JDBC

Objectives

- Learn (a little) about relational database concepts
- Learn (a little) about SQL
- Learn to use the Java database API – JDBC

SQL and Relational Databases

SQL Primer

- Two types of commands
  - Data Definition
    - Create Table
    - Drop Table
  - Data Manipulation
    - Select
    - Insert
    - Update
    - Delete

RDB Concepts

- Tables
  - A defined set of columns
  - Having zero or more rows of data
- Primary-Key
  - Those attributes whose values uniquely identify one row from all others
- Foreign-Key
  - Those columns in one table which hold the primary-key from another table
  - Define relationships between rows in different tables

RDB Concepts

- Index
  - A structure providing rapid access to the rows of a table based on the values of its ‘indexed’ columns
  - Applied to the columns which are frequently used to identify the rows of interest
  - Always created for columns in primary-key

Constraints

- Constraints
  - Constrain the values that may be placed in a table
    - `PRIMARY KEY`
      - Prohibits duplicate values in this column(s) and disallows `NULL`
    - `UNIQUE KEY`
      - Prohibits duplicate values in this column(s)
    - `FOREIGN KEY`
      - Identifies the column(s) migrated as a foreign key and the table migrated from
      - Allows optional specification of a delete rule

The Java Database API: JDBC

JDBC

- Not an acronym
  - Misinterpreted to be Java Database Connectivity
- A set of classes defined in the `java.sql` and `javax.sql` packages
  - Provides a standard mechanism for accessing relational databases that support Structured Query Language (SQL).
  - Provides support for ODBC through a JDBC-ODBC bridge.
  - Other JDBC drivers are preferred

JDBC Driver Types

- Type 1
  - JDBC-ODBC Bridge Technology
- Type 2
  - JNI drivers for C/C++ connection libraries
- Type 3
  - Socket-level Middleware Translator
- Type 4
  - Pure Java-DBMS driver

Type 1 Drivers

- JDBC-ODBC Bridge
  - Translates call into ODBC and redirects to the DBMS's ODBC driver
  - ODBC must exist on every client
  - Slow – due to translation

Type 2 Drivers

- Native API
  - Java driver makes JNI calls on the DBMS API (usually written in C or C++)
  - Requires client-side code libraries
  - Provided by DBMS vendor
  - Can crash JVMs
  - Fast

Type 3 Drivers

- Middleware Pure Java Driver
  - Process:
    1. JDBC driver translates JDBC calls into a DBMS-independent protocol
    2. Communicates over a socket with a middleware server that translates into native API DBMS calls
  - Single driver provides access to multiple DBMSs
  - No client code need be installed

Type 4 Drivers

- Pure Java Drivers
  - Driver talks directly to the DBMS using Java sockets
  - No Middleware layer needed
  - No client code need be installed

Establishing a Connection

- Connecting to a JDBC Data Source requires:
  - Loading the JDBC driver class
  - Connecting to the data source
  - Specify database as a URL
    - A database URL has the form:
      `jdbc:`*`subprotocol`*`:`*`subname`*

Loading Drivers

- Three approaches:
  1. Using `DataSource` class and JNDI
     - Preferred method
  2. Identify driver classes using system properties
     - When naming service isn't available
  3. Do nothing, rely on Service Provider mechanism
     - JDBC 4.0/Java SE 6
     - For quick and dirty development
     - Least desirable
  4. Load driver class explicitly
     - Pre JDBC 4.0/Java SE 6

Using JNDI

- **Configure** `DataSource` **in JNDI, then**
  - **Uses JNDI and** `javax.sql.DataSource`

```
Connection conn = null;
InitialContext ctx = new InitialContext();
DataSource ds = (DataSource)ctx.lookup( "jdbc/"+"mySrc" );
try
{
   conn = ds.getConnection();
}
catch( SQLException ex )
{
   ...
```

Using Properties

- Set `jdbc.drivers` property

  `-Djdbc.drivers=com.mysql.jdbc.Driver`

```
Connection conn = null;
String db = "jdbc:mysql://localhost/EmployeeDB";
String user = "student";
String pass = "student";
try
{
   conn = DriverManager.getConnection( db, user, pass );
}
catch( SQLException ex )
{
   ...
```

Rely on Service Provider

- The `DriverManager` "discovers" driver
  - Driver identified in jar file:

    `META-INF/services/java.sql.Driver`

```
Connection conn = null;
String db = "jdbc:mysql://localhost/EmployeeDB";
String user = "student";
String pass = "student";
try
{
   conn = DriverManager.getConnection( db, user, pass );
}
catch( Exception ex )
{
   ...
```

Using Explicit Driver

- **Load the driver manually**

```
Connection conn = null;
String db = "jdbc:mysql://localhost/EmployeeDB";
String driverClassName = "com.mysql.jdbc.Driver";
String user = "student";
String pass = "student";
try
{
     Class.forName( driverClassName );
     conn = DriverManager.getConnection( db, user, pass );
}
catch( Exception ex )
{
   ...
```

Using the Connection

- The `Connection` class provides methods for:
  - Creating statements for execution
  - Controlling the behavior of the connection.
- Virtually all JDBC methods throw `SQLException`

Connection Methods

```
Statement createStatement()
PreparedStatement prepareStatement(String sql)
CallableStatement prepareCall( String sql )
void setAutoCommit( boolean autoCommit )
boolean getAutoCommit()
void commit()
void rollback()
void close()
```

Statement

- **Once the statement is created it may be used to execute SQL.**

```
ResultSet executeQuery( String sql )
int executeUpdate( String sql )
```

Batch Processing

- Allows processing of multiple statements in one request
  - New in JDBC 2.0

```
void addBatch( String sql )
Void clearBatch()
int[] executeBatch()
```

Executing a Query

```
Connection conn = null;
String db = "jdbc:mysql://localhost/EmployeeDB";
String username = "student";
String password = ”student";
conn = DriverManager.getConnection( db, user, pass );


Statement stmt = conn.createStatement();
String query = "SELECT employee_name, salary"
             + "  FROM employee"
             + " ORDER BY employee_name";
ResultSet rs = stmt.executeQuery( query );
```

ResultSet

- Provides a mechanism for accessing the results of a query. Provides methods for:
  - Moving through the resultant records.
  - Getting the values out the records.
  - Closing the ResultSet.

```
boolean next()
void close()
ResultSetMetaData getMetaData()
```

ResultSetMetadata

- Provides a wide variety of methods for obtaining information about the `ResultSet`, the following are just a few.

```
int getColumnCount()
String getColumnName(int column)
int getColumnType( int column )
boolean wasNull()
```

Getting Values

- Get methods are provided for obtaining values from a record, all the "standard" SQL types are supported.
- Each method:
  - Accepts a single argument
    - Column index (indexes are 1-based, not 0-based)

    -or-
    - Column name
  - Returns the appropriate Java type

SQL-Java Type Mapping

- Methods for retrieving SQL types

SQL Type	Java Type	Method	2.0
ARRAY	Array	getArray	x
BIGINT	long	getLong	
BINARY	byte[]	getBytes	
BIT	boolean	getBoolean	
BLOB	java.sql.Blob	getBlob	x
CHAR	String	getString	
CLOB	java.sql.Clob	getClob	x
DATE	java.sql.Date	getDate	
DECIMAL	java.math.BigDecimal	getBigDecimal	
DOUBLE	double	getDouble	
FLOAT	double	getDouble	
INTEGER	int	getInt	

SQL-Java Type Mapping

SQL Type	Java Type	Method	2.0
LONGVARBINARY	java.io.InputStream	getBinaryStream	
LONGVARCHAR	java.io.InputStream	getAsciiStream getUnicodeStream	
NUMERIC	java.math.BigDecimal	getBigDecimal	
REAL	float	getFloat	
REF	java.sql.Ref	getRef	x
STRUCT	java.sql.Struct	getObject	x
SMALLINT	short	getShort	
TIME	java.sql.Time	getTime	
TIMESTAMP	java.sql.Timestamp	getTimestamp	
TINYINT	byte	getByte	
VARBINARY	byte[]	getBytes	
VARCHAR	String	getString	
<user-defined>	Object (optional map)	getObject	x

Using ResultSet

```
Connection conn = null;
String db = "jdbc:mysql://localhost/EmployeeDB";

String username = "student”;
String password = "student";


conn = DriverManager.getConnection( db, user, pass );
Statement stmt = conn.createStatement();
String query = "SELECT employee_name, salary"
             + "  FROM employee"
             + " ORDER BY employee_name";
ResultSet rs = stmt.executeQuery( query );
while( rs.next() )
{
   System.out.print( rs.getString("employee_name") );
   System.out.print( rs.getInt("salary") );
}
```

Scrollable `ResultSet`

- New in JDBC 2.0
  - Scrolling forward and backward
  - Absolute positioning
  - Direct insert of a row
  - Direct update of a row
  - Statement specifies type of `ResultSet` to create

Types of ResultSet

- **Scroll type**
  - TYPE_FORWARD_ONLY
    - **Cursor may move only forward**
  - TYPE_SCROLL_INSENSITIVE
    - **NOT sensitive to changes made by others**
  - TYPE_SCROLL_SENSITIVE
    - **Sensitive to changes made by others**
- **Concurrency**
  - CONCUR_READ_ONLY
    - **May NOT be updated**
  - CONCUR_UPDATABLE
    - **May be updated**

ResultSet Operations

- Provides a mechanism for accessing the results of a query. Provides methods for:
  - Navigating the resultant records
    - Locating cursor
    - Moving cursor
  - Determining where the cursor is
  - Modifying resultant records
    - Inserting (a special row is provided)
    - Updating
    - Deleting

Navigation Operations

```
boolean previous()
boolean first()
boolean last()
void absolute( int position )
void relative( int rows )
int getRow()
boolean isFirst()
boolean isLast()
boolean isBeforeFirst()
boolean isAfterLast()
```

Modification Operations

```
void moveToInsertRow()
void insertRow()
void updateRow()
int getRow()
void refreshRow()
void updateXxx( String colName, xxx value )
void deleteRow()
```

Obtaining Scrollable ResultSet

```
Connection conn = null;
String db = "jdbc:mysql://localhost/EmployeeDB";
String username = "student";
String password = "student";
conn = DriverManager.getConnection( db, user, pass );

Statement stmt;
stmt = conn.createStatement( ResultSet.TYPE_SCROLL_SENSITIVE,
                             ResultSet.CONCUR_UPDATABLE );
String query = "SELECT employee_name, salary"
             + "  FROM employee"
             + " ORDER BY employee_name";
ResultSet rs = stmt.executeQuery( query );
```

Updating via ResultSet

```
...
conn = DriverManager.getConnection( db, user, pass );

String query = "SELECT employee_id, salary"
             + "  FROM employee"
             + " WHERE employee_name = 'Barney Rubble'" );
Statement stmt;
stmt = conn.createStatement( ResultSet.TYPE_SCROLL_SENSITIVE,
                             ResultSet.CONCUR_ UPDATABLE );
rs = stmt.executeQuery( query );
...
rs.updateInt("salary", 55000 );
rs.updateRow();
...
```

Inserting via ResultSet

```
byte[] thePassword;
...
conn = DriverManager.getConnection( db, user, pass );

String query = "SELECT employee_id, employee_name, salary"
             + "  FROM employee" );
Statement stmt;
stmt = conn.createStatement( ResultSet.TYPE_SCROLL_SENSITIVE,
                             ResultSet.CONCUR_READ_ONLY );
rs.moveToInsertRow();
rs.updateString("employee_name", 'Bambam Rubble' );
rs.updateInt("salary", 25000 );
rs.insertRow();
...
```

Precompiled SQL

- Provides superior performance for queries which are executed repeatedly.
- Queries contain parameter markers to allow parameter replacement for each execution.

Parameter Markers

- Methods are provided for performing parameter substitution.
- Each of the set methods accepts:
  - A first argument that is the parameter index
    - Parameter indexes are 1-based, not 0-based
  - A second argument of the appropriate type
  - Third argument for length on stream methods

Parameter Replacement Methods

- **Set of methods for setting parameters**

```
setXxx( int paramIndex, xxx value )
```

SQL Type	Java Type	Method	2.0
ARRAY	Array	setArray	x
BIGINT	long	setLong	
BINARY	byte[]	setBytes	
BIT	boolean	setBoolean	
BLOB	java.sql.Blob	setBlob	x
CLOB	java.sql.Clob	setClob	x
DATE	java.sql.Date	setDate	
DECIMAL	Object	setObject (scale)	
DOUBLE	double	setDouble	
FLOAT	double	setFloat	
INTEGER	int	setInt	

Parameter Replacement Methods

SQL Type	Java Type	Method	2.0
LONGVARBINARY	java.io.InputStream	setBinaryStream	
LONGVARCHAR	java.io.InputStream	setAsciiStream setUnicodeStream setCharacterStream	
NUMERIC	Java.math.BigDecimal	setBigDecimal	
REF	java.sql.Ref	setRef	x
SMALLINT	short	setShort	
TIME	java.sql.Time	setTime	
TIMESTAMP	java.sql.Timestamp	setTimestamp	
TINYINT	byte	setByte	
VARBINARY	byte[]	setBytes	
VARCHAR	String	setString	
<user-defined>	Object (optional map)	setObject	x

Using Parameter Markers

```
String query = "SELECT employee_name, salary"
             + " FROM employee"
             + " WHERE employee_name = ?";

PreparedStatement ps = con.prepareStatement( query );
.
.
.
ps.setString( 1, "Barney Rubble" );
ResultSet rs = ps.executeQuery();
```

Auto Fields

- Fields automatically populated by the database
  - Typically integer
  - Commonly used to provide unique index
  - Database must provide a means of determining the value of the last generated value
    - Varies by database

```
SELECT LAST_INSERT_ID
```

Example

< Review Source Code >

`EmployeeDb.java`

SQL Backup Slides

Create

- **Creates a new table**

```
CREATE TABLE table_name
(column_def,... [, constraint_def,...])
```

- **Column definition**

```
column datatype [[NOT] NULL] [AUTO_INCREMENT]
[PRIMARY KEY]
```

- **Constraint definition**

```
[CONSTRAINT name]
UNIQUE (column,...) |
PRIMARY KEY (column,...) |
FOREIGN KEY(column,...) REFERENCES ref_tab(column,...)
```

Create Examples

```
CREATE TABLE employee
(employee_id INTEGER AUTO_INCREMENT,
 employee_name VARCHAR(30),
 salary INTEGER NOT NULL,
PRIMARY KEY (employee_id),
UNIQUE KEY(employee_name))
```

```
CREATE TABLE dependent_type
(relationship VARCHAR(10) PRIMARY KEY)
```

```
CREATE TABLE dependent
(employee_id INTEGER NOT NULL,
 dependent_name VARCHAR(30) NOT NULL,
 relationship VARCHAR(10) NOT NULL,
 PRIMARY KEY(employee_id,dependent_name),
 FOREIGN KEY(employee_id) REFERENCES employee(employee_id),
 FOREIGN KEY(relationship) REFERENCES dependent_type
(relationship))
```

Create Index

- **Creates an index on a set of columns in a table**

```
CREATE [UNIQUE] INDEX index_name
    ON table_name (column,...)
```

Select

- **Select a set of information which meets some criteria**

```
SELECT [DISTINCT] columns|*
  FROM tables
[WHERE criteria]
[ORDER BY column_list]
[UNION [ALL] select_statement]
```

```
SELECT employee_name, salary
  FROM employee
 WHERE salary > 50000
 ORDER BY salary, employee_name
```

Logical Operators

=	Equal to
>	Greater than
>=	Greater than equal to
<	Less than
<=	Less than equal to
<>	Not equal to
AND	Both conditions are true
OR	Either condition is true
NOT	Returns the opposite condition

SQL Operators

IN(*list*)	**Match any of a list of values**
LIKE	**Match character pattern**
	% **represents zero or more characters**
	_ **represents any single character**
NULL	**The null value**

Union

- Combines results of two queries
  - Queries must have same number of columns
  - Columns must be of same type
  - Order results of UNION not individual queries

```
SELECT employee_name
  FROM employee
 UNION
SELECT dependent_name
  FROM dependent
 ORDER BY 1
```

Join

- Combine columns from multiple tables
  - Construct a joined row from rows in each table, matching on a column(s) value
  - Use table aliases or table names to specify unique column names

Join Examples

```
SELECT employee_name, dependent_name
  FROM employee, dependent
 WHERE employee.employee_id = dependent.employee_id
 ORDER BY employee_name, dependent_name
```

```
SELECT e.employee_name, d.dependent_name
  FROM employee e, dependent d
 WHERE e.employee_id = d.employee_id
 ORDER BY e.employee_name, d.dependent_name
```

Subquery

- **Uses the results of one query as part of another query**

```
SELECT employee_name, salary
  FROM employee
 WHERE salary > (SELECT e.salary
                   FROM employee e
                  WHERE e.employee_name
                                    = 'Barney Rubble')
```

Column Functions

- **Summarize the contents of an entire column**
  - SUM()
  - AVG()
  - MIN()
  - MAX()
  - COUNT() and COUNT(*)

```
SELECT SUM(salary),AVG(salary),MAX(salary),MIN(salary)
  FROM employee
```

Insert

- **Adds new rows into a table**

```
INSERT INTO table [(column [, column2 ])]
     VALUES (value [, value])|select_statement
```

```
INSERT INTO employee
  VALUES ('Fred Flintstone', 55000 )
```

```
INSERT INTO employee ( employee_name, salary )
  VALUES ('Barney Ruble', 45000 )
```

Update

- **Modifies existing rows**

```
UPDATE table
   SET column = value [, column = value]
[WHERE condition]
```

Update Examples

```
UPDATE employee
   SET salary = 50000
 WHERE (employee_id = 1)
```

```
UPDATE employee
   SET salary = 50000
 WHERE employee_id IN (SELECT DISTINCT employee_id
                       FROM employee e
                       WHERE e.employee_name
                                  = 'Barney Rubble')
```

Delete

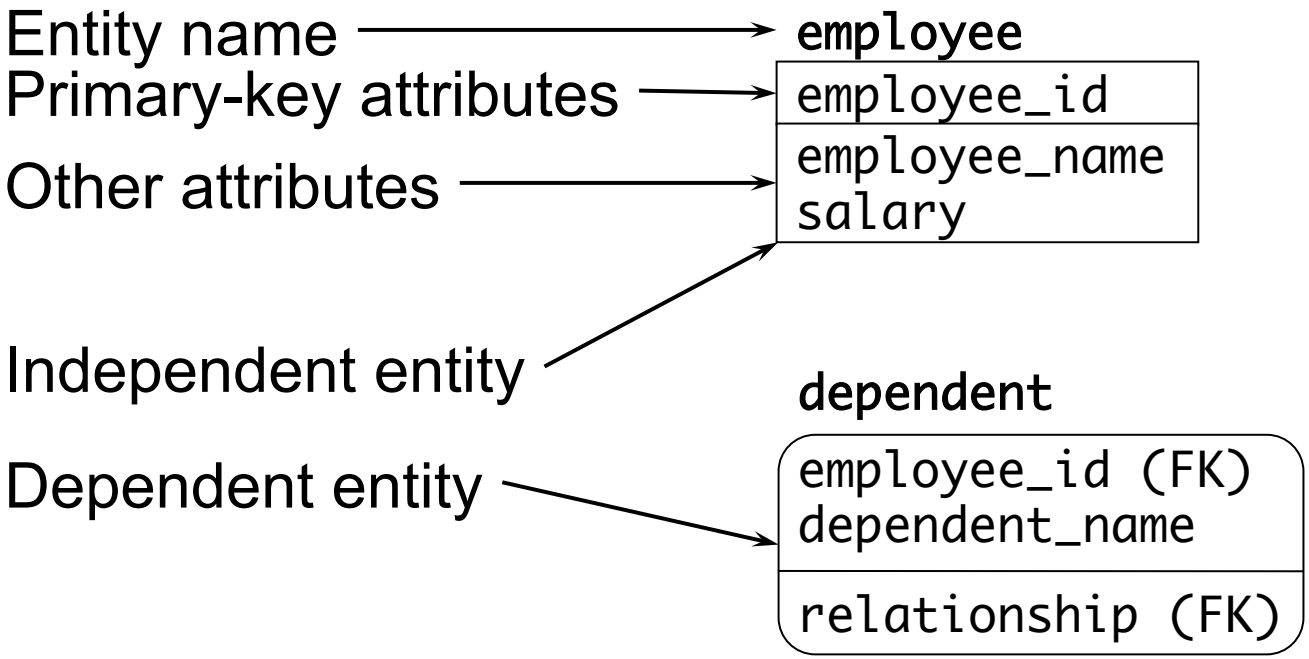
- **Remove existing rows**

```
DELETE FROM table
[WHERE condition]
```

```
DELETE FROM employee
WHERE (employee_id = 0)
```

```
DELETE FROM employee
WHERE employee_id IN (SELECT DISTINCT employee_id
                      FROM employee e
                      WHERE e.employee_name
                              = 'Fred Flintstone')
```

IDEF1X Entity Notation



Example Schema