JDBC de base

Université de Nice - Sophia Antipolis
Richard Grin
Version 3.9.6 – 18/9/12

Plan de cette partie

- Présentation de JDBC
- Drivers JDBC
- Travailler avec JDBC
- Interfaces et classes de JDBC
- Exceptions
- Lancer une requête SQL
- Transactions et exceptions

Présentation

- JDBC (*Java Data Base Connectivity*) est l'API de base pour l'accès à des bases de données relationnelles avec le langage SQL, depuis un programme en Java
- Il est fourni par le paquetage **`java.sql`**
- L'API JDBC est presque totalement indépendante des SGBDRs

Versions de SQL supportées

- Les premières versions de JDBC supportent le standard *SQL-2 Entry Level*
- JDBC 2, 3 et 4 offrent en plus des fonctionnalités de SQL3
- JDBC 4 accompagne Java 6 et ajoute quelques possibilités (nouvelles exceptions, chargement automatique du driver, meilleur support des LOB,…)
- Pour des raisons d'efficacité un driver peut utiliser les possibilités particulières d'un SGBD (permis par JDBC), mais au détriment de la portabilité

Contenu de `java.sql`

- Ce paquetage contient un grand nombre d'interfaces et quelques classes
- Les interfaces constituent l'interface de programmation
- JDBC ne fournit pas les classes qui implantent les interfaces

Drivers JDBC

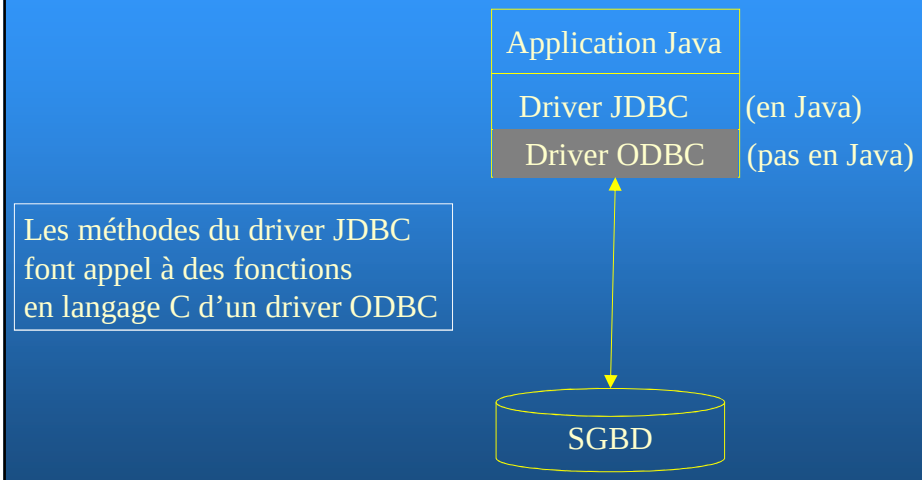
Drivers

- Pour travailler avec un SGBD il faut disposer de classes qui implantent les interfaces de JDBC
- Un ensemble de telles classes est désigné sous le nom de *driver* JDBC
- Les drivers dépendent du SGBD auquel ils permettent d'accéder
- Tous les SGBD importants du marché ont un (et même plusieurs) driver JDBC, fourni par l'éditeur du SGBD ou par des éditeurs de logiciels indépendants

Types de drivers JDBC

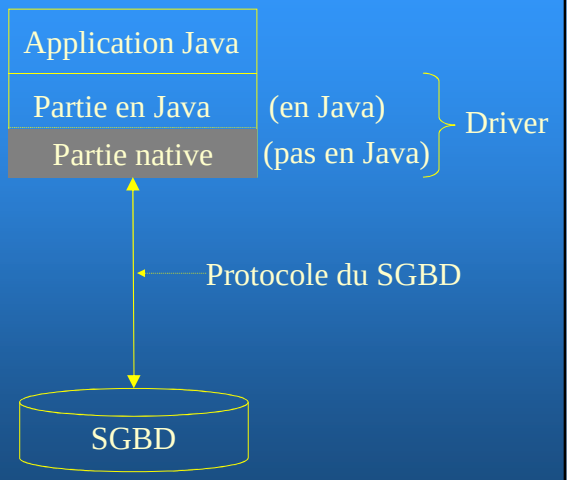
- Type 1 : pont JDBC-ODBC
- Type 2 : driver qui fait appel à des fonctions natives non Java (le plus souvent en langage C) de l'API du SGBD que l'on veut utiliser
- Type 3 : driver qui permet l'utilisation d'un serveur *middleware*
- Type 4 : driver écrit entièrement en Java, qui utilise le protocole réseau du SGBD

Type 1 : pont JDBC-ODBC



Les méthodes du driver JDBC font appel à des fonctions en langage C d'un driver ODBC

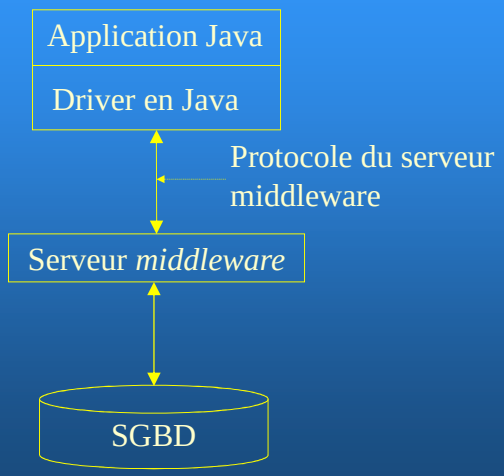
Type 2 : utilise une API native



Les méthodes du driver JDBC font appel à des fonctions d'une API du SGBD écrite dans un autre langage que Java

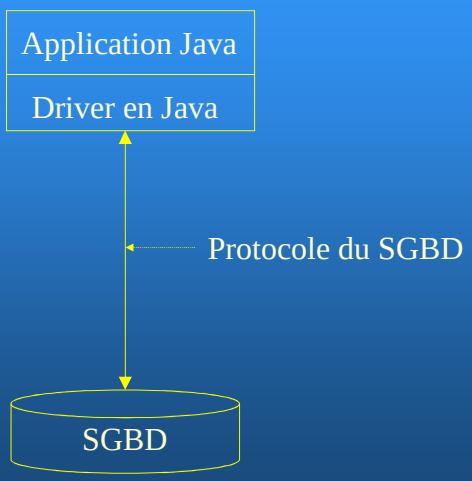
Type 3 : accès à un serveur *middleware*

Les méthodes du driver JDBC se connectent par *socket* au serveur *middleware* et lui envoient les requêtes SQL ; le serveur *middleware* les traitent en se connectant au SGBD



Type 4 : 100 % Java avec accès direct au SGBD

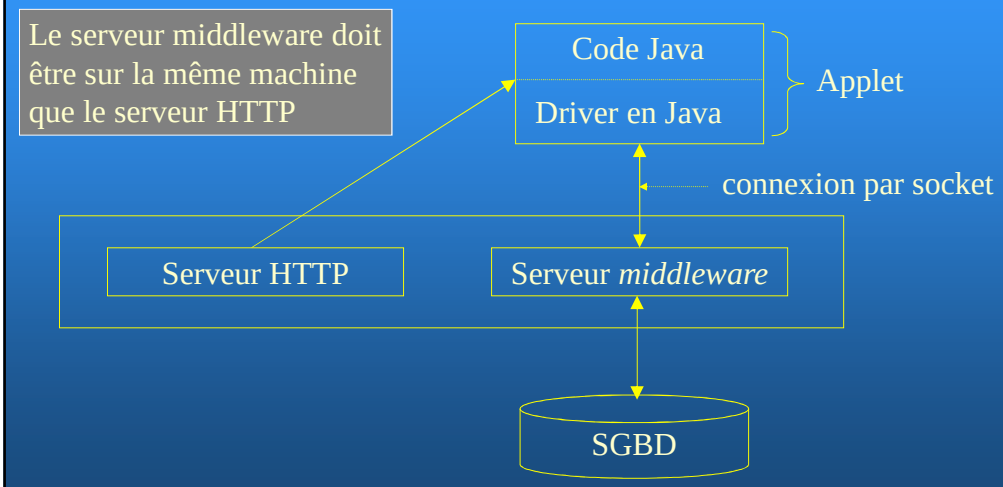
Les méthodes du driver JDBC utilisent des *sockets* pour dialoguer avec le SGBD selon son protocole réseau



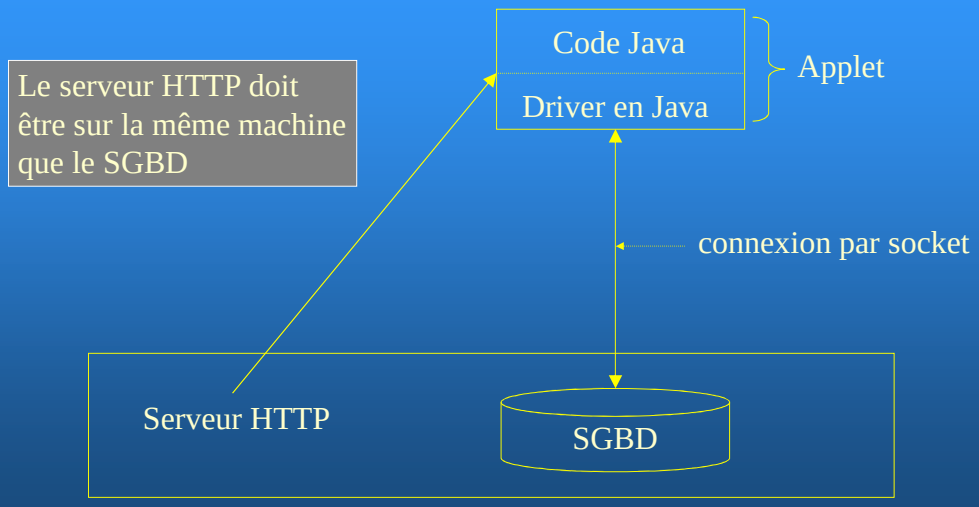
Types de drivers et applet *untrusted*

- Une applet ne peut pas charger à distance du code natif (non Java) ; elle ne peut donc pas utiliser les drivers de type 1 et 2
- Pour des raisons de sécurité, une applet *untrusted* (qui fonctionne dans le bac à sable ; voir cours sur la sécurité) ne peut échanger des données par *sockets* qu'avec la machine d'où elle provient, ce qui implique des contraintes avec les drivers de type 3 et 4

Driver de type 3 et applet *untrusted*



Driver de type 4 et applet *untrusted*



Travailler avec JDBC

Pour utiliser JDBC

- Pour l'exécution, ajouter le chemin des classes du (des) driver dans le *classpath* (option **`-classpath`** de la commande java)
- Par exemple, si Oracle est installé dans **`/oracle`**, le driver peut être dans le fichier **`/oracle/jdbc/lib/ojdbc6.jar`**
  et l'application sera lancée par la commande

```
java –classpath /oracle/jdbc/lib/ojdbc6.jar:… ...
```


Dans les classes qui utilisent JDBC

- Importer le paquetage **`java.sql`** :

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

- Charger en mémoire la classe du (des) driver (driver de type 4 fourni par Oracle pour cet exemple) avant d'utiliser JDBC :

```
Class.forName("oracle.jdbc.OracleDriver");
```

  Cette étape est inutile avec JDBC 4 (donc à partir du JDK 6) ; voir le transparent « JDBC 4 et le driver » dans la section suivante

Étapes du travail avec une base de données avec JDBC

1. Ouvrir une connexion (**`Connection`**)
2. Créer des instructions SQL (**`Statement`**, **`PreparedStatement`** ou **`CallableStatement`**)
3. Lancer l'exécution de ces instructions :
   - interroger la base (**`executeQuery`**)
   - ou modifier la base (**`executeUpdate`**)
   - ou tout autre ordre SQL (**`execute`**)

(4. Valider ou non la transaction avec **`commit`** ou **`rollback`**)

5. Fermer la connexion (**`close()`**)

Remarque importante

- Comme toutes les ressources qui doivent absolument être fermées, les connexions doivent être fermées dans un bloc **`finally`**
- Pour ne pas alourdir les exemples à venir, ceux-ci ne comportent pas cette fermeture
- Le schéma global de code, avec fermeture des ressources et validation/invalidation des transactions, est donné dans la section finale « Transactions et exceptions »

Classes et interfaces de JDBC

- Nous étudierons tout d'abord les classes et méthodes de base de JDBC
- Des nouvelles possibilités de JDBC 2, 3 et 4, en particulier celles qui sont liées à SQL3, seront abordées dans les parties « JDBC avancé » et « JDBC et objet-relationnel »

Avertissement

- Dans la suite du cours on utilise des raccourcis tels que « instance de **`Connection`** »
- Comme **`Connection`** est une interface, il faut traduire par « instance d'une classe qui implémente **`Connection`** »

Interfaces principales

- Driver : renvoie une instance de **`Connection`**
- Connection : connexion à une base
- Statement : ordre SQL
- PreparedStatement : ordre SQL paramétré
- CallableStatement : procédure stockée sur le SGBD
- ResultSet : lignes récupérées par un ordre SELECT
- ResultSetMetaData : description des lignes récupérées par un SELECT
- DatabaseMetaData : informations sur la base de données

Classes principales

- DriverManager : gère les drivers, lance les connexions aux bases
- Date : date SQL
- Time : heures, minutes, secondes SQL
- TimeStamp : date et heure, avec une précision à la microseconde
- Types : constantes pour désigner les types SQL (pour les conversions avec les types Java)

Interface `Driver`

- La méthode **`connect`** de **`Driver`** prend en paramètre un URL et divers informations (nom de login et mot de passe,…) et renvoie une instance de l'interface **`Connection`**
- Cette instance de **`Connection`** permettra de lancer des requêtes vers le SGBD
- **`connect`** renvoie **`null`** si le driver ne convient pas pour se connecter à la base désignée par l'URL
- Utilisée par **`DriverManager`** ; pas visible par l'utilisateur de l'API

URL d'une base de données

- Un URL pour une base de données est de la forme :
  jdbc:*sous-protocole*:*base de donnée*
- Par exemple, pour Oracle :
  jdbc:oracle:thin:@sirocco.unice.fr:1521:INFO
  - oracle:thin est le sous-protocole (driver « *thin* » ; Oracle fournit aussi un autre type de driver)
  - @sirocco.unice.fr:1521:INFO désigne la base de données INFO située sur la machine sirocco (le serveur du SGBD écoute sur le port 1521)
- La forme exacte des parties *sous-protocole* et *base de données* dépend du SGBD cible

Gestionnaire de drivers

- La classe **`DriverManager`** gère les drivers (instances de **`Driver`**) disponibles pour les différents SGBD utilisés par le programme Java
- Pour qu'un driver soit utilisable, sa classe doit être chargée en mémoire (inutile avec JDBC 4 ; voir transparent suivant) :

```
Class.forName("oracle.jdbc.OracleDriver");
```

- La classe crée alors une instance d'elle même et enregistre cette instance auprès de la classe **`DriverManager`**

JDBC 4 et le driver (1/2)

- Afin de charger un driver, JDBC 4 utilise le mécanisme Java pour les fournisseurs de service
- Tous les fichiers jar du classpath sont parcourus par le système d'exécution Java pour voir s'ils contiennent un répertoire **`META-INF/services`**
- Si le fichier jar contient le code pour fournir un service, ce répertoire contient un fichier avec le nom du service
- Le nom du service qui correspond à un driver JDBC est **`java.sql.Driver`**

JDBC 4 et le driver (2/2)

- Le fichier **`META-INF/services/java.sql.Driver`** du jar qui contient le driver JDBC doit contenir une ligne avec le nom de la classe qui implémente l'interface Driver
- Par exemple, ce fichier contiendra la ligne **`oracle.jdbc.OracleDriver`**
- A partir du JDK 6 il est donc inutile d'appeler la méthode **`Class.forName`** pour les drivers qui sont compatibles avec JDBC 4 (c'est pratiquement le cas pour tous les drivers)

Obtenir une connexion

- Pour obtenir une connexion à un SGBD, on demande cette connexion à la classe gestionnaire de drivers :

```
String url =
 "jdbc:oracle:thin:@sirocco.unice.fr:1521:INFO";
Connection conn =
  DriverManager.getConnection(url,
                              "toto", "mdp");
```

- La classe **`DriverManager`** s'adresse à tour de rôle à tous les drivers qui se sont enregistrés (méthode **`connect`**), jusqu'à ce qu'un driver lui fournisse une connexion (ne renvoie pas **`null`**)

Connexions et threads

- Les connexions sont des ressources coûteuses, et surtout longues à obtenir
- Attention, une instance de **`Connection`** ne peut pas être partagée par plusieurs *threads* car elle n'est pas protégée contre les accès concurrents
- Il faut plutôt utiliser les pools de connexions fournis avec les « sources de données » (étudiées dans une autre partie du cours)

Configuration d'une connexion

- Depuis JDBC 4 la classe **`Connection`** fournit les méthodes **`setClientInfo`** qui permettent de configurer une connexion
- Chaque driver JDBC a son propre type de configuration ; consulter sa documentation
- **`getClientInfo()`** fournit une instance de **`Properties`** qui donne les propriétés configurables de la connexion

Validité d'une connexion

- Depuis Java 6, on peut vérifier qu'une connexion est encore valide en lui envoyant la méthode **`isValid()`**
- Le driver s'arrange pour voir si la connexion fonctionne encore
- Il est possible de passer à la méthode un délai maximum en secondes au-delà duquel la méthode doit retourner (elle renvoie false si le délai est dépassé)

Transactions

- Par défaut la connexion est en « *auto-commit* » : un commit est automatiquement lancé après chaque ordre SQL qui modifie la base
- Le plus souvent il faut enlever l'auto-commit : **`conn.setAutoCommit(false)`**
- Il faut alors explicitement valider ou annuler la transaction par
  - **`conn.commit()`**
  - **`conn.rollback()`**

Niveau d'isolation

- Le niveau d'isolation d'une transaction peut être modifié :

```
conn.setTransactionIsolation(
  Connection.TRANSACTION_SERIALIZABLE);
```

Exceptions

SQLException

- SQLException : erreurs SQL ; exception contrôlée qui est la racine des exceptions et des « warnings » (avertissements) liés à SQL

Exceptions de JDBC 2

- Le cours « JDBC avancé » introduira des objets qui peuvent générer des exceptions
- BatchUpdateException : lié aux erreurs qui se produisent pendant l'exécution de groupement de requêtes (méthode **`executeBatch`** de **`Statement`**) ; donne des informations sur les requêtes qui ont eu des problèmes
- SerialException : lié à l'utilisation de BLOB, CLOB
- SyncFactoryException, SyncProviderException : liés à l'utilisation des RowSets déconnectés

Exceptions de JDBC 4 (1)

- Pour distinguer des types d'exception, JDBC 4 a introduit 3 sous-classes de **`SQLException`**
  - SQLNonTransientException : le problème ne peut être résolu sans une action externe ; inutile de réessayer la même action sans rien faire de spécial
  - SQLTransientException : le problème peut avoir été résolu si on attend un peu avant d'essayer à nouveau
  - SQLRecoverableException : l'application peut résoudre le problème en exécutant une certaine action mais elle devra fermer la connexion actuelle et en ouvrir une nouvelle

Exceptions de JDBC 4 (2)

- Classes filles de **`SQLNonTransientException`** : **`SQLDataException`**, **`SQLFeatureNotSupportedException`**, **`SQLIntegrityConstraintViolationException`**, **`SQLInvalidAuthorizationException`**, **`SQLNonTransientConnectionException`**, **`SQLSyntaxErrorException`**
- Classes filles de **`SQLTransientException`** : **`SQLTimeoutException`**, **`SQLTransactionRollbackException`**, **`SQLTransientConnectionException`**

Exceptions de JDBC 4 (3)

- SQLClientInfoException : lancé lorsqu'une connexion à une base de données ne supporte pas certaines configuration données par les méthodes **`setClientInfo`** de la classe **`Connection`**

Chaînage des exceptions

- Une requête SQL peut provoquer plusieurs exceptions
- On peut obtenir la prochaine exception par la méthode **`getNextException()`**
- Une exception peut avoir une cause ; on l'obtient par la méthode **`getCause()`**
- Toutes ces exceptions peuvent être parcourues par une boucle « for-each » :

```
catch(SQLException ex) {
  for (Throwable e : ex) { … }
```


Warnings

- Certaines méthodes peuvent provoquer l'attachement d'un avertissement à l'instance qui a reçu le message
- Par exemple, des **`SQLWarning`** peuvent être chaînés aux **`Statement`**, **`Connection`** ou **`ResultSet`**
- Le 1er avertissement est retrouvé avec la méthode **`getWarnings()`** de la classe **`Statement`** (ou autre classe) ; les suivants se retrouvent par la méthode **`getNextWarning()`** de la classe **`SQLWarning`**
- La levée d'un warning n'interrompt pas l'exécution comme la levée d'une exception

Quelques *warnings*

- SQLWarning : avertissements SQL (classe fille de SQLException) ; le mécanisme de récupération des avertissements est étudié plus loin
- DataTruncation : avertit quand une valeur est tronquée lors d'un transfert entre Java et le SGBD (classe fille de SQLWarning)
- RowSetWarning : ils sont attachés aux rowSets (étudiés dans le cours « JDBC avancé »)

Lancer une requête SQL

Instruction SQL simple

- Instance de l'interface Statement
- La création est effectuée par la méthode createStatement() de Connection :

```
Statement stmt = connexion.createStatement();
```

- Lorsqu'un **`Statement`** a été créé, il peut être utilisé pour lancer plusieurs requêtes différentes (**`executeQuery`** ou **`executeUpdate`**), après avoir traité complètement la requête précédente (car, par exemple un **`ResultSet`** d'une précédente requête est automatiquement fermé au lancement d'une nouvelle requête)

Exécution de l'instruction SQL simple

- La méthode à appeler dépend de la nature de l'ordre SQL que l'on veut exécuter :
  - consultation (select) : executeQuery renvoie un ResultSet pour récupérer les lignes une à une
  - modification des données (update, insert, delete) ou ordres DDL (create table,…) : executeUpdate renvoie le nombre de lignes modifiées
  - si on ne connaît pas à l'exécution la nature de l'ordre SQL à exécuter ou si l'ordre peut renvoyer plusieurs résultats : execute (voir transparent « Ordre SQL quelconque » plus loin sur ce support)

Consultation des données (SELECT)

```
Statement stmt = conn.createStatement();
// rset contient les lignes renvoyées
ResultSet rset =
  stmt.executeQuery("SELECT nomE FROM emp");
// Récupère chaque ligne une à une
while (rset.next())
  System.out.println (rset.getString(1));
  // ou . . . (rset.getString("nomE"));
stmt.close();
```

La première colonne a le numéro 1

Voir le schéma global de code pour la fermeture des connexions dans la section « Transactions et exceptions »

Interface `ResultSet`

- **`executeQuery()`** renvoie une instance de **`ResultSet`**
- **`ResultSet`** va permettre de parcourir toutes les lignes renvoyées par le **`select`**
- Au début, **`ResultSet`** est positionné avant la première ligne et il faut donc commencer par le faire avancer à la première ligne en appelant la méthode **`next()`**
- Cette méthode permet de passer à la ligne suivante ; elle renvoie **`true`** si cette ligne suivante existe et **`false`** sinon

Interface **`ResultSet`**

- Quand **`ResultSet`** est positionné sur une ligne les méthodes getXXX permettent de récupérer les valeurs des colonnes de la ligne :
  - getXXX(int *numéroColonne*)
  - getXXX(String *nomColonne*) (nom simple d'une colonne, pas préfixé par un nom de table ; dans le cas d'une jointure utiliser un alias de colonne)
- XXX désigne le type Java de la valeur que l'on va récupérer, par exemple **`String`**, **`Int`** ou **`Double`**
- Par exemple, **`getInt`** renvoie un **`int`**

`ResultSet` - performances

- Quand le réseau est lent et que l'on veut récupérer de nombreuses lignes, il est parfois possible d'améliorer sensiblement les performances en modifiant le nombre de lignes récupérées à chaque fois par le ResultSet (il faut effectuer des tests pour chaque cas)
- Pour cela, on utilise la méthode **`setFetchSize`** de **`Statement`**
- C'est seulement une indication qu'on donne au driver ; il n'est pas obligé d'en tenir compte

Types JDBC/SQL

- Tous les SGBD n'ont pas les mêmes types SQL ; même les types de base peuvent présenter des différences importantes
- Pour cacher ces différences, JDBC définit ses propres types SQL dans la classe **`Types`**, sous forme de constantes nommées
- Ils sont utilisés par les programmeurs quand ils doivent préciser un type SQL (**`setNull`**, **`setObject`**, **`registerOutParameter`**)
- Le driver JDBC fait la traduction de ces types dans les types du SGBD

Types JDBC/SQL (classe **`Types`**)

- CHAR, VARCHAR, LONGVARCHAR
- BINARY, VARBINARY, LONGVARBINARY
- BIT, TINYINT, SMALLINT, INTEGER, BIGINT
- REAL, DOUBLE, FLOAT
- DECIMAL, NUMERIC
- DATE, TIME, TIMESTAMP
- BLOB, CLOB
- ARRAY, DISTINCT, STRUCT, REF
- JAVA_OBJECT

Types SQL3 étudiés dans un autre support du cours

Correspondances entre types Java et SQL

- Il reste le problème de la correspondance entre les types Java et les types SQL
- Dans un programme JDBC, les méthodes getXXX, setXXX servent à préciser cette correspondance
- Par exemple, **`getString`** indique que l'on veut récupérer la donnée SQL dans une String
- C'est le rôle du driver particulier à chaque SGBD de faire les traductions correspondantes ; une exception peut être lancée si ça n'est pas possible

Correspondances avec getXXX()

- On a une grande latitude ; ainsi, presque tous les types SQL peuvent être retrouvés par getString()
- Cependant, des méthodes sont recommandées ; voici des exemples :
  - CHAR et VARCHAR : getString, LONGVARCHAR : getAsciiStream et getCharacterStream
  - BINARY et VARBINARY : getBytes, LONGVARBINARY : getBinaryStream
  - REAL : getFloat, DOUBLE et FLOAT : getDouble
  - DECIMAL et NUMERIC : getBigDecimal
  - DATE : getDate, TIME : getTime, TIMESTAMP : getTimestamp

Types Date en Java et en SQL

- **`java.sql`** contient une classe **`Date`** qui est utilisé par JDBC pour les échanges de dates entre Java et la base de données
- Cette classe hérite de la classe **`java.util.Date`**
- Elle correspond à un temps en millisecondes
- Normalement les dates SQL ne contiennent pas d'indication sur l'heure dans la journée ; il faut utiliser les types SQL TIME et TIMESTAMP pour l'heure dans la journée
- Pour passer de **`java.util.Date`** à **`java.sql.Date`**, utiliser la méthode **`getTime()`**

Exemple

- Pour passer de **`java.util.Date`** à **`java.sql.Date`**, utiliser la méthode **`getTime()`**:
-
```
java.util.Date date = new java.util.Date();
java.sql.Date dateSQL =
  new java.sql.Date(date.getTime());
java.sql.Time time =
  new Time(date.getTime());
java.sql.Timestamp time =
  new Timestamp(date.getTime());
```

Manipulation des dates

- Un petit rappel sur les dates en Java :
  - mise en forme avec la classe **`java.text.DateFormat`**
  - calculs sur les dates avec la classe **`java.util.Calendar`**
- Voir le cours sur les dates dans le support « Compléments divers »

Valeur `NULL`

```
Statement stmt = conn.createStatement();
ResultSet rset = stmt.executeQuery(
    "SELECT nomE, comm FROM emp");
while (rset.next()) {
  nom = rset.getString("nomE");
  commission = rset.getDouble("comm");
  if (rset.wasNull())
    System.out.println(nom + ": pas de comm");
  else
    System.out.println(nom + " a " + commission
                       + " € de commission");
}
```

Modification des données (INSERT, UPDATE, DELETE)

```
Statement stmt = conn.createStatement();
String ville = "NICE";
int nbLignesModifiees = stmt.executeUpdate(
    "INSERT INTO dept (dept, nomD, lieu) "
    + "VALUES (70, 'DIRECTION',"
    + "'" + ville + "')");
```

N'oubliez pas l'espace !

Instruction SQL paramétrée

- La plupart des SGBD (dont Oracle) peuvent n'analyser qu'une seule fois une requête exécutée un grand nombre de fois durant une connexion
- JDBC permet de profiter de ce type de fonctionnalité par l'utilisation de requêtes paramétrées
- Les requêtes paramétrées sont associées aux instances de l'interface PreparedStatement qui hérite de l'interface Statement

Création d'une requête paramétrée

```
PreparedStatement pstmt =
  conn.prepareStatement("UPDATE emp SET sal = ?"
                        + " WHERE nome = ?");
```

- Les "?" indiquent les emplacements des paramètres
- Cette requête pourra être exécutée avec plusieurs couples de valeurs : (2500, 'DUPOND'), (3000, 'DURAND'), etc.

Requête paramétrée – Valeurs des paramètres

- Les valeurs des paramètres sont données par les méthodes setXXX(n, valeur) (setDouble, setString,…)
- On choisit la méthode setXXX suivant le type Java de la valeur que l'on veut mettre dans la base de données
- C'est au programmeur de passer une valeur Java du bon type à la méthode setXXX
- Le driver JDBC fait la conversion dans le bon format pour le SGBD

Requête paramétrée - Exemple

```
PreparedStatement pstmt =
  conn.prepareStatement(
    "UPDATE emp SET sal = ? "
    + "WHERE nomE = ?");
for (int i=0; i<10; i++) {
  pstmt.setDouble(1,employe[i].getSalaire());
  pstmt.setString(2, employe[i].getNom());
  pstmt.executeUpdate();
}
```

commence à 1 et pas à 0

Requête paramétrée - **NULL**

- Pour passer la valeur NULL à la base de donnée, on peut
  - utiliser la méthode setNull(n, type) (type de la classe **Types**)
  - ou passer la valeur Java **null** si la méthode setXXX() attend un objet en paramètre (**setString**, **setDate**,…)

Avantages des `PreparedStatement`

- Leur traitement est plus rapide s'ils sont utilisés plusieurs fois avec plusieurs paramètres
- Ils améliorent aussi la portabilité car les méthodes setXXX gèrent les différences entre SGBD
- En effet, les SGBD n'utilisent pas tous les mêmes formats de date (`'JJ/MM/AA'` ou `'AAAA-MM-JJ'` par exemple) ou de chaînes de caractères (pour les caractères d'« échappement »)
- Mais on peut aussi utiliser pour cela la syntaxe (un peu lourde) « SQL Escape » (voir plus loin)
- Ils évitent l'injection de code SQL

Procédures stockées

- Les procédures stockées permettent non seulement de précompiler des ordres SQL mais aussi de les regrouper
- Comme les accès réseau aux bases de données ralentissent les applications, les procédures stockées permettent souvent d'améliorer les performances
- Mais elles nuisent aussi souvent à la portabilité des applications

Exemple de procédure stockée (Oracle)

```
create or replace procedure augmenter
  (unDept in integer, pourcentage in number,
   cout out number) is
begin
 select sum(sal) * pourcentage / 100
    into cout
    from emp
    where dept = unDept;
  update emp
    set sal = sal * (1 + pourcentage / 100)
    where dept = unDept;
end;
```


Création d'une procédure stockée

- Les procédures stockées sont associées aux instances de l'interface CallableStatement qui hérite de l'interface PreparedStatement
- La création d'une instance de CallableStatement se fait par l'appel de la méthode prepareCall de l'interface Connection
- On passe à cette méthode une chaîne de caractères qui décrit comment sera appelée la procédure stockée, et si la procédure renvoie une valeur ou non

Syntaxe pour les procédures stockées

- La syntaxe de l'appel des procédures stockées n'est pas standardisée ; elle diffère suivant les SGBD
- JDBC utilise sa propre syntaxe pour pallier ce problème :
  - si la procédure renvoie une valeur :
    `{ ? = call nom-procédure(?, ?,...) }`
  - si elle ne renvoie aucune valeur :
    `{ call nom-procédure(?, ?,...) }`
  - si on ne lui passe aucun paramètre :
    `{ call nom-procédure }`

Le driver traduira dans la syntaxe du SGBD

Exemple

```
CallableStatement cstmt =
  conn.prepareCall("{? = call augmenter(?,?)}");
```

Exécution d'une procédure stockée

1. Passage des paramètres « in » et « in/out » par les méthodes setXXX (idem requêtes paramétrées)
   Types des paramètres « out » et « in/out » indiqués par la méthode registerOutParameter
2. Exécution par une des méthodes executeQuery, executeUpdate ou execute, suivant le type des commandes SQL que la procédure contient
3. Paramètres « out », « in/out », et la valeur éventuelle retournée, récupérés par les méthodes getXXX (idem requêtes paramétrées)

Utilisation d'une procédure stockée

```
CallableStatement csmt = conn.prepareCall(
  "{ call augmenter(?, ?, ?) }");
// 2 chiffres après la virgule pour 3ème paramètre
csmt.registerOutParameter(3, Types.DECIMAL, 2);
// Augmentation de 2,5 % des salaires du dept 10
csmt.setInt(1, 10);
csmt.setDouble(2, 2.5);
csmt.executeQuery(); // ou execute()
double cout = csmt.getDouble(3);
System.out.println("Cout total augmentation : "
                   + cout);
```

Procédure stockée contenant plusieurs ordres SQL

- Une procédure stockée peut contenir plusieurs ordres SQL de divers types
- Pour retrouver tous les résultats de ces ordres (ResultSet ou nombre de lignes modifiées), on utilise la méthode **`getMoreResults()`** de la classe **`Statement`**
- Ainsi, si elle contient 2 ordres SELECT, le 1er ResultSet est récupéré par **`getResultSet`** ; on passe à la 2ème requête par **`getMoreResults`** et son ResultSet est récupéré par **`getResultSet`**

Ordre SQL quelconque

- On peut ne pas savoir quels ordres SQL sont contenus dans une procédure stockée
- Dans ce cas, on utilise le fait que
  - execute renvoie true si le 1er résultat est un ResultSet
  - getMoreResults renvoie true si le résultat suivant est un ResultSet
  - **`getUpdateCount()`** : renvoie le nombre de lignes modifiées, ou -1 s'il n'y a plus de résultat (ou si le résultat est un ResultSet)
- On peut exécuter tous les ordres dans une boucle dont la condition de fin est

```
!getMoreResults() && getUpdateCount() == -1
```

Schéma de code

```
boolean retval = cstmt.execute();
do {
  if (retval == false) { // pas un ResultSet
    int count = cstmt.getUpdateCount();
    if (count == -1) break; // c'est fini !
    else { // traite l'ordre SQL
      . . .
    }
  }
  else { // ResultSet
    ResultSet rs = cstmt.getResultSet();
    . . . // traite le ResultSet
  }
  retval = cstmt.getMoreResults();
while (true);
```


Renvoyer un `ResultSet` d'une procédure stockée avec Oracle

- Attention, les 4 transparents qui suivent sur ce sujet sont particuliers à Oracle ; consultez le manuel de votre SGBD si vous travaillez avec un autre SGBD
- Il faut utiliser le type « ref cursor » d'Oracle et des extensions JDBC fournies avec le driver distribué par Oracle
- Le curseur Oracle sera fermé quand l'instance de CallableStatement sera fermée

Fonction qui renvoie un curseur

- Il faut

1. Créer un type pour la référence de curseur qu'on va renvoyer
2. Créer la fonction qui renvoie la référence de curseur

Créer le type référence de curseur

- Créer un nom de type pour la référence de curseur qu'on va renvoyer
- Pour utiliser ensuite le type, il faut le créer dans un paquetage :

```
create or replace package Types AS
  type curseur_type is ref cursor;
end Types;
```

Créer la fonction

```
create or replace
function listdept(num integer)
  return Types.curseur_type
is
  empcursor Types.curseur_type;
begin
  open empcurseur
    for select dept, nomE
        from emp where dept = num;
  return empcurseur;
end;
```


Utiliser la fonction dans JDBC

```
CallableStatement cstmt =
   conn.prepareCall("{ ? = call list(?) }");
cstmt.setInt(2, 10);
cstmt.registerOutParameter(1,
  OracleTypes.CURSOR);
cstmt.execute();
ResultSet rs =
  ((OracleCallableStatement)cstmt)
    .getCursor(1);
while (rs.next()) {
  System.out.println(rs.getString("nomE")
     + ";" + rs.getInt("dept"));
}
```

Syntaxe spéciale de JDBC
(« *SQL Escape syntax* »)

- Comme avec les procédures stockées, JDBC a une syntaxe spéciale pour ne pas dépendre de particularités des différents SGBD :
  - dates : `{d '2000-10-5'}`
  - appels de fonctions : `{fn concat("Hot", "Java")}`
  - jointures externes :

```
con.createStatement("SELECT * FROM"
    + " {oj EMP RIGHT OUTER JOIN DEPT"
    + " ON EMP.DEPT = DEPT.DEPT}");
```

  - caractère d'échappement utilisé par LIKE :

```
WHERE Id LIKE '_%\\' {escape '\'}
```


Les Meta données

- JDBC permet de récupérer des informations sur le type de données que l'on vient de récupérer par un SELECT (interface ResultSetMetaData),
- mais aussi sur la base de données elle-même (interface DatabaseMetaData)
- Les données que l'on peut récupérer avec DatabaseMetaData dépendent du SGBD avec lequel on travaille

ResultSetMetaData

```
ResultSet rs =
  stmt.executeQuery("SELECT * FROM emp");
ResultSetMetaData rsmd = rs.getMetaData();
int nbColonnes = rsmd.getColumnCount();
for (int i = 1; i <= nbColonnes; i++) {
  String typeColonne = rsmd.getColumnTypeName(i);
  String nomColonne = rsmd.getColumnName(i);
  System.out.println("Colonne " + i + " de nom "
                     + nomColonne + " de type "
                     + typeColonne);
}
```


DatabaseMetaData

```
private DatabaseMetaData metaData;
private java.awt.List listTables = new List(10);
. . .
  metaData = conn.getMetaData();
  String[] types = { "TABLE", "VIEW" };
  ResultSet rs =
    metaData.getTables(null, null, "%", types);
  String nomTables;
  while (rs.next()) {
    nomTable = rs.getString(3);
    listTables.add(nomTable);
  }
```

Joker pour noms des tables et vues

Ordre SQL « dynamiques »

- Au moment où il écrit le programme, le programmeur peut ne pas connaître, le type SQL des valeurs insérées ou retrouvées dans la base de données
- Pour cela, JDBC a prévu les méthodes **`getObject`** et **`setObject`** qui effectuent des conversions automatiques (ou non automatiques) entre les types Java et SQL

Fermer les ressources

- Toutes les ressources JDBC doivent être fermées dès qu'elles ne sont plus utilisées
- Le plus souvent la fermeture doit se faire dans un bloc **`finally`** pour qu'elle ait lieu quel que soit le déroulement des opérations (avec ou sans erreurs)
- Un **`Statement`** ou un **`ResultSet`** fermé ne peut plus être utilisé
- Voir la section suivante « Transaction et exceptions » pour un schéma global de code (avec « try avec ressource » introduit par le JDK 7)

Les ressources à fermer

- **`Connection`** : leur fermeture est indispensable car c'est la ressource la plus coûteuse ; si on utilise un pool de connexions, la fermeture rend la connexion au pool
- **`Statement`**, et les sous-interfaces **`PreparedStatement`** et **`CallableStatement`**
- Un **`ResultSet`** est automatiquement fermé lorsque le *statement* qui l'a engendré est fermé ou réexécuté, ou utilisé pour retrouver le prochain résultat (**`getMoreResults`**)

Transactions et exceptions

Exception ⇒ Rollback ?

- Une exception ne provoque pas automatiquement le rollback d'une transaction
- C'est le code Java qui doit explicitement appeler la méthode **`rollback()`** s'il le faut
- Une exception peut signaler un problème récupérable ; en ce cas le programme peut choisir une alternative pour terminer correctement la transaction

Quelques règles usuelles

- Les exceptions non contrôlées correspondent souvent à des erreurs non récupérables ; en ce cas le **`catch`** correspondant lancera un *rollback*
- Les exceptions contrôlées sont parfois récupérables ; par exemple, si la clé existe déjà, on peut changer de clé, ou faire un update à la place d'un insert
- Chaque cas est particulier et peut suivre d'autres règles

Exemple de rollback

```
try {
  ...
  conn.commit();
}
catch(SQLException e) {
  // La situation ne permet pas
  // de corriger le problème
  conn.rollback();
}
```


Exemple de récupération

```
try {
  ...
  conn.commit();
}
catch(SQLException e) {
  // La situation est récupérable.
  // Le code qui suit corrige le problème
  . . .
  conn.commit();
}
```

Il faut terminer les transactions (1)

- Il ne faut jamais laisser une transaction ouverte
- Exemple d'erreur : une méthode effectue des modifications dans une base de données et une exception survient, pas attrapée dans la méthode
- L'exception est attrapée par une méthode appelante pour afficher un message d'erreur, mais la transaction n'est jamais fermée
- Les blocages effectués durant la transaction ne seront libérés que longtemps après par un timeout du SGBD

Il faut terminer les transactions (2)

- Si une transaction est en cours lors de la fermeture d'une connexion, le comportement dépend du SGBD et du driver JDBC
- Il peut y avoir un commit automatique, ou un rollback, ou un autre comportement
- Il faut donc absolument ne pas oublier de toujours terminer explicitement une transaction par un commit ou un rollback

Il faut terminer les transactions

- Le transparent suivant donne un schéma de code pour le cas où la méthode doit fermer la transaction en cours
- Ce schéma doit être adapté si la transaction ou la connexion est gérée par une méthode appelante
- Dans tous les cas, la transaction doit être terminée explicitement par un commit ou un rollback et la connexion fermée dans un bloc **`finally`**

Schéma global de code

```
try {
  ...
  conn.commit();
}
catch(SQLException e) {
  // Traiter l'exception ;
  // conn.commit() ou conn.rollback() selon les cas
  ...
}
finally {
  // Fermeture des ressources plus utilisées par la
  // suite (stmt.close(), conn.close(),...)
  ...
}
```

Fermeture automatique des connexions

- Le JDK 7 permet de fermer automatiquement des ressources, en particulier des connexions JDBC
- La clause **`try`** peut comporter une première section qui indique des ressources qui seront fermées automatiquement à la fin du bloc **`try`**
- Ces ressources doivent implémenter l'interface **`java.lang.AutoCloseable`**
- Les interfaces **`Connection`** et **`Statement`** héritent de cette interface et donc les connexions JDBC peuvent utiliser cette fonctionnalité

Exemple

```
try (
  Connection conn =
      DriverManager.getConnection(url, "toto", "mdp");
  Statement stmt = connexion.createStatement();
){
  // Travail avec la connexion et la requête
  ...
}
```

Attention,
`conn` et **`stmt`** seront fermés automatiquement à la sortie du bloc **`try`**, ce qui implique qu'il n'est pas possible d'invalider La transaction dans un bloc **`catch`**

Schéma global du code

- Pour pouvoir invalider les transactions dans un bloc catch, il faut prévoir « 2 niveaux » de blocs try avec ressource :
  - Un try « global » qui ouvre la connexion dans la section initiale du try (donc la connexion sera fermée automatiquement à la fin du bloc try)
  - Un ou plusieurs try avec ressource (ou try « simples ») qui correspondent à des transactions et qui ouvrent des Statement ; chaque try peut comporter un catch qui invalide la transaction en cours puisque la connexion est connue dans tout le try global

Code schématique global

```
try (Connection c =
        DriverManager.getConnection(url,"toto","mdp");) {
  // Transaction 1
  try (Statement stmt = c.createStatement();) {
    ...
    c.commit();
  }
  catch(SQLException e) {
    // Traiter exception et c.commit() ou c.rollback()
    ...
  }
  // Autres transactions éventuelles (semblables à la 1ère)
  ...
}
```

Codes d'erreur SQL

- Pour savoir l'action à exécuter quand une erreur survient, il faut savoir récupérer des informations sur les exceptions
- Les méthodes **`String getSQLState()`** (code SQL ANSI-92) et **`int getErrorCode()`** (code d'erreur propre au SGBD utilisé) de la classe **`SQLException`** fournissent ces informations

Quelques codes d'erreur Oracle

- 1 : violation de contrainte d'unicité (clé)
- 2291 : violation de contrainte d'intégrité référentielle (clé étrangère)
- 942 : Table ou vue inexistante
- 955 : Nom déjà utilisé (par exemple si on veut créer une table qui existe déjà)
- 1722 : mauvais format pour un nombre
- 1017 : mot de passe incorrect
- 1403 : aucune donnée n'a été trouvée