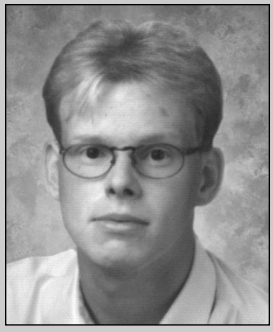
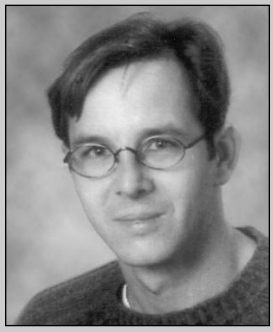
Achieving Platform-Independent Database Access with Open SQL/SQLJ — Embedded SQL for Java in the SAP Web Application Server

Andreas Fischbach and Adrian Görler



Andreas Fischbach, Java Server Technology Group, SAP AG



Adrian Görler, Java Server Technology Group, SAP AG

(complete bios appear on page 30)

The Java programming language has two important qualities that are indispensable for developing large-scale business applications:

- Java allows you to write entirely platform-independent code — a key premise of the language is "write once, run anywhere." At design time, the Java compiler checks the source code for syntax and semantics errors and converts it to platform-independent byte code. At runtime, this byte code is executed by a Java virtual machine regardless of the platform on which it is running.

- Java is a very safe programming language with strict design-time checks. Many issues (such as uninitialized variables) that would lead to runtime errors in other languages are detected at design time in Java.

Starting with Release 6.20, the SAP Web Application Server (Web AS) contains a full-fledged J2EE[1] server that provides standard Java application support for SAP environments. However, business applications usually store persistent data in a relational database, and although J2EE offers a standard for object persistence (entity beans), you might want to use relational persistence to achieve full control over the database access layer. So what support does Java provide for developing relational persistence code? Is there an offering with the same qualities — platform independence and design-time checks — as the Java programming language itself?

Unfortunately, Java alone does not offer a satisfying solution. Java relies on Java Database Connectivity (JDBC), which is merely an API

[1] Java 2 Platform, Enterprise Edition.

for executing SQL statements on the database and processing the results. Because JDBC doesn't use a standardized SQL grammar, the SQL statements are not checked. Typically, you need to manually adapt JDBC code to the individual target database and test it very thoroughly. And since JDBC is a dynamic protocol, the SQL statements sent to the database cannot be checked at design time, so errors might only be detected during productive use.

SQLJ (or "SQL Java") takes a different approach. SQLJ is an ISO[2] standard for accessing relational databases from Java by embedding static SQL statements, which can be checked at design time, directly in the Java code, providing the statement-checking capability that the JDBC API lacks. SQLJ does not specify the SQL grammar to be supported by the databases accessed, however, so you still need to adapt the code to the target database.

Release 6.30 of the Web AS offers an all-encompassing solution for developing relational persistence code — Open SQL for Java. A key part of this framework is Open SQL/SQLJ, which is an implementation of SQLJ that is truly platform-independent. With Open SQL/SQLJ, you can write persistence code that runs on any database supported by Web AS 6.30 with equivalent syntax and semantics, which will be checked at design time. It relieves you of the burden of adapting the persistence code to the target database. With Open SQL/SQLJ, the ability to "write once, run anywhere" and perform strict design-time checks becomes a reality for persistence code.

This article is the first in a series on Java persistence in the 6.30 release of the Web AS. It provides an introduction to using Open SQL/SQLJ for developing database-centric web applications with the SAP NetWeaver Developer Studio, SAP's new development environment for Java-based applications. We will first introduce you to the Open SQL for Java framework. We will then explain the fundamental concepts of SQLJ and walk you through a series of examples that illustrate how to apply the most important features of Open SQL/SQLJ. Finally, you will learn how Open SQL/SQLJ fits into the NetWeaver Developer Studio and the Web AS. Note that some familiarity with Java programming, especially using JDBC, is assumed.

[2] International Organization for Standardization (**www.iso.org**).

An Introduction to Open SQL for Java

Writing persistence code with JDBC, which is tailored to accommodate many different databases, is very cumbersome. Although SQL is an ISO standard, individual databases understand a wide variety of SQL dialects with many syntactical variants. In addition, the supported data types and type properties differ greatly between databases.

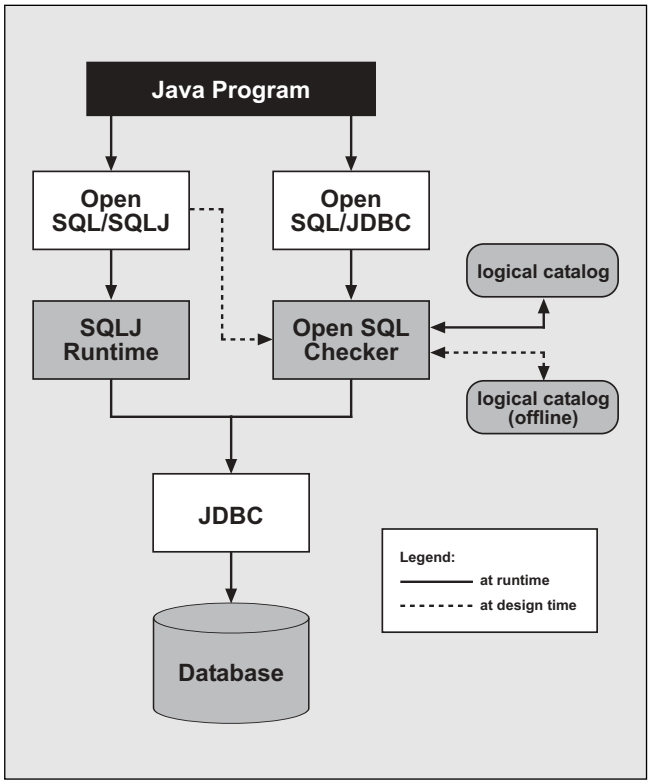
Open SQL for Java[3] is a new SAP framework contained in Web AS 6.30 that was developed to level these differences between individual databases and JDBC drivers. **Figure 1** illustrates the architecture of this framework. The heart of Open SQL for Java is the "Open SQL checker," which checks SQL statements for conformance with the Open SQL grammar. This grammar is a subset of the ISO standard SQL-92 that is syntactically recognized by all databases supported by Web AS 6.30. The Open SQL grammar roughly comprises SQL-92 Entry Level.[4] All SQL statements that are part of this subset are guaranteed to execute with identical semantics on any of the supported databases.

To abstract from the names and properties of the data types supported by individual database vendors, the Open SQL framework contains a "logical catalog" that is independent of the underlying database. The logical catalog is available at runtime in the J2EE server, as well as offline at design time in the NetWeaver Developer Studio. All database tables are first created in the design time catalog and

[3] A future article will cover Open SQL for Java in more detail.

[4] For more information on the Open SQL grammar, refer to the SAP NetWeaver Developer Studio product documentation (see the sidebar on page 30).

Figure 1 The Open SQL for Java Framework



then deployed to the J2EE server. As part of the deployment, the data types in the logical catalog are mapped to the data types of the database being used, and then the actual database table is created.

Overview of Basic SQLJ Concepts

As a foundation for the examples that follow, we begin with a discussion of SQLJ and how it works. You need to understand the following fundamental concepts:

- How the SQLJ standard evolved
- The role of the SQLJ translator as a preprocessor
- SQLJ statements and syntax
- Connecting to databases via a connection context
- Processing result sets with iterators

Evolution of the SQLJ Standard

SQLJ is a standard that specifies the interoperation between the Java programming language and SQL. It was developed by a consortium of companies comprising Cloudscape, Compaq, IBM, Informix, Oracle, Sun, and Sybase. SQLJ was originally filed as an ANSI standard consisting of three parts:

- *SQLJ Part 0* — Object Language Bindings (for using embedded static SQL in Java)
- *SQLJ Part 1* — SQL Routines Using the Java Language (for using Java in stored procedures)
- *SQLJ Part 2* — SQL Types Using the Java Programming Language (for using Java with user-defined data types)

In 2000 and 2002, an extended version of SQLJ was adopted as an ISO standard:

- *ISO/IEC 9075-10:2000* — SQL/Object Language Bindings (SQL/OLB)
- *ISO/IEC 9075-13:2002* — SQL Routines and Types Using the Java™ Programming Language (SQL/JRT)

Open SQL/SQLJ is the SAP implementation of ISO 9075-10, which is an extension of the original SQLJ Part 0 standard. This implementation consists of two pieces of software — the SQLJ translator, which processes the SQLJ source code at design time, and the SQLJ runtime, which executes SQLJ at runtime.

The SQLJ Translator

SQLJ allows you to embed SQL statements directly in the Java source code. The grammar of the SQLJ source code is standard Java, extended by special SQLJ statements that start with the token *#sql*. Obviously, the standard Java compiler (*javac*) cannot understand these SQLJ statements. Therefore, as a first step, a preprocessor called the “SQLJ translator” translates the SQLJ source file into an intermediate Java file. In this file, the SQLJ statements are

Figure 2 ISO Standard SQLJ Translator

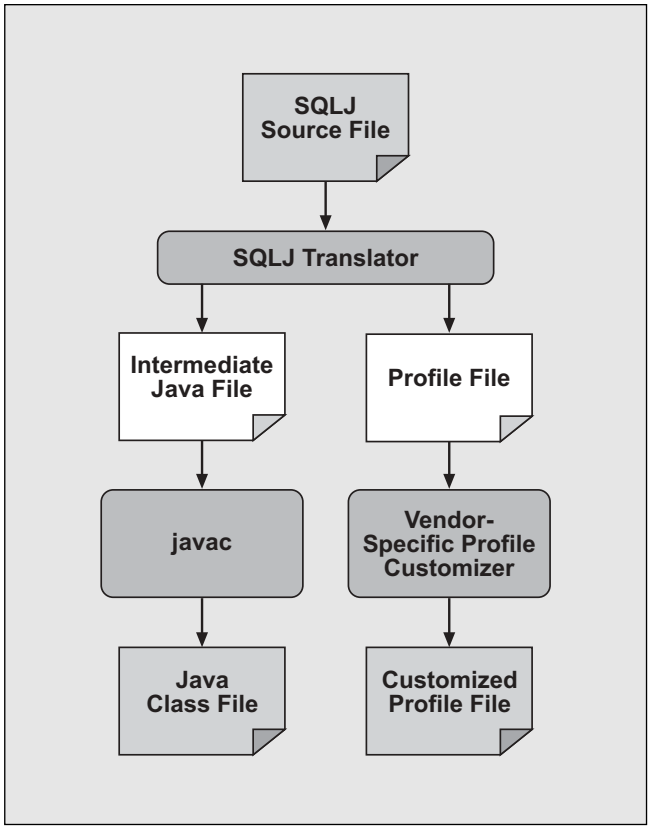
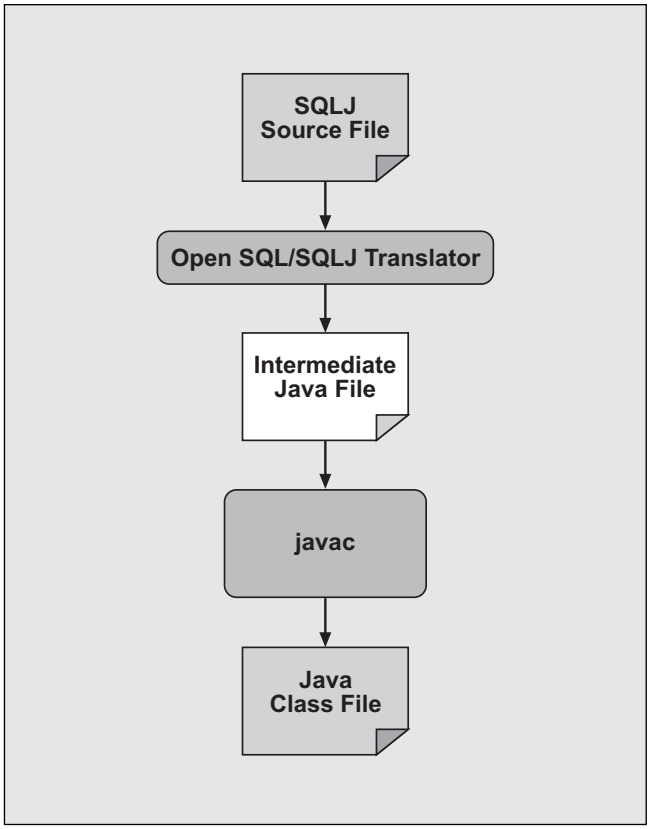


Figure 3 The Open SQL/SQLJ Translator



replaced by calls to the SQLJ runtime. The Java compiler translates this intermediate Java file to Java byte code, which can be executed by any Java virtual machine.

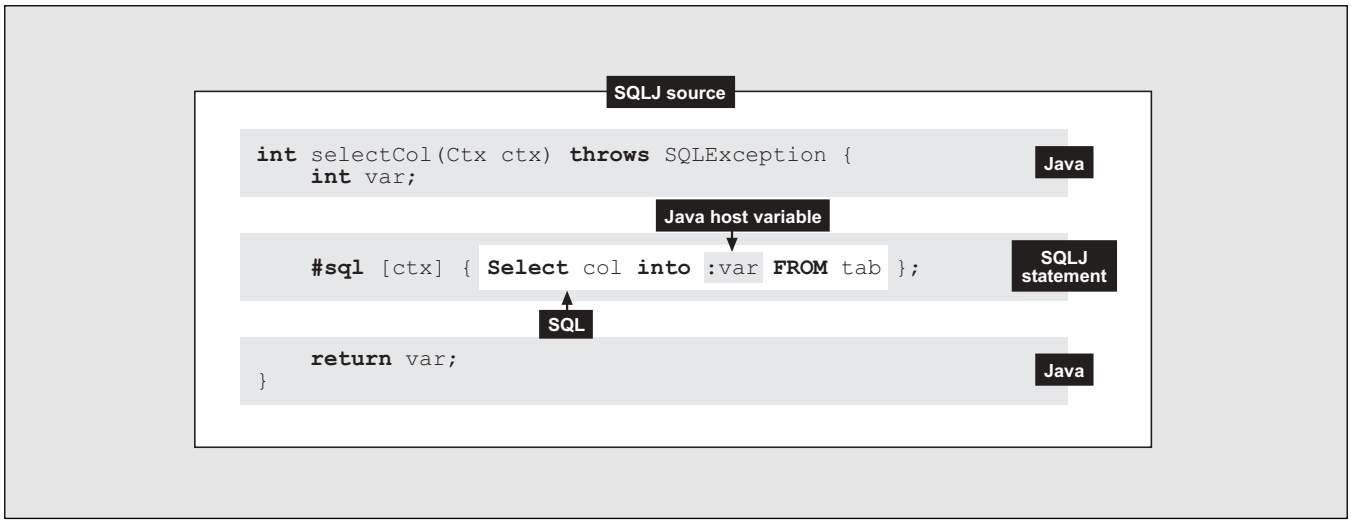
In the reference implementation of SQLJ, the intermediate Java files do not contain the embedded SQL statements. Instead, they are stored in separate profile files that are used at runtime for program execution. The intention was that these profile files would be customized for the database by a vendor-specific post-processor called a "profile customizer" in order to adapt the statements to the specifics of the vendor database (as shown in **Figure 2**). However, SQLJ does not specify the SQL subset that a profile customizer must be able to understand. Therefore, in the SQLJ reference implementation, the porting problem remains unsolved.

In the SAP implementation, the Open SQL/SQLJ translator takes a slightly different approach (as shown in **Figure 3**). The translator itself is capable of performing platform-independent syntax and semantics checks. Thus Open SQL/SQLJ requires no profile customization. The Java code produced by the Open SQL/SQLJ translator is entirely platform-independent and can be executed against the accompanying Open SQL/SQLJ runtime without any additional profile files.

At design time, the Open SQL/SQLJ translator performs syntax and semantics checks via the same SQL checker used by Open SQL/JDBC[5] at runtime. This ensures that Open SQL/SQLJ and Open SQL/JDBC use the same Open SQL grammar. The build environment automatically triggers the translation process. As a result, you never need to work with

[5] The Open SQL for Java framework also includes Open SQL/JDBC, which provides standard database access via JDBC with dynamic SQL.

Figure 4 *Embedded SQL in Java*



the intermediate Java file — you always modify the SQLJ source file.

SQLJ Statements and Syntax

Figure 4 shows the various elements of SQLJ source code. SQLJ statements start with the keyword *#sql* and end with a semicolon. The SQL text is enclosed in curly brackets (*{…}*). Java code, as well as the additional SQLJ keywords *#sql*, *context*, *iterator*, and *with*, are case-sensitive. However, SQL statement text is case-*in*sensitive. The SQL text may also contain "host variables," which are prefixed with a colon. Host variables are embedded Java variables that are used to exchange values between Java and SQL. Open SQL/SQLJ supports the full set of SQL statements specified by the Open SQL grammar, as well as single row queries, the transaction demarcation statements *COMMIT* and *ROLLBACK*, and the iterator conversion statement *CAST*.

Connecting to Databases via a Connection Context

A database connection specifies the database, the session, and the transaction to be used. In SQLJ, the database connection is represented by the "connection context." A connection context is an instance of a user-specified class that is declared using an SQLJ context declaration of the form:

```
#sql <modifiers> context
  <java class name> ;
```

From this declaration, the SQLJ translator generates a Java class *<java class name>* that implements the interface *sqlj.runtime.ConnectionContext*. This connection context is constructed from a JDBC connection, which is typically obtained from a JDBC *DataSource* looked up in the Java Naming and Directory Interface (JNDI):

```
Context jndiCtx =
  new InitialContext();

DataSource dataSource =
  (DataSource)jndiCtx.lookup(
  "jdbc/SQLJ_EXAMPLE");

Connection connection =
  dataSource.getConnection();

Ctx ctx = new Ctx(connection);
```

For convenience, you can directly associate a connection context class with a JDBC *DataSource*. In this case, the declaration takes the following form instead:

```
#sql <modifiers> context
  <java class name>
  [with ( dataSource =
  " <data source name> " )] ;
```

The default constructor of the context then automatically obtains the connection from the associated *DataSource*:

```
Ctx ctx = new Ctx();
```

In Open SQL/SQLJ, all queries and DML[6] statements must use an explicit connection context. In other words, these types of statements must contain an expression that designates the connection context object on which it will be executed. Otherwise, a syntax error will be issued.

Processing Result Sets with Iterators

In JDBC, you access the result set of a query through the JDBC *ResultSet* interface. This interface allows you to process result sets with any number of columns. However, it offers only generic getter methods that are not specific to the column type. Using a JDBC *ResultSet* can be very error-prone because type safety is not guaranteed. Any errors will only be detected at runtime.

SQLJ greatly facilitates processing result sets returned by queries. As opposed to JDBC, you do not need to access the JDBC result set directly. Instead, SQLJ allows you to declare a "result set iterator" with a well-defined (i.e., permanently defined at design time) number of strictly typed result set columns. SQLJ supports two types of result set iterators — "positional iterators" and "named iterators." The positional iterator specifies only the types of the result set columns. The named iterator also specifies the column names and has accessor methods with the same names as the result set columns. These named accessor methods make named iterators much more useful and easy to use than positional iterators. Therefore, we concentrate on named iterators in our examples.

[6] Data Manipulation Language (DML).

You declare named iterators using the following syntax:

```
#sql <modifiers> iterator
   <java class name> (
   <java datatype> <java id>
(, <java datatype> <java id>)* );
```

From this declaration, the SQLJ translator generates a Java class *<java class name>* with named accessor methods that retrieve the values of the result set column with the corresponding name. Iterator compatibility with the result set column is automatically checked at design time. Thus an iterator is a very convenient and type-safe way to access the result set columns.

Putting Open SQL/SQLJ to Work with Some Examples

Now that you are familiar with the basic concepts of SQLJ, let's take a closer look at Open SQL/SQLJ at work. To provide a practical foundation for learning, we will walk through a series of examples that illustrate the key features of Open SQL/SQLJ.

In the following examples, we want to focus on how to use Open SQL/SQLJ. To avoid detracting from the persistence code, we use small example Java classes to demonstrate the interesting features. These classes all implement the interface *SqljExample* (see **Figure 5**), which defines a *run* method with the signature in line 8[7]:

```
public void run(PrintWriter pw)
  throws SQLException;
```

The *run()* method takes *PrintWriter pw* as a parameter, and the example classes write their simple output using the *println* method of the *PrintWriter* supplied.

[7] Note that line numbers have been added to the code listings for your convenience.

Figure 5 The Interface SqljExample (SqljExample.java)

```
1 package com.sap.sqlj.examples;
2
3 import java.io.PrintWriter;
4 import java.sql.SQLException;
5
6 public interface SqljExample {
7
8   public void run(PrintWriter pw) throws SQLException;
9 }
```

In the last section of this article, you will learn to build a suitable framework that wraps the example classes discussed in the article in a Java servlet that executes in Web AS 6.30 and displays the output in a web browser.

The Data Model Used for the Examples

Our examples use a simple data model that consists of two database tables — *TMP_SQLJ_EMP* (shown in **Figure 6**), which contains employee data, and

Figure 6 Table TMP_SQLJ_EMP in the NetWeaver Developer Studio Table Editor

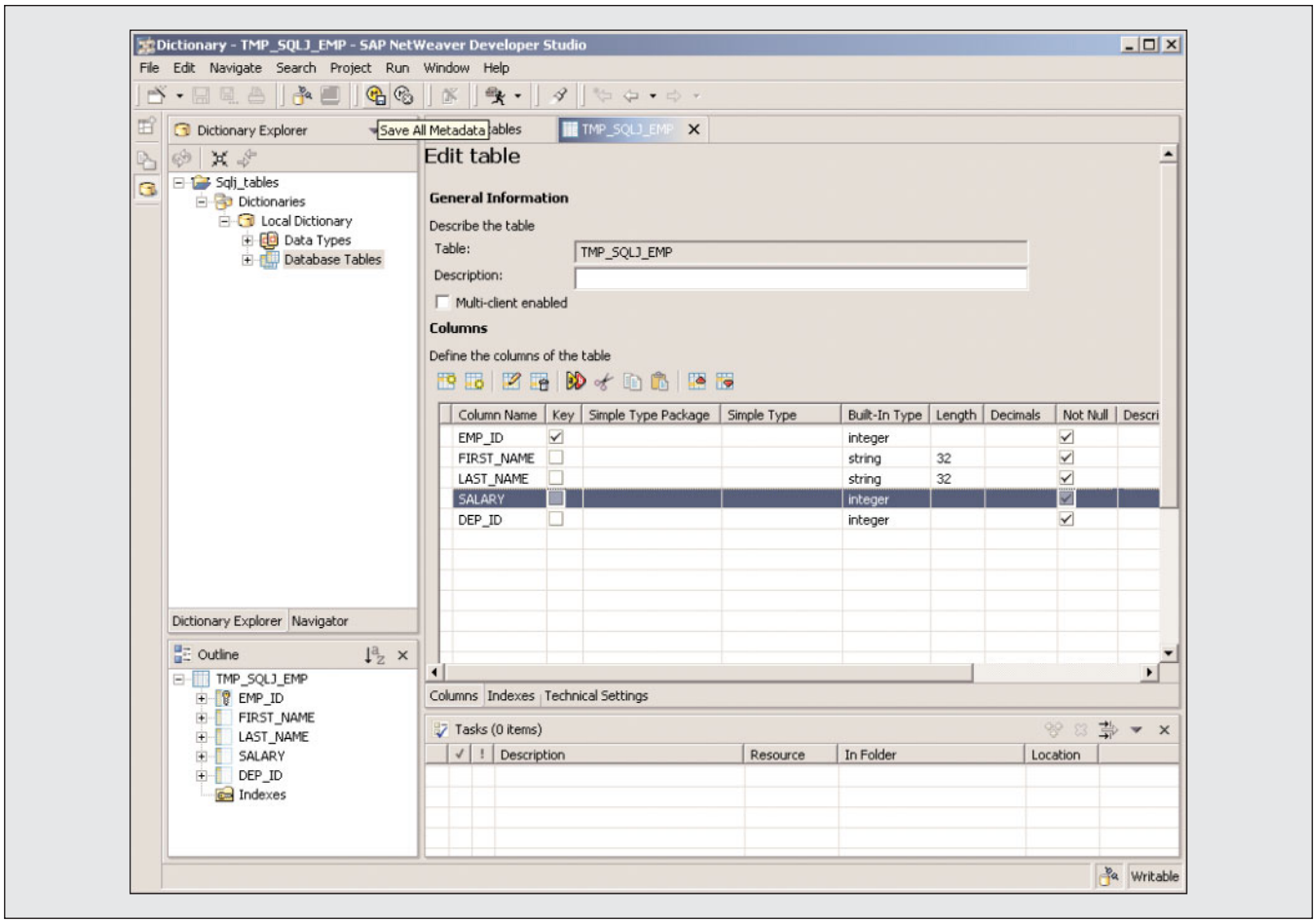
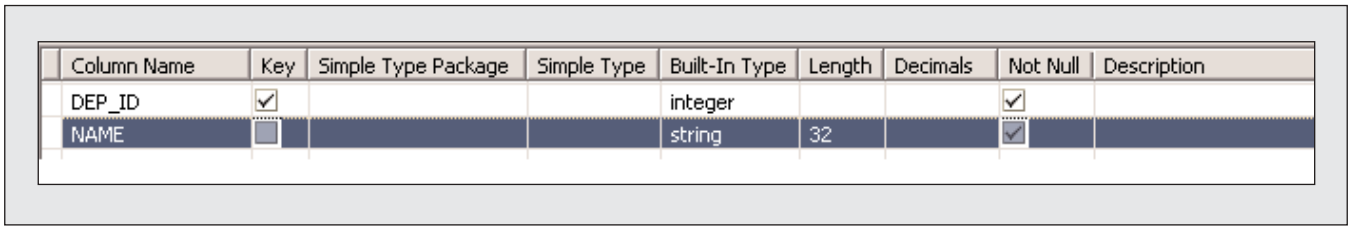


Figure 7 Table TMP_SQLJ_DEP in the NetWeaver Developer Studio Table Editor



Column Name	Key	Simple Type Package	Simple Type	Built-In Type	Length	Decimals	Not Null	Description
DEP_ID	☑			integer			☑	
NAME	☐			string	32		☑	

TMP_SQLJ_DEP (shown in **Figure 7**), which contains department data. The two tables are related by the column *DEP_ID*. (We will look at how to create these tables in a later section.)

Connection Context for the Examples

All of our example classes connect to the database using a JDBC data source that can be looked up in the JNDI under the name *jdbc/SQLJ_EXAMPLE*. To access this data source, in line 3 of **Figure 8** (*Ctx.sqlj*) we declare a connection context class named *Ctx* that all of the examples will use:

```
#sql context Ctx with
  (dataSource =
  "jdbc/SQLJ_EXAMPLE");
```

This declaration associates the connection context class *Ctx* with the data source *jdbc/SQLJ_EXAMPLE*. From this declaration, the SQLJ translator generates a Java class named *Ctx*. The default constructor of this class automatically looks up *jdbc/SQLJ_EXAMPLE* in JNDI and obtains a connection from the data source.

Figure 8 The Connection Context (Ctx.sqlj)

```
1 package com.sap.sqlj.examples;
2
3 #sql context Ctx with (dataSource
    = "jdbc/SQLJ_EXAMPLE");
```

Example #1: Executing DML Statements

In our first example, we illustrate the basic technique of using SQLJ for executing SQL statements on the database. The DML statements *INSERT*, *UPDATE*, and *DELETE* are a good place to start because they produce no result set.

The example in **Figure 9** (*SimpleStatement.sqlj*) initializes the data model for the examples that follow. To ensure reproducible results, we first erase all data from the database tables *TMP_SQLJ_EMP* and *TMP_SQLJ_DEP*. We then create an entry for the *Human Resources* department in the table *TMP_SQLJ_DEP*.

In order to access the database, we first create an instance *ctx* of the connection context class *Ctx*. Because the connection context class *Ctx* is associated with a data source, we can obtain a connection simply by calling its default constructor in line 10:

```
Ctx ctx = new Ctx();
```

Next, using the connection context *ctx*, in lines 14-26 we execute the SQLJ statements that are necessary to initialize the data model on the database:

```
#sql [ctx]
  { delete from TMP_SQLJ_DEP };
[...]
#sql [ctx]
  { delete from TMP_SQLJ_EMP };
[...]
#sql [ctx]
  { insert into TMP_SQLJ_DEP
    (dep_id, name)
    values ( 1, 'Human Resources') };
```

Remember that every executable SQLJ statement starts with the token *#sql*. Immediately after

Figure 9 Source Code for the DML Statements Example (SimpleStatement.sqlj)

```
 1 package com.sap.sqlj.examples;
 2
 3 import java.io.PrintWriter;
 4 import java.sql.SQLException;
 5 import sqlj.runtime.ExecutionContext;
 6
 7 public class SimpleStatement implements SqljExample{
 8
 9     public void run(PrintWriter pw) throws SQLException {
10         Ctx ctx = new Ctx();
11         ExecutionContext ecx = ctx.getExecutionContext();
12
13         try {
14             #sql [ctx] { delete from TMP_SQLJ_DEP };
15             int rc = ecx.getUpdateCount();
16             pw.println(rc +
17     " row[s] deleted from table TMP_SQLJ_DEP. <BR>");
18
19             #sql [ctx] { delete from TMP_SQLJ_EMP };
20             rc = ecx.getUpdateCount();
21             pw.println(rc +
22     " row[s] deleted from table TMP_SQLJ_EMP. <BR>");
23
24             #sql [ctx] {
25                 insert into TMP_SQLJ_DEP (dep_id, name)
26                    values ( 1, 'Human Resources') };
27             rc = ecx.getUpdateCount();
28             pw.println(rc +
29     " row[s] inserted into table TMP_SQLJ_DEP. <BR>");
30             #sql [ctx] { commit work };
31         } finally {
32             ctx.close();
33         }
34     }
35 }
```

this token, you specify the connection context instance *[ctx]* on which the statement is to be executed. The SQL statement text is embedded in curly brackets, and the SQLJ statement is terminated with a semicolon.

In the J2EE server, all applications share the same connection pool, so we must not forget to close the connection context after usage. To ensure that the used connections are returned to the pool under any circumstances, in lines 31-33 we call the *close()* method in the *finally* block corresponding to the *try* block that encloses the SQLJ statements (lines 13-33):

```
try {
[...]
} finally {
    ctx.close();
}
```

Closing the connection context in the *finally* block ensures that the underlying database connection, which is a scarce resource, is always closed and the associated resources are freed.

Example #2: Specifying Parameters in SQL Statements with Host Variables

Very rarely (such as in the previous example) you can statically specify all values contained in an SQL statement by SQL literals. More commonly, SQL statements require the use of parameters that are assigned at runtime. In SQLJ, Java variables or even complex Java expressions can be parameters of SQL statements. You embed these host variables and host expressions directly in the SQL statement text. In the Open SQL framework, the SQLJ translator has access to an offline representation of the database schema that contains the JDBC types for all database columns involved in the SQLJ statement. It uses this type information to perform strict type checks to ensure that the Java type of the host variable or the result type of the host expression can be safely used at its syntactical position in the SQL statement.

Figure 10 Source Code for the Host Variables Example (HostvarStatement.sqlj)

```
 1 package com.sap.sqlj.examples;
 2
 3 import java.io.PrintWriter;
 4 import java.sql.SQLException;
 5
 6 public class HostvarStatement implements SqljExample {
 7     private Ctx ctx = null;
 8
 9     private void insertEmployee(
10         PrintWriter pw,
11         int empId,
12         String firstName, String lastName,
13         int salary, int depId)
14         throws SQLException {
15         #sql [ctx] {
16             insert into TMP_SQLJ_EMP
17                 (EMP_ID, FIRST_NAME, LAST_NAME,
18                  SALARY, DEP_ID)
19                values
20                (:empId, :firstName, :lastName,
21                 :salary, :depId) };
22         pw.println("Employee " +
23             lastName + ", " + firstName +
24             " inserted.<BR>");
25     }
26
27     public void run(PrintWriter pw) throws SQLException {
28         String[] departments = {
29             "Financials",
30             "Research and Development" };
31
32         ctx = new Ctx();
33
```


In our next example, we use host variables to further populate *TMP_SQLJ_DEP*, the table of departments. This technique allows us to use a single *INSERT* statement that is executed with different parameters for each row to be inserted. An array named *departments* contains the names of the departments to be inserted.

As in the previous example (Figure 9), in **Figure 10** *(HostvarStatement.sqlj*) we start by obtaining a connection context instance named *ctx* in lines 28-32:

```
String[] departments = {
  "Financials",
  "Research and Development" };

ctx = new Ctx();
```

To insert the data, we loop over the array *departments* and insert each array element. On every iteration, we assign the department name to the Java host variable *name*. To demonstrate the use of a host expression, we also inserted a Java expression (the loop counter *i + 2*) as the department ID. Before the

Figure 10 (continued)

```
34          try {
35              for (int i = 0; i < departments.length; i++) {
36                  String name = departments[i];
37                  #sql [ctx] {
38                      insert into TMP_SQLJ_DEP
39                                (dep_id, name)
40                      values (:(i + 2), :name) };
41                  pw.println("Department "
42                        + name
43                        + " ("
44                        + (i + 2)
45                        + ") inserted.<BR>");
46              }
47                 pw.println("<BR>");
48              insertEmployee(pw, 1,
49                   "Johnny", "Weissmueller", 1000, 1);
50              insertEmployee(pw, 2,
51                   "Mark", "Spitz", 2000, 2);
52              insertEmployee(pw, 3,
53                   "Jenny", "Thompson", 350, 2);
54              insertEmployee(pw, 4,
55                   "Janet", "Evans", 350, 2);
56              insertEmployee(pw, 5,
57                   "Michael", "Gross", 4000, 3);
58              insertEmployee(pw, 6,
59                   "Grant", "Hacket", 100, 4);
60              insertEmployee(pw, 7,
61                   "Vladimir", "Salnikov", 500, 4);
62
63              #sql [ctx] { commit work };
64          } finally {
65              ctx.close();
66          }
67      }
68  }
```

statement is executed, the Java expression is evaluated and the result taken as the input value (lines 35-46):

```
for (int i = 0;
     i < departments.length;
     i++) {
  String name = departments[i];
  #sql [ctx] {
    insert into TMP_SQLJ_DEP
    (dep_id, name)
    values (:(i + 2), :name) };
[...]
}
```

To fill the data model with some employees, we create an *insertEmployee()* method (lines 9-25):

```
private void insertEmployee(
  PrintWriter pw, int empId,
  String firstName,
  String lastName,
  int salary, int depId)
  throws SQLException {
  #sql [ctx] {
    insert into TMP_SQLJ_EMP
    (EMP_ID,
     FIRST_NAME, LAST_NAME,
     SALARY, DEP_ID)
    values
    (:empId,
     :firstName, :lastName,
     :salary, :depId) };
  pw.println(
    "Employee " +
    lastName + ", " + firstName +
    " inserted.<BR>");
}
```

This method will be called with different arguments for each employee to be inserted. Again, we must not forget to close the connection context in the *finally* block (lines 34-66):

```
try {
[...]
} finally {
  ctx.close();
}
```

If you have used JDBC before, this simple example strikingly illustrates the benefits of using the SQLJ host variable approach instead of the JDBC *setXxx()* methods. In JDBC, correctly determining the ordinal position and type of the host variables in an SQL statement can be very error-prone. This problem is especially prevalent if the number of parameters changes during development. In SQLJ, you are completely relieved from this burden because the respective checks are performed at design time.

Example #3: Retrieving Single Row Result Sets with a Single Row Query

So far, we have only executed DML statements that manipulate data. Now, we are going to use simple queries to retrieve data from the database. If the result set returned by a query contains only one row, you can fetch the result set columns directly into host variables without using a result set iterator. In SQLJ, you can easily perform this operation with a "single row query." Typically, you use this technique to retrieve a single row of data that is uniquely identified by its primary key.

> ✓ ***Tip***
>
> *In contrast to a general SELECT statement, a single row query must not be a UNION and must not contain a GROUP BY, HAVING, or ORDER BY clause.*

In **Figure 11** *(SingleSelect.sqlj)*, we use a single row query to determine the first name and last name of an employee with a given ID *empId* (lines 13-25):

Figure 11 Source Code for the Single Row Query Example (SingleSelect.sqlj)

```
 1 package com.sap.sqlj.examples;
 2
 3 import java.io.PrintWriter;
 4 import java.sql.SQLException;
 5
 6 public class SingleSelect implements SqljExample{
 7
 8     public void run(PrintWriter pw) throws SQLException {
 9         Ctx ctx = new Ctx();
10         try {
11             int empId = 5;
12             String firstName, lastName;
13             #sql [ctx] {
14                 select FIRST_NAME, LAST_NAME
15                     into :firstName, :lastName
16                     from TMP_SQLJ_EMP
17                     where emp_id = :empId };
18             pw.println(
19                 "The name of the employee with the id "
20                     + empId
21                     + " is "
22                     + firstName
23                     + " "
24                     + lastName
25                     + ".<BR>");
26
27             int maximum;
28             #sql [ctx] { select max(salary)
29                                 into :maximum
30                                 from TMP_SQLJ_EMP };
```

(continued on next page)

```
#sql [ctx] {
  select FIRST_NAME, LAST_NAME
    into :firstName, :lastName
    from TMP_SQLJ_EMP
    where emp_id = :empId };
  pw.println(
    "The name of the employee
    with the id " + empId +
    " is " + firstName +
    " " + lastName +
    ".<BR>");
```

However, a single row query is not restricted to retrieving data with a fully specified primary key. You can use it for any statement that returns a result set with exactly one row. In our example, we use another single row query to determine the maximum salary of all employees (lines 28-30):

```
#sql [ctx] {
  select max(salary)
    into :maximum
    from TMP_SQLJ_EMP };
```

> ✓ ***Tip***
>
> *According to SQL syntax, the INTO clause must immediately follow the SELECT list.*

Figure 11 (continued)

```
31              pw.println(
32                   "The maximum salary of all employees is "
33                   + maximum + ".");
34          } finally {
35              ctx.close();
36          }
37      }
38  }
```

Example #4: Processing Multi-Row Result Sets with a Named Iterator

Generally, a query produces a result set with multiple rows. SQLJ offers a powerful tool called a "named iterator" for processing a multi-row result set in a convenient way. A named iterator is a Java class that can be thought of as an SQL cursor with a well-defined number of strictly typed and named columns. In this next example, we use a named iterator to retrieve the names of all employees.

In **Figure 12** *(MultiSelect.sqlj)*, we start by declaring the named iterator class *EmployeeIter* with the columns *emp_id*, *first_name*, and *last_name* (lines 8-9):

```
#sql private static iterator
EmployeeIter (int    emp_id,
              String first_name,
              String last_name);
```

From this declaration, the SQLJ translator generates an iterator class *EmployeeIter* with the following three accessor methods (with names that match the columns): *emp_id()*, *first_name()*, and *last_name()*.

Because the iterator is used only in this class, we declared it as a private static inner class of *MultiSelect*. If you intend to use an iterator class in other classes, you would declare an iterator in a separate source file as a (public) top-level class, allowing reuse of the iterator class.

In order to use the iterator, we declare an instance variable *iter* of type *EmployeeIter*. Then we assign the result of a query to *iter* (lines 14-18):

```
EmployeeIter iter;

#sql [ctx] iter = {
  select emp_id, first_name,
  last_name
    from TMP_SQLJ_EMP };
```

After this assignment, the iterator points to the position before the first row of the result set. You typically process a named iterator in a loop (lines 20-29):

```
while(iter.next()) {
  pw.println(iter.emp_id() +
    " " + iter.first_name() +
    " " + iter.last_name() +
    "<BR>");
}
iter.close();
```

On every iteration, the *next()* method advances the iterator to the next row of the result set. Then the generated accessor methods *emp_id()*, *first_name()*, and *last_name()* extract the values from the respective result set columns. The loop terminates when the return value of the *next()* method indicates that no more result set rows are available.

As in our previous examples, we close the connection context in the *finally* block (lines 30-32).

Figure 12 Source Code for the Named Iterator Example (MultiSelect.sqlj)

```
 1 package com.sap.sqlj.examples;
 2
 3 import java.io.PrintWriter;
 4 import java.sql.SQLException;
 5
 6 public class MultiSelect implements SqljExample{
 7
 8     #sql private static iterator EmployeeIter
 9       (int emp_id, String first_name, String last_name);
10
11     public void run(PrintWriter pw) throws SQLException {
12         Ctx ctx = new Ctx();
13         try {
14             EmployeeIter iter;
15
16             #sql [ctx] iter = {
17                  select emp_id, first_name, last_name
18                      from TMP_SQLJ_EMP };
19
20             while(iter.next()) {
21                  pw.println(
22                        iter.emp_id()
23                        + " "
24                        + iter.first_name()
25                        + " "
26                        + iter.last_name()
27                        + "<BR>");
28             }
29             iter.close();
30         } finally {
31             ctx.close();
32         }
33     }
34 }
```

Example #5: Obtaining Return Values from the Execution Context

Let's revisit our first example (refer back to Figure 9). There, in line 14, we used a *DELETE* statement as an example of a simple SQLJ statement:

```
#sql [ctx] {
  delete from TMP_SQLJ_DEP };
```

If this statement fails to execute on the database, an *SQLException* will be thrown. We could catch this exception and handle the error situation appropriately. But if the statement executes successfully, how would we determine the number of database rows affected by the statement? In JDBC, this information is returned by the *executeUpdate()* method of the JDBC *Statement* used to execute the *DELETE*. However, in SQLJ, a *DELETE* statement has no return value.

You obtain this information in SQLJ in a

different way. Every executable SQLJ statement (*SELECT*, *INSERT*, *UPDATE*, *DELETE*) runs in an "execution context," which is an instance of the class *sqlj.runtime.ExecutionContext*. The execution context gives you control over certain parameters that affect the execution of a statement, such as the maximum number of rows to be returned by a query. It also allows you to retrieve information related to the execution of a statement, like warnings returned by the database or, as in our example, the update count. If not otherwise specified, a statement runs in the default execution context of the current connection context.

In our example, we execute the *DELETE* statements using the default execution context, which is obtained from the *getExecutionContext()* method of the connection context (line 11 in Figure 9):

```
ExecutionContext ecx =
  ctx.getExecutionContext();
```

From this execution context, we obtain the update count of the *DELETE* statement executed through the *getUpdateCount()* method (lines 14-17 in Figure 9):

```
#sql [ctx] {
  delete from TMP_SQLJ_DEP };
  int rc = ecx.getUpdateCount();
  pw.println(rc +
" row[s] deleted [...]. <BR>");
```

Example #6: Achieving Better Performance via Batch Updates

In a previous example (refer back to Figure 10), we inserted data consecutively for multiple employees into the table *TMP_SQLJ_EMP*. For each employee, we executed a separate *INSERT* statement. All of these statements are basically identical, but parameterized differently. However, every execution requires multiple roundtrips between the application server and the database. First the statement is prepared, then the parameter (host variable) data is transferred, and finally the statement is executed.

You can dramatically improve performance by using "batch updates" instead of a series of identical statements. With this technique, the update statement is prepared once. Next, the parameter values of all statements in the batch are collected on the application server and sent to the database in a single roundtrip. Finally, the database executes the statement using the parameter batch. Thus the batch execution requires only one roundtrip, and the database can process the data with a highly efficient array operation.

Batch updates are a new, optional feature of the SQLJ standard; individual SQLJ implementations are not required to support it. In SQLJ, the execution context of a statement controls batching. Therefore, to use batching according to the SQLJ standard, you must follow these steps:

1. Enable batching on the execution context.
2. Execute a series of batchable statements that are mutually batch-compatible.
3. Execute the parameter batch on the execution context.
4. Disable batching on the execution context.

Unfortunately, the SQLJ standard does not specify exactly which statements are batchable, nor does it define the criteria for batch compatibility. Therefore, if you develop persistence code according to the SQLJ standard, you cannot rely on statement batching. The standard requires further extension in order to precisely define the criteria for batchability.

Even though batching is an optional SQLJ feature, SAP has chosen to support it in Open SQL/SQLJ. The implementation is compatible with the SQLJ standard, but the criteria for batchability are precisely defined, and the usage is slightly simplified. All DML statements are batchable in Open SQL/SQLJ. However, they are batchable only if executed via a special subclass (*com.sap.sql.BatchExecutionContext*) of the class *sqlj.runtime.ExecutionContext*. On any instance of *BatchExecutionContext*, batching is an immutable property of the class — it is always enabled and cannot be switched off. To ensure that all statements

in a batch are compatible, Open SQL/SQLJ requires a batch to comprise only statements with identical source code position. In other words, each batched statement must occupy the same position in the source code.

Therefore, to use batching in Open SQL/SQLJ, you must follow these steps:

1. Create an instance *bctx* of *BatchExecutionContext*.
2. Execute a series of statements with identical source code positions on *bctx*.
3. Execute the batch on *bctx*.

In this next example, we modify the previous example (once again, refer back to Figure 9) to use batch updates. Here we use batching to more efficiently insert additional employees into the table *TMP_SQLJ_EMP*. In **Figure 13** (*Batching.sqlj*), we start by creating a connection context and an instance of *BatchExecutionContext* (lines 30-31):

```
ctx = new Ctx();
bctx =
  new BatchExecutionContext();
```

The only necessary change to the *insertEmployee()* method is to denote the batch execution context *bctx* in the square brackets of the context clause (lines 17-23):

```
#sql [ctx, bctx] {
  insert into TMP_SQLJ_EMP
  (EMP_ID, FIRST_NAME, LAST_NAME,
   SALARY, DEP_ID)
  values
  (:empId, :firstName, :lastName,
   :salary, :depId) };
```

All consecutive calls of the *insertEmployee()* method add employee data to the batch, instead of directly inserting the data into the database table. Before committing the changes, we must execute the batch using the *executeBatch()* method of the batch execution context *bctx* (lines 47-49):

```
bctx.executeBatch();

#sql [ctx] { commit work };
```

Figure 13 Source Code for the Batching Example (Batching.sqlj)

```
 1 package com.sap.sqlj.examples;
 2
 3 import java.io.PrintWriter;
 4 import java.sql.SQLException;
 5 import com.sap.sql.BatchExecutionContext;
 6
 7 public class Batching implements SqljExample{
 8     private Ctx ctx = null;
 9     private BatchExecutionContext bctx = null;
10
11      private void insertEmployee(
12          PrintWriter pw,
13         int empId,
14         String firstName, String lastName,
15         int salary, int depId)
16         throws SQLException {
```

(continued on next page)

Figure 13 (continued)

```
17          #sql [ctx, bctx] {
18              insert into TMP_SQLJ_EMP
19                  (EMP_ID, FIRST_NAME, LAST_NAME,
20                   SALARY, DEP_ID)
21                  values
22                  (:empId, :firstName, :lastName,
23                   :salary, :depId) };
24          pw.println("Employee " +
25              lastName + ", " + firstName +
26              " inserted.<BR>");
27      }
28
29      public void run(PrintWriter pw) throws SQLException {
30          ctx = new Ctx();
31          bctx = new BatchExecutionContext();
32
33          try {
34              insertEmployee(pw, 8,
35                   "Ian", "Thorpe", 1100, 1);
36              insertEmployee(pw, 9,
37                   "Giorgio", "Lamberti", 3000, 2);
38              insertEmployee(pw, 10,
39                   "Franziska", "van Almsick", 750, 2);
40              insertEmployee(pw, 11,
41                   "Michael", "Phelps", 3000, 3);
42              insertEmployee(pw, 12,
43                   "Joerg", "Hoffmann", 100, 4);
44              insertEmployee(pw, 13,
45                   "Hannah", "Stockbauer", 700, 4);
46
47              bctx.executeBatch();
48
49              #sql [ctx] { commit work };
50          } finally {
51              ctx.close();
52          }
53      }
54  }
```

Example #7: Mixing SQLJ with JDBC to Implement Dynamic SQL

In SQLJ, the statement text of the embedded SQL is static and cannot be changed at runtime. Obviously this approach is insufficient, because you sometimes need the ability to use dynamic SQL. For example, you might want to select all rows of a database table matching a collection of key values. You cannot specify this type of query with static SQL in SQLJ. Therefore, you need to use JDBC for dynamic SQL.

Fortunately, Open SQL/SQLJ and Open SQL/JDBC use the same SQL checker and execute statements using the same runtime, so the two can seamlessly interact with each other. Open SQL/SQLJ and Open SQL/JDBC can share the same database connection and database transaction, and SQLJ result set iterators and JDBC result sets are mutually convertible. As a result, one logical unit of work can comprise both SQLJ and JDBC operations.

In this final example, we illustrate the cooperation between SQLJ and JDBC. Suppose we want to derive a list of employees who work in a particular set of departments. The departments are identified by their IDs, which are contained in the array *depArray* that is known only at runtime. For the purposes of this example, let's assume that the size of this array is also unknown, so we cannot specify the query statically. Instead, in **Figure 14** *(DynamicSQL.sqlj*), we dynamically create an SQL statement that contains the department IDs as an *IN* condition of the *WHERE* clause (lines 19-27):

```
StringBuffer stmtText =
  new StringBuffer();
stmtText.append(
  "SELECT first_name, last_name
   FROM TMP_SQLJ_EMP
   WHERE emp_id IN (");
for(int i = 0;
    i < depArray.length;
    i++) {
stmtText.append(Integer.toString(
  depArray[i]));
  if (i + 1 != depArray.length)
    stmtText.append(",");
  }
}
stmtText.append(")");
```

Figure 14 Source Code for the Dynamic SQL Example Using SQLJ and JDBC (DynamicSQL.sqlj)

```
 1 package com.sap.sqlj.examples;
 2
 3 import java.io.PrintWriter;
 4 import java.sql.SQLException;
 5 import java.sql.Connection;
 6 import java.sql.Statement;
 7 import java.sql.ResultSet;
 8
 9 class DynamicSQL implements SqljExample {
10
11     #sql private static iterator EmployeeIter (String first_name,
           String last_name);
12
13     public void run(PrintWriter pw) throws SQLException {
14             EmployeeIter iter;
15         int[] depArray = { 1, 3 }; // supposed to be known only at runtime
16
17         Ctx ctx = new Ctx();
18
19         StringBuffer stmtText = new StringBuffer();
20         stmtText.append("SELECT first_name, last_name FROM TMP_SQLJ_EMP
               WHERE emp_id IN (");
21         for(int i = 0; i < depArray.length; i++) {
22             stmtText.append(Integer.toString(depArray[i]))
```

(continued on next page)

Figure 14 (continued)

```
23              if (i + 1 != depArray.length)
24                  stmtText.append(",");
25              }
26          }
27          stmtText.append(")");
28
29          Connection conn = ctx.getConnection();
30          Statement stmt = conn.createStatement();
31          try {
32              ResultSet rs = stmt.executeQuery(stmtText.toString());
33              #sql iter = { CAST :rs };
34              try {
35                  while(iter.next()) {
36                      pw.println(
37                      iter.first_name() + " " +
38                      iter.last_name() + "<BR>");
39                  }
40              } finally {
41                  iter.close();
42              }
43          } finally {
44              stmt.close();
45          }
46      }
47 }
```

To execute the statement via JDBC, we obtain the JDBC connection *conn* underlying the SQLJ connection context *ctx* using the *getConnection()* method of the connection context. From this connection, we create a statement object *stmt* on which we execute the dynamically created query that returns the JDBC result set *rs* (lines 29-45):

```
Connection conn =
  ctx.getConnection();
Statement stmt =
  conn.createStatement();
try {
  ResultSet rs = stmt.executeQuery(
  stmtText.toString());
[...]
} finally {
  stmt.close();
}
```

The special SQLJ *CAST* statement converts a JDBC result set into an SQLJ iterator. Here, we convert the JDBC result set *rs* to the SQLJ iterator *iter* and process it as before (lines 33-42):

```
#sql iter = { CAST :rs };
try {
  while(iter.next()) {
    pw.println(
      iter.first_name() + " " +
      iter.last_name() + "<BR>");
    }
} finally {
  iter.close();
}
```

Closing the iterator *iter* and the statement *stmt* in *finally* blocks (lines 40-45), we ensure that the dependent resources (i.e., the underlying JDBC result set) get released in case of an error.

Now we'll show you how to run our Open SQL/SQLJ examples in your SAP environment.

Open SQL/SQLJ in SAP Web AS 6.30

There are no prerequisites for using Open SQL/SQLJ in Web AS 6.30. The Open SQL/SQLJ runtime is deeply integrated into Web AS 6.30 and is available out of the box. Open SQL/SQLJ runs on any Open SQL connection. All data sources created in the JDBC Connector Service of Web AS 6.30 return Open SQL connections by default. Therefore, you can use these connections directly with Open SQL/SQLJ.

Open SQL/SQLJ in the SAP NetWeaver Developer Studio

The NetWeaver Developer Studio supports primary development tasks such as editing, translating, and debugging Open SQL/SQLJ sources, as well as deploying the archives that are created as a result of the development project. In general, you can use SQLJ anywhere that you can use Java. The Open SQL/SQLJ translator is transparently integrated into the NetWeaver Developer Studio. It checks the SQL statements against an offline representation of the database schema (the "logical catalog"), which is provided to the Open SQL/SQLJ translator by a dictionary project that contains the schema of the tables created in this project.

Developing the Example Application

To help you get started, we will now show you, step by step, how we used Open SQL/SQLJ in the NetWeaver Developer Studio to develop our examples. For ease of use, we also provide a framework that wraps the examples in a simple web application so you can view their output in a browser. To develop, deploy, and execute an application that runs the examples in this article, follow this process:

1. Create and deploy the dictionary project and database tables.
2. Create the web application and the sources for the examples:
   - Create the project for the web application.
   - Create the HTML page.
   - Create the interface *SqljExample*.
   - Create the Java servlet.
   - Create the example classes.
3. Assemble and deploy the web application.
4. Run the application.

So, start the NetWeaver Developer Studio and follow along as we describe each of these steps.[8] To get the expected results, make sure to enter everything exactly as shown.

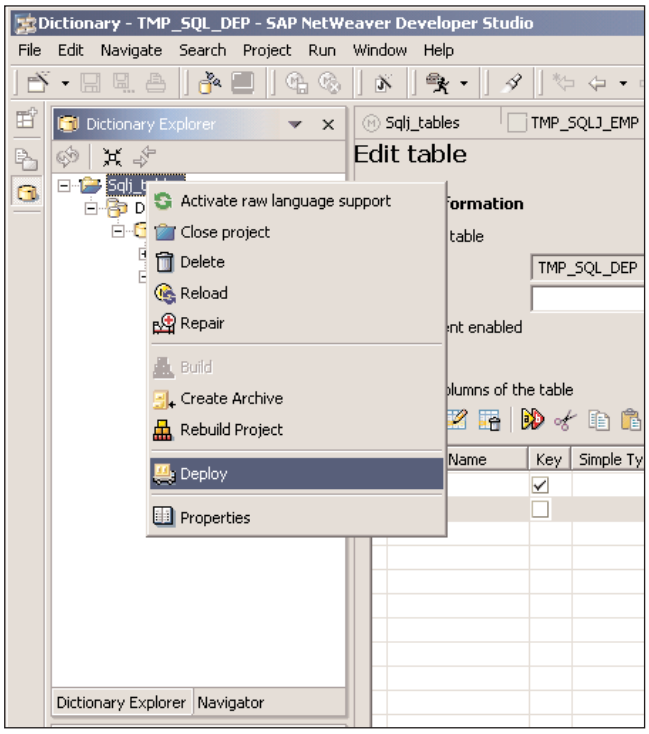
Creating and Deploying the Dictionary Project and Database Tables

The first step is to create and deploy a dictionary project. It contains the catalog information of the database tables used and provides this information to the Open SQL/SQLJ translator. To create a dictionary project, choose *File → New → Project → Dictionary → next* from the main menu. Enter *Sqlj_tables* as the project name and select *Finish*.

Next, declare the table definitions with the table editor. Open the project tree, mark the item *Database Tables*, and choose *create table* in the context menu.

[8] If you are new to the SAP NetWeaver Developer Studio, please refer to the product documentation for detailed instructions (see the sidebar on page 30).

Figure 15 Deploying the Dictionary Project



Choose the table names, column names, and data types as shown in Figures 6 and 7 (refer back to pages 9 and 10). Press the *Save All Metadata* button in the menu bar to save the table schema. Then deploy the table description on the Web AS to create the database tables on the database. Mark the project name *Sqlj_tables* in the project tree and choose *Create Archive* on the context menu.

Finally, to deploy the created archive into the Web AS, choose *Deploy* on the same *Sqlj_tables* context menu (as shown in **Figure 15**).

Creating the Web Application and Source Code for the Examples

As a framework for running the examples, we develop a simple web application that uses an HTML page as the user interface and a Java servlet to process user input and display the result. Here we show you how to create this application.

Start by creating a web project to organize the source files. From the main menu, choose *File → new → Project → J2EE → Web Project → next*. Enter *Sqlj_servlet* as the project name and select *Finish*.

Now we can develop our examples in this project. The first step is to create a new Java package. In the context menu of the project, choose *new → package*, enter *com.sap.sqlj.examples* as the package name, and select *Finish*. The J2EE Explorer view then opens, which is where you create the Java and SQLJ source files.

Begin by creating the HTML page from which you invoke the servlet:

1. From the main menu, choose *File → new → other → J2EE → Web → HTML File → next*.

2. Choose *webContent* as the folder and *index* as the page name (as shown in **Figure 16**).

3. Enter the HTML code as shown in **Figure 17**.

Figure 16 Creating the HTML Page

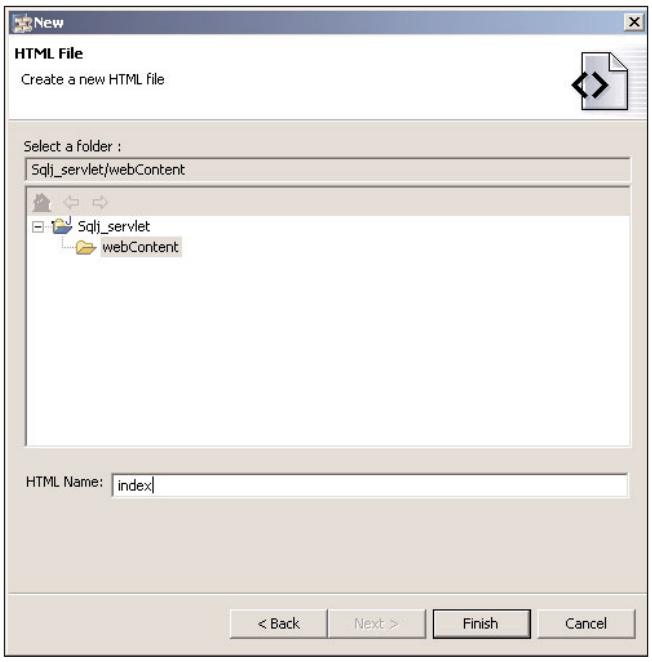


Figure 17 Source Code for the HTML Page (index.html)

```
<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.0 Transitional//EN">

<HTML>

<BODY BGCOLOR="#FFFFFF">
<hr>
<FORM METHOD="GET"
ACTION="./servlet/com.sap.sqlj.examples.SqljRunner">
<table>
<tr><td><INPUT TYPE="radio" checked name="examples"
    value="0">SimpleStatement</td></tr>
<tr><td><INPUT TYPE="radio"  name="examples"
    value="1">HostvarStatement</td></tr>
<tr><td><INPUT TYPE="radio"  name="examples"
    value="2">SingleSelect</td></tr>
<tr><td><INPUT TYPE="radio"  name="examples"
    value="3">MultiSelect</td></tr>
<tr><td><INPUT TYPE="radio"  name="examples"
    value="4">Batching</td></tr>
<tr><td><INPUT TYPE="submit" value="Execute">
</table>
</FORM>

<hr>

</BODY>

</HTML>
```

Next, create the interface *SqljExample* by following these steps:

1. From the main menu, choose *File → new → other → Java → Interface → next.*
2. Enter the appropriate package name and class name (*com.sap.sqlj.examples* and *SqljExample*), and select *Finish*.
3. Enter the source code for *SqljExample* (**Figure 18**).

Figure 18 Source Code for the Interface SqljExample (SqljExample.java)

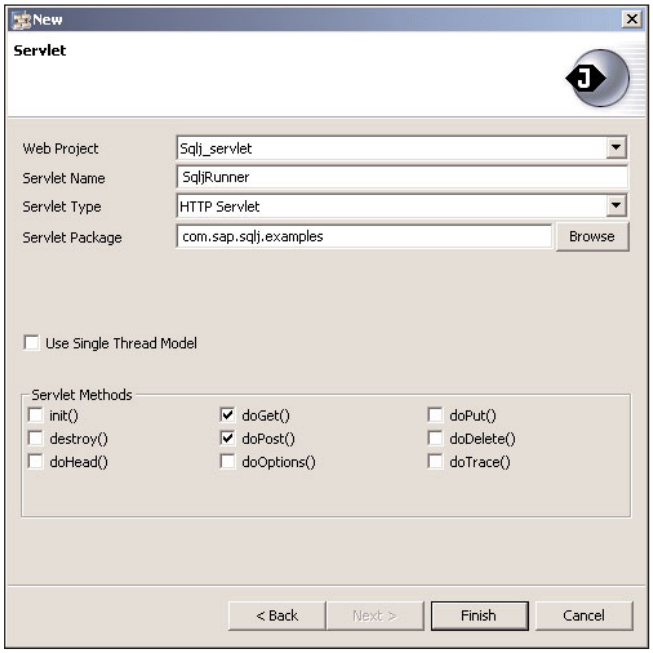
```
package com.sap.sqlj.examples;

import java.io.PrintWriter;
import java.sql.SQLException;

public interface SqljExample {

 public void run(PrintWriter pw) throws SQLException;
}
```

Figure 19 Creating the Java Servlet



Now follow these steps to create the Java servlet that processes the input data and displays the result:

1. From the main menu, choose *File → new → other → J2EE → Web → Servlet → next*.

2. Choose the names shown in **Figure 19** and select *Finish*.

Enter the source code for the class *SqljRunner* shown in **Figure 20**.

Before we can create the SQLJ source files, we must tell the SQLJ translator where to find the offline logical catalog data. To do this, we create a reference to the dictionary project *Sqlj_tables* (which we created earlier). On the context menu of the project *Sqlj_servlet*, choose *properties → Java Build Path → Projects*. Select the project *Sqlj_tables* and select *OK* (as shown in **Figure 21**).

Figure 20 Source Code for the Java Servlet (SqljRunner.java)

```
package com.sap.sqlj.examples;
import java.io.IOException;
import java.io.PrintWriter;

import javax.servlet.ServletException;
import javax.servlet.http.HttpServlet;
import javax.servlet.http.HttpServletRequest;
import javax.servlet.http.HttpServletResponse;

public class SqljRunner extends HttpServlet {

   protected void doGet(
         HttpServletRequest request,
         HttpServletResponse response)
         throws ServletException, IOException {

               PrintWriter out = response.getWriter();
               try {
                     SqljExample example = null;
                     Integer radio =
Integer.valueOf(request.getParameter("examples"));

                     switch (radio.intValue()) {
                        case 0: example = new SimpleStatement();
```


Figure 20 (continued)

```
                                    break;
                            case 1: example = new HostvarStatement();
                                    break;
                            case 2: example = new SingleSelect();
                                    break;
                            case 3: example = new MultiSelect();
                                    break;
                            case 4: example = new Batching();
                                    break;
                            case 5: example = new DynamicSQL();
                                   break;
                            default: out.println("invalid selection");

                        }

                        if(example!=null)
                            example.run(out);
                    }
                    catch(Exception ex) {
                           out.println("Example failed " + ex.getMessage());
                    }
        }
    }
```

Figure 21 *Declaring a Reference to the Dictionary Project*

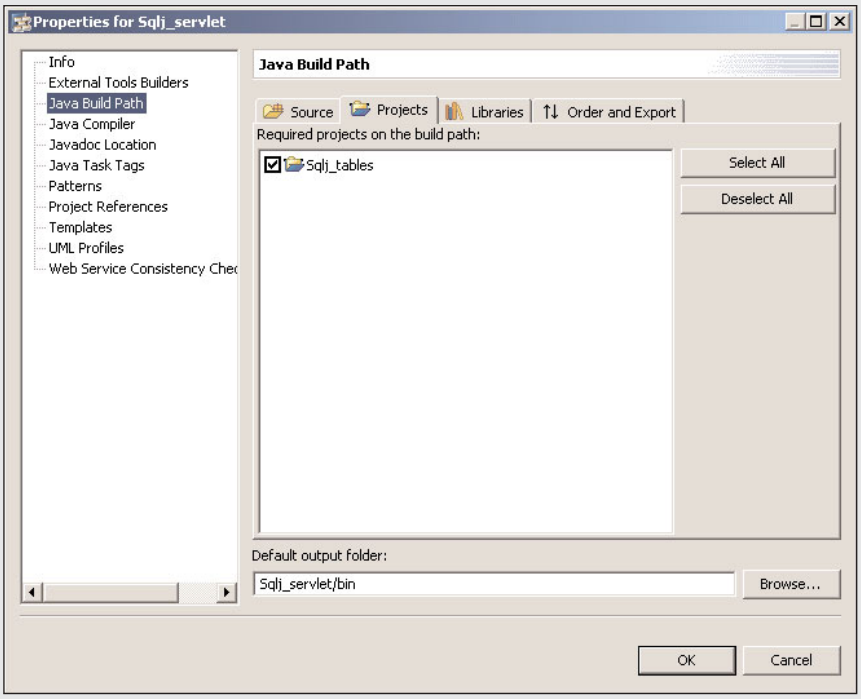


Figure 22 Creating the SQLJ Source

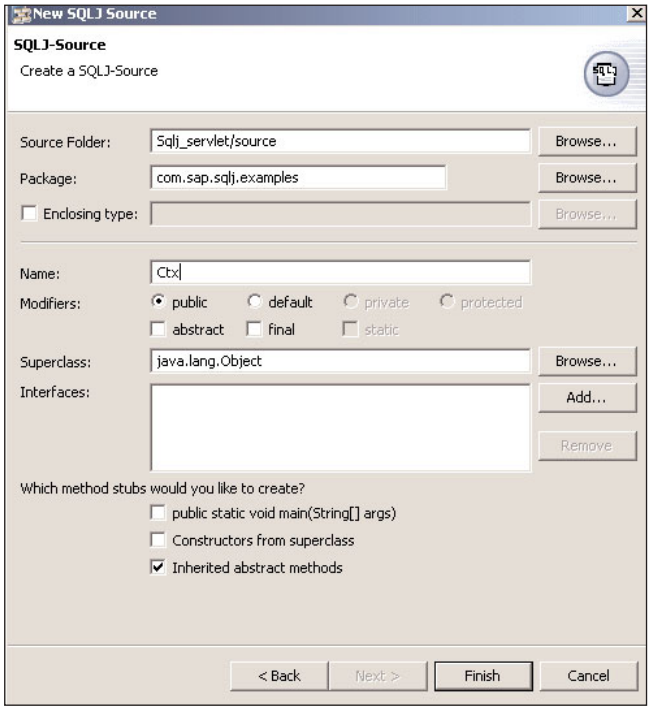


Figure 23 Editing the SQLJ Source Code

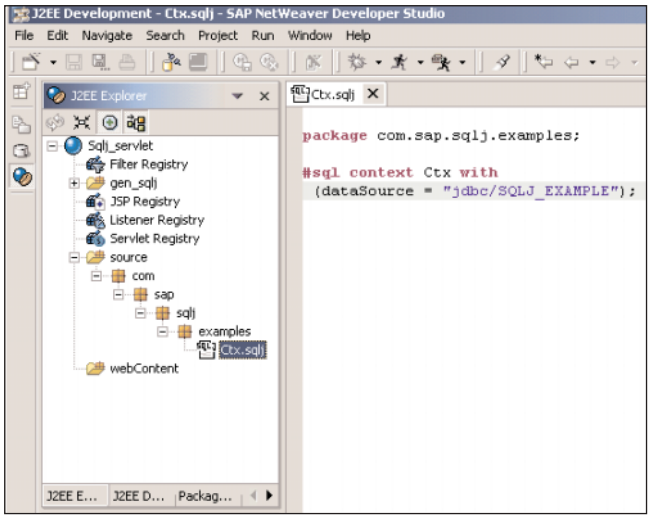


Figure 24 The Connection Context (Ctx.sqlj)

```
1 package com.sap.sqlj.examples;
2
3 #sql context Ctx with (dataSource
    = "jdbc/SQLJ_EXAMPLE");
```

Finally, create the SQLJ source files for the examples by following these steps:

1. From the main menu, choose *File → new → other → persistence → SQLJ Source → next*.

2. Enter the appropriate package and class names for the connection context as shown in **Figure 22**, and select *Finish*.

3. As you can see in **Figure 23**, you can now enter the source code. We'll start with the first example, which is the connection context (*Ctx.sqlj*) — enter the source code shown in **Figure 24**.

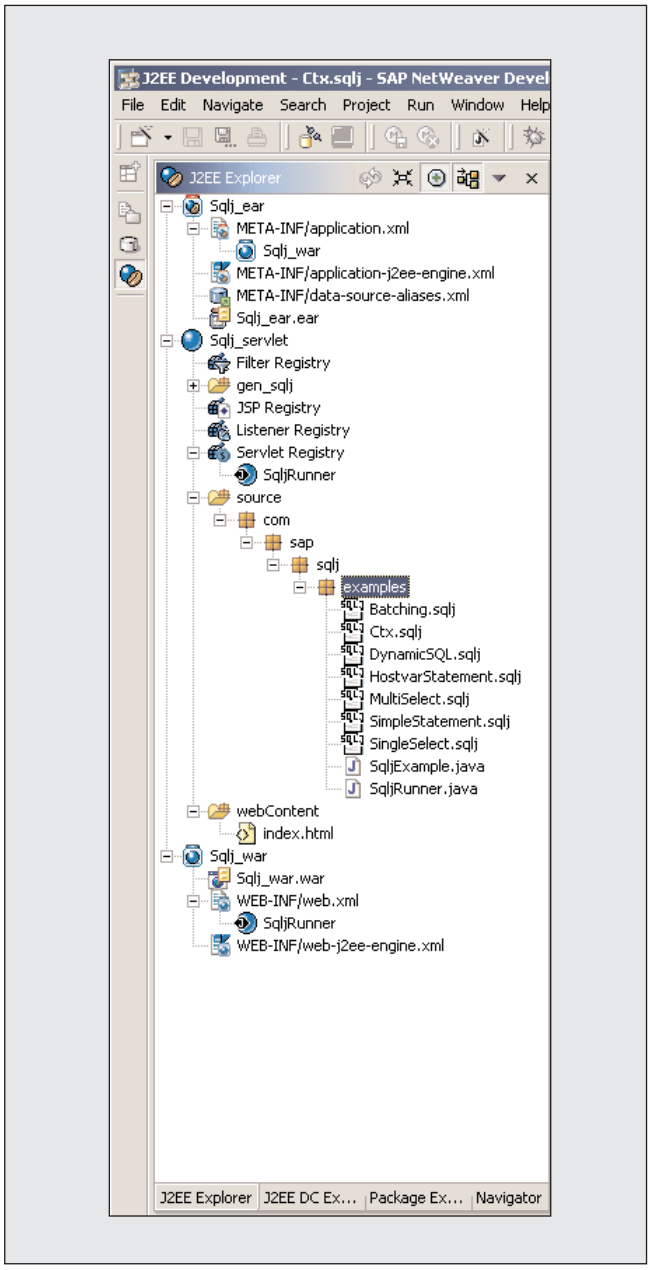
4. Repeat steps 1-3 for creating the SQLJ source files for each example, using the code shown in Figures 9-14. Be sure to use the correct names for each example.

Assembling and Deploying the Application

Before we can run the application, we must create a web archive (WAR) and an enterprise application archive (EAR) that contains the application. For this purpose, create a web application project that bundles the web resources (in this case, only the *Sqlj_servlet*) into a web archive. Choose *File → new → project → J2EE → Web Application Project → next* from the main menu. Name the project *Sqlj_war* and add a reference to the web project *Sqlj_servlet*.

In order to deploy the application, we also need an enterprise application project that bundles all components of the application (in our case, the *Sqlj_war* project). To create an enterprise application project, choose *File → new → project → Enterprise Application Project → next* from the main menu. Name the project *Sqlj_ear* and select *Next*. Add a reference to the web application project *Sqlj_war* and then select *Finish*. In the context menu of the *Sqlj_ear* project, use the menu item *new → data-source-aliases* to create the alias referenced in the class *Ctx*. Enter *SQLJ_EXAMPLE* as the alias name and select *Finish*. In the same context menu, use the item *Build ear file* to create an EAR file that contains the entire example application.

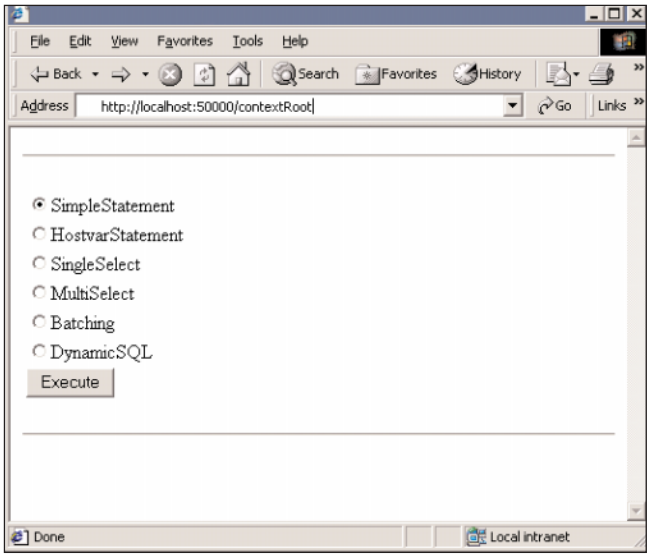
Figure 25 J2EE Explorer View



Finally, in the context menu of the EAR file (*Sqlj_ear.ear*), use the menu item *deploy* to deploy the EAR file into the Web AS. It is now ready for use.

If you created all of the sources, projects, and archives exactly as described, the J2EE Explorer of the NetWeaver Developer Studio should now look like **Figure 25**.

Figure 26 Running the Example Application



Running the Application

To run the example, start your web browser and connect to your J2EE engine using the URL *http://<hostname>:<portnumber>/contextRoot* where *<hostname>* and *<portnumber>* are the hostname and the port number of your J2EE server. You will see a page similar to **Figure 26**. The output of the examples will appear in your web browser. Run the examples from top to bottom to ensure that the data used in the examples is properly initialized.

Helpful Hints

☑ Remember to create the reference from the SQLJ project to the dictionary project. Otherwise, the table descriptions will be unknown, and the translation of the SQLJ sources will fail.

☑ Make sure that your data sources are configured to return Open SQL connections. Otherwise, Open SQL/SQLJ contexts will not accept the connection.

☑ For embedding static SQL statements, use Open SQL/SQLJ. For embedding dynamic SQL statements, use Open SQL/JDBC. And remember that the Open SQL for Java framework gives you the power to combine both static and dynamic SQL statements in the same Java source.

☑ Feel free to extend the output of these examples if you are interested in exploring more details (such as the return values of special methods).

☑ If the NetWeaver Developer Studio GUI changes in future releases, please refer to the documentation of your release to determine the appropriate steps.

Conclusion

Release 6.30 of the SAP Web Application Server provides a powerful framework for database access using relational persistence. Open SQL for Java comprises a platform-independent syntax and semantics checker. It guarantees that SQL statements embedded in Java code will execute on all supported databases with equal syntax and semantics, relieving you from the burden of worrying about persistence code portability. With Open SQL/SQLJ, you can access a database using static SQL that is automatically checked at design time. When you need dynamic SQL, you simply use Open SQL/JDBC instead. There are no restrictions — Open SQL/SQLJ and Open SQL/JDBC are fully interoperable. You can mix static SQL (using Open SQL/SQLJ) and dynamic SQL (using Open SQL/JDBC) in the same Java source, trouble-free.

With this framework, Open SQL/SQLJ solves the problems of portability and design-time checking with static SQL without sacrificing the flexibility of dynamic SQL, making developing database-centric applications easier, more efficient, and more enjoyable.

Further Reading

ISO standards (available at www.iso.org)

- **SQL 92:** ISO/IEC 9075:1992, Information technology – Database Languages – SQL
- **SQLJ:** ISO/IEC 9075-10:2000, Information technology – Database languages – SQL – Part 10: Object Language Bindings (SQL/OLB)

SAP NetWeaver Developer Studio Product Documentation

- **Open SQL/SQLJ Developer Documentation:** *SAP Web AS for J2EE Applications → Development Manual → Developing Web Applications → Developing Business Logic → Java Persistence → Relational Persistence → Open SQL/SQLJ*
- **Open SQL/SQLJ API Documentation:** *SAP Web AS for J2EE Applications → Reference Manual → Java Persistence Reference → SQLJ Runtime API*
- **Open SQL Grammar:** *SAP Web AS for J2EE Applications → Reference Manual → Java Persistence Reference → Open SQL Reference → Open SQL Grammar*

Andreas Fischbach studied Commercial Information Technology and received his diploma in 1996. During that same year, he joined the SAP/Informix porting team, where he was responsible for Informix-specific aspects of the SAP R/3 system. In 2001, Andreas started working on Java persistence in the Java Server Technology group. He focuses mainly on relational persistence, in particular SQLJ. Additionally, Andreas is engaged in object persistence with JDO. He can be reached at andreas.fischbach@sap.com.

Adrian Görler studied physics at the Ruprecht-Karls-University of Heidelberg, Germany, where he specialized in Computational Biophysics. He received his doctorate at the Max-Planck-Institute for Medical Research, Heidelberg, and did post-doctoral research work in various international laboratories. In 1999, Adrian joined SAP and became a member of the Business Programming Languages group, where he worked as a kernel developer responsible for the implementation and maintenance of Open SQL as well as Native SQL in ABAP. Since 2001, Adrian has been a member of the Java Server Technology group, where he has been working on the implementation of Open SQL for Java, especially Open SQL/SQLJ. He can be reached at adrian.goerler@sap.com.

