SQLJ: TRICKS, TRAPS, AND GEMS

AN IRREVERENT, INDISPENSABLE, AND INTERACTIVE PRIMER ON ORACLE SQLJ

Ekkehard Rohwedder, Oracle Corporation

WHO IS THIS FOR? You know SQL and Java, and you want to learn Oracle SQLJ — read on!

WHAT IS THIS? This is a *primer* —a small introductory pamphlet— on Oracle's version of SQLJ. If you use paint primer, you cover the entire area, although the primer might not cover everything in the required depth or finish. You also use primer to ignite an explosive charge. We hope you will find this primer wide-ranging, brief, indispensable, and lighthearted — in other words: *a blast.*

WHAT ELSE DO I GET? A truckload of warnings, references, and exercises.

- Throughout this primer you will find dire *warnings*.

> **Warning:** Even if you do not read anything else, read (and heed) these warnings! You will be glad you did and will save yourself time and frustration.

- You will also find a number of *references* to the real documentation, as well as to the demo programs. Follow these if you are stuck, if you need more information, or if you want to learn the truth. Also make sure you peruse the other available SQLJ information.

Ref *SQLJ Developer's Guide and Reference*, Part No. A64684-02
Ref SQLJ Demo Programs at [Oracle Home]`/sqlj/demo/`
Ref *JDBC Developer's Guide and Reference*, Part No. A64685-02
Ref *JPublisher Developer's Guide and Reference*, Part No. A68027-02
Ref SQLJ Whitepaper "An Overview of SQLJ - Embedded SQL in Java", at [Oracle Home]`/sqlj/doc/sqlj-overview.pdf`
Ref SQLJ Whitepaper "Using Oracle Objects in SQLJ Programs", at [Oracle Home]`/sqlj/doc/sqlj-objects.pdf`
Ref SQLJ Runtime Documentation, at [Oracle Home]`/sqlj/doc/runtime/javadoc/packages.html`
Ref Oracle SQLJ Website, at `http://www.oracle.com/java/sqlj/`
Ref Oracle Technology Network Website, at `http://technet.oracle.com -> java -> sqlj&jdbc`

- Finally, you will discover lots of *exercises* throughout. The number of stars indicate the difficulty of the exercise — easy (*), medium (**), and hard (***). If you are a learning-by-doing kind of guy (or gal), then just pick up some of the problems and have fun. At this point, there are no sample solutions to the exercises. You might want to check the preceding web site or —better yet— send in your solutions.

Exercise 1: (*) Why are the exercises given in small print? Answer: Just as in life, it's the small print that counts.

- This primer specifically addresses the *command line version* of SQLJ. Not everything covered here is equally applicable to the Oracle JDeveloper development environment or to the SQLJ translator that is part of the JServer VM.

INSTALLING SQLJ

Install the Java Development Kit 1.1 or later (you can get it from `www.sun.com`). Note that JDK 1.0.2 will not do. Make sure that the current directory "." is in your `CLASSPATH`. You should now be able to say `java` and `javac`.

Get an Oracle JDBC driver (unless you already have one installed). It contains the file `classes111.zip` (and `classes12.zip` if you are using JDK 1.2), which must be placed in your `CLASSPATH`. Depending on the type of driver you use, you might also need a dynamic library (*`.so` or *`.dll`) in your `LD_LIBRARY_PATH` or, respectively, `PATH`. Follow the JDBC installation instructions for this. You should now be able to compile and run JDBC programs.

Ref See also *JDBC Developer's Guide and Reference*, Chapter 2, Section "Verifying a JDBC Client Installation".

If you obtained Oracle SQLJ from an Oracle8*i* or later database installation, then you have an [Oracle Home]. The SQLJ executable (named `sqlj` or `sqlj.exe` depending on your flavor of operating system) lives in [Oracle Home]`/bin`, which will be in your `PATH`. You must also put [Oracle Home]`/sqlj/lib/translator.zip` —which contains the SQLJ translator— in your `CLASSPATH`.

If you obtained Oracle SQLJ from the Oracle website, you must perform a bunch of untarring and (g-)unzipping, and you will eventually end up with a `sqlj` directory. The executable lives in `sqlj/bin` and must be in your `PATH`, and the `sqlj/lib/translator.zip` file must be in your `CLASSPATH`.

Are you greeted with a help screen when you say `sqlj`? Yes? Then you are in business!

Warning: Make sure that you have a JDK and the Oracle JDBC drivers properly installed before starting the SQLJ installation. Also, if you have several versions of Java or Java Development Environments installed, we recommend that you "build" up your PATH and your CLASSPATH environment from scratch to make sure to properly pick up a known Java and JDBC configuration. On NT, consider creating a `.bat` file that you can use in a DOS window to provide the appropriate setup.

Warning: If you are using SQLJ version 8.1.5 on NT, you must `set SQLJ_OPTIONS=-passes.`

Ref For more details, see *SQLJ Developer's Guide and Reference*, Chapter 2, Sections "Checking the Installation and Configuration" and "Testing the Setup". While we mention only the `translator.zip` file in this discussion, you should distribute the `runtime.zip` file (not `translator.zip`) with any SQLJ applications that you create.

WHAT'S IN A NUMBER?

This primer describes the *Oracle SQLJ 8.1.6 release*. Applications that you created with the Oracle SQLJ 8.1.5 translator will continue to run with the 8.1.6 runtime, or you can recompile them to take advantage of new features, such as statement caching (see Section "5.8 Give Me Speed … or More Speed"). Although you would typically use the 8.1.6 version of SQLJ with Oracle's 8.1.6 JDBC driver, you can also use it together with the 8.0.6 or 7.3.4 JDBC versions.

The following features are new in version 8.1.6: the `-checksource` flag (see Section "1.1 I Need To Tell You"), the method `Oracle. close()` (see Section "1.4 Getting Connected"), support for JDBC 2.0 types and interfaces (`java.sql.Struct`/`Ref`/`Array`/`Blob`/ `Clob` and `SqlData` - see Sections "3.3 She Is An Oracle Type" and "5.5 Let's Get Objective"), the `-jdblinemap` and `-P-CshowThreads` options (see Section "4.2 Where Is The Bug"), the offline SQL parser (see Section "5.7 Isn't It — Portable"), performance enhancements (see Section "5.8 Give Me Speed … or More Speed"), and support for iterator subclassing (see Section "5.9 Classy Kinds of Iterators").

CONTENTS

SECTION 1 SKELETON OF A SQLJ PROGRAM: SQLJ command line / Online checking of SQLJ programs / SQLExceptions in SQLJ / Connecting to the database at runtime / Starting a TCP/IP listener on Unix

SECTION 2 THE MEAT OF A SQLJ PROGRAM: SQL statements in SQLJ / Host expressions / Stored function calls / Stored Procedure calls and argument modes / SQLJ iterator concept / Named iterators / Positional iterators

SECTION 3 A UNIVERSE OF TYPES: SOME SQL, SOME JAVA: JDBC types and NULL handling in SQLJ / SQLJ Stream types / Oracle type extensions

SECTION 4 YOUR OWN PRIVATE TRANSLATOR: Helpful SQLJ options / Debugging options: `-linemap` and `-P-debug` / Programming SQLJ applets

SECTION 5 THE REST OF THE STORY — ADVANCING THE FEATURES: SQLJ-JDBC interoperability / Using connection context instances / Using execution contexts / Using typed connections / Using Oracle Objects / SQLJ in JServer / Portability / Performance / Iterator Subclassing

FEEDBACK AND ACKNOWLEDGEMENTS

Oracle SQLJ is supported through Oracle's World Wide Support. Call 911 in the Bug Database to reach SQLJ — yes, the product number is 911! Please direct other questions, corrections, praise, scathing critique, postcards, solutions to exercises, suggestions, encouragement, and so on to the author of this pamphlet at `erohwedd@us.oracle.com`.

Thank you Brian Becker, Brian Wright, and Ellen Barnes for comments on an earlier draft! Thanks Adrian, Pierre, and Jeremy for the encouragement!

1 SKELETON OF A SQLJ PROGRAM

SQLJ COMMAND LINE / ONLINE CHECKING OF SQLJ PROGRAMS / SQLEXCEPTIONS IN SQLJ / CONNECTING TO THE DATABASE AT RUNTIME / STARTING A TCP/IP LISTENER ON UNIX

In this section we look at the essential components that every SQLJ program needs — in other words, the skeleton. Before getting the skeleton out of the closet, some preliminary remarks.

1.1 I NEED TO TELL YOU

Let's start out by looking at how you translate, compile, and run your SQLJ program. Make sure that it lives in a file with the extension *.`sqlj` (instead of *.`java`). Then translate and compile your files in one step.

```
sqlj MyFile.sqlj MyOtherFile.sqlj MyJavaFile.java
```

Yes, this even compiles your Java files in the same fell swoop. This should —if everything goes all right— create *.`class` files (and some *.`ser` files), and you can then issue

```
java MyFile
```

provided, of course, that `MyFile` has a method

```
public static void main(String[] args) { ... }
```

Even though you are familiar with `.class` files —the result of Java compilation— you will be curious about these `.ser` files that the SQLJ translator produces. We also call them *(serialized) profiles*. They are serialized Java objects that contain all the information about the static SQL statements in your `.sqlj` source files, such as the SQL code, the types and names of the host variables that occur in the SQL statement, and what kind of SQL statement this is (a commit/rollback, a query, a DML statement, and so on).

Warning: Make sure that all the Java classes referenced by your program are either passed as a `.sqlj` or `.java` source file on the SQLJ command line or can be accessed through your `CLASSPATH`.

Ref When you invoke `sqlj`, a number of things go on "under the covers". To get the full story, see *SQLJ Developer's Guide and Reference*, Chapter 1, Section "Basic Translation Steps and Runtime Processing", and Chapter 9, Section "Internal Translator Options".

Ref To learn more about profiles, see *SQLJ Developer's Guide and Reference*, Chapter 10, Section "More About Profiles".

Exercise 1: (**) What problem arises (and under what circumstances) when the preceding Warning is not followed? What difference in behavior do you notice when you give the option `-checksource=false` (Note: This option sets the same behavior that SQLJ version 8.1.5 has.)

1.2 WANT TO CHECK IT OUT? - GET ONLINE!

Without a database, the SQLJ translator can perform only *offline checking* of your SQL code. If you want to get your database involved, that is, you want SQLJ to perform *online checking*, then you must tell the translator how to connect to it. Specifically, you must supply a user name (corresponding to the database schema you want to connect to) and a password.

```
sqlj -user=scott/tiger MyFile.sqlj
```

Of course, you also want to be able to say which database you'd like to talk to and how — that is, with which protocol. Because SQLJ uses JDBC underneath, this is accomplished by a JDBC URL. By default, SQLJ uses the OCI8 JDBC URL. This is the string `"jdbc:oracle:oci8:@"` - see Section "1.4 Getting Connected". However, you can also specify your own URL. For example, you can request the Oracle OCI7 driver (if available) as follows.

```
sqlj -user=scott/tiger@jdbc:oracle:oci7:@  MyFile.sqlj
```

And there is a special shorthand notation if you use Oracle's thin JDBC driver.

```
sqlj -user=scott/tiger@my_host:1521:my_oracle_sid MyFile.sqlj
```

Finally, you can use "shorthand" on the command line and write `-u` instead of `-user=`, as follows.

```
sqlj -u scott/tiger MyFile.sqlj
```

Warning: The translator will try to check your SQLJ programs against the database if, and only if, you specify the `-user` option (or `-u` shorthand).

Ref The `-user` and `-u` flags are two of the 46 or so option flags that SQLJ accepts. To see a synopsis of all options, refer *to SQLJ Developer's Guide and Reference*, Chapter 8, Table 8-1 "SQLJ Translator Options". Or issue `sqlj -help` to get an introduction to the most important ones. See also Section "4.1 The Translator Is Talking Back".

Once you have gained some familiarity with SQLJ, you might want to try the exercises below to learn more about how the SQLJ translator reports errors, both offline and online (see also *SQLJ Developer's Guide and Reference*, Chapter 9, Section "Internal Translator Operations" for more details).

Exercise 1: (*) When run offline, SQLJ checks the legality of Java types used in SQL statements, and some (rather superficial) syntax. Show some errors caught by the translator offline.

Exercise 2: (*) When run online, SQLJ additionally checks the shape of SELECTs, type compatibility between SQL and Java, and asks the database to parse SQL DML statements. Show some errors caught by the translator online, but not offline.

Exercise 3: (*) Show some errors caught only at runtime, but not at translation time.

Exercise 4: (*) In the examples above, the password was given on the command line. Usually, you want to avoid doing this.
(a) What happens if you omit the password in the `-user` option?
(b) You can use the `sqlj.properties` file for storing command line options used for `sqlj` invocation. Investigate the format of this file, and store the password information in it. What happens if an option is given in both the command line and the `sqlj.properties` file?

1.3 ERRORS WANT TO BE CAUGHT

One of the first lines in your SQLJ program will be

```
import java.sql.SQLException;
```

Whenever something goes wrong while running your SQLJ program, your SQLJ statements and any methods in the SQLJ runtime API throw a `SQLException`. Either declare that your program throws a `SQLException`, or put

```
try { ... } catch (SQLException exn) { ... }
```

blocks in your program.

Ref For more details on `SQLExceptions` in SQLJ, see *SQLJ Developer's Guide and Reference*, Chapter 4, Section "Exception-Handling Basics".

Exercise 1: (*) Create the following file `test.sqlj`:

```
public class test {
   public static void main(String[] args)   {
     #sql { ROLLBACK };
} }
```

What do you see when you run `sqlj test.sqlj`? Why? How can you fix this? After fixing and translating, run `java test`. What happens? Why? Now read Section "1.4 Getting Connected" and fix this problem.

Exercise 2: (***) A `SQLException` can originate from the database, from JDBC, or from SQLJ itself. Can you write a SQLJ program that creates all three kinds of errors at runtime? *Hint*: you might want to translate this program offline, and you also want to read up on SQLJ-JDBC interoperability — see Section "5.1 A Dynamic Program".

1.4 GETTING CONNECTED

What good is a SQL program without a database connection? Another important import line is the following.

```
import oracle.sqlj.runtime.Oracle;
```

The first thing you must do before executing a SQLJ statement is to connect to the database. (Note: Not true in the server — the stored procedure or function that is implemented by a SQLJ method runs in a database session that already has a connection going for it!)

```
Oracle.connect("jdbc:oracle:oci8:@", "scott", "tiger");
```

Your user name —equivalent to the database schema you are connecting to— is `"scott"` and your password `"tiger"`. The first argument to `connect()` is the JDBC URL. If you want to connect to a different database, just place the database alias from your `$ORACLE_HOME/work/tnsnames.ora` after the "@". The following are connect strings for Oracle JDBC.

`jdbc:oracle:oci7:@`	For an OCI7 connection
`jdbc:oracle:oci8:@`	For an OCI8 connection
`jdbc:oracle:thin:@`*hostname*`:`*port*`:`*oracle-sid*	For a thin JDBC connection
`jdbc:oracle:kprb`	For the session in the server.

Table 1 - List of Oracle JDBC URLs.

The following is a sample thin JDBC URL: `"jdbc:oracle:thin:@localhost:1521:orcl"`.

The `connect()` method also has a twin: `Oracle.close()` — always invoke this method to close your connection!

Ref Refer to the JDBC documentation for specifics — *JDBC Developer's Guide and Reference*, Chapter 3, "First Steps in JDBC".

Ref There are many more ways to establish a connection in SQLJ than the particular `Oracle.connect()` method shown here. For more information, refer to *SQLJ Developer's Guide and Reference*, Chapter 4, Section "Connection Considerations" and —for advanced users— Chapter 7, Section "Connection Contexts".

Warning: The `Oracle.connect()` method sets the single, static connection for your program. If your program uses multiple connections, or you program an applet or a multithreaded application, you must use explicit connections, which are explained in Section "5.2 Being Well Connected — Explicitly".

Warning: `Oracle.connect()` has —by default— auto-commit turned off. You must issue a SQL COMMIT statement to make any changes permanent. The JDBC connection mechanism `DriverManager.getConnection()` turns —by default— auto-commit on.

Exercise 1: (*) Try to connect with the following JDBC URLs and observe what happens. Explain.

```
jdbc:notoracle:oci8:@
jdbc:oracle:oci:@
```

Exercise 2: (*) What happens when you connect a second time using the `Oracle.connect()` method? Is a new connection established, or do you continue to be connected with the original connection? Write a `sqlj` program to find out!

Exercise 3: (*) Look at the SQLJ demos in [Oracle Home]`/sqlj/demo/`. Instead of hard-coding connection parameters in the SQLJ program, these are loaded from a `connect.properties` file. Rewrite your examples to use this feature.

Exercise 4: (***) You can use the same `Oracle.connect()` and `Oracle.close()` code both, on the client and in the JServer VM. Explain how this is possible.

1.5 IS ANYBODY LISTENING?

If you want to connect to your database with the thin JDBC driver, then your database listener must listen on a TCP/IP port. If you belong to the GUI-challenged group of Unix users, you can achieve this by editing your `$ORACLE_HOME/work/listener.ora` file, adding an additional line to:

```
LISTENER = (ADDRESS_LIST=
 (ADDRESS=(PROTOCOL=ipc)(KEY=oracle-sid)) )
```

as follows:

```
LISTENER = (ADDRESS_LIST=
 (ADDRESS=(PROTOCOL=ipc)(KEY=oracle-sid))
 (ADDRESS=(PROTOCOL=tcp)(HOST=hostname)(PORT=port)) )
```

Now you must stop and then re-start your listener to pick up the new settings:

```
lsnrctl stop; lsnrctl start
```

Exercise 1: (*) Get your database listener to listen to a TCP/IP port. Write a SQLJ program that connects to this port using the thin driver and run it. What error is reported if there is no listener on the specified TCP/IP port?

2 THE MEAT OF A SQLJ PROGRAM

SQL STATEMENTS IN SQLJ / HOST EXPRESSIONS / STORED FUNCTION CALLS / STORED PROCEDURE CALLS AND ARGUMENT MODES / SQLJ ITERATOR CONCEPT / NAMED ITERATORS / POSITIONAL ITERATORS

When you want to embed SQL in Java, you will inevitably use SQLJ *statements* and —in most cases— SQLJ *iterators*. This chapter explains the basic concepts of both.

2.1 SQLJ IS EMBEDDED SQL

So how do we issue "COMMIT" to get the changes in our transaction committed, or —for that matter— how do we write other SQL statements? It's straightforward.

```
#sql { UPDATE emp SET sal = 3000 WHERE ename = 'SCOTT' };
#sql { COMMIT };
```

You can put any SQL statement (including DDL, DML, PL/SQL declarations and blocks) between the curly braces, and it will get sent to the database as is - SQL comments and all!

Warning: Every SQLJ statement must be terminated with a semicolon "`;`"

Ref See also *SQLJ Developer's Guide and Reference*, Chapter 3, Section "Overview of SQLJ Executable Statements".

Exercise 1: (*) What happens if you omit the semicolon "`;`" at the end of the SQLJ statement (after the closing curly brace)? What happens if you put a semicolon "`;`" at the end of the SQL statement (just before the closing curly brace)?

Exercise 2: (**) Write a DDL statement, a DML statement, a PL/SQL block, and a PL/SQL declaration in SQLJ. Sprinkle some SQL comments in.

2.2 COOLER THAN HOST VARIABLES: HOST EXPRESSIONS

SQL statements that cannot retrieve values from or send values to the database are not terribly programmable. That's where host variables come in. They are Java variables prefixed with "`:`", placed inside the SQL statement, that can retrieve and/or send data values.

```
String name  = "SCOTT";
Double raise = new Double(1.08);
Double salary;

#sql { UPDATE emp SET sal = sal * :raise WHERE ename = :name };
#sql { SELECT sal INTO :salary FROM emp WHERE ename = :name };
```

But SQLJ is more flexible than that — you can use Java expressions instead of host variables. Just make sure that the host expression is enclosed between "`:(`" and "`)`".

```
String[] emps = new String[] { "Scott", "Miller", "King" };
double[] raises = new double[] { 8.0, 4.0, 0.0 };

for (int i=0; i<emps.length; i++)
   #sql { UPDATE emp SET sal = sal * :(1.0 + raises[i] / 100.0)
          WHERE ename = :(emps[i].toUpperCase()) };

int j=0; double[] s = new double[emps.length];
while (j<emps.length) {
   #sql { SELECT sal INTO :(s[j]) FROM emp
          WHERE ename = :(emps[j++].toUpperCase()) };   }
```

Warning: All host expressions are evaluated once, and only once, from left to right (including side-effects) before any values are sent to the database.

Ref See also *SQLJ Developer's Guide and Reference*, Chapter 3, Section "Evaluation of Java Expressions at Runtime".

Ref An application with host expressions is [Oracle Home]`/sqlj/demo/ExprDemo.sqlj`.

Exercise 1: (*) You can use host expressions where values are expected. Write a SQLJ statement with a host expression in an illegal place. Translate and run it. What happens?

Exercise 2: (*) A host expression in an INTO list must be able to receive a data value. Write a host expression that is not legal in an INTO list. What happens when you compile and run your program?

Exercise 3: (**) Come up with more SQLJ statements that demonstrate that SQLJ evaluates host expressions from left to right.

Exercise 4: (*) Show that you can use SQL comments in SQL text (between `{` and `}`), and that you can use Java comments inside of Java host expressions in SQLJ statements.

2.3 LET'S GET RESULTS — FUNCTIONS FIRST

We already saw how results can be received from a SQL statement when we used the SELECT-INTO statement. More often, results from a SQL operation are received by a SQLJ assignment statement. Let's look at a call to the (built-in) SQL function SYSDATE().

```
java.sql.Date today;
#sql today = { VALUES( SYSDATE() ) };
System.out.println("The database thinks that today is "+today);
```

The VALUES(...) syntax is SQLJ-specific syntax for calling a stored function. Such functions might also take arguments, such as in the following code snippet.

```
String in10Days;
#sql in10Days = { VALUES( DELTA_DATE(:today, 10) ) };
```

Note that we can receive a SQL DATE value in different formats in Java — in our examples, as a `java.sql.Date` and as a `java.lang.String`.

Exercise 1: (**) Write the PL/SQL function DELTA_DATE that takes a DATE and an INTEGER and returns a new DATE that is INTEGER many days in the future. Now run the SQLJ program above.
Can you create the PL/SQL function in the SQLJ program itself?

2.4 ARE WE OUT-MODED YET? — GETTING INTO PROCEDURES

In our discussion above, we glossed over the fact that host variables (host expressions) are used in different modes.

- *IN* - The value of the expression is sent to the database.
- *OUT* - The expression denotes a location and receives a value from the database.
- *INOUT* - All of the above.

By default, host-expressions have the mode IN, with the exception of host-expressions in INTO-lists, which have the mode OUT. In all other cases, you have to explicitly prefix the host expression with the mode. For example:

```
int x;
int y = 10;
#sql { BEGIN :OUT x := :y + :y; END };
```

Oooops. There is one more exception (but this is the last one, I promise): in the SET statement, which is part of the SQLJ language, the left-hand side of the assignment is implicitly OUT. Thus the following is functionally identical to our BEGIN ... END block above.

```
#sql { SET :x = :y + :y };
```

Stored Procedures (and Oracle Stored Functions) can have parameters with all three modes. The SQLJ syntax for calling stored procedures is illustrated in the following code fragment.

```
int x = 10;
int y;
int z = 20;
#sql { CALL MyProc( :x, :OUT y, :INOUT z ) };
```

Warning: You must add OUT or INOUT modes to all host expressions in stored function and procedure arguments that do not have the mode IN. Otherwise, you will not see any values returned from the database in these positions.

Warning: You must add OUT or INOUT modes to all host expressions in PL/SQL blocks that do not have the mode IN. Otherwise, you will not see any values returned from the database from the PL/SQL block.

Ref See also *SQLJ Developer's Guide and Reference*, Chapter 3, Section "Stored Procedure and Function Calls". The SET statement is described in the same chapter in the Section "Assignment Statement (SET)".

Exercise 1: (*) Write a stored procedure MyProc. Call it using the SQLJ program fragment above.

Exercise 2: (**) Write a stored function MyFunc that takes all three kinds of arguments, as well. Show that the assignment of the result takes place after the assignments of the out parameters. Show that out parameters are assigned from left to right.

```
#sql ... = { VALUES( MyFunc( :..., :OUT ..., :INOUT ... ) };
```

Exercise 3: Show that all Java host expressions (including OUT or INOUT expressions that evaluate to assignable locations — so called "lvalues") are evaluated before the SQL statement is executed.

Exercise 4: (*) Omit the OUT and INOUT markers in your program. What happens? How can you catch this problem at translation time, rather than when you run your program?

Exercise 5: (**) Now write a PL/SQL block that takes IN, as well as OUT or INOUT arguments. Omit the OUT and INOUT markers in your program. What happens? Can you catch this problem at translation time, rather than when you run your program? Remember that PL/SQL blocks have one of the following forms.

```
#sql { BEGIN ... PL/SQL statements ... END };
#sql { DECLARE ... PL/SQL declarations ... BEGIN ... PL/SQL statements ... END };
```

Exercise 6: (**) What are the advantages to require that the modes of host variables must be specified syntactically in the SQLJ language? *Hint:* how could SQLJ determine the modes if they are not known? What consequence does this have for translating SQLJ programs?

2.5 LOOK MA: RESULT SETS ARE TYPED … AND ARE CALLED ITERATORS!

When you execute a query in JDBC, it will return a `java.sql.ResultSet`. You then retrieve the rows in the result set through a processing loop. The `next()` method on the `ResultSet` returns `true` if another row is available. In this case, the row is retrieved, and the individual columns can be accessed through `getXxxx(`*column_number*`)` calls, where `Xxxx` represents the Java type, with which you want to retrieve the column, such as `String`, `Int` (for `int`), `Double` (for `double`), ...

SQLJ does not have the "amorphous" result sets of JDBC. SQLJ query results are always strongly typed — each column in the result has a particular Java type. To differentiate these "typed result sets" from the JDBC notion of `ResultSet` and from the SQL notion of cursor, we call them *iterators*. SQLJ provides two flavors of iterators.

- *Positional iterators* are the "plain vanilla" variety. They are characterized by the Java types of the columns. You use a FETCH statement to retrieve the columns of a row from an iterator. This will look familiar, if you are used to other languages with embedded SQL.
- The *named iterators* have the "mocha flavor". You specify both the column types in Java, as well as the column name. This name also serves as the name of the accessor function, with which you retrieve the column value. This kind of iterator is most "Java*ish*", and JDBC programmers will immediately feel familiar with it.

Enough talk — you want to see code? Hold on!

2.6 WHAT'S IN A NAME?

So, how do you get your iterator with all of these names and types? You declare it, of course!

```
#sql iterator NamedIter (String ename, Double sal);
```

This line creates a Java class declaration for the `NamedIter` class — right where you wrote it. This class has `next()` and `close()` methods — just like the `java.sql.ResultSet`. Instead of the `getXxxx(`*column_name*`)` accessors, however, your `NamedIter` class sports two fully customized, tailor-made, individualized accessor methods known as `String ename()` and `Double sal()`. A minor detail: you will have most success with this declaration if you put it where Java class declarations are permitted.

Let's declare ourselves a `NamedIter`.

```
NamedIter n;
```

And —better yet— populate it with the result from a query.

```
#sql n = { SELECT ename, sal FROM emp };
```

How do you use this iterator? Whaddayaknow, I told you all about these methods that you find in `NamedIter`.

```
while (n.next()) {
    System.out.println(n.ename()+" would like to make "+ (n.sal()*2));
}
n.close();
```

Open questions. You should now have a gazillion questions about named iterators, such as: Where do you declare an iterator type? Does the order in the SELECT list matter? How do you match SQL and Java names? What about case sensitive and case insensitive names? Can you say "SELECT * FROM EMP"? and so on. Not to worry — you will discover the answers to all these questions from the exercises below!

Warning: If you want to declare an iterator locally (as an inner class), we recommend that you declare it as follows.

```
#sql public static iterator IteratorName( … );
```

Warning: You must always `close()` your iterators once you are done using them, or you will run out of cursors to connect to the database with. This is even more important if your Java code runs in the server: Mercilessly, the JavaVM in the JServer (unlike on the client) will not reclaim and close open cursors when your stored program terminates.

Ref *SQLJ Developer's Guide and Reference*, Chapter 3, Section "Multi-Row Query Results— SQLJ Iterators" and Section "Overview of SQLJ Declarations" describe iterators.

Ref The demo [Oracle Home]`/sqlj/demo/NamedIter.sqlj` contains named iterators.

Exercise 1: (**) Where do you declare an iterator type?Actually, you can declare an iterator type (or, equivalently, an iterator class) wherever you can declare a Java class.

(a) Show that you can declare iterators in different locations (top-level, nested, and so on).

(b) Show that you can use modifiers (for example `public`, `static`, ...) on iterator declarations, the same way you use them in class declarations.

(c) You want to declare an iterator as:

```
#sql public iterator PubEmp(String ename);
```

Where do you have to place this iterator declaration? Why?

(d) Why did we give the recommendation to declare nested iterator classes as `public static`? What happens if you omit `static`? What happens if you omit `public`? What if you try to return this iterator as a column of another iterator or as a parameter of a stored procedure or function?

Exercise 2: (*) Does the order in the SELECT list matter? Write a SQLJ program using the example above, and run it. Now reverse the order of the columns. Which behavior do you expect? Run the modified program and test your hypothesis.

Exercise 3: (*) Can you use the query SELECT * FROM EMP? Change the example to use this form of SELECT. What behavior do you expect? Run the program and verify your guess.

Is it a good idea to use a wildcard in SELECT statements in a SQLJ program? If yes, why? If no, why not?

Exercise 4: (*) How do you match SQL and Java names? They always match in a case insensitive manner!

(a) Show that the case does not matter by changing the case of the column names in the iterator declaration.

(b) How can you populate a `NamedIter` variable from a query, such as:

SELECT 'BILL', 5000.0 FROM dual

(*Hint:* change the query, use aliases)

(c) Show that the case does not matter, by changing the case of the column names in the SELECT statement. Also show that this is the case with case-sensitive column names.

(d) Which restrictions do you expect on column names in iterator declarations themselves? Show that SQLJ issues an error when these restrictions are violated.

(e) Which restrictions do you expect on column names in SELECT statements? When can SQLJ check these restrictions? Show that SQLJ can issue an error when these restrictions are violated.

Exercise 5: (***) It is rather peculiar that SQLJ always separates the declaration and the population of the iterator object.

```
NamedIter n;
#sql n = { SELECT ename, sal FROM emp };
```

Perhaps, one might prefer to combine declaration and population in a single statement, such as:

```
#sql NamedIter n = { SELECT ename, sal FROM emp };
```

Considering that a SQLJ statement expands into a block of statements, why does SQLJ not support this syntax — what disadvantages would this syntax have?

2.7 GET INTO POSITION!

Declarations for positional iterators are even easier than those for named iterators.

```
#sql iterator PosIter (String, Double);
```

In the processing loop for the positional iterator, you issue FETCH statements to retrieve the next row of data into host variables. After a FETCH, the `endFetch()` call returns `true` if the FETCH was successful, and `false` if there was no row left that could be fetched. Positional iterators require neither the `next()` method nor the accessors that we encountered on the SQLJ named iterator. All of this is best demonstrated by rewriting our earlier example to now use a positional iterator.

```
String name   = null;
Double salary = null;

PosIter p;
#sql p = { SELECT ename, sal  FROM emp };

while (true) {
   #sql { FETCH :p INTO :name, :salary };
   if (p.endFetch()) break;
   System.out.println(name + " would like to make " + (salary * 2));
}
p.close();
```

Warning: Even though it might look unusual, you should always employ the following template when using positional iterators.

```
while (true) {
    #sql { FETCH :p ..... };
    if (p.endFetch()) break;
    ..... process fetched data .....
 }
```

Otherwise, many different things can (*and will*) go wrong!

Warning: You must always `close()` your iterators once you are done using them, or you will run out of cursors to connect to the database with. This is particularly important in the JServer environment.

Ref *SQLJ Developer's Guide and Reference*, Chapter 3, Section "Multi-Row Query Results—SQLJ Iterators" and Section "Overview of SQLJ Declarations" describe iterators.

Ref The demo [Oracle Home]`/sqlj/demo/PosIter.sqlj` contains positional iterators.

Exercise 1: (*) Reverse the order of the columns in the SELECT statement: What happens at translate time? when you run the program? Are you surprised? Explain.

Exercise 2: (*) What do you expect to happen when the SELECT list has fewer columns than required or more columns than required by the positional iterator. Test your hypothesis. Can the SQLJ translator detect this discrepancy? What happens at translate time? at runtime?

Exercise 3: (*) What happens if you move the `endFetch()` test after the `println` statement? Why should you test `endFetch()` before processing the FETCH variables?

Exercise 4: (*) What happens if you use the `endFetch()` test as the test for the while loop condition (`while (!p.endFetch())`)? Are you surprised? Explain.

Exercise 5: (**) Note that the two FETCH variables `name` and `salary` were initialized outside of the `while` loop.
(a) What happens if you do not initialize these variables?
(b) Why does this happen? (You might want to take a peek at the generated Java code.)
(c) What happens to the FETCH variables if the FETCH failed?

Exercise 6: (*) The named iterator declarations use Java types and names, and the positional iterators use only Java types. Does it make sense to mix both kinds in the same declaration? What do you think? What does the SQLJ translator think?

3 A UNIVERSE OF TYPES: SOME SQL, SOME JAVA

JDBC TYPES AND CORRESPONDING SQL TYPES / NULL HANDLING IN SQLJ / SQLJ STREAM TYPES / ORACLE TYPE EXTENSIONS

So far, we just used a bunch of Java types in our SQLJ program, but we really had no clue which types are permitted and how they are used. SQLJ includes all of the types in JDBC —this is described in the next section— and then some.

3.1 WHAT TYPE ARE YOU?

A DESCRIPTION OF JDBC-SUPPORTED TYPES AND HOW THEY ARE USED IN SQLJ.

NUMERIC TYPES. This includes: `int`, `Integer`, `long`, `Long`, `short`, `Short`, `byte`, `Byte`, `boolean`, `Boolean`, `double`, `Double`, `float`, `Float`, and —just so you can see I am not stuttering— `java.math.BigDecimal`. So, what's the deal with supporting both the primitive type (such as `int`, or `double`) and the corresponding Java object type (such as `Integer`, or `Double`)? SQL NULL always maps to Java `null` — and the reverse. Thus, if you read a NULL value into an `Integer`, you receive a Java `null`, but if you read it into an `int`, you can get only a `SQLException`.

CHARACTER TYPES. The Java type `String` represents these very well, thank you. Note that the Java `char` and `Character` types are not supported by SQLJ or by JDBC (besides, they could only hold a single character, anyway). Also useful might be the character streams `sqlj.runtime.AsciiStream` and `sqlj.runtime.UnicodeStream`. We will examine them more closely in Section "3.2 Stumbling On Streams".

DATE AND TIME TYPES. These include `java.sql.Time`, `java.sql.Timestamp`, and `java.sql.Date`. Yes, that is `java.sql.Date`, and not `java.util.Date` — don't confuse the two!

RAW TYPES. Raw data can be represented as `byte[]`, aka "byte-array", or —in stream form— as `sqlj.runtime.BinaryStream`, which we discuss in the next section.

RESULT SETS, CURSORS, AND SO ON. What representation would you expect? `java.sql.ResultSet` and iterator types? Yes, right on the button! A little secret here: using these as host variables is not part of the JDBC specification, but permitted by Oracle.

Did we miss some types here? Yes — let's digress a bit into JDBC history: Java types for several useful SQL entities, such as ROWIDs, BFILEs, BLOBs, and structured types are, unfortunately, not in the JDBC 1.2 specification, which is what JDK 1.1.X uses. However, Java types for these are mentioned in JDBC 2.0, which goes together with JDK 1.2 — go figure! Anyway, this means that in a JDK 1.1 environment these types must be represented through Oracle-specific extensions that we talk about in Section "3.3 She Is An Oracle Type".

SO WHAT ARE THE CORRESPONDING SQL TYPES THAT YOU CAN USE FOR THESE JAVA TYPES?

NUMERIC TYPES. Use any of the numeric SQL types, such as INTEGER, NUMBER(*prec*[,*scale*]), REAL, SMALLINT, and so forth — these are all some form of NUMBER, anyway. Of course, you also must be sure that your Java type can hold the values that you expect in the SQL type and the reverse.

CHARACTER TYPES are CHAR, VARCHAR, VARCHAR2, and LONG.

DATE AND TIME TYPE is DATE.

RAW TYPES are RAW and LONG RAW.

RESULTSET/ITERATOR TYPE is REF CURSOR.

Additionally, SQL, as well as SQLJ with its underlying JDBC driver, perform several implicit conversions. For example, you can retrieve numeric or date values as `String`, or you can insert `Strings` that represent numbers where numeric SQL values are expected.

Warning: JDBC does not enforce retrieving a SQL NULL as a Java `null`, but SQLJ consistently does. You should be aware of this difference in behavior.

Warning: SQLJ (and SQL) perform implicit conversions between SQL and Java types. Although this can be useful, it also might lead to surprising and unexpected behavior. Do not rely on type-checking alone to ensure the correctness of your SQL code.

Ref For more information, see *SQLJ Developer's Guide and Reference*, Chapter 5, Section "Supported Types for Host Expressions", and Chapter 4, Section "Null Handling".

Exercise 1: (*) Show that a SQL NULL is retrieved as Java `null` in an "object wrapper" type, such as `Integer`, but results in a `SQLException` in a primitive type, such as `int`. When does the SQLJ translator detect this situation?

Exercise 2: (*) Write an example program where numeric values are read from or written to the database, using various SQL and Java types. Can you demonstrate loss of precision? In which situations does the SQLJ translator detect potential loss of precision? What JDBC type, if any, can you use if you want to ensure that you do not lose any precision?

Exercise 3: (*) Write an example program where SQL character string values are read from or written to the database. Do the various SQL character types behave differently? Describe your observations.

Exercise 4: (*) Experiment with Java types and SQL types to find out conversions that are performed implicitly. Can you find SQL-JDBC type combinations that are illegal (that is, types between which no conversion is performed)?
What does this mean for type checking between SQL and Java types by the SQLJ translator? Take, for example, a positional iterator that contains a `String` column and an `int` column. What happens, if you flip the corresponding host variables in the FETCH statement? What happens if you flip the corresponding columns in the SELECT statement?

Exercise 5: (*) What is the difference between `java.util.Date` and `java.sql.Date`?

Exercise 6: (***) One of the preceding SQL types can be read only from the database, but not be written to it. What is it? Demonstrate.

3.2 STUMBLING ON STREAMS

The SQLJ specification adds the new stream types `sqlj.runtime.BinaryStream`, `sqlj.runtime.AsciiStream`, and `sqlj.runtime.UnicodeStream` for "wrapping" a LONG (or LONG RAW) column in the database. All three stream types implement a `java.io.InputStream`. Note that when you retrieve the value of a LONG column, all data that occurs in the same row prior to that column is lost. This has a number of consequences.

Warning: When you use FETCH and a positional iterator, you can only have a single stream column, and this must be the last column of the iterator.

Warning: When using a named iterator, you must access the stream columns in sequence. Whenever you access a column that comes after the stream column, the data in the stream column is lost.

Ref See *SQLJ Developer's Guide and Reference*, Chapter 5, Section "Support for Streams".

Exercise 1: (*) Write a named iterator with two stream columns. Access the streams columns out of order. What happens? Show that the order in the SELECT statement must be obeyed, not the order in the iterator declaration.

Exercise 2: (**) What would happen if you permitted more than one stream column in a positional iterator and —consequently— in a FETCH statement? Why do the restrictions mentioned in the Warning not apply if you use `byte[]` instead of streams for retrieving LONGs?

3.3 SHE IS AN ORACLE TYPE

If you are willing to go out with an Oracle Type, we have a whole roster of them for you to choose from. They all live in the same place: `oracle.sql`. You might want to call up *JDBC Developer's Guide and Reference*, Chapter 4 "Oracle Extensions" for these types, rather than going on a blind date. Or, if you are more adventurous, perform a quick background check with `javap oracle.sql.Xxxxx` and then give `Xxxxx` a whirl!

`oracle.sql.ROWID` - represents a database ROWID.

`oracle.sql.CLOB` - represents a CLOB, a large character object.

`oracle.sql.BLOB` - represents a BLOB, a large binary object.

`oracle.sql.BFILE` - represents a BFILE, a binary file.

`oracle.sql.REF` - represents a REF, a reference to a structured object. See Section "5.5 Let's Get Objective".

`oracle.sql.ARRAY` - represents a VARRAY or a nested table. See Section "5.5 Let's Get Objective".

`oracle.sql.STRUCT` - represents a user-defined structured object. See Section "5.5 Let's Get Objective".

"Wait a minute!" you are now going to say. "All these types (with the exception of ROWID) look rather familiar — I remember now, these are JDBC 2.0 types (`java.sql.Clob/Blob/Bfile/Ref/Array/Struct`)." Right on the money! If you run SQLJ under JDK 1.2 (with the Oracle JDBC classes `classes12.zip`), you can also use these JDBC types instead of the Oracle types. In fact, all these Oracle types implement the interface of their corresponding JDBC type.

The next set of types represent efficient "wrappers" that preserve the binary representation of data in the database. These types are ultra-fast when exchanging information with the database, because they require zilch conversion effort. And they represent the information completely faithfully down to the last original bit. You can use these types the same way you use their corresponding JDBC cousins. Note that these types are also endowed with conversions methods and constructors involving the Java-native types that we talked about earlier.

`oracle.sql.NUMBER` - represents all numeric SQL data.

`oracle.sql.CHAR` - represents all character data in SQL.

`oracle.sql.DATE` - represents all date and time data in SQL.

`oracle.sql.RAW` - represents all raw data in SQL.

Ref See *SQLJ Developer's Guide and Reference*, Chapter 5, Section "Oracle Type Extensions".

Exercise 1: (*) What's with this `oracle.sql.ROWID`? Show that you can retrieve the ROWID in a SELECT statement, and subsequently employ the `oracle.sql.ROWID` in the WHERE clause of an INSERT, UPDATE, or DELETE statement.

Exercise 2: (*) Think big! Use one or more of the `oracle.sql.BLOB/CLOB/BFILE` types in an example.

Exercise 3: (**) Pick one or more of the `oracle.sql` "wrapper" types and compare them with the JDBC "native" types. Can you show efficiency savings due to skipped conversion? What happens if you want to manipulate values of these types? Describe the tradeoff.

Exercise 4: (***) If you are running JDK 1.1, you cannot use the `java.sql.Blob/Struct` and so on types. Why not?
However, in Oracle JDBC you can use `oracle.jdbc2.Blob/Struct` etc. types instead. (Note: SQLJ does *not* support types in `oracle.jdbc2`.) If your application uses `oracle.jdbc2` types, you must recompile it if you want to run it under JDK 1.2. Why?

4 YOUR OWN PRIVATE TRANSLATOR

HELPFUL SQLJ OPTIONS / DEBUGGING OPTIONS: -LINEMAP AND -P-DEBUG / PROGRAMMING SQLJ APPLETS

This chapter describes features of the SQLJ translator itself. However, we will only cover the fun parts here. For the full story about basic translator features, see *SQLJ Developer's Guide and Reference*, Chapter 8, Section "Basic Translator Options", and if —by golly— you want the advanced SQLJ translator features, too, you must see the doctor in *SQLJ Developer's Guide and Reference*, Chapter 8, Sections "Advanced Translator Options" and "Translator Support and Options for Alternative Environments".

4.1 THE TRANSLATOR IS TALKING BACK

So you are stuck, and you want to get help. Do not fret — the SQLJ translator might just be able to give you the assistance you need.

`sqlj -help` - gives you a short help message with the main SQLJ command line options. Additionally, whenever you just say `sqlj` without other arguments, this is interpreted as a cry for help.

`sqlj -help-long` - gives a really long message. This is most useful if you want to figure out which command line options the translator is actually using and where they come from. In this case, you might want to add your original command line as well. Or you can just check on some obscure translator option.

`sqlj -version-long` - shows you the SQLJ translator version, as well as the version of your Oracle JDBC driver and your Java VM. If the JDBC driver has version 0.0, you know that you need to install it and put it in your `CLASSPATH`!

`sqlj -status` … - add the `-status` flag to your command line if you want to be entertained during those really long translations/compiles. The SQLJ translator will make sure that it regularly gets back to you to let you know what it is up to at the moment.

`sqlj -explain` … - add this option to your command line if you get one or more of these !@#$!% (incomprehensible) error or warning messages. The SQLJ translator will look up the Cause and/or Action explanation for the message in the SQLJ manual and print it out, just for you. Isn't that a nice touch!

`sqlj -n` … - add this if you want to see what is actually invoked by the SQLJ wrapper script/wrapper executable. Or just use it to amaze yourself that you can use pretty much the same command line options you have grown to know and love in `javac` and in `loadjava`. And you can even see how they look in SQLJ-Translatorese. This option shows what would have been invoked, but does not run `sqlj` for real — just as `make -n` does.

Ref For a general overview of SQLJ options, see *SQLJ Developer's Guide and Reference*, Chapter 8, Section "Basic Translator Options".

Exercise 1: (*) Try all the options above. Have fun!

4.2 WHERE IS THE BUG?

We trust you will have noticed that error messages issued by your Java compiler on code that originates from a SQLJ file are reported on the SQLJ file, and not on the generated Java file. However, when your program throws exceptions at runtime, line numbers (such as those issued by `printStackTrace()`) are shown in terms of the generated Java files.

You knew, of course, that we'd have a cure for that problem, too. Just add the flag `-linemap` to your command line during translation. Then the translator will fix up the file names and the line numbers in those class files that were compiled from original SQLJ files.

If you now pick up Sun's Java debugger `jdb` to debug your SQLJ program, you'll see that … it does not work: `jdb` refuses to show the SQLJ source. No wonder — they only taught it about `.java` source files! Okay, so we give you another magic command line, spell: `-jdblinemap`, to be used instead of `-linemap` whenever you must trick that silly little (de)bugger.

If you must trace how your SQLJ program talks with the database, you can install a *profile auditor* in the SQLJ profile files (those pesky little `.ser` files that hold the static SQL part of your SQLJ program and that we first encountered in Section "1.1 I Need To Tell You"). After the usual SQLJ translation and compilation, tracing can be added by issuing the following command.

```
sqlj -P-debug *.ser
```

The above assumes that all the generated `.ser` files are in the current directory.

At the end of the day, SQLJ runtime calls turn into calls to Oracle's JDBC runtime. You can trace these, too with the following call to the JDBC API: `java.sql.DriverManager.setLogStream(System.out)`.

Ref *SQLJ Developer's Guide and Reference*, Chapter 8, Section "Basic Translator Options" - "Reporting and Line Mapping Options" describes the `-linemap` and `-jdblinemap` options.

Ref The profile auditor is explained in *SQLJ Developer's Guide and Reference*, Appendix A, Section "AuditorInstaller Customizer for Debugging".

Exercise 1: (**) Create `sqlj` programs with various bugs in the (non-SQLJ) Java code. What kinds of bugs are reported by the SQLJ translator? What kinds of bugs are reported by the Java compiler?

Exercise 2: (*) Show that stack traces in SQLJ programs refer to the generated Java file. Then use the `-linemap` option, and convince yourself that the stack trace refers to the original SQLJ file.
Debug your SQLJ programs with the `jdb` debugger. *Note:* make sure to use `-jdblinemap`, instead of `-linemap`.

Exercise 3: (*) Perform a trace of a SQLJ program with the profile auditor. Now also add JDBC tracing. Do the same for a multithreaded SQLJ program. *Note:* Make sure to use the `-P-CshowThreads=true` flag together with the `-P-debug` Auditor installer.

4.3 APPLETMANIA

Do not walk, but run, to the [Oracle Home]`/sqlj/demo/applets` directory. View `index.html` in your browser, and then click on the `Applet.readme` link, and do everything in it. See ya!

Already back? Now read the `AppletOracle.readme` file, and do what it says.

Congratulations — you are an applet expert! We just summarize the main gotcha's for you.

Warning: Use the `-ser2class` flag to convert serialized profile files `.ser` to classfiles — some browsers cannot handle `.ser` files.

Warning: Use the `-d` option to place all `.class` and `.ser` files (if any) in a directory hierarchy, which you can then easily archive later.

Warning: If you do not use the Java plugin when browsing the applet, you must specify the `-profile=false` flag to the SQLJ translator, and you cannot use the SET statement or Oracle specific types (including the use of iterators and result sets as parameter or column types).

Warning: If you do use the Java plugin, make sure that your `CLASSPATH` is empty before you start your browser.

Warning: We strongly recommend that you use explicit connection contexts on your applet's SQL statements. See Section "5.2 Being Well Connected — Explicitly" for details.

Ref You can find the applet demos at [Oracle Home]`/sqlj/demo/applets/`

Exercise 1: (*) Oracle-specific types (such as `oracle.sql.Xxxx`), and certain SQLJ constructs (such as `SET :x =` ...) require Oracle customization, that is `-profile=true` (the default). What happens if you write such a program, but set `-profile=false` during SQLJ translation?

Exercise 2: (**) Show that Netscape 4.X does not like `.ser` files.

5 THE REST OF THE STORY — ADVANCING THE FEATURES

SQLJ-JDBC INTEROPERABILITY / USING CONNECTION CONTEXT INSTANCES / USING EXECUTION CONTEXTS / USING TYPED CONNECTIONS / USING ORACLE OBJECTS / SQLJ IN JSERVER / PORTABILITY / PERFORMANCE / ITERATOR SUBCLASSING

This chapter delves into a few of the advanced SQLJ features - but not very deeply. We'll look at how to mesh dynamic SQL with SQLJ, at the mysteries of *connection contexts* and *execution contexts*, and we offer encouragement for those who want to start using Oracle Objects with SQLJ as well as for those who want to program with SQLJ in JServer. We consider how to write portable and performant SQLJ programs. The final SQLJ gem that we introduce in this primer is iterator subclassing.

5.1 A DYNAMIC PROGRAM

SQLJ works just fine and dandy with static SQL — where you know the shape of SQL statements and queries beforehand, and only the actual data that is passed to (or from) the database varies. Now imagine that you must write a program that can make up the WHERE clause of a SELECT on the fly. Guess you'd better forget all about SQLJ, right?

Nope — you can still use SQLJ! SQLJ and JDBC are close-knit buddies: JDBC connections and SQLJ connection contexts are mutually convertible, and so are java.sql.ResultSets and SQLJ iterators. Let's look at the specifics.

CONNECTING FROM JDBC TO SQLJ. All connection context constructors and initializers can take an existing JDBC connection. Example:

```
java.sql.Connection conn = DriverManager.getConnection(....);
Oracle.connect(conn);
```

Now SQLJ and JDBC share the same session.

Warning: When you set or create a SQLJ connection from a JDBC connection, you will inherit all the properties of the JDBC connection, including the auto-commit setting. Remember, JDBC auto-commit is off by default, SQLJ auto-commit is either on by default, or it has to be declared explicitly.

CONNECTING FROM SQLJ TO JDBC. All SQLJ connection contexts have the `getConnection()` method, which allows you to retrieve the underlying JDBC connection. What? You say, you do not know how to get the SQLJ connection context that you set with `Oracle.connect(....)`? Of course, you don't! Because I have not yet told you how. Now squint your eyes at the lines below.

```
java.sql.Connection conn =
    sqlj.runtime.ref.DefaultContext.getDefaultContext().getConnection();
```

As you can tell, the monster expression `sqlj.runtime.ref.DefaultContext.getDefaultContext()` gives you the value of the static(!) SQLJ default context.

PASSING RESULT SETS FROM JDBC TO SQLJ. We want to pass off a JDBC result set as a SQLJ iterator. Well, the SQLJ standards committee, in their wisdom, agreed that it was not sufficient to just construct a SQLJ iterator instance from a JDBC `ResultSet`. You must assign it explicitly with a SQLJ CAST statement.

```
SomeIterator iter;
java.sql.ResultSet rs = stmt.executeQuery();
#sql  iter = { CAST :rs };
```

PASSING ITERATORS FROM SQLJ TO JDBC. This one is a breeze. You just call the iterator's `getResultSet()` method and – voilà— your JDBC `ResultSet`.

Warning: The result set-to-iterator and the iterator-to-result set conversions must be made before the `next()` method is called, or a FETCH is performed on the original result set, or —respectively— iterator.

Ref *SQLJ Developer's Guide and Reference*, Chapter 7, Section "SQLJ and JDBC Interoperability" describes SQLJ-JDBC interoperability.
Ref An application demonstrating interoperability is at [Oracle Home]`/sqlj/demo/JDBCInteropDemo.sqlj`.

Exercise 1: (*) Demonstrate that you can convert a JDBC connection to a SQLJ connection and the reverse. Should it be possible to cascade several conversions, such as JDBC to SQLJ to JDBC to SQLJ? Why, or why not?

Exercise 2: (*) Demonstrate that you can convert a JDBC result set into a SQLJ iterator and the reverse.

Exercise 3: (**) Why is it useful to convert a JDBC result set into a SQLJ iterator? Give an example!

Exercise 4: (**) Can you also come up with a scenario where you would want to be able to convert SQLJ iterators into JDBC result sets?

Exercise 5: (**) According to the SQLJ specification, the behavior is undefined if you fetch results before you convert between result sets and iterators.
(a) Why was the behavior not prescribed in this case? What actual behavior do you see when you use Oracle SQLJ?
(b) What would you expect if you convert back and forth several times between iterators and result sets? What actual behavior do you see when you use Oracle SQLJ?

Exercise 6: (*)Another way to write dynamic SQL in SQLJ is to use PL/SQL! By starting a SQLJ statement with `BEGIN` (or with `DECLARE`) you get immediate access to the PL/SQL engine and to its EXECUTE IMMEDIATE syntax (just make sure to add the closing `END`). Demonstrate writing dynamic SQL statements in this way. *Note:* In the demo directory, you can find a solution `DynamicDemo.sqlj`.

5.2 BEING WELL CONNECTED - EXPLICITLY

Up until now you have been brainwashed. We made you believe that there is only a single, static connection in your SQLJ program, that you set once with `Oracle.connect(....)` and then forget about. Though this helps sufferers of carpal tunnel syndrome (never type an explicit connection!) and makes great copy for SQLJ marketers ("look how short SQLJ programs are"), this is not the way the world works.

- If you are connecting to more than one schema, or
- if you are running an applet in a browser, or
- if you are connecting to the database in a multithreaded program,

then you should, nay, you *must* use explicit SQLJ connections.

Don't worry — you'll learn all about explicit connections in a jiffy. The most bland SQLJ connections are called `sqlj.runtime.ref.DefaultContext`, and we parade them next.

```
import sqlj.runtime.ref.DefaultContext;
…
DefaultContext ctx1 =
      new DefaultContext("jdbc:oracle:oci8:@", "scott", "tiger", false);
DefaultContext ctx2 = new DefaultContext(aJdbcConnection);
#sql [ctx1] { UPDATE emp SET sal = sal / 2 };
#sql [ctx2] { UPDATE emp SET sal = sal * 2 };
```

You see, you can specify explicitly which connection your SQLJ statements are supposed to use — just put the connection context instance (or an expression evaluating to one) in those square brackets: `[`*context*`]`. If you do not do this, your statement will use the default context, which you have been setting all along with `Oracle.connect()`.

Note that the connection context constructor `DefaultContext()` supports the same signatures as `java.sql.DriverManager.getConnection()`, with an additional `boolean` argument at the end that specifies whether auto-commit is on or off. Additionally, you can create a new context from a JDBC connection or (not shown) from another SQLJ context — this, of course, inherits both the session, as well as the session's auto-commit setting from that connection.

Let's digress once more with a little JDBC/SQLJ background. In JDBC, auto-commit is by default on. This was considered a rather dorky default setting by the SQLJ proponents (which mostly come from big-database companies). Rather than having a default setting opposite to JDBC, the auto-commit on the SQLJ context must be specified explicitly. Still not satisfied, Oracle is providing the `Oracle.connect()` API that turns auto-commit off by default (although it also supports the extra `boolean` at the end for setting it explicitly).

Warning: Be a good citizen and do not follow the bad example we have been giving here: Always specify the auto-commit setting explicitly, whether you are using `Oracle.connect()`, `new DefaultContext()`, or your own connection context type. Thanks! We'll be eternally grateful.

Warning: Always use explicit connection contexts, unless you know that your program owns the world and requires only a single, static database connection.

Ref See *SQLJ Developer's Guide and Reference*, Chapter 4, Section "Connection Considerations".

Exercise 1: (*) Turn the preceding code fragment into a complete sample program, and run it. *Note:* The `Oracle.connect()` API explicitly loads and registers the Oracle JDBC driver. If you are using `sqlj.runtime.ref.DefaultContext`, you must perform this task yourself. Can you also demonstrate the different default settings for auto-commit?

Exercise 2: (*) The Oracle way to obtain an explicit SQLJ connection context (of type `sqlj.runtime.ref.DefaultContext`) is called `Oracle.getConnection(…)`. Demonstrate its use.

Exercise 3: (***) Why is it a bad idea to use a single, static connection context in applets or in multithreaded programs? Can you write a sample program that demonstrates the problem?

5.3 WRAPPING UP UPDATES

Whenever we executed a SQLJ statement, we either obtained data through host variables or by assignment, or we experienced a `SQLException` containing some error message. At times we would like to obtain additional information about the SQL statement, such as the following.

- The statement might result in a warning (not an exception) that we want to inspect.
- We are interested in the number of rows that was changed (or removed) by an UPDATE or DELETE statement.

This information is available on a `sqlj.runtime.ExecutionContext`. An example.

```
import oracle.sqlj.runtime.Oracle;
import sqlj.runtime.ref.DefaultContext;
import sqlj.runtime.ExecutionContext;
…
Oracle.connect(aJdbcConnection);
DefaultContext ctx = new DefaultContext("jdbc:oracle:oci8:@", "scott", "tiger",
false);

#sql { DELETE FROM emp WHERE sal > 5000 };
#sql [ctx] { UPDATE emp SET sal = sal * 2 };

ExecutionContext ec1 = DefaultContext.getDefaultContext().getExecutionContext();
System.out.println( ec1.getUpdateCount() + " employees are laid off.");

ExecutionContext ec2 = ctx.getExecutionContext();
System.out.println( ec2.getUpdateCount() + " employees are rejoicing.");
```

Ref See *SQLJ Developer's Guide and Reference*, Chapter 7, Section "Execution Contexts".

Ref Also consider the demo [Oracle Home]`/sqlj/demo/MultiThreadDemo.sqlj`.

Exercise 1: (*) How long is the update count available on the execution context?

Exercise 2: (*) What value does `getUpdateCount()` return when a `SQLException` occurred during execution of the SQL statement?

Exercise 3: (*) You can create an `ExecutionContext` instance (using the empty constructor), and pass it explicitly to the SQLJ statement, between the square brackets that usually hold execution contexts. Show that SQLJ will use the explicitly passed execution context, rather than the execution context that can be accessed from the connection context.

Exercise 4: (**) You can also use `ExecutionContexts` to explicitly set properties for the execution of SQL statements — refer to the SQLJ runtime documentation. Demonstrate some of the functions, such as `setMaxRows(int max)` — the maximum number of rows fetched in a SELECT statement, and `setQueryTimeout(int seconds)` — the maximum permitted time for executing a query.

Exercise 5: (**) How would you have to write a multithreaded application that is only allowed a single static connection context, but where each thread performs UPDATES independently?

5.4 WHAT TYPE IS YOUR CONNECTION?

Consider the following scenario: you want to write a SQLJ program that establishes connections to two different database schemas. In the PILOTS schema you keep personal data, schedules, flight hours, and so on of fighter pilots, and in the JETS schema you have the technical specs, the repair history, and maintenance schedule of fighter aircraft.[1] Naturally, the SQLJ statements that operate on each of these schemas will utilize different tables, views, and different sets of stored procedures and functions. It would be nice if you could verify the correctness of the SQL statements in your program against both of these schemas. So far, we explained to you only how to specify a single database connection for online checking.

This is where the notion of typed connection contexts comes in. We create two different context types.

```
#sql context Pilots;
#sql context Jets;
```

At runtime, you must connect to the corresponding schema — unfortunately, SQLJ cannot check that for you.

```
Pilots pconn = new Pilots("jdbc:oracle:oci8:@","PILOTS","ACE",false);
```

---

[1] Oracle's lawyers think it's a bad idea to use any product in "nuclear … or otherwise inherently dangerous applications." Of course, we are talking about some multi-player Internet game application here, *caprice*?

```
Jets jconn  = new Jets("jdbc:oracle:oci8:@","JETS","STRATOS",false);
```

However, at translate time, SQLJ can determine whether you used a `Pilots` connection context or a `Jets` connection context in your SQLJ statement.

```
#sql [pconn] { INSERT INTO pilot VALUES ( …. ) };                 // Pilots context
#sql [jconn] { UPDATE maintenance SET status = Checkup( …. ) };  // Jets context
```

Can we tell SQLJ at translate time how to connect to the database for these connection context types? Sure — that's easy!

```
sqlj -user@Pilots=pilots/ace -user@Jets=jets/stratos MyFile.sqlj
```

Ref See *SQLJ Developer's Guide and Reference*, Chapter 7, Section "Connection Contexts".
Ref The corresponding demo is [Oracle Home]`/sqlj/demo/MultiSchemaDemo.sqlj`.

Exercise 1: (*) What happens if you omit the setting `-user@Jets=jets/stratos`?
What happens if you replace `-user@Jets=jets/stratos` with `-user=jets/stratos`?
What happens if you replace both preceding `-user` settings with `-user=pilots/ace`?

Exercise 2: (***) Could we achieve the same functionality without introducing connection context types? Why or why not? Explain.

5.5 LET'S GET OBJECTIVE

Now what about those SQL objects, the SQL REFs and VARRAYS/Nested Tables? Yes, SQLJ supports all these features of Oracle SQL. You can use the "raw" representations (`oracle.sql.STRUCT/REF/ARRAY`) for these types — at least for receiving values from the database.

However, for full functionality, you should use JPublisher wrapper classes for these SQL types. This is a four-step process.

1. Create your SQL type in the database — for example ADDRESS.
2. Use the JPublisher to create a Java wrapper class for the SQL ADDRESS type in the file `Address.java`. For each attribute ATTR in the SQL type, the `Address` class will contain `setAttr()` and `getAttr()` setter/getter methods.
   ```
   jpub -sql=Address -user=scott/tiger -url=jdbc:oracle:oci8:@
   ```
3. Compile `Address.java`.
   ```
   javac Address.java
   ```
4. Now you can use the Java Address class, as if it were a built-in type that you can receive from the database, or send to the database. You just compile and run your SQLJ program normally.
   ```
   sqlj MyFile.sqlj
   ```

Easy, isn't it? Of course, there are a whole lot of additional details that we omitted. Get the full picture from the following sources.

Ref *SQLJ Developer's Guide and Reference*, Chapter 6 "Objects and Collections".
Ref Technical Whitepaper "Using Oracle Objects in SQLJ Programs" at [Oracle Home]`/sqlj/doc/sqlj-objects.pdf`.
Ref *JPublisher Developer's Guide and Reference.*
Ref Demo files in [Oracle Home]`/sqlj/demo/Objects/` and [Oracle Home]`/sqlj/demo/jpub/`.

Exercise 1: (*) Try out the objects features using the demos provided in the `Objects` and `jpub` directories.

Exercise 2: (**) Both JPublisher and SQLJ also support the JDBC 2.0 `java.sql.SqlData` interface. Transform some of the demos written using the `CustomDatum` interface to `SqlData`. What are the limitations of using `SqlData`? What are the advantages?

Exercise 3: (**) JPublisher provides a strong type association between a (generated) Java class and a SQL type. There are several Java classes, such as `oracle.sql.STRUCT`, `REF`, and `ARRAY` (as well as their JDBC 2.0 counterparts: `java.sql.Struct`, `Ref`, and `Array`) that you can use instead of the JPublisher-generated classes. What limitations would you expect when using these "generic" wrapper classes? Consider retrieving a value in a SELECT column, passing it as an IN parameters, and receiving it as an OUT parameters.
Now write SQLJ code to discover actual limitations. Why do these exist?

5.6 STUFFING SQLJ INTO THE SERVER

No way, José! We do not have space or time to explain this here. If you used the `Oracle.connect()` method and a default context on your SQL statements, you should be in good shape to debug and test your SQLJ program on the client. Then you "stuff it" into the server using the following steps (naturally, variations on this theme abound) .

1. Compile and `jar` your program (client-side) for upload to the server.
2. Load your program into the server with `loadjava`.
3. Publish one or more public static methods of your application as SQL stored procedures or functions.
4. Run your program from SQL, that is from a SQL environment, or from a SQLJ/JDBC or any other client.

Here we just pass a few tips along that relate to the client-side compilation Step 1.

Warning: Use the `-ser2class` option to convert `.ser` files into `.class` files. This usually makes the JServer much happier! Use the `-d` option to designate a root directory under which all the SQLJ-generated `.class` (and possibly `.ser`) files will be placed.

```
sqlj -ser2class -d rootdir *.sqlj *.java
```

Copy any `.properties` files that you require into the appropriate location under *rootdir*.
Change to *rootdir* before issuing `jar cvf0` *myjar*`.jar *`, then upload *myjar*`.jar` with `loadjava.`

Ref An entire manual *Java Stored Procedure Developer's Guide* is devoted to this topic.
Ref *SQLJ Developer's Guide and Reference*, Chapter 11 "SQLJ in the Server" discusses SQLJ-specific aspects in depth.
Ref You can find demo files in [Oracle Home]`/sqlj/demo/server/`.

5.7 ISN'T IT — PORTABLE!

Now that we have introduced all these great Oracle features — how can we get rid of them and put the (Oracle)Genie back in its bottle? First off, you want to turn on portability warnings, so you get notified about Oracle-specific type usage.

```
sqlj -warn=portable …
```

Note that some types, such as iterators and result sets, are not Oracle specific, per se, but the standard SQLJ driver does not let you use them as parameters or in iterator columns. You'll see warnings in these cases, as well. Secondly, you should avoid the use of Oracle-specific SQL constructs such as PL/SQL — you need some kind of standard SQL grammar in your SQLChecker component. There is good news and bad news on this. The good news is that we have a demo in [Oracle Home]`/sqlj/demo/components` with new SQL checkers (`ParsingJdbcChecker` and `ParsingOfflineChecker`) that use an actual SQL grammar to check the syntax of your SQL statements. Even better, you get the source for this grammar and can modify it to your heart's content. On the downside, this checker is not part of the Oracle SQLJ product and, therefore, unsupported.

Ref See *SQLJ Developer's Guide and Reference*, Chapter 8 "Translator Command Line and Options", Section "Basic Translator Options" - "Reporting and Line Mapping Options".

Exercise 1: (*) Go to [Oracle Home]`/sqlj/demo/checker/components` and kick the tires of the parsing checker.

Exercise 2: (*) Write some Oracle-specific code and test the `-warn=portable` flag and the parsing checker on it.

Exercise 3: (**) There is one set of types that are not Oracle-specific, but you will still see portability warnings about them. Which ones? Why? *Hint:* the SQLJ 8.1.6 runtime is built under JDK 1.1.

Exercise 4: (***) Improve the SQL grammar. (a) Rewrite the grammar. (Yes, it was a quick hack.) (b) Improve error recovery and error messages. (c) Make it recognize SQL-92 Entry Level only. (d) Make it recognize your favorite flavor of SQL.

5.8 GIVE ME SPEED OR … MORE SPEED!

You say you want speed, speed, and more speed? It's coming to you in the 8.1.6 SQLJ release. And you can pick it up with little or no effort — compared to using JDBC. Buckle up, as we put the pedal to the metal! In the examples below we assume that your program uses only the default connection context. Otherwise, any required `ExecutionContext` and JDBC `Connection` objects will have to be obtained from your actual connection context instance.

SPEED THROUGH STATEMENT CACHING. By default, SQLJ automatically caches the last five SQLJ statements that you executed on a given connection. Execution time is cut up to 50% if a statement can be pulled from the cache. If you like to, you can also set a specific statement cache size during SQLJ translation or profile customization as follows.

```
sqlj -P-Cstmtcache=cacheSize …
```

Now what does a *cacheSize* of 0 do? Yep, it turns off the cache! Use it to see how your program performed under the previous versions of SQLJ. Or, more sensibly, set it to a higher value.

SPEED THROUGH BATCHING. SQLJ now also supports batching through the `setBatching()` and `setBatchLimit()` methods on the `ExecutionContext`..

```
ExecutionContext ec = DefaultContext.getDefaultContext().getExecutionContext();
ec.setBatching(true);
```

This turns on batching of INSERT, DELETE, and UPDATE statements in your SQLJ program. If the same DML statement is executed repeatedly —such as in a loop— the parameter bindings are collected. Finally, when execution moves to a different SQLJ statement, the collected set of parameters is bound through an array bind and executed as a single statement.

Speed through Row Prefetching. You can also set a row prefetch size for your queries (the default size is 10) to save on round trips to the database.

```
OracleConnection conn = (OracleConnection) DefaultContext.getContext().getConnection();
conn.setDefaultRowPrefetch(prefetchSize);
```

Speed through Hints for Variable-Size Parameters. If you use variable-size SQL types, such as CHAR, VARCHAR, or RAW as bind parameters in your SQLJ statements, then JDBC has to prepare for the worst case (such as a PL/SQL function returning 32kB of character data). Often, you know the actual maximum size (in bytes!) of these parameters ahead of time and can give a hint to the underlying JDBC engine. An example.

```
    #sql s = { /*(10)*/ VALUES( to_uppercase(:t/*(10)*/) ) };
```

The size hint is always given as a comment `/*(`*size*`)*/` immediately after the host variable (or host expression), or —if this is the return value for a function— as the first comment in the SQL statement. You still have to tell SQLJ at translation or at customization time to pick up these hints through the `-P-Coptparams` flag.

```
    sqlj -P-Coptparams …
```

If you use a specific statement cache size and parameter size hints, you must specify both flags at the same time.

Warning: A fixed, limited number of statement cursors is available to your SQLJ program per JDBC connection - *not* per SQLJ connection context. The SQLJ statement cache effectively reduces the number of statement cursors available to JDBC.

Warning: If you turn batching on, also set an upper batch limit with `ec.setBatchLimit(`*size*) to not run out of memory.

Ref See *SQLJ Developer's Guide and Reference*, Appendix A "Performance and Debugging", Section "Performance Enhancement Features".
Ref For demos, see [Oracle Home]`/demo/PrefetchDemo.sqlj` and [Oracle Home]`/sqlj/demo/performance/`.

Exercise 1: (*) Run the performance demos. Which of the optimizations appears to offer the most improvement?

Exercise 2: (*) Write a program that will constantly overflow a five-element statement cache. Now demonstrate caching by increasing the cache size.

Exercise 3: (**) Write a program that shows that JDBC and SQLJ use statement handles from the same underlying connection.

Exercise 4: (*) What happens when the execution of a batched statement results in an error? When will you detect this? Demonstrate.

Exercise 5: (*) If you have several different DML statements in a loop, batching will not work. Can you give a workaround for this? Demonstrate.

Exercise 6: (*) Show what happens when the parameter size hints in your SQLJ program are exceeded. Read up on the `-P-Coptparams` and `-P-Coptparamdefaults` flags, and demonstrate the use of default parameter hints.

5.9 Classy Kinds Of Iterators

SQLJ iterator types have some "object flavor" but do not feel like genuine objects. You might wish that you could endow an iterator with a different character from that which SQLJ generates. Roll your own by subclassing an iterator type and providing your own (add-on) behavior! In the following example, we assume that we already have an `Emp` class.

```
    #sql iterator Iter(String ename, int empno);
    class SubIter extends Iter {
       SubIter(sqlj.runtime.profile.RTResultSet rs) throws SQLExecption { super(rs); }
       Emp getEmp() { return new Emp( ename(), empno() ); }
    }
    …
    SubIter iter;
    #sql iter = { SELECT * FROM EMP };
    while (iter.next()) { System.out.println(iter.getEmp()); }
```

Note the constructor *Subclass*`(sqlj.runtime.profile.RTResultSet)` that plugs into the corresponding superclass.

Exercise 1: (*) Play with the example in [Oracle Home]`/sqlj/demo/SubclassIterDemo.sqlj`. What are some interesting new behaviors for iterators? Write more examples.

Exercise 2: (***) Write a program that generates an iterator declaration together with a subclass that exhibits useful behavior, such as retrieving the entire result set as a collection, providing typed row objects, providing updateable typed row objects, and so on.

YOU HAVE NOW BECOME A SQLJ GURU

You made it. It's time to get a life and have some fun!