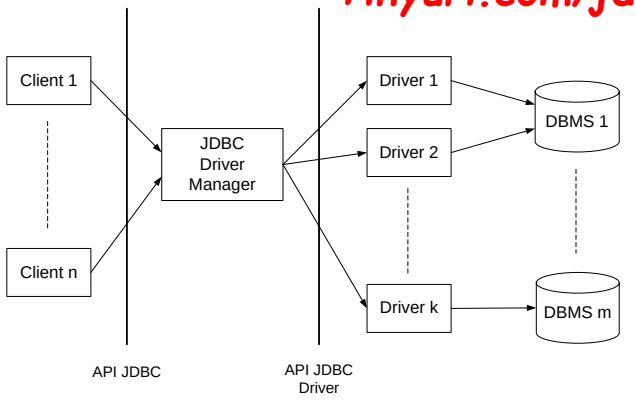
JDBC

Tinyurl.com/jdbc14



Alessandra Lumini
Alessandra.lumini@unibo.it

Concetti Fondamentali

- Riferimenti
- Introduzione
  - Architettura
  - Tipi di driver
- Un Esempio
  - Driver e Driver Manager
  - Connection
  - PreparedStatement
  - ResultSet
- Esercizi

Riferimenti

- Sito JDBC
  - http://www.oracle.com/technetwork/java/javase/jdbc/index.html
- API Java
  - <directory java>/docs/guide/jdbc
- Tutorial on-line
  - http://docs.oracle.com/javase/tutorial/jdbc/
- Cap. 6 del libro Atzeni, Ceri, Paraboschi, Torlone *Basi di dati,* McGraw-Hill Italia, 2009
- Dalla documentazione JavaDoc
  - java.sql: classi fondamentali
  - javax.sql: estensioni

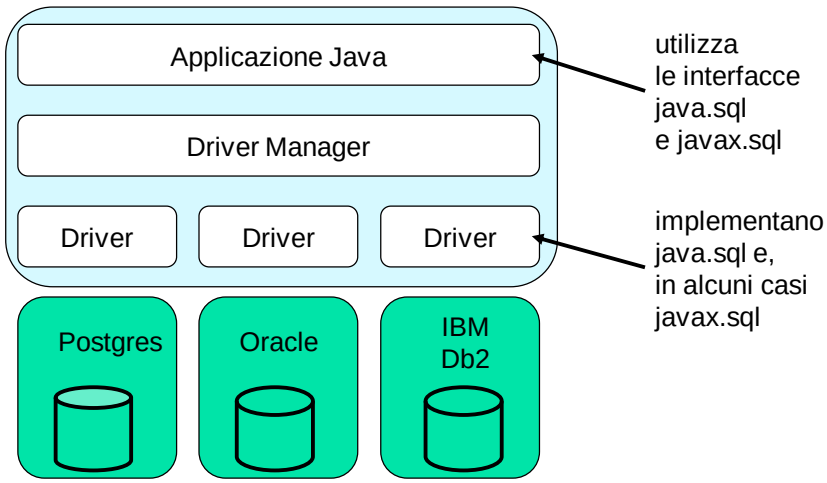
JDBC

- JDBC è una libreria di API per l'accesso **uniforme** a database relazionali
  - Definisce classi per:
    - connessione a database
    - invio di query SQL
    - navigazione dei risultati delle query
    - accesso ai metadati del database/risultati
- Versioni
  - JDBC 1.0 -- JDK 1.1
  - JDBC 2.0 -- JDK 1.3
  - JDBC 3.0 -- JDK 1.4
  - JDBC 4.0 -- JDK 1.5
  - JDBC 4.1 -- JDK 1.7
- Packages Java
  - **java.sql**
  - javax.sql

JDBC vs ODBC (1)

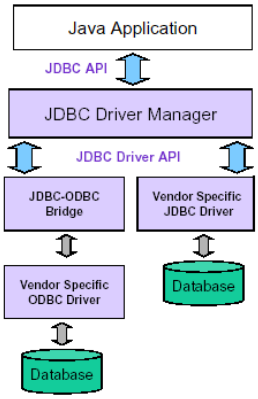
- ODBC
  - dipendente dal sistema
  - utilizzabile da più linguaggi di programmazione
  - scambio dei dati complesso
- JDBC
  - indipendente dal sistema
  - utilizzabile solo con il linguaggio JAVA
  - minore complessità dell'interfaccia
- Entrambe le API permettono di richiamare funzionalità specifiche dei DBMS

Architettura



Tipi di Driver

- Driver di tipo 1
  - traducono JDBC in ODBC
    - JDBC-ODBC Bridge
  - inefficienti
- Driver di tipo 2
  - parte in Java e parte in codice nativo
- Driver di tipo 3
  - pure Java
  - protocollo indipendente dal RDBMS
- Driver di tipo 4
  - pure Java
  - protocollo specifico del RDBMS



JDBC: Accesso ai dati

- L'API fornisce un meccanismo che carica dinamicamente i driver appropriati e li registra nel JDBC Driver Manager.
- Le connessioni JDBC supportano la creazione e l'esecuzione delle istruzioni (INSERT, UPDATE, DELETE, SELECT)
- I tipi di istruzioni supportati sono:
- **Statement** - l'istruzione viene inviata al database di volta in volta;
- **Prepared Statement** - l'istruzione viene compilata una sola volta, in modo che le chiamate successive siano più efficienti;
- **Callable Statement** - usati per chiamare le stored procedure.
- I comandi di scrittura restituiscono un valore che indica il numero di righe modificate
- Le interrogazioni restituiscono un result set (classe ResultSet).

Step di accesso ai dati

1. Registrare il driver JDBC
2. Connettersi al DB
3. Definire una query SQL
4. Elaborare i risultati della query
5. Chiudere la connessione

Step di accesso ai dati 1-3

1. **Il metodo Class.forName() carica la classe del driver JDBC.**
   - `Class.forName( "com.mioDbms.mioDriver" );`
2. **Poi, il metodo DriverManager.getConnection() crea una connessione.**
   - `Connection conn = DriverManager.getConnection( "jdbc:mioDbms:altri dati utili per il driver", "myLogin", "myPassword" );`
   - La stringa da utilizzare dipende dal driver JDBC. Inizia sempre con "jdbc:", il resto varia a seconda del prodotto scelto.
3. **Una volta stabilita la connessione, occorre passare una istruzione.**
   - `Statement stmt = conn.createStatement(); stmt.executeUpdate( "INSERT INTO myTab(myColumn) VALUES ( 'val') " );`

Step di accesso ai dati 4-5

4. **I dati vengono prelevati dal database e restituiti come RecordSet da cui vengono elaborati.**

```
Statement stmt = conn.createStatement();
ResultSet rs = stmt.executeQuery( "SELECT * FROM
myTable" );
while ( rs.next() ) {
    int numCol = rs.getMetaData().getColumnCount();
    for ( int i = 1 ; i <= numCol ; i++ )
    { // I numeri di colonna iniziano da 1.
System.out.println( "COL" + i + "=" +rs.getObject(i));
  } }
```

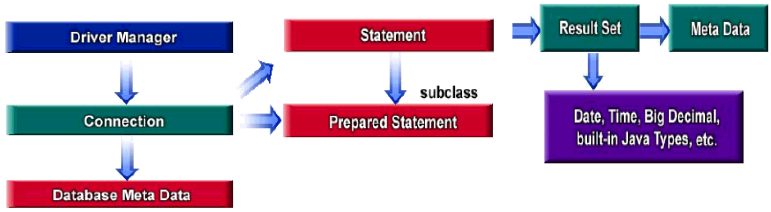
5. **Chiusura connessione**

```
rs.close();
stmt.close();
```


JDBC - classi e interfacce

- **java.sql.DriverManager:** ***classe***. Offre funzionalità di **gestione dei database driver**
- **java.sql.Driver:** ***interfaccia***. Astrae i dettagli del protocollo di **connessione al database**. I produttori di database implementano interfaccia
- **java.sql.Connection:** ***interfaccia***. Astrae dettagli di interazione con il database. Permette di **inviare statement SQL e di leggere i risultati** dell'esecuzione degli statement. I produttori di database implementano l'interfaccia

JDBC: Classi Java



Interfaccia Driver

- Rappresenta il punto di partenza per ottenere una connessione a un DBMS
- I produttori di driver JDBC implementano l'interfaccia Driver (mediante opportuna classe) affinché possa funzionare con un tipo particolare di DBMS
- La classe che implementa Drive può essere considerata la "factory" per altri oggetti JDBC (ad esempio, oggetti di tipo Connection)
- È possibile ottenere un'istanza effettiva della classe Driver ricercando la classe con il metodo forName:
  - ```
    Driver d =
    class.forName("com.mioDbms.mioDriver").newIn
    stance();
    ```
-
    ```

Classe java.sql.DriverManager

- Facilita la gestione di oggetti di tipo Driver
- Quando un oggetto Driver viene istanziato, esso viene automaticamente registrato nella classe DriverManager
- Ogni applicazione può registrare uno o più driver JDBC diversi tra loro
- Consente la connessione con il DBMS sottostante mediante il metodo statico getConnection
- Usa il driver opportuno tra quelli registrati

Interfaccia Connection

- Un oggetto di tipo Connection rappresenta una connessione attiva con il DB
- Il metodo getConnection di DriverManager, se non fallisce, restituisce un oggetto di tipo Connection

```
String url = "jdbc:odbc:dbname";
Connection db = DriverManager.getConnection(url);
```

- L'interfaccia mette a disposizione una serie di metodi per le operazioni, tra le quali:
- Preparazione di query SQL da inviare tramite oggetti
  - Statement
  - PreparedStatement
  - CallableStatement

Interfaccia Statement

- Gli oggetti di tipo Statement possono essere usati per inviare query SQL senza parametri al DBMS. La query può essere:
- di modifica UPDATE, INSERT, CREATE..
  - (StatementObj.) executeUpdate(stmt SQL)
  - Restituisce un intero rappresentante il numero di righe che sono state inserite/aggiornate/cancellate o 0 se comando DDL
- di selezione SELECT
  - (StatementObj.) executeQuery(stmt SQL), genera un unico result set
  - Per query di tipo SELECT il risultato è inserito in un oggetto ResultSet
- query con più di un risultato o più di un contatore di aggiornamento
  - (StatementObj.) execute(stmt SQL)
  - ```
    Statement s = db.createStatement();
    ResultSet r = s.executeQuery("Select * from Table");
    ```


Interfaccia PreparedStatement

- Gli oggetti di tipo PreparedStatement possono essere usati per creare query SQL parametriche (di tipo IN) e precompilate (“prepared”)
- Il valore di ciascun parametro non è specificato nel momento in cui lo statement SQL è definito, ma rimpiazzato dal carattere ‘?’
- Un oggetto PreparedStatement può essere creato con il metodo prepareStatement di Connection
  - ```
    String sSQL = "SELECT * FROM Table WHERE ID = ?";
    PreparedStatement pS = db.prepareStatement(sSQL);
    ```
- I parametri vengono poi settati mediante il metodo (StatementObj.)setXXX(n,value)
  - ```
    pS.setInt(1,1);
    ResultSet rs = pS.executeQuery(sSQL);
    ```
    ```

Interfaccia CallebleStatement

- Gli oggetti di tipo CallableStatement possono essere usati per definire query parametriche con parametri di tipo IN, OUT e INOUT
- Permettono di eseguire una invocazione a una stored procedure memorizzata sul server DB
- Un oggetto CallableStatement può essere creato con il metodo prepareCall di Connection
  - ```
    String sql = "{ call setSalary (?) }";
    CallableStatement cS = db.prepareCall(sql);
    cS.setInt (1,1000);
    cS.executeUpdate();
    ```
  - ```
    //ResultSet result = cS.executeQuery();
    ```


CallebleStatement: Batch Updates

- Si possono raggruppare chiamate multiple a una stored procedure in un batch update.
  - ```
    String sql = "{ call setSalary (?) }";
    CallableStatement cS = db.prepareCall(sql);
    cS.setInt (1,1000);
    cS.addBatch();
    cS.setInt (1,2000);
    cS.addBatch();
    cS.executeBatch();
    ```
    ```

CallebleStatement: OUT Parameters

- Se una stored procedure restutuisce dei parametri, sono accessibili dal callebleStatement object
- è necessario che ogni OUT e INOUT parameter venga registrato prima dell'invocazione alla stored procedure

```
String sql = "{ call calculateStatistics(?) }";
CallableStatement cS = db.prepareCall(sql);
cS.setInt (1,123);
cS.registerOutParameter(1, java.sql.Types.VARCHAR);
cS.registerOutParameter(2, java.sql.Types.INT);
ResultSet result = cS.executeQuery();
String out1 = cS.getString(1);
int out2 = cS.getInt (2);
```

```
//while(result.next()) { ... }
```


Interfaccia ResultSet

- L'oggetto ResultSet è il risultato di una query di selezione
- Rappresenta una tabella composta da righe (gli elementi selezionati) e colonne (gli attributi richiesti)
- Per ottenere il valore del cursore alla riga corrente:
  - (ResultSetObj.) getXXX(column-name)
  - (ResultSetObj.) getXXX(column-number)
  - I metodi getInt, getString,… permettono la lettura degli attributi di una tabella
- Per spostare il cursore dalla riga corrente a quella successiva:
  - (ResultSetObj.) next() (restituisce true in caso di successo; false se non ci sono più righe nell'insieme risultato)

```
while(result.next())
{ result.getString ("nome");
  result.getInt ("età"); // etc. }
```

Altri tipi di ResultSet

- È possibile creare dei ***ResultSet*** con maggiori funzionalità, passando degli opportuni parametri al momento della creazione dello ***Statement*** :
  - ***scrollable ResultSet*** :
    - le righe del risultato possono essere scandite in entrambe le direzioni
    - ci si può posizionare in maniera "assoluta" su una particolare riga
  - ***updatable ResultSet*** :
    - mettono a disposizione dei metodi ***updateXXX*** analoghi ai metodi ***getXXX***, che consentono di modificare il valore di un campo sulla riga corrente
    - l'aggiornamento si riflette sulla corrispondente tabella nel database automaticamente

Tipi di ResultSet

- ResultSet.TYPE_FORWARD_ONLY – Il resultset è navigabile solo in avanti
- ResultSet.TYPE_SCROLL_INSENSITIVE – Il resultset è scrollabile. Modifiche fatte da altri sulle tabelle da cui è stato derivato il resultset non sono visibili
- ResultSet.TYPE_SCROLL_SENSITIVE - Il resultset è scrollabile. Modifiche fatte sulle tabelle da cui è stato derivato il resultset sono visibili nel resultset

- ResultSet.CONCUR_READ_ONLY – È un resultset di sola lettura (default)
- ResultSet.CONCUR_UPDATABLE – È un resultset che può essere aggiornato, e le modifiche riportate nel DB

Esempio di UPDATE

```
Statement st =
db.createStatement(ResultSet.TYPE_SCROLL_SENSITIVE,
ResultSet.CONCUR_UPDATABLE);
ResultSet results = st.executeQuery("SELECT test_col
FROM test_table");
// Muove il cursore alla 3a riga
results.absolute(3);
// Update della 3a riga
results.updateString("test_col","nuovo_valore");
results.updateRow();
// In caso si abort si usa:
//results.cancelRowUpdates();
results.absolute(5);
results.deleteRow();
// Aggiorna il resultSet
results.refreshRow();
```


Metodi di navigazione

- Boolean next(), previous() - Muove il puntatore alla riga successiva (precedente). Se è stata raggiunta l'ultima (prima) riga il metodo restituisce false e il puntatore punta immediatamente dopo l'ultima riga
- Boolean first(), last() - Muove il puntatore alla prima/ultima riga del resultSet.
- Boolean absolute(n) – n>0: il puntatore si posiziona sulla riga n a partire dall'inizio (le righe si contano a partire da uno) n<0: come sopra ma a partire dalla fine
- Boolean relative(n) - il puntatore si posiziona di n righe in avanti (o indietro se n<0) a partire dalla posizione attuale
- Void afterLast(), void beforeFirst()
- Boolean isFirst(), isBeforeFirst(), isLast(), isAfterLast(),
- Int getRow() - Restituisce il numero della riga corrente

Conversione tipi SQL-JAVA

	INTEGER	REAL	CHAR	VARCHAR	DATE	CLOB
getInt	**X**	X	X	X		
getFloat	X	**X**	X	X		
getString	X	X	**X**	**X**	X	
getDate			X	X	**X**	
getClob						**X**
getObject	X	X	X	X	X	X

(1) Registrare il driver

- Significa caricare dinamicamente una classe Java
- sun.jdbc.odbc.JdbcOdbcDriver è il JDBC-ODBC Bridge fornito con JDK
- Altri driver:
  - SQL Server com.microsoft.jdbc.sqlserver.SQLServerDriver
  - IBM DB2 com.ibm.db2.jcc.DB2Driver
  - Oracle oracle.jdbc.OracleDriver
  - MySQL com.mysql.jdbc.Driver
- File Jar di SQL Server:
  - mssbase.jar
  - mssqlserver.jar
  - msutil.jar

(1) Caricare la classe

- Per caricare la classe occorre:
  - avviare il programma con –classpath <file_jar_del_driver>
  - o, modificare CLASSPATH con <file_jar_del_driver>
  - **o, copiare <file_jar_del_driver> in jre/lib/ext**

```
import java.sql.*; // package JDBC
public int count(String[] args)
    throws ClassNotFoundException
 {
    // registra il driver MS Sql Server
    Class.forName(" com.microsoft.jdbc.sqlserver.SQLServerDriver ");
    …
}
```

Url di database

- Sintassi (per collegarsi al db) simile a URL internet
- Sintassi generica
  jdbc:nom sottoprotocollo: altro
- Esempi
  - jdbc:db2:nomeDB
  - jdbc:pointbase:nomeDB
  - jdbc:odbc:nomeDB
  - jdbc:mysql://localhost:3306/nomeDB
  - jdbc:oracle:thin:@localhost:1521:xe

(2) Connessione al DB

- Per ottenere una connessione dal **`DriverManager`** occorre specificare
  - URL ("indirizzo" completo) della connessione (host, dbms, db)
  - utente e password
- Sintassi URL

  `jdbc:<sottoprotocollo>:<parametri>`
- Esempi
  - jdbc:db2:nomeDB
  - jdbc:pointbase:nomeDB
  - jdbc:odbc:nomeDB
  - jdbc:mysql://localhost:3306/nomeDB
  - jdbc:oracle:thin:@localhost:1521:xe

(2) Connessione al DB

```
String url = "jdbc:microsoft:sqlserver:"+ // driver
             "SI-SQLSERVER;" +            // host
      "DatabaseName=NorthwindIT";  //database
String user = "student";
String password = "pass";
Connection conn =
  DriverManager.getConnection(url, user, password);
```

(3) Definire query SQL

```
// oggetto per la gestione di comandi SQL
Statement stmt = con.createStatement();


String query = "SELECT * FROM myTable";
// metodo per l'esecuzione di query SQL
ResultSet rs = stmt.executeQuery( query );


String update = "UPDATE myTable SET col1= col1 + 1";
// metodo per l'esecuzione di update/insert/create
   table SQL
int affectedRows = stmt.executeUpdate(update );
```


(4) Elaborare i risultati

```
  …
// ResultSet è un iteratore sulle righe
  while ( rs.next() )
  {
// accesso ai valori dei campi per nome
    String nome = rs.getString("col1");
    int eta = rs.getInt("col2");

//accesso ai valori dei campi per posizione (a
  partire da 1)
    int eta2 = rs.getInt( 2 );

     System.out.println(nome + " " + eta);
  }
  …
```

(4) Scorrere il ResultSet

```
…
Statement stmt =
con.createStatement(ResultSet.TYPE_SCROLL_INSENSITIVE,
                    ResultSet.CONCUR_READ_ONLY);

String query = "SELECT col1,col2 FROM myTable";
ResultSet rs = stmt.executeQuery( query );

rs.previous();  // riga precedente
rs.relative(-5);      // 5 righe indietro
rs.relative(7); // 7 righe avanti
rs.absolute(100);     // 100-esima riga
…
```


(4) ResultSet aggiornabili

```
…
Statement stmt =
con.createStatement(ResultSet.TYPE_FORWARD_ONLY,
                    ResultSet.CONCUR_UPDATABLE);

String query = "SELECT col1,col2 FROM myTable";
ResultSet rs = stmt.executeQuery( query );
…
while ( rs.next() )
{
    int eta = rs.getInt("col2");
    rs.updateInt("col2", eta+1);
    rs.updateRow();
}
```

(5) Chiusura Connessione

```
...

  // chiude la connessione al DB
  conn.close();

...
```


Metadati

- La maggior parte dei metodi delle classi finora viste può sollevare delle eccezioni di tipo ***SQLException***
- Per avere informazioni sul database si può richiedere un oggetto ***DataBaseMetaData***
  - si hanno a disposizione metodi per ottenere il nome del db, il numero massimo di connessioni, il nome del driver, ecc.

```
DataBaseMetaData dbmd = conn.getMetaData();
```

- Per avere informazioni su un ***ResultSet*** si può richiedere un oggetto ***ResultSetMetaData***
  - si hanno a disposizione metodi per ottenere il numero di colonne restituite, i loro nomi e tipi, ecc.

```
ResultSetMetaData rsmd = rs.getMetaData();
```

Metadati dal database

```
…
Connection con = …. ;
DatabaseMetaData dbmd = con.getMetaData();
String catalog = null;  //null = tutti
String schema = null;
String table = "sys%";  //tabelle che iniziano per sys
String[] types = null;
ResultSet rs =
    dbmd.getTables(catalog , schema , table , types );
    …
```


Metadati dal resultset

```
public static void printRS(ResultSet rs) throws SQLException
{
  ResultSetMetaData md = rs.getMetaData();
       // output nomi delle colonne
  int nCols = md.getColumnCount();
  for(int i=1; i < nCols; ++i)
      System.out.print( md.getColumnName(i)+",");
  System.out.println( md.getColumnName(nCols));
// output resultset
while ( rs.next() )              {
              for(int i=1; i < nCols; ++i)
                       System.out.print( rs.getString(i)+",");
              System.out.println( rs.getString(nCols));
     }
}
```

Laboratorio

- Obiettivo: esemplificare l'API JDBC. Quindi:
  - scriviamo codice che ha unicamente lo scopo di evidenziare l'uso della API JDBC
  - non ci preoccupiamo della qualità del codice Java
- Caso di studio
  - Database «University»
- Operazioni
  - inserimento di una tupla nel db
  - cancellazione di una tupla
  - ricerca di un tupla per chiave primaria
  - ricerca di un insieme di tuple (per qualche proprietà)

Ambiente

- DBMS
  - SQLServer
- Driver JDBC per il DBMS scelto
  - mssbase.jar
  - mssqlserver.jar
  - msutil.jar
  - jtds-1.3.1.jar (non Microsoft)
- Ambiente Java standard: **`.jar/.zip`** del driver deve essere nel **`CLASSPATH`**
  - Eclipse Project →Properties → Java Build Path Libraries → Add JARs

Classe di riferimento

- Consideriamo la classe **`Student`**:

```
package it.unibo.bd.model;
import java.util.Date;
public class Student {
   private String firstName;
   private String lastName;
   private int code;
   private Date birthDate;

   public String getFirstName() {
     return firstName;
   }

   public void setFirstName(String firstname) {
     this.firstName = firstName;
   }
   // seguono tutti gli altri metodi getter e setter
}
```

Database

- E la relativa tabella nel database:

```
CREATE TABLE students
(
   code INT NOT NULL PRIMARY KEY,
   firstname CHAR(40),
   lastname CHAR (40),
   birthdate DATE
)
```

Le classi fondamentali di JDBC

- Package **`java.sql`** (va importato)

- Classe **`DriverManager`**
- Interfaccia **`Driver`**
- Interfaccia **`Connection`**
- Interfaccia **`PreparedStatement`**
- Interfaccia **`ResultSet`**
- **`Eccezione SQLException`**

Preparazione

- Importare il progetto Eclipse ANT-based presente nel file BD_Es1.zip senza esploderne l'archivio su file system (lo farà Eclipse)
- File → Import → General → Existing Projects into Workspace → Next → Select archive file
- Confiniamo nella classe **DBConnection** le operazioni necessarie per ottenere la connessione il suo compito è fornire connessioni alle altre classi che ne hanno bisogno
  - metodo Connection **getMsSQLConnection**() che restituisce una nuova connessione ad ogni richiesta

(1)(2) Driver e Connessione

```
package it.unibo.bd.db;
import java.sql.*;
public class DBConnection {
    private String dbName = "stud"; // nome del database

    public Connection getMsSQLConnection()  {

    String driver = "net.sourceforge.jtds.jdbc.Driver";
    String dbUri = "jdbc:jtds:sqlserver://137.204.74.1:1433/"+dbName;
    String userName = "student";
    String password = "student";

    Connection connection = null;

    Class.forName(driver);
    connection = DriverManager.getConnection(dbUri, userName, password);

}
```

(3) Istruzione SQL

- Vediamo ora il codice JDBC che esegue istruzioni SQL per:
  - salvare (rendere persistenti) oggetti nel db
  - cancellare oggetti dal db
  - trovare oggetti dal db
- Vedi classe **`StudentTable`**
- Per evitare di lavorare tutti sulla stessa tabella modificare il codice:
  - tableName="students"; → tableName="students_miaMatricola";

(3) Uso di Prepared Statement

- Per eseguire una istruzione SQL è necessario creare un oggetto della classe che implementa **PreparedStatement**
  - creato dall'oggetto **Connection** invocando il metodo:

```
PreparedStatement prepareStatement(String s);
```

- La stringa s è una istruzione SQL parametrica: i parametri sono indicati con il simbolo **?**
- Esempio 1

```
String insert = "insert into students(code, firstname,
lastname, birthdate) values (?,?,?,?)";
statement = connection.prepareStatement(insert);
```

- Esempio 2

```
String delete = "delete from students where code=?";
statement = connection.prepareStatement(delete);
```


(3) Uso di Prepared Statement

- I parametri sono assegnati mediante opportuni metodi della classe che implementa **PreparedStatement**
  - metodi `setXXX (<numPar>, <valore>)`
  - un metodo per ogni tipo, il primo argomento corrisponde all'indice del paramentro nella query, il secondo al valore da assegnare al parametro
- Esempio 1 (cont.)

```
PreparedStatement statement;
String insert = "insert into students(code, firstname,
lastname, birthdate) values (?,?,?,?)";

statement = connection.prepareStatement(insert);
statement.setInt(1, student.getCode());
statement.setString(2, student.getFirstName());
statement.setString(3, student.getLastName());

long secs = student.getBirthDate().getTime());
statement.setDate(4, new java.sql.Date(secs));
```

(3) Conversioni di formato

- JDBC usa **`java.sql.Date`**, mentre la classe Student usa **`java.util.Date`**
- Le istruzioni

```
long secs = student.getBirthDate().getTime());
statement.setDate(4, new java.sql.Date(secs));
```

servono a "convertire" una data da una rappresentazione all'altra

(3) Tipi di operazioni

- Una volta assegnati i valori ai parametri, l'istruzione può eseguita
- Distinguiamo due tipi di operazioni:
  - **aggiornamenti** (insert, update, delete)
    - modificano lo stato del database
    - vengono eseguiti invocando il metodo executeUpdate() sull'oggetto PrepareStatement
  - **interrogazioni** (select)
    - non modificano lo stato del database
    - ritornano una sequenza di tuple
    - vengono eseguite invocando il metodo executeQuery() che restituisce il risultato in un oggetto ResultSet

(3) Aggiornamenti

```
PreparedStatement statement;
String insert = "insert into students(code, firstname,
lastname, birthdate) values (?,?,?,?)";
statement = connection.prepareStatement(insert);
statement.setString(1, student.getCode());
statement.setString(2, student.getFirstName());
statement.setString(3, student.getLastName());
long secs = student.getBirthDate().getTime());
statement.setDate(4, new java.sql.Date(secs));

statement.executeUpdate();
```

(3) Interrogazioni (select)

```
PreparedStatement statement;
String query = "select * from students where code=?";
statement = connection.prepareStatement(query);
statement.setInt(1,code);
ResultSet result = statement.executeQuery();
```

(4) Gestire il risultato di una query

- Un oggetto della classe **`ResultSet`** rappresenta la collezione di ennuple restituita da una query SQL (istruzione **`SELECT`**)
- Per gestire il risultato offre vari metodi:
  - metodo **`boolean next()`** per scorrere le ennuple (analogo ad un iteratore)
  - metodi **`getXXX(String attributo)`** per acquisire i valori degli attributi

    Es.: `int getInt(String attributo);`

    Es.: `String getString(String attributo);`

(4) Gestire il risultato di una query

```
String retrieve = "select * from students where code=?";
PreparedStatement statement =
   connection.prepareStatement(retrieve);
statement.setInt(1, code);
ResultSet result = statement.executeQuery();
Student student = null;
if (result.next()) {
    student = new Student();
    student.setCode(result.getInt("code"));
    student.setFirtsName(result.getString("firstname"));
    student.setLastName(result.getString("lastname"));
    long secs = result.getDate("birthdate").getTime();
    birthDate = new java.util.Date(secs);
    student.setBirthDate(birthDate);
}
```

(4) Gestire il risultato di una query

- Un altro esempio (findAll())

```
List<Student> students = new LinkedList<Student>();
Connection connection = this.dataSource.getConnection();
PreparedStatement statement;
String query = "select * from students";
statement = connection.prepareStatement(query);
ResultSet result = statement.executeQuery();
while(result.next()) {
   Student student = new Student();
   student.setCode(result.getInt("code"));
   student.setFirstName(result.getString("firstname"));
   student.setLastName(result.getString("lastname"));
   student.setBirthDate(new java.util.Date(result.getDate
       ("birthdate").getTime()));
   students.add(student);
}
```

(5) Rilascio delle risorse

- Tutti i metodi delle classi dell'API JDBC “lanciano” una eccezione SQLException
- Connection, statement, ResultSet devono essere sempre "chiusi" per il rilascio di risorse
- Se si effettua la chiusura nella clausola finally si ha la garanzia che venga comunque effettuata

```
finally {
 try {
 if (statement != null)  statement.close();
 if (connection!= null)  connection.close();
 }
 catch (SQLException e) {
     throw new PersistenceException(e.getMessage());
 }
}
```

Esercizio 1

- Studiare il codice della classe **`StudentTable`**
  - Spiegare a che cosa serve l'istruzione

```
if (findByPrimaryKey(student.getCode())!=null)
```

  del metodo **`public void persist()`**
- Scrivere il codice del metodo:

  `public void update(Student student)`

  che aggiorna nel database la tupla corrispondente all'oggetto passato come parametro.
  - Suggerimento: usare il metodo findByPrimaryKey()

Esercizio 2

- Creare il database *olympic:*

```
CREATE TABLE athletes
(
  code int PRIMARY KEY,
  name char(40),
  nation char(40),
  birthDate date,
  height double precision
)
```

Esercizio 2

- Creare la classe **Athlete**:

```
import java.util.Date;
public class Athlete {
   private int code;
   private String name;
   private String nation;
   private double height;
   private Date birthDate;

   public Athlete(){}

   // metodi getter e setter
}
```


Esercizio 2

- Scrivere e testare il codice della classe **AthleteTable**
- Oltre ai metodi:

```
persist(Athlete a)
delete(Athlete a)
update(Athlete a)
findByPrimaryKey(int code)
findAll()
```

scrivere il codice del metodo

```
public List<Athlete>
        findTallAthletes(double h)
```

che ritorna gli atleti con altezza maggiore del parametro **h**