Imports

import java.sql.SQLException;
import sqlj.runtime.ExecutionContext;
import sqlj.runtime.ConnectionContext;
import sqlj.runtime.ResultSetIterator;
import sqlj.runtime.ref.DefaultContext;
import oracle.sqlj.runtime.Oracle;

Fundamental SQLJ Syntax
Statements

#sql { ...*SQL code*...}; // Use static default connection
#sql [*ctx*] {...*SQL code*...}; // Use context instance *ctx*

Bind Variables, Bind Expressions

#sql {... :*inVar* ... :OUT *outVar* ... :INOUT *inOutVar* ...};
#sql {.. :(*inExp*) .. :OUT (*outExp*) .. :INOUT (*ioExp*) ..};

Declarations

#sql *modifiers* context *ContextName*;
#sql *modifiers* iterator *IteratorName* (...);
Must be placed where classes may legally be declared. Use
public static modifiers for inner class declarations.

Connections

Static Default Connection

Oracle.connect(*Oracle_URL*, *user*, *password*);
...
Oracle.close();

This is equivalent to the following.
java.sql.DriverManager.registerDriver
      (new oracle.jdbc.driver.OracleDriver());
java.sql.Connection *conn* =
   java.sql.DriverManager.getConnection
            (*Oracle_URL*, *user*, *password*);
conn.setAutoCommit(false);
DefaultContext.setDefaultContext
            (new DefaultContext(*conn*));
...
DefaultContext.getDefaultContext().close();

Other Connect Signatures
(*Oracle_URL_with_user_and_password*),
(*JDBC_Connection*), (*SQLJ_context*),
(*Class_denoting_package*, "*properties_file_name*"), ...

Some Oracle URLs

"jdbc:oracle:thin:@*hostname*:*port*:*database_SID*"
     - thin JDBC driver
"jdbc:oracle:oci8:@*tns_alias_name*"
     - JDBC-OCI driver
"jdbc:oracle:kprb:" - server-side JDBC driver

Connection Context Instance *ctx*

Use DefaultContext or declare a context class
#sql context *MyCtx*; // **public static** for an inner class

Create a context instance

MyCtx ctx =   // last argument specifies autocommit
      new *MyCtx*(*Oracle_URL*, *user*, *password*, false);
...
ctx.close(); // close(false) to keep JDBC connection

Use a DataSource
// autocommit is inherited from the data source
#sql context *DSCtx*
      with (dataSource="*JNDI_name_of_data_source*");
DSCtx ctx = new *DSCtx*(*user*, *password*);
...
ctx.close(); // close(false) to keep JDBC connection

JDBC Interoperability

java.sql.Connection *conn* = DefaultContext
            .getDefaultContext().getConnection();
// or
conn = *ctx*.getConnection();
// and (note that autocommit is inherited from *conn*)
MyCtx ctx = new *MyCtx*(*conn*);

Queries
Single-Row Select

#sql { SELECT FROM *tab*
                  INTO :x_1, :x_2, ... WHERE ...};

Named Iterators

#sql iterator *NamedIter* (String col_1, int col_2, ...);
...   // use **public static** for inner class
NamedIter ni;
#sql *ni* = { SELECT col_1, col_2, ...
                        FROM *tab* WHERE ...};
while (*ni*.next()) {
   ...process... *ni*.col_1() ... *ni*.col_2() ...
}
ni.close();

Positional Iterators

#sql iterator *PosIter* (String, int, ...);
...   // use **public static** for inner class
PosIter pi; String x_1=*null*; int x_2=0; ...
#sql *pi* = { SELECT col_1, col_2, ...
                        FROM *tab* WHERE ...};
while (true) {
   #sql { FETCH FROM :*pi* INTO :x_1, :x_2, ...};
   if (*pi*.endFetch()) break;
   ...process...x_1..x_2...
}
pi.close();

ResultSet Iterators

ResultSetIterator *rsi*; String x_1; int x_2; ...
#sql *rsi* = { SELECT col_1, col_2, ...
                        FROM *tab* WHERE ...};
while (*rsi*.next()) {
   #sql { FETCH CURRENT FROM :*rsi*
                  INTO :x_1, :x_2, ...};
   ...process...x_1...x_2...
}
rsi.close();

Scrollable Iterators

#sql iterator *Iter*
      implements sqlj.runtime.Scrollable (...);
// or use
sqlj.runtime.ScrollableResultSetIterator *srsi*;

Predicates
isFirst(), isLast(), isBeforeFirst(), isAfterLast().

Movement Methods
previous(), first(), last(), beforeFirst(), afterLast(),
absolute(int), relative(int).

Positional Movement and Fetch Syntax
FETCH [NEXT | PRIOR | CURRENT | FIRST
        | LAST | ABSOLUTE :*x* | RELATIVE :*x*]
      FROM :*pi* INTO :x_1, :x_2 ...

JDBC Interoperability

java.sql.ResultSet *rs* = ...; *Iter iter* = ...;
// SQLJ iterator from result set
#sql *iter* = { CAST :*rs* }; ...process *iter*...; *iter*.close();
// result set from SQLJ iterator
rs = *iter*.getResultSet(); ...process *rs*...; *iter*.close();

Statements
Stored Procedures and Functions

#sql { CALL *Stored_Procedure*(...) };
#sql *assignable_expr* =
      { VALUES(*Stored_Function*(...)) };

SQL Expressions and PL/SQL

#sql { SET :*outputVar* = ...*SQL expression*... };
#sql { BEGIN ... END; };
#sql { DECLARE ... BEGIN ... END; }

Transaction Control

#sql { SET TRANSACTION ISOLATION
            READ COMMITTED }; // default setting!
#sql { SET TRANSACTION ISOLATION
            SERIALIZABLE };
#sql { ROLLBACK }; // Use these instead of the
#sql { COMMIT};   // JDBC methods!

Dynamic SQL

#sql { ... :{*runtime_expr*} ... };
#sql { ... :{*runtime_expr* :: *translate_time_SQL*} ...};
Example: String *table*="emp2"; String *col*="comm";
 #sql { select :{*col*::sal} INTO :*x* FROM :{*table*::emp} };

Execution Contexts

Create new or obtain it from a context
ExecutionContext *ec* = new ExecutionContext(); // or
ec = *ctx*.getExecutionContext(); // or
ec = DefaultContext.getDefaultContext()
                  .getExecutionContext();

APIs to set up execution properties
get/setMaxFieldSize() -max. size in bytes

get/setMaxRows() - max. rows returned in result set
get/setQueryTimeout() - timeout in seconds, default 0
get/setFetchSize() - result set prefetch size, default 10
get/setFetchDirection() - default FETCH_FORWARD
is/setBatching() - default is false (batching off)
get/setBatchLimit() - default 0 (=UNLIMITED_BATCH)

Execute with explicit context *ec*
#sql [*ec*] { ... };

APIs to control execution
executeBatch() - execute pending batch (if any) and
                  return the batch update counts
cancel() - cancel execution, cancel pending batch

APIs to retrieve execution results
getUpdateCount() - return number of rows updated or:
    - QUERY_COUNT (-1) - query execution
    - EXCEPTION_COUNT (-2) - execption occurred
    - NEW_BATCH_COUNT (-3) - new batch created
    - ADD_BATCH_COUNT (-4) - stmt. added to batch
    - EXEC_BATCH_COUNT (-5) - batch was executed
getBatchUpdateCounts() - after executing *n* batched
      statements: int[] with *n* elements containing -2 each

Type Support

Streams

For LONG columns use Ascii/UnicodeStream
sqlj.runtime.AsciiStream *as* =
      new sqlj.runtime.AsciiStream
            (*input_stream*, *input_stream_length*);
// For LONG RAW columns use BinaryStream

Retrieving streams
#sql { SELECT col INTO :*as* FROM stream_tab };
int len=*as*.getLength(); int c;
while((c= *as*.read()) ! = -1) System.out.println((char)c);
as.close();

Stream Limitations
- positional iterators/ResultSetIterators:
   only *one* long column, which must also be the last
- named iterators:
   column processing must obey the *select-list order*

LOBs

Reading LOBs
oracle.sql.BLOB blob; oracle.sql.CLOB clob;
oracle.sql.BFILE bfile;
...
// Note: the index starts from 1 (*not* from 0)
byte[ ] b = blob.getBytes(*begin_index*, *length*);
String s = clob.getSubString(*begin_index*, *length*);
b = bfile.getBytes(*begin_index*, *length*);

java.io.InputStream is = blob.getBinaryStream( );
is = clob.getAsciiStream( );
is = bfile.getBinaryStream( );
Write data into LOB
Create and select empty_lob(), except for JDBC-OCI.

```
byte[ ] b = ... ; String s = ...;
int amount_written = blob.putBytes(begin_index, b);
amount_written = clob.putString(begin_index,s);
```

Replace the LOB content from a stream
```
Writer w = clob.getCharacterOutputStream( );
OutputStream os = clob.getAsciiOutputStream( );
os = blob.getBinaryOutputStream( );
```

SQL-Java Type Compatibility

SQL Types	Java Types
NUMBER and all numeric types	boolean, Boolean, byte, Byte, short, Short, int, Integer, long, Long, float, Float, double, Double, String, java.math.Big-Decimal, oracle.sql.NUMBER,
CHAR, VARCHAR2, LONG	String, oracle.sql.CHAR, sqlj.runtime.AsciiStream, sqlj.runtime.UnicodeStream
RAW, LONG RAW	byte[], oracle.sql.RAW, sqlj.runtime.BinaryStream
DATE	java.sql.Date, java.sql.Time, java.sql.Timestamp, oracle.sql.DATE
BLOB	java.sql.Blob*, oracle.sql. BLOB, sqlj.runtime.Bina-ryStream**
CLOB	java.sql.Clob*, oracle.sql. CLOB, sqlj.runtime.Ascii-Stream**, sqlj.runtime.Uni-codeStream**
BFILE	oracle.sql.BFILE
ROWID	oracle.sql.ROWID
REF CURSOR	java.sql.ResultSet, SQLJ iter-ator instance
user-defined object	java.sql.Struct*, oracle.sql. STRUCT***, java.sql. SQLData* implementation, oracle.sql.ORAData implem.
user-defined reference	java.sql.Ref, oracle.sql. REF***, oracle.sql.ORAData implementation
user-defined collection	java.sql.Array, oracle.sql. ARRAY***, oracle.sql. ORAData implementation

* Type requires JDBC 2.0 or later (JDK 1.2 and above).
** Streaming to database only supported in JDBC-OCI.
*** Not supported as OUT, INOUT, or return parameter.

Object Type Support

Introduction to Object support
Use implementations of oracle.sql.ORAData in Java to read and write instances of SQL Object Types.
Define SQL Object Types (schema SCOTT)
```
CREATE TYPE ADDRESS AS OBJECT
    (street VARCHAR2(40));
CREATE TYPE PERSON AS OBJECT
    (name VARCHAR2(20), paddr ADDRESS);
CREATE TABLE PTAB AS TABLE OF PERSON;
CREATE TABLE ATAB
    (addr  ADDRESS, zip VARCHAR2(9));
```
Generate Java Wrappers with JPublisher
```
jpub -user=scott/tiger \
    -sql=ADDRESS:JAddress,PERSON:JPerson
javac JAddress*.* JPerson*.*    # use sqlj for .sqlj files
```

Use Java Wrappers in Test.sqlj
```
public class Test {
 public static void main(String[]args)
        throws SQLException
{ Oracle.connect("jdbc:oracle:oci8:@","scott","tiger");
  JAddress a; JPerson p = new JPerson();
  p.setName("John");
  #sql { INSERT INTO PTABLE VALUES(:p) };
  #sql { SELECT addr INTO :a FROM ATAB
                WHERE zip='12345' };
... } }  // To compile and run: sqlj Test.sqlj; java Test
```

SQLJ ISO Object Support
Using java.sql.SQLData from JDBC 2.0.
Create Java Wrappers With JPublisher
```
jpub -usertypes=jdbc -user=scott/tiger \
    -sql=ADDRESS:IsoAddress,PERSON:IsoPerson
javac IsoAddress*.* IsoPerson*.*
```
Create Type Map isoMap.properties
```
class.IsoAddress=STRUCT SCOTT.ADDRESS
class.IsoPerson=STRUCT SCOTT.PERSON
```

Use SQLData Wrappers in Iso.sqlj
```
public class Iso {
 #sql public static IsoCtx with (typeMap="isoMap");
 public static void main(String[]args)
        throws SQLException
{ new oracle.jdbc.driver.OracleDriver();
  IsoCtx ctx = new IsoCtx
        ("jdbc:oracle:oci8:@","scott","tiger",false);
  IsoAddress a; IsoPerson p = new IsoPerson();
  p.setName("John");
  #sql [ctx] { INSERT INTO PTAB VALUES(:p) };
  #sql [ctx] { SELECT addr INTO :a FROM ATAB
                WHERE zip='12345' };
... } }  // To compile and run: sqlj Iso.sqlj; java Iso
```

SQLJ Translator

CLASSPATH with Oracle 9i JDBC
JDK 1.1: translator.zip, runtime11.zip, classes111.zip.
JDK 1.2 or 1.3: translator.zip, runtime12.zip, classes12.zip.

sqlj [options] *.java *.sqlj

```
-user user/password -url url
    - connect to database at translate time as user
-d directory
    - place .class and .ser files under directory
-codegen=oracle
    - generate Oracle 9i JDBC code (no .ser files)
-version-long,  -help,  -help-long
    - show version and environment, give help
-explain,  -status
    - additional translation help and status messages
-linemap
    - map generated .class files to ..sqlj source files
```

sqlj [options] *.jar *.ser

```
-P-debug
    - add debugging auditor
-P-Cstmtcache=n
    - set SQLJ statement cache size n
```

Tools Reference

jpub [options]

```
-user=user/ password -url=url
    - connect to database as user
-sql=sql_name:java_name
    - generate Java wrapper java_name for sql_name
-package=package
    - use package for generated classes
-usertypes=[oracle|jdbc]
    - generate ORAData (SQLData) implementation
-methods=[always|false|true]
    - generate .sqlj files/.java files/.sqlj files iff methods
-input=file
    - read input for JPublisher translation from file
```

loadjava [options] *.jar
*.sqlj *.java *.ser *.properties
```
-user user/password[@host.port.sid] [-thin]
    - connect to database as user. Default: JDBC-OCI.
-resolve    - force immediate class resolution
-verbose    - print out what loadjava is doing
dropjava -user ... -verbose *.jar *.xxxx
    - drop classes and files
```
Tips for SQLJ translation prior to using loadjava
```
    - use -d directory to collect all .class and .ser files
    - optionally use -ser2class to avoid re-customization
    - optionally use jar to package application
    - remember to copy .properties files, if used!
```

SQLJ Resources

ORACLE
Quick Reference Card

June 2001   A90489-01