Osnovne karakteristike SQLJ

SQLJ predstavlja pristup koji omogućava da ugrađivanje SQL iskaza direktno u Java program. Za razliku od JDBC, kod SQLJ postoji sintaksna i semantička provera u vreme prevođenja. Rezultat toga jeste veća robusnost programa.

Prevodilac za SQLJ konvertuje Java program u kome se nalaze ugrađeni SQL iskazi tako da se dobije čist Java kod, koji se može izvršavati koristeći JDBC drajver za pristup bazi podataka.

Koraci neophodni prilikom pisanja SQLJ programa su:

1. Uvođenje neophodnih paketa. Pored JDBC klasa koje su obuhvaćene sa java.sql.*, potrebne su još i sqlj.runtime.* kao i sqlj.runtime.ref.*. Za uspostvljanje konekcije prema konkretnom DBMS potrebne su i dodatne klase (npr. za Oracle to je oracle.sqlj.runtime.*)

2. Registrovanje JDBC drajvera pozivom metode registerDriver() klase DriverManager.

3. Povezivanje na bazu podataka. Najpre se dohvata DefaultContext objekat koristeći getConnection metodu čiji je potpis:
public static DefaultContext getConnection (String url,String user,String password,boolean autoCommit)
throws SQLException

Tako dohvaćen DefaultContext je potrebno postaviti za statički podrazumevani kontekst pozivanjem metode setDefaultContext klase DefaultContext

4. Upotreba ugrađenih SQL iskaza upotrebom:
#sql {<sql-statement>}

Indikator #sql služi da SQLJ provodiocu ukaže da ono što sledi jeste SQL iskaz koji može, a ne mora sadržati host promeljive i host izraze koji služe za komunikaciju za host jezikom, odnosno okolnim Java programom.

Na primer:

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

public class Simple1 {

   public static void main (String args[]) throws SQLException {

      //Ukoliko se koristi Oracle baza podataka, onda nije potrebno registrovati drajver
      //jer je SQLJ proizvod kompanije Oracle

      DefaultContext cx1 = Oracle.getConnection("jdbc:oracle:oci8:@","book","book",true);

      DefaultContext.setDefaultContext(cx1);

      String cn;
      Double ap,bp,cp;
      //readEntry učitava jedan karakter sa standardnog ulaza
      String sym = readEntry("Enter symbol : ").toUpperCase();

      try {

         #sql {select cname,current_price,ask_price,bid_price
               into :cn,:cp,:ap,:bp
               from security
               where symbol = :sym };

      } catch (SQLException e) {
         System.out.println("Invalid symbol.");
         return;
        }

      System.out.println("\n Company Name = " + cn);
      System.out.println(" Last sale at = " + cp);

      if (ap == null)
         System.out.println(" Ask price = null");
         System.out.println(" Ask price = " + ap);
      if (bp == null)
         System.out.println(" Bid price = null");
      else
         System.out.println(" Bid price = " + bp);
      }
}
```

Važno je uočiti da ukoliko baza podatak vrati NULL, to se proverava tako što je vrednost host varijable null. Upravo zbog toga SQLJ implicira da na svakom mesto gde baza podataka može da vrati NULL, nije dozvoljeno koristiti host varijable koje su deklarisane kao Java primitivni tipovi.

Kompajliranje SQLJ programa

Prevodilac za SQLJ kao ulaz prihvata fajlove sa sufiksom .sqlj. Kao rezultat prevođenja nastaju .java fajlovi zajedno sa nekoliko SQLJ profilnih fajlova koji sadrže klase neophodne za izvođenje SQL operacija. Prevodilac automatski poziva Java kompajler kako bi bili proizvedeni .class fajlovi.

Prevodilac podržava i takozvanu online proveru sintakse. Odnosno u toku prevoženja prevodilac se povezuje na bazu koristeći navedeno korisničko ime i lozinku, kao npr:

> sqlj -url = jdbc:oracle:oci8:@ -user = book -password = book Simple1.sqlj

Ukoliko se želi uključivanje ili isključivanje određenih upozorenja:
> sqlj -warn = noprecision,nonulls Simple1.sqlj

Takođe, svi parametri prevođenja mogu biti navedeni u sqlj.properties fajlu, npr:

```
sqlj.driver = oracle.jdbc.driver.OracleDriver
sqlj.url = jdbc:oracle:oci8:@
sqlj.user = book
sqlj.password = book
sqlj.warn = noprecision,nonulls
```

Upotreba većeg broja konekcija

Programi koji su napisani za SQLJ mogu lako pristupati većem broju baza podataka tako što će kreirati jedan DefaultContext objekat za podrazumevanu konekciju i jedan ne podrazumevani kontekst konekcije za svaku dodatnu konekciju.

Na primer, kreiranje ne podrazumevanog konteksta konekcije, nazvan DbList se deklariše sa:

```
#sql context DbList;
```

Ova deklaracija će biti proširena od strane SQLJ prevodioca u odgovarajuću klasu DbList, koja može biti instancirana u SQLJ programu, čime se kreira novi kontekst konekcije, na primer:

```
DbList con2 = new DbList (Oracle.getConnection("jdbc:oracle:oci8:@", "book2", true));
```

Novi kontekst konekcije se dalje može koristiti za zadavanje SQL iskaza:

```
#sql   [con2]  {<sql-statement>};
```

U ovom slučaju sqlj.properties može izgledati ovako:

```
sqlj.driver = oracle.jdbc.driver.OracleDriver
sqlj.warn = noprecision,nonulls
#
sqlj.url = jdbc:oracle:oci8:@
sqlj.user = book
sqlj.password = book
#
sqlj.url@DbList = jdbc:oracle:oci8:@
sqlj.user@DbList = book2
sqlj.password@DbList = book2
```

Na primer:

```
import sqlj.runtime.*;
import sqlj.runtime.ref.*;
import java.sql.*;
import java.io.*;
import oracle.sqlj.runtime.Oracle;

#sql context Book2Context;

public class Simple2 {
   public static void main (String args[]) throws SQLException {

      DefaultContext x1 = Oracle.getConnection("jdbc:oracle:oci8:@","book","book",true);
      DefaultContext.setDefaultContext(x1);

      Book2Context x2 = new Book2Context(Oracle.getConnection(
                                        "jdbc:oracle:oci8:@","book2","book2",true));

      String dbname="";
      try {
         #sql [x2] { select db_name
                     into :dbname
                     from db_list
                     where db_name like 'C%' };

      } catch (SQLException e) {
          System.out.println("Error:" + e.getMessage());
          return;
        }

      System.out.println("DB name is " + dbname);
      String cn = "";
      String sym =

      readEntry("Enter symbol : ").toUpperCase();

      try {
        #sql { select cname
               into :cn
               from security
               where symbol = :sym };
      } catch (SQLException e) {
          System.out.println("Invalid symbol.");
          return;
        }

      System.out.println("\n Company Name = " + cn);
   }
}
```

Host varijable i izrazi

Host varijable i izrazi se koriste u SQLJ kako bi se prenosile vrednosti između SQL iskaza i okolnog Java programa. Host varijable su ili Java lokalne promenljive, Java deklarisani parametri, ili Java varijable klase/instance. Host izrazi su bilo koji validni Java izrazi.

Host varijable moraju počinjati sa dvotačkom koja treba da bude praćena sa IN, OUT ili INOUT, zavisno od smera prenošenja vrednosti. Podrazumevano je IN, pa se može izostaviti, npr:

```
#sql {select a into :a1 from t where b = :b1};
#sql {select a into :OUT a1 from t where b = :IN b1};
```

Host izrazi mogu da budu aritmetički izrazi, pozivi metoda sa povratnom vrednostima, elementi nizova, uslovni izrazi, logički izrazi itd.

Komplikovniji host izrazi treba da budu unutar zagrada kako bi se osigurala pravilna interpretacija, npr:

```
#sql {update member
      set cash_balance = :(x+100)
      where mid = :y };
```

Kada iz baze može doći NULL, onda je neophodno da se za host varijable koriste klase (Integer, Long, Float i Double), a ne primitivni tipovi.

Host varijable i izrazi treba da budu kompatibilni sa odgovarajućim SQL tipovima:

number, number(n), integer, integer(n)	→ int, long
number, number(n), number(n,d)	→ float, double
char, varchar2	→java.lang.String
date	→java.sql.Date
cursor	→ java.sql.ResultSet, SQLJ iterator objects

SQLJ iteratori

Obrada rezultata upita koji vraćaju veći broj redova zahteva upotrebu SQLJ iteratora koji odgovara Result Set kod JDBC, uz tu razliku da je strogo tipiziran i direktno povezan sa kursorom koji je definisan za određeni upit.

Deklaracija SQL iteratora specificira Java klasu koju SQLJ automatski konstruiše. Iterator klasa sadrži polja i metode koje odgovaraju tipovima kolona i opciono nazivima kolona za SQL upit koji je povezan sa tim iteratorom.

Kada se SQLJ iterator deklariše, programer može da specificira ili samo tipove podataka određenih kolona (pozicioni iterator) ili i tipove podataka i nazive kolona određenih kolona (imenovani iterator).

Objekat iteratora se instancira izvršavanjem SQL upita, dok se pritom SQL podaci dohvataju u iterator i konvertuju u odgovarajući Java tip specificiran deklaracijom iteratora.

Imenovani iterator specificira i tipove i nazive kolona, dok redosled nije bitan:

#sql iterator iterName (colType1 colName1, ..., colTypeN colNameN);

Na primer, za upit:

```
select to_char(trans_date,'DD-MON-YYYY') tdate,
       trans_type ttype, symbol, quantity
       price_per_share, commission, amount
from transaction
where mid = :mmid and to_char(trans_date, 'MM') = :month and to_char(trans_date, 'YYYY') = :year;
```

Odgovara sledeća deklaracija imenovanog iteratora:

```
#sql iterator TReport(String tdate, String ttype, String symbol, double quantity,
                      double price_per_share, double commission, double amount);
```

Nakon deklaracije iteratora, on može biti instanciran i popunjen u Java programu.

Na primer:

```
private static void printReport(String mmid) throws SQLException, IOException {

   String mmid2;

   try {
      #sql { select distinct mid
             into :mmid2
             from transaction
             where mid = :mmid };
   } catch (SQLException e) {
      System.out.println("No transactions");
      return;
     }
   String month = readEntry("Month(01 - 12): ");
   String year = readEntry("Year(YYYY): ");


   TReport t = null;

   #sql t=
      {select to_char(trans_date,'DD-MON-YYYY') tdate,
              trans_type ttype, symbol, quantity
              price_per_share, commission, amount
       from transaction
       where mid = :mmid and to_char(trans_date, 'MM') = :month
             and to_char(trans_date, 'YYYY') = :year };

   while(t.next()) {
      System.out.print(t.tdate() + " ");
      writeEntryRight(t.ttype(),6);
      writeSpaces(3);
      writeEntryLeft(t.symbol(),6);
      writeEntryRight(twoDigit(t.quantity()),10);
      writeEntryRight(twoDigit(t.price_per_share()),10);
      writeEntryRight(twoDigit(t.commission()),10);
      writeEntryRight(twoDigit(t.amount()),10);
      System.out.println();
   }
   writeDashes(68);
   System.out.println();
   t.close();
}

private static void writeEntryRight(String text, int width) {
   for (int i = 0; i < (width - text.length()); i++)
      System.out.print(" ");
   System.out.print(text);
}

private static void writeDashes(int width) {
   for (int i = 0; i < width; i++)
      System.out.print("-");
}

static String twoDigit(double f) {
   boolean neg = false;
   if (f < 0.0) {
      neg = true;
      f = -f;
   }
   long dollars = (long) f;
   int cents = (int) ((f - dollars) * 100);
   String result;
   if (cents <= 9) result = dollars + ".0" + cents;
   else result = dollars + "." + cents;
   if (neg) return "-" + result;
   else return result;
}
```

Pozicioni iterator specificira samo tipove, a ne i nazive kolona. Način instaciranja i popunjavanja je isti kao i kod imenovog iteratora, ali je način dohvatanja podataka iz iteratora drugačiji. Odnosno, za potrebe pozicionog iteratora se mora koristiti fetch into iskaz, zajedno sa endFetch metodom da bi se utvrdilo da li je poslednji red dohvaćen, npr:

#sql { fetch :iter into :var1, ..., :vark };

Metoda endFetch, kada se primeni nad interator objektom, inicijalno vraća true, pre nego je bilo koji red dohvaćen. Nakon toga vraća false posle svakog uspešnog dohvatanja reda, i konačno ponovo vraća true nakon što je poslednji red dohvaćen. Poziv metode endFetch mora biti napravljen pre nego je red dohvaćen, zbog toga što fetch ne baca nikakav izuzetak.

Na primer:

```
#sql iterator PQuote(String,String,double,Double,Double);

public static void getPriceQuoteByCname(String cn) throws SQLException, IOException {

   double cp=0.0;
   Double ap=null,bp=null;
   PQuote p = null;
   String sym = "";

   #sql p = {select symbol, cname, current_price, ask_price, bid_price
             from security
             where upper(cname) like :cn};

   #sql {fetch :p into :sym,:cn,:cp,:ap,:bp};

   if (!p.endFetch()) {
      System.out.print("Symbol"); writeSpaces(12);
      System.out.print("Company"); writeSpaces(17);
      System.out.print("Last Sale"); writeSpaces(4);
      System.out.print("Ask"); writeSpaces(7);
      System.out.println("Bid");
      writeDashes(74);
      System.out.println();

      while (!p.endFetch()) {
         writeEntryLeft(sym,9);
         writeEntryLeft(cn,30);
         writeEntryRight(twoDigit(cp),10);

         if (ap == null) System.out.print(" null");
         else writeEntryRight(twoDigit(ap.doubleValue()),10);

         if (bp == null) System.out.print(" null");
         else writeEntryRight(twoDigit(bp.doubleValue()),10);

         System.out.println();
         #sql {fetch :p into :sym,:cn,:cp,:ap,:bp};
      };
      writeDashes(74);
    } else {
        System.out.println("No company matches the name");
    }
    p.close();
}
```

Dinamički SQL

Po svojoj prirodi SQLJ podržava statički SQL, tako da se za potrebe dinamičkog SQL mora koristiti JDBC.

Na osnovu SQLJ podrazumevanog konteksta konekcije moguće je dohvatiti JDBC konekciju na sledeći način:

```
DefaultContext cx1 = Oracle.getConnection("jdbc:oracle:oci8:@","raj","raj",true);

DefaultContext.setDefaultContext(cx1);

Connection conn = DefaultContext.getDefaultContext().getConnection();
```

Slično, na osnovu JDBC konekcije je moguće dohvatiti SQLJ kontekst konekcije na sledeči način:

```
#sql context PortDB;

Connection conn = DriverManager.getConnection("jdbc:oracle:oci8:book/book");

PortDB cx2 = new PortDB(conn);
```

Na primer:

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

public class Dsqlj {
   public static void main (String args[]) throws SQLException {

       DefaultContext cx1 = Oracle.getConnection("jdbc:oracle:oci8:@", "book","book",true);
       DefaultContext.setDefaultContext(cx1);

      // Get a JDBC Connection object from an SQLJ DefaultContext object
     Connection conn = DefaultContext.getDefaultContext().getConnection();

     String sym = readEntry("Enter symbol substring: ").toUpperCase();
     String query ="select cname,current_price,ask_price,bid_price "+
                           "from security " +
                           "where symbol like '%" + sym + "%'";
     Statement stmt = conn.createStatement();
     ResultSet rs = stmt.executeQuery(query);
     while (rs.next()) {
         System.out.println("\n Company Name = " +
         rs.getString(1));
         System.out.println(" Last sale at = " + rs.getString(2));
         String ap = rs.getString(3);

         if (rs.wasNull()) System.out.println(" Ask price = null");
         else System.out.println(" Ask price = " + ap);

         String bp = rs.getString(4);

         if (rs.wasNull()) System.out.println(" Bid price = null");
        else System.out.println(" Bid price = " + bp);
     }
    rs.close();
    stmt.close();
  }
}
```

Poput konkcije, slično se i JDBC result set može transformisati u SQLJ iterator:

```
QueryIterator q;
#sql q = {<sql query>};
ResultSet rs = q.getResultSet();
```

i obratno:

```
ResultSet rs = ....;
#sql q = {CAST :rs};
```

Transakcije

Ukoliko se prilikom povezivanja na bazu podataka prilikom upotrebe getConnection metodu čiji je potpis:

public static DefaultContext getConnection (String url,String user,String password,boolean autoCommit)

odabere režim rada sa eksplicitnim transakcijama (odnosno, autoCommit je false) onda je kao kod JDBC potrebno zadavati komande COMMIT i ROLLBACK, i to koristeći:

```
#sql {COMMIT [WORK]};
#sql {ROLLBACK [WORK]};
```

Pri tome je ključna reč WORK opciona i ne menja način funkcionisanja komandi COMMIT i ROLLBACK.

Pozivanje uskladištenih procedura i funkcija iz SQLJ

SQLJ program može da sadrži tzv. anonimni blok u kome može biti definisan SQL proceduralni kod.

Neka je data tabela kreirana sa: create table squares (n number, nsquared number);

Onda sledeći SQLJ deklariše anonimni proceduralni kod za popunjavanje te tabele:

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

    public class Anon {
        public static void main (String args[]) throws SQLException {

            DefaultContext cx1 = Oracle.getConnection("jdbc:oracle:oci8:@", "book","book",true);
            DefaultContext.setDefaultContext(cx1);

            #sql {
                      declare n int;
                      begin
                          delete from squares;
                          set n := 1;
                          while (n <= 10)
                               insert into squares values (n,n*n);
                               n := n + 1;
                          end while;
                      end;
              };
           }
    }
```

Takođe, SQLJ program može da poziva uskladištene procedure. Na primer, ukoliko postoji procedura:

```
create procedure latestTransactionDate (midd in char[50], ldate out date) is
begin
     select max(trans_date)
     into ldate
     from transaction
     where mid = midd;
end;
```

Onda se ta procedura može pozvati iz SQLJ programa na sledeći način:

```
String m = "10000";
java.sql.Date lastDate;

#sql {CALL latestTransactionDate(:in m, :out lastDate)};
```

Slično, SQLJ program može da poziva uskladištene funkcije. Na primer, ukoliko postoji funkcija:

```
create function avgPP( mid in char[50], sym in char[50])
return decimal(10,2)
as...
```

Onda je poziv funkcije iz SQLJ moguć na sledeći način:

```
String m = readEntry("Member ID: ");
String sym = readEntry("Symbol: ");
double pp;

#sql pp = { VALUES (avgPP(:in m, :in sym)) };
```