Database
JDBC

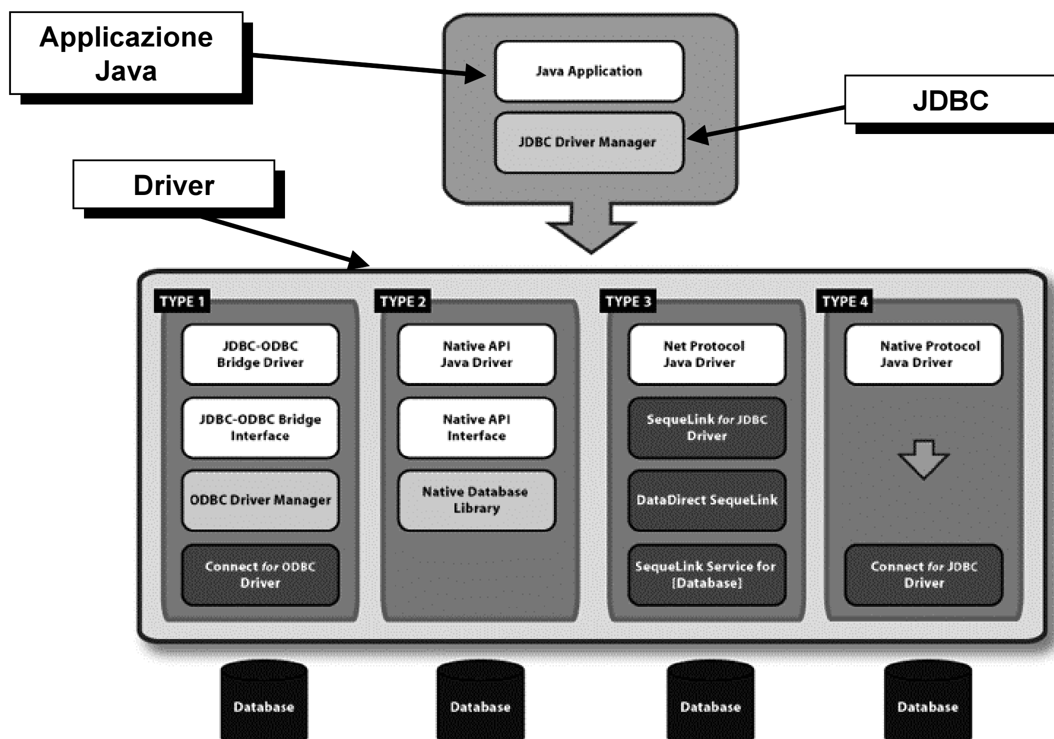
Che cos' è JDBC

- JDBC è un insieme di classi che consentono ad un programma Java di operare su un database
- Con JDBC è possibile:
  - Connettersi con un RDBMS
  - Aprire un database
  - Eseguire comandi SQL (sia DDL che DML)
  - Leggere i risultati di un estrazione fatta con il comando SELECT

Driver

- **JDBC permette di accedere in modo uniforme ai database ed è indipendente dal tipo di database**
- **Utilizza dei driver per accedere a specifici RDBMS**
- **I driver sono classi Java e il loro compito è quello di fare da "adattatori" fra JDBC e un determinato RDBMS**
- **JDBC viene quindi definito driver manager**
- **Esistono diversi tipi di driver: alcuni si connettono direttamente al database, altri utilizzano i driver nativi forniti dal produttore del RDBMS**

Schema esemplificativo



Esempio

- Anche per JDBC utilizzeremo il database SQLite
- E' infatti disponibile un driver JDBC che consente di operare con SQLite da un programma Java
- Come esempio immaginiamo di dover gestire un negozio che vende diversi tipi di caffè provenienti da diversi fornitori
- Definiremo una classe che rappresenta il negozio e opera sul DB per:
  - Creare le tabelle
  - Inserire, modificare e cancellare i dati
  - Estrarre i dati
  - Eliminare le tabelle

Struttura del DB

- Il database, che chiameremo CoffeeShop, sarà costituito da due tabelle: Coffees e Suppliers:

Coffees

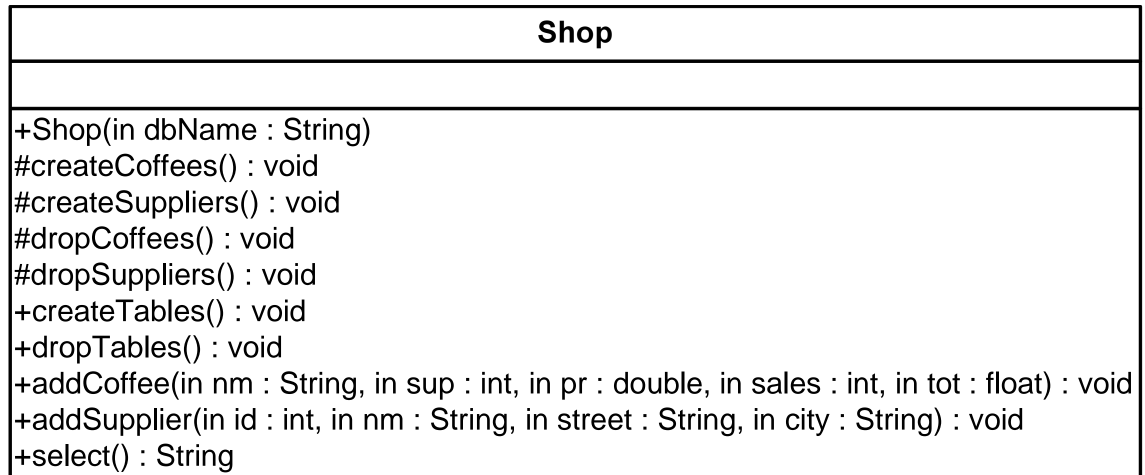
Name	Supplier	Price	Sales	Total
Colombian	101	7.99	0	0
French Roast	49	8.99	0	0
Espresso	150	9.99	0	0
Colombian Decaf	101	8.99	0	0
French Roast Decaf	49	9.99	0	0

Suppliers

ID	Name	Street	City
101	Acme, Inc.	99 Marker Street	Groundsville
49	Superior Coffee	1 Party Place	Mendocino
150	The High Ground	100 Coffee Lane	Meadows

Diagramma della classe Shop

- **Le operazioni sul database saranno gestito dalla classe Shop il cui diagramma UML è il seguente**



Definizione della classe

- **Le classi di JDBC sono contenute nel package java.sql**
- **Nella classe definiamo due attributi che ci serviranno successivamente**

```
import java.sql.*;

public class Shop
{
  Connection con;
  Statement stmt;
  ...
}
```

Il costruttore - Descrizione

- Il costruttore crea la connessione con il DB il cui nome viene passato come parametro
- Innanzitutto attiviamo il driver:

```
Class.forName("SQLite.JDBCDriver");
```

- Per creare la connessione utilizziamo il metodo statico getConnection() della classe DriverManager
- getConnection() richiede una stringa di connessione costituita dal nome del driver e dal nome del DB
- Otteniamo così un'istanza di Connection che memorizziamo nell'attributo con:

```
con =
  DriverManager.getConnection("jdbc:sqlite:/"+dbName);
```

- Infine creiamo un'istanza di Statement:

```
stmt = con.createStatement();
```

- I vari metodi possono generare eccezioni e quindi inseriamo il tutto in un blocco try/catch

Il costruttore - Implementazione

- Ecco qui l'implementazione completa:

```
public Shop(String dbName)
{
  try
  {
    Class.forName("SQLite.JDBCDriver");
    con = DriverManager.getConnection("jdbc:sqlite:/"+dbName);
    stmt = con.createStatement();
  }
  catch (ClassNotFoundException e) // generata da Class.forName
  {
    System.out.println("Driver non trovato");
  }
  catch (SQLException se)
  {
    System.out.println(se.getMessage());
  }
}
```

Per SQLite il nome del DB è il percorso dove si trova il file che lo contiene

La classe Statement

- **La classe Statement permette di eseguire comandi SQL passati sotto forma di stringa**
- **Statement definisce due metodi importanti:**
- **`executeUpdate()` che consente di eseguire un comando che non prevede l'estrazione di dati: quindi tutti i comandi DDL più INSERT, UPDATE, DELETE**
- **Prende come parametro una stringa e non ha valori di ritorno**
- **`executeQuery()` che consente di eseguire un comando che prevedono l'estrazione di dati, quindi i comandi SELECT**
- **Prende come parametro una stringa e restituisce un'istanza di una classe che implementa l'interfaccia ResultSet, che permette di accedere ai dati che costituiscono il risultato della query**

Creazione della tabella Coffees

- **Il createCoffees() ci consente di creare la tabella Coffees: usa executeUpdate per inviare il comando DDL**
- **E' protected perché è stato pensato per un uso interno alla classe: le altre classi invocheranno createTables()**
- **Non vogliamo infatti che si possa creare solo una parte del DB**

```
protected void createCoffees()
{
  try
  {
    stmt.executeUpdate(
     "CREATE TABLE COFFEES (NAME  VARCHAR(32), "+
     "SUPPLIER INTEGER, PRICE FLOAT, SALES INTEGER, "+
     "TOTAL FLOAT)");
    }
   catch (SQLException se)
   {
    System.out.println(se.getMessage());
   }
}
```

Distruzione della tabella Coffees

- **dropCoffees() ci consente invece di distruggere la tabella Coffees**
- **Anche dropCoffees() è protected**
- **Da notare il fatto che gestiamo in tutti i metodi le eccezioni con try/catch**

```
protected void dropCoffees()
{
  try
  {
    stmt.executeUpdate("DROP TABLE COFFEES");
  }
  catch (SQLException se)
  {
    System.out.println(se.getMessage());
  }
}
```


Creazione della tabella Suppliers

- **Il createSuppliers() ci consente di creare la tabella Suppliers**
- **Da notare la necessità di utilizzare la concatenazione di stringhe per gli statement lunghi: una stringa Java deve infatti stare su una sola riga**

```
protected void createSuppliers()
{
  try
  {
    stmt.executeUpdate(
     "CREATE TABLE SUPPLIERS (SUP_ID INTEGER, "+
     "SUP_NAME VARCHAR(32), STREET VARCHAR(32), "+
     "CITY VARCHAR(32))");
  }
  catch (SQLException se)
  {
    System.out.println(se.getMessage());
  }
}
```

Distruzione della tabella Suppliers

- **dropSuppliers() ci consente invece di distruggere la tabella dei fornitori**

```
protected void dropSuppliers()
{
  try
  {
    stmt.executeUpdate("DROP TABLE SUPPLIERS");
  }
  catch (SQLException se)
  {
    System.out.println(se.getMessage());
  }
}
```


createTables() e dropTables()

- **Vediamo infine i due metodi pubblici che creano e distruggono in modo consistente tutte le tabelle del database**
- **Utilizzano i metodi protetti definiti precedentemente**

```
public void createTables()
{
  createCoffees();
  createSuppliers();
}
```

```
public void dropTables()
{
  dropCoffees();
  dropSuppliers();
}
```

addCoffee()

- **addCoffee() inserisce una riga nella tabella Coffees usando il comando INSERT in cui inseriamo i parametri mediante concatenazione di stringhe**

```
public void addCoffee(String nm, int sup, double pr,
  int sales, int tot)
{
  try
  {
    stmt.executeUpdate(
      "INSERT INTO COFFEES "+
      "(name,supplier,price,sales,total) "+
      "VALUES "+
      "('"+nm+"',"+sup+","+pr+","+sales+","+tot+")");
  }
  catch (SQLException se)
  {
    System.out.println(se.getMessage());
  }
}
```


addSupplier()

- **addSupplier() inserisce invece una riga nella tabella dei fornitori**

```
public void addSupplier(int id, String nm, String
street,
  String city)
{
  try
  {
    stmt.executeUpdate(
     "INSERT INTO SUPPLIERS (id,name,street,city)"+
     "VALUES ("+id+",'"+nm+"','"+
     street+"','"+city+"')");
  }
  catch (SQLException se)
  {
    System.out.println(se.getMessage());
  }
}
```

L'interfaccia ResultSet

- Per eseguire comandi SELECT si utilizza il metodo executeQuery() della classe Statement
- Il metodo restituisce un'istanza di una classe che implementa l'interfaccia ResultSet, con cui possiamo accedere ai dati estratti.
- Con il metodo next() scorriamo le righe
- Con getString() e getDouble() leggiamo i campi della riga corrente per nome (tupla) o per posizione (n-upla):

```
ResultSet rs =
  stmt.executeQuery("SELECT Name, Price FROM Coffees");

while (rs.next())
{
   String name = rs.getString("Name");
   Double price = rs.getDouble("Price");
   String priceAsString = rs.getString("Price");
   /* oppure:
      String name = rs.getString(1);
      Double price = rs.getDouble(2); */
}
```

Indipendentemente dal tipo possiamo estrarre sempre un campo come stringa

Metadati

- ResultSet ci permette anche di accedere ai metadati: ovvero alla descrizione della tabella restituita dal comando SELECT
- Usando il metodo getMetaData() otteniamo un'istanza di una classe che implementa l'interfaccia ResultSetMetaData.
- Possiamo quindi scoprire, per esempio, il numero di colonne e i loro nomi:

```
ResultSet rs =
  stmt.executeQuery("SELECT Name, Price FROM Coffees");

ResultSetMetaData md = rs.getMetaData();

int n = md.getColumnCount();

String s = md.getColumnName(1);
```

select()

- **Per completare il nostro semplice esempio proviamo a scrivere un metodo select() che ci permette di estrarre dati dal DB restituendo i dati in una stringa:**

```
public String select(String query)
  {
    String result = "";
    try
    {
      ResultSet rs = stmt.executeQuery(query);
      int cnt = rs.getMetaData().getColumnCount();
      while (rs.next())
      {
        for (int i=1; i<=cnt; i++)
          result = result + rs.getString(i)+" ";
        result = result+"\n";
      }
    }
    catch (SQLException se)
    { System.out.println(se.getMessage()); }
    return result;
  }
}
```


Esempio di uso

- **Terminiamo con un esempio di utilizzo della nostra classe:**

```
Coffees cf = new Coffees("C:/sqlite/coffeeshop.db");

cf.createTables();

cf.addCoffee("Colombian", 101, 7.99, 0, 0);
cf.addCoffee("French Roast", 49, 8.99, 0, 0);
cf.addCoffee("Espresso", 150, 9.99, 0, 0);
cf.addCoffee("Colombian Decaf", 101, 8.99, 0, 0);
cf.addCoffee("French Roast Decaf", 49, 9.99, 0, 0);

cf.addSupplier(101,"Acme,Inc.","99 Market Street","Groundsville");
cf.addSupplier(49,"Superior coffee","1 Party Place","Mendocino");
cf.addSupplier(150,"The High Ground","100 Coffee Lane","Meadows");

String s = cf.select("select * from coffees");
System.out.println("Coffee list:\n"+s);

s = cf.select("select * from coffees, suppliers "+
              "where coffees.suppplier = suppliers.id");
System.out.println("Coffee/supplier list:\n"+s);
```

Per SQLite il nome del DB è il percorso dove si trova il file che lo contiene