JDBC

• JDBC (Java Database Connectivity) is a Java API for accessing databases
  – Early versions were JDBC-to-ODBC bridges
    • ODBC (Open Database Connectivity) uses a device driver model
    • Usually available as a C API
  – Newer JDBC versions are "pure Java"
JDBC

• Generally, any SQL statement can be executed using JDBC
  – Including DDL (data definition language) statements such as CREATE TABLE

• In most situations, though, the database schema is created manually and the applications use only DML (data manipulation language)
  – SELECT, INSERT, UPDATE, DELETE
Overview

1. Initialize driver
   - Only done once
   - Not necessary in JDBC 4.0 and newer
2. Open connection
3. Execute one or more statements
4. Use result set returned from executing statement
5. Commit or rollback transaction if autocommit is off
6. Close all resources (result set, statement, and connection)
Setup

- Old versions of JDBC required explicitly loading the driver class by name
  
  ```java
  Class.forName("org.postgresql.Driver");
  ```

- JDBC 4.0 and later no longer require this
  - But the appropriate jar must still be on the classpath
    - Each DBMS has its own implementation of JDBC
Connections

• Connection factory needs URL with
  – DBMS type
  – server name
  – port number
  – database name

• As well as username and password

    Connection conn = DriverManager.getConnection(
    "jdbc:postgresql://serverName:portNumber/databaseName",
    "username",
    "password");
Connections

• Very important to close connections!

Connection conn = DriverManager.getConnection("jdbc:postgresql://serverName:portNumber/databaseName", "username", "password");
try {
    // use connection
} catch (SQLException e) {
    // handle error
} finally {
    // very important to close the connection
    try {
        conn.close();
    } catch (Throwable th) {
        // it's annoying to always have to catch this
    }
}
Connection Pooling

• Opening a connection is an expensive operation
  – Most applications will use a connection pool
  • Can be tricky to implement: must be careful not to share connection between active threads
    – A thread should:
      » Get an unused connection (should keep the connection from being used by other threads)
      » Execute statements
      » Commit or rollback transaction
      » Release connection back to pool when done
Connection Pooling

• A thread should generally use only one connection at a time
  – A thread can self-deadlock if it starts using a second connection while holding open a transaction on another connection

• Third-party connection pooling libraries exist
  – C3P0
// -- create connection --

Statement stmt = conn.createStatement();
try {
    ResultSet rs = stmt.executeQuery("SELECT * FROM table WHERE ...");
    try {
        while (rs.next()) {
            // use result set
        }
    } finally {
        try {
            rs.close();
        } catch (Throwable th) {
        }
    }
} finally {
    try {
        stmt.close();
    } catch (Throwable th) {
    }
}

// -- close connection --
Closing Resources

• All resources, such as Statement and ResultSet, should be closed as soon as possible after use
  – Free up resources on both client and server
  – An active transaction could be holding locks on the database

• If autocommit is off then the transaction remains open until a commit or rollback or the connection is closed

• If autocommit is on then the transaction completes when the statement completes
  – For SELECT statements, this does not happen until the ResultSet is closed!
Transactions

// -- create connection --

conn.setAutoCommit(false);
Statement stmt = conn.createStatement();
try {
    int count = stmt.executeUpdate("INSERT INTO table ...");
    conn.commit();
} catch (SQLException e) {
    conn.rollback();
}
finally {
    try {
        stmt.close();
    } catch (Throwable th) {
    }
}

// -- close connection --
executeUpdate()

- INSERT, UPDATE, DELETE and DDL statements (e.g., CREATE TABLE) use `executeUpdate()` method
  - Returns the number of records affected

```java
int count = stmt.executeUpdate("INSERT INTO table ...");
```
executeQuery()

- SELECT statements use the executeQuery() method
  - Returns a ResultSet which can be iterated over to get the matching records

```java
ResultSet rs = stmt.executeQuery("SELECT * FROM table WHERE ...");
try {
    while (rs.next()) {
        // use result set
    }
} finally {
    try {
        rs.close();
    } catch (Throwable th) {
    }
}
```
ResultSet

• A ResultSet object can be viewed as allowing access to a set of rows
  – Can picture the ResultSet as having an internal cursor which points to the "current" row
  – Starts out initialized to before the first row
    • Must call next() to move to the first row before accessing it
    • next() will return false if there are no more rows
ResultSet rs = stmt.executeQuery("SELECT * FROM table ...");
   try {
      while (rs.next()) {

         // use result set

      }
   } finally {
      try {
         rs.close();
      } catch (Throwable th) {
      }
   }