Database Connectivity

Enterprise Application Development

Sanjay Goel

Topics

- Overview
  - JDBC
  - Types of Drivers
  - API
- Connecting to Databases
- Executing Queries & Retrieving Results
- Advanced Topics
  - Prepared Statements
  - Connection Pooling
- Assignment

Overview

JDBC

Definition

- JDBC: Java Database Connectivity
  - It provides a standard library for Java programs to connect to a database and send it commands using SQL
  - It generalizes common database access functions into a set of common classes and methods
  - Abstracts vendor specific details into a code library making the connectivity to multiple databases transparent to user
- JDBC API Standardizes:
  - Way to establish connection to database
  - Approach to initiating queries
  - Method to create stored procedures
  - Data structure of the query result

JDBC

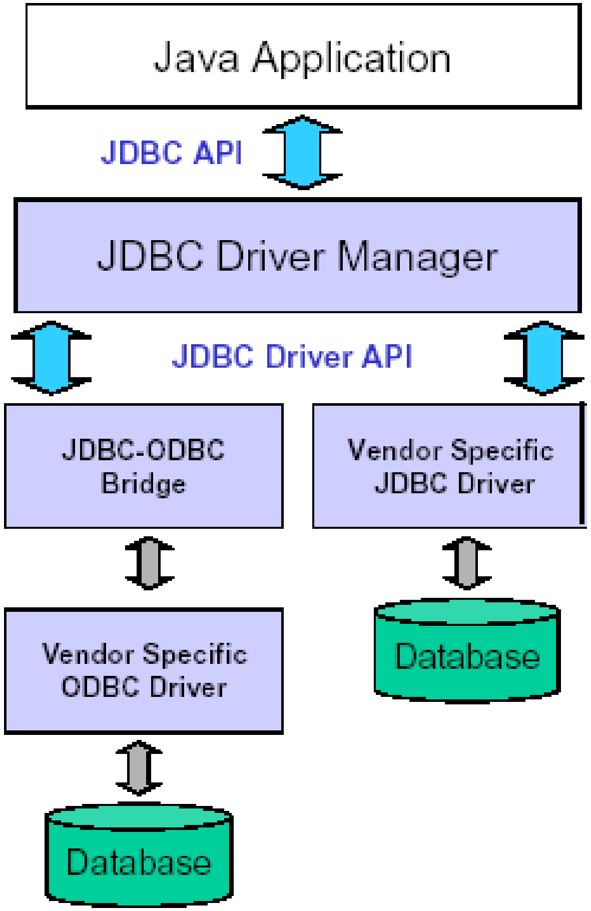
API

- Two main packages java.sql and javax.sql
  - **Java.sql** contains all core classes required for accessing database (Part of Java 2 SDK, Standard Edition)
  - **Javax.sql** contains optional features in the JDBC 2.0 API (part of Java 2 SDK, Enterprise Edition)
- Javax.sql adds functionality for enterprise applications
  - DataSources
  - JNDI
  - Connection Pooling
  - Rowsets
  - Distributed Transactions

JDBC

Architecture

- JDBC Consists of two parts:
  - JDBC API, a purely Java-based API
  - JDBC Driver Manager, which communicates with vendor-specific drivers that perform the real communication with the database
- Translation to the vendor format occurs on the client
  - No changes needed to the server
  - Driver (translator) needed on client



JDBC

Drivers

- JDBC uses drivers to translate generalized JDBC calls into vendor-specific database calls
  - Drivers exist for most popular databases
  - Four Classes of JDBC drivers exist

    Type I

    Type II

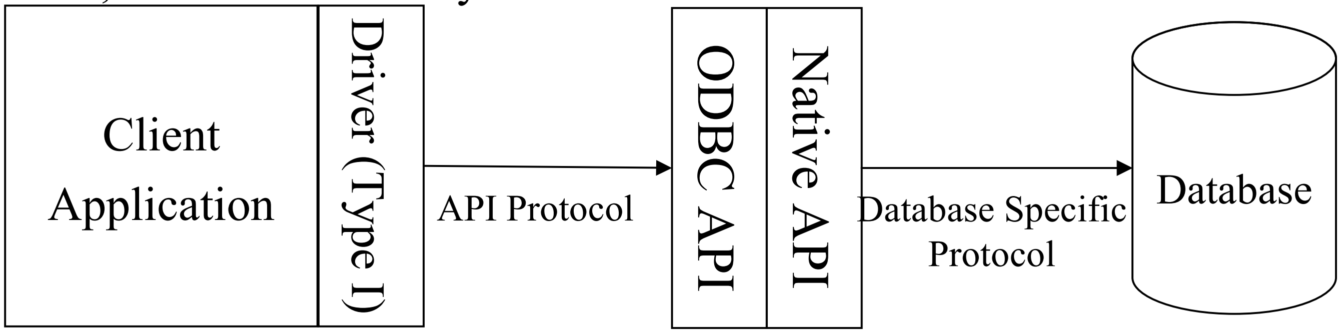
    Type III

    Type IV

JDBC

Drivers (Type I)

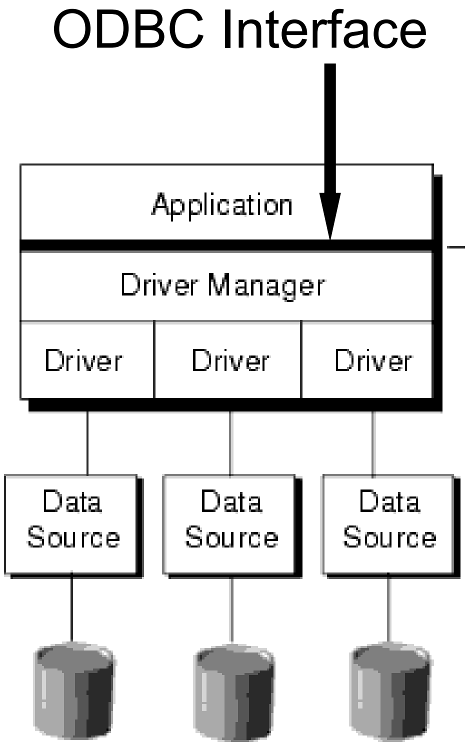
- Type I driver provides mapping between JDBC and access API of a database
  - The access API calls the native API of the database to establish communication
- A common Type I driver defines a JDBC to ODBC bridge
  - ODBC is the database connectivity for databases
  - JDBC driver translates JDBC calls to corresponding ODBC calls
  - Thus if ODBC driver exists for a database this bridge can be used to communicate with the database from a Java application
- Inefficient and narrow solution
  - Inefficient, because it goes through multiple layers
  - Narrow, since functionality of JDBC code limited to whatever ODBC supports



JDBC

Open Database Connectivity (ODBC)

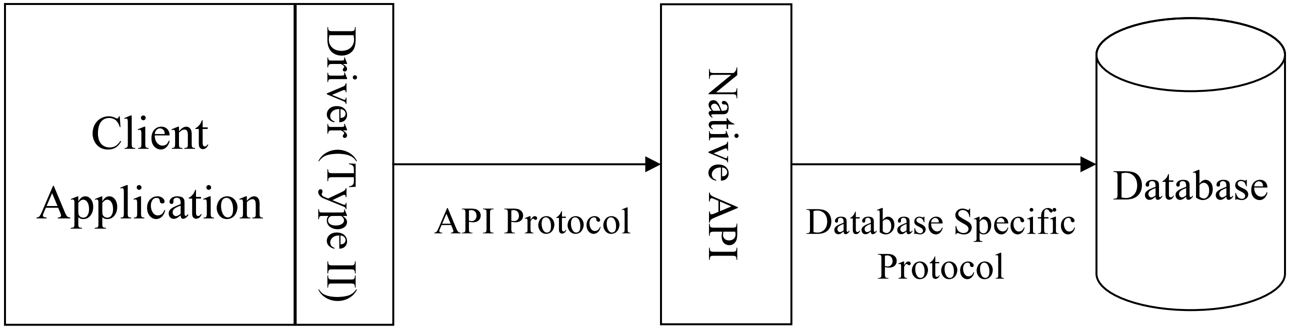
- A standard database access method developed by the SQL Access group in 1992.
  - The goal of ODBC is to make it possible to access any data from any application, regardless of which database management system (DBMS) is handling the data.
  - ODBC manages this by inserting a middle layer, called a database *driver* , between an application and the DBMS.
  - The purpose of this layer is to translate the application's data queries into commands that the DBMS understands.
  - For this to work, both the application and the DBMS must be *ODBC-compliant*, that is, the application must be capable of issuing ODBC commands and the DBMS must be capable of responding to them.



JDBC

Drivers (Type II)

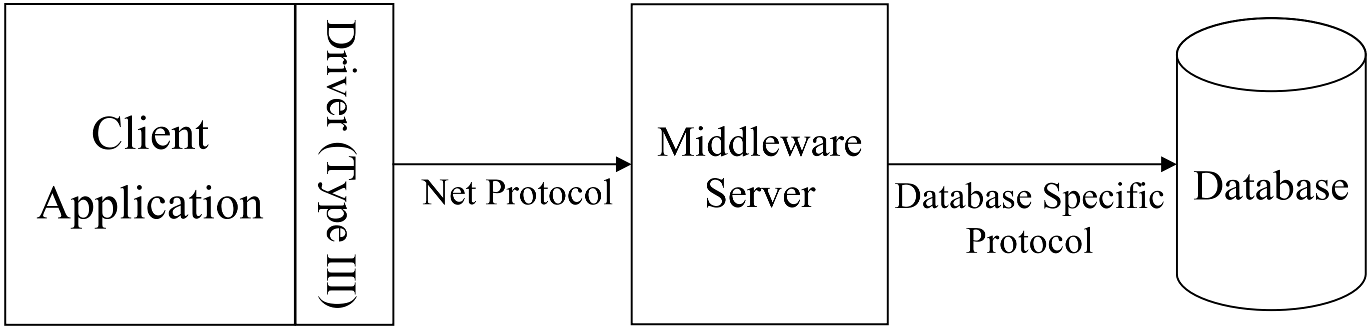
- Type II driver communicates directly with native API
  - Type II makes calls directly to the native API calls
  - More efficient since there is one less layer to contend with (i.e. no ODBC)
  - It is dependent on the existence of a native API for a database



JDBC

Drivers (Type III)

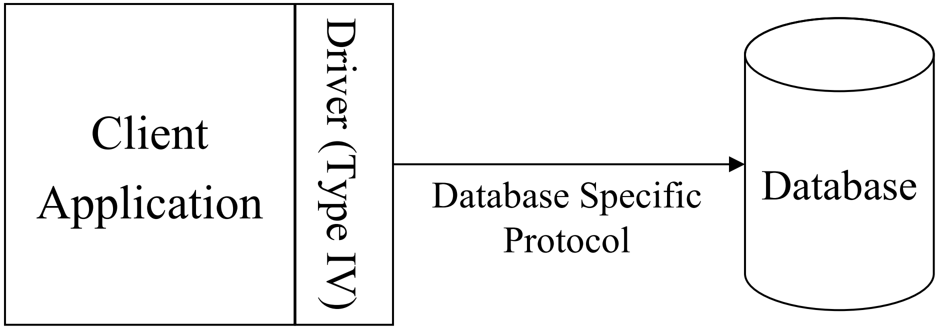
- Type III driver make calls to a middleware component running on another server
  - This communication uses a database independent net protocol
  - Middleware server then makes calls to the database using database-specific protocol
  - The program sends JDBC call through the JDBC driver to the middle tier
  - Middle-tier may use Type I or II JDBC driver to communicate with the database.



JDBC

Drivers (Type IV)

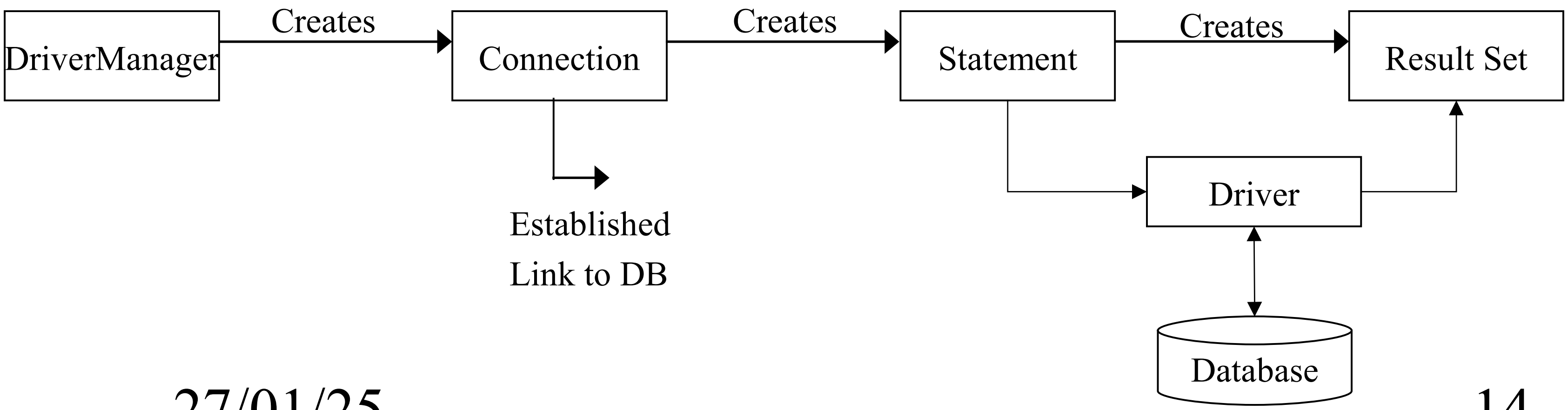
- Type IV driver is an all-Java driver that is also called a thin driver
  - It issues requests directly to the database using its native protocol
  - It can be used directly on platform with a JVM
  - Most efficient since requests only go through one layer
  - Simplest to deploy since no additional libraries or middle-ware



Connecting to Database

JDBC
Conceptual Components

- **Driver Manager:** Loads database drivers and manages connections between the application and the driver
- **Driver:** Translates API calls into operations for specific database
- **Connection:** Session between application and data source
- **Statement:** SQL statement to perform query or update
- **Metadata:** Information about returned data, database, & driver
- **Result Set:** Logical set of columns and rows of data returned by executing a statement



JDBC

Basic Steps

- Import the necessary classes
- Load the JDBC driver
- Identify the data source (Define the Connection URL)
- Establish the Connection
- Create a Statement Object
- Execute query string using Statement Object
- Retrieve data from the returned ResultSet Object
- Close ResultSet & Statement & Connection Object in order

JDBC

Driver Manager

- DriverManager provides a common access layer on top of different database drivers
  - Responsible for managing the JDBC drivers available to an application
  - Hands out connections to the client code
- Maintains reference to each driver
  - Checks with each driver to determine if it can handle the specified URL
  - The first suitable driver located is used to create a connection
- DriverManager class can not be instantiated
  - All methods of DriverManager are static
  - Constructor is private

JDBC Driver

Loading

- Required prior to communication with a database using JDBC
- It can be loaded
  - dynamically using Class.forName(String *drivername*)
  - System Automatically loads driver using jdbc.drivers system property
- An instance of driver must be registered with DriverManager class
- Each Driver class will typically
  - create an instance of itself and register itself with the driver manager
  - Register that instance automatically by calling RegisterDriver method of the DriverManager class
- Thus the code does not need to create an instance of the class or register explicitly using registerDriver(Driver) class

JDBC Driver

Loading: class.forName()

- Using forName(String) from java.lang.Class instructs the JVM to find, load and link the class identified by the String

```
e.g try {
        Class.forName(“COM.cloudscape.core.JDBCDriver”);
        } catch (ClassNotFoundException e) {
            System.out.println(“Driver not found”);
            e.printStackTrace();
        }
```

- At run time the class loader locates the driver class and loads it
  - All static initializations during this loading
  - Note that the name of the driver is a literal string thus the driver does not need to be present at compile time

JDBC Driver

Loading: System Property

- Put the driver name into the jdbc drivers System property
  - When a code calls one of the methods of the driver manager, the driver manager looks for the jdbc.drivers property
  - If the driver is found it is loaded by the Driver Manager
  - Multiple drivers can be specified in the property
  - Each driver is listed by full package specification and class name
  - a colon is used as the delimiter between the each driver

  e.g jdbc.drivers=com.pointbase.jdbc.jdbcUniversalDriver
- For specifying the property on the command line use:
  - java -Djdbc.drivers=com.pointbase.jdbc.jdbcUniversalDriver MyApp
- A list of drivers can also be provided using the Properties file
  - System.setProperty("jdbc.drivers", "COM.cloudscape.core.JDBCDriver");
  - DriverManager only loads classes once so the system property must be set prior to the any DriverManager method being called.

JDBC

URLs

- JDBC Urls provide a way to identify a database
- Syntax:

  <protocol>:<subprotocol>:<protocol>

  - Protocol: Protocol used to access database (jdbc here)
  - Subprotocol: Identifies the database driver
  - Subname: Name of the resource
- Example
  - Jdbc:cloudscape:Movies
  - Jdbc:odbc:Movies

Connection

Creation

- Required to communicate with a database via JDBC
- Three separate methods:

```
public static Connection getConnection(String url)
public static Connection getConnection(String url, Properties info)
public static Connection getConnection(String url, String user, String password)
```

- Code Example (Access)

```
try {// Load the driver class
    System.out.println("Loading Class driver");
    Class.forName("sun.jdbc.odbc.JdbcOdbcDriver");
    // Define the data source for the driver
    String sourceURL = "jdbc:odbc:music”;
    // Create a connection through the DriverManager class
    System.out.println("Getting Connection");
    Connection databaseConnection = DriverManager.getConnection(sourceURL);
    }
catch (ClassNotFoundException cnfe) {
          System.err.println(cnfe); }
 catch (SQLException sqle) {
          System.err.println(sqle);}
```

Connection

Creation

- Code Example (Oracle)

```
try {
        Class.forName("oracle.jdbc.driver.OracleDriver");
        String sourceURL = "jdbc:oracle:thin:@delilah.bus.albany.edu:1521:databasename";
        String user = "goel";
        String password = "password";
        Connection databaseConnection=DriverManager.getConnection(sourceURL,user,
    password );
        System.out.println("Connected Connection"); }
catch (ClassNotFoundException cnfe) {
        System.err.println(cnfe); }
catch (SQLException sqle) {
        System.err.println(sqle);}
```

Connection
Closing

- Each machine has a limited number of connections (separate thread)
  - If connections are not closed the system will run out of resources and freeze
  - Syntax: public void close() throws SQLException

- Naïve Way:

```
try {
      Connection conn
      = DriverManager.getConnection(url);
       // Jdbc Code
      …
      } catch (SQLException sqle) {
         sqle.printStackTrace();
      }
      conn.close();
```

- SQL exception in the Jdbc code will prevent execution to reach conn.close()

- Correct way (Use the finally clause)

```
try{
Connection conn =
     Driver.Manager.getConnection(url);
     // JDBC Code
     } catch (SQLException sqle) {
         sqle.printStackTrace();
     } finally {
         try {
              conn.close();
         } catch (Exception e) {
              e.printStackTrace();
         }
     }
```

Statement

Types

- Statements in JDBC abstract the SQL statements
- Primary interface to the tables in the database
- Used to create, retrieve, update & delete data (CRUD) from a table
  - Syntax: Statement statement = connection.createStatement();
- Three types of statements each reflecting a specific SQL statements
  - Statement
  - PreparedStatement
  - CallableStatement

Statement

Syntax

- Statement used to send SQL commands to the database
  - Case 1: ResultSet is non-scrollable and non-updateable

    ```
    public Statement createStatement() throws SQLException
    Statement statement = connection.createStatement();
    ```

  - Case 2: ResultSet is non-scrollable and/or non-updateable

    ```
    public Statement createStatement(int, int) throws SQLException
    Statement statement = connection.createStatement();
    ```

  - Case 3: ResultSet is non-scrollable and/or non-updateable and/or holdable

    ```
    public Statement createStatement(int, int, int) throws SQLException
    Statement statement = connection.createStatement();
    ```

- PreparedStatement

  ```
  public PreparedStatement prepareStatement(String sql) throws SQLException
  PreparedStatement pstatement = prepareStatement(sqlString);
  ```

- CallableStatement used to call stored procedures

  ```
  public CallableStatement prepareCall(String sql) throws SQLException
  ```

Statement

Release

- Statement can be used multiple times for sending a query
- It should be released when it is no longer required
  - Statement.close():
  - It releases the JDBC resources immediately instead of waiting for the statement to close automatically via garbage collection
- Garbage collection is done when an object is unreachable
  - An object is reachable if there is a chain of reference that reaches the object from some root reference
- Closing of the statement should be in the finally clause

```
try{
    Connection conn =
    Driver.Manager.getConnection(url
    );
    Statement stmt =
    conn.getStatement();
    // JDBC Code
    } catch (SQLException sqle) {
    sqle.printStackTrace();
    } finally {
      try {stmt.close();
            conn.close();
       } catch (Exception e) {
             e.printStackTrace();
       }
    }
```

JDBC

Logging

- DriverManager provides methods for managing output
  - DriverManagers debug output can be directed to a printwriter

    public static void setLogWriter(PrintWriter pw)
  - PrintWriter can be wrapped for any writer or OutputStream
  - Debug statements from the code can be sent to the log as well.

    public static void println(String s)
- Code

```
FileWriter fw = new FileWriter(“mydebug.log”);
PrintWriter pw = new PrintWriter(fw);
// Set the debug messages from Driver manager to pw
DriverManager.setLogWriter(pw);
// Send in your own debug messages to pw
DriverManager.println(“The name of the database is “ + databasename);
```

Querying the Database

Executing Queries

Methods

- Two primary methods in statement interface used for executing Queries
  - executeQuery Used to retrieve data from a database
  - executeUpdate: Used for creating, updating & deleting data
- executeQuery used to retrieve data from database
  - Primarily uses Select commands
- executeUpdate used for creating, updating & deleting data
  - SQL should contain Update, Insert or Delete commands
- Uset setQueryTimeout to specify a maximum delay to wait for results

Executing Queries

Data Definition Language (DDL)

- Data definition language queries use executeUpdate
- Syntax: int executeUpdate(String sqlString) throws SQLException
  - It returns an integer which is the number of rows updated
  - sqlString should be a valid String else an exception is thrown
- Example 1: Create a new table

```
Statement statement = connection.createStatement();
String sqlString =
“Create Table Catalog”
+ “(Title Varchar(256) Primary Key Not Null,”+
+ “LeadActor Varchar(256) Not Null, LeadActress Varchar(256) Not Null,”
+ “Type Varchar(20) Not Null, ReleaseDate Date Not NULL )”;
Statement.executeUpdate(sqlString);
```

  - executeUpdate returns a zero since no row is updated

Executing Queries

DDL (Example)

- Example 2: Update table

```
Statement statement = connection.createStatement();
String sqlString =
 “Insert into Catalog”
+ “(Title, LeadActor, LeadActress, Type, ReleaseDate)”
+ “Values(‘Gone With The Wind’, ‘Clark Gable’, ‘Vivien Liegh’,”
+ “’Romantic’, ‘02/18/2003’ ”
Statement.executeUpdate(sqlString);
```

  - executeUpdate returns a 1 since one row is added

Executing Queries

Data Manipulation Language (DML)

- Data definition language queries use executeQuery
- Syntax

  ResultSet executeQuery(String sqlString) throws SQLException

  – It returns a ResultSet object which contains the results of the Query
- Example 1: Query a table

```
Statement statement = connection.createStatement();
String sqlString = “Select Catalog.Title, Catalog.LeadActor, Catalog.LeadActress,” +
                   “Catalog.Type, Catalog.ReleaseDate From Catalog”;
ResultSet rs = statement.executeQuery(sqlString);
```

ResultSet

Definition

- ResultSet contains the results of the database query that are returned
- Allows the program to scroll through each row and read all columns of data
- ResultSet provides various access methods that take a column index or column name and returns the data
  - All methods may not be applicable to all resultsets depending on the method of creation of the statement.
- When the executeQuery method returns the ResultSet the cursor is placed before the first row of the data
  - Cursor refers to the set of rows returned by a query and is positioned on the row that is being accessed
  - To move the cursor to the first row of data next() method is invoked on the resultset
  - If the next row has a data the next() results true else it returns false and the cursor moves beyond the end of the data
- First column has index 1, not 0

ResultSet

- ResultSet contains the results of the database query that are returned
- Allows the program to scroll through each row and read all the columns of the data
- ResultSet provides various access methods that take a column index or column name and returns the data
  - All methods may not be applicable to all resultsets depending on the method of creation of the statement.
- When the executeQuery method returns the ResultSet the cursor is placed before the first row of the data
  - Cursor is a database term that refers to the set of rows returned by a query
  - The cursor is positioned on the row that is being accessed
  - First column has index 1, not 0
- Depending on the data numerous functions exist
  - getShort(), getInt(), getLong()
  - getFloat(), getDouble()
  - getClob(), getBlob(),
  - getDate(), getTime(), getArray(), getString()

ResultSet

- Examples:
  - Using column Index:

    Syntax:public String getString(int columnIndex) throws SQLException

    e.g. ResultSet rs = statement.executeQuery(sqlString);

        String data = rs.getString(1)
  - Using Column name

    public String getString(String columnName) throws SQLException

    e.g. ResultSet rs = statement.executeQuery(sqlString);

        String data = rs.getString(Name)
- The ResultSet can contain multiple records.
  - To view successive records next() function is used on the ResultSet
  - Example: while(rs.next()) {
  - System.out.println(rs.getString); }

Scrollable ResultSet

- ResultSet obtained from the statement created using the no argument constructor is:
  - Type forward only (non-scrollable)
  - Not updateable
- To create a scrollable ResultSet the following statement constructor is required
  - Statement createStatement(int resultSetType, int resultSetConcurrency)
- ResultSetType determines whether it is scrollable. It can have the following values:
  - ResultSet.TYPE_FORWARD_ONLY
  - ResultSet.TYPE_SCROLL_INSENSITIVE (Unaffected by changes to underlying database)
  - ResultSet.TYPE_SCROLL_SENSITIVE (Reflects changes to underlying database)
- ResultSetConcurrency determines whether data is updateable. Its possible values are
  - CONCUR_READ_ONLY
  - CONCUR_UPDATEABLE
- Not all database drivers may support these functionalities

Scrollable ResultSet

- On a scrollable ResultSet the following commands can be used
  - boolean next(), boolean previous(), boolean first(), boolean last()
  - void afterLast(), void beforeFirst()
  - boolean isFirst(), boolean isLast(), boolean isBeforeFirst(), boolean isAfterLast()
- Example

RowSet

- ResultSets limitation is that it needs to stay connected to the data source
  - It is not serializable and can not transporting across the network
- RowSet is an interface which removes the limitation
  - It can be connected to a dataset like the ResultSet
  - It can also cache the query results and detach from the database
- RowSet is a collection of rows
- RowSet implements a custom reader for accessing any tabular data
  - Spreadsheets, Relational Tables, Files
- RowSet object can be serialized and hence sent across the network
- RowSet object can update rows while diconnected fro the data source
  - It can connect to the data source and update the data
- Three separate implementations of RowSet
  - CachedRowSet
  - JdbcRowSet
  - WebRowSet

RowSet

- RowSet is derived from the BaseRowSet
  - Has SetXXX(…) methods to supply necessary information for making connection and executing a query
- Once a RowSet gets populated by execution of a query or from some other data source its data can be manipulated or more data added
- Three separate implementations of RowSet exist
  - CachedRowSet: Disconnected from data source, scrollable & serilaizable
  - JdbcRowSet: Maintains connection to data source
  - WebRowSet: Extension of CachedRowSet that can produce representation of its contents in XML

MetaData

- Meta Data means data about data
- Two kinds of meta data in JDBC
  - Database Metadata: To look up information about the database (here)
  - ResultSet Metadata: To get the structure of data that is returned (later)
- Example
  - connection.getMetaData().getDatabaseProductName()
  - connection.getMetaData().getDatabaseProductVersion()
- Sample Code:

```
private void showInfo(String driver,String url,String user,String password,
 String table,PrintWriter out) {
        Class.forName(driver);
        Conntection con = DriverManager.getConnection(url, username, password);
        DatabaseMetaData dbMetaData = connection.getMetaData();
        String productName = dbMetaData.getDatabaseProductName();
        System.out.println("Database: " + productName);
        String productVersion = dbMetaData.getDatabaseProductVersion();
        System.out.println("Version: " + productVersion);
}
```

Source Code

Connecting to Microsoft Access

```
/**
 * The code allows a user to connect to the MS Access Database and
 * run queries on the database. A sample query execution is provided
 * in this code. This is developed to help the students get initially
 * connected to the database.
 *
 * @author Sanjay Goel
 * @company School of Business, University at Albany
 *
 * @version 1.0
 * @created April 01, 2002 - 9:05 AM
 *
 * Notes 1: Statement is an interface hence can not be instantiated
 * using new. Need to call createStatement method of connection class
 *
 * Notes 2: Use executeQuery for DML queries that return a resultset
 * e.g., SELECT and Use executeUpdate for DDL & DML which do not
 * return Result Set e.g. (Insert Update and Delete) & DDL (Create
 * Table, Drop Table, Alter Table)
 *
 * */

import java.sql.*;

public class ConnectAccess {

  /**
   * This is the main function which connects to the Access database
   * and runs a simple query
   *
   * @param String[] args - Command line arguments for the program
   * @return void
   * @exception none
   *
   */
  public static void main(String[] args) {
```

```
// Load the driver
          try {
            // Load the driver class
            Class.forName("sun.jdbc.odbc.JdbcOdbcDriver");

            // Define the data source for the driver
            String sourceURL = "jdbc:odbc:music";

            // Create a connection through the DriverManager class
            Connection databaseConnection
               = DriverManager.getConnection(sourceURL);
            System.out.println("Connected Connection");

            // Create Statement
            Statement statement = databaseConnection.createStatement();
            String queryString
               = "SELECT recordingtitle, listprice FROM recordings";

            // Execute Query
            ResultSet results = statement.executeQuery(queryString);

            // Print results
            while (results.next()){
               System.out.println(results.getString("recordingtitle") +
                                                              "\t" +
          results.getFloat("listprice"));
            }

            // Close Connection
            databaseConnection.close();
          }
          catch (ClassNotFoundException cnfe) {
            System.err.println(cnfe);
          }
          catch (SQLException sqle) {
            System.err.println(sqle);
          }

  }
}
```

Connecting to Oracle

```
/**
 * The code allows a user to connect to the ORACLE Database and run
 * queries on the database. A sample query execution is provided in
 * this code. This is developed to help the students get initially
 * connected to the database.
 *
 * @author Sanjay Goel
 * @company School of Business, University at Albany
 *
 * @version 1.0
 * @created April 01, 2002 - 9:05 AM
 *
 * Notes 1: Statement is an interface hence can not be instantiated
 * using new. Need to call createStatement method of connection class
 *
 * Notes 2: Use executeQuery for DML queries that return a resultset
 * e.g., SELECT and Use executeUpdate for DDL & DML which do not
 * return Result Set e.g. (Insert Update and Delete) & DDL (Create
 * Table, Drop Table, Alter Table)
 *
 * */

import java.sql.*;

public class ConnectOracle {

  /**
   * This is the main function which connects to the Oracle database
   * and executes a sample query
   *
   * @param String[] args - Command line arguments for the program
   * @return void
   * @exception none
   *
   */
  public static void main(String[] args) {
```

```
// Load the driver
          try {
            // Load the driver class
            Class.forName("oracle.jdbc.driver.OracleDriver");

            // Define the data source for the driver
            String sourceURL
               = "jdbc:oracle:thin:@delilah.bus.albany.edu:1521:bodb01";

            // Create a connection through the DriverManager class
            String user = "goel";
            String password = "goel";
            Connection databaseConnection
               = DriverManager.getConnection(sourceURL, user, password);
            System.out.println("Connected to Oracle");

            // Create a statement
            Statement statement = databaseConnection.createStatement();

            // Create a query String
            String sqlString = "SELECT artistid, artistname FROM
          artistsandperformers";

            // Close Connection
            databaseConnection.close();
          }
          catch (ClassNotFoundException cnfe) {
            System.err.println(cnfe);
          }
          catch (SQLException sqle) {
            System.err.println(sqle);
          }
    }
}
```

Connecting to Cloudscape

```
/**
 * The code allows a user to connect to the Cloudscape Database and
 * run queries on the database. A sample query execution is provided
 * in this code. This is developed to help the students get initially
 * connected to the database.
 *
 * @author Sanjay Goel
 * @company School of Business, University at Albany
 *
 * @version 1.0
 * @created April 01, 2002 - 9:05 AM
 *
 * Notes 1: Statement is an interface hence can not be instantiated
 * using new. Need to call createStatement method of connection class
 *
 * Notes 2: Use executeQuery for DML queries that return a resultset
 * e.g., SELECT and Use executeUpdate for DDL & DML which do not
 * return Result Set e.g. (Insert Update and Delete) & DDL (Create
 * Table, Drop Table, Alter Table)
 *
 * */

import java.sql.*;

public class ConnectCloudscape {

    public static void main(String[] args) {

                // Load the driver
                try {
                    // Load the driver class
                    Class.forName("COM.cloudscape.core.JDBCDriver");

                    // Define the data source for the driver
                    String sourceURL = "jdbc:cloudscape:Wrox4370.db";
```

```
// Create a connection through the DriverManager class
                    Connection databaseConnection =
                DriverManager.getConnection(sourceURL);
                    System.out.println("Connected Connection");

                    // Create a statement
                    Statement statement = databaseConnection.createStatement();

                    // Create an SQL statement
                    String sqlString = "SELECT artistid, artistname FROM
                artistsandperformers";

                    // Run Query
                    ResultSet results = statement.executeQuery(sqlString);

                    // Print Results
                    while(results.next()) {
                          System.out.println(results.getInt("artistid") + "\t" +

                results.getString("artistname"));
                    }

                    // Close Connection
                    databaseConnection.close();
                }
                catch (ClassNotFoundException cnfe) {
                    System.err.println(cnfe);
                }
                catch (SQLException sqle) {
                    System.err.println(sqle);
                }
    }
}
```

Prepared Statement

// code from IVOr horton

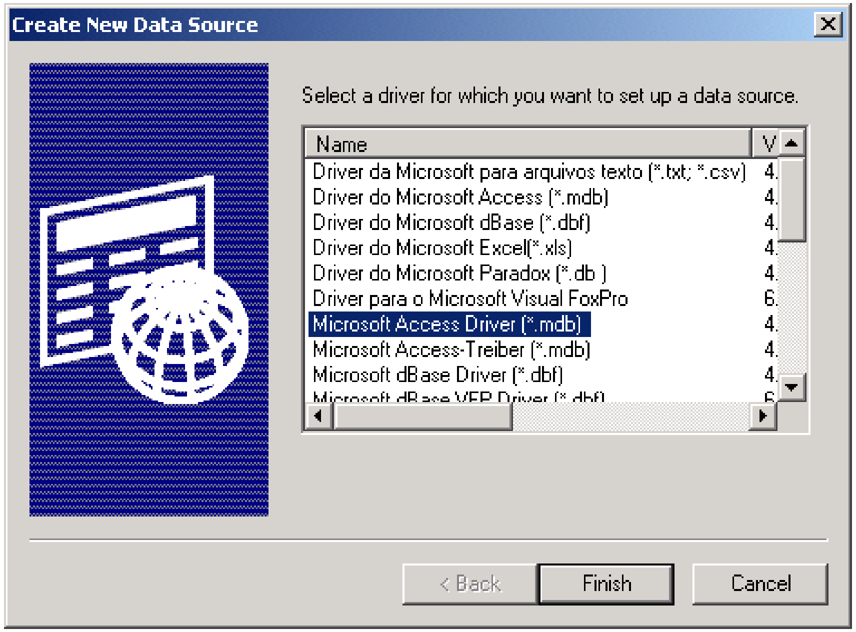
```
Import java.sql.*;

public class AuthorDatabase {
  public static void main(String[] args) {
    try {
          String url = “jdbc:odbc:library”;
          String driver = “sun.jdbc.odbc.JdbcOdbcDriver”;
          String user = “goel”
          String password = “password”;
          // Load the Driver
          Class.forName(driver);
          Connection connection = DriverManager.getConnection();
          String sqlString = “UPDATE authors SET lastname = ? Authid = ?”;
          PreparedStatement ps = connection.prepareStatement(sqlString);
          // Sets first placeholder to Allamaraju
          ps.setString(1, “Allamaraju”);
         // Sets second placeholder to 212
         ps.setString(2, 212);
         // Executes the update
         int rowsUpdated = ps.executeUpdate();
         System.out.println(“Number of rows changed = “ + rowsUpdated);
         connection.close();
         }
      catch (ClassNotFoundException cnfe) {
         System.out.println(“Driver not found”);
         cnfe.printStackTrace();
       }
       catch (SQLException sqle) {
         System.out.println(“Bad SQL statement”);
         sqle.printStackTrace();
       }
```

Access Data Source

- Create a database
- Select DataSources (ODBC) from the control panel

  (Start→ Settings→ ControlPanel→DataSources→AdministrativeTools→Data Sources)
- Select the System DSN tab
- On ODBC data source administrator click on add
- Select the database driver as Microsoft Access Driver

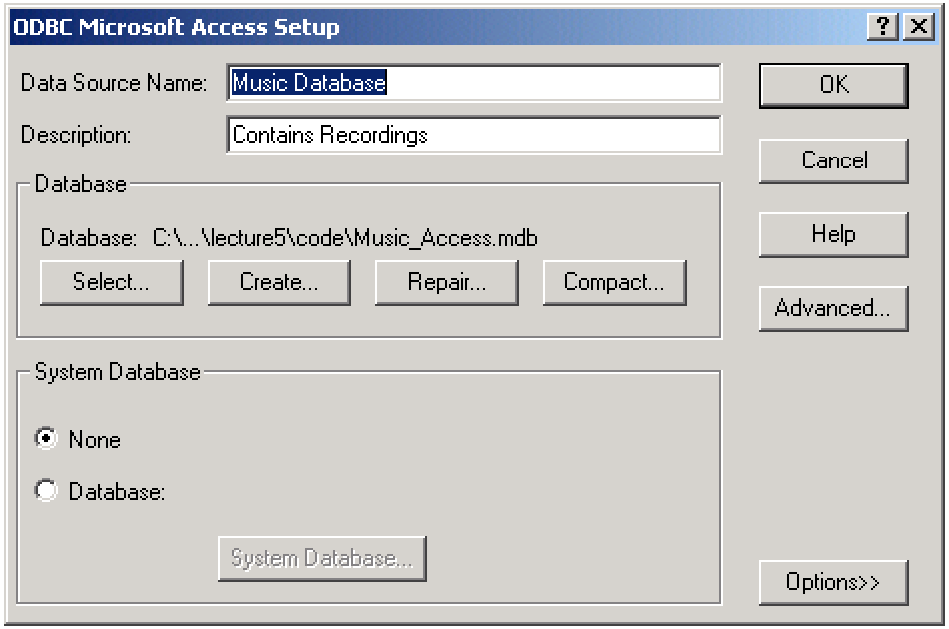


Select a driver for which you want to set up a data source.

Name	V
Driver da Microsoft para arquivos texto (*.txt; *.csv)	4.
Driver do Microsoft Access (*.mdb)	4.
Driver do Microsoft dBase (*.dbf)	4.
Driver do Microsoft Excel(*.xls)	4.
Driver do Microsoft Paradox (*.db)	4.
Driver para o Microsoft Visual FoxPro	6.
Microsoft Access Driver (*.mdb)	4.
Microsoft Access-Treiber (*.mdb)	4.
Microsoft dBase Driver (*.dbf)	4.
Microsoft dBase VFP Driver (*.dbf)	6.

Access Data Source

- Fill the ODBC Microsoft Access Setup Form
  - Write Data Source Name (Name of the data source that you have in the program)
  - Add description of database
  - Click on select and browse the directory to pick a database file
  - Click on OK



ODBC Microsoft Access Setup

Data Source Name: Music Database

Description: Contains Recordings

Database

Database: C:\...\lecture5\code\Music_Access.mdb

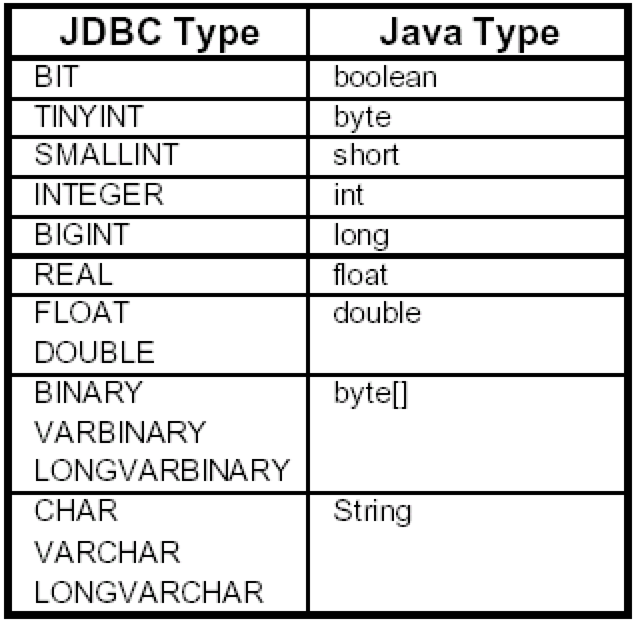
System Database

● None

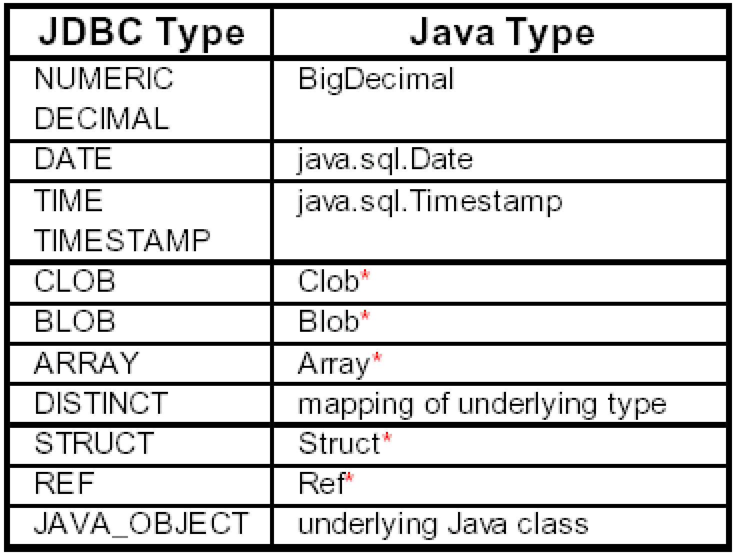
○ Database:

Advanced Topics

JDBC – Data Types



JDBC Type	Java Type
BIT	boolean
TINYINT	byte
SMALLINT	short
INTEGER	int
BIGINT	long
REAL	float
FLOAT DOUBLE	double
BINARY VARBINARY LONGVARBINARY	byte[]
CHAR VARCHAR LONGVARCHAR	String



JDBC Type	Java Type
NUMERIC DECIMAL	BigDecimal
DATE	java.sql.Date
TIME TIMESTAMP	java.sql.Timestamp
CLOB	Clob*
BLOB	Blob*
ARRAY	Array*
DISTINCT	mapping of underlying type
STRUCT	Struct*
REF	Ref*
JAVA_OBJECT	underlying Java class

Prepared Statement

- PreparedStatement provides a means to create a reusable statement that is precompiled by the database
- Processing time of an SQL query consists of
  - Parsing the SQL string
  - Checking the Syntax
  - Checking the Semantics
- Parsing time is often longer than time required to run the query
- PreparedStatement is used to pass an SQL string to the database where it can be pre-processed for execution

Prepared Statement

- It has three main uses
  - Create parameterized statements such that data for parameters can be dynamically substituted
  - Create statements where data values may not be character strings
  - Precompiling SQL statements to avoid repeated compiling of the same SQL statement
- If parameters for the query are not set the driver returns an SQL Exception
- Only the no parameters versions of executeUpdate() and executeQuery() allowed with prepared statements.

Prepared Statement

- Example

```
// Creating a prepared Statement
String sqlString = “UPDATE authors SET lastname = ? Authid = ?”;
PreparedStatement ps = connection.prepareStatement(sqlString);
ps.setString(1, “Allamaraju”);     // Sets first placeholder to Allamaraju
ps.setString(2, 212);                   // Sets second placeholder to 212
ps.executeUpdate();                     // Executes the update
```

Callable Statements & Stored Procedures

- Stored Procedures
  - Are procedures that are stored in a database.
  - Consist of SQL statements as well as procedural language statements
  - May (or may not) take some arguments
  - May (or may not) return some values
- Advantages of Stored Procedures
  - Encapsulation & Reuse
  - Transaction Control
  - Standardization
- Disadvantages
  - Database specific (lose independence)
- Callable statements provide means of using stored procedures in the database

Callable Statements & Stored Procedures

- Stored Procedures must follow certain rules
  - Names of the stored procedures and parameters must be legal
  - Parameter types must be legal supported by database
  - Each parameter must have one of In, Out or Inout modes
- Example

  // Creating a stored procedure using SQL

  - CREATE PROC procProductsList AS SELECT * FROM Products;
  - CREATE PROC procProductsDeleteItem(inProductsID LONG) AS DELETE FROM Products WHERE ProductsID = inProductsID;“
  - CREATE PROC procProductsAddItem(inProductName VARCHAR(40), inSupplierID LONG, inCategoryID LONG) AS INSERT INTO Products (ProductName, SupplierID, CategoryID) Values (inProductName, inSupplierID, inCategoryID);"
  - CREATE PROC procProductsUpdateItem(inProductID LONG, inProductName VARCHAR(40)) AS UPDATE Products SET ProductName = inProductName WHERE ProductID = inProductID;"

  Usage: procProductsUpdateItem(1000, “My Music”)

  (Sets the name of the product with id 1000 to 16.99)

Example of Using Blob (Images)

- Look at

JNDI

- Look at