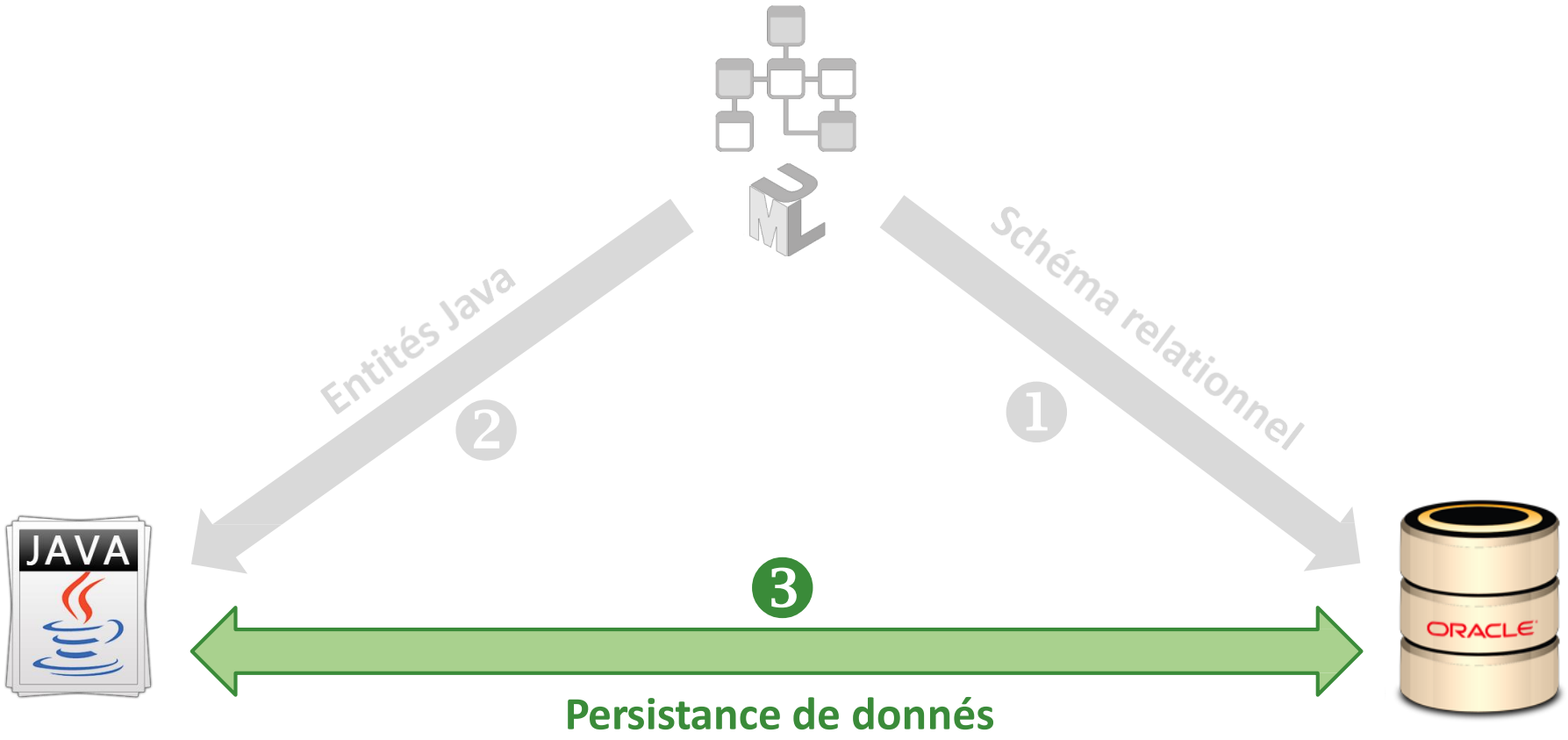
Bases de Données Avancées (BDA)

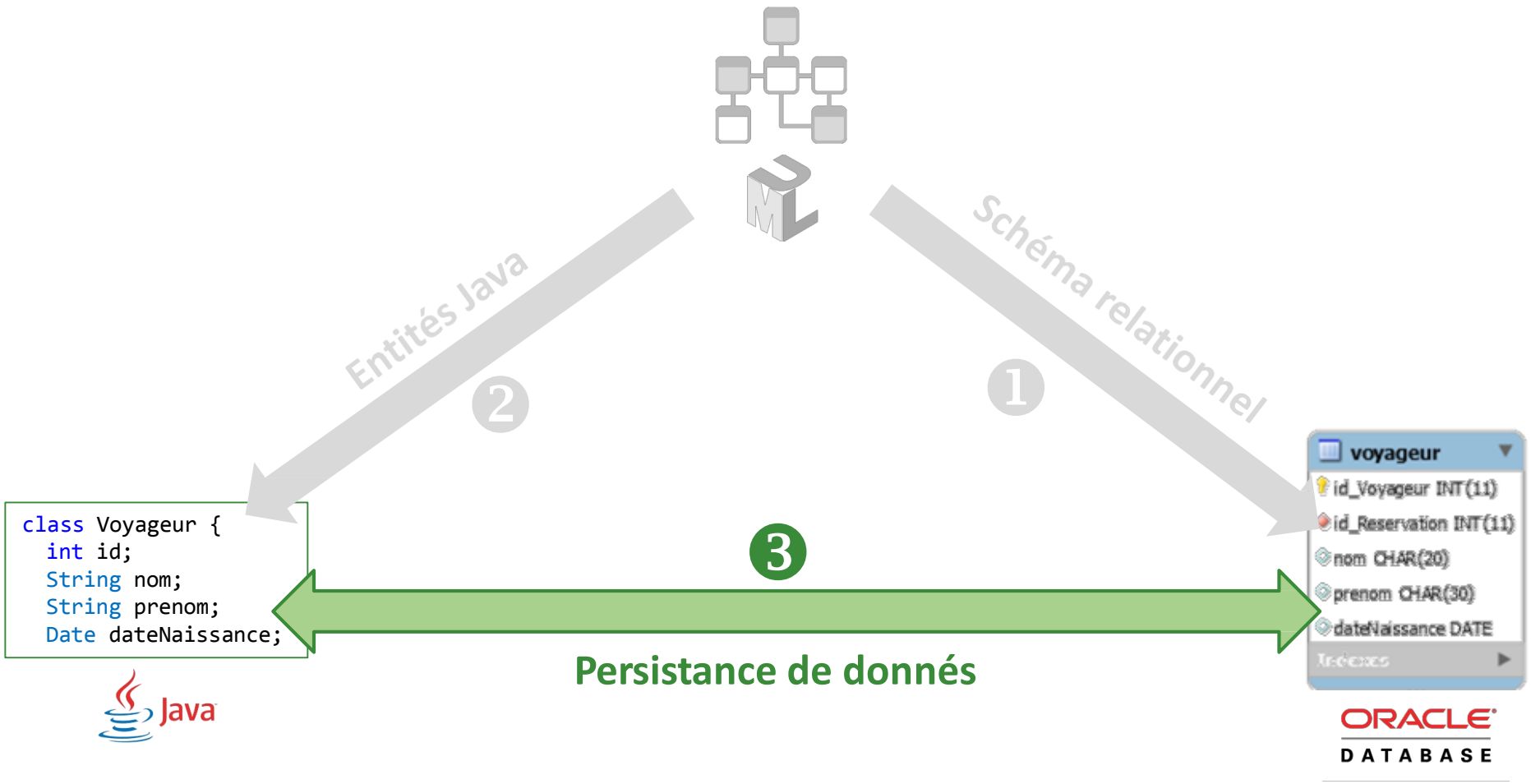
Chapitre 3-1 : Persistance de données non-transparente

Utilisation de l'API JDBC

Persistance de données (1/2)



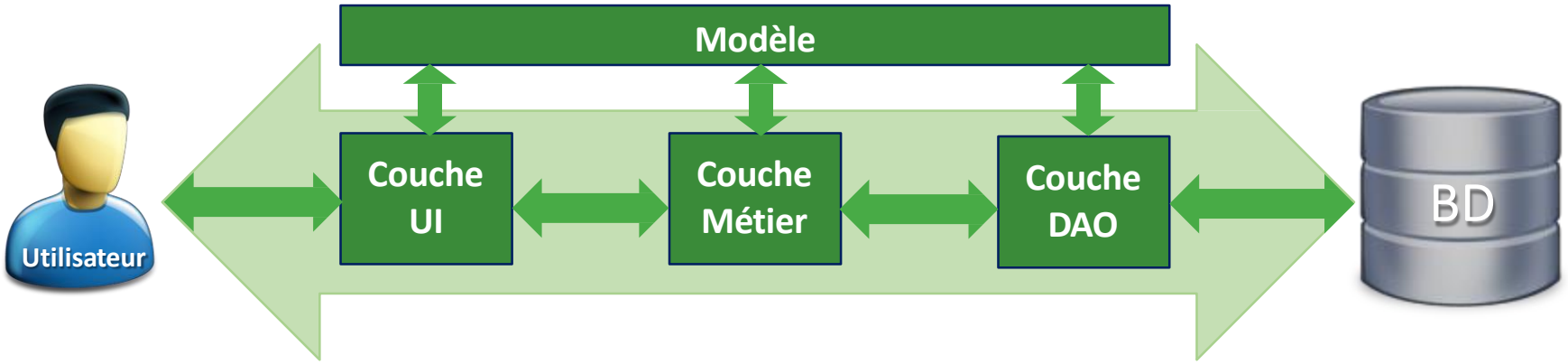
Persistance de données (2/2)



Mapping objet relationnel (1/2)

Conventionnellement, une application est divisée en 3 couches :

- La couche d'accès aux données (DAO) permet d'accéder aux données de l'applications
- La couche Métier regroupe l'ensemble des traitements que l'application doit effectuer
- La couche présentation (UI) s'occupe de la saisie des données et de l'affichage des résultats

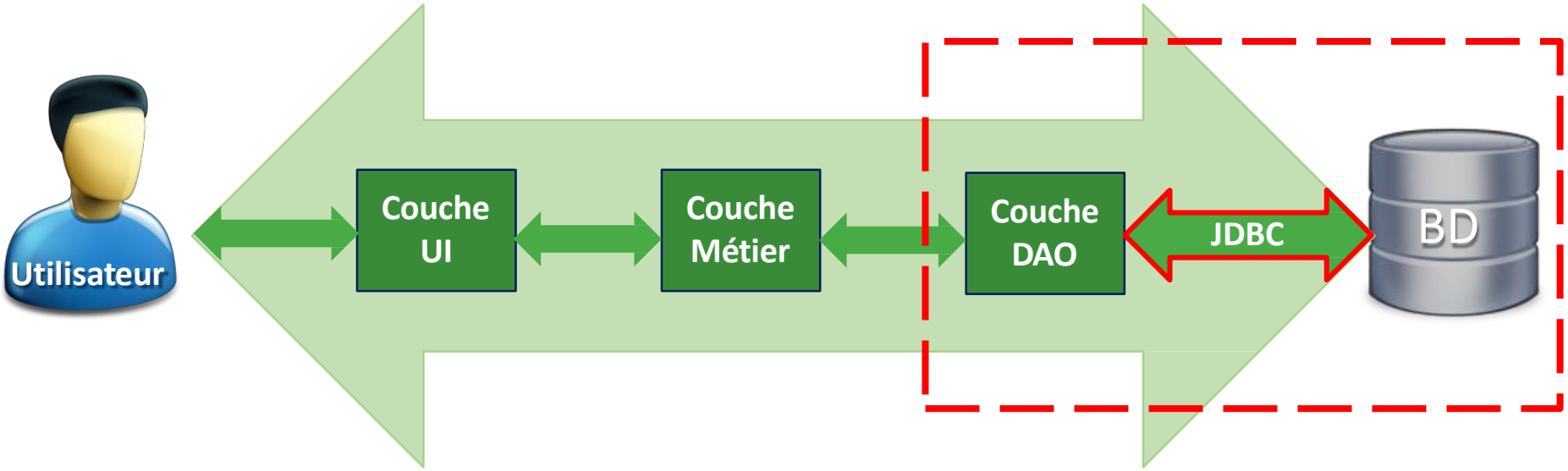


Mapping objet relationnel (2/2)

- La plupart des applications sont orientée objet :
  - Manipulation des objet et des classes
  - Utilisation de l'héritage et de l'encapsulation
  - Utilisation du polymorphisme
- Les données persistantes sont souvent stockées dans des BDR
- Le mapping objet relationnel consiste à faire correspondre un enregistrement d'une table de la BD à un objet d'une classe correspondante
- Une classe persistante est une classe dont l'état de ses objets sont stockés dans une unité de sauvegarde (BD, Fichier, etc.)

JDBC : Java DataBase Connectivity (1/2)

JDBC
Java Database Connectivity

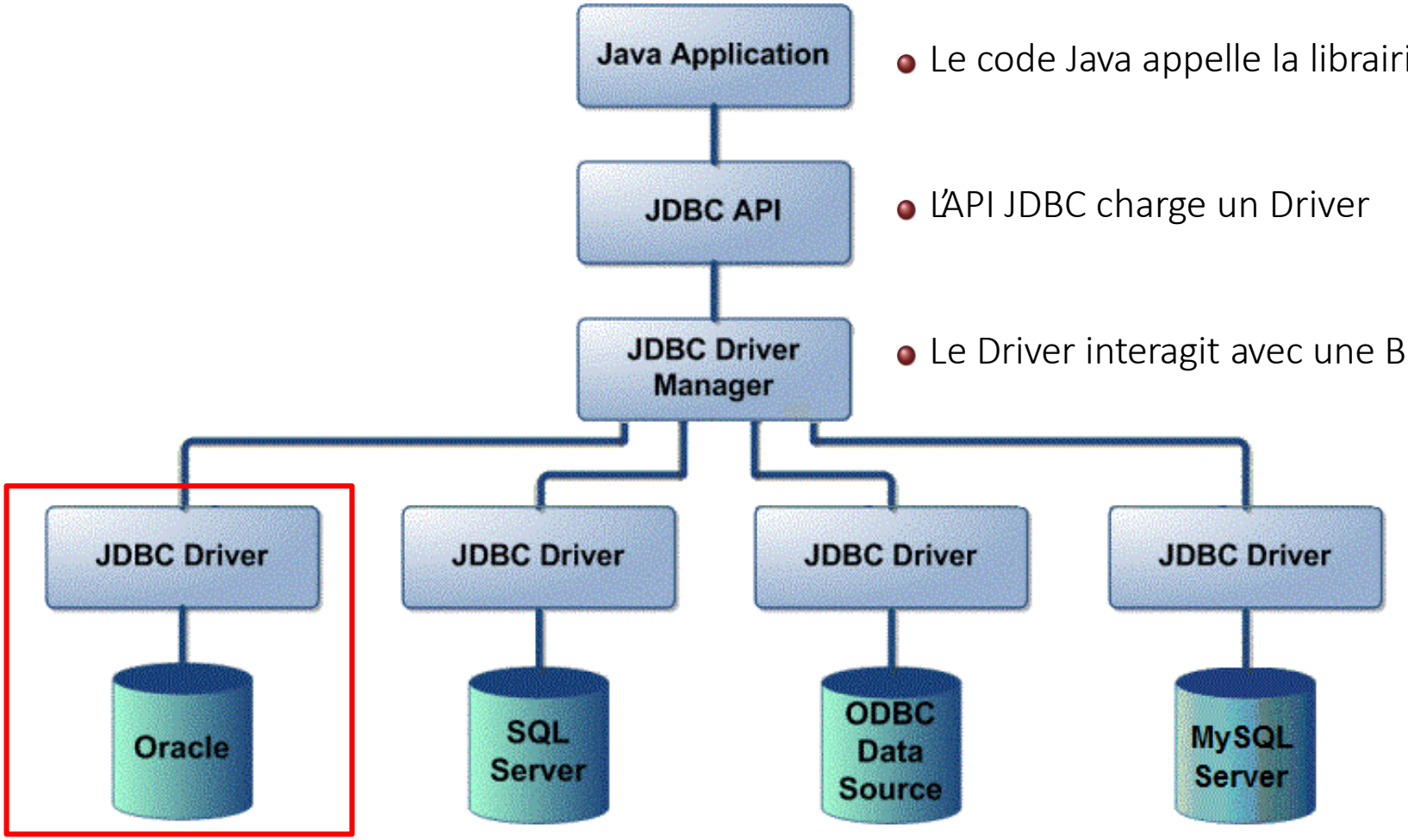


JDBC : Java DataBase Connectivity (2/2)

JDBC
Java Database Connectivity

- JDBC est une API Java permettant des connections indépendantes entre les applications Java et les BD relationnelles
- L'API JDBC permet le transfert des données à partir des tables vers des objets et vice versa,
- La correspondance des données entre le modèle relationnel et le modèle objet doit faire face à plusieurs problèmes :
  - Le modèle objet : héritage, polymorphisme, ...
  - Le modèle relationnel : indexation, intégrité, formes normales, ...
  - Les relations entre les entités des deux modèles sont différentes !

Architecture de JDBC



- Le code Java appelle la librairie JDBC
- L'API JDBC charge un Driver
- Le Driver interagit avec une BD

Etapes de connexion à une BD Oracle

Pré-connexion

1. Définir le Driver JDBC
2. Ouvrir une connexion (Connection)
3. Créer une commande (Statement)
4. Exécuter une commande
5. Récupérer le résultat d'une commande "SELECT"
6. Fermer la connexion (Connection)

Etapes de connexion à une BD Oracle

Pré-connexion

1. Démarrer le serveur Oracle dans Services (si nécessaire)

2. Télécharger le driver Oracle-JDBC (**`ojdbc7.jar`**) sur :
   - `http://www.oracle.com/technetwork/database/features/jdbc/jdbc-drivers-12c-download-1958347.html`

3. Ajouter l'API du driver Oracle-JDBC dans le projet
   - Librairies > Add Jar/Folder… > Choisir le chemin : `"ojdbc7.jar"`

4. Créer un nouveau package `dz.trash.jdbc`

Etapes de connexion à une BD Oracle

1. Définir le Driver JDBC

- Pour Oracle driver :

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

- Pour jdbc-odbc bridge driver :

```
Class.forName("sun.jdbc.odbc.JdbcOdbcDriver");
```

- Pour MySQL driver :

```
Class.forName("com.mysql.jdbc.Driver");
```

Etapes de connexion à une BD Oracle

2. Ouvrir une connexion

```
import java.sql.*;

...
static final String DB_URL = "jdbc:oracle:thin:@localhost:1521:XE";
static final String USER = "TrashAdmin";
static final String PASS = "******";

Connection con = DriverManager.getConnection(DB_URL, USER, PASS);
```

Etapes de connexion à une BD Oracle

3. Créer une commande

- Créer une simple commande SQL :

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

- Créer une commande SQL précompilée pour améliorer les performances :

```
PreparedStatement pstmt = con.prepareStatement();
```

- Faire appel à des procédures stockées dans la BD :

```
CallableStatement cstmt = con.prepareCall(name);
```

Etapes de connexion à une BD Oracle

4.a. Exécuter une commande `INSERT/UPDATE/DELETE`

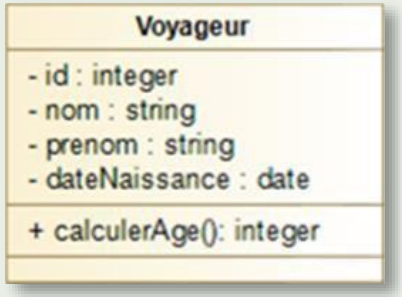
- Exemple 1 :

```
String sql = "INSERT INTO voyageur" +
             "VALUES(3,'Ali','Brahim','1-1-1995')";
stmt.executeUpdate(sql);
```

- Exemple 2 :

```
Voyageur v = new Voyageur(3, "Ali", "Brahim", "1-1-1995");
String sql ="INSERT INTO voyageur VALUES(?,'?',?,'?')";

pstmt.setInt(1, v.getId());
pstmt.setString(2, v.getName());
pstmt.setString(3, v.getCredit());
pstmt.setString(4, v.getContent());
...
pstmt.executeUpdate(sql);
```

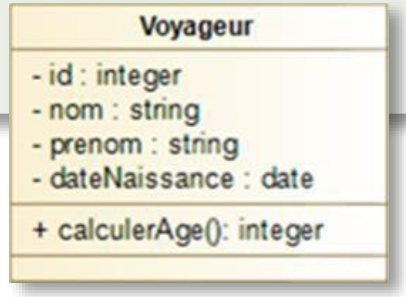


Etapes de connexion à une BD Oracle

4.b. Exécuter une commande SELECT

- Exemple :

```
String sql ="SELECT * FROM voyageur WHERE prenom ='Ali'";
stmt.executeQuery(sql);
// ou
String p_sql ="SELECT * FROM voyageur WHERE prenom = '?' ";
pstmt.setString(1, "Ali");
pstmt.executeQuery(p_sql);
```



Etapes de connexion à une BD Oracle

4.c. Faire appel à une procédure stockée

- Exemple :

```
String sql ="{call count_by_res_id(?)}";
CallableStatement cstmt = con.prepareCall(sql);
cstmt.setInt(1, 12);
...
cstmt.execute();
```

Etapes de connexion à une BD Oracle

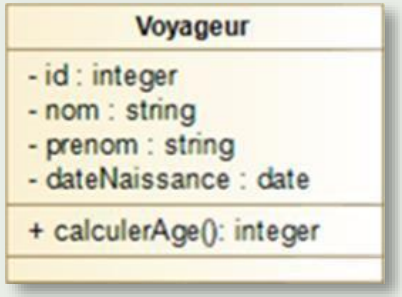
5. Récupérer le résultat d'une commande SELECT (1/2)

- Exemple :

```
Set<Voyageur> list = HashSet<>();

String sql ="SELECT * FROM voyageur";
ResultSet rs = stmt.executeQuery(sql);
while(rs.next()){
    int id = rs.getInt("id");
    String nom = rs.getString("nom");
    String prenom = rs.getString("prenom");
    ...

    Voyageur v = new Voyageur(id, nom, prenom, ...);
    list.put(v);
}
```



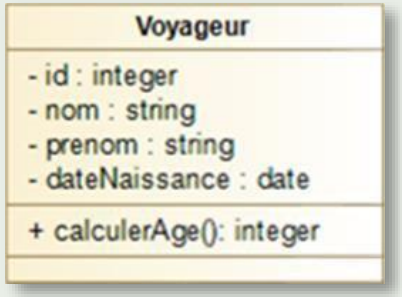
Etapes de connexion à une BD Oracle

5. Récupérer le résultat d'une commande SELECT (2/2)

- Récupération des colonnes :

```
ResultSetMetaData rsmd = rs.getMetaData();
int numcols = rsmd.getColumnCount();

for (int i = 1 ; i <= numcols; i++) {
    System.out.print(rsmd.getColumnLabel(i));
}
```



Types SQL (Oracle DB) vs. types Java

Types SQL (Oracle DB)	Types Java
NUMBER(1)	boolean
NUMBER(5)	short
NUMBER(10)	int
NUMBER(19)	long
NUMBER(19,4)	float
NUMBER(19,4)	double
NUMBER(38)	java.math.BigDecimal
CHAR(1)	char
VARCHAR2(255)	String
LONG	char[]
RAW, LONGRAW	byte[]
DATE	java.sql.Timestamp, Time, Date

Etapes de connexion à une BD Oracle

6. Fermer la connexion

```
...
Connection con = DriverManager.getConnection(DB_URL, USER, PASS);
Statement stmt = con.createStatement();

...
stmt.close();
con.close();
```

Etapes de connexion à une BD Oracle

Récapitulatif

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

2.
```
String DB_URL="..."; String USER="AGENT"; String PASS="****";
Connection con = DriverManager.getConnection(DB_URL,USER,PASS);
```

3.
```
Statement stmt = con.createStatement();
String sql ="SELECT * FROM voyageur";
```

4. `ResultSet rs = stmt.executeQuery(sql);`

5. `while(rs.next()){ ... }`

6.
```
stmt.close();
con.close();
```

Transactions

- **Une transaction** est un bloc d'une ou plusieurs commande(s) qui sont **exécutées ensemble**

Transaction

```
try{
Connection con = DriverManager.getConnection(DB_URL,USER,PASS);
con.setAutoCommit(false);

Statement stmt = con.createStatement();
stmt.executeUpdate("INSERT/UPDATE/DELETE...");
stmt.executeUpdate("INSERT/UPDATE/DELETE...");
…
con.commit();
con.setAutoCommit(true);

} catch(SQLException ex){
        con.rollback();
}
```

Gestion des exceptions SQL

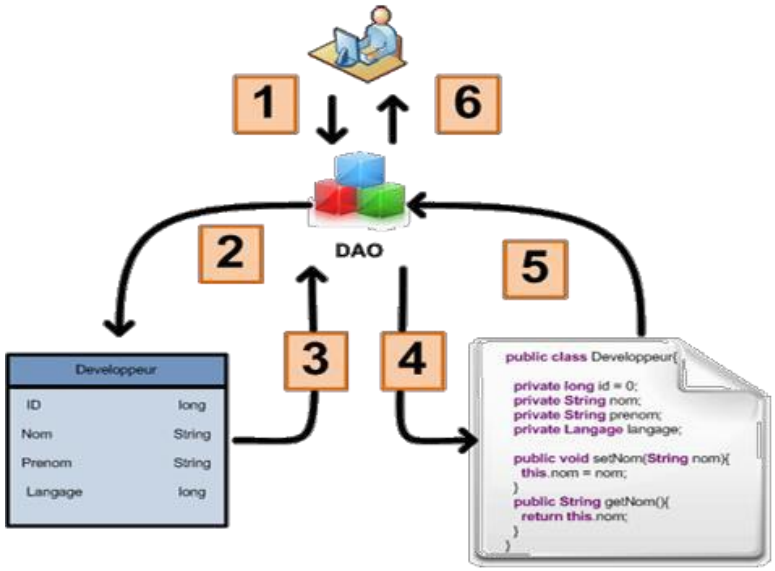
- **Une exception SQL** est une condition d'erreur lors de l'exécution d'un commande SQL.

```
try {
    Class.forName("com.oracle.jdbc.Driver");
} catch (ClassNotFoundException ex) {
    System.err.println("ClassNotFoundException: " + ex.getMessage());
}
try {
    // Mettre les commandes SQL ici
} catch (SQLException ex) {
    while(ex != null){
        System.err.println("SQLException: " + ex.getMessage());
        System.err.println("SQLState: " + ex.getSQLState());
        System.err.println("ErrorCode: " + ex.getErrorCode());
        ex = ex.getNextException();
    }
}
```

Design pattern DAO (1/2)

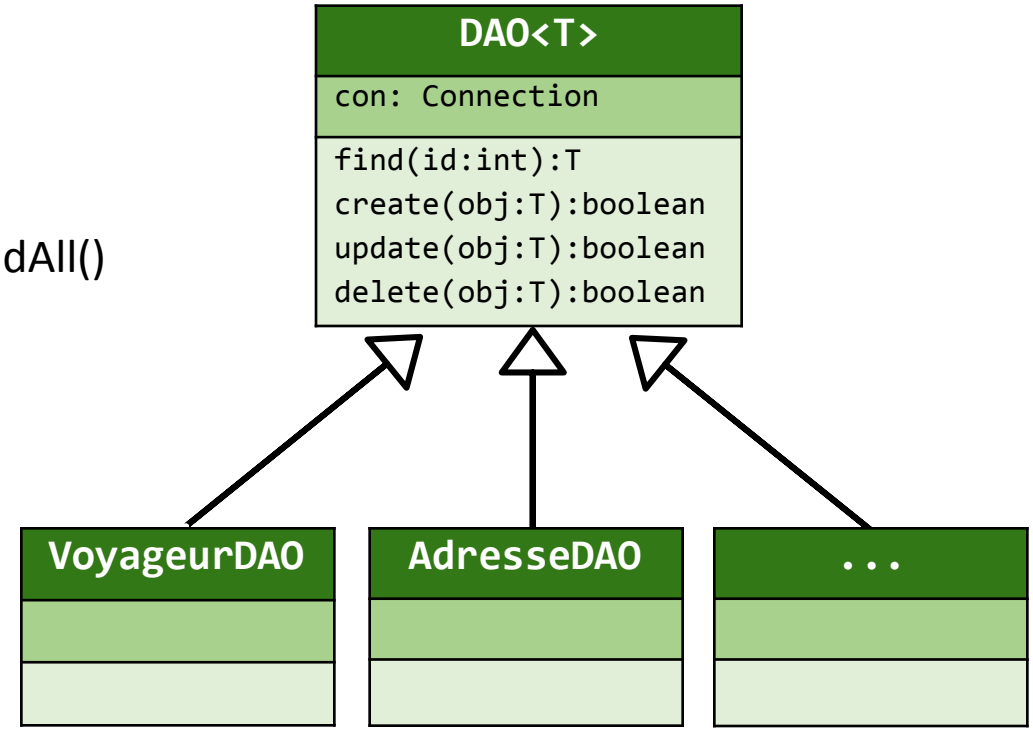
DAO : Data Access Object :

- La couche métier **utilise** la couche DAO pour interroger la BD
- DAO permet d'**isoler les opérations** sur la BD
- Pas de couplage entre l'application et la technologie de persistance
- Ainsi, le changement du mode de stockage **ne remet pas en cause** le reste de l'application
  - Seules ces classes dites **"techniques"** seront à modifier



Design pattern DAO (2/2)

- Chaque **classe persistante** doit posséder **sa classe DAO**
- DAO fournit les méthodes de
  - **Création** : create(obj)
  - **Suppression** : delete(obj)
  - **Mise à jour** : update(obj)
  - **Interrogation** : find(id), findAll()



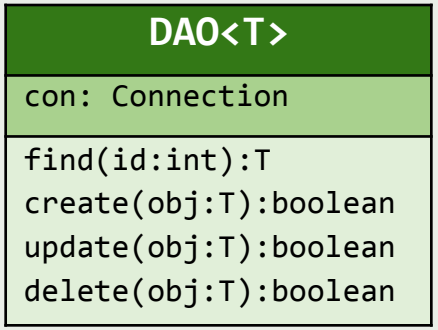
Design pattern DAO

Classe DAO<T>

```
public abstract class DAO<T> {
  protected Connection con = null;

  public DAO(Connection con){
    this.con = con;
  }

  public abstract boolean create(T obj);
  public abstract boolean delete(T obj);
  public abstract boolean update(T obj);
  public abstract T find(int id);
}
```



Design pattern DAO

Exemple : Classe `VoyageurDAO` (1/2)

```
public class VoyageurDAO extends DAO<Voyageur>{

  public VoyageurDAO(Connection con) {
    super(con);
  }

  public boolean create(Voyageur obj) { ... }
  public boolean delete(Voyageur obj) { ... }
  public boolean update(Voyageur obj) { ... }
  public Voyageur find(int id) { ... }
  ...
  public Set<Voyageur> findAll() { ... }
  ...
}
```

Design pattern DAO

Exemple : Classe `VoyageurDAO` (2/2) – `findAll()`

```
public class VoyageurDAO extends DAO<Voyageur>{
    ...
    public Set<Voyageur> findAll() {
        Set<Voyageur> list = HashSet<>();
        ResultSet rs = stmt.executeQuery("SELECT * FROM voyageur");
        while(rs.next()){
            int id = rs.getInt("id");
            String nom = rs.getString("nom");
            String prenom = rs.getInt("prenom");
            ...
            list.put(new Voyageur(id, nom, prenom, ...));
        }
        return list;
    }
}
```