Oracle Lite

Java 開発者ガイド

リリース 4.0.1

2000 年 10 月

部品番号：J02410-01

ORACLE®

Oracle Lite Java 開発者ガイド , リリース 4.0.1

部品番号：J02410-01

原本名：Oracle Lite Java Developer's Guide, Release 4.0.1

原本部品番号：A86148-01

Copyright © 2000, Oracle Corporation. All rights reserved.

Printed in Japan.

制限付権利の説明

プログラム（ソフトウェアおよびドキュメントを含む）の使用、複製または開示は、オラクル社との契約に記された制約条件に従うものとします。著作権、特許権およびその他の知的財産権に関する法律により保護されています。

当プログラムのリバース・エンジニアリング等は禁止されております。

このドキュメントの情報は、予告なしに変更されることがあります。オラクル社は本ドキュメントの無謬性を保証しません。

* オラクル社とは、Oracle Corporation（米国オラクル）または日本オラクル株式会社（日本オラクル）を指します。

危険な用途への使用について

オラクル社製品は、原子力、航空産業、大量輸送、医療あるいはその他の危険が伴うアプリケーションを用途として開発されておりません。オラクル社製品を上述のようなアプリケーションに使用することについての安全確保は、顧客各位の責任と費用により行ってください。万一かかる用途での使用によりクレームや損害が発生いたしましても、日本オラクル株式会社と開発元である Oracle Corporation（米国オラクル）およびその関連会社は一切責任を負いかねます。当プログラムを米国国防総省の米国政府機関に提供する際には、『Restricted Rights』と共に提供してください。この場合次の Notice が適用されます。

Restricted Rights Notice

Programs delivered subject to the DOD FAR Supplement are "commercial computer software" and use, duplication, and disclosure of the Programs, including documentation, shall be subject to the licensing restrictions set forth in the applicable Oracle license agreement. Otherwise, Programs delivered subject to the Federal Acquisition Regulations are "restricted computer software" and use, duplication, and disclosure of the Programs shall be subject to the restrictions in FAR 52.227-19, Commercial Computer Software - Restricted Rights (June, 1987). Oracle Corporation, 500 Oracle Parkway, Redwood City, CA 94065.

このドキュメントに記載されているその他の会社名および製品名は、あくまでその製品および会社を識別する目的にのみ使用されており、それぞれの所有者の商標または登録商標です。

目次

はじめに ix

このマニュアルについて x
表記規約 xi

1 Oracle Lite の Java サポート

Oracle Lite の Java 機能 1-2
ネイティブな JDBC サポート 1-2
Java ストアド・プロシージャとトリガー 1-3
Java アクセス・クラス（JAC） 1-3
SQLJ 1-4
データベース・アクセス・モデル 1-4
Java データ型 1-5
Java ツール 1-6
Oracle Lite Designer 1-7
loadjava 1-8
Oracle Lite が提供する Java 開発環境 1-8
環境の設定 1-8
JDK 1.1 の場合の変数の設定 1-9
JDK 1.2 の場合の変数の設定 1-9

2 Java ストアド・プロシージャとトリガー

Oracle Lite の新機能 2-2
Java ストアド・プロシージャとトリガーの概要 2-2
ストアド・プロシージャの使用方法 2-3
モデル 1: Load and Publish ストアド・プロシージャ開発モデルの使用 2-4

クラスのロード 2-4
loadjava 2-5
CREATE JAVA の使用方法 2-8
SQL へのストアド・プロシージャのパブリッシュ 2-8
パブリッシュされたストアド・プロシージャのコール 2-10
パブリッシュされたストアド・プロシージャの削除 2-11
dropjava の使用 2-11
SQL コマンドの使用方法 2-13
例 2-13
モデル 2: アタッチ済ストアド・プロシージャ開発モデルの使用 2-15
表レベルのストアド・プロシージャ 2-15
行レベルのストアド・プロシージャ 2-15
表への Java クラスのアタッチ 2-15
アタッチ済ストアド・プロシージャのコール 2-16
アタッチ済ストアド・プロシージャの削除 2-17
例 2-18
Java ストアド・プロシージャの ODBC からのコール 2-19
Java データ型 2-20
パラメータの宣言 2-21
ストアド・プロシージャを使用して複数行を返す方法 2-21
ODBC を使用して複数行を返す方法 2-22
例 2-22
トリガーの使用方法 2-23
文レベルのトリガーと行レベルのトリガーの比較 2-23
トリガーの作成 2-23
トリガーの使用可能と使用禁止 2-24
トリガーの削除 2-24
トリガーの例 2-24
トリガーの引数 2-27

3 Java Database Connectivity（JDBC）

Oracle Lite の新機能 3-2
JDBC 3-2
JDBC 環境の設定 3-2
Oracle Lite JDBC ドライバのロード 3-2
Oracle Lite データベースへの接続 3-3

JDBC からの Java ストアド・プロシージャの実行 3-3
ExecuteQuery メソッドの使用方法 3-4
Callable Statement の使用方法 3-5
Oracle Lite での拡張 3-5
データ型の拡張 3-6
データ・アクセスの拡張 3-7
BLOB の読取りを行うサンプル・プログラム 3-9
CLOB への書込みを行うサンプル・プログラム 3-10
JDBC と Java アクセス・クラス間の相互運用性 3-10
制限事項 3-11

4 Java 埋込み SQL（SQLJ）

SQLJ の概要 4-2
SQLJ の操作 4-2
SQLJ の使用方法 4-3
SQLJ の構成 4-3
Oracle Lite への接続 4-3

5 Java アクセス・クラス（JAC）

Oracle Lite の新機能 5-2
単純永続インタフェース 5-2
データ型サポートの拡張 5-3
データ型変換サポート 5-3
Java ストアド・プロシージャとトリガーのサポート 5-4
ガベージ・コレクション 5-4
レプリケーション・サポート 5-5
順序オブジェクト 5-7
索引の拡張 5-7
行識別子（ROWID）のサポート 5-8
POLConnection 5-8
Java アクセス・クラス（JAC）の概要 5-9
Java プロキシ・クラス 5-9
一時 Java クラスと永続 Java クラス 5-10
一時 Java クラス 5-10
Oracle8*i* Lite の永続 Java クラス 5-10

Java アクセス・クラスの主な機能 5-12
ステートフルな永続化とステートレスな永続化 5-12
Java オブジェクトの一意性 5-12
継承 5-12
Java の NULL と Oracle Lite の NULL 5-13
バイト配列とバイナリ 5-13
大 / 小文字区別 5-13
永続可能属性のデータ型 5-14
データベースの管理 5-14
データベースの作成 5-14
データベースの削除 5-15
データベースへの接続 5-15
トランザクションの管理 5-16
トランザクションの明示的な開始 5-16
トランザクションのコミット 5-16
トランザクションのロールバック 5-17
新しいセーブポイントの設定 5-17
セーブポイントの削除 5-17
セーブポイントまでのロールバック 5-17
データベース・スキーマの作成 5-17
ステートレス永続 Java クラスの作成 5-19
例 1: 永続インタフェースの実装 5-20
例 2: 永続クラスの継承 5-22
オブジェクトの管理 5-23
永続オブジェクトの作成 5-23
永続オブジェクトの問合せ 5-23
永続オブジェクトの削除 5-24
永続オブジェクトの属性の取出し 5-25
永続オブジェクトの属性の更新 5-25
永続バイナリ属性の操作 5-25
制約 5-26
永続オブジェクト・ディレクトリ 5-27
Java ストアド・プロシージャとトリガー 5-28
クラスのロード 5-29
トリガーの作成 5-30
ストアド・プロシージャのコール 5-30

リレーショナル・データベースのアクセス 5-30
単純永続オブジェクトの使用方法 5-31
単純永続クラスの作成 5-31
オブジェクトの作成と格納 5-32
プロキシ・クラスの使用 5-32
永続オブジェクトの管理 5-33
delete 5-33
reload 5-33
setDirty 5-33
getInstances 5-34
単純永続オブジェクトにおけるオブジェクト参照 5-34
例 5-34
Human.java 5-35
Professor.java 5-35
Student.java 5-35
Course.java 5-36
University.java 5-36
Java アクセス・クラス・パッケージ 5-37

6 シームレスな永続 Java

シームレスな永続 Java の概要 6-2
シームレスな永続クラスの作成 6-2
アプリケーションの構造 6-4
シームレスな永続 Java ポストプロセッサ 6-5
ポストプロセッサのオプション 6-5
ポストプロセッサの起動 6-7
サンプル 1 6-7
サンプル 2 6-7
サンプル 3 6-7
サンプル 4 6-7
ポストプロセッサの規則と参照 6-8
クラス永続性規則 6-8
フィールド永続性規則 6-8
処理対象の Java クラス 6-8
処理済クラスの CLASSPATH への指定 6-8
手動による手順 6-8

セマンティクス 6-9
ポストプロセス後のデバッグ 6-9
ステートフル永続オブジェクトの管理 6-9
新しいオブジェクトの作成 6-10
オブジェクトの削除 6-10
オブジェクトの取出し 6-10
オブジェクトのフィールド値の走査 6-10
イテレータ 6-10
属性値のアクセス 6-11
オブジェクトの変更 6-11
配列属性の変更 6-11
オブジェクトの廃棄 6-12
PostActivate 6-12
PriorDelete 6-12
トランザクション処理後またはデータベース切断後のオブジェクトの状態 6-13
リフレッシュ 6-13
ステートフル永続クラスの手動による実装 6-13
POLPersistentCapable インタフェースの実装 6-14
Persistent Glue Protocol 6-14
アクティブ化コンストラクタ 6-15
readObject 6-16
writeObject 6-17
オブジェクト状態のロード 6-18
更新の準備 6-18
書込み後のオブジェクトへのダーティのマーク付け 6-19
完全な例 6-19

A ストアド・プロシージャのチュートリアル

ストアド・プロシージャとトリガーの作成 A-2
SQL*Plus の起動 A-2
表の作成 A-2
Java クラスの作成 A-3
Java クラスのロード A-4
ストアド・プロシージャのパブリッシュ A-5
データベースへの値の移入 A-5
プロシージャの実行 A-5

電子メールアドレスの検証 A-6
トリガーの作成 A-6
トリガーのテスト A-6
電子メール・アドレスの検証 A-6
コミットまたはロールバック A-7

B JDBC アプリケーションのチュートリアル

環境の検査 B-2
アプリケーションへの JDBC ドライバのロード B-2
Oracle Lite データベースへの接続 B-2
問合せの発行と結果の参照 B-6
トランザクションのコミット B-7
完全な例 B-7
この例のコンパイルと実行 B-8

C JAC アプリケーションのチュートリアル

環境の検査 C-2
データベースへの接続 C-2
新規クラスの作成 C-2
新規オブジェクトの作成 C-3
既存オブジェクトの問合せ C-3
オブジェクトの属性値の検査 C-4
トランザクションのコミット C-4
データベースからの切断 C-4
完全な例 C-4
この例のコンパイルと実行 C-6

D サンプル・プログラム

Java サンプル・プログラムの概要 D-2
Java アクセス・クラス（JAC）のサンプル D-2
Java レプリケーション・クラスのサンプル D-3
JDBC のサンプル D-3
PL/SQL から Java サンプルへの変換 D-3
Java ストアド・プロシージャのサンプル D-3
トリガーのサンプル D-5

サンプルの実行 D-5
Java アクセス・クラス（JAC）のサンプルの実行 D-5
Java レプリケーション・クラスのサンプルの実行 D-6
JDBC のサンプルの実行 D-6
PL/SQL 変換のサンプルの実行 D-7
Java ストアド・プロシージャのサンプルの実行 D-7
トリガーのサンプルの実行 D-8

索引

はじめに

ここでは、『Oracle Lite Java 開発者ガイド』を紹介します。説明する内容は、次のとおりです。

- このマニュアルについて
- 表記規約

このマニュアルについて

『Oracle Lite Java 開発者ガイド』は、Oracle Lite の Java 機能を解説したマニュアルです。このマニュアルの使用にあたって、Java プログラミング言語を使い慣れていることが前提です。

このマニュアル『Oracle Lite Java 開発者ガイド』の構成は、次のとおりです。

第 1 章「Oracle Lite の Java サポート」	Oracle Lite の Java 機能を紹介し、開発環境の設定方法について説明します。
第 2 章「Java ストアド・プロシージャとトリガー」	Oracle Lite での Java ストアド・プロシージャとトリガーの開発方法を説明します。
第 3 章「Java Database Connectivity（JDBC）」	Oracle Lite での JDBC サポートを説明します。
第 4 章「Java 埋込み SQL（SQLJ）」	Java 埋込み SQL の標準である SQLJ の概要を説明します。
第 5 章「Java アクセス・クラス（JAC）」	JAC を使用して Oracle Lite にステートレス永続 Java クラスを作成する方法を説明します。
第 6 章「シームレスな永続 Java」	Oracle Lite のシームレスな永続 Java ポストプロセッサの使用方法を説明します。ステートフル永続 Java クラスを手動で作成する方法も説明します。
付録 A「ストアド・プロシージャのチュートリアル」	ストアド・プロシージャの作成と実行に関するチュートリアルを提供します。
付録 B「JDBC アプリケーションのチュートリアル」	Oracle Lite データベースにアクセスする JDBC プログラムの作成に関するチュートリアルを提供します。
付録 C「JAC アプリケーションのチュートリアル」	JAC プログラムの作成に関するチュートリアルを提供します。
付録 D「サンプル・プログラム」	Oracle Lite に同梱の Java のサンプル・プログラムについて説明します。

注意： このマニュアルでは、「Oracle8*i*」および「Oracle8*i* Server」という記述は、Oracle8*i* のリリース 8.1.5 以上を意味します。

表記規約

このマニュアルでの表記規約は次のとおりです。

イタリック	イタリックは変数を表します。変数には適切な値を代入します。
...	コードの一部で使用される省略記号は、その説明内容に関係のないコードを省略していることを意味します。
[ ]	SQL 文では、オプション項目は大カッコで囲まれます。たとえば、[a | b] は、*a* と *b* がどちらもオプションの引数であることを表します。カッコは入力しないでください。
{ }	中カッコで囲まれている項目は、いずれか 1 つを選択します。たとえば、*{a|b}* は、*a* か *b* の（両方ではなく）どちらかを選択する必要があることを表します。
|	垂直バーは、2 つのオプションがある場合に「または」を意味します。入力するオプションは 1 つのみです。垂直バー自体は入力しないでください。
英大文字	テキスト内の英大文字は、SELECT や UPDATE などの SQL キーワードを表します。

1

Oracle Lite の Java サポート

この章では、Oracle Lite でサポートされている Java インタフェースとツールを紹介します。説明する内容は、次のとおりです。

- Oracle Lite の Java 機能
- データベース・アクセス・モデル
- Java ツール
- Oracle Lite が提供する Java 開発環境

Oracle Lite の Java 機能

豊富な Java サポートを備え、コンパクトで、しかも Oracle8*i* との相互運用が可能な Oracle Lite は、モバイルなアプリケーションとデータの配置と管理には理想的なプラットフォームです。Oracle Lite の Java 機能には、広範囲なソフトウェア開発およびデータ管理方針をサポートするライブラリ、ツールおよびインタフェースが含まれています。

Java 機能には次のものが含まれます。

- ネイティブな JDBC ドライバ。
- Java ストアド・プロシージャおよびトリガーのサポート。
- Java アクセス・クラス（JAC）。コア・データベースへのオブジェクト指向インタフェースを提供するクラス・ライブラリです。
- SQLJ サポート。

Oracle Lite は、Java コンポーネントの開発および管理用のツールも提供します。Oracle Lite のシームレスな永続 Java ポストプロセッサは、一時 Java クラスを Oracle Lite のステートフル永続クラスに変換します。Oracle Lite Designer は、Java クラスとオブジェクトを管理するビジュアル・インタフェースで、データベースのリレーショナル・ビューとオブジェクト指向ビューを提供します。`loadjava` コマンドライン・ユーティリティは、Java クラスを Oracle Lite データベースにロードする作業を自動化します。

Oracle Lite が提供する Java 機能の多くは、Oracle8*i* と互換性があります。この互換性により、企業のデータとアプリケーションを軽量のモバイル・プラットフォーム（PC、ラップトップ、携帯端末、埋込みデバイスなど）にすばやく簡単に配置できます。

ネイティブな JDBC サポート

JDBC（Java Database Connectivity）は、Java プログラムで SQL を使用するための業界標準の API です。JDBC は Microsoft の Open Database Connectivity（ODBC）を基に作成されています。JDBC 実装のほとんどでは、DBMS へのアクセスを提供するために、JDBC コールが JDBC-ODBC ブリッジを経由する必要があります。しかし、Oracle Lite では、ネイティブなタイプ 2 JDBC ドライバを提供しています。Oracle Lite が提供するネイティブなドライバにより、JDBC アプリケーションは Oracle Lite オブジェクト・カーネルと直接対話できます。

Oracle Lite JDBC ドライバは、JDBC 1.22 仕様をサポートします。さらに、JDBC 2.0 標準の特定の機能（ラージ・オブジェクト・データ型やデータ・アクセス関数など）もサポートします。

Oracle Lite の JDBC 実装は、SQL92 Intermediate Level に準拠し、次の拡張が含まれています。

- Java ストアド・プロシージャおよびトリガー
- Oracle データ型（VARCHAR2 や BLOB など）

- CONNECT BY および START WITH 句による再帰問合せ
- Oracle 組込みファンクション

Java ストアド・プロシージャとトリガー

Java ストアド・プロシージャおよびトリガーは、直接 Oracle Lite データベースに格納される Java プログラムです。データベースにアクセスするアプリケーションは、Java ベースでない場合でも、ストアド・プロシージャを実行できます。トリガーは、行の更新や削除などの特定のデータベース・イベントが発生したときに実行されるストアド・プロシージャです。ストアド・プロシージャとトリガーの使用により、データベースに関連するロジックは、データベースにアクセスする各アプリケーションに格納するのではなく、一箇所に集中して格納できます。

アプリケーションが Java ストアド・プロシージャをコールすると、Oracle Lite のオブジェクト・カーネルは最初に Java 仮想マシンのインスタンスを作成します（まだ作成されていない場合）。次に、オブジェクト・カーネルはメソッドを含むクラスをデータベースから取り出し、これを Java 仮想マシンにロードし、このメソッドを実行します。

Java アクセス・クラス（JAC）

Oracle Lite JAC は、Oracle Lite データベースへのオブジェクト指向インタフェースを提供する Java クラス・ライブラリです。JAC により、Java オブジェクトを直接 Oracle Lite に格納できます。JDBC と比較して、JAC を使用した場合はアプリケーション開発で次のことができます。

- すばやく容易な開発。Java オブジェクト・モデルとリレーショナル・モデル間のマッピングを管理する必要がありません。
- より高いパフォーマンス。JAC ではリレーショナル・モデルへのマッピングに必要なオーバーヘッドがないため、アプリケーションのパフォーマンスが向上します。

JAC ではステートレスまたはステートフルな JAC クラスを作成できます。ステートレス・クラスは、値を Oracle Lite データベースに直接格納します。ステートフル・クラスでは、標準の一時 Java クラスと同じように、値はインスタンス変数内にあります。Oracle Lite ポストプロセッサが、一時 Java クラスをシームレスなステートフル永続クラスに変換します。

注意： JAC アプリケーションは、Oracle Lite でのみ使用できます。Oracle8*i* では JAC をサポートしていません。

SQLJ

SQLJ は、静的 SQL 文を Java プログラムに埋め込むための標準です。JDBC と比較すると、SQLJ には次のような利点があります。

- SQLJ の方が、ソース・プログラムがより簡潔であり、サイズも小さい。
- SQL 構文チェック、データベース・スキーマ・チェック、タイプ・チェックにより、プログラマによるエラーの確立が低くなる。
- SQLJ 文の方が実行速度が速い。

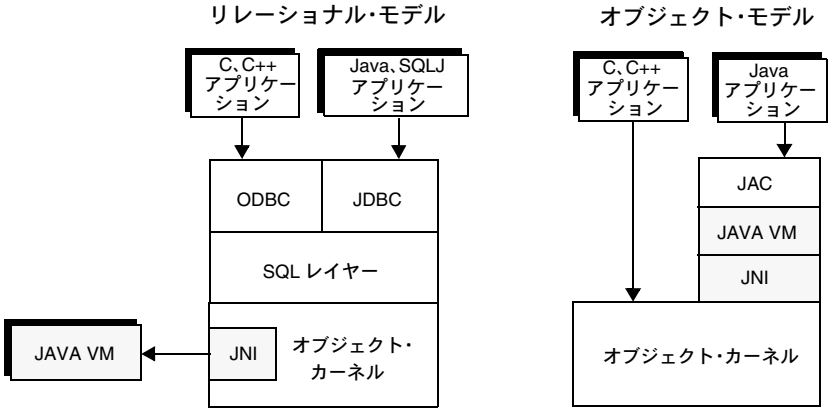
ほとんどの場合、SQLJ 文は JDBC 文と同じソース内にあります。SQLJ は静的 SQL 文に、JDBC は動的 SQL 文に対して使用します。

SQLJ を使用するには、Oracle SQLJ トランスレータ、ランタイム・コンポーネントおよび付属のドキュメンテーションを、次の Web サイトからダウンロードします。

http://www.oracle.com/java/sqlj/index.html

データベース・アクセス・モデル

次の図は、Oracle Lite がサポートするリレーショナル・プログラミング・インタフェースとオブジェクト・プログラミング・インタフェースを示しています。



リレーショナル・モデルを使用する必要があるのは、一般に、アプリケーションで複雑な問合せを使用する場合、すでに SQL 形式になっているデータにアクセスする場合、または他のリレーショナル・データベース管理システムとともに動作させる必要がある場合です。オ

ブジェクト・モデルは、アプリケーションで SQL 言語の全機能を必要とせず、軽量で高速実行が必要な場合に使用します。

この 2 つの設計モデルは互いに排他的ではありません。Java メソッドは、同一トランザクション内で JDBC コールと JAC コールを使用して Oracle Lite データをアクセスできます。ただし、アクセス・モードごとに、それぞれ個別の接続オブジェクトが必要です。JDBC コールには `Connection` オブジェクトを作成する必要があります。JAC コールには `POLConnection` オブジェクトを作成する必要があります。

注意： Oracle Lite は、JNI（Java Native Interface）をサポートする Java ランタイム環境であればどのランタイム環境でも使用できます。Oracle の JDeveloper、Sun の Java Development Kit、Symantec の Visual Cafe for Java および Borland の JBuilder は、すべてこのランタイム環境をサポートします。

Java データ型

Oracle Lite は、次の表に示すように Java と Oracle との間でデータ型を変換します。

Java データ型	SQL データ型
`byte[]`、`byte[][]`、`Byte[]`	BINARY、RAW、VARBINARY
`boolean`、`Boolean`	BIT
`String`、`String[]`	CHAR、VARCHAR、VARCHAR2
`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（精度の指定なし）
`BigDecimal`、`BigDecimal[]`	NUMBER(n)
`java.sql.Date`、`java.sql.Date[]`	DATE
`java.sql.Time`、`java.sql.Time[]`	TIME
`java.sql.Timestamp`、`java.sql.Timestamp[]`	TIMESTAMP
`java.sql.Connection`	データベースへのデフォルトの JDBC 接続

Java データ型	SQL データ型
`oracle.lite.jac.POLJACConnection`	データベースへのデフォルトの JAC 接続
`oracle.lite.jac.POLPersistantCapable`	オブジェクトの JAC インスタンス

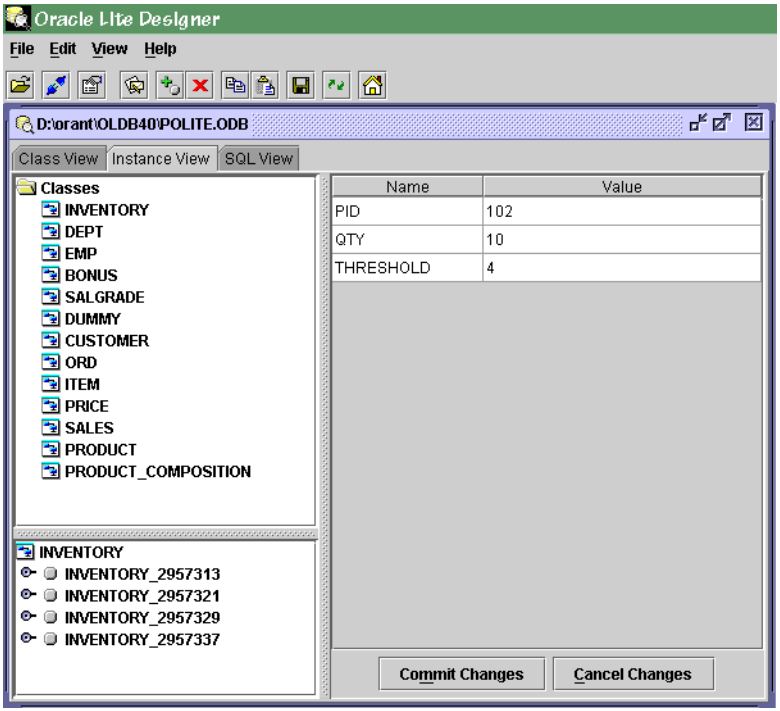
Java ツール

Oracle Lite は、Java 開発を管理する次のツールを提供します。

ツール	説明
Designer	Oracle Lite オブジェクト・クラスの階層とプロパティを管理します。Designer の実行可能ファイルは、**oldes.exe** です。
ポストプロセッサ	Java クラスを Oracle Lite データベース内で永続的にするための作業を自動化します。ポストプロセッサの使用方法は、第 6 章「シームレスな永続 Java」を参照してください。
`loadjava`	Java クラスを Oracle Lite にロードします。
`dropjava`	Java クラスを Oracle Lite から削除します。

Oracle Lite Designer

Oracle Lite Designer は、クラス、オブジェクトおよびデータベースを管理するビジュアル・インタフェースを提供します。



次のビューが提供されます。

ビュー	用途
クラス・ビュー	クラス定義の表示と操作を行います。
インスタンス・ビュー	オブジェクトの表示と操作を行います。
SQL ビュー	表としてのデータベースの表示と操作を行います。

インスタンス・ビューを使用すると、オブジェクトのブラウズ、作成、変更および削除ができます。インスタンス・ビューでは、左上のペインにはデータベースに定義されているクラスが表示され、左下のペインに各クラスのオブジェクトが表示され、右のペインには選択さ

れているオブジェクトの値が表示されます。これらの値を直接変更し、「Commit Changes」ボタンをクリックして変更を保存できます。

クラスのブラウズ、作成、変更および削除には、クラス・ビューを使用します。クラスを検査するには、「Class View」タブをクリックします。左下のペインに、選択されているクラスの属性とメソッドが表示されます。右のペインには、選択されている属性またはメソッドのプロパティが表示されます。

SQL ビューは、データベースのリレーショナル・ビューを示します。

loadjava

`loadjava` ユーティリティは、Java クラスとリソースを Oracle Lite データベースにロードする作業を自動化します。`loadjava` を使用すると、クラスおよびリソースを個別に、または ZIP あるいは JAR アーカイブとしてロードできます。

クラスをロードした後、SQL 文からアクセスするクラスのメソッドのコール仕様を作成します。値を返すストアド・プロシージャのコール仕様を作成するには、CREATE FUNCTION 文を使用します。ストアド・プロシージャが値を返す必要がない場合は、CREATE PROCEDURE 文を使用します。

クラスのアンロード用として、Oracle Lite では `dropjava` を提供していますが、これは `loadjava` と似たように機能します。

Oracle Lite が提供する Java 開発環境

Oracle Lite における Java 開発は、次のツールにより促進されます。

- Oracle Developer 2.1 は、Oracle Lite でのユーザー定義 Java ストアド・プロシージャをサポートしています。
- Oracle JDeveloper は、Oracle Lite での Java ストアド・プロシージャをサポートし、Java ストアド・プロシージャの開発と配置を支援するように設計された機能を装備しています。

環境の設定

この項では、Oracle Lite アプリケーションを作成する開発環境の設定方法を説明します。Java アプリケーションを開発するには、サン・マイクロシステムズ社の Java Development Kit（JDK）バージョン 1.1.8（以降）が必要です。

Oracle Lite のインストール後、Oracle Lite が JDK とともに稼働できるように、PATH および CLASSPATH 環境変数を設定します。PATH および CLASSPATH の設定は、次の項で説明されているように、使用する JDK のバージョンにより違います。

Oracle Lite をインストールする前に環境変数に CLASSPATH ユーザー変数が含まれており、このユーザー変数に CLASSPATH システム変数が含まれていない（`CLASSPATH=...;%CLASSPATH%` として指定されていない）場合、CLASSPATH ユーザー

変数を変更して *Oracle_Home*¥LITE¥CLASSES ディレクトリ内の **OLITE40.JAR** ファイルを組み込んでください。

注意： CLASSPATH に対する変更を反映するには、すべてのコマンド・プロンプト・ウィンドウを閉じてから再度オープンする必要があります。

JDK 1.1 の場合の変数の設定

JDK 1.1 を使用する場合は、次のように、PATH 変数内に JDK 実行可能ファイルを指定する必要があります。

```
set PATH=c:¥jdk1.1.8¥bin
```

Oracle Lite クラスを CLASSPATH に追加します。

```
set CLASSPATH=c:¥Oracle_Home¥lite¥classes¥oljdk11.jar
     c:¥Oracle_Home¥lite¥classes¥olite40.jar;c:¥JDK_Home¥lib¥classes.zip
```

Oracle Lite JDK 1.1 アーカイブ（**oljdk11.jar**）を CLASSPATH 内の最初の参照項目として指定する必要があります。

JDK 1.2 の場合の変数の設定

JDK 1.2 を使用する場合は、JDK 1.2 Java コンパイラ（**javac.exe**）が含まれたディレクトリを、他の Java コンパイラを含むディレクトリよりも前に PATH 変数内に指定する必要があります。

Classic Java 仮想マシン（JVM）の共有ライブラリ（**jvm.dll**）を含むディレクトリを PATH に追加します。**jvm.dll** は、*JDK_Home*¥jre¥bin¥classic ディレクトリにあります。

oljdk11.jar を CLASSPATH から削除します。これにより、Oracle Lite で使用されるシステム・クラスのバージョンが、JDK で使用されるバージョンと同じになります。例を次に示します。

```
set PATH=c:¥JDK_Home¥bin;c:¥JDK_Home¥jre¥bin¥classic
set CLASSPATH=c:¥JDK_Home¥jrc¥lib¥rt.jar;c:¥Oracle_Home¥lite¥classes¥olite40.jar
```

Classic JVM を使用するかわりに、HotSpot JVM を使用することもできます。HotSpot は、サン・マイクロシステムズ社により提供されている JDK アドオン・モジュールです。これは JDK 1.2 のパフォーマンスを強化しますが、Java クラスおよびオブジェクトの管理用のビジュアル・ツールである Oracle Lite Designer のパフォーマンスを改善します。HotSpot はサン・マイクロシステムズ社の Web サイトから入手できます。

HostSpot をインストールした後、次のように PATH を設定します。

```
set PATH=c:¥jdk1.2¥bin;c:¥jdk1.2¥jre¥bin¥hotspot;%PATH%
```

システムが正常に設定されたかどうかを調べるために、*Oracle_Home*¥Lite¥Examples¥Java ディレクトリにある Java のサンプルを実行します。

2

Java ストアド・プロシージャとトリガー

この章では、Oracle Lite リレーショナル・モデル内で Java ストアド・プロシージャとトリガーを使用する方法を説明します。説明する内容は、次のとおりです。

- Oracle Lite の新機能
- Java ストアド・プロシージャとトリガーの概要
- ストアド・プロシージャの使用方法
- Java データ型
- トリガーの使用方法

注意： Oracle Lite オブジェクト・モデルから Java ストアド・プロシージャとトリガーを使用する方法については、第 5 章「Java アクセス・クラス（JAC）」を参照してください。

また、SQL*Plus を使用してストアド・プロシージャを作成し実行する例については、付録 A「ストアド・プロシージャのチュートリアル」を参照してください。

Oracle Lite の新機能

この項では、今回のリリースで拡張された Oracle Lite の Java ストアド・プロシージャとトリガーのサポートについて説明します。

Oracle Lite では、Oracle8*i* のストアド・プロシージャ開発モデルをサポートするようになりました。このモデル（Load and Publish 開発モデル）では、クラスを表にアタッチするかわりに、`loadjava` コマンドライン・ユーティリティまたは CREATE JAVA コマンドを使用して、Java クラスを Oracle Lite データベースにロードします。クラスをデータベースにロードした後、コール仕様を使用して、SQL からコールするクラスのメソッドをパブリッシュします。コール仕様を作成するには、CREATE FUNCTION または CREATE PROCEDURE コマンドを使用します。詳細は、「モデル 1: Load and Publish ストアド・プロシージャ開発モデルの使用」を参照してください。

Oracle Lite では、ストアド・プロシージャに作成にあたり従来のモデルもサポートしています。従来のモデルでは、Java クラスを表にアタッチします。クラス内の静的メソッドがその表の表レベルのストアド・プロシージャになり、非静的（インスタンス）メソッドが、行レベルのストアド・プロシージャになります。

Oracle Lite には、今回、Java クラスをデータベースにロードする作業を自動化する `loadjava` ユーティリティが含まれています。`loadjava` を使用すると、Java クラス、ソースおよびリソースを個別にまたはアーカイブとしてロードできます。詳細は、「loadjava」を参照してください。

Java ストアド・プロシージャとトリガーの概要

Java ストアド・プロシージャは、Oracle Lite データベースに格納される Java メソッドです。このプロシージャは、データベースにアクセスするアプリケーションにより起動できます。トリガーは、行の更新、挿入または削除などの特定のイベントが発生したときに実行（起動）されるストアド・プロシージャです。列を更新した場合も、トリガーを起動できます。

トリガーは、文レベルまたは行レベルで操作できます。文レベルのトリガーは、複数の行が影響されても、トリガー文につき 1 回起動されます。行レベルのトリガーは、トリガー文により影響される行につき 1 回起動されます。Java ストアド・プロシージャは、1 つの値、1 行または複数行を返しますが、トリガーは値を返せません。

ストアド・プロシージャを作成するには、まず Oracle Lite に格納するクラスを作成します。Java IDE を使用してプロシージャを作成するか、または既存のプロシージャが要件に合うものであれば、それを再利用できます。

クラスを作成するときは、SQL DML 文からの Java ストアド・プロシージャのコールに関する次の制限を考慮してください。

- INSERT 文、UPDATE 文または DELETE 文からコールされたメソッドは、その文により変更されたデータベース表に対して問合せまたは変更できません。
- SELECT 文、INSERT 文、UPDATE 文または DELETE 文からコールされたメソッドは、COMMIT などの SQL トランザクション制御文を実行できません。また、CREATE など

の DDL 文には自動コミットが付随するため、ストアド・プロシージャでこのような DDL 文を実行できません。

ストアド・プロシージャ内に、以上の制限に違反する SQL 文があると、実行時に（その文が解析されるときに）エラーが生成されます

1 つの Oracle Lite アプリケーションには Java 仮想マシンが 1 つしかロードされないため、クラスには実行環境において一意の名前を付ける必要があります。アプリケーションが複数のデータベースからメソッドをコールする場合、それぞれのデータベースの Java クラスは同じ Java 仮想マシンにロードされます。Java クラス名にデータベース名を接頭辞として付けると、複数のデータベースにわたって Java クラス名を一意にできます。

Java ストアド・プロシージャが `java.sql.Connection` 型、`POLPersistentCapable` 型または `POLConnection` 型の引数を取る場合、Oracle Lite はカレント・トランザクションまたはカレント行から適切な値を最初の引数としてメソッドに渡します。このメソッドをコールするアプリケーション側でこのパラメータに値を指定する必要はありません。

ストアド・プロシージャの使用方法

Oracle Lite は、ストアド・プロシージャを作成する開発モデルをいくつかサポートします。Load and Publish モデルでは、Java クラスを Oracle Lite にロードし、次に、SQL からコールするクラスの静的メソッド用にコール仕様を作成します。このモデルは、Oracle8*i* でもサポートされています。このため、Oracle8*i* の企業向けアプリケーションの実装で培ったスキルとリソースを利用できます。

このモデルでは、次の手順を使用します。

1. 格納するメソッドを含む Java クラスを開発します。
2. `loadjava` ユーティリティまたは CREATE JAVA コマンドを使用して、クラスを Oracle Lite データベースにロードします。
3. SQL からアクセスするメソッドを、コール仕様を作成してパブリッシュします。メソッドのパブリッシュにより、SQL 名がメソッドに関連付けられます。SQL アプリケーションではこの名前を使用してメソッドをコールします。

Oracle Lite に格納してあるプロシージャをすべてパブリッシュする必要はありません。SQL からコールするプロシージャのみをパブリッシュします。他のストアド・プロシージャからコールされるだけのストアド・プロシージャも多数あり、これらをパブリッシュする必要はありません。このモデルを使用してストアド・プロシージャを開発する方法の詳細は、「モデル 1: Load and Publish ストアド・プロシージャ開発モデルの使用」を参照してください。Load and Publish モデルでサポートされているのは静的メソッドのみです。

次のモデルでは、クラスを表にアタッチし、クラス内のメソッドを名前によりコールします。これは、ストアド・プロシージャを開発する従来の Oracle Lite モデルです。このモデルを使用すると、クラス・レベルの（静的）メソッドとオブジェクト・レベルの（非静的）メソッドの両方を格納できます。

このモデルでは、次の手順に従います。

1. 格納するメソッドを含む Java クラスを開発します。
2. ALTER TABLE コマンドを使用して、クラスを表にアタッチします。

クラスをアタッチした後は、クラス内のメソッドを SQL から直接コールできます。メソッドは次の構文で指定します。

```
table_name.method_name
```

Java クラスを表にアタッチする方法は、「モデル 2: アタッチ済ストアド・プロシージャ開発モデルの使用」を参照してください。

Oracle Lite は、ストアド・プロシージャを削除するためのツールと SQL コマンドを提供しています。Oracle Lite ではクラス間の依存性を追跡しないため、データベースからプロシージャを削除するときは注意が必要です。削除するストアド・プロシージャが、他のストアド・プロシージャで参照されていないことを確認する必要があります。クラスを削除すると、直接的あるいは間接的にそのクラスに依存するクラスが無効になります。

モデル 1: Load and Publish ストアド・プロシージャ開発モデルの使用

この項では、Load and Publish 開発モデルを使用してストアド・プロシージャを作成する方法を説明します。ストアド・プロシージャを作成するには、まずクラスを作成します。クラスがエラーなしでコンパイルされ実行されることを確認します。次に、クラスを Oracle Lite データベースにロードします。最後に、SQL からコールするメソッドをパブリッシュします。Oracle Lite では、`main` メソッドにマップされるメソッドはパブリッシュできません。一方、Oracle8*i* では、`main` メソッドをパブリッシュするコール仕様を作成できます。

注意： Load and Publish 開発モデルでサポートされているのは Java 静的メソッドのみです。静的メソッドと非静的（インスタンス）メソッドの両方を格納するには、「モデル 2: アタッチ済ストアド・プロシージャ開発モデルの使用」で説明されているように、データベース表にクラスをアタッチする必要があります。

クラスのロード

Java クラスを Oracle Lite データベースにロードするには、次のいずれかを使用できます。

- `loadjava`
- CREATE JAVA コマンド

`loadjava` コマンドライン・ユーティリティは、Java クラスを Oracle Lite および Oracle8*i* データベースにロードする作業を自動化します。Java クラスを手動でロードするには、CREATE JAVA コマンドを使用します。

loadjava loadjava によりファイルからスキーマ・オブジェクトが作成され、そのオブジェクトがスキーマにロードされます。スキーマ・オブジェクトは、Java ソース、クラスおよびリソースから作成できます。リソースとしては、イメージ、SQLJ リソース、またはその他プロシージャがデータとしてアクセスする必要のあるものであれば、何でもかまいません。ファイルは、個別に、または ZIP や JAR アーカイブ・ファイルとして loadjava に渡すことができます。

Oracle Lite ではクラスの依存性を追跡しません。ストアド・プロシージャが必要とするクラスとリソースが、すべてデータベースにロードされているかまたは CLASSPATH に指定されていることを確認してください。

構文

loadjava の構文は次のとおりです。

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

引数

以下に、loadjava の引数について詳しく説明します。

user

user 引数には、ユーザー名、パスワードおよびデータベース接続文字列を次の書式で指定します。

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

例を次に示します。

```
scott/tiger@c:¥ORANT¥OLDB40¥Polite.odb
```

オプション

Oracle Lite では次のオプションをサポートします。

オプション	説明
-force | -f	ファイルのロードを施行します。すでにデータベースに同じ名前のスキーマ・オブジェクトが存在する場合でも施行されます。
-verbose | -v	実行中に詳細なステータス・メッセージを表示するように、loadjava に指示します。

複数のオプションを指定する場合は、オプション間をスペースで区切る必要があります。例を次に示します。

```
-force -verbose
```

Oracle8*i* は、『Oracle8*i* Java ストアド・プロシージャ開発者ガイド』で説明されているように、その他のオプションもサポートします。その他のオプションが Oracle Lite で使用された場合、認識はされますがサポートはされません。このようなオプションを使用しても、エラーにはなりません。

Oracle8*i* でサポートされているオプションを表示するには、次の構文で `loadjava` のヘルプ情報を参照してください。

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

filename

コマンドラインには、クラス、ソース、JAR、ZIP およびリソースを任意の順序でいくつでも指定できます。複数のファイル名を指定する場合は、カンマではなくスペースで区切る必要があります。ソースが渡されると、`loadjava` は Java コンパイラを起動して、ファイルを表にロードする前にコンパイルします。JAR ファイルまたは ZIP ファイルが渡されると、`loadjava` は JAR または ZIP 内の各ファイルを処理します。JAR または ZIP アーカイブの場合はスキーマ・オブジェクトは作成されません。`loadjava` は、JAR または ZIP アーカイブ内にある JAR または ZIP アーカイブは処理しません。

ファイルをロードする最善の方法は、JAR または ZIP 内にファイルを入れ、そのアーカイブをロードすることです。アーカイブをロードすることにより、リソースのスキーマ・オブジェクトの複雑な名前体系に煩わされません。JDK で有効な JAR または ZIP がある場合は、この JAR または ZIP を `loadjava` でロードできるため、リソースのスキーマ・オブジェクトの複雑な名前体系に煩わされなくてすみます。

`loadjava` では、データベースにファイルをロードするときに、ファイル用に作成するスキーマ・オブジェクトに名前を付ける必要があります。スキーマ・オブジェクトの名前は、ファイル名とは少し違います。また、異なるスキーマ・オブジェクトには異なるネーミング規則があります。クラスは名前で識別できるため、`loadjava` は自動的にファイル名をスキーマ名にマップできます。同様に、JAR および ZIP にはその中にあるファイルの名前が含まれています。

しかし、リソースは名前では識別できないため、コマンドラインに入力されるリテラル名（または JAR や ZIP アーカイブ内のリテラル名）から Java リソースのスキーマ・オブジェクト名が導出されます。クラスの実行中にリソースのスキーマ・オブジェクトが使用されるため、コマンドラインには正しいリソース名を指定することが重要です。

リソースを個別にロードする最善の方法は、`loadjava` をパッケージ・ツリーの最上位のディレクトリから実行し、そのディレクトリを基準にした相対的なリソース名を指定することです。リソースのロードをパッケージ・ツリーの最上位から行わない場合は、リソースの命名に関する次の情報を考慮してください。

リソースをロードするとき、`loadjava` は、コマンドラインに入力されたファイル名からリソースのスキーマ・オブジェクトの名前を導出します。次のような相対パス名と絶対パス名をコマンドラインに入力したとします。

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

同一ファイルを相対パス名と絶対パス名で指定しましたが、loadjava は 2 つのスキーマ・オブジェクトを作成します。

- alpha¥beta¥x.properties
- ¥scott¥javastuff¥alpha¥beta¥x.properties

loadjava は、入力されたファイル名からリソースのスキーマ・オブジェクトの名前を生成します。

注意： Oracle Lite では、スラッシュで始まるスキーマ・オブジェクト名を使用できます。Oracle8*i* では、スラッシュで始まるスキーマ・オブジェクトの前に「ROOT」が追加されます。

クラスからのリソースの参照は、相対的（たとえば、b.properties）または絶対的（たとえば、¥a¥b.properties）に指定できます。loadjava とクラス・ローダーでスキーマ・オブジェクトに必ず同じ名前が使用されるようにするには、クラスから java.lang.Object.getResource メソッドまたは java.lang.Class.getResourceAsStream メソッドに渡されるリソースの名前を loadjava に渡します。

コマンドラインに正しい名前を入力できるように、クラスがリソースを指定するために相対名を使用しているか絶対名を使用しているかを覚えておいてディレクトリを変更するという作業のかわりに、次のように、リソースを JAR ファイルにロードできます。

```
cd ¥scott¥javastuff
jar -cf alpharesources.jar alpha¥*.properties
loadjava options alpharesources.jar
```

または、もっと簡単にクラスとリソースの両方を JAR に入れると、次の起動方法はまったく同じことになります。

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

例

次の指定により、クラスとリソースが Oracle Lite データベースにロードされます。この指定では force オプションが使用されているため、指定された名前のオブジェクトがすでにデータベースにある場合は、loadjava により置き換えられます。

```
c:¥> loadjava -u scott/tiger@c:¥ORANT¥OLDB40¥Polite.odb -f Agent.class ¥
    images.dat
```

CREATE JAVA の使用方法

Java クラスを手動でロードするには、次の構文を使用します。

```
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>')
```

CREATE JAVA のパラメータは次のとおりです。

- OR REPLACE 句を指定すると、同じ名前のファンクションまたはプロシージャがデータベースにすでに存在する場合、再作成されます。
- Oracle8*i* と互換性を保つために、Oracle Lite では `<resolver_spec>` 句は認識されますが処理は行いません。Oracle8*i* と違って、Oracle Lite ではクラスの依存性は解決されません。クラスを手動でロードする場合は、必ず依存クラスをすべてロードするようにしてください。
- Oracle Lite では `<invoker_rights_clause>` が認識されますが処理は行いません。
- CREATE JAVA SOURCE 文は Oracle8*i* ではサポートされていますが Oracle Lite ではサポートされません。

Java 文を埋め込む際は、SQL*Plus 文の最後にセミコロン文字（;）を使用しないでください。行の最後の文字をセミコロンにする必要がある場合は、`/**/` の文字を使用して、空のコメント行を追加する必要があります。このコンテキストでは、通常のコメント記号（//）が機能しません。行の最後に `/**/` を置くことで、SQL*Plus によりセミコロンが SQL 文の末尾として解釈されることを防ぎます。

例

次に CREATE JAVA 文の例を示します。この例では、`Employee` というクラスがデータベースにロードされます。

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

SQL へのストアド・プロシージャのパブリッシュ

`loadjava` または CREATE JAVA を使用してクラスを Oracle Lite データベースにロードした後は、クラス内の静的メソッドで SQL からコールするメソッドをパブリッシュします。メソッドをパブリッシュするには、そのためのコール仕様を作成します。コール仕様は、Java メソッドの名前、パラメータ・タイプおよび戻り型を、対応する SQL 要素にマップします。

ストアド・プロシージャをすべてパブリッシュする必要はありません。アプリケーションのエントリ・ポイントとして機能するもののみをパブリッシュします。通常の実装では、スト

アド・プロシージャのほとんどは SQL ユーザーではなく他のストアド・プロシージャからのみコールされます。

コール仕様を作成するには、CREATE FUNCTION または CREATE PROCEDURE を使用します。値を返すメソッドには CREATE FUNCTION を、値を返さないメソッドには CREATE PROCEDURE を使用します。CREATE FUNCTION 文と CREATE PROCEDURE 文の構文は次のとおりです。

```
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>]';
   /
```

この文のキーワードとパラメータは次のとおりです。

- `<sql_parms>` は次の書式です。

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

- `<java_parms>` は、Java データ型の完全修飾名です。
- Oracle8*i* との互換性を保つために、Oracle Lite では `<invoker_rights_clause>` 句は認識されますが処理は行いません。
- `<java_fullname>`は、静的 Java メソッドの完全修飾名です。

注意： SQL*Plus では、CREATE PROCEDURE 文と CREATE FUNCTION 文の後ろにスラッシュ（/）を指定する必要があります。

たとえば、次のクラスがデータベースにロードされているとします。

```
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
```

クラス GenericDrop には、dropIt という名前のメソッドが 1 つあり、このメソッドはどのような種類のスキーマ・オブジェクトでも削除します。たとえば、引数に「table」と「emp」を dropIt に渡すと、データベース表 EMP をスキーマから削除します。

次のコール仕様により、このメソッドが SQL にパブリッシュされます。

```
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)';
   /
```

Java データ型パラメータには完全修飾名を指定する必要があるので、注意してください。

パブリッシュされたストアド・プロシージャのコール

ストアド・プロシージャを SQL にパブリッシュした後、SQL DML 文を使用してこのプロシージャをコールします。たとえば、次のクラスがデータベース内に格納されているとします。

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

クラス Formatter には formatEmp というメソッドが 1 つあり、このメソッドは、従業員の名前と職務状況を含む書式化された文字列を返します。Formatter 用のコール仕様を次のように作成します。

```
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';
/
```

このコール仕様により、メソッド formatEmp が format_emp としてパブリッシュされます。これを次のようにコールします。

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

この文により、次の出力が生成されます。

```
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
```

注意： Oracle Lite では、Oracle8*i* の CALL 文を使用してストアド・プロシージャを起動できません。

C および C++ アプリケーションからストアド・プロシージャをコールする方法は、「Java ストアド・プロシージャの ODBC からのコール」を参照してください。

パブリッシュされたストアド・プロシージャの削除

Oracle Lite からクラスを削除するには、次のいずれかを使用します。

- dropjava ユーティリティ
- DROP JAVA 文

コール仕様を削除するには、DROP FUNCTION または DROP PROCEDURE を使用します。

dropjava の使用 dropjava は、Oracle Lite および Oracle8*i* データベースから Java クラスを削除する作業を自動化するコマンドライン・ユーティリティです。dropjava は、ファイル

名をスキーマ・オブジェクト名に変換し、そのスキーマ・オブジェクトを削除します。dropjava の起動には次の構文を使用します。

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

引数

以下に、dropjava の引数について説明します。

user

user 引数には、ユーザー名、パスワードおよびデータベース接続文字列を次の書式で指定します。

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

例を次に示します。

scott/tiger@c:¥ORANT¥OLDB40¥Polite.odb

オプション

verbose オプション（-verbose | -v）を指定すると、実行中に詳しいステータス・メッセージを生成するように dropjava に指示できます。

Oracle8*i* は、『Oracle8*i* Java ストアド・プロシージャ開発者ガイド』で説明されているように、dropjava のその他のオプションもサポートします。その他のオプションが Oracle Lite で使用された場合、認識はされますがサポートはされません。このようなオプションを使用しても、エラーにはなりません。

サポートされるオプションと認識されるオプションの一覧を表示するには、コマンド・プロンプトで次のように入力します。

```
dropjava -help
```

filename

filename 引数には、Java クラス、ソース、JAR、ZIP およびリソースを任意の順序でいくつでも指定できます。JAR および ZIP ファイルは圧縮を解凍する必要があります。dropjava では、ほとんどのファイル名を loadjava と同じ方法で解釈します。

- クラスを指定した場合、ファイル内でクラス名を検索し、それに対応するスキーマ・オブジェクトを削除します。
- ソースを指定した場合、ファイル内の最初のクラス名を検索し、それに対応するスキーマ・オブジェクトを削除します。
- JAR および ZIP ファイルを指定した場合、アーカイブされているファイルの名前がコマンドラインで入力されたかのように、ファイル名を処理します。

ファイル名に **.java**、**.class**、**.jar** または **.zip** 以外の拡張子がある場合、または拡張子がない場合は、ファイル名がスキーマ・オブジェクト名であると解釈され、その名前のソース、クラス、またはリソースのスキーマ・オブジェクトがすべて削除されます。スキーマ・オブ

ジェクトのどの名前とも一致しないファイル名が検出されると、エラー・メッセージが表示され、残りのファイル名が処理されます。

SQL コマンドの使用方法 Oracle Lite データベースから Java クラスを手動で削除するには、DROP JAVA 文を使用します。DROP JAVA の構文は次のとおりです。

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

コール仕様を削除するには、DROP FUNCTION 文または DROP PROCEDURE 文を使用します。

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

スキーマ名が指定された場合、Oracle Lite では認識されますがサポートはされません。

例

次の例では、Load and Publish モデルを使用して Java ストアド・プロシージャを作成します。

この例では、Java メソッド `paySalary` を Oracle Lite データベースに格納します。`paySalary` は、従業員の手取り給与を計算します。Oracle Lite はカレントの Connection オブジェクトを `paySalary` に自動的に渡します。

この例には次の手順が含まれています。

- ステップ 1: Java クラスを作成する
- ステップ 2: データベースへの Java クラスのロード
- ステップ 3: ファンクションのパブリッシュ
- ステップ 4: ファンクションの実行

Java ストアド・プロシージャの例は、他にも *Oracle_Home*¥Lite¥Examples¥Java ディレクトリにあります。

ステップ 1: Java クラスを作成する

Java クラス `Employee` をファイル **Employee.java** 内に作成します。`Employee` クラスは `paySalary` メソッドを実装します。

```
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
}
```

注意： 最初の PUBLIC クラス文の前に、コメント内でキーワードの「public class」を使用しないでください。

ステップ 2: データベースへの Java クラスのロード

SQL*Plus から、次のように CREATE JAVA を使用して Java クラスをロードします。

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

このコマンドは、c:¥myprojects¥doc にある Java クラスを Oracle Lite データベースにロードします。

ステップ 3: ファンクションのパブリッシュ

paySalary メソッド用のコール仕様を作成します。次のコール仕様は、Java メソッド paySalary をファンクション pay_salary としてパブリッシュします。

```
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';
/
```

ステップ 4: ファンクションの実行

SQL*Plus 内で pay_salary を実行するには次のようにします。

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

ODBC 内で pay_salary を実行するには次のようにします。

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

pay_salary への引数が定数のため、FROM 句ではダミー表 DUAL を指定します。この SELECT 文により、次の出力が生成されます。

```
Net salary is 2502.6
```

モデル 2: アタッチ済ストアド・プロシージャ開発モデルの使用

この項では、クラスを表にアタッチしてストアド・プロシージャを作成する方法を説明します。この方法は Oracle Lite 固有のものです。ここで説明されている方法で Oracle8*i* の表にクラスをアタッチすることはできません。「モデル 1: Load and Publish ストアド・プロシージャ開発モデルの使用」に説明されているストアド・プロシージャ開発用の Load and Publish モデルがサポートするのは、クラスの（静的）メソッドのみですが、クラスを表にアタッチすると、Java クラスとインスタンスのメソッドを格納しコールできます。

アタッチ済ストアド・プロシージャを作成するには、アタッチするクラスを開発します。クラスがエラーなしでコンパイルされ実行されることを確認します。次に、クラスを Oracle Lite の表にアタッチします。クラスがアタッチされると、クラス内のメソッドがその表の表レベルおよび行レベルのストアド・プロシージャになります。

表レベルのストアド・プロシージャ

表レベルのストアド・プロシージャは、Java クラスの静的メソッドです。このため、メソッドの実行時には、メソッドが属するクラスはインスタンスを生成しません。コール文の中では、表レベルのストアド・プロシージャは *table_name.method_name* として参照されます。

文レベルのトリガーと BEFORE INSERT および AFTER DELETE 行レベル・トリガー（「文レベルのトリガーと行レベルのトリガーの比較」を参照）は、表レベルのストアド・プロシージャであることが必要です。

行レベルのストアド・プロシージャ

行レベルのストアド・プロシージャは、Java クラスの非静的メソッドです。行レベルのストアド・プロシージャを実行するために、そのプロシージャが属するクラスのインスタンスが生成されます。クラス・コンストラクタへの引数によって、コンストラクタがクラス・インスタンスの作成にパラメータとして使用する列値が決定されます。コール文の中では、行レベルのストアド・プロシージャは *method_name*（表修飾子なし）として参照されます。行レベル・トリガーは、行レベル・ストアド・プロシージャを間接的に実行できます。

表への Java クラスのアタッチ

表に Java クラスをアタッチするには、ALTER TABLE を使用します。ALTER TABLE コマンドの構文は次のとおりです。

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

アタッチできるのは、ソースまたはクラスです。ソースは、システム・パスにある Java コンパイラによりコンパイルされます。

`cls_or_src_name` は、クラスまたはソースの完全修飾名を示します。これには、クラス名が後に続くパッケージ名（`Oracle.lite.Customer` など）が含まれます。クラス名やソー

ス名にはファイル拡張子を指定してはいけません。名前の大 / 小文字は区別されます。小文字を使用する場合は、名前を二重引用符（""）で囲みます。cls_or_src_name で指定されるパッケージ内にソースまたはクラスがあることを確認します（サンプル・クラス Customer のソースには、行「package Oracle.lite;」が含まれています）。クラスは、データベースの同一パッケージ内に格納されます。パッケージが存在していない場合は作成されます。

Java クラスをデータベースの別の表にすでにアタッチしてある場合は、IN DATABASE 句を使用できます。クラスがまだアタッチされていない場合は、クラスまたはソースのディレクトリ位置を cls_or_src_path に指定します。

行レベルのストアド・プロシージャを実行する前に、その行に Java オブジェクトが作成されます（Java オブジェクトが存在しない場合）。ALTER TABLE 文に WITH CONSTRUCTOR 句が含まれている場合は、col_name_list に含まれている列のデータ型に最適のクラス・コントラクタを使用して、このオブジェクトが作成されます。ALTER TABLE 文に WITH CONSTRUCTOR 句が含まれていない場合は、デフォルトのコンストラクタが使用されます。

表の中に定義されているメソッドに関する情報を取り出すには、ODBC 関数の SQLProcedures と SQLProcedureColumns を使用できます。

ALTER TABLE 文については、次のことに注意してください。

- ALTER TABLE 文に引数を指定した場合、クラスのインスタンスは、引数リストで渡される値を使用して作成されます。その後で Java アプリケーションがこれらのパラメータの値を変更しても、新しい値はデータベースに自動的にはポストされません。新しい値でデータベースを更新するには、アプリケーションで JDBC コールまたは JAC コールを使用する必要があります。
- ALTER TABLE 文でコンストラクタが指定されていない場合に行レベル・トリガーが起動されると、デフォルト・コンストラクタを使用してオブジェクトが作成されます。

アタッチ済ストアド・プロシージャのコール

ALTER TABLE 文を使用してプロシージャを表にアタッチした後は、SELECT 文を使用してこのプロシージャをコールできます。表レベルのストアド・プロシージャは *table_name.method_name* として参照し、行レベルのプロシージャは *method_name* として参照します。

たとえば、表レベルのストアド・プロシージャを実行するには、次のように指定します。

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

proc_name は、表レベルのストアド・プロシージャの名前です。arg_list の各引数は、定数かまたは表内の列への参照です。arg_list のすべての引数が定数の場合は、FROM 句でダミー表 DUAL を参照する必要があります。

行レベルのストアド・プロシージャは、次のように実行します。

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

プロシージャを *table_name.method_name* の書式でコールし、その名前の表またはメソッドが存在しない場合、Oracle Lite では、*table_name* 名がスキーマ名を意味し、*method_name* がプロシージャ名を意味すると解釈されます。*method_name* のみを参照する場合は、参照されたメソッドが行レベル・プロシージャであると解釈されます。ただし、そのようなプロシージャが定義されていない場合、Oracle Lite では *method_name* が現在のスキーマ内のプロシージャを意味すると解釈されます。

注意： Oracle Lite では、Oracle8*i* の CALL 文を使用してストアド・プロシージャを起動できません。

Callable Statement を使用して、ODBC または JDBC アプリケーションからプロシージャを実行できます。詳細は、第 3 章「Java Database Connectivity（JDBC）」を参照してください。また、「Java ストアド・プロシージャの ODBC からのコール」も参照してください。

アタッチ済ストアド・プロシージャの削除

ストアド・プロシージャを削除するには、ALTER TABLE コマンドを使用します。ALTER TABLE コマンドの構文は次のとおりです。

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

注意： クラス名に小文字が含まれている場合は、クラス名を二重引用符（" "）で囲む必要があります。

Java クラスをデタッチしてもデータベースからは削除されません。データベースから Java クラスを削除するには、DETACH AND DELETE 文を使用します。

Java クラスをストアド・プロシージャまたはトリガーとして起動した後に、その Java クラスをデータベースから削除した場合、そのクラスはアプリケーションにアタッチされている Java 仮想マシンの中に残ります。クラスを Java 仮想マシンからアンロードするには、（必要なら）データベースの変更をコミットし、データベースに接続しているアプリケーションをすべてクローズします。Java クラスを置き換えるには、データベースへの接続をすべてクローズしてから Java クラスをリロードする必要があります。

例

次の例は、Oracle Lite に Java ストアド・プロシージャを作成する方法を示します。この例では、Java メソッド paySalary を表 EMP にアタッチします。paySalary は、従業員の手取り給与を計算します。Oracle Lite はカレントの Connection オブジェクトを最初の引数として paySalary に渡します。

この例には次の手順が含まれています。

- ステップ 1: 表を作成する
- ステップ 2: Java クラスを作成する
- ステップ 3: Java クラスを表へアタッチする
- ステップ 4: メソッドを実行する

ステップ 1: 表を作成する

次の SQL コマンドを使用して表を作成します。

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

ステップ 2: Java クラスを作成する

Java クラス Employee をファイル **Employee.java** 内に作成します。Employee クラスは paySalary メソッドを実装します。

```
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
}
```

ステップ 3: Java クラスを表へアタッチする

SQL*Plus から、次のように ALTER TABLE コマンドを使用して Java クラスをアタッチします。

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

このコマンドは、ディレクトリ C:¥tmp にある Employee クラスの Java ソースを EMP 表にアタッチします。

ステップ 4: メソッドを実行する

paySalary メソッドを SQL*Plus で実行するには、次の文を入力します。

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

ODBC から paySalary を実行するには、SQLExecDirect を起動します。

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

この文により、次の結果が生成されます。

```
Net salary is 2502.6
```

Java ストアド・プロシージャの ODBC からのコール

Java ストアド・プロシージャを起動するには、ODBC 関数 SQLExecute と SQLExecuteDirect を使用できます。SQLExecute は、SQLPrepare を使用して作成された SQL 文へのハンドルを引数として取ります。SQLExecute と SQLPrepare の使用例は、「ODBC を使用して複数行を返す方法」を参照してください。

C または C++ のマルチスレッド・アプリケーションから Java ストアド・プロシージャを起動するには、アプリケーションの main 関数から **loadjvm40.dll** をロードする必要があります。これにより、C または C++ アプリケーションがストアド・プロシージャを直接または間接的に起動する複数のスレッドを作成するときに、Java 仮想マシンのガベージ・コレクションで発生する問題が解決されます。スレッドは、終了するまで Java 仮想マシンからデタッチされないため、Java 仮想マシンのメモリーが足りなくなります。Oracle Lite では、そのスレッドを作成したのが Java 仮想マシンかユーザー・アプリケーションかを判断できないため、スレッドのデタッチは行われません。

main では、次のように、すべてのアクションの前にライブラリをロードする必要があります。

```
int main (int argc, char** argv)
{
   LoadLibrary("loadjvm40.dll");
   ...
}
```

このライブラリは、Java 仮想マシンをアプリケーションのメイン・スレッドにロードします。このライブラリは、スレッドがプロセスからデタッチされた場合は、スレッドを Java 仮想マシンからデタッチしようとします。**loadjvm40.dll** は、スレッドが Java 仮想マシンにアタッチされていない場合でも正しく動作します。

Java データ型

Oracle Lite は、SQL 標準規則に基づいて Java と SQL の間でデータ型の変換を実行します。たとえば、文字列を引数として取るストアド・プロシージャに整数を渡すと、整数は文字列に変換されます。行レベル・トリガーの引数の詳細は、「トリガーの引数」を参照してください。Java と SQL 間のデータ型の対応付け一覧は、第 1 章「Oracle Lite の Java サポート」の「Java データ型」を参照してください。

注意： Oracle8*i* では、DATE 列は TIMESTAMP として作成されます。トリガー・メソッドもそのように指定する必要があります。

Java では、メソッドの範囲外にある引数の値をメソッドで変更できません。ただし、Oracle Lite では、IN、OUT および IN/OUT パラメータをサポートします。

Java データ型の多くは不変であるか、または NULL 値をサポートしません。このようなデータ型に NULL 値を渡して IN/OUT パラメータを使用する場合は、ストアド・プロシージャでその型の配列を使用するか、または NULL をサポートするかわりのデータ型を使用します。次の表は、Java の整数データ型に IN/OUT パラメータを使用できるか、または NULL 値を設定できるかを示しています。

Java の引数	IN/OUT	NULL
`int`	いいえ	いいえ
`int[]`	はい	いいえ
`Integer`	いいえ	はい
`Integer[]`	はい	はい
`int[][]`	はい	はい

`Date` などの可変 Java データ型を使用すると、NULL または IN/OUT パラメータを渡すことができますが、ストアド・プロシージャから引数の NULL 状態を変更する必要がある場合は、`Date` 配列を使用してください。

注意： NULL ではあり得ない Java 引数に NULL を渡すと、エラーが発生します。

パラメータの宣言

Java メソッドの戻り値は、プロシージャの OUT パラメータに設定できます。プリミティブ型または不変参照型は、IN パラメータに設定できます。可変参照型または配列型は、IN/OUT パラメータに設定できます。次の表は、対応する Oracle Lite パラメータを IN/OUT パラメータにするために使用する Java データ型を示します。

IN/OUT パラメータの型	使用するデータ型
NUMBER	Number[]
Binary	byte[] または byte[][]
String	string[]

ストアド・プロシージャが java.sql.Connection、POLPersistentCapable または POLConnection 引数を取る場合、Oracle Lite はカレント・トランザクションまたはカレント行の値を使用して、引数を自動的に指定します。この引数は、プロシージャに渡される最初の引数です。

ストアド・プロシージャを使用して複数行を返す方法

ストアド・プロシージャを使用して複数の行を返すことができます。ただし、複数行を返すストアド・プロシージャを起動できるのは、JDBC アプリケーションまたは ODBC アプリケーションからのみです。ストアド・プロシージャで複数行を返すには、対応する Java メソッドが java.sql.ResultSet オブジェクトを返す必要があります。SELECT 文の実行により、Java メソッドが返される ResultSet を取得します。ResultSet の列名は SELECT 文に指定します。結果列を、表で使用されている名前ではなく別の名前で呼ぶ必要がある場合は、SELECT 文で結果列の別名を使用する必要があります。例を次に示します。

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

複数行を返すストアド・プロシージャの戻り型は java.sql.ResultSet でなければならないため、結果の列名または型の取得に、そのストアド・プロシージャのシグネチャは使用できません。この結果、ストアド・プロシージャの結果の列名または型を追跡する別の表を設計する必要があります。たとえば、前述の SELECT 文を Java メソッドに埋め込んだ場合、このメソッドの戻り型は char Name や char Dept ではなく、java.sql.ResultType です。

注意： 複数行を返す Java ストアド・プロシージャの作成に使用できるのは、「モデル 2: アタッチ済ストアド・プロシージャ開発モデルの使用」で説明されているアタッチ済ストアド・プロシージャ開発モデルのみです。

ODBC を使用して複数行を返す方法

複数行を返すストアド・プロシージャを ODBC アプリケーションで実行するには、次の CALL 文を使用します。P はストアド・プロシージャの名前で、a_1 から a_n はストアド・プロシージャへの引数です。

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

文を実行する前に値をバインドする必要がある引数には、マーカー（?）を使用します。文が実行されるとプロシージャが実行され、結果セット上のカーソルが文ハンドル内に格納されます。文ハンドルを使用して順次フェッチを行い、プロシージャから行を返します。

CALL 文を実行した後は、SQLNumResultCols を使用してそれぞれの結果行の列数を見つけ出します。列の名前とデータ型を返すには、SQLDescribeCol 関数を使用します。

例

次の例は、ODBC を使用して、複数行を返すストアド・プロシージャを実行する方法を示します。この例では、SQLNumResultCols 関数も SQLDescribeCol 関数も使用されていません。ここでは、ストアド・プロシージャが作成済で、PROC として SQL にパブリッシュされていると想定されています。PROC は引数として整数を取ります。

```
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);
}
```

トリガーの使用方法

トリガーとは、特定のイベントが発生したときに実行（起動）されるストアド・プロシージャのことです。トリガーは、列が更新されたとき、または行が追加あるいは削除されたときに起動されます。トリガーはイベントの前または後で起動できます。

トリガーは、一般的には、データベースのビジネス・ルールを施行するために使用します。たとえば、トリガーは入力値を検証し、無効な挿入を拒否できます。同様に、トリガーを使用すると、特定の行を削除する前に、その行に依存するすべての表を一貫性のある状態にすることができます。

文レベルのトリガーと行レベルのトリガーの比較

トリガーには、行レベルと文レベルの 2 種類があります。行レベル・トリガーは、データベースへの変更により影響される各行に対して 1 回起動されます。文レベル・トリガーは、変更により複数の行が影響される場合でも、1 回のみ起動されます。

行オブジェクトのインスタンスを生成してプロシージャをコールできないため、BEFORE INSERT トリガーと AFTER DELETE トリガーが起動できるのは、表レベルのストアド・プロシージャのみです。AFTER INSERT、BEFORE DELETE および UPDATE の各トリガーでは、表レベルまたは行レベルのストアド・プロシージャを起動できます。

トリガーの作成

トリガーを作成するには、CREATE TRIGGER コマンドを使用します。CREATE TRIGGER 文の構文は次のとおりです。

```
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)
```

CREATE TRIGGER の構文は、次のとおりです。

- OR 句は、複数のトリガー・イベントの指定に使用します。
- FOR EACH ROW は、行レベル・トリガーの作成に使用します。表レベル・トリガーの場合は、この句は指定しません。
- `procedure_ref` は、実行するストアド・プロシージャの識別に使用します。

1 つの表に対して、同じ種類のトリガーを複数作成できます。ただし、各トリガーにはスキーマ内で一意の名前を指定する必要があります。

次の例では、プロシージャを PROCESS_NEW_HIRE として格納しパブリッシュしてあるものとします。トリガー AIEMP は、EMP 表に行が 1 つ挿入されるたびに起動されます。

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

表の複数の列に対して同じストアド・プロシージャを使用する UPDATE トリガーは、文の中で列のサブセットが変更されたときに 1 回のみ起動されます。たとえば、次の文は表 T に BEFORE UPDATE トリガーを作成します。表 T には、C1、C2、C3 という列があります。

```
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);
```

この UPDATE 文は、T_TRIGGER を 1 回のみ起動します。

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

トリガーの使用可能と使用禁止

トリガーは、作成すると自動的に使用可能になります。トリガーを使用禁止にするには、ALTER TABLE 文か ALTER TRIGGER 文を使用します。

トリガーを個別に使用可能または使用禁止にするには、ALTER TRIGGER 文を使用します。ALTER TRIGGER 文の構文は次のとおりです。

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

表にアタッチされているすべてのトリガーを使用可能または使用禁止にするには、ALTER TABLE 文を使用します。

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

トリガーの削除

トリガーを削除するには、DROP TRIGGER 文を使用します。DROP TRIGGER 文の構文は次のとおりです。

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

トリガーの例

次の例ではトリガーを作成します。「モデル 2: アタッチ済ストアド・プロシージャ開発モデルの使用」で説明されている開発モデルに従って作成します。Load and Publish モデルを使用したトリガー作成の例は、「」を参照してください。この例では、まず、表と Java クラスを作成します。次に、クラスを表にアタッチします。最後に、トリガーを作成して起動します。

SalaryTrigger クラスには check_sal_raise メソッドが含まれています。このメソッドは、従業員が 10% を超える昇給を受ける場合はメッセージを出力します。EMP 表内の給与を更新する前に、トリガーがメソッドをコールします。

check_sal_raise はメッセージを標準出力に書き込むため、この例にある SQL コマンドは DOS 版の SQL*Plus を使用して発行します。DOS コマンド・プロンプトで、次のように入力して SQL*Plus を起動します。

```
plus80 username/password@connect_string
```

connect_string は ODBC:data_source_name です。たとえば、デフォルト・データベースにユーザー SYSTEM として接続するには、DOS プロンプトで次のように入力します。

```
plus80 system/pw@odbc:polite
```

SQL*Plus コマンドラインで、次のように EMP 表を作成して値を挿入します。

```
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);
```

次のクラスを **SalaryTrigger.java** に入れます。

```
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);
      }
   }
}
```

SalaryTrigger クラス・コンストラクタは整数を引数にとり、この整数を属性 eno（従業員番号）に割り当てます。表 EMP の各行（つまり各従業員ごと）に SalaryTrigger が作成されます。

check_sal_raise メソッドは非静的メソッドです。実行するには、そのクラスのオブジェクトがこのメソッドをコールする必要があります。EMP の行の salary 列が更新されるたびに、その行に対応する SalaryTrigger コンストラクタへの引数として従業員番号（*E#*）

を使用して、このコンストラクタのインスタンスが（存在しない場合は）作成されます。次に、トリガーが check_sal_raise メソッドをコールします。

Java クラスを作成した後は、次のように、クラスを表にアタッチします。

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

この文は、Oracle Lite に対して、現在のディレクトリにある Java ソース **SalaryTrigger.java** をコンパイルし、結果のクラスを EMP 表にアタッチするように指示します。また、この文は、Oracle Lite がクラスのインスタンスを生成するときに、E# 列にある値を引数として取るコンストラクタを使用するように指定します。

クラスを表にアタッチした後、次のように、トリガーを作成します。

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

この文は、check_raise というトリガーを作成します。このトリガーは、EMP 表のいずれかの行の salary 列が更新される前に、check_sal_raise メソッドをコールします。Oracle Lite は、salary 列の古い値と新しい値をメソッドの引数として渡します。

この例では、行レベル・トリガーが行レベル・プロシージャ（非静的メソッド）を起動します。行レベル・トリガーは、表レベル・プロシージャ（静的メソッド）を起動することもできます。ただし、文レベル・トリガーは文全体に対して 1 回起動され、文が複数の行に影響する可能性があるため、文レベル・トリガーは表レベル・プロシージャを 1 回しか起動できません。

次のコマンドは、salary 列を更新しトリガーを起動します。

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

これにより、次の出力が生成されます。

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

トリガーの引数

「モデル 2: アタッチ済ストアド・プロシージャ開発モデルの使用」で説明されているように、アタッチ済ストアド・プロシージャを使用する場合は、行レベル・トリガーは Java から SQL への型変換をサポートしません。このため、トリガー引数の Java データ型は、トリガー列の対応する SQL データ型（「Java データ型」を参照）に一致している必要があります。ただし、Load and Publish モデルを使用する場合は、Oracle Lite がデータ型のキャストをサポートします。

次の表に、それぞれの列のタイプで有効なトリガー引数を示します。

トリガーの引数	新規列のアクセス	古い列のアクセス
insert	はい	いいえ
delete	いいえ	はい
update	はい	はい

注意： `java.sql.Connection` オブジェクトを引数として持つトリガーは、リレーショナル・モデルを使用するアプリケーションでのみ使用できます。

次の例は、IN/OUT パラメータを使用するトリガーの作成方法を示します。

1. まず、Java クラス `EMPTrigg` を作成します。

```
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. 新規の表 EMPLOYEE を作成し、これに値を挿入します。

```
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. 次に、クラスを Oracle Lite にロードします。

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

4. CREATE PROCEDURE 文を使用して、コールする EMPTrigg メソッドをパブリッシュします。

```
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. ここで、各プロシージャにトリガーを作成します。

```
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. 次のコマンドを入力し、トリガーを起動して結果を表示します。

```
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;
```

3

Java Database Connectivity（JDBC）

この章では、Oracle Lite の Java Database Connectivity（JDBC）に対するサポートを説明します。説明する内容は、次のとおりです。

- Oracle Lite の新機能
- JDBC
- JDBC 環境の設定
- JDBC からの Java ストアド・プロシージャの実行
- Oracle Lite での拡張
- JDBC と Java アクセス・クラス間の相互運用性
- 制限事項

注意： Oracle Lite JDBC ドライバの使用例は、付録 B「JDBC アプリケーションのチュートリアル」を参照してください。

Oracle Lite の新機能

この項では、Oracle Lite の今回のリリースで強化された JDBC サポートについて説明します。

JDBC ドライバで、BLOB および CLOB データ型を含むラージ・オブジェクトをサポートするようになりました。このサポートは、JDBC 2.0 仕様の機能に準拠し、Oracle8*i* の JDBC 実装と互換性があります。

Oracle Lite には、ラージ・オブジェクトにアクセスするための新機能も含まれています。詳細は、「Oracle Lite での拡張」を参照してください。

JDBC

JDBC とは、Java プログラムからリレーショナル・データベースにアクセスするためのアプリケーション・プログラミング・インタフェースです。Oracle Lite では、Java アプリケーションが Oracle Lite のオブジェクト・リレーショナル・データベース・エンジンと直接通信できるネイティブな JDBC ドライバを提供しています。Oracle Lite の JDBC 実装は、JDBC 1.22 に準拠しています。さらに、Oracle Lite では JDBC 2.0 で指定されている特定の拡張機能を提供しています。Oracle Lite の拡張は、Oracle8*i* の JDBC 実装と互換性があります。JDBC の詳細は、サン・マイクロシステムズ社の Web サイトを参照してください。

JDBC 環境の設定

Oracle Lite データベースにアクセスするには、Java プログラムで最初に Oracle Lite JDBC ドライバをロードしてから、データベースへの接続を確立する必要があります。

Oracle Lite JDBC ドライバのロード

Oracle Lite JDBC ドライバをロードするには、Oracle Lite JDBC ドライバ・クラスの完全修飾名を `Class.forName` メソッドに渡します。

```
Class.forName("oracle.lite.poljdbc.POLJDBCDriver");
```

`Class.forName` メソッドは、指定された Oracle Lite JDBC ドライバ・クラスを検索し、ロードし、リンクします。`POLJDBCDriver` クラスのインスタンスを明示的に作成する必要はありません。`POLJDBCDriver` クラスのインスタンスは、`Class.forName` をコールしたときに JDBC ドライバ・マネージャにより自動的に作成され登録されます。

注意： Oracle Lite JDBC ドライバと JDK 1.2 JDBC ドライバでは特定のデータ型（CLOB や BLOB など）に対して同じ名前を使用するため、`java.sql.*` と `oracle.sql.*` の両方をインポートするプログラムからこのようなデータ型をコールすると、コンパイラ・エラーが返される可能性があります。このようなエラーの回避方法の詳細は、「制限事項」を参照してください。

Oracle Lite データベースへの接続

Oracle Lite データベースの JDBC 接続文字列は次のとおりです。

```
jdbc:polite:data_sourcename
```

`data_sourcename` は、**odbc.ini** ファイルで定義されているデータ・ソースの名前です。JDBC ドライバは、**odbc.ini** ファイルの *`data_sourcename`* により識別されるデータベースの物理的な位置を探します。

次の文では JDBC 接続文字列を使用して、データ・ソース名が `POLITE` の Oracle Lite データベースに接続します。

```
Connection conn = DriverManager.getConnection("jdbc:polite:POLITE",
   "system", "passwd");
```

ユーザー `system` として接続する場合は、パスワードには任意の文字を組み合せて指定できます。ただし、パスワードは空白にはできません。データベースが暗号化されている場合は、正しいパスワードを指定する必要があります。

Oracle Lite データベース内のデータにアクセスし操作するには、`getConnection` で返される `Connection` オブジェクトを使用します。

JDBC ドライバとの接続方法の詳細は、付録 B「JDBC アプリケーションのチュートリアル」を参照してください。

JDBC からの Java ストアド・プロシージャの実行

Java ストアド・プロシージャを作成した後は、次のようにして、JDBC アプリケーションからプロシージャを実行できます。

- ストアド・プロシージャを実行する SELECT 文字列を `Statement.executeQuery` メソッドに渡す。
- JDBC `CallableStatement` を使用する。

注意： ストアド・プロシージャの作成方法の詳細は、第 2 章「Java ストアド・プロシージャとトリガー」を参照してください。

executeQuery メソッドは、表レベルおよび行レベルのストアド・プロシージャを実行します。CallableStatement は、現時点では表レベルのストアド・プロシージャの実行しかサポートしていません。

ExecuteQuery メソッドの使用方法

executeQuery メソッドを使用してストアド・プロシージャをコールするには、まず、Statement オブジェクトを作成します。このオブジェクトには、カレントの Connect オブジェクトの createStatement メソッドにより返された値を割り当てます。次に Statement.executeQuery メソッドをコールします。このメソッドには、Java ストアド・プロシージャを起動する SELECT 文字列を渡します。

たとえば、変数 q に格納されている引数値を使用して、表 INVENTORY に対して行レベル・プロシージャ SHIP を実行するとします。変数 p には、ストアド・プロシージャの実行対象となる製品（行）の製品 ID が含まれています。

```
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;
```

パラメータを様々に変えてプロシージャを繰り返し実行する場合は、Statement のかわりに PreparedStatement を使用します。PreparedStatement 内の SQL 文はプリコンパイルされているため、PreparedStatement は効率良く実行できます。さらに、PreparedStatement は、文の中で疑問符（?）を使用して IN パラメータを設定できます。ただし、PreparedStatement が LONG や LONG RAW などの LONG 型パラメータを使用する場合は、setAsciiStream、setUnicodeStream または setBinaryStream メソッドを使用して、パラメータをバインドする必要があります。

前述の例で、プロシージャ SHIP でデータベースを更新する場合、上記の問合せを発行するトランザクションの分離レベルが READ COMMITTED であれば、SELECT 文に FOR UPDATE 句を追加する必要があります。次のように指定します。

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

Callable Statement の使用方法

Callable Statement を使用してストアド・プロシージャを実行するには、次のように、`CallableStatement` オブジェクトを作成してそのパラメータを登録します。

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

JDBC の Callable Statement の使用には、次の制限が適用されます。

- JDBC の Callable Statement で実行できるのは、表レベルのストアド・プロシージャのみです。
- IN パラメータと OUT パラメータの両方がサポートされるが、OUT パラメータとして使用する Java データ型に制限が適用されます。詳細は、第 2 章「Java ストアド・プロシージャとトリガー」を参照してください。
- プロシージャ名は Java メソッド名に対応し、大 / 小文字が区別されます。
- PreparedStatement の場合と同じように、Callable Statement が LONG 型（LONG、LONG VARBINARY、LONG VARCHAR、LONG VARCHAR2、LONG RAW など）を使用する場合は、`setAsciiStream`、`setUnicodeStream` または `setBinaryStream` メソッドを使用してパラメータをバインドする必要があります。

注意： `Resultset` オブジェクトや `Statement` オブジェクトなどの JDBC オブジェクトが不要になった場合、オブジェクトをクローズしてシステム・リソースを解放する必要があります。

Oracle Lite での拡張

Oracle Lite JDBC ドライバは JDBC 1.22 をサポートし、JDBC 2.0 で定義されている特定の機能拡張も提供します。この拡張には、BLOB（ラージ・バイナリ・オブジェクト）や CLOB（ラージ・キャラクタ・オブジェクト）データ型とデータ・アクセス関数に対するサポートが含まれています。Oracle Lite での JDBC 拡張は、Oracle8*i* の JDBC 実装と互換性があります。ただし、Oracle Lite では Oracle8*i* の JDBC データ型拡張、配列、構造体、`REF` はサポートしていません。

この項では、Oracle Lite のデータ型とデータ・アクセスに関する拡張を説明します。関数の構文やコール・パラメータの詳細は、次のサイトにあるサン・マイクロシステムズ社の『Java 2 仕様』を参照してください。

http://www.javasoft.com/products/jdk/1.2/docs/api/index.html

データ型の拡張

BLOB と CLOB は、データベース表に直接格納するには大きすぎるデータを格納するために使用されます。データベース表には、データではなく、実際のデータの場所を指すロケータが格納されます。BLOB には、非構造化バイナリ・データが大量に含まれ、CLOB には固定幅文字データ（1 文字につき一定のバイト数が必要な文字）が大量に含まれます。

BLOB または CLOB のロケータは、標準の SELECT 文を使用してデータベースから選択できます。SELECT 文を使用して BLOB または CLOB ロケータを選択すると、ラージ・オブジェクトのロケータのみが取得されます。データ自体は取得されません。ただし、ロケータがある場合、アクセス関数を使用してラージ・オブジェクトのデータを読み取ったり書き込んだりできます。

Oracle Lite の BLOB クラスには次のメソッドが含まれています。

機能	説明
`length`	BLOB の長さをバイト単位で返します。
`getBinaryOutputStream`	BLOB データを返します。
`getBinaryStream`	BLOB インスタンスをバイト・ストリームとして返します。
`getBytes`	指定された位置から、BLOB データをバッファに読み取ります。
`getConnection`	カレントの接続を返します。
`isConvertibleTo`	BLOB を特定のクラスに変換できるかどうかを判断します。
`putBytes`	BLOB データ内の指定された位置にバイトを書き込みます。
`makeJdbcArray`	JDBC 配列で表された BLOB を返します。
`toJdbc`	BLOB を JDBC クラスに変換します。

Oracle Lite の CLOB クラスには次のメソッドが含まれています。

機能	説明
`length`	CLOB の長さをバイト単位で返します。
`getSubString`	CLOB データ内の指定された位置からサブ文字列を取り出します。
`getCharacterStream`	CLOB データを Unicode ストリームとして返します。
`getAsciiStream`	CLOB インスタンスを ASCII ストリームとして返します。

機能	説明
getChars	CLOB データ内の指定された位置からの文字を文字配列に取り出します。
getCharacterOutputStream	Unicode ストリームから CLOB データを書き込みます。
getAsciiOutputStream	ASCII ストリームから CLOB データを書き込みます。
getConnection	カレントの接続を返します。
putChars	文字配列からの文字を、CLOB データ内の指定された位置に書き込みます。
putString	CLOB データ内の指定された位置にデータを書き込みます。
toJdbc	CLOB を JDBC クラスに変換します。
isConvertibleTo	CLOB を特定のクラスに変換できるかどうかを判断します。
makeJdbcArray	JDBC 配列で表された CLOB を返します。

データ・アクセスの拡張

Oracle Lite では、CLOB および BLOB データ型の値を設定および返すアクセス関数が提供されています。さらに、ラージ・オブジェクトに対するストリーム形式のアクセスを可能にする関数が、ストリーム・クラスにより提供されています。

OraclePreparedStatement クラス、OracleCallableStatement クラスおよび OracleResultSet クラスには、ラージ・オブジェクトに対するアクセス関数が含まれています。

OraclePreparedStatement クラスには、次のデータ・アクセス関数が含まれています。

機能	説明
setBLOB	BLOB データを設定します。
setCLOB	CLOB データを設定します。

OracleCallableStatement クラスには、次のデータ・アクセス関数が含まれています。

機能	説明
getBLOB	BLOB データを指すロケータを返します。
getCLOB	CLOB データを指すロケータを返します。

OracleResultSet クラスには、次のデータ・アクセス関数が含まれています。

機能	説明
getBLOB	BLOB データを指すロケータを返します。
getCLOB	CLOB データを指すロケータを返します。

POLLobInputStream、POLLobOutputStream、POLClobReader および POLClobWriter は、ストリームに対するアクセス・クラスです。

POLLobInputStream クラスには、次のデータ・アクセス関数が含まれています。

機能	説明
read	ラージ・オブジェクトから配列に読み取ります。

POLLobOutputStream クラスには、次のデータ・アクセス関数が含まれています。

機能	説明
write	出力ストリームからラージ・オブジェクトに書き込みます。

POLClobReader クラスは、クラス java.io.reader の継承です。次のデータ・アクセス関数が含まれています。

機能	説明
read	CLOB 内の文字を配列の一部に読み取ります。
ready	ストリームが読み取れる状態になっているかどうかを示します。

機能	説明
close	ストリームを閉じます。
markSupported	ストリームで mark 操作がサポートされているかどうかを示します。
mark	ストリーム内のカレント位置にマークを設定します。この後で reset 関数をコールすると、ストリームの位置がマークされた位置に変更されます。
reset	ストリーム内のカレント位置を、マークされた位置にリセットします。ストリームがマークされていない場合、この関数はそのストリームに適した（たとえば開始位置に戻すなどの）方法でストリームをリセットしようとします。
skip	ストリーム内の文字をスキップします。

POLClobWriter クラスは、クラス java.io.writer の継承です。次のデータ・アクセス関数が含まれています。

機能	説明
write	文字配列を出力ストリームに書き込みます。
flush	バッファ内のすべての文字を、目的の宛先に書き込みます。
close	ストリームをフラッシュして閉じます。

BLOB の読取りを行うサンプル・プログラム

次のサンプルでは、getBinaryStream メソッドを使用して BLOB データをバイト・ストリームに読み取ります。次に、バイト・ストリームをバイト配列に読み取り、読み取られたバイト数を返します。

```
// 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);
...
```

CLOB への書込みを行うサンプル・プログラム

次のサンプルでは、文字配列にデータを読み取り、次に getCharacterOutputStream メソッドを使用して文字配列を CLOB に書き込みます。

```
java.io.Writer writer;
// read data into a character array
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();
...
```

JDBC と Java アクセス・クラス間の相互運用性

JDBC インタフェースはデータのリレーショナル・モデルを使用するアプリケーションで使用され、Java アクセス・クラス（JAC）はデータのオブジェクト指向モデルを使用するアプリケーションで使用できます。どちらのインタフェースも同一のデータベースにデータを格納できます。一方のインタフェースで作成したデータは他方のインタフェースからアクセスできます。ただし、これには後述のような制限があります。

JDBC アプリケーションによって作成された表は、JAC インタフェースからはすべてクラスとして参照でき、JAC アプリケーションはその表の行をオブジェクトとして選択できます。表のプロシージャもすべて、対応するクラス内のメソッドとして JAC インタフェースから参照できます。

JAC を使用して作成したクラスは、次の条件がすべて満たされている場合、JDBC インタフェースからアクセスできます。

- JDBC アプリケーションで、JAC を介して作成されたクラスや属性名を二重引用符（" "）で囲んで参照している。これらの名前が大 / 小文字によって区別されるためです。
- JDBC で、JAC を介して作成されたクラスをドットをスラッシュに置き換えて参照している。たとえば、クラス oracle.personnel.Person からすべての Person オブジェクトの名前属性を JDBC で検索するには、次のように指定します。

  ```
  SELECT "Name"
  FROM "Oracle/personnel/Person" ;
  ```

  属性名とクラス名は二重引用符で囲まれ、クラス名のドットはすべてスラッシュに置き換えられています。
- クラスが最上位のクラスである。つまり、他のユーザー定義クラスのサブクラスではありません。

- クラスの属性のすべてが、JDBC のサポートするデータ型である。たとえば、JDBC では整数や文字列などのリテラル・データ型がサポートされますが、他のクラスを参照する属性タイプはサポートされません。サポートされている属性にアクセスするには、JAC クラスの SQL ビューを作成します。このビューは、JDBC でサポートされているデータ型の属性をクラスから選択します（クラス名は表名と同じです）。
- クラスのインスタンスが、グループを指定しないで作成されている。つまり、クラスのインスタンスがクラスのデフォルト・グループにおいて作成されているということです。Oracle Lite では、最上位クラスのそれぞれに対して、クラス名と同じ名前のデフォルト・グループを作成します。JAC を使用して作成したメソッドやトリガーは、すべて JDBC インタフェースからアクセスできます。

JDBC でサポートしていない属性の型は次のとおりです。

- すべての配列型（文字配列型とバイト配列型を除く）
- すべての OID 型

制限事項

書込み中にデータの切捨てが発生すると、SQL データ切捨て例外が発生します。読取り中にデータの切捨てが発生した場合は、SQL データ切捨て警告が発生します。しかし、データ切捨てオブジェクトには意味のある情報は含まれていません。

Oracle Lite の JDBC クラスと JDBC 2.0 クラスでは、特定のデータ型に同じ名前を使用しています（たとえば、`oracle.sql.Blob` と `java.sql.Blob`）。プログラムで `oracle.sql.*` と `java.sql.*` を両方ともインポートする場合、完全修飾名を指定せずに重複しているクラスにアクセスしようとすると、コンパイル・エラーが発生する可能性があります。この問題を回避するには、次の方法のいずれかを使用します。

1. BLOB、CLOB およびデータ・クラスには完全修飾名を使用する。
2. クラスは明示的にインポートする（たとえば、import `oracle.sql.Blob`）。
3. JDK 1.1 コンパイラで CLASSPATH に **oljdk11.jar** を指定しないでコンパイルする。

クラスには常に完全修飾のクラス名が含まれているため、データ型の名前が重複しても実行時に競合は発生しません。

4

Java 埋込み SQL（SQLJ）

SQLJ は、SQL 文を Java プログラムに埋め込むための規格です。SQLJ は、静的 SQL 文を必要とする Oracle Lite Java アプリケーションの中で使用できます。

この項では、Oracle Lite の開発における Oracle SQLJ ツールの使用方法の概要を説明します。説明する内容は、次のとおりです。

- SQLJ の概要
- SQLJ の操作
- SQLJ の使用方法

SQLJ の概要

SQLJ を使用すると、Java プログラムに静的 SQL 文を埋め込むことができます。静的 SQL 文は事前に定義された文です。つまり、ユーザーがプログラムを実行する際に変化しない文のことを指します。ただし、SQL 文に渡されるデータ値はホスト変数を使用して変更できます。現実には、アプリケーションには静的 SQL 文と動的 SQL 文の両方が必要になる場合がほとんどのため、SQLJ 文は JDBC 文と同じソース内に混在させて使用します。SQLJ は静的 SQL 文に、JDBC は静的 SQL 文と動的 SQL 文に使用します。

Oracle Lite の SQLJ サポートは Oracle8*i* と互換性があります。Oracle8*i* 用に作成された SQLJ プログラムは、あまり変更しなくても Oracle Lite で機能します。

SQLJ の操作

SQLJ 文は、`#sql` タグにより識別されます。SQLJ トランスレータは、SQLJ 文を SQLJ ランタイム・ライブラリへのコールに置き換え、クラスを生成します。トランスレータが SQLJ 文を変換するとき、構文がチェックされ、Java と SQL 間のデータ交換の互換性を確認するためにタイプ・チェックが実行されます。また、トランスレータはスキーマもチェックして、SQL 構文とデータベース・スキーマとの間で矛盾がないかどうか確認します。

次のサンプル・コードは、JDBC と SQLJ で同じ内容の SELECT 文を示しています。SQLJ 文では、入力および出力引数に、ホスト・プログラムで宣言された変数を使用します。JDBC には、各引数をバインドする文と、それぞれの結果を取り出す文が別々に必要です。

```
// SQLJ
   float w; java.sql.Date x; int y; String z;
   ...
   #sql { SELECT C1, C2 INTO :w, :x FROM TAB WHERE C3 = :y AND C4 = :z };


// JDBC
   float w; java.sql.Date x; int y; String z;
   ...
   PreparedStatement s = connection.prepareStatement(
      "SELECT C1, C2 FROM TAB WHERE C3 = ? AND C4 = ?");
   s.setInt(1, y);
   s.setString(2, z);
   ResultSet r = s.executeQuery();
   r.next();
   w = r.getFloat(1);
   x = r.getDate(2);
   r.close();
   s.close();
```

コンパイル済の SQLJ プログラムは、Java VM、SQLJ ランタイム・ライブラリおよび JDBC ドライバが利用できれば、どのような環境でも実行できます。

SQLJ の使用方法

SQLJ を使用するには、SQLJ トランスレータ、ランタイム・ライブラリおよび付属のマニュアルを、次の Oracle Web サイトからダウンロードします。

http://www.oracle.com/java/sqlj/index.html

配布パッケージには、SQLJ の詳しいドキュメンテーションが含まれています。以下の項では、Oracle Lite とともに SQLJ を使用する方法について説明します。

SQLJ の構成

SQLJ をインストールし構成するには、配布パッケージに含まれている手順に従います。Oracle Lite とともに使用するには、SQLJ パラメータを次のように構成します。

1. **connect.properties** ファイルで、SQLJ URL を次のように設定します。

   ```
   jdbc:polite:dsn_name
   ```

2. **sqlj.properties** ファイルで、SQLJ URL を次のように設定します。

   ```
   jdbc:polite:dsn_name
   ```

3. SQLJ ドライバを次のように設定します。

   ```
   oracle:lite:pol.jdbc.POLJDBCDriver
   ```

SQL スクリプトを実行して表をインストールした後、`conn.commit()` を使用して、変更をデータベースに明示的にコミットします（`conn` は Connection オブジェクト）。

Oracle Lite への接続

SQLJ アプリケーションでは、Oracle Lite データベースに接続する前に、まず Oracle Lite JDBC ドライバをロードする必要があります。JDBC ドライバをロードするには、アプリケーションで JDBC ドライバ・クラスの完全修飾名を引数として `Class.forName` メソッドに渡す必要があります。

```
class.forName("oracle.lite.poljdbc.POLJDBCDriver")
```

5

Java アクセス・クラス（JAC）

この章では、Oracle Lite Java アクセス・クラス（JAC）を使用してステートレス永続クラスを作成する方法を説明します。

説明する内容は、次のとおりです。

- Oracle Lite の新機能
- Java アクセス・クラス（JAC）の概要
- Java アクセス・クラスの主な機能
- データベースの管理
- ステートレス永続 Java クラスの作成
- オブジェクトの管理
- 単純永続オブジェクトの使用方法
- Java アクセス・クラス・パッケージ

注意： JAC アプリケーションのサンプルは、付録 C「JAC アプリケーションのチュートリアル」を参照してください。

Oracle Lite の新機能

Oracle Lite の今回のリリースでは、JAC の次の分野で拡張機能と新機能が追加されています。

- 単純永続インタフェース
- データ型サポートの拡張
- データ型変換サポート
- Java ストアド・プロシージャとトリガーのサポート
- ガベージ・コレクション
- レプリケーション・サポート
- 順序オブジェクト
- 索引の拡張
- 行識別子（ROWID）のサポート
- POLConnection

単純永続インタフェース

今までは、JAC はオブジェクトの永続性に関して 3 つの方法をサポートしていました。

- `POLPersistentCapable` クラスを実装して作成するステートレス永続クラス
- `POLPersistentCapableStateful` クラスを実装して作成するステートフル永続クラス
- Oracle Lite 永続 Java ポストプロセッサを使用して生成するシームレスな永続クラス

以上に加えて、JAC では単純永続インタフェースもサポートするようになりました。単純永続インタフェースには次のような特長があります。

- クラスを永続クラスにする場合に、（明示的な変更もポストプロセッサを介する変更も）変更は何も必要ない。クラスを継承することも、インタフェースを実装することも、永続フィールドをパブリックにすることも必要ありません。
- 同一クラスの一時インスタンスと永続インスタンスが、同一アプリケーション内で共存できる。
- 接続の管理、オブジェクトの永続化、データベース内の古いオブジェクトのリフレッシュ、オブジェクトへの変更済のマーク付けに関して、完全に制御できる（責任がある）。永続 Java ポストプロセッサを使用すると、以上のタスクの多くが自動的に管理されます。
- 多次元配列を使用できる。永続 Java ポストプロセッサは、多次元配列をサポートしません。

単純永続オブジェクトは、他のインタフェースのオブジェクトよりも大きいオーバーヘッドが必要になります。単純永続オブジェクトには POLObjHdl を直接指すポインタがないため、JAC ではこれらのオブジェクトをデータベース・オブジェクトにマップするハッシュ表をメンテナンスします。Oracle Lite では、同一アプリケーション内での複数インタフェースの使用をサポートしています。

詳細は、「単純永続オブジェクトの使用方法」を参照してください。

データ型サポートの拡張

スカラー型と一次元配列に加えて、JAC ではサポートされているアトミック・タイプの多次元配列をサポートするようになりました。多次元配列の属性を作成するには、POLAttrDesc コンストラクタを使用します。

```
POLAttrDesc(String name, Class type)
```

例を次に示します。

```
new POLAttrDesc("Names", String[].class); //  An Array of strings
new POLAttrDesc("id", int.class); // Integer attribute
new POLAttrDesc("data",
   java.math.BigDecimal[][].class); // Two-dimensional array of BigDecimal
```

属性オブジェクトの初期化方法としては、POLAttrDesc コンストラクタの使用が一般的です。他のコンストラクタは下位互換性を保つために提供されているもので、固定サイズ配列を作成したり、BigDecimal 属性用の明示的な精度とスケールを指定するために提供されています。新しい POLAttrDesc コンストラクタでは、精度とスケールは 19 と想定されています。JAC でサポートされているデータ型の詳細は、「永続可能属性のデータ型」を参照してください。

データ型変換サポート

JAC では、ユーザーが指定するデータ型とデータベースのデータ型との間で暗黙的な変換がサポートされるようになりました。たとえば、POLObjHdl.getStringVal を使用して整数属性を取り出すことができます。

JAC では次のような変換が実行されます。

1. クラス *A* には toString メソッドがあり、（別の）クラス *B* には、String 引数か valueOf メソッドをとるコンストラクタがあるとします。ここで、valueOf メソッドは String を引数として取り、*B* の新規インスタンスを返すとします。この場合は、クラス *A* のインスタンスを *B* にキャストできます。*A* と *B* が同じ場合、または *B* が *A* のスーパークラスである場合、オブジェクト・カーネルはクラスを変更しません。
2. プリミティブ型（int など）やラッパー型（java.lang.Integer など）は、相互に変換可能です。
3. POLObjHdl とその関連のユーザー・オブジェクトは、相互に変換可能です。

4. POLPersistentProxy とその関連の単純永続オブジェクトは、相互に変換可能です。
5. 配列 *A[]* は、配列の最初の要素を *A* にキャストできます。配列のサイズが 0 の場合は例外が発生します。この変換を繰り返して実行すれば、複数のディメンションを取得できます。
6. A は *A[]* にキャストして、要素が 1 つの配列を作成できます。

変換に失敗すると、POLException がエラー・コード POLException.TYPE_NOT_SAME で発生します。

次に示す POLObjHdl の新しい set メソッドでは、型変換が利用されています。このメソッドを使用して、属性を別の型の値に設定したり、プリミティブ型に NULL 値を設定したり、多次元配列に属性を設定できます。

```
public void set(POLAttr a, Object val)
public void set(int pos, Object val)
```

例を次に示します。

```
hdl.set(2, new int[][] {{1,2},{3,4}});
```

次に示す get メソッドも POLObjHdl に定義されていますが、これらは指定された Java 型に属性をキャストします。

```
public Object get(POLAttr a, Class domain)
public Object get(int pos, Class domain)
```

例を次に示します。

```
(int[][])hdl.get(2, int[][].class)
```

POLIterator も、列の型とユーザー指定値の型との間で暗黙的に値を変換します。

Java ストアド・プロシージャとトリガーのサポート

ストアド・プロシージャとトリガー用の JAC インタフェースが、新しい SQL モデルを反映して変更されています。詳細は、「Java ストアド・プロシージャとトリガー」を参照してください。

ガベージ・コレクション

JAC プログラムが JDK 1.2 とともに実行されると、Java 仮想マシンは JAC により返される永続オブジェクトに対してガベージ・コレクションを実行します。
POLObjHdl.discardObj を明示的にコールする必要はなくなりました。ただし、JDK 1.1 の場合はまだ明示的にオブジェクトを破棄する必要があり、パフォーマンスの最適化に POLClass.createNewObjQuiet を使用する必要があります。

POLConnection の新しいメソッドは、JAC キャッシュのサイズを監視します。

```
public synchronized int getCacheSize(int kind);
```

次の表は、kind に使用できる値を示します。

値	説明
CACHE_SIZE_NORMAL	オブジェクトをすべてカウントします。
CACHE_SIZE_PROXY	単純永続インタフェースを監視します。
CACHE_SIZE_DIRTY	ダーティ・オブジェクトをカウントします。

ガベージ・コレクション（または JDK 1.1 での明示的なオブジェクト管理）が正しく機能している場合は、キャッシュ・サイズが特定の範囲内で安定します。

レプリケーション・サポート

今回のリリースでは、POLClass オブジェクトに、整数のフラグを引数とするコンストラクタが追加されています。JAC では 2 種類のフラグがサポートされています。

値	説明
USE_SNAPSHOT	クラスのインスタンスに対して加えられるすべての変更（作成、変更、削除）のロギングを可能にします。
USE_DIRTY_VECTOR	各インスタンスに追加属性 OL__DIRTYMAP を追加します。この属性は、最後のリフレッシュ以降にどの属性が更新されているかを示す byte[] ビットマップです。

例を次に示します。

```
POLClass.createNewClass(conn, Person.class, POLClass.USE_SNAPSHOT)
```

変更されたインスタンスをすべて取得するには、型が POLIterator.SNAPSHOT のイテレータを作成します。たとえば、次のように指定します。

```
new POLIterator(cls, true, POLIterator.SNAPSHOT, "")
```

イテレータが返す各オブジェクトの状態を調べるには、POLObjHdl に定義されている getSnapState を使用します。戻り値として可能な値は次のとおりです。

値	説明
SNAP_CLEAN	オブジェクトは変更されていません。この値は、スナップショット・イテレータにより返されるオブジェクトには表示されません。他のオブジェクトに対して getSnapState をコールしたときには返されることがあります。
SNAP_NEW	オブジェクトは新規です。
SNAP_DIRTY	オブジェクトは変更されています。
SNAP_DELETED	オブジェクトは削除されています（スナップショットが切り捨てられるまでは、削除されたオブジェクトにスナップショットからアクセスできます）。
SNAP_GHOST	最後にリフレッシュされてから、オブジェクトが作成、削除されています。このようなオブジェクトは一般的には無視されます。

ローカルな変更が検出された場合、通常レプリケーション・プロトコルはスナップショットを消去し、サーバーから受け取った変更を適用している間はロギングを使用禁止にします。これは、POLClass に定義されている ctrlSnapshot をコールすることにより実行されます。ctrlSnapshot メソッドは、次のいずれかの引数をとります。

値	説明
SNAP_TRUNCATE	すべての変更（および変更されたインスタンスの OL__DIRTYMAP）を消去します。
SNAP_DISABLE	変更のロギングを一時停止します。これは一時的な操作です。セーブポイントのコミット、ロールバック、または取消しにより、ロギングは再び使用可能になります。
SNAP_ENABLE	ロギングを再開します。

POLAttrDesc クラスには setFlags メソッドが含まれています。このメソッドにより追加のレプリケーション・パラメータを設定できます。

値	説明
HIDDEN_ATTR	属性を SQL 文からは見えないようにします。これは JAC 自体では無視されますが、アプリケーションでは POLAttr.getMemberProperties を使用して調べることができます。
AUTO_INCREMENT	スカラー型の int 属性にのみ適用されます。このオプションを指定した属性には、新規オブジェクトの作成時に単調増分される値が割り当てられます。
PRIMARY_KEY	主キー列を、レプリケーション用に指定します。この属性が更新されると、スナップショットの 2 つのエントリが編集されます。古い主キー値を持つ削除済オブジェクトと、新しい主キー値を持つ新規オブジェクトです。

順序オブジェクト

POLSequence のインスタンスは、トランザクションとは独立して増分、減分、または問合せを実行できる永続整数値を保持します（ロールバックしても値は元には戻りません）。順序オブジェクトを使用すると、Users クラスの UserID 列のように、データベース内で一意性が保証される値を作成できます。

いずれかのトランザクション（順序を更新したトランザクションである必要はない）がコミットされると、順序がデータベースに書き込まれます。順序は複数のトランザクションが競合せずに更新できます。ただし、データベースへの唯一の接続を持つアプリケーションが順序を更新し、接続を閉じる前にクラッシュした場合、新しい値は保存されません。

新しい順序オブジェクトを作成したトランザクションがロールバックされると、その順序オブジェクトは消去されます。

索引の拡張

Oracle Lite では、索引オブジェクトに名前を付けられるようになりました。索引の名前は、各データベース固有にしてください。createNewIndex メソッドは、最初の引数として名前をとります。名前のない索引を作成するには、createNewIndex に NULL を渡します。索引オブジェクトは、名前により問い合せるか、またはクラスおよび索引が適用されるグループで問い合せることができます。

索引には新しい統計情報を利用できます。getStat メソッドは、次の情報を報告します。

- 索引内のオブジェクト数の合計
- 索引により占有される 4K バイト単位のデータベース・ページの数
- 個別のキー値の数

POLAttr.getRangeSize は、属性値に上限と下限を指定した問合せにより返されるオブジェクト数を返します。これらのメソッドは推定値を返すものであり、精度よりも速度に重点を置いて最適化されていることに注意してください。

行識別子（ROWID）のサポート

POLConnection の getObjById メソッドと POLObjHdl の getObjId は、永続オブジェクト・ハンドルと、行識別子（ROWID）がオブジェクトとして含まれているバイト配列との間の変換を行います。ROWID を使用すると、次のような作業を実行できます。

- オブジェクトに対する継続時間が長い参照を、別データベースまたはデータベースの外に格納する。
- 同一データベースへの複数の JAC 接続から、同一オブジェクトにアクセスする。
- JAC と JDBC から同一オブジェクトにアクセスする。

Oracle Lite では、通常、削除されたオブジェクトの ROWID は、削除されたオブジェクトの POLGroup が削除されるまでは再利用されません。この動作を変更するには、reuseoid オプションを指定して odbinfo ユーティリティを起動します。odbinfo の説明は、『Oracle Lite ユーザーズ・ガイド』を参照してください。その場合、JAC アプリケーションでは削除されたオブジェクトの ROWID を格納しないでください。

POLConnection

setIsolationLevel メソッドで、トランザクション分離レベルを SINGLE USER に設定できるようになりました。SINGLE USER 分離レベルは、JDBC と ODBC でまったく同じように機能します。一時点でデータベースに接続できる数は 1 つのみです。

CLASSLOADER 接続プロパティを使用すると、指定された POLClass に対応するユーザー・クラスを見つけるために JAC が使用するカスタム・クラス・ローダーをアプリケーションで指定できます。そのクラス・ローダーを使用してクラスを明示的にロードするには、loadClass メソッドを使用できます。

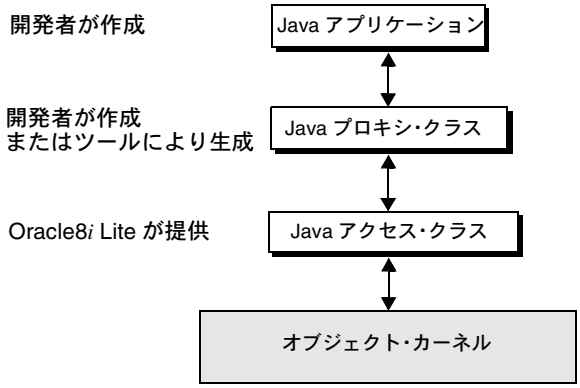
NOCLASSMAP 接続プロパティにより、POLClass とユーザー・クラスとのマッピングがすべて使用禁止になります。JAC では、システム・オブジェクト以外のオブジェクトはすべて POLPersistentObject のインスタンスとして返します。これは、データベース・クラスの操作時に、その名前と一致する名前を持つ Java クラスをロードしない設計ツールの場合に便利です。

getDatabaseID メソッドは、グローバルで一意のデータベース識別子を表す long 値を返します。データベース ID は、データベースの作成時に生成されます。

一時データベースに接続するには、getConnection にデータベース名として NULL を渡します。システムは一時データベースをメイン・メモリーに格納し、切断されるとメモリーを自動的に解放します。一時データベースではトランザクション操作はサポートされません。

Java アクセス・クラス（JAC）の概要

この項では、JAC ライブラリの機能とインタフェースの概要を説明します。JAC は、Java オブジェクトを直接 Oracle Lite に格納するためのオブジェクト指向インタフェースです。JAC は、次の図のように、Oracle Lite オブジェクト・カーネルと直接対話します。



JAC は、次の種類の永続オブジェクトをサポートします。

- ステートレス永続オブジェクト。
- ステートフル永続オブジェクト。詳細は、第 6 章「シームレスな永続 Java」を参照してください。
- 単純永続オブジェクト。

注意： JAC アプリケーションは、Oracle Lite でのみ使用できます。Oracle8*i* では JAC をサポートしていません。

Java プロキシ・クラス

Java アプリケーションと JAC レイヤーの中間に位置する Java プロキシ・クラスは、Oracle Lite のスキーマをモデル化します。Java プロキシ・クラスは、該当する JAC クラスに永続性の管理責任を委託します。プロキシ・クラスは、直接開発するか、Oracle Lite Designer を使用して生成するか、または永続 Java ポストプロセッサを使用して生成します。Oracle Lite 永続 Java ポストプロセッサの詳細は、第 6 章「シームレスな永続 Java」を参照してください。

注意： Oracle Lite Designer を使用する場合は、データベース・スキーマに単一継承が含まれる場合にのみ、Java プロキシ・クラスの生成が可能となります。

一時 Java クラスと永続 Java クラス

次の例は、一時 Java クラスの実装と Oracle Lite の永続 Java クラスの実装の違いを示します。

一時 Java クラス

Person クラスは一時クラスです。この Person から生成されるオブジェクトとその状態は、Java VM のガベージ・コレクションが実行されるとき、または対応する Java VM が終了するときに失われます。

```
public class Person
{
   private char[] name;
   private int age;
   private Person[] children;
   public Person(char[] name, int age, Person[] children)
   {
      this.name = name; this.age = age; this.children = children;
   }
   public char[] getName() { return name;}
   public void setName(char[] name) { this.name = name; }
   public int getAge() { return age;}
   public void setAge(int age) { this.age = age;}
   public Person[] getChildren() { return children; }
   public void setChildren(Person children) {this.children = children; }
}
```

Oracle8*i* Lite の永続 Java クラス

Person クラスを永続クラスにする方法を次に示します。この Person クラスから生成されるオブジェクトとその状態は、Oracle Lite データベースに保存されます。

```
import oracle.lite.jac.*;
public class Person extends POLPersistentObject
{
   public Person() {};

   // attribute getters and setters
   public synchronized char[] getName() throws POLException
   { return getObjHdl().getCharArrayValAt(0); }
```

```
   public synchronized void setName(char[] new_name)
   throws POLException
   { getObjHdl().setCharArrayValAt(0, new_name); }

   public synchronized int getAge() throws POLException
   { return getObjHdl().getIntValAt(1); }

   public synchronized void setAge(int age)  throws POLException
   { getObjHdl().setIntValAt(1, age); }

   public synchronized Person[] getChildren() throws POLException {
      return (Person[]) getObjHdl().getObjArrayValAt(2);

   }
   public synchronized void setChildren(Person [] new_children)
      throws POLException {
      getObjHdl().setObjArrayValAt(2, new_children); }
}
```

この例で示されているように、永続クラスには次のような特長があります。

- JAC クラス `POLPersistentObject` から派生する。`POLPersistentObject` には、`POLObjHdl` 型（永続オブジェクト・ハンドル）の変数が含まれています。永続オブジェクト・ハンドルは、Java 永続プロキシ・クラスとオブジェクト・カーネル内のオブジェクトの表記を結ぶものです。オブジェクト・ハンドルは、値の取出しと設定に使用します。
- 状態を何も持たない（ステートレス）。永続クラスにはインスタンス変数がありません。かわりに、アクセス用メソッドがデータベースに値を直接格納し取り出します。

永続クラスのサンプルは、「ステートレス永続 Java クラスの作成」に詳しく説明されています。

Java アクセス・クラスの主な機能

以下の項では、JAC パッケージの主な機能を説明します。

ステートフルな永続化とステートレスな永続化

Oracle Lite では、Java オブジェクトのステートフルな永続化とステートレスな永続化をサポートしています。

- ステートレスな永続化は、`POLPersistentCapable` インタフェースを実装するか、または `POLPersistentObject` クラスを継承するユーザー定義クラスを介してサポートされます。ステートレス・クラスは永続化状態情報を持たず、インスタンス変数もありません。
- ステートフルな永続化は、`POLPersistentCapableStateful` インタフェースを実装するか、または `POLPersistentObjectStateful` クラスを継承するユーザー定義の Java クラスを介してサポートされます。このようなクラスでは、Java フィールドに値を保持します。詳細は、第 6 章「シームレスな永続 Java」を参照してください。

Java オブジェクトの一意性

Oracle Lite では、同一接続の同一の永続データベース・オブジェクトに対して、ステートフル永続 Java クラスの Java オブジェクトが重複して作成されることはありません。一方、ステートレス永続 Java クラスの場合、デフォルトではその Java オブジェクトが重複して作成されます。この動作は、接続を確立するときに、`properties` 引数の該当するフラグを設定するとオーバーライドできます。

詳細は、Oracle8*i* Lite のドキュメンテーション・ページから参照できる『Oracle Lite Java アクセス・クラス API リファレンス』にある `POLConnection` を参照してください。

継承

Oracle Lite のオブジェクト・モデルでは、属性とメソッドの複数の継承がサポートされます。属性の継承には、次の規則が適用されます。

- 属性は、継承パスが複数指定されている場合でも、一度しか継承されません。たとえば、ダイヤモンド型継承グラフの場合、ダイヤモンドの最上部にある各属性に関して、最下部にあるクラスには各属性の 1 つのコピーしか継承されません。
- クラス内の各属性には、名前とソース ID（属性を定義したクラス）があります。

- Oracle Lite のオブジェクト・カーネルでは、クラス *C* の属性の定義を参照する際、次のアルゴリズムを使用します。
  - 属性の名前とソース・クラス *S* が指定されている場合、オブジェクト・カーネルはクラス *S* 内にあるその名前の属性を返します。
  - 名前のみが指定されていて、その名前の属性がクラス *C* に定義されている場合、オブジェクト・カーネルはその属性を返します。C に定義されていない場合は、オブジェクト・カーネルはその名前の属性をオブジェクトのスーパークラスから検索します。オブジェクト・カーネルは、クラス *C* が作成されたときに指定されている順序でスーパークラスを調べます。

Java の NULL と Oracle Lite の NULL

Java オブジェクト型に対応する永続オブジェクトの属性を初期化するには、オブジェクトの setter メソッド（`setXVals` と `setXValAts`。*X* は Java オブジェクト型）に Java の NULL 値を渡します。対応する getter メソッド（`getXVal` または `getXValAt`）を属性に対してコールすると、Java の NULL が返されます。

永続配列型の属性は、空の Java 配列で初期化できます。空の配列は、データベースの NULL とは等価ではありません。属性が NULL と等価のインスタンスを問い合せても、返されるインスタンスの属性は空の配列ではありません。

Java では、Java オブジェクト型ではないオブジェクトに関しては、`setXVals` と `setXValAts` メソッドに Java の NULL を渡すことは禁止されています。属性を NULL に初期化するには、永続オブジェクト・ハンドルに対して `setNullAttrValAt` または `setNullAttrVal` をコールします。属性が NULL に初期化された後、その属性に対して `getXVal` または `getXValAt` メソッドをコールすると、`POLNullValueException` が発生します。

バイト配列とバイナリ

永続属性がバイト配列の場合、適切な getter メソッド（`getByteArrayVal` または `getByteArrayValAt`）がコールされたときに、オブジェクト・カーネルはデータベースから配列全体を返します。ただし、永続属性がバイナリ型の場合は、その値の一部を取り出すことができます。このため、イメージやテキスト・データなど、値を部分的に取り出す必要のあるリソースには、バイナリ型を使用する必要があります。

大 / 小文字区別

Oracle Lite クラス名、グループ名、属性名およびメソッド名はすべて大 / 小文字が区別されます。

永続可能属性のデータ型

次のデータ型は、JAC を介して永続化できます。このようなデータ型は多次元でもかまいません。Oracle Lite は、Java と Oracle との間でデータ型を変換します。

Java の引数	SQL データ型
byte、Byte	BINARY、RAW、VARBINARY
boolean、Boolean	BIT
String	CHAR、VARCHAR、VARCHAR2
short、Short	SMALLINT
int、Integer	INT
float、Float	REAL
double、Double	DOUBLE、NUMBER（精度の指定なし）
BigDecimal	NUMBER(n)
java.sql.Date	DATE
java.sql.Time	TIME
java.sql.Timestamp	TIMESTAMP
oracle.lite.jac.POLPersistantCapable	オブジェクトの JAC インスタンス

データベースの管理

JAC は、データベースの作成と削除、およびデータベースへの接続を行う機能を提供します。

データベースの作成

データベースを作成するには、POLDatabase クラスに定義されている createNewDatabase メソッドをコールします。メソッドには完全修飾名と一意のボリューム識別子を渡します。createNewDatabase は、Java プログラムでのみ使用できるデータベースを作成します。指定されたデータベース名は大 / 小文字が区別されません。createNewDatabase には次のようなシグネチャがあります。

```
public static void createNewDatabase(String db_name, short volume_id)
   throws POLException
```

例を次に示します。

```
POLDatabase.createNewDatabase("C:¥¥DB¥¥TEST.odb", 1234);
```

これにより、TEST という名前の Oracle Lite データベースが C:¥DB ディレクトリに作成されます。

注意： Java 文字列リテラルでは、円記号文字は円記号 2 つ（¥¥）で表されます。

リレーショナル・アプリケーション（ODBC または JDBC アプリケーションなど）をデータベースに対して実行する場合は、データベースは Oracle Lite のユーティリティ **createdb.exe** を使用して作成する必要があります。

データベースの削除

POLDatabase に定義されている静的データベース削除メソッドを、完全修飾のデータベース名を使用してコールすると、データベースが削除されます。削除メソッドのシグネチャを次に示します。

```
public static void deleteDatabase(String db_name) throws POLException
```

データベース Test.odb を次のように削除します。

```
POLDatabase.deleteDatabase("C:¥¥DB¥¥TEST.odb");
```

createdb.exe により作成された Oracle Lite データベースを削除するには、**deletedb.exe** を実行する必要があります。詳細は「データベースの作成」を参照してください。

データベースへの接続

暗号化されていないデータベースに接続する場合は、次の構文を使用します。

```
public static POLConnection getConnection(String db_name, int options)
     throws POLException
```

パスワードの指定が必要なデータベースに接続する場合は、次の構文を使用します。

```
public static POLConnection getConnection(String db_name, String passwd,
     int options) throws POLException
```

たとえば、TEST.odb データベースに接続するには、次のように指定します。

```
POLConnection conn = POLConnection.getConnection("C:¥¥DB¥¥TEST.odb",
                     POLConnection.CONNECT_READ_WRITE);
```

TEST.odb がパスワード「secret」で暗号化されている場合は、次のように接続します。

```
POLConnection conn = POLConnection.getConnection("C:¥¥DB¥¥TEST.odb",
                    "secret", POLConnection.CONNECT_READ_WRITE);
```

最後に正常に接続されたデータベース接続がデフォルトの接続として扱われます。デフォルトの接続は、POLConnection オブジェクトを指定して setDefaultConnection をコールするとリセットできます。例を次に示します。

```
POLConnection.setDefaultConnection(conn);
```

トランザクションの管理

トランザクションとは、作業の単位を定義したものです。トランザクション中に行われた変更は、トランザクションをコミットした時点で確実に永続性を持つようになります。データベースに対して加えた変更は、トランザクションをロールバックするとすべて取り消されます。トランザクションが何もアクティブでない場合は、最初のデータベース操作が暗黙的にトランザクションを開始します。JAC では、トランザクションとセーブポイントのネストはサポートしていません。

Oracle Lite の Java インタフェースは、次の分離レベルをサポートします。

- READ COMMITTED
- REPEATABLE READ
- SERIALIZABLE
- SINGLE USER

トランザクションの分離レベルは、デフォルトでは REPEATABLE READ です。

トランザクションの明示的な開始

新規トランザクションを明示的に開始するには、POLConnection に定義されている beginWork メソッドを使用します。たとえば、分離レベルが SERIALIZABLE のトランザクションを開始するには、次のように指定します。

```
conn.beginWork(POLConnection.SERIALIZABLE);
```

トランザクションのコミット

トランザクションをコミットするには、次のようにします。

```
conn.commit();
```

このコマンドにより、データベースへの変更が保存されます。

トランザクションのロールバック

トランザクションをロールバックするには、次のようにします。

```
conn.rollback();
```

このコマンドにより、現在のトランザクションで加えられたデータベースに対する変更が取り消されます。

新しいセーブポイントの設定

セーブポイントにより、トランザクションのカレント位置をマークできます。セーブポイントをロールバックとともに使用すると、トランザクション全体ではなく、トランザクションの一部だけを元に戻すことができます。

新しいセーブポイントをマークするには、POLConnection で定義されている markSavePoint をコールします。

```
conn.markSavePoint();
```

セーブポイントの削除

保留中のセーブポイントとは、削除もロールバックもされていないセーブポイントを指します。保留中のセーブポイントを削除するには、removeSavePoint をコールします。

```
conn.removeSavePoint();
```

保留されているセーブポイントがない場合は、POLException が発生します。

セーブポイントまでのロールバック

セーブポイントより後にデータベースに対して加えられたすべての変更を取り消すには、rollbackToSavePoint をコールします。

```
conn.rollbackToSavePoint();
```

保留中のセーブポイントが複数存在する場合、rollbackToSavePoint メソッドは最後に作成されたセーブポイントを削除します。removeSavePoint も同じです。

保留中のセーブポイントが 1 つしかない場合、rollbackToSavePoint メソッドは、そのセーブポイント以降に行われた変更を取り消します。

データベース・スキーマの作成

Java オブジェクトを Oracle Lite に格納する前に、データベース・スキーマを作成する必要があります。JAC コア・クラスを使用すると、データベース・スキーマは永続 Java クラスの設計に基づいてプログラムで作成できます。データベース・スキーマと対話する永続 Java クラスは、手動で実装する必要があります。

Java クラスの設計から Oracle Lite のデータベース・スキーマにマップする場合は、次の指示に従ってください。

- Oracle Lite データベース・スキーマ・クラスは、各 Java クラスに対して同一名で作成する。
- Java クラスで定義された各属性に対し、Oracle Lite スキーマ・クラスに属性を作成する。Java クラスの属性のうち、Oracle Lite データベースにおいて直接永続的にできないデータ型を持つ属性については、Oracle Lite の永続的なデータ型に変換する必要があります。Oracle Lite のスキーマ・クラスの属性型は、`POLAttrDesc` クラスで定義されている該当の型を使用して、対応する Java の属性と同じ型にする必要があります。
- Oracle Lite のクラス属性は、作成時に NULL に指定できる。
- Java は単一クラス継承のみサポートする。
- Oracle Lite スキーマ・クラスの継承階層は、Java クラス設計で定義された階層に従う。Java クラスにより実装されたインタフェースは、Oracle Lite スキーマ・クラスでは無視されます。

次の例では、JAC データベース・スキーマをプログラムにより作成する方法を説明しています。

```
import oracle.lite.jac.*;

public class createschema {
   public createschema () {
   }
   public static void main(String args[]) {
      POLConnection conn = null;

      if (args.length !=1)
      {
         System.out.println("Usage: java createschema db_name!");
         return;
      }
      try {
         try {
             conn = POLConnection.getConnection("C:¥¥DB¥¥"+args[0], 0);
         }
         catch (POLException e)
         {
            System.out.println("Failed to connect ! errno ="+
            e.getErrNo());

            conn.rollback();
            return;
         }
```

```
            // create the Person class
            try {
               POLAttrDesc class1_attrs[] = new POLAttrDesc[3];
               class1_attrs[0] = new POLAttrDesc("Name",
                    POLAttrDesc.CHARARRAY_TYPE, POLAttrDesc.NONNULLABLE);
               class1_attrs[1] = new POLAttrDesc("Age",
                    POLAttrDesc.INT_TYPE, POLAttrDesc.NONNULLABLE);
               class1_attrs[2] = new POLAttrDesc("Children", "Person",
                   true, POLAttrDesc.NONNULLABLE);
               POLClass person = POLClass.createNewClass(conn,
                   new String("Person"),
                   null,
                   class1_attrs);
               conn.commit();
               System.out.println("end of Class1 creation¥n");
            }
            catch (POLException e1)
            {
               System.out.println("Failed to create Person class! errno = "
                   + e1.getErrNo());
               conn.rollback();
               return;
            }
            conn.disconnect();
         }
         catch (POLException out)
         {
            System.out.println(out);
         }
      }
   }
```

ステートレス永続 Java クラスの作成

この項では、Oracle Lite で JAC を使用してステートレスな永続 Java クラスを開発する方法を説明します。JAC ステートレス永続 Java クラスには、次のような特長があります。

- `POLPersistentCapable` インタフェースを実装するか、`POLPersistentObject` を継承する。
- インスタンス変数がない。
- 対応するデータベース・クラスと同じ名前に指定する必要がある（可能な場合）。
- `POLObjHdl` 型の引数をとるコンストラクタを定義する。

- 各永続属性用に setter メソッドおよび getter メソッドを提供する。
- クラスのインスタンスを生成する静的メソッドを提供する。

Person クラスの一時的な実装を次に示します。

```
public class Person
{
   private char[] name;
   private int age;
   private Person[] children;
   public Person(char[] name, int age, Person[] children)
   {
      this.name = name; this.age = age; this.children = children;
   }
   public char[] getName() { return name;}
   public void setName(char[] name) { this.name = name; }
   public int getAge() { return age;}
   public void setAge(int age) { this.age = age;}
   public Person[] getChildren() { return children; }
   public void setChildren(Person[] children) {this.children = children; }
}
```

次の例は、このクラスを永続化する方法を示します。最初の例では、POLPersistentCapable インタフェースを実装して永続 Person クラスを実装する方法を示します。2 番目の例では、POLPersistentObject クラスから継承することにより永続 Person クラスを実装する方法を示します。

例 1: 永続インタフェースの実装

この Person クラスは、POLPersistentCapable インタフェースを実装します。

```
import oracle.lite.jac.*;

public class Person implements POLPersistentCapable

{
   private static POLClass classHdl = null;
   private POLObjHdl objHdl;
   public Person (POLObjHdl hdl) {};

   // create a new persistent Person object
   public static Person NewPersonObject(POLConnection conn, char[] name,
            int age, Person[] children) throws POLException
      {
      if (classHdl == null || classHdl.getObjHdl() == null ||
                  conn == classHdl.getConnection())
         classHdl = POLClass.findClass(conn, "Person");
         Person obj = (Person)(classHdl.createNewObj(null,null));
```

```
            obj.setName(name);
            obj.setAge(age);
            obj.setChildren(children);
            return obj;
        }
    // implements the methods defined in POLPersistentCapable
    public synchronized void clearObjHdl() { objHdl.deleteObj(); objHdl =
        null; }
    public synchronized POLObjHdl getObjHdl() { return objHdl; }
    public synchronized void setObjHdl(POLObjHdl obj_hdl) { objHdl =
        obj_hdl;}
    public synchronized void deleteObj() throws POLException {
        getObjHdl().deleteObj();  }

    // attribute getters and setters
    public synchronized char[] getName() throws POLException  {
        return getObjHdl().getCharArrayValAt(0); }

    public synchronized void setName(char[] new_name)
            throws POLException
        { getObjHdl().setCharArrayValAt(0, new_name); }

    public synchronized int getAge() throws POLException {
        return getObjHdl().getIntValAt(1);
    }

    public synchronized void setAge(int age)  throws POLException {
        getObjHdl().setIntValAt(1, age);
    }

    public synchronized Person[] getChildren() throws POLException {
        return (Person[]) getObjHdl().getObjArrayValAt(2);
    }
    public synchronized void setChildren(Person [] new_children)
              throws POLException {
        getObjHdl().setObjArrayValAt(2, new_children);
    }
}
```

この場合、Person クラスは POLPersistentCapable インタフェースに定義されている clearObjHdl、getObjHdl および setObjHdl メソッドを実装します。また、各永続属性用の getter メソッドおよび setter メソッドと、Person オブジェクトの作成および検索に役立つメソッドも含まれています。

例 2: 永続クラスの継承

この永続 Person クラスは、POLPersistentObject クラスを継承します。clearObjHdl、getObjHdl および setObjHdl メソッドは実装されません。.

```
import oracle.lite.jac.*;
public class Person extends POLPersistentObject
{
   private static POLClass classHdl = null;
   public Person(POLObjHdl hdl) {};

   // create a new persistent Person object
   public static Person newPersonObject(POLConnection conn,
                       char[] name,
                       int age,
                       Person[] children)
                       throws POLException {
      if (classHdl == null || classHdl.getObjHdl() == null ||
                          conn == classHdl.getConnection())
         classHdl = POLClass.findClass(conn,"Person");
      Person obj = ((Person)classHdl.createNewObj(null, null));
      obj.setName(name);
      obj.setAge(age);
      obj.setChildren(children);
      return obj;
   }
   // attribute getters and setters
   public synchronized char[] getName() throws POLException  {
      return getObjHdl().getCharArrayValAt(0); }

   public synchronized void setName(char[] new_name)
                        throws POLException {
      getObjHdl().setCharArrayValAt(0, new_name);
   }
   public synchronized int getAge() throws POLException {
      return getObjHdl().getIntValAt(1);
   }
   public synchronized void setAge(int age)  throws POLException {
      getObjHdl().setIntValAt(1, age);
   }
   public synchronized Person[] getChildren() throws POLException {
      return (Person[]) getObjHdl().getObjArrayValAt(2);
   }
   public synchronized void setChildren(Person [] new_children)
                              throws POLException {
      getObjHdl().setObjArrayValAt(2, new_children);
   }
}
```

オブジェクトの管理

この項では、永続オブジェクトの作成方法と管理方法を説明します。

永続オブジェクトの作成

永続オブジェクトを作成するには、POLClass に定義されている静的メソッド createNewObj を使用します。createNewObj は、次のようにコールできます。

```
Person obj = ((Person)classHdl.createNewObj(null, null));
```

classHdl は POLClass のインスタンスです。createNewObj はユーザー定義のクラスを返します。または classHdl のマッピングが見つからない場合は POLPersistentObject を返します。

永続オブジェクトの問合せ

JAC では、属性値の走査または非定型問合せにより、オブジェクトの問合せをサポートしています。属性の走査は、永続オブジェクト・ハンドルに対して getObjVal、getObjValAt、getObjArrayVal および getObjArrayValAt をコールするという方法で行えます。

POLEnumeration とそのサブクラス POLIterator および POLOrderByIterator では、問合せ機能を持つ様々なイテレータが提供されています。POLIterator は基本的なイテレータです。POLOrderByIterator は、指定された順序でオブジェクトのリストを返します。これらのイテレータは、トランザクションが終了してもクローズされません。close メソッドを使用して明示的にクローズしない場合は、アプリケーションがデータベースから切断されるときにクローズされます。

イテレータ内に検索条件を作成するには、プログラムで POLSearchCond 式を使用するか、または述部の文字列表現を使用します。述部には検索条件が含まれます。述部でサポートされている演算子は次のとおりです。

演算子	定義
=	等しい
!=	等しくない
<	より小さい
<=	より小さいかまたは等しい
>	より大きい
>=	より大きいかまたは等しい
isnull	NULL である
isnotnull	NULL でない
&&	論理積
||	論理和

ブール型定数は TRUE と FALSE です。これらには大 / 小文字区別はありません。

POLClass に定義されているメソッド queryObjs を使用すれば、特定のクラスのオブジェクトを問い合せることができます。このメソッドは、文字列引数をとります。

一定の条件を満たすオブジェクトを検索するには、POLIterator を使用できます。例を次に示します。

```
POLClass a_class = POLClass.findClass(conn, "Person");
POLIterator it  = a_class.queryObjs(
         "age <40 or age > 60");
while (it.hasMoreElements())
   Person per = (Person)it.nextElement();
```

これで、61 才以上 40 才未満の Person オブジェクトがすべて検索されます。

永続オブジェクトの削除

永続オブジェクトを削除するには、次のように deleteObj メソッドをコールします。

```
objHdl.deleteObj();
```

このメソッドは、Java オブジェクトとデータベース・オブジェクト間のリンクを切断します。（POLObjHdl オブジェクトにより表される）リンクが切断されると、Java 仮想マシンはデータベース・オブジェクトに対してガベージ・コレクションを実行できます。

永続オブジェクトの属性の取出し

POLObjHdl クラスは、JAC の属性のタイプごとに getter メソッドを定義します。永続オブジェクトの属性値を取り出すには、取り出す属性の型に適した getter メソッドを使用します。たとえば、ユーザー定義型 Person の配列である属性 *X* を取り出すには、次のように getObjArrayValAt を使用します。

```
synchronized Person[] getX() throws POLException
{
   return (Person[])getObjHdl().getObjArrayValAt(2);
}
```

getObjArrayValAt は、属性の位置を表す整数値を引数に取ります。POLObjHdl クラスは、位置に基づき POLAttr オブジェクトに基づく属性の取出し用メソッドを提供します。POLAttr オブジェクトに基づく属性の取出し用メソッドは、getObjArrayVal です。

永続オブジェクトの属性の更新

POLObjHdl クラスは、JAC の属性のタイプごとに setter メソッドを定義します。永続オブジェクトの属性値を設定するには、設定する属性のタイプに適した setter メソッドを使用します。たとえば、ユーザー定義型 Person の配列である属性 *X* を設定するには、次のように setObjArrayValAt を使用します。

```
synchronized void setX(POLObjHdl [] new_val)
                   throws POLException
{
   objHdl.setObjArrayValAt (2, new_val);
}
```

setObjArrayValAt の最初の引数は、設定する属性の位置を示します。2 番目の引数は POLPersistentCapable 型の新しい値を表します。POLObjHdl クラスは、位置に基づき POLAttr オブジェクトに基づく属性の setter メソッドを提供します。POLAttr オブジェクトに基づく属性の setter メソッドは、setObjArrayVal です。

永続バイナリ属性の操作

Oracle Lite の永続バイナリ型属性は、実際はオブジェクト型です。次の例では、永続バイナリ属性の作成方法を示します。

```
POLAttrDesc attr_desc= new POLAttrDesc("Image",
   POLAttrDesc.BINARY_TYPE,
   POLAttrDesc.NULLABLE);
```

属性を作成した後、次のようにその属性を初期化します。

```
POLBlob blob = new POLBlob(group, size);
blob.setBlobVal(0, val);
objHdl.setObjVal(attr, blob);
```

この例では、

- group は、blob が格納されるグループです。
- size は、blob の初期の最大サイズです。
- val はバイト配列です。
- attr はイメージ・オブジェクトを保持します。
- objHdl は永続オブジェクト・ハンドル（POLObjHdl）を表します。

部分的なバイナリ値の取出しは、次のように行います。

```
POLBlob blob = (POLBlob)objHdl.getObjVal(attr);
byte[] val = blob.getBlobVal(100, 10);
```

これにより、100 番目のバイト（相対 0）から始まるバイト配列として 10 バイトが返されます。

制約

制約を作成するには、JAC の POLConstraint クラスを使用します。Oracle Lite のオブジェクト・カーネルは、JAC および SQL を使用して作成された制約を施行します。JAC では、次の宣言制約をサポートしています。

- 主キー制約
- 一意性制約
- NOT NULL 制約
- 外部キー制約

制約に違反すると、POLException が発生します。

POLConstraint クラスは、次のメソッドを提供します。

- 特定の種類の制約を作成するメソッド
- 特定の種類の制約を問い合せるメソッド
- 制約を削除するメソッド

制約を作成するには、createSimpleConstraint メソッドを使用します。たとえば、次のコードにより、Name 属性が主キーになります。

```
POLClass person_class = POLClass.findClass(conn, "Person");
POLAttr    name_attr = person_class.findAttr ("Name");
POLAttr[] attrs = new POLAttr[1];
   attrs[0] = name_attr;
POLConstraint   cons = POLConstraint.createSimpleConstraint
   ("name_unique", POLConstraint.PRIMARY,
   person_class, attrs);
```

name_unique は制約名を表します。制約を検索する最も簡単な方法は、次のように名前で問い合せる方法です。

```
POLConstraint cons = POLConstraint.findConstraint(conn, "name_unique");
```

制約を削除するには、POLConstraint に定義されている deleteObj メソッドを使用します。

クラスに対して制約を定義した後、JAC メソッドがそのクラスのオブジェクトを作成または変更するたびに、Oracle Lite オブジェクト・カーネルにより制約チェックが実行されます。このため、オブジェクトを作成または変更するときは必ず JAC メソッドを使用し、制約チェックが起動されるようにします。

たとえば、オブジェクトの作成には createNewObj を使用します。

```
POLClass.createNewObj(POLAttr[] attrs, object[] values)
```

属性値の設定には setAttrVals を使用します。

```
POLObjHdl.setAttrVals (POLAttr[] attrs, object[] values)
```

永続オブジェクト・ディレクトリ

JAC では、頻繁にアクセスされるオブジェクトに対して、永続オブジェクト・ディレクトリ・サービスが提供されます。このディレクトリは、オブジェクト・ハンドルと名前を関連付けます。オブジェクトのディレクトリ名を使用すると、オブジェクトをすばやく参照できます。JAC の POLDirectory クラスがオブジェクト・ディレクトリを管理します。

オブジェクト名をディレクトリに入れるには、putObject メソッドを使用します。putObject は String と POLObjHdl を引数に取ります。String 引数で渡されるオブジェクト名は、ディレクトリ内で一意にしてください。次の例は、ディレクトリにオブジェクトを入れる方法を示します。

```
POLDirectory root = POLDirectory.getRootDirectory(conn);
Person person =...;
...
root.putObject("Bill", person);
```

オブジェクトの取出しには getObject を使用します。

```
Person person = (Person)root.getObject("Bill");
```

オブジェクト名を使用すると、ディレクトリ全体を繰り返し取り出すことができます。例を次に示します。

```
Enumeration names = root.getNames();
while (names.hasMoreElement())
{
   String name = names.nextElement();
   System.out.println(root.getObject(name));
}
```

putObject への 2 番目の引数として null を渡して、別ディレクトリの下にサブディレクトリを作成できます。例を次に示します。

```
root.putObject("level1", null);
```

ディレクトリからオブジェクトを削除するには、removeObject を使用します。例を次に示します。

```
root.removeObject("Bill");
```

Java ストアド・プロシージャとトリガー

ストアド・プロシージャとは、データベース内に格納されている Java メソッドのことです。トリガーは、特定のデータベース・イベントが発生したときに実行されるストアド・プロシージャです。トリガーは次のような状況下で実行できます。

- オブジェクトの作成前
- オブジェクトの作成後
- オブジェクトの削除前
- オブジェクトの削除後
- オブジェクトの更新前
- オブジェクトの更新後

トリガーを作成するには、次の手順に従います。

1. トリガーのロジックを実装するクラスを作成しコンパイルします。
2. クラスとそのリソースを Oracle Lite データベースにロードします。
3. クラス・ハンドルを createTrigger メソッドに渡します。

注意： この項では、POLJavaObj クラスを使用してクラスをロードしトリガーを作成する方法を説明します。詳細は、Oracle8*i* Lite のドキュメンテーション・ページから参照できる『Java アクセス・クラス API リファレンス』を参照してください。

クラスのロード

クラスは個別にまたは JAR アーカイブとして Oracle Lite にロードできます。JAC ではクラスとリソースをデータベースにロードする方法をいくつか用意しています。1 つの方法は、オブジェクトを BLOB としてデータベースにロードし、次に POLJavaObj の loadClass メソッドか loadResource メソッドをコールします。loadClass には次のシグネチャがあります。

```
public static POLJavaObj loadClass(POLConnection conn, String schema,
                                   POLBlob blob);
```

リソースをロードするには、次のシグネチャを持つ loadResource を使用します。

```
public static POLJavaObj loadResource(POLConnection conn, String schema,
                                   String name, POLBlob blob, boolean isSource);
```

loadClass と loadResource は、クラスまたはリソースを表す POLJavaObj オブジェクトを返します。POLJavaObj を削除すると、BLOB も削除されます。loadclass は、次に示すように BLOB のかわりに Class 引数を受け入れるようオーバーロードされます。

```
public static POLJavaObj loadClass(POLConnection conn, String schema,
                                   Class hdl);
```

ストアド・プロシージャとトリガーは、データベース内に格納されていないクラスにアクセスできます。Java クラスを実際にデータベース内に格納しなくても、その Java クラスでストアド・プロシージャとトリガーにアクセスできるようにすることができます。これには、POLBlob のかわりに Class 引数を指定して loadClass メソッドをコールします。

JAR ファイルを Oracle Lite にロードしたり、ソースをコンパイルして結果のクラスを格納するには、次のシグネチャを持つ POLJavaObj を使用します。

```
public static POLJavaObj[] loadBundle(POLConnection conn, String schema,
                                   String path)
```

POLJavaObj への引数の中で、schema は NULL または有効なデータベース・スキーマです。Oracle Lite ではこの引数は処理されません。

トリガーの作成

クラスに POLJavaObj を作成した後、findMember または getMembers を使用して、属性およびメソッドのハンドルを取得します。メソッドの検索には、その名前と JNI シグネチャを使用するか、または名前と引数の数を使用します。これにより POLJavaMember のインスタンスが返されるので、これを使用してトリガー（POLClass.createTrigger）を作成します。

ストアド・プロシージャのコール

データベースに格納されているクラスの（トリガー以外の）Java メソッドをコールするには、POLConnection.getClassLoader を使用してクラスの classloader オブジェクトを取り出します。データベース・クラスとメソッドをコールするリフレクション API にアクセスするには、getClassLoader により返される classloader オブジェクトの loadClass を使用します。

リレーショナル・データベースのアクセス

Oracle Lite データベースに格納されている表とデータに関する情報にアクセスするには、JAC を使用します。オブジェクト・カーネルは、データベース内の表を、同じ名前の Oracle Lite クラスにマップします。名前の中のドット（.）はスラッシュ（/）に置き換えられます。同様に、表の列は Oracle Lite クラスの属性に自動的にマップされます。JAC は、リレーショナル・モデルで作成されたすべてのメタレベル情報とユーザー・データにアクセスできます。

次の表は、SQL 列のデータ型と Java 属性のデータ型とのマッピングを示します。

SQL データ型	Java データ型
CHAR	char
CHAR(len)	char[]
VARCHAR(len)	char[]
すべての LONG 型	POLBlob
DECIMAL/NUMERIC	java.math.BigDecimal
BIT	boolean
TINYINT	byte
SMALLINT	short
INTEGER	int
BIGINT	long
REAL	float

SQL データ型	Java データ型
FLOAT/DOUBLE	`double`
BINARY/BINARY(len)/ VARBINARY	`byte[]`
LONG VARBINARY	`POLBlob`
DATE	`java.sql.Date`
TIME	`java.sql.Time`
TIMESTAMP	`java.sql.Timestamp`

単純永続オブジェクトの使用方法

単純永続インタフェースを使用すると、最小限のコーディングで永続 Java オブジェクトを作成できます。単純永続クラスでは、特別のクラスを継承することも、インタフェースを実装することも、永続フィールドをパブリックにすることも必要ありません。単純永続インタフェースを使用すると、簡単に Java クラスを定義しデータベースに追加できます。

この項では、単純永続インタフェースの概要を説明します。詳細は、Oracle8*i* Lite のドキュメンテーション・ページから参照できる『Java アクセス・クラス API リファレンス』を参照してください。

単純永続クラスの作成

単純永続インタフェースを使用するには、まず、パブリック Java クラスの階層を定義します。クラス内の永続フィールドには、JAC でサポートされている型（たとえば、`int`、`String[][]`、`POLPersistentCapable[]` など）を使用します。また、他の単純永続オブジェクトを直接指すフィールドを定義することもできます。一時的な静的フィールドは永続型ではないため、どの型でもかまいません。

単純 Java クラスの例を次に示します。

```
public class Human {
   String name;
}
```

`Student` クラスでは `Human` を継承し、新規属性 `classes` を定義しています。

```
public class Student extends Human {
   Course[] classes=new Course[0];
}
```

`Human` と `Student` を永続的にするには、`createNewClass` を使用します。

```
POLClass student = POLClass.createNewClass(conn, Student.class)
```

これで、Student クラスと、java.* パッケージの階層にはないスーパークラスが作成されます。JAC では java.lang.Object 用の POLClass を作成しません。階層全体で 1 つの POLGroup を共有します。このため、Student オブジェクトを返すイテレータを Human に対してオープンできます。

このかわりに、createNewClass を使用してクラス階層を明示的に作成することもできます。これには、Java クラスに現れる順序で、互換性のある型を使用して属性を宣言します。これを行うには、特に POLAttrDesc コンストラクタが便利です。

クラスを作成後、そのクラスは名前により取り出すか、または findClass メソッドを使用して取り出すことができます。

```
student = POLClass.findClass(conn, Student.class)
```

オブジェクトの作成と格納

単純永続オブジェクトは、標準のコンストラクタを使用して作成できます。オブジェクトのインスタンスを生成した後、POLConnection に定義されている create メソッドを使用してそのオブジェクトをデータベースに格納します。

```
Student carole = new Student("Carole");
POLPersistentProxy proxy = conn.create(carole);
```

プロキシ・クラスの使用

JAC メソッドの多くは、POLPersistentCapable のインスタンスを引数として取ります。ただし、単純永続オブジェクトではこのインタフェースを実装しません。このため、POLDirectory や POLIterator などの JAC 機能を使用できるように、単純永続オブジェクトは（POLPersistentCapableStateful インタフェースを実装する）POLPersistentProxy のインスタンスと 1 対 1 で対応しています。

プロキシ・オブジェクトと対応するユーザー・オブジェクトをマップするには、POLPersistentProxy の次のメソッドを使用できます。オブジェクトを取り出すには、getObj を使用します。

```
public Object getObj();
```

プロキシ・オブジェクトを取り出すには、getProxy を使用します。

```
public static POLPersistentProxy getProxy(POLConnection conn, Object obj);
```

getProxy は、永続オブジェクト obj に対応するプロキシ・オブジェクトを取り出します。obj がデータベースにない場合は、NULL が返されます。POLConnection.create メソッドに対して既存のオブジェクトを引数として渡すと、このメソッドは、getProxy と同じように、そのオブジェクトのプロキシ・オブジェクトを取り出します。

永続オブジェクトの管理

単純永続オブジェクトの管理には、次のメソッドを使用できます。

delete

POLConnection に定義されている delete メソッドは、単純永続オブジェクトをデータベースから削除します。トランザクションまたはセーブポイントのロールバック後、削除されたオブジェクトが再び表示されることがあるので注意してください。delete には次のシグネチャがあります。

```
public void delete(java.lang.Object obj)
```

reload

reload メソッドは、ロールバックが発生して単純永続オブジェクトが古くなっているかどうかを確認し、必要であればオブジェクトをデータベースからリロードします。次のシグネチャがあります。

```
public void reload(java.lang.Object obj)
```

POLClass.getInstances メソッドと POLPersistentProxy.getObj メソッドは、reload を暗黙的にコールします。これにより、これらのメソッドが返すオブジェクトの妥当性が保証されます。これに比べて、シームレスな永続クラスでは、ポストプロセッサにより生成されたコードが、永続フィールドのアクセスの前に、reload と同等のものを暗黙的にコールします。

setDirty

POLConnection の setDirty は、オブジェクトが変更されていてデータベース内で更新の必要があることを JAC に通知します。次のシグネチャがあります。

```
public void setDirty(java.lang.Object obj)
```

JAC は次にコミットされる前にデータベースを更新します。setDirty 操作は、オブジェクトが最新でないと失敗します。このため、通常はオブジェクトを次のように更新する必要があります。

```
conn.reload(bankAccount);
bankAccount.balance += 1000000;
conn.setDirty(bankAccount);
```

マルチスレッド環境で更新漏れがないように、すべての変更が終わった後で必ず setDirty をコールします。また、単純永続オブジェクトの変更に、POLObjHdl の設定用メソッドを使用しないでください（ただし、オブジェクト・フィールドをすべて同じ値に明示的に設定する場合は使用してもかまいません）。そうでないと、オブジェクトとデータベースが同期しなくなります。

getInstances

POLClass の getInstances メソッドは、クラスの中で POLIterator の検索条件を満たす単純永続インスタンスをすべて返します。次のシグネチャがあります。

```
public java.lang.Object[] getInstances(java.lang.String query)
```

クラスの全オブジェクトを取り出すには、引数を指定しないで getInstances をコールします。

```
public java.lang.Object[] getInstances()
```

単純永続オブジェクトにおけるオブジェクト参照

単純永続オブジェクトは、他のオブジェクトを直接参照するポインタを持っています。JAC では、(データベース内の階層全体を格納することによる)「到達可能性」と(リロードにより階層全体を最新に保つ)「カスケード・ロード」により、単純永続オブジェクトの永続性をサポートします。循環データ構造は正しく処理されます。ただし、JAC ではカスケード・ロードを使用しているため、リンクされたリスト内のノードを取り出すと、リスト全体がメモリーにロードされます。これを避けるには、フィールドは直接参照としてではなく POLPersistentProxy として宣言する必要があります。

JAC では到達不能オブジェクトはデータベースから自動的には削除しません。データ構造を削除する場合、アプリケーションですべてのメンバー・オブジェクトを明示的に削除する必要があります。CASCADE_DEL オプションを指定した POLAttr.createRelation を使用すると、ほとんどの場合、この問題は解決されます。

例

次の例は、単純永続インタフェースの使用方法を示します。この例は次のファイルで構成されています。

クラス	説明
Human.java	Student と Professor の共通スーパークラス。
Professor.java	Professor は 0 クラス以上の授業を担当します。
Student.java	Student は 0 クラス以上の授業を受講します。
Course.java	Student と Professor オブジェクトをリンクします。
University.java	アプリケーションを駆動します。

Human.java

Human クラスには次の定義が含まれています。

```
public class Human {
   String name;
   Human(String name) { this.name = name; }
   public String toString() {
      return getClass().getName()+" "+name;
   }
}
```

Professor.java

Professor クラスは Human を継承します。

```
public class Professor extends Human {
   Course[] classes = new Course[0];
   public Professor(String name) { super(name); }
   public void teach(Course course) {
      course.prof = this;
      Course[] tmp = new Course[classes.length+1];
      System.arraycopy(classes, 0, tmp, 0, classes.length);
      tmp[classes.length] = course;
      classes = tmp;
    }
}
```

Student.java

Student クラスも Human を継承します。

```
public class Student extends Human {
    Course[] classes=new Course[0];
    public Student(String name) { super(name); }
    public void take(Course course) {
      Course[] tmp = new Course[classes.length+1];
      System.arraycopy(classes, 0, tmp, 0, classes.length);
      tmp[classes.length] = course;
      classes = tmp;
      course.addStudent(this);
    }
}
```

Course.java

Course クラスは、Professor と Student 配列で構成されます。

```
public class Course {
    String name;
    Professor prof;
    Student[] students = new Student[0];
    public Course(String name) { this.name = name; }
    void addStudent(Student student) {
      Student[] tmp = new Student[students.length+1];
      System.arraycopy(students, 0, tmp, 0, students.length);
      tmp[students.length] = student;
      students = tmp;
    }
    public String toString() {
      StringBuffer sb = new StringBuffer("*** Course "+name+"¥n");
      sb.append("Taught by: "+prof+"¥n");
      sb.append("Taken by: ");
      for (int i = 0; i < students.length; i++) {
          if (i > 0) sb.append(", ");
          sb.append(students[i].toString());
      }
      sb.append("¥n");
      return sb.toString();
    }
}
```

University.java

University クラスは main メソッドを提供します。このメソッドを使用して、Course、Professor および Student の各クラスをテストできます。

```
import oracle.lite.jac.*;

public class University {
    static final String dbName = "polite.odb";
    public static void main(String[] args) throws Throwable {

      // Create a database
        try { POLDatabase.deleteDatabase(dbName); }
        catch (POLException e) {}
      POLDatabase.createNewDatabase(dbName, (short)9999, 11);

      // Connect to the new database
      POLConnection conn = POLConnection.getConnection(dbName);
```

```
        // Create the persistent classes
        POLClass student = POLClass.createNewClass(conn, Student.class);
        POLClass prof = POLClass.createNewClass(conn, Professor.class);
        POLClass course = POLClass.createNewClass(conn, Course.class);

        // Create the object hierarchy
        Course cs134 = new Course("C.Sci 134"), cs217 = new Course("C.Sci 217"),
            art101 = new Course("Art 101");
        Professor alice = new Professor("Alice"), bob = new Professor("Bob");
        alice.teach(cs134); alice.teach(cs217);
        bob.teach(art101);
        Student carole = new Student("Carole"), david = new Student("David"),
            elaine = new Student("Elaine");
        carole.take(cs134); carole.take(art101);
        david.take(cs217);
        elaine.take(cs134); elaine.take(cs217);

        // Store the course information into the database
        conn.create(cs134); conn.create(cs217); conn.create(art101);

        // Verify that the information is indeed stored
        conn.commit();
        conn.disconnect();
        conn = POLConnection.getConnection(dbName);
        course = POLClass.findClass(conn, Course.class);
        Object[] list = course.getInstances();
        for (int i = 0; i < list.length; i++)
            System.out.println(list[i]);
        conn.disconnect();
    }
}
```

Java アクセス・クラス・パッケージ

JAC パッケージの主要コンポーネントは、次のクラスで構成されています。メソッドのパラメータや戻り値を含む JAC パッケージ全体の完全な情報は、Oracle8*i* Lite のドキュメンテーション・ページからアクセスできる『Java アクセス・クラス API リファレンス』を参照してください。

クラス	説明
POLAttr	クラス属性を表します。永続属性は、属性のプロパティ（属性の名前、型、属性が定義されているクラス、格納位置など）で問い合せることができます。
POLAttrDesc	永続属性のプロパティをカプセル化します。

クラス	説明
POLBlob	バイナリ・ラージ・オブジェクト（BLOB）を表します。BLOB は、イメージやテキスト・データ用に使用されることがよくあります。
POLClass	Oracle Lite クラスを記述するメタ・オブジェクト。永続クラスのプロパティ（クラスの名前や永続属性など）を、対応する `POLClass` オブジェクトに対して問い合せることができます。
POLClassMember	メンバー名とクラス名をカプセル化します。この 2 つは、クラス属性とクラス・メソッドの共通プロパティです。このクラスは、`POLAttrDesc` のスーパークラスです。
POLConnection	データベース接続セッションをモデル化します。
POLConstraint	制約のクラス・モデル。このクラスは、Oracle Lite データベース内の制約の作成、問合せおよび削除を実行するメソッドを提供します。
POLDatabase	静的データベース・オブジェクトをモデル化します。
POLDirectory	永続オブジェクトの名前に基づく永続オブジェクト・ディレクトリを提供します。永続オブジェクトを名前でアクセスできるようにします。一般に、ディレクトリは、頻繁にアクセスする永続オブジェクトにのみ使用するようにします。
POLEnumeration	様々な種類のイテレータ用のメソッドを提供します。`java.util.Enumeration` インタフェースを実装します。
POLException	Oracle Lite データベース操作中に検出されたエラーを示します。`POLException` オブジェクトの `getErrNo` メソッドをコールすると、エラー・コードが返されます。
POLGroup	データベース内に作成されたグループ・オブジェクトを表します。
POLIndex	Oracle Lite クラスの属性に対して作成された索引を表します。
POLIterator	指定された検索条件を満たす `POLPersistentCapable` インスタンスのコレクションを取り出し、取り出されたインスタンスを反復できます。
POLJavaMember	Oracle Lite に格納されている単純永続 Java クラスのフィールドまたはメソッドを表します。
POLJavaObj	Oracle Lite に格納されているオブジェクトまたはリソースを表します。

クラス	説明
POLObjHdl	Oracle Lite オブジェクト・カーネル内に格納されている永続オブジェクトへのアクセスを提供します。オブジェクト・ハンドルは、Java 永続プロキシ・オブジェクトとデータベース・オブジェクトをつなぐ役割を果たします。
POLOrderByIterator	指定された順序でオブジェクトを返すイテレータ。
POLPersistentCapable	ステートレス永続 Java クラスで実装する必要のあるプロトコルを定義します。Java クラスを永続的にするには、クラスまたはそのスーパークラスの 1 つでこのインタフェースを実装する必要があります。
POLPersistentCapableStateful	`POLPersistentCapable` を継承するインタフェース。このインタフェースは、ステートフル永続 Java クラスで実装する必要があります。
POLPersistentObject	`POLPersistentCapable` インタフェースのデフォルトの実装を提供します。ほとんどの永続 JAC クラスは、`POLPersistentObject` クラスから導出されます。このクラスを継承すると、Oracle Lite プロキシ・クラスを作成できます。
POLPersistentObjectStateful	`POLPersistentCapableStateful` インタフェースのデフォルトの実装を提供します。このクラスのサブクラスは、独自の `readObject` メソッドと `writeObject` メソッドを提供する必要があります。
POLPersistentProxy	単純永続オブジェクトを永続可能インタフェースにマップします。`POLPersistentProxy` は、`POLPersistentCapableStateful` インタフェースを実装します。
POLSearchCond	バイナリ比較式または単項比較式を表します。可能な Java リテラル値のそれぞれにコンストラクタを提供します。これらのコンストラクタは、Java 値をストリーム表現に変換します。`POLIterator` コンストラクタに渡される `POLSearchCond` オブジェクトの配列は、論理式の選言的正規形を表します。
POLSequence	順序を表します。順序を使用すると、データベースの表の数値列に対して一意の数値の連続リストを生成できます。
POLStatistics	グループまたは索引に関する情報を返します。
POLTrigger	Oracle Lite の永続トリガーをモデル化します。

6

シームレスな永続 Java

この章では、Oracle Lite のシームレスな永続 Java サポートについて説明します。説明する内容は、次のとおりです。

- シームレスな永続 Java の概要
- シームレスな永続クラスの作成
- ステートフル永続オブジェクトの管理
- ステートフル永続クラスの手動による実装

シームレスな永続 Java の概要

シームレスな永続 Java ポストプロセッサにより、Java クラスを Oracle Lite データベース内で永続的にするための作業が自動化されます。次のコンポーネントによりシームレスな永続 Java が使用できるようになります。

- 拡張 Java Access Class（JAC）パッケージ
- Java クラス・ポストプロセッサ

これらのコンポーネントを使用すると、Java クラスと Oracle Lite クラスのマッピングを意識せずに永続 Java クラスを開発できます。ポストプロセッサは、クラスのマッピング作業を管理し、永続性に必要なプロトコルを実装します。ポストプロセッサは、処理されるクラスのそれぞれに永続ステートフル・クラスを生成します。

永続クラスを作成するために、ポストプロセッサは次を実行します。

- 指定されたクラスを調べて、Oracle Lite の適切なスキーマを作成する。
- `POLPersistentCapable` インタフェースを実装し、かつ Oracle Lite の Persistent Glue Protocol に準拠するクラスを生成する。

ポストプロセッサはステートフル永続クラスを作成します。JAC は、クラスから生成されたオブジェクトの値を使用して Oracle Lite データベース内のデータ値を同期します。

シームレスな永続クラスの作成

シームレスな永続 Java クラスを作成するには、次の手順を実行します。

1. 一時 Java クラスを作成します。
2. ソースをコンパイルします。
3. シームレスな永続ポストプロセッサでクラスを処理します。
4. 生成されたクラスを CLASSPATH に入れます。
5. Java アプリケーションを実行します。

以上の手順の例を次に示します。まず、永続的な `Person` クラスを定義します。次のコードを持つ **Person.java** を作成します。

```
import oracle.lite.jac.*;
public class Person
{
   private String name;
   private int age;
   private String address;
```

```
   // other person methods
   public Person(String name, int age, String address)
   {  this.name = name; this.age= age;
      this.address = address;
   }

   public String getName()
   { return name; }

   public void setName(String name)
   {
      this.name = name;
   }

   public int getAge()
   { return age; }

   public void setAge(int age)
   {
      this.age = age;
   }

   public String getAddress()
   { return address; }

   public void setAddress(String address)
   {
      this.address = address;
   }
}
```

Person クラスをテストするには、次のコードを持つ **test.java** を作成します。

```
import oracle.lite.jac.*;
public class test {
   public static void main(String[] args) {
      try {
         POLConnection conn = POLConnection.getConnection(args[0],
            POLConnection. CONNECT_READ_WRITE);
         Person p = new Person("Kevin Smith", 30,
            "San Francisco, CA");
         conn.commit();
         conn.disconnect();
      }
      catch (POLException e)
```

```
        {
           System.out.println(e);
        }
    }
}
```

javac を使用してファイル **Person.java** と **test.java** をコンパイルします。

ファイルをコンパイルした後、コマンドラインから Oracle Lite ポストプロセッサを実行します。次の例では、`Person` クラスを Oracle Lite ポストプロセッサに渡しています。

```
java oracle.lite.polp.polp -v -d c:¥Orant¥Oldb40¥Polite.odb -src . -t
     c:¥temp Person
```

ポストプロセッサは Oracle Lite のクラス **Person.class** を作成し、これを C:¥temp に入れます。**Person.class** ファイルを CLASSPATH に指定し、次のようにテスト・プログラムを実行します。

```
java test C:¥Orant¥Oldb40¥Polite.odb
```

テスト・プログラムは、指定されたデータベース内に `Person` インスタンスを作成します。

アプリケーションの構造

一般に、Oracle Lite の永続 Java アプリケーションは次の手順を実行します。

1. `POLConnection.getConnection` を使用してデータベースに接続します。データベースに接続すると、トランザクションが暗黙的に開始されます。
2. アプリケーション・ロジックを実装します。
3. データベースの変更をコミットまたはロールバックします。
4. データベースから切断します。

プログラムでは、以上の手順は次のようになります。

```
try {
   POLConnection conn = POLConnection.getConnection(...); // Step 1
   ...                                                    // Step 2

   conn.commit();  // or conn.rollback();                   Step 3
   conn.disconnect();                                  //  Step 4
}
catch (POLException e)
{
   System.out.println(e);
   ...
}
```

try-catch ブロックがデータベースへのアクセスを囲んでいます。try-catch ブロックは、`POLException` 例外を検出します。

シームレスな永続 Java ポストプロセッサ

ポストプロセッサを起動するには、次を入力します。

```
java oracle.lite.polp.polp [-v | -d database_name  | -s class_name  | -np |
   -cn class_incl_exc_file ] -t target_dir -src source_dir class_names/jar file
```

ポストプロセッサのオプション

フラグとフラグ値を使用して、ポストプロセッサにオプションを指定できます。オプションを 2 つ以上指定する場合は、オプションとオプションの間にスペースを入れます。ポストプロセッサは次のオプションをサポートします。

フラグ	可能な値	説明
-v		処理中にメッセージを表示します。デフォルトでは、このオプションは使用可能になっていません。処理中に詳しい状態の出力を表示する場合は、このオプションを使用可能にします。
-d	database_name	オプションです。ポストプロセッサは、クラスのポストプロセス中に、指定されたデータベースに Oracle Lite スキーマを作成します。 database_name は、完全修飾パスを指定した **.odb** であることが必要です。例を次に示します。 -d C:¥Orant¥Lite¥Oldb40¥Polite.odb
-s	class_name	オプションです。ポストプロセッサは、Oracle Lite データベース内にスキーマを作成できる JAC アプリケーションを生成します。このオプションは、JAC アプリケーションのクラス名を指定するために使用します。テキスト・ファイルは、-t オプションで指定されたターゲット・ディレクトリに入れられます。
-t	target_dir	必須です。ポストプロセッサが生成済ファイルを入れるディレクトリを指定します。
-src	source_dir	必須です。処理対象のクラスまたは JAR ファイルの場所を指定します。source_dir と target_dir には同一ディレクトリを指定しないでください。
-np		オプションです。一時 Java クラスから永続クラスへの変換を抑止します。このオプションは、提供されている Java クラスを使用してデータベース・スキーマを生成するために、-s または -d オプションといっしょに使用します。例を次に示します。 -np -d C:¥Orant¥Lite¥Oldb40¥Polite.odb -s schema

フラグ	可能な値	説明
-cn	class_incl_exc_file	JAR ファイルまたは Java クラス名リストを指定する場合はオプションで、これらを指定しない場合は必須です。このファイルは、JAR ファイルから除外する Java クラスまたはポストプロセスに含める Java クラスをリストします。class_incl_exc_file は、完全修飾名であることが必要です。
	JAR ファイル名または Java クラス名リスト	Java クラス名のリストは、スペースで区切った一連の Java クラス名（完全修飾）であることが必要です。例を次に示します。 test.Person test.Student リスト内のクラスはどんな順序でもかまいません。
-gc		クラスを生成します。

-cn class_incl_exc_file オプションにより、処理対象のファイルを指定します。構文は次のとおりです。

```
[+ | -] file_name
```

ポストプロセッサは、名前の前に正符号（+）のあるファイルを変換し、負符号（-）の付いたファイルは除外します。名前の前に符号がないクラスはポストプロセスに含まれます。符号とクラス名の間にはスペースを挿入する必要があります。1 行に 1 つの名前を指定します。クラス名は完全修飾であることが必要です。例を次に示します。

```
- postprocess.test.NotPersistentCapable
postprocess.test.PersistentClass1
- postprocess.test.NotPersistentObject
+ postprocess.test.PersistentClass2
```

ポストプロセッサの起動

次に、ポストプロセッサの起動方法のサンプルを示します。

サンプル 1 次のコマンドにより、`Person`、`Student` および `Employee` の各クラスに対してポストプロセッサが起動されます。クラスは C:¥input ディレクトリにあります。新規クラスが C:¥output に入れられます。スキーマを再作成できる JAC アプリケーションが、C:¥output に入れられます。Oracle Lite スキーマが、データベース `C:¥orant¥lite¥oldb40¥polite.odb` に入れられます。

```
java oracle.lite.polp.polp -v -t C:¥output -d C:¥orant¥lite¥oldb40¥polite.odb -s
test_schema -src C:¥input test.Person test.Student test.Employee
```

サンプル 2 次のコマンドにより、C:¥input にある **test.jar** の中のすべてのクラスが処理されます。新規クラスが C:¥output に入れられます。スキーマを再作成できる JAC アプリケーションが、C:¥output ディレクトリに入れられます。ポストプロセッサがクラスを処理中に、データベース `C:¥orant¥lite¥oldb40¥polite.odb` に Oracle Lite スキーマが作成されます。

```
java oracle.lite.polp.polp -v -t C:¥output -d C:¥orant¥lite¥oldb40¥polite.odb -s
test_schema -src C:¥input test.jar
```

サンプル 3 次のコマンドにより、一時 Java クラスから永続クラスへの変換が抑止されます。新規クラスは作成されません。ただし、Java クラスにマップされる Oracle Lite スキーマは作成されます。

```
java oracle.lite.polp.polp -np -v -t C:¥output -d C:¥orant¥lite¥oldb40¥polite.odb -s
test_schema -src C:¥input test.Person test.Student test.Employee
```

サンプル 4 次のコマンドにより、**test.jar** にある `test.transientclass1` および `test.transientclass2` 以外のすべてのクラスが処理されます。新規クラスが C:¥output に入れられます。スキーマを再作成できる JAC アプリケーションも、C:¥output に入れられます。データベース `C:¥orant¥lite¥oldb40¥polite.odb` 内に Oracle Lite スキーマが作成されます。

```
java oracle.lite.polp.polp -v -t C:¥output -d C:¥orant¥lite¥oldb40¥polite.odb -cn
C:¥working¥excl.txt -s test_schema -src C:¥input test.jar
```

ここでは、**excl.txt** に次のエントリが含まれていることが前提です。

```
- test.transientclass1
- test.transientclass2
```

ポストプロセッサの規則と参照

Java ポストプロセッサには次に示す規則が適用されます。

クラス永続性規則

ポストプロセッサは、（-cn オプションを使用して）コマンドラインまたはファイル class_inc_file に指定されているクラスを処理します。ポストプロセッサが、-d オプションで指定されている名前と同じ名前のクラスをデータベース内で見つけた場合、Java クラスは永続です。永続 Java クラスは、そのスーパークラスも永続クラスであることを意味するものではありません。

フィールド永続性規則

Java クラス内のフィールドが、第 5 章「Java アクセス・クラス（JAC）」の「永続可能属性のデータ型」に示されている基本型のいずれかの場合、そのフィールドは永続です。また、フィールドがユーザー定義クラスあるいは（クラス永続性規則に基づき永続である）ユーザー定義型の配列の場合も、フィールドは永続です。

処理対象の Java クラス

永続にしたい Java クラスは、すべて 1 つのポストプロセス・コマンドに指定する必要があります。そうでないと、処理されていない他の Java クラスで構成される永続クラスのフィールドは、一時フィールドになります。この結果、データベース・スキーマが不完全になります。永続にするクラスと、永続可能 Java フィールドを直接アクセスする一時クラスの両方を処理する必要があります。

処理済クラスの CLASSPATH への指定

ポストプロセッサは、一連の Java クラスを新たに作成します。実行時での使用のために、新規クラスを CLASSPATH に指定する必要があります。元のクラスまたはアーカイブ・ファイルは CLASSPATH に含める必要はありません。

手動による手順

シームレスな永続 Java を使用するときは、次の手順を明示的に行う必要があります。

- 不要になった永続オブジェクトを削除する。
- データベースに接続および切断する。
- トランザクションを適切にコミットまたはロールバックする。
- 配列フィールドの要素が変更されときに、オブジェクト prepareForUpdate と dirty をマークする。

セマンティクス

シームレスな永続 Java クラスには、次の規則が適用されます。

- 任意のコンストラクタ（Java コンパイラにより生成されたコンストラクタを含む）を起動すると、データベース内にオブジェクトが自動的に作成される。永続オブジェクトは、JAC を使用して明示的に削除する必要があります。
- キャッシュされた永続 Java オブジェクトの状態は、選択した分離レベルを基にデータベースとの整合性が保たれる。
- オブジェクトを取り出した接続が切断された後で、キャッシュされた永続 Java オブジェクトを使用しない。
- Java クラスはすべて Oracle Lite クラスにマップされる。Oracle Lite クラスの名前には、対応する Java クラスの完全修飾名が使用されます。
- オブジェクトが変更または作成された時点での整合性制約チェックは保証されない。ただし、トランザクションがコミットされるまでに整合性制約のチェックが行われることは保証されます。変更されたオブジェクトに対して `POLConnection.store` をコールすると、整合性制約チェックが行われます。例を次に示します。

  ```
  Person p;
  ...
  p.name = "John Smith";
  conn.store(p);
  ```

  Oracle Lite オブジェクト・カーネルは、`Person` オブジェクトをデータベースに格納するときに、`name` フィールドに対して定義されている制約をチェックします。

ポストプロセス後のデバッグ

元のクラスにデバッグ情報が含まれている場合は、ポストプロセッサにより生成されたクラスをデバッグできます。

ステートフル永続オブジェクトの管理

シームレスな永続 Java サポートを利用することで、開発者は永続オブジェクトの作成、削除、取出しおよび変更ができます。永続オブジェクトには状態情報が保持されています。JAC により、Java オブジェクトとそれに対応するデータベース・オブジェクトとの間で、状態情報が同期されます。

データベースからステートフル永続オブジェクトを取り出すとき、そのオブジェクトの同一性はそのまま保持されます。Oracle Lite では、同一の接続内で重複する Java 永続オブジェクトを作成する必要はありません。

新しいオブジェクトの作成

永続 Java クラスのオブジェクト・コンストラクタをコールすると、データベース・オブジェクトが作成されます。たとえば、次の文により、指定された値で Person オブジェクトが作成され、このオブジェクトがデータベース内に入れられます。

```
Person p = new Person("John Smith", 30, "San Francisco");
```

オブジェクトの削除

Oracle Lite からオブジェクトを削除するには、次のシグネチャを持つ POLConnection.deleteObj をコールします。

```
public void deleteObj(Object obj) throws POLException;
```

次のように指定すると、Person オブジェクトが作成され、このオブジェクトがデータベースから削除されます。

```
Person p = new Person("John Smith", 30, "San Francisco");
conn.deleteObj(p);
```

オブジェクトの取出し

次のいずれかの方法で、データベースからオブジェクトを取り出すことができます。

- オブジェクトのフィールド値の走査
- POLIterator

オブジェクトのフィールド値の走査

データベースからオブジェクトを取り出すには、別の永続クラス型の属性値にアクセスできます。たとえば、p は Person クラスのインスタンスで、型が Parent の属性 parent を持つ場合、次のように指定します。

```
Parent parent = p.parent;
```

イテレータ

次の例では、POLIterator クラスを使用して、データベースからオブジェクトを取り出します。

```
POLClass person_class = POLClass.findClass(conn, "Person");
POLIterator it = new POLIterator(person_class, true, "");
Person p = (Person)it.next();
while (p != null)
   p = (Person)it.next();
```

これにより、永続 Person オブジェクトがすべてデータベースから取り出されます。オブジェクトの取出しに関する説明は、第 5 章「Java アクセス・クラス（JAC）」の「永続オブジェクトの問合せ」を参照してください。

属性値のアクセス

永続オブジェクトの属性値は、一時オブジェクトのフィールド値にアクセスする方法と同じ方法でアクセスできます。たとえば、name が Person クラスで定義されている属性であるとして、次のように指定します。

```
String name = person.name;
```

オブジェクトの変更

永続オブジェクトの属性値は、一時オブジェクトのフィールド値を変更する方法と同じ方法で変更できます。

```
person.salary += person.salary * 0.10;
```

配列属性の変更

Java の Array 型の永続属性を変更するには、コードを追加する必要があります。次の例では、配列の要素 1 つを新しい値に変更しています。

```
person.houses = new House[10];
for (int I=0; I < person.houses.length; I++)
   person.houses[I] = new House("house"+I);
```

永続配列属性の要素を置き換える前に、それが含まれるオブジェクトに対して prepareForUpdate メソッドをコールする必要があります。配列要素を変更した後は、それが含まれるオブジェクトに対して dirty メソッドをコールする必要があります。例を次に示します。

```
person.houses = new House[10];
conn.prepareForUpdate(person);
for (int I=0; I < person.houses.length; I++)
   person.houses[I] = new House("house"+I);
conn.dirty(person);
```

次のサンプルでは、新規配列が配列全体を置き換えるため、追加コードは必要ありません。

```
House houses = new House[10];
for (int i=0; i < houses.length; i++)
    houses[i] = new House("house"+i);
person.houses = houses;
```

特定のトランザクションで変更する永続配列オブジェクトに対して prepareForUpdate メソッドを 1 回コールする必要があります。ただし、配列オブジェクト自体を変更するときは、常に dirty をコールする必要があります。Oracle Lite オブジェクト・カーネルは、dirty メソッドがコールされないと、永続オブジェクトが変更されたことを認識できません。

Oracle Lite では、Java 配列は Oracle Lite オブジェクトとしては格納されません。Java 空間において 1 つの Java 配列オブジェクトを共有する 2 つの永続 Java オブジェクトは、データベースから取り出された時点で、それぞれが 1 つずつ独自の配列オブジェクトを持つようになります。2 つのオブジェクトが配列オブジェクトの同一のコピーを指すようにするには、配列を別の永続クラスとして明示的にモデル化する必要があります。

オブジェクトの廃棄

永続オブジェクトが不要になった場合、discardObj をコールしてオブジェクトとその関連リソース（対応するデータベース・オブジェクト・ハンドルなど）を解放します。たとえば、ステートフル永続オブジェクト p を廃棄するには、次のように指定します。

```
conn.discardObj(p);
```

JDK 1.2 を使用して実行中の場合は、Java 仮想マシンが、削除された JAC オブジェクトに対して自動的にガベージ・コレクションを実行するので注意してください。discardObj を明示的にコールする必要はありません。

PostActivate

データベース・オブジェクトを Java 空間に入れるプロセスをアクティブ化といいます。オブジェクトがデータベースからアクティブ化されたときに、オブジェクトに特別の処理が必要な場合は、対応するクラス内に postActivate コールバック・メソッドを実装できます。postActivate のシグネチャは次のとおりです。

```
public void postActivate()
```

このコールバック・メソッドは、この Java クラスの永続オブジェクトがデータベースから初めてアクティブ化されたときに実行されます。

PriorDelete

オブジェクトがデータベースから削除される前に、オブジェクトに特別の処理が必要な場合は、対応するクラス内に priorDelete コールバック・メソッドを実装できます。priorDelete は、データベース・オブジェクトが削除される前に実行されます。

トランザクション処理後またはデータベース切断後のオブジェクトの状態

トランザクション中またはセーブポイント中に取り出された永続オブジェクトは、トランザクションが終了するか、セーブポイントが取り消された後は無効な状態になります。このようなオブジェクトを新規トランザクションで再びアクセスすると、（必要な場合は）Oracle Lite オブジェクト・カーネルがオブジェクトを最新の状態情報と比較して次のように妥当性をチェックします。

- 削除されたすべての永続オブジェクトは、トランザクションがコミットされた後に一時状態になる。
- 新たに作成された永続オブジェクトは、トランザクションがロールバックされた後に一時状態になる。
- すべての永続オブジェクトは、データベースが切断された後に一時オブジェクトになる。
- 一時オブジェクトになった永続オブジェクトは使用しない。このようなオブジェクトはデータベースの状態を反映していない可能性や、オブジェクトへの更新がデータベースにポストされない可能性があるためです。

リフレッシュ

Oracle Lite では、接続内でアクティブ化されているすべての永続オブジェクトの妥当性をチェックするために、接続ベースの `refresh` メソッドを提供します。`refresh` メソッドは、次のようにコールします。

```
conn.refresh();
```

Oracle Lite カーネルは、（必要な場合は）ダーティ・オブジェクトをディスクに書き出します。

ステートフル永続クラスの手動による実装

場合によっては、ポストプロセッサを使用しないで、ステートフル永続クラスを手動で実装することがあります。ステートフル永続クラスを作成するには、次を実行する必要があります。

- `POLPersistentCapableStateful` インタフェースを直接的または間接的に実装する。
- Persistent Glue Protocol に準拠する。
- ステートフル永続 Java クラスを、同等の Oracle Lite クラスにマップする。データベース・スキーマの作成に関する説明は、第 5 章「Java アクセス・クラス（JAC）」の「データベースの管理」を参照してください。

POLPersistentCapable インタフェースの実装

ステートフル永続ルート・クラスでは、POLPersistentCapableStateful インタフェースを実装するか、または POLPersistentObjectStateful クラスを継承する必要があります。（POLPersistentObjectStateful は、POLPersistentCapableStateful インタフェースのデフォルトの実装です。）

POLPersistentCapableStateful インタフェースは POLPersistentCapable クラスを継承したもので、オブジェクト・ハンドルを設定、取得および消去するメソッドが含まれています。オブジェクト・ハンドル POLObjHdl は、Java オブジェクトとこれに対応するデータベース・オブジェクトとのリンクを表します。オブジェクト・ハンドルは次のメソッドで管理されます。

メソッド	説明
setObjHdl	Oracle Lite は、永続オブジェクトをデータベース・オブジェクト・ハンドルに関連付けるためにこのメソッドをコールします。
getObjHdl	POLObjHdl のインスタンスとして表された、対応するデータベース・オブジェクト・ハンドルを返します。
clearObjHdl	永続オブジェクトが一時オブジェクトになるたびにコールされます。clearObjHdl は、永続オブジェクトと対応するデータベース・オブジェクト・ハンドルとのリンクを切断します。

永続ルート・クラスが POLPersistentObjectStateful を継承する場合は、これらのメソッドを実装する必要はありません。

Persistent Glue Protocol

永続オブジェクトが正しく機能するようにするには、Oracle の Persistent Glue Protocol を使用して永続クラスを開発する必要があります。Persistent Glue Protocol には次のような特長があります。

- すべての永続クラスが独自のアクティブ化コンストラクタを実装する。このアクティブ化コンストラクタは、永続オブジェクトが Java 空間に入れられるときにコールされます。
- すべての永続クラスが独自の readObject メソッドと writeObject メソッドを実装する。
- 永続オブジェクトの状態がアクセスされる前に、永続オブジェクトの load メソッドがコールされる。
- 永続オブジェクトの状態が変更される前に、永続オブジェクトに対して prepareForUpdate メソッドがコールされ、オブジェクトの状態が変更された後は dirty メソッドがコールされる。

アクティブ化コンストラクタ

すべての永続クラスでは、次のシグネチャを持つアクティブ化コンストラクタを実装する必要があります。

```
ClassName(oracle.lite.jac.POLObjHdl obj_hdl)
```

クラスで POLPersistentCapableStateful インタフェースを直接実装する場合は、アクティブ化コンストラクタで次のようにオブジェクト・ハンドルを設定する必要があります。

```
setObjHdl(obj_hdl);
```

クラスで POLPersistentCapableStateful インタフェースを直接実装しない場合は、コンストラクタでは、POLObjHdl 引数を渡して単にスーパークラスのコンストラクタをコールするだけで済みます。

次の例では、Person クラスのアクティブ化コンストラクタを示します。このコンストラクタでは、POLPersistentCapableStateful インタフェースを直接実装します。オブジェクト・ハンドルを設定する必要があります。Student クラスは Person を継承します。

```
public class Person implements POLPersistentCapableStateful
{
   ...
   public Person(POLObjHdl obj) { setObjHdl(obj);}
   ...
}

public class Student extends Person
{
   ...
   public Student(POLObjHdl obj)
   {
      super(obj);
   }
   ...
}
```

次の例は、POLPersistentObjectStateful を直接継承する永続クラス Person のアクティブ化コンストラクタを示します。オブジェクト・ハンドルを直接設定する必要はありません。

```
public class Person extends POLPersistentObjectStateful
{
   ...
   public Person(POLObjHdl obj) { }
   ...
}

public class Student extends Person
```

```
{
   ...
   public Student(POLObjHdl obj)
   {
      super(obj);
   }
   ...
}
```

注意： アクティブ化コンストラクタを永続 Java アプリケーションから直接コールしないでください。

readObject

すべての永続 Java クラスでは、クラスで直接的または間接的に POLPersistentObjectStateful を継承する場合でも、POLPersistentCapableStateful に定義されている readObject メソッドを実装する必要があります。

readObject は、データベースから属性値を読み取り、この値を Java クラス内の対応するフィールドに割り当てます。オブジェクト・カーネルは、オブジェクトの状態情報をデータベースから読み取るときに readObject をコールします。load メソッドでも readObject メソッドをコールできます。

readObject を実装するときは、親クラスの readObject メソッドを最初にコールします。次に、DiskObjectReader を介してデータベースから読み取られた値を使用して、各 Java フィールドを初期化します。次にその例を示します。

```
public class Person extends POLPersistentObjectStateful {
   private String name;
   private int age;
   private String address;
   ...
   public void readObject(POLDiskObjectReader reader)
   {
     name = new String(reader.getCharArray());
     age = reader.getInt();
     address = new String(reader.getCharArray());
   }
   ...
}
```

次の Student クラスは Person を継承します。このクラスの readObject メソッドは、最初に親クラスの readObject メソッドをコールしてから、Student により継承された属性を読み取ります。

```
public class Student extends Person {
   private float GPA;
   private Professor advisor;

   ...

   public void readObject(POLDiskObjectReader reader)
   {
     super.readObject(reader);
     GPA = reader.getFloat();
     advisor = (Professor)reader.getObj();
   }
   ...
}
```

注意： readObject メソッドを永続 Java アプリケーションから直接コールしないでください。詳細は、「オブジェクト状態のロード」を参照してください。

---

writeObject

すべての永続 Java クラスは、POLPersistentCapableStateful インタフェースに定義されている writeObject メソッドを実装する必要があります。このメソッドは、オブジェクトの状態情報をデータベースに書き込みます。store メソッドは、writeObject メソッドをコールできます。writeObject を実装するときは、親クラスの writeObject メソッドを最初にコールします。その後で、次のように DiskObjectWriter を使用して、各永続 Java フィールドの値をデータベースに書き込みます。

```
public class Person extends POLPersistentObjectStateful {
   private String name;
   private int age;
   private String address;
   ...
   public void writeObject(POLDiskObjectWriter writer, boolean free)
   {
     writer.setCharArray(name.toCharArray());
     writer.setInt(age);
     writer.setCharArray(address.toCharArray());
     if (free)
     {
```

```
        name = null; address = null;
      }
   }
   ...
}

class Student extends Person {
   private float GPA;
   private Professor advisor;
   ...
   public void writeObject(POLDiskObjectWriter writer, boolean free)
   {
      super.writeObject(writer, free);
      writer.setFloat(GPA);
      writer.setObj(advisor);
      if (free)
         advisor = null;
   }
   ...
}
```

注意： writeObject メソッドを永続 Java アプリケーションから直接コールしないでください。

オブジェクト状態のロード

どのトランザクションの場合でも、オブジェクトの状態値にアクセスする前に、オブジェクトに対して load メソッドをコールする必要があります。例を次に示します。

```
Person p;
...
conn.load(p);
System.out.println(p.name);
```

更新の準備

prepareForUpdate メソッドは、必要であれば最新の状態を永続オブジェクトにロードし、永続オブジェクトの書込み用ロックを取得します。どのトランザクションの場合でも、オブジェクトの状態を変更する前に、オブジェクトに対して prepareForUpdate を少なくとも 1 回コールする必要があります。例を次に示します。

```
Person p;
...
conn.prepareForUpdate(p);
p.age += 1;
```

書込み後のオブジェクトへのダーティのマーク付け

オブジェクトを変更した後は、POLConnection に定義されている dirty メソッドにそのオブジェクトを渡す必要があります。dirty メソッドをコールすることにより、Oracle Lite に対してオブジェクトへの変更を通知します。dirty をコールしない限り、最後に dirty がコールされた後にオブジェクトに加えられた変更は失われます。

```
Person p;
...
conn.prepareForUpdate(p);
p.age += 1;
conn.dirty(p);
```

完全な例

ステートフル永続 Java クラスを手動で実装した完全な例を次に示します。この例では、Oracle Lite スキーマはすでに作成されているものとします。

```
import oracle.lite.jac.*;
public class Person implements POLPersistentCapableStateful
{
   // implements the POLPersistentCapable interface
   private transient POLObjHdl objHdl;  // make it transient
   public void setObjHdl(POLObjHdl obj_hdl)
   { objHdl = obj_hdl ; }
   public POLObjHdl getObjHdl() { return objHdl; }
   public void clearObjHdl() { objHdl = null; }

   private String name;
   private int age;
   private String address;

   // implement the persistent glue protocol
   public Person(POLObjHdl objhdl) { }
   public void readObject(POLDiskObjectReader reader)
   {
      name= reader.getString();
      age = reader.getint();
      address = reader.getString();
   }

   public void writeObject(POLDiskObjectWriter writer, boolean free)
   {
      writer.setString(name);
      writer.setint(age);
      writer.setString(address);
   }
```

```
   // other person methods
   public Person(String name, int age, String address)
   {  this.name = name; this.age= age;
      this.address = address;
      POLConnection.getDefaultConnection().toPersistent(this);
   }

   public String getName()
   { getObjHdl().load(); return name; }

   public void setName(String name)
   {
      getObjHdl().prepareForUpdate(); this.name = name;
      getObjHdl().dirty();
   }

   public int getAge()
   { getObjHdl().load(); return age; }

   public void setAge(int age)
   {
      getObjHdl().prepareForUpdate(); this.age = age;
      getObjHdl().dirty();
   }

   public String getAddress()
   { getObjHdl().load(); return address; }

   public void setAddress(String address)
   {
      getObjHdl().prepareForUpdate(); this.address = address;
      getObjHdl().dirty();
   }
}
```

次の Student クラスは Person を継承します。

```
public class Student extends Person
{
   private float gpa;
   private Person advisor;
     // implement the persistent glue protocol
   public Student(POLObjHdl objhdl) { super(objhdl); }
   public void readObject(POLDiskObjectReader reader)
   {
      super.readObject();
      gpa = reader.getfloat();
      advisor = (Person)reader.getObject();
```

```
    }
    public void writeObject(POLDiskObjectWriter writer, boolean free)
    {
       super.writeObject(writer, free);
       writer.setfloat(gpa);
       writer.setObject(advisor);
       if (free)
          advisor = null;
    }

    // other student method
    public Student(String name, int age, String address,
             float gpa, Person advisor)
    {
       super(name, age, address);
       this.gpa = gpa; this.advisor = advisor;
    }

    public float getGPA()
    {
       getObjHdl().load(); return gpa;
    }

    public void setGPA(float gpa)
    {
       getObjHdl().prepareForUpdate(); this.gpa = gpa;
       getObjHdl().dirty();
    }

    public Person getAdvisor()
    {
       getObjHdl().load(); return advisor;
    }

    public void setAdvisor(Person advisor)
    {
       getObjHdl().prepareForUpdate();
       this.advisor = advisor;
       getObjHdl().dirty();
    }
}
```

A

ストアド・プロシージャのチュートリアル

この付録では、Java ストアド・プロシージャとトリガーの作成方法を示します。説明する内容は、次のとおりです。

- ストアド・プロシージャとトリガーの作成
- トリガーの作成
- コミットまたはロールバック

ストアド・プロシージャとトリガーの作成

このチュートリアルでは、Java クラス EMAIL を作成し、Oracle Lite にロードし、このクラスのメソッドを SQL にパブリッシュし、このメソッドのトリガーを作成します。EMAIL クラスはソース **EMAIL.java** にあります。このファイルは、Java のサンプル・ディレクトリ *Oracle_Home*¥Lite¥Examples¥Java にあります。

EMAIL には assignEMailAddress というメソッドがあります。このメソッドは、従業員のファースト・ネームの最初の 1 文字とラスト・ネームの最大 7 文字に基づいて、その従業員の電子メール・アドレスを生成します。アドレスがすでに割り当てられている場合、このメソッドは、ファースト・ネームとラスト・ネームの文字を組み合せて一意の電子メール・アドレスを見つけようとします。

クラスを作成した後、SQL*Plus を使用してそのクラスを Oracle Lite にロードします。この例では、CREATE JAVA を使用します。かわりに、loadjava ユーティリティを使用してクラスを Oracle Lite にロードすることもできます。クラスをロードした後、assignEMailAddress メソッドを SQL にパブリッシュします。

最後に、T_EMP（従業員情報が含まれている表）に行が挿入されるたびに assignEMailAddress メソッドを起動するトリガーを作成します。

assignEMailAddress は、引数として、JDBC Connection オブジェクト、従業員の ID 番号、ファースト・ネーム、ミドル・イニシャル、ラスト・ネームを取ります。JDBC Connection オブジェクトは、Oracle Lite により提供されます。assignEMailAddress メソッドを実行するときに Connection オブジェクトの値を指定する必要はありません。このメソッドでは、JDBC Connection オブジェクトを使用して、生成された電子メール・アドレスが一意であることを確認します。

SQL*Plus の起動

SQL*Plus を起動し、デフォルトの Oracle Lite データベースに接続します。このチュートリアルの Java アプリケーションでは標準出力に出力するため、SQL*Plus は DOS 版を使用します。DOS プロンプトから次を入力します。

```
plus80 system/mgr@odbc:polite
```

SQL プロンプトが表示されます。

表の作成

表を作成するには、次を入力します。

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

Java クラスの作成

c:¥tmp の **EMAIL.java** ファイルに Java クラス `EMAIL` を作成しコンパイルします。**EMAIL.java** は `assignEMailAddress` メソッドを実装します。このファイルの内容は次のとおりです。このファイルは、*Oracle_Home*¥LITE¥EXAMPLES¥JAVA ディレクトリからコピーできます。

```
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;
       }
    }
```

Java クラスのロード

EMAIL クラスを Oracle Lite にロードするには、次を入力します。

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

クラスをロードした後、クラスに変更を加える場合は、次を行う必要があります。

1. `dropjava` または DROP JAVA CLASS を使用して、データベースからクラスを削除する。
2. 作業をコミットする。
3. SQL*Plus を終了する。
4. SQL*Plus を再起動する。

これで Java 仮想マシンからクラスがアンロードされます。

ストアド・プロシージャのパブリッシュ

ストアド・プロシージャのコール仕様を作成すると、そのストアド・プロシージャは SQL からコールできるようになります。`assignEMailAddress` は値を返さないため、次のように CREATE PROCEDURE コマンドを使用します。

```
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)';
/
```

SQL*Plus を使用する場合は、CREATE PROCEDURE 文に続いてスラッシュを入力する必要があります。

データベースへの値の移入

T_EMP に行を挿入します。

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

プロシージャの実行

プロシージャを実行するには、次のように入力します。

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

電子メールアドレスの検証

ASSIGN_EMAIL プロシージャの結果を表示するには、次のように入力します。

```
SELECT * FROM T_EMP;
```

このコマンドにより、次の出力が生成されます。

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

トリガーの作成

T_EMP に行が挿入されるたびに ASSIGN_EMAIL が実行されるようにするには、T_EMP に AFTER INSERT トリガーを作成します。このトリガーは次のように作成します。

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

SQL*Plus では、CREATE TRIGGER 文の後にピリオドとスラッシュを 1 つずつ別々の行に入力する必要があります。

T_EMP に 1 行挿入されるたびに、EMP_TRIGG というトリガーが起動されます。このプロシージャの実際の引数は、列 `eno`、`fname` および `lname` の値です。

`connection` 引数を指定する必要はありません。

トリガーのテスト

T_EMP に 1 行挿入して、トリガーをテストします。

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

電子メール・アドレスの検証

SELECT 文でトリガーの起動を確認します。

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

コミットまたはロールバック

最後に、変更をコミットして作業を保存するか、またはロールバックして変更を取り消します。

B

JDBC アプリケーションのチュートリアル

このチュートリアルでは、Oracle Lite で JDBC アプリケーションを作成する方法を説明します。説明する内容は、次のとおりです。

- 環境の検査
- アプリケーションへの JDBC ドライバのロード
- Oracle Lite データベースへの接続
- 問合せの発行と結果の参照
- トランザクションのコミット
- 完全な例
- この例のコンパイルと実行

環境の検査

開始する前に、**olite40.jar** が CLASSPATH 内にあることを確認します。CLASSPATH 環境変数には、Oracle Lite のインストール中に **olite40.jar** が設定されます。設定されていない場合は、この変数を追加します。JDK をインストールしていない場合はインストールします。

アプリケーションへの JDBC ドライバのロード

JDBC アプリケーションでは、最初に Oracle Lite JDBC ドライバをロードする必要があります。Oracle Lite JDBC ドライバをアプリケーションにロードするには、次のように `forName` 関数を使用します。

```
Class.forName("oracle.lite.poljdbc.POLJDBCDriver");
```

Oracle Lite データベースへの接続

Oracle Lite データベースに接続するには、次の 2 つの方法があります。

- データベースが JDBC アプリケーションと同じコンピュータにある場合は、ネイティブ・ドライバ接続 URL 構文を使用してデータベースに接続します。
- データベースがサーバー上にあり、クライアント・コンピュータから接続する場合は、タイプ 2 ドライバ接続 URL 構文を使用して接続します。
- どちらのドライバも **olite40.jar** のコンポーネントです。これら 2 つのドライバは、JDBC1.0 規格に準拠しています。

ネイティブ・ドライバ接続 URL 構文

`jdbc:polite[:`*uid*`/`*pwd*`]:`*dsn* `[;`*key*`=`*value*`]*`

次の引数は、DSN 句の一部とすることも、個別のキー値のペアとすることもできます。ドライバに追加情報を提供するキー値のペアは、出現する場合もあれば出現しない場合もあります。DSN で指定できる情報はすべて、キー値のペアとして指定することができます。情報をキー値のペアとして指定すると、DSN で指定した情報は必ず上書きされます。

URL の解析とキー値のペアのオプションは次のとおりです。

引数	説明
`jdbc`	プロトコルを JDBC として識別する。
`polite`	サブプロトコルを `polite` として識別する

引数	説明
uid / pwd	Oracle Lite データベースのオプションのユーザー ID およびパスワード。この引数を指定すると、データ・ソース名（DSN) およびキー値のペアでのユーザー ID およびパスワードの指定が上書きされる。
dsn	**odbc.ini** ファイル内のデータ・ソース名（DSN）のエントリを識別する。このエントリには、サーバーへの接続を完了するために必要な情報がすべて含まれている。
DataDirectory=	.odb ファイルが常駐しているディレクトリ。
Database=	作成時に指定したデータベース名。
IsolationLevel=	トランザクション分離レベル。READ COMMITTED、REPEATABLE READ、SERIALIZABLE または SINGLE USER。分離の詳細は、『Oracle Lite ユーザーズ・ガイド』を参照。
Autocommit=	コミット動作。ON または OFF。
CursorType=	カーソル動作。DYNAMIC、FORWARD ONLY、KEYSET DRIVE または STATIC。カーソル・タイプの詳細は、『Oracle Lite ユーザーズ・ガイド』を参照。
UID=	ユーザー名。
PWD=	パスワード。
key=value	

例

Oracle Lite JDBC ドライバをロードした後は、JDBC アプリケーションから Oracle Lite データベースに接続できます。例を次に示します。

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

try {
   Connection conn = DriverManager.getConnection(ConnectMe)
   }
catch (SQLException e)
{
   ...
}
```

タイプ 2 ドライバ接続 URL 構文

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

ネイティブ・ドライバ接続と同様、タイプ 2 ドライバ接続では、次の引数を、DSN 句の一部として使用することも、ドライバに追加情報を提供するオプションのキー値のペアとして使用することもできます。情報をキー値のペアとして指定すると、DSN で指定した情報は必ず上書きされます。

引数	説明
`jdbc`	プロトコルを JDBC として識別する。
`polite`	サブプロトコルを `polite` として識別する。
UID	ユーザー名。
PWD	パスワード。
host	Oracle Lite データベースのホストとなり、ODBC アダプタ・サービス **olsv2040.exe** が実行されるマシンの名前。このホスト名はオプション。未指定の場合のデフォルト値は、JDBC アプリケーションが実行されるローカル・マシンの名前。
port	ODBC アダプタ・サービスによってリスニングが行われるポート番号。このポート番号はオプションで、未指定の場合のデフォルト値はポート「100」。
dsn	**odbc.ini** ファイル内のデータ・ソース名（DSN）のエントリを識別する。このエントリには、サーバーへの接続を完了するために必要な情報がすべて含まれている。
DataDirectory=	.odb ファイルが常駐しているディレクトリ。
Database=	作成時に指定したデータベース名。
IsolationLevel=	トランザクション分離レベル。READ COMMITTED、REPEATABLE READ、SERIALIZABLE または SINGLE USER。分離の詳細は、『Oracle Lite ユーザーズ・ガイド』を参照。
Autocommit=	コミット動作。ON または OFF。
CursorType=	カーソル動作。DYNAMIC、FORWARD ONLY、KEYSET DRIVE または STATIC。カーソル・タイプの詳細は、『Oracle Lite ユーザーズ・ガイド』を参照。
UID=	ユーザー名。
PWD=	パスワード。

例

次に、このタイプの接続の例を示します。

```
try {
Connection conn = DriverManager.getConnection(
   "jdbc:polite
        :polite@URL_Name ;DataDirectory=c:¥Oracle_Home
        ;Database=polite
        ;IsolationLevel=SINGLE USER
        ;Autocommit=ON
        ;CursorType=DYNAMIC","UID","PWD")
}
catch (SQLException e)
{
   ...
}
```

`getConnection` メソッドの最初の引数は JDBC URL です。

2 番目の引数は、接続先のカタログ名またはユーザー名です。すべての Oracle Lite データベースには、`system` というカタログ名が付いています。

3 番目の引数はパスワードです。データベースが暗号化されている場合は、パスワードを指定する必要があります。

`getConnection` メソッドは try-catch ブロックで囲んで、接続試行中に SQL 例外が発生しないようにします。catch ブロックに例外ハンドリング文を挿入できます。

注意： Oracle8 Navigator を使用してデータベースを作成する場合は、JDBC から接続する前にデータベースのデータ・ソース名を作成する必要はありません。

Oracle8 Navigator は、データベースが接続されるディレクトリ内にデータベースを作成します。

問合せの発行と結果の参照

問合せを発行して結果を参照する前に、表を作成し、その表に新規行を移入する必要があります。

表の作成

JDBC を使用してデータベース内に表を作成するには、まず、Statement オブジェクトを作成します。

```
Statement stmt = conn.createStatement();
```

次に、表を作成する SQL 文とともにこの文を実行します。

```
stmt.executeUpdate("create table person (name char(20), age integer,
     address char(100))");
```

表への新規行の移入

表に新規行を挿入するには、INSERT 文を使用して PreparedStatement オブジェクトを作成します。

```
PreparedStatement pstmt = conn.prepareStatement("insert into person
     values(?,?,?)");
```

疑問符（?）のそれぞれが、値を入れるプレースホルダです。このプレースホルダには、文を実行する前に値をバインドしておく必要があります。

```
pstmt.setString(1, "Kevin Smith");
pstmt.setInt(2, 30);
pstmt.setString(3, "123 Never Land Dr, SF, CA 11111");
pstmt.executeUpdate();
```

表の問合せと結果の参照

表を問い合せるには、SQL 問合せを指定した文を実行します。

```
ResultSet rset = stmt.executeQuery("select * from person");
```

結果を参照するには、次のように指定します。

```
while (rset.next())
{
   System.out.println("Name : " + rset.getString(1)+"¥n");
   System.out.println("Age: " +rset.getInt(2)+"¥n");
   System.out.println("Address: "+rset.getString(3)+"¥n¥n");
}
```

結果セットが不要な場合はクローズします。

```
rset.close();
```

次のように文をクローズします。

```
stmt.close();
```

トランザクションのコミット

トランザクションをコミットするには、commit メソッドを使用します。

```
conn.commit();
```

データベース接続が不要な場合は、次のように指定してアプリケーションから切断できます。

```
conn.close();
```

完全な例

この例は、このチュートリアル全体を示したものです。

```
import java.sql.*;
public class jdbcexample {

   public static void main(String args[]) {
      // to load the driver
      try {
         Class.forName("oracle.lite.poljdbc.POLJDBCDriver");
         // connect to the database
         Connection conn = DriverManager.getConnection("jdbc:polite:polite",
               "system", "");
         // create a table
         Statement stmt = conn.createStatement();
         stmt.executeUpdate("create table person (name char(20), age integer, "+
               " address char(100))");
         // insert new rows into the table
         PreparedStatement pstmt = conn.prepareStatement(
               "insert into person values(?,?,?)");
         pstmt.setString(1, "Kevin Smith");
         pstmt.setInt(2, 30);
         pstmt.setString(3, "123 Never Land Dr, SF, CA 11111");
         pstmt.executeUpdate();
         // query the table
         ResultSet rset = stmt.executeQuery("select * from person");
         // browse the result
         while (rset.next())
         {
            System.out.println("Name : " + rset.getString(1)+"¥n");
```

```
            System.out.println("Age: " +rset.getInt(2)+"¥n");
            System.out.println("Address: "+rset.getString(3)+"¥n¥n");
         }
         rset.close();
         stmt.close();
         // commit the transaction
         conn.commit();
         // disconnect from database
         conn.close();
      }
      catch (SQLException e)
      // exception handling
      {
         System.out.println(e);
      }
      catch (ClassNotFoundException ee)
      {
         System.out.println(ee);
      }
   }
}
```

この例のコンパイルと実行

この例をコンパイルするには、次のように入力します。

```
javac jdbcexample.java
```

この例を実行するには、次のように入力します。

```
java jdbcexample
```

C

JAC アプリケーションのチュートリアル

Java アクセス・クラス（JAC）パッケージ `oracle.lite.jac` は、Oracle Lite データベースに対するオブジェクト指向インタフェースを提供します。このチュートリアルでは、JAC アプリケーションの作成方法を示します。説明する内容は、次のとおりです。

- 環境の検査
- データベースへの接続
- 新規クラスの作成
- 新規オブジェクトの作成
- 既存オブジェクトの問合せ
- オブジェクトの属性値の検査
- トランザクションのコミット
- データベースからの切断
- 完全な例
- この例のコンパイルと実行

環境の検査

開始する前に、変数 **olite40.jar** が CLASSPATH 内にあることを確認します。設定されていない場合は、この変数を追加します。次に、JDK を正しくインストールし構成してあることを確認します。

データベースへの接続

データベースに接続するには、まず、POLConnection オブジェクトのインスタンスを生成する必要があります。静的メソッド getConnection を次の引数を指定してコールします。

- **.odb** 拡張子の付いた完全修飾データベース名（DSN ではない）。たとえば、データベース名が **POLITE.ODB** で ¥ORAWIN¥OLDB40 にある場合は、¥ORAWIN¥OLDB40¥POLITE.ODB と指定します。
- アクセス・モード。これは、通常は CONNECT_READ_WRITE です。

例を次に示します。

```
POLConnection conn  = POLConnection.getConnection("db",
   POLConnection.CONNECT_READ_WRITE);
```

新規クラスの作成

クラスを作成するには、POLAttrDesc オブジェクトの配列を作成する必要があります。この配列はクラス属性を表します。POLAttrDesc コンストラクタへの最初の引数は、大 / 小文字を区別した属性名です。2 番目の引数には、属性のデータ型を指定します。3 番目の引数には、属性を NULL に設定できるかどうかを示します。TRUE の場合、属性は NULL に設定できます。

たとえば、次のように作成された Name、Age および Address 属性が Person クラスに含まれているとします。

```
POLAttrDesc attr_descs[] = new POLAttrDesc[3];
attr_descs[0] = new POLAttrDesc("Name", POLAttrDesc.CHARARRAY_TYPE, true);
attr_descs[1] = new POLAttrDesc("Age", POLAttrDesc.INT_TYPE, true);
attr_descs[2] = new POLAttrDesc("Address", POLAttrDesc.CHARARRAY_TYPE,
   true);
```

これらの属性を定義した後、POLClass に定義されている静的メソッド createNewClass を起動して新しい Oracle Lite クラスを作成します。

最初の引数は、カレント接続を表します。2 番目の引数は、作成するクラスの名前（大 / 小文字区別）です。3 番目の引数はスーパークラス（POLClass オブジェクトの配列）です。4 番目の引数には、新規属性（POLAttrDesc の配列）が含まれます。

```
POLClass person_class = POLClass.createNewClass(conn, "Person", null,
   attr_descs);
```

新規オブジェクトの作成

新規オブジェクトを作成するには、まず、POLClass オブジェクトの createNewObj をコールします。

```
POLPersistentCapable obj1 = person_class.createNewObj();
```

このメソッドは、POLPersistentCapable のインスタンスを返します。

次に、POLClass オブジェクトの findAttr メソッドを（属性名を引数として）コールし、属性オブジェクトを見つけます。

```
POLAttr name_attr = person_class.findAttr("Name");
POLAttr age_attr = person_class.findAttr("Age");
POLAttr address_attr = person_class.findAttr("Address");
```

最後に、POLObjHdl に定義されている適切な setter メソッドを使用して、属性の値を更新します。setter メソッドは、setxVal という名前で、x は属性の型です。属性の型に適した setter メソッドを使用します。

```
obj1.getObjHdl().setCharArrayVal(name_attr, "Kevin Smith".toCharArray());
obj1.getObjHdl().setIntVal(age_attr, 30);
obj1.getObjHdl().setCharArrayVal(address_attr,
   "123 Never Land Dr, SF, CA 11111".toCharArray());
```

POLPersistentCapable インタフェースに定義されている getObjHdl が、POLObjHdl のインスタンスとしてデータベース・オブジェクト・ハンドルを返します。このオブジェクト・ハンドルは、Java オブジェクトとデータベース・オブジェクトをリンクします。

既存オブジェクトの問合せ

データベース内のオブジェクトを問い合せるには、POLIterator を使用します。イテレータを作成するには、POLIterator コンストラクタをコールします。コンストラクタへの最初の引数は POLClass オブジェクトです。これにより、検索するクラスが識別されます。2 番目の引数は、指定されたクラスのサブクラスに属するオブジェクトも検索するかどうかを示すブール型の値です。true は、サブクラスも検索することを意味します。3 番目の引数は検索条件を表します。

一般に、検索条件の構文は次のとおりです。

```
Column_Name  Comparison_Operator Value
```

次の文によりイテレータが作成されます。

```
POLIterator it = new POLIterator(person_class,true, "");
```

オブジェクトの属性値の検査

イテレータを作成した後、イテレータを使用して該当するオブジェクトを取り出します。イテレータの next をコールすると、次に該当するオブジェクトが返されます。取り出すオブジェクトがない場合、このメソッドは null を返します。

```
POLPersistentCapable obj=(POLPersistentCapable)it.next();
```

取り出されたオブジェクトの属性値を検査するには、POLObjHdl に定義されている適切な getter メソッドをコールします。次のように指定します。

```
while (obj != null)  {
   System.out.println("Name: "+ new String(obj.getObjHdl().getCharArrayVal
      (name_attr)));
   System.out.println("Age: "+ obj.getObjHdl().getIntVal(age_attr));
   System.out.println("Address: "+new
   String(obj.getObjHdl().getCharArrayVal(address_attr)));
   obj = (POLPersistentCapable)it.next();
}
```

トランザクションのコミット

トランザクションをコミットするには、POLConnection オブジェクトの commit メソッドをコールします。

```
conn.commit();
```

データベースからの切断

データベースから切断するには、POLConnection オブジェクトの disconnect メソッドをコールします。

```
conn.disconnect();
```

完全な例

このチュートリアルで説明されている完全なアプリケーションを次に示します。

```
import oracle.lite.jac.*;
public class jacexample {
```

```
public static void main(String[] args) {
   try {
   // connect to db
      POLConnection conn  = POLConnection.getConnection("db",
      POLConnection.CONNECT_READ_WRITE);

      // to create a new class
      POLAttrDesc attr_descs[] = new POLAttrDesc[3];
      attr_descs[0] = new POLAttrDesc("Name", POLAttrDesc.CHARARRAY_TYPE,
                      true);
      attr_descs[1] = new POLAttrDesc("Age", POLAttrDesc.INT_TYPE, true);
      attr_descs[2] = new POLAttrDesc("Address", POLAttrDesc.CHARARRAY_TYPE,
            true);

      POLClass person_class = POLClass.createNewClass(conn, "Person", null,
            attr_descs);

      // create some new objects
      POLAttr name_attr = person_class.findAttr("Name");
      POLAttr age_attr = person_class.findAttr("Age");
      POLAttr address_attr = person_class.findAttr("Address");

      POLPersistentCapable obj1 = person_class.createNewObj();

      obj1.getObjHdl().setCharArrayVal(name_attr,
             "Kevin Smith".toCharArray());
      obj1.getObjHdl().setIntVal(age_attr, 30);
      obj1.getObjHdl().setCharArrayVal(address_attr,
            "123 Never Land Dr, SF, CA 11111".toCharArray());

      // query
      POLIterator it = new POLIterator(person_class,true, "");
      POLPersistentCapable obj=(POLPersistentCapable)it.next();

      while (obj != null)  {
         System.out.println("Name: "+ new
               String(obj.getObjHdl().getCharArrayVal(name_attr)));
         System.out.println("Age: "+ obj.getObjHdl().getIntVal(age_attr));
         System.out.println("Address: " +
              newString(obj.getObjHdl().getCharArrayVal(address_attr)));
              obj = (POLPersistentCapable)it.next();
      }

      // to commit a transaction
      conn.commit();

      // to disconnect from database
```

```
            conn.disconnect();
         }
         catch (POLException e)
         {
            System.out.println(e);
         }
      }
   }
```

注意： `getConnection` メソッドの「`db`」を該当するデータベースに置き換えてください。

この例のコンパイルと実行

この例のプログラムをコンパイルするには、次のように入力します。

```
javac jacexample.java
```

この例のプログラムを実行するには、次のように入力します。

```
java jacexample
```

D

サンプル・プログラム

この付録では、Oracle Lite に同梱された Java サンプル・プログラムの使用手順を説明します。説明する内容は、次のとおりです。

- Java サンプル・プログラムの概要
- サンプルの実行

Java サンプル・プログラムの概要

Oracle_Home¥Lite¥Examples¥Java ディレクトリには、Java ストアド・プロシージャ、Java アクセス・クラス、Java レプリケーション・クラスおよび JDBC を Oracle Lite で使用する方法を示すサンプル・プログラムが含まれています。

Java のサンプル用ディレクトリには、次のファイルが含まれています。

1. **Stoproex.sql**
2. **INVENTORY.java**
3. **JACEX.java**
4. **JRCEX.java**
5. **JDBCEX.java**
6. **plsqlex.sql**
7. **PLSQLEX.java**
8. **EMAIL.java**

次の項ではサンプルについて説明します。Java のクラス、メソッドおよびファイルの名前は、大 / 小文字が区別されます。SQL から Java プログラムを実行するときは、大 / 小文字をそのまま保持するために名前は二重引用符で囲む必要があります。

注意： サンプルを JDK 1.1.8 とともに実行すると、すべてのシステムで一貫性のある結果が生成されない場合があります。Symantec の JIT コンパイラを使用すると、この問題が解決される可能性があります。

Java アクセス・クラス（JAC）のサンプル

ファイル **JACEX.java** には、JAC を使用して新製品を作成する Java プログラムのサンプルが含まれています。このプログラムでは、Java ストアド・プロシージャのサンプル・スクリプト **Stoproex.sql** により作成されたスキーマを使用しています。

データ・ソース名 `polite` が含まれている Oracle Lite のサンプル・データベースを使用する場合は、**JACEX.java** を変更する必要はありません。Java ストアド・プロシージャのサンプル用に別のデータベースを使用する場合、または Oracle ホーム・ディレクトリが ORANT でない場合は、プログラムをコンパイルする前に、データベースへのパス ¥*Oracle_Home*¥OLDB40¥POLITE.odb を含む行を変更する必要があります。プログラムは Java コンパイラ（**javac**）を使用してコンパイルし、Java インタプリタ（**java**）を使用して実行します。

Java レプリケーション・クラスのサンプル

ファイル **JRCEX.java** には、Oracle Lite Java レプリケーション・クラスを使用して Oracle7 または Oracle8*i* Server からデータをレプリケートする Java プログラムのサンプルが含まれています。このプログラムを実行する前に、レプリケーション・マスター・サイトと表を設定し、レプリケーション接続を構成する必要があります。詳細は、『Oracle Lite レプリケーション・ガイド』を参照してください。

JDBC のサンプル

ファイル **JDBCEX.java** には、JDBC クラスを使用して PRODUCT 表の行を選択し情報を表示する Java プログラムのサンプルが含まれています。

PL/SQL から Java サンプルへの変換

Oracle の PL/SQL 言語で作成されたストアド・プロシージャとトリガーは Java に変換できます。**PLSQLEX.java** に含まれている Java プログラムのいくつかは、『Oracle Server PL/SQL ユーザーズ・ガイドおよびリファレンス』マニュアルに説明されている PL/SQL プログラムと対応しています。**Plsqlex.sql** には、Java ストアド・プロシージャを起動する SQL 文が含まれています。

Java ストアド・プロシージャのサンプル

Java ストアド・プロシージャのサンプルは、Oracle Lite の表にクラスを手動でアタッチする方法を示します。このかわりに、`loadjava` を使用してクラスを Oracle Lite データベースにロードし、CREATE PROCEDURE 文または CREATE FUNCTION 文を使用してそのクラスのメソッドを SQL にパブリッシュすることもできます。このモデルでは、クラスはデータベース表にはアタッチしません。ストアド・プロシージャを開発するパブリッシュ・モデルの説明は、第 2 章「Java ストアド・プロシージャとトリガー」の「モデル 1: Load and Publish ストアド・プロシージャ開発モデルの使用」を参照してください。

ファイル **Stoproex.sql** には、サンプル・スキーマを作成する SQL 文が含まれています。このスクリプトは、Java サンプルを実行する前に SQL*Plus を使用して実行する必要があります。サンプル・スキーマには、次の 3 つの表が含まれています。

表	説明
PRODUCT	製品に関する情報を格納します。
PRODUCT_COMPOSITION	製品の構成に関する情報を格納します。表の各行は、その製品の組立てに必要な部品の数量を追跡します。
INVENTORY	倉庫にある製品の数量を格納します。

Stoproex.sql には、表に行を挿入する文、INVENTORY 表に Java クラスをアタッチする文および INVENTORY 表の QTY 列に AFTER UPDATE トリガーを作成する文も含まれています。

INVENTORY 表にアタッチされる Java クラスは、ファイル **INVENTORY.java** に定義されています。このファイルには、`SHIP_PRODUCT` という静的メソッドが 1 つと、`SHIP` および `CHECK_INVENTORY` という 2 つの非静的（インスタンス）メソッドが含まれています。

`SHIP_PRODUCT` メソッドは引数を 3 つ取ります。Connection オブジェクト、製品 ID および顧客に出荷される製品の数です。

Stoproex.sql は、次の SQL 文を使用してこのメソッドをコールします。

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

次の点に注意してください。

1. 静的メソッドは、*table_name.method_name* として参照する必要がある。静的メソッドを実行するための FROM 句は、常に疑似表 DUAL を参照する必要があります。
2. SQL は `inventory.ship_product` を大文字に変換する。これは、**INVENTORY.java** ではメソッド名が `SHIP_PRODUCT` だからです。表に「Inventory」、メソッドに「`shipProduct`」という名前を付けた場合は、この両方の名前を二重引用符で囲む必要があります（`"Inventory"."shipProduct"`）。
3. Connection オブジェクトは、メソッドへの引数として明示的には指定されない。型が `java.sql.Connection` の引数には Oracle Lite がカレントの接続を提供します。

Java メソッド `SHIP` では、JDBC クラスを使用して Oracle Lite データベースにアクセスします。引数として渡される接続から文を作成し、SELECT 文を実行します。SELECT 文は、**INVENTORY.java** にも定義されている Java 非静的メソッド `SHIP` を実行します。

メソッド `SHIP` は、出荷する製品数を更新します。`SHIP` が非静的メソッドであるため、Oracle Lite はこのメソッドをコールする前に `INVENTORY` クラスのインスタンスを作成します。ATTACH 文の WITH CONSTRUCTOR ARGS 句に指定されている列を取るコンストラクタを使用して、インスタンスを作成します。

このサンプルでは、INVENTORY 表の QTY 列に対する AFTER UPDATE トリガーが作成されているため、UPDATE のたびに `CHECK_INVENTORY` メソッドが実行されます。`CHECK_INVENTORY` は非静的メソッドであるため、Oracle Lite では行インスタンスを使用するか、（行インスタンスがない場合は）新規インスタンスを作成します。

更新された在庫数が在庫のしきい値を下回った場合は、`CHECK_INVENTORY` メソッドが PRODUCT_COMPOSITION 表を使用して製品の構成部品を参照します。また、各部品の数を更新して、在庫の補充に必要な製品数に見合うように部品数を更新します。この更新は、最終的な数に到達するまで再帰的に発生します。到達した時点で、`CHECK_INVENTORY` により製品の注文が入れられます。

トリガーのサンプル

このサンプルは EMP 表を作成し、`assignEMailAddress` という静的メソッドを持つ `EMAIL` という Java クラスをこの表にアタッチします。このサンプルの詳細は、付録 A「ストアド・プロシージャのチュートリアル」を参照してください。

サンプルの実行

Java サンプルを実行するには、次の手順に従います。

1. サンプルのディレクトリ *Oracle_Home*¥Lite¥Examples¥Java に移動します。例を次に示します。

   ```
   cd ¥ORANT¥LITE¥EXAMPLES¥JAVA
   ```

2. 「.」（カレント・ディレクトリ）が CLASSPATH に含まれていない場合は、追加します。

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

3. DOS 版の SQL*Plus を使用して SQL スクリプトを実行します。例を次に示します。

   ```
   plus80 system/mgr@ODBC:polite @stoproex.sql
   ```

   PLUS80W.exe は SQL*Plus の Windows 版を実行しますが、これでは Java 仮想マシンからの出力が表示されない場合があります。このため、このマニュアルに記載のサンプルでは、すべて SQL*Plus の DOS 版 **PLUS80.exe** が必要になります。

Java アクセス・クラス（JAC）のサンプルの実行

JAC のサンプルを実行するには、次の手順に従います。

1. **Stoproex.sql** スクリプトを実行して POLITE データベースに PRODUCT 表をインストールします。コマンドラインで、次のように入力します。

   ```
   plus80 system/mgr@ODBC:polite @stoproex.sql
   ```

2. 次のコマンドを使用してソース・ファイルをコンパイルします。

   ```
   javac JACEX.java
   ```

3. コンパイルされたクラスを次のように実行します。

   ```
   java JACEX
   ```

表をロードするようにというメッセージが表示されます。

```
To run JACEX example correctly, you need to run "plus80
system/mgr@odbc:polite@STOPROEX.sql" first!
```

次に、PRODUCT 表と PRODUCT_COMPOSITION 表の列が表示されます。これには、表にアタッチされているクラスまたはサブクラスの情報も含まれます。サブクラスやスーパークラスがなくてもエラーではありません。

プログラムは、PRODUCT_COMPOSITION 表の内容（新規行を含む）を表示して終了します。

```
PID = 1000 NAME = Boat DESP = Water-Skiing Boat
```

最後に次のメッセージが表示されます。

```
JAC Example completed successfully!
```

Java レプリケーション・クラスのサンプルの実行

Java レプリケーション・クラスのサンプルを実行するには、（Oracle 7.3.3 以上が稼動する）Oracle マスター・サイト上に次を作成する必要があります。

- REPDEMO というユーザー
- REPDEMO というパスワード
- サーバー・データベースを指す TNS 接続文字列 REPSERVER

ユーザーを作成し、このユーザーがレプリケーション・パッケージに対する実行アクセス権を持つようにするには、Oracle Replication Manager を使用します。

REPDEMO という表をマスター・サイトに作成します。表で高速リフレッシュを想定する場合は、表のスナップショット・ログを作成し、Replication Manager を使用してレプリケーション・マスター・グループにこのログを追加します。

次のコマンドを使用してソース・ファイルをコンパイルします。

```
javac JRCEX.java
```

コンパイルされたクラスを次のように実行します。

```
java JRCEX
```

出力をチェックします。次のようになります。

```
Initializing... snapshotConnectionOpen... masterConnectionOpen...
```

JDBC のサンプルの実行

JDBC のサンプルを使用するには、**Stoproex.sql** スクリプトを実行して Oracle Lite データベースに PRODUCT 表をインストールします。

```
plus80 system/mgr@ODBC:polite @stoproex.sql
```

次のコマンドを使用してソース・ファイルをコンパイルします。

```
javac JDBCEX.java
```

コンパイルされたクラスを次のように実行します。

```
java JDBCEX
```

PL/SQL 変換のサンプルの実行

PLSQLEX.java を実行するには、SQL*Plus を起動します。

```
plus80 system/mgr@odbc:polite
```

SQL*Plus のプロンプトで次のスクリプトを実行します。

```
@plsqlex.sql
```

Java ソース・ファイル **PLSQLEX.java** を表にアタッチします。

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

表のメソッドをコールするには、次のように入力します。

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

結果を表示するには、次のように指定します。

```
select * from temp;
```

その他のサンプルの詳細は、**PLSQLEX.java** を参照してください。サンプルには、`sampleOne` から `sampleFour` という名前が付けられています。

Java ストアド・プロシージャのサンプルの実行

Stoproex.sql スクリプトを実行し、ストアド・プロシージャのサンプルに必要な表とストアド・プロシージャをインストールします。

```
plus80 system/mgr@ODBC:polite @stoproex.sql
```

スクリプトが完了すると、在庫表の内容が表示されます。SQL*Plus のプロンプトで次のように入力します。

```
select * from inventory;

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

表の中の負の数値は、部品 101、102、103 に補充が必要であることを示します。

トリガーのサンプルの実行

電子メール・トリガーのサンプルを実行するには、付録 A「ストアド・プロシージャのチュートリアル」を参照してください。

索引

A

ALTER TABLE 文，2-15，2-17，2-24
ALTER TRIGGER 文，2-24

B

BLOB，3-6，3-11
　値の取得，3-7
　値の設定，3-7

C

CallableStatement クラス，3-5
clearObjHdl，6-14
CLOB，3-6，3-11
　値の取得，3-7
　値の設定，3-7
close メソッド，3-9
Connection オブジェクト、引数に渡される，2-21
CREATE FUNCTION 文，2-9
CREATE JAVA 文，2-8，2-14
CREATE PROCEDURE 文，2-9，2-23
createdb.exe，5-15
createNewDatabase メソッド，5-14
createNewObj メソッド，5-23
createStatement メソッド，3-4

D

deleteDatabase メソッド，5-15
deletedb.exe，5-15
deleteObj メソッド，5-24
Designer，1-6，1-7，1-8
DETACH AND DELETE 文，2-17
DROP FUNCTION 文，2-13
DROP JAVA 文，2-13
DROP PROCEDURE 文，2-13
DROP TRIGGER 文，2-24
dropjava，1-8，2-11 ～ 2-13
　オプション，2-12
　引数，2-12
　ファイル名の指定，2-12

E

executeQuery メソッド，3-4

F

flush メソッド，3-9
force、loadjava オプション，2-5

G

getAsciiOutputStream，3-7
getAsciiStream，3-6
getBinaryOutputStream，3-6
getBinaryStream，3-6
getBLOB，3-8
getBytes，3-6
getCacheSize メソッド，5-5
getCharacterOutputStream，3-7
getCharacterStream，3-6
getChars，3-7
getCLOB，3-8
getConnection，3-6，3-7
getDatabaseID メソッド，5-8
getObjHdl，6-14
getSubString，3-6

I

isConvertibleTo，3-6，3-7

J

JAC，「Java アクセス・クラス」を参照
JAR ファイルのロード，2-6
Java
　Oracle8*i* Lite の機能，1-2
　管理ツール，1-6
　ストアド・プロシージャ，1-3，2-2
　静的メソッド，2-15
　トリガー，1-3
　非静的メソッド，2-15
Java Development Kit（JDK），1-5
Java アクセス・クラス
　JDBC との相互運用性，3-10
　主な機能，5-12
　概要，5-9
　説明，1-3
　属性のデータ型，5-14
　パッケージの説明，5-37
　リレーショナル・データベース，5-30
　例，C-1
Java クラス
　一時と永続，5-10
　永続の作成，5-19
Java 仮想マシン（JVM），1-3，2-3，2-17，2-19，5-4，A-5
JDBC
　JAC との相互運用性，3-10
　拡張，1-2，3-5 ～ 3-10
　サンプル・アプリケーション，B-2
　ストアド・プロシージャ，3-3
　設定，3-2
　説明，1-2，3-1
　標準実装，1-2
JDK（Java Development Kit），1-5
JVM，「Java 仮想マシン」を参照

L

length メソッド，3-6
loadBundle メソッド，5-29
loadClass メソッド，5-29
loadjava，1-8，2-5 ～ 2-7
　オプション，2-5
　構文，2-5
　引数，2-5
　ファイル名の指定，2-6
　例，2-7
loadjvm40.dll，2-19
load メソッド，6-18

M

makeJdbcArray，3-6，3-7
markSupported メソッド，3-9
mark メソッド，3-9

N

NULL 値，5-13

O

ODBC、ストアド・プロシージャのコール，2-19，2-22
Oracle8*i* Lite
　Designer，1-6，1-7，1-8
　Java 機能，1-2
　サンプル・アプリケーション，D-2
　ツール，1-6
　ポストプロセッサ、シームレスな永続，6-2，6-5
OracleCallableStatement クラス，3-8
OraclePreparedStatement クラス，3-7
OracleResultSet クラス，3-8

P

Persistent Glue Protocol，6-14
POLAttr，5-37
POLAttrDesc，5-37
　コンストラクタ，5-3
POLBlob，5-38
POLClass，5-38
POLClassMember，5-38
POLClobReader クラス，3-8
POLClobWriter クラス，3-9
POLConnection，2-21，5-38
POLConnection クラス，5-8
POLDatabase，5-38
POLException，5-38

POLGroup，5-38
POLIndex，5-38
POLIterator，5-38
POLLobInputStream クラス，3-8
POLLobOutputStream クラス，3-8
POLObjHdl，5-39
POLPersistentCapable，5-39
POLPersistentCapable インタフェース，5-20，6-14
POLPersistentObject，5-39
POLSearchCond，5-39
PreparedStatement クラス，3-4
prepareForUpdate，6-18
putBytes，3-6
putChars，3-7
putString，3-7

R

readObject メソッド，6-16
ready メソッド，3-8
read method，3-8
refresh メソッド，6-13
reset メソッド，3-9
ROWID サポート，5-8

S

SELECT 文、ストアド・プロシージャのコール，2-16
setBLOB，3-7
setCLOB，3-7
setIsolationLevel メソッド，5-8
setObjHdl，6-14
SINGLE USER 分離レベル，5-8
skip メソッド，3-9
SQLDescribeCol，2-22
SQLJ
　設定，4-3
　説明，1-4，4-1
SQLNumResultCols，2-22

T

toJdbc メソッド，3-6，3-7

V

verbose、loadjava オプション，2-5

W

writeObject メソッド，6-17
write メソッド，3-8，3-9

Z

ZIP ファイルのロード，2-6

あ

アクセス・モデル，1-4
アクティブ化関数，6-12
アタッチ、クラス，2-15

い

一時クラスと永続クラス，5-10
イテレータ，5-24，6-10

う

埋込み SQL，「SQLJ」を参照

え

永続オブジェクト
　管理，6-9
　削除，5-24
　作成，5-23
　手動による実装，6-13
　属性の更新，5-25，6-11
　属性の取出し，5-25，6-11
　問合せ，5-23，6-10
永続化
　ステートフル，5-12，6-9
　ステートレス，5-12
永続クラスと一時クラス，5-10
永続配列，6-11

お

オブジェクト・ハンドル
　メソッド，5-22
オブジェクト・モデル
　概要，1-4
　継承，5-12

か

ガベージ・コレクション，5-4
環境、設定，1-8

き

行レベルのトリガー，2-2
　文レベルのトリガーとの比較，2-23

く

クラス
　管理ツール，1-6

け

継承
　Oracle Lite と Java，5-12

こ

コール仕様
　作成，2-8，A-5
　サンプル，2-10，2-11，2-14
コール、ストアド・プロシージャ，2-10，2-19
コンストラクタのアクティブ化，6-15

さ

索引
　統計情報，5-7
　命名，5-7
削除
　永続オブジェクト，5-24
削除、ストアド・プロシージャ，2-11，2-12
作成
　永続オブジェクト，5-23
　データベース・スキーマ，5-17

し

順序オブジェクト，5-7
初期化、属性，5-3

す

スキーマ・オブジェクト名，2-6，2-7
スキーマの作成，5-17
ステートフル・オブジェクト，5-12，6-9
　手動による実装，6-13
ステートフルな永続化の説明，5-12
ストアド・プロシージャ，1-3
　SQL へのパブリッシュ，2-8，A-5
　行レベル，2-15
　コール，2-10，2-17，3-5
　削除，1-8，2-11，2-17
　説明，1-3，2-2
　表レベル，2-15
　複数行を返す，2-21
　命名，2-3
　例，2-13，A-1
削除、ストアド・プロシージャ，1-8
スレッド、マルチスレッドからのストアド・プロシージャのコール，2-19

せ

制約，5-26，5-27
セーブポイント，5-17
接続プロパティ，5-8
設定，1-8

そ

相互運用性、JAC と JDBC，3-10
属性
　Java アクセス・クラス，5-14
　永続バイナリの操作，5-25
　更新，5-25，6-11
　初期化，5-3
　取出し，5-25，6-11
　バイト配列とバイナリ，5-13
属性の継承，5-12

た

多次元配列，5-2，5-3，5-4
単純永続オブジェクト，5-2，5-31 ～ 5-37

つ

ツール
　開発，1-8
　クラス管理，1-6

て

ディレクトリ、オブジェクト，5-27
データ型，1-5，2-20
　JAC の属性，5-14
　変換，5-3
データベース
　JAC
　　削除，5-15
　　作成，5-14
　　接続，5-15
　JDBC による接続，3-3
　リレーショナル・アクセス，5-30
データベース・スキーマ、永続オブジェクトの作成，
　5-17

と

問合せ
　JDBC での，3-4
　永続オブジェクト，5-23，6-10
トランザクション，5-16，6-13
　分離レベル，3-4
トランザクションのコミット，5-16
トリガー，1-3
　JAC，5-28 ～ 5-30
　行レベル，2-2
　削除，2-24
　作成，2-23，A-6
　説明，1-3，2-2，2-23
　引数，2-27
　文レベル，2-2
　例，2-24，A-1
トリガーの使用可能，2-24
トリガーの使用禁止，2-24

は

バイト配列とバイナリ属性，5-13
バイナリ属性，5-25
配列、永続属性としての，6-11
パブリッシュ、ストアド・プロシージャ，2-8，A-5
パラメータ、ストアド・プロシージャ，2-21，3-5

ひ

引数、トリガー，2-27

ふ

プロキシ・クラス
　定義，5-9
分離レベル、トランザクション，3-4，5-8，5-16
文レベルのトリガー，2-2
　行レベルのトリガーとの比較，2-23

ほ

ポストプロセッサ、シームレスな永続，6-2，6-5 ～ 6-8

ま

マルチスレッド・プログラムからのストアド・プロ
　シージャのコール，2-19

め

命名、ストアド・プロシージャ，2-3

り

リレーショナル・データベース
　Java アクセス・クラス，5-30
リレーショナル・モデル
　概要，1-4

れ

レプリケーション，5-5

ろ

ロード
　JAR ファイル，2-6
　ZIP ファイル，2-6
　クラス，1-8，2-4，2-5，5-29