Basi Dati

JDBC

a.a. 2021/2022
Prof.ssa G. Tortora
Prof. M.Risi

Database airdb (airdb.sql)

```
DROP DATABASE IF EXISTS airdb;
CREATE DATABASE airdb;

DROP USER IF EXISTS 'airuser'@'localhost';
CREATE USER 'airuser'@'localhost' IDENTIFIED BY 'airuser';
GRANT ALL ON airdb.* TO  'airuser'@'localhost';

USE airdb;

DROP TABLE IF EXISTS aerei;
CREATE TABLE aerei (
  id char(20) primary key,
  produttore char(20) not null,
  modello char(10) not null,
  dataimm date,
  numposti int
);

LOAD DATA LOCAL INFILE 'datiaerei.sql' INTO TABLE aerei (id,produttore,modello,dataimm,numposti);

LOAD DATA LOCAL INFILE 'datiaerei2.sql' INTO TABLE aerei
FIELDS TERMINATED BY ',' ENCLOSED BY '"' IGNORE 1 LINES (id,produttore,modello,dataimm,numposti);
```

Prima di eseguire lo script, lanciare il commando: **`SET GLOBAL local_infile = true;`**
e aggiungere il parametro **`--local_infile=1`** al comando `mysql`.

Basic JDBC

- Import the packages
- Register Driver & Open Connection
- Execute a Query
- Extract data from result set
- Clean up the environment

Connessione.java (MySQL)

```
import java.sql.*;

class Connessione {
   public static void main(String args[]) throws Exception {
      try {
        Class.forName("com.mysql.cj.jdbc.Driver");

        String url = "jdbc:mysql://localhost:3306/airdb";
        Connection con = DriverManager.getConnection(url, "airuser", "airuser");

        System.out.println("Connessione OK \n");
        con.close();
      }
      catch (ClassNotFoundException e) {
        System.out.println("DB driver not found \n");
        System.out.println(e);
      catch(Exception e) {
        System.out.println("Connessione Fallita \n");
        System.out.println(e);
      }
   }
}
```

I Driver JDBC di MySQL

- Una volta costruito il database, dobbiamo procurarci i driver nativi Java MySQL, chiamati:

  MySQL Connector/J

- Dal pacchetto scaricato preleviamo il file:

  mysql-connector-java-8.X.X.jar

Compilare ed eseguire l'applicazione JDBC

- Bisogna istallare la distribuzione *Java SDK, Standard Edition (JDK)*.
- Per verificare la presenza di Java:

```
java –version
```

- Per compilare:

```
javac -cp .;mysql-connector-java-8.X.X.jar Connessione.java
```

- Per eseguire:

```
java -cp .;mysql-connector-java-8.X.X.jar Connessione
```

Obiettivo: creare una applicazione completa JDBC

- Implementare la classe `AirDB.java` per la gestione delle operazioni sul database **airdb**.

- Le operazioni CRUD riguardano la tabella **aerei**:
  - **C**reate
  - **R**etrive
  - **U**pdate
  - **D**elete

- Implementare un'interfaccia utente (tesuale) per poter richiamare le operazioni CRUD.

Preparazione

- Importare le classi:

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

- Usare i blocchi **`try`** … **`catch`**:
  - Il primo blocco **try** contiene il metodo **`Class.forName`**, dal package ***`java.lang`***. Questo metodo lancia un **`ClassNotFoundException`**, in maniera tale da permetter al blocco catch di gestire subito l'eccezione.
    - Può essere gestita una sola volta nel costruttore.
  - Il secondo blocco **try** contiene i metodi JDBC, che lanciano tutti un eccezione del tipo **`SQLException`**, così il blocco catch può gestire solamente oggetti di quel tipo.

Gestire le eccezioni

```
try {
  Class.forName("com.mysql.cj.jdbc.Driver");
} catch(ClassNotFoundException e) {
 System.err.print("ClassNotFoundException:" + e.getMessage());
}

try {
  // il codice JDBC che genera le eccezioni SQLException
  // vanno qui
} catch(SQLException ex) {
  System.err.println("SQLException:" + ex.getMessage());
}
```

Caricamento dei driver

```
// per un database MySQL
String driver = "com.mysql.cj.jdbc.Driver";

// se il database è Oracle
… driver = "oracle.jdbc.OracleDriver";

// se il driver è JDBC-ODBC (tipo 1)
… driver = "sun.jdbc.odbc.JdbcOdbcDriver";

// se il database è PostgresQL
… driver = "com.postgresql.Driver";

// il metodo forName forza il caricamento del driver
Class.forName(driver); // lancia una ClassNotFoundException
```

Creazione classe e costruttore

```
import java.sql.*;

public class AirDB {

  public AirDB() {
    try {
      Class.forName("com.mysql.cj.jdbc.Driver");
    }
    catch (ClassNotFoundException e) {
      System.err.print("ClassNotFoundException:" + e.getMessage());
    }
  }

}
```

Creazione della connessione

```
// per un database MySQL
String url = "jdbc:mysql://localhost:3306/airdb";
// per il database Oracle di tipo thin
… url = "jdbc:oracle:thin:@//localhost:1521/airdb";
// per il driver JDBC-ODBC (tipo1)
… url = "jdbc:odbc:airdb";
// airdb è il DSN (Data Source Name) di solito si crea nei
   sistemi Windows mediante il Pannello di Controllo.
// se il database è PostGreSQL
… url = "jdbc:postgresql://localhost:5432/airdb";

// Connection è un'interfaccia, getConnection() un metodo statico
   della classe DriverManager
Connection con = DriverManager
   .getConnection(url, "airuser", "airuser");
// lancia una SQLException
```

Connessione alternativa

```
String url = "jdbc:mysql://localhost:3306/airdb?
             user=airuser&password=airuser";

Connection con = DriverManager.getConnection(url);
// lancia una SQLException
```

Gestione delle connessioni

```
private Connection getConnection() throws SQLException {
  String url = "jdbc:mysql://localhost:3306/airdb";

  Connection connection = DriverManager.getConnection(url, "airuser", "airuser");
  System.out.println("Connessione OK \n");

  return connection;
}


private void releaseConnection(Connection connection) throws SQLException {
  if (connection != null) {
    connection.close();
    connection = null;
  }
}
```

Preparazione dell'istruzione SQL

```
// recuperiamo una connessione
Connection con = getConnection();

// l'oggetto st rappresenta l'istruzione SQL
// Statement è un'interfaccia

Statement st = con.createStatement();
// lancia una SQLException
```

- A questo punto **`st`** esiste, ma non contiene lo statement SQL da passare al DBMS.

Esecuzione della query (query di lettura)

```
String sql = "SELECT * FROM aerei";
// ResultSet è un'interfaccia
ResultSet rs = st.executeQuery(sql); //lancia una SQLException
// rs contiene le righe della tabella
```

- Il cursore (puntatore al record corrente) adesso è posizionato prima della prima riga.

- Per spostare il cursore in avanti, indietro …, possiamo usare i seguenti metodi (booleani):
  - **`next()`**, **`previous()`**, **`first()`**, **`last()`**, **`beforeFirst()`**, **`afterLast()`**,…

Elaborazione dei campi

```
while (rs.next()) {
   // modello è il nome della terza colonna della tabella
   String modello = rs.getString("modello");
   // posti è il nome della quinta colonna della tabella
   int posti = rs.getInt("numposti");



   String produttore = rs.getString(2);

   // elaborazione dei campi
   System.out.printf("%s %s %d\n",
                modello, produttore, posti);
} // fine while
```

ResultSet.getInt restituisce 0 quando il valore del campo è NULL

```
rs.getInt("numposti");
if (rs.wasNull()) {
   // gestire il valore del campo NULL
}
```

Chiusura degli oggetti

```
// gli oggetti vanno chiusi correttamente
  nell'ordine inverso a quello di apertura

rs.close(); // ResultSet, lancia una SQLException
st.close(); // Statement, lancia una SQLException
con.close(); // Connection, lancia una SQLException
```

Si può usare anche `relaseConnection(con)`

- L' apertura di una connessione DB è un'operazione costosa in termini di performance:
  - È necessario utilizzare un *ConnectionPool* per condividere le connessioni tra le diverse richieste.
  - In questo caso la Connection non deve essere chiusa.

Chiudere la connessione

```
try {
   Connection con = getConnection();

   Statement st = con.createStatement();
   //...
   ResultSet rs = st.executeQuery("SELECT ... ");
   while (rs.next()) {
        //...
   }
   if (rs != null) rs.close();
   if (st != null) st.close();
}
catch (SQLException ex) {
// gestisci l'eccezione
} finally {
   //chiude sempre la connessione
   if (con != null) con.close();
}
```

Aggiungere un try…catch per gestire la SQLException

DBConnectionPool

```
import java.sql.Connection;
import java.sql.DriverManager;
import java.sql.SQLException;
import java.util.LinkedList;
import java.util.List;
```

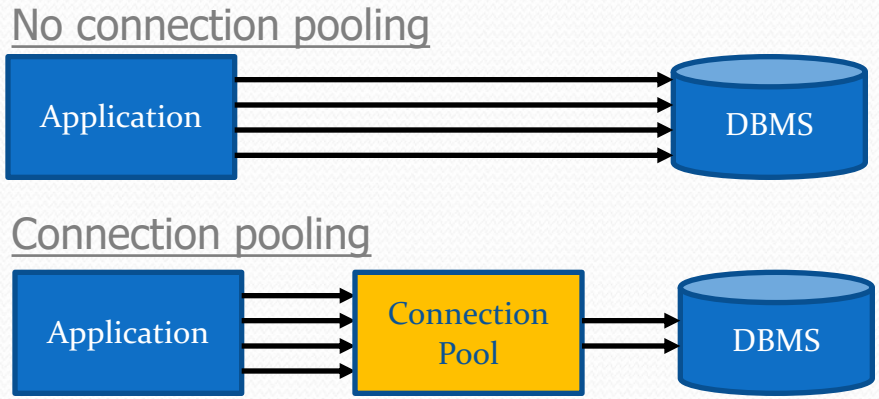
```
public class DBConnectionPool {

  private static List<Connection> freeDbConnections;

  static {
    freeDbConnections = new LinkedList<Connection>();

    try {
        Class.forName("com.mysql.cj.jdbc.Driver");
    } catch (ClassNotFoundException e) {
        System.err.println("DB driver not found!”+ e);
        System.err.println(e);
    }

  }
…
```



```
…

private static Connection createDBConnection() throws SQLException {
        Connection newConnection = null;
        String ip = "localhost";
        String port = "3306";
        String db = "airdb";
        String username = "airuser";
        String password = "airuser";
        String params = "";

        newConnection = DriverManager.getConnection(
                "jdbc:mysql://"+ ip +":"+ port +"/"+ db + params,
                username, password);

        newConnection.setAutoCommit(false);

        return newConnection;
}
…
```

Se disattivato l'autoCommit bisogna manualmente eseguire il commit delle transazioni.

```
…
public static synchronized Connection getConnection() throws SQLException {
        Connection connection;

        if (! freeDbConnections.isEmpty()) {
          connection = (Connection) freeDbConnections.get(0);
          DBConnectionPool.freeDbConnections.remove(0);

          try {
            if (connection.isClosed())
              connection = DBConnectionPool.getConnection();
          } catch (SQLException e) {
            if(connection != null)
              connection.close();

            connection = DBConnectionPool.getConnection();
          }
        } else { connection = DBConnectionPool.createDBConnection(); }
        return connection;
}

public static synchronized void releaseConnection(Connection connection) {
        DBConnectionPool.freeDbConnections.add(connection);
}
```

Usare la *ConnectionPool*

```
Connection con = null;
Statement st = null;
ResultSet rs = null;
try {
  con = DBConnectionPool.getConnection();
  st = con.createStatement();
  rs = st.executeQuery("SELECT ... ");

  while (rs.next()) {
    //...
  }
} catch(SQLException s) {
  //gestisci l'eccezione
} finally {
  try {
    if(rs != null) rs.close();
    if(st != null) st.close();
    DBConnectionPool.releaseConnection(con);
  } catch(SQLException s) {
    //gestisci l'eccezione
  }
}
```

Aggiornamento dei record

- Se si vuole effettuare un aggiornamento dei record con le istruzioni SQL: *INSERT, UPDATE, DELETE e CREATE*, si usa il metodo **`executeUpdate`** che restituisce il numero dei record aggiornati in caso di successo.

```
// elimina i record che soddisfano il criterio specificato
String usql = "DELETE FROM aerei WHERE id ='f4f4f'";
int result = st.executeUpdate(usql);
                              // lancia una SQLException
if (result > 0) {
        // tutto ok...
} else {
        // impossibile cancellare il/i record
}
```

Creare una tabella

```
Statement st = con.createStatement();

int n = st.executeUpdate("CREATE TABLE aerei_cr " +
  "(id CHAR(20) primary key, " +
  "produttore CHAR(20) not null, " +
  "modello CHAR(10) not null, " +
  "dataimm DATE, " +
  "numposti INT) " );
```

- Notare che se il valore di ritorno di executeUpdate è 0, significa:
  1. il comando eseguito ha modificato zero righe, o
  2. il comando eseguito è uno statement DDL.

Inserimento di dati

```
Statement st = con.createStatement();

int n = st.executeUpdate("INSERT INTO aerei " +
  "VALUES ('f4f4f', 'concorde', 'AZ12', '1990-1-15', 120)");
// n è uguale ad 1,
// tuttavia dipende dal numeri di tuple inserite

System.out.println(n);
```

Aggiornare i dati

```
String updateString = "UPDATE aerei " +
  "SET numposti = numposti * 0.8 " +
  "WHERE numposti > 100";

Statement st = con.createStatement();

int resultUpdate = st.executeUpdate(updateString);

System.out.println(resultUpdate);
```

Query parametriche

```
// il ? rappresenta il parametro
String sql = "SELECT * FROM aerei WHERE produttore = ?";
String produttore = "boeing";
// interfaccia PreparedStatement
PreparedStatement ps = con.prepareStatement(sql);
// associamo al (primo e unico) parametro la stringa
  produttore
ps.setString(1, produttore);
// eseguiamo la query
ResultSet rs = ps.executeQuery();
// nessun argomento per il metodo executeQuery, attenzione!
// si possono scorrere i record
while (rs.next()) {
//lettura ed elaborazione dei record
// ...
}
```

Query parametriche (2)

```
// il ? rappresenta il parametro
String sql = "SELECT * FROM aerei WHERE produttore = ?
            AND ( numposti > ? OR numposti IS NULL) ";
String produttore = "boeing";
int posti = 100;
// interfaccia PreparedStatement
PreparedStatement ps = con.prepareStatement(sql);
// associamo ai parametri i valori
ps.setString(1, produttore);
ps.setInt(2, posti);
// eseguiamo la query
ResultSet rs = ps.executeQuery();
//…
```

Procedure

- call procdura(?, ?)

Informazioni sulle eccezioni

```
try {
// codice che genera l'eccezione
} catch(SQLException ex) {
  System.out.println("Info sulla SQLException:\n");
  while (ex != null) {
   System.out.println("Message:" + ex.getMessage ());
   System.out.println("SQLState:" + ex.getSQLState ());
   System.out.println("ErrorCode:" + ex.getErrorCode ());

    ex = ex.getNextException();
  }

}
```

Informazioni sui warning

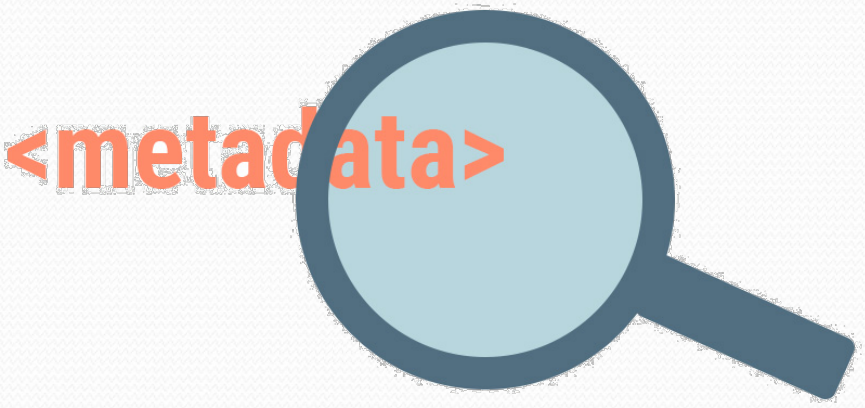
```
Statement st = con.createStatement();
ResultSet rs = st.executeQuery("SELECT/UPDATE/INSERT ...");

SQLWarning warning = st.getWarnings();

if (warning != null) {
  System.out.println("Info sui Warning:\n");
  while (warning != null) {
    System.out.println("Message:" + warning.getMessage());
    System.out.println("SQLState:" + warning.getSQLState());
    System.out.println("Code:" + warning.getErrorCode());

    warning = warning.getNextWarning();
  }
}
```

Advanced JDBC



Creating a Scrollable Result Set

- You need to create a ResultSet object that is scrollable:

```
Statement stmt = con.createStatement(
    ResultSet.TYPE_SCROLL_INSENSITIVE,
    ResultSet.CONCUR_READ_ONLY);
```

  - `TYPE_FORWARD_ONLY` *: creates a non-scrollable result set, (the cursor moves only forward).
  - `TYPE_SCROLL_INSENSITIVE`: does not reflect changes.
  - `TYPE_SCROLL_SENSITIVE`: reflects changes (by others).

- The following line checks whether the ResultSet object srs is scrollable:

```
int type = srs.getType();
```

* = default

Creating a Scrollable Result Set (2)

```
Statement stmt = con.createStatement(
    ResultSet.TYPE_SCROLL_INSENSITIVE,
    ResultSet.CONCUR_READ_ONLY);
```

- `CONCUR_READ_ONLY` *: result set is read-only.
- `CONCUR_UPDATABLE`: result set is updatable.

Creating a Scrollable Result Set (3)

```
ResultSet srs = stmt.executeQuery(
  "SELECT produttore, modello, numposti FROM aerei");

//...
srs.afterLast();
while (srs.previous()) {
  String prod = srs.getString("produttore");
  int posti = srs.getInt("numposti");
  System.out.println(prod+ " " + posti);
}
```

Getting the Cursor Position

```
srs.absolute(4);
int rowNum = srs.getRow(); // rowNum should be 4

srs.relative(-3);
rowNum = srs.getRow(); // rowNum should be 1

srs.relative(2);
rowNum = srs.getRow(); // rowNum should be 3
```

Getting the Cursor Position (2)

- Four additional methods let you verify whether the cursor is at a particular position.
- The position is stated in the method names: **`isFirst`**, **`isLast`**, **`isBeforeFirst`**, **`isAfterLast`**.
- These methods all return a boolean and can therefore be used in a conditional statement:

```
if (!srs.isAfterLast()) {
  String name = srs.getString("produttore");
  int posti = srs.getInt("numposti");
  System.out.println(prod + " " + posti);
}
```

Creating an Updatable Result Set

```
Statement stmt = con.createStatement(
  ResultSet.TYPE_SCROLL_SENSITIVE,
  ResultSet.CONCUR_UPDATABLE);

ResultSet usrs = stmt.executeQuery(
  "SELECT produttore, modello, numposti FROM aerei");
```

- The following line of code checks whether the ResultSet object is updatable:

```
int concurrency = usrs.getConcurrency();
```

Creating an Updatable Result Set (2)

```
usrs.last();
usrs.updateInt("numposti", 125);
usrs.updateRow();
usrs.beforeFirst();
```

- The following code fragment makes an update and then cancels it:

```
usrs.last();
usrs.updateInt("numposti", 125);
//...
usrs.cancelRowUpdates();
```

Inserting and Deleting Rows

```
Statement stmt = con.createStatement(
  ResultSet.TYPE_SCROLL_SENSITIVE,
  ResultSet.CONCUR_UPDATABLE);

ResultSet uprs = stmt.executeQuery("SELECT * FROM aerei");
//...
uprs.moveToInsertRow();
uprs.updateString("id", "f2ff3");
uprs.updateString("produttore", "concorde");
uprs.updateString("modello", "AZ21");
uprs.updateString("dataimm", "2004-01-24");
uprs.updateInt("numposti", 75);
uprs.insertRow();
```

Using Transactions

```
con.setAutoCommit(false);  // disable Auto-commit Mode

PreparedStatement updatePlaces = con.prepareStatement(
        "UPDATE aerei SET numposti = ? WHERE produttore = ?");
updateSales.setInt(1, 50);
updateSales.setString(2, “boeing");
updateSales.executeUpdate();
//...

PreparedStatement updateTotal = con.prepareStatement(
        "UPDATE aerei SET numposti = numposti+ ? WHERE modello= ?");
updateTotal.setInt(1, 10);
updateTotal.setString(2, “AZ12");
updateTotal.executeUpdate();

con.commit();  // commit a transaction

con.setAutoCommit(true);  // enable Auto-commit Mode
```

Using Transactions to Preserve Data Integrity

- In addition to grouping statements together for execution as a unit, transactions can help to preserve the integrity of the data in a table.

- After inserting the outdated data, the user realizes that they are no longer valid and calls the Connection method **`rollback()`** to undo their effects.
  - The method rollback aborts a transaction and restores values to what they were before the attempted update.

Using *Statement* Objects for Batch Updates

```
con.setAutoCommit(false);
Statement stmt = con.createStatement();
stmt.addBatch(
  "INSERT INTO aerei VALUES (’78fs2', 'concorde', 'AZ22', '1992-3-15', 125) ");
stmt.addBatch(
  "INSERT INTO aerei VALUES (’79fs3', 'concorde', 'AZ22', '1993-6-20', 125) ");
stmt.addBatch(
  "INSERT INTO aerei VALUES (’80fs3', ’boing', ’BG33', '1995-9-02', 150) ");
stmt.addBatch(
  "INSERT INTO aerei VALUES (’81fs4', ’boing', ’BG39', '1998-3-15', 175) ");

int []  updateCounts = stmt.executeBatch();

con.commit();
con.setAutoCommit(true);
```

Batch Update Exceptions

```
try {
  // make some updates
} catch(BatchUpdateException b)  {
  System.err.println("BatchUpdateException");
  System.err.println("SQLState: " + b.getSQLState());
  System.err.println("Message: " + b.getMessage());
  System.err.println(”Code: " + b.getErrorCode());
  System.err.print("Update counts: ");
  int [] updateCounts = b.getUpdateCounts();
  for (int i = 0; i < updateCounts.length; i++) {
       System.err.print(updateCounts[i] + " ");
  }
}
```

Creating a *Date* Object

```
Calendar cal = Calendar.getInstance();
  cal.set(Calendar.YEAR, 2020);
  cal.set(Calendar.MONTH, Calendar.DECEMBER);
  cal.set(Calendar.DATE, 16);
  cal.set(Calendar.HOUR, 0);
  cal.set(Calendar.MINUTE, 0);
  cal.set(Calendar.SECOND, 0);
  cal.set(Calendar.MILLISECOND, 0);
  java.sql.Date d = new java.sql.Date(cal.getTimeInMillis());
  System.out.println("Date:" + d);
```

```
java.sql.Date d2 = java.sql.Date.valueOf("1999-05-31");
```

Date '0000-00-00'

- java.sql.SQLException: Value '0000-00-00' can not be represented as java.sql.Date

```
String url = "jdbc:mysql://localhost:3306/airdb?
             zeroDateTimeBehavior=CONVERT_TO_NULL";
```

- exception → Throws an SQLException
- round → '0001-01-01'
- convertToNull → null

serverTimezone=UTC
useLegacyDatetimeCode=false
useUnicode=true
useJDBCCompliantTimezoneShift=true
zeroDateTimeBehavior=CONVERT_TO_NULL
EXCEPTION
ROUND
autoReconnect=true
useSSL=false

Using a ResultSetMetaData Object

- You can get information about the columns in this ResultSet object by creating a ResultSetMetaData object and invoking ResultSetMetaData methods on it.

```
Statement stmt = con.createStatement();
ResultSet rs = stmt.executeQuery(
      "SELECT * FROM aerei");

ResultSetMetaData rsmd = rs.getMetaData();
//...
```

Using the Method getColumnCount

```
Statement stmt = con.createStatement();
ResultSet rs = stmt.executeQuery("SELECT * FROM aerei");

ResultSetMetaData rsmd = rs.getMetaData();
int numberOfColumns = rsmd.getColumnCount();
for (int i = 1; i <= numberOfColumns; i++) {
       String columnName = rsmd.getColumnLabel(i);
       System.out.print(columnName);
}
while (rs.next ()) {
  for (int i=1; i<=numberOfColumns; i++) {
       String columnValue = rs.getString(i);
       System.out.print(columnValue);
  }
}
stmt.close();
```

Getting Column Type Information

```
ResultSetMetaData rsmd = rs.getMetaData();

int columns = rsmd.getColumnCount();
String tableName = rsmd.getTableName(1);
System.out.println("Table: " + tableName);

for (int i = 1; i <= columns; i++) {
  String colLabel = rsmd.getColumnLabel(i);
  String colName = rsmd.getColumnName(i);
  int jdbcType = rsmd.getColumnType(i);
  String type= rsmd.getColumnTypeName(i);
  System.out.print(colName + " " +colLabel+ " of type "+
                   type + " (" + jdbcType+ ")");
}
```

Print a table

```
Statement stmt = con.createStatement();
ResultSet rs = stmt.executeQuery("SELECT * FROM aerei");
ResultSetMetaData md = rs.getMetaData();

//print the column labels
for( int i = 1; i <= md.getColumnCount(); i++ )
        System.out.print( md.getColumnLabel(i) + " " );
System.out.println();

//loop through the result set
while( rs.next() ) {
        for( int i = 1; i <= md.getColumnCount(); i++ )
                System.out.print( rs.getString(i) + " " );
        System.out.println();
}

if(rs != null) rs.close();
if(rs != null) stmt.close();
```

Using a DatabaseMetaData Object

- The interface DatabaseMetaData has over 150 methods for getting information about a database or DBMS.
- Once you have an open connection with a DBMS, you can create a DatabaseMetaData object that contains information about that database system.

```
DatabaseMetaData dbmd = con.getMetaData();
\\...
```

Available types

```
ResultSet rsm = dbmd.getTypeInfo();

while (rsm.next()) {
  String typeName = rsm.getString("TYPE_NAME");
  short dataType = rsm.getShort("DATA_TYPE");
  String createParams = rsm.getString("CREATE_PARAMS");
  int nullable = rsm.getInt("NULLABLE");
  boolean caseSensitive = rsm.getBoolean("CASE_SENSITIVE");

  System.out.println("DBMS type " + typeName);
  System.out.println("java.sql.Types: "  + dataType);
  System.out.println("parameters: " + createParams);
  System.out.println("nullable?: "  + nullable);
  System.out.println("case sensitive?: " + caseSensitive);
}
```

Gestione menu

```
import java.io.*;

public class EsempioMenu {
public static void main(String args[]) throws Exception {//inizio main

   InputStreamReader keyIS;
   BufferedReader keyBR;
   int i = 0;
   String scelta;

   keyIS = new InputStreamReader(System.in);
   keyBR = new BufferedReader(keyIS);

   while (i != 1000) {//inizio while
         System.out.println(”Operazione:");
         System.out.println("1, Visualizza aerei");
         System.out.println("2, Inserimento aereo");
         System.out.println("3, Cancella aereo");
         System.out.println("1000, Per uscire");

         System.out.print("Inserisci scelta: ");
         scelta = keyBR.readLine();
…
```

…

```
        try {//inizio try-catch
          i = Integer.parseInt(scelta);
         }
         catch(NumberFormatException e) {
          i = 999;
         } //fine try-catch
        switch (i) {
            case 1: { System.out.println("Visualizza aerei"); break;
            }
            case 2: { System.out.println("Inserimento aereo"); break;
            }
            case 3: { System.out.println("Cancella aereo"); break;
            }
            case 1000: { System.out.println("Uscita"); break;
            }
            default: { System.out.println("Scelta non presente");
            }
        } //fine switch
   } //fine while
 } //fine main
} //fine EsempioMenu
```