IMPLEMENTATION OF AN IMPROVED EMBEDDED SQL FOR JAVA

by

Louis M Bradley

A Thesis Submitted to the Faculty of

The College of Engineering and Computer Science

in Partial Fulfillment of the Requirements for the Degree of

Master of Science

Florida Atlantic University

Boca Raton, Florida

December, 2012

IMPLEMENTATION OF AN IMPROVED EMBEDDED SQL FOR JAVA

by

Louis M. Bradley

This thesis was prepared under the direction of the candidate's thesis advisor, Dr Martin Solomon, Department of Computer Science, and has been approved by the members of his supervisory committee. It was submitted to the faculty of the College of Engineering and Computer Science and was accepted in partial fulfillment of the requirements for the degree of Master of Science.

SUPERVISORY COMMITTEE:

Martin K. Solomon, Ph.D.
Thesis Advisor

Robert B. Cooper, Ph. D.

Ionut Cardei, Ph. D.

Borko Furht, Ph.D.
Chair, Department of Computer and Electrical Engineering and Computer Science

Mohammad Ilyas, Ph.D.
Interim Dean, The College of Engineering and Computer Science

Barry T. Rosson, Ph.D.
Dean, Graduate College

November 19, 2012
Date

ACKNOWLEGDMENTS

I would like thank my advisor, Dr Martin K. Solomon, for putting up with me during the design, development and testing of the SQLJE pre-compiler. His guidance, knowledge and suggested improvements were greatly appreciated. I would also like to thank Dr Ionut Cardei and Dr Robert Cooper for serving on my thesis committee and providing their valuable insights and to Jose Luis Hurtado for his thesis, upon which this project was based.

ABSTRACT

Author: Louis M. Bradley

Title: Implementation of an Improved Embedded SQL for Java

Institution: Florida Atlantic University

Thesis Advisor: Dr. Martin K. Solomon

Degree: Master of Science

Year: 2012

The Java Development Environment defines SQLJ as a standard way of embedding the relational database language SQL in the object-oriented programming language Java. Oracle Corporation provides an extension of SQLJ that supports dynamic SQL constructs for the processing of SQL commands that are not completely known at compile time. Unfortunately, these constructs are not sufficient to handle all dynamic situations, so that the programmer has to depend on other SQL embeddings, such as JDBC, in addition to Oracle's SQLJ.

In this thesis we implement several extensions to Oracle's SQLJ so that all dynamic situations can be programmed in SQLJ, without resorting to other SQL embeddings. We also add a sub-query based for loop facility, similar to the one provided in Oracle's database programming language PL/SQL, as an improvement over the iterator constructs that SQLJ provides.

This thesis discusses the design, development and implementation of these SQLJ extensions, and provides applications that show the utility of these extensions in terms of clarity and power.

IMPLEMENTATION OF AN IMPROVED EMBEDDED SQL FOR JAVA

1. INTRODUCTION

The ability to intuitively and efficiently access Relational Database Management Systems (RDMS) from systems development platforms has been a topic over which many papers and systems proposals have been based (see [1] for references). The inability to seamlessly marry the two technologies was named the "impedance mismatch" problem by Copeland and Maier in their proposal to add database commands to the SmallTalk language [2].

Since then, two suppliers of development environments and RDMSs have emerged as leaders in the industry, Microsoft with its Dot Net suite of programming languages coupled with its SQL Server RDMS, and Oracle with its Java development platform and Oracle RDMS. Both suppliers have added "extensions" to their languages to try and reduce the impedance mismatch, but neither has completely integrated Standard Query Language (SQL) statements directly into their programming languages [4, 7].

In his 2012 thesis [1], Jose Luis Hurdato discussed the limitations found in the current interface between programming languages and RDMSs including surveying the current abilities provided by Microsoft and Oracle. In that thesis he detailed the different approaches to providing access to RDMS data used by different platforms and concluded that the best approach available to date was the Standard Query Language for Java (SQLJ) pre-compiler used by the Java language for Oracle connectivity. There he also

pointed out that SQLJ lacked several features that many times required the use of both SQLJ and direct Java Database Connectivy (JDBC) calls to compensate.

He then recommended several additions to the SQLJ command set to both reduce the need to use JDBC calls and to make the programs easier to read. These recommendations included a new Dynamic Host Expression, a dynamic sub-query based for loop and a new SQLJ command used to retrieve metadata. These recommendations were implemented by this author in a pre-compiler to SQLJ written in Java. This thesis discusses the design, development and implementation of this pre-compiler and presents comparisons between the standard SQLJ and the enhanced commands now available.

2. HISTORY

When the electronic processing of data was at its infancy, the options and abilities for data storage were limited. At the start of the computer generation, punched card media combined with magnetic tape storage was the norm. Using this media required the data stored in these early systems to be of a fixed length and sequential nature. The access and manipulation of the data in these systems was a simple task requiring only a few basic I/O statements (Read and Write) coupled with exact data definitions within the language. To facilitate this access, each computer manufacturer supplied development languages such as COBOL or FORTRAN that included these basic I/O commands [6].

With the introduction of disk-based processing came the ability to individually access a specific record on the disk, and the option to update that record or remove it from the file. Such direct access file systems added new I/O commands and abilities, and the hardware manufacturers added these commands to their base programming languages. A similar approach was followed when early hierarchical proprietary database systems were developed [6].

As the use of computers to process more and different types of data increased, the "all in one" approach where the computer manufacturer supplied the hardware, operating systems, database management system and programming languages began to be split with many "third party" companies emerging to compete for database management and software development systems. This divergence of hardware, development software and

database management systems became even more prevalent with the introduction of personal computers and client/server processing. Now, one vendor supplied the hardware, another the operating system for that hardware, a third the computer language used for software development and a fourth the database management routines [6].

Although each component that made up a development environment was superior to the all-in-one approach taken during the mainframe days, we were now faced with how to best integrate a programming language and a database system that were most of the times developed by separate companies. In addition, each programming language requires access to each of the RDMSs, and, although the different SQL "languages" are similar, there are subtle differences in syntax that must be taken into consideration in the interface. Initially these interfaces were implemented using embedded SQL and a pre-compiler as was done with IBM's System R and Ingres. These interfaces later changed to application programmer interfaces (API) that basically passed strings with encoded SQL statements from the program to the DBMS. Such interfaces include ODBC (Open Database Connectivity) and OLEDB (Object Linking and Embedding Interface) from Microsoft and JDBC (Java Database Connectivity) from Java [4,6,7].

These "string" type interfaces became the norm for integrating database access into programs, but still left a lot to be desired due to the hidden nature of the actual SQL commands within these strings. Programmers were faced with one way of accessing the database through external tools using the base SQL commands, or the more advanced procedural commands supplied by the RDMS, and a second way within their programs where similar commands were first assigned to string variables, and those variables then passed into the database management routines.

As one solution, a pre-compiler for the C and C++ languages was developed named Pro*C. The Pro*C pre-compiler reads files of type "PC" that include SQL commands embedded within a C program. The embedding includes the ability to define a set of variables that are accessible both by the host program and the SQL commands being interpreted, and a structure where by SQL statements can be written that closely match the same statement that would be entered into a query tool outside of the C environment. [8]

As mentioned previously, the separation of database management, software development and hardware pushed the specialization that caused the impedance mismatch issue where the interface between host languages and database management were no longer seamless as they had been when one entity developed all components. What is ironic is that today, the two largest suppliers of database management systems, Microsoft and Oracle, now are also the suppliers of the major development environments with Microsoft supplying the Visual Studio Dot Net suite of development tools and Oracle acquiring the Java development platform from Sun Systems. Now one would think that since we have both database management and program development are back in the hands of single suppliers that they would modify their development environments so that more direct access to at least their own database management systems would be available. Unfortunately, that is not the case. Although each supplier has added structures to their development languages, the additions are far from true integration between their System Development Environments (SDKs) and DBMS. [4,7]

On the Microsoft side, they developed an interface called LINQ to SQL. LINQ makes use of an object relational mapping facility along with its own set of "query

expressions". These expressions are similar but not the same as standard SQL, with their semantics and syntax being quite different. A programmer familiar with normal SQL query will have to relearn how to do even simple queries with LINQ. [7]

On the Oracle side, their improved interface is called SQLJ and is a Java pre-compiler that takes specifically marked statements and converts them into JDBC calls with minimal change in the syntax of the SQL statement. This approach is similar to what was introduced in Pro*C and SQLJ statements closely follow the same syntax and semantics as SQL statements processed by the host DMBS. In addition, the Java development environment and the SQLJ pre-compiler allow for syntax checking and column validation at compile time. Although a version of SQLJ introduced with Oracle 9i was touted as able to handle any dynamic SQL by the lead developer ([4] preface), the pre-compiler cannot handle true dynamic SQL statements and does not have a facility to expose metadata to the programmer, still causing the programmer to revert to JDBC to accomplish those tasks [4,5].

These various approaches to connectivity between development environments and database management systems were covered in Hurtado's thesis in which he suggests that the SQLJ interface, even with its limitations, was the best implementation of an interface between an SDK and an RDMS. His paper suggestes several enhancements to the SQLJ interface that would provide for dynamic SQL statement processing, metadata support, and an improved method of retrieving data from Select statements [1].

3. PROJECT OVERVIEW

3.1 Jose Luis Hurtado's Thesis

This project is a follow-on to Jose Luis Hurtado's July 2012 thesis, and implements the recommendations he made for enhancements to the SQLJ Java to Oracle interface. Based on his findings, this interface already provides a great deal of control to the application programmer while fairly closely aligning itself with similar syntax and structure as regular SQL statements. The main issues with the current SQLJ implementation dealt with limitations requiring the possible use of both SQLJ and JDBC within the same program, and the awkwardness of the way that Select statements are processed [1].

This thesis project will add four new features to the SQLJ command set that should make the source code more readable while eliminating the need of using both JDBC and SQLJ within the same program. The four features include: Dynamic Host Expressions, a Dynamic sub-query based for loop, metadata support and the ability to process SQL commands when nothing is known about the command.

3.2 Dynamic Host Expressions

Currently SQLJ supports two variable types that allow SQLJ to share information between Oracle and Java, Host Expressions and Meta Bind Variables. A Host Expression allows a programmer to pass a Java variable into SQLJ as a parameter, but the exact type of the Host Expression is needed, and each host expression must be

individually declared in the statement. A Meta Bind Variable allows the programmer to replace part of an SQLJ statement with a string built within the Java code.

The new **Dynamic Host Expression** will allow programmers to define an array of type Object, and programmatically fill the array with values. This allows statements that rely on repetitive parameters or values to dynamically determine the number of parameters at run time. Dynamic Host Expressions can be used in the Values clause of an Insert statement, in the parameter list of a function or procedure call, or can be used in other commands when associated with the new verb Applying;

An example usage of the Dynamic Host Expression in an Insert statement could be:

```
1   #sqlje {insert into :{tablename} values (:[tablevalues])};
```

In the above example, the table into which one is inserting will be determined by the value of the host express tablename which could change based on the processing of the program. The tablevalues array would be instantiated and filled with the correct number of values that match the table into which the values are to be inserted.

Other examples that could use the Dynamic Host Expression include:

```
1   #sqlje {call :{procedure_name} (:[procedure_params])};
2   #sqlje function_result = {values :{function_Name}(:[function_input])};
3   #sqlje {update :{tablename} set Field1 = ?, Field2 = ?
4           where Field3 = ? APPLYING :[update_Values]};
```

3.3 Dynamic sub-query based for loop

SQLJ's current set-up for processing Select statements relies on the use of an Iterator class that is pre-defined with Select statement information before processing. Then the Iterator class is used to traverse the result-set returned from the Select statement. For SQLJE, we will eliminate the need for an Iterator class and instead allow the programmer to both instantiate a class to hold result-set information and pass the Select statement to that class in a single statement followed by a processing loop in which the data from the result-set is processed.

For example, in normal SQLJ, the following steps would be used to set up a query of data from a table (See Appendix A for table definition):

```
1  #sql iterator SIterator   (String snum, String sname, int status, String city);
2  SIterator sx;
3  #sql sx =  {SELECT s# snum, sname, status, city FROM s ORDER BY snum};
4  while(sx.next()) {
5      System.out.println(sx.snum()+"  "+ sx.sname()+"  "+ sx.status()+"  "+sx.city());
6  }
```

With the new sub-query based **For** loop, this code can be replaced by:

```
3  #sqlje {for sx (String snum, String sname, int status, String city) in
   (select s#, sName, status, city from s order by s#)};
   {
   System.out.println(sx.snum+ "  " + sx.sname + "  " + sx.status + " " + sx.city);
   }
```

In addition we will also allow the programmer to completely eliminate the static declaration of internal variables and automatically generate these variables as an array within the result-set class that can then be accessed by ordinal number or by name. The following statements use this approach to retrieve the same data as above:

```
1  #sqlje {for sx in (select s#, sName, status, city from s order by s#)};
2  {System.out.println(sx.Column("s#")+ "  " + sx.Column("sname") + "  " +
3  sx.Column("status") + " " + sx.Column("city"));}
```

The dynamic ability of the new result-set class lets us take this one step further, allowing for our pre-compiler to handle a Select statement entirely encoded in a string and nothing else is known about the statement.

```
1  String sSelect = "Select s#, sName, status, city from s order by s#";
2  #sqlje {for sx in (:{sSelect})};
3      {
4       System.out.println(sx.Column("s#")+ "  " +   sx.Column("sname") + "  " +
5            sx.Column("status") + " " + sx.Column("city"));
6       }
```

3.4 Metadata support

Metadata for a database object provides structural information about that object. Metadata for a Select statement provides information about the result-set of that Select statement which includes the name of each column of the result-set, the type of data, the maximum length of the data, whether or not the data can be null and, for numeric data, its precision. There is not a structure in the normal SQLJ implementation that provides this information requiring programmers to make use of JDBC calls to get this information.

In Hurtado's thesis he suggests the use of a "Declare" statement to both set up the result-set class for a Select statement, and to acquire metadata information from that Select. In our implementation we chose to instead remove the need for the Declare statement for our dynamic sub-query based for loop and to include a new command, Describe to specifically acquire meta-data information. The verb Describe seemed to be more conducive to the action that was taking place. An example usage could be:

```
1  Describe dbTableDef = new Describe();
2  #sqlje {Describe into dbTableDef (Select * from S)};
```

In addition, when a dynamic Select statement is processed (one without the explicit list of fields), the result-set class created also contains basic metadata that includes the number of columns in the result-set and the names for each column. This is how the dynamic Select statement can return values based on column name.

3.5 Ability to process a dynamic SQL command

Many times what needs to be modified in a database requires the program to dynamically create SQL statements that could change which table is being modified, which fields within the table and under which conditions. The original JDBC easily handled this situation since all of the commands used in the JDBC interface were really just the value of strings being passed through to the Oracle RDMS. SQLJ does not have this ability.

To provide a way to process dynamic non-Select SQL commands, a new verb was added to SQLJE, **Execute**, which will allow a programmer to code the entire SQL command and then pass it through the SQLJE pre-compiler. The feature can be used with or without a Dynamic Host Expression identified by the Applying command. An example using the S-P-J database structure (Appendix A) could look as follows:

```
1  String sSQL = "UPDATE P SET pname = ? WHERE p# = ?";
2  Object pData = new Object[2];
3  pData[0] = "Marty";
4  pData[1] = "P1";
5  #sqlje {EXECUTE :{sSQL} APPLYING pData};
```

4. IMPLEMENTATION

The implementation of the enhancements described in Chapter Three required a four step process to design and code the program that converted the new SQLJE statements into JDBC commands understandable by the Java compiler. The first step was to see how the SQLJ preprocessor handled existing commands, and what differences could be identified based on the SQLJ statements processed. The next step was to determine how to integrate the enhancements into the existing SQLJ framework. The third step was to determine the best tool in which to parse the statements and code the commands. Finally the program itself was coded and then subjected to a series of tests to ensure it was operating correctly.

4.1 Coding Requirements

For this author to be able to code the enhancements required, a more in-depth "behind the scenes" look at the way SQLJ processes different types of SQL statements was needed. To accomplish this, the author used the Oracle Jdeveloper SDK platform to code each different type of SQLJ command, and then analyzed the output to determine how the final command was constructed and what internal database management routines were used to process the statement.

This analysis resulted in breaking the SQLJ commands into three different categories,

1. Comma Separated Immediate Commands: This category of commands include the INSERT, Procedure Call and Function Call statements. In all three cases the commands include a block where either a variable list or a parameter list is included within parenthesis separate by commas.
2. Non-Comma Separated Immediate Commands: This category of commands include the Update and Delete statements. In these cases both the parameter list and (in the case of the Update command) the SET list use a Variable = Value structure.
3. Iterators and SQLJ Select statements: This category of commands include both the construction of the Iterator class itself, how that class makes itself available to the main Java program and the construction of the SQL Select statement passed to the class to create a result-set. This category is also where the most extensive changes to the SQLJ logic would be performed since the concept of the Iterator would be replaced.

To ensure compatibility with the usage of SQLJ in Java servlets, a separate set of statements were processed that made use of JDBC default contexts instead of SQLJ connections to the database. The resulting use of different internal method calls and DBMS routines were documented.

4.2 Integrating New Functionality

The next step was to develop sample Java programs for each category of statement that would implement the new enhancements using the structures and internal JDBC function calls discovered in the previous step. For the Immediate Command categories, this involved determining how the Dynamic Host Expression would be integrated into the SQL statement generation logic both when used in comma separated commands and when used in conjunction with the Applying statement. A small sample program for each command type was developed that would be used in the development and testing step as the template for that command.

A similar process was followed for the For -> Select statements, but with separate results depending upon whether or not the For->Select was static or dynamic. In both cases, a new class is created, but the actual working of the class is completely different. Sample classes and in-line replacement code for both the static and dynamic case were created and tested.

For the new Describe command, a new Java class (called Describe) was developed that formed the backbone for exposing metadata to the program. The class can be instantiated in multiple places within a Java program with different Select statements. A small "Test Describe" program was written to act as the template for the conversion of the command to actual Java/JDBC code.

Based on the findings from the immediate commands, a template was developed that could handle any non-Select SQL statement when nothing was known about the

statement at compile time. A sample program and template was developed to be used when this feature was coded.

It was decided that the SQLJE pre-compiler would not check for variable existence and provide for in-line error checking, although this feature may be added in a subsequent version. So, the pre-compile will directly pass through any of the three variable/expression types through without determining if the variables behind those expressions exist.

4.3 Tool Selection

Once the sample programs and templates were complete, the way that the SQLJE statements would be converted to Java/JDBC code needed to be determined. What was needed was a tool that could read a program, find the SQLJE statements and parse the statements. The tool would then need to create the required output to replace the statement with the correct calls to JDBC routines while passing all other lines in the program directly through. The tool would also have to create new classes for the For-Select statements and be able to distinguish different structures based on the syntax of the command.

Two different approaches were reviewed. The first was to make use of a lexical/syntax analysis tools such as Flex and Bison to scan and parse the various SQLJE commands and then provide the parsed lines to a Java program. The second approach eliminated the use of the lexical/syntax analysis tools and instead used regular expressions coded directly within a Java program to scan and parse the input. After reviewing the two approaches, the option using only a Java program and regular

expressions was selected. Using this approach would concentrate all development effort in one place allowing for changes and additions to be more easily coded. It also reduced the learning curve required to make use of the lexical tool and incorporate its output into the Java program.

4.4 Development and Testing

An incremental prototyping methodology was used in the development of the SQLJE pre-compiler. The first command implemented was the INSERT command using a Dynamic Host Expression. This command required the potential parsing of both a variable list and a parameter list and provided several methods that were used in subsequent command conversions. It also required the ability to recognize and process the Dynamic Host Expression which provided a base method to be used by other commands in a Applying clause or within a function or store procedure call.

The Update and Delete commands were coded next and were perhaps the easiest to convert due to their fairly rigid structure. The methods developed for decoding the different host expressions were refactored and reused by these commands.

The next command to be implemented was the **For->Select** statement that used a variable list. This Select command required that the parser generate both a class to process result-set data at the same time that the in-line conversion of the command was being performed. To handle this, a separate buffer was created to hold the code generated for the result-set class. The buffer was then appended to the end of the pre-compiled code once all in-line code was completed for all commands in the program being pre-compiled.

The generated result-set classes were then enhanced to be able to process a dynamic **For-Select** statement. Instead of specific fields being named in the class, an array of objects was used that would receive the data from the result-set row by row. At the initial implementation, only ordinal value assignments were going to be allowed, but this seemed to be very restrictive. To improve upon the ordinal only assignment so that column named value assignments could be implemented, the metadata for the Select statement was retrieved and used to generate an overloaded method so that either column name or ordinal position could be used.

The ability to extract metadata information was then coded. This task involved coding a general purpose **Describe** class with the basic components required for metadata extraction and exposure along with the code to convert the **Descibe** command into code that would access that class so that the data would be populated.

A basic Procedure Call was the next command to be implemented. The initial version allowed only "IN" type parameters and closely followed the code written for the Insert statement.

Function Calls using the **Values** command were then coded. Although initially it was thought that the same methods for parameter parsing could be used that were developed for the Insert and Procedure Call, because of the unique way that the Function returned a result, these routines needed to be enhanced so that parameter tags (":1") instead of question marks ("?") could be used so that the result could be registered as a out parameter. These changes were required both to the static parsing of the functions parameters, and when a Dynamic Host Expression was used.

The final command to be implemented was the **Execute** command that would take any non-Select SQL command stored in a Meta Bind Expression and process it with or without a Applying. The same routine that processed the **Update** and **Delete** commands was used to compile the **Execute**.

Once all the initial commands were coded and tested, several of the repetitive tasks in the program were refactored so that the code was placed in general purpose methods, and the commands retested to ensure they continued to work correctly.

The next step involved the ability to handle comments and strings within programs and to be able to correctly know when a command was a true SQLJE command. To accomplish this several regular expressions were developed that would identity comments and pass these directly to the output until the comment was completed. Other regular expressions were used to find strings both with double and single quotes. To keep the integrity of the strings while enabling the processing of the commands, each string was replaced with a coded literal. The command was then parsed and converted as needed, and then the coded literals were replaced with their string counterparts. This enabled the command parser to operate without having to be concerned if it was dealing with a string or a literal pushing those decisions to the front end string scanner and the routine that writes the line of code to the output file.

After the initial commands, comment scanner and string scanner were completed, several other changes were added to the program as new ideas were explored or features missed in the initial implementation were discovered. These changes included the addition of ability to use a “Default Context” database connection so that the SQLJE pre-

compiler can be used for Java Servlet development and the addition of Out parameters to the Store Procedure function call.

Small programs were used through-out the development life cycle to test the functionality of the pre-compiler. As each command was completed, that command was used in several different ways to determine that the generated code worked correctly. Once all commands were completed, several mini-programs as described in Chapter 6 were written, passed through the pre-compiler and run to both test the usability of the pre-compiler and to demonstrate the differences between the SQLJE commands and similar programs written in SQLJ and JDBC.

5. USING THE EXTENDED SQLJ STATEMENTS

As discussed previously, our extensions to Oracle's SQLJ database access statements (#SQLJE) provide dynamic SQL capabilities to access data from an Oracle database without directly using the JDBC statements. These features should allow a programmer to use only the SQLJ and SQLJE statements and to no longer have to switch between SQLJ and JDBC, as must be done with the normal Java SQLJ in many dynamic SQL situations.

In addition, we provide a facility for processing Select statement output that we believe is superior to the standard SQLJ iterator facility.

5.1 Extended SQLJ Dynamic Host Expressions

Before going through the syntax for each new or updated extended SQLJ statement, we will first discuss how Java expressions can appear within these statements.

There are three different Java "expression types" that are available in SQLJE statements:

Expressions of the form :{Expression} represent "**Meta Bind Expressions**".

Expressions of the form :(Expression) represent "**Standard Host Expressions**".

and

Expressions of the form :[Expression] represent "**Dynamic Host Expressions**"

Meta Bind Expressions are implemented in SQLJE in a similar manner as SQLJ. The value of the expression will replace the bind expression during the execution of the statement:

For example, let's assume that we have three Java String variables as follows:

```
1  String sTableName1 = "Employee";
2  String sTableName2 = "Discount";
3  String sFields = "Emp_No, Discount_Percent";
```

and that we use these Java variables in a SQLJE statement as follows:

```
1  #SQLJE {Insert into :{sTableName1 + "_" + sTableName2}
2          (:{sFields}) Values (10,34.5)};
```

When parsing is completed for this statement, the resulting SQL statement would then read:

```
1  sSQL = "Insert Into " + sTableName1 + "_" + sTableName
2          +"(" + sFields + ") Values (10,34.5)";
```

When the program actually processes, the values from sTableName1, sTableName2 and sFields will replace the variables so that the resulting SQL statement would be:

```
1  "Insert Into Employee_Discount (Emp_No, Discount_Percent) Values (10,34.5)"
```

Standard Host Expresssions are also implemented in a similar fashion in SQLJE as they are in SQLJ. Each Standard Host Expression represents a single value that can be used in the Where Clause or Select-List of a SQL statement. Expanding on the above example, we will add two new variables to our Java program:

```
1   Int nEmpNo = 101;
2   Float nDiscountPercent = 6.5;
```

And our SQL statement would be changed to:

```
1   #SQLJE {Insert into :{sTableName1 + "_" + sTableName2
2           (:{sFields}) Values (:nEmpNo, :nDiscountPercent);
```

At run time after the above is parsed and the values of the variables are used, our SQL statement will be converted to the following:

```
1   "Insert Into Employee_Discount (Emp_No, Discount_Percent) Values (?,?)
```

In addition the parser will add statements that will use the Standard Host Expressions in SetObject statements to introduce the variables as the resolution of parameters for the SQL statement:

```
1   "sSql.SetObject(1,nEmpNo);"
2   "sSql.SetObejct(2,nDiscountPercent);"
```

The prepared statement would then be sent to the Oracle Database Management System for execution.

A **Dynamic Host Expression** is a new host expression type recognized only by SQLJE dynamic SQL statements. The Dynamic Host Expression must have as its value an Array of type Object and can hold multiple values used in a single SQL statement. Continuing with the example above, one would first define an Object Array, and then set each element of the array to the value needed for the SQL statement:

```
1  Object[] oEmpData = new Object[2];
2  oEmpData[0] = 123;
3  oEmpData[1] = 6.5;
```

One could then change the Insert statement we have been using as follows:

```
1  #SQLJE {Insert into :{sTableName1 + "_" + sTableName2}
2          (:{sFields}) Values (:[oEmpData])};
```

When the parser encounters this notation, the SQL statement is rewritten at run time in the same way as when Standard Host Expressions were used:

```
1  "Insert Into Employee_Discount (Emp_No, Discount_Percent) Values (?,?)
```

The number of "?s" added to the statement will match the number of elements of the array. The parser will additionally add the SetObject statements to the program, one for each element in the Array.

Dynamic Host Expressions can be used in three different contexts. In an Insert (as above) the Dynamic Host Expression replaces the contents of a Values clause. For a Procedure or Function call, it replaces the list of parameters being passed to the Procedure or Function. For all other statements, the Dynamic Host Expression must be used in conjunction with the verb "Applying". Please see how these Dynamic Host expressions can be used in each type of statement below.

5.2 Select Statement Processing with SQLJE For Statements

In standard SQLJ, an object called an Iterator is used to provide access to the rows returned in a Select statement. In the SLQJ implementation an Iterator is in reality a class generated by the SQLJ parser where the actual retrieval of data and assignment to java variables is accomplished. Although this structure does provide for data access, the use of the separate Iterator feels awkward to programmers.

In our SQLJE enhanced Select statements, we continue to make use of a separate class to do the actual work of retrieving records and assigning database information to Java variables, but instead of using a separate Iterator declaration, we use a "For" syntax which provides for more convenient processing as in Oracle PL/SQL and ANSI SQL-1999 SQL/PSM.

The SQLJE implementation of Select statements supports both a **static** and a **dynamic** version. In the **static** version, all information needed to correctly process the Select is incorporated into the SQLJE statement. An example query could be:

```
1  #SQLJE {for Discounts (int nEmpNo, float nDiscount)
2          in (select * from Employee_Discount)};
```

```
{
System.out.println “The discount for Employee “ + Discounts.nEmpNo
        + “ is “ + Discounts.nDiscount;
}
```

In this case, the SQLJE parser will create a new class and then assign the name “Discounts” to it. The new class will be specifically coded with only the field names mentioned in the field list of the “for Discounts” clause, so even though the Select clause said to pull all fields from the table, only the two fields mentioned in the field list will be included in the class.

The statement would then be parsed and passed to the “Discounts” class. The instructions after the SQLJE statement would be converted to:

```
while (Discounts.Next() == true){
        System.out.println “The discount for Employee “ + Discounts.nEmpNo
        “ is “ + Discounts.nDiscount;
}
```

In the dynamic implementation of a Select statement, the field list is removed and replaced by special code added to the class doing the data retrieval. For example, the above Select statement could be coded as follows:

```
#SQLJE {FOR Discounts IN
        (SELECT Emp_No, Discount_Percent from Employee_Discount)};
```

Since no field list is included, the Query Class must use result-set metadata returned from the query to set up the column values for each row in the query. This allows for the processing statements associated with the SQLJE statement to provide for two different ways to access the data, by ordinal value or by column name.

An example processing block for the above statement could be:

```
{
        System.out.println "The discount for Employee " + Discounts.Column(1) +
        + " is "  + Discounts.Column("Discount_Percent"));
}
```

In this example, both ways of accessing data from the query are shown, Discounts.Column(1) will return the data in the first column as specific in the Select clause which in this case would be "Emp_No". The Discounts.Column("Discount_Percent") entry would return the value of the column named "Discount_Percent"). (Please note, the indexes into the columns is "one" based to match the standard used by Oracle.)

The Select statement syntax for SQLJE allows Dynamic Host Expressions, Standard Host Expressions and Meta Bind Expressions to be used as described at the beginning of this chapter. For example, the following Statement could be a valid SQLJE for Select statement:

```
#SQLJE {for Discounts (int nEmpNo, float nDiscount)
        in (select * from :{sTableName1 + "_" + sTableName2}
```

```
3           Where Emp_No = ? and Discount_Percent = ? and Emp_No > :nEmpNo)
4           Using :[oEmpData]};
5           {
6           System.out.println "The discount for Employee " + Discounts.nEmpNo
7           + " is " + Discounts.nDiscount;
8           }
```

In the above example, the parser will replace the ":{sTableName1 + "_" + sTableName2}" with the values found in those two variables concatenated together with an underscore. It would then prepare the statement with three parameter place holders, assign the values from the oEmpData array to the first two parameters, and assign the value of nEmpNo to the third parameter. The statement would then be passed to Oracle for processing.

At the most flexible, the entire statement can be loaded into a Meta Bind Variable resulting in an SQLJE statement like the following:

```
1   String sSelectDiscount = "Select * From Employee_Discount";
2   #SQLJE {for Discounts in (:{sSelectDiscount})};
3           {
4           System.out.println "The discount for Employee " +
5           Discounts.Column("EmpNo")
6           + " is "  + Discounts.Column("Discount_Percent");
7           }
```

5.3 Using the Insert statement with SQLJE

The major enhancement included in the SQLJE extensions for the Insert statement involves the use of Dynamic Host Expressions. For an Insert statement, these expressions can be directly included within a Values statement, or appear as part of a Using directive in either an Insert…Values or Insert…Select statement.

Here are several examples using the data definitions from above showing how Dynamic Host Expressions can be used:

```
#sqlje {Insert Into :{sTableName1 + "_" + sTableName2
        (:{sFields})
        Values (:[oEmpData])};
```

```
#sqlje {Insert Into :{sTableName1 + "_" + sTableName2}
        (Emp_No, Discount_Percent)
        Values (?,?) Using :[oEmpData]};
```

```
String sUpdateFields = "Emp_No + 1, Discount_Percent * .9";
#sqlje {Insert Into Employee_Discount (:{sFields})
        Select :{ sUpdateFields } from Employee_Discount
        Where Emp_No = ? and Discount_Percent = ? Using (:[oEmpData])};
```

5.4 Using the Update statement with SQLJE

The SQLJE Update statement can use Dynamic Host Expression arrays to assign values to variables for either in the Set clause or the Where clause of the command. An example Update command using the Employee Discount table include:

```
Object[] oDiscountData = new Object[2];
        oDiscountData[0] = .20;
        oDiscountdata[1] = 231;
        #sqlje {Update Employee_Discount
                Set Discount_Percent = ? Where Employee_ID = ?
                Using :[oDiscountData]};
```

5.5 Using the Delete statement with SQLJE

The SQLJE Delete statement can use Dynamic Host Expressions arrays to supply values to the Where clause of the command. An example Delete command could be:

```
#sqlje {Delete from :{sTableName1 + "_" + sTableName2}
        Where Employee_Number = ? and Discount_Percent = ?
        Using :[oEmpData]};
```

5.6 Calling a Stored Procedure with SQLJE

The call to a stored procedure can use Dynamic Host Expression arrays to supply parameter value to the procedure. The call to the stored procedure will be modified based on the number of element in the Dynamic Host Expression.

Suppose our Employee Discount system includes a stored procedure name "BumpDiscount" that accepts an Employee_Number and a percent increase/decrease to the Discount Percent, the SQLJE statement to call that store procedure could be coded as follows:

```
1   #sqlje {call BumpDiscount (:[oEmpdata])};
```

The resulting SQL statement would be generated with "?"s in the parameter field with the number of "?" equal to the size of the oEmpdata array. The program would generate setObject statements iterating through oEmpData array.

```
1   Call BumpDiscount (?, ?)
```

A procedure call may contain parameters that are input only, output only or in/out. The verb "Via" has been included into the SQLJE syntax to help handle this situation. "Via" will be the tag to indicate that the next token is another Dynamic Host Expression with the same number of items as in the Dynamic Host Expression used in the parameter list. This Dynamic Host Expression will be an array of integers. If the value of an array item is zero, the corresponding parameter enter will be considered an input parameter. If it is a one, the parameter will be an output parameter, and if it is a two, the parameter will be in/out. The SQLJE pre-compiler will then create the appropriate parameter registration entries to handle the In and Out type parameters. For example, assume we have set up an array name oParmType with a 0 in both elements:

```
1   #sqlje {call BumpDiscount (:[oEmpdata]) VIA :[oParmType]};
```

5.7 Calling a Function using SQLJE

Function calls are implemented in SQLJ and SQLJE through the use of the **VALUES** statements. Let's assume that we have an Oracle Function as part of our Employee Disocunt System named "DoubleDiscount" that when passed an Employee_ID and a Check_Percent will check to see if the Check_Percent > 2 * Discount_Percent of that employee. If it is, the function will return the Check_Percent otherwise it will return t2 * the Discount_Percent for that employee.

The syntax for returning a value from a function call with our SQLJE Extensions would be:

```
1  float nDDiscount;
2  #sqlje nDDiscount = {VALUES DoubleDiscount(:[oEmpData])};
```

The parser would check to see how many entries are in the oEmpData array, and set up that many parameters for the call. It would then convert the statement to a PL/SQL statement that calls the function, provides the values in the oEmpData array as "IN" parameters and sets up the nDDiscount as the "Out" parameter for the call.

5.8 Using a Dynamic Host Expression for SQL Statement Where Nothing is Known

A new command has been added to the SQLJE syntax called Execute that allows the programmer to pass a Meta Bind Expression

```
1  String sSQL = "UPDATE P SET pname = ? WHERE p# = ?"
2  Object pData = new Object[2];
3  pData[0] = "Marty";
```

```
4   pData[1] = “P1”;
5   #sqlje {EXECUTE :{sSQL} APPLYING :[pData]};
```

5.9 Acquiring Metadata with the SQLJE Describe command

One missing ability of the standard SQLJ instruction set is the absence of a way to easily retrieve metadata such as column name, column type and column length. To gain access to this information required the programmer to include direct JDBC calls in Java to load this data.

This ability was included in our SQLJE by adding a new **Describe** command in association with a Describe class included in a separate JAR file.

The basic syntax for using the Describe statement is:

```
1   Describe dbTableData = new Describe();
2   #sqlje {Describe into dbTableData (Select Statement)};
```

The dbTableData instance of the Describe class includes the following;

- nColumns – The number of columns returned by the Select statement
- sError – An error message if the processing of the Select statement caused an exception
- sColumnName[i] – The column name of the ith column. (Columns are “1” based)
- sColumnType[i] – The type of the ith column.
- nColumnLength[i] – The length of the ith column.

- nPrecision[i] – For float type columns, the digits maintained to the right of the decimal for the ith column.
- nIsNullable[i] – 1 indicates that the ith column is nullable, 0 that it is not

The Select Statement included in the Describe command can follow any of the formatting discussed above for an SQLJE Select Statement from being hard-coded in the statement:

```
1   #sqlje {Describe into dbTableData (Select * from Employee_Discount)};
```

to being completely dynamic

```
1   #sqlje {Describe into dbTableData (:{sSelectDiscount})};
```

6. COMPARISON PROGRAMS

To demonstrate the differences between standard SQLJ and the new SQLJE command, four different mini-programs using the S-P-J database (see Appendix A) were coded.

6.1 Two-Level Hierarchy

The first mini-program retrieves the Suppliers from our S-P-J database, and beneath each Supplier lists the Parts for which that the Supplier currently has orders. The Iterator class must be first declared for the SQLJ version, and then referenced within the body of the program.

```
1  /* SQLJ version of TwoLevel.java from jdbc.doc */
2  /* declare named iterator classes */
3  #sql iterator SIterator
4            (String snum, String sname, int status, String city);
5  #sql iterator PIterator
6            (String pnum, String pname, int weight, String color, String city);
7
8  public class TwoLevelsqlj
9  { public static void main(String[]args) throws SQLException
10   {
11     /* connect to Oracle with the thin JDBC driver*/
12     Oracle.connect
13        ("jdbc:oracle:thin:@db11.eng.fau.edu:1521:R11g",
14         "scott", "tiger");
15
16     /* declare iterator objects */
17     SIterator sx;
18     PIterator px;
19
20     /* populate iterator object for s */
```

```
21      #sql sx =
22        {SELECT s# snum, sname, status, city FROM S ORDER BY snum};
23
24      /* for each supplier record */
25      while(sx.next())
26      {
27        /* print the supplier record */
28        System.out.println(sx.snum()+"  "+ sx.sname()+"  "+
29                           sx.status()+"  "+sx.city());
30
31      /* populate iterator object for p */
32      #sql px =
33        {SELECT DISTINCT p.p# pnum, p.pname, p.weight, p.color, p.city
34          FROM P, SPJ
35          WHERE spj.s# = :(sx.snum()) and p.p# = spj.p#
36          ORDER BY pnum};
37
38        /* for each part record supplied by the current supplier */
39        while(px.next())
40        {
41         /* print the part record */
42         System.out.println("    "+ px.pnum()+"  " +
43                            px.pname()+"  "+ px.weight()+
44                            "  " +px.color()+"  "+px.city());
45        }
46        px.close();
47      }
48
49      /* close the SIterator and the connection */
50      sx.close();
51      Oracle.close();
52    }
53  }
```

A comparable program that does the same function using SQLJE could be the following. Note how no Iterator class is required and the syntax more closely follows PL/SQL conventions. Also note that the variables returned to the program are defined as fields in the S and P result-sets instead of methods of the Iterator.

```
public class TwoLevel {
public static void main(String[]args) throws
        SQLException, IOException, ClassNotFoundException {

//  Connect to Oracle
   Oracle.connect("jdbc:oracle:thin:@db11.eng.fau.edu:1521:R11g","lbradle6", "oracle");



   #sqlje {for s (String snum, String sname, int status, String city) in
      (select s#, sName, status, city from s order by s#)};
   {
      String sNum = s.snum;
      System.out.println(s.snum+ "  " + s.sname + "  " + s.status + " " + s.city);

      #sqlje {for p (String pnum, String pname, String color, int weight, String city) in
         (select distinct p.p#, pname, color, weight, city from spj, p
          where spj.p# = p.p# and spj.s# = :sNum order by p.p#)};
      {
         System.out.println("   "+p.pnum+ "  " + p.pname + "  " + p.color + "   " +
                        p.weight + "  " + p.city);

      }
   }
   Oracle.close();
  }
}
```

6.2 General Retrieval Utility

The next mini-program allows a user to enter a table name a field name. The program then retrieves metadata for that table and field, and if all checks out asks the user for a value for the field. Once the user enters the field value, the program generates a Select statement to retrieve all records where the entered field matches the value of that field in the database. Here is the SQLJ version of the program:

```
/* SQLJ version of gret.java from jdbc3.doc */
public class gretsqlj
{ public static void main(String[]args) throws
```

```
4       SQLException, IOException
5      {
6       /* create a BufferedReader object for standard input (this allows
7       us to read from standard input a line at a time) */
8       BufferedReader input =
9         new BufferedReader(new InputStreamReader(System.in));
10
11      /* connect to Oracle with the thin JDBC driver*/
12      /* have to get the DefaultContext object that wraps the Connection
13      object, so that we can obtain the Connection object, which is
14      required by the JDBC metadata stuff, from it. */
15      DefaultContext dc =
16      Oracle.connect
17          ("jdbc:oracle:thin:@db11.eng.fau.edu:1521:R11g",
18           "scott", "tiger");
19      Connection conn = dc.getConnection();
20
21      /* input from user the table name and field name for the query */
22      System.out.println
23     ("Please enter table name and field name for query in upper case");
24      String line = input.readLine();
25      StringTokenizer tk = new StringTokenizer(line);
26      String tablename = tk.nextToken();
27      String fieldname = tk.nextToken();
28
29      /* Get field info for specified field in specified table.  We do
30      this so that we can check that the table has such a field, and so
31      that we can get the type name of that field. */
32      DatabaseMetaData d = conn.getMetaData();
33      Result-set rm = d.getColumns(null, null, tablename, fieldname);
34
35      /* no records in result-set means specified field is not in
36      specified table */
37      if (!rm.next())
38      {
39        System.err.println
40        ("bad table name or bad field name, retrieve terminated");
41        System.exit(1);
42      }
43
44      /* get type name of field from the result-set, for later use */
```

```
45      String typename = rm.getString(6);
46
47      /* input field value for query from user */
48      System.out.println("please enter field value for query");
49      line = input.readLine();
50      tk = new StringTokenizer(line);
51      String fieldvalue = tk.nextToken();
52
53      /* Set parameter in query to the input field value.  We assume
54      field is either varchar, float, or integer.  Note that the Oracle
55      type name for integer is NUMBER */
56      if (typename.equals("VARCHAR2")) fieldvalue = "'"+fieldvalue+"'";
57
58      /* execute query into Result-setIterator object */
59      /* SQLJ dynamic SQL is based on meta bind expressions, which can
60      appear in only certain clauses of certain SQL statements.  Please
61      see sqldynamicsql.doc on Blackboard for more details. */
62      Result-setIterator ri;
63      #sql ri =
64        {select * from :{tablename} where :{fieldname} = :{fieldvalue}};
65
66      /* get Result-set from Result-setIterator object */
67      Result-set r = ri.getResult-set();
68
69      /* get number of fields in field list of query */
70      Result-setMetaData rmd = r.getMetaData();
71      int nfields = rmd.getColumnCount();
72
73      /* print output from query */
74      while (r.next())
75      {
76
77        /* note that field numbers in result-set record
78        start with 1, not 0 */
79        for (int i = 1; i <= nfields; i++)
80        {
81          /* since we are only getting the fields to print them,
82          we can get them as strings */
83          System.out.print("   " + r.getString(i));
84        }
85        System.out.println();
```

```
86          System.out.println();
87        }
88
89        /* close stuff */
90        r.close();
91        rm.close();
92        ri.close();
93        Oracle.close();
94        System.out.println("retrieve complete");
95      }
96    }
```

Here is the same mini-program using SQLJE. Note the absence of the iterator class since these have been replaced by the dbTab class in the For statement. The Describe class and command are used to retrieve the metadata for the table, and can be called without adding a default context to the Oracle connection and using JDBC commands. Also note that the SQLJ For->Select can accept a Standard Host Expression in a Where clause. The SQLJ version can only accept Meta Bind Variables there requiring the program to place single quote marks around the field.

```
1   public class GRET
2   { public static void main(String[]args) throws
3       SQLException, IOException, ClassNotFoundException
4     {
5       /* create a BufferedReader object for standard input (this allows
6       us to read from standard input a line at a time) */
7       BufferedReader input =
8         new BufferedReader(new InputStreamReader(System.in));
9
10      /* connect to Oracle with the thin JDBC driver*/
          Oracle.connect("jdbc:oracle:thin:@db11.eng.fau.edu:1521:R11g","lbradle6",
11  "oracle");
12
13      /* input from user the table name and field name for the query */
14      System.out.println
15     ("Please enter table name and field name for query in upper case");
```

```
16      String line = input.readLine();
17      StringTokenizer tk = new StringTokenizer(line);
18      String tablename = tk.nextToken();
19      String fieldname = tk.nextToken();
20
21      Describe dbTable = new Describe();
22      String sSQL = "Select * from " + tablename;
23      #sqlje {DESCRIBE INTO dbTable (:{sSQL})};
24      if (dbTable.nColumns == 0){
25          System.err.println
26          ("bad table name or bad field name, retrieve terminated");
27          System.exit(1);
28      }
29      Boolean bFound = false;
30      for (int i=1;i<=dbTable.nColumns;i++){
31         if (fieldname.compareTo(dbTable.sColumnName[i])== 0) bFound = true;
32      }
33      if (bFound == false)  {
34                System.err.println
35                ("bad field name, retrieve terminated");
36                System.exit(1);
37           }
38      /* input field value for query from user */
39      System.out.println("please enter field value for query");
40      line = input.readLine();
41      tk = new StringTokenizer(line);
42      String fieldvalue = tk.nextToken();
43
44      #sqlje {for dbTab in (select * from :{tablename} where :{fieldname} = :fieldvalue)};
45        {
46          for (int i=1;i<=dbTab.nColumns;i++)   System.out.print("  " + dbTab.Column(i));
47          System.out.println();
48        }
49        System.out.println();
50      /* close stuff */
51      Oracle.close();
52      System.out.println("retrieve complete");
53    }
54  }
```

6.3 General Load Utility

The General Load Utility mini-program asks the user for a table into which s/he would like to insert records and the number of records to insert. The program then goes and retrieves metadata for that table and informs the user of the fields and types that need to be entered. The user enters the fields and the program verifies the input and inserts the new records into the table.

Because the General Load Utility can be used to insert values into any table, the number of columns and type of columns are not know at compile time. Consequently, the current version of SQLJ cannot be used to code this fairly simple program. The comparison example program then is one written using JDBC.

```
1  public class LoadUtil {
2  public static void main(String[]args) throws
3          SQLException, IOException, ClassNotFoundException {
4     BufferedReader ioInput =
5      new BufferedReader(new InputStreamReader(System.in));
6
7     /* load jdbc drivers and
8        connect to Oracle with the thin JDBC driver*/
9     Class.forName("oracle.jdbc.OracleDriver");
10    Connection dbConnection =
11     DriverManager.getConnection
12       ("jdbc:oracle:thin:@db11.eng.fau.edu:1521:R11g",
13        "scott", "tiger");
14
15    /* input from user the table name into which they want to insert */
16    System.out.println
17    ("Please enter table name into which you would like to insert new records a space and
18    String sLine = ioInput.readLine();
19    StringTokenizer tk = new StringTokenizer(sLine);
20    String sTable = tk.nextToken().toUpperCase();
21    int nRecords = 0;
22    try
23       { nRecords = Integer.parseInt(tk.nextToken());}
```

```
24      catch(Exception e)
25          {System.out.println(sTable + " Invalid Input " + e.toString() + " Insert Aborted");
26          System.exit(1);}
27
28      if (nRecords == 0){
29          System.out.println(sTable + " Number of Records Can't Be Zero");
30          System.exit(1);}
31
32      //Go see if this is a good table and if it is, get the column names and definitions and store
33      DatabaseMetaData dbTableColumns = dbConnection.getMetaData();
34      Result-set dbTableResults = dbTableColumns.getColumns(null, null, sTable,null);
35      ArrayList<TableDefinition> arColumns = new ArrayList<TableDefinition>();
36      while (dbTableResults.next())
37      {
38              TableDefinition oTable;
39              oTable = new
40              arColumns.add(oTable);
41              System.out.println(dbTableResults.getString(4)+" " +
42          }
43          if (arColumns.size() == 0){
44              System.out.println(sTable + " Table Not Found - Insert Aborted");
45              System.exit(1);
46          }
47  //   Create the Insert Statement
48          String sInsert = "";
49          String sValues = "";
50          String sEnter = "";
51          for (TableDefinition ele:arColumns) {
52              if (sInsert != "") sInsert = sInsert + "," ;
53              sInsert = sInsert + ele.GetName();
54              if (sValues != "") sValues =  sValues + "," ;
55              sValues = sValues + "?";
56              sEnter = sEnter + ele.GetName()+ " ";
57              }
58          sInsert = "INSERT INTO " + sTable + "(" + sInsert + ") VALUES (" + sValues + ")";
59          System.out.println(sInsert);
60
61  //  Print out what is in the table now
62          PrintTable(dbConnection, sTable);
63  // Prepare The Statement
64          PreparedStatement dbInsert = dbConnection.prepareStatement(sInsert);
```

```
65
66  // Let's go insert the records
67          for (int i = 1; i <= nRecords; i++){
68              boolean bContinue = true;
69              System.out.println("Enter Record " + i + " of " + nRecords + " " + sEnter);
70              sLine = ioInput.readLine();
71              tk = new StringTokenizer(sLine);
72              int nCol = 0;
73              // Look through the columns and update the entries based on the column type
74              for (TableDefinition ele:arColumns) {
75                  nCol = nCol + 1;
76                  try{
77                      String sFieldValue = tk.nextToken();
78                      if (ele.GetType().equals("VARCHAR2"))
79                      else if (ele.GetType().equals("FLOAT"))
80                      else dbInsert.setInt(nCol,Integer.parseInt(sFieldValue));
81                  }
82                  catch (Exception e){
83                      System.out.println("!!!Input Incorrect - This Record Skipped!!! " +
84                      bContinue= false;
85                  }
86              }
87              if (bContinue){
88                  try{
89                      int nInsertOK = dbInsert.executeUpdate();
90                      if (nInsertOK == 1) System.out.println("+++Insert Successful+++");
91                          else System.out.println("!!!Insert Failed!!!");
92                  }
93                  catch (Exception e){
94                      System.out.println("!!!Insert Failed!!! " + e.toString() );
95                  }
96              }
97          }
98          System.out.println("End of Program");
99          System.exit(0);
100     }
101 // Class to hold a structure for table definitions
102     public class TableDefinition
103     {
104       private String sColumnName;
105       private String sColumnType;
```

```
106         public  TableDefinition ( String psColumnName, String psColumnType)
107         {
108         sColumnName = psColumnName;
109         sColumnType = psColumnType;
110           }
111          public String GetName(){
112             return sColumnName;
113         }
114          public String GetType(){
115              return sColumnType;
116          }
117       }
118   }
```

The SQLJE version of the General Load Utility makes used of the Describe class and command to acquire the metadata information needed for the table. Once that is available, the program just needs to iterate through the list of columns in the table and create the information for the user. The user can then enter the information which is split into the oData array, and the single command

```
1   #sqlje {INSERT INTO :{sTable} (:{sColumns}) Values (:[oData])};
```

is used to generate the code necessary to create the actual insert statement and set the parameter entries. Here is the SQLJE version of the program:

```
1   public class GeneralLoad {
2       public static void main(String[] args)throws IOException,
3                 SQLException,ClassNotFoundException {
4           Oracle.connect("jdbc:oracle:thin:@db11.eng.fau.edu:1521:R11g","scott", "tiger");
5           InputStreamReader isr = new InputStreamReader(System.in);
6             BufferedReader br = new BufferedReader(isr);
7           System.out.println("Into What Table Would You Like To Insert?");
8           String sTable = br.readLine();
9           System.out.println("How Many Records do you Want To Insert?");
10           String sCount = br.readLine();
```

```
11          int nCount = 0;
12                try {
13                  nCount = Integer.valueOf(sCount).intValue();
14                } catch (NumberFormatException nfe) {
15                  System.out.println("Incorrect Number of Records!");
16                  System.exit(1);
17                }
18
19
20          Describe dbTable = new Describe();
21          String sSelect = "Select * from " + sTable;
22          String sColumns = "";
23          #sqlje {DESCRIBE INTO dbTable (:{sSelect})};
24
25          System.out.println("Enter data separated by commas");
26          for (int i = 1;i <= dbTable.nColumns;i++){
               System.out.print(dbTable.sColumnName[i] + "(" + dbTable.sColumnType[i] + ")
27    ");
28             if (sColumns.compareTo("")==0) sColumns = dbTable.sColumnName[i];
29             else sColumns = sColumns + "," + dbTable.sColumnName[i];
30          }
31          Object [] oData;
32          for (int i = 0;i<nCount;i++){
33             String sLine = br.readLine();
34             oData = sLine.split(",");
35             if (oData.length != dbTable.nColumns){
36                System.out.println ("Incorrect number of data fields " + sLine);
37             }
38             else {
39                #sqlje {INSERT INTO :{sTable} (:{sColumns}) Values (:[oData])};
40             }
41          }
42       }
43    }
```

6.4 Process Any Statement

The final mini-program we are going to compare is one where the use can enter any standard SQL statement. The program then checks to see whether or not the

statement is a Select statement (requiring a result-set class) or not (an immediate command). Based on this information, the program generates the correct call to the Oracle RDMS using SQLJE, and executes the command.

This is another program that cannot be coded in the current version of SQLJ since there is no mechanism to pass a completely unknown command through to SQLJ and have it execute it.

Here is an example program written in JDBC that implements the Any Statement min-program:

```
1  public class AnyJDBC {
2
3
4     public static void main(String[] args) throws SQLException, IOException,
   ClassNotFoundException {
5        /* create a BufferedReader object for standard input
6        (this allows us to read from standard input
7        a line at a time) */
8         BufferedReader ioInput = new BufferedReader(new InputStreamReader(System.in));
9
10       /* load jdbc drivers and
11       connect to Oracle with the thin JDBC driver*/
12        Class.forName("oracle.jdbc.OracleDriver");
13        Connection dbConnection =
14             DriverManager.getConnection("jdbc:oracle:thin:@db11.eng.fau.edu:1521:R11g", "scott", "tiger");
15       /* input from user the table name into which they want to insert */
16       System.out.println("Please enter the SQL Statement you would like to process");
17       String sSQL = ioInput.readLine();
18       if (sSQL.toUpperCase().substring(0, 6).compareTo("SELECT") == 0) {
19          try {
20             Statement dbTableStatement = dbConnection.createStatement();
21             ResultSet dbTableResult = dbTableStatement.executeQuery(sSQL);
22             ResultSetMetaData dbTableDef = dbTableResult.getMetaData();
23             int nFields = dbTableDef.getColumnCount();
24             for (int i = 1; i <= nFields; i++) {
25                System.out.print("  " + dbTableDef.getColumnName(i));
```

```
26                }
27                System.out.println();
28                //  Print out the data for the table
29                while (dbTableResult.next()) {
30                   for (int i = 1; i <= nFields; i++) {
31                      System.out.print("  " + dbTableResult.getString(i));
32                   }
33                   System.out.println();
34                }
35             } catch (SQLException ex) {
36                System.out.println("Error in SQL statement " + ex.getMessage());
37             }
38          }
39       else    {
40          try {
41             PreparedStatement dbInsert = dbConnection.prepareStatement(sSQL);
42             int nRowsAffected = dbInsert.executeUpdate();
43             System.out.println("Command completed sucessfully");
44          } catch (SQLException ex) {
45             System.out.println("Error in SQL statement " + ex.getMessage());
46          }
47       }
48          System.out.println("End of Program");
49          System.exit(0);
50
51       }
52    }
```

Here is the same program written using our new SQLJE. The program is very similar except the SQLJE version uses our new Describe command to retrieve the metadata from the column names.

```
1     public class AnyStatement {
2        public static void main(String[]args) throws
3          SQLException, IOException, ClassNotFoundException
4         {
5           BufferedReader input =
6            new BufferedReader(new InputStreamReader(System.in));
7
```

```
8           Oracle.connect("jdbc:oracle:thin:@db11.eng.fau.edu:1521:R11g","scott", "tiger");
9
10
11        /* input from user the SQL Statement to process*/
12        System.out.println
13       ("Please enter the SQL Statement you would like to process");
14        String sSQL = input.readLine();
15        if (sSQL.toUpperCase().substring(0,6).compareTo("SELECT")==0){
16           try {
17        Describe dbTable = new Describe();
18        #sqlje {DESCRIBE INTO dbTable (:{sSelect})};
19        for (int i = 1;i <= dbTable.nColumns;i++){
20           System.out.print(dbTable.sColumnName[i]);}
21              #sqlje {for dbTable in (:{sSQL})};
22              {for (int i = 1;i<=dbTable.nColumns;i++){
23                 System.out.print(dbTable.Column(i)+" ");
24              }
25              System.out.println();
26              }
27           }
28           catch (SQLException ex){
29              System.out.println("Error in SQL statement " + ex.getMessage());
30           }
31        }
32        else{
33           try {
34              #sqlje {execute :{sSQL}};
35              System.out.println("Command completed sucessfully");
36           }
37           catch (SQLException ex){
38              System.out.println("Error in SQL statement " + ex.getMessage());
39           }
40          }
41        }
42  }
```

7. SUMMARY AND CONCLUSION

This thesis documents a development project that implemented of an extended SQLJ embedded database interface with new commands and structures. In Chapter Two we briefly reviewed the history of interfaces between relational database management systems and systems development environments and introduced improvements suggested in Hurdato's Summer 2012 thesis. In Chapter Three we defined the overall project and specified the four major changes we planned on adding to the SQLJ syntax: Dynamic Host Expressions, Select For Statements, a Describe facility and a new command, Execute, that provides the ability to process a SQL statement unknown at compile time. Chapter four detailed the design, development and testing that resulted in a pre-compiler to SQLJ implementing the changes. Chapter Five documented how the new features can be used with examples for each command type. Finally in Chapter Six we compared source code written using SQLJ and JDBC with the code that performed the same functions with the SQLJE extensions.

The main intent of making the additions to SQLJ was to eliminate the need to use JDBC with SQLJ in the same program while improving the clarity and brevity of the source code. As shown in the sample mini-programs these goals were achieved with the use of Dynamic Host Expressions, the Describe command and the Execute command.

We also wanted to replace the result-set iterator used by SQLJ with something that was closer to the way other embedded database interfaces traversed a result-set. This

was accomplished using the For->Select command. In addition, we changed the return type of static field names so that they were no longer methods but instead fields from the class, and we added the ability to reference result-set columns for the dynamic case with indexes that match the column names.

> Although these changes resulted in the accomplishment of the goals of the project, the pre-compiler that adds these new features is fairly straightforward and was written over a four month period by one developer. This implies that adding these features directly to the real SQLJ pre-compiler or even directly to the Java language itself may not be that large a task, and something that we would recommend to the Oracle Corporation to review.

APPENDIX A – S-P-J DATABASE

The S-P-J Database is a set of tables used in C.J. Date's "An Introduction to Database Systems" textbook that is used for many of the examples in this thesis. The database represents a classic supplier, project, part relationship and is made up of four tables as follows:

S – Table that holds information about a specific supplier:

s# - The unique identifier for a Supplier (Varchar (5))

sname – The name of the supplier (Varchar(20))

status – The status of the supplier (integer)

city – The city where the supplier is based (Varchar(10)

P – Table that holds information about a specific part:

p# - The unique identifier for a Part (Varchar(5))

pname – The name of the Part (Varchar (20))

color – The color of the Part (Varchar (10))

weight – The weight of the Part (integer)

city – The city where the Part is located (Varchar(10))

J – Table the holds information about a specific Job/Project

j# - The unique identier for a Job (Varchar(5))

jname – The name of the Job (Varchar(20))

city – The city where the Job is being worked (Varchar(10))

SPJ – Table that holds all orders for Parts by Supplier and Job

s# - Key to the Supplier (S) table (Varchar(5))

j# - Key to the Job/Project (J) table (Varchar(5))

p# - Pointer to the Parts (P) table (Varchar(5))

qty – The number of Parts supplied to this Job by this Supplier

APPENDIX B - SOURCE CODE

```
1  //SQLJE.JAVA – Main Program SQLJE Pre-Compiler
2  // Program that adds new features to the SQLJ Oracle DMS Commands
3  // Includes the addition of Dynamic Host Expressions
4  // Select For Processing without Iterators - Both static and dynumic
5  // Support for a Describe Command
6  // Suppoer for an Execute Command
7  public class SQLJE {
8     static Pattern compiledRegex;
9     static Matcher regexMatcher;
10    static BufferedWriter writer;
11    static int mnClassCount = 0; // Holds the current number of select command classes.
12    static int mnCommandCount = -1; // Holds the current number of commands.  Used for tag name
13    static String msSelectClass = ""; // Holds the clases created for the Select statement
14    static String msParams = ""; // Holds the param entries needed for "?"
15    static String msTagName = "";
16    static String msContext = null;
17    static List msLiterals = new ArrayList(); // Holds comments and literals removed the each line
   and the put back by AppendLine
18
19    public SQLJE() {
20    }
21
22    public static void main(String[] args) {
23       SQLJE parseSQLJ = new SQLJE();
24
25       String sInFile = args[0] + ".sqlje";
26       String sOutFile = args[1] + ".sqlj";
27
28
29       File inFile = new File(sInFile); // File to read from.
30       File outFile = new File(sOutFile); // File to write to
31       try {
32          parseFile(inFile, outFile);
33
34       } catch (IOException e) {
```

```
35              System.err.println(e);
36              System.exit(1);
37          }
38      }
39
40      public static void parseFile(File fromFile, File toFile) throws IOException {
41          //  This is the main loop for the SQLJE pre-compile.  The method reads
42          //  each line of input, and replaces literals with special strings
43          //  so they are not processed.  It then checkes for comments and passes those
44          //  directly through to the output file.  Finally, it checks to see if the
45          //  sqlje# tag is present, and if it is pulls together the entire sqlje statement
46          //  and passes it for processing.
47          BufferedReader reader = new BufferedReader(new FileReader(fromFile));
48          writer = new BufferedWriter(new FileWriter(toFile));
49
50          String sLine = null;    //  The line being scanned
51          String sCommand = null;
52          String sComments = "(?:/\\*(?:[^*]|(?:\\*+[^*/]))*\\*+/)|(?://.*)";   //Regex for a comment
53          String sDoubleLit = "\"([^\"\\\\]|\\\\.)*\"";       //Regex for a literal in double quotes
54          String sSingleLit = "\'([^\'\\\\]|\\\\.)*\'";       //Regex for a literal in single quotes
55          boolean bGotOne = false;                            //Boolean to mark that the line has an sqlje command
56          boolean bInComment = false;                         //Boolean to indicate currently in a comment
57          int nComment;
58
59          while ((sLine = reader.readLine()) != null) {
60              // If we are in a comment, copy over until we get the end comment sentinel
61              if (bInComment == true) {
62                  nComment = sLine.indexOf("*/");
63                  if (nComment > -1) { // We're going to split the line so that the comment has ended and the command is on a new lin
64                      AppendLine(sLine.substring(0, nComment + 2));
65                      sLine = sLine.substring(nComment + 2);
66                      bInComment = false; // we need to process what is to the right of the
67                  } else { // we just need to write this line out and move on
68                      AppendLine(sLine);
69                  }
70              }
71              if (bInComment == false) {
72                  //Step one - remove all quote literals from the line just read in
73                  sLine = RemoveLiterals(sLine, sDoubleLit);
74                  sLine = RemoveLiterals(sLine, sSingleLit);
```

```
75              // Step two - Remove any comments from the line just read in
76              // First - see if there are end of line comments (//)
77              nComment = sLine.indexOf("//");
78              if (nComment > -1) {
79                 msLiterals.add(sLine.substring(nComment));
80                 sLine = sLine.substring(0, nComment) + "!%&" + msLiterals.size() + "&%!";
81              }
82
83              // Remove any complete comments (/* to */) in the line just read
84              sLine = RemoveLiterals(sLine, sComments);
85
86              // See if there is just a start of a comment.  If there is set the InComments switch
87              nComment = sLine.indexOf("/*");
88              if (nComment > -1) {
89                 msLiterals.add(sLine.substring(nComment));
90                 sLine = sLine.substring(0, nComment) + "!%&" + msLiterals.size() + "&%!";
91                 bInComment = true;
92              }
93
94
95              if (sLine.toUpperCase().indexOf("#SQLJE") > -1) {
96                 bGotOne = true;
97                 sCommand = sLine;
98                 AppendLine("//************************************");
99
100             } else {
101                if (bGotOne == true)
102                   sCommand = sCommand + " " + sLine;
103             }
104
105             if (bGotOne == true)
106                sLine = "//" + sLine;
107             AppendLine(sLine);
108             if (bGotOne == true) {
109                if (sLine.indexOf(";") > 0) {
110                   DecodeCommand(sCommand, writer);
111                   AppendLine("//************************************");
112                   bGotOne = false;
113                   msLiterals.clear();
114                }
115             }
```

```
116             }
117         }
118         msLiterals.clear();
119 
120         String SelClass[] = msSelectClass.split("\n");
121         for (int i = 0; i < SelClass.length; i++) {
122             AppendLine(SelClass[i]);
123         }
124 
125         reader.close();
126         writer.close();
127     }
128 
129     public static void DecodeCommand(String psCommand, BufferedWriter writer) throws IOException, StringIndexOutOfBoundsException {
130         //  This method determines which type of command is within the SQLJE statements.
131         //  It also ensures correct formating of the statement with curly brackets and semicolon
132         //  and determines if an alternate database context is to be used instead of the normal SQLJ connection
133         try {
134             msContext = null;
135             String sInCommand = psCommand.trim();
136 
137 
138             int nPosition = sInCommand.indexOf("{");
139             if (nPosition == -1) {
140                 AppendLine("//Missing Starting Curly Bracket - Statement Not Processed\n");
141                 return;
142             }
143 
144             //Check for alternate database context
145             int nLBracket = sInCommand.indexOf("[");
146             if ((nLBracket > -1) && (nLBracket < nPosition)){
147                 int nBracket = sInCommand.indexOf("]");
148                 if (nBracket > -1) {
149                     msContext = sInCommand.substring(nLBracket + 1, nBracket);
150                     sInCommand = sInCommand.substring(0, nLBracket) + sInCommand.substring(nBracket + 1);
151                     nPosition = sInCommand.indexOf("{");
152                 } else {
153                     AppendLine("//Cannot decypher command\n");
154                     return;
155                 }
```

```
156             }
157
158             // Remove the #sqlje and (if included) database context from the command
159             String sRight = sInCommand.substring(nPosition + 1).trim();
160             nPosition = sRight.indexOf(" ");
161             if (nPosition == -1) {
162                 AppendLine("//Cannot decypher command\n");
163                 return;
164             }
165             String sCommand = sRight.substring(0, nPosition);
166
167             //Get rid of any spaces in Host Variables so they won't be split later
168             String sHostVariable = ":\\{.*?\\}";
169             String sTempSQL = "";
170             compiledRegex = Pattern.compile(sHostVariable);
171             try {
172                 Boolean bContinue = true;
173                 while (bContinue == true) {
174                     regexMatcher = compiledRegex.matcher(sRight);
175                     if (regexMatcher.find() == true) {
176                         String sHost = regexMatcher.group();
177                         sHost = sHost.replaceAll(" ", "");
178                         //              sRight = regexMatcher.replaceFirst("%&!\\{" + sHost + "\\}");
179                         int nEnd = regexMatcher.end();
180                         sRight = regexMatcher.replaceFirst(sHost);
181                         sTempSQL = sTempSQL + " " + sRight.substring(0, nEnd);
182                         sRight = sRight.substring(nEnd);
183
184                     } else
185                         bContinue = false;
186                 }
187             } catch (IllegalStateException ex) {
188                 System.out.println(ex.getMessage());
189             }
190             sTempSQL = sTempSQL + " " + sRight;
191             sRight = sTempSQL;
192
193              String sUpper = sCommand.toUpperCase();
194             if (sCommand.toUpperCase().compareTo("INSERT") == 0)
195                 DecodeInsert(sRight, writer);
196             else if (sCommand.toUpperCase().compareTo("FOR") == 0)
```

```
197                 DecodeSelectFor(sRight, writer);
198             else if (sCommand.toUpperCase().compareTo("UPDATE") == 0)
199                 DecodeUpdateDelete(sRight, writer);
200             else if (sCommand.toUpperCase().compareTo("DELETE") == 0)
201                 DecodeUpdateDelete(sRight, writer);
202             else if (sCommand.toUpperCase().compareTo("CALL") == 0)
203                 DecodeProcedure(sRight, writer);
204             else if (sCommand.toUpperCase().compareTo("EXECUTE") == 0)
205                 DecodeUpdateDelete(sRight, writer);
206             else if (sCommand.toUpperCase().compareTo("DESCRIBE") == 0)
207                 DecodeDescribe(sRight, writer);
208             else if (sCommand.toUpperCase().indexOf("VALUES") == 0)
209                 DecodeFunction(sInCommand, writer);
210             else {
211                 AppendLine("//Command incorrect or not implemented\n");
212                 return;
213             }
214         } catch (IOException ex) {
215             System.out.println("Error in Command" + psCommand);
216         } catch (StringIndexOutOfBoundsException oob) {
217             System.out.println("Error in Command" + psCommand);
218 
219         }
220     }
221 
    private static void DecodeInsert(String psCommand, BufferedWriter writer) throws IOException
222 {
223         //OK - we have an insert statement
224         // declare the variables needed to access Oracle
225         AppendLine("{");
226         SetOracleContext();
227 
        //  First see if there is a VALUES clause and split the command.  While I'm at it, find out if
228 there is a SELECT
229         // which means no VALUES, and if there is a APPLYING
230         int nValues = psCommand.toUpperCase().indexOf("VALUES");
231         int nSelect = psCommand.toUpperCase().indexOf("SELECT");
232         int nUsing = psCommand.toUpperCase().indexOf("APPLYING");
233         if ((nValues == 0) && (nSelect == 0)) {
            AppendLine("//Command incorrect or not implemented no VALUES or SELECT clause " +
234 psCommand + "\n");
235             return;
236         }
```

```
237         String sLeft;
238         String sValue;
239         if (nValues >= 0) { //Splitting on VALUES
240            sLeft = psCommand.substring(0, nValues);
241            sValue = psCommand.substring(nValues + 6).trim();
242         } else { //Splitting on SELECT
243            sLeft = psCommand.substring(0, nSelect);
244            sValue = psCommand.substring(nSelect).trim();
245         }
246
247
248         //Check for the INTO and then get the table name.  This could be a host variable.
249         int nPosition;
250         nPosition = sLeft.toUpperCase().indexOf("INTO");
251         if (nPosition == 0) {
              AppendLine("//Command incorrect or not implemented - Missing INTO " + psCommand +
252    "n");
253            return;
254         }
255         sLeft = sLeft.substring(nPosition + 4);
256         String sVariables = "";
257         String sTable = "";
258         compiledRegex = Pattern.compile("\\(.*\\)");
259         try {
260            regexMatcher = compiledRegex.matcher(sLeft);
261            if (regexMatcher.find()) {
262               sVariables = regexMatcher.group();
263               sLeft = regexMatcher.replaceFirst("").trim();
264            }
265         } catch (IllegalStateException ex) {
266            System.out.println(ex.getMessage());
267            return;
268         }
269         sTable = sLeft.trim();
270
271         //  See what we have as the table name.  A Host Variable or a table name
272
273         DetermineTableorProcedure(sTable, "INSERT INTO ");
274
275         //  OK - We Have "INSERT INTO TABLENAME".  Next, is there a field list??
276         if (sVariables.indexOf("(") == 0) {
277            sVariables = sVariables.substring(1, sVariables.length() - 1);
```

```
278             String[] sEachVariable = sVariables.split(",");
279             ProcessInsertParen(sEachVariable, "V");
280          }
281          //This is a VALUES statement - process it and return to the main routine
282          if (nValues >= 0) {
283             AppendLine("     __fau_ssql = __fau_ssql + \" VALUES \";");
284
285             //See if we have a APPLYING - If yes, grab the Dynamic Host Expression there
286             String[] sGetUsing = sValue.split("APPLYING");
287             String sUsing = "";
288             if (sGetUsing.length == 2) {
289                sUsing = sGetUsing[1];
290                sValue = sGetUsing[0];
291             }
292
293             //We have the left side ready to go.  See if the right side has a Dynamic Host Expression
294             //  Get just what is inbetween the parenthesis
295             compiledRegex = Pattern.compile("\\(.*\\)");
296             try {
297                regexMatcher = compiledRegex.matcher(sValue);
298                if (regexMatcher.find()) {
299                   sValue = regexMatcher.group();
300                   sValue = sValue.substring(1, sValue.length() - 1);
301                }
302             } catch (IllegalStateException ex) {
303                System.out.println(ex.getMessage());
304                AppendLine("Values Clause Poorly Formated");
305                return;
306             }
307             String[] sEachValue = sValue.split(",");
308             msParams = "";
309             ProcessInsertParen(sEachValue, "?");
310             SetParamInsertProc(sEachValue);
311
312
313             //  If we have a Using, stick this at the end to set any other parameters
314             if (sUsing.compareTo("") != 0) {
315                sUsing = sUsing.replace(";", "");
316                sUsing = sUsing.replace("}", "").trim();
317                SetUsing(sUsing);
318                if (msParams.trim().compareTo("") != 0) {
```

```
319                 String SelParam[] = msParams.split("\n");
320                 for (int i = 0; i < SelParam.length; i++) {
321                     AppendLine(SelParam[i]);
322                 }
323             }
324         }
325         AppendLine("    __fau_ec.oracleExecuteBatchableUpdate();");
326         AppendLine("}");
327         return;
328       }
329       //This is a SELECT statement instead of Values.  Run it through the ClauseReplace
330       if (nSelect >= 0) {
331         if (nUsing != 0) {
332             String[] sGetUsing = sValue.split("APPLYING");
333             String sUsing = "";
334             if (sGetUsing.length == 2) {
335                 sUsing = sGetUsing[1];
336                 sValue = sGetUsing[0];
337             }
338             sValue = sValue.replace(";", "");
339             AppendLine("__fau_ssql = __fau_ssql +  = " + ClauseReplace(sValue) + ";");
340             sUsing = sUsing.replace(";", "");
341             sUsing = sUsing.replace("}", "").trim();
342             SetUsing(sUsing);
343         } else {
344             AppendLine("__fau_ssql = __fau_ssql +  =  " + ClauseReplace(sValue) + ";");
345         }
346         mnCommandCount = mnCommandCount + 1;
347         AppendLine("        String __fau_tag = \"" + mnCommandCount + msTagName + ":\" +
__fau_ssql;");
348         AppendLine("                                                                    __fau_st          =
__fau_ec.prepareOracleBatchableStatement(__fau_cc,__fau_tag,__fau_ssql);");
349
350         if (msParams != "") {
351             String SelParam[] = msParams.split("\n");
352             for (int i = 0; i < SelParam.length; i++) {
353                 AppendLine(SelParam[i]);
354             }
355         }
356         AppendLine("    __fau_ec.oracleExecuteBatchableUpdate();");
357         return;
358       }
```

```
359         // If we made it here (although we should not have, we didn't have either a Values or a Select.
            AppendLine("//Command incorrect or not implemented no VALUES or SELECT clause " +
360 psCommand + "\n");
361         return;
362
363       }
364
          static void ProcessInsertParen(String[] psEachValue, String psQuesorName) throws
365 IOException {
366         // The method accepts an array of strings representing either a variable list or a Values list.
            // The psQuerorName parameter is used to determine if question marks should be placed into
367 the statement
368         // or if the actual variable name should be used.
369
370         boolean nFirst = true;
371         String sNext = "";
372
373         // Is this a single host variable for the variable list.  If yes, process and get out
374         sNext = psEachValue[0].trim();
            if ((psEachValue.length == 1) && (psQuesorName.compareTo("V") == 0) &&
375 (psEachValue[0].trim().substring(0, 2).compareTo(":{")) == 0) {
376           sNext = psEachValue[0].substring(2);
377           sNext = sNext.replace("}", "");
378           AppendLine("     __fau_ssql = __fau_ssql + \"(\" + " + sNext + " + \")\";");
379           return;
380         }
381
382         //  For each string in the variable list, determine the type and add it to the overall command
383         int nParam = 0;
384         if (psQuesorName.compareTo("F") == 0) nParam = 1;
385        for (String sOneValue : psEachValue) {
386           sOneValue = sOneValue.trim();
387           if (sOneValue.substring(0, 1).compareTo(":") == 0) // This is a variable
388           {
389             if (sOneValue.substring(1, 2).compareTo("[") == 0) // And it is an Array
390             {
391               String sArray = sOneValue.substring(2, sOneValue.length() - 1);
392               sArray = sArray.replace("]", "");
393               AppendLine("    String sArrayAdd=null;");
394               AppendLine("    for (int __faui = 0;__faui<" + sArray + ".length;__faui++) ");
395               if (psQuesorName.compareTo("?") == 0)
                    AppendLine("          {if (sArrayAdd==null) sArrayAdd = \"?\"; else sArrayAdd =
396 sArrayAdd + \",?\";}");
397               else if (psQuesorName.compareTo("F") == 0)
```

```
                    AppendLine("            {if (sArrayAdd==null) sArrayAdd = \":\" + (__faui+2); else
398 sArrayAdd = sArrayAdd + \",:\" + (__faui+2);}");
399                 else
                    AppendLine("            {if (sArrayAdd==null) sArrayAdd = sArray[__faui]; else
400 sArrayAdd = sArrayAdd + \",\" + sArray[__faui];}");
401                 if (nFirst) {
402                    AppendLine("     __fau_ssql = __fau_ssql + \"(\" + sArrayAdd; ");
403                    nFirst = false;
404                 } else {
405                    AppendLine("     __fau_ssql = __fau_ssql + \",\" + sArrayAdd;");
406                 }
407              } else {
408                 String sArray = sOneValue.substring(1);
409                 if (nFirst) {
410                    if (psQuesorName.compareTo("?") == 0)
411                       AppendLine("     __fau_ssql = __fau_ssql +\"(" + "?" + "\";");
412                    else if (psQuesorName.compareTo("F") == 0){
413                       nParam = nParam + 1;
414                       AppendLine("     __fau_ssql = __fau_ssql +\"(" + ":" + nParam + "\";");
415                    }
416                    else
417                       AppendLine("     __fau_ssql = __fau_ssql + \"(\" + " + sArray + ";");
418                    nFirst = false;
419                 } else {
420                    if (psQuesorName.compareTo("?") == 0)
421                       AppendLine("     __fau_ssql = __fau_ssql + \"," + "?" + "\";");
422                    else if (psQuesorName.compareTo("F") == 0){
423                       nParam = nParam + 1;
424                       AppendLine("     __fau_ssql = __fau_ssql + \"," + ":" + nParam  + "\";");
425                    }
426                    else
427                       AppendLine("     __fau_ssql = __fau_ssql + \",\" + " + sArray + ";");
428                 }
429              }
430           } else if (nFirst) {
431              sOneValue = sOneValue.replace(")", "").trim();
432              AppendLine("     __fau_ssql = __fau_ssql +\"(" + sOneValue + "\";");
433              nFirst = false;
434           }
435
436           else {
437              sOneValue = sOneValue.replace(")", "").trim();
```

```
438                 AppendLine("     __fau_ssql = __fau_ssql + \"," + sOneValue + "\";");
439             }
440         }
441         AppendLine("     __fau_ssql = __fau_ssql + \")\";");
442
443     }
444
445
    private static void DetermineTableorProcedure(String psTable, String psType) throws
446 IOException {
        // This method is used by the Insert and Procesure call to set up the SQL command for
447 processing
448         String sTable = psTable;
        if (sTable.substring(0, 1).compareTo(":") == 0) { // This is a host variable - not a table
449 name/proceudue name
450             sTable = sTable.substring(1).trim();
451             String sFirst = sTable.substring(0, 1);
452             String sLast = sTable.substring(sTable.length() - 1, sTable.length());
453             if ((sFirst.compareTo("{") != 0) || (sLast.compareTo("}") != 0)) {
                AppendLine("//Command incorrect or not implemented - Bad Table / Procedure Name " +
454 sTable);
455                 return;
456             }
457             sTable = sTable.substring(1, sTable.length() - 1);
458             AppendLine("   String __fau_ssql = \"" + psType + "\";");
459             AppendLine("     __fau_ssql = __fau_ssql + " + sTable + ";");
460         } else {
461             AppendLine("   String __fau_ssql = \"" + psType + "\";");
462             AppendLine("     __fau_ssql = __fau_ssql + \"" + sTable + "\";");
463         }
464     }
465     private static void SetParamInsertProc(String[] psEachVariable) throws IOException {
466         // This method is used by Insert, Function Call and Procedure Call
467         // It goes through the list of Values or Parameter Call Variables and creates
468         // Param entries for each one
469         String[] sEachVariable = psEachVariable;
470         AppendLine("    int __fau_param = 0;");
471         msParams = " ";
472         String sNext = "";
473         mnCommandCount = mnCommandCount + 1;
        AppendLine("         String __fau_tag = \"" + mnCommandCount + msTagName + ":\" +
474 __fau_ssql;");
        AppendLine("                                                                    __fau_st          =
475 __fau_ec.prepareOracleBatchableStatement(__fau_cc,__fau_tag,__fau_ssql);");
```

```
476
477         for (String sOneValue : sEachVariable) {
478            if (sOneValue.substring(0, 1).compareTo(":") == 0) // This is a variable
479            {
480               if (sOneValue.substring(1, 2).compareTo("[") == 0) // And it is an Array
481               {
482                  String sArray = sOneValue.substring(2, sOneValue.length() - 1);
483                  sArray = sArray.replace("]", "");
484                  AppendLine("    for (int __faui = 0;__faui<" + sArray + ".length;__faui++) { ");
485                  AppendLine("      __fau_param = __fau_param + 1;");
486                  AppendLine("      __fau_st.setObject(__fau_param ," + sArray + "[__faui]);}");
487
488               } else {
489                  AppendLine("    __fau_param = __fau_param + 1;");
490                  AppendLine("     __fau_st.setObject(__fau_param ," + sOneValue.substring(1) + ");");
491               }
492
493            } else
494               sNext = sOneValue;
495         }
496
497      }
498
499     private static void DecodeUpdateDelete(String psCommand, BufferedWriter writer) throws
   IOException {
500        // Method used to scan all Update or Delete commands and reformat them so they can be
   processed.
501         AppendLine("{");
502         SetOracleContext();
503         String sCommand = psCommand.trim();
504         if (sCommand.length() >= 7) {
505            if (sCommand.toUpperCase().substring(0, 7).compareTo("EXECUTE") == 0) {
506               sCommand = sCommand.substring(7);
507            }
508         }
509
510         msParams = "";
511         boolean bHasUsing = false;
512         int nPosition = sCommand.toUpperCase().indexOf("APPLYING");
513         if (nPosition > 0)
514            bHasUsing = true;
515         if (bHasUsing) {
```

```
516             String[] sParts = sCommand.split("APPLYING");
517             sParts[0] = sParts[0].trim();
518             AppendLine("String __fau_ssql = " + ClauseReplace(sParts[0]) + ";");
519             sParts[1] = sParts[1].replace(";", "");
520             sParts[1] = sParts[1].replace("}", "").trim();
521             SetUsing(sParts[1]);
522          } else {
523             AppendLine("String __fau_ssql = " + ClauseReplace(sCommand) + ";");
524          }
525          mnCommandCount = mnCommandCount + 1;
526          AppendLine("          String __fau_tag = \"" + mnCommandCount + msTagName + ":\" + __fau_ssql;");
527          AppendLine("       __fau_st = __fau_ec.prepareOracleBatchableStatement(__fau_cc, __fau_tag, __fau_ssql);");
528          if (msParams != "") {
529             String SelParam[] = msParams.split("\n");
530             for (int i = 0; i < SelParam.length; i++) {
531                AppendLine(SelParam[i]);
532             }
533          }
534
535          AppendLine("    __fau_ec.oracleExecuteBatchableUpdate();");
536          AppendLine("}");
537
538       }
539
540       private static void DecodeDescribe(String psCommand, BufferedWriter writer) throws IOException {
541          // This method is used to process the DESCRIBE command that passes a Select clause into the
542          // the Describe class to get metadata information
543          String sCommand = psCommand;
544          int nSide = 0;
545          // Remove the FOR
546          sCommand = sCommand.substring(3).trim();
547          int nIn = sCommand.toUpperCase().indexOf("INTO");
548          if (nIn == 0) {
549             AppendLine("// Poorly Formated Describe Command ");
550             return;
551          }
552
553          //Get rid of in INTO - Trim it and get the class name
554          sCommand = sCommand.substring(nIn + 4).trim();
555          int nLeft = sCommand.indexOf("(");
```

```
556
557         String sClass;
558         if (nIn > nLeft) {
559            sClass = sCommand.substring(0, nLeft).trim();
560            sCommand = sCommand.substring(nLeft);
561         } else {
562            sClass = sCommand.substring(0, nIn).trim();
563            sCommand = sCommand.substring(nIn + 2).trim();
564         }
565
566         msParams = "";
567         String sRightSide = sCommand;
568         sRightSide = sRightSide.replace("(", "");
569         sRightSide = sRightSide.replace("}", "");
570         sRightSide = sRightSide.replace(")", "");
571         sRightSide = sRightSide.replace(";", "");
572
573          if (sRightSide.toUpperCase().indexOf("APPLYING") > 0) {
574            String[] sParts = sRightSide.split("APPLYING");
575            sRightSide = sParts[0];
576            sParts[1] = sParts[1].replace(";", "");
577            sParts[1] = sParts[1].replace("}", "").trim();
578            SetUsing(sParts[1]);
579         }
580
581         AppendLine("{");
582         AppendLine("String __fauSel = " + ClauseReplace(sRightSide) + ";");
583
584         if (msParams.compareTo("") != 0) {
585            String SelParam[] = msParams.split("\n");
586            for (int i = 0; i < SelParam.length; i++) {
587               AppendLine(SelParam[i]);
588            }
589         }
590         if (msContext != null)
591            AppendLine("sqlj.runtime.ref.DefaultContext __fau_cc = " + msContext + ";");
592         else
               AppendLine("sqlj.runtime.ref.DefaultContext __fau_cc =
593    sqlj.runtime.ref.DefaultContext.getDefaultContext();");
         AppendLine("if (__fau_cc==null)
594    sqlj.runtime.error.RuntimeRefErrors.raise_NULL_CONN_CTX();");
595         AppendLine("    " + sClass + ".LoadDescription(__fauSel,__fau_cc);");
```

```
596         AppendLine("}");
597     }
598
599     private static void DecodeProcedure(String psCommand, BufferedWriter writer) throws IOException {
600         //  This method handles calling a stored procedure.
601         String sCommand = psCommand;
602         int nPosition;
603         nPosition = sCommand.toUpperCase().indexOf("CALL"); // Get rid of the call statement
604         sCommand = sCommand.substring(nPosition + 4).trim();
605
606         AppendLine("{");
607         SetOracleContext();
608
609
610         // Get the parameter list and the procedure name
611         String sVariables = "";
612         String sProcedure = "";
613         compiledRegex = Pattern.compile("\\(.*\\)");
614         try {
615             regexMatcher = compiledRegex.matcher(sCommand);
616             if (regexMatcher.find()) {
617                 sVariables = regexMatcher.group();
618                 sCommand = regexMatcher.replaceFirst("").trim();
619             }
620         } catch (IllegalStateException ex) {
621             System.out.println(ex.getMessage());
622             return;
623         }
624         sProcedure = sCommand.replace(";", "").trim();
625
626         DetermineTableorProcedure(sProcedure, "CALL ");
627
628
629         // Go through and set the PARAMs.  For regular variables, just stick them in but for Arrays will need to iterate
630         String[] sEachVariable = null;
631         ;
632         if (sVariables.indexOf("(") == 0) {
633             sVariables = sVariables.substring(1, sVariables.length() - 1);
634             sEachVariable = sVariables.split(",");
635             ProcessInsertParen(sEachVariable, "?");
```

```
636         }
637         SetParamInsertProc(sEachVariable);
638
639         AppendLine("    __fau_ec.oracleExecuteBatchableUpdate();");
640         AppendLine("}");
641
642         return;
643     }
644     private static void DecodeFunction(String psCommand, BufferedWriter writer) throws IOException {
645         // This method parses out the information for an Oracle Function call.  It has special logic
646         // to handle the Out Param
647         String sCommand = psCommand;
648
649         // Get rid of the first part
650         int nPosition;
651         nPosition = sCommand.toUpperCase().indexOf("#SQLJE");
652         sCommand = sCommand.substring(nPosition + 6).trim();
653         String[] sEachSide;
654         sEachSide = sCommand.split("=");
655         if (sEachSide.length != 2) {
656             System.out.println("Poorly Formed Function Call" + psCommand + "\n");
657             return;
658         }
659         String sVariable = sEachSide[0].trim(); // This is the java variable that will receive the output from the function call
660
661         //  Remove Values and trim
662         nPosition = sEachSide[1].toUpperCase().indexOf("VALUES");
663         sCommand = sEachSide[1].substring(nPosition + 6).trim();
664         if (sCommand.substring(0, 1).compareTo("(") != 0) {
665             System.out.println("Poorly Formed Function Call" + psCommand + "\n");
666             return;
667         }
668         // Get rid of the left paren and we should be good to go
669         sCommand = sCommand.substring(1).trim();
670
671         AppendLine("{");
672         AppendLine("oracle.jdbc.OracleCallableStatement  __fau_st = null;");
673         if (msContext != null)
674             AppendLine("sqlj.runtime.ref.DefaultContext __fau_cc = " + msContext + ";");
675         else
```

```
676             AppendLine("sqlj.runtime.ref.DefaultContext __fau_cc = sqlj.runtime.ref.DefaultContext.getDefaultContext();");
677             AppendLine("if (__fau_cc==null) sqlj.runtime.error.RuntimeRefErrors.raise_NULL_CONN_CTX();");
678             AppendLine("sqlj.runtime.ExecutionContext.OracleContext __fau_ec = ((__fau_cc.getExecutionContext()==null) ? sqlj.runtime.ExecutionContext.raiseNullExecCtx() : __fau_cc.getExecutionContext().getOracleContext());");
679
680
681         // Get the parameter list and the function name
682         String sParams = "";
683         String sFunction = "";
684
685         compiledRegex = Pattern.compile("\\(.*\\)");
686         try {
687             regexMatcher = compiledRegex.matcher(sCommand);
688             if (regexMatcher.find()) {
689                 sParams = regexMatcher.group();
690                 sCommand = regexMatcher.replaceFirst("").trim();
691             }
692         } catch (IllegalStateException ex) {
693             System.out.println(ex.getMessage());
694             return;
695         }
696         sFunction = sCommand.replace(";", "").trim();
697         sFunction = sFunction.replace("}", "");
698
699          if (sFunction.substring(0, 1).compareTo(":") == 0) { // This is a host variable - not a table name/proceudue name
700             sFunction = sFunction.substring(1).trim();
701             String sFirst = sFunction.substring(0, 1);
702             String sLast = sFunction.substring(sFunction.length() - 1, sFunction.length());
703             if ((sFirst.compareTo("{") != 0) || (sLast.compareTo("}") != 0)) {
704                 AppendLine("//Command incorrect or not implemented - Function Name " + sFunction);
705                 return;
706             }
707             sFunction = sFunction.substring(1, sFunction.length() - 1);
708             AppendLine("   String __fau_ssql = \"  BEGIN :1 := \";");
709             AppendLine("     __fau_ssql = __fau_ssql + " + sFunction + ";");
710         } else {
711             AppendLine("   String __fau_ssql = \"  BEGIN :1 := \";");
712             AppendLine("     __fau_ssql = __fau_ssql + \"" + sFunction + "\";");
713         }
714
```

```
715
        // Go through and set the PARAMs.  For regular variables, just stick them in but for Arrays will
716 need to iterate
717         String[] sEachVariable = null;
718
719         if (sParams.indexOf("(") == 0) {
720             sParams = sParams.substring(1, sParams.length() - 1);
721             sEachVariable = sParams.split(",");
722             ProcessInsertParen(sEachVariable, "F");
723         }
724
725         AppendLine("     __fau_ssql = __fau_ssql + \"\\n; END;\";");
726         AppendLine("    int __fau_param = 1;");
727         msParams = " ";
728         String sNext = "";
729         mnCommandCount = mnCommandCount + 1;
        AppendLine("         String __fau_tag = \"" + mnCommandCount + msTagName + ":\" +
730 __fau_ssql;");
731         AppendLine("       __fau_st = __fau_ec.prepareOracleCall(__fau_cc,__fau_tag,__fau_ssql);");
732
733         for (String sOneValue : sEachVariable) {
734             sOneValue = sOneValue.replace(")","").trim();
735             if (sOneValue.substring(0, 1).compareTo(":") == 0) // This is a variable
736             {
737                 if (sOneValue.substring(1, 2).compareTo("[") == 0) // And it is an Array
738                 {
739                     String sArray = sOneValue.substring(2, sOneValue.length() - 1);
740                     sArray = sArray.replace("]", "");
741                     AppendLine("    for (int __faui = 0;__faui<" + sArray + ".length;__faui++) { ");
742                     AppendLine("      __fau_param = __fau_param + 1;");
743                     AppendLine("       __fau_st.setObject(__fau_param ," + sArray + "[__faui]);}");
744
745                 } else {
746                     AppendLine("    __fau_param = __fau_param + 1;");
747                     AppendLine("     __fau_st.setObject(__fau_param ," + sOneValue.substring(1) + ");");
748                 }
749
750             } else
751                 sNext = sOneValue;
752         }
753
754
```

```
755         AppendLine("    __fau_st.registerOutParameter(1,oracle.jdbc.OracleTypes.VARCHAR);");
756         AppendLine("    __fau_ec.oracleExecuteUpdate();");
            AppendLine("    " + sVariable + " =  __fau_st.getObject(1); if (__fau_st.wasNull()) throw new
757   sqlj.runtime.SQLNullException();");
758         AppendLine("}");
759
760         return;
761
762       }
763
764
765
      private static void DecodeSelectFor(String psCommand, BufferedWriter writer) throws
766   IOException {
767         //The following method handles parsing and conversion for the For->Select clause including
768         //creating the result set classes for both the regular and default context oracle connections
769         String sCommand;
770         int nSide = 0;
771         mnClassCount = mnClassCount + 1;
772         sCommand = psCommand.trim();
773         // Remove the FOR
774         sCommand = sCommand.substring(3).trim();
775         int nLeft = sCommand.indexOf("(");
776         int nIn = sCommand.toUpperCase().indexOf("IN");
777         if (nLeft == 0) {
778            AppendLine("// Poorly Formated For Command ");
779            return;
780         }
781         String sIterator;
782         if (nIn > nLeft) {
783            sIterator = sCommand.substring(0, nLeft).trim();
784            sCommand = sCommand.substring(nLeft);
785         } else {
786            sIterator = sCommand.substring(0, nIn).trim();
787            sCommand = sCommand.substring(nIn + 2).trim();
788         }
789
790         //Split the command on the IN.  To the left would be a possible variable list
791         //and to the right the select statement to be processed
792         String[] sEachSide = sCommand.split(" in | IN | In | iN ");
793         String[] sVariables = null;
794
```

```
795         //Begin creating the result set class.
796         AppendBuff("class __fauRead" + mnClassCount + "{");
797         if (sEachSide.length == 2) {
798            nSide = 1;
799            String sLeft = sEachSide[0];
800            sLeft = sLeft.replace(")", "");
801            sLeft = sLeft.replace("(", "");
802            sVariables = sLeft.split(",");
803            for (int i = 0; i < sVariables.length; i++) {
804               String sVariable = sVariables[i].trim();
805               String[] sVarName = sVariable.split(" ");
806               AppendBuff("   public " + sVarName[0].trim() + " " + sVarName[1].trim() + ";");
807            }
808         }
809         AppendBuff("    ResultSet dbTableResult;");
810
811         // If there was nothing to the left of IN, then this is a dynamic Select statement
812         // Set up all of the variables to hold meta data and create the methods to return information
813         if (sVariables == null) {
814            AppendBuff("    private Object[] dbColumn;");
815            AppendBuff("    private String[] dbColumnName;");
816            AppendBuff("    int nColumns=0;");
817            AppendBuff("    public Object Column(int pnColumn){");
818            AppendBuff("            if((pnColumn > 0) && (pnColumn <= nColumns)) return dbColumn[pnColumn];");
819            AppendBuff("    else return null;}");
820            AppendBuff("    public Object Column(String psColumn){");
821            AppendBuff("    for (int nCol= 1;nCol <= nColumns;nCol++){");
822            AppendBuff("            if (psColumn.toUpperCase().compareTo(dbColumnName[nCol])==0) return dbColumn[nCol];}");
823            AppendBuff("    return null;}");
824
825         }
826         AppendBuff("     public  __fauRead" + mnClassCount + "(PreparedStatement pdbStatement, Connection pdbConnection)");
827         AppendBuff("     throws SQLException, ClassNotFoundException { ");
828         AppendBuff("      dbTableResult = pdbStatement.executeQuery();");
829
830         // If this was a dynamic Select statements, set up the metadata from the resultset
831          if (sVariables == null) {
832            AppendBuff("    ResultSetMetaData dbRsmd = dbTableResult.getMetaData();");
833            AppendBuff("    nColumns = dbRsmd.getColumnCount();");
834            AppendBuff("    dbColumn = new Object[nColumns+1];");
```

```
835             AppendBuff("    dbColumnName = new String[nColumns+1];");
836             AppendBuff("    for (int nCol = 1;nCol <= nColumns; nCol++){ ");
                AppendBuff("                                                dbColumnName[nCol]      =
837 dbRsmd.getColumnName(nCol).toUpperCase();}");
838             AppendBuff("            return;");
839         }
840         AppendBuff(" }");
841
842         // Second method for default connections.  Repeat everything above, but with an overloaded
843         // method with the different objects needed.
            AppendBuff("    public __fauRead" + mnClassCount + "(oracle.jdbc.OraclePreparedStatement
844 pdbStatement, sqlj.runtime.ExecutionContext.OracleContext pdbConnection)");
845         AppendBuff("    throws SQLException, ClassNotFoundException { ");
846         AppendBuff("     dbTableResult = pdbStatement.executeQuery();");
847         if (sVariables == null) {
848             AppendBuff("    ResultSetMetaData dbRsmd = dbTableResult.getMetaData();");
849             AppendBuff("    nColumns = dbRsmd.getColumnCount();");
850             AppendBuff("    dbColumn = new Object[nColumns+1];");
851             AppendBuff("    dbColumnName = new String[nColumns+1];");
852             AppendBuff("    for (int nCol = 1;nCol <= nColumns; nCol++){ ");
                AppendBuff("                                                dbColumnName[nCol]      =
853 dbRsmd.getColumnName(nCol).toUpperCase();}");
854             AppendBuff("            return;");
855         }
856         AppendBuff(" }");
857
            // Create the Next method that is used to actualy move the data from Oracle to the Java
858 variables
            AppendBuff("public boolean Next() throws java.sql.SQLException,
859 ClassNotFoundException{");
860         AppendBuff("  if (dbTableResult.next()) {");
861         // If this is not the Dynamic version, assign each variable it's Oracle value
862         if (sVariables != null) {
863             for (int i = 0; i < sVariables.length; i++) {
864                 String sVariable = sVariables[i].trim();
865                 String[] sVarName = sVariable.split(" ");
866                 String sGet = "";
867                 if (sVarName[0].compareTo("String") == 0)
868                     sGet = "getString";
869                 if (sVarName[0].compareTo("int") == 0)
870                     sGet = "getInt";
871                 if (sVarName[0].compareTo("Date") == 0)
872                     sGet = "getDate";
873                 if (sVarName[0].compareTo("Float") == 0)
```

```
874                 sGet = "getFloat";
875
876             //AppendBuff("db" + sVarName[1] + " = dbTableResult." + sGet + "(" + (i + 1) + ");");
877             AppendBuff(sVarName[1] + " = dbTableResult." + sGet + "(" + (i + 1) + ");");
878           }
879         // If this is a Dynamic version, then assign the data to the Column array
880         } else {
881           AppendBuff("    for (int __faui=1;__faui<=nColumns;__faui++){");
882           ;
883           AppendBuff("      dbColumn[__faui] = dbTableResult.getObject(__faui);");
884           AppendBuff("      }");
885
886         }
887
888         AppendBuff("    return true;");
889         AppendBuff("} else {");
890         AppendBuff("   dbTableResult.close();");
891         AppendBuff("    return false;");
892         AppendBuff("}");
893         AppendBuff("}");
894         AppendBuff("}");
895
896
897         msParams = "";
898         String sRightSide = sEachSide[nSide];
899         sRightSide = sRightSide.replace("(", "");
900         sRightSide = sRightSide.replace("}", "");
901         sRightSide = sRightSide.replace(")", "");
902         sRightSide = sRightSide.replace(";", "");
903
904         if (sRightSide.toUpperCase().indexOf("APPLYING") > 0) {
905           String[] sParts = sRightSide.split("APPLYING");
906           sRightSide = sParts[0];
907           sParts[1] = sParts[1].replace(";", "");
908           sParts[1] = sParts[1].replace("}", "").trim();
909           SetUsing(sParts[1]);
910         }
911
912         AppendLine("__fauRead" + mnClassCount + " " + sIterator + ";");
913         AppendLine("{");
914         AppendLine("String __fauSel = " + ClauseReplace(sRightSide) + ";");
```

```
915
916         if (msContext != null)
917         {
918             AppendLine("sqlj.runtime.ref.DefaultContext __fau_cc = " + msContext + ";");
                AppendLine("sqlj.runtime.ExecutionContext.OracleContext __fau_ec =
    ((__fau_cc.getExecutionContext()==null) ? sqlj.runtime.ExecutionContext.raiseNullExecCtx() :
919 __fau_cc.getExecutionContext().getOracleContext());");
                AppendLine("oracle.jdbc.OraclePreparedStatement __fau_st =
920 __fau_ec.prepareOracleStatement(__fau_cc,\"fauRead\",__fauSel);");
921         }
922         else
923         {
                AppendLine("sqlj.runtime.ref.DefaultContext __fau_cc =
924 sqlj.runtime.ref.DefaultContext.getDefaultContext();");
                AppendLine("if (__fau_cc==null)
925 sqlj.runtime.error.RuntimeRefErrors.raise_NULL_CONN_CTX();");
                AppendLine(" Connection __fauConnection =
926 __fau_cc.getDefaultContext().getConnection();");
                AppendLine("PreparedStatement __fau_st =
927 __fauConnection.prepareStatement(__fauSel);");
928         }
929
930         if (msParams.compareTo("") != 0) {
931             String SelParam[] = msParams.split("\n");
932             for (int i = 0; i < SelParam.length; i++) {
933                 AppendLine(SelParam[i]);
934             }
935         }
936
937         if (msContext != null){
938             AppendLine(sIterator + " = new __fauRead" + mnClassCount + "( __fau_st ,__fau_ec);");
939             AppendLine("__fau_ec.oracleCloseQuery();");
940         }
941         else
                AppendLine(sIterator + " = new __fauRead" + mnClassCount + "( __fau_st
942 ,__fauConnection);");
943
944         AppendLine("}");
945         AppendLine("while (" + sIterator + ".Next() == true)");
946     }
947
948
949     private static String ClauseReplace(String psIn)throws IOException {
950         // This methos scans a
951         String sIn = psIn;
```

```
952         String sOut = "\"\"";
953         String[] sWord = sIn.split(" ");
954         boolean bComma = false;
955
956         for (int i = 0; i < sWord.length; i++) {
957            bComma = false;
958            if (sWord[i].indexOf(",") > 0) {
959               bComma = true;
960               sWord[i] = sWord[i].replace(",", "");
961            }
962            sWord[i] = sWord[i].replace("}", "");
963            sWord[i] = sWord[i].replace(";", "");
964            if (sWord[i].length() > 0) {
965               if (sWord[i].substring(0, 1).compareTo(":") == 0) {
966                  if (sWord[i].substring(1, 2).compareTo("{") == 0) {
967                     String sNext = sWord[i].substring(1);
968                     sNext = sNext.replace("{", "");
969                     sOut = sOut + " + " + sNext + " + \" \"";
970                  } else if (sWord[i].substring(1, 2).compareTo("[") == 0) {
971                     AppendLine(" A Dynamic Host Expression should not be included in this stream");
972                        sOut = "";
973                        return sOut;
974                  } else {
975                     sOut = sOut + " + \"? \"";
976                     if (sWord[i].indexOf(")") > 0)
977                        sOut = sOut + " + \") \"";
978                     if (msParams == "")
979                        msParams = "    int __fau_param = 0;\n";
980                     msParams = msParams + "     __fau_param = __fau_param + 1;\n";
                        msParams = msParams + "                __fau_st.setObject(__fau_param ," +
981   sWord[i].replace(")", "").substring(1) + ");\n";
982                  }
983               } else
984                  sOut = sOut + " + \"" + sWord[i] + " \"";
985               if (bComma) {
986                  sOut = sOut + " + \"" + "," + " \"";
987               }
988            }
989         }
990         return sOut;
991      }
992      private static void SetOracleContext() throws IOException {
```

```
993         // This method sets the appropriate Oracle context
994         AppendLine("oracle.jdbc.OraclePreparedStatement  __fau_st = null;");
995         if (msContext != null)
996            AppendLine("sqlj.runtime.ref.DefaultContext __fau_cc = " + msContext + ";");
997         else
               AppendLine("sqlj.runtime.ref.DefaultContext                    __fau_cc                    =
998   sqlj.runtime.ref.DefaultContext.getDefaultContext();");
            AppendLine("if                                                              (__fau_cc==null)
999   sqlj.runtime.error.RuntimeRefErrors.raise_NULL_CONN_CTX();");
            AppendLine("sqlj.runtime.ExecutionContext.OracleContext              __fau_ec              =
      ((__fau_cc.getExecutionContext()==null)  ?  sqlj.runtime.ExecutionContext.raiseNullExecCtx()  :
1000  __fau_cc.getExecutionContext().getOracleContext());");
1001     }
1002      private static void SetUsing(String psIn) {
1003         //  This method accepts a "Dynamic Field Variable" and generates an iteration
1004         //  through the array to set parameter values.  The code assumes that some parameters
1005         //  may have already been set by other fields in the command
1006        String sIn = psIn;
1007         sIn = sIn.replace("[", "");
1008         sIn = sIn.replace("]", "");
1009         sIn = sIn.replace(":", "").trim();
1010         if (msParams == "")
1011            msParams = "    int __fau_param = 0;\n";
1012         msParams = msParams + "    for (int __faui = 0;__faui<" + sIn + ".length;__faui++){ \n";
1013         msParams = msParams + "     __fau_param = __fau_param + 1;\n";
1014         msParams = msParams + "     __fau_st.setObject(__fau_param ," + sIn + "[__faui]" + ");}\n";
1015         return;
1016      }
1017
1018     private static String RemoveLiterals(String psLine, String psPattern) {
1019         //  This method scans for literals and certain comments
1020         //  When found, the text is saved in the msLiteral ArrayList
1021         //  and the text replaced with !%&N&%!
1022         //  This allows literals to be ignored by all other routines
1023         //  The AppendLine routine replaces these literals back when each line if written
1024         String sLine = psLine;
1025         boolean bContinue = true;
1026
1027         while (bContinue) {
1028            compiledRegex = Pattern.compile(psPattern);
1029            try {
1030               regexMatcher = compiledRegex.matcher(sLine);
1031               if (regexMatcher.find()) {
```

```
1032
1033                msLiterals.add(regexMatcher.group());
1034                sLine = regexMatcher.replaceFirst("!%&" + msLiterals.size() + "&%!");
1035             } else
1036                bContinue = false;
1037          } catch (IllegalStateException ex) {
1038             System.out.println(ex.getMessage());
1039             return psLine;
1040          }
1041       }
1042       return sLine;
1043    }
1044
1045    static void AppendLine(String psLine) throws IOException {
1046       // This methos writes lines to the output file
1047       String sLine = psLine;
1048       String sValue = "";
1049       compiledRegex = Pattern.compile("!%&\\d+&%!");
1050       try {
1051          boolean bContinue = true;
1052          while (bContinue == true) {
1053             regexMatcher = compiledRegex.matcher(sLine);
1054             if (regexMatcher.find()) {
1055                sValue = regexMatcher.group();
1056                sValue = sValue.substring(3, sValue.length() - 1);
1057                sValue = sValue.substring(0, sValue.length() - 2);
1058                int iReplace = Integer.parseInt(sValue) - 1;
1059                String msReplace = msLiterals.get(iReplace).toString();
1060                sLine = regexMatcher.replaceFirst(msLiterals.get(iReplace).toString());
1061             } else {
1062                bContinue = false;
1063             }
1064          }
1065
1066
1067       } catch (IllegalStateException ex) {
1068          System.out.println(ex.getMessage());
1069          AppendLine("Error in ParseSQLJ - " + ex.getMessage());
1070          return;
1071       }
1072
```

```
1073            writer.write(sLine);
1074            writer.newLine();
1075
1076        }
1077
1078        static void AppendBuff(String psLine) {
1079        //  This method adds lines to the Class Buffer
1080            msSelectClass = msSelectClass + psLine + "\n";
1081
1082        }
1083    }
```

```
1  // Describe Class – Compiled into JAR file and included in any sqlje program
2  // wishing to use the Describe command
3  public class Describe {
4     public String[] sColumnName;
5     public String[] sColumnType;
6     public String sError;
7     public int[] nColumnLength;
8     public int[] nPrecision;
9     public int[] nIsNullable;
10
11    private ResultSet rm = null;
12    private PreparedStatement s = null;
13    Connection conn = null;
14    int nColumns = 0;
15
16    public Describe() {
17    }
18
19    public void LoadDescription(String psStatement, sqlj.runtime.ref.DefaultContext __fau_cc)
   throws SQLException, ClassNotFoundException {
20       try {
21       Connection dbConnection = __fau_cc.getDefaultContext().getConnection();
22       Statement stmt = dbConnection.createStatement();
23       ResultSet rs = stmt.executeQuery(psStatement);
24       ResultSetMetaData dbRsmd = rs.getMetaData();
25       nColumns = dbRsmd.getColumnCount();
26       //test = new Object[nColumns];
27       sColumnName = new String[nColumns + 1];
28       sColumnType = new String[nColumns + 1];
29       nColumnLength = new int[nColumns + 1];
30       nPrecision = new int[nColumns + 1];
31       nIsNullable = new int[nColumns + 1];
32       sColumnName[0] = "";
33       sColumnType[0] = "";
34       nColumnLength[0] = 0;
35       nPrecision[0] = 0;
36       nIsNullable[0]=0;
37
38
39       for (int nCol = 1; nCol <= nColumns; nCol++) {
40          sColumnName[nCol] = dbRsmd.getColumnName(nCol);
```

```
41              sColumnType[nCol] = dbRsmd.getColumnTypeName(nCol);
42              nColumnLength[nCol] = dbRsmd.getColumnDisplaySize(nCol);
43              nPrecision[nCol] = dbRsmd.getPrecision(nCol);
44              nIsNullable[nCol] = dbRsmd.isNullable(nCol);
45          }
46          }
47              catch (SQLException ex){
48                  sError = ex.getMessage();
49              }
50      }
51
52  }
```

BIBLIOGRAPHY

[1] Jose Luis Hurtado. *Reducing Impedance Mismatch in SQL Embeddings for Object-Oriented Programming Languages***.** Florida Atlantic University, College of Electrical Engineering and Computer Science, Summer 2012.

[2] George Copeland and David Maier. *Making Smalltalk a Database System.* SIG-MOD Rec., 14(2):316-523, June 1984

[3] Jason Price. *Java Programming with Oracle SQLJ.* O'Reilly & Associates Inc., August 2001, ISBN 8173663807

[4] Nirva Morisseau-Leroy, Martin K Solomon and Gerald Momplaisir. *Oracle91 SQLJ Programming*. Oracle Press, 2001. ISBN 0072190930

[5] C.J.Date. *An Introduction to Database Systems, Eighth Edition.* Addison-Wesley, August 2003. ISBN 0321197844

[6] Gary B. Shelly, Thomas J. Cashman. Introduction to Computers and Data Processing. Anaheim Publishing Company, 1980. ISBN 0882361155

[7] Andrew Troelsen. *Pro C# 2010 and the .Net 4 Platfor, Fifth Edition*. Apress, 2010. ISBN 1430225492

[8] Syed Mujeeb Ahmed, Jack Melnick, Neelam Singh, Tim Smith. *Pro*C/C++ Precompiler Programmer's Guide, Release 9.2*, Oracle Corporation, March 2002

[9] John Levine. *flex & bison*. O'Reilly Media, 2009. ISBN 0596155972

[10] Mehran Habibi. *Java Regular Expressions: Taming the java.util.regex Engine.* Apress, 2004. ISBN 1590591070