Oracle® Database Lite

Developer's Guide for Java

10*g* (10.0.0)

Part No. B13811-01

June 2004

ORACLE®

Oracle Database Lite Developer's Guide for Java 10*g* (10.0.0)

Part No. B13811-01

Contents

3 JDBC Programming

4 Java Stored Procedures and Triggers

Send Us Your Comments

Oracle Database Lite Developer's Guide for Java 10*g* (10.0.0)

Part No. B13811-01

Oracle Corporation welcomes your comments and suggestions on the quality and usefulness of this publication. Your input is an important part of the information used for revision.

- Did you find any errors?
- Is the information clearly presented?
- Do you need more information? If so, where?
- Are the examples correct? Do you need more examples?
- What features did you like most about this manual?

If you find any errors or have any other suggestions for improvement, please indicate the title and part number of the documentation and the chapter, section, and page number (if available). You can send comments to us in the following ways:

- Electronic mail: helplite_ca@oracle.com
- FAX: (650) 506-7355. Attn: Oracle Database Lite
- Postal service:

  Oracle Corporation
  Oracle Database Lite Documentation
  500 Oracle Parkway, Mailstop 1op2
  Redwood Shores, CA 94065
  U.S.A.

If you would like a reply, please give your name, address, telephone number, and electronic mail address (optional).

If you have problems with the software, please contact your local Oracle Support Services.

Preface

This preface introduces you to the *Oracle Database Lite Developer's Guide for Java*, discussing the intended audience, documentation accessibility, and structure of this document.

Intended Audience

This manual is intended for application developers as the primary audience and for database administrators who are interested in application development as the secondary audience.

Documentation Accessibility

Our goal is to make Oracle products, services, and supporting documentation accessible, with good usability, to the disabled community. To that end, our documentation includes features that make information available to users of assistive technology. This documentation is available in HTML format, and contains markup to facilitate access by the disabled community. Standards will continue to evolve over time, and Oracle is actively engaged with other market-leading technology vendors to address technical obstacles so that our documentation can be accessible to all of our customers. For additional information, visit the Oracle Accessibility Program Web site at

`http://www.oracle.com/accessibility/`

Accessibility of Code Examples in Documentation JAWS, a Windows screen reader, may not always correctly read the code examples in this document. The conventions for writing code require that closing braces should appear on an otherwise empty line; however, JAWS may not always read a line of text that consists solely of a bracket or brace.

Structure

This guide includes the following topics:

- Chapter 1, "Overview"

  This chapter provides an overview of native Java applications for developers.

- Chapter 2, "Application Development"

  This chapter describes how to develop and test Java applications.

- Chapter 3, "JDBC Programming"

This chapter discusses the Oracle Database Lite support for JDBC programming.

- Chapter 4, "Java Stored Procedures and Triggers"

  This chapter describes how to use Java stored procedures and triggers within the Oracle Database Lite relational model.

- Chapter 5, "Java Support on Windows CE"

  This chapter describes Java support for Windows CE devices using the Java Interface.

- Appendix A, "Stored Procedure Tutorial"

  This appendix demonstrates how to create a Java stored procedure and trigger.

- Appendix B, "Sample Programs"

  This appendix provides instructions for using the sample Java programs provided with Oracle Database Lite.

1 Overview

This chapter provides an overview of native Java applications for developers. Topics include:

1.1 Concepts

Oracle Database Lite facilitates the development, deployment, and management of offline mobile database applications for a large number of mobile users. An offline mobile application is an application that can run on mobile devices without requiring constant connectivity to a server. An offline database application requires a local database on the mobile device whose content is a subset of data that is stored in the enterprise data server. The modifications made to the local database by the application are occasionally reconciled with server data. The technology used for reconciling changes between the mobile database and the enterprise database is known as data synchronization.

For more information about Oracle Database Lite concepts, refer the *Oracle Database Lite Developer's Guide*.

1.2 Application Development Steps Overview

This section provides an overview of the Java application development process for mobile applications. Topics include:

- Setup Enterprise Data Subset Definition
- Package the Application
- Publish the Application
- Test the Application

1.2.1 Setup Enterprise Data Subset Definition

The enterprise data subset definition setup process can be accomplished by performing tasks in the phases listed below.

1. Mobile application developers must first define the subset of enterprise data for users of mobile applications.

2. The enterprise data subset is defined as a publication, which is instantiated as the Oracle Database Lite schema on the mobile client.
3. The primary data subsetting mechanism of a publication is the publication item, which is a parameterized query that defines the data subset, based on specified parameter values.
4. When a publication is instantiated, a snapshot is created for each publication item (with variables bound to values) in Oracle Database Lite.

Packaging Applications

After completing tasks under phases listed above, mobile application developers can develop mobile applications against Oracle Database Lite. Upon completion of application development and testing, developers can package applications to present them in a format which is ready to be published to Oracle Database Lite.

Provisioning Applications to Users

The provisioning process involves assigning privileges for application usage and associated data subsets to users. To provision an application to a user, the Mobile Server Administrator creates a subscription for a user, from the publication that is associated with an application.

Creating Application Subscriptions

To create a subscription, the Mobile Server Administrator must assign values for the subscription parameters of the publication. These values collectively determine the enterprise data subset for a user.

Generating Database Schema

When a user logs into the Mobile Server, the Mobile Server installs Oracle Database Lite on the client machine and creates a database or schema for each subscription that is associated with applications that are provisioned to a user. At this stage, database schema are populated with tables and rows that are retrieved from the server, based on subscription definitions. The Mobile Server also installs mobile applications on the client machine.

1.2.2 Develop the Application

Using the Mobile Development Kit, you can develop Java applications. You can build Java applications using Java Servlets, Java Server Pages (JSP), and Java Beans. After creating your Java applications, you must perform the following tasks.

1. Create database objects in Oracle Database Lite.
2. Write the application code.
   a. Set the `CLASSPATH` to include required libraries.
   b. Compile the Java Servlet and JavaBean.
   c. Install the JSP.
3. Compile the application.
4. Define the application and register the servlet.
   a. Start the Packaging Wizard in debug mode. Using the Command Prompt, enter the following:

      ```
      cd <Oracle_home>\mobile\sdk\bin
      ```

```
wtgpack -d
```

The Packaging Wizard appears. Using the Packaging Wizard, create a new application, select a platform, specify your Java application settings, select application files, compile JSP files, select your Java application servlets, and specify registry settings. For more information about the Packaging Wizard, see the *Oracle Database Lite Tools and Utilities Guide*.

5. Run the application.

   To run your Java application, start the Mobile Client Web Server on the development computer. Using your Java browser, access your Java application and connect to the application's URL.

   The default port for the Mobile Client Web Server is 7070. To configure the port that will be used by the Mobile Client Web Server, change the port entry in the `webtogo.ora` file.

   ```
   <Oracle_home>\mobile\sdk\bin\webtogo.ora
   ```

   For information on how to edit the `webtogo.ora` file, see Section 11.3, "Editing the webtogo.ora File," in the *Oracle Database Lite Administration and Deployment Guide*.

   For additional information about configuration parameters in the `webtogo.ora` file, see Appendix B, "Mobile Server Configuration Paramters" in the *Oracle Database Lite Administration and Deployment Guide*.

   a. Start the Mobile Client Web Server. Using the Command Prompt, enter the following.

      ```
      cd <Oracle_home>\mobile\sdk\bin
      ```

      ```
      wtgdebug.exe
      ```

      The Mobile Client Web Server starts and reports which servlets are loaded. If your servlets contain any `System.out.printIn()` statements, the messages appear in this window.

   b. Start your web browser and connect to the following URL.

      ```
      http://your_machine:7070/
      ```

      The browser displays a list of applications that are currently known to the Web-to-Go system.

   c. Click your Java application.

The Mobile Development Kit for Web-to-Go always uses Oracle Database Lite as the development database. You can create one yourself using the `CREATEDB` statement.

The Mobile Development Kit for Web-to-Go also uses a web server that is referred to as the Mobile Client Web Server.

During the development phase, your Java application's servlet stores application items in Oracle Database Lite. During the deployment phase, you must copy the database objects from Oracle Database Lite to the Oracle database.

1.2.3 Package the Application

To prepare Java applications for publishing to the Mobile Server, you must package these applications using the Packaging Wizard. The following steps enable you to publish your Java applications.

1. Define your Java application using the Packaging Wizard.

a. To define Java applications, start the Packaging Wizard using the Command Prompt and enter the following.

```
cd <Oracle_home>\mobile\sdk\bin

wtgpack
```

b. Choose **Edit an existing application** and select your Java application from the list displayed.

c. Review application information and settings provided under the platform, application, files, servlets, and database tabs.

2. Using the Database tab, define the application connection to the Oracle database.
3. Using the Roles tab, define application roles.
4. Using the Snapshots tab, define snapshots for your Java applications. To define snapshots, import table definitions from the development database.
5. Using the Sequences tab, define sequences that your Java application will use in offline mode. At a later stage, you can create the actual sequences in the Oracle database. During synchronization, Web-to-Go automatically creates a local copy of your Java application's sequence on your client.
6. Create SQL files for your Java application. After you specify sequences, the Application Definition Completed dialog appears. Using this dialog, you can choose to generate SQL scripts for database objects.
7. You are now ready to package your Java application. Using the Application Definition Completed dialog, you can package your Java application into a jar file.

1.2.4 Publish the Application

After packaging your Java applications, you are ready to publish them. The following steps enable you to publish your Java applications.

1. Create the 'Table Owner' account. The 'Table Owner' is effectively the database user who will own the Java application that you just packaged.
2. Create database objects in the Oracle database by running the SQL master script. The script creates your Java application's table and its corresponding sequence.
3. Using the SQL script, start the Mobile Server.
4. Login to the Mobile Server and start the Mobile Manager.
5. Upload the jar file containing your Java application.

1.2.5 Test the Application

Before you can test your Java application, you must administer your Java applications by creating users, setting application properties, granting user access to applications, and defining snapshot template values for a user's snapshot template variables.

After administering Java applications as indicated above, you must install the Mobile Client for Web-to-Go using the Mobile Client Setup program, login to the Mobile Client for Web-to-Go, and synchronize the Mobile Client for Web-to-Go.

For detailed information on how to develop, package, publish, and test your applications, refer Chapter 2, "Application Development".

1.3 Configuring the Development System

This section discusses how to configure the development system. Topics include:

1.3.1 Java Development Kit (JDK)

As part of the development system's configuration, install the Java Development Kit (JDK) 1.3.1 or higher.

1.3.2 Install and Configure the Oracle Database or Enterprise Database

Install the appropriate Oracle Database or Enterprise Database. For more information, refer the *Oracle Database Lite Installation and Configuration Guide for Windows NT/2000/XP*.

1.3.3 Install the Mobile Server

For more information on how to install the Mobile Server, refer the *Oracle Database Lite Installation and Configuration Guide for Windows NT/2000/XPs*.

1.3.4 Configure the Mobile Server

For more information, refer the *Oracle Database Lite Installation and Configuration Guide for Windows NT/2000/XP.*

1.3.5 Install the Mobile Development Kit

For more information on how to install the Mobile Development Kit, refer the *Oracle Database Lite Installation and Configuration Guide for Windows NT/2000/XP.*

Note: Section 2.2 to Section 2.5 are only required when you synchronize data between client and server databases.

2

Application Development

This chapter describes how to develop and test Java applications. Topics include:

2.1 Oracle Database Lite Java Support

This section describes Java interfaces and tools supported by Oracle Database Lite. Topics include:

2.1.1 Java Datatypes

Oracle Database Lite performs type conversions between Java and Oracle datatypes as indicated by the following table. Table 2–1 lists the Java datatypes and the corresponding SQL datatypes that result from the type conversion.

Table 2–1 Datatype Conversions

Java Datatype	SQL Datatype
byte[], byte[][], Byte[]	BINARY, RAW, VARBINARY, BLOB
boolean, Boolean	BIT
String, String[]	CHAR, VARCHAR, VARCHAR2, CLOB
short, short[], short[][], Short, Short[]	SMALLINT
int,int[], int[][], Integer, Integer[]	INT
float, float[], float[][], Float, Float[]	REAL
double, double[], double[][], Double, Double[]	DOUBLE, NUMBER (without precision)
BigDecimal, BigDecimal[]	NUMBER(n)

Table 2–1 (Cont.) Datatype Conversions

Java Datatype	SQL Datatype
java.sql.Date, java.sql.Date[]	DATE
java.sql.Time, java.sql.Time[]	TIME
java.sql.Timestamp, java.sql.Timestamp[]	TIMESTAMP
java.sql.Connection	Default JDBC connection to database

2.1.2 Java Tools

Oracle Database Lite provides tools to manage Java development. Table 2–2 lists these Java tools and their descriptions.

Table 2–2 Java Tools

Tool	Description
`loadjava`	Loads Java classes into Oracle Database Lite.
`dropjava`	Removes Java classes from Oracle Database Lite.

2.1.2.1 loadjava

The `loadjava` utility automates the task of loading Java class and resource files into Oracle Database Lite. Using `loadjava`, you can load class and resource files individually, or in ZIP or JAR archives.

After you load the class, create a call specification for the methods in the class that you want to make accessible to SQL statements. To create a call specification for a stored procedure that returns a value, use the `SQL CREATE FUNCTION` statement. If the stored procedure does not return a value, use the `CREATE PROCEDURE` statement.

For unloading classes, Oracle Database Lite provides `dropjava`, which works in a similar manner as the `loadjava` utility.

2.1.3 Oracle Database Lite Java Development Environment

The following tools facilitate Java development for Oracle Database Lite.

- Oracle Developer 2.1 supports Java stored procedures written in Oracle Database Lite by users.
- Oracle JDeveloper also supports Java stored procedures in Oracle Database Lite, and includes features designed specifically to help develop and deploy Java stored procedures.

2.1.3.1 Environment Setup

This section describes how to set up your development environment to create Oracle Database Lite applications. To develop Java applications, you must have the Sun Microsystems Java Development Kit (JDK), version 1.3.1 (or higher).

To enable Oracle Database Lite to work with the JDK, set your `PATH` and `CLASSPATH` environment variables, after you install Oracle Database Lite. Depending on the version of JDK that you are using, the `PATH` and `CLASSPATH` settings may vary. The following sections summarize these variations.

If your environment includes a CLASSPATH user variable before you install Oracle Database Lite, and the user variable does not include the CLASSPATH system variable (is not specified as CLASSPATH=...;%CLASSPATH%), you must modify the CLASSPATH user variable to include the **OLITE40.JAR** file in the <Oracle_home>\mobile\classes directory.

> **Note:** All command prompt windows must be closed and reopened to reflect changes made to your CLASSPATH.

Setting Variables for JDK 1.3.1

If you are using JDK 1.3.1, the directory with the JDK 1.3.1 Java compiler (**javac.exe**) should be in the PATH variable before any other directories that contain other Java compilers.

Add the directory that contains the Classic Java Virtual Machine (JVM) shared library, **jvm.dll**, to the PATH. **jvm.dll** should be in your JDK_Home\jre\bin\classic directory.

For example,

```
set PATH=C:\JDK_Home\bin;c:\JDK_Home\jre\bin\classic

set CLASSPATH=c:\JDK_Home\jrc\lib\rt.jar;c:\<Oracle_
home>\Mobile\classes\olite40.jar
```

As an alternative to using the Classic JVM, you can use the HotSpot JVM. HotSpot is a JDK add on module provided by Sun Microsystems. HotSpot is available from the Sun Microsystems Web site.

After installing HotSpot, set your PATH as given below.

```
set PATH=c:\jdk1.3.1\bin;c:\jdk1.3.1\jre\bin\hotspot;%PATH%
```

In the example above, your installation of the JDK and HotSpot is on Drive C. You should verify the location of your installation before amending your PATH statement. To test whether your system is set up correctly, run the Java examples in the <Oracle_home>\Mobile\Sdk\Samples\JDBC directory.

2.2 Java Development Tools

To write and debug Java programs, you can use any Java development tool. However, you must ensure that you set the CLASSPATH and PATH correctly.

2.3 Developing and Testing the Application

Before synchronizing your database with the Mobile Server, you must create a seed database by publishing a dummy application and synchronize it to the client machine. This creates a sample database with the correct schema and table definitions. If you are not synchronizing your database, you should first create the database with sample tables and data.

2.4 Packaging the Application

At this stage, you must package and publish the application.

To package Java applications using the Packaging Wizard, refer the *Oracle Database Lite Tools and Utilities Guide*.

To publish the application, refer to Section 4.6, "Uploading Applications to the Mobile Server Repository" in the *Oracle Database Lite Administration and Deployment Guide*.

2.5 Testing

The publishing phase can be termed as the testing phase for your applications. To test your Java applications, publish the application using the Packaging Wizard. For more information, refer Section 4.6, "Uploading Applications to the Mobile Server Repository" in the ***Oracle Database Lite*** *Administration and Deployment Guide*.

2.6 MSync/OCAPIs/mSyncCom

For more information, refer the *Java mSync API Specification*.

3

JDBC Programming

This chapter discusses the Oracle Database Lite support for JDBC programming. It includes the following topics:

3.1 JDBC Compliance

JDBC is an application programmer's interface for accessing relational databases from Java programs. Oracle Database Lite supplies a native JDBC driver that allows Java applications to communicate directly with Oracle Database Lite's object relational database engine. Oracle Database Lite's implementation of JDBC complies with JDBC 1.22. In addition, Oracle Database Lite provides certain extensions specified by JDBC 2.0. Oracle Database Lite's extensions are compatible with the Oracle8*i* JDBC implementation. For a complete JDBC reference, see the Sun Microsystems web site.

3.2 JDBC Environment Setup

If you are using the client/server model, include the **olite40.jar** in the 'system' classpath on the server machine. Include the '`user`' classpath on the client machine.

For more information on how to start the Multiuser Oracle Database Lite Database Service, see Section 2.2.1.2, "Starting a Multi-User Oracle Database Lite Database Service," in the *Oracle Database Lite Developer's Guide*.

3.3 Connect to Oracle Database Lite

JDK 1.3.x or higher is required to connect to Oracle Database Lite.

There are three ways to connect to Oracle Database Lite.

Oracle Database Lite supports two types of drivers namely, Type 2 and Type 4. The Type 2 driver requires a native code on the client side. The Type 2 driver interfaces with the Oracle Database Lite ODBC driver through this native code.

Note: On the Windows platform, the Type 2 driver uses the `oljdbc40.dll`.

The Type 4 JDBC driver is a pure Java driver and uses the Oracle Database Lite network protocol to communicate with the Oracle Database Lite service. Before using this driver, ensure that you start Oracle Database Lite. A Java applet can use the Type 4 JDBC driver.

Type 2 Driver Connection URL Syntax

```
DriverManager.getConnection("jdbc:polite@URL_Name:100:polite","system","admin");
```

This syntax is used to make a direct connection to a database on a client machine. Enter the URL definition as given below.

```
jdbc:polite@host:port:dsn
```

The following arguments can be made as part of the URL clause or as a separate key-value pair. There may be none or many occurrences of the key-value pair which provide additional information to the driver. All information that can be specified in the URL can be specified as a key-value pair. The information that is specified as a key-value pair always overrides the information that is specified in the URL.

The URL interpretation and key-value pair options for each argument are described in the following table.

Argument	Description
`jdbc`	Identifies the protocol as JDBC.
`polite`	Identifies the subprotocol as `polite`.
uid / pwd	The optional user ID and password for Oracle Database Lite. If specified, this overrides the specification of a user ID and password. If the database is encrypted, you must include the password in the key-value pair.
dsn	Identifies the data source name (DSN) entry in the **odbc.ini** file. This entry contains all the necessary information to complete the connection to the server. **Note**: For a JDBC program, you need not create a DSN if you have supplied all the necessary values for the data directory and database through key=value pairs. On the windows platform, you can use the ODBC administrator to create a DSN. For more information, refer the *Oracle Database Lite Developer's Guide*.
DataDirectory=	Directory in which the `.odb` file resides.
Database=	Name of database as given during its creation.
IsolationLevel=	Transaction isolation level: READ COMMITTED, REPEATABLE READ, SERIALIZABLE or SINGLE USER. For more information on isolation levels, see the *Oracle Database Lite Developer's Guide*.
Autocommit=	Commit behavior, either ON or OFF.
CursorType=	Cursor behavior: DYNAMIC, FORWARD ONLY, KEYSET DRIVEN or STATIC. For more information on cursor types, see the *Oracle Database Lite Developer's Guide*.
UID=	User name

Argument	Description
PWD=	Password

Example

```
String ConnectMe=("jdbc:polite:SCOTT/tiger:polite;DataDirectory=<Oracle_
home>;Database=polite;IsolationLevel=SINGLE
USER;Autocommit=ON;CursorType=DYNAMIC")


try
   {Class.forName("oracle.lite.poljdbc.POLJDBCDriver")
   Connection conn = DriverManager.getConnection(ConnectMe)
   }
catch (SQLException e)
{



   ...
}
```

Type2 Client/Server Driver Connection URL Syntax

jdbc:polite[:*uid* / *pwd*]@[*hosf*]:[*porf*]:*dsn* [;*key=value*]*

The URL can be used to connect to the Oracle Database Lite service using the Type 2 JDBC driver. For more information on how to install and start the Multiuser Oracle Database Lite Database Service, refer to Section 2.2.1.2, "Starting a Multi-User Oracle Database Lite Database Service," in the *Oracle Database Lite Developer's Guide*.

Argument	Description
host	The name of the machine that hosts Oracle Database Lite and on which the Oracle Database Lite service **olsv2040.exe** runs. This host name is optional. If omitted, it defaults to the local machine on which the JDBC application runs.
port	The port number at which the Oracle Database Lite service listens. The port number is optional and if omitted defaults to port "100".

Example

An example of this type of connection is given below.

```
try {
Connection conn = DriverManager.getConnection(
   "jdbc:polite@yourhostname
        ;DataDirectory=<Oracle_home>
        ;Database=polite
        ;IsolationLevel=SINGLE USER
        ;Autocommit=ON
        ;CursorType=DYNAMIC", "Scott", "tiger")
}

catch (SQLException e)
{
```

You should enclose the `getConnection` method in a try-catch block to intercept any SQL exception thrown during the connection attempt. You can insert an exception handling statement in the catch block.

Type4 (Pure Java) Driver Connection URL Syntax

The URL syntax for the type4 driver is given below.

```
jdbc:polite4[:uid/pwd]@[host]:[port]:dsn[;key=value]*
```

The parameter `4` indicates that the type4 driver is being used. For the rest of the parameters, see the definitions of those parameters for the type2 driver as described above.

Note: The URL works with the Oracle Database Lite service only. For more information on how to start and stop the Oracle Database Lite service, refer the *Oracle Database Lite Developer's Guide*.

3.4 Executing Java Stored Procedures from JDBC

After creating a Java stored procedure, you can execute the procedure from a JDBC application by performing the following steps.

- By passing an `SQL SELECT` string that executes the stored procedure to the `Statement.executeQuery` method.
- By using a JDBC `CallableStatement`.

Note: For more information on creating stored procedures, see Chapter 4, "Java Stored Procedures and Triggers".

The `executeQuery` method executes table-level and row-level stored procedures. `CallableStatement` currently only supports execution of table-level stored procedures.

3.4.1 Using the executeQuery Method

To call a stored procedure using the `executeQuery` method, first create a `Statement` object, which you assign the value returned by the `createStatement` method of the current connection object. You then execute the `Statement.executeQuery` method, by passing the `SQL SELECT` string that invokes the Java stored procedure.

For example, suppose you want to execute a row-level procedure `SHIP` on a table named `INVENTORY` with the argument value stored in the variable *q*. The variable `p` contains the product ID for the product (row) for which you want to execute the stored procedure.

```
int res = 0;
Statement s = conn.createStatement();
ResultSet r = s.executeQuery("SELECT SHIP(" + q + ")" +
   "FROM INVENTORY WHERE PID = " + p);
if(r.next()) res = r.getInt(1);
r.close();
s.close();
return res;
```

If you need to execute a procedure repeatedly with varying parameters, use `PreparedStatement` instead of `Statement`. Because the SQL statements in a `PreparedStatement` are pre-compiled, `PreparedStatements` execute more efficiently. Additionally, a `PreparedStatement` can accept `IN` parameters, represented in the statement with a question mark `(?)`. However, if the `PreparedStatement` takes a "`long`" type parameter, such as `LONG` or `LONG RAW`, you must bind the parameter using the `setAsciiStream`, `setUnicodeStream`, or `setBinaryStream` methods.

In the preceding example, if the procedure `SHIP` updates the database and the isolation of the transaction that issues the above query is `READ COMMITTED`, you must append the `FOR UPDATE` clause to the `SELECT` statement, as given below.

```
"SELECT SHIP(" + q + ")" +
   FROM INVENTORY WHERE PID = " +
   p + "FOR UPDATE");
```

3.4.2 Using a Callable Statement

To execute the stored procedure using a callable statement, create a `CallableStatement` object and register its parameters as given below.

```
CallableStatement cstmt = conn.prepareCall(
   "{?=call tablename.methodname() }");
cstmt.registerOutParameter(1, ...);
cstmt.executeUpdate();
cstmt.get..(1);
cstmt.close();
```

The following restrictions apply to JDBC callable statements.

- JDBC callable statements can only execute table-level stored procedures.
- Both IN and OUT parameters are supported. However, not all Java datatypes can be used as OUT parameters. For more information, see Chapter 4, "Java Stored Procedures and Triggers".
- Procedure names correspond to the Java method names, and are case-sensitive.
- As with prepared statements, if the callable statement has a "`long`" type, such as: LONG, LONG VARBINARY, LONG VARCHAR, LONG VARCHAR2, or LONG RAW, you must bind the parameter using the `setAsciiStream`, `setUnicodeStream`, or `setBinaryStream` methods.

> **Note:** When no longer needed, you should reclaim system resources by closing JDBC objects, such as `Resultset` and `Statement` objects.

3.5 Oracle Database Lite Extensions

The Oracle Database Lite JDBC driver supports JDBC 1.22 and provides extensions that support certain features defined in JDBC 2.0. The extensions include support for BLOB (large binary object) and CLOB (large character object) datatypes and scrollable result sets. The Oracle Database Lite JDBC extensions are compatible with the Oracle8i JDBC implementation. However, Oracle Database Lite does not support the Oracle8i JDBC datatype extensions, `Array`, `Struct`, or `REF`.

This section lists and describes the Oracle Database Lite datatype and data access extensions. For details regarding function syntax and call parameters, see the Sun Microsystems Java 2 specification at the Sun Javasoft website.

3.5.1 Datatype Extensions

BLOBs and CLOBs store data items that are too large to store directly in a database table. Rather than storing the data, the database table stores a locator that points to the location of the actual data. BLOBs contain a large amount of unstructured binary data items and CLOBs contain a large amount of fixed-width character data items (characters that require a fixed number of bytes per character).

You can select a BLOB or CLOB locator from the database using a standard SELECT statement. When you select a BLOB or CLOB locator using SELECT, you acquire only the locator for the large object, not the data itself. Once you have the locator, however, you can read data from or write data to the large object using access functions.

Table 3–1 lists the methods included in the Oracle Database Lite BLOB class and their descriptions:

Table 3–1 Methods in the Oracle Database Lite BLOB Class

Function	Description
`length`	Returns the length of a BLOB in bytes.
`getBinaryOutputStream`	Returns BLOB data.
`getBinaryStream`	Returns a BLOB instance as a stream of bytes.
getBytes	Reads BLOB data, starting at a specified point, into a buffer.
`getConnection`	Returns the current connection.
`isConvertibleTo`	Determines if a BLOB can be converted to a particular class.
`putBytes`	Writes bytes to a specified point in the BLOB data.
`makeJdbcArray`	Returns the JDBC array representation of a BLOB.
`toJdbc`	Converts a BLOB to a JDBC class.
`trim`	Trims to length.

Table 3–2 lists the methods included in the Oracle Database Lite CLOB class and their descriptions.

Table 3–2 Methods in the Oracle Database Lite CLOB Class

Function	Description
`length`	Returns the length of a CLOB in bytes.
`getSubString`	Retrieves a substring from a specified point in the CLOB data.
`getCharacterStream`	Returns CLOB data as a stream of Unicode characters.
`getAsciiStream`	Returns a CLOB instance as an ASCII stream.
`getChars`	Retrieves characters from a specified point in the CLOB data into a character array.
`getCharacterOutputStream`	Writes CLOB data from a Unicode stream.
`getAsciiOutputStream`	Writes CLOB data from an ASCII stream.
`getConnection`	Returns the current connection.

Table 3–2 (Cont.) Methods in the Oracle Database Lite CLOB Class

Function	Description
`putChars`	Writes characters from a character array to a specified point in the CLOB data.
`putString`	Writes a string to a specified point in the CLOB data.
`toJdbc`	Converts a CLOB to a JDBC class.
`isConvertibleTo`	Determines if a CLOB can be converted to a particular class.
`makeJdbcArray`	Returns a JDBC array representation of a CLOB.
`trim`	Trims to length.

3.5.2 Data Access Extensions

Oracle Database Lite provides access functions to set and return values of the CLOB and BLOB datatypes. In addition, stream classes provide functions that enable stream-format access to large objects.

The large object access functions are located in the `OraclePreparedStatement`, the `OracleCallableStatement`, and the `OracleResultSet` class.

Table 3–3 lists the data access functions included in the `OracleResultSet` class.

Table 3–3 Data Access Functions in the OracleResultSet Class

Function	Description
`getBLOB`	Returns a locator to BLOB data.
`getCLOB`	Returns a locator to CLOB data.

The stream format access classes are `POLLobInputStream`, `POLLobOutputStream`, `POLClobReader`, and `POLClobWriter`.

The `POLLobInputStream` class includes the following data access function.

Function	Description
`read`	Reads from a large object into an array.

The `POLLobOutputStream` class includes this data access function.

Function	Description
`write`	Writes from an output stream into a large object.

The `POLClobReader` class extends the class `java.io.reader`. It includes these data access functions.

Function	Description
`read`	Reads characters from a CLOB into a portion of an array.
`ready`	Indicates whether a stream is ready to read.
`close`	Closes a stream.
`markSupported`	Indicates whether the stream supports the `mark` operation.

Function	Description
mark	Marks the current position in the stream. Subsequent calls to the reset function reposition the stream to the marked location.
reset	Resets the current position in the stream to the marked location. If the stream has not been marked, this function attempts to reset the stream in a way appropriate to the particular stream, such as by repositioning it at its starting point.
skip	Skips characters in the stream.

The POLClobWriter class extends the class java.io.writer. It includes these data access functions:

Function	Description
write	Writes an array of characters to the output stream.
flush	Writes any characters in a buffer to their intended destination.
close	Flushes and closes the stream.

3.5.2.1 Reading from a BLOB Sample Program

The following sample uses the getBinaryStream method to read BLOB data into a byte stream. It then reads the byte stream into a byte array, and returns the number of bytes read.

```
// Read BLOB data from BLOB locator.
InputStream byte_stream = my_blob.getBinaryStream();
byte [] byte_array = new byte [10];
int bytes_read = byte_stream.read(byte_array);
...
```

3.5.2.2 Writing to a CLOB Sample Program

The following sample reads data into a character array, then uses the getCharacterOutputStream method to write the array of characters to a CLOB.

```
java.io.Writer writer;
char[] data = {'0','1','2','3','4','5','6','7','8','9'};

// write the array of character data to a CLOB
writer = ((CLOB)my_clob).getCharacterOutputStream();
writer.write(data);
writer.flush();
writer.close();
...
```

3.6 Limitations

If data truncation occurs during a write, a SQL data truncation exception is thrown. A SQL data truncation warning results if data truncation occurs during a read.

The Oracle Database Lite JDBC classes and the JDBC 2.0 classes use the same name for certain datatypes (for example, oracle.sql.Blob and java.sql.Blob). If your program imports both oracle.sql.* and java.sql.*, attempts to access the overlapping classes without fully qualifying their names may result in compiler errors. To avoid this problem, use one of the following steps:

1. Use fully qualified names for BLOB, CLOB, and data classes.
2. Import the class explicitly (for example, import `oracle.sql.Blob`).

Class files always contain fully qualified class names, so the overlapping datatype names do not cause conflicts at runtime.

3.7 New JDBC 2.0 Features

This section describes JDBC 2.0 methods or interfaces that are supported by the Oracle Database Lite JDBC driver. Topics include:

- Section 3.7.1, "Interface Connection"
- Section 3.7.2, "Interface Statement"
- Section 3.7.3, "Interface ResultSet"
- Section 3.7.4, "Interface Database MetaData"
- Section 3.7.5, "Interface ResultMetaData"
- Section 3.7.6, "Interface PreparedStatement"

3.7.1 Interface Connection

This section describes the JDBC 2.0 Interface methods that are implemented by the Oracle Database Lite JDBC driver.

3.7.1.1 Methods

Statement

```
createStatement(int resultSetType, int resultSetConcurrency)
```

Creates a statement object that generates ResultSet objects with the given type and concurrency.

Map

```
getTypeMap()
```

Gets the TypeMap object associated with this connection.

CallableStatement

```
prepareCall(String sql, int resultSetType, int
resultSetConcurrency)
```

Creates a CallableStatement object that generates ResultSet objects with the given type and concurrency.

PreparedStatement

```
prepareStatement(String sql, int resultSetType, int
resultSetConcurrency)
```

Creates a PreparedStatement object that generates ResultSet objects with the given type and concurrency.

void

```
setTypeMap(Map map)
```

Installs the given type map as the type map for this connection.

3.7.2 Interface Statement

This section describes the JDBC 2.0 Interface Statement methods that are implemented by the Oracle Database Lite JDBC driver.

Connection

`getConnection()`

Returns the Connection object that produced this Statement object.

int

`getFetchDirection()`

Retrieves the direction for fetching rows from database tables that is the default for result sets generated from this Statement object. Only FETCH_FORWARD is supported for now.

int

`getFetchSize()`

Retrieves the number of result set rows that is the default fetch size for result sets generated from this Statement object. Only fetch size = 1 is supported for now.

int

`getResultSetConcurrency()`

Retrieves the result set concurrency. Only CONCUR_READ_ONLY is supported for now.

int

`getResultSetType()`

Determine the result set type. Only TYPE_FORWARD_ONLY and TYPE_SCROLL_INSENSITIVE are supported for now.

void

`setFetchDirection(int direction)`

Gives the driver a hint as to the direction in which the rows in a result set will be processed.

void

`setFetchSize(int rows)`

Gives the JDBC driver a hint as to the number of rows that should be fetched from the database when more rows are needed.

3.7.3 Interface ResultSet

This section describes the JDBC 2.0 Interface ResultSet methods that are implemented by the Oracle Database Lite JDBC driver.

3.7.3.1 Fields

The following fields can be used to implement the Interface ResultSet feature.

static int

`CONCUR_READ_ONLY`

The concurrency mode for a ResultSet object that may NOT be updated.

static int

`CONCUR_UPDATABLE`

The concurrency mode for a ResultSet object that may be updated. Not supported for now.

static int

`FETCH_FORWARD`

The rows in a result set will be processed in a forward direction; first-to-last.

static int

`FETCH_REVERSE`

The rows in a result set will be processed in a reverse direction; last-to-first. Not supported for now.

static int

`FETCH_UNKNOWN`

The order in which rows in a result set will be processed is unknown.

static int

`TYPE_FORWARD_ONLY`

The type for a ResultSet object whose cursor may move only forward.

static int

`TYPE_SCROLL_INSENSITIVE`

The type for a ResultSet object that is scrollable but generally not sensitive to changes made by others.

static int

`TYPE_SCROLL_SENSITIVE`

The type for a ResultSet object that is scrollable and generally sensitive to changes made by others. Not supported for now.

3.7.3.2 Methods

This section describes the JDBC 2.0 ResultSet method implemented by the Oracle Database Lite JDBC driver.

boolean

`absolute(int row)`

Moves the cursor to the given row number in the result set.

void

```
afterLast()
```

Moves the cursor to the end of the result set, just after the last row.

void

```
beforeFirst()
```

Moves the cursor to the front of the result set, just before the first row.

boolean

```
first()
```

Moves the cursor to the first row in the result set.

Array

```
getArray(String colName)
```

Gets an SQL ARRAY value in the current row of this ResultSet object.

BigDecimal

```
getBigDecimal(int columnIndex)
```

Gets the value of a column in the current row as a java.math.BigDecimal object with full precision.

BigDecimal

```
getBigDecimal(String columnName)
```

Gets the value of a column in the current row as a java.math.BigDecimal object with full precision.

int

```
getConcurrency()
```

Returns the concurrency mode of this result set.

Date

```
getDate(int columnIndex, Calendar cal)
```

Gets the value of a column in the current row as a java.sql.Date object.

int

```
getFetchDirection()
```

Returns the fetch direction for this result set.

int

```
getFetchSize()
```

Returns the fetch size for this result set.

int

```
getRow()
```

Retrieves the current row number.

Statement

`getStatement()`

Returns the Statement that produced this ResultSet object.

int

`getType()`

Returns the type of this result set.

boolean

`isAfterLast()`

boolean

`isBeforeFirst()`

boolean

`isFirst()`

boolean

`isLast()`

boolean

`last()`

Moves the cursor to the last row in the result set.

boolean

`previous()`

Moves the cursor to the previous row in the result set.

void

`refreshRow()`

Refreshes the current row with its most recent value in the database. Currently does nothing.

boolean

`relative(int rows)`

Moves the cursor a relative number of rows, either positive or negative.

3.7.3.3 Methods that Return False

The following three methods always return false because this release does not support deletes, inserts, or updates.

boolean

`rowDeleted()`

Indicates whether a row has been deleted.

boolean

`rowInserted()`

Indicates whether the current row has had an insertion.

boolean

`rowUpdated()`

Indicates whether the current row has been updated.

void

`setFetchDirection(int direction)`

Gives a hint as to the direction in which the rows in this result set will be processed.

void

`setFetchSize(int rows)`

Gives the JDBC driver a hint as to the number of rows that should be fetched from the database when more rows are needed for this result set.

3.7.4 Interface Database MetaData

This section describes the JDBC 2.0 Interface Database MetaData methods that are implemented by the Oracle Database Lite JDBC driver.

3.7.4.1 Methods

The following methods can be used to implement the Interface Database MetaData feature.

Connection

`getConnection()`

Retrieves the connection that produced this metadata object.

boolean

`supportsResultSetConcurrecny(int type, int concurrency)`

Supports the concurrency type in combination with the given result set type.

boolean

`supportsResultSetType(int Type)`

Supports the given result set type.

3.7.4.2 Methods that Return False

The following methods return false, because this release does not support deletes or updates.

boolean

`deletesAreDetected(int Type)`

Indicates whether or not a visible row delete can be detected by calling ResultSet.rowDeleted().

boolean

`insertsAreDetected(int Type)`

Indicates whether or not a visible row insert can be detected by calling ResultSet.rowInserted().

boolean

`othersDeletesAreVisible(int Type)`

Indicates whether deletes made by others are visible.

boolean

`othersInsertsAreVisible(int Type)`

Indicates whether inserts made by others are visible.

boolean

`othersUpdatesAreVisible(int Type)`

Indicates whether updates made by others are visible.

boolean

`ownDeletesAreVisible(int Type)`

Indicates whether a result set's own deletes are visible.

boolean

`ownInsertsAreVisible(int Type)`

Indicates whether a result set's own inserts are visible.

boolean

`ownUpdatesAreVisisble(int Type)`

Indicates whether a result set's own updates are visible.

boolean

`updatesAreDetected(int Type)`

Indicates whether or not a visible row update can be detected by calling the method ResultSet.rowUpdated.

3.7.5 Interface ResultMetaData

This section lists methods that can be implemented using the Interface ResultMetaData feature.

3.7.5.1 Methods

The following method can be used to implement the Interface ResultMetaData feature.

String

`getColumnClassName(int column)`

Returns the fully-qualified name of the Java class whose instances are manufactured if the method ResultSet.getObject is called to retrieve a value from the column.

3.7.6 Interface PreparedStatement

This section describes methods that can be implemented using the Interface PreparedStatement feature.

3.7.6.1 Methods

The following methods can be used to implement the Interface PreparedStatement feature.

Result

```
SetMetaDatagetMetaData()
```

Gets the number, types and properties of a ResultSet's columns.

void

```
setDate(int parameter Index, Date x, Calendar cal)
```

Sets the designated parameter to a java.sql.Date value, using the given Calendar object.

void

```
setTime(int parameterIndex, Time x, Calendar cal)
```

Sets the designated parameter to a java.sql.Time value, using the given Calendar object.

void

```
setTimestamp(int parameter Index, Timestamp x, Calendar cal)
```

Sets the designated parameter to a java.sql.Timestamp value, using the given Calendar object.

3.7.6.1.1 Limitation currently, the option `setQueryTimeOut` is not supported.

4

Java Stored Procedures and Triggers

This chapter describes how to use Java stored procedures and triggers within the Oracle Database Lite relational model. Topics include:

4.1 New Features in Oracle Database Lite

Oracle Database Lite supports the Oracle database server development model for stored procedures. In this model (referred to as the "load and publish" development model), instead of attaching classes to tables, you load the Java class into the Oracle Database Lite database using the `loadjava` command-line utility or the SQL statement `CREATE JAVA`. After loading the class into the database, you use a call specification to publish the methods in the class that you want to call from SQL. You use either the `CREATE FUNCTION` or `CREATE PROCEDURE` command to create a call specification. For more information, see "Model 1: Using the Load and Publish Stored Procedure Development Model".

Oracle Database Lite still supports the traditional model of creating stored procedures. In the traditional model, you attach the Java class to a table. The static methods in the class become the table-level stored procedures of the table, and the non-static (instance) methods become the row-level stored procedures.

Oracle Database Lite now includes the `loadjava` utility, which automates the task of loading Java classes into the database. Using `loadjava`, you can load Java class, source, and resource files, individually or in archives. For more information, see "loadjava".

4.2 Stored Procedures and Triggers Overview

A Java stored procedure is a Java method that is stored in Oracle Database Lite. The procedure can be invoked by applications that access the database. A trigger is a stored procedure that executes, or "fires", when a specific event occurs, such as a row update, insertion, or deletion. An update of a specific column can also fire a trigger.

A trigger can operate at the statement-level or row-level. A statement-level trigger fires once per triggering statement, no matter how many rows are affected. A row-level trigger fires once for every row affected by the triggering statement. Java

stored procedures can return a single value, a row, or multiple rows. Triggers, however, cannot return a value.

The first step to creating a stored procedure is to create the class that you want to store in Oracle Database Lite. You can use any Java IDE to write the procedure, or you can simply reuse an existing procedure that meets your needs.

When creating the class, consider the following restrictions on calling Java stored procedures from SQL DML statements:

- When called from an INSERT, UPDATE, or DELETE statement, the method cannot query or modify any database tables modified by that statement.
- When called from a SELECT, INSERT, UPDATE, or DELETE statement, the method cannot execute SQL transaction control statements, such as COMMIT or ROLLBACK.

Any SQL statement in a stored procedure that violates a restriction produces an error at run time (when the statement is parsed).

You must provide your class with a unique name for its deployment environment, since only one Java Virtual Machine is loaded for each Oracle Database Lite application. If the application executes methods from multiple databases, then the Java classes from these databases are loaded into the same Java Virtual Machine. By prefixing the Java class name with the database name, you ensure that the Java class names are unique across multiple databases.

If a Java stored procedure takes an argument of type `java.sql.Connection`, then Oracle Database Lite supplies the appropriate argument value from the current transaction or row as the first argument to the method. The application executing the method does not need to provide a value for this parameter. In this case, DMLs executed inside the procedure are executed in the invoker's transaction context.

4.3 Using Stored Procedures

Oracle Database Lite supports several development models for creating stored procedures. In the load and publish model, you load the Java class into Oracle Database Lite, then create a call specification (call spec) for the static methods in the class that you want to call from SQL. This model is also supported by Oracle database, which enables you to utilize skills and resources you have already developed in implementing Oracle database enterprise applications and data.

This model consists of the following steps:

1. Develop a Java class that contains the methods you want to store.
2. Use the `loadjava` utility or the SQL CREATE JAVA command to load the class into the Oracle Database Lite.
3. Publish the methods that you want to make accessible to SQL by creating call specs for those methods. By publishing a method, you associate a SQL name to the method. SQL applications use this name to invoke the method.

You do not need to publish every procedure that you store in Oracle Database Lite, only those that should be callable from SQL. Many stored procedures are only called by other stored procedures, and do not need to be published. For more information on using this model for developing stored procedures, see "Model 1: Using the Load and Publish Stored Procedure Development Model". The load and publish model only supports static methods.

In the second model, you attach the class to a table and invoke methods in the class by name. This is the traditional Oracle Database Lite model for developing stored procedures. Using this model, you can store both class-level (static) methods and object-level (non-static) methods.

For this model, follow these steps:

1. Develop a Java class with the methods you want to store.
2. Attach the class to a table using the SQL ALTER TABLE command.

After attaching the class, you can invoke methods in the class directly from SQL. You identify the method with the following syntax:

```
table_name.method_name
```

For more information on attaching Java classes to tables, see "Model 2: Using the Attached Stored Procedure Development Model".

Oracle Database Lite provides tools and SQL commands for dropping stored procedures. You should use caution when dropping procedures from the database, since Oracle Database Lite does not keep track of dependencies between classes. You must ensure that the stored procedure you drop is not referenced by other stored procedures. Dropping a class invalidates classes that depend on it directly or indirectly.

4.3.1 Model 1: Using the Load and Publish Stored Procedure Development Model

This section describes how to create stored procedures using the load and publish development model. The first step in creating a stored procedure is to write the class. Make sure that the class compiles and executes without errors. Next, load the class into Oracle Database Lite. Finally, publish the methods that you want to call from SQL. In Oracle Database Lite, you cannot publish a method that is mapped to a `main` method. Oracle database, on the other hand, permits call specs that publish `main` methods.

Note: The load and publish development model only supports Java static methods. To store static and non-static (instance) methods, you must attach the class to database tables, as described in "Model 2: Using the Attached Stored Procedure Development Model".

4.3.1.1 Loading Classes

To load Java classes into the Oracle Database Lite database, you can use either:

- `loadjava`
- the SQL statement `CREATE JAVA`

The loadjava command-line utility automates the task of loading Java classes into Oracle Database Lite and Oracle databases. To load Java classes manually, use the SQL statement `CREATE JAVA`.

4.3.1.1.1 loadjava loadjava creates schema objects from files and loads them into the database. Schema objects can be created from Java source files, class files, and resource files. Resource files may be image files, resources, or anything else a procedure may need to access as data. You can pass files to `loadjava` individually, or as ZIP or JAR archive files.

Oracle Database Lite does not keep track of class dependencies. Make sure that you load into the database, or place in the CLASSPATH, all supporting classes and resource files required by a stored procedure. To query the classes that are loaded in the database, you can query the `okJavaObj` meta class.

> **Note:** The table name and column names are case sensitive.

Syntax

`loadjava` uses the following syntax:

```
loadjava {-user | -u} username/password[@database]
   [-option_name -option_name ...] filename filename ...
```

Arguments

This section discusses the `loadjava` arguments in detail.

User

The `user` argument specifies a username, password, and database directory in the following format:

```
<user>/<password>[@<database>]
```

For example:

```
scott/tiger@<Oracle_home>\Mobile\Sdk\OLDB40\Polite.odb
```

Options

Oracle Database Lite supports the following options that are listed and described in Table 4–1.

Table 4–1 Options

Option	Description
`-force | -f`	Forces files to be loaded, even if a schema object with the same name already exists in the database.
`-verbose | -v`	Directs `loadjava` to display detailed status messages while running.
`-meta | -m`	Creates the meta information in the database but does not load the classes. This is useful when the classes are in a .jar file and are not loaded into the database.

When specifying multiple options, you must separate the options with spaces. For example:

```
-force -verbose
```

Oracle database supports additional options, as described in the *Oracle9i Java Stored Procedures Developer's Guide*. If used with Oracle Database Lite, the additional options are recognized but not supported. Using them does not result in an error.

To view the options supported by Oracle database, see the `loadjava` help information using the following syntax.

```
loadjava {-help | -h}
```

Filenames

On the command line, you can specify as many class, source, JAR, ZIP, and resource files as you like, in any order. You must separate multiple file names with spaces, not commas. If passed a source file, `loadjava` invokes the Java compiler to compile the file before loading it into the database. If passed a JAR or ZIP file, `loadjava` processes each file in the JAR or ZIP. It does not create a schema object for the JAR or ZIP archive. `loadjava` does not process a JAR or ZIP archive within another JAR or ZIP archive.

The best way to load files is to place them in a JAR or ZIP and then load the archive. Loading archives avoids the complications associated with resource schema object names. If you have a JAR or ZIP that works with the JDK, then you can be sure that loading it with `loadjava` also works, and you can avoid the complications associated with resource schema object naming.

As it loads files into the database, `loadjava` must create a name for the schema objects it creates for the files. The names of schema objects differ slightly from filenames, and different schema objects have different naming conventions. Class files are self-identifying, so `loadjava` can map their filenames to the names of schema objects automatically. Likewise, JAR and ZIP archives include the names of the files they contain.

However, resource files are not self-identifying; `loadjava` derives the names of Java resource schema objects from the literal names you enter on the command-line (or the literal names in a JAR or ZIP archive). Because classes use resource schema objects while executing, it is important that you specify the correct resource file names on the command line.

The best way to load individual resource files is to run `loadjava` from the top of the package tree, specifying resource file names relative to that directory. If you decide not to load resource files from the top of the package tree, consider the following information concerning resource file naming.

When you load a resource file, `loadjava` derives the name of the resource schema object from the file name that you enter on the command line. Suppose you type the following relative and absolute pathnames on the command line:

```
cd \scott\javastuff
loadjava options alpha\beta\x.properties
loadjava options  \scott\javastuff\alpha\beta\x.properties
```

Although you have specified the same file with a relative and an absolute pathname, `loadjava` creates *two* schema objects:

- `alpha\beta\x.properties`
- `\scott\javastuff\alpha\beta\x.properties`.

`loadjava` generates the resource schema object's name from the file names *you entered*.

Classes can refer to resource files relatively (for example, `b.properties`) or absolutely (for example, `\a\b.properties`). To ensure that `loadjava` and the class loader use the same name for a schema object, pass `loadjava` *the name of the resource that the class passes to the* `java.lang.Object.getResource` *or* `java.lang.Class.getResourceAsStream` *method.*

Instead of remembering whether classes use relative or absolute resource names and changing directories so that you can enter the correct name on the command line, you can load resource files into a JAR file, as follows:

```
cd \scott\javastuff
```

```
jar -cf alpharesources.jar alpha\*.properties
loadjava options alpharesources.jar
```

Or, to simplify further, put both the class and resource files in a JAR, which makes the following invocations equivalent:

```
loadjava options alpha.jar
loadjava options \scott\javastuff\alpha.jar
```

Example

The following loads a class and resource file into Oracle Database Lite. It uses the `force` option; if the database already contains objects with the specified names, `loadjava` replaces them.

```
c:\> loadjava -u scott/tiger@c:\Olite\Mobile\Sdk\OLDB40\Polite.odb -f Agent.class\
images.dat
```

4.3.1.1.2 Using CREATE JAVA

To load Java classes manually, use the following syntax:

```
CREATE [OR REPLACE] [AND RESOLVE] [NOFORCE]
   JAVA {CLASS [SCHEMA <schema_name>] |
   RESOURCE NAMED [<schema_name>.]<primary_name>}
   [<invoker_rights_clause>]
   RESOLVER <resolver_spec>]
   USING BFILE ('<dir_path>', '<class_name>')
```

The following apply to the CREATE JAVA parameters:

- The OR REPLACE clause, if specified, recreates the function or procedure if one with the same name already exists in the database.
- For compatibility with the Oracle database, Oracle Database Lite recognizes but ignores the `<resolver_spec>` clause. Unlike the Oracle database, Oracle Database Lite does not resolve class dependencies. When loading classes manually, be sure to load all dependent classes.
- Oracle Database Lite recognizes but ignores `<invoker_rights_clause>`.

Example

The following demonstrates a CREATE JAVA statement. It loads a class named `Employee` into the database.

```
CREATE JAVA CLASS USING BFILE ('c:\myprojects\java',
   'Employee.class');
```

4.3.1.2 Publishing Stored Procedures to SQL

After loading the Java class into the Oracle Database Lite database using `loadjava` or CREATE JAVA, you publish any static method in the class that you want to call from SQL. To publish the method, create a call specification (call spec) for it. The call spec maps the Java method's name, parameter types, and return types to SQL counterparts.

You do not need to publish every stored procedure, only those that serve as entry points for your application. In a typical implementation, many stored procedures are called only by other stored procedures, not by SQL users.

To create a call spec, use the SQL commands CREATE FUNCTION or CREATE PROCEDURE. Use CREATE FUNCTION for methods that return a value, and

CREATE PROCEDURE for methods that do not return a value. The CREATE FUNCTION and CREATE PROCEDURE statements have the following syntax.

```
CREATE [OR REPLACE]
   { PROCEDURE [<schema_name>.]<proc_name> [([<sql_parms>])] |
   FUNCTION [<schema_name>.]<func_name> [([<sql_parms>])]
   RETURN <sql_type> }
   <invoker_rights_clause>
   { IS | AS } LANGUAGE JAVA NAME
   '<java_fullname>  ([<java_parms>])
   [return <java_type_fullname>]';
   /
```

The following apply to this statement's keywords and parameters:

- <sql_parms> has the following format:

  ```
  <arg_name> [IN | OUT | IN OUT]
           <datatype>
  ```

- <java_parms> is the fully qualified name of the Java datatype.
- For compatibility with the Oracle database, Oracle Database Lite recognizes but ignores the `<invoker_rights_clause>` clause.
- <java_fullname> is the fully qualified name of a static Java method.
- `IS` and `AS` are synonymous.

For example, assume the following class has been loaded into the database:

```
import java.sql.*;
import java.io.*;

public class GenericDrop {
   public static void dropIt (Connection conn, String object_type,
                       String object_name) throws SQLException {
      // Build SQL statement
      String sql = "DROP " + object_type + " " + object_name;
      try {
         Statement stmt = conn.createStatement();
         stmt.executeUpdate(sql);
         stmt.close();
      } catch (SQLException e) {
         System.err.println(e.getMessage());}
   }  // dropIt
}  // GenericDrop
```

Class `GenericDrop` has one method named `dropIt`, which drops any kind of schema object. For example, if you pass the arguments "table" and "emp" to `dropIt`, the method drops the database table EMP from your schema.

The following call specification publishes the method to SQL:

```
CREATE OR REPLACE PROCEDURE drop_it (
      obj_type VARCHAR2,
      obj_name VARCHAR2)
   AS LANGUAGE JAVA
   NAME 'GenericDrop.dropIt(java.sql.Connection,
      java.lang.String, java.lang.String)';
   /
```

Notice that you must fully qualify the Java datatype parameters.

Given that you have a table named TEMP defined in your schema, you can execute the drop_it procedure from SQL Plus as follows.

```
Select drop_it('TABLE', 'TEMP') from dual;
```

You can also execute the drop_it procedure from within a ODBC application using ODBC CALL statement. For more information, refer Section 4.3.3, "Calling Java Stored Procedures from ODBC".

4.3.1.3 Calling Published Stored Procedures

After publishing the stored procedure to SQL, you call it by using a SQL DML statement. For example, assume that this class is stored in the database:

```
public class Formatter {
   public static String formatEmp (String empName, String jobTitle) {
      empName = empName.substring(0,1).toUpperCase() +
         empName.substring(1).toLowerCase();
      jobTitle = jobTitle.trim().toLowerCase();
      if (jobTitle.equals("analyst"))
         return (new String(empName + " is an exempt analyst"));
      else
      return (new String(empName + " is a non-exempt " + jobTitle));
  }
}
```

Class `Formatter` has one method named `formatEmp`, which returns a formatted string containing an employee's name and job status. Create a call spec for `Formatter` as follows:

```
CREATE OR REPLACE FUNCTION format_emp (ename VARCHAR2, job VARCHAR2)
   RETURN VARCHAR2
   AS LANGUAGE JAVA
   NAME 'Formatter.formatEmp (java.lang.String, java.lang.String)
   return java.lang.String';
   /
```

The call spec publishes the method `formatEmp` as `format_emp`. Invoke it as follows:

```
SELECT FORMAT_EMP(ENAME, JOB) AS "Employees" FROM EMP
   WHERE JOB NOT IN ('MANAGER', 'PRESIDENT') ORDER BY ENAME;
```

This statement produces the following output:

```
Employees
-----------------------------------------
Adams is a non-exempt clerk
Allen is a non-exempt salesman
Ford is an exempt analyst
James is a non-exempt clerk
Martin is a non-exempt salesman
Miller is a non-exempt clerk
Scott is an exempt analyst
Smith is a non-exempt clerk
Turner is a non-exempt salesman
Ward is a non-exempt salesman
```

Note: Oracle Database Lite does not support the Oracle database SQL CALL statement for invoking stored procedures.

For information on calling stored procedures from C and C++ applications, see "Calling Java Stored Procedures from ODBC".

4.3.1.4 Dropping Published Stored Procedures

To remove classes from Oracle Database Lite, use either of the following:

- the `dropjava` utility
- the SQL DROP JAVA statement

To drop call specifications, use either DROP FUNCTION or DROP PROCEDURE.

4.3.1.4.1 Using dropjava `dropjava` is a command-line utility that automates the task of dropping Java classes from Oracle Database Lite and Oracle databases. `dropjava` converts file names into the names of schema objects and drops the schema objects. Use the following syntax to invoke `dropjava`:

```
dropjava {-user | -u} username/password[@database]
  [-option] filename filename ...
```

Arguments

This section describes the arguments to `dropjava`.

User

The `user` argument specifies a username, password, and absolute path to the database file in the following format:

```
<user>/<password>[@<database>]
```

For example:

```
scott/tiger@c:\Olite\Mobile\Sdk\OLDB40\Polite.odb
```

Option

By specifying the verbose option (`-verbose | -v`), you can direct `dropjava` to produce detailed status messages while running.

Oracle database supports additional options for `dropjava`, as described in the *Oracle9i Java Stored Procedures Developer's Guide*. If used with Oracle Database Lite, the additional options are recognized but not supported. Using them does not result in an error.

For a complete list of supported and recognized options, from the command prompt type:

```
dropjava -help
```

Filename

For the `filename` argument, you can specify any number of Java class, source, JAR, ZIP, and resource files, in any order. JAR and ZIP files must be uncompressed. `dropjava` interprets most file names the same way `loadjava` does:

- For class files, `dropjava` finds the class name in the file and drops the corresponding schema object.

- For source files, `dropjava` finds the first class name in the file and drops the corresponding schema object.
- For JAR and ZIP files, `dropjava` processes the archived file names as if they had been entered on the command line.

If a file name has an extension other than **.java**, **.class**, **.jar**, or **.zip**, or has no extension, then `dropjava` assumes that the file name is the name of a schema object, then drops all source, class, and resource schema objects with that name. If `dropjava` encounters a file name that does not match the name of any schema object, it displays an error message and then processes the remaining file names.

4.3.1.4.2 Using SQL Commands To drop a Java class from Oracle Database Lite manually, use the DROP JAVA statement. DROP JAVA has the following syntax:

```
DROP JAVA { CLASS | RESOURCE } [<schema-name> .]<object_name>
```

To drop a call specification, use the DROP FUNCTION or DROP PROCEDURE statement:

```
DROP { FUNCTION | PROCEDURE } [<schema-name>.]<object_name>
```

The schema name, if specified, is recognized but not supported by Oracle Database Lite.

4.3.1.5 Example

The following example creates a Java stored procedure using the load and publish model.

In this example, you store the Java method `paySalary` in the Oracle Database Lite. `paySalary` computes the take-home salary for an employee.

This example covers the following steps.

- Step 1: Create the Java Class
- Step 2: Load the Java Class into the Database
- Step 3: Publish the Function
- Step 4: Execute the Function

More examples of Java stored procedures are located in the `<Oracle_home>\Mobile\SDK\samples\jdbc` directory.

Step 1: Create the Java Class

Create the Java class `Employee` in the file **Employee.java**. The `Employee` class implements the `paySalary` method:

```
import java.sql.*;
public class Employee {
   public static String paySalary(float sal, float fica, float sttax,
                           float ss_pct, float espp_pct)  {
      float deduct_pct;
      float net_sal;
        // compute take-home salary
      deduct_pct = fica + sttax + ss_pct + espp_pct;
      net_sal = sal * deduct_pct;
      String returnstmt = "Net salary is " + net_sal;
      return returnstmt;
   } // paySalary
}
```

Note: The keyword "public class" should not be used in a comment before the first public class statement.

Step 2: Load the Java Class into the Database

From MSQL, load the Java class using CREATE JAVA, as follows:

```
CREATE JAVA CLASS USING BFILE ('c:\myprojects\doc',
'Employee.class');
```

This command loads the Java class located in c:\myprojects\doc into the Oracle Database Lite.

Step 3: Publish the Function

Create a call spec for the `paySalary` method. The following call spec publishes the Java method `paySalary` as function `pay_salary`:

```
CREATE FUNCTION pay_salary (
sal float, fica float, sttax float, ss_pct float, espp_pct float)
RETURN VARCHAR2
AS LANGUAGE JAVA NAME
'Employee.paySalary(float, float, float, float, float)
return java.lang.String';
/
```

Step 4: Execute the Function

To execute `pay_salary` in MSQL:

```
SELECT pay_salary(6000.00, 0.2, 0.0565, 0.0606, 0.1)
FROM DUAL;
```

To execute `pay_salary` in ODBC:

```
SQLExecDirect(hstm,
   "SELECT pay_salary(6000.00,0.2,0.0565,0.0606,0.1)
   FROM DUAL);
```

Because the arguments to `pay_salary` are constants, the FROM clause specifies the dummy table DUAL. This SELECT statement produces the following output:

```
Net salary is 2502.6
```

4.3.2 Model 2: Using the Attached Stored Procedure Development Model

This section describes how to create stored procedures by attaching classes to tables. This information is specific to Oracle Database Lite; you cannot attach classes to Oracle database tables as described here. The load and publish model for developing stored procedures, described in "Model 1: Using the Load and Publish Stored Procedure Development Model", only supports class (static) methods. By attaching classes to tables, however, you can store and call Java class and instance methods.

To create attached stored procedures, develop the class that you want to attach. Make sure that the class compiles and executes without errors. Then attach the class to an Oracle Database Lite table. Once the class is attached, the methods in the class become the table-level and row-level stored procedures of the table.

4.3.2.1 Attaching a Java Class to a Table

To attach a Java class to a table, use the SQL command ALTER TABLE. The ALTER TABLE command has the following syntax:

```
ALTER TABLE [schema.]table
   ATTACH JAVA {CLASS|SOURCE} "cls_or_src_name "
   IN {DATABASE|'cls_or_src_path '}
   [WITH CONSTRUCTOR ARGS (col_name_list )]
```

You can attach either a source file or a class file. Source files are compiled by the Java compiler found in the system path.

`cls_or_src_name` specifies a fully qualified name of a class or source file. This includes the package name followed by class name, such as `Oracle.lite.Customer`. Do not include the file extension in the class or source file name. The name is case-sensitive. If you use lowercase letters, enclose the name in double quotes (" "). Make sure that the source or class is in the package specified by `cls_or_src_name`. (The source file of the example class `Customer` should contain the line "`package Oracle.lite;`".) The class file is stored in the database in the same package. Oracle Database Lite creates the package if it does not already exist.

If you have already attached the Java class to another table in the database, you can use the IN DATABASE clause. If the class has not yet been attached, specify the directory location of the class or source file in `cls_or_src_path`.

Prior to executing a row-level stored procedure, Oracle Database Lite creates a Java object for the row, if one does not already exist. If the ALTER TABLE statement includes a WITH CONSTRUCTOR clause, Oracle Database Lite creates the object using the class constructor that is the best match given the datatypes of the columns included in `col_name_list`. If the ALTER TABLE statement does not include a WITH CONSTRUCTOR clause, Oracle Database Lite uses the default constructor.

You can use the ODBC functions `SQLProcedures` and `SQLProcedureColumns` to retrieve information about methods defined in a table.

4.3.2.2 Table-Level Stored Procedures

Table-level stored procedures are the static methods of the attached Java class. Therefore, when executing the method, Oracle Database Lite does not instantiate the class to which it belongs. In a call statement, you refer to table-level stored procedures as *table_name.method_name*.

Statement-level triggers and BEFORE INSERT and AFTER DELETE row-level triggers (see section ""Statement-Level vs. Row-Level Triggers"") must be table-level stored procedures.

4.3.2.3 Row-Level Stored Procedures

Row-level stored procedures are the non-static methods in the attached Java class. To execute a row-level stored procedure, Oracle Database Lite instantiates the class to which the procedure belongs. The arguments to the class constructor determine which column values the constructor uses as parameters to create the class instances. In a call statement, you refer to row-level stored procedures as method_name (without the table qualifier). Row-level triggers can indirectly execute row-level stored procedures.

4.3.2.4 Calling Attached Stored Procedures

After attaching the class to a table using the ALTER TABLE statement, you can call it with a SELECT statement. Refer to table-level stored procedures as *table_name.method_name* and row-level procedures as *method_name*.

For example, to execute a table-level stored procedure:

```
SELECT table_name.proc_name[arg_list]
   FROM {DUAL|[schema.]table WHERE condition};
```

The `proc_name` is the name of the table-level stored procedure. Each argument in `arg_list` is either a constant or a reference to a column in the table. If all the arguments of `arg_list` are constants, the FROM clause should reference the dummy table DUAL.

Execute a row-level stored procedure as follows:

```
SELECT [schema.]proc_name[arg_list]
   FROM [schema.]table
   WHERE condition;
```

If you call a procedure in the form *table_name.method_name*, and a table or method with that name does not exist, Oracle Database Lite assumes that table_name refers to a schema name and *method_name* refers to a procedure name. If you reference method_name only, Oracle Database Lite assumes that the referenced method is a row-level procedure. If there is no such procedure defined, however, Oracle Database Lite assumes that method_name refers to a procedure in the current schema.

Note: Oracle Database Lite does not support the Oracle8*i* SQL CALL statement for invoking stored procedures.

You can use a callable statement to execute a procedure from ODBC or JDBC applications. For more information, see Chapter 3, "JDBC Programming". For additional information, see "Calling Java Stored Procedures from ODBC".

4.3.2.5 Dropping Attached Stored Procedures

You use the ALTER TABLE command to drop stored procedures. ALTER TABLE has the following syntax:

```
ALTER TABLE [schema.]table
   DETACH [AND DELETE] JAVA CLASS "class_name"
```

Note: You must enclose the class name in double quotes (" ") if it contains lowercase letters.

Detaching the Java class does not delete it from the database. To delete the Java class file from the database, use the DETACH AND DELETE statement.

If you delete a Java class from the database after invoking it as a stored procedure or trigger, the class remains in the Java Virtual Machine attached to the application. To unload the class from the Java Virtual Machine, commit changes to the database, if necessary, and close all applications connected to the database. To replace a Java class, you must close all connections to the database and reload the class.

4.3.2.6 Example

The following example shows how to create a Java stored procedure in Oracle Database Lite. In this example, you attach the Java method `paySalary` to the table EMP. `paySalary` computes the take-home salary for an employee.

This example covers the following steps:

Step 1: Create the Table

Create the table using the following SQL command:

```
CREATE TABLE EMP(Col1 char(10));
```

Step 2: Create the Java Class

Create the Java class `Employee` in the file **Employee.java**. The `Employee` class implements the `paySalary` method:

```
import java.sql.*;
public class Employee {
   public static String paySalary(float sal, float fica, float sttax,
                           float ss_pct, float espp_pct)  {
      float deduct_pct;
      float net_sal;
        // compute take-home salary
      deduct_pct = fica + sttax + ss_pct + espp_pct;
      net_sal = sal * deduct_pct;
      String returnstmt = "Net salary is " + net_sal;
      return returnstmt;
   } // paySalary
}
```

Step 3: Attach the Java Class to the Table

From MSQL, attach the Java class using the ALTER TABLE command:

```
ALTER TABLE EMP ATTACH JAVA SOURCE "Employee" IN 'C:\tmp';
```

This command attaches the Java source file for the `Employee` class, which resides in the directory `C:\tmp`, to the EMP table.

Step 4: Execute the Method

To execute the `paySalary` method in MSQL, type the following statement:

```
SELECT EMP."paySalary"(6000.00,0.2,0.0565,0.0606,0.1)
   FROM DUAL;
```

To execute `paySalary` from ODBC, invoke `SQLExecDirect`:

```
SQLExecDirect(hstm,
   "SELECT EMP.\"paySalary\"(6000.00,0.2,0.0565,0.0606,0.1)
   FROM DUAL);
```

This statement produces the following result:

```
Net salary is 2502.6
```

4.3.3 Calling Java Stored Procedures from ODBC

When invoking a Java stored procedure from a multithreaded C or C++ application, you should load **jvm.dll** from the application's `main` function. This resolves a problem

that occurs with the Java Virtual Machine's garbage collection when a C or C++ application creates multiple threads that invoke a stored procedure directly or indirectly. The Java Virtual Machine runs out of memory because the threads do not detach from the Java Virtual Machine before exiting. Since Oracle Database Lite cannot determine whether the Java Virtual Machine or the user application created the thread, it does not attempt to detach them.

`main` should load the library before taking any other action, as follows:

```
int main (int argc, char** argv)
{
   LoadLibrary("jvm.dll");
   ...
}
```

The library loads the Java Virtual Machine into the application's main thread. It attempts to detach any thread from the Java Virtual Machine if the thread detaches from the process. The **jvm.dll** behaves correctly even if the thread is not attached to a Java Virtual Machine.

4.4 Java Datatypes

Oracle Database Lite performs type conversion between Java and SQL datatypes according to standard SQL rules. For example, if you pass an integer to a stored procedure that takes a string, Oracle Database Lite converts the integer to a string. For information about row-level triggers arguments, see Trigger Arguments. For a complete list of Java to SQL datatype mappings, see Chapter 2, "Application Development", Section 2.1.1, "Java Datatypes", Table 2–1, " Datatype Conversions".

> **Note:** In Oracle database, DATE columns are created as TIMESTAMP. You must specify trigger methods accordingly.

Java does not allow a method to change the value of its arguments outside the scope of the method. However, Oracle Database Lite supports IN, OUT, and IN/OUT parameters.

Many Java datatypes are immutable or do not support NULL values. To pass NULL values and use IN/OUT parameters for those datatypes, a stored procedure can use an array of that type or use the equivalent object type. Table 4–2 shows the Java integer datatypes you can use to enable an integer to be an IN/OUT parameter or carry a NULL value.

Table 4–2 The Java Integer Datatypes

Java Argument	Can Be IN/OUT	Can Be NULL
`int`	No	No
`int[]`	Yes	Yes
`Integer`	No	Yes
`Integer[]`	Yes	Yes
`int[][]`	Yes	Yes

You can use mutable Java datatypes, such as `Date`, to pass a NULL or an IN/OUT parameter. However, use a `Date` array if a stored procedure needs to change the NULL status of its argument.

Note: Passing a NULL when the corresponding Java argument cannot be NULL causes an error.

4.4.1 Declaring Parameters

The return value of a Java method is the OUT parameter of the procedure. A primitive type or immutable reference type can be an IN parameter. A mutable reference type or array type can be an IN/OUT parameter. Table 4–3 shows the Java type to use to make the corresponding Oracle Database Lite parameter an IN/OUT parameter.

Table 4–3 Java Types for Oracle Database Lite IN/OUT Parameters

For IN/OUT parameters of type...	Use...
Number	`Integer[]` or `int[]`
Binary	`byte[]` or `byte[][]`
String	`string[]`

If the stored procedure takes a `java.sql.Connection`, Oracle Database Lite automatically supplies the argument using the value of the current transaction or row. This argument is the first argument passed to the procedure.

4.4.2 Using Stored Procedures to Return Multiple Rows

You can use stored procedures to return multiple rows. You can invoke stored procedures that return multiple rows only from JDBC or ODBC applications, however. For a stored procedure to return multiple rows, its corresponding Java method must return a `java.sql.ResultSet` object. By executing a SELECT statement, the Java method obtains a `ResultSet` object to return. The column names of the `ResultSet` are specified in the SELECT statement. If you need to address the result columns by different names than those used in the table, the SELECT statement should use aliases for the result columns. For example:

```
SELECT emp.name Name,  dept.Name Dept
   FROM  emp, dept
   WHERE emp.dept# = dept.dept#;
```

Because the return type of a stored procedure that returns multiple rows must be `java.sql.ResultSet`, the signature of that stored procedure cannot be used to obtain the column names or types of the result. Consequently, you should design additional tables to track the column names or result types for the stored procedures. For example, if you embed the preceding SELECT statement in a Java method, the method return type should be `java.sql.ResultType`, not `char Name` and `char Dept`.

Note: You can only create Java stored procedures that return multiple rows using the attached stored procedure development model, described in "Model 2: Using the Attached Stored Procedure Development Model".

4.4.2.1 Returning Multiple Rows in ODBC

To execute a stored procedure that returns multiple rows in an OBDC application, use the following CALL statement, in which P is the name of the stored procedure and a_1 through a_n are arguments to the stored procedure.

```
{CALL P(a1,...,an)}
```

You use a marker (?) for any argument that should be bound to a value before the statement executes. When the statement executes, the procedure runs and the cursor on the result set is stored in the statement handle. Subsequent fetches using this statement handle return the rows from the procedure.

After you execute the `CALL` statement, use `SQLNumResultCols` to find the number of columns in each row of the result. Use the `SQLDescribeCol` function to return the column name and datatype.

4.4.2.2 Example

The following example shows how to use ODBC to execute a stored procedure that returns multiple rows. This example does not use the `SQLNumResultCols` or `SQLDescribeCol` functions. It assumes that you have created a stored procedure, which you have published to SQL as PROC. PROC takes an integer as an argument.

```
rc = SQLPrepare(StmtHdl, "{call PROC(?)}", SQL_NTS);
CHECK_STMT_ERR(StmtHdl, rc, "SQLPrepare");

rc = SQLBindParameter(StmtHdl, 1, SQL_PARAM_INPUT_OUTPUT,
   SQL_C_LONG,SQL_INTEGER, 0, 0, &InOutNum, 0, NULL);
CHECK_STMT_ERR(StmtHdl, rc, "SQLBindParameter");

rc = SQLExecute(StmtHdl);
CHECK_STMT_ERR(StmtHdl, rc, "SQLExecute");

/* you can use SQLNumResultCols and SQLDescribeCol here */

rc = SQLBindCol(StmtHdl, 1, SQL_C_CHAR, c1, 20, &pcbValue1);
CHECK_STMT_ERR(StmtHdl, rc, "SQLBindCol");

rc = SQLBindCol(StmtHdl, 2, SQL_C_CHAR, c2, 20, &pcbValue2);
CHECK_STMT_ERR(StmtHdl, rc, "SQLBindCol");

while ((rc = SQLFetch(StmtHdl)) != SQL_NO_DATA_FOUND) {
   CHECK_STMT_ERR(StmtHdl, rc, "SQLFetch");
   printf("%s, %s\n", c1, c2);
}
```

4.5 Using Triggers

Triggers are stored procedures that execute, or "fire", when a specific event occurs. A trigger can fire when a column is updated, or when a row is added or deleted. The trigger can fire before or after the event.

Triggers are commonly used to enforce a database's business rules. For example, a trigger can verify input values and reject an illegal insert. Similarly, a trigger can ensure that all tables depending on a particular row are brought to a consistent state before the row is deleted.

4.5.1 Statement-Level vs. Row-Level Triggers

There are two types of triggers: row-level and statement-level. A row-level trigger is fired once for each row affected by the change to the database. A statement-level trigger fires only once, even if multiple rows are affected by the change.

The BEFORE INSERT and AFTER DELETE triggers can only fire table-level stored procedures, since a row object cannot be instantiated to call the procedures. The AFTER INSERT, BEFORE DELETE, and UPDATE triggers may fire table-level or row-level stored procedures.

4.5.2 Creating Triggers

Use the CREATE TRIGGER statement to create a trigger. The CREATE TRIGGER statement has the following syntax:

```
CREATE [OR REPLACE] TRIGGER trigger_name {BEFORE | AFTER} [{INSERT | DELETE |
   UPDATE [OF column_list]} [OR ]] ON table_reference
   [FOR EACH ROW] procedure_ref
   (arg_list)
```

In the CREATE TRIGGER syntax:

- Use the OR clause to specify multiple triggering events.
- Use FOR EACH ROW to create a row-level trigger. For a table-level trigger, do not include this clause.
- Use `procedure_ref` to identify the stored procedure to execute.

You can create multiple triggers of the same kind for a table if each trigger has a unique name within a schema.

In the following example, assume that you have stored and published a procedure as PROCESS_NEW_HIRE. The trigger AIEMP fires every time a row is inserted into the EMP table.

```
CREATE TRIGGER AIEMP AFTER INSERT ON EMP FOR EACH ROW
   PROCESS_NEW_HIRE(ENO);
```

UPDATE triggers that use the same stored procedure for different columns of a table are fired only once when a subset of the columns is modified within a statement. For example, the following statement creates a BEFORE UPDATE trigger on table T, which has columns C1, C2, and C3:

```
CREATE TRIGGER T_TRIGGER BEFORE UPDATE OF C1,C2,C3 ON T
   FOR EACH ROW trigg(old.C1,new.C1,old.C2,new.C2,
   old.C3,new.C3);
```

This update statement fires `T_TRIGGER` only once:

```
UPDATE T SET C1 = 10, C2 = 10 WHERE ...
```

4.5.2.1 Enabling and Disabling Triggers

When you create a trigger, it is automatically enabled. To disable triggers, use the ALTER TABLE or ALTER TRIGGER statement.

To enable or disable individual triggers, use the ALTER TRIGGER statement, which has the following syntax:

```
ALTER TRIGGER <trigger_name> {ENABLE | DISABLE}
```

To enable or disable all triggers attached to a table, use ALTER TABLE:

```
ALTER TABLE <table_name> {ENABLE | DISABLE} ALL TRIGGERS
```

4.5.3 Dropping Triggers

To drop a trigger, use the DROP TRIGGER statement, which has the following syntax:

```
DROP TRIGGER [schema.]trigger
```

4.5.4 Trigger Example

This example creates a trigger. It follows the development model described in "Model 2: Using the Attached Stored Procedure Development Model". For an example of creating triggers using the load and publish model, see "Trigger Arguments Example". In the example, you first create a table and a Java class. Then you attach the class to the table. And finally, you create and fire the trigger.

The `SalaryTrigger` class contains the `check_sal_raise` method. The method prints a message if an employee gets a salary raise of more than ten percent. The trigger fires the method before updating a salary in the EMP table.

Since `check_sal_raise` writes a message to standard output, use MSQL to issue the MSQL commands in the example. To start MSQL, invoke the Command Prompt and enter the following.

```
msql username/password@connect_string
```

`connect_string` is JDBC URL syntax. For example, to connect to the default database as user SYSTEM, at the Command Prompt.

```
msql system/passwd@jdbc:polite:polite
```

At the MSQL command line, create and populate the EMP table as follows.

```
CREATE TABLE EMP(E# int, name char(10), salary real,
   Constraint E#_PK primary key (E#));

INSERT INTO EMP VALUES (123,'Smith',60000);
INSERT INTO EMP VALUES (234,'Jones',50000);
```

Place the following class in **SalaryTrigger.java**:

```
class SalaryTrigger {
   private int eno;
   public SalaryTrigger(int enum) {
      eno = enum;
   }
   public void check_sal_raise(float old_sal,
      float new_sal)
   {
      if (((new_sal - old_sal)/old_sal) > .10)
      {
         // raise too high  do something here
         System.out.println("Raise too high for employee " + eno);
      }
   }
}
```

The SalaryTrigger class constructor takes an integer, which it assigns to attribute eno (the employee number). An instance of SalaryTrigger is created for each row (that is, for each employee) in the table EMP.

The check_sal_raise method is a non-static method. To execute, it must be called by an object of its class. Whenever the salary column of a row in EMP is modified, an instance of SalaryTrigger corresponding to that row is created (if it does not already exist) with the employee number (*E#)* as the argument to the constructor. The trigger then calls the check_sal_raise method.

After creating the Java class, you attach it to the table, as follows:

```
ALTER TABLE EMP ATTACH JAVA SOURCE "SalaryTrigger" IN '.'
   WITH CONSTRUCTOR ARGS(E#);
```

This statement directs Oracle Database Lite to compile the Java source file **SalaryTrigger.java** found in the current directory, and attach the resulting class to the EMP table. The statement also specifies that, when instantiating the class, Oracle Database Lite should use the constructor that takes as an argument the value in the E# column.

After attaching the class to the table, create the trigger as follows:

```
CREATE TRIGGER CHECK_RAISE BEFORE UPDATE OF SALARY ON EMP FOR EACH ROW
   "check_sal_raise"(old.salary, new.salary);
/
```

This statement creates a trigger called check_raise, which fires the check_sal_raise method before any update to the salary column of any row in EMP. Oracle Database Lite passes the old value and the new value of the salary column as arguments to the method.

In the example, a row-level trigger fires a row-level procedure (a non-static method). A row-level trigger can also fire table-level procedures (static methods). However, because statement-level triggers are fired once for an entire statement and a statement may affect multiple rows, a statement-level trigger can only fire a table-level procedure.

The following command updates the salary and fires the trigger:

```
UPDATE EMP SET SALARY = SALARY + 6100  WHERE E# = 123;
```

This produces the following output:

```
Raise too high for employee 123
```

4.5.5 Trigger Arguments

If using attached stored procedures, as described in "Model 2: Using the Attached Stored Procedure Development Model", row-level triggers do not support Java-to-SQL type conversion. Therefore, the Java datatype of a trigger argument must match the corresponding SQL datatype (shown in section "Java Datatypes") of the trigger column. However, if you are using the load and publish model, Oracle Database Lite supports datatype casting.

Table 4–4 describes how trigger arguments work in each type of column.

Table 4–4 Trigger Arguments

Trigger Argument	New Column Access	Old Column Access
`insert`	Yes	No
`delete`	No	Yes
`update`	Yes	Yes

Note: Triggers that have a `java.sql.Connection` object as an argument may be used only with applications that use the relational model.

4.5.6 Trigger Arguments Example

The following example shows how to create triggers that use IN/OUT parameters.

1. First, create the Java class `EMPTrigg`.

```
import java.sql.*;

public class EMPTrigg {
   public static final String goodGuy = "Oleg";

   public static void NameUpdate(String oldName, String[] newName)
   {
      if (oldName.equals(goodGuy))
         newName[0] = oldName;
   }

   public static void SalaryUpdate(String name, int oldSalary,
          int newSalary[])
   {
      if (name.equals(goodGuy))
         newSalary[0] = Math.max(oldSalary, newSalary[0])*10;
   }

   public static void AfterDelete(Connection conn,
                     String name, int salary) {
      if (name.equals(goodGuy))
         try {
            Statement stmt = conn.createStatement();
            stmt.executeUpdate(
             "insert into employee values('" + name + "', " +
                       salary + ")");
            stmt.close();
         } catch(SQLException e) {}
     }
 }
```

2. Create a new table EMPLOYEE and populate it with values.

```
CREATE TABLE EMPLOYEE(NAME VARCHAR(32), SALARY INT);
INSERT INTO EMPLOYEE VALUES('Alice', 100);
INSERT INTO EMPLOYEE VALUES('Bob', 100);
INSERT INTO EMPLOYEE VALUES('Oleg', 100);
```

3. Next, load the class into Oracle Database Lite.

```
CREATE JAVA CLASS USING BFILE ('c:\myprojects', 'EMPTrigg.class');
```

4. Use the CREATE PROCEDURE statement to publish the `EMPTrigg` methods that you want to call:

```
CREATE PROCEDURE NAME_UPDATE(
   OLD_NAME IN VARCHAR2, NEW_NAME IN OUT VARCHAR2)
   AS LANGUAGE JAVA NAME
   'EMPTrigger.NameUpdate(java.lang.String, java.lang.String[])';
   /

CREATE PROCEDURE SALARY_UPDATE(
   ENAME VARCHAR2, OLD_SALARY INT, NEW_SALARY IN OUT INT)
   AS LANGUAGE JAVA NAME
   'EMPTrigger.SalaryUpdate(java.lang.String, int, int[])';
   /

CREATE PROCEDURE AFTER_DELETE(
   ENAME VARCHAR2, SALARY INT)
   AS LANGUAGE JAVA NAME
   'EMPTrigger.AfterDelete(java.sql.Connection,
           java.lang.String, int)';
   /
```

5. Now, create a trigger for each procedure:

```
CREATE TRIGGER NU BEFORE UPDATE OF NAME ON EMPLOYEE FOR EACH ROW
   NAME_UPDATE(old.name, new.name);

CREATE TRIGGER SU BEFORE UPDATE OF SALARY ON EMPLOYEE FOR EACH ROW
   SALARY_UPDATE(name, old.salary, new.salary);

CREATE TRIGGER AD AFTER DELETE ON EMPLOYEE FOR EACH ROW
   AFTER_DELETE(name, salary);
```

6. Enter the following commands to fire the triggers and view the results:

```
SELECT * FROM EMPLOYEE;
UPDATE EMPLOYEE SET SALARY=0 WHERE NAME = 'Oleg';
SELECT * FROM EMPLOYEE;

DELETE FROM EMPLOYEE WHERE NAME = 'Oleg';
SELECT * FROM EMPLOYEE;

UPDATE EMPLOYEE SET NAME='TEMP' WHERE NAME = 'Oleg';
DELETE FROM EMPLOYEE WHERE NAME = 'TEMP';

SELECT * FROM EMPLOYEE;
```

5

Java Support on Windows CE

This chapter describes Java support for Windows CE devices using the Java Interface. Topics include:

5.1 Overview

Using the Java interface for Mobile Sync client-side synchronization tasks, programs written in Java can use the functionality provided by the OCAPI library. The Java interface resides in the `oracle.lite.msync` package.

The Java interface provides for the following functions:

- Setting client side user profiles containing data such as user name, password, and server
- Starting the synchronization process
- Tracking the progress of the synchronization process

The Java interface consists of two files, `mSync.jar` and `msync_java.dll`. To use the Java interface, the `mSync.jar` file must be included in the classpath. The `mSync.jar` file is located in the following directory.

```
<Oracle_home>\Mobile\classes
```

The `msync_java.dll` file is located in the following directory.

```
<Oracle_home>\Mobile\bin
```

There are four parts to the Java interface. They are:

- Sync Class
- SyncException Class
- SyncOption Class
- SyncProgressListener Interface

The following sections describe the Java interface.

5.2 Sync Class

This class initiates synchronization by using the provided synchronization options. The parameters for the constructor are listed in Table 5–1.

Constructors

`Sync(SyncOption option)`

Table 5–1 Sync Class Constructor

Parameter	Description
`option`	Instance of the `SyncOption` Class. This contains all the parameters needed to perform synchronization.

Public Methods

To monitor the progress of the synchronization process, the public method `SyncProgressListener` adds a progress listener to the object.

`SyncProgressListener add(ProgressListener listener)`

The parameters for the `SyncProgressListener` method are described in Table 5–2.

Table 5–2 Sync Class Public Method

Parameter	Description
`listener`	An object that implements the `ProgressListener` interface. The synchronization object calls the `progress()` function of this object to notify it of the synchronization progress.
`void doSync ()`	Starts a synchronization session and blocks that thread until synchronization is complete.
`void abort ()`	Aborts the synchronization session.

The following code demonstrates how to start a session using the default settings.

```
try

{
  Sync mySync = new Sync( new SyncOption());
  mySync.doSync();
}
catch ( SyncException e)
{
  System.err.println( "Sync Error:"+e.getMessage());
}
```

5.3 SyncException Class

This class signals a non recoverable error during the synchronization process. The `SyncException()` Class constructs a "`clear`" object. The parameters for the constructor are listed inTable 5–3:

Constructors

```
SyncException()

SyncException(int errorCode, string errorMessage)
```

Table 5–3 syncException Constructor Parameter Description

Parameter	Description
`errorCode`	The error. Refer the *Oracle Database Lite Message Reference*.
`errorMessage`	A readable text message that provides extra information.

Public Methods

The methods for the `SyncException` are listed in Table 5–4.

Table 5–4 SyncExceptionClass Public Methods

Parameters	Description
`int getErrorCode()`	Gets the error code.
`String getErrorMessage`	Gets the error message.

5.4 SyncOption Class

The `SyncOption` class is used to define the parameters for the synchronization process. It can either be constructed manually, or can save or load data from the user profile.

Constructors

```
SyncOption()

SyncOption

   ( String user,

     String password,

     String syncParam,

     String transportDriver,

     String transportParam)
```

The parameters for the SyncOption constructor are listed in Table 5–5:

Table 5–5 SyncOption Constructors

Parameter	Description
`user`	A string containing the name used for authentication by the Mobile Server.
`password`	A string containing the user's password.
`syncParam`	A string which defines an optional list of parameters for the synchronization session. See Section 5.5, "Java Interface SyncParam Settings" for more information.
`transportDriver`	A string containing the name of the transport driver. Currently, only "HTTP" is supported.

Table 5–5 (Cont.) SyncOption Constructors

Parameter	Description
`transportParam`	A string containing all the parameters needed for the specified driver to operate. See Section 5.6, "Java Interface TransportParam Parameters" for more information.
`priority`	A boolean value which limits synchronization to server tables flagged as high priority, otherwise all tables are synchronized.
`pushOnly`	A boolean value which makes synchronization push only.

Public Methods

These methods load and save the user profile. The parameters of the public methods are listed in Table 5–6:

Table 5–6 Sync Option Public Method Parameters

Parameter	Description
`void load(String username)`	This loads the profile for the specified user name. If the user name is left null, the profile is loaded for the last user to synchronize.
`void save()`	This saves the settings to the profile for the active user.
`void setUser(String username)`	This is used to set and get the current user.
`String getuser()`	
`void setPassword(String password)String getPassword()`	This is used to set and get the password.
`void setSyncParam(String syncParam) string getSyncParam()`	This is used to set and get the synchronization parameters.
`void setTransportDriver(String driverName) String getTransportDriver()`	This is used to set and get the driver name. Release 5.0.2 supports the "HTTP" driver.
`void setTransportParam(String transportParam) String getTransportParam()`	Set and get the transport parameters.

Example

The following code example demonstrates how to start a synchronization session using the default settings:

```
SyncOption opt = new SyncOption

("sam","lion","pushonly","HTTP","server=server1;proxy=www-proxy.us.oracle.com;prox
yPort=80");

opt.save();
```

5.5 Java Interface SyncParam Settings

The `syncParam` is a string that can be passed when creating the `SyncOption` object. It allows support parameters to be specified to the synchronization session. The string is constructed of name-and-value pairs. For example:

```
"name=value;name2=value2;name3=value3, ...;"
```

The names are not case sensitive, but the values are. The field names which can be used are listed in Table 5–7.

Table 5–7 Java Interface SyncParamSettings

Name	Value/Options	Description
"reset"	N/A	Clear all entries in the environment before applying any remaining settings.
"security"	SSL or CAST5	Use the appropriate selection to choose either SSL or CAST5 stream encryption.
"push only"	N/A	Use this setting to upload changes from the client to the server only, do not download.This is useful when data transfer is one way, client to server.
"noapps"	N/A	Do not download any new or updated applications. This is useful when synchronizing over slow connection or on a slow network.
"syncDirection"	"sendonly" "receiveonly"	"SendOnly" is the same as "pushonly". "ReceiveOnly" allows no changes to be posted to the server.
"noNewPubs"	N/A	This setting prevents any new publications created since the last synchronization from being sent, and only synchronizes data from the current publications.
"tableFlag"	"enable"	The "enable" setting allows [Publication.Item] to be synchronized, "disable" prevents synchronization.
[Publication.Item]	"disable"	
"fullrefresh"	N/A	Forces a complete refresh.
"clientDBMode"	"EMBEDDED" or "CLIENT"	If set to "EMBEDDED", access to the database is by conventional ODBC, if set to "CLIENT" access is by multi-client ODBC.

Example 1

The first example enables SSL security and disables application deployment for the current synchronization session:

```
"security=SSL; noapps;"
```

Example 2

The second example resets all previous settings, activates upload for the "Dept" table only:

```
"reset;pushOnly;tableFlag[TestApp.Emp]=disable;tableFlag[TestApp.Dept]=enable;"
```

5.6 Java Interface TransportParam Parameters

The format of the TransportParam string is used to set specific parameters using a string of name-and-value pairs, for example:

```
"name=value;name2=value2;name3=value3, ...;"
```

The names are not case sensitive, but the values are. The field names which can be used are listed in Table 5–8.

Table 5–8 TransportParam Parameters

Name	Value	Description
`"reset"`	N/A	Clear all entries in the environment before applying the rest of the settings.
`"server"`	server hostname	The hostname or IP address of the Mobile Server.
`"proxy"`	proxy server hostname	The hostname or IP address of the proxy server.
`"proxyPort"`	port number	The port number of the proxy server.
`"cookie"`	cookie string	The cookie to be used for transport.

Example

The example directs the Mobile Sync engine to use the server at "test.oracle.com" through the proxy "proxy.oracle.com" at port 8080:

```
"server=test.oracle.com;proxy=proxy.oracle.com;proxyPort=8080;"
```

5.7 SyncProgress Listener Service

The SyncProgressListener is an interface that allows progress updates to be trapped during synchronization.

This class initiates synchronization by using the provided synchronization options. The parameters for the method are listed in Table 5–9:

Method

```
void progress

   (int progressType,

    int completed);
```

Table 5–9 SyncProgressListener Abstract Method

Parameter	Description
`progressType`	This is set to one of the constants listed in Table 5–10.
`completed`	This is the percentage of completion for specific `progressType`.

The names of the constants which report the synchronization progress are listed in Table 5–10.

Table 5–10 SyncProgressListener Interface Constants

Constant Name	Progress Type
`PT_INT`	States that the synchronization engine is in the initializing stage. The current and total counts are set to 0.

Table 5–10 (Cont.) SyncProgressListener Interface Constants

Constant Name	Progress Type
PT_PREPARE_SEND	States that the synchronization engine is preparing local data to be sent to the server. This includes getting locally modified data. For streaming implementations this takes a shorter amount of time.
PT_SEND	States that the synchronization engine is sending data to the network.
	The total count equals the number of bytes to be sent, and the current count equals the byte count being sent currently.
PT_RECV	States that the synchronization engine is receiving data from the server.
	The total count equals the number of bytes to be received, and the current count equals the byte count being received currently.
PT_PROCESS_RECV	States that the synchronization engine is applying the newly received data from the server to the local data stores.
PT_COMPLETE	States that the synchronization engine has completed the synchronization process.

Example

This simple class implements the `SyncProgressListener`.

```
class myProgressTracker implements SyncProgress Listener;

{
  public void progress
     (int progressType,
     int completed)
    {
      System.out.println( "Status: "+progressType+"="+ completed+"%" );
     } //progress
 }
```

A

Stored Procedure Tutorial

This appendix demonstrates how to create a Java stored procedure and trigger. Topics include:

- Section A.1, "Creating a Stored Procedure and Trigger"
- Section A.2, "Create a Trigger"
- Section A.3, "Commit or Roll Back"

A.1 Creating a Stored Procedure and Trigger

In this tutorial, you create a Java class `EMAIL`, load the class into Oracle Database Lite, publish its method to SQL, and create a trigger for the method. The `EMAIL` class appears in the source file **EMAIL.java**, and is available in the Java examples directory at the following location.

```
<Oracle_home>\Mobile\Sdk\Samples\JDBC
```

`EMAIL` has a method named `assignEMailAddress`, which generates an email address for an employee based on the first letter of the employee's first name and up to seven letters of the last name. If the address is already assigned, the method attempts to find a unique email address using combinations of letters in the first and last name.

After creating the class, you load it into Oracle Database Lite using MSQL. For this example you use the SQL statement CREATE JAVA. Alternatively, you can use the `loadjava` utility to load the class into Oracle Database Lite. After loading the class, you publish the `assignEMailAddress` method to SQL.

Finally, you create a trigger that fires the `assignEMailAddress` method whenever a row is inserted into T_EMP, the table that contains the employee information.

As arguments, `assignEMailAddress` takes a JDBC connection object, the employee's identification number, first name, middle initial, and last name. Oracle Database Lite supplies the JDBC connection object argument. You do not need to provide a value for the connection object when you execute the method. `assignEMailAddress` uses the JDBC connection object to ensure that the generated e-mail address is unique.

A.1.1 Start MSQL

Start MSQL and connect to the default Oracle Database Lite. Since the Java application in this tutorial prints to standard output, use the DOS version of MSQL. From a DOS prompt, type:

```
msql system/mgr@jdbc:polite:polite
```

The SQL prompt should appear.

A.1.2 Create a Table

To create a table, type:

```
CREATE TABLE T_EMP(ENO INT PRIMARY KEY,
   FNAME VARCHAR(20),
   MI CHAR,
   LNAME VARCHAR(20),
   EMAIL VARCHAR(8));
```

A.1.3 Create a Java Class

Create and compile the Java class `EMAIL` in the file **EMAIL.java** in C:\tmp. **EMAIL.java** implements the `assignEMailAddress` method. The code sample given below lists the contents of this file. You can copy this file from the following location.

```
<Oracle_home>\Mobile\Sdk\Samples\JDBC
```

```
import java.sql.*;

public class EMAIL {
   public static void assignEMailAddress(Connection conn,
            int eno, String fname,String lname)
            throws Exception
   {
      Statement stmt = null;
      ResultSet retset = null;
      String emailAddr;
      int i,j,fnLen, lnLen, rowCount;

      /* create a statement */
      try {
         stmt = conn.createStatement();
      }
      catch (SQLException e)
      {
         System.out.println("conn.createStatement failed: " +
         e.getMessage() + "\n");
         System.exit(0);
      }
      /* check fname and lname */
      fnLen = fname.length();
      if(fnLen > 8) fnLen = 8;
      if (fnLen == 0)
         throw new Exception("First name is required");
      lnLen = lname.length();
      if(lnLen > 8) lnLen = 8;
      if (lnLen == 0)
         throw new Exception("Last name is required");
      for (i=1; i <= fnLen; i++)
      {
         /* generate an e-mail address */
         j = (8-i) > lnLen? lnLen:8-i;
         emailAddr =
               new String(fname.substring(0,i).toLowerCase()+
               lname.substring(0,j).toLowerCase());
         /* check if this e-mail address is unique  */
         try {
            retset = stmt.executeQuery(
```

```
                    "SELECT * FROM T_EMP  WHERE email = '"+
                    emailAddr+"'");
                if(!retset.next()) {
                   /* e-mail address is unique;
                   * so update the email column */
                   retset.close();
                   rowCount = stmt.executeUpdate(
                       "UPDATE T_EMP SET EMAIL = '"
                       + emailAddr + "' WHERE ENO = "
                       + eno);
                   if(rowCount == 0)
                      throw new Exception("Employee "+fname+ " " +
                              lname + " does not exist");
                   else return;
                }
             }
             catch (SQLException e) {
                while(e != null) {
                   System.out.println(e.getMessage());
                   e = e.getNextException();
                }
             }
          }
          /* Can't find a unique name */
          emailAddr = new String(fname.substring(0,1).toLowerCase() +
               lname.substring(0,1).toLowerCase() + eno);
          rowCount = stmt.executeUpdate(
               "UPDATE T_EMP SET EMAIL = '"
               + emailAddr + "' WHERE ENO = "
               + eno);
          if(rowCount == 0)
             throw new Exception("Employee "+fname+ " " +
                  lname + " does not exist");
          else return;
       }
    }
```

A.1.4 Load the Java Class File

To load the `EMAIL` class file into Oracle Database Lite, type:

```
CREATE JAVA CLASS USING BFILE
   ('c:\tmp', 'EMAIL.class');
```

If you want to make changes to the class after loading it, you need to:

1. Drop the class from the database, using `dropjava` or DROP JAVA CLASS
2. Commit your work
3. Exit MSQL
4. Restart MSQL

This unloads the class from the Java Virtual Machine.

A.1.5 Publish the Stored Procedure

You make the stored procedure callable from SQL by creating a call specification (call spec) for it. Since `assignEMailAddress` does not return a value, use the CREATE PROCEDURE command, as follows:

```
CREATE OR REPLACE PROCEDURE
   ASSIGN_EMAIL(E_NO INT, F_NAME VARCHAR2, L_NAME VARCHAR2)
   AS LANGUAGE JAVA NAME 'EMAIL.assignEMailAddress(java.sql.Connection,
int, java.lang.String,
     java.lang.String)';
```

A.1.6 Populate the Database

Insert a row into T_EMP:

```
INSERT INTO T_EMP VALUES(100,'John','E','Smith',null);
```

A.1.7 Execute the Procedure

To execute the procedure, type:

```
SELECT ASSIGN_EMAIL(100,'John','Smith')
  FROM dual
```

A.1.8 Verify the Email Address

To see the results of the ASSIGN_EMAIL procedure, type:

```
SELECT * FROM T_EMP;
```

This command produces the following output:

```
     ENO  FNAME                M LNAME                EMAIL
     ---- -------------------   - -------------------- --------
     100  John                  E Smith                jsmith
```

A.2 Create a Trigger

To make ASSIGN_EMAIL execute whenever a row is inserted into T_EMP, create an AFTER INSERT trigger for it. Create the trigger as follows:

```
CREATE TRIGGER EMP_TRIGG AFTER INSERT ON T_EMP FOR EACH ROW
  ASSIGN_EMAIL(eno,fname,lname);
```

A trigger named EMP_TRIGG fires every time a row is inserted into T_EMP. The actual arguments for the procedure are the values of the columns `eno`, `fname`, and `lname`.

You do not need to specify a `connection` argument.

A.2.1 Testing the Trigger

Test the trigger by inserting a row into T_EMP:

```
INSERT INTO T_EMP VALUES(200,'James','A','Smith',null);
```

A.2.2 Verify the Email Address

Issue a SELECT statement to verify that the trigger has fired:

```
SELECT * FROM T_EMP;
   ENO FNAME                M LNAME                EMAIL
   --- -------------------- - -------------------- --------
   100 John                 E Smith                jsmith
```

```
200 James               A Smith               jasmith
```

A.3 Commit or Roll Back

Finally, commit your changes to preserve your work, or roll back to cancel changes.

B

Sample Programs

This appendix provides instructions for using the sample Java programs provided with Oracle Database Lite. Topics include:

- Section B.1, "Java Samples Overview"
- Section B.2, "Running the Samples"

B.1 Java Samples Overview

The `<Oracle_home>\Mobile\SDK\Samples\JDBC` directory contains sample programs that demonstrate the use of Java stored procedures, Java Replication Classes, and JDBC with Oracle Database Lite.

The Java examples directory contains these files:

1. Stoproex.sql
2. INVENTORY.java
3. JDBCEX.java
4. plsqlex.sql
5. PLSQLEX.java

The following sections describe the samples. Java class, method, and file names are case-sensitive. When running Java programs from SQL, you must enclose names in double quotes to preserve their case.

B.1.1 JDBC Sample

The file **JDBCEX.java** contains a sample Java program that uses JDBC classes to select the rows of the PRODUCT table and display information.

B.1.2 PL/SQL Conversion to Java Samples

You can convert stored procedures and triggers written in Oracle's PL/SQL language to Java. Several of the Java programs in **PLSQLEX.java** correspond to PL/SQL programs described in the *Oracle Server PL/SQL Users Guide and Reference* manual. **Plsqlex.sql** contains SQL statements that invoke the Java stored procedures.

B.1.3 Java Stored Procedures Sample

The Java stored procedures sample shows how to manually attach a class to an Oracle Database Lite table. Alternatively, you can load the class into the Oracle Database Lite database using `loadjava`, and publish its methods to SQL using the CREATE

PROCEDURE or CREATE FUNCTION statements. In this model, you do not attach the class to a database table. For more information on the publish model of developing stored procedures, see "Model 1: Using the Load and Publish Stored Procedure Development Model" in Chapter 4, "Java Stored Procedures and Triggers".

The file **Stoproex.sql** contains SQL statements to create a sample schema. You must run this script using MSQL before running the Java samples. The sample schema contains the following three tables:

Table	Description
PRODUCT	Stores information about products.
PRODUCT_COMPOSITION	Stores information about the composition of products. Each row of the table keeps track of the quantity of a sub-product required to build the product.
INVENTORY	Stores the quantity of products in the warehouse.

Stoproex.sql also contains statements that insert rows into the tables, attach a Java class to the INVENTORY table, and create an AFTER UPDATE trigger in the INVENTORY table's QTY column.

The Java class attached to the INVENTORY table is defined in the file, **INVENTORY.java**. It has one static method called `SHIP_PRODUCT`, and two non-static (instance) methods called `SHIP` and `CHECK_INVENTORY`.

The `SHIP_PRODUCT` method takes three arguments: a connection object, a product ID, and the quantity of the product to be shipped to the customer.

Stoproex.sql invokes the method with the following SQL statement:

```
SELECT inventory.ship_product(100,1) FROM DUAL FOR UPDATE;
```

Notice the following:

1. Static methods must be referred to as *table_name.method_name*. The FROM clause for static method execution must always refer to the pseudo table DUAL.
2. SQL converts `inventory.ship_product` into uppercase because the method name is `SHIP_PRODUCT` in **INVENTORY.java**. If you name the table "Inventory" and the method "`shipProduct`", you must double-quote both names: `"Inventory"."shipProduct"`.
3. The connection object is not explicitly given in the arguments to the method. Oracle Database Lite supplies the current connection for any argument of type `java.sql.Connection`.

The Java method `SHIP` uses JDBC classes to access Oracle Database Lite. It creates a statement from the connection passed as an argument and executes a SELECT statement. The SELECT statement executes the Java non-static method `SHIP`, also defined in **INVENTORY.java**.

The method `SHIP` updates the quantity of products to ship. Since `SHIP` is a non-static method, Oracle Database Lite creates an instance of the class `INVENTORY` before calling this method. It creates the instance using the constructor that takes the columns specified in the WITH CONSTRUCTOR ARGS clause of the ATTACH statement.

Since this sample creates an AFTER UPDATE trigger on the QTY column of the INVENTORY table, each UPDATE executes the `CHECK_INVENTORY` method. Since `CHECK_INVENTORY` is a non-static method, Oracle Database Lite uses the row instance or creates a new instance if one does not exist.

If the updated quantity-on-hand drops below the inventory threshold, the CHECK_INVENTORY method uses the PRODUCT_COMPOSITION table to look up the constituent parts of the product. It also updates the quantity of each to reflect the fact that a certain quantity of this product must be manufactured to replenish inventory. This update happens recursively until an end product is reached, at which point CHECK_INVENTORY places an order for the product.

B.2 Running the Samples

To run the Java samples:

1. Go to the samples directory, <Oracle_home>\Mobile\Sdk\Samples\JDBC. For example:

   ```
   cd <Oracle_home>\Mobile\Sdk\Samples\JDBC
   ```

2. Add "." (the current directory) to the CLASSPATH, if it is not already included:

   ```
   set CLASSPATH=.;%CLASSPATH%
   ```

3. Execute the SQL scripts using the DOS command-line version of MSQL. For example:

   ```
   msql system/mgr@jdbc:polite @stoproex.sql
   ```

B.2.1 Running the JDBC Sample

To use the JDBC sample, install the PRODUCT table in Oracle Database Lite by running the **Stoproex.sql** script:

```
msql system/mgr@jdbc:polite @stoproex.sql
```

Compile the source file using the command:

```
javac JDBCEX.java
```

Run the compiled class:

```
java JDBCEX
```

B.2.2 Running the PL/SQL Conversion Samples

To run **PLSQLEX.java**, start MSQL:

```
msql system/mgr@jdbc:polite
```

At the MSQL prompt, run the script:

```
@plsqlex.sql
```

Attach the Java source file **PLSQLEX.java** to the table:

```
alter table temp attach java source "PLSQLEX" in '.';
```

To execute the table method, type:

```
select temp."sampleOne"() from dual for update;
```

To view the results:

```
select * from temp;
```

See the file **PLSQLEX.java** for information regarding additional samples. The samples are named `sampleOne` to `sampleFour`.

B.2.3 Running the Java Stored Procedures Sample

Run the **Stoproex.sql** script to install the tables and stored procedures required for the stored procedures sample:

```
msql system/mgr@jdbc:polite @stoproex.sql
select inventory.ship_product(p,q) from dual;
```

When the script completes, display the contents of the inventory table. At the MSQL prompt, type:

```
select * from inventory;

PID   QTY THRESHOLD
---- ---- ---------
100     1     1
101    -6     2
102   -26     8
103   -26     8
```

Negative numbers in the table indicate that parts 101, 102, and 103 need to be restocked.

Index

F

G

I

J

L

M

N

O

P

Q

R

S

T

U

V

W

Z