Object Oriented Programming

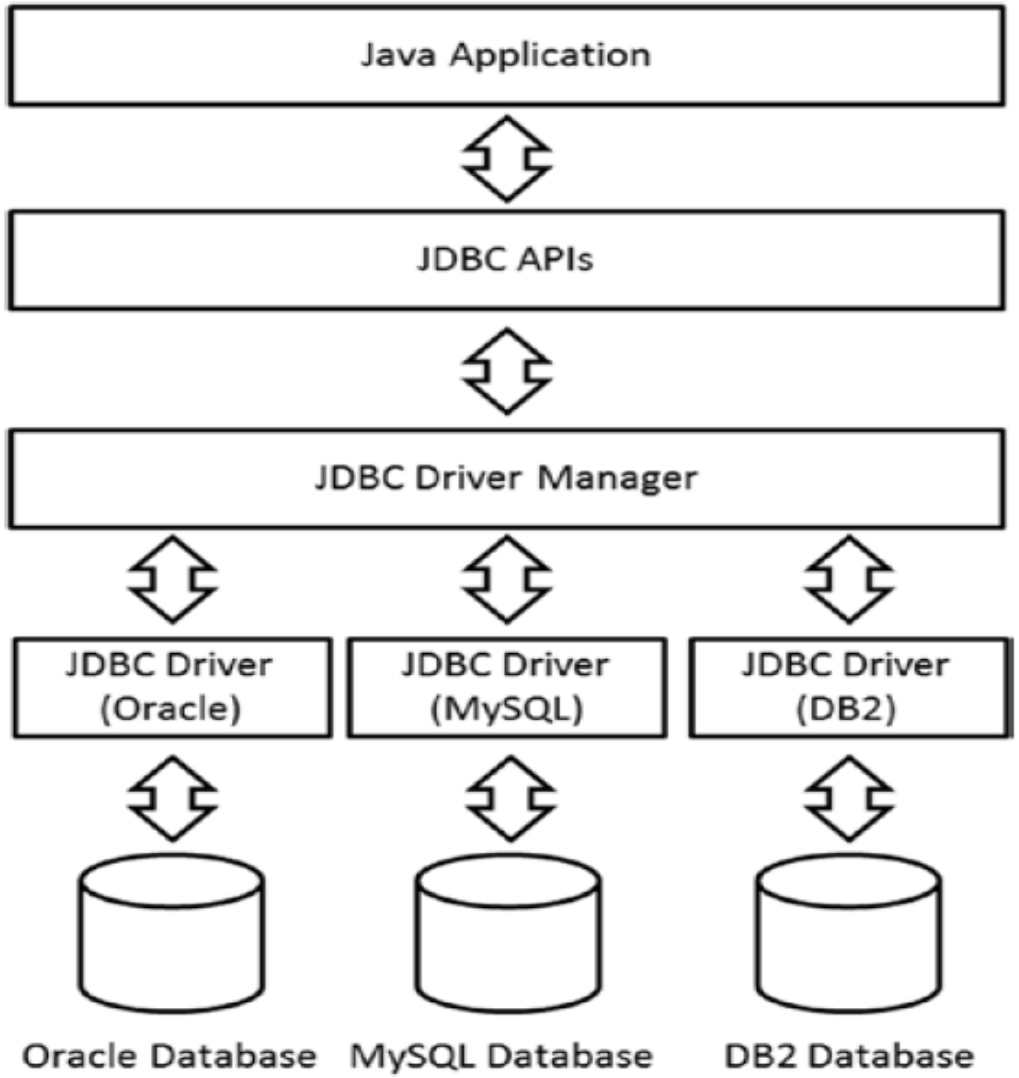
(CSC 302)

Lecture Note

JDBC

4th December, 2019

The JDBC Architecture



How to use JDBC to work with databases

- An introduction to database drivers

Before you can connect to a database, you must make a database driver available to your application

- The four types of JDBC database drivers.
  - **Type 1** - A JDBC-ODBC bridge driver converts JDBC calls into ODBC calls that access the DBMS protocol. For this data access method, an ODBC driver must be installed on the client machine. A JDBC-ODBC bridge driver was included as part of the JDK prior to Java 8 but is not included or supported with Java 8 and later.
  - **Type 2** - A native protocol partly Java driver converts JDBC calls into the native DBMS protocol. Since this conversion takes place on the client, the database client library must be installed on the client machine.
  - **Type 3** - A net protocol all Java driver converts JDBC calls into a net protocol that's independent of any native DBMS protocol. Then, middleware software running on a server converts the net protocol to the native DBMS protocol. Since this conversion takes place on the server side, the database client library isn't required on the client machine.
  - **Type 4** - A native protocol all Java driver converts JDBC calls into the native DBMS protocol. Since this conversion takes place on the server, the database client library isn't required on the client machine.

- How to download a database driver
  - For MySQL databases, you can download a JDBC driver named **Connector/J** from the MySQL website. This driver is an open-source, type-4 driver that's available for free.
  - For other databases, you can usually download a type-4 JDBC driver from the database's website.
  - The Connector/J driver for MySQL databases is included with NetBeans. As a result, if you're using NetBeans, you don't need to download this driver.
- How to make a database driver available to an application
  - Before you can use a database driver, you must make it available to your application. The easiest way to do this is to use your IDE to add the JAR file for the driver to your application.
  - To add the MySQL JDBC driver to a NetBeans project, right-click on the Libraries folder, select the Add Library command, and use the resulting dialog box to select the MySQL JDBC Driver library.
  - To add any JDBC driver to a NetBeans project, right-click on the Libraries folder, select the Add JAR/Folder command, and use the resulting dialog box to select the JAR file for the driver.

- How to connect to a database
- A package for working with databases

  java.sql
- Interfaces for connecting to a database
  - **Driver:** Knows how to get a connection to the database
  - **Connection:** Knows how to communicate with the database
  - **Statement:** Knows how to run the SQL
  - **ResultSet:** Knows what was returned by a SELECT query

- **Database URL syntax**
  - jdbc:subprotocolName:databaseURL
- mysql url
  - jdbc:mysql://localhost:3306/dbname
- Oracle url

  jdbc:oracle:thin@localhost/dbname
- postgreSQL url
  - jdbc:postgresql://localhost:5432/dbname
- MSSQL Server
  - jdbc:microsoft:sqlserver://dbname:1433

The Connection Interface

- The Connection interface of the **java.sql** package represents a connection from application to the database.
- It is a channel through which your application and the database communicate. The table below lists important methods in the Connection interface. All of these methods throw **SQLExceptions**.
- **Statement createStatement()** - Creates a Statement object that can be used to send SQL statements to the database.
- **PreparedStatement prepareStatement(**String sql**)** - Creates a PreparedStatement object that can contain SQL statements. The SQL statement can have IN parameters; they may contain ? symbol(s), which are used as placeholders for passing actual values later.
- **CallableStatement prepareCall(**String sql**)** - Creates a CallableStatement object for calling stored procedures in the database. The SQL statement can have IN or OUT parameters; they may contain ? symbol(s), which are used as placeholders for passing actual values later.

- **DatabaseMetaData getMetaData()** - Gets the DataBaseMetaData object. This metadata contains database schema information, table information, and so on, which is especially useful when you don not know the underlying database.
- **Clob createClob()** - Returns a Clob object (Clob is the name of the interface). Character Large Object (CLOB) is a built-in type in SQL; it can be used to store a column value in a row of a database table.
- **Blob createBlob()** - Returns a Blob object (Blob is the name of the interface). Binary Large Object (BLOB) is a built-in type in SQL; it can be used to store a column value in a row of a database table.
- **void setSchema(**String schema**) -** When passed the schema name, sets this Connection object to the database schema to access.
- **String getSchema()** Returns the schema name of the database associated with this Connection object; returns null if no schema is associated with it.

The DriverManager Class

- The **DriverManager** class helps establish the connection between the program (the user) and the JDBC drivers. This class also keeps track of different data sources and JDBC drivers. Hence, there is no need to explicitly load the JDBC driver.
- DriverManager searches for a suitable driver and, if found, automatically loads it when you call the **getConnection**() method.
- You can code the following statement to get connection (given within a try-with-resources statement) when you don not explicitly load the JDBC driver:
  - `Connection connection = DriverManager.getConnection(url + database, userName, password);`

Important Static Methods in the DriverManager Class

Method	Description
Connection getConnection(String url) getConnection(String url, Properties info) getConnection(String url, String user, String password)	Attempts to establish a connection given the database URL. Additionally, you can provide Information such as a username and password directly as String arguments or through a Properties file. This method throws an SQLException if the connection can't be established.
Driver getDriver(String url)	Searches the list of registered JDBC drivers and, if found, returns the appropriate Driver object matching the database URL.
void registerDriver(Driver driver)	Add to the list of registered Driver objects in the DriverManager.
deregisterDriver(Driver driver)	Deregisters a driver from the list of registered Driver objects in the DriverManager

- **How to get Connection**
- Before you can get or modify the data in a database, you need to connect to it. To do that, you use the **getConnection** method of the **DriverManager** class to return a Connection object.
- When you use the getConnection method of the DriverManager class, you must supply a URL for the database. In addition, you usually supply a username and a password, though you might not for an embedded database. This method throws a **SQLException**.
- With JDBC 4.0 and later, the SQLException class implements the **Iterable** interface. As a result, you can use an enhanced for statement to loop through any nested exceptions.
- With JDBC 4.0 and later, the database driver is loaded automatically. This feature is known as **automatic driver loading**.
- Prior to JDBC 4.0, you needed to use the **forName** method of the **Class** class to load the driver. This method throws a **ClassNotFoundException**.

- Although the connection string for each driver is different, the documentation for the driver should explain how to write a connection string for that driver.
- Typically, you only need to connect to one database for an application. However, it's possible to load multiple database drivers and establish connections to multiple types of databases.
- **How to connect to a database**
  - Connection connection = DriverManager.getConnection(url, user, password)
  - Connection connection = DriverManager.getConnection(url)

- How to return a result set and move the cursor through it
  - To return a result set, you use the **createStatement** method of a Connection object to create a Statement object. Then, you use the **executeQuery** method of the Statement object to execute a **SELECT** statement that returns a **ResultSet** object.
  - By default, the createStatement method creates a *forward-only*, *read-only* result set. This means that you can only move the cursor through it from the first row to the last and that you can't update it. Although you can pass arguments to the createStatement method that create other types of result sets, the default is appropriate for most applications.
  - When a result set is created, the cursor is positioned before the first row. Then, you can use the methods of the ResultSet object to move the cursor. To move the cursor to the next row, for example, you call the next method. If the row is valid, this method moves the cursor to the next row and returns a true value. Otherwise, it returns a false value.
  - The createStatement, executeQuery, and next methods throw a SQLException. As a result, any code that uses these methods needs to catch or throw this exception.

- How to create a result set that contains 1 row and 1 column

```
Statement statement = connection.createStatement();
ResultSet product = statement.executeQuery(
        "SELECT ProductCode FROM Product " +
        "WHERE ProductID = '1'");
```

- How to create a result set that contains multiple columns and rows

```
Statement statement = connection.createStatement();
ResultSet products = statement.executeQuery(
      "SELECT * FROM Product ");
```

- How to move the cursor to the first row in the result set

```
boolean productExists = product.next();
```

- How to loop through a result set

```
while (products.next()) {
   // statements that process each row
}
```

- ResultSet methods for forward-only, read-only result sets
  - **next()** - Moves the cursor to the next row in the result set.
  - **last()** - Moves the cursor to the last row in the result set.
  - **close()** - Releases the result set's resources.
- How to get data from a result set
- Methods of a ResultSet object that return data from a result set

Method	Return Type	Example database type
getBoolean()	boolean	BOOLEAN
getDate()	java.sql.date	DATE
getString()	String	CHAR, VARCHAR
getDouble()	double	DOUBLE
getTime()	java.sql.time	TIME
getTimeStamp()	java.sql.TimeStamp	TIMESTAMP
getInt()	Int	INTEGER
getObject()	Object	Any type
getLong	Long	BIGINT

- Code that uses indexes to return columns from the students result set
  - int studID = students.getInt(1);
  - String fname = students.getString(2);
  - String surname = students.getString(3);
  - double gpa = students.getDouble(4);
- Code that uses names to return the same columns
  - int studID = students.getInt(“StudentID");
  - String fname = students.getString(“FirstName");
  - String surname = students.getString(“Surname");
  - double gpa = students.getDouble(“GPA");

- Code that creates a Student object from the students result set
  - Student st = new Student();
  - st.setId(studID);
  - st.setFirstName(fname);
  - st.setSurname (surname);
  - st.setGPA(price);

- How to insert, update, and delete data
- How to add a row

```
String query = "INSERT INTO student (studID, FirstName, Surname, GPA) "
             + "VALUES ('" + student.getID() + "', " +
                        "'" + student.getFirstName() + "', " +
                         "'" + student.getSurname() + "', " +
                         "'" + product.getGPA() + "')";
Statement statement = connection.createStatement();
int rowCount = statement.executeUpdate(query);
```

- How to delete a row

```
String query = "DELETE FROM Student " +
               "WHERE studID = '" + studID + "'";
Statement statement = connection.createStatement();
int rowCount = statement.executeUpdate(query);
```

- The **executeUpdate** method is an older method that works with most JDBC drivers. Although there are some newer methods that require less SQL code, they may not work properly with all JDBC drivers.
- The executeUpdate method returns an *int* value that identifies the number of rows that were affected by the SQL statement.

Warning

- If you build a SQL statement from user input and use a method of the Statement object to execute that SQL statement, you may be susceptible to a security vulnerability known as a SQL **injection attack**.
- A SQL injection attack allows a hacker to bypass authentication or to execute SQL statements against your database that can read sensitive data, modify data, or delete data.
- To prevent most types of SQL injection attacks, you can use prepared statements as described in the next slide.

- How to work with prepared statements
  - When you use **prepared** statements in your Java applications, the database server only has to check the syntax and prepare an execution plan once for each SQL statement. This improves the efficiency of the database operations. In addition, it prevents most types of SQL injection attacks.
  - To specify a parameter for a prepared statement, type a question mark (?) in the SQL statement.
  - To supply values for the parameters in a prepared statement, use the set methods of the **PreparedStatement** interface. For a complete list of set methods, look up the PreparedStatement interface of the **java.sql** package in the documentation for the Java API.
  - To execute a SELECT statement, use the **executeQuery** method. To execute an INSERT , UPDATE, or DELETE statement, use the **executeUpdate** method.

- How to use a prepared statement
- To return a result set

```
String sql = "SELECT ProductCode, ProductDescription, ProductPrice " +
                "FROM Product WHERE ProductCode = ?";
PreparedStatement ps = connection.prepareStatement(sql);
ps.setString(1, productCode);
ResultSet product = ps.executeQuery();
```

- To modify a row

```
String sql = "UPDATE Student SET "
          + " studID = ?, "
          + " FirstName = ?, "
          + " Surname = ?, "
          + " GPA = ? "
          + "WHERE ProductCode = ?";
PreparedStatement ps = connection.prepareStatement(sql);
ps.setString(1, student.getId()); ps.setString(2, student.getFirstName());
ps.setString(3, student.getSurname());  ps.setDouble(4, product.getGPA());
ps.setString(5, product.getId()); ps.executeUpdate();
```

STUDENT MANAGEMENT SYSTEM