JDBC

Massimo Mecella

Dipartimento di Ingegneria informatica automatica e gestionale
Antonio Ruberti
Sapienza Università di Roma

Progetto di Applicazioni Software

Introduzione

- I DBMS relazionali supportano interfacce interattive per l'immissione di comandi SQL e la presentazione del risultato (ad es., l'interprete dei comandi di MySQL)
- L'utilizzo di tali interfacce è sufficiente solo se il lavoro può essere svolto interamente tramite comandi SQL
- Nella pratica abbiamo bisogno di poter accedere ad una base di dati a partire da applicazioni scritte in un qualsiasi linguaggio di programmazione

Accedere ai DBMS dalle applicazioni

- Le applicazioni scritte in un generico linguaggio di programmazione sono gestite come processi separati che si connettono al DBMS per interagire con esso
- Stabilita la connessione, si utilizza:
  - un meccanismo per incapsulare i comandi SQL nel linguaggio di programmazione
    $\rightarrow$ tramite SQL, si inseriscono, modificano, reperiscono i dati
  - un meccanismo di cursori per manipolare collezioni dati e scorrere i risultati uno alla volta
    $\rightarrow$ mentre SQL opera su insiemi (o multinsiemi) di valori, nativamente, il linguaggio di programmazione non ha tale capacità

SQL incapsulato

- **Inserimento**- Inseriamo i comandi SQL in un programma scritto in un **linguaggio ospite** (ad es., il linguaggio C)
- **Segnalazione**- I comandi SQL devono essere segnalati in modo che il preprocessore possa trasformarli in chiamate di funzioni del linguaggio ospite
- **Variabili speciali**-Nei comandi SQL devono essere dichiarate alcune variabili speciali del linguaggio ospite

- **Compatibilità di tipo**- E' necessario stabilire la compatibilità fra i tipi di dato SQL e quelli del linguaggio ospite (SQL-92 stabilisce la corrispondenza per diversi linguaggi di programmazione)
- **Gestione delle eccezioni**- Servono meccanismi per segnalare al programma ospite eventuali errori sorti durante l'esecuzione dell'SQL

Esempio

Si assuma che nel database sia presente la tabella
`Velisti(nome,identificativo,esperienza, eta)`
Si consideri il seguente codice (incapsulato in un progr. C)

```
EXEC SQL BEGIN DECLARE SECTION
char c_vname[20];
long c_vid;
short c_esperienza;
short c_eta;
EXEC SQL END DECLARE SECTION

EXEC SQL BEGIN
INSERT INTO Velisti VALUES (:c_vname,:c_vid,
                  :c_esperienza,:c_eta);
EXEC SQL END
```

Cursori

- Permettono di reperire una alla volta le righe di una tabella relazionale o restituite da una interrogazione
- Possono essere quindi dichiarati su qualsiasi interrogazione e su qualsiasi tabella
- Le operazioni di base su un cursore sono
  - apertura (il che posiziona il cursore appena prima della prima riga)
  - lettura della riga corrente
  - movimento del cursore alla riga successiva, alla precedente
  - chiusura di un cursore

Esempio

```
EXEC SQL DECLARE vinfo CURSOR FOR
SELECT V.name, V.eta
FROM Velisti V
WHERE V.esperienza> :c_minesperienza
```

Dove `:c_minesperienza` è una variabile C precedentemente dichiarata in una `DECLARE SECTION`
Per aprire il cursore usiamo `OPEN vinfo;`
Per leggere la prima riga del cursore nelle variabili del linguaggio ospite `FETCH vinfo INTO :c_vnome, :c_eta;`
Eseguendolo ripetutamente (ad es. in un ciclo `While`) possiamo leggere tutte le righe calcolate dall'interrogazione una alla volta.

Proprietà dei cursori

- Al termine del suo utilizzo è bene chiudere il cursore con il comando `CLOSE` (ad es. `CLOSE vinfo;`)
- Un cursore può essere **di sola lettura** o **aggiornabile**. In quest'ultimo caso, ad esempio, possiamo eseguire il seguente comando per fare aggiornamenti

```
UPDATE velisti V
SET V.esperienza=V.esperienza-1
WHERE CURRENT of vinfo;
```

oppure, possiamo cancellare tuple tramite il seguente comando

```
DELETE Velisti
WHERE CURRENT of vinfo;
```

SQL incapsulato: critiche

- Il codice ottenuto facendo uso di SQL incapsulato è una naturale estensione del codice del linguaggio ospite, quindi semplice da leggere
- D'altro canto, nell'SQL incapsulato, il preprocessore trasforma i comandi SQL in chiamate di funzione del linguaggio ospite, ed il compilatore fa uso di particolari librerie, specifiche per ciascun tipo di DBMS
  $\Rightarrow$ trasformazione che varia da DBMS a DBMS: le applicazioni che usano SQL incapsulato sono indipendenti dal DBMS a livello di codice sorgente, ma non di eseguibile
  $\Rightarrow$ **portabilità** limitata dell'applicazione

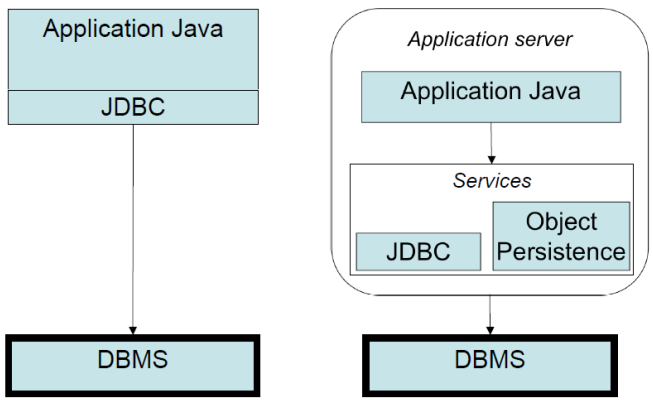
Verso una maggiore indipendenza dal DBMS: JDBC

- Tale limite viene superato con l'uso di interfacce (API) standard per la comunicazione con DBMS, come ODBC o **JDBC** (**Java DataBase Connectivity**)
- JDBC è un' API Java per l'esecuzione di comandi SQL indirizzati ad un DBMS: serie di classi ed interfacce che implementano una modalità standardizzata per l'interazione con il DBMS da parte di applicazioni Java
- Ciascun DBMS fornisce un driver specifico che:
  - viene caricato a run-time
  - traduce le chiamate alle funzioni JDBC in chiamate alle funzioni del DBMS

Modalità di accesso al database

JDBC è generalmente usato in due architetture di sistema:

- l'applicazione Java "parla" direttamente col database
- un livello intermedio invia i comandi SQL al database, eseguendo controlli su accessi e aggiornamenti



Architettura di JDBC

Un sistema che usa JDBC ha quattro componenti principali

Applicazione: inizia e termina la connessione, imposta le transazioni, invia comandi SQL, recepisce risultati. Tutto avviene tramite l'API JDBC (nei sistemi three tiers se ne occupa lo strato intermedio)

Gestore di driver: carica i driver, passa le chiamate al driver corrente, esegue controlli sugli errori

Driver: stabilisce la connessione, inoltra le richieste e restituisce i risultati, trasforma dati e formati di errore dalla forma dello specifico DBMS allo standard JDBC

Sorgente di dati: elabora i comandi provenienti dal driver e restituisce i risultati

Caratteristiche di JDBC

- esecuzione di comandi SQL
  1. DDL (Data Definition Language)
  2. DML (Data Manipulation Language)
- manipolazione dei risultati tramite **result set** (una forma di cursore)
- reperimento di **metadati**
- gestione di **transazioni**
- gestione di **errori** ed **eccezioni**
- definizione di **stored procedure** scritte in Java (supportate da alcuni DBMS)

Software: cosa occorre

1. Piattaforma Java
   - Java Standard Edition (anche conosciuta come Java 2 platform o J2SE)
   - il package `java.sql` - funzionalità di base di JDBC
   - il package `javax.sql` - funzionalità più avanzate (ad es. utili per la gestione di pooling di connessioni o per la programmazione a componenti)
2. Driver JDBC per il DBMS a cui ci si vuole connettere
3. DBMS

Driver JDBC

Il driver JDBC per un certo DBMS viene distribuito in genere dalla casa produttrice del DBMS. È di fatto una libreria Java che va opportunamente linkata in fase di esecuzione dell'applicativo. Questo va fatto:

(*i*) settando opportunamente le variabili d'ambiente

(*ii*) indicando da riga di comando la libreria da linkare quando si lancia l'applicazione

(*iii*) configurando l'ambiente di sviluppo Java che state usando.

Fasi dell'interazione con una base di dati via JDBC

L'interazione con una base di dati via JDBC prevede le seguenti fasi:

1. Caricamento dinamico del driver JDBC opportuno per la sorgente dati
2. Connessione con la sorgente di dati
3. Sessione di interazione
4. Chiusura della connessione

Prima fase: caricamento del driver JDBC

- I driver delle sorgenti dati sono gestiti dal **Driver Manager**, i.e. una classe che opera tra l'utente e i driver
- Il caricamento del driver avviene tramite il meccanismo Java per il caricamento dinamico delle classi, i.e. utilizzando il metodo `Class.forName(String s)`
  $\rightarrow$ il metodo statico `forName` della classe `Class` restituisce una istanza della classe Java specificata nella stringa passata come parametro, che indica il nome completamente qualificato (cioè con il package a cui appartiene) della classe stessa

$\rightarrow$ Nel nostro caso, passiamo una stringa per la creazione di un oggetto della classe `Driver` (specifico per il DBMS selezionato), il quale automaticamente "registra" se stesso con la classe `DriverManager`

Ad esempio:

```
Class.forName("oracle.lite.poljdbc.POLJDBCDriver");
Class.forName("com.mysql.jdbc.Driver");
```

Seconda fase: connessione con la sorgente dati

- Per connettersi ad una sorgente, si usa il metodo **statico** `getConnection` della classe `DriverManager` che restituisce un oggetto che appartiene ad una classe che implementa l'interfaccia `Connection` specifica per il DBMS per il quale si è precedentemente caricato il driver a run-time
- I parametri di `getConnection` sono un'**URL JDBC**, lo **username** e la **password**, dove l'URL JDBC ha la forma `jdbc:<sub-protocollo>:<altri-parametri>`

Connessione con la sorgente dati (esempio)

```
String url="jdbc:postgresql://localhost/myDatabase";
String login="postgres";
String password="password";
Connection conn=DriverManager.getConnection(url,
                                username,password);
```

- `postgresql` è il subprotocollo per PostgreSQL;
- `//localhost/` indica che ci stiamo connettendo al DBMS dalla stessa locazione su cui è installato, `myDatabase` è il nome del database a cui ci si vuole connettere (può anche mancare).

La classe `DatabaseMetaData`

Permette di ottenere informazioni sul sistema di basi di dati stesso, come ad esempio informazioni sul catalogo.
Il codice seguente mostra come ottenere nome e versione di un driver JDBC

```
DatabaseMetaData md=con.getMetaData();
System.out.println("Informazioni sul driver:");
System.out.println("Nome: "+ md.getDriverName() +
     "; versione: " + md.getDriverVersion());
```

Si noti che il metodo `getMetaData()` è invocato tramite un oggetto di tipo `Connection`.

Alcuni metodi della classe `DatabaseMetaData`

- `getConnection()` restituisce l'oggetto di tipo `Connection` a cui si riferisce l'oggetto di classe `DatabaseMetaData`
- `getURL()` restituisce l'URL per il database in uso
- `getUserName()` restituisce il nome dell'utente connesso
- `getMaxConnections()` restituisce il numero massimo di connessioni possibili

Terza fase: sessione di interazione

L'interazione con la sorgente di dati avviene attraverso l'uso di oggetti di classi che implementano per lo specifico DBMS che si accede, le seguenti interfacce:

- `Statement`: istruzione SQL nota a tempo di compilazione
- `PreparedStatement`: istruzione SQL la cui struttura è fissata a tempo di compilazione, ma che può ammettere parametri il cui valore può variare a run-time
- `CallableStatement`: istruzione necessaria per accedere alle stored procedures
- `ResultSet`: struttura dati simile ad un cursore

Statement

- L'oggetto che consente di eseguire una istruzione SQL è lo **statement** (interfaccia `Statement`)
- Uno statement è sempre relativo ad una connessione $\Rightarrow$ serve una istanza di una connessione attiva per avere uno statement (metodo `createStatement` di `Connection`)

Ad esempio:

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

Metodi importanti della classe `Statement`

- `executeUpdate` per creazione, eliminazione, aggiornamento
- `executeQuery` per interrogazione
- Entrambi prendono in ingresso una stringa, che di fatto è la stringa SQL che si vuole immettere (senza il punto e virgola finale!)

Stringhe Java/SQL

Attenzione!

- Java è case-sensitive, mentre SQL lo è solamente per i valori di tipo stringa
- In Java gli apici che delimitano stringhe sono doppi, mentre in SQL sono singoli

Stringhe Java/SQL (Esempio)

Esempio di stringa che si può usare come parametro di input del metodo `executeUpdate`:

```
String createTableCoffees="CREATE TABLE COFFEES"+
"(COF_NAME VARCHAR(32), SUP_ID INTEGER, PRICE FLOAT,"+
"SALES INTEGER, TOTAL INTEGER)";
```

N.B. La stringa è concatenata con + perché Java non ammette definizioni di stringhe su più righe

executeUpdate

Creiamo la tabella COFFEES con la stringa SQL di prima:

```
stmt.executeUpdate(createTableCoffees);
```

Inseriamo due righe in COFFEES; si osservi che utilizziamo sempre lo stesso oggetto Statement per le diverse operazioni.

```
stmt.executeUpdate("INSERT INTO COFFEES "+
  "VALUES ('Colombian',101,7.99,0,0)" );

stmt.executeUpdate("INSERT INTO COFFEES "+
  "VALUES ('Espresso',150,9.99,0,0)" );
```

Il metodo executeUpdate restituisce un int pari al numero di righe modificate. Nel caso di uno statement DDL, ad esempio la creazione di una tabella, viene restituito 0.

executeQuery

Il metodo executeQuery di Statement restituisce un oggetto di tipo ResultSet

```
ResultSet rs=stmt.executeQuery("SELECT * FROM COFFEES");
```

o anche

```
String query="SELECT COF_NAME,PRICE FROM COFFEES";
ResultSet rs=stmt.executeQuery(query);
```

`ResultSet`

- Un `ResultSet` è un oggetto che rappresenta il risultato di una query, ovvero un insieme di tuple
- Come un cursore, un `ResultSet` è inizialmente posizionato **prima della prima riga**
- Il `ResultSet` può avanzare col metodo `next()`, che una volta invocato, sposta il cursore di una riga in avanti e restituisce `false` (booleano) se non ci sono più righe da analizzare, `true` altrimenti

ResultSet (esempio)

```
String query="SELECT COF_NAME,PRICE FROM COFFEES";
ResultSet rs=stmt.executeQuery(query);
while(rs.next()) {
  //elaborazione dei dati
 }
```

→ Come si accede ai dati "puntati" dall'oggetto `rs`?

I metodi `getXXX`

I metodi `getXXX()` restituiscono il valore della colonna specificata dal parametro, corrispondente alla **riga corrente** (quella su cui è posizionato il cursore). Il parametro può essere:

- il **nome** della colonna;
- il **numero d'ordine** della colonna

N.B. Il numero d'ordine è **quello della tabella dei risultati** e non quello della tabella interrogata.
`XXX` va sostituito con il tipo di dato che si vuole prelevare dal cursore nella posizione indicata dal parametro. Ovviamente tale tipo di dato deve essere compatibile con il tipo di dato prelevato dal DB.

Esempio

```
String query="SELECT COF_NAME,PRICE FROM COFFEES";
ResultSet rs=stmt.executeQuery(query);
while(rs.next()) {
  String s=rs.getString("COF_NAME");
  float n=rs.getFloat("PRICE");
  System.out.println(s+"   "+n); }
```

Cosa stampa?

I metodi `getXXX` (cont.)

Istruzioni equivalenti alle precedenti righe all'interno del ciclo `while`:

```
String s=rs.getString(1);
float n=rs.getFloat(2);
```

Abbiamo usato `getString()` e `getFloat()` per leggere dati dal `ResultSet()`. Esistono anche `getInt()`, `getDate()`, `getBoolean()`, `getTimestamp()` e altri.

Corrispondenza tra tipi di dato Java e SQL

Tipo SQL	Tipo o Classe Java	Metodo di lettura di ResultSet
BIT	boolean	getBoolean()
CHAR (..)	String	getString()
VARCHAR (..)	String	getString()
DOUBLE	double	getDouble()
FLOAT	double	getDouble()
INTEGER	int	getInt()
NUMERIC	int	getInt()
REAL	float	getFloat()

Corrispondenza tra tipi di dato Java e SQL

Tipo SQl	Tipo o Classe Java	Metodo di lettura di `ResultSet`
`DATE`	`java.sql.Date`	`getDate()`
`TIME`	`java.sql.Time`	`getTime()`
`TIMESTAMP`	`java.sql.Timestamp`	`getTimestamp()`

La corrispondenza fra i tipi SQL ed i tipi Java è comunque definita in maniera piuttosto "elastica". Si noti, ad esempio, che per prelevare il prezzo di un caffè (tipo SQL `FLOAT`) dal result set dell'interrogazione effettuata, abbiamo in precedenza usato `getFloat()` al posto di `getDouble()`.

Nota sulla classe `java.sql.Date` (Cont.)

Il metodo `getDate()` restituisce un oggetto di classe `java.sql.Date` (che estende la classe `java.util.Date`) nel formato 'yyyy-MM-dd' (anno, mese, giorno). La classe `java.sql.Date` mette a disposizione alcuni metodi per eseguire elementari confronti sulle date (ad es., `compareTo`).
Molti altri metodi della classe sono però deprecati.

Nota sulla classe `java.sql.Date` (Cont.)

Per fare operazioni più complesse conviene trasformare l'oggetto di classe `java.sql.Date` in un oggetto di classe `java.util.GregorianCalendar`. Un possibile modo è riportato di seguito

```
String s = data.toString();
int year=Integer.parseInt(s.substring(0,4));
int month=Integer.parseInt(s.substring(5,7));
int day=Integer.parseInt(s.substring(8,10));
GregorianCalendar g=
new GregorianCalendar(year,month,day);
```

Dove `data` è un oggetto di classe `java.sql.Date`.

Nota sulla classe `java.sql.Date` (Cont.)

Per ottenere una stampa da un `GregorianCalendar` in formato 'dd-mm-yyyy', si può ad esempio procedere come segue

```
System.out.print( g.get(Calendar.DAY_OF_MONTH));
System.out.print( "-");
System.out.print( g.get(Calendar.MONTH));
System.out.print( "-");
System.out.println(g.get(Calendar.YEAR));
```

Metadati su un Result Set

È possibile ottenere i dati su di un oggetto di tipo `ResultSet` creando un oggetto di tipo `ResultSetMetaData` e invocando i metodi della classe `ResultSetMetaData` su di esso.

- `ResultSetMetaData getMetaData()`, metodo della classe `ResultSet`, restituisce i metadati dell'oggetto su cui è invocato
- `int getColumnCount()` restituisce il numero di colonne del `ResultSet` associato
- `String getColumnLabel(int column)` riceve un intero *i* come argomento, e restituisce il **nome** della *i*-esima colonna del `ResultSet` associato

Esempio

COF_NAME	SUP_ID	PRICE	SALES	TOTAL
Colombian	101	7.99	0	0
French Roast	49	8,99	0	0
Espresso	150	9.99	0	0
Colombian Decaf	101	8.99	0	0
French Roast Decaf	49	10.75	0	0

Esempio

```
Statement stmt=conn.createStatement();
ResultSet rs=stmt.executeQuery("SELECT * FROM COFFEES");
ResultSetMetaData rsmd=rs.getMetaData();
int numeroColonne=rsmd.getColumnCount();
for(int i=1; i<=numeroColonne; i++)
{ if( i > 1 ) System.out.print(",  ");
  String nomeColonna=rsmd.getColumnLabel(i);
   System.out.print(nomeColonna);
}
System.out.println("");
```

```
while(rs.next())
{ for(int i=1; i<=numeroColonne; i++)
  { if( i > 1 ) System.out.print(",  ");
    String valoreColonna=rs.getString(i);
    System.out.print(valoreColonna);
  }
  System.out.println("");
}
stmt.close();
```

Cosa fa?

Esempio: output

```
COF_NAME,  SUP_ID,  PRICE,  SALES,  TOTAL
Colombian,  101,  7.99,  0,  0
French Roast,  49,  8.99,  0,  0
Espresso,  150,  9.99,  0,  0
Colombian Decaf,  101,  8.99,  0,  0
French Roast Decaf,  49,  10.75,  0,  0
```

Tipo di una colonna di un ResultSet

È possibile anche ottenere il tipo di una colonna di un `ResultSet`.

```
ResultSetMetaData rsmd=rs.getMetaData();
int jdbcType=rsmd.getColumnType(2);
String jdbcTypeName=rsmd.getColumnTypeName(2);
```

Ogni tipo del DBMS è identificato da un nome e da un codice (intero). Ad esempio al tipo intero corrispondono il codice `4` e il nome `INTEGER`.

Altri metodi di `ResultSetMetaData`

- `String getTableName (int column)` restituisce il nome della tabella a cui appartiene la colonna designata tramite il parametro
- `String getSchemaName (int column)` restituisce il nome dello schema della tabella a cui appartiene la colonna designata tramite il parametro

$\rightarrow$ entrambi restituiscono la stringa vuota se il metodo non è applicabile

Altri metodi di `ResultSetMetaData` (cont.)

- `boolean isReadOnly (int column)` restituisce `true` se la colonna designata tramite il parametro **non è scrivibile**
- `boolean isWritable (int column)` restituisce `true` se la colonna designata tramite il parametro **è scrivibile**
- `boolean isSearchable (int column)` restituisce `true` se la colonna designata tramite il parametro **può essere usata in una clausola where**

Prepared Statement

- Un `PreparedStatement` è una specializzazione di `Statement`
- Contiene una query, dalla struttura fissata, ma in grado di ricevere parametri che sono istanziati a run-time
- Può quindi essere usata più volte con effetti diversi

Prepared Statement (esempio)

```
PreparedStatement updateSales=conn.prepareStatement(
  "UPDATE COFFEES SET SALES=? WHERE COF_NAME=?" );
```

I punti interrogativi sono **parametri** che vengono passati con i metodi `setXXX()`, analoghi ai `getXXX()`.

```
updateSales.setInt(1,75);
updateSales.setString(2,"Colombian");
```

Il primo argomento indica il numero d'ordine del parametro, il secondo argomento il valore.

Prepared Statement (cont.)

Per eseguire un `PreparedStatement`:

```
updateSales.executeUpdate();
```

oppure

```
updateSales.executeQuery();
```

Si noti che `executeUpdate` ed `executeQuery` in questo caso non hanno bisogno di parametri.
Un valore assegnato con un metodo `setXXX` rimane inalterato fino ad una nuova assegnazione oppure finché non viene invocato il metodo `clearParameters`

Esempio

COF_NAME	SUP_ID	PRICE	SALES	TOTAL
Colombian	101	7.99	0	0
French Roast	49	8.99	0	0
Espresso	150	9.99	0	0
Colombian Decaf	101	8.99	0	0
French Roast Decaf	49	10.75	0	0

Esempio

```
PreparedStatement updateSales;
String updateString="UPDATE COFFEES SET SALES=? "+
  "WHERE COF_NAME=?";
updateSales=conn.prepareStatement(updateString);
int[] salesForWeek={175,150,60,115,90};
String[] coffees={"Colombian","French Roast","Espresso",
  "Colombian Decaf","French Roast Decaf"};
int len=coffees.length;
for(int i=0;i<len;i++) {
  updateSales.setInt(1,salesForWeek[i]);
  updateSales.setString(2,coffees[i]);
  updateSales.executeUpdate();
}
```

Esempio

COF_NAME	SUP_ID	PRICE	SALES	TOTAL
Colombian	101	7.99	175	0
French Roast	49	8.99	150	0
Espresso	150	9.99	60	0
Colombian Decaf	101	8.99	115	0
French Roast Decaf	49	10.75	90	0

Esempio

Creiamo la tabella persona con

```
CREATE TABLE persona (
cf INT PRIMARY KEY,
nomepers VARCHAR(20),
professione VARCHAR(20),
citta VARCHAR(20) );
```

e popoliamola con

```
INSERT INTO persona VALUES (11,'aldo','fornaio','firenze');
INSERT INTO persona VALUES (12,'ugo','fabbro','napoli');
INSERT INTO persona VALUES (15,'anna','ingegnere','napoli');
```

```
String crea="CREATE TABLE persona ("+
  "cf INT PRIMARY KEY,"+
  "nomepers VARCHAR(20),"+
  "professione VARCHAR(20),"+
  "citta VARCHAR(20))";
Statement st=conn.createStatement();
st.executeUpdate(crea);
PreparedStatement pst=conn.prepareStatement("INSERT INTO "+
  "persona VALUES (?,?,?,?)");
int[] codici={11,12,15};
String[] nomi={"aldo","ugo","anna"};
String[] professioni={"fornaio","fabbro","ingegnere"};
String[] citta={"firenze","napoli","napoli"};
for(int i=0;i<3;i++) {
  pst.setInt(1,codici[i]);
  pst.setString(2,nomi[i]);
  pst.setString(3,professioni[i]);
  pst.setString(4,citta[i]);
  pst.executeUpdate();
}
```

Ora troviamo il nome delle persone che sono nate a Napoli.

```
ResultSet rs;
rs=st.executeQuery("SELECT nomepers FROM persona "+
  "WHERE citta='napoli'");
while(rs.next()) {
  n=rs.getString(1);
  System.out.println(n);
}
```

L'output prodotto sarà

```
ugo
anna
```

Note sui metodi setXXX

- Per sapere quali metodi `setXXX` usare in modo da impostare correttamente i parametri di un prepared statement, fate riferimento alla tabella di corrispondenza riportata in precedenza per i metodi `getXXX`. Valgono le stesse regole di "elasticità"
- In particolare, quando dovete inserire un valore decimale che in SQL corrisponda ad un Float, Double o Real, si consiglia l'uso di `setDouble()`

- Il metodo `setDate(Date d)` vuole in input una data con il formato 'yyyy-MM-dd' (anno, mese, giorno). Per creare un oggetto di questo tipo potete utilizzare il metodo statico `valueOf (String s)` della classe `java.sql.Date` in cui la stringa passata come parametro abbia il formato 'yyyy-MM-dd'. Ad esempio,

```
java.sql.Date data=java.sql.Date.valueOf("2000-01-12");
```

CallableStatement

- L'interfaccia `CallableStatement`, sottoclasse di `PreparedStatement`, permette di chiamare stored procedure
- Come per gli oggetti della classe `PreparedStatement`, è possibile specificare usare il punto interrogativo per indicare un parametro che sarà istanziato a run-time

CallableStatement (esempio)

```
CREATE PROCEDURE PersoneNateInCitta(IN c VARCHAR(25))
SELECT nomepers FROM persona
WHERE citta=c;

(...)
CallableStatement cstmt=
          con.prepareCall("{call PersoneNateInCitta(?)}");
ResultSet rs;
for(int i=0;i<3;i++) {
  cstmt.setString(1,citta[i]);
  rs = cstmt.executeQuery();
  while (rs.next()) {....}
}
```

Quarta fase: chiusura della connessione

Gli oggetti di tipo `Connection`, `Statement`, `ResultSet`, quando non più utilizzati, devono essere **chiusi** con il metodo `close()`.

Eccezioni

- L'esistenza di una eccezione indica che si è verificato un problema durante l'esecuzione di un programma
- Le eccezioni vengono **gestite** da codice opportuno; tale codice è "sparpagliato" nel codice dell'applicazione
- Le eccezioni servono a gestire situazioni che un metodo non è in grado di controllare

Eccezioni in Java

- Il programmatore racchiude in un blocco `try` il codice che può generare una eccezione
- Il blocco `try` è immediatamente seguito da zero o più blocchi `catch`
- Un blocco `catch` specifica i tipi di eccezioni che può gestire, e il codice che le gestisce
- Un blocco `finally` **facoltativo**, dopo l'ultimo blocco `catch`, specifica del codice che viene **sempre** eseguito

Eccezioni (cont.)

```
try
{ /* blocco try  */ }

catch(TipoEcc1 e1)
{ /* blocco catch */ }
catch(TipoEcc2 e2)
{ /* blocco catch */ }
...
catch(TipoEcc2 e2)
{ /* blocco catch */ }

finally
{ /* blocco finally */ }
```

Eccezioni (cont.)

- Quando viene lanciata una eccezione, il programma **abbandona il blocco** `try` e ricerca il gestore appropriato nei blocchi `catch`
- Se il tipo di eccezione lanciata corrisponde a quello di un blocco `catch`, allora il codice di quel blocco viene eseguito, l'esecuzione riprende dopo l'ultimo blocco `catch`
- Se non sono lanciate eccezioni nel blocco `try`, l'esecuzione riprende dopo l'ultimo blocco `catch`
- Se c'e' un blocco `finally`, questo viene **sempre** eseguito

Nota

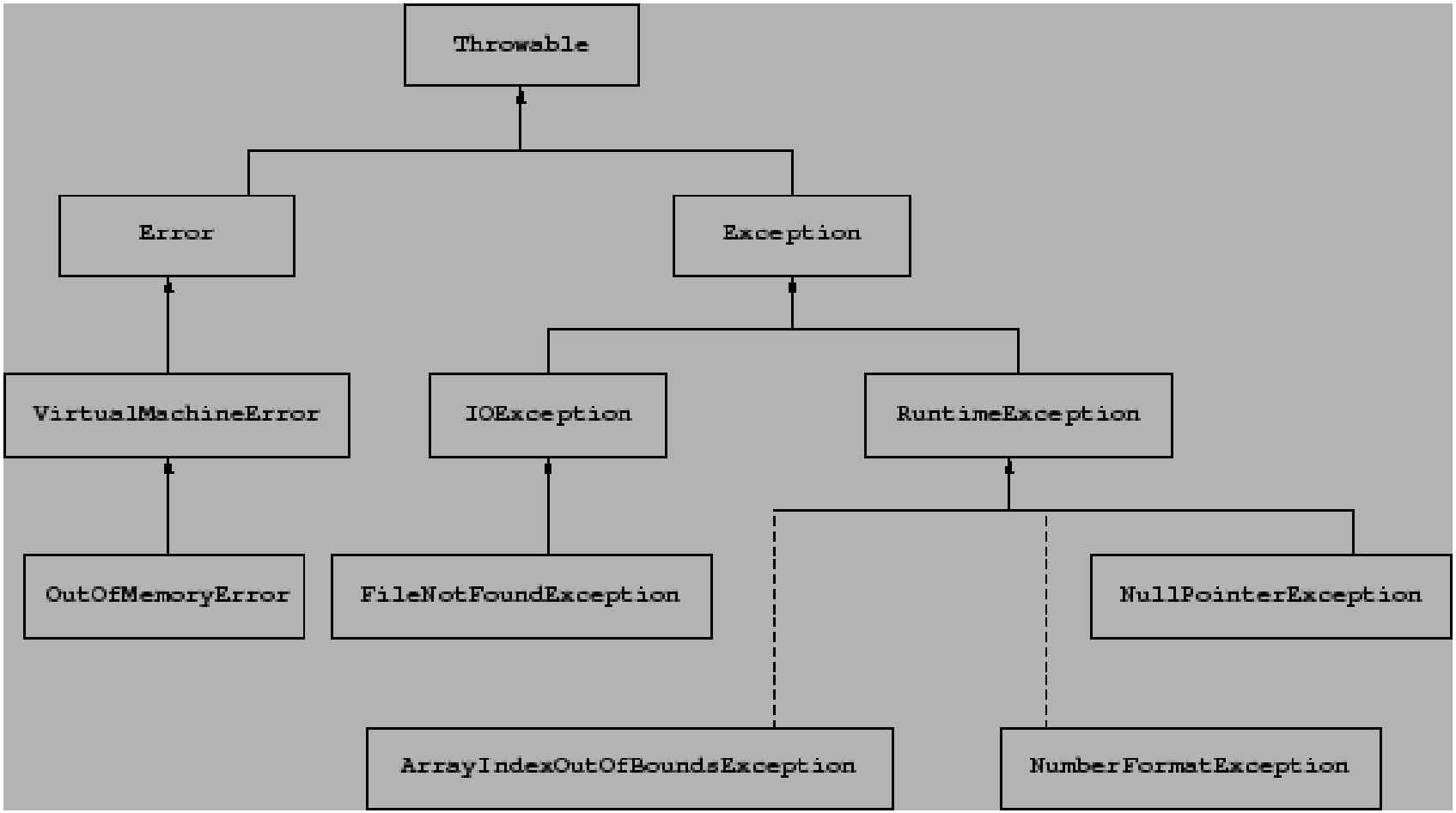
Le eventuali istruzioni del blocco `finally` vengono eseguite sempre, anche in presenza di una eccezione verificatasi nel blocco `try` che non viene catturata da alcun blocco `catch`, o in presenza di un `return` del blocco `try` o `catch`. Il blocco `finally` può contenere delle istruzioni che chiudono dei files oppure rilasciano delle risorse, per garantire la consistenza dello stato.

La clausola `throws`

- Le eccezioni che non sono state trattate tramite le istruzioni `try-catch-finally` vanno dichiarate nella clausola `throws`, facente parte del prototipo del metodo in cui l'eccezione si può verificare
- Non è necessario specificare nella clausola `throws` tutti i tipi di eccezione non "catturate"
- Non si ha quest'obbligo per le eccezioni che ereditano dalla classe `RuntimeException` (**unchecked**)
- Lo stesso dicasi per gli errori, che sono oggetti che ereditano dalla classe `Error`

- Si faccia presente che `Exception` e `Error` sono immediate sottoclassi di `throwable`, mentre `RuntimeException` è immediata sottoclasse di `Exception`
- Oggetti di tipo `Error` sono errori di sistema che non devono essere gestiti

Gerarchia delle eccezioni



Come lanciare una eccezione

- Per lanciare una eccezione si usa l'istruzione `throw`, che accetta un qualunque oggetto `Throwable`
- Si possono anche definire proprie classi di eccezioni, derivandole da `Exception`

```
MyException e;
...
throw e;
```

Le eccezioni lanciate vanno segnalate nella clausola `throws` (ovviamente questo non è necessario se si tratta di `RuntimeException`)

Cosa fare

Java richiede che quando un metodo lancia (`throws`) una eccezione deve esserci un meccanismo per gestire tale eccezione. Ci sono tre alternative:

- catturare e gestire l'eccezione;
- catturare l'eccezione e lanciare una propria eccezione dichiarata nella clausola `throws`;
- dichiarare l'eccezione nella propria clausola `throws` lasciando così che si propaghi.

Esempi: Il metodo `Class.forName` può lanciare una eccezione di tipo `ClassNotFoundException`; i metodi JDBC possono tutti lanciare una eccezione di tipo `SQLException`.

La classe `ClassNotFoundException`

- Nella gerarchia delle eccezioni è derivata direttamente dalla classe `Exception` (`java.lang.Exception`, se vogliamo indicare il suo nome completo)
- Nel nostro caso, occorre quando la java virtual machine non riesce ad istanziare la classe passata come parametro di tipo `String` al metodo `Class.forName`
- È una eccezione che deve essere verificata

La classe `SQLException`

- È derivata da `java.lang.Exception`
- Contiene:
  - una `String` che è una descrizione dell'errore, restituita da `getMessage` (metodo ereditato dalla classe `Throwable`)
  - una stringa `SQLState`, che identifica l'eccezione (Open Group SQL Specification)
  - un codice di errore, che è quello del DBMS
  - un collegamento alla eccezione successiva, nel caso ne sia verificata più di una

Esempio: SQLException

Si consideri il seguente frammento di codice, che gestisce eccezioni concatenate:

```
try
{ // codice che potrebbe generare l'eccezione
} catch (SQLException ex)
  { System.out.println("Catturata una SQLException!");
    while(ex != null)
    { System.out.println("Stato SQL: "+ex.getSQLState());
      System.out.println("Messaggio: "+ex.getMessage());
      System.out.println("Errore: "+ex.getErrorCode());
      ex=ex.getNextException();
    }
}
```

La classe `SQLException`

Metodi specifici della classe:

`public String getSQLState();` restituisce lo SQLState

`public int getErrorCode();` restituisce il codice di errore

`public SQLException getNextSQLException();` restituisce la eccezione successiva concatenata all'oggetto di invocazione

Attenzione:

- Quando viene lanciata una `SQLException` è possibile che il metodo che ha invocato l'eccezione abbia effettuato delle modifiche sui dati.
- Tali modifiche potrebbero avere portato i dati in uno stato non consistente
- Pertanto è auspicabile che il programmatore consideri la possibilità di invocare il metodo `rollback` per riportare i dati nell'ultimo stato consistente in cui essi si trovavano

Gestore della connessione

- Si possono **incapsulare** gli oggetti di tipo `Connection` all'interno di una classe, che li restituisce all'occorrenza
- Il modulo demandato a tale compito è il `ConnectionManager`
- il `ConnectionManager` restituisce ogni volta una connessione diversa

Gestore della Connessione

```
import java.sql.*;

public class ConnectionManager
{
   private ConnectionManager(){};
   private static boolean driverLoaded = false;

   private static final String MY_DRIVER =
                                    "org.postgresql.Driver";
   private static final String MY_URL =
                       "jdbc:postgres://localhost/myDatabase";
   private static final String LOGIN = "postgres";
   private static final String PASSWD = "password";
```

```
 public static Connection getConnection()
           throws ClassNotFoundException, SQLException
  {
   if(!driverLoaded)
      {
       Class.forName(MY_DRIVER);
       driverLoaded= true;
      }
   return DriverManager.getConnection(MY_URL,LOGIN,PASSWD);
  }
}// end of class ConnectionManager
```

Nel seguito viene proposto esempio che fa uso della classe ConnectionManager appena definita

Un esempio completo

```
import java.sql.*;

public class Esempio1 {
 public static void main(String[] argv)
 {
   Statement stmt = null;
   Connection con = null;
   ResultSet rs = null;
   PreparedStatement UpdatePrices = null;
```

```
// creazione di una tabella
try
{
  con = ConnectionManager.getConnection();
  stmt = con.createStatement();
  String createTableCoffees="CREATE TABLE COFFEES"+
   "(COF_NAME VARCHAR(32), SUP_ID INTEGER, PRICE FLOAT,"+
      "SALES INTEGER, TOTAL INTEGER)";
  stmt.executeUpdate(createTableCoffees);

// inserimento di tuple

 stmt.executeUpdate("INSERT INTO COFFEES VALUES "+
                         "('Kimbo', 103, 6.82, 0, 0)");
 stmt.executeUpdate("INSERT INTO COFFEES VALUES "+
                       "('Espresso', 105, 7.82, 0, 0)");
```

```
// esecuzione di una query

rs = stmt.executeQuery("SELECT * FROM COFFEES");
String s = null;
float n = 0;
while (rs.next())
 {
  s = rs.getString("COF_NAME");
  n = rs.getFloat("PRICE");
  System.out.println("COF_NAME: "+ s +
                              ", PRICE: " + n);
 }

 // aggiornamento di una tabella ed interrogazione

 UpdatePrices = con.prepareStatement("UPDATE "+
                "COFFEES SET PRICE=? WHERE COF_NAME=?");
 UpdatePrices.setDouble(1,8.2);
 UpdatePrices.setString(2,"Kimbo");
 UpdatePrices.executeUpdate();
 rs = stmt.executeQuery("SELECT * FROM COFFEES");
```

```
 while (rs.next())
  {
   s = rs.getString("COF_NAME");
   n = rs.getFloat("PRICE");
   System.out.println("COF_NAME: "+ s +", PRICE: " + n);
  }

 // rilascio delle risorse
 stmt.close();
 UpdatePrices.close();
}//chiusura del blocco try
catch(ClassNotFoundException e)
   { System.out.println("Driver NON TROVATO");}
catch(SQLException e)
   {
    System.out.println("Eccezione" + e.toString());
    e.printStackTrace();
   }
```

```
  finally
   {
    if(conn != null) {
    try {
        conn.close();
      }
    catch(SQLException ce)
     {
      System.out.println("Eccezione" + ce.toString());
      ce.printStackTrace();
     }
   }
 }
}
```

Gestore della Connessione “Statica”

Quando vogliamo usare un solo oggetto connection alla volta all’interno del nostro programma (come in realtà avviene nell’esempio precedente), è possibile utilizzare una classe Connection Manager che gestisce un oggetto statico di tipo Connection.

```
public class ConnectionManager {
 private ConnectionManager() {}
private static final String MY_DRIVER =
                                    "org.postgresql.Driver";
   private static final String MY_URL =
                        "jdbc:postgres://localhost/myDatabase";
   private static final String LOGIN = "postgres";
   private static final String PASSWD = "password";

 public static Connection getConnection()
```

```
            throws SQLException, ClassNotFoundException  {
    if(conn == null) {
       Class.forName(MY_DRIVER);
       conn = DriverManager.getConnection(MY_URL,
                                          LOGIN, PASSWD);
    } else {
       if(conn.isClosed())
          conn = DriverManager.getConnection(MY_URL,

                                          LOGIN, PASSWD);
       else {
       // In questo caso si e' verificato un comportamento
       // non atteso. Inserisco comandi che corrispondono
       // ad una politica da seguire in caso di errore

       conn.rollback(); //si e' scelto di disfare tutto il
                        //lavoro eseguito dall'ultimo commit.
        }
     return conn; }
}
```

La classe `Esempio1` vista in precedenza può usare indifferentemente uno dei due connection manager proposti.

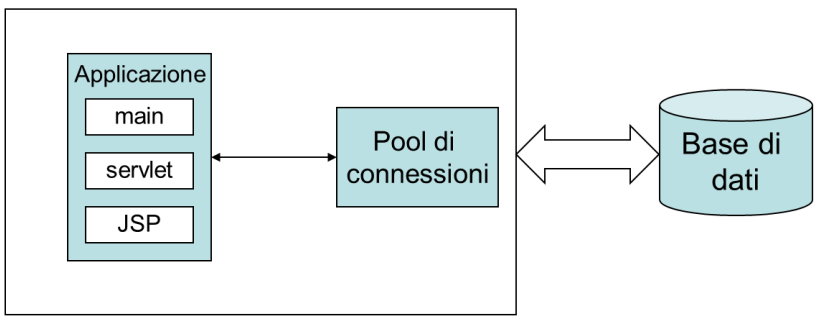
Connessione “Statica” vs. non “Statica”

– Nel caso in cui il connection manager gestisca un oggetto statico, l'applicazione non deve mai avere contemporaneamente due metodi attivi che operano sulla base dati: questi utilizzerebbero lo stesso oggetto di tipo `Connection`, e quindi, le operazioni dei due metodi interferirebbero a vicenda, compromettendo le caratteristiche transazionali dell'applicazione

\+ D'altro canto, quando l'applicazione usa più oggetti di tipo connection, è facile saturare il numero di connessioni supportate dal DBMS, provocando forti rallentamenti dell'applicazione, o addirittura causandone l'arresto.

Pool di connessioni

Una soluzione interessante che media fra la necessità di avere più di una connessione con la base di dati, ma al tempo stesso consente di non superare un numero di connessioni che garantiscano un buon livello di efficienza per l'applicazione, è la realizzazione di un pool di connessioni.



Il connection manager gestisce diverse connessioni, in genere tutte aperte alla prima richiesta da parte dell'applicazione

- ad ogni nuova richiesta viene restituita una connessione "libera" se disponibile, altrimenti si mette il richiedente "in attesa"
- quando tutti comandi SQL sono stati eseguiti, la connessione viene restituita al pool

Un'applicazione può implementarsi il proprio algoritmo di pooling, oppure usare librerie di terzi quali il software open-source C3P0.

Pool di connessioni - Vantaggi

Tre sono i principali vantaggi:

- minimizzare i costi di apertura e chiusura delle connessioni tipicamente molto costosi
- mantenere molte connessioni attive contemporaneamente può essere molto costoso per un DBMS - il pool può ottimizzare l'uso delle connessioni, e rilasciarne se non ci sono richieste
- creare prepared statement può essere costoso per alcuni driver - il pool di connessioni può usare un sistema di caching per una connessione che è mantenuto tra una richiesta e l'altra

Impostare transazioni tramite JDBC

Quando una connessione viene creata, essa è in modalità **auto-commit**. Quindi ogni esecuzione di uno statement SQL viene resa immediatamente **committed**, ed è quindi considerata come una **transazione**.
Per impostare transazioni costituite da più di un comando SQL tramite JDBC, dobbiamo modificare lo stato dell'**auto-commit** degli oggetti di tipo connection. Per fare questo usiamo il comando

```
con.setAutoCommit(false);
```

In questo modo le operazioni sono rese **committed** solo quando si chiama esplicitamente il metodo `commit()`.

Esempio

COF_NAME	SUP_ID	PRICE	SALES	TOTAL
Colombian	101	7.99	75	90
French Roast	49	8.99	32	108

All'aggiornamento della colonna SALES (vendite settimanali) deve corrispondere un opportuno aggiornamento della colonna TOTAL, altrimenti i dati non sono consistenti (si perde la **integrità**). Si vuole che i due aggiornamenti siano eseguiti sempre e solo insieme, pertanto occorre che essi vengano accorpati in una **transazione**. Per l'atomicità della transazione i due aggiornamenti verranno eseguiti entrambi o non verranno eseguiti affatto.

Esempio

Si suppone di avere già un oggetto `con` di tipo `Connection`.

```
con.setAutoCommit(false);
PreparedStatement updateSales=
  con.prepareStatement("UPDATE COFFEES"+
  "SET SALES=? WHERE COF_NAME=?");
updateSales.setInt(1,50);
updateSales.setString(2,"Colombian");
updateSales.executeUpdate();
```

```
PreparedStatement updateTotal=
  con.prepareStatement("UPDATE COFFEES" +
    "SET TOTAL=TOTAL+? WHERE COF_NAME=?");
updateTotal.setInt(1,50);
updateTotal.setString(2,"Colombian");
updateTotal.executeUpdate();
con.commit();
con.setAutoCommit(true);
```

Impostare transazioni tramite JDBC

- Come visto, l'impostazione a `false` della modalità auto-commit ci consente di racchiudere più comandi SQL all'interno di una singola transazione
- Oltre questo, vogliamo anche impostare il livello di isolamento che "riteniamo opportuno" per la transazione che stiamo definendo

Livelli di isolamento in JDBC

E' possibile impostare il livello di isolamento delle transazioni tramite JDBC con il comando:

con.setTransactionIsolation(<costante>);

Per conoscere invece il livello di isolamento si può eseguire il comando

con.getTransactionIsolation();

I livelli di isolamento sono identificati in JDBC tramite costanti intere definite nella classe `Connection`.

Livelli di isolamento supportati da JDBC

`TRANSACTION_NONE` (intero pari a 0) le transazioni non sono supportate;

`TRANSACTION_READ_UNCOMMITTED` (intero pari a 1) possono aver luogo **dirty read**, **unrepeatable read**, **phantom read**;

`TRANSACTION_READ_COMMITTED` (intero pari a 2) sono impediti i **dirty read** rispetto al livello precedente;

`TRANSACTION_REPEATABLE_READ` (intero pari a 4) sono impediti i **unrepeatable read** rispetto al livello precedente;

`TRANSACTION_SERIALIZABLE` (intero pari a 8) **dirty read**, **unrepeatable read**, **phantom read** sono impediti.

Impostare transazioni tramite JDBC

- la scelta del livello di isolamento per una transazione viene in genere effettuata bilanciando opportunamente requisiti di integrità (evitare anomalie) e di efficienza (diminuire il numero di lock sulla base dati)
- La regola che seguiremo è **garantire l'assenza di anomalie per ogni singola transazione facendo uso del meccanismo di lock più efficiente possibile**
- Ad esempio, una volta elencate le anomalie che possono interessare la transazione, se è sufficiente impostare READ COMMITTED al fine di evitarle, non ricorreremo a SERIALIZABLE

Esempio

- Nell'esempio precedente (aggiornamento contemporaneo delle vendite settimanali e delle vendite totali), dal momento che la transazione non effettua letture al suo interno, le uniche anomalie a cui può essere esposta sono di tipo `Lost Update (WW)`.
- Per garantire che tali anomalie non si verfichino è sufficiente impostare il livello di isolamento a `READ COMMITTED` (il livello `READ UNCOMMITTED` non è utilizzabile in quanto la transazione non è `READ ONLY`)

Esempio

Si consideri nuovamente la tabella COFFEES:

```
CREATE TABLE COFFEES(
COF_NAME VARCHAR(32),
SUP_ID INTEGER,
PRICE FLOAT,
SALES INTEGER,
TOTAL INTEGER);

INSERT INTO COFFEES VALUES ('Colombian', 1, 7.99, 20, 80);
INSERT INTO COFFEES VALUES ('French_Roast', 2, 10.60, 10, 100);
INSERT INTO COFFEES VALUES ('Espresso', 3, 12, 9.99, 75);
INSERT INTO COFFEES VALUES ('Colombian_Decaf', 4, 15, 20, 63);
INSERT INTO COFFEES VALUES ('French_Roast_Decaf', 5, 12.33, 20,
```

Esempio

```
import java.sql.*;
public class TransactionPairs {
 public static void main(String[] args)
  { String url="jdbc:postgres://localhost/myDatabase";
    Connection con=null;
    PreparedStatement updateSales=null;
    PreparedStatement updateTotal=null;
    String updateString="UPDATE COFFEES "+
                                "SET SALES=? WHERE COF_NAME=?";
    String updateStatement="UPDATE COFFEES "+
                                "SET TOTAL=TOTAL+?" +
                                "WHERE COF_NAME=?";
    try
     { Class.forName(...); }
    catch (ClassNotFoundException e)
     { System.err.print(e.getMessage()); }
    try
```

```
{ con=DriverManager.getConnection
      (url,"postgres","password");
  updateSales=con.prepareStatement(updateString);
  updateTotal=con.prepareStatement(updateStatement);
  int[] salesForWeek={175,150,60,155,90};
  String[] coffees={"Colombian","French_Roast",
    "Espresso","Colombian_Decaf","French_Roast_Decaf"};
  int len=coffees.length;
  con.setAutoCommit(false);
  con.setTransactionIsolation(
    Connection.TRANSACTION_READ_COMMITTED);
  for(int i=0; i<len; i++)
   { updateSales.setInt(1,salesForWeek[i]);
     updateSales.setString(2,coffees[i]);
     updateSales.executeUpdate();
     updateTotal.setInt(1,salesForWeek[i]);
     updateTotal.setString(2,coffees[i]);
     updateTotal.executeUpdate();
     con.commit();
   }//end for
 }// end try
```

```
catch (SQLException ex)
 { System.err.println("Error: "+ex.getMessage());
   if (con!=null)
    { try
       { System.err.
         println("Transaction is being "+"rolled back");
         con.rollback(); }
      catch (SQLException excep)
       { System.err.
         println("Error: "+excep.getMessage());}
   }//end if
  }// end catch

 finally
  { if (con != null)
    try
     { updateSales.close();
       updateTotal.close();
       con.setAutoCommit(true);
       con.setTransactionIsolation(
         Connection.<default-isolation-level>);
```

```
            con.close(); }
          catch (SQLException ex2)
           { System.err.println(ex2.getMessage());}
        }//end finally
  }// main
}//end class
```