**JDBC**

- JDBC (Java Database Connectivity) is a standard SQL database access interface.
- The JDBC API defines classes to represent constructs such as database connections, SQL statements, result sets, and database metadata.
- JDBC allows a Java program to issue SQL statements and process the results.

**JDBC Goals**

- To provide
  - Java programmers with a uniform, simple interface to a wide range of relational databases
  - DB independence. Can replace underlying database with minimal code impact.
  - a common base on which higher level tools and interfaces can be built.
Trivial First Example

- **Trivial.java**
  - Compile
  - Execute
    - Error!
  - Create an ODBC entry
  - Execute
    - see the result

A JDBC Application

[Diagram showing the two layers of the JDBC Architecture]
Pieces Continued

• A JDBC application consists of
  • Java client: code implementing the application
  • JDBC API. Provides DB independent abstraction to
    • establish a connection with a database
    • send SQL statements
    • process the results

• JDBC Driver
  • Translates API calls to requests made against the specific database.
  • Specific driver for the specific database.
  • Installed on the client. Usually a set of class files placed in the class path.
  • All large databases are now supported.

• Database client software (optional)
  • Establishes communication between DB clients and the DB server.
  • Depending on the DB Manufacturer, you may need to install database client software on the client machine.
  • JDBC Driver may already include this functionality
Pieces Continued

- Database server:
  - The actual database engine
  - Oracle, MSAccess, SQL Server, Postgresql etc.

Alternatives (ODBC)

- What is ODBC
  - Microsoft's version of JDBC.
  - Many drivers exist for ODBC.
  - Sun provides a JDBC-ODBC driver to allow Java applications connectivity to databases that are only ODBC enabled.
API: Establish Connection

1. `Class.forName(...)` make the driver class available

2. `String url = "jdbc:odbc:supplier-part";`  
   This is the connect string. The connect string mentions the driver as well as the database. For the example above, the driver is the `jdbc:odbc` bridge driver. The database is `supplier-part`.

3. `Connection con=DriverManager.getConnection(url, uID, pw);`  
   Get a connection (session) with a specific database. Within the context of a Connection, SQL statements are executed and results are returned.

   A Connection can be used to obtain information about the DB. By default a Connection automatically commits changes after each statement.

   Typically, setting up a connection is an expensive operation.

---

API: Executing Queries

- A query can return many rows, each with many attributes

- Steps are
  1. Send query to the database
  2. Retrieve one row at a time
  3. For each row, retrieve attributes
API: Executing Queries

Example:
Statement stmt = con.createStatement();

// Send the query to the DB, get back a ResultSet
ResultSet rs = stmt.executeQuery("SELECT * FROM PART;");

// Go through all rows returned by the query
while(rs.next()){
    // Pull out individual columns from the current row
    int pno=rs.getInt("PNO");
    String pname=rs.getString("PNAME");

    // Print out the values
    System.out.println(pno+"\t"+pname);
}
rs.close();

API: Updates

Int rowsEffected=
    stmt.executeUpdate(
        "DELETE * FROM ACCOUNTS;");

Executes a SQL INSERT, UPDATE or DELETE statement. Returns the number of rows affected.
## API: Prepared Statements

- Is a parameterized SQL statement.
- Used to speedup query parsing (statement does not need to be reparsed).
- Used to simplify development (clumsy strings do not have to be created and recreated for each update).

**Example:**

```java
String insert = "INSERT INTO ACCOUNT(NAME,AMOUNT)VALUES(?,?);";
PreparedStatement ps = con.prepareStatement(insert);
ps.setString(1,"Charlie"); // Fill in the first ?
ps.setDouble(2,23.45); // Fill in the second ?
rowsEffected = ps.executeUpdate();
ps.setInt(1,"Arnold"); // Fill in the first ?
ps.setInt(2,102.23); // Fill in the second ?
rowsEffected = ps.executeUpdate();
```

## Transactions

- A transaction is a collection of DB modifications, which is treated as an atomic DB operation. All take place or none do.
- Transactions are used to make sure that a collection of updates leaves the database in a consistent state (as defined by the application program).
- By default the Connection automatically commits changes after executing each statement. If auto commit has been disabled, the method commit must be called explicitly; otherwise, database changes will not be saved.
API: Transactions

Example:

// Change to transactional mode
con.setAutoCommit(false);

// Transaction A begins here
stmt.executeUpdate("DELETE * FROM ACCOUNT..."); // 1
stmt.executeUpdate("INSERT INTO ACCOUNT ...."); // 2
stmt.executeUpdate("INSERT INTO ACCOUNT ...."); // 3
stmt.executeUpdate("INSERT INTO ACCOUNT ...."); // 4
con.commit();

// Commit changes to database
// All of 1,2,3,4 take effect

API: Transactions

Example:

// Transaction B begins here
stmt.executeUpdate("DELETE * FROM SALES..."); // 5
stmt.executeUpdate("INSERT INTO SALES ...."); // 6
stmt.executeUpdate("INSERT INTO SALES ...."); // 7
stmt.executeUpdate("INSERT INTO SALES ...."); // 8
con.rollback();

// Rollback to before transaction B began
// None of 5,6,7,8 effects the DB

- Example: Transaction.java
Scope of DB Connections

1. **Servlet**: Open/close connection on each servlet invocation
   - Will work, but very slow.
2. **Session**: Keep 1 open connection associated with the session: HttpSession.getParameter("dbCon")
   - Slow
   - Dangerous (multiple concurrent servlets in same session): concurrency problems
   - Awkward (when to close?)
3. **Web App**: Keep 1 open connection for the web application and re-use it
   - Dangerous: Concurrency issues: bottlenecks

Connection Pooling

- **The Solution**:  
  - Maintain a pool of open connections that know how to time themselves out
  - Each servlet calls:
    
    ```java
    DbConnectionPool dbcp = getServletContext().getAttribute("dbConPool");
    DbConnection con = dbcp.checkoutConnection();
    ...
    Statement s = con.createStatement();
    ...
    dbcp.checkinConnection(con);
    ```
**Connection Pooling**

- The `DbConnectionPool` allocates a new `Connection` object when asked to do so.
  - It wraps it in a `DbConnection` object that delegates calls to its enclosed `Connection` object and returns it to the client.
  - It sets a timeout that will reclaim the `Connection` for a pool of free connections
    - If client accesses `DbConnection` after timeout, `DbConnection` will request a fresh connection from the pool.
- Client returns `DbConnection` to the pool when done with it.
  - `DbConnectionPool` does not close the `Connection`, rather saves it for the next request
- Client may block awaiting a freed connection if some maximum upper limit of open connections is reached.

**EJBs**

- **Enterprise Java Beans**
  - Part of Sun J2EE initiative
- **Goal:**
  - Ease development of Web-oriented, 3-tier, RDBMS-based, applications.
    - Entity bean:
      - Encapsulates a persistent object that is mirrored in an RDBMS
    - Session bean:
      - Encapsulates transactions
- **Benefits:**
  - Handles all the crap for you (e.g., sessions, connection pooling, ...)

52 - JDBC  
CSC309  
19