Lezione 19

Accesso ai database con JDBC

JDBC (non è una sigla, secondo quanto sostenuto da Sun, anche se molti la interpretano come *Java DataBase Connectivity*) è un'interfaccia di programmazione che lavora da tramite tra codice Java e database. Più in particolare, JDBC racchiude una serie di classi che permettono l'accesso ad una base di dati mediante metodi e schemi di funzionamento che sono intuitivi e perfettamente in linea con lo stile di programmazione tipico del linguaggio di Sun. In sostanza, quindi, è possibile connettersi ad un particolare database sfruttando un apposito driver JDBC, costituito da una classe Java. Tutti i principali DBMS dispongono oramai di un driver JDBC appositamente studiato. Esiste poi un particolare driver, chiamato *ponte JDBC-ODBC*, che permette l'utilizzo di qualsiasi fonte di dati per la quale è disponibile un driver ODBC. Ogni DBMS dotato di un'interfaccia ODBC, ad esempio Microsoft Access, può così essere immediatamente sfruttato da Java e JSP, senza la necessità di un driver appositamente studiato per la connettività da applicazioni Java. Tra JDBC ed il ponte ODBC, quindi, Java è virtualmente dotato della possibilità di interagire con tutti i DBMS in circolazione.

19.1 - Lavorare con JDBC

L'impiego di JDBC è semplice, e solitamente si articola attraverso quattro passi:

1. Per prima cosa, è necessario caricare il driver idoneo per l'utilizzo del particolare database che si intende sfruttare. Può essere caricato un apposito driver JDBC installato in precedenza nel sistema, oppure può essere sfruttato il ponte JDBC-ODBC. Non è importante il nome o il funzionamento interno del particolare driver selezionato: l'interfaccia di programmazione sarà sempre la medesima.
2. Si apre una connessione verso il particolare database necessario all'applicazione, sfruttando il driver caricato al passo precedente.
3. Si impiegano l'interfaccia di JDBC ed il linguaggio SQL per interagire con la base di dati. Generalmente, viene sottoposta al DBMS una query volta all'ottenimento di alcuni risultati.
4. I risultati ottenuti possono essere manipolati sfruttando le classi JDBC e del codice Java studiato per il compito.

Questo corso non si occupa di alcun particolare DBMS. Tutto quello che sarà fornito in questa sede, pertanto, sarà un approccio generico a JDBC e all'impiego dei database, attraverso la descrizione del pacchetto *java.sql* e l'utilizzo di alcuni celebri DBMS. Saranno inoltre descritte le più basilari istruzioni di SQL.

19.2 - Un test con MySQL

Il primo test sarà effettuato servendosi di MySQL, un noto DBMS Open Source. Per prima cosa, dunque, è necessario installare tale software nella propria postazione di lavoro. MySQL è disponibile per diverse piattaforme. Il download può essere eseguito gratuitamente partendo dall'indirizzo http://www.mysql.com/.
MySQL dispone sia di un driver JDBC sia di un driver ODBC. Per questo primo test sarà impiegato il driver JDBC chiamato *Connector/J*, che può essere prelevato dalla pagina http://dev.mysql.com/downloads/connector/j/. Affinché tale driver possa essere visto dalle applicazioni Java, è necessario notificare alla macchina virtuale la presenza dell'archivio JAR contenuto nel pacchetto scaricato. Per far ciò, è possibile operare in diversi modi. Il modo più semplice è depositare una copia dell'archivio JAR di Connector/J all'interno della

directory *jre/lib/ext*, nella cartella che contiene l'installazione di Java. Un'altra efficace tecnica è l'aggiunta del percorso dell'archivio JAR nella variabile di sistema *CLASSPATH*.
Una volta completate le operazioni preliminari, è necessario realizzare un database MySQL utile per il test. Si attivi il DBMS, quindi ci si connetta ad esso servendovi dell'apposito programma da riga di comando contenuto nella cartella *bin* di MySQL (*mysql.exe*, per la versione Windows). Il primo comando da impartire è il seguente:

```
CREATE DATABASE javatest;
```

Quindi, si selezioni il database per l'uso appena creato:

```
USE javatest;
```

Si realizzi la tabella che sarà impiegata per il test, sfruttando il seguente codice SQL:

```
CREATE TABLE Persone (
  Nome      VARCHAR (50) NOT NULL,
  Cognome   VARCHAR (50) NOT NULL,
  Indirizzo VARCHAR (50) NOT NULL
);
```

Infine, è necessario popolare la tabella con alcuni record esemplificativi:

```
INSERT INTO Persone (
  Nome, Cognome, Indirizzo
) VALUES (
  'Mario', 'Rossi', 'Via Roma 25'
);

INSERT INTO Persone (
  Nome, Cognome, Indirizzo
) VALUES (
  'Luigi', 'Bianchi', 'Via Milano 38'
);

INSERT INTO Persone (
  Nome, Cognome, Indirizzo
) VALUES (
  'Antonio', 'Verdi', 'Via Genova 12'
);
```

Giunti a questo punto si è ottenuta una tabella *Persone* come quella mostrata di seguito:

Nome	***Cognome***	***Indirizzo***
Mario	Rossi	Via Roma 25
Luigi	Bianchi	Via Milano 38
Antonio	Verdi	Via Genova 12

Ci si prepari per utilizzare la tabella da una pagina JSP esemplificativa. Il codice necessario è riportato di seguito:

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

```
public class JDBCTest1 {

  public static void main(String[] args) {
    // Nome del driver.
    String DRIVER = "com.mysql.jdbc.Driver";
    // Indirizzo del database.
    String DB_URL = "jdbc:mysql://localhost:3306/javatest";
    try {
      // Carico il driver.
      Class.forName(DRIVER);
    } catch (ClassNotFoundException e1) {
      // Il driver non può essere caricato.
      System.out.println("Driver non trovato...");
      System.exit(1);
    }
    // Preparo il riferimento alla connessione.
    Connection connection = null;
    try {
      // Apro la connesione verso il database.
      connection = DriverManager.getConnection(DB_URL);
      // Ottengo lo Statement per interagire con il database.
      Statement statement = connection.createStatement();
      // Interrogo il DBMS mediante una query SQL.
      ResultSet resultset = statement.executeQuery(
          "SELECT Nome, Cognome, Indirizzo FROM Persone"
      );
      // Scorro e mostro i risultati.
      while (resultset.next()) {
        String nome = resultset.getString(1);
        String cognome = resultset.getString(2);
        String indirizzo = resultset.getString(3);
        System.out.println("Lette informazioni...");
        System.out.println("Nome: " + nome);
        System.out.println("Cognome: " + cognome);
        System.out.println("Indirizzo: " + indirizzo);
        System.out.println();
      }
    } catch (SQLException e) {
      // In caso di errore...
    } finally {
      if (connection != null) {
        try {
          connection.close();
        } catch (Exception e) {
        }
      }
    }
  }

}
```

Si passeranno ora velocemente in rassegna le operazioni effettuate.

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

Con questa riga viene reso visibile il contenuto del pacchetto *java.sql*, che contiene le API JDBC.

```
// Nome del driver.
```

```
String DRIVER = "com.mysql.jdbc.Driver";
// Indirizzo del database.
String DB_URL = "jdbc:mysql://localhost:3306/javatest";
```

Qui vengono dichiarate due stringhe. La prima tiene traccia del nome del driver JDBC di cui ci si dovrà servire. Nel caso di Connector/J, installato in precedenza, il percorso da indicare è proprio *com.mysql.jdbc.Driver*. Con la stringa *DB_URL*, invece, si tiene traccia del database cui connettersi. La sintassi desiderata da Connector/J è la seguente:

```
jdbc:mysql://[hostname][:port]/[dbname][?param1=value1][&param2=value2]...
```

In questo caso, si è supposto che MySQL sia in esecuzione sulla stessa macchina che eseguirà la classe (*localhost*) e che la porta ascoltata dal DBMS sia la numero 3306 (la predefinita di MySQL). Quindi, è stato specificato il nome del database con cui lavorare (*javatest*, creato poco sopra). Non sono stati impiegati parametri aggiuntivi. Se il database realizzato è stato protetto con una coppia di dati username-password, bisognerà impiegare una lista di parametri così organizzata:

```
jdbc:mysql://localhost:3306/JSPTest?user=nome&password=pass
```

In definitiva, il percorso specificato nel codice di esempio deve essere variato in base alle esigenze.

```
Class.forName(DRIVER);
```

Questa istruzione carica in memoria il driver, in modo che possa poi essere sfruttato dal codice Java che segue. E' necessario circondare la chiamata con una struttura *try ... catch*. Una *java.lang.ClassNotFoundException*, infatti, può essere propagata nel caso in cui non sia possibile trovare il driver nel *CLASSPATH* della macchina virtuale.

```
Connection connection = null;
```

Viene preparato il riferimento ad un oggetto *java.sql.**Connection***, necessario per stabilire una connessione con il DBMS. Tutto quello che segue è racchiuso in un blocco *try ... catch*, poiché a rischio di eccezione: i disguidi, infatti, sono sempre in agguato (il DBMS, ad esempio, potrebbe essere momentaneamente non disponibile).

```
connection = DriverManager.getConnection(DB_URL);
```

Questa istruzione stabilisce la connessione ricercata, servendosi del metodo statico *getConnection()* della classe *java.sql.**DriverManager***.

```
Statement statement = connection.createStatement();
```

Questa riga prepara ed ottiene un oggetto *java.sql.**Statement***. Questo tipo di oggetto è necessario per interagire con il DBMS attraverso delle query espresse servendosi del linguaggio SQL.

```
ResultSet resultset = statement.executeQuery(
  "SELECT Nome, Cognome, Indirizzo FROM Persone"
);
```

Una query SQL che recupera l'intero contenuto della tabella *Persone* viene sottoposta al

DBMS. I risultati saranno conservati in un oggetto di tipo *java.sql.**ResultSet***.

```
while (resultset.next()) { ... }
```

Questo ciclo scorre l'insieme dei risultati ottenuti. Il corpo del ciclo sarà ripetuto tante volte quanti sono i record restituiti da DBMS. Il metodo *next()* di un oggetto *RecordSet*, infatti, ha duplice funzionalità: scorre in avanti l'elenco dei record ottenuti e restituisce *true* fin quando ci non si giunge al termine dell'esplorazione.

```
String nome       =  resultset.getString(1);
String cognome    =  resultset.getString(2);
String indirizzo  =  resultset.getString(3);
```

All'interno del ciclo *while*, i singoli campi della tabella *Persone* vengono recuperati e memorizzati all'interno di apposite stringhe.

```
System.out.println("Nome: " + nome);
System.out.println("Cognome: " + cognome);
System.out.println("Indirizzo: " + indirizzo);
```

Questo codice mostra i risultati in output.

```
if (connection != null) {
  try {
    connection.close();
  } catch (Exception e) {
  }
}
```

La connessione, al termine di ogni operazione (clausola *finally*) viene chiusa, se ancora attiva.

L'output che ci si aspetta di ricevere all'esecuzione del programma è:

```
Lette informazioni...
Nome: Mario
Cognome: Rossi
Indirizzo: Via Roma 25

Lette informazioni...
Nome: Luigi
Cognome: Bianchi
Indirizzo: Via Milano 38

Lette informazioni...
Nome: Antonio
Cognome: Verdi
Indirizzo: Via Genova 12
```

19.3 - Un test con Microsoft Access

Si andrà ora a sperimentare una connessione JDBC-ODBC, servendoci di un database Microsoft Access. Ovviamente, per mettere in pratica l'esperimento, è necessario servirsi di un sistema Windows, giacché Access è un DBMS disponibile esclusivamente per questa piattaforma. Si replichi quanto fatto per MySQL, realizzando con Access un file *javatest.mdb* che contenga una tabella *Persone* così strutturata:

1. *Nome*, di tipo *Testo.*
2. *Cognome*, di tipo *Testo.*
3. *Indirizzo*, di tipo *Testo*.

La tabella deve essere popolata con i medesimi dati mostrati nel paragrafo precedente. In questo caso, non è necessario servirsi della riga di comando, giacché Access dispone nativamente di un'interfaccia grafica che semplifica la creazione e l'utilizzo di un database.

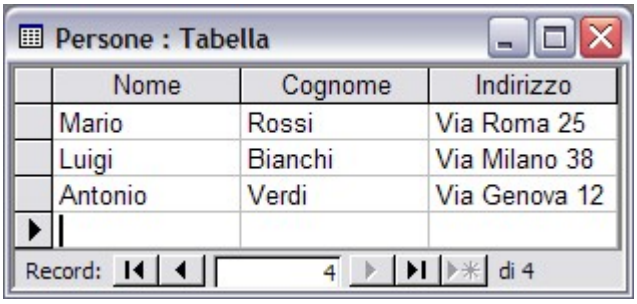

Figura 19.1
Creare e popolare un database Access è molto semplice.

A questo punto, è necessario registrare il file *javatest.mdb* come sorgente ODBC. Per far ciò, si deve sfruttare l'apposita utility disponibile in Windows, solitamente nominata "Origine Dati (ODBC)". Nei sistemi di tipo 2000/XP è possibile reperire tale voce tra gli strumenti di amministrazione. Si avvii il tool, quindi si aggiunga un nuovo riferimento nella scheda “DSN di sistema”. Si selezioni il driver “Microsoft Access Driver (*.mdb)”, si associ al DSN il nome “javatest” e quindi si selezioni il file *javatest.mdb*. Si dovrebbe ottenere una scheda come quella mostrata in **Figura 19.2**. Si confermi ogni operazione.

Figura 19.2
La finestra di Windows XP utile per la creazione di una nuova origine dati ODBC.

Il codice della classe da impiegare per il test è riportato di seguito:

```
import java.sql.*;

public class JDBCTest2 {

  public static void main(String[] args) {
```

```
    // Nome del driver.
    String DRIVER = "sun.jdbc.odbc.JdbcOdbcDriver";
    // Indirizzo del database.
    String DB_URL = "jdbc:odbc:javatest";
    try {
      // Carico il driver.
      Class.forName(DRIVER);
    } catch (ClassNotFoundException e1) {
      // Il driver non può essere caricato.
      System.out.println("Driver non trovato...");
      System.exit(1);
    }
    // Preparo il riferimento alla connessione.
    Connection connection = null;
    try {
      // Apro la connesione verso il database.
      connection = DriverManager.getConnection(DB_URL);
      // Ottengo lo Statement per interagire con il database.
      Statement statement = connection.createStatement();
      // Interrogo il DBMS mediante una query SQL.
      ResultSet resultset = statement.executeQuery(
          "SELECT Nome, Cognome, Indirizzo FROM Persone"
      );
      // Scorro e mostro i risultati.
      while (resultset.next()) {
        String nome = resultset.getString(1);
        String cognome = resultset.getString(2);
        String indirizzo = resultset.getString(3);
        System.out.println("Lette informazioni...");
        System.out.println("Nome: " + nome);
        System.out.println("Cognome: " + cognome);
        System.out.println("Indirizzo: " + indirizzo);
        System.out.println();
      }
    } catch (SQLException e) {
      // In caso di errore...
    } finally {
      if (connection != null) {
        try {
          connection.close();
        } catch (Exception e) {
        }
      }
    }
  }

}
```

Come è possibile osservare, l'unica reale differenza con il codice sviluppato per la gestione del database di MySQL risiede nella parte:

```
// Nome del driver.
String DRIVER = "sun.jdbc.odbc.JdbcOdbcDriver";
// Indirizzo del database.
String DB_URL = "jdbc:odbc:javatest";
```

JDBC, infatti, espone un'interfaccia generica che permette di lavorare alla medesima maniera con tutti i DBMS. In questo caso, infatti, è stato sufficiente specificare il percorso del driver JDBC-ODBC *(sun.jdbc.odbc.JdbcOdbcDriver*) e l'URL del database Access

precedentemente associato ad un DSN di sistema (*jdbc:odbc:javatest*).

Connessioni DSN-less con Access e JDBC-ODBC
Con Access, evitando la definizione di un DSN, è addirittura possibile stabilire connessioni che puntino direttamente ad un file MDB. Basta sfruttare un URL analogo al seguente:

```
String DB_URL = "";
DB_URL += "jdbc:odbc:";
DB_URL += "driver={Microsoft Access Driver (*.mdb)};"
DB_URL += "dbq=C:\\NomeCartella\\JSPTest.mdb;";
```

19.4 - Connessione ad un database, tutti i particolari

Per connettersi ad un database è prima di tutto necessario caricare in memoria il driver corrispondente, affinché questo sia già disponibile nel momento in cui si richiederanno ulteriori servizi. La sintassi per effettuare l'operazione è la seguente:

```
Class.forName(stringa_driver);
```

A questo punto entrano in gioco l'interfaccia *java.sql.**Connection*** e la classe *java.sql.**DriverManager***. La prima descrive le funzionalità necessarie per entrare in comunicazione con uno specifico database, mentre *DriverManager* offre una serie di metodi statici, utili per stabilire qualsiasi tipo di connessione consentita dai driver JDBC già caricati in memoria. Il modello generalmente osservato dai programmatori Java è il seguente:

```
Connection conn = DriverManager.getConnection(url_database);
```

Una volta ottenuta una connessione attiva, diventa possibile sfruttare i metodi descritti da *Connection*. I più frequentemente utilizzati sono:

- ***close()***. Chiude la connessione.
- ***createStatement()***. Crea e restituisce un oggetto *java.sql.Statement*, utile per interagire con il database mediante dei comandi SQL.

19.5 - L'interfaccia Statement

Il linguaggio SQL comprende istruzioni utili per interagire con una base di dati. In particolare, mediante SQL è possibile compiere tre principali operazioni:

1. Eseguire selezioni e ricerche all'interno di una o più tabelle, con l'istruzione *SELECT*.
2. Modificare il contenuto di una tabella, con istruzioni come *DELETE*, *INSERT* e *UPDATE*.
3. Modificare la struttura del database, ad esempio con *CREATE TABLE*.

L'interfaccia *java.sql.Statement* comprende i metodi necessari per fornire al DBMS le istruzioni SQL appena descritte:

- ***executeQuery()*** commissiona le istruzioni di tipo *SELECT*.
- ***executeUpdate()*** commissiona le istruzioni di aggiornamento delle tabelle (*DELETE*, *INSERT* e *UPDATE*) e della base di dati (*CREATE TABLE*, *CREATE*

INDEX e così via).

Il metodo ***executeQuery()*** restituisce sempre un oggetto che implementa l'interfaccia *java.sql.ResultSet*. Grazie ad essa è possibile prendere in esame i risultati restituiti dall'istruzione SQL di ricerca commissionata ad DBMS. ***executeUpdate()***, al contrario, non ha risultati da restituire. Nonostante questo, è possibile ottenere indietro un intero che riporta il numero delle righe coinvolte dall'esecuzione di istruzioni di tipo *DELETE*, *INSERT* e *UPDATE*. Negli altri casi, dove realmente non c'è nulla da restituire, tale valore di ritorno sarà sempre 0 (zero).

19.6 - L'interfaccia ResultSet

L'interfaccia *java.sql.ResultSet* comprende i metodi indispensabili per scorrere l'insieme dei risultati restituiti da una query SQL. Il metodo ***next()*** scorre in avanti tale insieme. Si supponga di aver eseguito una query che restituisce due record. Inizialmente, il cursore del corrente oggetto *ResultSet* sarà posizionato antecedentemente al primo dei due record restituiti. In questa condizione, non è possibile svolgere operazioni di analisi dei risultati, giacché nessun record è puntato dal cursore corrente. Una prima chiamata a *next()* farà in modo che il cursore venga spostato sul primo record restituito dalla query. Ogni volta che un record è puntato dal cursore, diventa possibile estrapolarne i contenuti. Una seconda chiamata a *next()* sposterà il cursore in avanti di una posizione. A questo punto, il secondo (ed ultimo) record ottenuto potrà essere esaminato ed utilizzato. Una terza chiamata a *next()* porterà il cursore oltre l'ultimo record disponibile. Si ritorna, così, ad una condizione simile a quella iniziale, quando nessun record era puntato. Il metodo *next()* fornisce un'ulteriore funzionalità: restituisce un valore booleano che è *true* quando un record è puntato, *false* in caso contrario. In termini pratici, un intero *ResultSet* può essere passato in rassegna con un codice del tipo:

```
while (resultSet.next()) {
  // Esamina il record corrente.
}
```

Un ciclo di questo tipo termina non appena tutti i record restituiti dalla query eseguita sono stati passati in rassegna.
Quando un record è correttamente puntato dal cursore, è possibile esaminare i suoi campi attraverso dei metodi che hanno tutti la forma:

```
getTipo(int indiceColonna)
```

Ad esempio, si supponga di voler ottenere il contenuto del primo campo del record corrente, sotto forma di stringa:

```
String stringa = resultSet.getString(1);
```

Se si conoscono i nomi associati ai singoli campi del record, è possibile usare la variante:

```
getTipo(String nomeColonna)
```

Ad esempio:

```
String stringa = resultSet.getString("Nome");
```

Il seguente elenco riporta i metodi di questa famiglia più frequentemente utilizzati:

- ***getBoolean()***. Restituisce il campo specificato sotto forma di *boolean*.
- ***getByte()***. Restituisce il campo specificato sotto forma di *byte*.
- ***getDate()***. Restituisce il campo specificato sotto forma di oggetto *java.util.Date*.
- ***getDouble()***. Restituisce il campo specificato sotto forma di *double*.
- ***getFloat()***. Restituisce il campo specificato sotto forma di *float*.
- ***getInt()***. Restituisce il campo specificato sotto forma di *int*.
- ***getLong()***. Restituisce il campo specificato sotto forma di *long*.
- ***getShort()***. Restituisce il campo specificato sotto forma di *short*.
- ***getString()***. Restituisce il campo specificato sotto forma di oggetto *java.lang.String*.

E i campi con valore NULL?
Se un campo ha valore *NULL* è possibile saperlo con un codice come:

```
String dato = resultSet.getString("NomeColonna");
boolean datoNull = resultSet.wasNull();
```

Il metodo *wasNull()* restituisce *true* se l'ultimo campo letto dal *ResultSet* è *NULL*. Quindi, stando all'esempio appena visto, sarà possibile conoscere l'eventuale nullità del dato letto consultando il contenuto del booleano *datoNull*.

19.7 - Un esempio di ricerca

Il codice riportato di seguito sfrutta il database di MySQL descritto in precedenza ed esegue una ricerca al suo interno. A questo punto, tutte le righe dovrebbero risultare di semplice comprensione:

```
import java.sql.*;

public class JDBCTest3 {

  public static void main(String[] args) {
    // Nome del driver.
    String DRIVER = "com.mysql.jdbc.Driver";
    // Indirizzo del database.
    String DB_URL = "jdbc:mysql://localhost:3306/javatest";
    try {
      // Carico il driver.
      Class.forName(DRIVER);
    } catch (ClassNotFoundException e1) {
      // Il driver non può essere caricato.
      System.out.println("Driver non trovato...");
      System.exit(1);
    }
    // Preparo il riferimento alla connessione.
    Connection connection = null;
    try {
      // Apro la connesione verso il database.
      connection = DriverManager.getConnection(DB_URL);
      // Ottengo lo Statement per interagire con il database.
      Statement statement = connection.createStatement();
      // Interrogo il DBMS mediante una query SQL.
      ResultSet resultset = statement.executeQuery(
          "SELECT Nome, Cognome, Indirizzo FROM Persone " +
          "WHERE Nome = 'Mario'"
      );
      // Scorro e mostro i risultati.
```

```
      while (resultset.next()) {
        String nome = resultset.getString(1);
        String cognome = resultset.getString(2);
        String indirizzo = resultset.getString(3);
        System.out.println("Lette informazioni...");
        System.out.println("Nome: " + nome);
        System.out.println("Cognome: " + cognome);
        System.out.println("Indirizzo: " + indirizzo);
        System.out.println();
      }
    } catch (SQLException e) {
      // In caso di errore...
    } finally {
      if (connection != null) {
        try {
          connection.close();
        } catch (Exception e) {
        }
      }
    }
  }

}
```

19.8 - Aggiunta di un nuovo record

Il seguente codice aggiorna la tabella *Persone* del database in uso, creando automaticamente un nuovo record.

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

public class JDBCTest4 {

  // Nome del driver.
  private static final String DRIVER = "com.mysql.jdbc.Driver";

  // Indirizzo del database.
  private static final String DB_URL = "jdbc:mysql://localhost:3306/javatest";

  // Questo metodo aggiunge un nuovo record alla tabella nel DB.
  private static boolean aggiungiRecord(String nome, String cognome,
      String indirizzo) {
    // Preparo il riferimento alla connessione.
    Connection connection = null;
    try {
      // Apro la connesione verso il database.
      connection = DriverManager.getConnection(DB_URL);
      // Ottengo lo Statement per interagire con il database.
      Statement statement = connection.createStatement();
      // Aggiungo il nuovo record.
      statement.executeUpdate(
          "INSERT INTO Persone (      " +
          "  Nome, Cognome, Indirizzo " +
          ") VALUES (                 " +
          "  '" + nome + "',          " +
          "  '" + cognome + "',       " +
          "  '" + indirizzo + "'      " +
          ")"
      );
      return true;
```

```
    } catch (SQLException e) {
      // In caso di errore...
      return false;
    } finally {
      if (connection != null) {
        try {
          connection.close();
        } catch (Exception e) {
        }
      }
    }
  }

  public static void main(String[] args) throws IOException {
    try {
      // Carico il driver.
      Class.forName(DRIVER);
    } catch (ClassNotFoundException e1) {
      // Il driver non può essere caricato.
      System.out.println("Driver non trovato...");
      System.exit(1);
    }
    // Interagisco con l'utente.
    BufferedReader reader = new BufferedReader(
        new InputStreamReader(System.in)
    );
    while (true) {
      System.out.print("Nome: ");
      String nome       =  reader.readLine();
      System.out.print("Cognome: ");
      String cognome    =  reader.readLine();
      System.out.print("Indirizzo: ");
      String indirizzo  =  reader.readLine();
      System.out.println();
      if (aggiungiRecord(nome, cognome, indirizzo)) {
        System.out.println("Record aggiunto!");
      } else {
        System.out.println("Errore!");
      }
      System.out.println();
      String ris;
      do {
        System.out.print("Vuoi aggiungerne un altro (si/no)? ");
        ris = reader.readLine();
      } while (!ris.equals("si") && !ris.equals("no"));
      if (ris.equals("no")) {
        break;
      }
      System.out.println();
    }
  }

}
```

Si provi pure ad aggiungere diversi nuovi record alla tabella *Persone*, sfruttando la routine interattiva allestita nel *main()* del programma. Come è possibile osservare, in questo caso si è fatto ricorso al metodo *executeUpdate()* di *Statement*. L'operazione richiesta, infatti, rientra nella categoria degli aggiornamenti, e non in quella delle query. Potrete verificare le aggiunte effettuate tornando ad eseguire la classe *JDBCTest1*.

19.9 - Codice SQL più semplice con PreparedStatement

L'interfaccia *java.sql.PreparedStatement*, che estende la già nota *Statement*, permette di concepire più semplicemente e velocemente il codice SQL per l'interazione con il DB. L'elaborazione di query e di istruzioni di aggiornamento, infatti, spesso comporta alcuni problemi di conflitto. Si prenda in considerazione la classe *JDBCTest4*, elaborata nel corso del paragrafo predente. Dopo aver avviato il programma si inseriscano i seguenti dati:

```
Nome: Franco
Cognome: Grigi
Indirizzo: Via L'Aquila 18
```

Consegnati i dati non si potrà non ottenere il messaggio:

```
Errore!
```

Perché? Il problema è tutto nel dato:

```
Via L'Aquila 18
```

Questa stringa contiene un carattere particolare: l'apice, usato come apostrofo tra L e Aquila. Si torni ad esaminare il codice SQL generato nel metodo *aggiungiRecord()* e dato in pasto ad *executeUpdate()*:

```
statement.executeUpdate(
  "INSERT INTO Persone (      " +
  "  Nome, Cognome, Indirizzo " +
  ") VALUES (                 " +
  "  '" + nome + "',          " +
  "  '" + cognome + "',       " +
  "  '" + indirizzo + "'      " +
  ")"
);
```

Sostituendo manualmente i dati si ottiene l'istruzione SQL:

```
INSERT INTO Persone (
  Nome, Cognome, Indirizzo
) VALUES (
  'Franco',
  'Grigi',
  'Via L'Aquila 18'
)
```

Il problema è chiaramente qui:

```
'Via L'Aquila 18'
      ^
```

L'apice usato come apostrofo termina la stringa SQL, causando un errore di interpretazione del codice.

In casi come questo si può scegliere di eseguire la sostituzione dei caratteri problematici, tecnica consentita dai differenti dialetti SQL esistenti. Ad esempio per MySQL è valido correggere le stringhe inserite dagli utenti con l'inserimento di sequenze di backslash:

```
'Via L\'Aquila 18'
```

Tuttavia questa soluzione non è né maneggevole né affidabile. JDBC offre di meglio.

L'interfaccia *PreparedStatement* permette di scrivere codice SQL parametrico, sostituendo i dati in input concatenati via stringa con dei caratteri di punto interrogativo. L'istruzione di inserimento prima mostrata può essere così riscritta:

```
"INSERT INTO Persone (      " +
"  Nome, Cognome, Indirizzo " +
") VALUES (                 " +
"  ?, ?, ?                  " +
")"
```

Fatto ciò, è possibile impostare uno ad uno i parametri espressi attraverso i metodi *setter* esposti da *PreparedStatement*. Questi metodi permettono di inserire i parametri senza preoccuparsi della conflittualità dei loro contenuti: sarà JDBC, in collaborazione con il driver dello specifico DBMS, a risolvere ogni problema. I *setter* di *PreparedStatement*, un po' come i *getter* di *ResultSet*, esistono per i principali tipi di Java. Il seguente elenco li riassume:

- ***setBoolean(int p, boolean value)***. Imposta il booleano *value* come valore del parametro alla posizione *p*.
- ***setByte(int p, byte value)***. Imposta il *byte value* come valore del parametro alla posizione *p*.
- ***setDate(int p, Date value)***. Imposta l'oggetto *java.util.Date value* come valore del parametro alla posizione *p*.
- ***setDouble(int p, double value)***. Imposta il *double value* come valore del parametro alla posizione *p*.
- ***setFloat(int p, float value)***. Imposta il *float value* come valore del parametro alla posizione *p*.
- ***setInt(int p, int value)***. Imposta l'intero *value* come valore del parametro alla posizione *p*.
- ***setLong(int p, long value)***. Imposta il *long value* come valore del parametro alla posizione *p*.
- ***setShort(int p, short value)***. Imposta lo *short value* come valore del parametro alla posizione *p*.
- ***setString(int p, String value)***. Imposta la stringa *value* come valore del parametro alla posizione *p*.

Parametri NULL
Il metodo *setNull(int numeroParametro, int tipoCampo)* permette di impostare sul valore nullo un parametro di una procedura memorizzata. Il secondo parametro accettato dal metodo specifica il tipo SQL del campo da verificare. Ogni tipo supportato da JDBC è identificato da un intero. Non è necessario avere una tabella delle corrispondenze, giacché la classe *java.sql.Types* fornisce una serie di appigli mnemonici di più semplice impiego.

Le *PreparedStatement*, oltre a risolvere i problemi illustrati, garantiscono inoltre maggiori prestazioni ed un alto grado di riusabilità. Quando si prepara un oggetto di questo tipo, infatti, il suo codice SQL viene precompilato. La medesima *PreparedStatement*, inoltre, può essere sfruttata più volte consecutivamente, cambiando semplicemente i parametri

attraverso i suoi metodi *setter* per ottenere risultati diversi, senza dover ogni volta riprocessare il codice SQL che le compone, come avverrebbe con una *Statement* classica.

Per lavorare con un oggetto *PreparedStatement* in luogo di un semplice *Statement* è necessario chiamare:

```
PreparedStatement statement = connection.prepareStatement(CODICE_SQL);
```

in luogo di

```
Statement statement = connection.createStatement();
```

Il codice SQL, con *PreparedStatement*, va specificato al momento della creazione dell'oggetto, e non quando si richiamano i metodi *executeQuery()* o *executeUpdate()*. Le due varianti di questi metodi offerte da *PreparedStatement*, infatti, sono prive di argomenti.

L'esempio del paragrafo precedente può allora essere riscritto alla seguente maniera:

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

public class JDBCTest5 {

  // Nome del driver.
  private static final String DRIVER = "com.mysql.jdbc.Driver";

  // Indirizzo del database.
  private static final String DB_URL = "jdbc:mysql://localhost:3306/javatest";

  // Questo metodo aggiunge un nuovo record alla tabella nel DB.
  private static boolean aggiungiRecord(String nome, String cognome,
      String indirizzo) {
    // Preparo il riferimento alla connessione.
    Connection connection = null;
    try {
      // Apro la connesione verso il database.
      connection = DriverManager.getConnection(DB_URL);
      // Preparo lo Statement per interagire con il database.
      PreparedStatement statement = connection.prepareStatement(
          "INSERT INTO Persone (      " +
          "  Nome, Cognome, Indirizzo " +
          ") VALUES (                 " +
          "  ?, ?, ?                  " +
          ")"
      );
      // Imposto i parametri.
      statement.setString(1, nome);
      statement.setString(2, cognome);
      statement.setString(3, indirizzo);
      // Eseguo l'aggiornamento.
      statement.executeUpdate();
      return true;
    } catch (SQLException e) {
      // In caso di errore...
      return false;
    } finally {
      if (connection != null) {
        try {
```

```
            connection.close();
          } catch (Exception e) {
          }
        }
      }
    }

    public static void main(String[] args) throws IOException {
      try {
        // Carico il driver.
        Class.forName(DRIVER);
      } catch (ClassNotFoundException e1) {
        // Il driver non può essere caricato.
        System.out.println("Driver non trovato...");
        System.exit(1);
      }
      // Interagisco con l'utente.
      BufferedReader reader = new BufferedReader(
          new InputStreamReader(System.in)
      );
      while (true) {
        System.out.print("Nome: ");
        String nome       =  reader.readLine();
        System.out.print("Cognome: ");
        String cognome    =  reader.readLine();
        System.out.print("Indirizzo: ");
        String indirizzo  =  reader.readLine();
        System.out.println();
        if (aggiungiRecord(nome, cognome, indirizzo)) {
          System.out.println("Record aggiunto!");
        } else {
          System.out.println("Errore!");
        }
        System.out.println();
        String ris;
        do {
          System.out.print("Vuoi aggiungerne un altro (si/no)? ");
          ris = reader.readLine();
        } while (!ris.equals("si") && !ris.equals("no"));
        if (ris.equals("no")) {
          break;
        }
        System.out.println();
      }
    }

}
```

Questa nuova release del software permette finalmente l'inserimento di un indirizzo come “Via L'Aquila”. Provare per credere.

19.10 - Richiamare le procedure memorizzate con CallableStatement

Con *PreparedStatement*, come si è appurato, è possibile sfruttare delle istruzioni SQL precompilate, che garantiscono migliori prestazioni e maggiore chiarezza. Le query desiderate sono date in pasto al DBMS al momento della creazione di un oggetto *PreparedStatement*, e possono poi essere ripetutamente sfruttate quante volte si desidera. Tuttavia, JDBC ed il DBMS devono ricompilare la medesima istruzione SQL ogni volta che l'oggetto viene ricreato. Si può fare di meglio, se il DBMS lo permette, ottenendo incrementi ancora più significativi sia nelle prestazioni sia nella semplicità del codice.

L'interfaccia *CallableStatement* estende *PreparedStatement*, e permette di richiamare delle procedure memorizzate all'interno del database. Una query, con *CallableStatement*, non deve essere specificata in linea all'interno del codice Java, ma può essere memorizzata perennemente all'interno della base di dati, pronta ad essere sfruttata più e più volte, in tutte le parti di tutte le applicazioni realizzate.
Ricollegandosi a quanto già fatto, si impieghi un database di Access come esempio guida. Si crei un database inizialmente vuoto, quindi si inserisca al suo interno una tabella, nominata *Utenti*. La struttura è riportata di seguito:

- *ID*, di tipo *Contatore*, da impiegare come chiave primaria.
- *Nome*, di tipo *Testo*.
- *Cognome*, di tipo *Testo*.
- *Email*, di tipo *Testo*.
- *AnnoNascita*, di tipo *Numerico*.

Si popoli la tabella con qualche record arbitrario, ad esempio:

ID	***Nome***	***Cognome***	***Email***	***AnnoNascita***
1	Mario	Rossi	rossi@tin.it	1958
2	Luigi	Bianchi	bianchi@libero.it	1972
3	Antonio	Verdi	verdi@tiscali.it	1967
4	Vittorio	Neri	*NULL*	1948
5	Franco	Grigi	*NULL*	1979
6	Massimo	Gialli	*NULL*	1981

Si registri un DSN di sistema associato alla base di dati. Si scelga il nome *javatest2*.

Ogni DBMS impiega tecniche differenti per la preparazione di procedure memorizzate. Con Access, ad esempio, è possibile servirsi di una comoda e semplice interfaccia utente.

1. Aprendo il database con Access, si attivi la linguetta laterale nominata “Query”.
2. Si avvii la procedura “Crea una query in visualizzazione struttura”.
3. Si selezioni la tabella *Utenti* come sorgente di dati.
4. Nell'elenco disponibile, si faccia in modo che tutti i campi della tabella siano selezionati e restituiti (se non siete pratici di Access, fate riferimento alla **Figura 19.3**).
5. Sotto la colonna associata al campo *Email*, alla voce “Criteri”, si introduca la limitazione “Is Not Null”.
6. Si salvi la query generata con il nome *NotNullMail*.

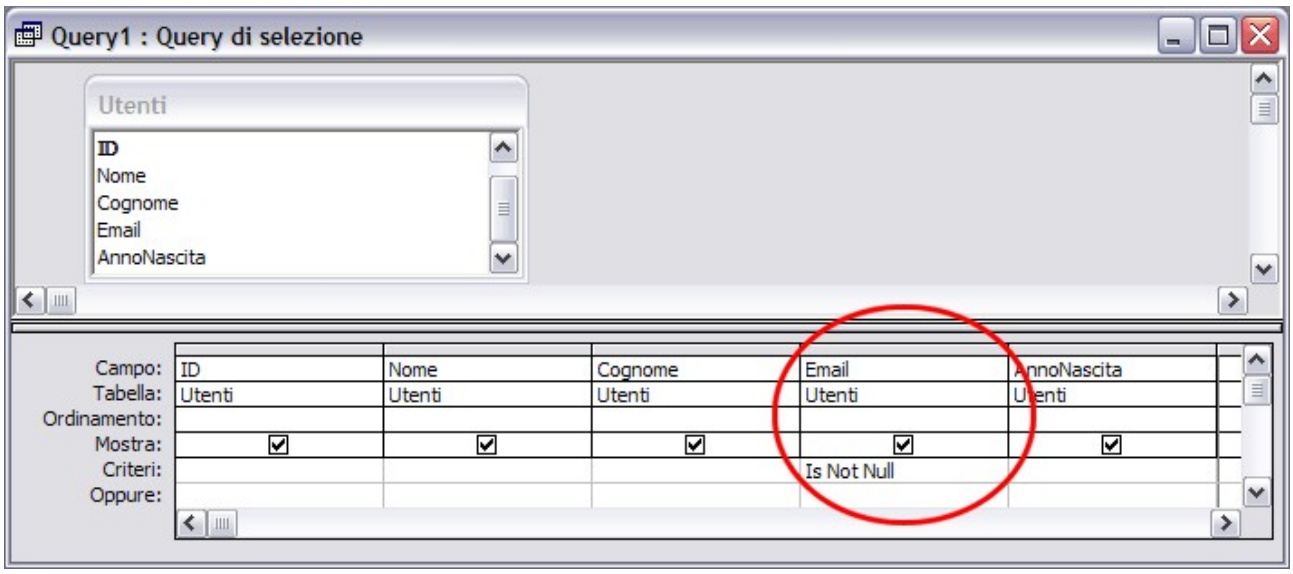

Figura 19.3
La creazione della procedura memorizzata suggerita con Access.

Tramite questi passi, è stata memorizzata una query del tipo:

```
SELECT
  ID, Nome, Cognome, Email, AnnoNascita
FROM
  Utenti
WHERE
  Email IS NOT NULL
```

Lo scopo è restituire l'elenco di tutti gli utenti di cui è noto l'indirizzo e-mail.

Adesso è possibile richiamare la query *NotNullMail* direttamente da codice Java:

```
// Ottengo un oggetto CallableStatement che richiama
// la procedura memorizzata NotNullMail.
CallableStatement statement = connection.prepareCall(
  "{call NotNullMail}"
);
// Eseguo la query.
ResultSet resultset = statement.executeQuery();
```

Ecco una classe completa che lo dimostra:

```
import java.sql.*;

public class JDBCTest6 {

  public static void main(String[] args) {
    // Nome del driver.
    String DRIVER = "sun.jdbc.odbc.JdbcOdbcDriver";
    // Indirizzo del database.
    String DB_URL = "jdbc:odbc:javatest2";
    try {
      // Carico il driver.
      Class.forName(DRIVER);
```

```
    } catch (ClassNotFoundException e1) {
      // Il driver non può essere caricato.
      System.out.println("Driver non trovato...");
      System.exit(1);
    }
    // Preparo il riferimento alla connessione.
    Connection connection = null;
    try {
      // Apro la connesione verso il database.
      connection = DriverManager.getConnection(DB_URL);
      // Mi preparo a richiamare la procedura memorizzata.
      CallableStatement statement = connection.prepareCall(
          "{call NotNullMail}"
      );
      // Interrogo il DBMS.
      ResultSet resultset = statement.executeQuery();
      // Scorro e mostro i risultati.
      while (resultset.next()) {
        int id = resultset.getInt(1);
        String nome = resultset.getString(2);
        String cognome = resultset.getString(3);
        String email = resultset.getString(4);
        int annoNascita = resultset.getInt(5);
        System.out.println("Lette informazioni...");
        System.out.println("ID: " + id);
        System.out.println("Nome: " + nome);
        System.out.println("Cognome: " + cognome);
        System.out.println("Email: " + email);
        System.out.println("Anno di nascita: " + annoNascita);
        System.out.println();
      }
    } catch (SQLException e) {
      // In caso di errore...
    } finally {
      if (connection != null) {
        try {
          connection.close();
        } catch (Exception e) {
        }
      }
    }
  }

}
```

L'output mostrato sarà:

```
Lette informazioni...
ID: 1
Nome: Mario
Cognome: Rossi
Email: rossi@tin.it
Anno di nascita: 1952

Lette informazioni...
ID: 2
Nome: Luigi
Bianchi
bianchi@libero.it
Cognome: Bianchi
Email: bianchi@libero.it
```

```
Anno di nascita: 1972

Lette informazioni...
ID: 3
Nome: Antonio
Cognome: Verdi
Email: verdi@tiscali.it
Anno di nascita: 1967
```

Anche una *CallableStatement* può sfruttare uno o più parametri. Andiamo a dimostrare un esempio di questo tipo. Tornando ad Access, si realizzi una nuova procedura memorizzata, chiamata *AnnoNascita* (**Figura 19.4**). Come nel caso precedente, si selezioni la tabella *Utenti* e si trasportino tutti i campi disponibili. Il criterio di restrizione, questa volta, lo si applicherà al campo *AnnoNascita*, con il codice:

```
Between [min] And [max]
```

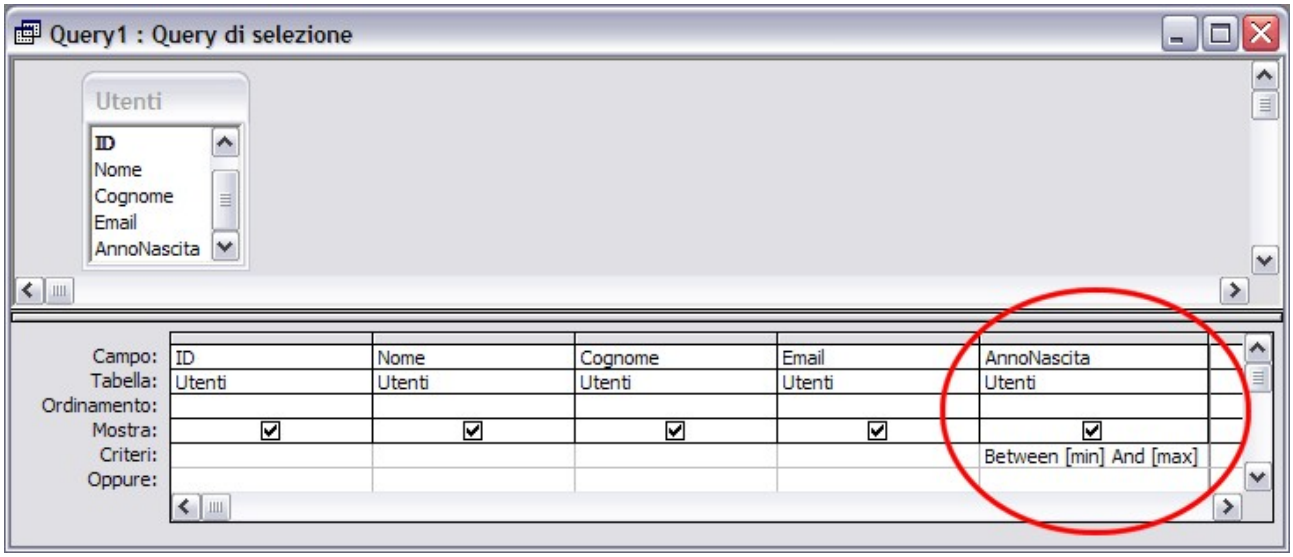

Figura 19.4
La creazione della procedura memorizzata con parametri.

Questa query, sostanzialmente, corrisponde più o meno ad una *PreparedStatement*:

```
SELECT * FROM Utenti WHERE AnnoNascita BETWEEN ? AND ?
```

Potrà essere richiamata alla seguente maniera:

```
CallableStatement statement = connection.prepareCall(
  "{call AnnoNascita(?, ?)}"
);
```

I parametri potranno essere impostati come con una *PreparedStatement*:

```
statement.setInt(1, estremoInferiore);
statement.setInt(2, estremoSuperiore);
```

L'intero codice che se ne desume è riportato di seguito:

```
import java.sql.*;

public class JDBCTest7 {

  public static void main(String[] args) {
    // Nome del driver.
    String DRIVER = "sun.jdbc.odbc.JdbcOdbcDriver";
    // Indirizzo del database.
    String DB_URL = "jdbc:odbc:javatest2";
    try {
      // Carico il driver.
      Class.forName(DRIVER);
    } catch (ClassNotFoundException e1) {
      // Il driver non può essere caricato.
      System.out.println("Driver non trovato...");
      System.exit(1);
    }
    // Preparo il riferimento alla connessione.
    Connection connection = null;
    try {
      // Apro la connesione verso il database.
      connection = DriverManager.getConnection(DB_URL);
      // Mi preparo a richiamare la procedura memorizzata.
      CallableStatement statement = connection.prepareCall(
          "{call AnnoNascita(?,?)}"
      );
      // Imposto i parametri.
      statement.setInt(1, 1960);
      statement.setInt(2, 1980);
      // Interrogo il DBMS.
      ResultSet resultset = statement.executeQuery();
      // Scorro e mostro i risultati.
      while (resultset.next()) {
        int id = resultset.getInt(1);
        String nome = resultset.getString(2);
        String cognome = resultset.getString(3);
        String email = resultset.getString(4);
        int annoNascita = resultset.getInt(5);
        System.out.println("Lette informazioni...");
        System.out.println("ID: " + id);
        System.out.println("Nome: " + nome);
        System.out.println("Cognome: " + cognome);
        System.out.println("Email: " + email);
        System.out.println("Anno di nascita: " + annoNascita);
        System.out.println();
      }
    } catch (SQLException e) {
      // In caso di errore...
    } finally {
      if (connection != null) {
        try {
          connection.close();
        } catch (Exception e) {
        }
      }
    }
  }

}
```

Il codice desidera avere i record con anno di nascita compreso tra il 1960 ed il 1980. L'output prodotto dalla sua esecuzione è:

```
Lette informazioni...
ID: 2
Nome: Luigi
Cognome: Bianchi
Email: bianchi@libero.it
Anno di nascita: 1972

Lette informazioni...
ID: 3
Nome: Antonio
Cognome: Verdi
Email: verdi@tiscali.it
Anno di nascita: 1967

Lette informazioni...
ID: 5
Nome: Franco
Cognome: Grigi
Email: null
Anno di nascita: 1979
```

Valori restituiti

Le procedure memorizzate possono restituire dei valori. In tal caso, per poter leggere quanto elaborato, è necessario ricorrere alla sintassi:

```
CallableStatement statement = connection.prepareCall(
  "{? = call NomeProcedura(?, ?, ...)}"
);
```

Maggiori informazioni nella documentazione ufficiale.