

Complément 2

L'approche SQLJ

La technologie SQLJ (norme ISO) permet d'intégrer du code SQL dans un programme Java. Oracle est conforme aux spécifications de la norme. Alors que celle-ci inclut seulement des aspects statiques de SQL, nous verrons en fin de chapitre qu'Oracle dispose d'extensions pour programmer des instructions dynamiques.

Généralités

Les programmes SQLJ sont traduits en classes Java par l'intermédiaire d'un précompilateur (*Oracle SQLJ translator*) : classes qu'il faut compiler avant qu'une machine virtuelle puisse les interpréter.

Blocs SQLJ

Le précompilateur analyse une source d'extension `sqlj` qui est écrit comme une classe Java intégrant des instructions SQL à l'intérieur de blocs – entre accolades et préfixées de `#sql` comme le montre le source `ExempleSQLJ.sqlj` suivant. Par simplicité, nous n'avons précisé ni la connexion à la base ni l'appel à d'autres méthodes indispensables que nous aborderons dans cette section.

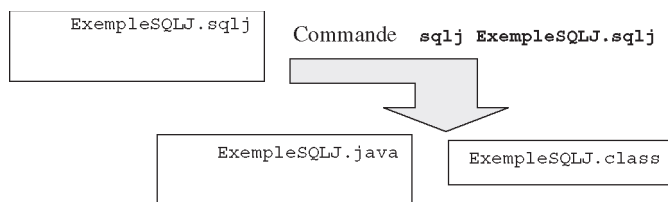
Figure C2-1 Structure générale d'une source SQLJ

```
import java.sql.*;                                ExempleSQLJ.sqlj
import oracle.sqlj.runtime.Oracle;
class ExempleSQLJ
{ public static void main (String args[])
  { try
    { ...
      #sql { DELETE FROM Pilote };
      #sql { INSERT INTO Pilote VALUES ('PL-1', 'Tanguy') };
      ...
    }
    catch (SQLException e) { ... }
  }
}
```

Précompilation

Comme l'illustre la figure suivante, le précompilateur traduit un source SQLJ (en s'appuyant sur la technologie JDBC) en produisant la classe Java (extension `java`) qu'il n'est pas conseillé de modifier, l'exécutable (extension `class`) contenant du code interprétable par les machines virtuelles Java, et, dans certains cas, des fichiers de configuration (extension `ser`) et d'autres classes résultant de la traduction des blocs SQL.

Figure C2-2 Précompilation SQLJ



Configurations

L'environnement de précompilation SQLJ nécessite de devoir configurer un certain nombre de variables et de fichiers.

Variables d'environnement

Il faut s'assurer que la variable `PATH` contient `Oracle_Home/bin` (répertoire d'installation d'Oracle) pour pouvoir invoquer le précompilateur (commande `sqlj`).

La variable `CLASSPATH` doit inclure le paquetage JDBC à utiliser (en fonction du pilote choisi par l'application) :

- `Oracle_Home\sqlj\lib\translator.zip` (ou `.jar`) pour inclure le précompilateur.
- `Oracle_Home\jdbc\lib\classes11.zip` ou `classes12.zip` (les versions `jar` sont aussi disponibles) pour le JDK selon la version (1.1.x ou 1.2.x et 1.3.x), ou `ojdbc14.jar` pour les versions 1.4.x.
- `Oracle_Home\sqlj\lib\runtime11.zip` ou `runtime12.zip` (les versions `jar` sont aussi disponibles) pour utiliser des pilotes JDBC Oracle couplés au JDK versions 1.1.x ou 1.2.x et supérieures.

D'autres bibliothèques peuvent être aussi incluses :

- `Oracle_Home\sqlj\lib\runtime.zip` (ou `.jar`) pour utiliser des pilotes JDBC Oracle plus anciens, indépendamment du JDK.
- `Oracle_Home\sqlj\lib\runtime-nonoracle.zip` (ou `.jar`) pour utiliser des pilotes non Oracle, indépendamment du JDK.

Afin d'initialiser la variable CLASSPATH, écrivez la commande `set classpath=.; C:\oracle\ora92\sqlj\lib\translator.zip;autreChemin;...` dans le fichier `autoexec.bat` ou dans un fichier `.bat` que vous exécuterez à la demande.

Informations de connexions (connect.properties)

En plus des paramétrages des variables d'environnement, il faut configurer SQLJ pour désigner la base de données utilisée par défaut (en l'absence d'un autre contexte). Ces informations relatives à la connexion sont regroupées dans le fichier `connect.properties` (un exemple est situé dans le répertoire `Oracle_Home\sqlj\demo`).

Les lignes précédées du caractère `#` ne sont pas analysées. Dans l'exemple qui suit, on retrouve la chaîne de connexion (qu'Oracle appelle *URL*) étudiée dans la section relative à JDBC. Il faut aussi renseigner les variables `sqlj.url` pour désigner la base, `sqlj.user` et `sqlj.password` pour identifier le schéma cible.

```
#Pilote de type 4 sur une base locale   Fichier connect.properties
sqlj.url=jdbc:oracle:thin:@localhost:1521:BDsoutou
sqlj.user=soutou
sqlj.password=ingres
```

Exécution

Une fois que le source SQLJ a été précompilé, il faut exécuter la classe Java de même nom générée (exemple : `ExempleSQLJ.class`). L'exécution est lancée soit en ligne (`java ExempleSQLJ`), soit à l'aide d'un environnement de développement (dans *JCreator* par le menu `Build/Execute File`).

Test d'une configuration

Une fois que le fichier `connect.properties` est à jour en fonction des paramètres de votre base (port UDP, nom de l'instance, utilisateur, mot de passe), vous pouvez tester votre environnement en utilisant le fichier `TestSQLJ.sqlj`. Cet exemple crée une table, insère un enregistrement, extrait un enregistrement dans un curseur et détruit la table précédemment créée.

D'autres exemples se trouvent dans le répertoire `Oracle_Home\sqlj\demo`, citons :

- `TestInstallSQLJ.sqlj` pour tester le précompilateur et l'environnement. Exécutez la classe pour voir le message `Hello, SQLJ!`
- `TestInstallSQLChecker.sqlj` qui renvoie un message d'erreur à l'exécution. Il convient alors de modifier à la ligne 54, `ITEM_NAMAE` par `ITEM_NAME`. Compilez et exécutez pour voir apparaître `Hello, SQLJ Checker!`

Affectations (SET)

La clause `SET` permet d'affecter une valeur à une variable hôte Java dans un bloc `SQLJ` :

```
#sql { SET : [OUT] variableHôte = expression };
```

L'expression peut être numérique, arithmétique, être un appel d'une fonction, etc. Par défaut la variable hôte est de type OUT, il n'est pas possible de la qualifier par IN ou INOUT.

Le code suivant (`TestSET.sqlj`) décrit deux affectations via des fonctions d'Oracle : la variable `dat` reçoit la date du jour d'exécution du programme, la variable `i` reçoit les sommes de 750 et 250.

Tableau C2-1 Affectation de variables hôtes

Code SQLJ	Commentaires
<pre>int i; java.sql.Date dat;</pre>	Déclarations.
<pre>#sql { SET :dat = sysdate }; #sql { SET :i = TO_NUMBER('750') + TO_NUMBER('250') };</pre>	Affectations.
<pre>System.out.println("dat = " + dat + ", i = " + i);</pre>	dat = 2003-07-24, i = 1000

Intégration de SQL

Cette section décrit l'intégration d'instructions SQL dans un programme SQLJ. Nous verrons ensuite comment prendre en compte les différentes exceptions SQL pouvant survenir au cours de l'exécution du programme. Les instructions SQL sont disposées à l'intérieur de blocs délimités par des accolades et préfixés de la directive `#sql`.

Instructions du LDD

Le code suivant présente des exemples de définition de modification et de suppression d'une table par un programme SQLJ :

Tableau C2-2 LDD dans SOLJ

Code SQLJ	Commentaires
try	
{Oracle.connect(Exemple.class,"connect.properties");	Connexion à la base.
#sql { CREATE TABLE Avion (immat VARCHAR(6), typeAvion VARCHAR(15), cap NUMBER(3), CONSTRAINT pk_Avion PRIMARY KEY(immat)) };	Création d'une table.
#sql { ALTER TABLE Avion ADD compa VARCHAR(4)};	Ajout de la colonne compa.
#sql { DROP TABLE Avion };	Suppression d'une table.
Oracle.close();	Fermeture de la connexion.
} catch(SQLException ex) { ... }	Gestion des erreurs.

Il est aussi possible de créer, modifier ou supprimer d'autres objets (index, vue, séquence, etc.).

Instructions du LMD

Le code suivant présente des exemples de manipulation de données par un programme SQLJ :

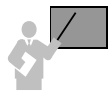
Tableau C2-3 LMD dans SQLJ

Code SQLJ	Commentaires
try { Oracle.connect(Exemple.class, "connect.properties");	Connexion à la base.
#sql { INSERT INTO Avion VALUES ('AV1', 'A340', 248, 'AF') };	Ajout d'un avion.
#sql { UPDATE Avion SET cap = cap + 10 WHERE typeAvion='A340' };	Modification de la colonne cap.
#sql { DELETE FROM Avion WHERE cap < 50 };	Suppression dans Avion.
Oracle.close();	Fermeture de la connexion.
} catch(SQLException ex) { ... }	Gestion des erreurs.

Il est aussi possible d'utiliser des variables hôtes (préfixées de :) au sein d'instructions SQL afin de paramétrer les ordres.

Requêtes

Contrairement aux instructions SQL précédentes, **SELECT** retourne un ou plusieurs enregistrements.



Si un seul enregistrement est retourné (extraction monoligne), il faut utiliser des variables hôtes et la directive **INTO** de l'instruction **SELECT**. Si plusieurs enregistrements sont retournés (extraction multiligne), il faudra utiliser des itérateurs (nom donné aux curseurs SQLJ).

Extraction monoligne

Comme dans le cas de PL/SQL, l'utilisation de la directive **INTO** de l'instruction **SELECT** permet d'extraire des valeurs de la base. Ces valeurs sont insérées dans des variables hôtes (préfixées de :). Il convient de déclarer ces variables Java au préalable en utilisant les conversions de type étudiées au chapitre 9.

La syntaxe générale d'une extraction monoligne avec SQLJ est la suivante :

```
#sql { SELECT col1 [,col2...] INTO :var1 [,:var2 ...]  
      FROM table1 [,table2...] WHERE condition ... };
```

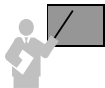
Le code suivant (TestVarHotes.sqlj) présente un exemple d'extraction monoligne :

Tableau C2-4 Extraction monoligne

Code SQLJ	Commentaires
try {... String nc ;	Déclaration de la variable hôte.
#sql { SELECT nomComp INTO :nc FROM Compagnie WHERE comp = 'AF' };	Requête qui charge la variable.
System.out.println("Nom de AF : " + nc);	Affichage du résultat.
...} catch(SQLException ex) { ... }	Gestion des erreurs.

Nature des variables hôtes

Il est possible de préciser la nature d'une variable hôte afin de spécifier si elle peut être utilisée en tant qu'entrée ou en tant que sortie d'un bloc SQLJ.



Une variable hôte peut être déclarée en tant qu'entrée d'un bloc SQLJ (préfixe :IN), en tant que sortie (préfixe :OUT), ou tant qu'entrée ou sortie (préfixe :INOUT).

Il est aussi possible de déclarer, d'une manière similaire, une méthode Java ou une expression combinant variables et méthodes Java.

Le code suivant (TestVarHotes.sqlj) présente un exemple de variables hôtes utilisées en entrée et en sortie :

Tableau C2-5 Nature des variables hôtes

Code SQLJ	Commentaires
try {... String code_comp, nom_comp;	Déclaration des variables hôtes.
#sql { SELECT comp INTO :OUT code_comp FROM Avion WHERE immat = 'F-WTSS' };	Requête qui charge une variable en sortie.
#sql { SELECT nomComp INTO :OUT nom_comp FROM Compagnie WHERE comp = :IN code_comp };	Requête qui charge une variable en sortie en utilisant une variable en entrée.
System.out.println ("Compagnie de F-WTSS : " + nom_comp);	Affichage du résultat.
} catch(SQLException ex) { ... }	Gestion des erreurs.

Extraction multiligne

- La sélection de plusieurs lignes peut être réalisée de différentes manières :
- par des variables hôtes et un curseur Java (instance de la classe `ResultSet` recevant le résultat d'une requête) ;
 - par itérateurs (nom des curseurs dans le vocable SQLJ). On distingue les itérateurs nommés (*named iterator*), pour lesquels on déclare le nom et le type de chaque colonne du curseur résultat, et les itérateurs positionnels (*positional iterator*) pour lesquels seul le type des colonnes du curseur résultat est déclaré.

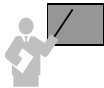
Variables hôtes

Le code suivant (`TestVarHotes2.sqlj`) extrait la liste des compagnies par l'intermédiaire de variables hôtes utilisées en sortie (directive `:OUT`) et d'un curseur Java :

Tableau C2-6 Extraction multiple par variables hôtes

Code SQLJ	Commentaires
<pre>try {... ResultSet rs;</pre>	Déclaration du curseur résultat.
<pre> #sql {BEGIN OPEN :OUT rs FOR SELECT comp, nomComp FROM Compagnie; END; };</pre>	Ouverture et chargement du curseur.
<pre> while(rs.next()) { System.out.print("Numéro: "+rs.getString(1)); System.out.println(" Nom: "+rs.getString(2)); } rs.close(); ...</pre>	Affichage des résultats.
<pre>} catch(SQLException ex) { ... }</pre>	Gestion des erreurs.

Itérateurs nommés



Plusieurs étapes sont nécessaires pour utiliser un itérateur nommé : déclaration du type de l'itérateur, déclaration d'un itérateur, chargement par la requête et fermeture.

```
#sql iterator nomTypeItérateur (type1 colonne1 [,type2 colonne2...]);
nomTypeItérateur nomItérateur;

...

#sql nomItérateur = { SELECT ... };

...

nomItérateur.close();
```

L'extraction d'une colonne d'un itérateur nommé est réalisée via l'instruction `nomCurseur.nomColonne()`. Le type du curseur est considéré comme une classe qu'il est possible d'alimenter en variables curseurs. Ces variables sont chargées par une requête. Le code suivant (`TestVarHotes3.sqlj`) extrait la liste des compagnies par l'intermédiaire d'un itérateur nommé :

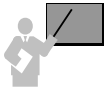
Tableau C2-7 Extraction multiple par un itérateur nommé

Code SQLJ	Commentaires
<pre>try {... #sql iterator CompagnieIter (String comp, String nomComp); CompagnieIter cp_iter; #sql cp_iter = { SELECT comp, nomComp FROM Compagnie }; while(cp_iter.next()) { System.out.print("Numéro: "+cp_iter.comp()); System.out.println(" Nom: "+cp_iter.nomComp()); } cp_iter.close(); ... } catch(SQLException ex) { ... }</pre>	<p>Déclaration du type de l'itérateur nommé.</p> <p>Déclaration d'un itérateur nommé.</p> <p>Chargement de l'itérateur nommé.</p> <p>Affichage des résultats (parcours de l'itérateur nommé).</p> <p>Fermeture de l'itérateur nommé.</p> <p>Gestion des erreurs.</p>



Les colonnes de l'itérateur doivent porter le même nom que les colonnes des tables.

Itérateurs positionnels



Plusieurs étapes sont nécessaires pour utiliser un itérateur positionnel : déclaration du type de l'itérateur, déclaration d'un itérateur, chargement par la requête et fermeture.

```
#sql iterator nomTypeItérateur (type1 [,type2 ...]);
nomTypeItérateur nomItérateur;

...

#sql nomItérateur = { SELECT ... };

...

nomItérateur.close();
```

Comme pour les curseurs PL/SQL, l'instruction `FETCH` permet de charger l'enregistrement courant dans le curseur. La méthode `endFetch` signale la fin du parcours. L'extraction d'une colonne est réalisée via l'instruction `nomCurseur.nomColonne()`.

Le code suivant (TestVarHotes4.sqlj) extrait la liste des compagnies par l'intermédiaire d'un itérateur positionnel :

Tableau C2-8 Extraction multiple par un itérateur positionnel

Code SQLJ	Commentaires
try {... #sql iterator CompagnieIter (String, String);	Déclaration du type de l'itérateur positionnel.
CompagnieIter cp_iter; String numcp = null; String nomcp = null;	Déclaration d'un itérateur positionnel.
#sql cp_iter = { SELECT comp, nomComp FROM Compagnie };	Chargement de l'itérateur positionnel.
while(true){ #sql { FETCH :cp_iter INTO :numcp, :nomcp }; if (cp_iter.endFetch()) break; System.out.print("Numéro: "+numcp); System.out.println(" Nom: "+nomcp);}	Affichage des résultats (parcours de l'itérateur positionnel).
cp_iter.close(); ...	Fermeture de l'itérateur positionnel.
} catch(SQLException ex) { ... }	Gestion des erreurs.

À propos des itérateurs

Cette section résume et complète la description des fonctionnalités des itérateurs.

Méthodes

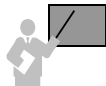
Les interfaces ResultSetIterator et PositionedIterator utilisées lors de la précompilation d'un itérateur (nommé ou positionnel) disposent des méthodes suivantes :

Tableau C2-9 Méthodes pour les itérateurs

Méthodes	Fonctions
close()	Fermeture de l'itérateur.
ResultSet getResultSet()	Extrait le resultset équivalent à l'itérateur au format JDBC. Exemple : #sql iter = { SELECT * FROM Avion }; ResultSet rs = iter.getResultSet();
boolean isClosed()	Détermine l'état de l'itérateur (renvoie true si ouvert).
boolean next()	Déplacement vers le prochain enregistrement de l'itérateur (retourne true si cet enregistrement existe).
boolean endFetch()	Détermine si la fin de l'itérateur positionnel est atteinte.

Navigation dans un itérateur

Oracle répond à la norme SQLJ qui prévoit la possibilité de naviguer dans les itérateurs (*scrollable iterators*). Ce mécanisme est basé sur l'interface `Scrollable` de la spécification JDBC 2.0 (*scrollable result sets*).



Il faut suivre les étapes suivantes pour pouvoir naviguer dans un itérateur : déclaration d'une classe d'itérateurs navigables, création d'un itérateur, chargement par requête et exploitation des diverses méthodes disponibles avant de fermer l'itérateur.

```
#sql public static iterator classeItérateur implements
    sqlj.runtime.Scrollable(type1 colonne1 [,type2 colonne2...]);
classeItérateur iter;

...

#sql iter = { SELECT ... };

... iter.méthodesNavigation();
```

Les principales méthodes relatives à la navigation dans un itérateur sont résumées dans le tableau suivant. Les quatre dernières méthodes retournent *false* si aucun enregistrement n'existe.

Tableau C2-10 Méthodes de navigation dans un itérateur

Méthodes	Fonctions
<code>void setFetchDirection(int)</code>	Précise la direction dans laquelle le parcours se fera. Les valeurs permises sont <code>FETCH_FORWARD</code> (par défaut si une direction a été donnée via l'interface <code>ExecutionContext</code>), <code>FETCH_REVERSE</code> ou <code>FETCH_UNKNOWN</code> .
<code>int getFetchDirection()</code>	Extrait la direction courante.
<code>boolean isBeforeFirst()</code>	Indique si le curseur est positionné avant le premier enregistrement.
<code>boolean isFirst()</code>	Indique si le curseur est positionné sur le premier enregistrement.
<code>boolean isLast()</code>	Indique si le curseur est positionné sur le dernier enregistrement.
<code>boolean isAfterLast()</code>	Indique si le curseur est positionné après le dernier enregistrement.

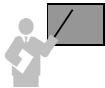
Le code suivant (`IterNavigable.sqlj`) utilise ces méthodes sur un itérateur reflétant l'état de la table `Compagnie`.

Tableau C2-11 Navigation dans un itérateur

Code SQLJ	Commentaires
<pre>import oracle.sqlj.runtime.Oracle; import java.sql.* ; class IterNavigable { #sql public static iterator ItérateurNavi implements sqlj.runtime.Scrollable (String comp, String nomComp);</pre>	Importations. Déclaration de la classe de l'itérateur.
<pre>public static void main (String args[]) {try {Oracle.connect (IterNavigable.class, "connect.properties"); ItérateurNavi compIter; #sql compIter = { SELECT comp, nomComp FROM Compagnie };</pre>	Déclaration et chargement de l'itérateur.
<pre> if (compIter.isBeforeFirst()) System.out.println("Positionné au début"); if (compIter.isFirst()) System.out.println("Positionné sur le 1er déjà");</pre>	Test renvoyant true. Test renvoyant false.
<pre> while(compIter.next()) {if (compIter.isFirst()) System.out.println("1ère compagnie : "); if (compIter.isLast()) System.out.println("Dernière compagnie : "); System.out.print("Numéro: "+compIter.comp()); System.out.println("Nom: "+compIter.nomComp());}</pre>	Parcours de l'itérateur avec affichage du premier et du dernier élément.
<pre> if (compIter.isAfterLast()) System.out.println("Positionné après la fin"); compIter.close(); Oracle.close(); }</pre>	Test renvoyant true. Fermeture de l'itérateur.
<pre> } catch(SQLException ex) { ... }</pre>	Gestion des erreurs.

Sensibilité

Un itérateur peut être déclaré « sensible » (*sensitive*) aux changements de la base durant le temps de son existence. Il fournit ainsi une vue actualisée des données. Ce mécanisme est semblable à celui étudié au chapitre 9.



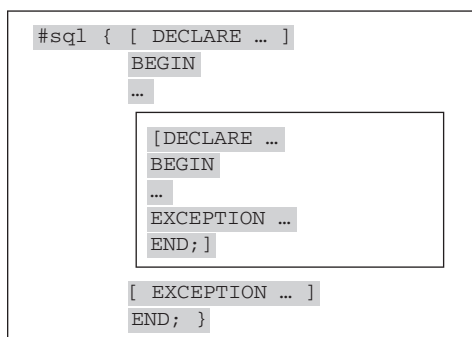
Pour déclarer un itérateur sensible, il faut ajouter une clause à la déclaration d'un itérateur :

```
#sql public static iterator classeItérateur implements
    sqlj.runtime.Scrollable(type1 colonne1 [,type2 colonne2...])
    with (sensitivity=SENSITIVE);
```

Transactions

Au même titre que les instructions SQL, il est possible d'intégrer un programme PL/SQL dans son intégralité dans un bloc SQLJ comme le montre la figure suivante. Des transactions peuvent ainsi être programmées et la gestion des exceptions au sein du bloc est également possible.

Figure C2-3 Programmes PL/SQL sous SQLJ

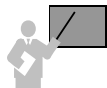


Intégration de blocs PL/SQL

Un simple bloc PL/SQL permet de remplacer la clause SET qui affecte une valeur à une variable hôte Java dans un bloc SQLJ :

```
#sql { BEGIN :OUT variableHôte:= expression; END };
```

Le code suivant (TestBlocPL.sqlj) décrit l'intégration d'une transaction dans un bloc SQLJ. La transaction consiste à insérer la compagnie 'BW' et un Concorde rattaché à cette compagnie. La capacité de cet avion sera diminuée de 10 places par rapport au Concorde de plus grande capacité de la base. Si aucun Concorde n'existe la transaction est annulée. Si d'autres erreurs surviennent il convient de les traiter dans le bloc d'exception de SQLJ (voir plus loin).



Il est possible de valider un ensemble d'instructions en écrivant le bloc « #sql{COMMIT}; » ou au contraire d'invalider la transaction par « #sql{ROLLBACK}; ».

Tableau C2-12 Intégration d'une transaction

Code SQLJ	Commentaires
<pre>try {... #sql { DECLARE v_capacité NUMBER(3); pas_de_Concorde EXCEPTION; BEGIN SELECT MAX(cap) INTO v_capacité FROM Avion WHERE typeAvion = 'Concorde'; IF v_capacité IS NULL THEN RAISE pas_de_Concorde; END IF; v_capacité := v_capacité - 10; INSERT INTO Compagnie VALUES ('BW', 'British Airways'); INSERT INTO Avion VALUES ('F-FGFB', 'Concorde', v_capacité, 'BW'); COMMIT; EXCEPTION WHEN pas_de_Concorde THEN ROLLBACK; END; }; } catch(SQLException ex) { ... }</pre>	<p>Déclarations.</p> <p>Extraction de la plus grande capacité des Concorde (si pas trouvé, annulation).</p> <p>Ajout de deux enregistrements.</p> <p>Validation ou invalidation.</p> <p>Gestion des erreurs.</p>

Points de validation

Comme avec JDBC, la technologie SQLJ d’Oracle intègre la programmation de points de validation (*savepoints*). Le tableau suivant décrit les instructions SQLJ de la norme concernée :

Tableau C2-13 Gestion des points de validation

Code SQLJ	Commentaires
<pre>#sql { SET SAVEPOINT :nomPoint };</pre>	Déclare un point de validation.
<pre>#sql { ROLLBACK TO :nomPoint };</pre>	Invalide la transaction depuis ce point.
<pre>#sql { RELEASE :nomPoint };</pre>	Annule le point de validation.



Oracle ne prend pas encore en charge la fonctionnalité d'annulation d'un point de validation.

Le code suivant (TestSavePoint.sqlj) décrit une transaction partagée en deux phases. L’instruction de non validation permet de valider seulement la première insertion :

Tableau C2-14 Point de validation

Code SQLJ	Commentaires
try {...	
String p1 = "P1";	Variable décrivant le point de validation.
#sql { INSERT INTO Avion VALUES ('F-TOTO', 'A340', 256, 'AF') };	
#sql { SET SAVEPOINT :p1 };	Déclaration du point de validation.
#sql { INSERT INTO Avion VALUES ('F-TITI', 'A340', 256, 'AF') };	
#sql { ROLLBACK TO :p1 };	Annulation de la dernière partie de la transaction.
#sql { COMMIT };	
} catch(SQLException ex) { ... }	Gestion des erreurs.

Appels de sous-programmes

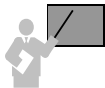
SQLJ permet d’inclure des appels à des procédures ou fonctions stockées qui peuvent être des éléments de paquetages applicatifs. Ces sous-programmes sont en général codés en PL/SQL mais peuvent être écrits depuis la version 8 en Java, C, C++, etc.

Nous distinguons deux cas :

- le sous-programme retourne un résultat scalaire (qui peut être composé d’une ou de plusieurs valeurs). Le cas particulier concerne le sous-programme qui ne retourne aucun résultat ;
- le sous-programme retourne un résultat complexe (traité par un curseur).

Résultats scalaires

Une fonction stockée nécessite une expression SQLJ qui reçoit un résultat et peut inclure des paramètres d’entrée. Une procédure stockée peut inclure des expressions SQLJ en paramètres d’entrée ou de sortie.



La syntaxe pour appeler un sous-programme retournant un résultat scalaire dans un bloc SQLJ est la suivante :

- #sql { CALL nomProcédure(paramètres) } pour une procédure ;

- `#sql variableHôte = { VALUES(nomFonction(paramètres)) }` pour une fonction ;
- L'expression `paramètres` désigne une liste de constantes ou de variables hôtes (`:IN v1, :OUT v2...`) si on désire faire passer des valeurs de variables Java en paramètres d'appel ou de réception.

Fonction

Intégrons dans un bloc SQLJ l'appel de la fonction PL/SQL `LeNomCompagnieEst` (décrite au chapitre 9, section « Appel d'une fonction ») qui retourne le nom de la compagnie d'un avion dont le numéro d'immatriculation est passé en paramètre. Si le numéro d'immatriculation n'est pas référencé dans la base, la valeur `NULL` est retournée. Le code suivant (`TestFonctPL.sqlj`) décrit l'appel de cette fonction par la directive `VALUES` :

Tableau C2-15 Intégration d'une fonction

Code SQLJ	Commentaires
<pre>try {... String immatAvion = "F-WTSS"; String nomCompF_WTSS;</pre>	Déclaration des variables hôtes d'entrée et de sortie.
<pre> #sql nomCompF_WTSS = { VALUES(LeNomCompagnieEst(:IN immatAvion)) }; System.out.println("Résultat : "+nomCompF_WTSS); } catch(SQLException ex) { ... }</pre>	Appel de la fonction (si numéro pas trouvé, null retourné). Affichage du résultat. Gestion des erreurs.

Procédure

Le code suivant (`TestProcPL.sqlj`) intègre l'appel de la procédure PL/SQL `AugmenteCapacité` (décrite au chapitre 9) qui augmente la capacité d'un avion dont le numéro est passé en paramètre. L'appel de la procédure est rendu possible par la directive `CALL`.

Tableau C2-16 Intégration d'une procédure

Code SQLJ	Commentaires
<pre>try {... String immatAvion = "F-WTSS"; int enPlus = 5;</pre>	Déclaration des variables hôtes d'entrée.
<pre> #sql { CALL AugmenteCapacité(:IN immatAvion, :IN enPlus); } catch(SQLException ex) { ... }</pre>	Appel de la procédure. Gestion des erreurs.



Si l'appel d'un sous-programme (fonction ou procédure) ne nécessite pas de paramètre, écrivez l'appel sans parenthèses : `CALL proc` au lieu de `CALL proc()` et `VALUES(fct)` au lieu de `VALUES(fct())`.

Résultats complexes

Les sous-programmes PL/SQL retournent des résultats complexes par l'intermédiaire de variables curseurs (`REF CURSOR`). Ces résultats sont affectés à des itérateurs SQLJ ou des curseurs Java (instances de la classe `ResultSet`).

Considérons la fonction `retourneCompagnies` du paquetage `GestionAvs` qui retourne le code et le nom des compagnies par la variable curseur `résultat` :

Tableau C2-17 Intégration de l'appel d'une fonction

Spécification	Corps
<pre>CREATE PACKAGE GestionAvs AS TYPE Comp_Curtype IS REF CURSOR; FUNCTION retourneCompagnies RETURN Comp_Curtype; END GestionAvs;</pre>	<pre>CREATE PACKAGE BODY GestionAvs AS FUNCTION retourneCompagnies RETURN Comp_Curtype IS résultat Comp_Curtype; BEGIN OPEN résultat FOR SELECT comp, nomComp FROM Compagnie; RETURN résultat; END;</pre>

Le code suivant (`TestProcRefCur.sqlj`) illustre l'appel de cette fonction par l'intermédiaire d'un itérateur nommé :

Tableau C2-18 Intégration de l'appel d'une fonction

Code SQLJ	Commentaires
<pre>#sql public static iterator CompIter (String comp, String nomComp);</pre>	Déclaration de la classe de l'itérateur.
<pre>try {... CompIter curs_comp; #sql curs_comp = { VALUES(GestionAvs.retourneCompagnies) }; while (curs_comp.next()) { System.out.print("Numéro: " + curs_comp.comp()); System.out.println("Non: " + curs_comp.nomComp()); curs_comp.close(); } catch(SQLException ex) { ... }</pre>	Déclaration d'un itérateur. Appel de la fonction. Parcours de l'itérateur.
	Gestion des erreurs.

Traitement des exceptions

Les exceptions qui ne sont pas traitées dans les blocs PL/SQL intégrés, ou celles qui sont volontairement retournées par un sous-programme ou un déclencheur, doivent être prises en compte au niveau du code Java (bloc `SQLException`). Ce bloc d'instructions permet de programmer des traitements en fonction des codes d'erreurs Oracle.

Définition des données

Écrivons le programme SQLJ qui crée une table en gérant la possibilité qu'elle existe déjà par le biais d'une exception (message d'Oracle : `ORA-00955: Ce nom d'objet existe déjà`).

Le code suivant (`ExceptLDD.sqlj`) crée la table `Temp` en gérant l'exception précédente. Dans ce cas la table est détruite puis recréée.

Tableau C2-19 Exception Oracle LDD

Code SQLJ	Commentaires
<pre>import java.sql.*; import oracle.sqlj.runtime.Oracle; class ExceptLDD { public static void main (String args[]) { try { Oracle.connect (ExceptLDD.class, "connect.properties"); #sql { CREATE TABLE Temp(colonne1 VARCHAR2(50)) }; } catch (SQLException e) { if (e.getErrorCode() == 955) {System.out.println("Table déjà existante!"); try { #sql { DROP TABLE Temp}; #sql { CREATE TABLE Temp(colonne1 VARCHAR2(50)) }; } catch (SQLException ex) { System.err.println("Erreur : " + ex);}} else System.err.println("Autre erreur : " + e);} finally { try { Oracle.close(); System.out.println("Table Temp créée"); } catch (SQLException e) {System.err.println("Erreur : " + e); } } }</pre>	<p>Importation des paquetages.</p> <p>Classe principale et méthode <code>main</code> contenant l'instruction LDD.</p> <p>Gestion de l'erreur.</p> <p>Fin du traitement.</p>

Manipulation des données

Le code suivant (ExceptLMD.sqlj) insère un enregistrement dans la table Avion en gérant quelques-unes des exceptions potentielles :

Tableau C2-20 Exceptions Oracle LMD

Code SQLJ	Commentaires
<pre>import java.sql.*; import oracle.sqlj.runtime.Oracle; class ExceptLMD { public static void main (String args[]) { try { Oracle.connect (ExceptLMD.class, "connect.properties"); #sql { INSERT INTO Avion VALUES ('AV1', 'A340', 248, 'AF') }; } catch (SQLException ex) {if (ex.getErrorCode() == 1) System.out.println("Avion déjà existant!"); else if (ex.getErrorCode() == 913) System.out.println("Trop de valeurs!"); else if (ex.getErrorCode() == 942) System.out.println("Nom de table inconnue!"); else if (ex.getErrorCode() == 947) System.out.println("Manque de valeurs!"); else if (ex.getErrorCode() == 1401) System.out.println("Valeur trop longue!"); else if (ex.getErrorCode() == 1438) System.out.println("Valeur trop importante!"); else if (ex.getErrorCode() == 2291) System.out.println("Compagnie inconnue!"); } finally { try { Oracle.close(); System.out.println("Déconnexion OK."); } catch (SQLException e) {System.err.println("Erreur : " + e); } } }</pre>	<p>Importation des paquetages.</p> <p>Classe principale et méthode main contenant l'instruction LMD.</p> <p>Gestion des erreurs potentielles.</p> <p>Fin du traitement.</p>

Interrogation des données

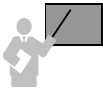
Le code suivant (ExceptLID.sqlj) extrait un enregistrement de la table Avion. Les événements « aucune ligne sélectionnée » (getSQLState renvoie "2000") et « plusieurs lignes extraites » (getSQLState renvoie "21000") sont pris en compte par des exceptions.

Tableau C2-21 Exceptions Oracle LID

Code SQLJ	Commentaires
<pre>import java.sql.*; import oracle.sqlj.runtime.Oracle; class ExceptLID { public static void main (String args[]) { try { Oracle.connect (ExceptLID.class, "connect.properties"); String nc; #sql { SELECT nomComp INTO :nc FROM Compagnie WHERE comp = 'AF'}; System.out.println("Nom de la compagnie : " + nc); catch (SQLException ex) { if (ex.getSQLState() == "2000") System.out.println("Pas de compagnie!"); else if (ex.getSQLState() == "21000") System.out.println ("Trop d'enregistrements renvoyés"); else System.out.println ("Autre erreur : "+ex.getMessage()); } finally { try { Oracle.close(); System.out.println("Déconnexion OK."); } catch (SQLException e) {System.err.println("Erreur : " + e); } } } }</pre>	<p>Importation des paquetages.</p> <p>Classe principale et méthode <code>main</code> contenant la requête et l'affichage du résultat si aucune exception n'est levée.</p> <p>Gestion des erreurs.</p> <p>Fin du traitement.</p>

Contextes de connexion

Le fichier `connect.properties` décrit la connexion par défaut du programme SQLJ. Comme sous JDBC, SQLJ permet de travailler simultanément avec plusieurs contextes de connexion. Ces connexions peuvent concerner différents schémas d'une même base ou divers schémas d'instances distinctes.



Le travail avec un contexte de connexion s'effectue à plusieurs niveaux :

- de la classe de contexte de connexion (`#sql context MaConnexion`);
- de l'instance de la classe de contexte connexion pour chaque connexion (`cx = new MaConnexion (chaîneConnexion, user, password, autoCommit)`);
- du bloc SQLJ qui est précédé du contexte de connexion (`#sql [cx] {...}`). Ici les crochets ne signifient pas une option mais un symbole à utiliser impérativement.

Le code suivant (`TestCONTEXT.sqlj`) travaille avec deux connexions. La connexion `connexScott` extrait les enregistrements de la table `Compagnie`. La connexion courante

concerne un schéma qui contient la table Avion. Les avions de chaque compagnie sont affichés à l’aide de deux boucles imbriquées.

Tableau C2-22 Contextes

Code SQLJ	Commentaires
import java.sql.*; import oracle.sqlj.runtime.Oracle;	Importation des paquetages.
#sql context MaConnexion;	Déclaration de la classe de contexte de connexion.
class TestCONTEXT { public static void main (String args[]) { try { Oracle.connect (TestCONTEXT.class, "connect.properties"); MaConnexion connexScott = new MaConnexion ("jdbc:oracle:thin:@camparols:1521:BDSoutou", "scott", "tiger", false); ResultSet cp; #sql [connexScott] { BEGIN OPEN :OUT cp FOR SELECT comp,nomComp FROM Compagnie; END; };	Classe principale et méthode main. Déclaration de la deuxième connexion.
while (cp.next()) {String codeComp = cp.getString(1); System.out.println ("Avions de la compagnie "+cp.getString(2)); ResultSet av; #sql { BEGIN OPEN :OUT av FOR SELECT immat, typeAvion FROM Avion WHERE comp = :IN codeComp; END; };	Parcours des compagnies.
boolean trouve = false; while (av.next()) { System.out.println ("Immatriculation : "+av.getString(1)+ " Type : "+av.getString(2)); trouve=true; } if (!trouve) System.out.println("Aucun avion"); av.close(); } cp.close(); connexScott.close(); Oracle.close(); }	Requête dans la connexion courante.
catch (SQLException ex) { System.err.println("Erreur : " + ex); } }	Fermeture des connexions.
	Fin du traitement.

SQL dynamique

Des extensions de SQLJ qui permettent de programmer des instructions SQL dynamiques sont proposées par Oracle. Ces aspects ne sont pas présents dans la norme SQLJ. Une instruction SQL dynamique n'est pas prédéfinie dans la source et peut évoluer au cours du programme. Les expressions du SQL dynamique qui sont intégrées au code SQLJ sont dites *meta bind expressions*.

Expression

Une expression SQL dynamique SQLJ contient un ou plusieurs identifiants Java (String) qui seront interprétés durant l'exécution. Une expression peut remplacer :

- le nom d'une table ;
- le nom d'une colonne dans un SELECT ;
- tout ou partie de conditions dans la clause or WHERE ;
- le nom d'un rôle, schéma, paquetage dans une instruction LDD ou LMD ;
- une valeur littérale dans une expression SQL.

Deux écritures sont possibles pour définir une expression SQL dynamique SQLJ. Un bloc peut contenir plusieurs expressions de ces types.

```
: { expressionJavaBind }
: { expressionJavaBind :: codeSQLquiRemplaceExécution }
```

Le code suivant (SQLDynamique.sqlj) illustre les deux écritures possibles d'une expression SQL dynamique. La première réalise une insertion dans une table dont le nom passe en paramètre. La deuxième permet de substituer à l'exécution un paramètre (ici la compilation génère une instruction d'insertion dans la table Avion, l'exécution insérera dans la table Avion2). Ce principe permet de compiler le programme sans que la table Avion2 existe forcément dans le schéma.

Tableau C2-23 Expressions SQLJ pour SQL dynamique

Code SQLJ	Instruction SQL générée à l'exécution
String x = "F-GAFU"; String y = "A380"; int nbPlaces = 345; String nomTable = "Avion"; #sql { INSERT INTO :{nomTable} VALUES (:x, :y, :nbPlaces, 'AF') };	INSERT INTO Avion VALUES ('F-GAFU', 'A380', 345, 'AF')
String nomTable2 = "Avion2"; #sql { INSERT INTO :{nomTable2 :: Avion} VALUES ('F-ENTE', :y, 350, 'AF') };	INSERT INTO Avion2 VALUES ('F-ENTE', 'A380', 350, 'AF')

Restrictions



Une expression SQL dynamique SQLJ ne peut pas :

- être associée à un mode (IN, OUT, ou INOUT) ;
 - être le premier mot-clé d'une instruction SQL dynamique ;
 - inclure de clause INTO dans une requête ;
 - apparaître dans une des clauses suivantes : CALL, VALUES, COMMIT, ROLLBACK, FETCH INTO ou CAST.
-