JDBC

Objectives for This Session

- Exposure to relational database concepts
- Exposure to the Structured Query Language, SQL
- Introduction to Java's JDBC facility

What Is A Database?

- Any organized collection of data
  - A single byte
  - Really, really big
    - (The EMC Isilon Scale-Out NAS Storage “Scales easily to 50 PB”)
  - Airline reservation and tracking
  - The data on your phone

What Is A Relational Database?

- Data is organized into tables
  - Columns represent named fields
  - Rows represent blocks of data corresponding to those fields
  - Some fields are designated keys
  - Rows can be accessed by any field…
  - … accessing key fields is fast
  - A field in one table may be used to access a row in a related table

Database Tables, Example

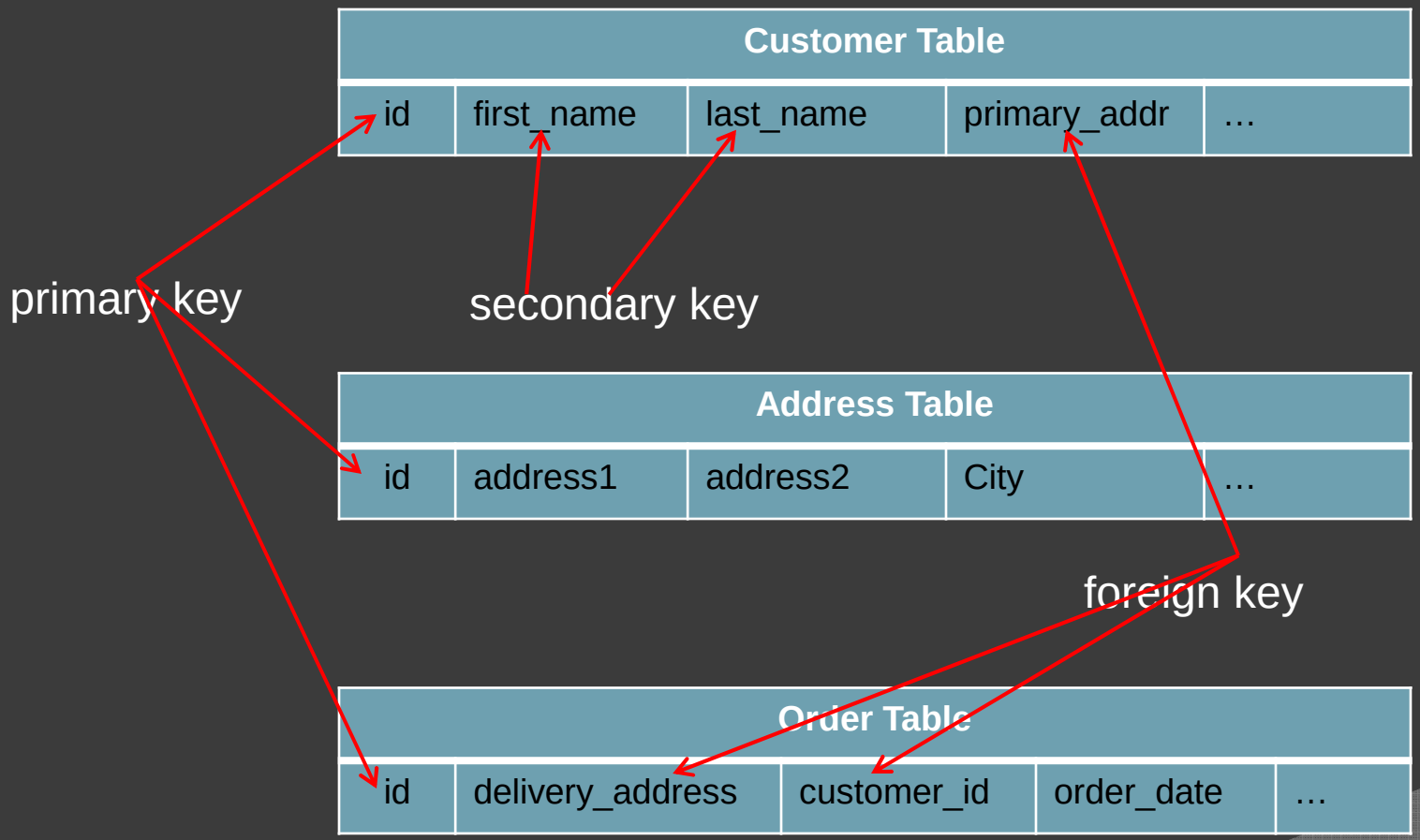
Customer Table					
id	first_name	last_name	phone	street1	…
01	Fred	Flintstone	212-555-1111	…	…
02	Wilma	Flintstone	212-555-1111	…	…
03	Betty	Rubble	212-555-2222	…	…
…					

Order Table					
id	cust_id	deliver_street1	deliver_street2	deliver_city	…
01	02	122 Main	#45	Seattle	…
02	03	124 Main	Null	Seattle	…
03	02	124 Main	#45	Seattle	…
…					

Keys

- Primary Key
  - Refers to the table that contains it
  - Must be unique
  - Often automatically generated
- Secondary Key
  - Refers to the table that contains it
  - May or may not be unique
- Foreign Key
  - Links to another table

Keys, Example



Index

- A column used to frequently access a row in a table, usually a key
- Provides very fast access
- Requires additional storage and memory resources

What Is The Structured Query Language (SQL)?

- A language to create, interrogate and update databases
- Originally developed at IBM about the same time C was developed at AT&T
  - Uses “unusual” naming conventions
- Similar operations do not necessarily use similar language constructs
- Widely, but inconsistently, adopted

SQL Commands

- Data Definition Language (DDL)
  - Create a table
  - Drop (delete) a table
- Data Manipulation Languate (DML)
  - Add a row to a table
  - Delete a row from a table
  - Update a row in a table
  - Select rows from a table

Some Common SQL Data Types

- CHARACTER(*n*)
  Character string of exactly length *n*
- VARCHAR(*n*)
  Character string of up to length *n*
- INTEGER
  Binary integer, up to 10 digits
- DATE
  Month, day, year
- BOOLEAN
  True or false

Other SQL Data Types

- INTEGER(*p*)
- DECIMAL(*p, s*)
- REAL
- TIME
- ARRAY
- BLOB
- Others…

More SQL Terms

- Connection
  - A link to a database; may be local or remote
- Statement
  - A SQL directive associated with a connection
  - A connection may associate with many statements
- Result Set
  - Data (if any) obtained by executing a SQL statement
  - A statement has at most one result set

JDBC

- Not an acronym
- An API for connecting to a database and executing SQL commands
- Lives in java.sql and javax.sql packages

JDBC Driver Types

- Type 1
  - JDBC-ODBC bridge technology
- Type 2
  - JNI drivers for C/C++ libraries
- Type 3
  - Socket-level middleware translators
- Type 4
  - Pure Java DBMS driver

Type 1 Drivers

- JDBC-ODBC Bridge
  - Translates call to ODBC and forwards to ODBC driver
  - ODBC must exist on every client
  - Slow

Type 2 Drivers

- Native API
  - Java makes direct access to C/C++ libraries
  - Requires client-side libraries
  - Provided by DBMS vendor
  - Can crash JVMs
  - Fast

Type 3 Drivers

- Middleware Java Driver
  - Translates JDBC calls into DBMS-specific calls
  - Communicates via a socket with a middleware server; server provides libraries (no client-side libraries necessary)
  - One driver can provide access to multiple DBMSs

Type 4 Drivers

- Pure Java Drivers
  - Driver talks directly to DBMS via socket
  - No middleware needed
  - No client-side libraries needed

Establishing A Connection

- Load the JDBC driver class
- Connect to the data source
  - Location of data source provided as a URL, e.g.:

```
jdbc:derby://localhost:1527/memory:scgDb
```

Loading Drivers

1. Use DataSource interface and JNDI
   - Preferred method
   - Identify drivers using system properties
   - How most of my examples do it
2. Do nothing; rely on service provider mechanism
   - How you will do it for your project
3. Load driver class explicitly
   - Least desirable method

The Derby Database Server

- A product from Apache
- Add a repository to your POM file:

```
…
</properties>
<repositories>
  <repository>
    <id>cp125-repository</id>
    <name>Repository for CP125</name>
    <url>http://faculty.washington.edu/rmoul/repository</url>
  </repository>
</repositories>
<dependencies>
…
```

Continued on next slide

The Derby Database Server (continued)

- Add a dependency to your POM file:

```
  …
  <dependency>
    <groupId>edu.uw.ext</groupId>
    <artifactId>cp125-support</artifactId>
    <version>1.0</version>
  </dependency>
</dependencies>
…
```

The Derby Database Server (continued)

- Some of the examples in this lecture use these properties:

```
public class DBConstants
{
    public static final String  DB_URL                  =
        "jdbc:derby:EmployeeDb;create=true";
    public static final String  DRIVER_CLASS_NAME       =
        "org.apache.derby.jdbc.EmbeddedDriver";
    public static final String  DERBY_DIR_NAME          =
        "target/derbyDb";
    public static final String  DERBY_HOME_PROP_NAME    =
        "derby.system.home";
    public static final String  DERBY_HOME              =
        "target/derbyDb";
    …
```

The Derby Database Server (continued)

- And this initialization method:

```
…
public static void init()
    throws ClassNotFoundException
{
    File    derbyDir    = new File( DERBY_DIR_NAME );
    derbyDir.mkdirs();
    System.setProperty( DERBY_HOME_PROP_NAME, DERBY_HOME );
    Class.forName( DRIVER_CLASS_NAME );
}
…
```

The Derby Database Server (continued)

- And connect like this:

```
try
{
    DBConstants.init();
    Connection  conn = DriverManager.getConnection(DB_URL);
    // do something interesting with the connection
    DBConstants.closeConnection( conn );
}
catch ( ClassNotFoundException exc )
{
    log.error( "driver not found", exc );
}
catch ( SQLException exc )
{
    log.error( "db connection failed", exc );
}
```

The Derby Database Server And Your Homework

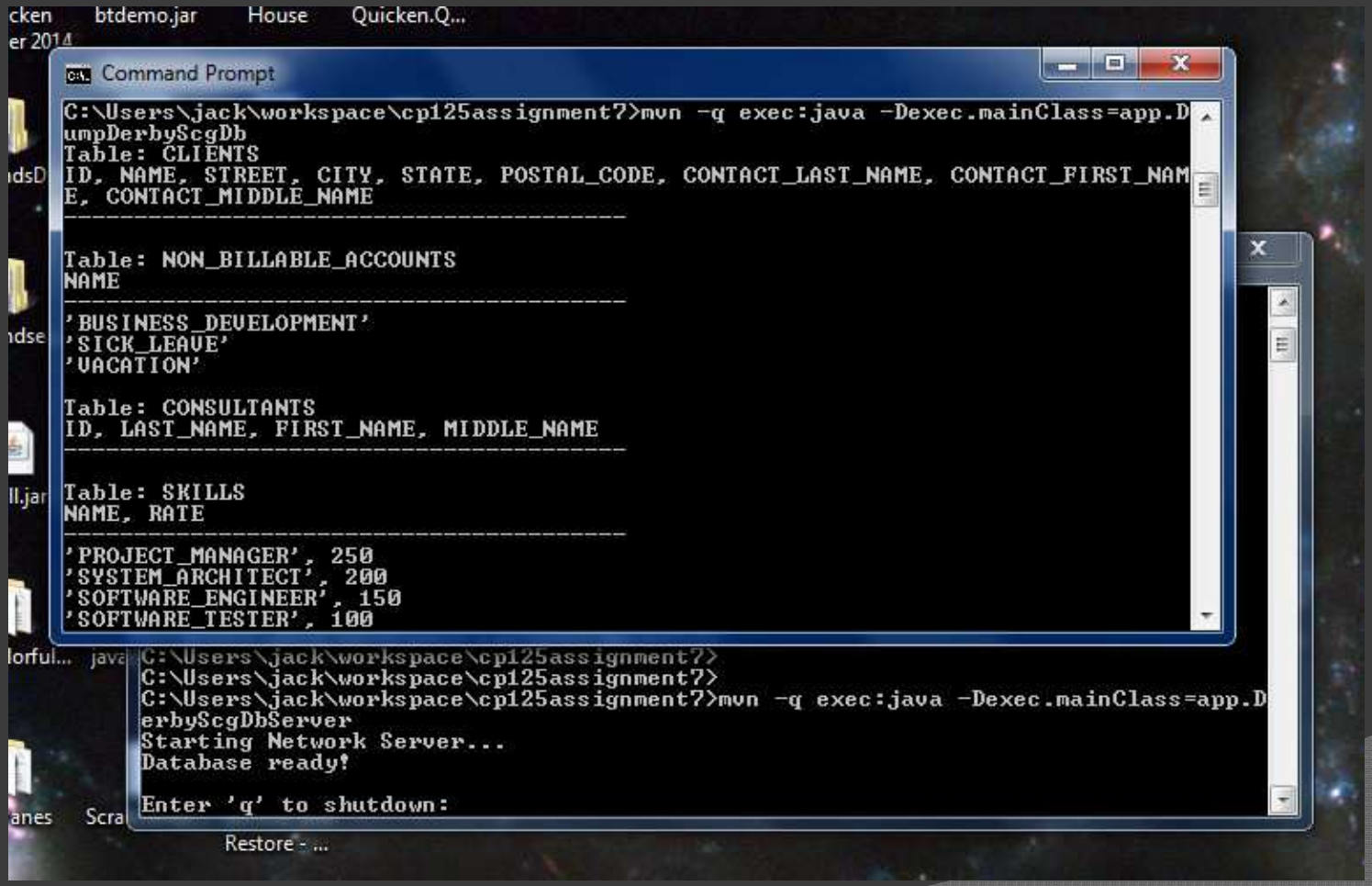
- Start the Derby server in a separate, dedicated window using the command:

  ```
  > mvn -q exec:java
      -Dexec.mainClass=app.DerbyScgDbServer
  ```

- Dump the SCG database (from another command line) using this command:

  ```
  > mvn -q exec:java
     -Dexec.mainClass=app.DumpDerbyScgDb
  ```

The Derby Database Server And Your Homework (continued)



The Derby Database Server And Your Homework (continued)

- All the new database methods in the sample homework solution follow this strategy:

```
public type operation ( type param )
    throws SQLException
{
    try ( Connection conn = getConnection() )
    {
        operation( conn, param )
    }
}
```

The Derby Database Server And Your Homework Example

```
public void addClient( ClientAccount client )
    throws SQLException
{
    try ( Connection conn = getConnection() )
    {
        addClient( conn, client );
    }
}

private Connection getConnection()
    throws SQLException
{
    Connection  conn    =
        DriverManager.getConnection( dbURL, dbUserName,
            dbPassword );
    return conn;
}
```

Working With A Connection Object

General Strategy:

1. Open a connection
2. Use the connection to create/prepare *statements*
3. Use the statement to execute SQL, possibly obtaining a *result set*
4. Close all result sets, statements and connection…

Working With A Connection Object (continued)

- A statement has at most one result set
  - If you need more than one result set, you need more than one statement
  - When the statement is closed, its result set is closed
- A connection may have many statements
  - When the connection is closed, all its statements are closed

Connection Class Methods

- **Statement createStatement()**
- **PreperedStatement prepareStatement(String sql)**
- **PreparedStatement prepareStatement(String sql, int getKeys)**

Also:

- prepareCall()
- commit()
- rollBack()
- etc.

Statement Class Methods

- **ResultSet executeQuery(String sql)**
- **int executeUpdate(String sql)**
- **int executeUpdate(String sql, int getKeys)**
- **ResultSet getGeneratedKeys()**

Also

- void addBatch()
- void clearBatch()
- int[] executeBatch()
- etc.

ResultSet Class Methods

- Used to iterate through result of a query
- boolean next() – moves the cursor to the next result; returns false if none
- *type* get*type*(int) – gets a value from a field given its index (1-based!!)
- *type* get*type*(String) – gets a value from a field given its name
- void close() – releases resources
- ResultSetMetaData getMetaData() - gets data about the ResultSet

ResultSetMetaData Class

- Provides data about the result set, such as:
  - int getColumnCount()
  - String getColumnName(int column)
  - int getColumnType(int column)
  - boolean wasNull()

Getting Values From A ResultSet

- Use the method that returns the value type

SQL Type	Java Type	Method
INTEGER	int	getInt()
CHAR	java.lang.String	getString()
VARCHAR (*N*)	java.lang.String	getString()
DATE	java.sql.Date	getDate()
BOOLEAN	boolean	getBoolean()
DECIMAL	java.math.BigDecimal	getBigDecimal()
DOUBLE	double	getDouble()
etc.		

SQL Select Statement

- SELECT *col1, col2…* FROM *table*
  ORDER BY *colA, colB*
- SELECT * FROM *table*

```
private static final String SELECT_SQL  =
    "SELECT first_name, last_name, birth_date "
    + "FROM student ORDER BY last_name, first_name";
```

Simple Query Example

```
private Connection getConnection()
    throws SQLException
{
    Connection  conn    =
        DriverManager.getConnection( DBConstants.DB_URL );
    return conn;
}

public void execute()
    throws SQLException
{
    try ( Connection conn = getConnection(); )
    {
        execute( conn );
    }
}
```

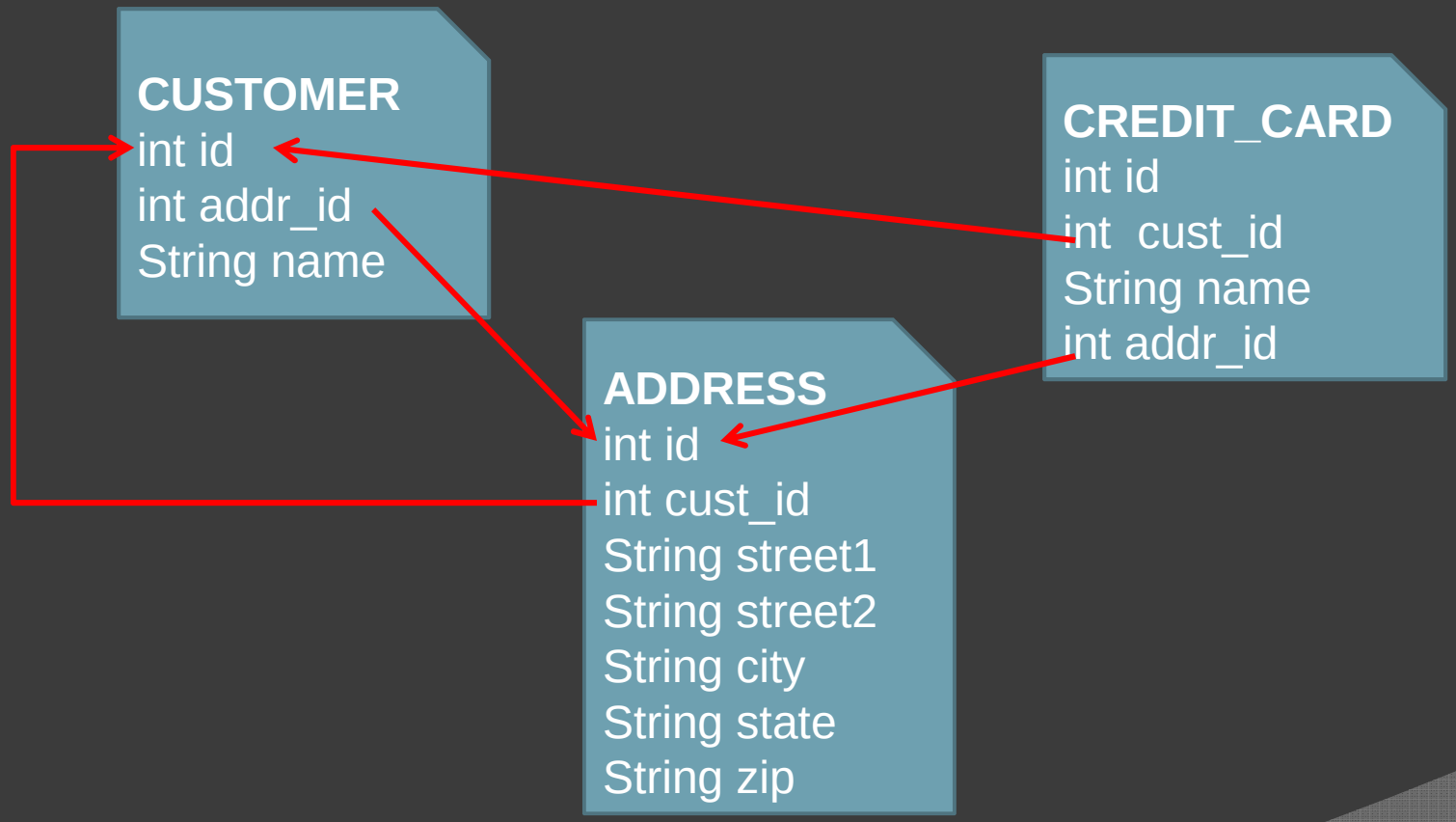
Simple Query Example (continued)

```
public void execute( Connection conn )
    throws SQLException
{
    Statement   statement   = conn.createStatement();
    ResultSet   resultSet   = statement.executeQuery( SELECT_SQL );
    String      fmt         = "%s, %s: born on %s%n";

    while ( resultSet.next() )
    {
        String  first   = resultSet.getString( "first_name" );
        String  last    = resultSet.getString( "last_name" );
        Date    born    = resultSet.getDate( "birth_date" );
        System.out.printf( fmt, last, first, born.toString() );
    }

    statement.close();
}
```

Splitting Data Between Tables



Splitting Data Between Tables (continued)

- A *customer* may have multiple addresses; *addr_id* is a *foreign key* that links the customer to her principal address
- Every *address* has a *foreign key* that identifies the customer it belongs to
- A customer may have multiple credit cards; a *credit_card* has a *foreign key* that links it to the *customer* who owns it, and another that links to a billing address

Precompiled SQL

- A powerful, efficient means for interrogating a database

```
SELECT * FROM table WHERE field1 = ? AND field2 = ?
```

- Statement is compiled before starting query
- Values are substituted for the question marks when executing the query
- Supported by the *PreparedStatement* interface

PreparedStatement: Setting Data Fields

- PreparedStatement has methods for setting fields of different types:

```
setT( int inx, T value )
```

- For example:

```
void setString( inx, “bob” )
void setInt( inx, 5 );
void setBoolean( inx, true )
```

Related Tables, Example

```
public class Customer
{
    private Integer        ident;
    private Name           name;
    private Address        address;
    private final
        List<CreditCard> creditCards;
```

Related Tables, Example (continued)

```
public class CreditCard
{
    private Integer ident   = null;
    private String  name;
    private Address address;
    private String  number;
    private String  cvv;
    private Date    expiry;
    …
}
```

Related Tables, Example (continued)

```
private static final String SELECT_BY_NAME_SQL  =
    "SELECT * FROM customer "
    + "WHERE first_name = ? AND last_name = ?";
…
public List<Customer> selectAll()
    throws SQLException
{
    try ( Connection conn = getConnection() )
    {
        List<Customer>  list    = selectAll( conn );
        return list;
    }
}
…
```

Related Tables, Example (continued)

```
private List<Customer>
buildCustomerList( Connection conn, ResultSet results )
    throws SQLException
{
    CreditCardTable     ccTable     =
        CreditCardTable.getInstance(
            connName, connPassword, connURL );
    List<Customer>      list        = new ArrayList<>();

    while ( results.next() )
    {
        int     ident   = results.getInt( "ident" );
        String  first   = results.getString("first_name" );
        String  last    = results.getString( "last_name" );
        String  middle  = results.getString("middle_name");
        Name    name    = new Name( first, last, middle );
        …
```

Related Tables, Example (continued)

```
        …
        String  addr1   = results.getString( "address1" );
        String  addr2   = results.getString( "address2" );
        String  city    = results.getString( "city" );
        String  state   = results.getString( "state" );
        String  zip     = results.getString( "zip" );
        Address addr    =
            new Address( addr1, addr2, city, state, zip );
        …
}
```

Related Tables, Example (continued)

```
            …
            Customer            customer    =
                new Customer( name, addr, ident );
            List<CreditCard>    creditCards =
                ccTable.selectAll( conn, customer );
            for ( CreditCard card : creditCards )
                customer.addCreditCard( card );

            list.add( customer );
        }

        return list;
    }
```

Related Tables, Example (continued)

```
// class CreditCardTable
private static final String SELECT_ALL_SQL  =
    "SELECT * FROM credit_card WHERE cust_id = ?";

public List<CreditCard> selectAll( Connection conn,
Customer customer )
    throws SQLException
{
    List<CreditCard>    list        = new ArrayList<>();
    PreparedStatement   statement   =
        conn.prepareStatement( SELECT_ALL_SQL );
    statement.setInt( 1, customer.getIdent() );
    ResultSet           results     =
        statement.executeQuery();

    while ( results.next() )
    {
        …
```

Inserting Into A Table

- To insert a row in a table use the *INSERT* statement

```
INSERT INTO table ( field, field, … )
    VALUES ( ?, ?, … )
```

- Must have as many ?s as fields
- At execution time, use a PreparedStatement to set values for the fields by index

Inserting Rows, Example

```
// DRIVER
CustomerTable   table   =
    CustomerTable.getInstance(
        connName, connPassword, connURL );

log.info( "adding customers" );
for ( int inx = 0 ; inx < names.length ; ++inx )
{
    Customer    customer    =
        new Customer( names[inx], addresses[inx] );
    for ( CreditCard card : creditCards[inx] )
        customer.addCreditCard( card );
    table.insert( customer );
}
```

Inserting Rows, Example (continued)

```
// class CustomerTable
private static final String INSERT_SQL  =
    "INSERT INTO customer ("
    +   "first_name, "
    +   "last_name, "
    +   "middle_name, "
    +   "address1, "
    +   "address2, "
    +   "city, "
    +   "state, "
    +   "zip"
    + ")"
    + "VALUES ( ?, ?, ?, ?, ?, ?, ?, ? )";
```

Inserting Rows, Example (continued)

```
// class CustomerTable
private Connection getConnection()
    throws SQLException {
    Connection conn =
        DriverManager.getConnection(
            connURL, connName, connPassword );
    return conn;
}
public void insert( Customer customer )
    throws SQLException {
    try ( Connection conn = getConnection() )
    {
        insert( conn, customer );
    }
}
```

Inserting Rows, Example (continued)

```
// class CustomerTable
private void
insert( Connection conn, Customer customer )
    throws SQLException
{
    Address             addr    = customer.getAddress();
    String              addr1   = addr.address1;
    String              addr2   = addr.address2;
    String              city    = addr.city;
    String              state   = addr.state;
    String              zip     = addr.zip_code;

    Name                name    = customer.getName();
    String              first   = name.first;
    String              last    = name.last;
    String              middle  = name.middle;
    …
```

Inserting Rows, Example (continued)

```
// class CustomerTable
    …
    int                 flags       =
        Statement.RETURN_GENERATED_KEYS;
    PreparedStatement   statement   =
        conn.prepareStatement( INSERT_SQL, flags );
    statement.setString( 1, first );
    statement.setString( 2, last );
    statement.setString( 3, middle );
    statement.setString( 4, addr1 );
    statement.setString( 5, addr2 );
    statement.setString( 6, city );
    statement.setString( 7, state );
    statement.setString( 8, zip );
    statement.executeUpdate();
…
```

Update A Row In A Table

- To update a row in a table use the *UPDATE* statement

```
UPDATE table ( field1 = val1, field2 = val2, … )
WHERE fieldN = valN
```

- You may specify ? as a value…
- … at execution time, use a PreparedStatement to set values for the fields by index

Update A Row In A Table Example

```
// class CustomerTable

 private static final String UPDATE_SQL  =
        "UPDATE customer SET "
        +   "first_name = ?, "
        +   "last_name = ?, "
        +   "middle_name = ?, "
        +   "address1 = ?, "
        +   "address2 = ?, "
        +   "city = ?, "
        +   "state = ?, "
        +   "zip = ? "
        + "WHERE ident = ?";
```

Update A Row In A Table Example

```
private void update( Connection conn, Customer
customer )
    throws SQLException
{
    PreparedStatement   statement   =
        conn.prepareStatement( UPDATE_SQL );
    Name         name         = customer.getName();
    Address      addr         = customer.getAddress();
    int          recID        = customer.getIdent();
    …
```

Update A Row In A Table Example (continued)

```
    …
    statement.setString( 1, name.first );
    statement.setString( 2, name.last );
    statement.setString( 3, name.middle );
    statement.setString( 4, addr.address1 );
    statement.setString( 5, addr.address2 );
    statement.setString( 6, addr.city );
    statement.setString( 7, addr.state );
    statement.setString( 8, addr.zip_code );
    statement.setInt( 9, recID );
    statement.executeUpdate();

    statement.close();
}
```

JDBC 2.0: Scrollable Result Sets

- You must create a statement with scrollable result sets enabled
- Move cursor forward, backward, or to a specific position
- Update or insert rows directly from result set

JDBC 2.0: Scrollable Result Sets (continued)

```
Connection.createStatement(
    int resultSetType,
    int resultSetConcurrency
);
```

- Flag parameters are located and documented in class ResultSet

JDBC 2.0: Scrollable Result Sets (continued)

resultSetType	
TYPE_FORWARD_ONLY	Cursor can move forward only; not sensitive to database changes
TYPE_SCROLL_INSENSITIVE	Cursor is scrollable; not sensitive to database changes
TYPE_SCROLL_SENSITIVE	Cursor is scrollable; generally sensitive to database changes

resultSetConcurrency	
CONCUR_READ_ONLY	Result set cannot be used to perform database updates
CONCUR_UPDATABLE	Result set can be used to perform database updates

Scrollable ResultSet Operations

Operations for:

- Navigating the ResultSet
  - Locate the cursor
  - Move the cursor
- Modifying records:
  - Insert
  - Update
  - Delete

Scrollable ResultSet Navigation Operations

Moving the Cursor	
boolean	next()
boolean	previous()
boolean	first()
boolean	last()
void	absolute(rowNum)
void	relative(numRows)

Scrollable ResultSet Modification Operations

Modifying Data	
void	moveToInsertRow()
void	insertRow()
void	updateRow()
void	deleteRow()
void	refreshRow()
void	update*T*(String field, *T* value)

Scrollable ResultSet: Insert Example

```
private void execute( Connection conn )
    throws SQLException
{
    int         type        =
        ResultSet.TYPE_SCROLL_SENSITIVE;
    int         curr        = ResultSet.CONCUR_UPDATABLE;
    Statement   statement   =
        conn.createStatement( type, curr );
    statement.executeQuery( SELECT_ALL_SQL );
    ResultSet   resultSet   = statement.getResultSet();
       resultSet.moveToInsertRow();
    …
```

Scrollable ResultSet: Insert Example (continued)

```
    …
    for ( int inx = 0 ; inx < firstNames.length ; ++inx )
    {
        String  first   = firstNames[inx];
        String  last    = lastNames[inx];

        resultSet.updateString( "first_name", first );
        resultSet.updateString( "last_name", last );
        resultSet.updateInt( "credits", 1 );
        resultSet.updateBoolean( "blocked", false );
        resultSet.insertRow();
    }

    statement.close();
}
```

Scrollable ResultSet: Update Example

```
private void execute( Connection conn )
    throws SQLException
{
    int         type    = ResultSet.TYPE_SCROLL_SENSITIVE;
    int         curr    = ResultSet.CONCUR_UPDATABLE;
    Statement   statement   =
        conn.createStatement( type, curr );
    statement.executeQuery( SELECT_ALL_SQL );
    ResultSet   resultSet   = statement.getResultSet();
    …
```

Scrollable ResultSet: Update Example (continued)

```
    …
    while ( resultSet.next() )
    {
        String  first = resultSet.getString("first_name");
        String  last  = resultSet.getString("last_name");

        int     credits = resultSet.getInt( "credits" );
        resultSet.updateInt( "credits", ++credits );
        resultSet.updateRow();
    }

    statement.close();
}
```

SQL Create Statement

CREATE TABLE *table_name (*

col_def, col_def,… constr, constr, …)

- Column Definition:
  name type options
  - (NOT) NULL, AUTOINCREMENT, GENERATED
- Constraint:
  - UNIQUE (*colNum, colNum,* ...)
  - PRIMARY KEY (*colNum*)
  - FOREIGN KEY (*colNum*)
    REFERENCES *table* (*colNum)*

SQL Create Table Example 1

```
private static final String CREATE_TABLE_SQL    =
    "CREATE TABLE account ( "
    + "ident INTEGER NOT NULL “
    +     “GENERATED ALWAYS AS IDENTITY, "
    + "first_name VARCHAR( 30 ), "
    + "last_name VARCHAR( 30 ) NOT NULL, "
    + "credits INTEGER, "
    + "blocked BOOLEAN, "
    + "PRIMARY KEY ( ident )"
    + ")";
```

SQL Create Table Example 1 (continued)

```
private void createTable( Connection conn )
    throws SQLException
{
    log.info( "creating account table" );
    Statement   statement   =
        conn.createStatement();
    statement.executeUpdate( CREATE_TABLE_SQL );
    statement.close();
}
```

SQL Create Table Example 2

```
//CUSTOMER TABLE
private static final String CREATE_TABLE_SQL    =
    "CREATE TABLE customer ( "
    + "ident INTEGER NOT NULL GENERATED “
    +     "ALWAYS AS IDENTITY, "
    + "first_name VARCHAR( 30 ), "
    + "last_name VARCHAR( 30 ) NOT NULL, "
    + "middle_name VARCHAR( 30 ), "
    + "address1 VARCHAR( 30 ) NOT NULL, "
    + "address2 VARCHAR( 30 ), "
    + "city VARCHAR( 30 ), "
    + "state CHARACTER( 2 ), "
    + "zip VARCHAR( 10 ), "
    + "PRIMARY KEY ( ident )"
    + ")";
```

SQL Create Table Example 2 (continued)

```
//CREDIT CARD TABLE
private static final String CREATE_TABLE_SQL    =
    "CREATE TABLE credit_card ( "
    + "ident INTEGER NOT NULL GENERATED ALWAYS AS IDENTITY, "
    + "cust_id INTEGER, "
    + "name VARCHAR(45), "
    + "number VARCHAR(25) NOT NULL, "
    + "cvv VARCHAR(5) NOT NULL, "
    + "expiration_date DATE NOT NULL, "
    + "address1 VARCHAR(30) NOT NULL, "
    + "address2 VARCHAR(30), "
    + "city VARCHAR(30) NOT NULL, "
    + "state VARCHAR(30) NOT NULL, "
    + "zip_code CHARACTER(5) NOT NULL, "
    + "PRIMARY KEY ( ident ), "
    + "FOREIGN KEY ( cust_id ) "
    +     "REFERENCES customer ( ident ) "
    + ")";
```

Logical Operators

Logical Operators		
Operator	Meaning	Relative Precedence
=	Equal to	7
>	Greater than	7
>=	Greater than or equal to	7
<	Less than	7
<=	Less than or equal to	7
<>	Not equal to	7
NOT	Logical negation (!)	6
AND	Boolean ‘and’ (&&)	5
OR	Boolean ‘or’ (||)	4

SQL Operators

SQL Operators		
Operator	Meaning	Relative Precedence
LIKE	Similar to; use with %	8
IN (list)	Present in a list	8
BETWEEN	Between two values	8
NULL	Field is unpopulated	8

Operators, Examples

```
SELECT * FROM customer WHERE last_name = ‘moe’
```

```
SELECT * FROM customer WHERE
    last_name = ‘moe’ AND first_name = ‘molly’
```

```
SELECT * FROM employee WHERE
    (dept = ‘manufacturing’ OR dept = ‘finance’)
    AND salary > 50000
```

```
SELECT * FROM employee WHERE last_name LIKE ‘smo%’
```

Union

Creates the union of two queries:

- Queries must have same number of cols
- Corresponding cols must be same type

```
SELECT employee_name FROM staff
UNION
Select student_name FROM student
```

Join

Combine columns from multiple tables

- Columns from matching rows are assembled in a single row
- Use table names or aliases to disambiguate columns from different tables with the same name

Join Types

- INNER JOIN
  - Returns all rows with match in all tables
- LEFT JOIN
  - Return all rows from the left table, and any matching rows in the right table
- RIGHT_JOIN
  - Return all rows from the right table, and any matching rows from the left table
- FULL JOIN
  - Return all rows with a match in any table

Inner Join Example

```
private final String JOIN_SQL   =
    "SELECT credit_card.name, "
    +    "credit_card.number, "
    +    "credit_card.expiration_date, "
    +    "customer.first_name, "
    +    "customer.last_name "
    +    "FROM credit_card "
    + "INNER JOIN customer "
    +    "ON credit_card.cust_id = customer.ident";

private final int   CC_NAME_INX         = 1;
private final int   CC_NUMBER_INX       = 2;
private final int   CC_EXPIRY_INX       = 3;
private final int   C_FIRST_NAME_INX    = 4;
private final int   C_LAST_NAME_INX     = 5;
```

Inner Join Example (continued)

```
public void execute( Connection conn )
    throws SQLException
{
    Statement   statement   = conn.createStatement();
    statement.executeQuery( JOIN_SQL );
    ResultSet   resultSet   = statement.getResultSet();

    System.out.println( "dumping result set" );
    ...
```

Inner Join Example (continued)

```
…
while ( resultSet.next() )
{
    String  cFirst   = resultSet.getString( C_FIRST_NAME_INX );
    String  cLast    = resultSet.getString( C_LAST_NAME_INX );
    String  ccName   = resultSet.getString( CC_NAME_INX );
    String  ccNumber = resultSet.getString( CC_NUMBER_INX );
    Date    ccExpiry = resultSet.getDate( CC_EXPIRY_INX );

    StringBuilder   bldr    = new StringBuilder();
    bldr.append( cLast ).append( ", " );
    bldr.append( cFirst ).append( ": " );
    bldr.append( "\"" ).append( ccName ).append( "\" " );
    bldr.append( ccNumber ).append( " " );
    bldr.append( "expires " ).append( ccExpiry );
    log.info( ": " + bldr.toString() );
}
…
```

SQL Sub-Query

- Use the result of one query as input to another

```
SELECT … WHERE col [comparison operator]
    ( SELECT colB FROM tableB )
```

SQL Sub-Query Example

```
// Select all credit cards for the customer having
// given first and last names.
private final String SUBQUERY_SQL   =
    "SELECT credit_card.name, "
    +    "credit_card.number, "
    +    "credit_card.expiration_date "
    + "FROM credit_card "
    + "WHERE cust_id = "
    +     "( SELECT ident FROM  customer "
    +         "WHERE first_name = ? "
    +         "AND last_name = ? )";

private final int   CC_NAME_INX          = 1;
private final int   CC_NUMBER_INX        = 2;
private final int   CC_EXPIRY_INX        = 3;
…
```

SQL Sub-Query Example (continued)

```
public void
execute( Connection conn, String firstName, String lastName )
    throws SQLException
{
    PreparedStatement   statement =
    conn.prepareStatement( SUBQUERY_SQL );
    statement.setString( 1, firstName );
    statement.setString( 2, lastName );
    ResultSet           results = statement.executeQuery();

    StringBuilder   bldr    = new StringBuilder();
    bldr.append( "All credit cards for " ).append( lastName );
    bldr.append( " " ).append( firstName );
    log.info( bldr.toString() );
    …
```

SQL Sub-Query Example (continued)

```
        …
        while ( results.next() )
        {
            String  name    = results.getString( CC_NAME_INX );
            String  number  = results.getString( CC_NUMBER_INX );
            String  expiry  = results.getString( CC_EXPIRY_INX );
            bldr = new StringBuilder( "    " );
            bldr.append( "\"" ).append( name ).append( "\" " );
            bldr.append( number ).append( " " );
            bldr.append( "expires " ).append( expiry );
            log.info( ": " + bldr.toString() );
        }

        statement.close();
    }
```

Column Functions

- Calculate a value for some column
  - SUM(*name*)
  - AVG(*name*)
  - MIN(*name*)
  - MAX(*name*)

Column Function Example

```
private static final String GET_AVERAGES_SQL    =
    "SELECT AVG( temp ), AVG( humidity ), AVG( pressure )"
    + "FROM weather "
    + "WHERE date = ?";

public void getAverages( Connection conn, Date date )
    throws SQLException, Exception
{
    WeatherTable    weather = new WeatherTable();
    weather.clean();
    weather.populateTable( conn );

    PreparedStatement   statement   =
        conn.prepareStatement( GET_AVERAGES_SQL );
    statement.setDate( 1, date );
    ResultSet           resultSet   = statement.executeQuery();
    if ( !resultSet.next() )
        throw new Exception( "weather table query failure" );
    ...
```

Column Function Example (continued)

```
    …
    String          fmt         = "%5.3f";
    double          temp        = resultSet.getDouble( 1 );
    double          humidity    = resultSet.getDouble( 2 );
    double          pressure    = resultSet.getDouble( 3 );
    String          sTemp       = String.format( fmt, temp );
    String          sHumidity   = String.format( fmt, humidity );
    String          sPressure   = String.format( fmt, pressure );
    StringBuilder   bldr        = new StringBuilder();
    bldr.append( "Averages for " ).append( date ).append( '\n' );
    bldr.append( "temp = " ).append( sTemp ).append( ", " );
    bldr.append( "humidity = " ).append( sHumidity ).append( ", " );
    bldr.append( "pressure = " ).append( sPressure );
    log.info( bldr.toString() );
}
```

IDEF1X

"IDEF1X is a method for designing relational databases with a syntax designed to support the semantic constructs necessary in developing a conceptual schema. A conceptual schema is a single integrated definition of the enterprise data that is unbiased toward any single application and independent of its access and physical storage. Because it is a design method, IDEF1X is not particularly suited to serve as an AS-IS analysis tool, although it is often used in that capacity as an alternative to IDEF1. IDEF1X is most useful for logical database design after the information requirements are known and the decision to implement a relational database has been made. Hence, the IDEF1X system perspective is focused on the actual data elements in a relational database. If the target system is not a relational system, for example, an object-oriented system, IDEF1X is not the best method."

- *from www.idef.com*

Miscellaneous

All execution methods in the Statement interface implicitly close a statment's current ResultSet object if an open one exists.

When a Statement object is closed, its current ResultSet object, if one exists, is also closed.

SQL requires single quotes around text values (most database systems will also allow double quotes)…

… however, numeric fields should not be enclosed in quotes:

sql was being developed at IBM at the same time C was being developed at Bell Labs