Java

java database connectivity -- jdbc

G. Prencipe
prencipe@di.unipi.it

Indice

- JDBC: Java Data Base Connectivity
  - connessioni a basi di dati
  - esecuzione di statement SQL
  - accesso ai result set

JDBC: Java Data Base Connectivity

- JDBC è l'interfaccia **di base** standard di Java ai database relaionali
- Fornisce un insieme completo di funzionalità per eseguire comandi SQL arbitrari su qualunque database supportato

JDBC: Java Data Base Connectivity

- Un problema fondamentale che si presenta immediatamente quando si vuole fornire qualche metodo generale per accedere a qualsiasi DB è come affrontare le diversità interne dei vari DB
- Infatti, il formato di un DB Oracle è diffrente da quello di un DB Access o MySQL
- Al fine di poter utilizzare JDBC con differenti tipi di DB è necessario avere a disposizione un software di mediazione che permetta la comunicazione tra JDBC e lo specifico DB

JDBC: Java Data Base Connectivity

- Questi software sono i **driver**
- Essi sono forniti o dal produttore stesso del DB o da terze parti
- JDBC 1.0 affida quasi tutte le operazioni a statement SQL "classici":
  - CREATE TABLE ... , CREATE VIEW ...
  - SELECT ... FROM ... WHERE ...
  - UPDATE ... SET ... WHERE ...
  - INSERT INTO ... VALUES ...
  - DELETE ...
- JDBC 2.0 (da Java 1.2) introduce un certo numero di metodi Java per manipolare direttamente i DB

JDBC e ODBC

- Prima che Java entrasse in scena, Microsoft aveva introdotto la propria soluzione al problema della compatibilità nei metodi d'accesso a diversi DB: *ODBC*
- I database supportati da JDBC includono tutti quelli supportati da ODBC (su Windows) *più* un certo numero di DB per Unix / Linux…. in pratica, JDBC supporta quasi tutto!

ODBC

- Open Database Connectivity (ODBC) è una API standard per la connessione ai DBMS
- Questa API è indipendente dai linguaggi di programmazione, dai sistemi di database, e dal sistema operativo
- ODBC si basa sulle specifiche di Call Level Interface (CLI) di SQL, X/Open (ora parte di The Open Group) e ISO/IEC
- È stata creata dall'SQL Access Group e la sua prima release risale al settembre 1992

ODBC

- ODBC è un'interfaccia nativa alla quale si può accedere tramite linguaggi che siano in grado di chiamare funzioni di librerie native
  - Nel caso di Microsoft Windows, questa libreria è una DLL
- La prima versione è stata sviluppata su Windows; altre release sono state scritte per UNIX, OS/2 e Macintosh

ODBC

- In aggiunta al software ODBC, c'è bisogno di un driver specifico per poter accedere ad ogni diverso tipo di DBMS
- ODBC permette ai programmi che lo usano di inviare ai database stringhe SQL senza che ci sia bisogno di conoscerne le API proprietarie
  - Genera automaticamente richieste che il sistema di database utilizzato sia in grado di capire
- In tal modo, i programmi possono connettersi a diversi tipi di database utilizzando più o meno lo stesso codice.

JDBC-ODBC

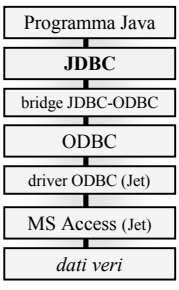
- Un JDBC-ODBC Bridge è un driver JDBC che impiega un driver ODBC per connettersi al DBMS
  - È il modo con cui JDBC può utilizzare ODBC
- Questo driver traduce le chiamate a metodi JDBC in chiamate a metodi ODBC
  - Può introdurre notevoli ritardi per DB di grosse dimensioni, dato il passaggio di conversione extra
- Il bridge, in genere, viene utilizzato quando non esiste un driver JDBC per un certo DBMS a
  - Accadeva spesso quando JDBC era ancora poco diffuso, mentre oggi è abbastanza raro
  - UnixODBC è l'implementazione ODBC più usata per piattaforme UNIX.

JDBC

- Il pacchetto che comprende il nucleo di JDBC è **java.sql**
- JDBC 3.0 comprende anche **javax.sql**

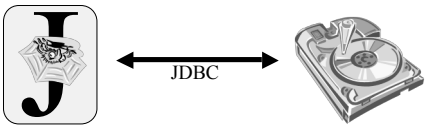
JDBC: struttura generale

- JDBC è un sistema basato su **driver**
- Un programma Java passa comandi a JDBC, che li passa a un driver specifico
- ODBC è un driver per JDBC (*JDBC-ODBC bridge*); al suo interno ODBC ha altri driver
- Il driver passa i comandi al DBMS effettivo (per esempio, Access)



JDBC: struttura generale

- Per usare JDBC, l'applicazione deve prima ***caricare un driver*** adatto al DBMS che ha i dati
- Deve poi ***aprire una connessione*** verso il database a cui si vuole accedere
- Infine, può ***mandare comandi SQL*** al database, che verranno eseguiti come se fossero stati dati dall'utente



Caricamento di un driver

- Per caricare un driver, è sufficiente conoscere il nome della classe che lo implementa
- I driver forniti con il JDK sono in `sun.jdbc` o `javax.jdbc`
  - Per esempio, il bridge JDBC-ODBC è in `sun.jdbc.odbc.JdbcOdbcDriver`
  - Questo driver funziona solo con Windows e Linux
  - Per MacOSX si può utilizzare il driver specifico
- Altri driver possono essere forniti dai singoli produttori
  - Basta scaricarli e metterli da qualche parte

Caricamento di un driver

- Sotto MacOSX una alternativa a ODBC è di utilizzare un database con i driver JDBC forniti dalla casa produttrice
  - Es.: MySql (gratuito)
- In questo caso il driver (tipicamente un JAR) va messo da qualche parte e poi caricato come jar esterno al momento della compilazione

Caricamento di un driver

- Questa fase consiste in una sola riga di codice:

Nome della classe del driver

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

Apertura della connessione

- L'apertura di una connessione è affidata a un metodo statico della classe DriverManager, che gestisce tutti i driver JDBC:

```
Connection con =
  DriverManager.getConnection(url, login, passwd);
```

- Il DB a cui connettersi è indicato tramite una URL
- È anche necessario fornire un login e una password (accettati dal DBMS)

Apertura della connessione

- Le URL di JDBC hanno il formato

protocollo	:	sotto protocollo	:	sorgente dati

- Per esempio, il database "Alberghi" raggiungibile tramite ODBC sarà indicato dalla URL

jdbc	:	odbc	:	Alberghi

- Un database Sybase "Bacheche", sulla macchina dbtest alla porta 1455

jdbc	:	sybase	:	//dbtest:1455/bacheche

Apertura della connessione

- Per esempio, per aprire una connessione verso "Alberghi":

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

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



    Connection conn=
       DriverManager.getConnection(  "jdbc:odbc:Alberghi",
                                     "pino", "topsecret");
```

Apertura della connessione
(con gestione delle eccezioni)

- Per esempio, per aprire una connessione verso "Alberghi":

```
import java.sql.*;
/* ... */
  try {
    Class.forName("sun.jdbc.odbc.JdbcOdbcDriver");
  } catch (ClassNotFoundException e) {
    /* il driver non è installato sulla macchina */
  }
  try {
    Connection conn=
       DriverManager.getConnection(  "jdbc:odbc:Alberghi",
                                     "pino", "topsecret");
  } catch (SQLException e) {
    /* errore nell'apertura del database */
  }
```

Connessioni e statement

- Una volta stabilita la connessione, è possibile usarla per inviare ***comandi*** (*statement*) SQL al database
- A questo scopo, bisogna prima creare un oggetto *Statement* per la connessione:

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

Statement

- Una volta ottenuto uno **Statement** collegato al DB tramite la connessione, possiamo usarlo per inviare comandi SQL
- Distinguiamo due tipi di comandi:
  - quelli che modificano i dati (CREATE, INSERT, UPDATE, DELETE)
  - quelli che restituiscono risultati (SELECT)
- I due tipi sono trattati da metodi differenti

Update Statement

- Sia *s* una variabile stringa che contiene un comando SQL di modifica dei dati
- Il comando SQL viene eseguito chiamando il metodo

```
stmt.executeUpdate(s);
```

Update Statement

- Esempio: supponiamo di avere un DB "Prenotazioni", con una tabella "Alberghi", accessibile via ODBC
- Vogliamo aggiungere un nuovo albergo:

```
import java.sql.*;

Class.forName("sun.jdbc.odbc.JdbcOdbcDriver");
Connection conn=
       DriverManager.getConnection("jdbc:odbc:Prenotazioni",
                                                  login,passwd);
Statement stmt=conn.createStatement();
stmt.executeUpdate(    "INSERT INTO Alberghi " +
                       "VALUES ('Miramonti', 12,'***',47.50)"  );
```

Query Statement

- Sia *s* una variabile stringa che contiene un comando SQL SELECT ... FROM ... WHERE
- Il comando viene eseguito chiamando il metodo

```
rs=stmt.executeQuery(s);
```

- I risultati della query sono in **`rs`**, che è un oggetto di classe **`ResultSet`**

Query Statement

- Esempio: vogliamo sapere quali alberghi a tre stelle costano meno di 50 euro a notte
- *(nota: una volta ottenuto uno stmt, lo si può usare per più comandi, qui ripetiamo la parte blu per chiarezza)*

```
import java.sql.*;

Class.forName("sun.jdbc.odbc.JdbcOdbcDriver");
Connection conn=
       DriverManager.getConnection("jdbc:odbc:Prenotazioni",
                                                    login,passwd);
Statement stmt=conn.createStatement();
ResultSet rs;
rs=stmt.executeQuery("SELECT * FROM Alberghi " +
                     "WHERE cat=`***' AND prezzo<50.00"  );
```

ResultSet

- ResultSet rappresenta una *tabella*
- In ogni istante, c'è una *riga corrente* su cui si opera
- All'inizio la riga corrente è posizionata prima della prima riga
- Il metodo *next()* del **ResultSet** avanza di una posizione la riga corrente
- *next()* restituisce *true* se ci sono altre righe, *false* se siamo alla fine della tabella

ResultSet

- Nota: dopo aver chiamato *next()* la prima volta, la riga corrente sarà la prima riga della tabella
- Queste convenzioni su *next()* lo rendono molto comodo per scorrere tutte le righe di un **ResultSet**:

```
ResultSet rs;
rs=stmt.executeQuery("SELECT * FROM Alberghi " +
                     "WHERE cat=`***' AND prezzo<50.00"  );

while (rs.next()) {
  /* processa la riga */
}
```

ResultSet

- Altri metodi per spostare la riga corrente:

Metodo ResultSet	Effetto
previous()	riga precedente
first()	prima riga
last()	ultima riga
beforeFirst()	prima della prima riga
afterLast()	dopo l'ultima riga
absolute(n)	riga numero *n*
relative(n)	+/- *n* righe dalla riga corrente

ResultSet

- L'accesso ai singoli campi della riga corrente avviene attraverso i metodi **`getXXX(campo)`**
- *XXX* è il tipo di dato che si vuole leggere (**String**, **int**, **float**, ...)
- *campo* è il nome della colonna oppure il numero d'ordine della colonna all'interno del **ResultSet**

ResultSet

- Esempio: stampiamo nomi e costi per i tre-stelle economici:

```
ResultSet rs;
rs=stmt.executeQuery("SELECT nome, prezzo FROM Alberghi " +
                     "WHERE cat=`***' AND prezzo<50.00"  );

while (rs.next()) {
  System.out.println(  rs.getString("nome")+ ", costa " +
                       rs.getFloat("prezzo")+ "€ a notte"  );

  /* oppure: */

  System.out.println(  rs.getString(1)+ ", costa " +
                       rs.getFloat(2)+ "€ a notte"   );
}
```

ResultSet

- Naturalmente, il metodo chiamato deve corrispondere al tipo SQL della colonna
- JDBC è molto tollerante, e consente di usare anche metodi relativi a tipi diversi, purché almeno *compatibili* con il tipo del dato
- Per esempio:
  - chiamando **getString()** su un dato intero, si ottiene la versione stringa del numero
  - chiamando **getInt()** su una stringa, JDBC converte la stringa in intero – purché la stringa rappresenti un numero

Tipo SQL / Metodo ResultSet	TINYINT	SMALLINT	INTEGER	BIGINT	REAL	FLOAT	DOUBLE	DECIMAL	NUMERIC	BIT	CHAR	VARCHAR	LONGVARCHAR	BINARY	VARBINARY	LONGVARBINARY	DATE	TIME	TIMESTAMP
getByte	✔	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓							
getShort	✓	✔	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓							
getInt	✓	✓	✔	✓	✓	✓	✓	✓	✓	✓	✓	✓							
getLong	✓	✓	✓	✔	✓	✓	✓	✓	✓	✓	✓	✓							
getFloat	✓	✓	✓	✓	✔	✓	✓	✓	✓	✓	✓	✓							
getDouble	✓	✓	✓	✓	✓	✔	✔	✓	✓	✓	✓	✓							
getBigDecimal	✓	✓	✓	✓	✓	✓	✓	✔	✔	✓	✓	✓							
getBoolean	✓	✓	✓	✓	✓	✓	✓	✓	✓	✔	✓	✓							
getString	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✔	✔	✓	✓	✓	✓	✓	✓	✓
getBytes														✔	✔	✓			
getDate											✓	✓	✓				✔		✓
getTime											✓	✓	✓					✔	✓
getTimeStamp											✓	✓	✓				✓	✓	✔
getAsciiStream											✓	✓	✔	✓	✓	✓			
getUnicodeStream											✓	✓	✔	✓	✓	✓			
getBinaryStream														✓	✓	✔			
getObject	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓

Riepilogo

- Con quanto abbiamo visto finora, possiamo già fare (quasi) **tutto** quello che si può fare con un DB
- Sappiamo collegarci a un DB ed eseguire comandi SQL arbitrari
  - CREATE, UPDATE, INSERT, DELETE, SELECT
- ... abbiamo già in mano tutte le carte che ci servono!

Riepilogo

- Uso tipico:

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

Connection conn=
       DriverManager.getConnection("jdbc:odbc:Prenotazioni",
                                                  login,passwd);
Statement stmt=conn.createStatement();

stmt.executeUpdate(upd SQL);

ResultSet rs;
rs=stmt.executeQuery(select SQL);

while (rs.next()) {
  ... rs.getDate(nome campo) ...
  ... rs.getInt(numero campo) ...
}
```

Esempio con MySql

```
public class TestMySQL {
   public static void main(String argv[]) throws Exception {
       Class.forName("com.mysql.jdbc.Driver");
       Connection conn =
       DriverManager.getConnection("jdbc:mysql:///test","","");
       Statement stmt = conn.createStatement();
       ResultSet rset = stmt.executeQuery("SELECT now();");
       while (rset.next()) {
              System.out.println(rset.getString(1));
       }
       rset.close();
       stmt.close();
       conn.close(); } }
```

Caratteristiche avanzate

- JDBC è stato progettato perché le cose semplici (es.: mandare comandi SQL) si potessero fare semplicemente
- Ci sono tuttavia molte caratteristiche avanzate che possono tornare utili
- **Non** le vediamo esaustivamente – ci limiteremo a citare l'esistenza delle caratteristiche più importanti

Gestione delle transazioni

- Capita spesso che si vogliano fare *aggiornamenti multipli atomici* a un DB
- Vogliamo che
  - **tutti gli aggiornamenti vengano effettuati**, oppure
  - **nessuno degli aggiornamenti venga effettuato**
- Come può il DB garantire questa proprietà di *atomicità* della transazione?

Gestione delle transazioni

- Quando si crea una connessione, essa è in modo *auto-commit*
  - ogni singolo comando SQL viene considerato una transazione
  - o viene completata con successo, o è come se non fosse mai stata fatta
- È possibile disabilitare l'auto-commit chiamando
  `conn.setAutoCommit(false);`

Gestione delle transazioni

- Quando auto-commit è disabilitato, deve essere il programmatore a *chiudere* le transazioni esplicitamente:
  - `conn.commit()` se tutto è andato bene
  - `conn.rollback()` se qualcosa è andato male e si vuole riportare il DB allo stato in cui era al momento dell'ultima **commit()** precedente
- È bene rimettere auto-commit a true dopo aver completato una transazione (così non ci dimentichiamo qualche transazione aperta)

Gestione delle transazioni

- Un uso tipico è dunque:

```
Statement stmt=conn.createStatement();
conn.setAutoCommit(false);
try {
      stmt.executeUpdate("CREATE TABLE ...");
      stmt.executeUpdate("INSERT ...");
      stmt.executeUpdate("INSERT ...");
      stmt.executeUpdate("UPDATE ...");
      conn.commit();
} catch (SQLException e) {
      conn.rollback();
} finally {
      conn.setAutoCommit(true);
}
```

Gestione delle transazioni

- Quando un DB viene acceduto da più applicazioni contemporaneamente, entrano in gioco altre caratteristiche delle transazioni
- Il **livello di isolamento** viene letto/impostato con `get/setTransactionIsolation()` di `conn`
  - `TRANSACTION_READ_COMMITED`
    - i dati scritti non sono visibili ad altri fino al `commit()` o al `rollback()`
  - ... e altre quattro modalità

Prepared Statement

- Se si usano certi comandi SQL frequentemente, è possibili prepararli in anticipo (pre-compilarli)
- In questi comandi, alcuni parametri possono essere sostituiti da "?", e poi rimpiazzati volta per volta con i valori opportuni
- In questo modo, si guadagna un po' di tempo, che in caso di DB ad alte prestazioni può essere importante

Prepared Statement

- Esempio

```
PreparedStatement nuovoTreStelle =
   conn.prepareStatement(
            "INSERT INTO Alberghi " +
            "VALUES (?, ?, '***', ?)"   );

nuovoTreStelle.setString(1, "Il Casone");
nuovoTreStelle.setInt(2, 40);
nuovoTreStelle.setFloat(3, 45.00);
nuovoTreStelle.executeUpdate();

nuovoTreStelle.setString(1, "Miramonti");
nuovoTreStelle.setInt(2,12);
nuovoTreStelle.executeUpdate();
```

... e analogamente per SELECT e executeQuery()

Update a **ResultSet**

- JDBC 2.0 permette di aggiornare i valori nel DB chiamando alcuni metodi del **ResultSet** restituito da una query
- In questo modo, non c'è bisogno di dare comandi SQL (INSERT, DELETE o UPDATE) separati
- Tuttavia, per applicazioni semplici è più semplice usare **executeUpdate()** come abbiamo visto

Altre estensioni in JDBC 2.0

- JDBC 2.0 permette di gestire i tipi SQL "nuovi" definiti nella versione 3 dello standard:
  - ARRAY, BLOB, CLOB, REF, tipi strutturati
- È anche possibile fare gli aggiornamenti cumulativi (*batch update*)
  - si definiscono tanti update, poi si mandano al DB tutti insieme in una sola volta: è più efficiente
- E diverse altre migliorie per il supporto a DB ad altissime prestazioni (*connection pooling*, supporto alle transazioni distribuite, ecc.)

Java

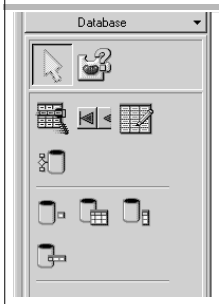
java database connectivity -- jdbc

fine

Accesso ai dati via JavaBean

- I **Data Access Bean** consentono di accedere alle basi di dati *senza* usare direttamente JDBC
- Si prestano bene alla programmazione visuale
- VisualAge ha una palette speciale per i bean "Database"

I DABean in VisualAge



- La maggior parte dei Data Access Bean sono *non-visuali* – non hanno corrispondenti visibili nell'interfaccia utente del programma
- VisualAge offre anche un bean visuale DBNavigator che offre una toolbar predefinita per navigare fra i dati

Elenco dei DABean

	Bean	Vis	Descrizione
Comandi SQL	Select	No	Esegue un comando SQL SELECT
	Modify	No	Esegue un comando SQL INSERT, UPDATE o DELETE
	ProcedureCall	No	Chiama una procedura SQL memorizzata nel DB
Selezione dati	CellSelector	No	Seleziona una cella da una tabella
	ColumnSelector	No	Seleziona una colonna da una tabella
	RowSelector	No	Seleziona una riga da una tabella
	CellRangeSelector	No	Seleziona un gruppo di celle da una tabella
GUI	DBNavigator	Si	Toolbar di controlli visuali

DB Access Class

- Il ruolo del driver, della connessione e dello statement sono raccolti con i DABean in una **DB Access Class**
- Nei diversi bean, una proprietà seleziona una DB Access Class, una connessione al suo interno, e un comando SQL da eseguire
- VisualAge crea per voi una DB Access Class se necessario – vi basta fornire:
  1. un nome per la DB Access class
  2. un nome per la connessione
  3. una URL di tipo jdbc:...

Bean Select

- Il bean **Select** consente di eseguire comandi SQL SELECT; include sia la *query* che il suo ResultSet
  - le query possono essere composte in maniera assistita
  - è possibile specificare dei parametri (`:par`), il cui valore viene assegnato separatamente – tramite una proprietà
- Il bean ha due proprietà per ogni colonna del ResultSet, il cui valore dipende dai valori contenuti nella riga corrente
  - versione nativa e versione String
- L'intero bean implementa TableModel, e può essere usato con JTable (tabella Swing)
- Modifiche alle proprietà del bean vengono salvate nel DB

Bean Modify

- Tutti i comandi che causano modifiche ai dati sono implementati dal bean Modify
  - UPDATE, INSERT, DELETE
- La proprietà *action* si comporta come la *query* del bean Select
- Il bean Modify si usa principalmente per apportare modifiche al DB *senza* aver prima eseguito una query tramite il bean Select

Bean Selezione

- Tutti i bean di selezione servono a estrarre un sottoinsieme dal ResultSet di un bean Select
  - CellSelector, ColumnSelector, RowSelector, CellRangeSelector
- Usati soprattutto per facilitare la connessione (visuale) con altri bean
- Per esempio:
  - possiamo fare una query da un DB, poi estrarre una sola colonna con ColumnSelector, e usare questa colonna per definire il contenuto di una *combo box*

Bean DBNavigator

- DBNavigator fornisce una toolbar con pulsanti per:
  - eseguire *query* o *action* dei bean collegati
  - spostare la riga corrente in un ResultSet
    - avanti, indietro, prima, ultima
  - aggiungere o cancellare una riga
  - fare un aggiornamento dei risultati di una query
  - eseguire una *stored procedure*
  - fare il *commit()* o il *rollback()* di una transazione
- Ogni pulsante può essere mostrato o nascosto
- La proprietà *model* del DBNavigator va collegata al Bean su cui si vuole navigare
  - tipicamente, un bean Select o uno dei *Selector

DABean in VisualAge

- Come al solito, il funzionamento dei Bean e il loro uso visuale è più facilmente dimostrato con un esperimento *in vivo*
- Realizzeremo con VisualAge una semplice applicazione Java per fare ricerche mirate su un DB
- Naturalmente, questi Bean si possono usare anche nell'ambito della programmazione tradizionale!