Accesso ai dati da software

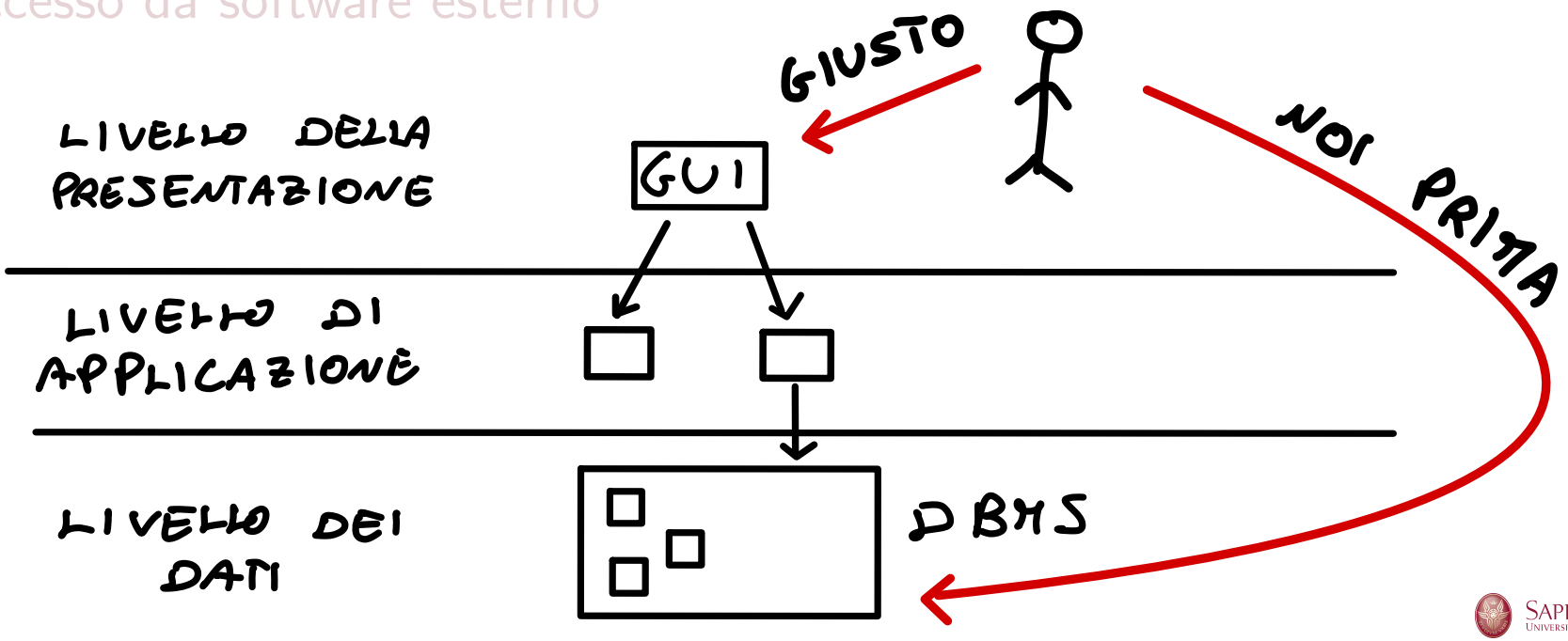
Gianluca Cima, Maurizio Lenzerini

SAPIENZA
UNIVERSITÀ DI ROMA

Anno accademico 2023/2024

Overview

Overview



Accesso da software

Finora abbiamo analizzato il caso in cui alla base di dati accede un utente umano utilizzando SQL. Nella pratica, però, nasce spesso l'esigenza di accedere ai dati da parte di moduli software, e quindi con meccanismi che sono propri dei linguaggi di programmazione. I DBMS moderni offrono due possibilità per fare questo:

1. Associare alla base di dati dei "moduli sofware" definiti all'interno del DBMS stesso e quindi nel data layer. In questo caso si parla di accesso da software interno ed i moduli possono essere scritti in un linguaggio di programmazione qualunque oppure in un linguaggio proprietario del DBMS (nel caso di PostgreSQL: PL/pgSQL). Questi moduli software sono poi invocati da utenti umani o da altri moduli software, sia interni sia esterni.
2. Accedere alla base di dati da software esterno (accesso da software esterno). Nell'architettura del software a tre livelli, questo corrisponde ad accedere al data layer dall'application layer. Siccome l'application layer è costituito da moduli software, nasce l'esigenza di accedere ai dati con meccanismi che sono propri dei linguaggi di programmazione. Tutti i linguaggi di programmazione, come ad esempio Python, Java, C, C++, consentono questo accesso. Qui noi studiamo il meccanismo offerto da Java per accedere a basi di dati relazionali, ossia quello che si chiama il protocollo JDBC.

SAPIENZA UNIVERSITÀ DI ROMA

Overview

PL/pgSQL

- Noi illustriamo le caratteristiche salienti di PL/pgSQL, che è un linguaggio di programmazione procedurale supportato da PostgreSQL. Ma esistono altri linguaggi analoghi nei vari DBMS.
- PL/pgSQL, includendo la possibilità di usare cicli e strutture di controllo avanzate, è stato creato in modo da svolgere operazioni complesse che non sono propriamente in linea con la filosofia e le potenzialità di SQL.
- I moduli sofware creati in questo linguaggio sono chiamati "function" (noti anche come **stored procedure**). Una volta che si fa compilare una function al DBMS, essa viene memorizzata dal sistema nel contesto del database in cui viene definita, e può essere chiamata in qualsiasi momento come parte di un'istruzione SQL, o attivata da un **trigger** (vedere dopo).
- Il compilatore di PL/pgSQL è installato di default su PostgreSQL.

PL/pgSQL: sintassi di base per una stored procedure

L'istruzione per creare una "function" o "stored procedure" è la CREATE FUNCTION (oppure CREATE OR REPLACE FUNCTION)

```
CREATE FUNCTION function_name (arg_1 arg_type, ..., arg_n arg_type)
RETURNS {return_type | TABLE (col_1 col_type, ...,) | TRIGGER} AS
$$
  DECLARE
    <dichiarazioni di variabili>
  BEGIN
    <istruzioni>
  END;
$$ LANGUAGE plpgsql;
```

Se *return_type* è VOID, allora la function è in realtà una procedura.

Per maggiori dettagli si veda la documentazione ufficiale: `https://www.postgresql.org/docs/9.6/static/plpgsql-structure.html`.

Stored Procedure: esempio - parte 1

Assumiamo di avere la seguente relazione:
fattura(soggetto: VARCHAR(100), imponibile: REAL, aliquota: INTEGER)
$\{\langle a, 1000.00, 22\rangle, \langle a, 30.00, 4\rangle, \langle b, 10.000, 10\rangle\}$

ottenuta tramite:

```
CREATE TABLE fattura (soggetto VARCHAR(100), imponibile REAL,
aliquota INTEGER);

INSERT INTO Fattura VALUES ('a', 1000.00, 22), ('a', 30.00, 4), ('b',
10.000, 10);
```

Supponiamo adesso di voler calcolare, dato un soggetto, una tabella con una riga che indichi tale soggetto ed il totale che esso dovrà versare di tasse. Lo possiamo fare attraverso la seguente stored procedure:

Stored Procedure: esempio - parte 2

```
CREATE OR REPLACE FUNCTION totale_fattura(sogg VARCHAR)
RETURNS TABLE (sgt VARCHAR, tot REAL) AS
$$
  DECLARE     r record; totale REAL;
  BEGIN
   totale := 0;
   FOR r IN SELECT * FROM Fattura WHERE soggetto=sogg LOOP
    totale := totale + r.imponibile*(100+r.aliquota)/100;
   END LOOP;
   CREATE TEMPORARY TABLE return_table (sgt VARCHAR, tot REAL);
   INSERT INTO return_table VALUES (sogg, totale);
   RETURN QUERY SELECT * FROM return_table;
  END;
$$ LANGUAGE plpgsql;
```

Esempio di chiamata: SELECT * from totale_fattura('a');

Se il tipo di ritorno di f(...) è VOID, la chiamata è SELECT f(...);

Triggers in PL/pgSQL

- Un trigger è un meccanismo per specificare che una certa stored procedure deve essere eseguita in maniera automatica in coincidenza di un determinato evento, in particolare l'inserimento, la cancellazione o l'aggiornamento di una tupla da una relazione.
- Viene associato ad una tabella e viene attivato in modo automatico quando il determinato evento relativo a tale tabella avviene

Sintassi di base per un trigger in PL/pgSQL:

CREATE TRIGGER *trigger_name* [BEFORE | AFTER | INSTEAD OF] [INSERT | DELETE | UPDATE] ON *table_name*

[FOR EACH ROW | FOR EACH STATEMENT] EXECUTE PROCEDURE *procedure_name*;

dove la procedura *procedure_name* deve avere come ritorno il tipo TRIGGER.

Trigger: semantica

- BEFORE | AFTER | INSTEAD OF: Un trigger può essere attivato prima (BEFORE) che l'operazione di modifica avvenga, oppure dopo (AFTER) che l'operazione sia stata eseguita, o può essere attivato in sostituzione dell'operazione stessa (INSTEAD OF, solo sulle viste).
- INSERT | DELETE | UPDATE: L'evento che attiva il trigger.
- FOR EACH ROW | FOR EACH STATEMENT:
  - Se il trigger viene definito FOR EACH ROW, allora viene attivato per ognuna delle tuple coinvolte nell'operazione di modifica;
  - se invece il trigger viene definito FOR EACH STATEMENT (default), allora viene attivato una sola volta per ciascuna operazione di modifica, indipendentemente dalle tuple coinvolte.

Trigger: osservazioni importanti

- All'interno del blocco di istruzioni di una funzione definita per un trigger (cioè il cui tipo è TRIGGER), sono disponibili alcune variabili speciali di tipo record tra cui:
  - NEW: rappresenta la nuova tupla per operazioni INSERT/UPDATE nei trigger FOR EACH ROW. Viene settato a NULL nei trigger FOR EACH STATEMENT oppure per operazioni di DELETE.
  - OLD: rappresenta la vecchia tupla per operazioni UPDATE/DELETE nei trigger FOR EACH ROW. Viene settato a NULL nei trigger FOR EACH STATEMENT oppure per operazioni di INSERT.

Trigger: osservazioni importanti

- Se il trigger è attivato da una istruzione che è a sua volta una transazione (INSERT, DELETE, UPDATE, ad esempio), le opreazioni nell'ambito del trigger sono considerate appartenenti a tale transazione.
- Se il trigger è definito BEFORE e FOR EACH ROW, allora
  - se il valore v che la procedura invocata dal trigger che opera sulla tupla t restituisce è NULL, allora l'operazione che ha scatenato il trigger non viene eseguita sulla tupla t,
  - se invece non è NULL, allora è di tipo record e in questo caso l'operazione che ha scatenato il trigger viene eseguita ed usa come tupla t proprio v.
- Se il trigger è definito AFTER o FOR EACH STATEMENT, il valore di ritorno è ignorato.

Trigger: esempio - parte 1

Assumiamo di avere i seguenti schemi:

lavoratore(codice: numeric, nome: varchar(20), cognome: varchar,
inprogetto: numeric, insindacato: varchar(20));
progetto (codice: numeric, usura: numeric, responsabile: numeric);

ed assumiamo che al momento dell'inserimento di un nuovo progetto, questo lo si debba assegnare a tutti i lavoratori che hanno il campo "inprogetto" pari a NULL, cioè i lavoratori che sono senza un progetto oppure sono tali che non si conosca il progetto a cui lavorano.

Trigger: esempio - parte 2

```
CREATE OR REPLACE FUNCTION assegna_progetto() RETURNS
TRIGGER AS
$$
  BEGIN
    UPDATE lavoratore
    SET inprogetto=NEW.codice WHERE inprogetto IS NULL;
    RETURN NEW;
  END;
$$ language plpgsql;

CREATE TRIGGER trigger_progetto AFTER INSERT ON progetto
  FOR EACH ROW EXECUTE PROCEDURE assegna_progetto();
```

Trigger, altro esempio: chiusura transitiva di un grafo

Supponiamo di avere la tabella `grafo` che rappresenta gli archi di un grafo G (f sta per "from" e t sta per "to"), definita tramite:

```
CREATE TABLE grafo (
  f varchar,      t varchar,      PRIMARY KEY (f,t)
);
```

e la tabella `reachable` definita tramite:

```
CREATE TABLE reachable (
  f varchar,      t varchar,      PRIMARY KEY (f,t)
);
```

Vogliamo calcolare, ad ogni inserimento di un arco in G (ossia di una tupla nella tabella `grafo`), la chiusura transitiva del grafo G e memorizzarla nella tabella `reachable` (ovvero, vogliamo sempre avere in `reachable` tutte e sole le tuple $\langle x, y \rangle$ tali che y è raggiungibile da x mediante un cammino nel grafo).

Trigger, altro esempio: chiusura transitiva di un grafo

Si potrebbe pensare che il calcolo della chiusura transitiva si possa esprimere in SQL. In realtà è stato dimostrato che il potere espressivo dell'algebra relazionale (e del frammento di SQL che studiamo noi) non è sufficiente per calcolare la chiusura transitiva di una relazione binaria, nel senso che *non esiste alcuna query Q in algebra relazionale (e nel frammento di SQL che noi studiato) tale che per ogni base di dati B con relazione binaria R il risultato ottenuto valutando Q rispetto a B coincide con la chiusura transitiva della relazione binaria R.*

Al contrario siccome PL/PGSQL è un linguaggio "Turing completo", la chiusura transitiva si può esprimere in PL/PGSQL.
Allora risolviamo il problema definendo una stored procedure per il calcolo della chiusura transitiva ed un opportuno trigger per lanciare tale procedura nel momento in cui al grafo aggiungiamo un arco.

Chiusura transitiva di un grafo - Parte 1

```
CREATE OR REPLACE FUNCTION compute_transitive_closure() RETURNS TRIGGER AS
$$   DECLARE edgeobject record; leastfixpoint bool; r record;
  BEGIN
   IF (NEW.f,NEW.t) not in (SELECT f,t FROM reachable)
   THEN INSERT INTO reachable VALUES (NEW.f, NEW.t);
          leastfixpoint := false;
          WHILE NOT leastfixpoint LOOP
            leastfixpoint := true;
            FOR r IN SELECT * FROM reachable LOOP
               FOR edgeobject IN SELECT * FROM reachable LOOP
                  IF (r.t = edgeobject.f)
                  THEN IF NOT EXISTS (SELECT * FROM reachable WHERE
                                              reachable.f=r.f AND reachable.t=edgeobject.t)
                          THEN leastfixpoint := false;
                                 INSERT INTO reachable VALUES (r.f, edgeobject.t);
                          END IF;
                  END IF;
              END LOOP;
             END LOOP;
          END LOOP;
     END IF;
     RETURN NULL;
  END;
$$ LANGUAGE plpgsql;
```

Chiusura transitiva di un grafo - Parte 2

A questo punto possiamo imporre che la funzione compute_transitive_closure() venga invocata ogni volta che inseriamo una tupla in `grafo`:

```
CREATE TRIGGER trigger_t_c AFTER INSERT ON grafo
  FOR EACH ROW
  EXECUTE PROCEDURE compute_transitive_closure()
```

Chiusura transitiva di un grafo - Esercizi

Si noti che nella soluzione presentata abbiamo optato per un trigger di tipo "FOR EACH ROW" e questo significa che per ogni tupla inserita nel grafo verrà invocata la stored procedure. L'aspetto non proprio soddisfacente è che essa calcola ex-novo tutta la chiusura transitiva e non solo la porzione che dipende dagli archi inseriti.

- **Esercizio 1**. Se vogliamo insistere con l'approccio di calcolare ex-novo la chiusura transitiva, allora possiamo cambiare la definizione del trigger in modo che sia di tipo "FOR EACH STATEMENT", cosicchè sia invocata una volta per statement (invece che per row). Cambiare opportunamente sia la stored procedure sia il trigger al fine di seguire questa strada.
- **Esercizio 2**. Se vogliamo cambiare approccio, possiamo confermare la decisione di definire il trigger "FOR EACH ROW", ma cambiare la stored procedure in modo che essa calcoli solo la porzione della chiusura transitiva che è rilevante rispetto alle tuple inserite. Cambiare opportunamente la stored procedure al fine di seguire questa strada.

Overview

Il framework JDBC

- L'accesso ad una base di dati relazionale dall'"application layer" avviene mediante l'uso di interfacce (API) standard per la comunicazione con DBMS, come ODBC o **JDBC** (**Java Data Base 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:
  - deve essere caricato a run-time dal programma che vuole usare il DBMS
  - traduce le chiamate alle funzioni JDBC in chiamate alle funzioni del DBMS

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, previa installazione di specifici package - noi non affronteremo questo aspetto, perché per l'accesso interno usiamo PL/pgSQL)

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 (quindi nel nostro programma dovremo inserire il comando `import java.sql.*;`)
   - 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

Un 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 si sta usando.

Elementi di JDBC

Come abbiamo già detto, JDBC è un API Java di connessione a dati residenti in DB relazionali e consiste di un insieme di elementi (classi e interfacce) scritte nel linguaggio di programmazione Java (si trovano nel package `java.sql` appunto).

I più importanti elementi di JDBC sono:

- la classe `Driver`
- la classe `DriverManager`
- l'interfaccia `Connection`
- la classe `DatabaseMetaData`
- l'interfaccia `Statement`
- l'interfaccia `ResultSet`
- l'interfaccia `ResultSetMetaData`
- l'interfaccia `PreparedStatement`
- l'interfaccia `CallableStatement`

Fasi dell'interazione con una base di dati via JDBC

L'interazione con una base di dati via JDBC prevede le seguenti fasi di esecuzione nell'ambito del programma software:

1. Caricamento dinamico della classe corrispondente al driver JDBC opportuno per la sorgente dati e registrazione del driver presso il driver manager
2. Connessione con la sorgente di dati, stabilita attraverso il driver manager
3. Sessione di interazione attraverso la connessione, in cui si definiscono gli "statement" e si eseguono tali statement comunicandoli alla sorgente
4. Chiusura della connessione

Prima fase: caricamento del driver JDBC

- I driver delle sorgenti dati sono gestiti dal **Driver Manager**, cioè una classe che opera tra il programma applicativo ed i driver
- Il caricamento del driver avviene tramite il meccanismo Java per il caricamento dinamico delle classi, cioè utilizzando il metodo `Class.forName(String s)`.

Prima fase: caricamento del driver JDBC

- Il metodo statico `forName` della classe `Class` carica dinamicamente la classe Java specificata nella stringa passata come parametro, che indica il nome completamente qualificato (cioè con il package a cui appartiene) della classe stessa.
- 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");
Class.forName("org.postgresql.Driver");
```

Seconda fase: connessione con la sorgente dati

- Per parlare ad una sorgente, ovvero un DBMS, occorre farlo attraverso una connessione. La connessione si stabilisce attraverso il metodo **statico** `getConnection` della classe `DriverManager`. Quest'ultimo restituisce un oggetto di una classe che implementa l'interfaccia `Connection` in maniera 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>`

Seconda fase: connessione con la sorgente dati

IMPORTANTE

```
Class.forName("org.postgresql.Driver");
String url="jdbc:postgresql:test";
String username="root";
String password="";
Connection conn=DriverManager.getConnection(url,
                    username,password);
```

- `postgresql` è il subprotocollo per PostgreSQL; `test` è il nome del database a cui ci si vuole connettere (può anche mancare).

Seconda fase: la classe `DatabaseMetaData`

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

```
DatabaseMetaData md=conn.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`.

Seconda fase: 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, per lo specifico DBMS che si accede, implementano 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

Terza fase: Statement

- Gli oggetti che consentono di eseguire una istruzione SQL sono istanze di classi **statement** che implementano l'interfaccia `Statement`
- Uno statement è sempre relativo ad una connessione $\Rightarrow$ serve una istanza di una connessione attiva per predisporre uno statement (metodo `createStatement` di `Connection`)

Ad esempio:

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

Terza fase: metodi importanti della classe `Statement`

- executeUpdate per eseguire comandi di creazione, eliminazione, aggiornamento
- executeQuery per eseguire interrogazioni
- Entrambi prendono in ingresso una stringa, che di fatto è la stringa corrispondente al comando SQL che si vuole passare al DBMS (senza il punto e virgola finale!) e che il DBMS eseguirà.

Terza fase: 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

Terza fase: 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 + perchè Java non ammette definizioni di stringhe su più righe

Terza fase: `executeUpdate`

Creiamo la tabella `COFFEES` con la stringa SQL di prima:

stmt.executeUpdate(createTableCoffees);

Inseriamo due tuple in `COFFEES`; si osservi che in questo caso 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 ha l'effetto di eseguire il comando rappresentato dalla stringa passata come parametro e restituisce un int pari al numero di tuple modificate. Nel caso di uno statement DDL, ad esempio la creazione di una tabella, viene restituito 0.

Terza fase: `executeQuery`

Il metodo executeQuery di `Statement` ha l'effetto di eseguire il comando `select` rappresentato dalla stringa passata come parametro e 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);
```

Terza fase: `ResultSet`

- Un oggetto di tipo ResultSet è un oggetto che consente di analizzare (come una sorta di cursore o iteratore) la sequenza di tuple che costituiscono il risultato fornito da executeQuery
- Come cursore, un oggetto di tipo ResultSet è inizialmente posizionato **prima della prima riga (tupla)** del risultato
- L'oggetto di tipo ResultSet può poi avanzare col metodo next() che, una volta invocato, sposta il cursore di una tupla in avanti e restituisce false (booleano) se non ci sono più tuple da analizzare, true altrimenti

Terza fase: ResultSet (esempio)

```
String query="SELECT COF_NAME,PRICE FROM COFFEES";
ResultSet rs=stmt.executeQuery(query);
while (rs.next()) {
   // elaborazione dei dati che si trovano
   // nella tupla sotto esame
}
```

→ Come si analizza la tupla "puntata" dall'oggetto `rs`?

Terza fase: i metodi `getXXX`

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

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

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

Terza fase: esempio di uso dei metodi

```
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?

Terza fase: 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.

Terza fase: 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()

Terza fase: 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()`.

Terza fase: nota sulla classe `java.sql.Date`

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));
```

Terza fase: metadati su un Result Set

È possibile ottenere i metadati (ad esempio, il numero di colonne, ecc.) su 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

Terza fase: esempio

Assumiamo di avere definito nella nostra base di dati la seguente tabella.

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

Terza fase: esempio di calcolo del result set

```
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?

Terza fase: esempio - output

L'output del precedente frammento di codice sarà:

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

Terza fase: 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`.

Terza fase: 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

→ entrambi restituiscono la stringa vuota se il metodo non è applicabile

Terza fase: 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**

Terza fase: Prepared Statement

Il "prepared statement" è una caratteristica dei DBMS basati su SQL. Con un prepared statement possiamo fornire ad un DBMS relazionale la definizione di un "comando parametrico", ossia di una query o di un update in cui inseriamo del punti interrogativi come parametri in posizioni in cui ci si aspetta dei dati. Tali parametri saranno poi instanziati nel momento in cui si invocherà lo statement per eseguirlo. Quando un DBMS elabora un prepared statement, non lo esegue, ma lo "compila" per sceglierne il piano di esecuzione, in modo che quando verrà invocato disporrà dell'algoritmo per eseguirlo.
In JDBC:

- 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

Terza fase: 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.

Terza fase: 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

Terza fase: 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

Terza fase: 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();
}
```

Terza fase: 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

Terza fase: esercizio

Supponendo di avere già a disposizione l'oggetto conn di tipo Connection che rappresenta una connessione attiva con la corretta base di dati gestita dal nostro DBMS, scrivere il codice Java per creare la tabella persona con attributi

cf, di tipo int (PRIMARY KEY),
nomepers, di tipo VARCHAR(20),
professione, di tipo VARCHAR(20),
citta, di tipo VARCHAR(20)

popolarla mediante un prepared statement con le tuple

(11,'aldo','fornaio','firenze')
(12,'ugo','fabbro','napoli')
(15,'anna','ingegnere','napoli')

e poi stampare, con uno statement, il nome delle persone che hanno Napoli come città.

Creiamo e popoliamo la tabella.

```
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 stampiamo il nome delle persone che hanno Napoli come città.

```
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
```

Terza fase: 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 un oggetto della classe `java.sql.Date`. 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' (anno-mese-giorno). Ad esempio,
  java.sql.Date data=java.sql.Date.valueOf("2000-01-12");

Terza fase: differenza tra statement e prepared statement

Illustreremo ora un esempio che sottolinea la differenza tra statement e prepared statement nell'utilizzo di JBDC.

Prima di farlo, ricordiamo che

- quando usiamo uno statement, noi confezioniamo una stringa che rappresenta una istruzione SQL e poi mandiamo al DBMS tale stringa allo scopo di eseguire l'istruzione.
- quando usiamo un prepared statement, la stringa che confezioniamo **non** è una istruzione completa, ma una istruzione parametrica, in cui ci sono dei simboli speciali (il ?) al posto di porzioni della istruzione che rappresentano dei valori mancanti. Quando mandiamo al DBMS la stringa, il sistema **non** esegue l'istruzione (non potrebbe, visto che non è completa), ma la "compila" al fine di derivare il cosiddetto "piano di esecuzione". L'esecuzione dell'istruzione avviene al momento in cui si chiede al DBMS di eseguire lo statement utilizzando il meccanismo che sostituisce il simbolo speciale con i valori concreti che avremo preparato.

Terza fase: differenza tra statemente e prepared statement

Riconsideriamo l'esempio visto prima trasformando le istruzioni:

```
ResultSet rs;
rs = st.executeQuery("SELECT nomepers FROM persona "+
                     "WHERE citta='napoli'");
```

in

```
acquisisci da un pagina web la stringa s inserita dall'utente
ResultSet rs;
rs = st.executeQuery("SELECT nomepers FROM persona "+
                     "WHERE citta=' " + s + " ' ");
```

Ovviamente, se l'utente ci fornisce napoli come stringa s, otteniamo esattamente lo stesso comportamento di prima, poichè il comando mandato a SQL è: SELECT nomepers FROM persona WHERE citta='napoli'.
Quindi, in queste condizioni, il programma stamperà in output ancora:

ugo
anna

Terza fase: differenza tra statemente e prepared statement

Ma supponiamo che l'utente fornisca come stringa s:

' OR 1=1 --

Adesso il comando che il programma manda ad SQL diventa:

SELECT nomepers FROM persona WHERE citta=' ' OR 1=1 -- '

Poichè il simbolo -- è usato in SQL per iniziare un commento, il compilatore SQL lo ignora e ignora tutto ciò che segue tale simbolo. Ne consegue che in questo caso il programma stamperà in output:

ugo
anna
aldo

Abbiamo così ottenuto dalla base di dati una informazione (il nome di tutte le persone) che magari non era lecito rivelare. Quello appena mostrato è un esempio di SQL injection, un tipo di attacco software molto comune che ha lo scopo di "rubare" dati privati, facendo leva sul meccanismo degli "statement" SQL invocati da software (in questo caso da JDBC).

Terza fase: differenza tra statemente e prepared statement

Consideriamo ancora il caso che l'utente fornisca come stringa s:

' OR 1=1 --

ma adesso assumiamo di avere scritto il programma in questo modo:

```
PreparedStatement pst = conn.prepareStatement("SELECT nomepers" +
                                              "FROM persona" +
                                              "WHERE citta = ?");
pst.setString(1,s);
pst.executeQuery();
```

L'ultimo comando ha l'effetto di eseguire la query
SELECT nomepers FROM persona WHERE citta= α

con α che viene istanziato con la stringa s. Questa query, ovviamente, non restituisce alcun risultato.

Abbiamo così sventato il pericolo di eseguire una query che poteva essere pericolosa rispetto alla sicurezza dei dati nella base di dati.

Terza fase: CallableStatement

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

Terza fase: CallableStatement (esempio)

Supponiamo di aver definito la seguente
CREATE FUNCTION PersoneNateInCitta(c VARCHAR(25))
...
SELECT nomepers FROM persona
WHERE citta=c;
(...)
RETURN QUERY ...

Nel programma Java possiamo scrivere:

```
CallableStatement cstmt = conn.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

Dopo le fasi:

- Prima fase: caricamento dinamico della classe corrispondente al driver JDBC
- Seconda fase: connessione con la sorgente di dati
- Terza fase: sessione di interazione attraverso la connessione, in cui si definiscono gli "statement" e si eseguono tali statement comunicandoli alla sorgente

nel momento in cui si decide che l'esigenza di accesso alla base di dati è esaurita, si procede alla quarta fase, quella in cui si chiude la connessione. Gli oggetti di tipo `Connection`, `Statement`, `ResultSet`, che evidentemente non si vogliono più utilizzare, devono essere **chiusi** con il metodo `close()`.