ESPRIT SYSTEMS

Esprit Database Programming

How to use DBObjects as Persistency Layer for high performance Database Applications

June 2010
Rainer Büsch

Inhaltsverzeichnis

1 Introduction....5
1.1 About this document....5
1.1.1 What's covered by this document....5
1.1.2 What's not covered by this document....5
1.1.3 Presuppositions....5
1.2 Installing the software....5
1.2.1 Software prerequisites....5
1.2.2 Installing the software....5
1.2.3 Setting the classpath....5
1.3 What's a DBObject?....6
1.3.1 Object relational Mapping....6
1.3.2 Composed persistent objects....6
1.3.3 DBObjects for database views....7
2 Creating the Database....7
3 Creating DBObjects....7
3.1 How a DBObject looks like....8
3.1.1 Static methods....8
3.1.2 Instance level methods....8
3.1.3 Member variables....8
3.1.4 The master instance....9
3.2 Column type mapping....9
3.2.1 Available data types....9
3.2.2 Column type mapping by naming conventions....10
3.2.3 Using custom column type mapping 10
3.2.4 Using primary keys....10
3.2.5 Primary/foreign key faking....11
4 Connecting to a database....11
4.1 Database credentials....11
4.1.1 Creating a credential....11
4.1.2 Reading credentials from the commandline....12
4.1.3 Reading credentials from a file....12
4.1.4 Advanced Credential parameters....13
4.2 Establishing connections....13
4.2.1 Establishing a single database connection....13
4.2.2 Establishing a pool of database connections....13
4.2.3 Closing the database connection....14
5 Working with DBObjects....14
5.1 Selecting records from the database....14
5.1.1 Selecting a single record....14
5.1.2 Selecting multiple records by condition....15
5.1.3 Iterating through query results....15
5.1.4 Sorting records....16
5.1.5 Monitoring asynchroneous record loads....16
5.1.6 Using more complex SQL conditions....17
5.2 4.2 Inserting new records....17
5.2.1 How to instantiate and insert a record....17

5.2.2 Using the master instance for inserts....18
5.2.3 Automatic primary key generation....18
5.2.4 Using a primary key factory....18
5.2.5 Using a factory for setting default values on insert....19
5.3 Updating records....19
5.4 Deleting records....20
5.4.1 Deleting a single record....20
5.4.2 Deleting multiple records by condition....20
5.4.3 Primary-Key only selects....21
5.5 Forced insert and update....21
6 Running transactions....22
6.1 How to use transactions....22
6.2 Running nested transactions....22
6.3 Running multiple transactions in parallel....23
7 Multiple database connections....23
7.1 Connecting to multiple databases....23
7.2 Database cross copy example....23
7.3 Conversion between different table structures....24
8 Importing/Exporting data....25
8.1 Formatting output....25
8.1.1 Creating HTML output....25
8.1.2 Creating XML output....25
8.2 Loading/Unloading data with UNL....26
8.2.1 Writing UNL output....26
8.2.2 Reading UNL input....26
8.2.3 Handling UNL parsing errors....27
9 Special features....27
9.1 Comparing DBObjects....27
9.1.1 Checking equality of primary keys....27
9.1.2 Checking equality of attributes....27
9.1.3 Using Comparable implementation....28
9.2 Table oriented actions....28
9.2.1 Checking table existence....28
9.2.2 Creating the table....28
9.2.3 Dropping a table....28
9.2.4 Counting records in a table....29
9.3 Using database schemas....29
9.4 Storing zero and blank values....29
9.4.1 Storing empty strings....29
9.4.2 Storing numeric zero values....30
9.5 Logging....30
9.5.1 Using a custom Logger....30
9.5.2 Switching off messages....30
9.5.3 Printing different LogLevels....31
10 Example partlist program....31
11 General purpose database tools....32

11.1 Tools for executing SQL commands....32
11.1.1 DBExecute Tool....32
11.1.2 DBSelect Tool....32
11.2 Tools for Exporting/Importing data in UNL format....33
11.2.1 UnlExport Tool....33
11.2.2 UnlImport Tool....33
11.2.3 UnlExport Tool....34
11.3 Interactive Tools....34
11.3.1 TableEditTool....34
12 Additional information....34

1 Introduction

1.1 About this document

1.1.1 What's covered by this document

This manual describes the basic features of the *DBObject* based persistency solution provided by the the *EspritAppSuite* software. You will learn how to connect to a database and how to work with raw *DBObjects* for reading/writing data from/to the database.

1.1.2 What's not covered by this document

There are advanced features of the software which are *not* covered here:

- ➔ Using *DBObjects* for building composed persistent objects (*ComposedRecord* instances)
- ➔ Writing database applications using the graphical user interface API of the *EspritAppSuite*.
- ➔ Using DBObjects in a client-server environment with remote database access.

1.1.3 Presuppositions

The reader should be quite firm in standard Java coding and think object-oriented. Knowledge about multi-threading is also very helpful.

For all coding done in this manual we suppose that your CLASSPATH environment variable contains at least these additional entries:

- ➔ The jdbc driver jar-file of your database system vendor.
- ➔ The *EspritAppSuite.jar* file which resides in your <installDir>/lib directory.
- ➔ The base directory for finding the DBObject subclasses of your database.

1.2 Installing the software

1.2.1 Software prerequisites

You need to have installed the Java Runtime Environment 1.7x. To find out your installed java version type the following command on your terminal command line:

```
java -version
```

1.2.2 Installing the software

The *EspritAppSuite* software is shipped in a single zipped file called *EspritAppSuite.zip* which you have to extract into any target directory which we refer to as *ESPRIT_HOME* (we recommend *C:/Esprit/EspritAppSuite* as installation directory). If you haven't got a zip tool you can also use the jar command line tool which is included in the Java SDK installation:

```
mkdir C:\Esprit\EspritAppSuite
jar xvf EspritAppSuite.zip C:\Esprit\EspritAppSuite
```

In the target directory you will find a *Readme.txt* file that helps you further.

1.2.3 Setting the classpath

The most important thing for running the *EspritAppSuite* software (as for any other Java Program)

is the CLASSPATH environment variable beeing set correctly so that all necessary classes and resources can be found. The following excerpt of the *dbocompile.bat* file shows what settings you must have as a minimum.

```
set ESPRIT_HOME=C:\Esprit\EspritAppSuite
set CLASSPATH=%CLASSPATH%;%ESPRIT_HOME%\lib\EspritAppSuite.jar // Esprit software
set CLASSPATH=%CLASSPATH%;%ESPRIT_HOME%\lib\jdbcdriver.jar // !Your! JDBC Driver
set CLASSPATH=%CLASSPATH%;%YOURPROJECT%\classes    // Your compiled DBObject subclasses
```

The value of the *ESPRIT_HOME* variable must be adapted to your real installation directory. The *EspritAppSuite.jar* entry is obvious - it contains the whole *EspritAppSuite* software. The *jdbcdriver.jar* stands for the actual JDBC driver software of your particular database.

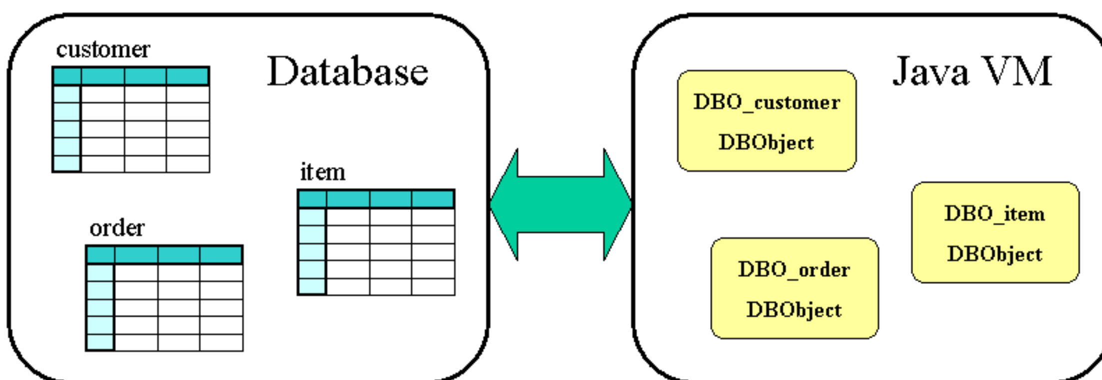
1.3 What's a DBObject?

A *DBObject* is a Java class, which models a database record. An instance of this class carries the information of the record in member variables and knows by itself how to read/write from/to the database (SELECT, INSERT, UPDATE, DELETE). The programmer uses a *DBObject* just like any other Java object and needs not care about database specifica. Any SQL statements are dynamically generated internally and not visible to the outside world (except in logging output).

1.3.1 Object relational Mapping

A database table is represented by a particular subclass of the *DBObject* superclass. The *DBOCompiler* generates the source code for such subclasses automatically derived from the meta-information found in the database. There is a simple naming convention: the Java class-name is equal to the table name just with a 'DBO_' prefix. A table called *customer* for example will be represented by a *DBObject* subclass called *DBO_customer.*

➔ Note that you can consider the DBObject class as a representation of the database table and the DBObject instance as a representation of a table record.



As the picture above indicates, you have got a simple *one to one* relationship between the tables in the database and the Java objects representing them. This simple object view may be good enough for most cases but note, that you are not restricted to this schema.

1.3.2 Composed persistent objects

Be aware that *DBObjects* can be used as building blocks for constructing composed persistent objects up to any complexity. How to do this is out of the scope of this document but you find examples in the free sample Project *EspritAppLab* within the package *de.esprit.applab.dbo.epart*.

1.3.3 DBObjects for database views

Not only database *tables* but also *views* can be compiled to a *DBObject*. A database *view* looks similiar to a table but acutally executes an underlying SQL statement when you select records from it. You can see *views* as an alternative way for constructing composed persistent objects. There is only one problem with views: they haven't got a primary key which is needed for record updates. In special cases a *view* would in fact be updatable if it only had a primary key (if for example a view does not contain joins). In such cases you can 'fake' a primary key as described later.

There is one major advantage in using *views* instead of composed persistent objects: it's underlying SELECT statement can be well tuned by the database administrator and thus will run with very good performance. Composed persistent objects in contrast are much more handy and flexible for the developer to work with.

2 Creating the Database

3 Creating DBObjects

The first step to working with *DBObjects* is of course to create them. You do this by running the *DBOCompiler* as a batch job. The *DBOCompiler* needs a *DBOTypeMapCredential* object for running, which contains all the information needed to create the *DBObjects*. In the package *de.esprit.workbench.dbo* of the free *EspritWorkbench* project you find the *EwbDBOCredential* which derives from *DBOTypeMapCredential* and adds the required compile information.

You need to specify a the following parameters for compiling DBObjects. Mandatory parameters are marked with an '*':

- *setTargetDir(File)* * Target output directory
  This is the base directory where the gernerated *DBObject* subclass is copied to, typically the source folder of your project. Note that below this target directory you will find the package hierarchy as subfolders.
- *setTargetPackage(String)* * Target package name
  This package name beeing used for the package statement in the gernerated *DBObject* subclass, i.e. *de.esprit.workbench.dbo*.
- *setUseTypeConventions(boolean)* Use DBO-conventions for column type mapping
  If *true* then the data type used for columns will be derived from the name of the column (only applies if otherwise no clear mapping is achieved). The column-type naming conventions are described below in this document.
- *setRebuildExistingDBOs(boolean)* Rebuild existing DBObjects only
  If *true* then only such DBObjects will be recreated, which already exist in the output directory. That way only those DBObjects are created which are actually needed. Otherwise DBObjects will be created for all tables and views found in the database.
- *setSchema(String)* Defines a database schema
  If a schema is defined, only those tables of this schema will be compiled. Otherwise only tables of the database internal default-schema are compiled.

➔ *addTableExcludePattern(String)* Exclude tables from compiling
Allows for defining a reqular expression. All tables that match this expression will be exclude from compilation. You may add multiple patters.

➔ *setTranslationAppKey(String)* Defines the application key for table/column translations
This allows for automatically translate table- and column-names in GUI input forms using the translation system of the Esprit software.

➔ *put(DBOColumnType, JavaColumnType)* Define a column-type explicitely
Allows for defining a column-type explicitely. For instance if you got a *CHAR* column named *was_easy* which you want to have represented as a Java *boolean* value in the according *DBObject*, you can achieve it with this method.

Refer to the *EspritWorkbench* example *EwbDBOCompile* for how to run the *DBOCompiler* with the *EwbDBOCredential*. And have a look to the freshly generated DBObjects in your target package.

➔ It is strongly recommended to recompile the DBObjects whenever the structure of the underlying database table has been changed.

3.1 How a DBObject looks like

As mentioned already each *DBObject* subclass that you have created with the *DBOCompiler* represents a table in your database. Instances of them represent the table records. Compiling the table *customer* for example results in a *DBObject* subclass called *DBO_customer.* Printing out the full *DBObject* source code would be quite lengthy, therefore you are encouraged to take some time for examining the Java source code of one you have created.

3.1.1 Static methods

In a particular *DBObject* subclass you will find a set of static methods which typically perform table oriented tasks. Methods like *getPrimaryKeyCount(), getPrimaryKeys()* etc. tell meta information about the underlying database table. Other methods like *createTable(), dropTable()* etc. perform table based actions in the database. There is also a set of static constants (in upper case letters) representing the names of the table columns (just table meta information).

3.1.2 Instance level methods

Furthermore you will find a set of instance level methods like *dbInsert(), dbUpdate(), dbDelete()* etc. which perform record based tasks. There is also a set of member variables that represent the record values. They are named exactly as the columns in your database table (in lower case letters and just preceeded by a '$' sign for making them unique) and have got a datatype that best matches the real SQL type definition (see column type mapping below).

Reading a record from the database basically means: instantiating your subclass and setting the member variables to values read from the database fields. Reading from the database requires an SQL statement of course, which is internally created by the *DBObject* itself. So this is what the *DBObject* superclass does for you: it hides the complexity of how to talk with databases. You won't need to deal with SQL any more.

3.1.3 Member variables

Note that the member variables are declared beeing private and thus you cannot access them directly. Rather you will find a *get$xxx()* and *set$xxx(...)* method for each of them in order to read or modify its value. Furthermore a *DBObject* has got an internal *isDirty()* flag that makes a guess whether your DBObject is in sync with the database or not. This flag is automatically set, whenever a *set$xxx(...)* method was called.

```
DBO_person p = new DBO_person(200);  // read record from the database
String lastName = p.get$LastName();  // read a value
p.set$LastName("Schroeder"); // modify a value, object is dirty now
```

Once you have read the *DBObject* from the database it exists in your memory and reflects the state of the record at the time it was read. If another user changes the record in the database you would not know. Therefore you should not rely too much on the *isDirty()* flag - it's just a guess (but usefull anyway). At any time you can refresh your object with the current database values using the *dbRehash()* call, which cleans the dirty flag.

Note that in other persistency solutions calling a getter/setter method actually results in a read/write from/to the database - which occasionally may result in serious performance problems. *DBObjects* leave it up to you when database access happens - you call *dbRehash()* or *dbUpdate()* when you want it to happen, giving you better control about performance.

3.1.4 The master instance

Each *DBObject* subclass contains an internal singleton instance of itself, which is called the *master instance*. You get a reference to the master instance as shown in this example:

```
DBO_customer master = DBO_customer.getMaster();
```

If you have a look to the *DBO_customer.java* source code you will find that most static methods just delegate to the master instance, so it is mainly used internally. But you may also 'abuse' it for your purposes. You will see a few examples later in this document.

3.2 Column type mapping

One of the main problems in object relational mapping is how to find the Java datatype that matches best your database datatype. For example in an *Oracle* database every datetime value is stored as a DATE data type. But your application may want to see it as a Java *Date*, *Time* or *Timestamp* value depending on it's actual meaning. So how to distinguish those?

3.2.1 Available data types

Lets first see, which Java data types are supported on DBObjects. Here is a sample *DBObject* - provided by the *EspritAppSuite* software - using all available types.

```
DBO_tnt_alltype a = new DBO_tnt_alltype();
a.set$Pk("1");  // Priamry key - always a String
a.set$TypeString("Test object");
a.set$TypeInt(Integer.MAX_VALUE);
a.set$TypeLong(Long.MAX_VALUE);
a.set$TypeFloat(Float.MAX_VALUE);
a.set$TypeDouble(Double.MAX_VALUE);
a.set$TypeBoolean(true); // boolean stored in a BOOLEAN database field
a.set$TypeBoolstr(true); // boolean stored in a CHAR(1) database field
a.set$TypeTime(new DBTime());
a.set$TypeTs(new DBTimestamp());
a.set$TypeDate(new DBDate());
a.set$TypeClob(new TestClob());
a.set$TypeBlob(new TestBlob());
a.dbInsert();
```

Note that storing time information is limited to the data types which JDBC offers: *Date, Time* and

Timestamp. In the example above we use *DBDate, DBTime* and *DBTimestamp*, which are special implementations with improved *toString()* output an better compare behaviour.

3.2.2 Column type mapping by naming conventions

The easiest way of getting a reasonable mapping between your database datatypes and the Java datatypes is by defining a naming convention: If for example the fieldname of a SQL DATE field ends with '_ts' then it will be considered to be a *Timestamp* value. As described earlier the type-mapping is done with the DBOTypeMapCredential class which is used as input for the DBOComiler. The method *setUseTypeConventions(true)* activates the following naming conventions:

```
Columnname   Java data type used
*_ts         Timestamp
*_time       Time
*_date       Date
is_*         boolean
use_*        boolean
has_*        boolean
```

3.2.3 Using custom column type mapping

Unfortunately defining a naming convention is not sufficient or sometimes isn't even possible because the existing database tables are not allowed to be renamed accordingly. Furthermore there are situations for instance where a RDBMS system does not support the BOOLEAN datatype. People then use to store booleans as 'T' or 'F' characters in a CHAR(1) database column. How to map those to Java booleans?

Another issue is the fact that often databases running in production environment are poorly designed but not allowed to be changed anymore in order to avoid the risk of running applications beeing unpredictably affected. For example you may find numeric values stored in SQL VARCHAR columns, which of course you rather would like to treat as Java double values. In order to get a 'clean' and well defined mapping you can set up a column type mapping in the *DBOTypeMapCredential* instance like this:

DBOTypeMapCredential

```
// Sample column type mapping file.
// Note that all entries are case-insensitive.

// the enter_time field (DATE) should be mapped as a Java Timestamp value
put("customer.enter_time", Timestamp.class);

// the preferred field (CHAR(1)) should be mapped as a Java boolean value
put("customer.preferred", boolean.class);
```

The valid Java types you can map to are:

```
boolean, int, long, float, double, String, Date, Time, Timestamp, Clob, Blob
```

Specific column type-map definitions of course have precedence over type-mapping conventions.

3.2.4 Using primary keys

Primary keys, unlike attribute fields, are always of type String. One of the reasons for this is, that they must be nullable. Another reason is that there are JDBC drivers which only accept String values as primary keys.

The same is valid for foreign keys. A foreign key value of "0" for example would be a valid reference pointing to an existing primary key in the target table. Whereas a foreign key value of null

means, that there is no such reference.

Not all datatypes make sense to be used as primary or foreign keys. The following sample *DBO_tnt_pktype* (included in the *EspritAppSuite* software) has got a composed primary key that includes all datatypes which are supported within primary keys:

```
DBO_tnt_pktype a = new DBO_tnt_pktype();

a.set$Pk("TestPK");
a.set$PkInt(Integer.MAX_VALUE.toString());
a.set$PkLong(Long.MAX_VALUE.toString());
a.set$PkFloat(Float.MAX_VALUE.toString());
a.set$PkDouble(Double.MAX_VALUE.toString());
a.set$PkBoolean(Boolean.TRUE.toString());  // stored in BOOLEAN database field
a.set$PkBoolsStr(Boolean.TRUE.toString()); // stored in CHAR(1) database field
a.set$PkTime(new DBTime().toString());
a.set$PkTs(new DBTimestamp().toString());
a.set$PkDate(new DBDate().toString());
a.set$Value("sample value");

a.dbInsert();
```

Please remember that all primary key values must be set as Strings.

3.2.5 Primary/foreign key faking

As mentioned already there are databases out in production environment which are poorly desinged. Occasionally you find tables which haven't got a primary key definition although they got a unique index on a particular column. *DBObjects* however need a primary key in order to be capable to update or rehash records. Therefore it is possible to 'fake' primary key as well as foreign key definitions as shown in the *AppLabDBCredential* class in the free *EspritAppLab* project:

AppLabDBCredential

```
addFakedPrimaryKey(DBO_trainer_addr_view.TRAINER_ID);

addFakedForeignKey(DBO_trainer_addr_view.TRAINER_ID, DBO_trainer.TRAINER_ID);
```

As you see the method arguments are constants of an already existing DBObject. Thus for faking keys you need a second DBObject compilation. A 'faked' primary key definition in the column type map even overrides the real primary key definition in the database.

4 Connecting to a database

4.1 Database credentials

Before working with *DBObjects* we must establish a database connection. Therefore you need to provide four database connection parameters: *driver, dburl, user* and *password*. These parameters are wrapped into a *Credential* object wich is passed to the *DBObject.connect(...)* method. There are serveral ways to create the *Credential* object, depending on what way you perfer.

4.1.1 Creating a credential

You may directly pass your connection parameters to the constructor of the Credential class. Here is an example for accessing the *Derby* standard Java Database.

```
Credential cred = new Credential(
     "org.apache.derby.jdbc.ClientDriver",
     "jdbc:derby://localhost:1527/traindb;create=true",
     "user1",  // user
     "user1"   // password
);
```

4.1.2 Reading credentials from the commandline

Usually it's more flexible if you let the parameters be specified from outside of your application code. There is a *DatabaseCommandLineParser* that makes it easy to fetch the parameters from the users command line input:

```
public static void main(String[] args) throws Exception {

     DatabaseCommandLineParser p = new DatabaseCommandLineParser(Main.class, args);

     Credential cred = p.getCredential();

}
```

This is how the command line arguments are expected by the *DatabaseCommandLineParser*:

```
-driver org.apache.derby.jdbc.ClientDriver
-dbUrl jdbc:derby://localhost:1527/espritdb;create=true
-user user1
-password user1
```

4.1.3 Reading credentials from a file

The most flexible and recommend way is to store your connection parameters in a file and read the credential information from there. This makes it reusable by other database tools as well:

```
try {
     Credential cred = new Credential("C:/Databases/espritdb/espritdb.cred");
     ...
} catch (Exception e) {
     // handle error
}
```

This is an expample how a credential file may look like:

espritdb.cred file:

```
###################################################
# Database credential
#
<
     ndfDocVersion = "1.0"
     ndfDocCreated = "2013-09-27 06:08:35"
     ndfDocType = "DatabaseCredential"
>
@ DatabaseCredential {
<
     jdbcDriver = "org.apache.derby.jdbc.EmbeddedDriver"
     connUrl = "jdbc:derby:espritdb"
     dbUser = "user1"
     dbPassword = "user1"
     dbSchema = null

     minConns = 1        # min number of connections in pool
     maxConns = 1        # max number of connections in pool
```

```
      loginTimeout = 10  # database login timeout in seconds
      >
}
@ SystemProperties {
      # Note that the following system property definition is only used
      # if not already defined in the JVM. Thus a JVM option like
      # -Dkey=value always precede the value defined here!
      <derby.system.home = "C:\Projects\EspritAppLab\dbmaster\derby">
}
```

4.1.4 Advanced Credential parameters

Besides the pure JDBC connection parameters a Credential object may contain further optional parameters like *minConns, maxConns* and *loginTimeout*, which are supported by the database connection pool in the *Esprit* software.

4.2 Establishing connections

4.2.1 Establishing a single database connection

Once you got the Credential object you can use it to establish the database connection tike this:

```
try {
    DBObject.connect(cred);
```

➔ If the connection fails, first check if your database server actually is running. Futhermore check the URL and datbase login parameters - there may be a spelling mistake. Also make sure to use your real hostname or your IP-address instead of localhost in your connection URL. Furthermore check whether you included the correct driver jar-file of your database vendor in your CLASSPATH.

The above statement is actually a shortcut for this:

```
try {
    DataSource cp = new DBConnectionPool(cred);
    DBObject.setDataSource(cp);
```

Thus in fact you have opened a connection pool maintaining just a single database connection, which is shared among all *DBObject* subclasses for accessing the database.

4.2.2 Establishing a pool of database connections

You can open a pool of database connections by specifying a *mincon* and *maxConn* parameter in your Credential object:

```
cred.setMinConns(10);  // minimum number of connections requested
cred.setMaxConns(15);  // maximum number of connections to grow up to
DataSource cp = new DBConnectionPool(cred);
```

This call initializes the *DBConnectionPool* with 10 initial connections and allows it to dynamically grow up to 15 if needed. After having grown up the pool will automatically shrink down to 10 when traffic gets less.

➔ The connection pool allows as many transactions running in prarallel as you have got connections. This means a significant performance improvement in multi-threaded environments, in particular when the database host is a multi-CPU machine.

Here is a typical complete source code snippet for starting a database application. Note the usage of the convenience class *DatabaseCLP* which tries to get the *Credential* parameters for you by first

looking for a credfile argument (for reading credentials from a file); if that fails it tries to find the *-driver, -dburl, -user, -password, -mincon, -maxcon* and *-debug* arguments (whereas the latter three are optional). If the connection attempt fails, then an Exception is thrown.

```
public static void main(String[] args) throws Exception {
     DatabaseCommandLineParser p = new DatabaseCommandLineParser(Main.class, args);
     DBObject.connect(p);
     /* Run your application code here */
}
```

Is that easy enough? The *DBObject.connect(p)* method fetches the required *Credential* object directly from the *DatabaseCommandLineParser*. If command line parsing fails, an informative help message is printed automatically. See the javadoc docu for more information.

4.2.3 Closing the database connection

Before your application terminates it should cleanly close all database connections in order to free up the database resources. This is how you do it:

```
DBObject.close();  // close the database connection
```

Actually this call is a shortcut for shutting down the underlying *ConnectionPool*. But by default it's done in a quite brutal manner, because all currently running user-transactions will be interrupted and rolled back immediately. You could do it more user by providing a shutdown-delay in milliseconds. The special value 0 indicates: wait until all currently running user-transactions have finished (new transactions are refused of course).

```
DBObject.close(5000);  // close after a maximum wait of 5 seconds
```

5 Working with DBObjects

5.1 Selecting records from the database

5.1.1 Selecting a single record

The following example shows how to select a particular record from the database. Once you got the record you may print it using it's *toString()* method, which prints it's table name and primary key. Note that the *DBObject* class implements the *Dumpable* interface - so it can dump it's full content by calling the *toDumpString()* method. Furthermore it implements *Serializable* which makes it transferable through streams (like the network).

For reading a particular object from the database you just need to pass a primary key value to the constructor. A new *DBObject* is beeing instantiated by reading it's field values from the database. A *RecordNotFoundException* is thrown if it cannot be found.

```
try {
     DBO_customer c1 = new DBO_customer(10);
     System.out.println("Found customer: "+c1);
     System.out.println("Content: "+c1.toDumpString());
```

If the desired object has got a composite primary key you got to pass it as a *String[ ]* to the constructor.

```
String[] primaryKey = {"1", "T255"};
DBO_trskill ts = new DBO_trskill(primaryKey);
System.out.println("Content: "+ts.toDumpString());
```

At any time you may check if a record still exists in the database (it could eventually have been deleted by another user!). The *exists()* method just checks if it still can be found in the database.

```
boolean isThere = ts.exists();
```

You may at any time refetch the object's attributes from the database (it could have been changed by another user!) using the *dbRehash()* call. Any unsaved previous settings done by you will be lost of course. After a rehash the *isDirty()* method will return *false* because now the object is known to be in sync with the database.

```
ts.dbRehash();
System.out.println("Object rehashed from database: "+ts.toDumpString());
```

5.1.2 Selecting multiple records by condition

This example shows how to query a set of records by a given condition whereas the condition can be any valid SQL condition clause. If the condition is null then all records of the table are read.

```
SqlCondition condition = new SqlCondition();
condition.addEqual("course_type", "DB");
DBO_course[] courses = DBO_course.select(condition);

for (int i = 0; i < courses.length; i++) {
     System.out.println(courses[i]); // prints table name and primary key
}
```

It is strongly recommended to use a type-safe notation in your condition strings. The condition above could better be written as follows:

```
SqlCondition condition = new SqlCondition();
condition.addEqual(DBO_course.COURSE_TYPE, "DB");
```

➔ All column-names are available as static public constants within the particular *DBObject* subclasses. Using them makes your code safe against column renaming in the database.

5.1.3 Iterating through query results

Note that the *select(...)* method returns an array of the loaded records. This is quite convenient, but there is a potential problem: what if the number of records is so big, that it would exhaust your memory resources? In this case you may prefer to iterate through the records using the *RecordIterator*. The difference is that not all records will be in memory at the same time, but rather they are fetched from the database in chunks while iterating through them.

```
SqlCondition condition = new SqlCondition();
condition.addEqual(DBO_course.COURSE_TYPE, "DB");

SqlOrder order = new SqlOrder(DBO_course.COURSE_NAME);
DBObject master = DBO_course.getMaster();
```

```
RecordIterator<DBO_course> it = new RecordIterator(master, condition, order);
while (it.hasNext()) {
     DBO_course course = it.next();
     System.out.println(course.get$CourseName());
}
DBUtil.close(it);
```

Normally the *RecordIterator* instance automatically closes it's underlying database *ResultSet* when all records have been read. But you must invoke it's *close()* method by yourself if you break the iteration before reaching the end.

5.1.4 Sorting records

You can enforce any particular ordering of the returned records by using a *SqlOrder* object for providing an SQL order clause to the *select(...)* call or to the *RecordIterator's* constructor:

```
SqlOrder order = new SqlOrder();
order.add(DBO_course.COURSE_TYPE, false);  // descending
order.add(DBO_course.COURSE_NAME, true);   // ascending

DBO_course[] courses = DBO_course.select(order);
```

In this case the returned records are ordered descending by the *cours_type* field and subordered by the *course_name* field. The order string can be any valid SQL order clause of your database.

Sorting is done by the database server and is somewhat expensive, in particular if the number of records is high. Therefore you should request sorting only when you actually need it.

➔ Note that if you don't specify an order clause then there is a default ordering by primary key (only if one exists of course). If you don't want sorting at all you should explicitely pass a null as an order argument.

5.1.5 Monitoring asynchroneous record loads

Occasionally there are database queries which take so much time (due to slow database, slow network or heavy load) that the user might want to cancel it. This means that the database query has to run asyncroneously, not blocking the users GUI, so that he is still able to press the Cancel-button. We can achieve this by using a *DBLoadMonitor* in conjunction with the *DBLoadThread*. You can write your own sublclass *MyLoadMonitor* deriving from *AbstractDBLoadMonitor* and override the only required method *loadedRecord(DBPersistent)*. The latter method is called each time a record has been loaded by the asynchroneously running *DBLoadThread*. The *DBLoadMonitor* can be cancelled any time by calling it's *cancel()* method, which in fact halts the *DBLoadThread*. Here is an example:

```
MyLoadMonitor monitor = new AbstractDBLoadMonitor() {
     public boolean loadedRecord(DBPersistent record) {
            super.loadedRecord(record);  // counts the records
            System.out.println("Loaded record: "+record);
     }
};

SqlCondition condition = new SqlCondition("course_type = 'DB'");
SqlOrder order = new SqlOrder("course_name");
DBRecord master = DBO_course.getMaster(); // the type of record to be loaded

// Load records asynchroneously
new DBLoadThread(monitor, master, condition, order).start();

Util.sleep(1000);    // let it run a while
monitor.cancel();    // cancels loading, the DBLoadThread will die
```

```
int count = monitor.getRecordCount();  // tell how many records had been loaded
```

If a *DBLoadMonitor* is used it will be responsible for collecting and processing the loaded records.

➔ It is absolutely legal to run several load-threads on different tables at the same time. But you cannot expect better performance unless you create a *DBConnectionPool* with more than one connection. You can run as many database actions in parallel as you have got connections in your pool.

5.1.6 Using more complex SQL conditions

SQL conditions can become very lengthy and tricky. In particular using brackets in mixed AND/OR conditions is error prone and must be checked very carefully. The *SqlCondition* class helps you to avoid typical errors and makes your conditions more easy to write and to maintain. Here is an example:

```
SqlCondition c = new SqlCondition();
c.add("id > 0");
c.and("type = 5");
c.or("owner = 8");
System.out.println(c);
```

This results in a condition string as follows:

```
(id > 0 AND type = 5 OR owner = 8)
```

Mixing AND and OR operators like this is dangerous and you cannot be sure whether the database server understands what you mean. Lets improve this:

```
SqlCondition c1 = new SqlCondition("id > 0");
SqlCondition c2 = new SqlCondition("type = 5");
c2.or("owner = 8");
c1.and(c2);
System.out.println(c1);
```

This results in a better condition string looking like this:

```
(id > 0 AND (type = 5 OR owner = 8))
```

As you see you can nest conditions to any complexity and the Conditon class builds the real SQL string for you with the brackets correctly set.

➔ Writing conditions is the only place where you still got to cope with SQL. Be aware that the SQL syntax is typically RDBMS specific. You are strongly recommended to remain as platform independent as possible. For example when comparing values with the SQL not-equals operator use '<>' instead of '!=' because the latter is not supported by all RDMBS systems.

5.2 4.2 Inserting new records

5.2.1 How to instantiate and insert a record

Creating a new *DBObject* is easy - just instantiate one with an empty constructor. Note that the newly created *DBObject* does NOT exist in the database yet unless you call it's *dbInsert()* method. The *isDirty()* method tells you whether the object is known to be in sync with the database. The newly created DBObject is dirty but it will become clean when the database insert was successfull.

➔ Note that prior to calling dbInsert() all not-null attributes must be set to proper values!

```
DBO_customer c = new DBO_customer(); // creates an empty object
boolean ok = c.isDirty(); // is true, because not in database yet!

c.set$CustomerId(10);      // assign primary key
c.set$Lastname("Reich");
c.set$Company("Cash & Co.");
c.dbInsert();              // inserts the record into the database table

boolean ok = c.isDirty(); // is false now, because in sync with database!
```

5.2.2 Using the master instance for inserts

When doing mass-inserts, it is would be more efficient to reuse the same *DBObject* instance for each insert instead of creating a new one every time. Using the master instance for this would be a perfect solution. Just before you perform the insert you should call *clearValues()* in order to nullify all values from the previous insert.

```
DBO_customer c = DBO_customer.getMaster();  // get the static master instance
c.clearValues();           // ensure all member variables are nullified

c.set$CustomerId(10);      // assign primary key
c.set$Lastname("Reich");   // and attribute values
...
c.dbInsert();              // inserts the record into the database table
```

It is perfectly legal to 'abuse' the master instance for whatever you like, but be aware that it is also used internally. You may consider it to be a *DBObject* for temporary usage only.

5.2.3 Automatic primary key generation

If the primary key exists already in the database the *dbInsert()* will fail and throw an Exception. Note that usually you won't have to assign a primary key value (at least if the primary key is a simple integer value). If no PK-value is given, then it will be calculated automatically:

```
c = new DBO_customer(); // creates an empty object
c.set$Lastname("Arm");
c.set$Company("Pech & Panne");

c.dbInsert(); // inserts a record with the next available primary key
```

The primary key is calculated by a '*select max(customer_id) from customer*' in this case, which is OK for a single insert, but may raise to a performance problem on mass-inserts. For increasing the insert-performance you may consider to use the *PrimaryKeyFactory* class, which manages the primary keys with a much better performance.

5.2.4 Using a primary key factory

This example shows how to use a *PrimaryKeyFactory* for high performance key-generation on record inserts. For instantiating it you need to tell what tables it should manage by passing an array of *DBObjects* to it's constructor (in this example only one). When the factory initializes it will find out the current maximum key value for each managed table and then calculates the new keys based on that.

➔ The *PrimaryKeyFactory* can only manage tables that have got a single-column numeric primary key. A table must be managed by only one *PrimaryKeyFactory* at a time.

```
DBObject[] tables = new DBObject[] {
     DBO_customer.getMaster()
};

DBConnectionPool cp = DBObject.getConnectionPool();

PrimaryKeyFactory keyFac = new PrimaryKeyFactory(cp, tables);
DBObject.setPrimaryKeyFactory(keyFac); // make DBObjects using it

DBO_customer c = new DBO_customer(); // creates an empty object
c.dbInsert(); // inserts record fetching the primary key from the PK-factory
```

➔ Once a primary key value has been requested from the *PrimaryKeyFactory* this value is actually consumed - even if the actual record insert may have failed due to a constraint violation.

5.2.5 Using a factory for setting default values on insert

Often it is required to provide field values for particular columns automatically when the record is inserted. Most database systems provide a way to define such default values, but some don't. In order to remain platform independent you are encouraged to use the *InsertDefaultFactory* class which provides (or calculates) default values for you. Let's suppose we want to set the *customer.enter_ts* column automatically to the current timestamp and give a default email-address on each new customer record. Here is how you can achieve that:

```
String dboPkg = "de.esprit.dbobject.demo.dbo");

InsertFactory insFac = new InsertDefaultFactory(dboPkg);
insFac.put(DBO_Customer.ENTER_TS, CurrentTime.CURRENT_TIME); // evolves to "now"
insFac.put(DBO_Customer.EMAIL, "services@tntsoft.de");

DBObject.setInsertFactory(insFac);   // set the insert-defaults factory

DBO_customer c = new DBO_customer();
c.dbInsert();  // automatically fetches values from the InsertDefaultFactory
```

The *InsertDefaultFactory* is filled with values just like a *HashMap* using the *table.column* as a key. When the factory is present it will be scanned for proper values each time a record is inserted. The special value *CurrentTime.CURRENT_TIME* evolves to the current datetime value, no matter if it is a *Date, Time* or *Timestamp* field. For providing a particular timestamp you may have put a particular *Timstamp* object as value.

The *InsertDefaultFactory* needs to know in which package your *DBObjects* are, because it uses the Java introspection mechanism in order to find out what datatype a particular field has and if it matches with the datatype provided by you.

➔ Note that the *InsertDefaultFactory* implements the *InsertFactory* interface. You could provide your own implementation which calculates insert-default values your way - no matter how complex that is.

Sometimes it is desired to store the insert-default-values in a database table rather than hardcoding them in the program. The *InsertDefaultFactory's* method *loadFromDatabase()* reads the default value definitions from a database table called *insert_def*. You will have to create this table before using this feature, which you can easily do with the call:

```
DBO_insert_def.createTable();
```

5.3 Updating records

You may easily change a *DBObject* just by invoking any *set$xxx(...)* method on it. After the assigment of new values the object is out of sync with the database and *isDirty()* will return *true*.

Now you may call it's *dbUpdate()* method in order to write the changes to the database.

➔ Only attributes should be canged! Do NOT attempt to assign a new primary key value - this would result in updating a different record in the database. DBObjects and database records match by their primary keys. Updating is not possible if no primary key exists.

```
DBO_customer c = new DBO_customer("10");
c.set$Lastname("Huber");    // change the name
c.set$Phone("089/123456");  // change phone number
c.isDirty()      // now returns true because object was modified

c.dbUpdate();    // write changes to the database
c.isDirty()      // now returns false because in sync with database again
```

5.4 Deleting records

5.4.1 Deleting a single record

In order to delete a single record you just call the *DBObject's dbDelete()* method. After a successfull deletion the object has changed to a dirty state which can be checked with the *isDirty()* method.

```
DBO_customer c = new DBO_customer("10");
c.dbDelete();   // deletes the record from the database table
c.isDirty();    // returns true because no more in sync with database
```

➔ It is NOT an error when the *DBObject* could not be found in the database for deletion. It may have been deleted by another user already, which is considered to be a legal situation.

5.4.2 Deleting multiple records by condition

You also can delete many records by condition by using the static *delete(...)* method.

```
SqlCondition condition = new SqlCondition("enter_date < '2003-01-01'");
int deletedRecs = DBO_customer.delete(condition);
```

Hereby the number of actually deleted records is returned. But be aware that there is a potential problem doing it this way: if the number of records is big your database system is likely to run into a long transaction - performing a subsequent rollback, which would result in a huge waste of computer power. Therefore we strongly recommend to either delete each record individually or in well defined chunks in order to remain safe and robust:

```
DBO_customer[] custs = DBO_customer.select(condition);
for (int i=0; i < custs.length; i++) {
     custs[i].dbDelete();
}
```

Because *DBObjects* use prepared statements internally single-record-deletes are very fast! But there is still a problem using this methodology: a query returning a huge number of records may exhaust your memory resources before you even can start the deletion. This is the case where you should use primary-key-only selects as described below.

5.4.3 Primary-Key only selects

Lets think about writing an archiving tool that deletes all records older than a particular timestamp. For finding all 'old' records you run a *select()* with an appropriate condition, but you cannot tell how many records will be hit - it may be a quite huge number. Thus just reading the records may exhaust your memory resources.

In order to save memory it is possible to enforce only the primary key to be read from the database - skipping any attribute values. That way, you will need much less memory and much less data will be transferred through the network. Having only got the primary key value is still enough information for deleting the record with a *dbDelete()* call.

Note that the r*eadPrimaryKeyOnly*-flag does NOT affect single records reads. Thus a *dbRehash()* call will always complete your *DBObject* with attribute values. The same is valid for instantiating *DBObjects* using one of its constructors like: *new DBO_tnt_logmsg(1234)*;

```
SqlCondition condition = new SqlCondition();
condition.add(DBO_tnt_logmsg.ENTER_TS+ " < '2003-11-01 12:00:00'");

try {
     DBO_tnt_logmsg.getMaster().setReadPrimaryKeyOnly(true);
     DBO_tnt_logmsg [] recs = DBO_tnt_logmsg.select(condition);
     for (int i = 0; i < recs.length; i++) {
            recs[i].dbDelete();
     }

} catch (Exception e) {
     // handle error

} finally {
     // Do NOT forget to reset the flag after reading
     DBO_tnt_logmsg.getMaster().setReadPrimaryKeyOnly(false);
}
```

5.5 Forced insert and update

There are cases where you just want to store a record in the database - not worrying about whether it exists already or not. If it exists, it should be updated, if it doesn't, it should just be inserted. This is exactly what the *forceInsert()* and *forceUpdate()* methods provide. Lets suppose we have got an object like this:

```
DBO_customer c = new DBO_customer();
c.set$Customer_id("1001");
c.set$Lastname("Buesch");
c.set$Company("EsprIT-Systems");
c.set$Email("rainer.buesch@esprit-systems.de");
```

The *forceUpdate()* method tries to update the record. If it fails because the record does not exist yet an insert is performed instead.

```
c.forceUpdate();  // try to update - if it fails => insert
```

The *forceInsert()* method tries to insert the record. If it fails because the record exists already it performs an update instead.

```
c.set$Email("service@tntsoft.de");
c.forceInsert();  // try to insert - if it fails => update
```

Both the *forceInsert()* and *forceUpdate()* methods lead to the same result. Which one to use depends on which is the more likely case you expect. If you expect that in most cases the record might exist already you would prefer the *forceUpdate()* method due to better performance.

6 Running transactions

6.1 How to use transactions

There are cases where you need to perform a group of database changes all of which must succeed. If one of them fails all previous changes have to be undone. This is what database transactions do for you. *DBObject's* support transactions with three static methods: *begin(), commit()* and *rollback()*.

➔ A *DBObject.begin()* must be followed by eiter a *DBObject.commit()* or *DBObject.rollback()*. You typically use a try-catch notation to ensure this. Forgetting to finish a transaction is a fatal programming error!

The following example shows how to insert 'mother' and 'child' records into two tables which have a foreign key relationship. A new course is inserted and an according new skill is added for all trainers. The whole procedure is done in a single database transaction which is rolled back if any of the inserts fails.

```
Try {
     DBObject.begin(); // start a transaction

     DBO_course tc = new DBO_course();
     tc.set$Course_id("T201");
     tc.set$Course_name("Database Design");
     tc.set$Days(3);
     tc.dbInsert();

     DBO_trainer[] trainers = DBO_trainer.select();
     DBO_trskill skill = null;

     for (int i=0; i<trainers.length; i++) {
            skill = new DBO_trskill();
            skill.set$CourseId(tc.get$CourseId());              // primary key
            skill.set$TrainerId(trainers[i].get$TrainerId());  // primary key
            skill.set$PrefGrade(1);
            skill.dbInsert();
     }

     DBObject.commit(); // commit the transaction

} catch (Exception e) {
     Util.error(this, "Transaction failed", e);
     DBObject.rollback(); // transaction must be rolled back!
}
```

6.2 Running nested transactions

Database systems do typically not allow to nest transactions. Thus after a BEGIN the database expects a COMMIT or ROLLBACK, but will not accept a second BEGIN. This is somewhat inconvenient for you as a programmer because when you open a transaction and then call a method on any object you probably do not know whether within that method again a transaction is opened or not.

Anyway *DBObjects* support nested transactions, so that you need not worry about. Nesting works as follows: If a thread calls the *DBObject.begin()* method twice the system will discover, that it already is within a transaction and just keeps track of the nesting depth. A subsequent

DBObject.commit() call will decrease the nesting depth. If the nesting depth is back to zero then the real database COMMIT is performed. A *DBObject.rollback()* call will allways break the transaction at any nesting level and perform the real database ROLLBACK immediately.

Here is an example using nested transactions. Note that the *Resistor* class is a *ComposedRecord* and it's *dbUpdate()* method in fact updates several internally managed *DBObjects* within a transaction - this is where the nesting comes in.

```
Try {
     DBObject.begin();

     DBO_tnt_logmsg l = new DBO_tnt_logmsg();
     l.set$Message("Changed resistor value");
     l.dbInsert();

     Resistor r = new Resistor("123"); // read by primary key
     r.setOhmValue(100);  // is a ComposedRecord field
     r.setTolerance(2);   // is a ComposedRecord field

     // This runs in a nested transaction within the Resistor object
     r.dbUpdate();

     DBObject.commit();   // end of nested transaction

} catch (Exception e) {
     DBObject.rollback();
}
```

6.3 Running multiple transactions in parallel

DBObject transactions are thread safe - thus many threads may call the *DBObject.begin()* method in order to start a transaction. If there is only one database connection available in the underlying *DBConnectionPool* the first thread who starts a transaction will block it until his transaction has been finished by either *DBObject.commit()* or *DBObject.rollback()*. Other threads will have to wait.

If however your *DBConnectionPool* has got several connections available other threads may run their transactions in parallel.You can run as many transactions in parallel as you have got connections in the pool.

➔ Note that you will see the best performance if your database host is a multi-CPU machine, because then it is possible that transactions run in parallel on several CPUs at actually the same time.

7 Multiple database connections

7.1 Connecting to multiple databases

The *DBObject* superclass allows for setting a *DBConnectionPool* instance. This instance is shared by all *DBObjects* for performing database access. At any time you can easily replace this instance by another one which is connected to a different database (even on a RDBMS system of a different vendor). That way you can easily read a record from one database and write it back to another one.

7.2 Database cross copy example

This example shows how to copy DBObjects across RDBMS systems of different vendors. For example lets move all logging records older than a particular date from an Informix database to an Oracle database:

```
try {
     // Read the connection params for both Oracle and Informix
     Credential oraCred = new Credential("C:/credentials/oracle.cred");
     Credential ifxCred = new Credential("C:/credentials/informix.cred");

     // Create a ConnectionPool for both Oracle and Informix
     DataSource oraCP = new DBConnectionPool(oraCred);
     DataSource ifxCP = new DBConnectionPool(ifxCred);

     // Lets read the records from the Informix database
     DBObject.setDataSource(ifxCP);

     SqlCondition condition = new SqlCondition();
    condition.add(DBO_tnt_logmsg.ENTER_TS+" < '2003-06-01 12:00:00')"

     DBO_tnt_logmsg[] recs = DBO_tnt_logmsg.select(condition);
     DBO_tnt_logmsg r;

     for (int i=0; i<recs.length; i++) {
            r = recs[i];
            try {
                   DBObject.begin();  // starts a transaction on Informix
                   r.dbDelete();      // deletes from the Informix database

                   DBObject.setDataSource(oraCP);
                   r.dbInsert();      // inserts into the Oracle database

                   DBObject.setDataSource(ifxCP);
                   DBObject.commit(); // commits the transaction on Informix

            } catch (Exception e) {
                   DBObject.setDataSource(ifxCP); // reset to Informix
                   DBObject.rollback();  // rolls back on Informix
            }
     }
} catch (Exception e) {
     Util.error(this, e);
}
```

Moving from one database to another is just as easy as switching the *DBConnectionPool*. Be aware that the transaction shown above runs on the Informix system only. If the INSERT into the Oracle database fails then the transaction on the Informix database is rolled back and leaves the data unchanged. That way we have got a safe data transfer.

7.3 Conversion between different table structures

The example above requires the tables on Informix and Oracle to have exactly the same structure. If this is not the case you need to convert the Informix record into an Oracle record by writing a simple conversion method that copies the values of interest - like this:

```
private DBO_ora_record toOracleRecord(DBO_ifx_record i) {
     DBO_ora_record o = new DBO_ora_record();
     o.set$RecId(i.get$RecId());
     o.set$EnterTs(i.get$EnterTs());
     ...
     return o;
}
```

Of course you are not limited to two database systems. This way of interfacing databases is as easy as it can be if you compare it with other ways people typically use to do it: unloading data into a file from one system and uploading it from the file onto another system.

8 Importing/Exporting data

DBObjects serve as data container objects representing records read from the database. But they also contain meta information which can be used to present the data in various formats like HTML, XML or UNL.

8.1 Formatting output

The *DBObjectSuite* software contains paritcular writer classes like *DBOHtmlWriter* and *DBOXmlWriter*, both derived from *TagWriter* which create tag-based output for you. Tag's can be opened recursively with an unlimited nesting depth.

8.1.1 Creating HTML output

When you use *DBObjects* as persistency layer for web applications (based on *Servlets*), then you can use the *DBOHtmlWriter* class for writing their content in a HTML table format. The following code snipplet could easily be used from within a servlet:

```
DBObject master = DBO_trainer.getMaster();
PrintStream ps = System.out;

try {
     DBOHtmlWriter w = new DBOHtmlWriter(ps)
     w.openTag("html");
     w.openTag("body");
     w.openTable(master);

     DBObject[] recs = (DBObject[]) master.dbSelect();
     for (int i = 0; i < recs.length; i++) {
            w.writeDBObject(recs[i]);
     }

     w.closeTable();
     w.closeTag(); // body
     w.closeTag(); // html
     w.close();    // flushes and closes the writer

} catch (Exception e) {
     // handle error
}
```

This example writes the content of the selected *DBObjects* out to the given *PrintStream* (which of course could be a *ServletOutputStream*) in a default HTML table format. For customizing the table look you can easily create your own *MyDBOHtmlWriter* that extends from *DBOHtmlWriter* and overrides the *getTableAttributes()* method for returning different table properties. Refer to the javadoc API docu for details.

8.1.2 Creating XML output

It is getting more and more common to exchange data with other systems using the XML format. Here is an example how you can use the *DBOXmlWriter* class for generating XML output from the *DBObject's* record information.

```
DBObject master = DBO_trainer.getMaster();

PrintStream ps = System.out;
try {
     DBOXmlWriter w = new DBOXmlWriter(ps);
     w.setWriteEmptyTags(true);
     w.openTag("courses");
```

```
        DBO_course[] courses = DBO_course.select(null);
        for (int i = 0; i < courses.length; i++) {
                w.writeDBObject(courses[i]);
        }

        w.closeTag(); // courses
        w.close();  // flushes and closes the writer
} catch (Exception e) {
        // handle error
}
```

Feel free to create your own *MyDBOXmlWriter* class deriving from *DBOXmlWriter* that customizes any behaviour as needed.

8.2 Loading/Unloading data with UNL

The UNL format is used for exporting database data into ASCII files. A table record is represented as a line of text whereas the various field values are separated by a special character (by default '|'). This format does not store any meta information like column names, just pure data only. Obviously binary large objects cannot be stored that way and are just treated as empty fields.

See also: Tools for Exporting/Importing data in UNL format

8.2.1 Writing UNL output

The *DBOUnlWriter* class allows for writing UNL output directly from a *DBObject* instance. The following example reads the table data from the database and extracts it into a file in UNL format.

```
DBOUnlWriter wr = new DBOUnlWriter("C:/temp/trainer.unl");
wr.writeRecords(DBO_trainer.select());
wr.close();
```

The resulting output file looks like this: **C:/temp/trainer.unl**

```
1|Mary|Poppins||||
2|Harry|Hirsch||||
```

As you can see the fields are separated by a special character (by default '|') which you can change with the *setFieldSeparator(...)* method. The number of separators matches the number of fields. If there is no information found between two separators it's considered to be a null value.

Note that a field value may contain the field separator character by incidence. In this case it will be quoted with a '\' in the output in order to mark it as a non-special character.

➔ All data is written in a neutral format - no localization is done! Therefore floating point numbers use a '.' as decimal separator, a date is formatted like yyyy-MM-dd etc.

8.2.2 Reading UNL input

The *DBOUnlReader* class allows for reading UNL input and converting it to *DBObject* instances. The following example reads the UNL file that was just written in the previous chapter. It parses the UNL data line by line and prints the resulting *DBObjects* to the system console.

```
DBObject master = DBO_trainer.getMaster();
DBOUnlReader reader = new DBOUnlReader(master, "C:/temp/trainer.unl");
DBObject record;
```

```
while (reader.hasNext()) {
     record = reader.next();
     System.out.println(record.toDumpString());
}
reader.close();
```

As you see in the example there is a *DBObject* master instance passed to the constructor. Thus all resulting *DBObjects* will be instances of that type and be castable as such. Once a *DBObject* is read that way it's easy to insert it into a database table by just calling it's *dbInsert()* method.

➔ Note that the number of fields in the UNL file must match the number of fields of the master *DBObject*, otherwise an *Exception* is thrown on parsing.

8.2.3 Handling UNL parsing errors

Lets suppose you received an UNL file containing records which you have got to insert into your database table. Occasionally it happens that data are not clean. For instance a date value may be formatted wrong beeing not parsable. Such bad records need manual rework and should be separated from the good ones. The *DBOUnlReader* povides an easy way for doing this:

```
reader.setViloationOutput("C:/temp/violation.unl");
```

When the violation output file is defined, then all non-parsable input lines are written into it. Each violation record is followed by an additional comment line telling details about the error. After having fixed the problems manually you can use the same file as input for another try.

9 Special features

9.1 Comparing DBObjects

9.1.1 Checking equality of primary keys

DBObjects can be compared using their *equalPrimaryKeys(...)* method, which is in fact the same as what the *equals(...)* method does. Two *DBObjects* are defined to be equal when all their primary key values are not null and equal. Attributes are not checked. If the *DBObject* hasn't got a primary key the equality check always returns *false*. Lets compare a few *DBObjects*:

```
DBO_customer c1 = new DBO_customer(1000);
DBO_customer c2 = new DBO_customer(1000);
boolean isSame = c1.equals(c2); // evaluates to true
c2 = new DBO_customer(1111);
boolean isSame = c1.equals(c2); // evaluates to false
```

9.1.2 Checking equality of attributes

Sometimes it is usefull to check whether two *DBObjects* have got identical attributes even if they have got different primary keys. You can do this using the *equalAttributes(...)* method:

```
boolean hasSameAttributes = c1.equalAttributes(c2); // checks attributes only
```

Unlike primary key values two attribute values are considered to be equal if both are null.

9.1.3 Using Comparable implementation

Note that *DBObjects* also implement the standard *Comparable* interface. Lets compare them using their *Comparable* implementation:

```
int cmp = c1.compareTo(c2); // -1 because key of c2 is bigger then key of c1
cmp = c1.compareTo(c1);     //  0 because both got the same primary key
```

You may use the Java *Collection* API in order to sort records as follows:

```
List list = new ArrayList();
list.add(c1);
list.add(c2);

Collections.sort(list); // sorts the list by using Comparable implementation
```

9.2 Table oriented actions

A *DBObject* subclass represents a table in the database. Nevertheless somebody may have dropped the underlying database table while your *DBObject* still exists in memory. Any try to access it would certainly fail.

9.2.1 Checking table existence

You may want to check if the real table still exists with the static *existsTable()* method.

```
boolean exists = DBO_tnt_logmsg.existsTable(); // false, because does not exist
```

9.2.2 Creating the table

Because the *DBObject* has full knowledge about the table meta data it is capable to reconstruct the missing database table by it's own even including primary/foreign key relationships.

```
DBO_tnt_logmsg.createTable(); // creates the table in the database
boolean exists = DBO_tnt_logmsg.existsTable(); // true, because now exists
```

The actual '*create table...*' statement looks quite different on different RDBMS systems - these differences are handled internally by the by *DBConnectionPool* which uses different *RdbmsSupport* objects depending on which RDBMS it is connected to.

If the table has foreign key definitions, we must also create those. For doing this any referenced table must actually exist already in the database. Thus a method call like the following is usually the last one after having created all database tables:

```
DBO_tnt_logmsg.createForeignKeyConstraints();
```

9.2.3 Dropping a table

Dropping the table from the database is quite easy. Be aware that all data in the table will be lost of course.

```
DBO_tnt_logmsg.dropTable();
boolean exists = DBO_tnt_logmsg.existsTable(); // false, because it's gone
```

➔ Note that dropping a table is a very different database operation than deleting all records from the table. It's much faster and there is no risk to run into a long transaction.

9.2.4 Counting records in a table

You can easily find out the number of records in a particular table by invoking the static *countRecords()* method. Internally the *DBObject* performs a highly performant key-only select on the primary key index. If the table hasn't got a primary key the count would be much more expensive.

```
int numRecs = DBO_customer.countRecords(); // returns number of records
```

There is a second *countRecords(...)* method which accepts and *SqlCondition* as argument and only counts the records, that match the given condition. So we can find out easily whether a customer with a particular name exists already in the database:

```
SqlCondition cond = new SqlCondition();
cond.addEquals(DBO_customer.NAME, "EsprIT-Systems");
int numRecs = DBO_customer.countRecords(cond); // returns number of matches
```

➔ For executing the above statement with good performance it is hardly recommended that the search column should be indexed in the database.

9.3 Using database schemas

Database users normally are not much aware of database schemas. When a user connects to the database there is always a default-schema active and you can access your tables like this:

```
>> select * from hobbies;
```

But there are cases where you want to access a table in another user's schema. You then have to use select statements like this in order to see for example TOM's table:

```
>> select * from "TOM".hobbies;
```

Setting the schema in the *DBObject* class does exactly this for you.

```
DBObject.setSchema("TOM");
```

After this call all *DBObject* subclasses will prepend the schema name to the table name whenever a table is accessed.

➔ "TOM".hobbies and hobbies are different tables in the database, but we require them to have exactly the same structure. Creating equally named tables with different structures in different schemas would anyway be a good way to confuse everybody including yourself.

You can switch back to the default schema with:

```
DBObject.setSchema(null);
```

9.4 Storing zero and blank values

9.4.1 Storing empty strings

Usually it is considered to be unintended to store a string only consisting of whitespaces in a database CHAR field. Therefore String values are trimmed when they are read/written from/to the

database. The default behaviour of *DBObjects* is that pure white-space-strings are treated as null values.

But there are cases where it in fact may be intended to store just a a blank character or to read leading and trailing blanks of a field (although you are not encouraged to do this). You can achieve this with these calls:

```
DBObject.setTrimStringsOnRead(false);  // default is true
DBObject.setTrimStringsOnWrite(false); // default is true
```

9.4.2 Storing numeric zero values

A *DBObject* uses an Java *int* variable to model a database INTEGER column (*long, float, double* respectively). The INTEGER database type however makes a difference between a NULL and a 0 value, which is not possible with Java *int* variables. Using the *Integer* class would solve this problem but to the high cost of memory and performance. As a compromise *DBObjects* let you determine whether a numeric zero should be stored as a NULL or as a 0 value.

```
DBObject.setStoreZeroAsNull(true); // default is false
```

Note that this distinction does not affect primary or foreign key columns because those are always modelled as Strings and thus are nullable anyway.

9.5 Logging

9.5.1 Using a custom Logger

DBObjects report logging information to a built-in default *LogSupport*, which just prints to the system console. You may supply your own *LogSupport* as shown below:

```
DBO_customer c = new DBO_customer();
c.update(); // fails ==> watch message of default error handler on console

DBObject.setErrorLogger(new MyErrorLogger()); // Set a custom log handler
c.update(); // fails ==> watch message of custom error handler on console

DBObject.setErrorLogger(null); // switches logging off totally
```

The *MyErrorLogger* is a class that implements the *LogSupport* interface. Here is an example for an implementation. Note that you are free to implement it your way.

```
public class MyErrorLogger extends LogSupportAdapter {
    public MyErrorLogger() {
          super("DBO");
    }
    public void logInfo(Object caller, String msg) {
          super.logInfo(caller, "#DBO# "+msg);
    }
    /* override all other logXX methods accordingly */
}
```

9.5.2 Switching off messages

Somtimes it is desired not to see any error or info messages at all. Therefore you can use the *DBObject.setSilent(true)* call for suppressing message reporting, which in fact is a shortcut for setting the *ErrorLogger* to null.

```
DBObject.setSilent(true); // no messages are reported at all
```

Setting the silent mode back to *false* will reinstall the previous *ErrorLogger*.

9.5.3 Printing different LogLevels

DBObjects internally use the logging API provided with this *EspritAppSuite* software. You may increase the verbosity of the output by exlicitely setting the *LogLevel*:

```
DBObject.getErrorLogger().setLogLevel(LogLevel.LOG_VERBOSE); // Make it telling more
```

10 Example partlist program

This sample program demonstrates the usage of *DBObjects* for data extraction. It finds out all parts of a particular car (*car* table), counts the number of parts (*part* table) that comprise the car and summarizes it's prices (which are stored in the *stock* table).

```
public class Partlist {

     public Partlist(String carId) throws Exception {
             DBO_car car = new DBO_car(carId);

             // find all parts of this car in the part table
             SqlCondition cond = new SqlCondition();
             condition.add(DBO_part.CAR_ID, car.get$CarId());

             DBO_part[] parts = DBO_part.select(condition);

             if (parts.length == 0) {
                    System.out.println("No parts found for car: "+car);
                    return;
             }

             // Let's step through the partlist and find out the part
             // prices which are stored in the 'stock' table.
             DBO_part part;
             DBO_stock stock;

             int totParts = 0;
             double totPrice = 0;

             for (int i=0; i<parts.length; i++) {
                    part = parts[i];

                    // Note that the stock table has a composed primary key
                    String[] key = {part.get$SupplierId(), part.get$StockId()};
                    stock = new DBO_stock(key);

                    totParts += part.quantity;
                    totPrice += stock.get$PartPrice() * part.get$Quantity();
             }

             System.out.println("The car "+car+" has: "+totParts+" parts");
             System.out.println(" The total price is: "+totPrice+" bucks");
             DBObject.close();  // close the database connection
     }
```

```
    public static void main(String[] args) throws Exception {
        DatabaseCLP p = new DatabaseCLP(args);
        DBObject.connect(p.getCredential());
        new PartList(p.getString("-carid"));
    }
}
```

11 General purpose database tools

The *EspritAppSuite* Software contains usefull commandline tools that operate on your database. All these tools can be used in the same way on all supported database systems. You may use these tools also within shell scripts. All of them require database credential information to be passed at startup, either with the *-credfile* option or a set of connection parameters (*-driver, -dburl, -user, -password*), so that a database connection can be established.

11.1 Tools for executing SQL commands

11.1.1 DBExecute Tool

The purpose of the *DBExecute* tool is to run a single or a set of SQL statement(s) in the database. It can be started by executing the batchfile dbexecute.bat in the bin subdirectory. There are a few options in order to specify what it should do for you:

- -set <table.column=value>
  Allows for setting a particular table column to the given value
- -cond <sql_condition>
  Defines a SQL condition which is used in conjunction with the -set option
- -sql <sql_statement>
  Just runs the given SQL statement in the database
- -script <sql_script_file>
  Executes all SQL statements of the given scriptfile in the database
- -skiperrors
  Continues execution of a script even if a statement fails

Please examine the dbexecute.bat file to find a few examples how to use.

11.1.2 DBSelect Tool

The purpose of the *DBSelect* tool is to run a single query or a set of queries in the database (SQL SELECT statements only). It can be started by executing the batchfile *dbselect.bat* in the bin subdirectory. There are a few options in order to specify what it should do for you:

- -outfile <output_file>
  Redirects the output to the given file. By default the output is printed to the shell console.
- -sql <sql_select_statement>
  Defines what SELECT statement should be executed in the database
- -script <sql_script_file>
  Executes all SQL SELECT statements of the given scriptfile in the database
- -delim <delimiter_char>
  Defines which character is used as the field separator in the output. The default is '|'.

- ➔ -header
  Tells whether a headline with the column-name information should be printed
- ➔ -skiperrors
  Continues execution of a script even if a statement fails
- ➔ -silent
  Print pure SELECT output only, suppresses any other info messages on the console.

Please examine the *dbselect.bat* file to find a few examples how to use it.

11.2 Tools for Exporting/Importing data in UNL format

The UNL format is a simple ASCII format which represents your table data (without any meta information). It is quite usefull for backing up data externally of the database or exchanging data with other systems. Another common usage is for example this one: extract the table data into a file, perform any processing (manual or automatic) on the data and re-import it into the database table. The tools that allow you to do this are *UnlExport* and *UnlImport*.

See also: Loading/Unloading data with UNL

Note that these tools use *DBObjects* internally and thus require that a *DBObject* subclass exists for each of your database tables. Therefore - besides the credential parameters - both tools have got the following mandatory options in common:

- ➔ -dbopkg <dbo_package>
  Tells in which package it can find the DBObject subclasses for your database. Of course these classes must be available in your CLASSPATH.
- ➔ -table <tablename>
  The name of the affected database table.
- ➔ -record <full_dbobject_classname>
  Specifies the full classpath of the DBObject that represents a table. The *DBObject* class file must be found in the CLASSPATH of course. You can use this option as an alternative to the *-dbopkg* and *-table* options.

11.2.1 UnlExport Tool

The *UnlExport* tool (see *bin/unlexport.bat* commandline script) reads records from a table and extracts it's data into an UNL file. Besides the common parameters there are a few options to specify what it should do for you:

- ➔ -outfile <UNL_output_file> (mandatory)
  Redirects the UNL output to the given file.
- ➔ -cond <sql_ condition> (optional)
  Allows to specify an SQL condition for filtering the records

11.2.2 UnlImport Tool

The *UnlImport* tool (see *bin/unlimport.bat* commandline script) reads records from an UNL file and inserts them as new records or updates existing ones respectively in the database table. Besides the common parameters there are a few options to specify what it should do for you:

- ➔ -unlfile <UNL_input_file> (mandatory)
  The UNL input file to be read.

- -viofile <violation_file> (otional)
  If there are UNL parsing errors they will be reported to this file together with the failing input line. This file can manually be reworked and then be used as input for another try.
- -update (optional)
  Enforces updates of existing records only. No new records are inserted.
- -addnew (optional)
  Enforces adding new records only. Existing records remain untouched.
- -newpk (optional)
  Enforces all records from the UNL file to be added as new records with a new primary key. This only works if the according table has a single numeric primary key.

11.2.3 UnlExport Tool

The *UnlExport* tool (see *bin/unlexport.bat* commandline script) reads records from a table and extracts it's data into an UNL file. Besides the common parameters there are a few options to specify what it should do for you:

- -outfile <UNL_output_file> (mandatory)
  Redirects the UNL output to the given file.
- -cond <sql_ condition> (optional)
  Allows to specify an SQL condition for filtering the records

11.3 Interactive Tools

11.3.1 TableEditTool

The *TableEditTool* (see *bin/table_edit.bat* commandline script) is a graphical user interface for editing records of database tables based on *DBObjects*. In order to edit a table you must enter the fully qualified name of a *DBObject* subclass into the class name field and then press the load button. If the class can be found then the records of the according tables will be loaded and you may use the GUI for editing them. The most important command line options are:

- -credfile <credential_file> (mandatory)
  The database credential file for connecting the database.
- -record <DBO_class_file_name> (otional)
  The fully qualified name of a DBObject subclass. The according table content is read by default. Note that the argument may also be any class that implements the DBRecord interface like for instance any ComposedRecord subclass. Composed records are another very powerfull application of DBObjects, just beyond the skope of this manual.

12 Additional information

- For detailed information about all classes within the EspritAppSuite software please refer to the javadoc documentation provided in the ESPRIT_HOME/docs directory, which you can browse with your webbrowser (start at index.html).
- For an easy to understand discussion of the DBObject philosophy please refer to the article published in JavaMAGAZINE in July 2004 (german), which you can find in the internet at: http://www.sigs.de/publications/js/2003/04/buesch_JS_04_03.pdf.
- This article is also available in english at: http://www.tntsoft.de There you can also find information about related products such as DynamIT (a database browser and editor).

- ➔ Included in the EspritAppSuite software there is a set of PowerPoint foils that give an overview about DBObjects and related products (in the ESPRIT_HOME/foils directory).
- ➔ There is a special package called de.esprit.dbosuite.demo (ESPRIT_HOME/src directory) which contains various source codes examples of how to use DBObjects. Feel free to explore these files and use them as templates for your projects.