Developing Stored Procedures In Java™

An Oracle Technical White Paper

April 1999

Developing Stored Procedures In Java

April 1999

ORACLE
SOFTWARE POWERS THE INTERNET™

INTRODUCTION

In three years, Java has matured from a programming language used to develop simple graphical user interface (GUI) programs that could be downloaded over the Web, to a platform for developing and deploying enterprise and Internet applications. Although Java was developed as a language for client applications, it is now being used in a variety of applications that are helping make network computing a reality. Java applications can easily be deployed on small embedded devices as well as high-end, mainframe-class servers. Java's true potential is now being realized, as it forms the basis for developing business logic and deploying it on scalable servers. Oracle's powerful database platform is a natural to build on Java's ubiquity in the industry. With the release of Oracle8*i*™, Oracle delivers a truly scalable, high-performance and robust platform for deploying intranet/Internet applications based on network computing. Oracle has embedded a Java Virtual Machine (VM) in its database engine. Now, database programmers can use both PL/SQL and Java stored procedures to develop their application logic. This paper focuses on how to develop applications using Java and deploy them in the Oracle database as Java stored procedures. It is a part of a series of technical papers produced by Oracle describing the architecture and programming models of its Java Virtual Machine.

This paper contains five sections:

- The first section discusses why Java is becoming a popular server programming language and the challenges in developing scalable Java servers.
- The second section outlines the different programming models that Oracle8*i* supports: stored procedures and component oriented development.
- The third section offers a detailed example of how to write stored procedures in Java and deploy them in the Oracle8*i* database. This section describes the steps involved in developing your application, loading the application into the Oracle database, publishing your Java applications as stored procedures and invoking Java stored procedures in a variety of contexts.
- The fourth section discusses interoperability among Java, Structured Query Language (SQL) and PL/SQL™. It gives examples of how Java calls SQL and PL/SQL. It also describes the different invocation scenarios for Java stored procedures.
- Finally, the fifth section presents the benefits of developing your applications using Java stored procedures and deploying them on the Oracle JServer VM.

JAVAS EASE AND SIMPLICITY

Why has Java become so popular among application developers? Primarily because Java makes application developers more productive. It is a modern, robust, object-oriented language. Java's unprecedented popularity stems from the following three benefits:

- *Application Development Is Simpler* - Java offers features such as support for networking, multithreading and automatic garbage collection that make application development simpler than most of its predecessors.

- *Applications Are Platform Independent* - The same Java binaries can run on any platform that supports a Java Virtual Machine, from embedded systems to mainframes. This reduces, if not completely eliminates, the task of porting applications.

Applications Can Be Developed As Components - Java offers a component model, JavaBeans, that allows application developers to design and package components that can be assembled with components written by other application developers. Enterprise JavaBeans extend the component model for Java to server-side components. Server-side components enable application developers to develop "business logic" and package it as a component that can be assembled into applications. The JavaBeans component model enables the development of a robust market for plug-and-play software modules. This new application development model enables rapid assembly of applications that can be customized and deployed on any platform and adapted as a company's business needs change.

JAVA FOR ENTERPRISE APPLICATIONS — A SERVER LANGUAGE

Much of Java's initial popularity came from its suitability for adding dynamic content to web pages, in the form of mini applications — "applets" — that run on the client machine. However, Java is moving rapidly toward its primary use in the future: writing enterprise and Internet applications that can be deployed in either client-server or intranet/Internet configurations. There are several reasons for Java's popularity as a server language:

• Java is a safe language and is therefore ideally suited for database integration. Because the database must be a safe environment and provide the foundation for several mission-critical applications, Oracle does not allow application code that could potentially compromise the integrity of the database to run inside the database. For example, Oracle does not permit application code written in "unsafe" languages like C or C++ to be executed in the same address space as the database. Although languages like C provide tremendous flexibility to application developers, they also have several pitfalls. Even experienced C and C++ programmers make these errors:

– *Overwriting Array Boundaries* - Because arrays and strings are manipulated as pointers, programmers inadvertently overwrite the boundaries of arrays and strings.

– *Memory Leaks And Corruption* - Memory allocation and deallocation is left to the programmer and results in a variety of programming problems, including memory corruption and memory leaks.

Any of these errors in an application can compromise the integrity of the database and halt mission-critical applications. In contrast, Java is a strongly-typed language and includes native support for arrays and strings and built-in support for memory management. The garbage collection mechanism built into the Java language frees application developers from allocating and deallocating memory, thereby eliminating problems of memory corruption and memory leaks.

• Apart from being a safe language, Java on the server offers many of the same benefits to application developers that it does on the client. Because Java is platform-independent, application logic developed in Java can easily be deployed on any server that supports Java. This enables true application partitioning in a multi-tier environment. Application programs written in Java can easily be migrated to another server without having to rewrite the application. Enterprises can leverage these benefits to improve time-to-market and lower system development and administration costs.

• Java is a full-featured language whose expressive power is close to C. It can be used to develop any kind of applications in a client or a server environment.

PROGRAMMING USING ORACLE JSERVER

Application developers familiar with procedural programming styles developed business application logic using languages like PL/SQL. The business logic they developed was deployed as stored program units that run in the database as stored procedures, functions or triggers (henceforth referred to as "stored procedures"). A stored procedure is a set of SQL and PL/SQL statements grouped together as an executable unit. They are defined and stored in the database, thereby providing efficient access to SQL data. Because stored procedures execute in the database, they minimize the network traffic between applications and the database, increasing application and system performance. Starting with the Oracle8*i* release, application developers can now use Java to develop their business logic and deploy it in the database as stored procedures, functions and triggers. Stored procedures developed in Java run in the same address space as SQL and PL/SQL, so they can seamlessly interoperate and invoke business logic that had been written using SQL or PL/SQL.

While PL/SQL shares the same datatypes and is therefore seamless with SQL, Java is an opensystem alternative to PL/SQL — an open, general purpose programming language with which you can program the server. And Oracle8*i* Java implementation is fully functional. In other words, you can now use Java in all the contexts in which you traditionally used PL/SQL. Further, database stored procedures can be implemented using either standard Java or SQLJ. SQLJ is a standard way of embedding static SQL statements in Java programs (similar to how Pro*C™ is used to embed static SQL statements in C programs). SQLJ is an excellent choice for writing stored procedures and triggers because these are typically data-or SQL-intensive and an embedded SQL language is ideal for this purpose. Java stored procedures can execute in three different runtime contexts:

• First, you can use them to write top-level SQL functions and procedures in Java. This allows users to extend SQL with any business logic written in Java. It is callable both from within

SQL DML and at the top level, just like PL/SQL.

• Second, database triggers can also be implemented in Java. These can run in a variety of contexts related to changes in database rows. Because Oracle8*i* views are fully updateable, triggers can also augment views to support update, insert or drop operations. You can now write in Java all five types of triggers that Oracle supports.

• Third, Oracle8 provided object-relational functionality, allowing users to define complex or composite datatypes in SQL. While Oracle8 enabled users to implement methods of these types either in PL/SQL or as external procedures in C or C++, Oracle8*i* adds the ability to implement

HOW TO WRITE JAVA STORED PROCEDURES

Now that we have outlined Oracle JServer and described what Java stored procedures are, let's look at how to implement them in the Oracle environment. We start this section with an analogy between the

Java execution environment on a client and the Java execution environment in the Oracle8*i* server, to help you understand the similarities and differences between these two environments. We then present a conceptual overview of how Java programs are loaded, stored, resolved and executed in the database. Finally, with the help of an example, we detail the steps involved in developing and deploying Java programs in the Oracle database.

First, let's look at how Java programs are compiled and executed using JavaSoft's JDK in a client environment. When Java source programs are compiled, each public class is stored in a separate .class file in the operating system environment. Java classes needed to execute a Java program can be physically located in different file system directories. When Java classes are loaded by the Java Virtual Machine, it searches different file system directories specified in the CLASSPATH environment variable to resolve references to the classes needed for execution.

An analogous mechanism is used when you compile and run Java programs in the Oracle8*i* database. Once you have written your Java program and tested it, you need to load it into Oracle8*i* — that is, onto Oracle JServer — and resolve all references. The database supports a variety of different forms in which Java programs can be loaded, including Java source text, standard Java .class files, or Java archives (.jar). Java source loaded into the database is automatically compiled by the Java byte-code compiler hosted in the database. The Java programs loaded into the database are stored as "library units" in a database schema similar to how PL/SQL program units are stored in the database. Each Java source file and Java class is stored in its own library unit. Java library units needed to execute a Java program can be physically located in different database schemas.

Similarly to the client execution environment, Java classes need to be resolved before they can beexecuted. Because Java classes in the database are stored as library units, a Java class library unit is resolved when all of its external references to Java names are bound. Analogous to the CLASSPATH, the RESOLVER is used by the database to resolve/link all the classes needed for server side execution. The RESOLVER provides a search path, but instead of using file system directories, SQL schemas are specified. When resolution searches for a binary corresponding to a Java full name, it searches that list of schemas in order until a library unit matching that name is found. It also builds the depends-on list for the library units containing each class. The depends-on list has the same purpose here as for PL/SQL: As long as the referenced library units are unchanged, then the Java program is, in effect, "pre-linked" and ready to run. Here is the syntax for a RESOLVER:

```
RESOLVER ((<match string> <schema name>) ...)
```

The <match string> is either a Java name or a wildcard name that can appear in a Java language import statement, or the single wildcard '*' that matches any name. The <schema name> specifies a schema to be searched for a corresponding "library unit." For example:

```
RESOLVER (("some.particular.class" FRED)

                ("com.oracle.employee.*" SCOTT)

                   (* PUBLIC))
```

Now that you are familiar with the Oracle8*i* execution environment for Java, let's look at the steps

needed to load and run Java programs in the Oracle8*i* server. Writing Java stored programs in Oracle8*i* can be broken into four steps: write your Java code, load and resolve it, publish the

code and then run it.

WRITE YOUR JAVA CODE

Begin writing the Java program to make a stored procedure, using a standard Java tool or editor. Oracle8*i* provides an Integrated Development Environment (IDE) for Java — JDeveloper™ — to facilitate writing Java applications and deploying them in the Oracle8*i* database. Iterate your Java code through the develop, test and debug cycle until you are satisfied with its functionality.

The example below uses a Java class, 'emp', with two Java methods ('approve_raise' and 'sal_grade') to show how Java stored procedures are created and deployed in the Oracle environment. The method 'approve_raise' uses an employee's current salary and raise to verify that the raise given to the employee follows some business logic. The method 'sal_grade' uses an employee's current salary to return the grade an employee belongs to. The Java code in this example does not make any SQL or PL/SQL calls. In the next section, we extend the example to show how Java interoperates with SQL.

```
// File: emp.java

package com.oracle.employee;

public class emp {

      public static String sal_grade (float current_sal) {

                if (current_sal > 0 && current_sal < 50000) return

" MTS";

                if (current_sal >= 50000 && current_sal < 100000)

return " SMTS";

                return "Salary out of Range";

      }

      public static void approve_raise (float current_sal,

float raise, float new_sal [ ]) {

                // new_sal is an array of size one to handle OUT
```

```
parameters in Java.

                float percent_raise = (raise / current_sal) * 100;

                if (percent_raise > 15 || percent_raise < 5)

                        new_sal[0] = -1;

                else {

                        new_sal[0] = current_sal + raise;

                }

      }

}
```

LOAD AND RESOLVE YOUR JAVA CODE

Once you have written the Java program, you need to load it into the database. This section describes the different forms used for loading Java into the database, how to load Java and from where you can load Java into the database.

• *You Can Load Java Into The Database In Three Forms* - First, Java programs can be loaded into the database as Java source and compiled by a byte-code compiler hosted in the database. Second, Java source files that have been compiled outside the database can also be loaded into the database as .class files or .jar files. Finally, a Java resource file (Java data in standard portable format — for example, images) can also be loaded into the database.

• *You Can Load Java Into Oracle8i In Two Ways* - First, a new DDL command of the form "CREATE JAVA …" can be issued from SQL*Plus® to load Java source, binaries, or resource files into the database. The Oracle8*i* database accepts two input sources for Java: from a binary file on the operating system level or from a LOB column in the database. The name of the library unit for each Java class loaded into the database is derived from the name of the class that it contains. For example, if the fully qualified name of a Java class is com.oracle.employee.emp, the corresponding library unit name in the database will be com/oracle/employee/emp.

The following examples show the two different mechanisms that you can use to load a Java class file into the database:

- Using a binary file on the operating system level:

  — Create a directory object on the server's file system

- SQL> CREATE DIRECTORY bfile_dir as
  '/home/user/com/oracle/employee';

- Statement processed.

- Then load the Java class file using the "CREATE JAVA CLASS …" DDL statement

- SQL> CREATE OR REPLACE JAVA CLASS USING BFILE (bfile_dir, 'emp.class');

- Statement processed.

Using a LOB column in the database:

1. Load the Java emp.class file into the LOB column of a table in the database. (Details not shown here.)

2. Create the Java class from the LOB column in the table

- SQL> CREATE JAVA CLASS RESOLVER USING BLOB

- >(SELECT ALOB FROM LOB_TABLE WHERE ID=123);

- Statement processed.

To simplify the loading process, Oracle8*i* provides a utility written in Java, called LOADJAVA, that automates the process of loading Java into the database. It takes Java input in three forms: Java source, Java binaries (.class) and entire Java archives (.jar). The LOADJAVA utility performs for you all the steps mentioned above. It first creates a system-generated database table into which Java can be loaded. This table is called `create$java$lob$table`. Next, it loads the Java binaries and .jar into a Binary LOB or BLOB column of this system generated table. Finally, it implicitly invokes "CREATE JAVA …" DDL to load Java from the BLOB column into database library units.

The LOADJAVA utility can also take a RESOLVER specification as an argument that it uses to resolve the Java classes. Along with the LOADJAVA utility, Oracle provides a DROPJAVA utility for dropping Java binaries and archives from the database "library units."

The following example shows how to use LOADJAVA to load Java programs into the Oracle database. This example loads the emp.class file into user SCOTT schema. It connects to the schema with the password "tiger." The Java class is loaded into the database on host oudelsrv-1 with database SID ORCL. The RESOLVER specification used to resolve external references for this class is specified by (("employee.*" SCOTT) (* PUBLIC)).

Using LOADJAVA to load Java program units into the database:

```
loadjava -user scott/tiger@oudelsrv-1:5521:ORCL -r

‘(("employee.*" SCOTT) (* PUBLIC))’ emp.class
```

You can load Java over the network: Because the LOADJAVA utility uses Oracle’s Java Database Connectivity (JDBC) drivers to communicate with the database and load Java into the database, Java program units can be loaded into the database over the network.By default, the LOADJAVA utility uses the Thin JDBC driver. The user also has the option to specify the JDBC/Oracle® Call Interface driver.

PUBLISH YOUR JAVA CODE

Once the Java program is loaded into the database, you need to register it with SQL. To register the Java program with SQL you need only expose the top-level Java entry point, so that it is callable from SQL. What does that mean? Consider a database trigger that is implemented in Java. The Java trigger may have several classes and method calls. Only one of these is initially called from SQL when the DML statement is executed. Subsequent Java-to-Java calls happen within the Java VM. Java classes and methods are not automatically published to the data dictionary. Many Java methods are called only from other Java classes. The reason for publishing Java to SQL, therefore, is to register this top-level Java entry point — that is, the Java class or method that SQL initially calls with SQL. Three issues need to be addressed before publishing Java to SQL:

Mapping SQL Types To Java Types - When Java programs are invoked from SQL or PL/SQL, data is passed to and from SQL to Java in parameters and results. Because Java and SQL support different datatypes, they need to be mapped from SQL to Java or from Java to SQL during a stored program invocation. Java native types include BYTE, SHORT, INT, FLOAT and DOUBLE while SQL native types include NUMBER, CHAR, VARCHAR2, DATE. When a Java stored program is published to SQL, you have to specify how the parameters and return values are mapped between SQL and Java. You can map SQL native types directly to Java native types but there are two problems with doing so:

— First, you may lose information in some applications when SQL native types are mapped to Java native types. For example, mapping a SQL NUMBER to a Java ‘int’ could result in loss of information in financial applications that use very large numbers.

— Second, SQL data can have NULL values. Scalar types in Java can not be NULL. When SQL NULL data are passed to parameters of Java scalar types, a NULL CONSTRAINT exception is raised before the call to Java is attempted.

You can handle these problems in two ways:

- To address the SQL NULL issue, you can use the Java classes corresponding to the Java scalar types. For example, you can map the SQL NUMBER type to the java.lang.Integer class. SQL NULL values can now be passed to Java because Java “null” represents SQL NULL. However, using these Java classes still does not address the possibility of loss of information.

- To address both problems mentioned above, Oracle8*i* provides an Oracle.SQL Java package that provides Java wrappers for native SQL types. These classes are designed to hold SQL data in SQL format (in byte arrays) rather than converting them to Java native format. Every primitive SQL type has a corresponding class definition in the oracle.sql package. These classes provide conversions from SQL format to primitive Java types when appropriate. Because data is kept in SQL native format, there is no loss of information and it can also store NULL values. In operations that involve moving SQL data from one table to another, using the oracle.sql package also proves to be very efficient because no conversion occurs from SQL native types to Java native types. For example, the oracle.sql.NUMBER class can be used to pass a SQL number to Java programs.

- *Mapping SQL Parameter Modes To Java Parameter Modes* - SQL supports parameter modes IN, OUT and IN OUT whereas Java methods only support the IN parameter mode. In Java, SQL OUT and IN OUT parameter modes are mapped to Java parameter types by using an array of length one and getting and setting its single element as if it were an IN OUT parameter.

- *Privileges Under Which The Java Programs Will Execute* - In Oracle 7™ and Oracle8, PL/SQL stored programs execute with "definer's-privileges." Such PL/SQL programs bind early to the tables that they name. A definer's-privileges program executes at the defining site, in the definer's schema, with the definer's visibility and permissions for accessing SQL names. The caller of a definer's-privileges program class must have EXECUTE privilege on that class. This requires customers to load all of the (PL/SQL) code for an application into every schema where it will be used, in order to bind early to tables. This unnecessarily stores multiple copies of the program and requires extra compilations.

Oracle8*i* supports a new privilege mechanism called "invoker's-privileges" that Java programs can use. This privilege mechanism allows late binding to tables using the visibility and authorization of the caller of the procedure. If an invoker's-privilege program executes a SQL statement that queries a table T, it will access the table T in A's schema when user A invokes the program and will access the table T in B's schema when user B invokes the program. When Java programs are invoked from SQL or PL/SQL, they can be declared to execute with definer's privileges or with invoker's privileges. However, when Java programs invoke other Java programs, they always execute with invoker's privileges.

After you address the preceding three issues, you can publish a Java method to SQL using a CALL-spec. A CALL-spec is a PL/SQL subprogram spec that is annotated to indicate that a Java method implements the subprogram. A CALL-spec is similar to a PL/SQL external-procedure declaration and declares a SQL name and parameters and result types for a method that is implemented in Java. The CALL-spec can either be written by application developers to control the mapping between SQL and Java names and types or be generated by tools that automate the mapping of Java and SQL type mapping. The CALL-spec provides the following information about the program it describes:

- The language in which the designated subprogram is implemented, here "JAVA"

- The Java full name of the method, here "com.oracle.employee.emp.sal_grade" or "com.oracle.employee.emp.approve_raise"

- The mapping of SQL types to Java types of parameters and result

- The parameter modes

- Privileges with which the Java program will execute. This is optional. If not specified, the Java program executes with the definer's privileges

The following CALL-specs help clarify the concepts cited on the previous page:

```
      CALL-spec for the sal_grade Java method of class
employee.emp
SQL> CREATE OR REPLACE FUNCTION SGRADE (SAL IN NUMBER) RETURN
VARCHAR2
      2> IS LANGUAGE JAVA NAME
      3> 'com.oracle.employee.emp.sal_grade (float) return
java.lang.String';
      /
      Statement processed.
      CALL-spec for the approve_raise Java method of class
employee.emp
      SQL> CREATE OR REPLACE PROCEDURE APPRAISE (SAL IN NUMBER,
RAISE IN
      2> NUMBER, NSAL OUT NUMBER) AUTHID INVOKER
IS LANGUAGE JAVA NAME
'com.oracle.employee.emp.approve_raise(float, float, float [
])';
/
      Statement processed.
```

NOTE: The following example shows you how to use the Oracle SQL package. This hypothetical CALL-spec does not correspond to the example we are working with in this section. If you want to use the Oracle SQL

native mapping with this example, you will first have to modify your Java code and use the appropriate parameter types in the method declarations.

```
      CALL-spec for the approve_raise Java method of class

employee.emp with Oracle SQL native mapping

      SQL> CREATE OR REPLACE PROCEDURE APPRAISE (SAL IN NUMBER,

RAISE IN

      2> NUMBER, NSAL OUT NUMBER) AUTHID INVOKER

                3> IS LANGUAGE JAVA NAME

'com.oracle.employee.emp.approve_raise(oracle.sql.NUMBER,

oracle.sql.NUMBER,

oracle.sql.NUMBER [ ])';

/

      Statement processed.
```

RUN YOUR JAVA CODE

Now that the Java stored procedure has been registered with SQL, SQL can call it. Java can be called from SQL and PL/SQL in a variety of contexts: traditional database stored functions and procedures, database triggers, object type methods, or from within a PL/SQL package or subprogram (procedure or function). SQL can invoke Java in two different ways:

- *Using the "CALL" Statement* - A new SQL statement, the "CALL" statement, is introduced in Oracle8*i*. It begins with the keyword "CALL." It is similar in effect to the EXECUTE command of SQL*PLUS. However, it does not expand to a BEGIN...END. Using the "CALL" statement, you can invoke a Java stored program as a top-level procedure.The "CALL" statement invokes the CALL-spec that is used to publish the Java methods to SQL. The "CALL" statement can also be used with functions. Here is an example of how to use the CALL statement. Note that SGRADE and APPRAISE were the CALL-specs used to publish the Java methods.

- Calling a Java Stored Procedure from SQL as a top-level procedure using the "CALL" statement:

```
SQL> VARIABLE NEWSAL NUMBER

SQL> VARIABLE CURSAL NUMBER
```

```
SQL> VARIABLE RAISE NUMBER

                SQL> VARIABLE GRADE VARCHAR2

            SQL> EXECUTE :CURSAL := 50000;

            Statement processed.

            SQL> EXECUTE :RAISE := 5000;

            Statement processed.

            SQL> CALL APPRAISE(:CURSAL, :RAISE, :NEWSAL);

            Statement processed.

            SQL> PRINT NEWSAL

            NEWSAL

            -------------

             55000

                SQL> CALL SGRADE(:CURSAL) INTO :GRADE

                Statement processed.

                SQL> PRINT GRADE

                GRADE

                ------------

                   SMTS
```

- *Using a SQL Query* -Just as you can employ user-defined PL/SQL functions with SQL query statements, you can also use Java stored functions to manipulate the results returned from a SQL query. Again, from SQL, you need to invoke the CALL-spec that was used to publish the Java method to SQL. The following example shows how to use a Java method in a SQL query statement.

- Calling a Java Stored Function from a SQL Query:

```
            SQL> SELECT ENAME, SAL, SGRADE(SAL) FROM EMP;
```

Like SQL, PL/SQL can also invoke Java stored procedures and functions. PL/SQL can call Java stored procedures directly. Let us look at how this mechanism works. Because the Java stored program is wrapped with a PL/SQL proxy call descriptor, the calling PL/SQL program can simply invoke the PL/SQL CALL-spec. The syntax to call the proxy is essentially identical to calling another PL/SQL program. When the proxy subprogram is invoked, the Java VM automatically invokes the Java method. Here is an example of calling a Java method directly from PL/SQL:

- Calling a Java Stored Procedure from PL/SQL directly:

```
DECLARE

newsal NUMBER(7,2);

cursal NUMBER(7,2);

raise NUMBER(7,2);

BEGIN

…

APPRAISE(CURSAL, RAISE, NEWSAL);

…

END;
```

ACCESSING SQL AND PL/SQL FROM JAVA STORED PROCEDURES

So far, we have discussed how to develop and deploy Java stored programs in the Oracle8*i* database. With the help of some simple examples we demonstrated the steps needed to load, resolve, publish and invoke Java programs in the database. However, these examples were simplistic and did not involve access to any persistent state in the Oracle database. In this section, we look at how Java stored programs access persistent data and how they interoperate with PL/SQL. Java programs running in the database can access persistent data using JDBC or SQLJ. The Oracle JServer Java Virtual Machine has a special version of a JDBC driver that runs in the database. This embedded JDBC driver complies with the JDBC 1.22 specification and supports the same APIs that the client-side JDBC drivers support. Because this JDBC driver runs in the same address and process space as the database, it directly accesses SQL and PL/SQL without making any network round trips.

Along with the embedded JDBC driver, the Oracle JServer also has an embedded SQLJ translator that allows application developers to write applications that access persistent data using SQLJ. SQLJ is a standard way of embedding static SQL statements in Java programs, similar to how PRO*C allows you to embed static SQL statements in C. The embedded SQLJ translator translates the SQLJ programs into

pure Java programs with underlying calls to JDBC. The translated Java program is 100 per cent pure Java and can be compiled with the embedded Java byte-code compiler. Once again, the translated programs can use the embedded JDBC driver to access persistent data. Both the embedded JDBC driver and the SQLJ translator facilitate flexible application partitioning because applications developed on a client machine that access the database using JDBC or SQLJ can be easily packaged and deployed in the database with only minor changes.

Let's now enhance our previous example to access persistent data using SQLJ. The methods of the 'emp' class are being modified in the following ways: First, the 'sal_grade' method does not take the employees salary as an input parameter. It takes the employee name as the input parameter. Second, the 'approve_raise' method does not use the current salary, raise and new salary as parameters.

Instead, this method now takes the employee name, raise and new salary as its parameters. Here is what the new Java stored procedures look like:

```
package com.oracle.employee;

public class emp {

      public static String sal_grade (String empname) {

                float current_sal;

      #sql {select salary into :current_sal from emp where ename =

:empname };

                if (current_sal > 0 && current_sal < 50000) return "

MTS";

                      if (current_sal >= 50000 && current_sal < 100000)

return " SMTS";

                return ("Salary out of Range");

      }



      public static void approve_raise (String empname, float

raise, float new_sal [ ]) {
```

```
            float current_sal;
            int hire_time;
      #sql {select salary into :current_sal from emp where ename =
:empname };
      // We are assuming that employment_period is a PL/SQL
function that has been previously defined in the
      // the database that looks at the employees hire date and
returns the number of months the employee
      // has been with the company.
      #sql hire_time = { VALUES(employment_period (:empname)) };
            float percent_raise = (raise / current_sal) * 100;
            if (hire_time > 9) {
               if (percent_raise > 15 || percent_raise < 5) {
                  new_sal[0] = -1;
            }
      else {
                  new_sal[0] = current_sal + raise;
                  }
      }
      else {
                  new_sal[0] = -2;
                  }
            }
```

```
}
```

The preceding program shows how SQL can be called from Java using SQLJ. It can be translated and compiled on a client machine and then loaded into the database for execution as described in the previous section. The same program can also be directly loaded into the database. The embedded SQLJ translator converts the SQLJ program into a Java program with underlying calls to JDBC and the byte-code compiler will convert this Java program into the corresponding binary class files.

INVOCATION SCENARIOS FOR JAVA STORED PROCEDURES

Java stored procedures can be deployed in several configurations, including the traditional two-tier client-server configuration similar to PL/SQL stored procedures. They can also be called from an application running in a middle-tier application. Java stored procedures can be invoked from any database/Net8™ client. Net8 Connection Manager is a connection concentrator that joins a large number of users to a single server. Here are some invocation scenarios for Java stored procedures:

- *Java Client Via JDBC or SQLJ* - A Java client application can invoke Java stored procedures using JDBC calls it uses to invoke PL/SQL stored procedures today — either SQL92 escape syntax or Oracle SQL syntax. Both procedures are supported by Oracle's JDBC Drivers.

- Pro*, Oracle Call Interface and ODBC clients can also access the stored procedure.

- Oracle Developer™ FORMS clients can access Java stored procedures.

This paper has outlined Java stored procedures and how an application developer can develop and deploy business logic as a Java stored procedures in the Oracle8*i* database. In this section, we discuss some of the benefits of using Java stored procedures. The most important reason for the widespread use of stored procedures is that result set processing improves application performance by eliminating network traffic bottlenecks and allows more efficient use of server resources. Instead of sending large quantities of data across the network from the server to the client, data can be operated on in the database server, with only the results sent across the wire to the client.

Second, the enforcement of business rules can be centralized in a widely distributed organization. From the central Oracle8*i* database, these Java stored procedures can be replicated in two ways: first, to other Oracle databases to enforce rules between a central server and a branch office database for instance. Further, Java stored procedures can be replicated between an Oracle database and mobile clients with Oracle Lite databases.

Third, Java stored procedures are a part of the SQLJ standardization effort that has been underway for the past two years. Oracle has played a leading role in driving this standards effort. Oracle, IBM, Tandem, Sybase, JavaSoft, Informix and other partners have helped to define the standard. The standards effort has focused on two parts: Part 0 of the standard provides a standard syntax for embedding SQL statements in Java programs. Part 1 is aimed at providing standard syntax for database stored procedures and triggers written in Java. Oracle8*i* implementation of Java stored procedures complies with upcoming ANSI/ISO standards for Java and SQL interoperability. These stored procedures are portable across databases from multiple vendors.

Fourth, Java stored procedures complement PL/SQL stored procedures as an open alternative for both client-server as well as transactional intranet/Internet applications that are deployed in multitier configurations. By adding Java as a server programming language, Oracle aims to open the RDBMS as a general purpose server platform to all Java programmers.

Finally, the two-way interoperability Oracle provides between Java and PL/SQL allows reuse of applications. Existing PL/SQL stored procedures can easily be reused from Java. In addition, Java stored procedures can be reused from PL/SQL.

These benefits bridge a number of deployment configurations — two-tier client-server configurations and multi-tier intranet and extranet deployments.

SUMMARY

This paper has given you an overview of how to develop enterprise applications in Java and deploy them in the Oracle8*i* database. The Oracle JServer Java Virtual Machine integrated with the Oracle database provides an open server platform for deploying Java applications that is scalable, robust, secure, highly available and easily manageable. To deploy Java stored programs in the Oracle8*i* database you need to load your Java programs into the database, publish the Java methods to SQL and call the Java programs in a variety of different contexts. Because these Java programs run in the same address space as the database server and can leverage the embedded JDBC drivers, they need not make network round trips to access SQL data and are therefore highly optimized to deliver high performance.

Java is tightly integrated with the database and can seamlessly interoperate with SQL and PL/SQL. Oracle8*i* implementation of Java stored programs offers a variety of benefits to application developers, including high application throughput, less network traffic and seamless interoperability among Java, SQL and PL/SQL. Java stored procedures are easily replicated between Oracle databases and Oracle's mobile database (Oracle Lite), allowing you to centralize enforcement of business rules in widely distributed organizations. In addition, because Java stored procedures are based on open standards, they allow portability across several database vendors.

Oracle Corporation
World Headquarters
500 Oracle Parkway
Redwood Shores, CA 94065
U.S.A.

Worldwide Inquiries:
+1.650.506.7000
Fax +1.650.506.7200
http://www.oracle.com/