# COSC 304
# Introduction to Database Systems

## Database Programming

**Dr. Ramon Lawrence**
**University of British Columbia Okanagan**
**ramon.lawrence@ubc.ca**

# Database Programming Overview

Most user interaction with a database is through programs instead of directly using SQL commands to the database.

A programmer developing a database application executes SQL directly or indirectly in code using one of these methods:

- standard application programming interfaces (APIs) such as JDBC
- custom APIs specific to the database vendor
- object-relational mapping technologies

All methods must process queries in code, send them to the database, retrieve database results, and map the result data into program variables so that they can be used by the program.

# *JDBC Overview*

**JDBC** is the most popular method for accessing databases using Java programs.

JDBC is an ***application programming interface*** (***API***) that contains methods to connect to a database, execute queries, and retrieve results.

For each DBMS, the vendor writes a JDBC driver that implements the API. Application programs can access different DBMSs simply by changing the driver used in their program.

# JDBC Interfaces

The JDBC API consists of a set of interfaces.

- A Java *interface* is an abstract class consisting of a set of methods with no implementation.
- To create a JDBC driver, the DBMS vendor must implement the interfaces by defining their own classes and writing the code for the methods in the interface.

The main interfaces in the JDBC API are:

- `Driver` - The main class of the entire driver.
- `Connection` - For connecting to the DB using the driver.
- `Statement` - For executing a query using a connection.
- `ResultSet` - For storing and manipulating results returned by a `Statement`.
- `DatabaseMetaData` - For retrieving metadata (schema) information from a database.

# JDBC Program Example

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

Import JDBC API

DB Connection Info

```
public class TestJDBCMySQL
{   public static void main(String[] args)
    {   String url = "jdbc:mysql://cosc304.ok.ubc.ca/WorksOn";
        String uid = "user";
        String pw = "testpw";

        try {          // Load driver class
            Class.forName("com.mysql.jdbc.Driver");
        }
        catch (java.lang.ClassNotFoundException e) {
            System.err.println("ClassNotFoundException: " +e);
        }
```

Load JDBC Driver
(for MySQL – optional)

# JDBC Program Example (2)

```
Connection con = null;
try {
    con = DriverManager.getConnection(url, uid, pw);
    Statement stmt = con.createStatement();
    ResultSet rst = stmt.executeQuery("SELECT ename,salary
                                       FROM Emp");
    System.out.println("Employee Name,Salary");
    while (rst.next())
       System.out.println(rst.getString("ename")
              +","+rst.getDouble("salary"));
}
catch (SQLException ex) { System.err.println(ex);}
finally
{   if (con != null)
        try
        {   con.close(); }
        catch (SQLException ex) { System.err.println(ex); }
    }
}
```

Make DB connection

Create statement

Execute statement

Iterate through ResultSet

# JDBC Program Example
# Try-with-Resources Syntax (Java 7+)

```
try (Connection con = DriverManager.getConnection(url, uid, pw);
     Statement stmt = con.createStatement();)
{
    ResultSet rst = stmt.executeQuery("SELECT ename,salary
                                       FROM Emp");
    System.out.println("Employee Name,Salary");
    while (rst.next())
        System.out.println(rst.getString("ename")
                           +","+rst.getDouble("salary"));
}
catch (SQLException ex)
{
    System.err.println(ex);
}
```

Declare resources

`Statement` and `Connection` objects closed by end of `try`

# JDBC `Driver` Interface

The `Driver` interface is the main interface that must be implemented by a DBMS vendor when writing a JDBC driver.

- The class itself does not do very much except allow a connection to be made to a database through the driver.
- Note that you do not call the `Driver` class directly to get a connection. Drivers self-register with the `DriverManager` so `Class.forName()` is no longer needed.
- When you call `DriverManager.getConnection()`, the `DriverManager` will attempt to locate a suitable driver.

# JDBC `Driver` Interface (2)

The `DriverManager` determines which JDBC driver to use based on the URL used as a parameter:

```
// Components of a URL
String url = "jdbc:mysql://cosc304.ok.ubc.ca/testDB";
```

JDBC protocol

subprotocol - used to select driver to use

URL of DB server

Name of DB on server

The interface method `Driver.connect()` is called by the `DriverManager` on the driver object that will allow access based on the subprotocol specified in the URL.

- Inside this method the JDBC driver writer performs the actions to connect to the DBMS and return a `Connection` object.
- A `try .. catch` is also needed here if the driver is unable to successfully connect to the DB at the specified URL.

# JDBC Driver Vendor Compliance

Each DBMS vendor may create a JDBC driver and provide any level of JDBC support that they desire.

- Some methods in the JDBC API are not available on all drivers.
- The basic support required is Entry Level SQL92.

Historical JDBC versions with key features:

- JDBC 1.0 (1997) - `Driver`, `Connection`, `Statement`, `ResultSet`, `DatabaseMetadata`, `ResultSetMetaData`
- JDBC 2.0 (1999) - BLOBs, updatable `ResultSet`, pooled connections
- JDBC 3.0 (2001) - improved transactions (savepoints), SQL99, retrieve auto-generated keys, update BLOBs
- JDBC 4.0 (2006) - XML support, better BLOB support, auto driver loading, do not need `Class.forName()`
- JDBC 4.1 (Java 7), JDBC 4.2 (Java 8)

# JDBC `Connection` Interface

The `Connection` interface contains abstract methods for managing a connection or session.

- A connection is opened after the call to `getConnection()` and should be explicitly closed when you are done.
- The `Connection` interface is used to create statements for execution on the database.

```
Connection con = DriverManager.getConnection(url, uid, pw);
Statement stmt = con.createStatement();
...
con.close();
```

# `Connection` Interface and MetaData

A `Connection` to a database can also be used to retrieve the database metadata (or schema).

- This is useful when you are writing generic tools where you do not know the schema of the database that you are querying in advance.

The method `getMetaData()` can be used to retrieve a `DatabaseMetaData` object.

# DatabaseMetaData Example

```
String []tblTypes = {"TABLE"};// What table types to retrieve

try {
    DatabaseMetaData dmd = con.getMetaData();  // Get metadata
    ResultSet rs1, rs2, rs5;

    System.out.println("List all tables in database: ");

    rs1 = dmd.getTables(null, null, "%", tblTypes);
    while (rs1.next()) {
        String tblName = rs1.getString(3);

        Statement stmt = con.createStatement();
        rs2 = stmt.executeQuery("SELECT Count(*) FROM "+tblName);
        rs2.next();
        System.out.println("Table: "+tblName+" # records: "+
                                rs2.getInt(1));
```

# DatabaseMetaData Example (2)

```
        rs5 = dmd.getColumns(null,null,tblName,"%");
        System.out.println("  Attributes: ");

        while (rs5.next()) {
            System.out.println(rs5.getString(4));
        }
    } // end outer while
} // end try
```

# JDBC `Statement` Interface

The `Statement` interface contains abstract methods for executing a single static SQL statement and returning the results it produces.

```
Statement stmt = con.createStatement();
ResultSet rst = stmt.executeQuery("SELECT ename, salary
                                   FROM Emp");
```

- The `Statement` object is created by calling `Connection.createStatement()`.
- The statement is then executed by calling `executeQuery()` and passing the SQL string to execute.

# JDBC `Statement` Interface (2)

There are two important variations of executing statements that are important and are used often.

- 1) The `Statement` executed is an `INSERT`, `UPDATE`, or `DELETE` and no results are expected to be returned:

```
rowcount = stmt.executeUpdate("UPDATE Emp Set salary=0");
```

- 2) The `Statement` executed is an `INSERT` which is creating a new record in a table whose primary key field is an autonumber field:

```
rowcount = stmt.executeUpdate("INSERT Product
                               VALUES ('Prod. Name')",
                               Statement.RETURN_GENERATED_KEYS );
ResultSet autoKeys = stmt.getGeneratedKeys();
```

# PreparedStatement and CallableStatement

There are two special types of `Statement` objects:

- `PreparedStatement` - extends `Statement` and is used to execute precompiled SQL statements.
  - Useful when executing the same statement multiple times with different parameters as the DBMS can optimize its parsing and execution.
  - Also useful to prevent SQL injection attacks.

```
String SQL = "UPDATE Emp SET salary = ? WHERE ID = ?";
PreparedStatement pstmt = con.prepareStatement(SQL);
pstmt.setBigDecimal(1, 55657.34); // Set parameters
pstmt.setString(2,"E1");
int rowcount = pstmt.executeUpdate();
```

- `CallableStatement` - extends `PreparedStatement` and is used to execute stored procedures.
  - Stored procedures are precompiled SQL code stored at the database that take in parameters for their execution.

# JDBC `ResultSet`

The `ResultSet` interface provides methods for manipulating the result returned by the SQL statement.

- Remember that the result is a relation (table) which contains rows and columns.
- The methods provide ways of navigating through the rows and then selecting columns of the current row.
- A `ResultSet` object maintains a cursor pointing to its current row of data. Initially the cursor is positioned before the first row.
- The `next()` method moves the cursor to the next row, and because it returns false when there are no more rows in the `ResultSet` object, it can be used in a while loop to iterate through the result set.

# JDBC ResultSet (2)

By default a `ResultSet` is not *updatable* and the cursor can move *forward-only* (can only use `next()` method).

```
while (rst.next())
{   System.out.println(rst.getString("ename")
                                +","+rst.getDouble(2));
}
```

- Remember, the first call to `next()` places the row cursor on the first row as the cursor starts off before the first row.
- Use the `getType()` methods to retrieve a particular type.
  - `getArray(), getBlob(), getBoolean(), getClob(), getDate(), getDouble(), getFloat(), getInt(), getLong(), getObject(), getString(), getTime()`
  - All methods take as a parameter the column index in the `ResultSet` (indexed from 1) or the column name and return the requested type.
  - Java will attempt to perform casting if the type you request is not the type returned by the database.

# Scrollable ResultSets

It is also possible to request ResultSets that allow you to navigate backwards as well as forwards.

- Request ResultSet type during createStatement.

```
// rs will be scrollable and read-only
Statement stmt = con.createStatement(
                    ResultSet.TYPE_SCROLL_INSENSITIVE,
                    ResultSet.CONCUR_READ_ONLY);
ResultSet rs = stmt.executeQuery("SELECT eno, ename FROM Emp");
```

Scrollable ResultSets allow you to navigate in any direction through it, and these methods can now be used:

- absolute(int row) - set cursor to point to the given row (starting at 1)
- afterLast(), beforeFirst(), first(), last(), next(), previous()
- Scrollable ResultSets were introduced in JDBC 2.0 API and may be less efficient than forward-only ResultSets.

# *Updatable* `ResultSets`

*Updatable* `ResultSets` allow you to update fields in the query result and update entire rows.

Updating an existing row:

```
// rs will be scrollable and updatable record set
Statement stmt = con.createStatement(
                    ResultSet.TYPE_SCROLL_INSENSITIVE,
                    ResultSet.CONCUR_UPDATABLE);
ResultSet rs = stmt.executeQuery("SELECT eno, ename FROM Emp");

rs.absolute(2);                 // Go to the 2nd row
rs.updateString(2,"Joe Blow");// Change name of employee
rs.updateRow();                 // Update data source
```

# Updatable ResultSets (2)

Adding a new row to a ResultSet:

```
// rs will be scrollable and updatable record set
Statement stmt = con.createStatement(
                    ResultSet.TYPE_SCROLL_INSENSITIVE,
                    ResultSet.CONCUR_UPDATABLE);
ResultSet rs = stmt.executeQuery("SELECT eno, ename FROM Emp");

rs.moveToInsertRow();          // Move cursor to insert row
rs.updateString(1,"E9");
rs.updateString("ename","Joe Smith");
rs.insertRow();                // Insert new row in DB
rs.moveToCurrentRow();         // Move cursor to row you were on
                               //  before insert
```

# JDBC API Question

***Question:*** Which one is not a JDBC API interface?

**A)** `Driver`
**B)** `Connection`
**C)** `Statement`
**D)** `ResultSet`
**E)** `HashMap`

# JDBC API Question

***Question:*** Select a **true** statement.

**A)** When a `ResultSet` is first opened, the current row is 0.

**B)** When a `ResultSet` is first opened, the current row is 1.

**C)** When asking for columns, the first column is index 0.

**D)** The method call `first()` is allowed for a forward-only `ResultSet`.

# Custom APIs

Many databases have custom APIs for various languages.

- For example, MySQL has APIs for C/C++ and PHP in addition to ODBC and JDBC drivers.

When to use a custom API?

- Custom APIs may be useful when the ODBC/JDBC standard does not provide sufficient functionality.
- Also useful if *know* that application will only access one database. (Be careful with this .. think of the future).

Custom APIs may have improved performance and increased functionality. The disadvantage is that your code is written for a specific DBMS which makes changes difficult.

- If you use a custom API, always isolate the database access code to a few general classes and methods!

# Aside: Direct Connection using `Driver`

It is possible to directly create a driver class object and call its connect method instead of using `DriverManager`.

- NOT recommended as puts vendor specific code in your code!

```
// Create new structure for passing info to database
Properties prop = new Properties();
prop.put("user","rlawrenc");
prop.put("password","pw");

Connection con = null;
try {
    Driver d = new com.mysql.jdbc.Driver();
    con = d.connect(url, prop);
```

This also demonstrates the use of the `Properties` class that allows you to pass key/value pairs to the database that are not in the URL. Can also use with `DriverManager`:

```
con = DriverManager.getConnection(url, prop);
```

# *Aside: Making a Connection using `DataSource`*

`DataSource` in package `java.sqlx` is the preferred way for connecting to a database as it hides the URL and vendor specific code (if used in conjunction with JNDI).

```
import javax.sql.DataSource;
import com.mysql.jdbc.jdbc2.optional.*;

MysqlDataSource ds = new MysqlDataSource();
ds.setUser("rlawrenc");
ds.setPassword("pw");
ds.setServerName("cs-suse-4.ok.ubc.ca");
ds.setDatabaseName("WorksOn");
con = ds.getConnection();
```

Note without Java Naming and Directory Interface (JNDI) that vendor specific code is still in the application.

# Object-Relational Mapping Java Persistence Architecture (JPA)

A huge challenge with database programming is converting the database results to and from Java objects. This is called object-relational mapping, and it is tedious and error-prone.

- ***Impedance mismatch*** - Database returns values in tables and rows and Java code manipulates objects, classes, and methods.

Various vendors (e.g. Hibernate) have produced object-relational mapping technologies that help the programmer convert database results into Java objects.

The Java Persistence Architecture (JPA) has been developed as a standard interface. Vendors can then implement the interface in their products.

# JDBC Question

Create a JDBC program that:

- Connects to `WorksOn` database on `cosc304.ok.ubc.ca`.
- Prints on the console each department and its list of projects.

Variant:

- Output in reverse order by department number. Two versions:
  - Change SQL
  - Use scrollable ResultSets (hint: previous() method).

Challenge:

- Improve your code so that it prints the department number, name, and how many projects in that department THEN the list of projects.

# Conclusion

**JDBC** is a standard APIs for connecting to databases using Java. Querying using JDBC has these steps:

- Load a `Driver` for the database
- Make a `Connection`
- Create a `Statement`
- Execute a query to produce a `ResultSet`
- Navigate the `ResultSet` to display results or update the DB

There are other ways for accessing a database:

- Custom APIs specific to each database vendor
- object-relational mapping and JPA

***Java Persistence Architecture*** (***JPA***) is a standard API for object-relational mapping.

# *Objectives*

- Explain the general steps in querying a database using Java.
- List the main JDBC classes (`Driver`, `Connection`, `Statement`, `ResultSet`) and explain the role of each in a typical program.
- Discuss the different types of `ResultSets` including scrollable and updatable `ResultSets`.
- Write a simple JDBC program (given methods of the JDBC API).
- Discuss and explain the advantages and disadvantages of using a standard API versus a vendor-based APIs.
- Explain object-relational mapping and the impedance mismatch.