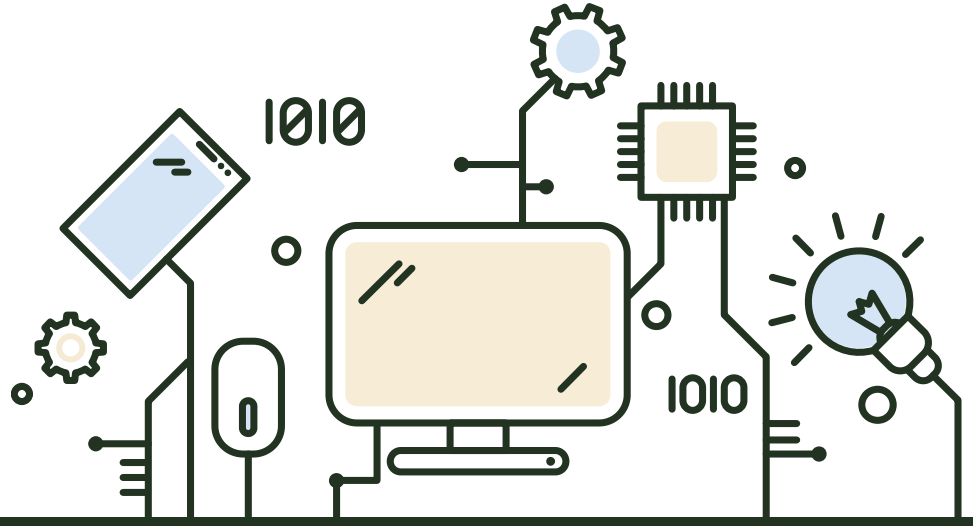
1010 LAB Digital Infomation Engineering

JDBC
(Java DataBase Connection)

LD0230 웹정보시스템 프로젝트

성신여자대학교
정보시스템공학과
홍 기 형



Introduction

1010 LAB

- **JDBC is an alternative to ODBC and ADO that provides database access to programs written in Java.**
- **JDBC stands for Java DataBase Connectivity.**
- **JDBC drivers are available for most DBMS products:**
  - http://devapp.sun.com/product/jdbc/drivers
  - for MySQL : Connector/J

JDBC Driver Types

1010 LAB

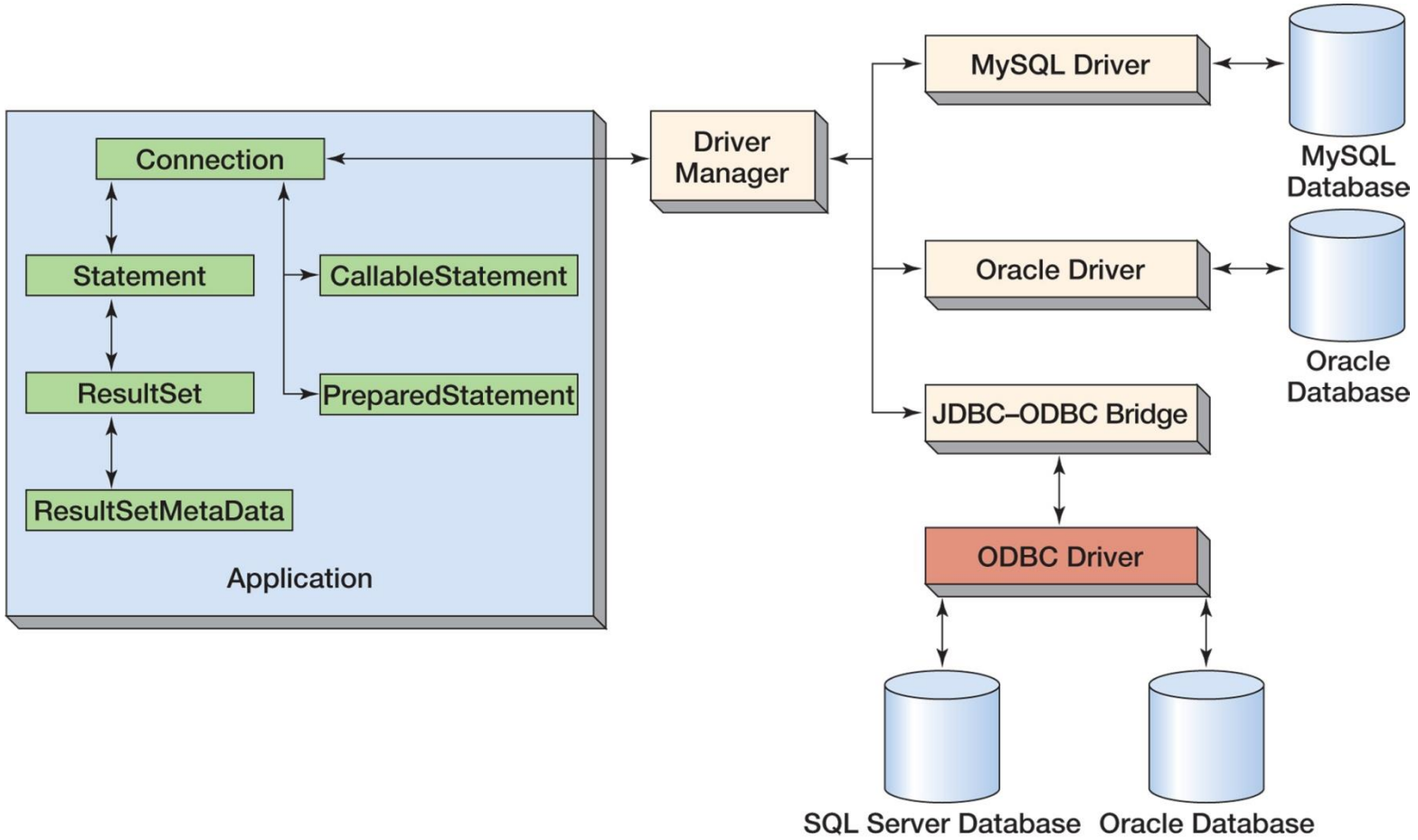
Driver Type	Characteristics
1	JDBC-ODBC bridge. Provides a Java API that interfaces to an ODBC driver. Enables processing of ODBC data sources from Java.
2	A Java API that connects to the native-library of a DBMS product. The Java program and the DBMS must reside on the same machine, or the DBMS must handle the intermachine communication, if not.
3	A Java API that connects to a DBMS-independent network protocol. Can be used for servlets and applets.
4	A Java API that connects to a DBMS-dependent network protocol. Can be used for servlets and applets.

Java Servlets and Applets

1010 LAB

- Java programs are compiled into an operating system independent bytecode.
- Various operating systems use their own bytecode interpreters a.k.a. Java virtual machines.
- An applet is transmitted to a browser via HTTP and is invoked on the client workstation using the HTTP protocol.
- A servlet is a Java program that is invoked on the server to respond to HTTP requests.
- Type 3 and Type 4 drivers can be used for both applets and servlets.
- Type 2 drivers can be used only in servlets.

JDBC Components



JDBC References

1010 LAB

- **Tutorial**
  - http://download.oracle.com/javase/tutorial/jdbc/basics/index.html
- **JDBC Package**
  - java.sql.*;
- **MySQL Connector/J Driver**
  - http://www.mysql.com/downloads/connector/j/5.1.html

Processing SQL Statements with JDBC

1. Establishing a connection.
2. Create a statement.
3. Execute the query.
4. Process the ResultSet object.
5. Close the connection.

Establishing Connections

1010 LAB

- **`DriverManager` Class**
  - `getConnection` method
    - database URL
    - user name and passwd required to access the DBMS with a `Properties` Object

MySQL Connector/J Database URL

1010 LAB

- **jdbc:mysql://[*host*] [,*failoverhost...*][:*port*]/[*database*]**

[?*propertyName1*][=*propertyValue1*][&*propertyName2*][=*propertyValue2*]

...

  - *host*:*port* is the host name and port number of the computer hosting your database. If not specified, the default values of *host* and *port* are 127.0.0.1 and 3306, respectively.
  - *database* is the name of the database to connect to. If not specified, a connection is made with no default database.
  - *failover* is the name of a standby database (MySQL Connector/J supports failover).
  - *propertyName*=*propertyValue* represents an optional, ampersand-separated list of properties. These attributes enable you to instruct MySQL Connector/J to perform various tasks. (user name and passwd)

MySQL JDBC connection

```
import java.sql.Connection;
import java.sql.DriverManager;
import java.util.Properties;


    Connection conn = null;
    Properties connectionProps = new Properties();
    connectionProps.put("user", "root");
    connectionProps.put("password", "");


    try {
        conn = DriverManager.
        getConnection("jdbc:" + "mysql" + "://" + "localhost" +
                      ":" + "3306" + "/"+"database", connectionProps);
    } catch (SQLException e) {
        // TODO Auto-generated catch block
        e.printStackTrace();
    }
```

Creating Statements

1010 LAB

- **`Statement`**
  - an interface that represents a SQL statement
  - You execute `Statement` objects, and they generate `ResultSet` objects, which is a table of data representing a database result set.
  - You need a `Connection` object to create a `Statement` object.

```
stmt = conn.createStatement();
```

- **There are three different kinds of statements:**
  - `Statement`: Used to implement simple SQL statements with no parameters.
  - `PreparedStatement`: (Extends Statement.) Used for precompiling SQL statements that might contain input parameters.
  - `CallableStatement`: (Extends PreparedStatement.) Used to execute stored procedures that may contain both input and output parameters.

Executing Queries

1010 LAB

- **`statement` object methods**
  - `execute`: Returns true if the first object that the query returns is a `ResultSet` object.
    - Use this method if the query could return one or more `ResultSet` objects.
    - Retrieve the `ResultSet` objects returned from the query by repeatedly calling `Statement.getResultSet`.
  - `executeQuery`: Returns one `ResultSet` object.
  - `executeUpdate`: Returns an integer representing *the number of rows affected by the SQL statement*.
    - Use this method if you are using INSERT, DELETE, or UPDATE SQL statements.

Create statement and execute

```
String st = "Select * From customer";


try {
    Statement sqlst = conn.createStatement();

    sqlst.execute(st);

    ResultSet rs = sqlst.getResultSet();

    while(rs.next()) {
        String rst = rs.getString(1);
        System.out.println(rst);
    }

} catch (SQLException e1) {
    // TODO Auto-generated catch block
    e1.printStackTrace();
}
```

Processing ResultSet Objects

1010 LAB

- **You access the data in a `ResultSet` object through a cursor.**
  - repeatedly calls the method `ResultSet.next` to move the cursor forward by one row.
  - Every time it calls `next`, the method outputs the data in the row where the cursor is currently positioned.

Using ResultSet

```
String query = "select COF_NAME, SUP_ID, PRICE, SALES, TOTAL from COFFEES";


try (Statement stmt = con.createStatement()) {
    ResultSet rs = stmt.executeQuery(query);
    while (rs.next()) {
        String coffeeName = rs.getString("COF_NAME");
        int supplierID = rs.getInt("SUP_ID");
        float price = rs.getFloat("PRICE");
        int sales = rs.getInt("SALES");
        int total = rs.getInt("TOTAL");
        System.out.println(coffeeName + ", " + supplierID + ", " + price +
                           ", " + sales + ", " + total);
    }
} catch (SQLException e) {
    JDBCTutorialUtilities.printSQLException(e);
}
```

Closing statements and Connections

- When you are finished using a Statement, call the method `Statement.close` to immediately release the resources it is using. When you call this method, its `ResultSet` objects are closed.
- Releases this `Connection` object's database and JDBC resources immediately instead of waiting for them to be automatically released.

```
 try {
    Statement sqlst = conn.createStatement();
    sqlst.execute(st);
    ResultSet rs = sqlst.getResultSet();
    while(rs.next()) {
         String rst = rs.getString(1);
         System.out.println(rst);
    }

    sqlst.close();
} catch (SQLException e1) {
    // TODO Auto-generated catch block
    e1.printStackTrace();
}

try {
    conn.close();
} catch (SQLException e) {
    // TODO Auto-generated catch block
    e.printStackTrace();
}
```

Insert a tuple

```
st = "INSERT INTO user (id, passwd) VALUES ('khhong', 'test')";
try {
    Statement sqlst = conn.createStatement();

    int i = sqlst.executeUpdate(st);
    System.out.println("Insert" + i);

    sqlst.close();
} catch (SQLException e1) {
    // TODO Auto-generated catch block
    e1.printStackTrace();
}
```

`ResultSet` interface

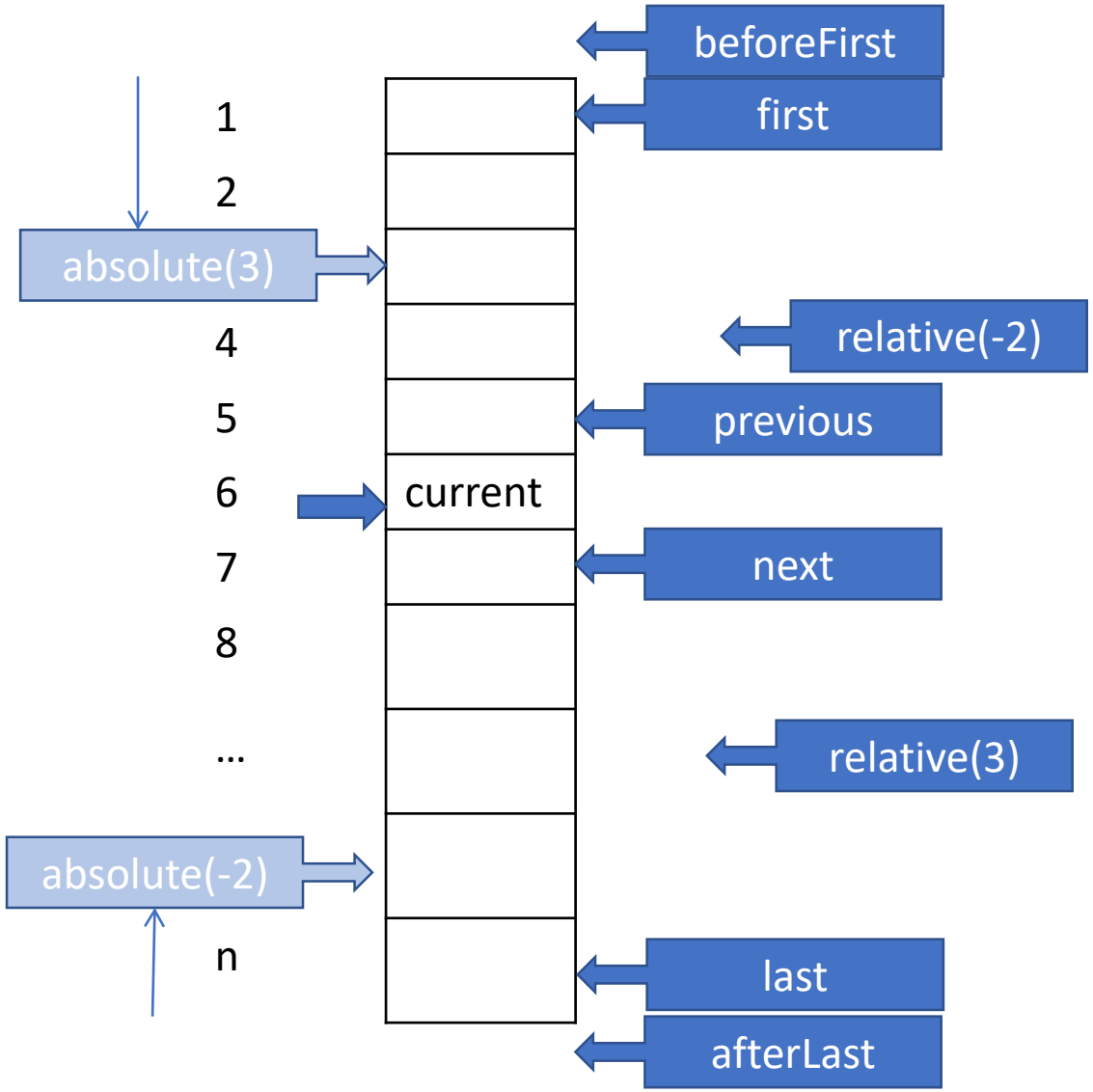
- **ResultSet objects can have different functionality and characteristics.**
  - type
  - concurrency
    - two concurrency levels:
      - CONCUR_READ_ONLY: The ResultSet object cannot be updated using the ResultSet interface. (default)
      - CONCUR_UPDATABLE: The ResultSet object can be updated using the ResultSet interface.
  - cursor *holdability*.

ResultSet Types

- **TYPE_FORWARD_ONLY: (default)**
  - The result set cannot be scrolled; its cursor moves forward only, from before the first row to after the last row. The rows contained in the result set depend on how the underlying database generates the results. That is, it contains the rows that satisfy the query at either the time the query is executed or as the rows are retrieved.
- **TYPE_SCROLL_INSENSITIVE:**
  - The result can be scrolled; its cursor can move both forward and backward relative to the current position, and it can move to an absolute position. The result set is insensitive to changes made to the underlying data source while it is open. It contains the rows that satisfy the query at either the time the query is executed or as the rows are retrieved.
- **TYPE_SCROLL_SENSITIVE:**
  - The result can be scrolled; its cursor can move both forward and backward relative to the current position, and it can move to an absolute position. The result set reflects changes made to the underlying data source while the result set remains open.

cursor move (ResultSet)

- next
- previous
- first
- last
- beforeFirst
- afterLast
- relative(int rows)
- absolute(int row)
- moveToInsertRow



cursor holdability

- **Calling the method Connection.commit can close the ResultSet objects that have been created during the current transaction. In some cases, however, this may not be the desired behavior..**
- **HOLD_CURSORS_OVER_COMMIT:**
  - ResultSet cursors are not closed; they are *holdable*: they are held open when the method commit is called. Holdable cursors might be ideal if your application uses mostly read-only ResultSet objects.
- **CLOSE_CURSORS_AT_COMMIT:**
  - ResultSet objects (cursors) are closed when the commit method is called. Closing cursors when this method is called can result in better performance for some applications.
- **The default cursor holdability varies depending on your DBMS.**
- **Note: Not all JDBC drivers and databases support holdable and non-holdable cursors.**

Retrieving Columns from Rows

```
public static void viewTable(Connection con) throws SQLException {
        Statement stmt = null;
        String query = "select id, passwd from user";

        try {
                stmt = con.createStatement();

                ResultSet rs = stmt.executeQuery(query);
                while (rs.next()) {
                        String uID = rs.getString(1);
                        String passwd = rs.getString(2);
                        System.out.println(uID + "\t" + passwd);
                }
        } catch (SQLException e) {
                // TODO Auto-generated catch block
                e.printStackTrace();
        } finally {
                if (stmt != null) { stmt.close(); }
        }

}
```

main() for testing

```
public static void main(String[] args) {
   Connection conn = null;
   Properties connectionProps = new Properties();
   connectionProps.put("user", "root");
   connectionProps.put("password", "");

   try {
      conn = DriverManager.
      getConnection("jdbc:" + "mysql" + "://" + "localhost" +
                  ":" + "3306" + "/"+"vrg1", connectionProps);
   } catch (SQLException e) {
     // TODO Auto-generated catch block
     e.printStackTrace();
   }

   System.out.println("Connected to database");

   try {
        databaseAccess.viewTable(conn);
   } catch (SQLException e1) {
       e1.printStackTrace();
   }
```

```
   try {
       conn.close();
   } catch (SQLException e) {
     e.printStackTrace();
   }

}
```

Updating Rows in ResultSet objects

1010 LAB

```
public static void modifyPasswd(Connection con, String uid, String newpasswd)
                                                        throws SQLException {
    Statement stmt = null;
    try {
        stmt = con.createStatement(ResultSet.TYPE_SCROLL_SENSITIVE,
                                   ResultSet.CONCUR_UPDATABLE);
        ResultSet uprs = stmt.executeQuery("SELECT * FROM user WHERE " + "id=" + "'"+ uid + "'");

        while (uprs.next()) {
            String oldpasswd = uprs.getString("passwd");
            uprs.updateString("passwd", newpasswd);
            uprs.updateRow();
        }

    } catch (SQLException e) {
        e.printStackTrace();
    } finally {
        if (stmt != null) {stmt.close();}
    }
}
```

Inserting Rows in ResultSet Objects

```
public static void insertRow(Connection con, String newId, String newpasswd)
                                                              throws SQLException {
        Statement stmt = null;
        try {
                stmt = con.createStatement(ResultSet.TYPE_SCROLL_SENSITIVE,
                                           ResultSet.CONCUR_UPDATABLE);
                ResultSet uprs = stmt.executeQuery("SELECT * FROM user");

                uprs.moveToInsertRow();

                uprs.updateString("id", newId);
                uprs.updateString("passwd", newpasswd);

                uprs.insertRow();
                uprs.beforeFirst();

        } catch (SQLException e) {
                e.printStackTrace();
        } finally {
                if (stmt != null) {stmt.close();}
        }
}
```

Using PreparedStatement

```
public static void insertWithParam(Connection con, String newId, String newpasswd)
                                        throws SQLException {
        PreparedStatement pstmt = null;
        try {

                con.setAutoCommit(false);

                pstmt = con.prepareStatement("INSERT INTO user VALUES( ?, ?)");
                pstmt.setString(1, newId);
                pstmt.setString(2, newpasswd);

                pstmt.executeUpdate();

                con.commit();

                con.setAutoCommit(true);
        } catch (SQLException e) {
                e.printStackTrace();
        } finally {
                if (pstmt != null) {pstmt.close();}
        }
}
```