

Oracle SQLJ Roadmap

An Oracle White Paper
July 2004

INTRODUCTION	3
1. Continue Use of The sqlj API.....	4
1.1 Leaving Existing Applications Unchanged.....	4
1.2 Using SQLJ through JPublisher.....	4
2.1 Migration Options.....	Error! Bookmark not defined.
2.1.1 -migrate	7
2.1.2 -migconn	7
2.1.3 -migrsi.....	8
2.1.4 -migsync	8
2.1.5 -migdriver.....	8
2.1.6. -migcodegen	9
2.1.7 -migserver	9
2.1.8 Examples.....	Error! Bookmark not defined.
2.2 Migration Path	10
2.3 DefaultContext	Error! Bookmark not defined.
2.3.1 Example	Error! Bookmark not defined.
2.4 Generic Iterator	17
2.5 User-defined ConnectionContext and Iterator.....	23
2.5.1 Example	Error! Bookmark not defined.
2.6 ExecutionContext.....	26
2.7 Signature Conflicts	28
2.8 Close Connections and Statements.....	30
2.9 Migration-Prompted Code Change	31
2.9.1 Signature Change	31
2.9.2 The Default Connection.....	32
2.9.3 Property and Type Map Files	36
2.9.4 Additional .class Files.....	36
2.9.5 Thread Safty.....	37
2.9.6 More Examples	38
CONSLUSION	38

INTRODUCTION

SQLJ is the umbrella name for the ANSI SQL-1999 (SQL3 or SQL-99) multiparts specification.

SQLJ Part-0, or *Embedded SQL in Java*, specifies the ability to embed SQL statements in Java. The Oracle implementation is referred to as Oracle SQLJ. It simplifies JDBC programming, and is to JDBC what JavaServer Pages are to Java Servlets. Because we have not seen a widespread adoption of SQLJ in the Java community in general, and the J2EE community in particular, Oracle is desupporting SQLJ—starting with Oracle Database 10g Release 1 and Oracle Application Server 10g Release 10.1.3. The goal of this white paper is to both address the impact of SQLJ desupport for developers, and discuss the migration paths to JDBC.

SQLJ Part-1, or *SQL Routines Using Java*, specifies the ability to invoke static Java methods from SQL as procedures and functions. This is also referred to as Java Stored Procedures, of which there is a growing and widespread adoption.

SQLJ Part-2, or *SQL Types Using Java*, specifies the ability to use Java classes as templates for user-defined data types in SQL. The Oracle implementation is referred to as SQL Object types.

As of Oracle Database 10g Release 1, Oracle no longer supports the SQLJ translator. Specifically, you can no longer translate `.sqlj` programs into `.java` and `.class` files using the `sqlj` command line. The SQLJ runtime, however, is still shipped and supported. That is, if your SQLJ client applications or SQLJ stored procedures are already translated, then those applications can still run against (client) or inside (stored procedures) Oracle Database 10g.

As of Oracle Application Server 10g Release 10.1.3, Oracle no longer supports the SQLJ translator in the application server, JSP, and JDeveloper. Specifically, the SQLJ command line has been removed. In addition, Oracle no longer supports the SQLJ JSP page (`.sqljsp`), and JDeveloper no longer takes `.sqlj` files as input files.

Following SQLJ product desupport, you have two options:

- Continue using the SQLJ-API, despite desupport.
- Migrate your SQLJ applications to JDBC.

The remainder of this paper discusses these two options in detail.

1.0 CONTINUE USING THE SQLJ API

Oracle is aware that customers may want to preserve their investment in the SQLJ API, including those who cannot migrate existing SQLJ applications or those who simply want to continue using the API—despite desupport.

1.1 Leaving Existing Applications Unchanged

SQLJ customers who wish to continue SQLJ development can use the SQLJ translator from previous releases, which comes with Oracle Database 9*i* or earlier, and with Oracle Application Server 10g (10.0.2) or earlier. You can also download the SQLJ translator Release 8*i* or Release 9*i* from the OTN Web site

http://otn.oracle.com/software/tech/java/sqlj_jdbc/index.html.

Existing pre-Oracle Database 10g SQLJ applications work against Oracle Database 10g. Similarly, pre-Oracle Database 10g SQLJ stored procedures work unchanged within an Oracle Database 10g Java Virtual Machine (JVM).

1.2 Using SQLJ Through JPublisher

SQLJ customers who want to continue using the Oracle SQLJ API can do so by using the `-sqlj` option in Oracle JPublisher 10g, to translate SQLJ programs. Oracle offers this option as a convenience and does not represent our product decision.

Here is the code for translating SQLJ programs using JPublisher:

```
jpub -sqlj <sqlj options> <list of .sqlj files>
```

This is equivalent to the following command in Oracle 9*i*:

```
sqlj <sqlj options> <list of .sqlj files>
```

For example, the command

```
jpub -sqlj -d . -explain foo.java bar.sqlj
```

works the same as the following SQLJ command in an Oracle 9*i* database:

```
sqlj -d . -explain foo.java bar.sqlj
```

When you use the SQLJ translator furnished by JPublisher in Oracle Database 10g, the `sqlj` command is replaced by `jpub -sqlj`, but the build, translation, and compilation processes do not change.

1.3 Migrate Your SQLJ Applications to JDBC

Oracle encourages customers to migrate their applications to JDBC, to be free of SQLJ runtime dependencies. The SQLJ translator translates a SQLJ program into a Java program that depends on SQLJ runtime APIs. We refer to SQLJ migration as converting SQLJ programs into pure JDBC programs.

A SQLJ program is referred to as a `.java` or `.sqlj` file that uses SQLJ `#sql` statements or SQLJ runtime APIs. SQLJ statements, appearing in files with the suffix `.sqlj`, start with `#sql`. For example:

```
#sql iter = {select ename from emp};
```

SQLJ runtime classes reside under the package hierarchies `sqlj.runtime` and `oracle.sqlj.runtime`—for example, `sqlj.runtime.ref.DefaultContext` and `oracle.sqlj.runtime.Oracle`.

A JDBC program is made up of `.java` files. It uses JDBC APIs for database access, but does not contain calls to SQLJ APIs.

You can migrate a SQLJ program into a JDBC program in either of two ways:

- Manually—by hand
- Automatically—using the SQLJ migration tool supplied with JPublisher

Migrating a SQLJ application manually is a tedious job. To ease the amount of work in migrating SQLJ to JDBC applications, Oracle JPublisher 10g Release 2 offers a migration tool through the `-migrate` option. Although the migration tool provides migration assistance, it does not offer a totally automatic solution. Some applications require human inspection and manual change to the migrated code generated by the migration tool. However, the migration tool greatly reduces the workload for migrating a SQLJ application. JPublisher is a database resource publishing tool distributed with Oracle Database releases. To learn more about Oracle JPublisher, refer to the *JPublisher 10g User's Guide*.

Employing Oracle JPublisher, you can migrate a SQLJ program into JDBC programming using the `jpub` command line and the `-migrate` flag. For example, the following command converts and compiles the SQLJ programs `Demo1.sqlj` and `Demo2.java` into the JDBC programs `Demo1.java` and `Demo2.java`:

```
jpub -migrate Demo1.sqlj Demo2.java
```

The `jpub` command also compiles `Demo1.java` and `Demo2.java` into `.class` code.

Any SQLJ program specified on the command line is translated into a JDBC program. If a SQLJ file has the suffix `.sqlj`, then Oracle JPublisher generates a JDBC file with a `.java` suffix, as well as `.class` files.

If the SQLJ file has a `.java` suffix, say `Demo2.java`, then Oracle JPublisher renames that file with the suffix `.migrate`—that is, `Demo2.java.migrate`—and generates new content for the `.java` file. If the file `Demo2.java.migrate` already exists, then the original SQLJ file `Demo2.java` is overwritten. In this case, no copy of the original SQLJ program is saved.

If you specify a JDBC file on the Oracle JPublisher command line, then the JDBC file is compiled into `.class` code, without generating any new `.java` files.

If the `-dir` setting in the Oracle JPublisher command line is specified differently from the current directory, then a SQLJ file with a `.java` suffix is not renamed or overwritten. Instead, the newly generated JDBC file is generated under the directory specified by the `-dir` setting.

In summary, the following files are generated by the preceding command:

<code>Demo1.java</code>	SQLJ-free Java program
<code>Demo1.class</code>	SQLJ-free Java class
<code>Demo2.java</code>	SQLJ-free Java program
<code>Demo2.class</code>	SQLJ-free Java class
<code>Demo2.java.migrate</code>	Content of the original <code>Demo2.java</code> file

In migration mode, Oracle JPublisher translates SQLJ programs using a style similar to the SQLJ setting `-codegen=oracle`. The migrated JDBC programs are Oracle-specific. If you specify `-codegen=iso` in the Oracle JPublisher command line, then Oracle JPublisher automatically turns off migration mode and generates SQLJ programs with `-codegen=iso`. For instance, the following command translates `Demo1.sqlj` into SQLJ programs in the form of `.java`, `.class`, and `.ser` files, as produced by SQLJ `iso` code generation.

```
jpub -migrate -codegen=iso Demo1.sqlj
```

The setting `-migrate` is, in fact, ignored.

2.1 Migration Options

Oracle JPublisher 10g Release 2 supplies the following options to assist in SQLJ migration.

<code>-migrate</code>	Turn on SQLJ migration.
<code>-migconn</code>	Specify the default connection in the migrated code.
<code>-migrsi</code>	Specify an interface for all ResultSet iterator classes.
<code>-migsync</code>	Mark static variables synchronized.
<code>-migdriver</code>	Specify the JDBC driver registered by the migrated code.
<code>-migcodegen</code>	Oracle JDBC-specific code generation or not.

`-migserver` Migrate the program to be used on the server side.

The following sections discuss the syntax of these options.

2.1.1 -migrate

Here is the syntax for `-migrate`:

```
-migrate[=true|false]
```

For example: `-migrate`, `-migrate=true`, or `-migrate=false`. The settings `-migrate` and `-migrate=true` are equivalent, indicating that Oracle JPublisher migrates SQLJ programs specified in the command line into JDBC programs. The setting `-migrate=false` turns off the migration mode; a SQLJ program is translated and compiled into a Java program that possibly depends on the SQLJ runtime. By default, if the no `-migrate` option is specified, then JPublisher behaves like `-migrate=false`.

Do not confuse `-rmslqj` with `-sqlj`, which is a JPublisher option for translating and compiling SQLJ programs. The setting `-sqlj` indicates that JPublisher will translate and compile `.java` or `.sqlj` files into `.class` files. The setting `-migrate` indicates that the generated `.class` files must be free of SQLJ runtime. Consider the following commands:

```
jpub -sqlj <sqlj options> <.sqlj or .java files>
jpub -migrate <.sqlj or .java files>
jpub -migrate -sqlj <sqlj options> <.sqlj or
.java files>
```

The first command generates `.class` files that use SQLJ runtime APIs; the second and third commands generate SQLJ runtime-free `.class` files. Note that when no SQLJ options are specified, the `-sqlj` switch can be omitted, in which case JPublisher translates and compiles any `.java` or `.sqlj` file specified.

2.1.2 -migconn

The `-migconn` option specifies the default JDBC connection used by the migrated code. The default connection replaces the default `DefaultContext` instance in the SQLJ runtime. The syntax for `-migconn` is the following:

```
-migconn=getter:<getter>,setter:<setter>
-migconn=<name>[:<datasource>|<modifier>][,<modifier>]*]
```

The term `<modifier>` is a Java modifier, including `protected`, `public`, `private`, `package`, and `static`. In the first syntax, the `<setter>` and `<getter>` settings specify the setter and getter methods for the default connection. For instance:

```
-migconn=getter:Test.getDefConn,setter:Test.setDefConn
```

This setting indicates that the methods `getDefConn` and `setDefConn` in the class `Test` are the accessors for the default connection. The preceding setting assumes that these two methods exist in `Test`. The second syntax for `-migconn` specifies a variable, called `<name>`, as the default connection. The optional `<datasource>` setting provides a JNDI data source location for initializing the default connection. The `<modifier>` settings add the modifiers for the default connection variable. Here are examples of the second syntax:

```
-migconn=_defaultConn:public,static
-migconn=Test._defaultConn
-migconn=Test._defaultConn;jdbc/MyDataSource
-migconn=_defaultConn:public,static,final
```

The first setting indicates that each migrated file contains a public static variable, `_defaultConn`, as the default connection for that file. The second setting identifies the default connection as the variable `_defaultConn` in `Test`. The third setting specifies that a data source be used to initialize the default connection variable `_defaultConn` in `Test`.

2.1.3 -migrsi

Here is the syntax for `-migrsi`:

```
-migrsi=<java interface name>
```

Here are some examples of the `-migrsi` option:

```
-migrsi=ResultSetInterface
-migrsi=Test.ResultSetInterface
```

2.1.4 -migsync

Here is the syntax for `-migsync`:

```
-migsync[=true|false]
```

The settings `-migsync` and `-migsync=true` mark certain of the variables generated for migration purpose as synchronized. The setting `-migsync=false` turns off that behavior. By default, `JPublisher` behaves like `-migsync=true`.

2.1.5 -migdriver

The syntax for `-migdriver` settings are shown below.

```
-migdriver=<class name>|no
```

For instance:

```
-migdriver=oracle.jdbc.driver.OracleDriver
-migdriver=com.jdbc.Driver
```

`-migdriver=no`

The first two settings specify the driver classes used to register the JDBC driver in the migrated code. The last setting indicates that driver registration code will not be generated during migration. By default, JPublisher behaves like `-migdriver` set as `oracle.jdbc.driver.OracleDriver`.

2.1.6 `-migcodegen`

The `-migcodegen` option indicates that the migrated code depends on an Oracle JDBC driver or a generic JDBC driver. Here is the syntax for `-migcodegen`:

`-migcodegen=oracle|jdbc`

The default behavior is `-migcodegen=oracle`—that is, Oracle-specific JDBC APIs are used in the migrated code.

2.1.7 `-migserver`

Here is the syntax for the `-msqljserver` option:

`-migserver`

This option indicates that the code is to be generated for server-side usage.

2.1.8 Examples

Following are some examples of the migration commands.

```
jpub -migrate Coffe.java Bean.sqlj
```

```
jpub -migrate -migserver Coffe.java Bean.sqlj
```

```
jpub -migrate \  
-
```

```
-migconn=getter:ConnManager.getConnection,setter:ConnManager.setConn\  
-
```

```
-migcodegen=jdbc\  
-
```

```
-migdriver=no\  
-
```

```
Coffe.java \  
-
```

```
Bean.sqlj
```

```
jpub -migrate \  
-
```

```
-migconn=ConnManager.defaultConnection \  
-
```

```
-migrsi=ConnManager.ResultSetIntf \  
-
```

```
-migdriver=no\  
-
```

```
-migsync=false \  
-
```

```
Coffe.java \  
-
```

2.2 Migration Path

Connection management is an important consideration in migrating a SQLJ application. The SQLJ runtime provides a default connection context as a built-in connection management mechanism for SQLJ applications. When you migrate a SQLJ application that relies on the default SQLJ connection context, you must decide on a replacement connection management mechanism in the migrated code. The migration option `-migconn` allows you to specify centralized accesses to a default JDBC connection, which plays the same role in the migrated code as the default connection context in the SQLJ runtime. For instance, you can create a class called `ConnectionManager` with two methods, `setConnection` and `getConnection`, and specify these two methods in the `-migconn` option such that all references to the default connection context are directed to `ConnectionManager`.

If the SQLJ application does not use the default connection context, then the preceding discussion is not applicable, because the `-migconn` option handles the default connection context. However, it is atypical for applications not to use the default connection context. Note that many `#sql` statements with no connection context specified use the default connection context. Many SQLJ APIs, including `sqlj.runtime.ref.DefaultContext#getDefaultContext`, `oracle.sqlj.runtime.Oracle#connect`, and `oracle.sqlj.runtime.Oracle#getConnection` access and manipulate the default connection context.

Follow these steps to migrate a SQLJ application:

1. Provide a default JDBC connection, by creating a new JDBC connection variable or a pair of JDBC connection accessors in an existing class or a new class. This step may need manual code changes.
2. Run the migration tool against all the SQLJ programs, including those `.java` programs. Specify `-migconn`, using the connection management mechanism provided in Step 1. If necessary, specify other migration options.
3. Inspect the migrated code and the code you created in Step 1. Verify that the migrated code works properly. This step may need manual code changes.
 - a. Make sure the default JDBC connection is initialized appropriately.
 - b. Inspect the migrated code for change signatures, which may require manual code changes to associate programs. In the migrated code, SQLJ APIs are mapped into JDBC APIs; therefore, some method signatures in the SQLJ application may change. You must manually update any external reference to those change signatures.
 - c. Inspect the migrated code for potential resource issues. The SQLJ runtime recycles resources with `sqlj.runtime.ExecutionContext` and

`oracle.sqlj.runtime.Oracle` APIs. These APIs are mapped into operations, based on individual data created in each migrated file. You must inspect the migrated code to verify that JDBC resources such as statements and connections are appropriately closed at the end of their life cycles.

- d. Inspect the migrated code for potential performance issues. The SQLJ runtime manages performance-related factors such as caching and batching with `sqlj.runtime.ExecutionContext` APIs. In the migrated code, these APIs are replaced by operations on a hash table created for each migrated file. All the connections and statements in the same file share the same hash table, which stores properties such as cache size and batch size.
4. Run a test to verify the migrated application.
 5. Adjust the application for changes to property and type map files. After migration, the connection property and type map files are no longer used. Such changes may require adjustment in packaging and installation.

So far, we have looked at client-side SQLJ applications. Migrating server-side SQLJ applications is different from migrating client-side applications. On the server side, the default connection is provided by a server-side JVM. Step 1 above is no longer needed for server-side migration. Notably, the `-migconn` option is not needed. The typical migration procedure includes first dropping the SQLJ files from the server, next running the migration tool against the application outside the database, then inspecting and code adjusting, and, finally, reloading the migrated application into the server.

2.3 Default Context

Many SQLJ applications utilize the default `DefaultContext` instance defined in the SQLJ runtime. This default instance provides the default JDBC connection for SQL statement execution. During migration, this default instance is mapped in a JDBC connection, regarded as the default connection in the migrated program.

This setting controls how the default `DefaultContext` instance in the SQLJ program is handled:

```
-migconn
```

Here is the default setting, which specifies a public static `java.sql.Connection` variable, named `DefaultContext`, to be created in each migrated program:

```
-migconn=DefaultContext:public,static
```

That is:

```
public static java.sql.Connection DefaultContext;
```

The connection variable specified in `-migrate` holds the underlying JDBC connection corresponding to the default connection context instance in the SQLJ runtime. If no class name is specified, then every migrated file contains a static `java.sql.Connection` variable named `DefaultContext`. The `-migconn` option can specify the variable name prefixed with the name of the class in which the variable exists or should be created. If that variable does not exist in that class, and that class is also specified to be migrated, then that variable is created as the result of the `jpub` command. Consider the command:

```
jpub -migrate -migconn=Test._defaultConn Demo1.sqlj
```

With the preceding setting, any reference to the `DefaultContext` instance in the original `Demo1.sqlj` is replaced by the variable `Test._defaultConn`. No `java.sql.Connection` variable is created in `Demo1.sqlj`. However, if translating `Demo1.sqlj` triggered translation of `Test.sqlj` or compilation of `Test.java`, then the new variable `defaultConn` is created in the `Test` class if `Test.sqlj` or `Test.java` does not already contain a variable called `_defaultConn`.

If a class is defined with such a connection variable, then its subclasses do not have that variable redefined. If a SQLJ program file contains several classes—for example, inner classes or nested classes—then that variable is created for the public top-level class, and other inner classes or nested classes, if necessary, reference that variable.

In addition to mapping the default connection context into a static variable, you can also model it by using a pair of setter and getter methods. The following command generates JDBC programs where accessing the default context is translated into setter and getter method calls:

```
jpub
-migconn=accessors:DemoDriver.getDefConn,DemoDriver.setDefConn
Demo1.sqlj Demo2.java
```

For instance, the two statements

```
DefaultContext.getDefaultContext.setConnection(myConn);
foo(DefaultContext.getDefaultContext())
```

are translated into the following two statements, respectively

```
DemoDriver.setDefConn(myConn);
foo(DemoDriver.getDefConn());
```

By default, each migrated program also contains the following static block to make sure `OracleDriver` is registered so that the connection variable instantiated for the default context can be created properly.

```
static {
```

```

try { java.sql.DriverManager.registerDriver(new oracle.jdbc.OracleDriver());}
catch(java.sql.SQLException e) { /* No Driver available */ }
}

```

Use the setting `-migdriver=no` to turn off generation of the preceding block. You can also use the `-migdriver=<driver name>` setting to set other driver classes. For instance, the command:

```
jpub -migrate -migdriver=MyDriver Demo.sqlj
```

produces the following static block:

```

static {
    try { java.sql.DriverManager.registerDriver(new MyDriver()); }
    catch(java.sql.SQLException e) { /* No driver available */ }
}

```

With the `-migrate` option, you can also instantiate the default connection using a data source. For instance, the setting:

```
-migrate=_defaultConnection;jdbc/defaultDataSource
```

produces the following static block along each declaration of the connection variable `_defaultConnection`:

```

static
{
    try
    {
        Class contextClass = Class.forName("javax.naming.Context");
        Class dataSourceClass = Class.forName("javax.sql.DataSource");
        Object c =
Class.forName("javax.naming.InitialContext").newInstance();
        Object ds = contextClass.getMethod("lookup", new
Class[] {String.class})
            .invoke(c,new Object[] {"jdbc/defaultDataSource"});
        DefaultContext
            = (java.sql.Connection)
dataSourceClass.getMethod("getConnection",new Cla
ss[] {}).invoke(ds, new Object[] {});

```

```

    }
    catch (Throwable t) {}
}

```

2.3.1 Example

Consider the following program, `TestDCtx.sqlj`:

```

import sqlj.runtime.ref.DefaultContext;

public class TestDCtx
{
    public void foo() throws java.sql.SQLException
    {
        bar(DefaultContext.getDefaultContext().getConnection());
    }
    public void bar(DefaultContext defCtx) throws java.sql.SQLException
    {
        DefaultContext.setDefaultContext(defCtx);
    }
}

```

The command:

```
jpub -migrate TestDCtx.sqlj
```

produces the following `TestDCtx.java` file:

```

public class TestDCtx
{
    public void foo() throws java.sql.SQLException
    {
        bar(TestDCtx.DefaultContext);
    }
    public void bar(java.sql.Connection defCtx) throws
java.sql.SQLException
    {
        TestDCtx.DefaultContext = defCtx;
    }
}

```

```

//
// Utilities Generated by -migrate, the option for migrating SQLJ to JDBC
//
public static java.sql.Connection DefaultContext;

public static java.sql.Statement _migDefaultStatement;
public static java.util.Hashtable ExecutionContext = new
java.util.Hashtable();

static {
    try { java.sql.DriverManager.registerDriver(new
oracle.jdbc.OracleDriver()); }
    catch(java.sql.SQLException e) { /* No Driver available */ }

    //Initialize Statement Holder
    ExecutionContext.put("Statements", new java.util.Vector());
}
}

```

The command:

```

jpub -migrate -migconn=accessors:Test.getConn,Test.setConn
TestDCtx.sqlj

```

produces the following TestDCtx.java file:

```

public class TestDCtx
{
    public void foo() throws java.sql.SQLException
    {
        bar(Test.getConn());
    }

    public void bar(java.sql.Connection defCtx) throws
java.sql.SQLException
    {
        Test.setConn(defCtx);
    }
}

```

```

public static java.sql.Statement _migDefaultStatement;
public static java.util.Hashtable ExecutionContext = new
java.util.Hashtable();
static {
    try { java.sql.DriverManager.registerDriver(new
oracle.jdbc.OracleDriver()); }
    catch(java.sql.SQLException e) { /* No Driver available */ }
    //Initialize Statement Holder
    ExecutionContext.put("Statements", new java.util.Vector());
}
}

```

Both examples show that a `DefaultContext` instance is mapped into a `java.sql.Connection` instance. The default `DefaultContext` instance is mapped into the default connection, which is accessed either as a variable or with a pair of setter and getter methods. The second command assumes that there is a program called `Test.java` in the classpath, which declares two methods: `getConn` and `setConn`.

The `oracle.sqlj.runtime.Oracle` connect methods are often used to initialize the default connection context. Suppose we add the following statement:

```
Oracle.connect(TestDCtx.class, "connect.properties");
```

into `TestDCtx.sqlj`.

The command:

```
jpub -migrate TestDCtx.sqlj
```

translates the code into a statement that sets the default connection with connection information that was read during migration time.

```
TestDCtx.DefaultContext=(TestDctx.DefaultContext!=null?TestDCtx.DefaultCon
text:java.sql.DriverManager.getConnection("jdbc:oracle:oci8:@",
```

```
"scott","tiger"));
```

We also assume that the content of `connect.properties` looks like this:

```
sqlj.user=scott
```

```
sqlj.password=tiger
```

The `connect.properties` file is read during migration. Its content no longer takes effect at runtime.

2.4 Generic Iterator

The following interfaces may appear in SQLJ programs, to declare iterator variables or parameters.

```
sqlj.runtime.ResultSetIterator
sqlj.runtime.PositionedIterator
sqlj.runtime.NamedIterator
sqlj.runtime.Scrollable
```

For a SQLJ program using these generic iterators, the migration tool creates an inner class called `ResultSetIterator` to replace the generic iterator type. Each class created for an iterator in that SQLJ program extends that `ResultSetIterator` class. In addition, if the `jpub` command contains the following option, then each iterator class created extends the class "`<class name>`".

```
-migrsi=<class name>
```

If the term `<class name>` is prefixed with a class name different from the one being migrated, then no inner class is created. Otherwise, an inner class is created in the migrated program.

Consider the program `TestRSI.sqlj`:

```
import sqlj.runtime.ResultSetIterator;

public class TestRSI
{
    #sql public static iterator EmpIter(String ename);
    public void foo() throws java.sql.SQLException
    {
        EmpIter iter=null;
        iter = (EmpIter) bar(iter);
    }
    public ResultSetIterator bar(ResultSetIterator iter) throws
java.sql.SQLException
    {
        #sql iter = {SELECT ename from emp};
        return iter;
    }
}
```

The command:

```
jpub -migrate TestRSI.sqlj
```

produces the `TestRSI.java` file sketched below.

```
public class TestRSI
{
    // SQLJ iterator declaration:
    // *****

    public static class EmpIter extends TestRSI.ResultSetIterator
    {
        public EmpIter(java.sql.ResultSet resultSet)
            throws java.sql.SQLException
        {
            m_rs = (oracle.jdbc.OracleResultSet) resultSet;
            m_isClosed = false;
            enameNdx = m_rs.findColumn("ename");
        }
        public EmpIter(ResultSetIterator rsi)
            throws java.sql.SQLException
        {
            this(rsi.getResultSet());
        }
        public String ename()
            throws java.sql.SQLException
        {
            return (String)m_rs.getString(enameNdx);
        }
        private int enameNdx;
    }
    public static class ResultSetIterator
```

```

{
    private java.util.Map m_typeMap;
    public void setTypeMap(java.util.Map typeMap) { m_typeMap
= typeMap; }
    public java.util.Map getTypeMap() { return m_typeMap; }
    public ResultSetIterator(java.sql.ResultSet resultSet)
        throws java.sql.SQLException
    {
        m_rs = (oracle.jdbc.OracleResultSet) resultSet;
        m_isClosed = false;
    }
    private oracle.jdbc.OracleResultSet m_rs;
    public java.sql.ResultSet getResultSet() { return m_rs; }
    ....
}
public void foo() throws java.sql.SQLException
{
    EmpIter iter=null;
    iter = new EmpIter(bar(iter));
}
public ResultSetIterator bar(ResultSetIterator iter) throws
java.sql.SQLException
{
    // #sql iter = { SELECT ename from emp };
    // *****
{
    // declare temps
    oracle.jdbc.OraclePreparedStatement __sJT_st = null;
    String theSqlTS = "SELECT ename from emp";
    __sJT_st =

```

```

(oracle.jdbc.OraclePreparedStatement)((oracle.jdbc.OracleConnection)TestRSI.DefaultContext).prepareStatement(theSqlTS);
    _migDefaultStatement = __sJT_st;
    if (ExecutionContext.get("Statements")!=null)
        ((java.util.Vector)
ExecutionContext.get("Statements")).addElement(_migDefaultStatement);
    // execute query
    iter = new ResultSetIterator( __sJT_st.executeQuery());
}
    return iter;
}
//
// Utilities Generated by -migrate, the option for migrating SQLJ
to JDBC
//
public static java.sql.Connection DefaultContext;
public static java.sql.Statement _migDefaultStatement;
public static java.util.Hashtable ExecutionContext = new
java.util.Hashtable();
static {
    try { java.sql.DriverManager.registerDriver(new
oracle.jdbc.OracleDriver()); }
    catch(java.sql.SQLException e) { /* No Driver available */ }
    //Initialize Statement Holder
    ExecutionContext.put("Statements", new java.util.Vector());
}
}

```

The preceding migrated code contains an inner class, `ResultSetIterator`, to serve as a generic iterator class. All iterator classes extend this class. Alternatively, you can specify an existing class for `ResultSetIterator`. In this case, the migration code does not have to define its own copy of the `ResultSetIterator` class.

The command:

```
jpub -migrate -migrsi=Test.RSIter TestRSI.sqlj
```

produces the following `TestRSI.java` file:

```
import java.sql.*;

public class TestRSI
{
    // SQLJ iterator declaration:
    // *****

    public static class EmpIter extends Test.RSIter
    {
        public EmpIter(java.sql.ResultSet resultSet)
            throws java.sql.SQLException
        {
            m_rs = (oracle.jdbc.OracleResultSet) resultSet;
            m_isClosed = false;
            enameNdx = m_rs.findColumn("ename");
        }

        public EmpIter(TestRSIter rsi)
            throws java.sql.SQLException
        {
            this(rsi.getResultSet());
        }

        public String ename()
            throws java.sql.SQLException
        {
            return (String)m_rs.getString(enameNdx);
        }

        private int enameNdx;
    }
}
```

```

public void foo() throws java.sql.SQLException
{
    EmpIter iter=null;
    iter = new EmpIter(bar(iter));
}

public ResultSetIterator bar(ResultSetIterator iter) throws
java.sql.SQLException
{

// #sql iter = { SELECT ename from emp };
// *****
{
// declare temps
oracle.jdbc.OraclePreparedStatement __sJT_st = null;
String theSqlTS = "SELECT ename from emp";
__sJT_st =

(oracle.jdbc.OraclePreparedStatement)((oracle.jdbc.OracleConnection)TestRSI.DefaultContext).prepareStatement(theSqlTS);
    _migDefaultStatement = __sJT_st;
    if (ExecutionContext.get("Statements")!=null)
        ((java.util.Vector)
ExecutionContext.get("Statements")).addElement(_migDefaultStatement);
// execute query
    iter = new ResultSetIterator(__sJT_st.executeQuery());
}
    return iter;
}

//
// Utilities Generated by -migrate, the option for migrating SQLJ to JDBC
//

```

```

public static java.sql.Connection DefaultContext;

public static java.sql.Statement _migDefaultStatement;
public static java.util.Hashtable ExecutionContext = new
java.util.Hashtable();

static {
    try { java.sql.DriverManager.registerDriver(new
oracle.jdbc.OracleDriver()); }
    catch(java.sql.SQLException e) { /* No Driver available */ }

    //Initialize Statement Holder
    ExecutionContext.put("Statements", new java.util.Vector());
}
}

```

2.5 User-Defined ConnectionContext and Iterator

SQLJ `ConnectionContext` instances are mapped into `java.sql.Connection` instances. The connection context class—for example, `DeptContext`—is mapped into a class containing utilities for migrating connection management APIs and connection context type maps. Consider the following example program, `TestCtx.sqlj`.

The iterator class is mapped into a class with the same name to facilitate migration of iterator-related APIs, such as Type map support and field accesses. Iterator instances are mapped directly into `java.sql.ResultSet`.

2.5.1 Example

Consider the program `TestIter.sqlj`:

```

import java.sql.SQLException;

public class TestIter
{
    #sql public static context CtxISO with (typeMap="TestIterTypesMap");
    #sql public static iterator IterISO with (typeMap="TestIterTypesMap")
(StructISO objVal);

    public void foo(CtxISO ctx) throws java.sql.SQLException
    {
        IterISO iter;

```

```

#sql [ctx] iter={ SELECT * from struct_tab };
}
public static class StructISO implements java.sql.SQLData
{
    public String getSQLTypeName() throws SQLException
    { return (m_type!=null) ? m_type : "STRUCT_TY"; }
    public void readSQL(java.sql.SQLInput si, String type) throws
SQLException
    { date = si.readDate(); num = si.readInt(); m_type = type; }
    public void writeSQL(java.sql.SQLOutput so) throws SQLException
    { so.writeDate(date); so.writeInt(num); }
    private java.sql.Date date;
    private int num;
    private String m_type;
    public int getNum() { return num; }
    public java.sql.Date getDate() { return date; }

    public void setNum(int i) { num = i; }
    public void getDate(java.sql.Date d) { date = d; }
}
}

```

The command:

```
jpub -migrate TestIter.sqlj
```

produces the following `TestIter.java` file:

```

import java.sql.SQLException;

public class TestIter
{
    // SQLJ context declaration:
    // *****

    public static class CtxISO

```

```

{
    private static java.util.Map m_typeMap = null;
    static
    {
        m_typeMap = new java.util.Hashtable();
        java.util.ResourceBundle res = null;
        java.util.Properties props = null;
        java.util.Enumeration e = null;
        try
        {
            ...code loading the type map from TestIterTypesMap.properties ...
        }
    }
    public static java.sql.Connection getConnection(String url, String user,
        String password, boolean autoCommit) throws java.sql.SQLException
    {
        java.sql.Connection conn =
        java.sql.DriverManager.getConnection(url, user, password);
        conn.setAutoCommit(autoCommit);
        if (m_typeMap!=null) conn.setTypeMap(m_typeMap);
        return conn;
    }
}

```

The context class, `CtxISO`, is mapped into the inner class `CtxISO`, which contains type map initialization and `getConnection` methods. The type map, `TestIterTypesMap.properties`, is read during migration and is no longer used at runtime. The `getConnection` methods are convenient utilities for migrating `oracle.sqlj.runtime.OraclegetConnection` APIs.

Instances of `CtxISO` are mapped into `java.sql.Connection`, as shown by the method in the `TestIter.java` file:

```
public void foo(java.sql.Connection ctx) throws java.sql.SQLException
```

The query is executed with the `Connection` instance `ctx`.

The `IterISO` class is mapped into inner class `IterISO`, with field access method `objVal` and methods delegating `java.sql.ResultSet` APIs. The type map defined for `IterISO` is utilized in the field access method `objVal`. Instances of `IterISO` remain as instances of the inner class `IterISO`. The

IterISO class is instantiated with a `java.sql.ResultSet` instance, as shown by the statement:

```
iter = new TestIter.IterISO(__sJT_st.executeQuery());
```

2.6 ExecutionContext

An `ExecutionContext` instance is mapped into `java.sql.Statement`. In the migrated code, `ExecutionContext` property setters, such as `setQueryTimeout` and `setFetchSize`, are converted into statements, storing the property in the hash table named `ExecutionContext`. The `ExecutionContext` property getters, such as `getUpdateCount`, are converted into the corresponding `java.sql.Statement` APIs. Consider the program `TestEctx.sqlj`:

```
import sqlj.runtime.ExecutionContext;

public class TestEctx
{
    public void run() throws java.sql.SQLException
    {
        ExecutionContext execCtx = new ExecutionContext();
        execCtx.setQueryTimeout(1000);
        int raise = 10;
        #sql [execCtx] { UPDATE EMP SET sal = sal + :raise };
        int updateCount = execCtx.getUpdateCount();
        execCtx.close();
    }
}
```

The command:

```
jpub -migrate TestEctx.sqlj
```

migrates the preceding program into `TestEctx.java` as the following:

```
import java.sql.*;

public class TestEctx
{
    public void run() throws java.sql.SQLException
```

```

    {
        java.sql.Statement execCtx = null/* new ExecutionContext()*/;
        ExecutionContext.put("QueryTimeout", new Integer(1000));
        int raise = 10;
        // #sql [execCtx] { UPDATE EMP SET sal = sal + :raise };
        // *****
    {
        // declare temps
        oracle.jdbc.OraclePreparedStatement __sJT_st = null;
        String theSqlTS = "UPDATE EMP SET sal = sal + :1";
        __sJT_st =
(oracle.jdbc.OraclePreparedStatement)((oracle.jdbc.OracleConnection)TestECtx.De
faultContext).prepareStatement(theSqlTS);
        execCtx = __sJT_st;
        // set IN parameters
        __sJT_st.setInt(1,raise);
        // execute statement
        __sJT_st.executeUpdate();
    }
        int updateCount = execCtx.getUpdateCount();
        execCtx.close();
    }
    //
    // Utilities Generated by -migrate, the option for migrating SQLJ to JDBC
    //
    public static java.sql.Connection DefaultContext;
    public static java.sql.Statement _migDefaultStatement;
    public static java.util.Hashtable ExecutionContext = new
java.util.Hashtable();
    static {
        try { java.sql.DriverManager.registerDriver(new

```

```

oracle.jdbc.OracleDriver()); }
    catch(java.sql.SQLException e) { /* No Driver available */ }
    //Initialize Statement Holder
    ExecutionContext.put("Statements", new java.util.Vector());
}
}

```

2.7 Signature Conflicts

DefaultContext and user-defined contexts are mapped into the class java.sql.Connection. ExecutionContext is mapped into a java.sql.Statement. As an example, if one method involves DefaultContext and another involves java.sql.Connection, then two migrated methods with the same name may carry the same signature. When the mapping of DefaultContext, user-defined contexts, and ExecutionContext causes signature conflicts, then those constructors and methods with the signature change are removed and do not show up in the migrated code.

Consider the program TestConflicts.sqlj:

```

import sqlj.runtime.ref.DefaultContext;
import sqlj.runtime.ConnectionContext;
import java.sql.Connection;
public class TestConflicts
{
    ConnectionContext m_ctx;
    public TestConflicts(ConnectionContext ctx) throws
java.sql.SQLException
    {
        m_ctx = ctx;
    }
    public TestConflicts(Connection conn) throws java.sql.SQLException
    {
        m_ctx = new DefaultContext(conn);
        foo(m_ctx);
    }
}

```

```

public void foo(ConnectionContext ctx) throws java.sql.SQLException
{
    m_ctx = ctx;
}
public void foo(java.sql.Connection conn) throws java.sql.SQLException
{
    m_ctx = new DefaultContext(conn);
}
}

```

The command:

```
jpub -migrate TestConflicts.sqlj
```

produces the following `TestConflicts.java` file:

```

import java.sql.*;
public class TestConflicts
{
    java.sql.Connection m_ctx;
    /* migrate: Constructor removed due to signature conflict */
    public TestConflicts(Connection conn) throws java.sql.SQLException
    {
        m_ctx = conn;
        foo(m_ctx);
    }
    /* migrate: Method < foo> removed due to signature conflict */
    public void foo(java.sql.Connection conn) throws java.sql.SQLException
    {
        m_ctx = conn;
    }
    //
    // Utilities Generated by -migrate, the option for migrating SQLJ to JDBC
    //

```

```

public static java.sql.Connection DefaultContext;
public static java.sql.Statement _migDefaultStatement;
public static java.util.Hashtable ExecutionContext = new
java.util.Hashtable();
static {
    try { java.sql.DriverManager.registerDriver(new
oracle.jdbc.OracleDriver()); }
    catch(java.sql.SQLException e) { /* No Driver available */ }
    //Initialize Statement Holder
    ExecutionContext.put("Statements", new java.util.Vector());
}
}

```

Due to signature conflicts, two methods in `TestConflicts.sqlj` are removed:

```

public TestConflicts(ConnectionContext ctx) throws
java.sql.SQLException
public void foo(ConnectionContext ctx) throws java.sql.SQLException

```

After migration, method calls to a removed method are directed to the method with which that removed method conflicts. For instance, the following method call invokes `foo(ConnectionContext)` in `TestConflicts.java`:

```
foo(m_ctx);
```

2.8 Close Connections and Statements

In SQLJ programs, the `oracle.sqlj.runtime.Oracle.close` method is often used to close JDBC connections and statements. In the same way, the statement:

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

is migrated into:

```

if (DefaultContext!=null) DefaultContext.close();
if (ExecutionContext.get("Statements")!=null)
{
    java.util.Vector _stmts = (java.util.Vector)
    ExecutionContext.get("Statements");

```

```

    for( int _stmts_i = 0; _stmts_i < _stmts.size(); _stmts_i++)
        ((java.sql.Statement) _stmts.elementAt(_stmts_i)).close();
    }

```

The default connection, represented by `DefaultContext`, is closed. Note that if the default connection is accessed with the getter and setter methods, as specified by:

```
-migconn=accessors:<getter>,<setter>
```

then no closing statement regarding the default connection is generated from `Oracle.close`.

As shown in previous examples, each SQL statement is registered in the hash table called `ExecutionContext`, with the key "Statements". To release those registered statements, the `for` statement iterates over the statements opened, and closes those statements one by one.

Note that the preceding block is not thread safe. We recommend that the migrated code be manually modified so that connections and statements are closed individually, at the end of their usage. After the manual change, the preceding statement block is not needed.

2.9 Migration-Prompted Code Change

The following are some typical cases where you must perform manual code changes after `.sqlj` or `.java` programs are migrated into a SQLJ-free application, through the migration tool.

2.9.1. Signature Change

Use of `ExecutionContext` must be updated into use of `java.sql.Connection`. Calls to a method in a migrated program may need modification because of signature change during migration. However, if the caller program is also processed by the migration tool, then manual signature modification is not needed.

Consider a call to the method:

```
public void foo (sqlj.runtime.ConnectionContext cc);
```

in a Java program:

```
foo (new sqlj.runtime.ref.DefaultContext(conn));
```

If the program containing `foo` has been migrated by the JPublisher tool, but the Java program that is invoking `foo` has not been, then you must modify that Java program manually to accommodate the signature change in `foo`. For example, converting the method call into:

```
foo (conn);
```

2.9.2 The Default Connection

The implicit use of the SQLJ default `DefaultContext` instance may need to be explicitly assigned, expressed with assignments of `java.sql.Connection` to the migrated program. Consider two programs, `Caller.java` and `Callee.sqlj`:

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

public class Caller
{
    public static void main(String[] args) throws java.sql.SQLException
    {
        Oracle.connect("jdbc:oracle:oci8:@", "scott", "tiger");
        Callee.callee();
    }
}
```

```
import sqlj.runtime.ResultSetIterator;

public class Callee
{
    public static void callee() throws java.sql.SQLException
    {
        ResultSetIterator rsi;
        #sql rsi = { select * from emp };
        System.out.println("done");
    }
}
```

Normally, you would migrate the two programs using the command:

```
% jpub -migrate Caller.java Callee.sqlj
```

However, the migrated program throws a `NullPointerException` at runtime.

```
% java Caller
Exception in thread "main" java.lang.NullPointerException
    at Callee.callee(Callee.java:15)
```

at Caller.main(Caller.java:7)

The `NullPointerException` is caused by the uninitialized default connection in the Callee program. Let us examine the generated code for both programs.

```
public class Caller
{
    public static void main(String[] args) throws java.sql.SQLException
    {
        Caller.DefaultContext=
        (Caller.DefaultContext!=null?
        Caller.DefaultContext
        ;java.sql.DriverManager.getConnection("jdbc:oracle:oci8:@",
"scott", "tiger"));
        Callee.callee();
    }
    public static java.sql.Connection DefaultContext;
    ....
}
```

```
public class Callee
{
    public static void callee() throws java.sql.SQLException
    {
        Callee.ResultSetIterator rsi;
        // #sql rsi = { select * from emp };
        // *****
11    {
12        // declare temps
13        oracle.jdbc.OraclePreparedStatement __sJT_st = null;
14        String theSqlTS = "select * from emp";
15        __sJT_st =
```

```

(oracle.jdbc.OraclePreparedStatement)((oracle.jdbc.OracleConnection)Callee.DefaultContext).prepareStatement(theSqlTS);
16     _migDefaultStatement = __sJT_st;
17     if (ExecutionContext.get("Statements")!=null)
18         ((java.util.Vector)
ExecutionContext.get("Statements")).addElement(_migDefaultStatement);
19     // execute query
20     rsi = new Callee.ResultSetIterator(__sJT_st.executeQuery());
21 }
22 System.out.println("done");
23 }
24
25 public static java.sql.Connection DefaultContext;
...
}
}

```

The preceding generated code shows that there is a broken communication between the two programs, regarding the default connection. The Caller program initializes its `DefaultContext` variable; the Callee program attempts to use its own `DefaultContext` variable to evaluate the SQL statement, which results in a `NullPointerException` because `Callee.DefaultContext` is apparently null.

These two programs would be able to share the same default connection in the original SQLJ model, where the default `sqlj.runtime.ref.DefaultContext` instance provides the default connection to all the programs at runtime.

After migration, you must explicitly the default connection for each program, or you must specify a shared connection at the migration time. Subsequently, there are two approaches to fix the problem observed above.

First, set `Callee.DefaultContext` explicitly in Caller. For instance, rewriting the generated `Caller.java` as follows allows the preceding test to run through.

```
public class Caller
```

```

{
    public static void main(String[] args) throws java.sql.SQLException
    {
        Caller.DefaultContext=
        (Caller.DefaultContext!=null?
        Caller.DefaultContext
        ;java.sql.DriverManager.getConnection("jdbc:oracle:oci8:@",
        "scott", "tiger"));
        Callee.DefaultContext = DefaultContext;
        Callee.callee();
    }
    public static java.sql.Connection DefaultContext;
    ....
}

```

Second, specify a common `java.sql.Connection` variable to be shared by all the programs being migrated. Re-do the migration as follows.

```

% mv Caller.java.migrate Caller.java
% rm Callee*.class
% jpub -migrate -migconn=Caller._defConn Caller.java Callee.sqlj
% java Caller
done

```

The first command recovers the `Caller.java` program from the first migration attempt. Remember, a `.java` file is renamed into a `.java.migrate` file by the migration command.

The second command removes the `Callee$ResultSetIterator.class` file, among other generated files, to avoid conflicts between `sqlj.runtime.ResultSetIterator` and the inner class `Callee$ResultSetIterator`, during the migration.

The third command migrates the two programs, specifying that the two programs share the same default connection variable, `Caller._defConn`.

The last command ran the test successfully.

2.9.3 Property and Type Map Files

As previously noted, after migration the connection property file—for example, `connect.properties`—used by the `oracle.sqlj.runtime.Oracle` APIs is no longer read at runtime. The properties are read by the migration tool and hard-coded in the migrated code. If the original SQLJ application depends on the runtime connection property file, then you must adjust the migrated code in the way `oracle.sqlj.runtime.Oracle` APIs are converted.

Similarly, the type map file for the SQLJ context and iterator is no longer read at runtime. The type map entries are read during migration and hard-coded in the migrated code. Corresponding adjustment is required if the type map entries differ between the translation time and the runtime.

2.9.4 Additional .class Files

Compared with the original SQLJ translator, the migration tool may produce additional `.class` files from the same `.java` or `.sqlj` file—notably, the `.class` file for the inner class, `ResultSetIterator`. When the `.java` or `.sqlj` program contains references to `sqlj.runtime.ResultSetIterator` and its subclasses, then the migration tool generates the inner class `ResultSetIterator`. For instance, the inner class `TestRSI$ResultSetIterator` is generated by the command:

```
jpub -migrate TestRSI.sqlj
```

However, if the option `-migrsi` specifies another inner class to represent `ResultSetIterator`, then the additional inner class is not generated. For instance, the command:

```
jpub -migrate -migrsi=Test.CommonRSI TestRSI.sqlj
```

does not generate the additional inner class in `TestRSI`. The inner class, `CommonRSI`, may be generated for `Test` if `Test.java` or `Test.sqlj` is, in turn, translated as the result of the preceding command.

If the original SQLJ application is sensitive to the `.class` files generated, then you must make corresponding adjustments to accommodate the additional `.class` files that the migration tool may generate.

Problems can occur when the migration tool runs more than once for a `.java` or `.sqlj` program that generates additional inner classes, and those classes are visible from the `classpath`. The solution is to remove the generated class when the migration tool gives errors on a repeated run. For instance, running the migration tool against `TestRSI.sqlj` the second time encounters a compilation error, as the following shows.

```
% jpub -migrate TestRSI
```

```

% jpub -migrate TestRSI
TestRSI.sqlj:12.5-12.39: Error: Return type incompatible with SELECT
statement: TestRSI$ResultSetIterator is not an iterator type.

Total 1 error.

% ls TestRSI*.class

TestRSI$EmpIter.class      TestRSI$ResultSetIterator.class

TestRSI.class

```

The second `jpub` command throws the error, because the class file `TestRSI$ResultSetIterator.class` interferes with `sqlj.runtime.ResultSetIterator` actually referenced by the `TestRSI.sqlj` program. Removing the generated `.class` files allows the migration tool to run successfully.

2.9.5. Thread Safety

The migrated code is not thread-safe if the original SQLJ program exercises `ExecutionContext` APIs. In the migrated code, all `ExecutionContext` instances are mapped into static variables, `ExecutionContext` of type `java.util.Hashtable`, which may be simultaneously manipulated by various threads. You must inspect the operations around `ExecutionContext` in the migration code to make sure thread issues are resolved appropriately. Consider the following statement generated in the `TestRSI.sqlj` example.

```

    if (ExecutionContext.get("Statements")!=null)
        ((java.util.Vector)
            ExecutionContext.get("Statements")).addElement(_migDefaultStatement);

```

This code stores the statement used in the query into the `ExecutionContext` variable, in order for that statement to be closed by the migrated code from `oracle.sqlj.runtime.Oracle` close APIs. If the migrated code is to be run in a multithreaded environment, then you must make sure that the preceding statement is protected—for instance, by rewriting it as a synchronized block:

```

synchronized (ExecutionContext)
{
    if (ExecutionContext.get("Statements")!=null)
        ((java.util.Vector)
            ExecutionContext.get("Statements")).addElement(_migDefaultStatement);
}

```

2.7.6 More Examples

Together with the migration tool, we also furnish the SQLJ demos distributed with the SQLJ 9i release and show how they can be migrated to JDBC. The README.txt file details the command and expected result for each demo. Typically the sqlj or javac command is replaced by the migration command jpub -migrate. The rest of the demo remains unchanged.

CONSLUSION

You have two choices for coping with SQLJ desupport in Oracle Database 10g and later releases, and in Oracle Application Server 10g (10.1.3) and later releases. The first option is to continue creating code with SQLJ, using either the SQLJ 9i translator or JPublisher-provided SQLJ translation in Oracle Database 10g. The second option is to migrate SQLJ programs to pure JDBC programs, with or without the migration assistance from JPublisher.

To use the JPublisher-provided migration tool, we recommend the following procedures.

```
#If necessary, specify -migconn to provide centralized connection management.  
#Run jpub -migrate to migrate the SQLJ program  
#Inspect and adjust generated .java program for potential issues.  
#Specify -migrsi and re-run the jpub -migrate command, to customize iterator  
migration  
#Customize the treatment of oracle.sqlj.runtime.Oracle and  
sqlj.runtime.ExecutionContext APIs for thread-safety and performance  
concerns.
```



Oracle SQLJ Roadmap
July 2004
Author: Quan Wang
Contributing Authors:
Ekkehard Rohwedder
Kuassi Mensah
Oracle Corporation
World Headquarters
500 Oracle Parkway
Redwood Shores, CA 94065
U.S.A.

Worldwide Inquiries:
Phone: +1.650.506.7000
Fax: +1.650.506.7200
www.oracle.com

Oracle Corporation provides the software
that powers the internet.

Oracle is a registered trademark of Oracle Corporation. Various
product and service names referenced herein may be trademarks
of Oracle Corporation. All other product and service names
mentioned may be trademarks of their respective owners.

Copyright © 2000 Oracle Corporation
All rights reserved.