleDA exampleDA = new ExampleDA();
      exampleDA.createOrReplaceTablesSequencesAndPackages();
      exampleDA.insert();
      exampleDA.update();
      exampleDA.select();
      exampleDA.delete();
      exampleDA.executePLSQL();

      jdbAccess.end();
    } catch (ApplicationException e) {
      System.out.println(e.toString());
    }
  }

  public void createOrReplaceTablesSequencesAndPackages() {
    try {
      Transaction t = DAFactory.getTransaction();
      t.begin();
      EmployeeDB empDB = new EmployeeDB(t);
      AddressDB addrDB = new AddressDB(t);
      EmpAddrDB empAddrDB = new EmpAddrDB(t);
      try {empDB.dropTable();} catch (ApplicationException e) {} // if it does not exist do nothing
      try {empDB.dropSequence();} catch (ApplicationException e) {} // if it does not exist do nothing
      try {addrDB.dropTable();} catch (ApplicationException e) {} // if it does not exist do nothing
      try {addrDB.dropSequence();} catch (ApplicationException e) {} // if it does not exist do nothing
      try {empAddrDB.dropTable();} catch (ApplicationException e) {} // if it does not exist do nothing
```

```
A console is not available.
```

And you see two example applications as Eclipse launchable Run/Debug applications:



Name: JDBaccess - EmployeeExampleDA

Project: JDBaccessExample

Main class: com.your_domain.your_product.main.ExampleDA

☐ Include libraries when searching for a main class

☐ Include inherited mains when searching for a main class

☐ Stop in main

The first example “EmployeeExampleDA” demontrates the use of data access operations such as insert, update, delete, select, function and procedure. To see what happens debug it and step by step through each operation. Some information is automatically logged to standard output.
The second example “EmployeeExampleGUI” shows the usage of JDBaccess in a typical three-layer-architecture with presentation, application and persistence layer. It starts a main window with a list of employees which can be edited.

You only need little database space (3 tables with some hundred rows of data).

To start the examples perform the following steps:

1. Create an example database user:
   - MySql:
     a) Create database:
     ```
     CREATE DATABASE employee;
     ```
     b) Create user with the appropriate permissions:
     ```
     GRANT SELECT,INSERT,UPDATE,DELETE,CREATE,DROP,CREATE ROUTINE,ALTER
     ROUTINE,EXECUTE
     ON employee.*
     TO 'scott'@'%'
     IDENTIFIED BY 'tiger';
     ```
   - Oracle:
     a) Create user with the appropriate permissions:
     ```
     CREATE USER joe
     IDENTIFIED BY "tiger"
     DEFAULT TABLESPACE users
     ```

```
        TEMPORARY TABLESPACE temp
        PROFILE DEFAULT;

        GRANT CONNECT TO joe;
        GRANT RESOURCE TO joe;
```

2. Provide your data source settings in: com/your_domain/your_product/main/ExampleJDBaccess.java: change the first 6 instance variables: dataSourceName, dbName, dbHost, dbPort, dbUser, dbUserPW) to appropriate values.
3. Compile the sources.
4. Start "EmployeeExampleDA" or "EmployeeExampleGUI" in run or debug mode.

4 Working with JDBaccess

For working with JDBaccess the following operations are important. For a deeper insight we propose to have a look at the two example applications in the source code delivered with your JDBaccess installation.

4.1 Initializing a JDBaccess application

First the data source is defined:

```
DataSource ds = new DataSource(“dsName”, “dbUser”, “dbUserPW”,
            “dbServiceName”, “dbPort”, “dbHostName”);
```

If necessary, connection init commands can be added. For example for MySql you could add the following command:

```
ds.addConnectionInitSqlString(“set @hostname = 'pc4711'”);
```

Or for Oracle you could add the following command

```
ds.addConnectionInitSqlString(“alter session set current_schema = test”);
```

Finally JDBaccess is started:

```
JDBaccess.setDataSource(ds);
JDBaccess.start();
```

That’s all. Connection and statement pools have been initialized and you are now able to fully perform all database access operations.

You can also see this initialization process in class “ExampleJDBaccess” in package “com.your_domain.your_product.main”.

4.2 Transaction

A transaction is similar to a database session. You start a transaction (get a separate database connection), perform a sequence of database access operations (statements) on it and if no error has occurred you commit it else you rollback it. Finally you end your transaction.

You start a transcation by:

```
Transaction t = DAReader.getTransaction();
t.begin();
```

You commit a transcation by:

```
t.commit();
```

You rollback a transcation back by:

```
t.rollback();
```

You end a transcation by:

```
t.end();
```

If you begin a transaction you should not forget the commit and end of that transaction so that the underlying database connection can be reused.

With MySql to achieve such transaction handling JDBaccess internally sets the transaction isolation level from "repeatable read" to "read committed". Also you should activate the storage engine “InnoDB” (see chapter 7.1 for details).

4.3 Insert

An Insert object automatically fetches sql types from the underlying table before it automatically inserts all transfer object field values according to the appropriate type. Id field values can be generated automatically by sequence numbers if the sequence name is defined in the transfer object class. All transfer objects are inserted batched because of performance reasons.

You insert rows by:

```
try {
  Insert insert = DAReader.getInsert(t);
  Row row1 = new Row();
  row1.setValueModified(“id”, new Long(4711));
  row1.setValueModified(“name”, “yourName”);
  Row row2 = new Row();
  row2.setValueModified(“id”, new Long(4712));
  row2.setValueModified(“name”, “yourName”);
  ArrayList transferObjects = new ArrayList();
  transferObjects.add(row1);
  transferObjects.add(row2);
  insert.setPackage(„yourPackage“);
  insert.setName(“yourName”);
  insert.setTOs(transferObjects);
  ArrayList createdTransferObjects = insert.execute();
  insert.end();
} catch (ApplicationException e) {
  t.rollback();
  ...
}
```

Or you define your insert operation in a data access file (da.xml) and execute it by:

```
try {
  Insert insert = DAReader.getInsert(t, „yourType“, „yourName“, yourStartClass);
  insert.setTO(yourTO);
  insert.execute();
  insert.end();
} catch (ApplicationException e) {
  t.rollback();
  ...
}
```

Or you define the sql code in your insert object by:

```
try {
  Insert insert = DAReader.getInsert(t);
  insert.setPackage(„yourPackage“);
  insert.setName(“yourName”);
  insert setSql(“insert into yourTable (id, name) values (?,?)”);
  insert.setParameter(“id” new Long(4711));
  insert.setParameter(“name” “yourName”);
  insert.execute();
  insert.end();
} catch (ApplicationException e) {
  t.rollback();
  ...
}
```

Or you insert transfer objects. If you define a sequence name in your transfer object, then this sequence name is used for automatically generating a new value in the id field of your transfer object:

```
try {
  Employee emp = new Employee();
  emp1.setValueModified("name", "tim");
  BigDecimal timSal = BigDecimal.valueOf(5000002, 2);
  emp.setValueModified("salary", timSal);
  emp.setValueModified("description", "description .....");
  emp.setValueModified("picture", picture);
  emp.setValueModified("creation", now);
  emp.setValueModified("modification", now);
  Insert insert = DAReader.getInsert(t);
  insert.setTO(emp);
  insert.execute();
  insert.end();
} catch (ApplicationException e) {
  t.rollback();
  ...
}
```

4.4 Update

You update a row (transfer object Employee) by id by:

```
try {
  Update update = DAFactory.getUpdate(t);
  Employee emp = new Employee();
  emp.setId(new Long(4711));
  emp.setValueModified("salary", new BigDecimal(20000));
  update.setTO(emp);
  update.execute();
  update.end();
} catch (ApplicationException e) {
  t.rollback();
  ...
}
```

Or you define your update operation in a data access file (da.xml) and execute it by:

```
try {
  Update update = DAFactory.getUpdate(t, "employee", "updateSalaryById", EmployeeDao.class);
  ArrayList values = new ArrayList();
  values.add(new BigDecimal(20000));
  values.add(new Long(4711));
  update.setParameters(values);
  update.execute();
  update.end();
} catch (ApplicationException e) {
  t.rollback();
  ...
}
```

Or you define the sql code in your update object by:

```
try {
  Update update = DAFactory.getUpdate(t);
  String sql = "update employee set salary = ? where id = ?";
  update.setSql(sql);
  ArrayList values = new ArrayList();
  values.add(new BigDecimal(20000));
  values.add(new Long(4711));
  update.setParameters(values);
  update.execute();
  update.end();
} catch (ApplicationException e) {
  t.rollback();
  ...
}
```

Or you update rows by a where condition (here name) by:

```
try {
  Update update = DAFactory.getUpdate(t);
  Employee empPeter = new Employee();
  empPeter.setName("peter");
  empPeter.setValueModified("salary", new BigDecimal(20000));
  ComparisonCondition nameEqualsComparison =
    new ComparisonCondition("name", ComparisonCondition.EQUALS, null);
  LogicalCondition nameEqualsCondition =
    new LogicalCondition(nameEqualsComparison, LogicalCondition.EMPTY, null);
  empPeter.setWhereCondition(nameEqualsCondition);
  update.setTO(empPeter);
  update.execute();
  update.end();
} catch (ApplicationException e) {
  t.rollback();
  ...
}
```

4.5 Delete

You delete a row (transfer object Employee) by id by:

```
try {
  Delete delete = DAFactory.getDelete(t);
  Employee emp = new Employee();
  emp.setId(new Long(4711));
  delete.setTO(empDao);
  delete.execute();
  delete.end();
} catch (ApplicationException e) {
  t.rollback();
  ...
}
```

Or you delete rows by a where condition (here name) by:

```
try {
  Delete delete = DAFactory.getDelete(t);
  Employee empPeter = new Employee();
  empPeter.setName("peter");
  ComparisonCondition nameEqualsComparison =
    new ComparisonCondition("name", ComparisonCondition.EQUALS, null);
  LogicalCondition nameEqualsCondition =
    new LogicalCondition(nameEqualsComparison, LogicalCondition.EMPTY, null);
  empPeter.setWhereCondition(nameEqualsCondition);
  delete.setTO(empPeter);
  delete.execute();
  delete.end();
} catch (ApplicationException e) {
  t.rollback();
  ...
}
```

4.6 Select

You select all rows by:

```
try {
  Select select = new Select();
  select.setSql("select * from employee");
  Result result = select.execute();
  ArrayList list = result.getAllElements();
  long count = result.getSize();
  select.end();
} catch (ApplicationException e) {
  ...
}
```

Or you select some rows of type Employee with where condition (own sql) and with order by clause by:

```
try {
  Select select = DAFactory.getSelect();
  String sql = "select * from employee where salary < ?";
  select.setSql(sql);
  select.setResultType(Employee.class);
  ArrayList params = new ArrayList();
  params.add(new Long(50000));
  select.setParameters(params);
  ArrayList orderBy = new ArrayList();
  orderBy.add("salary desc");
  select.setOrderBy(orderBy);
  Result result = select.execute();
  ArrayList list = result.getAllElements();
  select.end();
} catch (ApplicationException e) {
  t.rollback();
  ...
}
```

4.7 Procedure

You execute a procedure (see EmployeeDaoImpl in your example sources for the database code) by:

```
try {
  Procedure procedure = DAReader.getProcedure(t);
  ArrayList params = new ArrayList();
  params.add(new Integer(4711));
  ArrayList outputParamTypes = new ArrayList();
  outputParamTypes.add("varchar");
  outputParamTypes.add("numeric");
  outputParamTypes.add("cursor");
  outputParamTypes.add("cursor");
  procedure.setProcedureName("prc_addSalaryPercent");
  procedure.setModuleName("pkg_employee");
  procedure.setParameters(params);
  procedure.setOutputParamTypes(outputParamTypes);
  ArrayList outputParams = (ArrayList) procedure.execute();
  String success = (String) outputParams.get(0);
  // first result: employees before execution of procedure
  CallResult result1 = procedure.getResult(1);
  ArrayList elems1 = result1.getAllElements();
  // second result: employees after execution of procedure
  CallResult result2 = procedure.getResult(2);
  ArrayList elems2 = result2.getAllElements();
  // count of all updated rows of this procedure
  Long updateCount = procedure.getSize();
  procedure.end();
} catch (ApplicationException e) {
  t.rollback();
  ...
}
```

4.8 Function

You execute a function (see EmployeeDaoImpl in your example sources for the database code) by:

```
try {
  Function function = DAReader.getFunction(t);
  function.setFunctionName("fnc_addSalaryPercent");
  function.setModuleName("pkg_employee");
  ArrayList params = new ArrayList();
  params.add(new Integer(4711));
  function.setParameters(params);
  String outputParamType = "numeric";
  function.setOutputParamType(outputParamType);
  Object addedSalary2 = function.execute();
  function.end();
} catch (ApplicationException e) {
  t.rollback();
  ...
}
```

4.9 Result and CallResult

A result object (interface "Result") is returned by execution of a select object. In a result you have methods for getting elements (getAllElements, getNextElements, getPreviousElements, getFirstElement), for getting/setting a position (getPosition, setPosition), for setting the page size (setPageSize) and for reading large objects (setReadLobsFull). A result object automatically fetches result set meta data such as sql type, precision and scale and uses it to fill the result elements (row objects or data access objects) accordingly.

A call result (interface "CallResult") can be returned by a procedure. It is similar to a selection result but some methods such as page up and setting a position are not allowed.

Please have a look at the interface classes Result and CallResult to see all available methods.

4.10 ApplicationException

If an application error such as incorrect sql code appears an ApplicationException is thrown. All exceptions have a type, a number, a message string, a cause string and action instructions which can be accessed by getter methods.

4.11 Ending a JDBaccess application

You end your JDBaccess application (see ExampleJDBaccess.java) by calling:

```
      JDBaccess.end();
```

All open database connections and statements are closed.

4.12 Data access files

Following data access file defines some typical types of data access operations:

```
<?xml version="1.0" encoding="ISO-8859-1"?>
<!DOCTYPE da SYSTEM "com/jdbaccess/da/da.dtd">
<da>
  <insert>
    <object>employee</object>
    <name>insertWithoutLobs</name>
    <sql>insert into employee(id,name,salary,creation,modification) values(?,?,?,?,?)</sql>
  </insert>
  <update>
    <object>employee</object>
    <name>updateSalaryById</name>
    <sql>update employee set salary = ? where id = ?</sql>
  </update>
  <select>
    <object>employee</object>
    <name>all</name>
    <sql>select * from employee</sql>
    <sql-count>select count(id) from employee</sql-count>
  </select>
  <procedure>
    <object>employee</object>
    <name>prc_addSalaryPercent</name>
    <module-name>pkg_employee</module-name>
    <proc-name>prc_addSalaryPercent</proc-name>
    <output-params>
      <param>varchar</param>
      <param>numeric</param>
      <param>cursor</param>
      <param>cursor</param>
    </output-params>
  </procedure>
  <function>
    <object>employee</object>
    <name>fnc_addSalaryPercent</name>
    <module-name>pkg_employee</module-name>
    <func-name>fnc_addSalaryPercent</func-name>
    <output-params>
      <param>numeric</param>
    </output-params>
  </function>
  <procedure>
    <object>employee</object>
    <name>prc_insert</name>
    <sql>
      begin
```

```
        insert into employee (id, name, salary, creation, modification)
        values (seq_employee.nextval,?,?,?,?)
        returning id into ?;
      end;
    </sql>
    <output-params>
      <param>bigint</param>
    </output-params>
  </procedure>
</da>
```

4.13 Typical programming errors

Don’t forget to end an operation after you have used it:

1. When you have started a transaction you should not forget to commit and to end that transaction, so that the underlying database connection can be reused. In a typical GUI application a main transaction is started at the beginning of the application. This transaction is used by all application data access operations. At some points, as definded in the business logic, the application commits this transaction. At the end of the application the main transaction is ended.
2. When you have executed a data access object you should not forget to end that object so that the JDBC statement object (“open cursor”) can be reused.

Don’t forget to start a transaction with begin() if you want to use that transaction in data access operations.

When you have started JDBaccess (“JDBaccess.start()”) at the beginning of your application you should not forget to end JDBaccess (“JDBaccess.end()”) at the end of your application.

When you execute data access operations you have to surround that with try/catch-blocks. When an ApplicationException is thrown (e.g. your sql code is wrong) you should react in an appropriate way (e.g. rollback the underlying transaction and log a message to an error file). The exception also has an error type (e.g. fatal) which can be used in your application.

5 Designing your application

You should use JDBaccess in a typical two-layer-architecture with a rich client and a database server. But within your client JDBaccess allows you to develop your application in a typical three-layer-architecture with presentation, application and persistence layer. JDBaccess is the software between your persistence layer in the client and the JDBC driver for your database server.

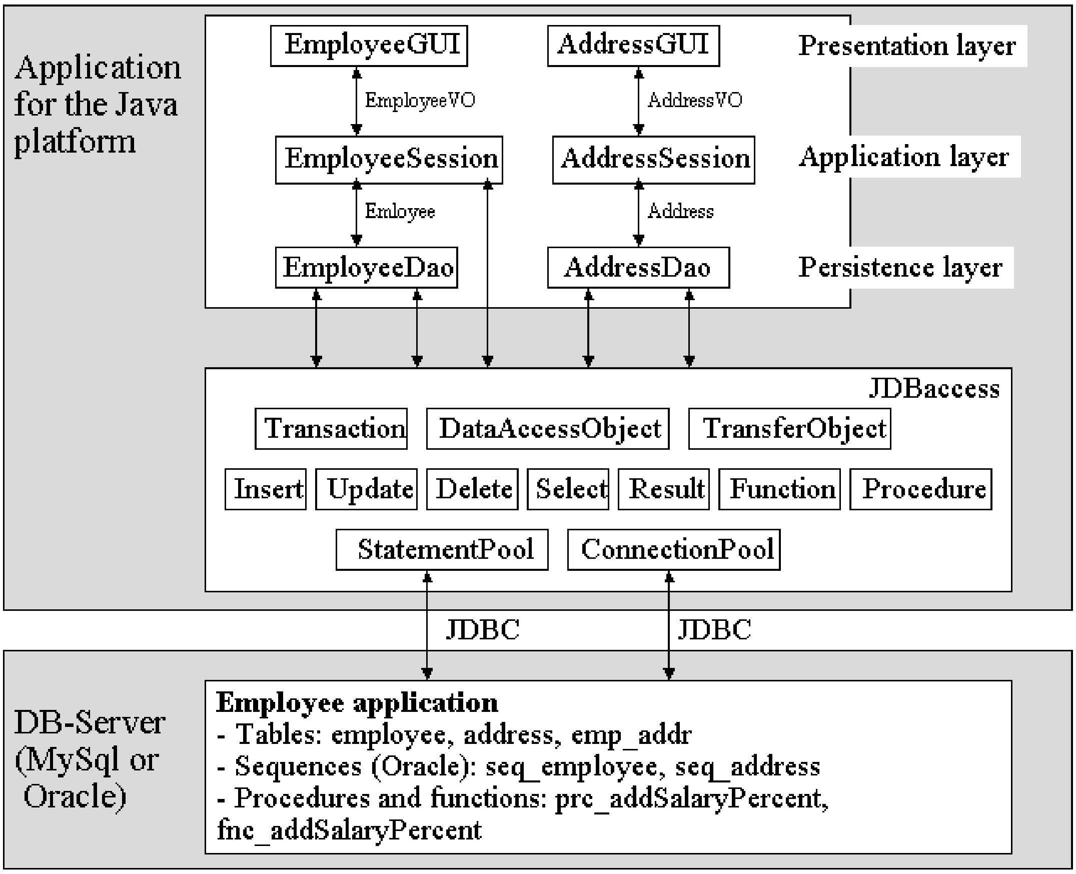


Figure 1: JDBaccess architecture of the employee example

5.1 Persistence layer

A persistence layer defines the database logic of your application.

JDBaccess supports the data access object pattern (see java.sun.com/blueprints/corej2eepatterns/Patterns/DataAccessObject.html).

You should have a look at the following packages in your JDBaccess installation for example classes:

- com.your_domain.your_product.address.dao
- com.your_domain.your_product.employee.dao

All data access object implementation classes should be subclasses of class DataAccessObjectImpl and all data access object interface classes should be subclasses of interface DataAccessObject which are provided in your JDBaccess library.

5.2 Application layer

An application layer defines all business logic of your application.

You should have a look at the following package in your JDBaccess installation for an example class:

- com.your_domain.your_product.employee.session

5.3 Presentation layer

A presentation layer defines the graphical user interface of your application.

You should have a look at the following package in your JDBaccess installation for example classes:

- com.your_domain.your_product.employee.gui

5.4 Transfer objects

Transfer objects are objects for exchanging data between layers. Value objects are used to exchange data between presentation and application layer. Transfer objects are used to exchange data between application layer and persistence layer.

JDBaccess supports the transfer object pattern (see java.sun.com/blueprints/corej2eepatterns/Patterns/TransferObject.html).

You should have a look at the following packages in your JDBaccess installation for example classes:

- com.your_domain.your_product.address.transfer
- com.your_domain.your_product.employee.transfer

All transfer object classes should be subclasses of class TransferObject which is provided in your JDBaccess library.

6 Mapping SQL types to types of the Java programming language

JDBaccess maps SQL types to types of the Java programming language and vice versa in conformity with the Sun JDBC standard (see Mapping SQL and Java Types, Sun JDBC, java.sun.com/j2se/1.4.2/docs/guide/jdbc/getstart/mapping.html).

6.1 Setting parameters/values

Following mapping for setting parameters is supported.

Java type	**SQL type**	**Oracle type**	**MySql type**	**JDBC standard**	**Comments**
Boolean	BOOLEAN BIT	n/a	BOOLEAN BIT	Yes	Oracle: as a workaround, you can create a column of type NUMBER(1) and set the appropriate value (0 = false; 1 = true)
String	VARCHAR CHAR CLOB LONGVARCHAR	VARCHAR2 CHAR CLOB LONG	VARCHAR CHAR TINYTEXT TEXT MEDIUMTEXT LONGTEXT	Yes	Depending on SQL type of table column; CLOB is not standard SQL type and is added, so that strings can be inserted/updated directly
BigDecimal	DECIMAL NUMERIC	NUMBER	DECIMAL NUMERIC	Yes	
Integer	INTEGER	NUMBER	TINYINT SMALLINT MEDIUMINT INTEGER BIGINT YEAR	Yes	
Long	BIGINT	NUMBER	MEDIUMINT INTEGER BIGINT YEAR	Yes	
Float	REAL	NUMBER	FLOAT	Yes	
Double	DOUBLE	NUMBER	DOUBLE	Yes	
byte[]	BINARY VARBINARY BLOB LONGVARBINARY	RAW BLOB LONG RAW	BINARY VARBINARY TINYBLOB BLOB MEDIUMBLOB LONGBLOB	Yes	Depending on SQL type of table column; BLOB is not standard SQL type and is added, so that byte arrays can be inserted/updated directly
java.util.Date	TIMESTAMP	DATE TIMESTAMP	DATE DATETIME TIME TIMESTAMP	No	Added
java.sql.Date	DATE	DATE TIMESTAMP	DATE DATETIME TIME TIMESTAMP	Yes	Usable but you should better use java.util.Date
java.sql.Time	TIME	DATE	DATE	Yes	Usable but vou should

		TIMESTAMP	DATETIME TIME TIMESTAMP		better use java.util.Date
java.sql.Timestamp	TIMESTAMP	DATE TIMESTAMP	DATE DATETIME TIME TIMESTAMP	Yes	Usable but you should better use java.util.Date
Clob	CLOB	CLOB	CLOB	Yes	Not needed: handled with String
Blob	BLOB	BLOB	BLOB	Yes	Not needed: handled with byte[]

Following types of the Java programming language are not supported:

Java type	**SQL type**	**Oracle type**	**MySql type**	**JDBC standard**	**Comments**
Array	ARRAY	VARRAY	n/a	Yes	Object relational: not supported yet
Struct	STRUCT	OBJECT	n/a	Yes	Object relational: not supported yet
Ref	REF	REF	n/a	Yes	Object relational: not supported yet
java.net.URL	DATALINK	n/a	n/a	Yes	no corresponding Oracle/MySql type
Java class	JAVA_OBJECT	n/a	n/a	Yes	no corresponding Oracle/MySql type

Remark: For setting sql paramters you have the “normal” database limitations. For example it is not possible to select by a longvarbinary value.

6.2 Select: Getting values

SQL type	**Oracle type**	**MySql type**	**Java type**	**JDBC standard**	**Comments**
BOOLEAN	n/a	BOOLEAN	Boolean	Yes	Oracle: as a workaround, you can create a column of type NUMBER(1) and set the appropriate value (0 = false; 1 = true)
BIT	n/a	BIT	Boolean	Yes	Oracle: as a workaround, you can create a column of type NUMBER(1) and set the appropriate value (0 = false; 1 = true)
CHAR	CHAR	CHAR	String	Yes	
VARCHAR	VARCHAR2	VARCHAR	String	Yes	
NUMERIC	NUMBER	NUMERIC	BigDecimal	Yes	

DECIMAL	NUMBER	DECIMAL	BigDecimal	Yes	
TINYINT	n/a	TINYINT	Integer	Yes	Oracle recommends to use a column of type NUMBER(3) instead
SMALLINT	NUMBER	SMALLINT	Integer	Yes	
INTEGER	NUMBER	INTEGER	Integer	Yes	
BIGINT	n/a	BIGINT	Long	Yes	Oracle recommends to use a column of type NUMBER instead
REAL	NUMBER REAL	n/a	Float	Yes	
FLOAT	NUMBER REAL	FLOAT	Float	Yes	
DOUBLE	NUMBER REAL	DOUBLE	Double	Yes	
TIME	n/a	TIME	java.util.Date	No	with date set to 1.1.1970 and time value (standard Java type is java.sql.Time); Oracle recommends to use a column of type DATE instead
DATE	DATE TIMESTAMP	DATE	java.util.Date	No	with date and time set to null values (standard Java type is java.sql.Date)
TIMESTAMP	DATE TIMESTAMP	DATETIME TIMESTAMP	java.util.Date	No	with date and time value (standard Java type is java.sql.Timestamp)
CLOB	CLOB	n/a	String	No	is read by streaming methods; in result you can specify to read the full value (standard Java type is Clob)
LONGVARCHAR	LONG	TINYTEXT TEXT MEDIUMTEXT LONGTEXT	String	Yes	
BLOB	BLOB	n/a	byte[]	No	byte[] is read by streaming methods; in result you can specify to read the full value (standard Java type is Blob)
BINARY	n/a	BINARY	byte[]	Yes	Oracle recommends to use BLOB, RAW or LONG RAW instead
VARBINARY	RAW	VARBINARY	byte[]	Yes	
LONGVARBINARY	LONG RAW	TINYBLOB BLOB MEDIUMBLOB LONGBLOB	byte[]	Yes	

With “yourSelect.setResultFieldType(yourClass)” the default Java type of the value can further be casted to the given Java type (yourClass). At this time BigDecimal, Short, Integer, Long, Float, Double can be casted into each other (e.g. BigDecimal into Double etc.)

Following SQL types are not supported:

SQL type	Java type	JDBC standard	Comments
DISTINCT	Object type of underlying type	Yes	Object relational: not supported yet
ARRAY	Array	Yes	Object relational: not supported yet
STRUCT	Struct or SQLData	Yes	Object relational: not supported yet
REF	Ref	Yes	Object relational: not supported yet
DATALINK	java.net.URL	Yes	no corresponding Oracle/MySql type
JAVA_OBJECT	underlying Java class	Yes	no corresponding Oracle/MySql type

6.3 Procedure and Function: Setting output parameter types and getting output parameters

Before getting output parameters of functions and procedures, output parameter types have to be set. You set output parameter types with the method "setOutputParamTypes" in Interface "Procedure" as an ArrayList of Strings of SQLTypes (see above the example procedure).

In MySql functions it is not needed to set the output parameter type. Also if MySql procedures delivers selection results as "open cursors" they must not be set as output paramter types but could be accessed in the same way as in Oracle with getResult(n).

Following output parameter types are supported (first column "SQL type" is the sql type which is set as output parameter type and column "Java type" is the Java class which you get in the output paramter after execution of the procedure):

SQL type	Oracle type	MySql type	Java type	Comments
BOOLEAN	n/a	BOOLEAN	Boolean	Oracle: as a workaround, you can create a wrapper PL-SQL-function- or procedure that returns a type, which is supported by Oracle such as TINYINT or INTEGER (with 0 = false; 1 = true)
BIT	n/a	BIT	Boolean	Oracle: as a workaround, you can create a wrapper PL-SQL-function- or procedure that returns a type, which is supported by Oracle such as TINYINT or INTEGER (with 0 = false; 1 = true)
CHAR	CHAR	CHAR	String	
VARCHAR	VARCHAR2	VARCHAR	String	
NUMERIC	NUMBER	NUMERIC	BigDecimal	
DECIMAL	NUMBER	DECIMAL	BigDecimal	
TINYINT	NUMBER INTEGER	TINYINT	Integer	

SMALLINT	NUMBER INTEGER	SMALLINT	Integer	
INTEGER	NUMBER INTEGER	INTEGER	Integer	
BIGINT	BIGINT	BIGINT	Long	
REAL	NUMBER FLOAT	n/a	Float	
FLOAT	NUMBER FLOAT	FLOAT	Double	
DOUBLE	NUMBER FLOAT	DOUBLE	Double	
TIME	n/a	TIME	java.util.Date	with date set to 1.1.1970 and time value (standard Java type is java.sql.Time); Oracle recommends to use a column of type DATE instead
DATE	DATE TIMESTAMP	DATE	java.util.Date	With date and time value
TIMESTAMP	DATE TIMESTAMP	DATETIME TIMESTAMP	java.util.Date	With date and time value
CLOB	CLOB	n/a	String	is read by streaming methods; in procedure or function you can specify to read the full value
LONGVARCHAR	LONG	TINYTEXT TEXT MEDIUMTEXT LONGTEXT	String	
BLOB	BLOB	n/a	byte[]	is read by streaming methods; in procedure or function you can specify to read the full value
BINARY	n/a	BINARY	byte[]	Oracle: use BLOB or VARBINARY or LONGVARBINARY
VARBINARY	RAW	VARBINARY	byte[]	
LONGVARBINARY	LONG RAW	TINYBLOB BLOB MEDIUMBLOB LONGBLOB	byte[]	
CURSOR	CURSOR	n/a	CallResult	Supported in Oracle, in MySql not needed; with “getResult(n)” the appropriate result is fetched

Following SQL types are not supported yet:

SQL type	**Java type**	**Comments**
DISTINCT	Object type of underlying type	Object relational: not supported yet
ARRAY	Array	Object relational: not supported yet
STRUCT	Struct or SQLData	Object relational: not supported yet
REF	Ref	Object relational: not supported yet

DATALINK	java.net.URL	no corresponding Oracle/MySql type
JAVA_OBJECT	underlying Java class	no corresponding Oracle/MySql type

6.4 Oracle/MySql standard SQL types

Standard SQL type	Oracle type	MySql type
ARRAY	VARRAY	n/a
BIGINT	n/a	BIGINT
BINARY	n/a	BINARY
BIT	n/a	BIT
BLOB	BLOB BFILE	TINYBLOB BLOB MEDIUMBLOB LONGBLOB
BOOLEAN	n/a	BOOLEAN
CLOB	CLOB	TINYTEXT TEXT MEDIUMTEXT LONGTEXT
CHAR	CHAR	CHAR
DATALINK	n/a	n/a
DATE (without time)	n/a	DATE
DECIMAL	NUMBER	DECIMAL
DISTINCT	n/a	n/a
DOUBLE	FLOAT NUMBER	DOUBLE
FLOAT	FLOAT NUMBER	FLOAT
INTEGER	NUMBER	INTEGER
JAVA_OBJECT	n/a	n/a
LONGVARBINARY	LONG RAW	TINYBLOB BLOB MEDIUMBLOB LONGBLOB
LONGVARCHAR	LONG	TINYTEXT TEXT MEDIUMTEXT LONGTEXT
NUMERIC	NUMBER	NUMERIC
REAL	FLOAT, NUMBER	n/a
REF	REF	n/a
SMALLINT	NUMBER	SMALLINT
STRUCT	OBJECT	n/a
TIME	n/a	n/a

TIMESTAMP (date and time)	DATE, TIMESTAMP	DATETIME TIMESTAMP
TINYINT	n/a	TINYINT
VARBINARY	RAW	VARBINARY
VARCHAR	VARCHAR2	VARCHAR

Additional Oracle/MySql SQL types which are not part of the standard yet (e.g. ROWID, YEAR, INTERVAL YEAR, CURSOR, SET, ENUM, etc.) are not listed here.

6.5 Java: Value limitations in standard SQL types

Standard SQL type	**value limitation**
ARRAY	
BIGINT	64 bit signed integer: -9223372036854775808 to 9223372036854775807
BINARY	254 bytes
BIT	0, 1
BLOB	
BOOLEAN	true, false
CHAR	254 (8-bit characters)
CLOB	
DATALINK	
DATE	year, month, day
DECIMAL	15 for precision (total number of digits) and for scale (number of digits after the decimal point)
DISTINCT	
DOUBLE	15 bits of mantissa (fractional part)
FLOAT	15 bits of mantissa (fractional part)
INTEGER	32 bit signed integer: -2147483648 to 2147483647
JAVA_OBJECT	
LONGVARBINARY	1 GB
LONGVARCHAR	1 GB (8-bit characters)
NUMERIC	15 for precision (total number of digits) and for scale (number of digits behind the decimal point)
REAL	7 bits of mantissa (fractional part)
REF	
SMALLINT	16 bit signed integer: -32768 to 32767
STRUCT	
TIME	hours, minutes, seconds

TIMESTAMP	year, month, day, hours, minutes, seconds, nanoseconds
TINYINT	8 bit signed or unsigned integer: -128 to 127 (8 bit signed) or 0 to 254 (8 bit unsigned)
VARBINARY	254 bytes
VARCHAR	254 bytes (8-bit characters)

7 Database specialties

7.1 MySql transactions with InnoDB

The MySql standard installation works with the “MyIsam” storage engine which does not support a standard transaction handling with commit and rollback. But if you want to achieve such a transaction handling you should activate the storage engine “InnoDB”. InnoDB provides MySQL with a transaction-safe storage engine that has commit, rollback, and crash recovery capabilities. InnoDB does locking on the row level and also provides an Oracle-style consistent non-locking read in select statements. For activating this non-locking read JDBaccess internally sets the transaction isolation level from "repeatable read" to "read committed" (“set session transaction isolation level read committed”). Each select even within the same transaction reads the data which is committed at that time. These features increase multi-user concurrency and performance. InnoDB is included in binary MySql distributions by default.
To activate the storage engine “innodb” you have to specify it in your create table statement:

```
  create table (id int not null, name varchar(300), ... ) engine = innodb
```

7.2 Oracle nchar

If you want to write Unicode characters to Oracle nchar, nvarchar2 and nclob columns you have to perform 2 steps:

- Oracle database:
  set NLS_NCHAR_CHARACTERSET to a Unicode value (e.g. AL16UTF16)
- Java application initialization:
  System.setProperty("oracle.jdbc.defaultNChar", "true");

8 Limitations

With JDBaccess following limitations exist:

- JDBacces supports exactly one data source with its dependent connection and statement pool. We think that for most applications this is sufficient.
- JDBaccess is not applicable in application server environments with own data source and connection pool implementations
- Updateable result sets are not supported

With the Oracle JDBC thin driver following limitations exist:

- update of longvarchar/longvarbinary columns: In versions <= 10.2.0.1. it is not possible to update more than one row at the time if the value which has to be updated is bigger than 2000/4000 characters/bytes.
- setting parameters of type longvarchar or longvarbinary in PL-SQL functions and procedures: values must be smaller than 32000 bytes.
- in versions <= 9.2.0.5 problems exist with the writing of CLOBS such as inserting bigger values by a pl-sql procedure
- If PL-SQL calls are performed in SQL-select statements Oracle’s JDBC driver always returns 0 as precision and scale in its result set meta data implementation. In JDBaccess these are set to correct values.
-

With the MySql JDBC driver Connector/J following limitations exist:

- selection of all rows of a table (ps.executeQuery()) which contains more than 1000 rows each with 1 MB of a LONGTEXT value

9 Next version

In the next release it is planned to support more than one data source with their connection and statement pools. Also, it is planned to put JDBaccess to the server side. And it is planned to support further useful JDBC functionality such as named parameters in functions and procedures.
Further on it is planned to support additional RDBMS with their JDBC drivers such as DB2 UDB, MS-SQL-Server and MaxDB.

10 Performance

JDBaccess can't be faster than JDBC itself, but it performs important operations by better usage of the JDBC methods. The average performance and throughput of applications with JDBaccess is much better than applications working with JDBC directly.

10.1 MySql

We did a performance test on the following machines:
Java-Client

- Hardware: AMD-Athlon-Computer (one AMD-Athlon processor 1,4 Ghz, 528 MB RAM)
- Operational system: Windows 2000 Professional
- Java platform: J2SE 5.0
- JDBC-driver: MySql-Connector/J 5.0.3

DB-Server

- Hardware: AMD-64-Computer (one AMD-64 3500+ processor, 1 GB RAM)
- Operational system : Linux (Suse Linux Enterprise Server 9)
- RDBMS: MySql 5.0.22 Standard

The test gave the following results (all operations are from the example employee application provided with JDBaccess):

Insert (insert an employee)

- 1000 times (without LOBs, batched): 1,3 sec
- 1000 times (without LOBs, not batched): 1,6 sec
- 1000 times (LONGTEXT with 1000, 10000, 100000, 1000000 chars, batched): 1,0 sec, 3,6 sec, 27,5 sec, 560,5 sec
- 1000 times (LONGBLOB with 1000, 10000, 100000, 1000000 bytes, batched): 0,8 sec, 3,4 sec, 22,2 sec, 386,1 sec

Update (update an employee by id)

- 1000 times (without LOBs): 2,3 sec
- 1000 times (LONGTEXT with 1000, 10000, 100000, 1000000 chars, LONGBLOB with 1000, 10000, 100000, 1000000 bytes): 2,5 sec, 5,5 sec, 33,0 sec, 623,4 sec

Delete (delete an employee by id)

- 1000 times: 1,0 sec

Select (select all employees)

- 1000 times: 1,1 sec

Result (get employees or rows)

- 1000 employees(without LOBs): 0,3 sec
- 1000 rows(LONGTEXT with 1000, 10000, 100000, 1000000 chars): 1,2 sec, 1,8 sec, 12,0 sec, 181,6 sec
- 1000 rows(LONGBLOB with 1000, 10000, 100000, 1000000 bytes): 0,7 sec, 0,7 sec, 0,8 sec, 1,0 sec

Function (fnc_addSalaryPercent)

- 1000 times: 8,8 sec

Procedure (prc_addSalaryPercent)

- 1000 times: 9,8 sec

10.2 Oracle

We did a performance test on the following machines:

Java-Client

- Hardware: Intel-Computer (one Intel processor 3,2 Ghz)
- Operational system: Windows 2000 Professional
- Java platform: J2SE 5.0
- JDBC-driver: Oracle thin driver 10.1.0.4

DB-Server

- Hardware: Intel-Computer (two Xeon processors 2,4 Ghz)
- Operational system: Windows 2000 Server
- RDBMS: Oracle 10.2.0.2 Enterprise

The test gave the following results (all operations are from the example employee application provided with JDBaccess):

Insert (insert an employee)

- 1000 times (without LOBs, batched): 1,0 sec
- 1000 times (without LOBs, not batched): 1,3 sec
- 1000 times (CLOB with 1000, 10000, 100000, 1000000 chars, batched): 0,9 sec, 16,8 sec, 39,1 sec, 213,8 sec
- 1000 times (BLOB with 1000, 10000, 100000, 1000000 bytes, batched): 0,75 sec, 16,8 sec, 39,1 sec, 244,5 sec

Update (update an employee by id)

- 1000 times (without LOBs): 1,55 sec
- 1000 times (CLOB with 1000, 10000, 100000, 1000000 chars, BLOB with 1000, 10000, 100000, 1000000 bytes): 1,8 sec, 35,5 sec, 85,9 sec, 457,2 sec

Delete (delete an employee by id)

- 1000 times: 0,8 sec

Select (select all employees)

- 1000 times: 1,0 sec

Result (get employees or rows)

- 1000 employees(without LOBs): 0,2 sec
- 1000 rows(CLOB with 1000, 10000, 100000, 1000000 chars): 1,3 sec, 3,4 sec, 18,0 sec, 159,8 sec
- 1000 rows(BLOB with 1000, 10000, 100000, 1000000 bytes): 0,8 sec, 3,4 sec, 13,4 sec, 112,1 sec

Function (fnc_addSalaryPercent)

- 1000 times: 12,5 sec

Procedure (prc_addSalaryPercent)

- 1000 times: 15,5 sec