JDBC: SQL nei linguaggi di programmazione

Matteo Cannaviccio
mcannaviccio@gmail.com

Basi di Dati I
2017/18

Panoramica

1. Dialogare con un DBMS
   - Applicazioni software ⟷ DBMS
2. Introduzione a **JDBC**
   - Scopo
   - Possibili architetture
   - Accesso al database
3. Esercizio

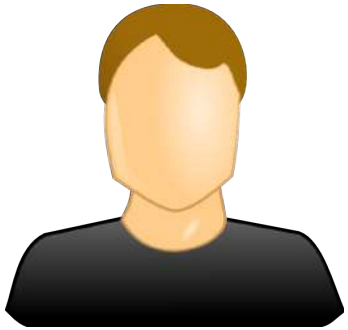
Panoramica

1. Dialogare con un DBMS
   - Applicazioni software ⟷ DBMS
2. Introduzione a JDBC
   - Scopo
   - Possibili architetture
   - Accesso al database
3. Esercizio

SQL e Applicazioni

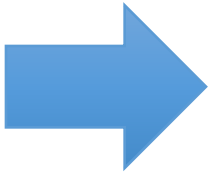
- I database immagazinano **informazioni strutturate** sottoforma di **ennuple** e **relazioni**, che possono essere accedute tramite comandi **SQL**
- In applicazioni complesse, l'utente alla ricerca di una data informazione non vuole eseguire comandi SQL
- Inoltre SQL non basta, per offrire un servizio sono necessarie altre funzionalità:
  - gestire input (scelte dell'utente e parametri)
  - gestire output (con dati che non sono relazioni o se si vuole una presentazione complessa)
  - gestire il controllo

Esempio: Ritiro Contanti



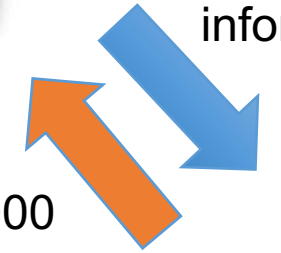
Utente 0011

Prelievo di 1000€



ATM verifica che l'utente può richiedere il prelievo

Il software richiede di **visualizzare** un informazione nel DB

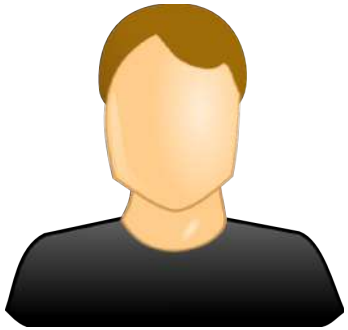


10.000

Conti

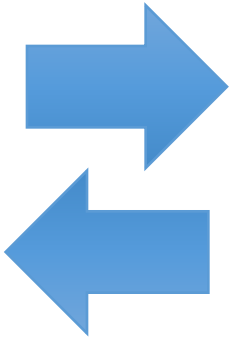
codice	saldo
0011	10.000
0231	1.500

Esempio: Ritiro Contanti



Utente 0011

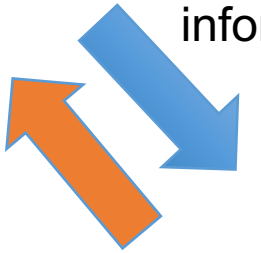
Prelievo di 1000€



ricevuta

ATM conferma prelievo

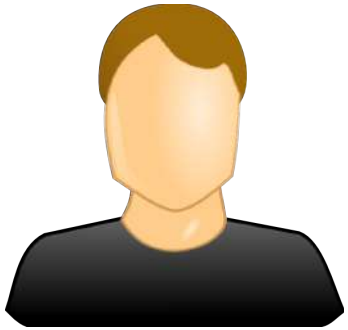
Il software richiede di **modificare** un informazione nel DB



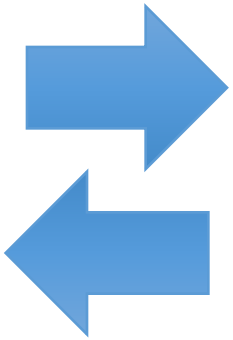
Conti

codice	saldo
0011	**9.000**
0231	1.500

Esempio: Ritiro Contanti



Utente 0011

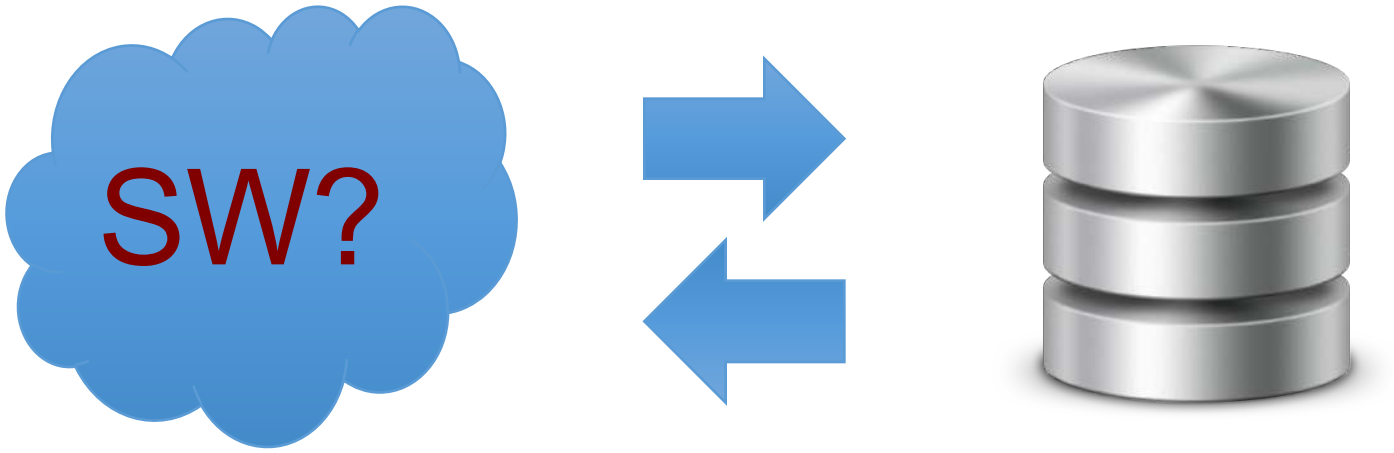


SW?

Conti

codice	saldo
0011	**9.000**
0231	1.500

Due mondi



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

Due mondi: conflitto d'impedenza

Applicazione Software		DBMS
Linguaggio di programmazione	**Linguaggio**	SQL
Numeri, Stringhe, Booleani	**Tipi di base**	VARCHAR, DATE
Oggetti/variabili in strutture dati	**Tipi strutturati**	Ennuple in relazioni
Java, Python, Ruby, etc.	**Esempio**	Stored Procedure, Linguaggi 4GL

Stored Procedure

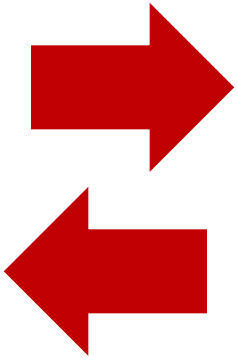
DBMS
SQL
VARCHAR, DATE
Ennuple in relazioni
Stored Procedure, Linguaggi 4GL

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

```
procedure AssegnaCitta(: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;
```

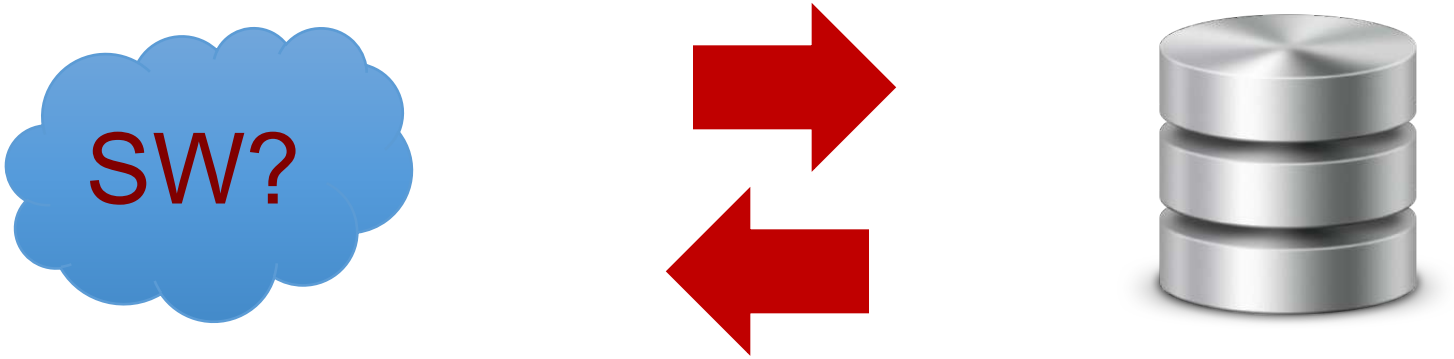
- Ogni DBMS adotta la propria versione

SW dialogano con il DBMS



Applicazione Software		DBMS
Linguaggio di programmazione	**Linguaggio**	SQL
Numeri, Stringhe, Booleani	**Tipi di base**	VARCHAR, DATE
Oggetti/variabili in strutture dati	**Tipi strutturati**	Ennuple in relazioni
Java, Python, Ruby, etc.	**Esempio**	Stored Procedure, Linguaggi 4GL

Soluzioni più comuni



1. **SQL immerso (“embedded”)**
2. **Call Level Interface (CLI)**

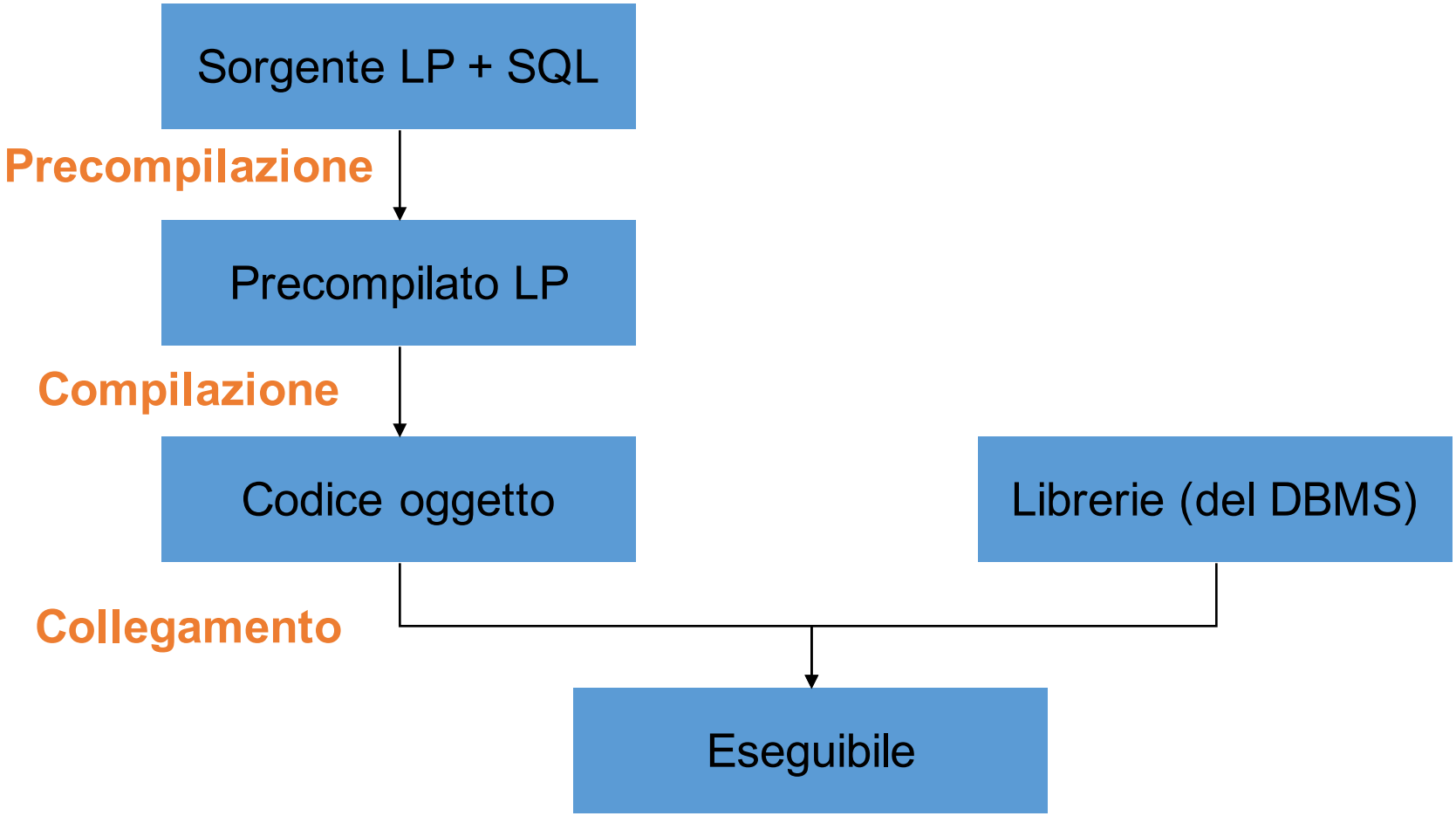
SQL immerso

- Il *linguaggio ospite* contiene statement SQL (opportunamente evidenziati).
- Il codice sorgente e l'SQL sono **pre-processati**.
- Un compilatore (legato al DBMS) precompila il sorgente, traducendo invocazioni SQL in chiamate al DBMS
- Utilizzano comandi SQL specifici, ad esempio:
  - `exec sql` delimitare l'SQL nel linguaggio
  - `declare` dichiarare variabili condivise tra SQL e linguaggio

SQL immerso

```
#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



SQL immerso

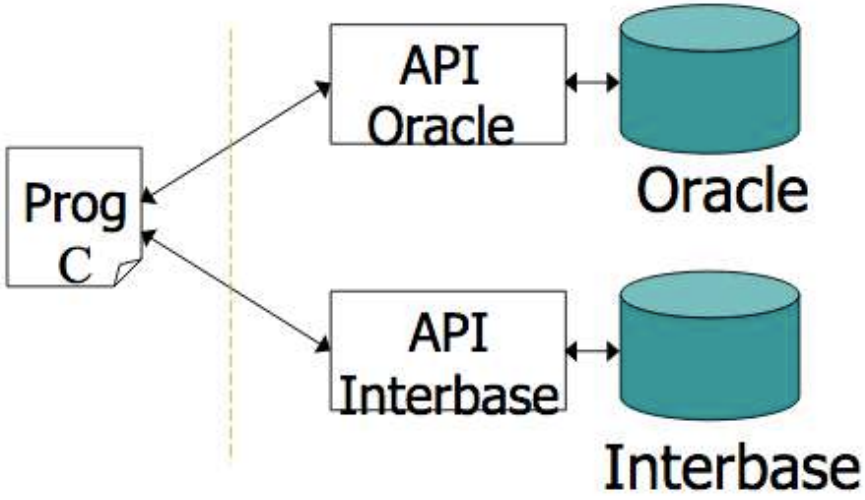
- **Vantaggi**
  - prestazioni (precompilazione ottimizzata)

    (analisi e ottimizzazione della query avviene durante la compilazione, quindi una sola volta)

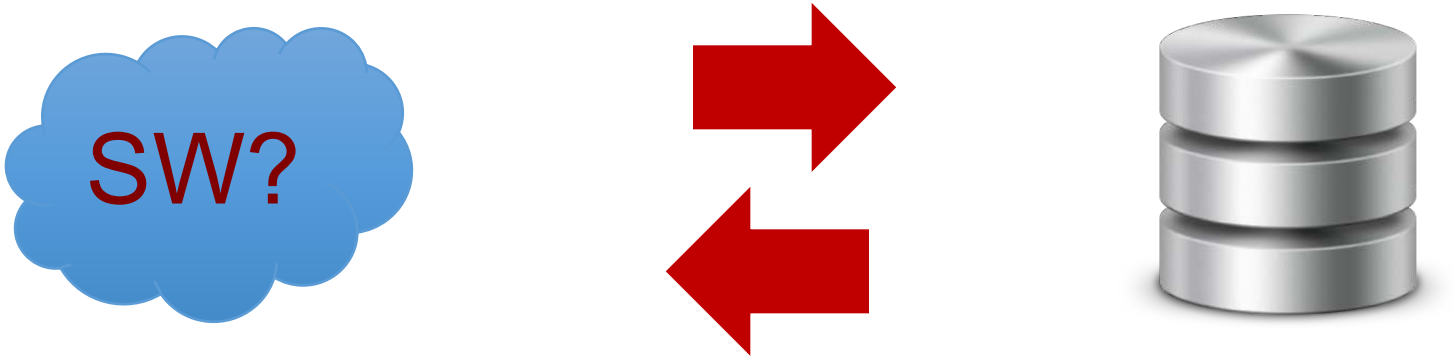
- **Svantaggi**
  - limiti di portabilità
  - applicazione dipende dal precompilatore
  - difficile sviluppare su DBMS diversi

CLI – Call Level Interface

- Approccio più utilizzato.
- Si scrive codice ordinario nel linguaggio di programmazione ospite.
- Utilizzo di una libreria di funzioni per:
  - connettersi al DBMS
  - eseguire statement SQL



Soluzioni più comuni



1. **SQL immerso (“embedded”)**
   - precompilazione (e quindi efficienza)
   - uso di SQL completo

2. **Call Level Interface (CLI)**
   - indipendente dal DBMS
   - permette di accedere a più basi di dati, anche eterogenee
   - un esempio: **JDBC**

Panoramica

1. Dialogare con un DBMS
   - Applicazioni software ⟷ DBMS
2. Introduzione a JDBC
   - Scopo
   - Possibili architetture
   - Accesso al database
3. Esercizio

JDBC: Java Database Connectivity

- Libreria standard per accedere a DBMS relazionali
- Compatibile con la maggior parte di database
- Definita nei package `java.sql` e `javax.sql`
- Standardizza il meccanismo di accesso al DB

- Riferimenti:
  - Overview: http://www.oracle.com/technetwork/java/overview-141217.html
  - Tutorial: http://docs.oracle.com/javase/tutorial/jdbc/TOC.html

Scopo di JDBC

Abilitare applicazioni Java ad accedere ai DBMS attraverso le operazioni di base CRUD

(Create, Read, Update, Delete)

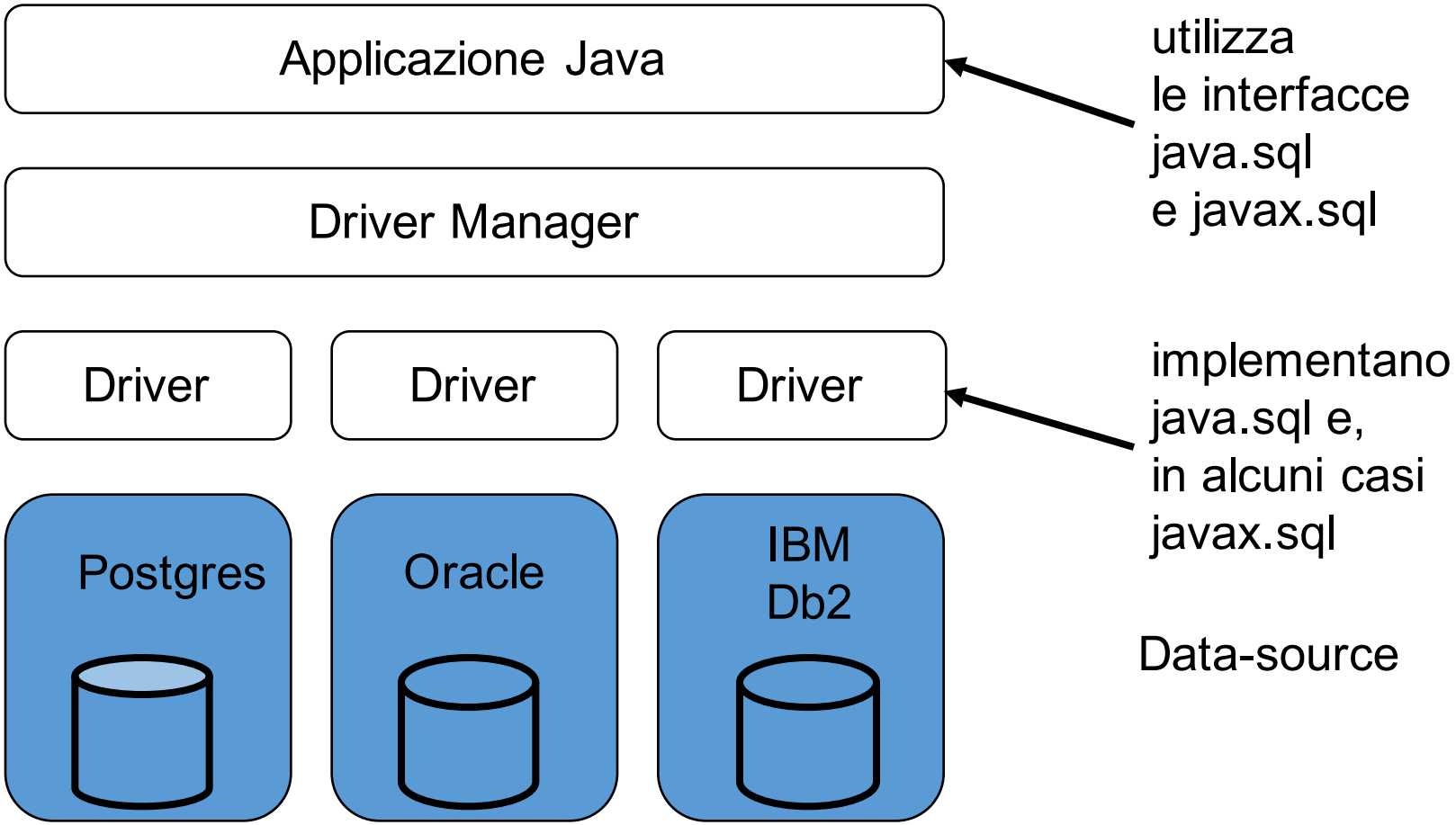
Ogni DBMS (PostgreSQL, MySQL, etc..) offre:

1. Un meccanismo per la connessione al DBMS;
2. Sintassi per inviare una query;
3. Una struttura per rappresentare il risultato

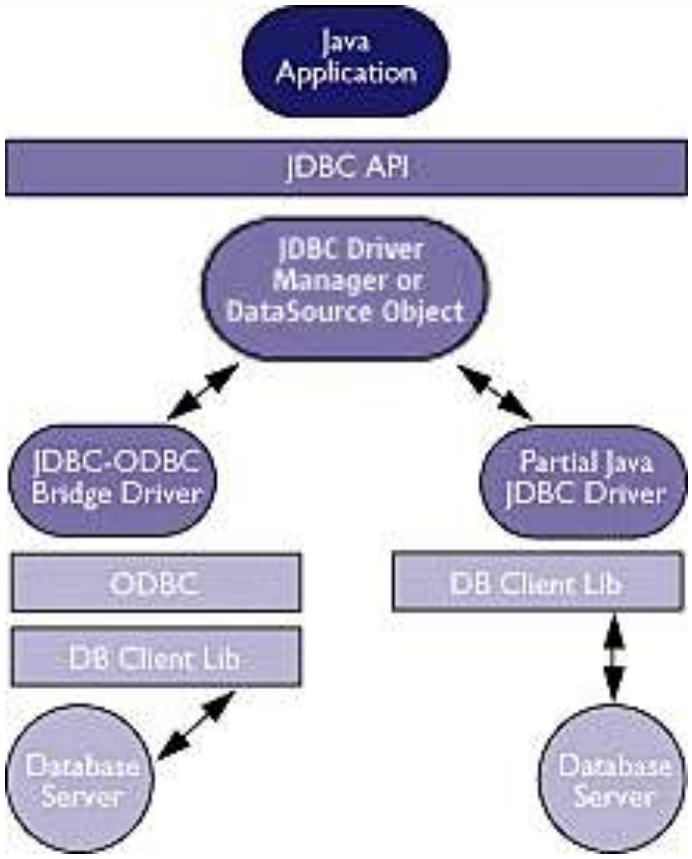
La **libreria JDBC**:

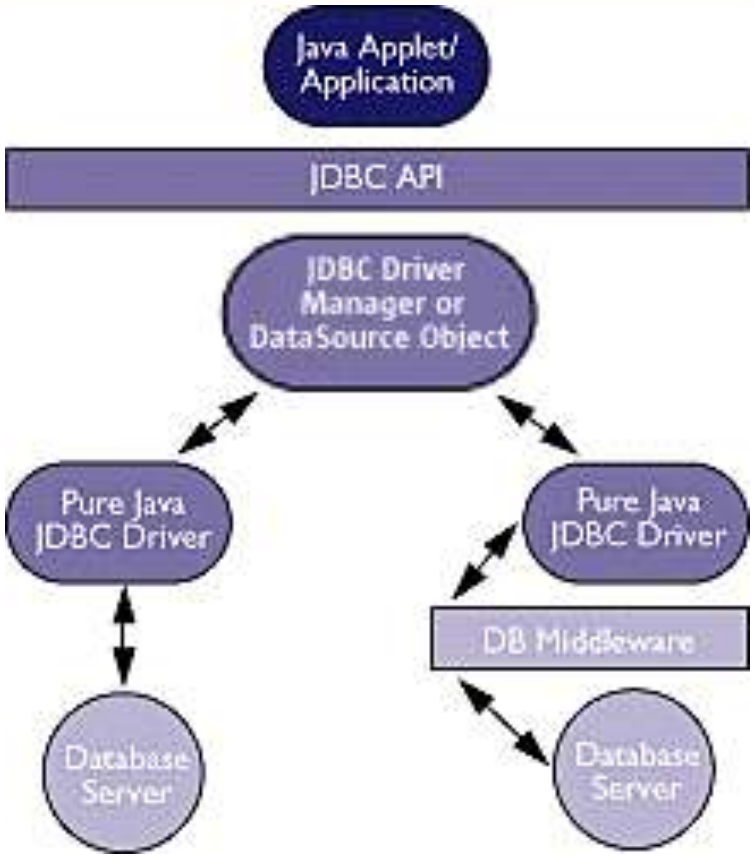
- astrae il processo (1,2,3) dallo specifico **Driver**;
- nasconde le differenze tra i diversi database (che le fanno internamente in modo diverso!)

Componenti principali



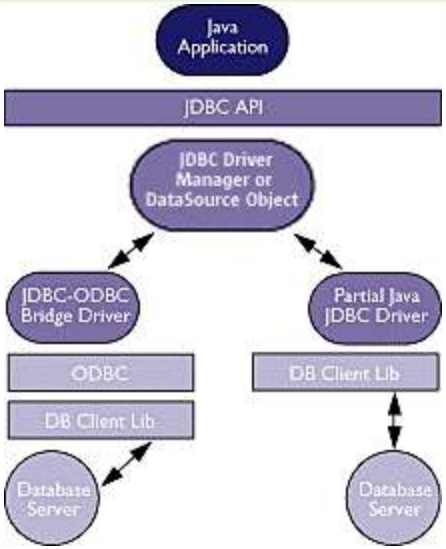
Possibili tipologie di driver JDBC





Type 1 **Type 2** **Type 4** **Type 3**

Driver: Type1, Type2

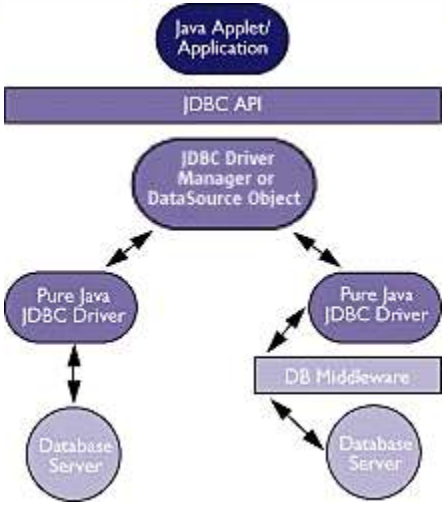


- ***Type 1***:
  JDBC-ODBC Bridge plus ODBC Driver
  This combination provides JDBC access via ODBC drivers. **ODBC** binary code -- and in many cases, database client code -- **must be loaded on each client machine** that uses a JDBC-ODBC Bridge.

- ***Type 2***:
  A native API partly Java technology-enabled driver
  This type of driver converts JDBC calls into calls on the client API for Oracle, Sybase, Informix, DB2, or other DBMS.

  Note that, like the bridge driver, this style of driver requires that some binary code be loaded on each client machine.

Driver: Type4, Type3



- ***Type 4***:
  Direct-to-Database Pure Java Driver
  This style of driver converts JDBC calls into the network protocol used directly by DBMSs, allowing **a direct call from the client machine to the DBMS** server and providing a practical solution for intranet access.

- ***Type 3***:
  Pure Java Driver for Database Middleware
  This style of driver translates JDBC calls into the **middleware** vendor's protocol, which is then translated to a DBMS protocol by a middleware server. The middleware provides connectivity to many different databases.

Confronto tra driver

Type 1 (Ponte JDBC-ODBC)
- prestazioni scadenti
- non indipendente dalla piattaforma
- fornito a corredo di SDK

Type 2
- migliori prestazioni
- non indipendente dalla piattaforma

Type 3
- client indipendente dalla piattaforma
- servizi avanzati (caching)
- architettura complessa

Type 4
- indipendente dalla piattaforma
- buone prestazioni
- scaricabile dinamicamente

Per approfondimenti:
http://java.sun.com/products/jdbc/driverdesc.html

Panoramica

1. Dialogare con un DBMS
   - Applicazioni software ⟷ DBMS
2. Introduzione a JDBC
   - Scopo
   - Possibili architetture
   - Accesso al database
3. Esercizio

Accesso al database

L'accesso al database avviene tramite:

1. Creazione connessione
   `java.sql.Connection`

2. Esecuzione istruzione SQL
   `java.sql.Statement`

3. Elaborazione dei risultati
   `java.sql.ResultSet`

4. Rilascio delle risorse

Esempio tabella

```
CREATE TABLE prodotti
(
  cp character varying(64) NOT NULL,
  nome character varying(64) NOT NULL,
  marca character varying(64) NOT NULL,
  modello character varying(64) NOT NULL,
  CONSTRAINT pk_prodotti PRIMARY KEY (cp)
)
```

Esempio programma JDBC

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

public class TestJDBC {public static void main(String[] args) {
  // carica il driver
  try {
      Class.forName("org.postgresql.Driver");
  } catch (ClassNotFoundException e) {
      e.printStackTrace();
  }

  try {
         // crea una connessione
         Connection c = DriverManager.getConnection("jdbc:postgresql:nome_db", “user", “pass");

         // crea uno statement
         String query = "SELECT * from prodotti”;
         PreparedStatement st = connection.prepareStatement(query);

         // itera risultati
         ResultSet rs = st.executeQuery();
         while (rs.next()){
            System.out.println(“Nome prodotto: ” + rs.getString(“nome”));
         }

         // rilascia risorse
         st.close()
         rs.close()
         c.close();

   } catch (SQLException e) {
       e.printStackTrace();
   }
}
```

Esempio programma JDBC

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

public class TestJDBC {public static void main(String[] args) {
  // carica il driver
```

Carica il driver

Crea una connessione

Crea uno statement

Gestisce il risultato (itera)

Rilascia le risorse

```
    } catch (SQLException e) {
        e.printStackTrace();
    }
}
```

Accesso al database - Panoramica

L'accesso al database avviene tramite:

1. **Creazione connessione**
   `java.sql.Connection`

2. Esecuzione istruzione SQL
   `java.sql.Statement`

3. Elaborazione dei risultati
   `java.sql.ResultSet`

4. Rilascio delle risorse

Creazione connessione

1. Caricare il driver
2. Definire connessione (*parametri*)
3. Stabilire la connessione

Caricare il Driver

Un driver è una classe messa a disposizione dallo specifico fornitore DBMS (PostgreSQL, MySQL,..)

- Deve essere disponibile all'applicazione Java
- L'applicazione non la conosce direttamente (per facilitare la migrazione verso altri DBMS)
- Ad esempio, usando PostgreSQL,viene caricata una classe Driver a ***run-time*** tramite l'istruzione:

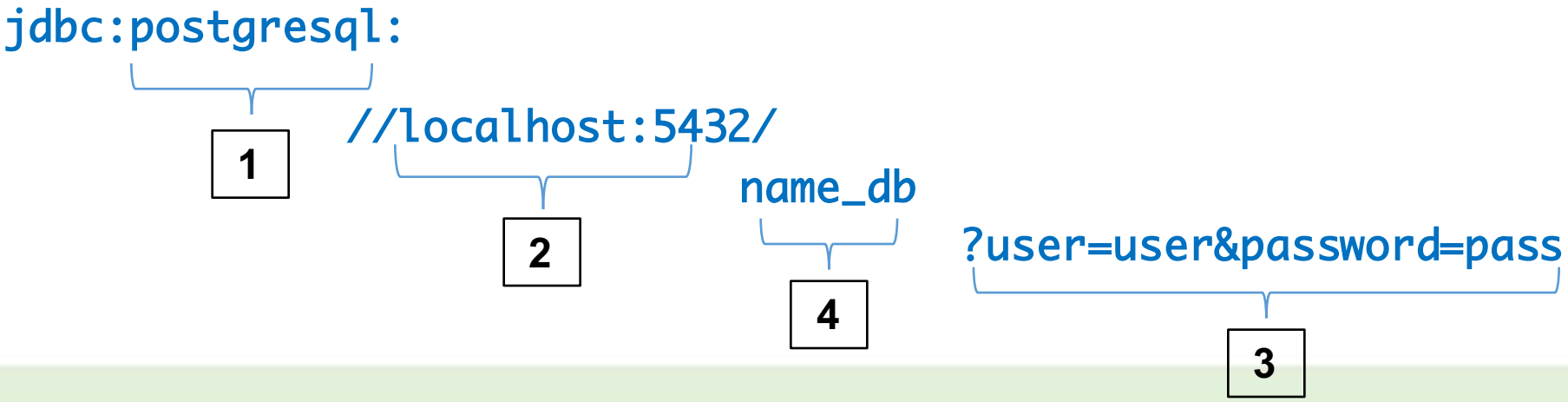
```
Class.forName("org.postgresql.Driver")
```

Definire una connessione

Il **DriverManager** ha bisongo di alcune informazioni:

1. Tipo di database (postgresql, db2, mysql, etc.)
2. L'indirizzo del server (es. "127.0.0.1, localhost")
3. Informazioni per l'autenticazione (user/pass)
4. Database / Schema al quale connettersi

Tutti questi parametri vengono concatenati in una stringa (la sintassi può variare leggermente tra diversi DB):



Stabilire la connessione

Il **DriverManager** ci restituisce un oggetto `java.sql.Connection` tramite il metodo `getConnection()`:

```
Connection c =
        DriverManager.getConnection(stringa_connessione)
```

- Segue le istruzioni fornite nella stringa URL
- Restituisce un oggetto Connection

Alternativa:

```
Connection c =
DriverManager.getConnection(
        “jdbc:postgresql:name_db”, “user”, “pass”)
```

Nota: Sostituire i valori name_db, user e pass!

Esempio connessione

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

public class TestJDBC {public static void main(String[] args) {
  // carica il driver
  try {
      Class.forName("org.postgresql.Driver");
  } catch (ClassNotFoundException e) {
      e.printStackTrace();
  }

  try {
          // crea una connessione
          Connection c = DriverManager.getConnection("jdbc:postgresql:nome_db", “user", “pass");

          // ... utilizza la connessione ...

          // chiudi la connessione
          c.close();

   } catch (SQLException e) {
       e.printStackTrace();
   }
}
```

Accesso al database - Panoramica

L'accesso al database avviene tramite:

1. Creazione connessione
   `java.sql.Connection`

2. **Esecuzione istruzione SQL**
   `java.sql.Statement`

3. Elaborazione dei risultati
   `java.sql.ResultSet`

4. Rilascio delle risorse

Creare uno Statement

Da un'istanza di Connection si possono creare oggetti `java.sql.Statement` che ci consentono di interrogare il DBMS.

Esistono due tipi di Statement:

1. `createStatement()`
   - Restituisce un oggetto di tipo **Statement**.
   - Non ha associata alcuna query SQL.

2. `prepareStatement(Q)`
   - Restituisce un oggetto di tipo **PreparedStatement**.
   - Q è una query SQL passata come stringa.

Eseguire una query

Eseguire una query a questo punto consiste nel richiamare uno dei seguenti metodi:

1. `executeQuery(Q)`
   - Si applica ad uno **Statement**
   - Esegue la query Q e ritorna un ResultSet con le tuple del risultato
2. `executeQuery()`
   - Si applica ad un **PreparedStatement**
   - Esegue la query associata al PreparedStatement e ritorna un ResultSet con le tuple del risultato

Eseguire un update

Un update è una query che modifica lo stato del database ma non restituisce alcun risultato.

1. `executeUpdate(U)`
   - Si applica ad uno **Statement**
   - U è un non-query statement:
     INSERT, UPDATE, DELETE
   - Ritorna il numero di righe aggiornate
     (0 se non significativo).

2. `executeUpdate()`
   - Si applica ad un **PreparedStatement**
   - Agisce quando il PreparedStatement ha associato un non-query statement SQL.
   - Ritorna il numero di righe aggiornate
     (0 se non significativo).

Esempio Statement

```
try {
   // crea una connessione
   Connection  c = DriverManager.getConnection("jdbc:postgresql:nome_DB“,“user",  “pass");

   // utilizza la connection

   Statement st = c.createStatement();
   String query = "SELECT * from prodotti";
   ResultSet rs = st.executeQuery(query);

   // chiudi connessione
   c.close();

} catch (SQLException  e) {
       e.printStackTrace();
}}
```

Esempio PreparedStatement

```
try {
   // crea una connessione
   Connection  c = DriverManager.getConnection("jdbc:postgresql:nome_DB“,“user",  “pass");

   // utilizza la connection

   String query = "SELECT * from prodotti";
   PreparedStatement  st = c.prepareStatement(query);
   ResultSet  rs = st.executeQuery();

   // chiudi connessione
   c.close();

} catch (SQLException  e) {
        e.printStackTrace();
}}
```

Statements & PreparedStatements

I **PreparedStatement**:

- separano la creazione e esecuzione dello Statement
- sono compilati (nel DBMS) immediatamente al momento della creazione.
- non devono essere ricompilati ad ogni esecuzione.

Sono preferibili perché permettono di eseguire ripetutamente la query, mettendo in comune:

- Parsing
- Calcolo (di parte) del piano d'accesso
- Compilazione

Si possono scegliere gli **Statement** semplici quando si è sicuri che la query verrà eseguita una sola volta.

PreparedStatements: passaggio dei parametri

In fase di creazione:

- viene definita la sintassi SQL (**template**) con dei segnaposto per le quantità variabili (**parametri**)
- i segnaposto sono indentificati da (**?**) nel testo della query e da un valore numerico che ne indica la posizione (partendo da 1)

Ad es.

```
String query =
"SELECT nome from prodotti where CP = ?";
```

PreparedStatements: passaggio dei parametri

In fase di esecuzione:

- vengono definite le quantità variabili
- viene eseguita la query

L'assegnazione dei parametri avviene assegnando la variabile Java alla posizione del segnaposto.

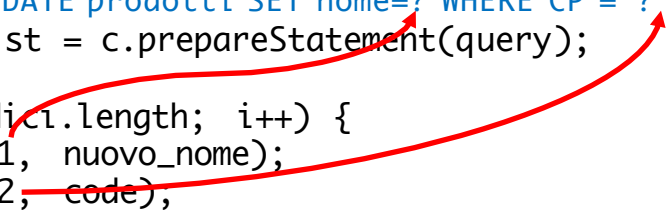
- **`setString(int pos, String value)`**
- **`setInt(int pos, int value)`**
- …

Ad es.

```
String productId = "21de4"
st.setString(1, productId);
```

Esempio PreparedStatement

```
String nuovo_nome = “Laptop“;
String[ ] codici = {“21de4“,“43f23“,“43d24“};

try {
  // crea una connessione
  Connection c = DriverManager.getConnection("jdbc:postgresql:nome_DB", “user", “pass");

  String query = "UPDATE prodotti SET nome=? WHERE CP = ?";
  PreparedStatement st = c.prepareStatement(query);

  for(int i=0; i<codici.length; i++) {
       st.setString(1, nuovo_nome);
       st.setString(2, code);
       st.executeUpdate();
   }

   // chiudi connessione
   c.close();

} catch (SQLException e) {
       e.printStackTrace();
}}
```



Accesso al database - Panoramica

L'accesso al database avviene tramite:

1. Creazione connessione
   `java.sql.Connection`

2. Esecuzione istruzione SQL
   `java.sql.Statement`

3. **Elaborazione dei risultati**
   `java.sql.ResultSet`

4. Rilascio delle risorse

Elaborazione dei risultati

I risultati delle interrogazioni sono forniti in oggetti di tipo `java.sql.ResultSet`:

- è una sequenza di ennuple
- si può "navigare" una riga alla volta (default in avanti)
- si possono estrarre i valori degli attributi dalla ennupla "corrente"

Metodi principali:

- `next()`
- `getXXX(posizione)`
  - es: `getString(3); getInt(2)`
- `getXXX(nomeAttributo)`
  - es: `getString("Cognome"); getInt("Codice")`

Elaborazione dei risultati (2)

“SELECT * FROM conto WHERE saldo > 1000”

`Il ResultSet` cattura il risultato tabellare

`next()`

- itera il “cursore” (si muove sulla tupla successiva)
- ritorna “false” se non ci sono più tuple

`getInt(“saldo”)`

`getInt(2)`

- restituisce il valore nella seconda colonna

codice_utente	saldo
0011	10.000
0231	1.500
0110	900
0123	25
1000	300

`ResultSet`

codice_utente	saldo
0011	**10.000**
0231	1.500

Esempio ResultSet

```
try {
  // crea una connessione
  Connection c = DriverManager.getConnection("jdbc:postgresql:nome_DB", “user", “pass");

   // fai qualcosa con la connection
   Statement st = c.createStatement();
   String query = "SELECT * from prodotti";
   ResultSet rs = st.executeQuery(query);

   while (rs.next()){
     String codice_prodotto = rs.getString(“CP”);
     String nome_prodotto = rs.getString(“nome”);
     String marca_prodotto = rs.getString(“marca”);
     String modello_prodotto = rs.getString(“modello”);

     System.out.println(nome_prodotto + “\t” + marca_prodotto = + “\t” + modello_prodotto)

   }
   // chiudi connessione e resultset
   rs.close();
   c.close();

} catch (SQLException e) {
      e.printStackTrace();
}}
```

Accesso al database - Panoramica

L'accesso al database avviene tramite:

1. Creazione connessione
   `java.sql.Connection`

2. Esecuzione istruzione SQL
   `java.sql.Statement`

3. Elaborazione dei risultati
   `java.sql.ResultSet`

4. **Rilascio delle risorse**

Rilascio delle risorse

Al termine del loro utilizzo Connection, Statement e ResultSet devono essere “chiusi”, invocando il metodo `close()`

Per default:

- ResultSet chiuso automaticamente alla chiusura dello Statement che l’ha creato.
- Statement chiuso automaticamente alla chiusura della Connection che l’ha generato.

Ma:

- Possono verificarsi problemi: chiudere preventivamente è più economico localmente.
- I default possono non valere in environment particolari (ad es. chiusura Connection con pooling).

Attenzione: una grande quantità di problemi critici è dovuta a risorse JDBC non rilasciate correttamente.

Rilascio delle risorse (2)

```
Connection c = null;
PreparedStatement st = null;
ResultSet rs = null;

try {


// … codice che usa JDBC...


} catch (SQLException e) {
    e.printStackTrace();


} finally {

    if (rs!=null)
        try {
            rs.close();
        } catch(SQLException e) {
            e.printStackTrace();
        }

    if(st!=null)
        try {
            st.close();
        } catch(SQLException e) {
            e.printStackTrace();
        }

    if (connection!=null)
        try {
            c.close();
        } catch(SQLException e) {
            e.printStackTrace();
        }

}
```

Panoramica

1. Dialogare con un DBMS
   - Applicazioni software ⟷ DBMS
2. Introduzione a JDBC
   - Scopo
   - Possibili architetture
   - Accesso al database
3. Esercizio

Ambiente

- DBMS
  - a scelta
    (PostgreSQL, MySQL, DB2, Oracle, SQLServer)
  - consigliato: PostgreSQL
- Driver JDBC per il DBMS scelto
  - con postgresql 9.4 scaricare il driver
    https://jdbc.postgresql.org/download/postgresql-9.4.1212.jre6.jar
  - con altri DBMS, scaricare un driver appropriato
- Ambiente Java standard
- Nota bene: il **.jar** del driver deve essere nel **`CLASSPATH`**

Esercizio

Implementare **DbHandler**, una classe Java che ha l'obiettivo di nascondere tutti i dettagli di creazione di una connessione.

In particolare dovrà implementare i seguenti metodi:

- **getConnection()** restituisce un oggetto Connection
- **closeConnection()** chiude la connessione attiva

Ed inoltre:

- **getPreparedStatement(String query)**
  restituisce un PreparedStatement data una query

Esercizio (DbHandler)

```
public class DbHandler {

    private Connection con;
    private final Properties connectionProperties;

    private final static String USER = "postgres";
    private final static String PASSWORD = "mypass";
    private final static String DB_URL = "jdbc:postgresql:prodotti_fornitori";

    private static final DbHandler ourInstance = new DbHandler();
    public static DbHandler getInstance() {
        return ourInstance;
    }

    private DbHandler() {

        this.connectionProperties = new Properties();
        this.connectionProperties.put("user", DbHandler.USER);
        this.connectionProperties.put("password", DbHandler.PASSWORD);

        try {
            Class.forName("org.postgresql.Driver");
        } catch (ClassNotFoundException e) {
            e.printStackTrace();
            System.exit(1);
        }
    }
    private Connection getConnection() throws SQLException {
        if (this.con == null || this.con.isClosed()) {
            this.con = DriverManager.getConnection(DB_URL, this.connectionProperties);
        }
        return this.con;
    }

    public PreparedStatement getPreparedStatement(String query) throws SQLException {
        return this.getConnection().prepareStatement(query);
    }

    public void closeConnection() throws SQLException {
        if(this.con!=null)
            this.con.close();
    }
```

Esercizio

Con riferimento alla base di dati **“prodotti_fornitori”**:

Fornitori (CF, Nome, Indirizzo, Città)

Prodotti (CP, Nome, Marca, Modello)

Catalogo (F, P, Costo)

Prodotto

toString(): che restituisce una stringa con codice, nome, marca e modello di un prodotto.

Fornitore

printFornitoreById(): che stampa, per il fornitore corrispondente, tutti i prodotti forniti da quel fornitore, con codice, nome, marca e modello.

Creare il database

- Andare su pgAdmin e creare un nuovo database chiamato “prodotti_fornitori”
- Eseguire le query per la generazione delle tabelle.
- Ad esempio:

```
CREATE TABLE Prodotti
(
  cp character varying(64) NOT NULL,
  nome character varying(64) NOT NULL,
  marca character varying(64) NOT NULL,
  modello character varying(64) NOT NULL,
  CONSTRAINT pk_prodotti PRIMARY KEY (cp)
)
```

Esercizio (Prodotto)

```
class Prodotto {

    private String cp;
    private String nome;
    private String marca;
    private String modello;

    public Prodotto() { }

    public Prodotto(String cp, String nome, String marca, String modello) {
        this.cp = cp;
        this.nome = nome;
        this.marca = marca;
        this.modello = modello;
    }

      // ... all get and set methods ....

   public String toString() {
       return this.cp + ": " + this.nome + ", " + this.marca + " - " + this.modello;
   }

}
```

Esercizio (Fornitore)

```
class Fornitore {
    private String cf;
    private String nome;
    private String indirizzo;
    private String citta;
    private final List<Prodotto> prodotti = new ArrayList<>();

    public Fornitore() { }

    public Fornitore(String cf, String nome, String indirizzo, String citta) {
        this.cf = cf;
        this.nome = nome;
        this.indirizzo = indirizzo;
        this.citta = citta;
    }

    // .... get and set ...

    public void printFornitoreById() {
        this.loadFornitoreById(this.cf);
        System.out.println(this);
    }
```

Esercizio (Fornitore)

```
private void loadFornitoreById(String id) {
    String query = "SELECT F.\"Nome\", F.\"Indirizzo\", F.\"Città\", P.\"CP\", P.\"Nome\", P.\"Marca\", P.\"Modello\"" +
            "from \"prodotti_fornitori\".\"Prodotti\" P " +
            "JOIN \"prodotti_fornitori\".\"Catalogo\" C " +
            "ON (P.\"CP\" = C.\"P\")" +
            "JOIN \"prodotti_fornitori\".\"Fornitori\" F " +
            "ON (F.\"CF\" = C.\"F\")" +
            "WHERE \"CF\" = ?";

    PreparedStatement ps = null;
    ResultSet rs = null;

    try {
        ps = DbHandler.getInstance().getPreparedStatement(query);
        ps.setString(1,id);
        rs = ps.executeQuery();

        if(rs.next()) {
            this.setCf(id);
            this.setNome(rs.getString(1));
            this.setIndirizzo(rs.getString(2));
            this.setCitta(rs.getString(3));

            Prodotto p;

            do {
                p = new Prodotto(rs.getString(4),rs.getString(5),rs.getString(6),rs.getString(7));
                this.prodotti.add(p);
            } while (rs.next());
        }
```

Esercizio (Fornitore) (2)

```
        } catch (SQLException e) {
            e.printStackTrace();
        } finally {
            try {
                if (rs!=null)
                    rs.close();
            } catch (SQLException e) {
                e.printStackTrace();
            }
            try {
                if(ps!=null)
                    ps.close();
            } catch (SQLException e) {
                e.printStackTrace();
            }
            try {
                DbHandler.getInstance().closeConnection();
            } catch (SQLException e) {
                e.printStackTrace();
            }
        }
    }

    public String toString() {
        StringBuffer sb = new StringBuffer();
        prodotti.forEach(x -> sb.append(x.toString()).append("\n"));
        return sb.toString();
    }
```