JDBC

The slides adapted from different resources including:

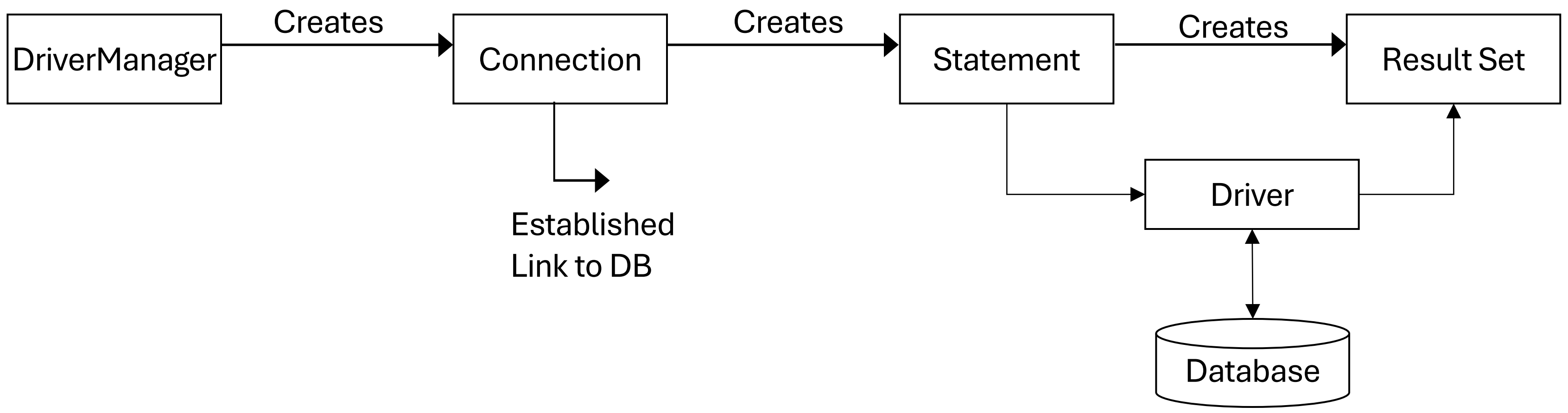
- Hitha Paulson, Assistant Professor, Dept. of Computer Science LF College, Guruvayoor
- MET CS 667: 9. JDBC, Servlets, JSP Zlateva
- Sanjay Goel

JDBC

- JDBC (Java Database Connectivity) is an interface between Java and Database
- JDBC receives queries from a Java Application program and communicate with Database
- All the communications are in the form of SQL commands
- JDBC is reponsible for
    - Open a Connection
    - Communicate with database
    - Execute SQL statements
    - Retrive query results

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

1. Import the necessary classes
2. Load the JDBC driver
3. Identify the data source (Define the Connection URL)
4. Establish the Connection
5. Create a Statement Object
6. Execute query string using Statement Object
7. Retrieve data from the returned ResultSet Object
8. 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

Load and Register Driver

Class.forName("`Driver ClassName`");

e.g., 1 Class.forName(“sun.jdbc.odbc.JdbcOdbcDriver”)

2 Class.forName("com.mysql.jdbc.Driver");

Note: Calling the Class.forName automatically creates an instance of a driver and registers it with the DriverManager, so you don't need to create an instance of the 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

Establish Connection

```
Connection conn = DriverManager.getConnection("URL");

Connection conn = DriverManager.getConnection(url, username, password);
```

- The drivers loaded recognizes, the JDBC URL in DriverManager.getConnection, that driver establishes a connection to the DBMS specified in the JDBC URL.
- The DriverManager class,manages all of the details of establishing the connection
- The connection returned by the method DriverManager.getConnection is an open connection you can use to create JDBC statements that pass your SQL statements to the DBMS.

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 (cont.)

- Code Example (Oracle)

```
 try {
     Class.forName("oracle.jdbc.driver.OracleDriver");
     String sourceURL = "jdbc:oracle:thin:@hostname:1521:databasename";
     String user = “myUserName";
     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

- 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();

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.getConne
  ction(url);
  Statement stmt = conn.
  createStatement();
  // JDBC Code
  } catch (SQLException
  sqle) {
  sqle.printStackTrace();
  } finally {
    try {stmt.close();
           conn.close();
     } catch (Exception
  e) {

  e.printStackTrace();
     }
  }
```

Statement Methods - Executing Queries

- Two primary methods in statement interface used for executing Queries
    - executeQuery Used to retrieve data from a database
    - executeUpdate: Used for inserting, 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 - Data Manipulation Language (DML)

- 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
- 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(1) ); }`

ResultSet

- Access Rows of a ResultSet
  - The basic loop for analyzing a result set uses format:

```
ResultSet rs = stmt.executeQuery (“select * from books”);
while ( rs.next()) {
   // rs.next pointers to the next row
      analyze a row of the result set;
   }
```

- Access Attributes of a row
  - Get the value of an attribute by methods of format:

```
Xxx getXxx (int clmn_num) or
Xxx getXxx (String clmn_name)
e.g., String cofName = rs.getString(1) or
       String cofName = rs.getString(“COF_NAME”);
```

  - There are many other getXxx methods including:

```
      int   supId = rs.getInt (“SUP_ID”);
      float price = rs.getDouble (“PRICE”);
```

JDBC – Metadata from RS

```
public static void printRS(ResultSet rs) throws
  SQLException
{
  ResultSetMetaData md = rs.getMetaData();
  // get number of columns
  int nCols = md.getColumnCount();
  // print column names
  for(int i=1; i < nCols; ++i)
        System.out.print( md.getColumnName( i)+",");
      / / output resultset
  while ( rs.next() )
  {     for(int i=1; i < nCols; ++i)
                System.out.print( rs.getString( i)+",");
        System.out.println( rs.getString(nCols) );
  }
}
```