AN OVERVIEW OF SQLJ: EMBEDDED SQL IN JAVA

Julie Basu, Oracle Corporation

INTRODUCTION

Over the past two years, Java has become the language of choice for developing Internet and intranet applications. Its promise of a standardized, portable, and network-centric application development solution is being realized steadily, and Java *applets* and *servlets* are appearing all over the World Wide Web, bringing rich and varied functionality to the previously static medium. The industry-leading Oracle8*i* data server provides comprehensive Java support at both the client and the server in a variety of ways [1]. In this paper, we present an overview of the features of SQLJ, a tool that allows embedding of static SQL statements textually in Java programs. The SQLJ translator converts such programs to pure Java code, optionally checking the static SQL statements against a database schema. The generated Java code can be compiled, and then executed through a JDBC driver against the database. Oracle8*i* also supports SQLJ stored procedures, functions, and triggers, which execute on a Java Virtual Machine integrated with the data server [2], thus enabling flexible deployment configurations, and code partition and migration across different tiers. Some major benefits of SQLJ are: increased user productivity through compact syntax, robust programming due to SQL checking at development time, standardization by major database vendors, and portability of binary code across different vendors' databases.

The paper is organized as follows. In section 1, we review the SQLJ framework for program development and its benefits. Section 2 provides a technical overview of the SQLJ language, including the use of complex Java host expressions as bind parameters, SQLJ iterators for querying data, and calls to stored procedures. We illustrate through a detailed example the steps involved in developing a SQLJ program, showing how to have the SQLJ translator check all the static SQL statements against a database schema. Next, we demonstrate the use of dynamic SQL, and show how SQLJ interoperates seamlessly with JDBC code, allowing dynamic SQL statements written in JDBC to coexist with the static ones in SQLJ. Finally, we describe the process of defining and running a SQLJ stored procedure in the Oracle8*i* data server. We conclude with a comparison of the SQLJ API with the JDBC interface, and summarize the benefits of SQLJ and Oracle's SQLJ strategy.

1. THE SQLJ FRAMEWORK AND ITS BENEFITS

This section describes the application development framework for SQLJ, in terms of the steps required to translate, compile, and run a SQLJ program. We also discuss the benefits of the SQLJ standard, and of its implementation by Oracle.

1.1 DEVELOPING APPLICATIONS USING SQLJ

The SQLJ translator converts Java programs containing embedded static SQL statements into Java programs with calls to the SQLJ runtime. This process is similar to the Pro*C precompiler translating SQL statements embedded in C, a main difference being that generated code in SQLJ conforms to a vendor-neutral standard. Vendor-specific features and extensions are well-supported in SQLJ through subsequent *customization* of Java byte code. Developing and running a SQLJ application with Oracle consists of four steps, which are shown in Figure 1.1 and listed below:

1. Translation of SQLJ source files with the SQLJ translator. This generates Java files with calls to the SQLJ runtime, as well as binary *SQLJ profile files* that contain information about the static SQL statements present in the SQLJ source.
2. Compilation of Java code with a Java compiler.
3. Customization of the generated SQLJ profiles for using Oracle-specific datatypes and extensions.
4. Running the application, using the SQLJ runtime library and a JDBC driver for Oracle.

SQLJ can perform Steps 1 to 3 by transparently invoking the SQLJ translator, a Java compiler, and then a profile customizer. At translation time, the translator can optionally check the SQL syntax and semantics against a user schema, and verify the type compatibility of host variables with SQL types for all static SQL statements in the program. In general, the SQLJ runtime is able to use any JDBC driver for execution, and will work even with a non-JDBC implementation. The Oracle SQLJ runtime can use either Oracle's JDBC/OCI or JDBC thin drivers, as specified by the user. Details of the JDBC/OCI and JDBC thin drivers from Oracle can be found in [4].

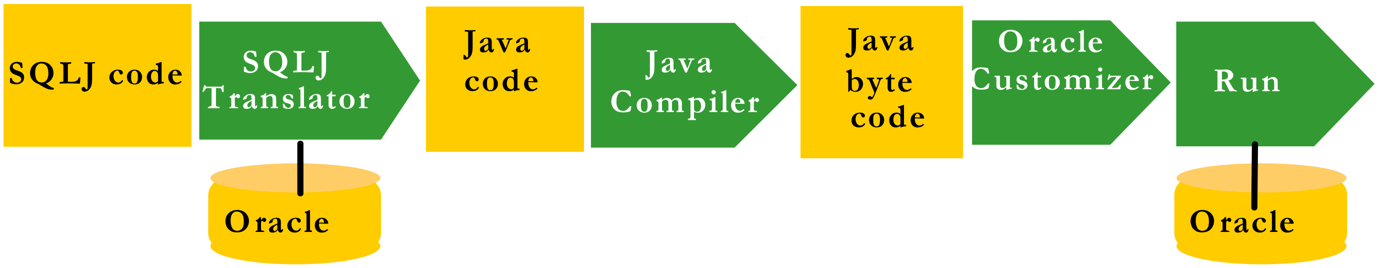

Figure 1-1 - The SQLJ Application Development Process

A *Reference Implementation* of SQLJ has been developed at Oracle, in collaboration with major vendors such as IBM, Informix, JavaSoft, Sybase, Tandem, and others. This Reference Implementation is written entirely in Java, and has a plug-in architecture for easy integration into development tools. The SQLJ translator architecture is also open and extensible, which permits SQL checking and runtime customization for arbitrary databases through Java plug-ins. Source code for the SQLJ Reference Implementation has been made available freely to the public and to other database vendors.

1.2 BENEFITS OF THE SQLJ STANDARD

SQLJ was developed cooperatively by the major database vendors primarily to provide Java application developers with a simple and highly productive database programming tool. The major benefits of the SQLJ standard are enumerated below.

1. A much more concise syntax for database access, and improved developer productivity.
2. Translate-time checking of the syntax and semantics of all static SQL statements.
3. Strong typing of queries through *iterator* types, and of database connections through *connection context* types.
4. Dynamic SQL support through the established JDBC interface for database access. SQLJ and JDBC constitute *complementary* technologies, and it is possible to mix-and-match SQLJ and JDBC code in the same application.
5. Source-level portability through vendor-neutral language syntax, and binary portability through ANSI standardization of SQLJ profiles and runtime. Additional uniformity is obtained from the shared Reference Implementation and standardized JDBC drivers.
6. Database independence through *customization* of the binary SQLJ profiles that describe the static SQL statements. Customization can be utilized for vendor-specific features, for migration to different databases, for debugging and logging purposes, and for performance enhancements through caching or pre-compilation of SQL operations.
7. Flexible deployment configurations. The SQLJ runtime is written in pure Java, which means that SQLJ applications can be deployed wherever JDBC is deployed, such as on a thin client, a thick client, in the middle tier, or in the server.

As part of the multi-vendor standards effort, the *SQLJ Part 0* specification for the embedding of static SQL statements in the Java language has been submitted to ANSI as Draft X3H2 98-227. *SQLJ Part 1* and *Part 2* proposals deal with SQLJ stored procedures and stored Java classes, and are at present being developed by the SQLJ partners, including Oracle.

1.3 STRENGTHS OF ORACLE'S SQLJ IMPLEMENTATION

As the industry leader, Oracle has not only helped to develop the SQLJ standard, but has also adopted it fully. Additionally, Oracle has integrated SQLJ in its database and tools. We note below the major strengths of Oracle's SQLJ implementation.

1. Tight integration with the JavaVM embedded in the Oracle8*i* data server. Integrating SQLJ with the server allows stored procedures, functions, and triggers to be written in SQLJ, and supports a uniform programming style at the client and the server.
2. Integration of the SQLJ translator with Oracle's JDeveloper tool. JDeveloper is a graphical IDE that allows SQLJ translation, Java compilation, and profile customization to be performed in one step, also provides debugging support at the SQLJ source code level. A production version of JDeveloper is currently available.
3. Oracle's own *SQL Checker* module for translate-time verification of the syntax and semantics of static SQL statements.

4. Support for Oracle-specific datatypes such as `REFCURSORs`, `LOBs`, and Oracle8 object types through Oracle's own profile customizer. The SQLJ customization process allows Oracle-specific extensions without jeopardizing standards compliance.
5. Improved performance for the Oracle database through customized profiles.

Further extensions and improvements are planned in Oracle's SQLJ support, such as integration with the Personal Oracle Lite database, complete NLS messages for globalization, and better performance and scalability through further integration with Oracle's JDBC drivers.

2. SQLJ LANGUAGE FEATURES

In this section, we illustrate the main concepts and constructs in the SQLJ language with the help of an example. Topics covered include using host expressions in SQL statements, performing updates, querying data through SQLJ iterators, calling PL/SQL stored procedures and functions, and using JDBC for dynamic SQL operations.

2.1 EXAMPLE

We consider a mini project-tracking system consisting of a table `PROJECTS`, with a nested table column `OWNERS` of type `OWNER_SET` that contains the names of the owners of a project:

```
  CREATE TYPE OWNER_SET AS TABLE OF VARCHAR(20)
  /
  CREATE TABLE PROJECTS (
        ID NUMBER(4),
        NAME VARCHAR(30) PRIMARY KEY,
        START_DATE DATE,
        DURATION NUMBER(3),
        OWNERS OWNER_SET ) NESTED TABLE OWNERS STORE AS OWNERS_TAB
  /
```

Let us design an application that can: (1) update the duration of a project, (2) list the open projects that are yet to be completed, (3) show the time to complete all open projects, (4) calculate which projects are due this month, etc. We will implement each of these functions using a different feature of SQLJ. Below is the skeleton of the Java class `ProjDemo` for this application. Following Java's file naming rules, the file containing this public class must be named `ProjDemo.sqlj`.

```
import sqlj.runtime.*;
import sqlj.runtime.ref.*;
import java.sql.*;

public class ProjDemo {
  public static void main(String[] args)  {
    try {
      dbConnect();                           //  logon to the database
      int numDays = 5;
      String projName = "That Project";
      updateDuration(projName, numDays);     // update project duration
      listOpenProjects();                    // list open projects using named iterator
      listOwners(projName);                  // list owners using positional iterator
      getMaxDeadline();                      // get max deadline via PL/SQL procedure
      boolean dueThisMonth = true;
      projectsDue(dueThisMonth );            // list due projects via dynamic SQL/JDBC
      Date deadline = projDeadline(projName); // get deadline via SQLJ stored function
    }
      catch (Exception e) {
         System.err.println( "Error running ProjDemo: " + e );
      }
  …   Definitions of the above methods   …
  }
```

We will gradually fill in the method definitions of the `ProjDemo` class in the following sections.

2.2 SQLJ STATEMENTS

All SQLJ statements start with the `#sql` prefix and end with a semi-colon. SQLJ statements are classified into two categories: *declarative* and *executable*. Declarative statements introduce Java types in the program, whereas executable statements specify database operations. Java types for SQLJ *iterators* and *connection contexts* are defined through the declarative statements, and such type declarations may appear where a Java class definition can legally appear. In contrast, executable SQLJ statements contain a static SQL operation within curly braces, and can be placed wherever a Java statement may appear. Examples of declarative and executable SQLJ statements are given below.

```
#sql iterator ProjIter (String name, int id, Date deadline);    // declares Java type

#sql { CREATE UNIQUE INDEX projid_index ON projects (id) };     // executes SQL
```

2.3 HOST EXPRESSIONS

In executable SQLJ statements, the inputs and outputs to SQL occur through Java *host expressions* that are embedded into the SQL statements with a colon prefix. In its basic form, a host expression is the simple name of any valid Java variable, field, or parameter. In our example, the method `updateDuration()` can be implemented with host variables `projName` and `numDays` as follows:

```
public static void updateDuration(String projName, int numDays) throws SQLException {
   #sql { UPDATE projects SET duration = duration + :numDays
          WHERE  name = :projName };
   #sql { COMMIT };
}
```

SQLJ also supports the use of qualified names and complex Java host expressions, which must appear in parentheses after the colon prefix, as shown in the example below.

```
#sql { UPDATE PROJECTS SET duration = :(getNewDuration(id)) WHERE ID = :id };
```

Here, `getNewDuration(id)` is a Java method call that returns a numeric value denoting the updated duration of the project. At runtime, this expression is evaluated in Java, and then passed to the SQL statement as an input bind parameter.

All standard JDBC types, such as boolean, byte, short, int, String, byte[], Integer, Double, java.sql.Date etc. are valid host expression types in SQLJ. Additionally, Oracle's SQLJ translator supports the use of Oracle7 and Oracle8 types such as ROWID, CLOB, BLOB, as well as Object and REF types. Details on use of the Oracle8 types may be found in [3] and [4].

2.4 ITERATORS

In a SQLJ program, a result set returned by a SQL query can be represented as an *iterator* object that is used to examine the data. An iterator object is an instance of an iterator class, which may be defined using the `#sql iterator` <*name*> `...` declaration. An iterator type declaration is expanded by the SQLJ translator to a Java class declaration of the same name. A SQLJ iterator class corresponds to the JDBC result set, with the additional important property that it is strongly typed in terms of the *shape* of the SQL query as defined by the number and Java types of selected attributes, and optionally their names too. This strong typing of SQLJ iterators allows them to be treated as type-safe, first-class Java objects with known row shapes, so that both the SQLJ translator and the Java compiler can statically check the validity of column data access wherever the iterator is used. SQLJ supports two different types of iterators - *named* and *positional*, both of which are illustrated below.

2.4.1 NAMED ITERATORS

A named iterator declaration specifies both column accessor names and their Java types. The accessor names must match the names (or aliases) of the selected columns in a SQL query bound to the iterator, and the accessor types must be valid JDBC or Oracle types compatible with the corresponding SQL datatypes. For our example, the following declarative statement introduces a named iterator type `ProjIter` with three columns `name`, `id`, and `deadline`:

```
#sql iterator ProjIter (String name, int id, Date deadline);
```

This type declaration can be placed where a Java class definition can appear, for example, after the imports in the file `ProjDemo.sqlj`. During translation of this SQLJ program, the above declaration generates a Java class named `ProjIter` with three special *column accessor methods*: `name()`, `id()`, and `deadline()`, that return `String`, `int`, and java.sql.`Date` values respectively, and are used to access the fetched column values. Other methods of the `ProjIter` class are similar to those found on JDBC result sets, such as `next()` to iterate over fetched rows, and `close()` to release the resources held by the iterator.

After an iterator type is defined, a SQLJ program can declare instances of this iterator type, and populate it using a SQL query. During execution, the SQLJ runtime matches the names of the column accessors in the iterator with the SQL column names in a *case-insensitive* way. Note that column aliases must be used for those SQL columns whose names are not valid Java identifiers. We now illustrate these concepts in the method `listOpenProjects()` for our example:

```
  public static void listOpenProjects() throws SQLException {
    ProjIter projs = null;               // Declare the iterator instance
                                         // Populate the iterator with a SQL query
    #sql projs = { SELECT start_date + duration as deadline, name, id
                    FROM projects  WHERE start_date + duration >= sysdate

    while (projs.next()) {               // Loop through the result rows
                                         // Access data via column accessors
            System.out.println("Project named ’" + projs.name() + "’ id " +
                               projs.id() + " completes on " + projs.deadline());
    }
    projs.close();
  }
```

Notice that the columns selected by the query do not match the iterator accessors in order, but have the same names. An alias named `deadline` denotes the completion time of a project, as computed by adding its `start_date` to its `duration`.

2.4.2 POSITIONAL ITERATORS

In contrast to named iterators, positional iterators specify only the number and types of columns, and not their names, as in:

```
    #sql iterator OwnerIter (String);
```

As for named iterators, this declaration generates a Java class named `OwnerIter`, but it does not have any special methods for column access; however, internally it encodes the query shape. Column data may be accessed by position only, through traditional `FETCH..INTO` syntax. The same benefits of strong typing as for named iterators are applicable for positional iterators also. The only difference is in the way column data is accessed, named access being more flexible and less error-prone in some cases, while `FETCH..INTO` may be adequate and more convenient in others. The choice is left to the SQLJ user, depending on the requirements of his/her program. Below we define the method `listOwners()` for our example using a positional iterator `owners` of type `OwnerIter`:

```
  public static void listOwners(String projName) throws SQLException {

    OwnerIter owners = null;                       // Declare the iterator instance
    #sql owners = { SELECT *                       // Populate iterator with SQL query
                    FROM THE(SELECT (p.owners) FROM projects p
                             WHERE p.name = :projName) };
    String ownerName = null;
    while (true) {                                 // Loop though the results
      #sql { FETCH :owners INTO :ownerName };  // FETCH implicitly gets next row
      if (owners.endFetch()) break;            // Check if no more rows
      System.out.println(ownerName);           // Else print data
    }
    owners.close();                                // Close the iterator
  }
```

Observe that the termination condition in the `while` loop for the `FETCH` statement is detected by calling the `endFetch()` method on the `owners` iterator, and this condition must always be checked before fetched data is accessed. This method is available only for positional iterators, and not for named ones. Positional and named iterators are separate Java entities, and the two paradigms cannot be mixed for the same iterator.

2.5 CALLING PL/SQL STORED PROCEDURES, FUNCTIONS, AND ANONYMOUS BLOCKS

SQLJ provides convenient short-hand syntax to call stored procedures and functions stored in the database, as well as anonymous procedural blocks. Assume for our example that we wish to print the completion date of all open projects.

We can define a stored procedure named `MAX_DEADLINE` using PL/SQL in the Oracle database, as follows:

```
CREATE OR REPLACE PROCEDURE MAX_DEADLINE (deadline OUT DATE) IS
   BEGIN    SELECT MAX(start_date + duration) INTO deadline FROM projects;
   END;
```

```
/
```

Then, we can call this stored procedure from SQLJ, as in the method `getMaxDeadline()` below.

```
 public static void getMaxDeadline() throws SQLException  {
    Date maxDeadline;
    #sql  { CALL MAX_DEADLINE(:OUT maxDeadline) };    // CALL syntax for stored procedures
    System.out.println("Last project completes on " + maxDeadline);
  }
```

OUT and INOUT parameter modes must be declared explicitly for stored procedures, functions, and anonymous blocks. The syntax for a function call is different from that of a procedure call, in that it uses the `VALUES` construct instead of `CALL`. For example, if we defined a PL/SQL function `GET_MAX_DEADLINE` returning a `DATE` instead of the stored procedure `MAX_DEADLINE` with an out parameter, the call to this function would appear in SQLJ as:

```
  #sql maxDeadline = { VALUES(GET_MAX_DEADLINE()) }; // VALUES syntax for stored functions
```

Anonymous blocks in SQLJ statements are placed within the curly braces using `BEGIN..END` syntax.

2.6 CONNECTING TO THE DATABASE

Database connections come into the picture at application runtime, and at translation time as well. Runtime SQL operations using the default context require the SQLJ `DefaultContext` to be initialized with a connection from the JDBC driver. Establishing a runtime connection to the Oracle database consists of the steps shown below.

```
public static void dbConnect() throws Exception
  { DriverManager.registerDriver(new oracle.jdbc.driver.OracleDriver());// register driver
    String url  = "jdbc:oracle:oci8:@";             // Using JDBC/OCI driver to connect
    String user = "scott"; String pwd  = "tiger";  // Logon user id and password
    DefaultContext.setDefaultContext(              // Initialize SQLJ default context
       new DefaultContext(url, user, pwd, false)); // with autoCommit off
  }
```

This connection is used to execute all `#sql` statements that do not have a context *tag*, as in the `ProjDemo` examples above. To enable SQL checking at translation time, database connection information can also be specified on the SQLJ command-line, or more conveniently, in a *SQLJ properties file*. Below we show an example of SQLJ command-line invocation that uses the JDBC/OCI driver and the scott/tiger schema during translation to check static SQL operations on the default context:

```
    sqlj -user=scott/tiger -url=jdbc:oracle:oci8:@   ProjDemo.sqlj
```

For flexibility, SQLJ supports many other options, such as for specification of the JDBC driver class used at translation time. Most options have common default values, e.g., the driver is preset to `oracle.jdbc.driver.OracleDriver`.

2.6.1 CONNECTING TO MULTIPLE DATABASES

In the examples given above, we have used a single default connection context for all our SQL operations. SQLJ supports connecting to multiple schemas in the same program. Different schemas used at runtime are modeled as distinct *connection context* classes in SQLJ programs. For example, a non-default connection context class `ProjDb` is defined as:

```
    #sql context ProjDb;
```

This declaration is expanded by the SQLJ translator to a Java class named `ProjDb`, of which an instance can be declared and initialized with a database connection as follows:

```
    ProjDb myProjDb;               // Declare instance of ProjDb connection context class
    myProjDb = new ProjDb(url, user, pwd, true);         // Initialize with autoCommit on
```

Subsequently, this connection context instance may be used in an embedded SQL operation as follows:

```
    #sql [myProjDb] { UPDATE … };  // Execute SQL on myProjDb connection context
```

The SQLJ translator also supports SQL checking on multiple connection contexts at translate time, through command-line options that are (optionally) *tagged* with the connection context class name. An example of such use is:

```
 sqlj -user=scott/tiger -user@ProjDb=roger/lion -url=jdbc:oracle:oci8:@  ProjDemo.sqlj
```

Such an invocation makes the SQLJ translator use two different schemas for checking SQL operations: scott/tiger for those that use the default context, and roger/lion for the SQL executed on instances of the `ProjDb` connection context class.

2.7 USING DYNAMIC SQL THROUGH JDBC

In some cases, for example when a `WHERE` clause condition or the set of selected attributes is unknown, the SQL statement is not known in advance, and therefore dynamic SQL must be used. The dynamic SQL API for SQLJ is JDBC, and a SQLJ program may contain both SQLJ code and JDBC calls. Access to JDBC connections and result sets from a SQLJ program might also be necessary for finer granularity of control. The two paradigms interoperate seamlessly with each other through conversions between JDBC connections and SQLJ connection contexts, and between JDBC result sets and SQLJ iterators. For example, a SQLJ connection context can be initialized with an existing JDBC connection:

```
     java.sql.Connection conn = …;     // Create JDBC connection
     ProjDb pdb = new ProjDB(conn);    // Use to initialize SQLJ connection context
```

Conversely, it is also possible to extract a JDBC connection object from a SQLJ connection context instance. This feature is illustrated below, where we define the method `projectsDue()` using dynamic SQL via JDBC statements and connections:

```
public static void projectsDue(boolean dueThisMonth) throws SQLException {
     // get JDBC connection from previously initialized SQLJ DefaultContext
     Connection conn = DefaultContext.getDefaultContext().getConnection();

     String query =  "SELECT name, start_date + duration FROM projects " +
                     "WHERE start_date + duration >= sysdate ";    // Query open projects
     if (dueThisMonth)                            // Add condition to check month due
       query += " AND to_char(start_date + duration, 'Month') " +
                " = to_char(sysdate, 'Month') ";  // Extract and compare month from dates

     PreparedStatement pstmt = conn.prepareStatement(query);
     ResultSet rs = pstmt.executeQuery();
     while (rs.next()) {
      System.out.println("Project: " + rs.getString(1) + " Deadline: " + rs.getDate(2));
     }
     rs.close(); pstmt.close();
 }
```

Similar conversions as for database connections are also supported between JDBC result sets and SQLJ iterators:

```
     ProjIter projs;                                    // Declare iterator instance
     #sql projs = { SELECT … };                         // Initialize it
     java.sql.ResultSet rsProjs = projs.getResultSet(); // Get its JDBC result set
```

Likewise, we can instantiate a SQLJ iterator object using a JDBC result set. The example below shows such a conversion to the SQLJ named iterator `projs`. In this case, the column names in the SQL query must match the accessors in the iterator.

```
     java.sql.ResultSet rs = …;   // Create and initialize a JDBC result set
     #sql projs = {CAST :rs};     // Cast the result set to a SQLJ iterator
```

Notice the use of the `CAST` operator above, which is special syntax provided for conversion of JDBC result sets to iterators.

3 DEFINING SQLJ STORED PROCEDURES ON THE SERVER

Finally, we demonstrate the use of SQLJ stored procedures and functions on the Oracle8*i* data server, which provides an integrated JavaVM [2], and supports SQLJ. SQLJ code defined at the server is automatically translated into Java byte code, and then stored in the database. For example, using *SQL*Plus*, we can define a static function `getDeadline()` in a Java class named `ProjUtil` as follows:

```
CREATE OR REPLACE JAVA SOURCE NAMED "ProjUtil" AS
  import java.sql.*;
  public class ProjUtil {
    public static Date getDeadline (String projName) {
      // Note: connection is automatic for server-side execution of SQLJ programs
      try {
            Date completionDate;
            #sql { SELECT start_date + duration INTO :completionDate FROM projects
                   WHERE name = :projName };  // single-row query by key project name
            return completionDate;
          } catch (SQLException e)  { return null; } } }
/
```

Once this SQLJ function has been created successfully, it can be invoked from SQL statements. But first a *wrapper* has to be defined for mapping the Java invocation signature to SQL. This wrapper definition is based on PL/SQL syntax, e.g.,

```
CREATE OR REPLACE FUNCTION getDeadline(projName VARCHAR2) RETURN DATE
  AS LANGUAGE JAVA NAME 'ProjUtil.getDeadline(java.lang.String) return java.sql.Date';
/
```

Then, the stored function or procedure can be invoked from both client-side and server-side SQLJ code, just like any other stored PL/SQL function or procedure. Thus, we can define the method `projDeadline()` using the above SQL wrapper:

```
public static Date projDeadline(String projName) throws SQLException {
    Date deadline = null;
    #sql deadline = { VALUES(getDeadline(:IN projName)) };
    return deadline;
}
```

4 CONCLUSIONS

This paper has presented a comprehensive overview of the goals, benefits, and capabilities of SQLJ. Using a detailed example, we have explained the various language features and demonstrated their usage. From the examples in this paper, we see that SQLJ statements are usually much shorter than the equivalent dynamic SQL calls in JDBC, because SQLJ uses embedded Java host variables to pass arguments to SQL. In contrast, the JDBC user must write separate calls to bind each argument and to retrieve each result. Another important advantage of SQLJ is that it supports translate-time checking of all static SQL statements against a database schema, and verification of the type compatibility of host variables with SQL datatypes. Additionally, strongly-typed SQLJ iterators enable Java type checking and SQL schema checking where data is fetched from an iterator (either using named column accessors or through a `FETCH..INTO` statement), because the iterator's class defines the number and types of the fetched columns. This type-checking greatly enhances program robustness by catching errors at development time, rather than at application runtime, and such benefits cannot be obtained in a dynamic API like JDBC. However, SQLJ and JDBC are complementary to each other - the former handles dynamic SQL, while the latter addresses static SQL. The user can easily combine both SQLJ and JDBC code in the same application program, and have them interoperate at the level of connections/contexts and result sets/iterators.

Besides the inherent power of Java's portable development and flexible deployment model, the SQLJ application development framework provides many other important advantages. The SQLJ language and runtime are being standardized by ANSI, so that SQLJ programs written for one vendor's database can be easily adapted to another's. By design, SQLJ supports code portability not only at the source level through standard syntax, but also at the level of binaries, since it adds vendor-specific customizations to the binary profile files. Additionally, a Reference Implementation of SQLJ has been developed at Oracle using Java, and its source code is available freely to the public and to other database vendors. SQLJ is blazing a new path in its domain, in terms of its openness, robustness, flexibility, extensibility, standardization, and integration with tools. As the industry leader, Oracle has further reinforced the power of SQLJ through tightly-integrated support for it on the Oracle8*i* data server, allowing SQLJ procedures and functions to be stored in the database and invoked from SQL, PL/SQL, and Java. Oracle-specific datatypes and SQL extensions are fully accessible in SQLJ through binary profile customization, without compromising standards compliance. Oracle8 types such as LOBs, objects, and collections are also easily manipulated in SQLJ [3]. Oracle's JDeveloper tool provides an integrated development environment with built-in support for editing, translation, and debugging of SQLJ code. Further improvements in integration, tools, and performance are on the way.

ACKNOWLEDGMENTS

This paper is based on the work of the SQLJ, JDBC, and JavaVM teams at Oracle. I would like to thank Brian Becker, Ragamayi Bhyravabhotla, Cheuk Chau, Rakesh Dhoopar, Pierre Dufour, Thomas Kurian, Ekkehard Rohwedder, and Brian Wright for their contributions and helpful comments.

REFERENCES

[1] *Oracle8 and Java,* Technical White Paper, Oracle Corporation, 1997.

[2] *Industrial Strength Java: Overview of Oracle's Server-Side Java*, Dave Rosenberg, Oracle Open World, Nov. 1998.

[3] *Using Oracle Objects in SQLJ Programs*, Ekkehard Rohwedder, Oracle Open World, Nov. 1998.

[4] Oracle8*i* SQLJ and JDBC User Documentation, Oracle Corporation, 1998.