JDBC: SQL NEI LINGUAGGI DI PROGRAMMAZIONE

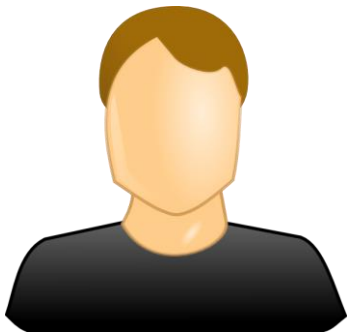
Disheng Qiu
disheng.qiu@gmail.com

Emanuel Weitschek
emanuel@dia.uniroma3.it

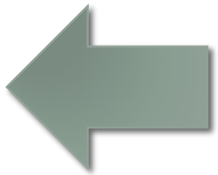
SQL e Applicazioni

- In applicazioni complesse, l'utente non vuole eseguire comandi SQL, ma programmi, con poche scelte
- SQL non basta, sono necessarie altre funzionalità, per gestire:
  - input (scelte dell'utente e parametri)
  - output (con dati che non sono relazioni o se si vuole una presentazione complessa)
  - per gestire il controllo

Esempio: Ritiro Contanti



Prelievo di 1000€



Commessa usa SW per richiedere il prelievo

Commessa riceve Ok e fornisce 1000€

Il software richiede aggiornamento DB

Conto

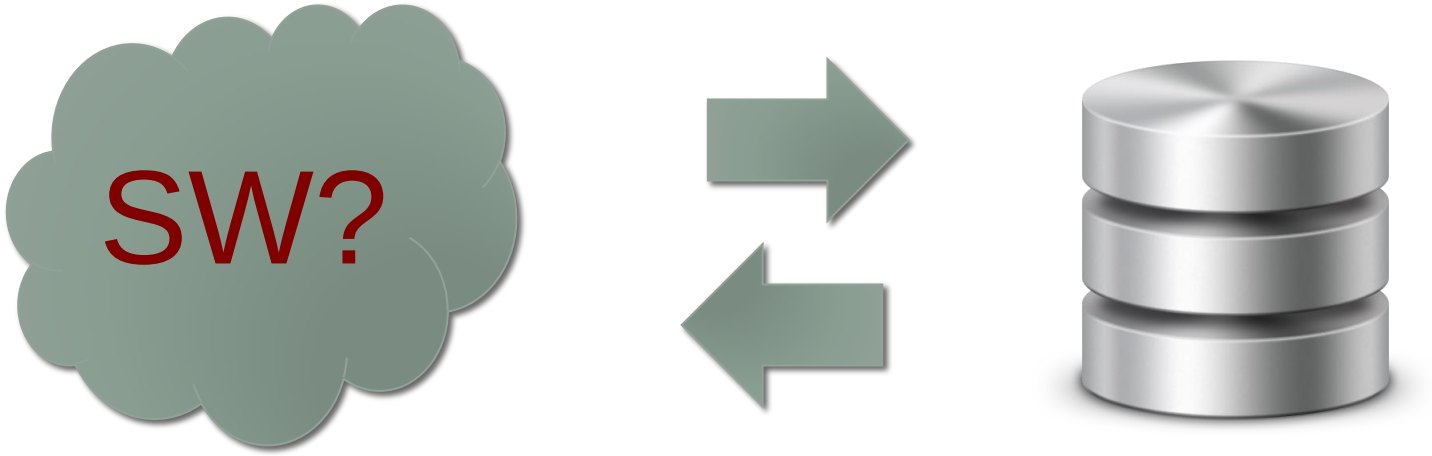
codice	saldo
0011	10.000
0231	1.500

Esempio: Ritiro Contanti

Conto

codice_utente	saldo
0011	9.000
0231	1.500

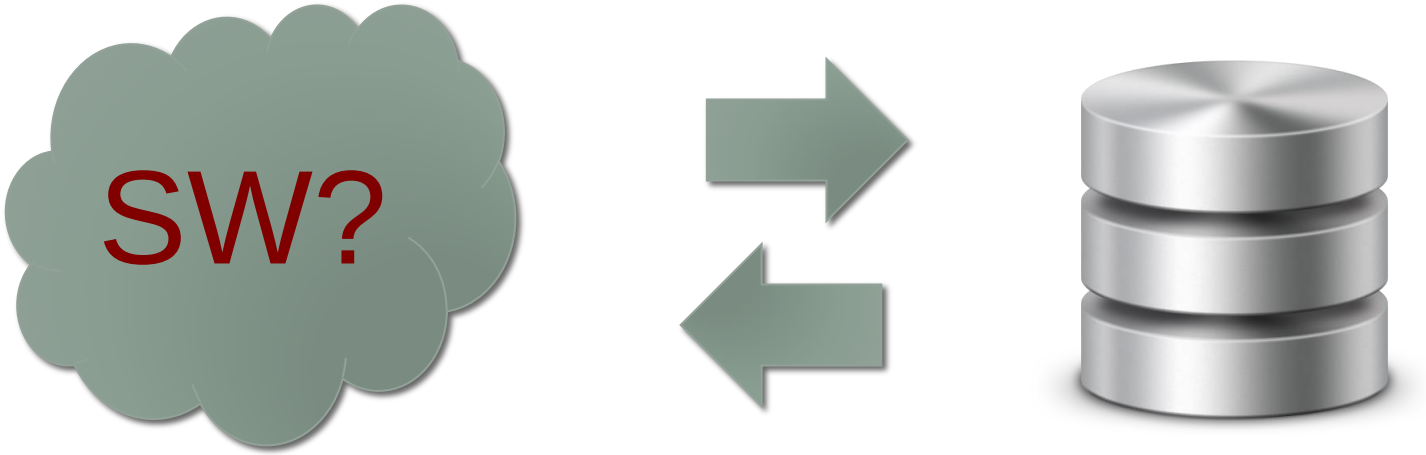
Due mondi



Conflitto di impedenza (“disaccoppiamento di impedenza”) fra base di dati e linguaggio:

- linguaggi: operazioni su singole variabili o oggetti
- SQL: operazioni su relazioni (insiemi di ennuple)

Due mondi



- Tipi di base:
  - linguaggi: numeri, stringhe, booleani
  - SQL: CHAR, VARCHAR, DATE, ...
- Tipi “strutturati” disponibili:
  - linguaggio: dipende dal paradigma
  - SQL: relazioni e ennuple
- Accesso ai dati e correlazione:
  - linguaggio: dipende dal paradigma e dai tipi disponibili; ad esempio scansione di liste o “navigazione” tra oggetti
  - SQL: join (ottimizzabile)

DBMS con più funzionalità

Incremento delle funzionalità di SQL

- Stored procedure
- Trigger
- Linguaggi 4GL

Stored procedure

- Sequenza di istruzioni SQL con parametri
- Memorizzate nella base di dati

```
procedure AssegnaCitta(:Dip varchar(20),
                       :Citta varchar(20))
update Dipartimento
set Città = :Citta
where Nome = :Dip;
```

Estensioni SQL per il controllo

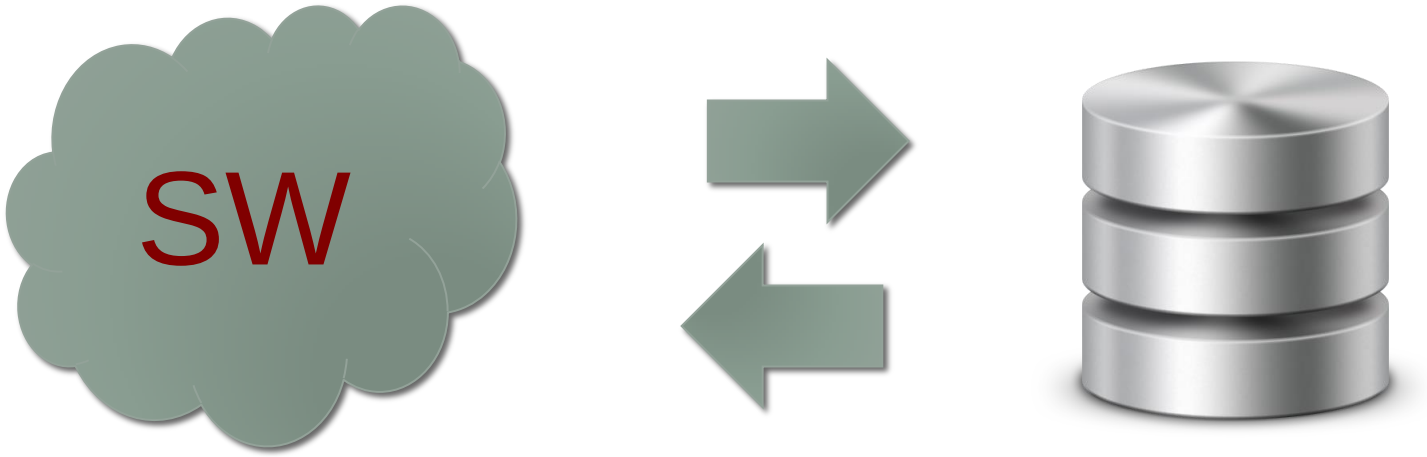
- Esistono diverse estensioni

```
procedure CambiaCittaADip(:NomeDip varchar(20),
                          :NuovaCitta varchar(20))
   if(select *
      from Dipartimento
      where Nome = :NomeDip) = NULL
      insert into ErroriDip values (:NomeDip)
   else
      update Dipartimento
      set Città = :NuovaCitta
      where Nome = :NomeDip;
   end if;
  end;
```

Linguaggi 4GL

- Ogni sistema adotta, di fatto, una propria estensione
- Diventano veri e propri linguaggi di programmazione proprietari “ad hoc”:
  - PL/SQL,
  - Informix4GL,
  - PostgreSQL PL/pgsql,
  - DB2 SQL/PL

SW dialogano con il DBMS



Linguaggi di programmazione (Java, Ruby, Python .. etc):

- SQL immerso (“Embedded SQL”)
- SQL dinamico
- Call Level Interface (CLI):
  - SQL/CLI, ODBC, JDBC

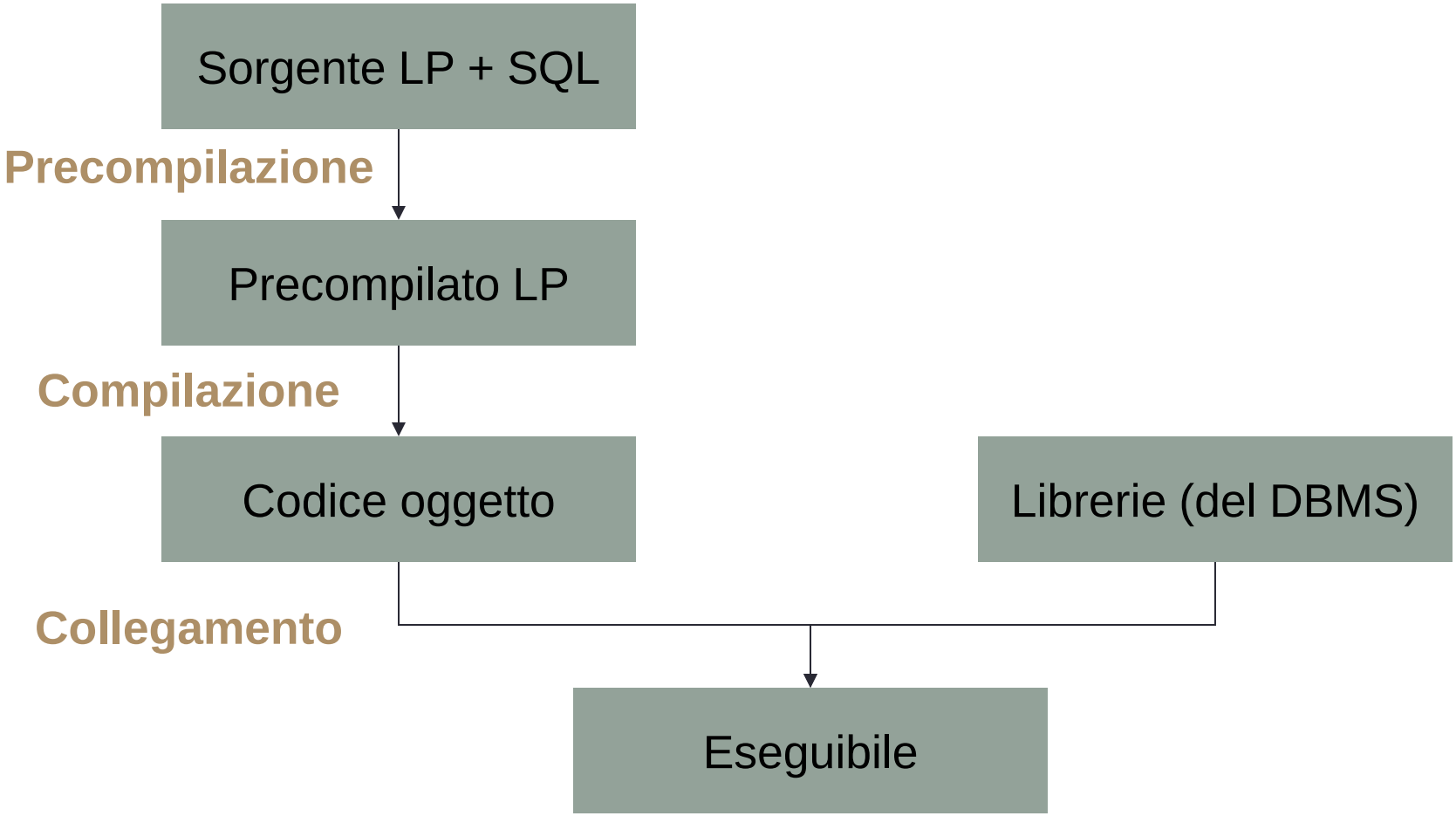
SQL immerso

- le istruzioni SQL sono “immerse” nel programma redatto nel linguaggio “ospite”
- un precompilatore (legato al DBMS) viene usato per analizzare il programma e tradurlo in un programma nel linguaggio ospite (sostituendo le istruzioni SQL con chiamate alle funzioni di una API del DBMS)

SQL immerso, un esempio

```
#include<stdlib.h>
main(){
  exec sql begin declare section;
    char *NomeDip = "Manutenzione";
    char *CittaDip = "Pisa";
    int NumeroDip = 20;
  exec sql end declare section;
  exec sql connect to utente@librobd;
  if (sqlca.sqlcode != 0) {
    printf("Connessione al DB non riuscita\n"); }
  else {
    exec sql insert into Dipartimento
           values(:NomeDip,:CittaDip,:NumeroDip);
    exec sql disconnect all;
  }
}
```

SQL immerso, fasi



Call Level Interface

- Indica genericamente interfacce che permettono di inviare richieste a DBMS per mezzo di parametri trasmessi a funzioni
- standard SQL/CLI (’95 e poi parte di SQL-3)
- ODBC: implementazione proprietaria di SQL/CLI
- JDBC: una CLI per il mondo Java

SQL immerso vs CLI

- SQL immerso permette
  - precompilazione (e quindi efficienza)
  - uso di SQL completo
- CLI
  - indipendente dal DBMS
  - permette di accedere a più basi di dati, anche eterogenee

JDBC

- Una API (Application Programming Interface) di Java (intuitivamente: una libreria) per l'accesso a basi di dati, in modo indipendente dalla specifica tecnologia
- JDBC è una interfaccia, realizzata da classi chiamate driver:
  - l'interfaccia è standard, mentre i driver contengono le specificità dei singoli DBMS (o di altre fonti informative)

I driver JDBC

- (A titolo di curiosità; ne basta uno qualunque)
  Esistono quattro tipi di driver (chiamati, in modo molto anonimo, tipo 1, tipo 2, tipo 3, tipo 4):
  1. Bridge JDBC-ODBC: richiama un driver ODBC, che deve essere disponibile sul client; è comodo ma potenzialmente inefficiente
  2. Driver nativo sul client: richiama un componente proprietario (non necessariamente Java) sul client
  3. Driver puro Java con server intermedio ("middleware server"): comunica via protocollo di rete con il server intermedio, che non deve risiedere sul client
  4. Driver puro Java, con connessione al DBMS: interagisce direttamente con il DBMS

Il funzionamento di JDBC, in breve

- Caricamento del driver
- Apertura della connessione alla base di dati
- Richiesta di esecuzione di istruzioni SQL
- Elaborazione dei risultati delle istruzioni SQL

Un programma con JDBC

```
import java.sql.*;
public class PrimoJDBC {
   public static void main(String[] arg){
     Connection con = null ;
     try {
        Class.forName("sun.jdbc.odbc.JdbcOdbcDriver");
        String url = "jdbc:odbc:Corsi";
        con = DriverManager.getConnection(url);
     } catch(Exception e){
        System.out.println("Connessione fallita");
     }
      try {
        Statement query = con.createStatement();
        ResultSet result =
        query.executeQuery("select * from Corsi");
        while (result.next()){
            String nomeCorso = result.getString("NomeCorso");
            System.out.println(nomeCorso);
        }
   }catch (Exception e){
        System.out.println("Errore nell'interrogazione");
     }
   }
}
```

Un altro programma con JDBC, 1

```
import java.lang.*;
import java.sql.*;
class ProvaSelectJDBC
{
  public static void main(String argv[])
  {
      Connection con = null;
      try {
         Class.forName("com.ibm.db2.jcc.DB2Driver");
      } catch (ClassNotFoundException exClass) {
           System.err.println("Fallita connessione al database. Errore 1");
      }
      try {
            String url = "jdbc:db2:db04";
            con = DriverManager.getConnection(url);
      }
      catch (SQLException exSQL) {
          System.err.println("Fallita connessione al database. "+
          exSQL.getErrorCode() + " " + exSQL.getSQLState() +
           exSQL.getMessage());
      }
```

Un altro programma con JDBC, 2

```
    try{
        String padre = "";  String figlio = "" ; String padrePrec = "";
        Statement query = con.createStatement();
        String queryString =
                "SELECT Padre, Figlio FROM Paternita ORDER BY Padre";
        ResultSet result = query.executeQuery(queryString);
        while (result.next()){
            padre = result.getString("Padre");
            figlio = result.getString("Figlio");
            if (!(padre.equals(padrePrec))){
                System.out.println("Padre: " + padre +
                    "\n  Figli:  " + figlio);}
            else System.out.println( "          " + figlio ) ;
            padrePrec = padre ;
        }
      } catch (SQLException exSQL) {
        System.err.println("Errore nell'interrogazione. "+
           exSQL.getErrorCode() + " " + exSQL.getMessage() );
      }
   }
}
```

Preliminari

- L'interfaccia JDBC è contenuta nel package `java.sql`

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

- Il driver deve essere caricato (trascuriamo i dettagli)

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

- Connessione: oggetto di tipo `Connection` che costituisce un collegamento attivo fra programma Java e base di dati; viene creato da

```
String url = "jdbc:odbc:Corsi";
con = DriverManager.getConnection(url);
```

Preliminari dei preliminari: origine dati ODBC

- Per utilizzare un driver JDBC-ODBC, la base di dati (o altro) deve essere definita come "origine dati ODBC"
- In Windows (con YYY, avendo già definito la base di dati xxx.yyy da collegare):
  - Pannello di controllo
  - Strumenti di amministrazione
  - Opzione "Origini dati ODBC"
  - Bottone "Aggiungi" ("Add")
  - Nella finestra di dialogo "Crea Nuova origine dati" selezionare “YYY Driver" e nella successiva
    - selezionare il file xxx.yyy
    - attribuirgli un nome (che sarà usato da ODBC e quindi da JDBC)

Esecuzione dell'interrogazione ed elaborazione del risultato

Esecuzione dell'interrogazione

```
Statement query = con.createStatement();
ResultSet result =
  query.executeQuery("select * from Corsi");
```

Elaborazione del risultato

```
while (result.next()){
      String nomeCorso =
            result.getString("NomeCorso");
      System.out.println(nomeCorso);
}
```

Statement

- Un'interfaccia i cui oggetti consentono di inviare, tramite una connessione, istruzioni SQL e di ricevere i risultati forniti
- Un oggetto di tipo `Statement` viene creato con il metodo `createStatement` di `Connection`
- I metodi dell'interfaccia `Statement`:
  - `executeUpdate` per specificare aggiornamenti o istruzioni DDL
  - `executeQuery` per specificare interrogazioni e ottenere un risultato
  - `execute` per specificare istruzioni non note a priori
  - `executeBatch` per specificare sequenze di istruzioni
- Vediamo `executeQuery`

ResultSet

- I risultati delle interrogazioni sono forniti in oggetti di tipo **`ResultSet`** (interfaccia definita in **`java.sql`**)
- In sostanza, un result set è una sequenza di ennuple su cui si può "navigare" (in avanti, indietro e anche con accesso diretto) e dalla cui ennupla "corrente" si possono estrarre i valori degli attributi
- Metodi principali:
  - **`next()`**
  - **`get`***`XXX`*`(`*`posizione`*`)`
    - es: **`getString(3); getInt(2)`**
  - **`get`***`XXX`*`(`*`nomeAttributo`*`)`
    - es: **`getString("Cognome"); getInt("Codice")`**

Specializzazioni di Statement

- **`PreparedStatement`** premette di utilizzare codice SQL già compilato, eventualmente parametrizzato rispetto alle costanti
  - in generale più efficiente di **`Statement`**
  - permette di distinguere più facilmente istruzioni e costanti (e apici nelle costanti)

  i metodi **`set`***`XXX`***`( , )`** permettono di definire i parametri
- **`CallableStatement`** premette di utilizzare "stored procedure", come quelle di Oracle PL/SQL o anche le query memorizzate (e parametriche) di Access

ResultSet

- Query: “Select * from conto where saldo > 500”
- `*Statement` e il metodo `executeQuery` invia la query al DBMS

codice_utente	saldo
0011	10.000
0231	1.500
0110	1.050
0123	25
1000	300

ResultSet

- Query: “Select * from conto where saldo > 500”
- `*Statement` e il metodo `executeQuery` invia la query al DBMS
- `ResultSet` cattura il risultato tabellare
  - `next()` itera il “cursore”
  - `getInteger(nomeColonna)` restituisce il valore

codice_utente	saldo
0011	10.000
0231	1.500
0110	1.050
0123	25
1000	300

```
import java.sql.*;
import javax.swing.JOptionPane;
public class SecondoJDBCprep {
   public static void main(String[] arg){
      try {
         Class.forName("sun.jdbc.odbc.JdbcOdbcDriver");
         String url = "jdbc:odbc:Corsi";
         Connection con = DriverManager.getConnection(url);
         PreparedStatement pquery = con.prepareStatement(
              "select * from Corsi where NomeCorso = ?");

        pquery.setString(1,param);
         ResultSet result = pquery.executeQuery();
         while (result.next()){
           String nomeCorso = result.getString("NomeCorso");
           System.out.println(nomeCorso);
        }
      } catch (Exception e){
         System.out.println("Errore");
       }
   }
}
```

```
import java.sql.*;
import javax.swing.JOptionPane;

public class TerzoJDBCcall {
   public static void main(String[] arg){
      try {
         Class.forName("sun.jdbc.odbc.JdbcOdbcDriver");
         String url = "jdbc:odbc:Corsi";
         Connection con = DriverManager.getConnection(url);
         CallableStatement pquery =
              con.prepareCall("{call queryCorso(?)}");
         String param = JOptionPane.showInputDialog(
               "Nome corso (anche parziale)?");
         param = "*" + param + "*";
         pquery.setString(1,param);
         ResultSet result = pquery.executeQuery();
         while (result.next()){
         String nomeCorso = result.getString("NomeCorso");
         System.out.println(nomeCorso);
       }
      }catch (Exception e){
         System.out.println("Errore");
       }
   }
}
```

Altre funzionalità

- Molte, fra cui
  - username e password
  - aggiornamento dei ResultSet
  - richiesta di metadati
  - gestione di transazioni

Transazioni in JDBC

- Scelta della modalità delle transazioni: un metodo definito nell'interfaccia `Connection`:

  `setAutoCommit(boolean autoCommit)`

- **`con.setAutoCommit(true)`**
  - (default) "autocommit": ogni operazione è una transazione
- **`con.setAutoCommit(false)`**
  - gestione delle transazioni da programma

    `con.commit()`

    `con.rollback()`
  - non c'è **`begin transaction`**

Esempio

- Consideriamo la classe **Student**:

```
package 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
}
```

Esempio

- E il database *university:*

```
CREATE TABLE students
(
  code integer NOT NULL,
  firstname character varying(64) NOT NULL,
  lastname character varying(64) NOT NULL,
  birthdate date NOT NULL,
  CONSTRAINT pk_students PRIMARY KEY (code)
)
```

Ambiente

- DBMS
  - a scelta
    (PostgreSQL, MySQL, DB2, Oracle, SQLServer)
  - consigliato: PostgreSQL
- Driver JDBC per il DBMS scelto
  - con postgresql 8.4 scaricare il driver
    http://jdbc.postgresql.org/download/postgresql-8.4-701.jdbc4.jar
  - con altri DBMS, scaricare un driver appropriato
    (se non lo trovate:
    http://developers.sun.com/product/jdbc/drivers)
- Ambiente Java standard
- Nota bene: il **.jar** del driver deve essere nel **`CLASSPATH`**

Le classi fondamentali di JDBC

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

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

Un Esempio

- Nel seguito, con riferimento ad nostro studio di caso, descriveremo le seguenti operazioni
  - Operazione n.1: Caricare il driver
  - Operazione n.2: Aprire una connessione
  - Operazione n.3: Definire l’istruzione SQL
  - Operazione n.4: Gestire il risultato
  - Operazione n.5: Rilascio delle risorse

Operazione n.1: Caricare il Driver

- Creare un oggetto della classe **`Driver`**

```
Driver d = new org.postgresql.Driver();
```

- Registrare il driver sul **`DriverManager`**

```
DriverManager.registerDriver(d);
```

- A questo punto il driver è disponibile

Operazione n.1: Caricare il Driver

- Le operazioni precedenti sono equivalenti alla istruzione:

```
Class.forName("org.postgresql.Driver");
```

- Vedi classe documentazione classe **`java.lang.Class`**
- Soluzione vantaggiosa:
  - Il nome della classe è indicato con una stringa che può essere letta da un file di configurazione
  - Disaccoppiamento dallo specifico DBMS
  - Non è necessario ricompilare il codice se si cambia il DBMS, basta modificare la stringa nel file di configurazione

Operazione n.2: Connessione

- Ottenere una connessione dal **`DriverManager`**
  - per farlo è necessario specificare:
    - host, dbms, db
    - utente e password
- URI (“indirizzo” completo) della connessione
  - specifica server, porta e database
- Sintassi

  `jdbc:<sottoprotocollo>:<parametri>`

Operazione n.2: Connessione

- URI per PostgreSQL

  `jdbc:postgresql:<baseDati>`

  `jdbc:postgresql://<host>/<baseDati>`

  `jdbc:postgresql://<host>:<porta>/<baseDati>`

- URI per Access

  `jdbc:odbc:<sorgenteODBC>`

  NB: la sorgente ODBC deve essere registrata

  - sotto Windows: pannello di controllo, Strumenti di amministrazione, origine dati odbc
  - sotto Linux: file **`odbc.ini`** (o **`.odbc.ini`**)

- Esempi

  `jdbc:postgresql:university`

  `jdbc:postgresql://127.0.0.1/university`

  `jdbc:postgresql://193.204.161.14:5432/university`

  `jdbc:odbc:university`

Operazione n.2: Connessione

- Creazione della connessione

```
Connection DriverManager.getConnection(
        String uri,String utente,String password);
```

- Esempio

```
Connection connection;
connection = DriverManager.getConnection(
"jdbc:postgresql:university”,"postgres"postgres”);
```

- Attenzione: ottenere una connessione è un’operazione costosa
  - creare troppe connessioni comporta problemi di prestazioni
  - nel corso di SIW saranno illustrate tecniche per superare questo problema

Nell’ Esempio

- Confiniamo in una classe, **`DataSource`**, le operazioni necessarie per ottenere la connessione
  - il suo compito è servire connessioni alle altre classi che ne hanno bisogno
  - metodo **`Connection getConnection()`** che restituisce una nuova connessione ad ogni richiesta
- E' una soluzione artigianale usata solo a fini didattici

La classe DataSource

```
import java.sql.*;

public class DataSource {
  private String dbURI = "jdbc:postgresql://localhost/university";
  private String user = "postgres";
  private String password = "postgres";

  public Connection getConnection() throws Exception {
        Class.forName("org.postgresql.Driver");
        Connection connection = DriverManager.getConnection(dbURI, user,
  password);
        return connection;
  }
}
```

Operazione n.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 **`StudentRepository`**
- Concentriamoci in particolare sul codice dei singoli metodi, piuttosto che del progetto di tale classe

Operazione n.3: Istruzione SQL

- 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);
```

Operazione n.3: Istruzione SQL

- 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));
```

Operazione n.3: Istruzione SQL

- 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

- Per i dettagli vedi la documentazione

Operazione n.3: Istruzione SQL

- 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
  - interrogazioni (select)
    - non modificano lo stato del database
    - ritornano una sequenza di tuple

Operazione n.3: Istruzione SQL

- Aggiornamenti (insert, delete, update)
  - vengono eseguiti invocando il metodo **`executeUpdate()`** sull'oggetto **`PrepareStatement`**

- Esempio 1 (cont.)

```
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();
```

Operazione n.3: Istruzione SQL

- Interrogazioni (select)
  - vengono eseguiti invocando il metodo **`executeQuery()`**
  - che ritorna il risultato in un oggetto **`ResultSet`**
- Esempio 2 (cont.)

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

Operazione n.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);`

Operazione n.4: Gestire il risultato di una query

- Il metodo **next()**
  - *Moves the* **cursor forward** *one row from its current position.*
    A **ResultSet** *cursor is initially positioned before the first row;*
    the first call to the method next makes the first row the current row;
    the second call makes the second row the current row, and so on.
  - *When a call to the next method returns false, the cursor is positioned after the last row. Any invocation of a* **ResultSet** *method which requires a current row will result in a* **SQLException** *being thrown.*
  - *Returns:* **true** *if the new current row is valid;* **false** *if there are no more rows*

Operazione n.4: Gestire il risultato di una query

```
Connection connection = this.dataSource.getConnection();
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);
}
```

Operazione n.4: Gestire il risultato di una query

```
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);
}
```

Operazione n.5: rilascio delle risorse

- `connection`, `statement`, `ResultSet` devono essere sempre "chiusi" dopo essere stati usati
- l'operazione di chiusura corrisponde al rilascio di risorse
- L'approccio seguito in queste lezioni è semplicistico, un modo più sofisticato e corretto richiede la conoscenza del meccanismo delle eccezioni che verrà introdotto nel corso di "Sistemi informativi su Web"

Esercizio

- Studiare il codice della classe

  `StudentRepository`

- 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

- Creare il database *olympic:*

```
CREATE TABLE athletes
(
  code integer NOT NULL,
  name character varying(64) NOT NULL,
  nation character varying(64) NOT NULL,
  birthdate date NOT NULL,
  height double precision,
  CONSTRAINT pk_athletes PRIMARY KEY (code)
)
```

Esercizio

- Creare la classe **Athlete**:

```
package olympic;
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

- Scrivere e testare il codice della classe **`AthleteRepository`**

- 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`**