



A6

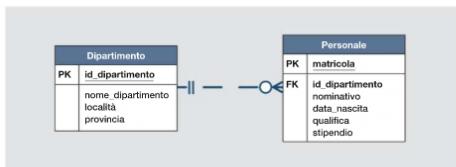
Accesso a una base di dati in linguaggio Java con JDBC

Com'è stato già anticipato nei capitoli precedenti, il linguaggio SQL può essere utilizzato come linguaggio «ospite» di un linguaggio di programmazione: in generale, questa modalità è relativa all'implementazione di applicazioni software, spesso di tipo gestionale, che prevedono un *front-end* per utenti che interagiscono mediante una GUI (*Graphical User Interface*) con i dati contenuti in un database. Il linguaggio di programmazione permette la realizzazione dell'interfaccia utente e l'implementazione delle procedure di elaborazione dei dati, mentre ai comandi SQL è demandata l'interazione con il database: ricerca, inserimento, aggiornamento e cancellazione dei dati.

ESEMPIO
Un programma di gestione aziendale integrata consente agli operatori di accedere ai dati relativi alla propria realtà aziendale memorizzati in un database gestito da un DBMS per compiere attività come la gestione della contabilità, della fatturazione, del magazzino e così via.

Nel caso in cui il linguaggio di programmazione ospitante sia Java, il modulo software che permette di accedere a un database è noto come JDBC (*Java DataBase Connectivity*): in questo capitolo vedremo come utilizzare la tecnologia JDBC per interagire con DBMS MySQL/MariaDB¹, due prodotti distribuiti gratuitamente rispettivamente da Oracle Corporation e MariaDB Foundation.

Gli esempi di questo capitolo² sono basati su un semplice database che rappresenta la realtà di un'azienda suddivisa in dipartimenti distribuiti geograficamente, ognuno dei quali ha un certo numero di impiegati. Di seguito se ne riporta il diagramma delle tabelle e il DB-schema in linguaggio SQL (comprendente delle istruzioni `INSERT` necessarie per popolare le tabelle del database con un insieme minimo di dati a cui verranno applicati gli esempi presentati in questo capitolo).



```

CREATE DATABASE Azienda;
USE Azienda;

CREATE TABLE Dipartimento (
    id_dipartimento VARCHAR(3) NOT NULL,
    nome_dipartimento VARCHAR(30),
    localita VARCHAR(20),
    provincia VARCHAR(2),
    CONSTRAINT chiave_primaria PRIMARY KEY (id_dipartimento)
);

INSERT INTO Dipartimento (id_dipartimento, nome_dipartimento,
                           localita, provincia) VALUES
('D1', 'ALFA', 'Livorno', 'LI'),
('D2', 'BETA', 'Pisa', 'PI'),
('D3', 'GAMMA', 'Piombino', 'LI'),
('D4', 'DELTA', 'Lucca', 'LU'),
('D5', 'OMEGA', 'Firenze', 'FI');

CREATE TABLE Personale (
    matricola VARCHAR(5) NOT NULL,
    id_dipartimento VARCHAR(3),
    nominativo VARCHAR(50),
    data_nascita DATE,
    qualifica VARCHAR(2),
    stipendio DOUBLE,
    CONSTRAINT chiave_primaria PRIMARY KEY (matricola),
    CONSTRAINT dipartimenti_personale
        FOREIGN KEY (id_dipartimento)
        REFERENCES Dipartimento(id_dipartimento)
);

INSERT INTO Personale (matricola, id_dipartimento, nominativo,
                      data_nascita, qualifica, stipendio) VALUES
('00001', 'D1', 'ROSSI PIERO', '1965-01-15', '01', 1640.00),
('00004', 'D2', 'NERI GIOVANNI', '1964-08-05', '02', 2530.00),
('00075', 'D4', 'ARNETTI MARIA', '1967-01-15', '01', 1750.00),
('04346', 'D3', 'BELLINI DANIELA', '1967-02-25', '01', 1740.00),
('04434', 'D2', 'VERDI MARCO', '1977-05-09', '03', 3480.00),
('04450', 'D3', 'SANDRI DONATA', '1969-05-01', '02', 2470.00),
('04532', 'D3', 'GIANNINI PIETRO', '1968-06-04', '01', 1580.00),
('04541', 'D3', 'TESINI MARIO', '1968-12-28', '02', 2740.00),
('04551', 'D1', 'BIANCHI MAURO', '1968-07-09', '02', 2580.00),
('04717', 'D2', 'PIERINI MARIO', '1969-06-20', '01', 1520.00),
('04794', 'D2', 'CARLETTI PAOLO', '1971-07-02', '01', 1670.00),
('05019', 'D4', 'SOLDANI GIULIO', '1973-03-27', '02', 2480.00),
('05462', 'D3', 'LAPINI PAOLO', '1967-01-11', '03', 3960.00),
('05477', 'D4', 'BRESCHI CARLA', '1976-05-02', '02', 2600.00);

```

Le varie soluzioni proposte nel seguito del capitolo prevedono una stretta integrazione del linguaggio SQL come ospite del linguaggio Java: l'interazione con il DBMS avviene sempre mediante stringhe che rappresentano comandi SQL che specifici metodi di classi Java inviano al DBMS stesso.

1 Architettura client/server e API

Java DataBase Connectivity

L'architettura di riferimento in cui si opera con JDBC è il classico modello client/server (FIGURA 1), in cui:

- l'applicazione client e il server DBMS sono normalmente eseguiti su computer distinti che comunicano tramite una rete locale o geografica;
- un singolo server può servire numerosi client;
- un singolo client può utilizzare più server.

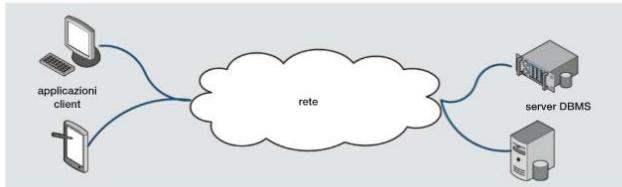


FIGURA 1

Un'analisi più approfondita evidenzia i seguenti aspetti:

- in un'architettura client/server coesistono applicazioni client e server DBMS eterogenei e, di conseguenza, si individuano almeno due componenti distinti: il *front-end* (costituito da GUI o da interfacce web) e il *back-end* (i server DBMS);
- è necessario stabilire come le applicazioni client comunicano con i server DBMS, in particolare come gestire i risultati delle *query*;
- è importante rendere l'accesso ai dati indipendente dallo specifico server DBMS utilizzato in modo che l'eventuale cambiamento non imponga modifiche alle applicazioni client.

Effettuare un accesso al database indipendente dallo specifico server DBMS utilizzato rappresenta il problema principale nell'implementazione di applicazioni distribuite in rete, la cui soluzione richiede funzionalità di adattamento e di conversione che permettano lo scambio di informazione tra sistemi e applicazioni eterogenei.

Per far fronte a queste problematiche nello sviluppo di applicazioni software si ricorre a tecnologie standard implementate in forma di API (*Application Program Interface*):

- **ODBC (Open DataBase Connectivity)**: sviluppata da Microsoft negli anni Novanta del secolo scorso e proposta con l'intento di definire un'interfaccia standard per l'accesso da codice in linguaggio C/C++ a database in modo indipendente sia dal sistema operativo sia dal DBMS;

- **JDBC (Java DataBase Connectivity)**: specificatamente sviluppata come componente standard del *Java Development Kit* a partire dal 1997, permette l'accesso a un database in modo indipendente dal DBMS e dalla piattaforma di esecuzione.

Un'applicazione software che opera su un database è un programma che invoca specifiche funzioni API per accedere ai dati gestiti da un DBMS. La tipica sequenza di interazione è la seguente:

- connessione alla sorgente dei dati (database contenuto in un DBMS);
- esecuzione di comandi/*query* SQL;
- recupero dei risultati ed eventuale gestione degli errori;
- disconnessione dalla sorgente dei dati.

Nel contesto descritto l'architettura software di riferimento è quella illustrata in FIGURA 2, in cui il *driver manager* espone all'applicazione software un'API standard, anche se i *driver* specifici dei vari DBMS dispongono di un'API proprietaria.

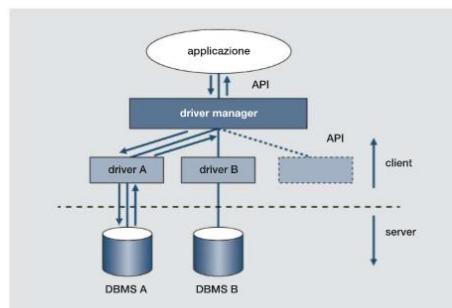


FIGURA 2

Driver manager. È un componente software che gestisce la comunicazione tra l'applicazione e i *driver* specifici dei singoli DBMS; risolve problematiche comuni a tutte le applicazioni:

- selezionare il *driver* da utilizzare sulla base delle informazioni fornite dall'applicazione;
- gestire l'invocazione delle funzioni dei *driver*.

Driver. Sono generalmente librerie caricate dinamicamente che implementano le funzioni API; ne esiste una specifica per ogni particolare DBMS che ha il compito di:

- nascondere le differenze di interazione dovute ai vari DBMS, sistemi operativi e protocolli di rete impiegati;
- trasformare le invocazioni delle funzioni API nel «dialetto» SQL usato dal DBMS, o nelle corrispondenti funzioni API supportate dal DBMS;
- gestire le transazioni, eseguire i comandi e le *query* SQL, inviare e recuperare i dati, gestire gli errori, ecc.

DBMS. È il server che gestisce la base di dati; in questo contesto:

- riceve le richieste esclusivamente nello specifico linguaggio supportato;
- esegue i comandi e le *query* SQL ricevute dal *driver* e restituisce i relativi risultati.

Il primo componente software conforme a questa architettura è stato ODBC supportato praticamente da tutti i DBMS relazionali. ODBC presenta tutta una serie di problemi legati principalmente a una limitata portabilità e inoltre:

- un linguaggio SQL con un insieme minimale di funzionalità;
- un'interfaccia non sempre di agevole utilizzazione.

Per ovviare a questi inconvenienti nel 1996 Sun Microsystems³, allo scopo di realizzare un'API standard per l'accesso ai database in linguaggio Java, ha sviluppato la tecnologia JDBC. JDBC è una soluzione «*pure Java*» inclusa nello JDK (Java Development Kit), il sistema di sviluppo del linguaggio Java che espone un'interfaccia flessibile per la gestione dell'interazione con i DBMS garantendo al tempo stesso l'indipendenza dalla piattaforma su cui viene eseguita.

In FIGURA 3 è schematizzata l'architettura generale di JDBC. JDBC prevede quattro tipi di *driver*.

- Tipo 1 (JDBC-ODBC bridge).** Fino alla versione JDBC4.1 era possibile accedere al DBMS da parte di JDBC tramite un *driver* ODBC (uno specifico per ogni DBMS relazionale). Dato che il *driver* ODBC non è in genere portabile tra diverse piattaforme di esecuzione, questa soluzione veniva adottata solo quando non vi erano alternative perché non adeguate al principio «*write once, run anywhere*» che caratterizza le applicazioni Java. Nelle versioni più recenti non è più supportato.
- Tipo 2 (driver realizzato solo in parte in Java).** Le chiamate inoltrate a JDBC sono convertite in chiamate all'API del DBMS, di conseguenza il *driver* contiene codice Java che invoca funzioni normalmente codificate in linguaggio C/C++; anche in questo caso il *driver* non è portabile tra piattaforme di esecuzione diverse.
- Tipo 3 (driver completamente realizzato in Java e DBMS middleware).** Le chiamate inoltrate a JDBC sono elaborate da un *driver* locale completamente codificato in Java che le trasforma utilizzando un protocollo indipendente da quello del server DBMS in invocazioni per un'applicazione *middleware* che, a sua volta, si interfaccia con lo specifico DBMS.

3. Sun Microsystems è stata acquistata nel 2009 da Oracle Corporation che ha mantenuto il supporto all'evoluzione del linguaggio Java e alle tecnologie correlate.

Middleware
Il termine *middleware* viene utilizzato per riferirsi a un insieme di componenti software intermediari tra applicazioni diverse. I componenti *middleware* sono spesso utilizzati come supporto per sistemi distribuiti complessi.

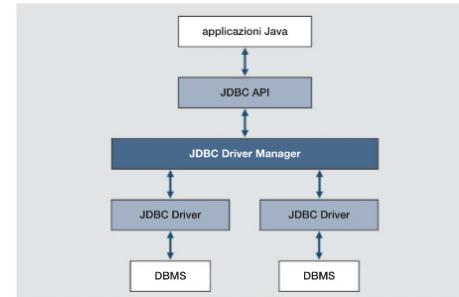


FIGURA 3

- Tipo 4 (driver pure Java con connessione diretta al server DBMS).** Il *driver* JDBC è in questo caso completamente realizzato in Java e – essendo specifico per il DBMS utilizzato – converte le chiamate JDBC direttamente nel protocollo di rete usato dal server DBMS.

Lo schema di FIGURA 4 sintetizza graficamente le differenze tra i quattro tipi di *driver* JDBC.

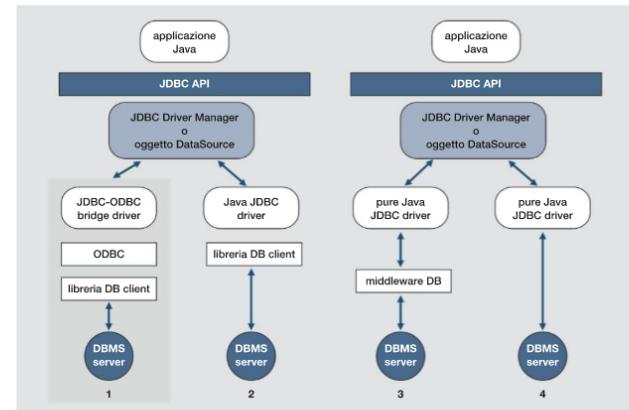


FIGURA 4

2 Connessione a un DBMS ed elaborazione di comandi e query SQL in linguaggio Java

Nel seguito ci riferiremo a una versione semplificata dell'architettura interna del *package java.sql* che implementa l'API JDBC, schematizzata in FIGURA 5.

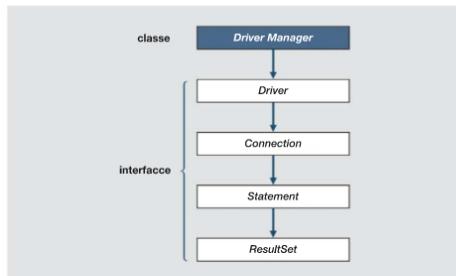


FIGURA 5

Prendiamo in esame i singoli passi previsti dall'API JDBC per l'interazione con un database contenuto in uno specifico DBMS.

1. Caricamento del driver

Il *driver manager* mantiene una lista di classi che implementano l'interfaccia *Driver*, ma la specifica implementazione di un *driver* deve essere registrata nel *driver manager*. Nelle prime versioni di JDBC la classe che realizza il *driver* implementando l'interfaccia *Driver* doveva essere esplicitamente caricata in modo da farne la registrazione mediante invocazione del metodo statico *registerDriver* della classe *DriverManager* da parte del codice di initializzazione della classe stessa. In alternativa era ed è possibile registrare direttamente un'istanza del *driver*.

ESEMPIO La seguente istruzione registra il *driver* di MySQL/MariaDB caricando la rispettiva classe
`Class.forName("com.mysql.jdbc.Driver");`
mentre la seguente istruzione registra esplicitamente il *driver bridge* per ODBC:
`DriverManager.registerDriver(new sun.jdbc.odbc.JdbcOdbcDriver());`

Dalla versione 4.0 di JDBC la registrazione dei *driver* avviene automaticamente in fase di connessione con il server DBMS e non è necessario effettuarla esplicitamente.

2. Connessione al server DBMS

Il metodo statico *getConnection* della classe *DriverManager* restituisce un oggetto che implementa l'interfaccia *Connection* e che permette di interagire con il DBMS; oltre allo *username* e alla *password*, il metodo *getConnection* richiede come parametro un URL il cui formato dipende dallo specifico DBMS utilizzato.

ESEMPIO L'accesso a un server MySQL in esecuzione sul computer stesso in cui viene eseguita l'applicazione Java richiede di specificare il seguente URL

`jdbc:mysql://localhost:3306/database`

dove *database* è il nome del database sul quale si intende operare. Nell'ipotesi di accedere al server DBMS con *username root* privo di *password*, la richiesta di connessione al database *Azienda* si effettua con la seguente istruzione:

`Connection con = DriverManager.getConnection(
 "jdbc:mysql://localhost:3306/Azienda",
 "root", "");`

In questi esempi e quelli che seguono si è fatto riferimento al DBMS MySQL, mentre per MariaDB è sufficiente sostituire il termine *mysql* con *mariadb*. In ogni caso la notazione con *mysql* e i relativi *driver* funzionano correttamente anche per MariaDB.

4. Questa configurazione è assolutamente sconsigliata sotto il profilo della sicurezza, ma è accettabile per un server DBMS utilizzato esclusivamente per sviluppare e testare il codice Java che accede a un database.

Nell'URL dell'esempio precedente *localhost* è l'identificatore di rete del server DBMS, mentre 3306 è il numero di porta standard su cui sono attivi MySQL/MariaDB; questa configurazione è tipica per un server DBMS installato sulla stessa macchina che ospita il sistema di sviluppo allo scopo di verificare il codice prodotto.

La connessione al server DBMS avviene invocando il metodo *getConnection* della classe *DriverManager*, che restituisce un oggetto che implementa l'interfaccia *Connection*; se uno dei *driver* caricati riconosce l'URL fornito dal metodo, il *driver* stabilisce la connessione, in caso contrario viene sollevata un'eccezione di tipo *SQLException*.

ESEMPIO Il codice del metodo *main* della seguente classe dimostra la connessione al database *Azienda* dell'esempio precedente gestito da un server DBMS MySQL in esecuzione sullo stesso sistema che esegue l'applicazione:

```

import java.sql.*;
public class TestConnection {
    public static void main(String[] args)
        // throws ClassNotFoundException
    {
        Connection con;
        String URL = "jdbc:mysql://localhost:3306";
        String database = "Azienda";
        // String driver = "com.mysql.jdbc.Driver";
        String user = "root";
        String password = "";
        try {
            // Class.forName(driver);
        }
    }
}
  
```



```

        con = DriverManager.getConnection(URL+"/"+database, user, password);
        System.out.println("Connessione al server DBMS effettuata.");
        con.close(); // disconnessione dal server DBMS
    }
    catch (SQLException exception) {
        System.out.println("Errore di connessione al server DBMS!");
    }
}

```

Le righe di codice commentate nell'esempio precedente si riferiscono all'eventuale uso di un *driver JDBC* di tipo obsoleto, per il quale è necessaria la registrazione esplicita.

Nelle versioni più recenti di JDBC è stato reso disponibile un tipo di costrutto, denominato *try-with-resources*, per chiudere automaticamente risorse del tipo *Connection*, *ResultSet* e *Statement*, a prescindere dal fatto che venga sollevata un'eccezione di tipo *SQLException* o di ogni altro tipo. Un'istruzione *try-with-resources* consiste in un'istruzione **try** con la dichiarazione di una o più risorse separate da «».

Relativamente all'esempio precedente le istruzioni per la connessione potrebbero essere riscritte come segue:

```

try (con = DriverManager.getConnection(URL+"/"+database, user,
    password);
    System.out.println("Connessione al server DBMS effettuata.")
);
catch (SQLException exception) {
    System.out.println("Errore di connessione al server DBMS!");
}

```

Come accennato, all'interno delle parentesi del **try** è possibile aggiungere più risorse:

```

Statement stm = con.createStatement();
ResultSet rs = stm.executeQuery("SELECT ...");

```

Nel caso in cui l'esecuzione di qualche istruzione che le genera avesse esito negativo tutte le risorse chiamate in causa sarebbero automaticamente rilasciate.

3. Esecuzione di comandi/query SQL

Per creare uno *statement* SQL – destinato a eseguire un comando o una *query* – si invoca il metodo *createStatement* di un oggetto che implementa l'interfaccia *Connection*; il metodo restituisce un oggetto che implementa l'interfaccia *Statement*.

Per eseguire un comando o una *query* è necessario distinguere tra interrogazioni dei dati e comandi DML/DDL; i metodi resi disponibili dall'interfaccia *Statement* a questo scopo sono riportati nella **TABELLA 1**.

TABELLA 1

execute	Esegue il comando o la <i>query</i> SQL fornita come parametro; restituisce true se il risultato è un oggetto che implementa l'interfaccia <i>ResultSet</i> , false altrimenti (l'eventuale risultato viene reso disponibile dai metodi <i>getResultSet</i> o <i>getUpdateCount</i>).
executeQuery	Esegue la <i>query</i> SQL fornita come parametro; restituisce un oggetto che implementa l'interfaccia <i>ResultSet</i> .
executeUpdate	Esegue il comando SQL fornito come parametro; restituisce un valore intero che rappresenta il numero di record coinvolti nell'esecuzione del comando; può essere utilizzato per l'esecuzione di comandi DDL.

Il seguente frammento di codice Java esegue una *query*

```

Connection con;
ResultSet result;
...
Statement stat = con.createStatement();
result = stat.executeQuery("SELECT * FROM Personale");
mentre quello che segue esegue un comando DML:
Connection con;
int result;
...
Statement stat = con.createStatement();
result = stat.executeUpdate("INSERT dipartimento(id_dipartimento, nome_dipartimento,
localita, provincia) VALUES ('D6', 'TEIA', 'Rosignano', 'LI');");

```

Il simbolo terminatore dei comandi SQL «», se non specificato, viene inserito automaticamente dal *driver* prima di inviare il comando al DBMS per l'esecuzione.

Il metodo *prepareStatement* di un oggetto che implementa l'interfaccia *Connection* consente di creare *statement* predefiniti eseguibili più volte (*prepared statement*); questa tecnica riduce i tempi di esecuzione del comando/*query* per comandi/*query* che devono essere eseguiti molte volte.

Utilizzando *statement* di questo tipo è possibile inserire nella stringa che rappresenta il comando SQL uno o più parametri identificati dal simbolo «». Alcuni metodi previsti dall'interfaccia *PreparedStatement* consentono di valorizzare i parametri – identificati dalla loro posizione nella stringa – prima dell'esecuzione; l'elenco che segue riporta solo quelli relativi ai tipi Java più comunemente utilizzati:

- *setTime*;
- *setString*;
- *setShort*;
- *setNull*;
- *setLong*;
- *setInt*;
- *setFloat*;
- *setDouble*;
- *setDate*.

ESEMPIO Il seguente frammento di codice Java crea uno *statement* predefinito con un parametro e lo istanzia con una stringa prima di eseguirlo:

```
Connection con;
ResultSet result;
...
PreparedStatement query = con.prepareStatement("SELECT * FROM Personale WHERE Nominativo = ?");
query.setString(1, "BIANCHI MAURO");
result = query.executeQuery();
```

4. Recupero dei risultati

Il metodo *executeQuery*, definito dalle interfacce *Statement* e *PreparedStatement*, restituisce un oggetto che implementa l'interfaccia *ResultSet*⁵. Un oggetto di questo metodo può contenere un risultato composto da una sequenza di record che sono iterabili mediante il metodo *next*: l'accesso ai valori dei campi del singolo record avviene mediante metodi specifici che accettano come parametro l'indice positionale del campo o il suo nome; l'elenco che segue riporta solo quelli relativi ai tipi Java più comunemente utilizzati:

- *getTime*;
- *getString*;
- *getShort*;
- *getLong*;
- *getInt*;
- *.getFloat*;
- *getDouble*;
- *getDate*.

ESEMPIO Il seguente frammento di codice Java esegue una *query* e visualizza il valore di alcuni campi del risultato:

```
Connection con;
...
Statement stat = con.createStatement();
ResultSet result = stat.executeQuery("SELECT * FROM Personale");
while (result.next()) {
    String nominativo = result.getString("nominativo");
    float stipendio = result.getFloat("stipendio");
    System.out.println(nominativo + ": " + stipendio);
}
result.close();
stat.close();
```

La prima invocazione del metodo *next* di un oggetto che implementa l'interfaccia *ResultSet* posiziona il «cursor» dei record sulla prima posizione della sequenza dei risultati; ogni successiva invocazione lo posiziona sulla posizione successiva: una volta raggiunta l'ultima posizione il metodo restituisce il valore **false**.

L'invocazione dei metodi *close* sugli oggetti che implementano l'interfaccia *ResultSet* e *Statement* forza il rilascio immediato delle risorse che utilizzano.

ESEMPIO Il seguente frammento di codice Java è analogo a quello dell'esempio precedente, ma accede ai valori dei singoli campi specificando la loro posizione nel risultato dell'interrogazione:

```
Connection con;
...
Statement stat = con.createStatement();
ResultSet result = stat.executeQuery("SELECT nominativo, qualifica,
                                         stipendio FROM Personale");
while (result.next()) {
    String nominativo = result.getString(1);
    float stipendio = result.getFloat(3);
    System.out.println(nominativo + ": " + stipendio);
}
result.close();
stat.close();
```

5. Disconnessione dal server DBMS

La disconnessione dal server DBMS si effettua invocando il metodo *close* dell'oggetto restituito dall'invocazione del metodo *getConnection*.

Gli esempi che seguono illustrano l'intera sequenza di interazione di un'applicazione JDBC con un DBMS.

ESEMPIO Il codice del metodo *main* della seguente classe visualizza l'elenco in ordine alfabetico del personale del database *Azienda* riportando il dipartimento a cui ogni unità di personale è assegnata (la connessione avviene a un server DBMS MySQL in esecuzione sullo stesso sistema che esegue l'applicazione):

```
import java.sql.*;
public class TestQuery {
    public static void main(String[] args) {
        Connection con;
        Statement stat;
        ResultSet result;
        String URL = "jdbc:mysql://localhost:3306";
        String database = "Azienda";
        String user = "root";
        String password = "";

        try {
            con = DriverManager.getConnection(URL+"/"+database, user,
                                            password);
            System.out.println("Effettuata connessione al server DBMS.");
            stat = con.createStatement();
            result = stat.executeQuery("SELECT * FROM Personale,
                                         Dipartimento WHERE
                                         Personale.id_dipartimento =
                                         Dipartimento.id_dipartimento
                                         ORDER BY nominativo");
            System.out.println("Elenco personale:");
            while (result.next()) {
                String nominativo = result.getString("nominativo");
                String dipartimento = result.getString("nome_dipartimento");
                System.out.println(nominativo + " - " + dipartimento);
            }
            result.close();
            stat.close();
        }
```



```

        con.close();
        System.out.println("Effettuata disconnessione dal server DBMS.");
    }
    catch (SQLException exception) {
        System.out.println("Errore!");
    }
}

Con i dati del database di esempio viene prodotto il seguente output:
Effettuata connessione al server DBMS.
Elenco personale:
ARNETTI MARIA - DELTA
BELLI DANIELA - GAMMA
BIANCHI MAURO - ALFA
BRESCHI CARLA - DELTA
CARLETTI PAOLO - BETA
GIANNINI PIETRO - GAMMA
LAPINI PAOLO - GAMMA
NERI GIOVANNI - BETA
PIERINI MARIO - BETA
ROSSI PIERO - ALFA
SANDRI DONATA - GAMMA
SOLDANI GIULIO - DELTA
TESINI MARIO - GAMMA
VERDI MARCO - BETA
Effettuata disconnessione dal server DBMS.

```

ESEMPIO Il codice del metodo *main* della seguente classe visualizza il numero delle unità di personale del database *Azienda* prima e dopo l'esecuzione di un comando di eliminazione di alcuni record della tabella *Personale* (la connessione avviene a un server DBMS MySQL in esecuzione sullo stesso sistema che esegue l'applicazione).

```

import java.sql.*;
public class TestDML {
    public static void main(String[] args) {
        Connection con;
        Statement stat;
        ResultSet result;
        int personale;
        String URL = "jdbc:mysql://localhost:3306";
        String database = "Azienda";
        String user = "root";
        String password = "";
        try {
            con = DriverManager.getConnection(URL+"/"+database, user, password);
            System.out.println("Effettuata connessione al server DBMS.");
            stat = con.createStatement();
            result = stat.executeQuery("SELECT COUNT(*) AS numero FROM Personale");
            result.next(); // posizionamento primo risultato
            personale = result.getInt(1);
            System.out.println("Unità personale: " + personale);
        }

```

```

        if (stat.executeUpdate("DELETE FROM Personale
                               WHERE stipendio > 3000;") > 0) {
            System.out.println("Eliminato personale con stipendio maggiore di 3000€.");
        }
        result = stat.executeQuery("SELECT COUNT(*) AS numero FROM Personale");
        result.next(); // posizionamento primo risultato
        personale = result.getInt(1);
        System.out.println("Unità personale: " + personale);
        result.close();
        stat.close();
        con.close();
        System.out.println("Effettuata disconnessione dal server DBMS.");
    }
    catch (SQLException exception) {
        System.out.println("Errore!");
    }
}
}

Con i dati del database di esempio viene prodotto il seguente output:
Effettuata connessione al server DBMS.
Unità personale: 14
Eliminato personale con stipendio maggiore di 3000€.
Unità personale: 12
Effettuata disconnessione dal server DBMS.

```

3 Classi CRUD in linguaggio Java: corrispondenza tra tipi SQL e tipi Java

Le operazioni che normalmente si effettuano su una base di dati sono riportate dall'acronimo CRUD:

- **Create:** creazione di nuovi record;
- **Read:** ricerca e lettura di record;
- **Update:** aggiornamento di record esistenti;
- **Delete:** eliminazione di record.

Ogni operazione CRUD è chiaramente associata a uno specifico comando del linguaggio SQL come riportato nella [TABELLA 2](#).

TABELLA 2

Create	INSERT
Read	SELECT
Update	UPDATE
Delete	DELETE

La realizzazione di una classe CRUD in linguaggio Java deve prevedere – oltre alla definizione di specifiche classi per la rappresentazione dei

Java Data Objects

Java Data Objects (JDO) è un API per la gestione della persistenza degli oggetti in un database relazionale. JDO consente di memorizzare in modo permanente e trasparente, per ripristinare successivamente, oggetti istanze di classi definite in linguaggio Java. JDO realizza un vero e proprio Object-Relational Mapping (ORM) tra gli oggetti Java tradizionali (POJO, Plain Old Java Object) e i database relazionali.





```

String query = "SELECT id_dipartimento FROM Dipartimento
               WHERE nome_dipartimento = '" +
               + personale.getDipartimento() + "'";
result = stat.executeQuery(query);
result.next();
id_dipartimento = result.getString(1);
result.close();
stat.close();
}
catch (SQLException exception) {
    return false;
}

try {
    stat = con.createStatement();
    data_nascita = personale.getDataNascita().toString();
    String command = "INSERT INTO Personale(matricola,
                                              id_dipartimento, nominativo, data_nascita,
                                              qualifica, stipendio)
                      VALUES ('" + personale.getMatricola() + "', '" +
                      id_dipartimento + "', '" +
                      personale.getNominativo() + "', '" +
                      data_nascita + "', '" +
                      personale.getQualifica() + "', '" +
                      personale.getStipendio() + "');";
    if (stat.executeUpdate(command) == 0) {
        return false;
    }
    stat.close();
}
catch (SQLException exception) {
    return false;
}
return true;
}

/* Aggiornamento dati unità di personale nel database */
public boolean aggiornaPersonale(Personale personale) {
    Statement stat;
    ResultSet result;
    String id_dipartimento;
    String data_nascita;

    try { // verifica presenza nel database unità di personale
        stat = con.createStatement();
        String query = "SELECT COUNT(*) AS numero FROM Personale
                       WHERE matricola = '" + personale.getMatricola() + "'";
        result = stat.executeQuery(query);
        result.next();
        if (result.getInt(1) == 0) {
            return false;
        }
        result.close();
        stat.close();
    }
    catch (SQLException exception) {

```

```

        return false;
    }

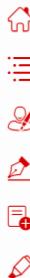
    try { // ricerca identificatore dipartimento a partire dal nome
        stat = con.createStatement();
        String query = "SELECT id_dipartimento FROM Dipartimento
                       WHERE nome_dipartimento = '" +
                       + personale.getDipartimento() + "'";
        result = stat.executeQuery(query);
        result.next();
        id_dipartimento = result.getString(1);
        result.close();
        stat.close();
    }
    catch (SQLException exception) {
        return false;
    }

    try {
        stat = con.createStatement();
        data_nascita = personale.getDataNascita().toString();
        String command = "UPDATE Personale SET id_dipartimento='" +
                         id_dipartimento + ", nominativo='" +
                         personale.getNominativo() +
                         "', data_nascita= '" + data_nascita +
                         "', qualifica=' + personale.getQualifica() +
                         "', stipendio=' + personale.getStipendio() +
                         ' WHERE matricola=' + personale.getMatricola() +
                         "'";;
        if (stat.executeUpdate(command) == 0) {
            return false;
        }
        stat.close();
    }
    catch (SQLException exception) {
        return false;
    }
    return true;
}

/* Ricerca nel database i dati di una unità di personale */
public Personale datiPersonale(String matricola) {
    Personale personale;
    Statement stat;
    ResultSet result;
    String nominativo;
    Localdate data_nascita;
    String nome_dipartimento;
    String qualifica;
    double stipendio;

    try {
        stat = con.createStatement();
        String query = "SELECT * FROM Personale, Dipartimento
                       WHERE Personale.id_dipartimento =
                           Dipartimento.id_dipartimento AND matricola = '" +
                           matricola + "'";

```



```

result = stat.executeQuery(query);
result.next();
nominativo = result.getString("nominativo");
data_nascita = result.getDate("data_nascita").toLocalDate();
nome_dipartimento = result.getString("nome_dipartimento");
qualifica = result.getString("qualifica");
stipendio = result.getDouble("stipendio");
personale = new Personale(matricola, nome_dipartimento,
                           nominativo, qualifica, data_nascita,
                           stipendio);
result.close();
stat.close();
return personale;
}
catch (SQLException exception) {
    return null;
}
}

/* Ricerca nel database i dati di tutte le unità di personale */
public Personale[] elencoPersonale() {
    Personale[] elenco_personale = {};
    Statement stat;
    ResultSet result;
    int numero_personale;
    String matricola;
    String nominativo;
    LocalDate data_nascita;
    String nome_dipartimento;
    String qualifica;
    double stipendio;

    try { // conteggio unità di personale presenti nel database
        stat = con.createStatement();
        String query = "SELECT COUNT(*) AS numero FROM Personale";
        result = stat.executeQuery(query);
        result.next();
        numero_personale = result.getInt(1);
        if (numero_personale > 0) {
            elenco_personale = new Personale[numero_personale];
        }
        else {
            return null;
        }
        result.close();
        stat.close();
    }
    catch (SQLException exception) {
        return null;
    }
    try {
        stat = con.createStatement();
        String query = "SELECT * FROM Personale, Dipartimento
                       WHERE Personale.id_dipartimento =
                             Dipartimento.id_dipartimento;";
        result = stat.executeQuery(query);
    }

```



```

        numero_personale = 0;
        while (result.next()) {
            matricola = result.getString("matricola");
            nominativo = result.getString("nominativo");
            data_nascita = result.getDate("data_nascita").toLocalDate();
            nome_dipartimento = result.getString("nome_dipartimento");
            qualifica = result.getString("qualifica");
            stipendio = result.getDouble("stipendio");
            elenco_personale[numero_personale] =
                new Personale(matricola, nome_dipartimento, nominativo,
                               qualifica, data_nascita, stipendio);
            numero_personale++;
        }
        result.close();
        stat.close();
        return elenco_personale;
    }
    catch (SQLException exception) {
        return null;
    }
}

/* Eliminazione unità di personale dal database */
public boolean eliminaPersonale(String matricola) {
    Statement stat;
    ResultSet result;
    try {
        stat = con.createStatement();
        String command = "DELETE FROM Personale
                         WHERE matricola = '" + matricola + "'";
        if (stat.executeUpdate(command) == 0) {
            return false;
        }
        stat.close();
    }
    catch (SQLException exception) {
        return false;
    }
    return true;
}
}

```



In un contesto reale nel codice del metodo *elencoPersonale* della classe dell'esempio precedente potrebbe essere opportuno introdurre un limite alla dimensione massima che l'*array* restituito può assumere.

Per verificare i metodi della classe CRUD dell'esempio precedente è possibile definire il seguente metodo *main*:

```

public static void main(String args[]) {
    LocalDate data_nascita = LocalDate.of(1965, 2, 23);
    Personale personale = new Personale("01234", "ALFA", "MEINI GIORGIO",
                                         "09", data_nascita, 9999.99);
    GestionePersonale gestione_personale;
    Personale[] elenco_personale = {};

```



```

try {
    gestione_personale = new GestionePersonale();
}

catch (SQLException exception) {
    System.out.println("Errore connessione server DBMS.");
    return;
}

if (gestione_personale.aggiungiPersonale(personale)) {
    System.out.println("Unità di personale aggiunta al database.");
}
else {
    System.out.println("Errore aggiunta unità di personale.");
}

elenco_personale = gestione_personale.elencoPersonale();
if (elenco_personale != null) {
    System.out.println("Elenco personale: ");
    for (int i=0; i<elenco_personale.length; i++) {
        System.out.println(elenco_personale[i]);
    }
}
else {
    System.out.println("Errore ricerca unità di personale.");
}

personale.setQualifica("00");
personale.setStipendio(999.99);
if (gestione_personale.aggiornaPersonale(personale)) {
    System.out.println("Unità di personale del database aggiornata.");
}
else {
    System.out.println("Errore aggiornamento unità di personale.");
}

personale = gestione_personale.datiPersonale("01234");
if (personale != null) {
    System.out.println("Unità di personale: " + personale);
}
else {
    System.out.println("Errore ricerca unità di personale.");
}

if (gestione_personale.eliminaPersonale("01234")) {
    System.out.println("Unità di personale eliminata dal database.");
}
else {
    System.out.println("Errore eliminazione unità di personale.");
}
}

```



3.1 Corrispondenza tra tipi SQL e tipi Java

La scelta del metodo corretto da impiegare per acquisire i valori dei singoli campi da un oggetto che implementa l'interfaccia *ResultSet* richiede di stabilire una corrispondenza tra i tipi dei campi delle tabelle di un database e i tipi delle variabili utilizzate.

Nella **TABELLA 3** sono riportate le corrispondenze tra i tipi di dati Java e i più comuni tipi di dati del linguaggio SQL.

TABELLA 3

SQL	Java
CHAR	String
VARCHAR	String
NUMERIC	java.math.BigDecimal
DECIMAL	java.math.BigDecimal
BIT	boolean
INTEGER	int
REAL	float
FLOAT	double
DOUBLE	double
BINARY	byte[]
VARBINARY	byte[]
DATE	java.sql.Date
TIME	java.sql.Time
TIMESTAMP	java.sql.Timestamp

CHAR e VARCHAR

I tipi del linguaggio SQL **CHAR** e **VARCHAR** sono strettamente collegati tra loro: **CHAR** rappresenta una stringa di caratteri di lunghezza fissa, mentre **VARCHAR** rappresenta una stringa di caratteri a lunghezza variabile. Nel linguaggio Java non è possibile effettuare questa distinzione: entrambi i tipi possono essere rappresentati in Java come **String**, o come **char[]**, anche se la prima soluzione è in generale la più appropriata. Il metodo *getString* definito dall'interfaccia *ResultSet* restituisce un oggetto di tipo **String** ed è adatto per dati sia di tipo **CHAR** sia **VARCHAR**.

NUMERIC e DECIMAL

I tipi **NUMERIC** e **DECIMAL** del linguaggio SQL sono molto simili: entrambi rappresentano numeri in virgola fissa e la rappresentazione più conveniente in linguaggio Java per tali tipi è la classe **BigDecimal** del package *java.math*. Essa infatti permette operazioni matematiche sul tipo **BigDecimal** anche in combinazione con tipi interi e in virgola mobile. Oltre che con il metodo *getBigDecimal* dell'interfaccia *ResultSet* è possibile accedere a questi tipi di valori invocando il metodo *getString*.

**BIT**

Il tipo SQL **BIT** rappresenta un singolo bit che può assumere i valori 0 o 1; può essere rappresentato mediante il tipo **boolean** del linguaggio Java restituito dal metodo `getBoolean` dell'interfaccia `ResultSet`.

INTEGER

Il tipo SQL **INTEGER** e le varianti MySQL **TINYINT**, **SIMALLINT**, **INT** e **BIGINT** rappresentano valori numerici interi che trovano una corrispondenza nei tipi Java **byte**, **short**, **int** e **long** restituiti rispettivamente dai metodi `getByte`, `getShort`, `getInt` e `getLong` dell'interfaccia `ResultSet`.

REAL, FLOAT e DOUBLE

Il tipo SQL **REAL** rappresenta numeri in virgola mobile in singola precisione, mentre i tipi **FLOAT** e **DOUBLE** rappresentano numeri in virgola mobile in doppia precisione. È quindi raccomandabile utilizzare per il tipo **REAL** il tipo Java **float** restituito dal metodo `getFloat` dell'interfaccia `ResultSet`, e per i tipi **FLOAT** e **DOUBLE** il tipo Java **double** restituito dal metodo `getDouble` dell'interfaccia `ResultSet`.

BINARY e VARBINARY

Questi tipi di dati SQL sono strettamente correlati: **BINARY** rappresenta dati binari di lunghezza fissa, mentre **VARBINARY** rappresenta dati binari di lunghezza variabile. Nel linguaggio Java non si ha la necessità di distinguere tra questi tipi perché essi possono essere rappresentati tramite *array* di byte in ambedue i casi: il metodo `getBytes` dell'interfaccia `ResultSet` viene usato per recuperare valori di tipo **BINARY** e **VARBINARY**.

DATE, TIME e TIMESTAMP

Questi tre tipi di dati del linguaggio SQL fanno riferimento a valori temporali: il tipo **DATE** rappresenta date con i valori di giorno, mese e anno, il tipo **TIME** un orario con valori di ore, minuti e secondi e il tipo **TIMESTAMP** valori in cui sono compresi una data e un orario integrato con il valore di millisecondi.

In accordo con la nuova API Java `java.time` per la gestione di date e orari, nelle versioni più recenti di JDBC sono state rese disponibili le tre classi del package `java.time`: `LocalDate` (usata negli esempi di questo capitolo), `LocalTime` e `LocalDateTime`, con le seguenti corrispondenze SQL:

- `LocalDate` per dati SQL di tipo **DATE**;
- `LocalTime` per dati SQL di tipo **TIME**;
- `LocalDateTime` per dati SQL di tipo **TIMESTAMP**.

Queste nuove classi sostituiscono la classe standard, ormai deprecata, `Date` del package `java.util` finalizzata alla gestione di dati di tipo data e/o orario. Anche in questo caso, non essendoci una corrispondenza diretta con nessuno dei tre tipi SQL per le date e gli orari, nel package `java.sql` sono definite le tre sottoclassi di **DATE**: `Date`, `Time` e `Timestamp`.



4 Uso di oggetti RowSet

`RowSet` è un'interfaccia definita nel package `javax.sql` che deriva dall'interfaccia `ResultSet`, ma che espone la tipica interfaccia di un oggetto **JavaBeans**: un oggetto che implementa l'interfaccia `RowSet` ha lo scopo di mantenere una corrispondenza tra i dati che contiene nella forma di una sequenza di record e la loro sorgente, generalmente un database relazionale al quale si connette utilizzando JDBC.

Alcuni vantaggi del ricorso a `RowSet` rispetto a `ResultSet` consistono nel fatto che `RowSet` è sempre «navigabile» in entrambe le direzioni e che l'aggiornamento dei dati che un oggetto `RowSet` contiene comporta l'aggiornamento della sorgente dei dati (nel caso di un database relazionale di record di una o più tabelle).

L'impostazione di alcune proprietà previste dall'interfaccia `RowSet` condizionano il funzionamento degli oggetti che la implementano ([TABELLA 4](#)).

TABELLA 4

concurrency	L'accesso concorrente ai dati può essere di tipo <code>CONCUR_READ_ONLY</code> (default, sola lettura dei dati), oppure <code>CONCUR_UPDATABLE</code> (permesso aggiornamento concorrente dei dati).
maxRows	Massimo numero di record gestiti.
password	<code>Password</code> per l'accesso al database.
queryTimeout	Tempo massimo di attesa per l'esecuzione di un comando (secondi).
readOnly	Accesso ai dati senza possibilità di modifica.
type	Lo scorrimento dei dati può essere di tipo <code>TYPE_FORWARD_ONLY</code> (unidirezionale), <code>TYPE_SCROLL_INSENSITIVE</code> (bidirezionale non sensibile al cambiamento della sorgente di dati) o <code>TYPE_SCROLL_SENSITIVE</code> (bidirezionale, sensibile al cambiamento della sorgente di dati).
url	URL per la connessione JDBC al database.
username	<code>Username</code> per l'accesso al database.

OSSERVAZIONE Dato che gli oggetti `RowSet` sono componenti **JavaBeans**, le proprietà sono impostate e acquisite invocando metodi `set/get` specifici.

L'interfaccia `RowSet` prevede due metodi fondamentali per il funzionamento degli oggetti che la implementano:

- **setCommand**: per impostare la query SQL che definisce i dati che l'oggetto contiene;
- **execute**: per eseguire la query SQL che carica i dati come contenuto dell'oggetto.

I seguenti metodi che l'interfaccia `RowSet` eredita dall'interfaccia `ResultSet` consentono di impostare il valore dei campi del record corrente senza aggiornare i corrispondenti valori nel database, cosa che può essere fatta successivamente invocando il metodo `updateRow`:

- `updateBigDecimal`;
- `updateBoolean`;





- *updateByte;*
- *updateBytes;*
- *updateDate;*
- *updateDouble;*
- *updateFloat;*
- *updateInt;*
- *updateLong;*
- *updateNull;*
- *updateShort;*
- *updateString;*
- *updateTime;*
- *updateTimestamp.*

OSSERVAZIONE Per l'acquisizione del valore dei campi del record corrente sono disponibili i seguenti metodi ereditati dall'interfaccia *ResultSet*:

- *getBigDecimal;*
- *getBoolean;*
- *getByte;*
- *getBytes;*
- *getDate;*
- *getDouble;*
- *getFloat;*
- *getInt;*
- *getLong;*
- *getNull;*
- *getShort;*
- *getString;*
- *getTime;*
- *getTimestamp.*

L'interfaccia *RowSet* eredita dall'interfaccia *ResultSet* i seguenti metodi per lo scorrimento della sequenza di record che contiene:

- *afterLast*: posiziona il cursore oltre l'ultimo record della sequenza (come quando termina lo scorrimento della sequenza);
- *beforeFirst*: posiziona il cursore prima del primo record della sequenza (come quando lo scorrimento della sequenza non è ancora iniziato);
- *first*: posiziona il cursore sul primo record della sequenza;
- *last*: posiziona il cursore sull'ultimo record della sequenza;
- *next*: posiziona il cursore sul successivo record della sequenza;
- *previous*: posiziona il cursore sul precedente record della sequenza.



L'eliminazione di un record da un oggetto che implementa l'interfaccia *RowSet* e contemporaneamente dei dati corrispondenti nel database avviene invocando il metodo *deleteRow*.

L'inserimento di un nuovo record in un oggetto che implementa l'interfaccia *RowSet* e contemporaneamente nella/e tabella/e corrispondente/i nel database avviene invocando il metodo *insertRow* dopo il posizionamento sul record speciale d'inserimento mediante invocazione del metodo *moveToInsertRow*.

Le classi che implementano l'interfaccia *RowSet* normalmente ne implementano una delle seguenti specializzazioni:

- ***JdbcRowSet***: prevede la connessione permanente a un database mediante un *driver JDBC*;
- ***CachedRowSet***: prevede che la connessione alla propria sorgente di dati possa avvenire mediante sincronizzazioni discontinue nel tempo;
- ***WebRowSet***: prevede lo stesso comportamento di *CachedRowSet*, oltre a specifiche funzionalità per l'importazione e l'esportazione in formato XML dei dati che contiene;
- ***JoinRowSet***: prevede lo stesso comportamento di *WebRowSet*, oltre alla possibilità di effettuare *join* senza la necessità di connettersi alla sorgente dei dati;
- ***FilteredRowSet***: prevede lo stesso comportamento di *WebRowSet*, oltre alla possibilità di filtrare i dati che risultano visibili senza la necessità di connettersi alla sorgente dei dati.

Nel package *javax.sql* non sono definite classi che implementano l'interfaccia *RowSet*, ma è in ogni caso possibile utilizzare una delle implementazioni standard di riferimento contenute nel package *com.sun.rowset*, come *JdbcRowSetImpl* che implementa l'interfaccia *JdbcRowSet*. Il modo più semplice per creare un oggetto di classe *JdbcRowSetImpl* è quello di invocarne il costruttore fornendo come argomento un oggetto di tipo *Connection* già istanziato.



La seguente classe Java rivisita quella dell'esempio precedente relativo agli oggetti *ResultSet* allo scopo di dimostrare l'uso degli oggetti *RowSet*:

```
import java.sql.*;
import javax.sql.*;
import java.time.*;
import javax.sql.rowset.*;
import com.sun.rowset.*;
```

```
public class GestionePersonale {
    final String URL = "jdbc:mysql://localhost:3306";
    final String database = "Azienda";
    final String user = "root";
    final String password = "";
    private Connection con;
```



```

private JdbcRowSet rowset;
public GestionePersonale() throws SQLException {
    con = DriverManager.getConnection(URL+"/"+database, user, password);
    rowset = new JdbcRowSetImpl(con);
    rowset.setCommand("SELECT * FROM Personale");
    rowset.execute();
}
/* Inserimento unità di personale nel database */
public boolean aggiungiPersonale(Personale personale) {
    Statement stat;
    ResultSet result;
    String id_dipartimento;
    try { // ricerca identificatore dipartimento a partire dal nome
        stat = con.createStatement();
        String query = "SELECT id_dipartimento FROM Dipartimento
                       WHERE nome_dipartimento = '" +
                      + personale.getDipartimento() + "'";
        result = stat.executeQuery(query);
        result.next();
        id_dipartimento = result.getString(1);
        result.close();
        stat.close();
    } catch (SQLException exception) {
        return false;
    }
    try {
        rowset.moveToInsertRow();
        rowset.updateString("matricola", personale.getMatricola());
        rowset.updateString("id_dipartimento", id_dipartimento);
        rowset.updateString("nominativo", personale.getNominativo());
        rowset.updateDate("data_nascita", Date.valueOf(personale.getDataNascita()));
        rowset.updateString("qualifica", personale.getQualifica());
        rowset.updateDouble("stipendio", personale.getStipendio());
        rowset.insertRow();
        return true;
    } catch (SQLException exception) {
        return false;
    }
}
/* Aggiornamento dati unità di personale nel database */
public boolean aggiornaPersonale(Personale personale) {
    Statement stat;
    ResultSet result;
    String id_dipartimento;
    String data_nascita;
    try { // ricerca identificatore dipartimento a partire dal nome
        stat = con.createStatement();
        String query = "SELECT id_dipartimento FROM Dipartimento WHERE
                       nome_dipartimento = '" + personale.getDipartimento() + "'";
        result = stat.executeQuery(query);
        result.next();
    }

```

```

        id_dipartimento = result.getString(1);
        result.close();
        stat.close();
    } catch (SQLException exception) {
        return false;
    }
    try {
        rowset.beforeFirst();
        while (rowset.next()) { // ricerca record
            if (rowset.getString("matricola").equals(
                (personale.getMatricola()))) {
                rowset.updateString("id_dipartimento", id_dipartimento);
                rowset.updateString("nominativo", personale.getNominativo());
                rowset.updateDate("data_nascita", Date.valueOf(personale.getDataNascita()));
                rowset.updateString("qualifica", personale.getQualifica());
                rowset.updateDouble("stipendio", personale.getStipendio());
                rowset.updateRow();
                return true;
            }
        }
        return false;
    } catch (SQLException exception) {
        return false;
    }
}
/* Ricerca nel database i dati di una unità di personale */
public Personale datiPersonale(String matricola) {
    Personale personale;
    Statement stat;
    ResultSet result;
    String nominativo;
    LocalDate data_nascita;
    String id_dipartimento;
    String nome_dipartimento;
    String qualifica;
    double stipendio;
    try {
        rowset.beforeFirst();
        while (rowset.next()) { // ricerca record
            if (rowset.getString("matricola").equals(matricola)) {
                id_dipartimento = rowset.getString("id_dipartimento");
                nominativo = rowset.getString("nominativo");
                data_nascita = rowset.getDate("data_nascita").toLocalDate();
                qualifica = rowset.getString("qualifica");
                stipendio = rowset.getDouble("stipendio");
                stat = con.createStatement();
                String query = "SELECT nome_dipartimento FROM Dipartimento
                               WHERE id_dipartimento = '" + id_dipartimento + "'";
                result = stat.executeQuery(query);
                result.next();
                nome_dipartimento = result.getString(1);
                result.close();
            }
        }
    }

```



```

        stat.close();
        personale = new Personale(matricola, nome_dipartimento,
                                   nominativo, qualifica, data_nascita,
                                   stipendio);
        return personale;
    }
    return null;
}
catch (SQLException exception) {
    return null;
}
}

/* Ricerca nel database i dati di tutte le unità di personale */
public Personale[] elencoPersonale() {
    Personale elenco_personale[];
    Statement stat;
    ResultSet result;
    int numero_personale;
    String matricola;
    String nominativo;
    LocalDate data_nascita;
    String id_dipartimento;
    String nome_dipartimento;
    String qualifica;
    double stipendio;

    try { // conteggio record
        rowset.beforeFirst();
        numero_personale = 0;
        while (rowset.next()) {
            numero_personale++;
        }
        if (numero_personale > 0) {
            elenco_personale = new Personale[numero_personale];
        }
        else {
            return null;
        }
    }
    catch (SQLException exception) {
        return null;
    }

    try {
        rowset.beforeFirst();
        numero_personale = 0;
        while (rowset.next()) {
            matricola = rowset.getString("matricola");
            id_dipartimento = rowset.getString("id_dipartimento");
            nominativo = rowset.getString("nominativo");
            data_nascita = rowset.getDate("data_nascita").toLocalDate();
            qualifica = rowset.getString("qualifica");
            stipendio = rowset.getDouble("stipendio");
            stat = con.createStatement();
    }

```



```

String query = "SELECT nome_dipartimento FROM Dipartimento
                WHERE id_dipartimento = '" + id_dipartimento + "'";
result = stat.executeQuery(query);
result.next();
nome_dipartimento = result.getString(1);
result.close();
stat.close();
elenco_personale[numero_personale] = new Personale(matricola,
                                                       nome_dipartimento,
                                                       nominativo, qualifica,
                                                       data_nascita, stipendio);

numero_personale++;
}
return elenco_personale;
}
catch (SQLException exception) {
    return null;
}
}

/* Eliminazione unità di personale dal database */
public boolean eliminaPersonale(String matricola) {
    Statement stat;
    ResultSet result;

    try {
        rowset.beforeFirst();
        while (rowset.next()) { // ricerca record
            if (rowset.getString("matricola").equals(matricola)) {
                rowset.deleteRow();
                return true;
            }
        }
        return false;
    }
    catch (SQLException exception) {
        return false;
    }
}

```

Non essendo cambiata l'interfaccia della classe *GestionePersonale* il *main* per il test della classe non subisce variazioni rispetto al metodo *main* della classe originale.

Il codice dell'esempio precedente – oltre all'oggetto *JdbcRowSet* che comprende i dati della tabella *Personale* del database *Azienda* – utilizza oggetti *ResultSet* temporanei per risolvere l'accesso alla tabella *Dipartimento*.

La mancanza di metodi che permettono di selezionare i singoli record di un oggetto *JdbcRowSet* costringe a effettuare la ricerca dei record di interesse iterando i record contenuti nell'oggetto.

Relativamente ai *RowSet*, nelle recenti versioni di JDBC sono state introdotte l'interfaccia *RowSetFactory* e la classe *RowSetProvider*, che permet-

Eventi e GUI

Gli eventi generati da un oggetto di classe *RowSet*, intercettati da uno specifico *RowSetListener* registrato da un componente della GUI di un'applicazione Java (per esempio un componente del framework *Swing*), permettono di aggiornare la visualizzazione dei dati in modo sincrono con il loro cambiamento anche nel caso che la variazione sia determinata da operazioni estranee al componente stesso.



tono la creazione di tutti i tipi di *RowSet* supportati dagli specifici *driver JDBC*.

ESEMPIO
Creazione di *RowSet* con l'interfaccia *RowSetFactory* e la classe *RowSetProvider*.
Viene usata un'istanza di *RowSetFactory* per creare l'oggetto *JdbcRowSet* *jdbcRs*:

```
import java.sql.*;
import javax.sql.rowset.*;
...
public void testJdbcRowSet(String username, String password)
throws SQLException {
RowSetFactory myRowSetFactory = null;
JdbcRowSet jdbcRs = null;
ResultSet rs = null;

try {
myRowSetFactory = RowSetProvider.newFactory();
jdbcRs = myRowSetFactory.createJdbcRowSet();
jdbcRs.setUrl("jdbc:mysql://localhost:3306/Azienda");
jdbcRs.setUsername(username);
jdbcRs.setPassword(password);

jdbcRs.setCommand("SELECT * FROM Personale");
jdbcRs.execute();
while (jdbcRs.next()) {
System.out.println("Matricola: " +
jdbcRs.getString(1) +
" Nome: " + jdbcRs.getString(3));
}
} catch (SQLException exception) {
// In caso di errore...
}
}
```

L'istruzione

```
myRowSetFactory = RowSetProvider.newFactory();
crea l'oggetto myRowSetFactory di tipo RowSetProvider usando l'implementazione RowSetFactory di default com.sun.rowset.RowSetFactoryImpl. In alternativa, se lo specifico driver JDBC utilizzato ha una sua propria implementazione per RowSetFactory, è possibile specificarla come parametro per newFactory. Le successive quattro istruzioni creano l'oggetto jdbcRs di tipo JdbcRowSet e configurano le proprietà di connessione al database.
```

L'interfaccia *RowSetFactory* contiene metodi per creare i differenti tipi di *RowSet* implementabili:

- *createCachedRowSet*;
- *createFilteredRowSet*;
- *createJdbcRowSet*;
- *createJoinRowSet*;
- *createWebRowSet*.



4.1 Notifica degli eventi

L'interfaccia *RowSet* prevede l'implementazione del modello a eventi dei componenti *JavaBeans*. Gli oggetti *RowSet* notificano a un eventuale *listener* tre diversi tipi di eventi:

- movimento del cursore dei record (metodo *cursorMoved*);
- aggiornamento, inserimento e cancellazione di un record (metodo *rowChanged*);
- modifica del contenuto dell'oggetto (metodo *rowSetChanged*).

Gli oggetti *listener* che ricevono gli eventi generati da oggetti *RowSet* devono implementare l'interfaccia *RowSetListener* ed essere registrati per ogni specifico oggetto ai cui eventi sono interessati.

ESEMPIO
La seguente classe realizza a solo titolo di esempio un visualizzatore degli eventi generati da un oggetto *JdbcRowSet* nel corso del suo funzionamento:

```
import java.sql.*;
JdbcRowSet jdbcRs = null;
import javax.sql.rowset.*;
import com.sun.rowset.*;

class TestListener implements RowSetListener {
public void cursorMoved(RowSetEvent event) {
System.out.println("Spostamento cursore.");
}

public void rowChanged(RowSetEvent event) {
System.out.println("Cambiamento record.");
}

public void rowSetChanged(RowSetEvent event) {
System.out.println("Modifica RowSet.");
}
}
```

La registrazione del *listener* avviene aggiungendo questa riga nel costruttore della classe *GestionePersonale*:

```
rowset.addRowSetListener(new TestListener());
```

L'esecuzione del metodo *main* della classe produce in output le stringhe di testo «Spostamento cursore» e «Cambiamento record» in corrispondenza dell'esecuzione delle istruzioni di iterazione sui dati e di modifica (inserimento, aggiornamento o cancellazione) dei record.

5 Gestione delle transazioni

Al momento della creazione di una connessione JDBC, questa si trova in uno stato di *autocommit*: ogni singolo comando SQL viene gestito come una transazione per la quale viene eseguito un *commit* automatico dopo la sua esecuzione. Avendo la necessità che un insieme di comandi sia eseguito in modo atomico è necessario disabilitare la modalità *autocommit*:

```
Connection con = DriverManager.getConnection(...);
con.setAutoCommit(false);
```



Accesso a una base di dati in linguaggio Java con JDBC

Una volta disabilitato l'*autocommit*, l'esecuzione dei comandi SQL non modifica il database fino a quando non viene eseguito esplicitamente il metodo *commit*:

```
con.commit();
```

In questo modo, se si verifica un qualsiasi errore, è possibile annullare tutte le modifiche apportate, ma non ancora confermate, eseguendo un *rollback*:

```
con.rollback();
```

OSSERVAZIONE Normalmente l'invocazione del metodo *commit* avviene dopo l'esecuzione dell'ultimo comando SQL che costituisce la transazione, mentre l'invocazione del metodo *rollback* avviene come parte del codice di gestione di un'eventuale eccezione:

```
try {
    ...
    ...
    ...
    con.commit();
}
catch (SQLException exception) {
    con.rollback();
}
```

ESEMPIO Il seguente metodo della classe *GestionePersonale* applica a tutto il personale di una specifica qualifica un aumento percentuale limitando il nuovo stipendio a un valore massimo:

```
public void aumentoStipendio(String qualifica, double percentuale,
    double massimo) throws SQLException {
    con.setAutoCommit(false);

    try {
        PreparedStatement aumenta = con.prepareStatement("UPDATE Personale SET stipendio =
            (stipendio + stipendio*?/100) WHERE qualifica = ?;");
        aumenta.setDouble(1, percentuale);
        aumenta.setString(2, qualifica);
        aumenta.executeUpdate();
        PreparedStatement limita = con.prepareStatement("UPDATE Personale
            SET stipendio=? WHERE qualifica =
            AND stipendio > ?;");
        limita.setDouble(1, massimo);
        limita.setString(2, qualifica);
        limita.setDouble(3, massimo);
        limita.executeUpdate();
        con.commit();
    }
    catch (SQLException exception) {
        con.rollback();
    }
    con.setAutoCommit(true);
}
```

Metodi per JdbcRowSet

L'interfaccia *JdbcRowSet* prevede, per la gestione delle transazioni, i metodi indicati:

- *setAutoCommit*: abilita/disabilita la modalità *autocommit*;
- *commit*: termina ed esegue il *commit* della transazione;
- *rollback*: termina e annulla la transazione.

I CONCETTI CHIAVE

ACCESSO A DATABASE INIDIPENDENTE DAL SERVER DBMS UTILIZZATO.

Effettuare un accesso al database indipendentemente dal specifico server DBMS utilizzato rappresenta il problema principale nell'implementazione di applicazioni distribuite in rete, la cui soluzione richiede funzionalità di adattamento e di conversione che permettano lo scambio di informazione tra sistemi e applicazioni eterogenei. Per far fronte a queste problematiche nello sviluppo di applicazioni software si ricorre a tecnologie standard implementate in forma di API (*Application Program Interface*).

DRIVER JDBC (OPEN DATABASE CONNECTIVITY). Tecnologia sviluppata da Microsoft negli anni Novanta del secolo scorso con l'intento di definire un'interfaccia standard per l'accesso da codice in linguaggio C/C++ a database in modo indipendente sia dal sistema operativo sia dal server DBMS.

DRIVER JDBC PURE JAVA CON CONNESSIONE DIRETTA AL SERVER DBMS. Il driver JDBC è in questo caso completamente realizzato in Java e – essendo specifico per il DBMS utilizzato – converte le chiamate JDBC direttamente nel protocollo di rete usato dal server DBMS.

USO DI API JDBC PER L'INTERAZIONE CON UNA BASE DI DATI GESTITA DA UNO SPECIFICO DBMS.

La tipica sequenza di interazione tra un'applicazione e un server DBMS per l'accesso a una base di dati è la seguente:

- 1) caricamento del driver (opzionale);
- 2) connessione al server DBMS;
- 3) esecuzione di comandi/*query* SQL;
- 4) recupero dei risultati ed eventuale gestione degli errori;
- 5) disconnessione dal server DBMS.

CARICAMENTO DEL DRIVER. Il driver manager mantiene una lista di classi che implementano l'interfaccia *Driver*, ma la specifica implementazione di un driver deve essere registrata nel driver manager. Nelle versioni più recenti di JDBC la registrazione dei driver avviene automaticamente in fase di connessione con il server DBMS e non è necessario effettuarla esplicitamente.

DRIVER. Sono generalmente librerie caricate dinamicamente che implementano le funzioni API; ne esiste una specifica per ogni particolare DBMS con il compito di:

- nascondere le differenze di interazione dovute ai vari DBMS, sistemi operativi e protocolli di rete impiegati;
- trasformare le invocazioni delle funzioni API nel «dialetto» SQL usato dal server DBMS, o nelle corrispondenti funzioni API supportate dal server DBMS;
- gestire le transazioni, eseguire i comandi e le *query* SQL, inviare e recuperare i dati, gestire gli errori, ecc.

JDBC-ODBC BRIDGE. Driver JDBC per l'accesso al ser-

ver DBMS tramite un *driver ODBC*. Non supportato a partire dalla versione JSE8.

DRIVER JDBC REALIZZATO SOLO IN PARTE IN JAVA E DBMS MIDDLEWARE. Le chiamate inoltrate a JDBC sono elaborate da un driver locale completamente codificato in Java che le trasforma utilizzando un protocollo indipendente da quello del server DBMS in invocazioni per l'applicazione *middleware* che, a sua volta, si interfaccia con lo specifico DBMS.



terfaccia `Statement` che dispone di metodi per eseguire interrogazioni su dati o comandi DML/DDL.

PREPARED STATEMENT. Il metodo `prepareStatement` di un oggetto che implementa l'interfaccia `Connection` consente di creare `statement` predefiniti eseguibili più volte (`prepared statement`); questa tecnica riduce i tempi di esecuzione del comando/`query` per comandi/`query` che devono essere eseguiti molte volte. Utilizzando `statement` di questo tipo è possibile inserire nella stringa che rappresenta il comando SQL uno o più parametri identificati da simboli `<?>`, i cui valori dovranno essere forniti al momento dell'esecuzione effettiva della `query`.

RECUPERO DEI RISULTATI. Il metodo `executeQuery`, definito dalle interfacce `Statement` e `PreparedStatement`, restituisce un oggetto che implementa l'interfaccia `ResultSet`, che può contenere un risultato composto da una sequenza di record iterabili mediante il metodo `next`; l'accesso ai valori dei campi del singolo record avviene mediante metodi specifici che accettano come parametro l'indice posizionale del campo o il suo nome.

DISCONNESSIONE DAL SERVER DBMS. La disconnessione dal server DBMS si effettua invocando il metodo `close` dell'oggetto restituito dall'invocazione del metodo `getConnec-`

CRUD (CREATE, READ, UPDATE, DELETE). Acronimo con cui ci si riferisce alle operazioni che normalmente si effettuano su una base di dati: *Create* (creazione di nuovi record di dati), *Read* (ricerca e lettura di record di dati), *Update* (aggiornamento di record di dati esistenti), *Delete* (eliminazione di record di dati). La realizzazione di una classe CRUD in linguaggio Java deve prevedere – oltre alla definizione di specifiche classi per la rappresentazione dei record di dati – lo sviluppo di una classe di gestione dei dati i cui metodi incapsulino l'accesso al database che rende persistenti i dati stessi.

OGGETTI ROWSET. `RowSet` è un'interfaccia che deriva dall'interfaccia `ResultSet`, ma che espone la tipica inter-

faccia di un oggetto `JavaBeans`: un oggetto che implementa l'interfaccia `RowSet` ha lo scopo di mantenere una corrispondenza tra i dati che contiene nella forma di una sequenza di record e la loro sorgente, generalmente un database relazionale al quale si connette utilizzando JDBC. Alcuni vantaggi del ricorso a `RowSet` rispetto a `ResultSet` consistono nel fatto che `RowSet` è sempre «navigabile» in entrambe le direzioni e che l'aggiornamento dei dati che un oggetto `RowSet` contiene comporta l'aggiornamento della sorgente dei dati (nel caso di un database relazionale dei record di una o più tabelle).

NOTIFICA EVENTI CON OGGETTI ROWSET. L'interfaccia `RowSet` prevede l'implementazione del modello a eventi dei componenti `JavaBeans`. Gli oggetti `RowSet` notificano a un eventuale *listener* tre diversi tipi di eventi: movimento del cursore dei record (metodo `cursorMoved`); aggiornamento, inserimento e cancellazione di un record (metodo `rowChanged`). Gli oggetti *listener* che ricevono gli eventi generati da oggetti che implementano l'interfaccia `RowSet` devono implementare l'interfaccia `RowSetListener` ed essere registrati per ogni specifico oggetto ai cui eventi sono interessati.

JDBC E TRANSAZIONI. JDBC prevede la gestione transazionale di tutte quelle operazioni che prevedono una modifica dei dati contenuti nel database. Al momento della creazione di una connessione JDBC questa si trova in uno stato di *autocommit*: ogni singolo comando SQL viene gestito come una transazione per la quale viene eseguito un *commit* automatico dopo la sua esecuzione. Avendo la necessità che un insieme di comandi sia eseguito in modo atomico è necessario disabilitare la modalità *autocommit* così che l'esecuzione dei comandi SQL non modifichi il database fino a quando non viene eseguito esplicitamente il *commit*. In questo modo, se si verifica un qualsiasi errore, è possibile annullare tutte le modifiche apportate, ma non ancora confermate, eseguendo un *rollback*.



RIPASSA CON LA MAPPA

