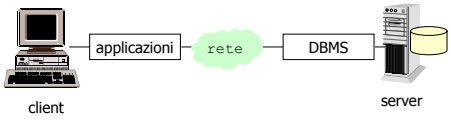
Accesso a Database con JDBC

Sommario

- Introduzione
  - architettura generale di riferimento
  - flusso tipico delle applicazioni ODBC/JDBC
- ODBC
  - architettura
  - il flusso generale di un'applicazione ODBC
- JDBC
  - architettura
  - tipologie di driver
  - le classi principali
  - il flusso generale di un'applicazione JDBC

Architettura client/server



Il SI è costituto da applicazioni che accedono a DB secondo il paradigma client/server

- il client e il server (DBMS) possono essere eseguiti su macchine diverse
- uno stesso server può servire più client
- un client può utilizzare più server

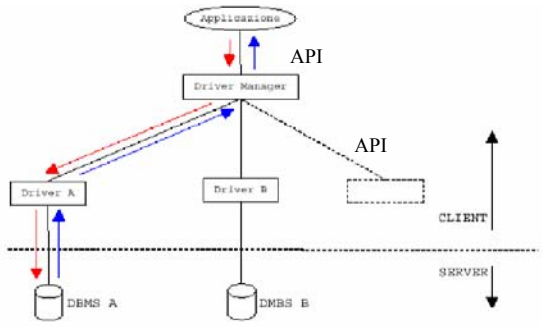
Introduzione

- Il problema principale nelle architetture client-server è l'esistenza di client e server (DBMS) eterogenei:
  - *Front-end*: gui, form, interfacce web
  - *Back-end*: DBMS
- E' necessario di stabilire come le applicazioni client possano comunicare con il server:
  - come rappresentare le query
  - come rappresentare i risultati delle query
- E' opportuno rendere l'accesso ai dati indipendente dallo specifico server considerato
  - se cambia il server non cambia l'applicazione
  - i tempi di sviluppo applicazioni client si riducono

Introduzione

- Accesso indipendente dal server
  - rappresenta il problema principale nello sviluppo di applicazioni eterogenee per sistemi distribuiti
  - richiede funzionalità di adattabilità e di conversione che permettano lo scambio di informazione tra sistemi, reti ed applicazioni, anche se eterogenei
- Per i sistemi di gestione dei dati esistono standard sviluppati in forma di API (Application Program Interface)
  - ODBC
  - JDBC

Architettura di riferimento



Flusso tipico delle applicazioni su basi di dati

- Applicazione su DB
  - è un programma che chiama specifiche funzioni API per accedere ai dati gestiti da un DBMS
- Flusso tipico:
  1. connessione alla sorgente dati (DBMS e specifico database)
  2. sottomissione di statement SQL per l'esecuzione
  3. recupero dei risultati e processamento degli errori
  4. *commit* o *rollback* della transazione che include lo statement SQL
  5. disconnessione

Driver Manager

- È una libreria che gestisce la comunicazione tra applicazione e driver
- risolve problematiche comuni a tutte le applicazioni
  - decide quale driver caricare, sulla base delle informazioni fornite dall'applicazione
  - caricamento del driver
  - chiamate alle funzioni dei driver
- l'applicazione interagisce solo con il driver manager

Driver

- Sono librerie dinamicamente connesse alle applicazioni che implementano le funzioni API
  - ciascuna libreria è specifica per un particolare DBMS
- Mascherano le differenze di interazione dovute ai DBMS, sistema operativo e protocollo di rete usati
- Traducono le varie funzioni API nel dialetto SQL usato dal DBMS (o nell'API supportata dal DBMS)
- Si occupano in particolare di:
  - iniziare transazioni
  - sottomettere statement SQL
  - inviare e recuperare dati
  - gestire errori

DBMS

- Il DBMS sostanzialmente rimane inalterato nel suo funzionamento
- riceve sempre e solo richieste nel linguaggio che supporta
- esegue lo statement SQL ricevuto dal driver e invia i risultati

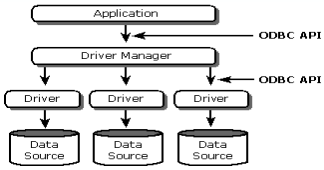
ODBC (Open DataBase Connectivity)

- Standard proposto da Microsoft nel 1991
- Supportato da praticamente tutti i DBMS relazionali
- Offre all'applicazione un'interfaccia che consente l'accesso ad un DB indipendente da:
  - il particolare dialetto di SQL
  - il protocollo di comunicazione da usare con il DBMS
  - la posizione del DBMS (locale o remoto)
- è possibile connettersi ad un particolare DB tramite un DSN (Data Source Name), che contiene tutti i parametri necessari alla connessione con il DB:
  - protocollo di comunicazione
  - tipo di sorgente dati (es: Oracle DBMS)
  - specifico database

ODBC

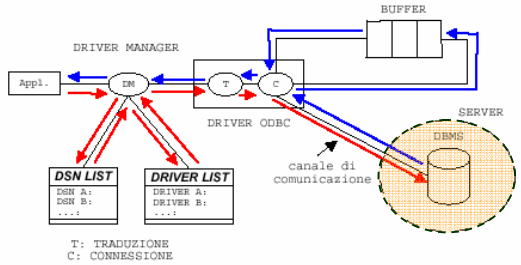
- L'architettura è conforme allo schema generale
- Le funzioni sono implementate in C e quindi non sono completamente portabili
- Il linguaggio supportato da ODBC è un SQL ristretto, costituito da un insieme minimale di istruzioni
  - scelta obbligata se si vuole permettere un cambio di DBMS lasciando inalterati i client
- Il linguaggio supporta sia query statiche (per applicazioni tradizionali) sia query dinamiche (per applicazioni interattive)
  - nel primo caso eventuali errori di SQL sono riportati al momento stesso della compilazione

ODBC: Architettura



- Driver Manager:
  - fornito con i sistemi operativi Windows
  - deve essere installato con Linux
- Driver:
  - forniti con il DBMS corrispondente
  - spesso installati e registrati a livello Driver Manager quando si installa il DBMS client

Funzionamento di query ODBC

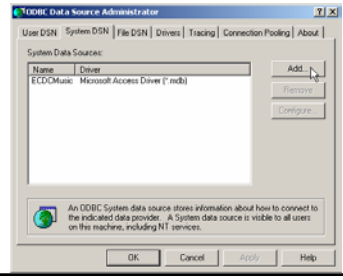


Come registrare un DSN?

- *Data Source Name* (DSN) è una stringa che indica:
  - tipo di driver
  - tipo di comunicazione
  - tipo di database
  - nome del DBMS e del database
- Tool di amministrazione dei driver ODBC
  - permettono di specificare tutti i dettagli di una sorgente dati DSN
  - dopo la registrazione, queste informazioni sono a disposizione del Driver Manager

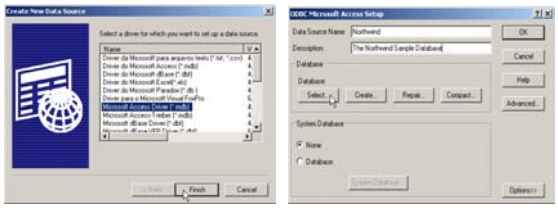
Uso di una base di dati Access via ODBC

1. clickare
   Start → Impostazioni → Pannello di Controllo → Strumenti di Amministrazione → Origine Dati → DSN System
2. premere
   Aggiungi



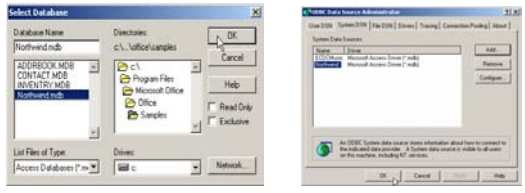
Uso di una base di dati Access via ODBC

3. Selezionare Microsoft Access Driver e premere Finish
4. Scrivere un nome in Data Source Name e premere Select



Uso di una base di dati Access via ODBC

5. Selezionare Northwind.mdb nella directory MSOffice\Samples e premere OK
6. premere OK nelle due finestre seguenti



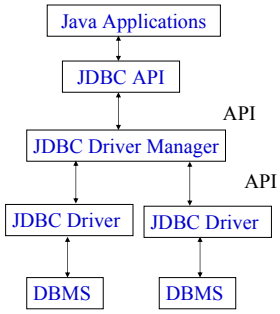
JDBC

- Limitazioni nell'uso di ODBC:
  - ODBC è un'API sviluppata in C che richiede API intermedie per essere usata con altri linguaggi
  - Interfaccia di non agevole interpretazione:
    - alcuni concetti sono portati a livello interfaccia ma sarebbe meglio mantenerli nascosti
- JDBC è stato sviluppato nel 1996 dalla Sun per superare questi problemi
  - non è un acronimo ma un trademark della SUN

JDBC

- Rappresenta una API standard per interagire con basi di dati da applicazioni Java
- Permette di ottenere una soluzione "pure Java" per l'interazione con DBMS
  - indipendenza dalla piattaforma
  - inclusa in JDK
- Interfaccia chiara e, nel contempo, flessibile

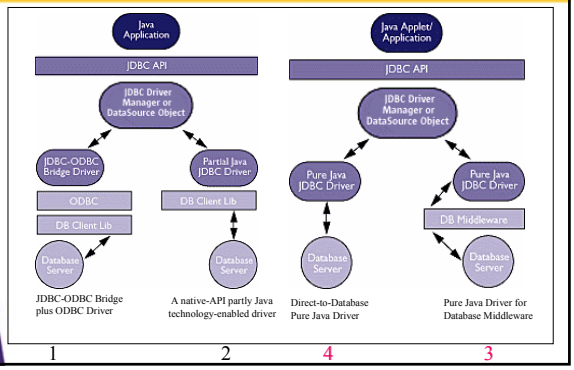
JDBC: Architettura generale



Tipi di driver

- Esistono quattro tipi di driver:
  - JDBC-ODBC Bridge + ODBC Driver
  - native-API partly Java technology-enabled driver
  - Pure Java Driver for Database Middleware
  - Direct-to-Database Pure Java Driver

Tipi di driver



Driver di tipo 1

- Accesso a JDBC tramite un driver ODBC
  - uso di JDBC-ODBC bridge
  - il codice ODBC binario e il codice del DBMS client, deve essere caricato su ogni macchina client che usa JDBC-ODBC bridge
- si consiglia di utilizzare questa strategia quando non esistono soluzioni alternative …
  - è necessario installare le librerie sul client perché non sono trasportabili su HTTP
  - compromette "write once, run anywhere"

Driver di tipo 2

- Le chiamate a JDBC vengono tradotte in chiamate all'API del DBMS prescelto
  - es. OCI Oracle
- Il driver contiene codice Java che chiama metodi scritti in C/C++
  - non utilizzabile per applet/servlet
  - anche in questo caso, codice binario deve essere caricato sul client
  - compromette "write once, run anywhere"

Driver di tipo 3

- Basato completamente su Java
  - utilizzabile con *applet/servlet*
- converte le chiamate JDBC nel protocollo di una applicazione middleware
  - che traduce quindi la richiesta del client nel protocollo proprietario del DBMS
  - l'applicazione middleware può garantire l'accesso a diversi DBMS
  - il protocollo middleware dipende dal DBSM sottostante

Driver di tipo 4

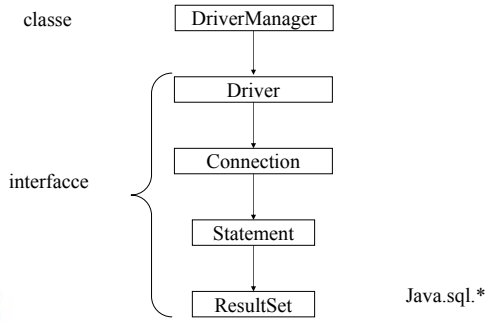
- Basato completamente su Java
  - utilizzabile con *applet/servlet*
- converte le chiamate JDBC nel protocollo di rete usato direttamente dal DBMS
  - permette quindi un accesso diretto dalla macchina client alla macchina server
- è la soluzione più pura in termini Java

Tipi di dato

ODBC/JDBC usano tipi SQL, mappati in tipi Java
- Gli identificatori sono definiti nella classe `java.sql.Types`

JDBC Types Mapped to Java Types	
JDBC Type	**Java Type**
CHAR	String
VARCHAR	String
LONGVARCHAR	String
NUMERIC	java.math.BigDecimal
DECIMAL	java.math.BigDecimal
BIT	boolean
TINYINT	byte
SMALLINT	short
INTEGER	int
BIGINT	long
REAL	float
FLOAT	double
DOUBLE	double
BINARY	byte[]
VARBINARY	byte[]
LONGVARBINARY	byte[]
DATE	java.sql.Date
TIME	java.sql.Time
TIMESTAMP	java.sql.Timestamp

Interfaccia JDBC (semplificata - solo le classi che vedremo)



Passo 1: caricamento driver

- Il driver manager mantiene una lista di classi che implementano l'interfaccia *java.sql.Driver*
  - l'implementazione del Driver deve essere registrata in qualche modo nel DriverManager
- Quando una classe *Driver* viene caricata, se ne crea un'istanza e ne viene effettuata anche la registrazione
  - Due modi:
    - `Class.forName("sun.jdbc.odbc.JdbcOdbcDriver");`
    - `DriverManager.registerDriver(new sun.jdbc.odbc.JdbcOdbcDriver());`
  - Il nome della classe da usare viene fornito con la documentazione relativa al driver

Passo 2: Connessione

- Per realizzare la connessione vengono utilizzate le seguenti classi ed interfacce:
  - classe `java.sql.DriverManager`: gestisce la registrazione dei driver
  - interfaccia `java.sql.Driver`: non viene esplicitamente utilizzata a livello applicativo
  - interfaccia `java.sql.Connection`: permette di inviare una serie di richieste SQL al DBMS
- È possibile connettersi a qualunque database, locale e remoto, specificandone l'URL
- l'URL ha tre parti: jdbc: <subprotocol>: <subname>
  - <subprotocol> identifica il driver o il meccanismo di connessione al database
  - <subname> dipende da subprotocol ed identifica lo specifico database

Passo 2: Connessione

- Esempi:
  - jdbc:oracle:thin:@everest:1521:GEN:
    - subprotocol: Oracle
    - subname:
      - *Thin:* specifica che si deve essere utilizzare *Oracle ODBC Thin driver*
      - *Everest:* specifica il nome della macchina
      - *1521:* numero porta
      - *GEN:* nome database Oracle
  - **`jdbc:mysql://cannings.org:3306/test`**
    - subprotocol: MySQL
    - subname:
      - `cannings.org` specifica il nome della macchina
      - `3306` : numero porta
      - `test` : nome database MySQL
  - **se si usa JDBC-ODBC driver: `jdbc:odbc:subname`**

Passo 2: Connessione

- La connessione avviene chiamando il metodo `getConnection` della classe `DriverManager`,
  - che restituisce un oggetto di tipo `Connection`

```
Connection con = DriverManager.getConnection(
                              "jdbc:odbc:miodatabase",
                              "myLogin",
                              "myPassword");
```

- Se uno dei driver caricati riconosce l'URL fornito dal metodo, il driver stabilisce la connessione

Passo 3: Creazione ed esecuzione degli statement

- Per creare ed eseguire uno statement vengono utilizzate le seguenti classi:
  - **`java.sql.Connection`**: per creare lo statement
  - **`java.sql.Statement`**: permette di eseguire gli statement
  - **`java.sql.ResultSet`**: per analizzare i risultati delle query
- un oggetto di tipo Statement viene creato a partire da un oggetto di tipo Connection e permette di inviare comandi SQL al DBMS :

```
Connection con;
…
Statement stmt = con.createStatement();
```


Esecuzione di statement

- Distinzione tra statement di interrogazione e statement di aggiornamento (come in ODBC)
  - per eseguire query:

```
stmt.executeQuery(”SELECT * FROM IMPIEGATI");
```

  - per eseguire operazioni di aggiornamento (inclusi gli statement DDL)

```
stmt.executeUpdate(”INSERT INTO IMPIEGATI
  VALUES‘AB34’,‘Gianni’, ‘Rossi’,’GT67’,1500");
stmt.executeUpdate("CREATE TABLE … )”);
```

- Il terminatore dello statement (es. ‘;’) è inserito dal driver
  - prima di inviare lo statement al DBMS per l'esecuzione

Prepared Statement

- E' possibile preparare gli statement prima dell'esecuzione (come in ODBC)
  - statement precompilati
- L'uso di prepared statement riduce i tempi di esecuzione dell'operazione
  - sono usati per statement utilizzati molte volte

```
PreparedStatement queryImp =
con.prepareStatement(”SELECT *
                      FROM IMPIEGATI");
                    …
queryImp.executeQuery();

queryImp.executeQuery();
```

Prepared Statement: uso di parametri

- La stringa SQL può contenere parametri identificati da '?' (eventualmente letti da input)
- Metodi **set*XXX*** (dove XXX è il tipo Java del parametro) consentono di completare gli statement SQL con i valori dei parametri

```
PreparedStatement queryImp =
    con.prepareStatement("SELECT * FROM IMPIEGATI
                          WHERE Nome = ?");
                    …
queryImp.setString(1, 'Rossi');
queryImp.executeQuery():
```


Passo 4: Elaborazione risultato

- JDBC restituisce i risultati di esecuzione di una query in un result set (come ODBC)

```
String query = " SELECT * FROM IMPIEGATI ";
ResultSet rs = stmt.executeQuery(query);
```

- Il result set è costruito solo per query e non per INSERT, DELETE, UPDATE (come in ODBC)
  - in questo caso viene restituito un intero, che rappresenta il numero di tuple modificate

Metodi per accedere alle tuple del risultato: *next*

- Il metodo next permette di iterare su un result set:
  - inizialmente il cursore è posizionato prima della prima tupla
  - il metodo diventa *false* quando non ci sono più tuple
  - iterazione tipica:

```
while (rs.next()) {
      /* elabora la tupla corrente */
}
```

  - Esempio:

```
while (rs.next()) {
      String s = rs.getString("Cognome");
      float n = rs.getFloat("Stipendio");
      System.out.println(s + "   " + n);
}
```


Metodi per accedere agli attributi: *getXXX*

- Permette di recuperare il valore associato ad un attributo della tupla puntata dal cursore
  - XXX è il tipo Java in cui si deve convertire il valore
    - getInt per valori numerici
    - getString per char, varchar
  - Esempio:

```
String s = rs.getString("Cognome");
```

- Gli attributi possono anche essere acceduti tramite la notazione posizionale

```
String s = rs.getString(2);
int n = rs.getInt(5);
```


Esempio

Nel seguito gli esempi si riferiranno alla tabella *Impiegati* definita in SQL come segue:

```
CREATE TABLE Impiegati
     (Matr VARCHAR(4),
     Nome VARCHAR(20),
     Cognome VARCHAR(20),
     Manager VARCHAR(4),
     Stipendio NUMBER);
```


Esempio esecuzione diretta

```
import java.sql.*;
import java.io.*;
class JdbcTest {
  public static void main (String args []) {
    Class.forName ("sun.jdbc.JdbcOdbcDriver");
    Connection con=
        DriverManager.getConnection(
            jdbc:odbc:dbimpiegati, "", "");
    Statement st = con.createStatement ();
    String stmt = "SELECT * FROM Impiegati
                   WHERE Cognome ='Rossi'";
    ResultSet  impRossi = st.executeQuery (stmt);
    while ( impRossi.next() )
        System.out.println(
            impRossi.getString("Stipendio"));
  }
}
```

Esempio prepared statement

```
import java.sql.*;
import java.io.*;
class JdbcTest {
   public static void main (String args []) {
      Class.forName ("oracle.jdbc.OracleDriver");
      Connection con = DriverManager.getConnection(
             jdbc:odbc:dbimpiegati,"", "");

       String stmt ="SELECT * FROM Impiegati
                          WHERE Cognome = 'Rossi'";

      PreparedStatement prepSt =
                   con.prepareStatement (stmt);
       ResultSet impRossi = prepSt.executeQuery ();
       while ( impRossi.next() )
          System.out.println (
             impRossi.getString ("Stipendio"));
   }
}
```


Passo 6: Disconnessione (chiusura)

- Il metodo `close()` permette di risparmiare risorse:
  - rilascio degli oggetti di classe **`Connection`**, **`Statement`**, **`ResultSet`** non più utilizzati
- la chiusura di un oggetto di tipo `Connection` chiude tutti gli `Statement` associati
- la chiusura di uno `Statement` chiude `ResultSet` associati

Eccezioni: classe *java.sql.SQLException*

- Estende la classe `java.lang.Exception`
- Fornisce informazioni ulteriori in caso di errore di accesso al database
  - la stringa SQLState che rappresenta la codifica dell'errore in base allo standard X/Open
    - `getSQLState()`
  - il codice di errore specifico al DBMS
    - `getErrorCode()`
  - una descrizione dell'errore
    - `getMessage()`

Esempio

```
import java.sql.*;
import java.io.*;
class JdbcTest {
   static String ARS_URL = "…";
   public static void main (String args []) {
     try{
        Class.forName ("…");
        Connection ARS;
        ARS =DriverManager.getConnection(ARS_URL, "…", "…");
     }
     catch (SQLException e) {
        while( e!=null){
           System.out.println("SQLState: "+ e.getSQLState());
           System.out.println(" Code: " +e.getErrorCode());
           System.out.println(" Message: " +e.getMessage());
           e = e.getNextException();
        }
     }
   }
}
```


Warning: la classe java.sql.SQLWarning

- Sottoclasse di `SQLException`
- Fornisce informazioni su warning innescati dall'accesso al DB
  - Non bloccano l'esecuzione del programma
- I warning vengono collegati agli oggetti i cui metodi hanno generato l'eccezione
  - connection
  - statement
  - resultset
- Sono recuperabili con i metodo `getWarnings()`

Esempio

```
Statement selImp = con.createStatement ();
String stmt =  "SELECT * FROM Impiegati
                      WHERE Cognome ='Rossi'";
ResultSet  impRossi = selImp.executeQuery (stmt);
while ( impRossi.next() ) {
    System.out.println (impRossi.getString ("Stipendio"));
    SQLWarning warning_stmt = selImp.getWarnings();
    while (warning_stmt != null) {
        System.out.println("Message: " +
                      warning_stmt.getMessage());
        System.out.println("SQLState: " +
                      warning_stmt.getSQLState());
        System.out.println("Vendor error code: " +
                      warning_stmt.getErrorCode());
        warning_stmt = warning_stmt.getNextWarning();
    }
}
```