Advanced Java Programming

JDBC

v2.1

What is JDBC?

• The JDBC API is the Java Data Access API

• JDBC is not an acronym, although it is sometimes referred to as “Java Database Connectivity”

• JDBC is used for accessing virtually any kind of tabular data, especially that stored in relational databases

• The JDBC API consists of a set of classes and interfaces that allow developers to write database applications

Introduction

• JDBC allows Java to execute SQL statements on relational database systems

• All SQL-92 & SQL-99 are supported
  – Also supports some vendor specific SQL eg: oracle XQUERY

• Most database vendors have driver support:
  – Oracle, IBM DB2, Sybase, MS SQL Server, Informix, Ingres, Mysql, Postgres, SQLite, embedded db
  – Otherwise use JDBC-ODBC bridge driver

Topics

• JDBC 1.0 Introduction
  – Drivers
  – Connections
  – Statements
  – Result Sets
  – Prepared Statements
  – Stored procedures

• JDBC 2.0
  – Scrollable result sets
  – Row sets
  – Connection pools
  – Data sources

JDBC & J2EE

• JDBC allows Java to execute SQL statements on relational database systems

• All SQL-92 & SQL-99 are supported
  – Also supports some vendor specific SQL eg: oracle XQUERY

• Most database vendors have driver support:
  – Oracle, IBM DB2, Sybase, MS SQL Server, Informix, Ingres, Mysql, Postgres, SQLite, embedded db
  – Otherwise use JDBC-ODBC bridge driver

JDBC versions

• The JDBC API specification is an evolving API

• JDBC 1.0 was originally an option to JDK 1.0 → java.sql packages

• JDBC 2.0 was included as part of Java 2 (JDK 1.2). → javax.sql packages

• JDBC 3.0 is latest version with Java SE 5
## JDBC & Java EE

- JDBC is mostly independent of Java EE
  - Built into default JDK
- You need to use Web Application server for advanced features such as
  - connection pooling, datasources
  - distributed transactions
- Other Java API's do similar things to JDBC
  - Entity EJB \( \rightarrow \) *part of Java EE 5*
  - Java Data Objects (JDO)
  - Hibernate
  - Java Persistence API (JPA) \( \rightarrow \) *part of Java EE 5*

## java.sql * API

- DriverManager, Driver
  - Call vendor driver to get connections to the database
- Connection
  - used to connect to the database
- Statement
  - used to send SQL statements to the database
- ResultSet
  - used to retrieve results from the database
- SQLException, SQLWarning
  - used to throw errors or warnings from the database

### What Does the JDBC API Do?

- A JDBC class generally:
  1. Loads a JDBC driver
  2. establishes a connection to the database
  3. executes SQL statements
  4. processes the results
  5. Releases resources

```java
Connection con = DriverManager.getConnection(" ... ");
Statement stmt = con.createStatement();
ResultSet rs = stmt.executeQuery("SELECT * FROM T1");
while (rs.next()) {
  x = rs.getString("1"); // first column of result table
}
rs.close()
```

### JDBC Drivers

- To use the JDBC API a Java program must load a JDBC driver
- A JDBC driver is a java library that implements the interfaces of the java.sql package
- JDBC drivers are provided by third parties (usually the database vendor). The java.sql package provided with the JDK defines just the interfaces and some key classes, such as the DriverManager class

```java
Class.forName("sun.jdbc.odbc.JdbcOdbcDriver");
```

### JDBC Drivers

- You need to load a JDBC driver
  - Use the \(-Djdbc.driver=\) class java option
  - Use the Java classloader: Class.forName(\(\)class\)
- eg: JDBC-ODBC Bridge driver:

```java
Class.forName("sun.jdbc.odbc.JdbcOdbcDriver");
```

- Eg: Oracle driver:

```java
Class.forName("oracle.jdbc.driver.OracleDriver");
```
Classification of JDBC Drivers

Sun define 4 distinct types of JDBC driver

- **Type 1 - JDBC-ODBC Bridge**
  - (Driver not pure Java, connects to local ODBC DB)
  - Type 1 provides JDBC access via ODBC drivers
  - Supplied with base JDK
  - ODBC drivers are readily available
  - Suitable where client installations are not a problem
  - Cons:
    - ODBC binary code must be loaded on each client
    - Performance is degraded - not suitable for large-scale applications

- **Type 2 - Native API**
  - (Driver not pure Java, connects to local DB)
  - Type 2 converts JDBC calls into client API (DLL or shared library)
  - Requires that some operating system-specific binary code be loaded on client machine
  - Not suitable for the Internet
  - Licensing issues

- **Type 3 - JDBC-NET**
  - (Driver may be pure Java, connects to DB via network)
  - Type 3 translates JDBC calls into a database independent net protocol
  - Not suitable for the Internet

- **Type 4 - Native Database Protocol**
  - (Driver pure Java, connects to DB via network)
  - Type 4 is better performance than the JDBC-ODBC bridge
  - Suitable for a corporate LAN
  - Cons:
    - Requires that some operating system-specific binary code be loaded on client machine
    - Cannot be used for the Internet
    - Licensing issues
    - Slow compared to type 3 & 4 drivers (next slides)
Classification of JDBC Drivers

- Type 3 - JDBC-Net pure Java driver
  - pros
    - middleware can connect to different databases
    - pure Java driver; small and fast to load
    - extremely flexible
    - provides (typical) caching, load balancing, logging etc.
    - same driver regardless of the underlying database
  - cons
    - database-specific coding done in the middle tier
    - can hurt performance - extra indirection
    - Need extra server in the middle

- Type 4 - Native-protocol pure Java driver
  - converts JDBC calls into the network protocol used by the database vendor
  - pros
    - direct call from client machine to the database server
    - performance (typically) is good - better than types 1 & 2
    - no need to install special software on the client
    - often used in distributed applications
  - cons
    - user needs a different driver for each database - tight coupling

- Best drivers to use are types 3 & 4

Connections

- Once a JDBC driver has been loaded, the DriverManager class can be used to create and manage database connections

Connection objects are created using the DriverManager.getConnection() method.
- This method is overloaded - all versions must specify the JDBC driver along with details of the database to be accessed and a username and password

General syntax:
- DriverManager.getConnection( url, "userid", "password" );
- url - see next slide

JDBC driver URL

- The format of the URL
  - protocol:sub-protocol:database_name,parameters

- Has three parts
  - Protocol: jdbc is the protocol
  - Sub-protocol: the name of a database connectivity mechanism
    - odbc, oracle, db2 etc
  - Database_name: driver-specific; a driver may choose any appropriate syntax
### JDBC driver URL

- Examples for JDBC URLs
  - JDBC-ODBC bridge
    - `jdbc:odbc:myDSN` where `myDSN` is the name of the ODBC DSN
  - MySQL
    - `jdbc:mysql://dragon.it.uts.edu.au:3306/myDB`
  - Cloudscape
    - `jdbc:cloudscape:Movies;autocommit=false;create=true`
  - Oracle
    - `jdbc:oracle:thin:@dbserver:1522:myDB`

### Connecting to an Oracle DB

- Getting a connection to an Oracle database:
  - Step 1
    - load (register) the driver
    ```java
    Class.forName("oracle.jdbc.driver.OracleDriver");
    ```
  - Step 2
    - get connection
    ```java
    Connection connection = DriverManager.getConnection("jdbc:oracle:thin:@smaug.it.uts.edu.au:1522:ell", "user", "pw");
    ```

### Connections

- Connections represent the client’s connection to the database
- Has various methods to interact with database
  - factory methods such as
    - `createStatement()`, `prepareStatement()`, `prepareCall()`
    - creates Statement objects which hold the SQL
  - transaction commands: `commit()`, `rollback()` etc
  - get metadata about the database

### Statements – Executing SQL

- Get a statement object from your connection
  ```java
  Statement stmt = connection.createStatement();
  ```
- Use the various `execute()` methods to run SQL
  - Use `executeQuery()` for read only
    ```java
    eg: SELECT
    ```
  - Use `executeUpdate()` for read/write
    ```java
    eg: INSERT, UPDATE, DELETE
    ```
  - Use `execute()` if unknown...
    ```java
    eg: dynamically generated SQL
    ```
  - Use `executeBatch()` to do multiple SQL
    ```java
    eg: for batch inserts
    ```

### Result Sets

- If you use `executeQuery()`, the query results are stored in a `ResultSet` object
- This is like a table with columns and rows
- Retrieve data by using various `getXXX()` methods where `XXX` = SQL type eg: for CHAR column, use `getString()`
  - you can retrieve by column number eg: 1
  - or by column name eg: “address”
- You can navigate the table via “cursor” methods such as
  - `first()`, `next()`, `previous()`, `absolute()`, `relative()`
- Use informational methods such as
  - `getMetaData()`, `getResultSet()`, `getMoreResults()`

### Example – Executing a query

```java
// create statement
Statement stmt = connection.createStatement();

// execute query
ResultSet rs = stmt.executeQuery("SELECT NAME FROM INVENTORY");

// display some query results
while (rs.next()) {
    System.out.println("Item name "+ rs.getString(1));
}
```
Prepared Statements

- Regular Statement objects represent an SQL statement that must be compiled each time they are executed.

- It is often more convenient or more efficient to use a PreparedStatement object for sending SQL statements to the database. This special type of statement is derived from the Statement class.

- PreparedStatement address the following requirements:
  - parameters can be substituted dynamically
  - values cannot always be represented as character strings
  - avoids repeated compiling of the same SQL statements

As with Statement objects, you create PreparedStatement objects with a Connection method.

- This example creates a PreparedStatement object that takes two input parameters:

```java
PreparedStatement updateSales = con.prepareStatement("UPDATE COFFEES SET SALES = ? WHERE COF_NAME = ?");
```

- The PreparedStatement contains an SQL statement which has, in most cases, been sent to the database and been precompiled.

Values to be used in place of any question mark placeholders need to be supplied before a PreparedStatement object can be executed.

- This is done by calling the setxxx methods.

For example, the following line of code sets the first question mark placeholder to an int with a value of 75:

```java
updateSales.setInt(1, 75);
```

PreparedStatement vs Statement

```java
PreparedStatement update = connection.prepareStatement("UPDATE COFFEES SET SALES = ? WHERE NAME = ? ");
update.setInt (1, 75);
update.setString (2, "Colombian");
updateSales.executeUpdate();
```

```java
Statement stmt = connection.createStatement();
String update = "UPDATE COFFEES SET SALES = 75 WHERE NAME = 'Colombian'";
stmt.executeUpdate(update);
```

SQL Escape sequences

- Inside the SQL statement you can use special JDBC only extensions.

- General syntax: `{ keyword }

- `{d '2009-05-01'}` represents a date

- `{fn function}` calls a function eg: `sqrt()`

- `{escape \\}` escapes the wildcard in LIKE expression

- `{oj outer-join-sql-statement}` does outer joins

Stored Procedures

- A stored procedure is a program written to run within the database server as a single logical unit. This may be a general language (like Java) or database specific (like Oracle PL/SQL). Use an escape sequence to define the call.

- use the `Connection.prepareCall("string")` to create the `CallableStatement` class

- If the stored procedure does not return results, use the `"(call proctype(?,?))"` string.

- If results are returned, use `"(? = call proctype(?,?))"` string
**Stored Procedures**

- As per preparedStatements:
  - use the setxxx() methods to set the parameters
  - use getxxx() methods to retrieve the result columns as usual

- eg:
  ```java
  CallableStatement cs = con.prepareCall("call SHOW_SUPPLIERS(?)");
  cs.setString(1,"Columbian");
  ResultSet rs = cs.executeQuery();
  ```

  ps: use the registerOutParameter() method if you have OUT parameters in your call

**Advantages over embedded SQL**

- precompiled - faster similar to PreparedStatement
- syntax errors caught at compile time
- developers need to know name of the procedure and its inputs only - tables and their structures are unimportant

**Cons**

- variations in the syntax and capabilities
  - stored procedure written in Java will work with database that implements SQLJ specification

**JDBC Metadata**

- It is sometimes necessary to get information about the database you are connected to

- Use `DatabaseMetaData o = Connection.getMetaData()`
- Returns:
  - info about database, table schemas, table descriptions, database driver details, level of SQL support, stored procedure listings and many other database parameters

**Advanced Result Sets**

- Use the alternative `Connection.getStatement()` method to get these advanced resultSets
  - eg: default
    ```java
    rs = conn.createStatement(sql)
    ```
  - eg: scrollable result set
    ```java
    rs = conn.createStatement(sql, ResultSet.TYPE_SCROLL_INSENSITIVE)
    ```
  - eg: updatable result set
    ```java
    rs = conn.createStatement(sql, ResultSet.TYPE_SCROLL_INSENSITIVE, ResultSet.CONCUR_UPDATABLE)
    ```

**Scrollable Result Sets**

- Forward-only **default**
  ```java
  ResultSet.TYPE_FORWARD_ONLY
  ```
- Scroll-insensitive
  ```java
  ResultSet.TYPE_SCROLL_INSENSITIVE
  ```
- Scroll-sensitive
  ```java
  ResultSet.TYPE_SCROLL_SENSITIVE
  ```

**Updateable Result Sets**

- Read-only **default**
  ```java
  ResultSet.CONCUR_READ_ONLY
  ```
- Updateable
  ```java
  ResultSet.CONCUR_UPDATABLE
  ```

**Advanced Result Sets**

- By default, Result Sets represent SQL cursors
  - the default ResultSet only can scroll forwards.
- JDBC has 2 optional ResultSet extensions:
  - **Scrollable Result Sets**
  - **Updateable Result Sets**

  ```java
  res = conn.createStatement(sql, ResultSet.TYPE_SCROLL_INSENSITIVE, ResultSet.CONCUR_UPDATABLE)
  ```

**Scrolled Result Sets**

- Forward-only → result sets allow only ResultSet.next()

- Scroll-insensitive
  - result set allows full navigation
  - changes to underlying rows not reflected

  - CAUTION: causes table locking!!!
Updateable Result Sets

- Updateable result sets allow for the update, insertion and deletion of rows in the database using the result set itself.

- ResultSets can be further classified into 2 types as a result of this:
  - Read-only (ResultSet.CONCUR_READ_ONLY)
  - Updateable (ResultSet.CONCUR_UPDATABLE)

Example– Updateable ResultSet

```java
String sql = "SELECT * from CUSTOMER";
// return scroll-sensitive, updateable result set
PreparedStatement ps = connection.prepareStatement
                        ( sql,
                          ResultSet.TYPE_SCROLL_SENSITIVE,
                          ResultSet.CONCUR_UPDATABLE );
ResultSet rs = ps.executeQuery();
rs.next();
rs.updateString ("FIRST_NAME", "Tony");
rs.updateRow();
```

Row Sets

- `javax.sql.RowSet` similar to ResultSet
  - RowSet interface extends the ResultSet interface
  - Acts like a JavaBean (use get/set methods on column names!)

- Implementations
  - JdbcRowSet – JavaBeans on top of ResultSet
  - CachedRowSet – extends JdbcRowSet, caches its data in memory, can run disconnected to database
  - WebRowSet – extends CachedRowSet, has XML interfaces

Transaction support

- JDBC supports simple SQL units of work
- Use the following connection object methods:
  - `setAutoCommit(True/False)`
    - defaults to True ie: each SQL will update underlying database
  - `commit()` – update the changes
  - `rollback()` – undo the changes
- See the Transactions lecture later for more details

Batch Updates

- For efficiency, you can have more than 1 SQL statement to be executed
- Use statement `addBatch(sql)`
- Execute batch via statement `executeBatch()`
- Use `commit()` / `rollback()` to control logical units of work
Example – Batch Update

// only commit changes when asked to
collection.setAutoCommit(false);

statement.addBatch("insert into state values ('NSW', 'New South Wales')");
statement.addBatch("insert into state values ('QLD', 'Queensland')");

// perform batch update
int[] nResults = statement.executeBatch();

// commit changes
connection.commit();

Exceptions

• java.sql classes throw SQLException on errors
• Methods to use:
  – getMessage() - returns the error message
  – getSQLState() – returns SQL standard error state
  – getErrorCode() – returns vendor error code
  – getNextException() – returns the next SQLException (if there are more than one)
• Use Statement.getWarnings() to get SQLWarnings.
  • This object returns non-fatal SQLStates and the statement will continue executing

Advanced JDBC

• The following classes require you to have an application server (such as Tomcat or Weblogic) to provide container services for the implementation

Connection Pools

• Pooling of resources in general is very important for enterprise systems
• The basic concept of a resource pool is a collection of resources that are created at system start up and maintained through the life of the system. Clients request a resource from the pool when they need to make use of it, then return it to the pool when they are finished
• Resource pooling is done in order to make a system scalable and able to support large numbers of users
• Database connections are one resource that is especially useful to pool, as database connections can take a long time to establish and then tear down

Data Sources

• The javax.sql.DataSource interface is generally implemented by application server providers
• Applications running within an application server can look up DataSources using JNDI (JNDI is explained later)
• The DataSource itself is created by the application server – the creation of the DataSource and the size and configuration of the underlying connection pool is done as part of the application server configuration
• Applications that have obtained a DataSource can then get a database connection by calling the DataSource.getConnection() method
Example– Data Source

```java
import java.sql.*;
import javax.sql.DataSource;
import javax.naming.*;

// initial contexts are a JNDI concept explained later
Context ctx = new InitialContext();

// get the data source
DataSource ds = (DataSource) ctx.lookup("myDataSource");

// open a connection
Connection conn = ds.getConnection();

// & do normal jdbc stuff from here on
```

New Developments

- **JDO (Java Data Objects)**
  - JDO is a new API for transparent database access
  - It provides an object-relational mapping layer that developers can use to transparently access underlying data stores without using database-specific code

- **JPA (Java Persistence API)**
  - New API for Object-Relational Mapping
  - Replaces the Entity Enterprise Java Bean in Java EE 5
  - Implemented by various existing ORM frameworks
    - Example: Hibernate, Toplink/EclipseLink, Kodo etc

- **JDBC 4.0** – (Java SE 6)
  - Removes need for class.forName(Driver), now DriverManager automatically finds them
  - Event listeners
  - Streaming API (set/update methods using IOStreams)

Best Practices for JDBC

- Prefer DataSource over DriverManager for establishing a connection
  - No need to hardcode driver information as with the DriverManager
  - DataSource takes advantage of connection pooling and distributed transactions
- Release the JDBC resources appropriately
  - Use finally block (try { ... })
- If possible use explicit commits
- Use batch updates if possible
- Set the min connection pool size to max (WLS)
  - Minimize response time by creating connections at server startup

Summary

- JDBC offers a simple, database-independent means for establishing database connections, executing SQL statements, manipulating query result data and obtaining metadata about the database itself

- Most of the core JDBC features were presented, along with some advanced features

- Later we look at a higher-level mechanism for accessing database data – EJB Entity Beans
  - Here we've considered "pure" JDBC independent of app servers

- Later we also look at JDBC Distributed Transactions