# Database access

# Topics

- RPG IV database access review
- SQL review
- JDBC introduction, classes
- JDBC steps for use
- Commitment control
- Metadata
- Calling stored procedures

# Unit objectives

After completing this unit, you should be able to:

- Describe basic SQL syntax
- Describe the structure of JDBC API, including the interfaces it provides and the process for accessing iSeries and IBM i data
- Leverage commitment control and stored procedures within JDBC

# RPG IV DB access

How you access data with RPG IV:

| Step | Description |
|---|---|
| Declare | On F-spec, declare external file with DISK keyword |
| Open | Implicitly with F-spec, or explicitly with USROPN F-spec keyword and OPEN op-code |
| Pos'n cursor | Using SETLL, SETGT or CHAIN op-codes |
| Read | Using READ, READP, READE, or READPE op-codes |
| Write | Using WRITE, UPDATE or DELETE op-codes |
| Close | Implicitly with exit and LR ON, or explicitly with CLOSE op-code |

- **Note**: Use NOMAIN H-spec keyword for cycle-less "procedure only" modules

# RPG IV commitment control

- How you do commitment control with RPG IV:

| Step | Description |
|---|---|
| Prepare | CRTJRN, CRTJRNRCV and STRJRNRCV CL cmds |
| Start | Using STRCMTCTL CL command |
| Declare | F-spec COMMIT keyword. Can use optional runtime field or parameter for dynamic control of commitment |
| Open, Pos'n, Write | Using usual RPG op-codes |
| Commit or Cancel | Using COMMIT or ROLBK op-codes |
| Close | Implicitly with exit and LR ON, or explicitly with CLOSE |
| End | Using ENDCMTCTL CL command |

# SQL

- RPG DISK access is called *direct database access*
  - Position exactly to record you want and work with it

- Database standard access is via Structured Query Language (SQL)
  - Standards set by X/Open.
  - SQL is "result set" based
    - You work with a set of records defined using filter criteria
      - For example: WHERE STATE='PA' (get all records where field STATE is 'PA')
  - Can retrieve, update, insert, delete entire set of records in single statement

# SQL and DB2/400

- DB2 UDB for IBM i supports SQL:
  - Imbedded in source: RPG, COBOL, C, C++, REXX
  - Entered interactively: STRSQL command
  - Entered via source member: RUNSQLCMD cmd
  - Need DB2 Query Manager and *SQL Development Kit for iSeries* (57xx-ST1) for SQL CL commands and to create SQL applications
    - Do not need 57xx-ST1 to run SQL applications
  - SQL- and DDS-created files are interchangeable

- SQL and RPG
  - Can imbed SQL "statements" in RPG source
  - Start with C/Exec SQL and end with C/End-Exec
  - Compile with CRTSQLRPGI for RPG IV

# Static versus dynamic SQL

- SQL statements can be static:
  - – Can only be used imbedded inside HLL like RPG
  - – Requires hard coded database, record and field names
  - – Offer best performance (optimized at compile time)
- SQL statements can be dynamic:
  - – Imbedded in HLL or issued via STRSQL or CLI API
  - – Can choose at runtime which statement to run
  - – Can prepare often-needed statements once and replace dynamical field values on each run
- Which to use?
  - – Static if you can hardcode the queries themselves
    - You can still specify field values (versus names) via variables
  - – Dynamic if user decides the queries to run

# CLI APIs

- SQL statements can be run by calling APIs:
  - Call Level Interface (CLI) APIs part of OS/400
  - Can use them to run dynamic SQL statements
  - CLI is an SQL standard
  - CLI offers most flexibility: everything is dynamic
- When to use CLI?
  - Cannot hardcode database file or query
  - Want to support multiple database vendors
  - Want ultimate flexibility (versus performance)
- Microsoft's ODBC is a form of CLI
  - A set of C APIs that is database vendor-neutral
  - Includes APIs to query capabilities of the database
  - Good for database-neutral applications and tools

# SQL terms

- SQL uses different terminology than DB2/400:

| iSeries | SQL |
|---|---|
| DB2/400 | Database |
| Library | Collection |
| Library + objects in it | Schema |
| Physical File | Table |
| Logical File | View |
| Locical Keyed File | Index |
| Record Format | MetaData |
| Record | Row |
| Field | Column |

# SQL DDL statements

- Data Definition Language (DDL) SQL statements deal with creation and management of database (much like DDS does). For example:

| Statement | Description |
|---|---|
| CREATE SCHEMA, COLLECTION | Create a schema or collection (library) |
| CREATE TABLE | Create a new table (file) |
| CREATE INDEX | Create a new index (keyed LF) |
| CREATE VIEW | Create a new view (non-keyed LF) |
| ALTER TABLE | Add/Delete/Change columns ("record format"), add constraints, add/delete a key field |
| DROP XXX | Deletes (physically) COLLECTION/SCHEMA, TABLE, INDEX, VIEW or PACKAGE |

# SQL DML statements

- Data Management Language (DML) SQL statements deal with actual data access and manipulation (like RPG op-codes do). For example:

| Statement | Description |
|---|---|
| DECLARE CURSOR | Declares a cursor (embedded SQL only) |
| OPEN, CLOSE | Opens, closes a cursor (embedded SQL only) |
| FETCH | Reads next row from result defined by cursor (embedded SQL only) |
| SELECT | Reads one or more rows (CLI only) |
| INSERT, UPDATE, DELETE | Inserts, updates and deletes one or more rows |
| COMMIT, ROLLBACK | Commit or undo previous "transaction" (one or more INSERT, UPDATE, DELETE stmtmnts |

# SQL SELECT statement

- Most used SQL statement: SELECT
  - For example: `SELECT * FROM CUSTOMER WHERE STATE='PA'`
    - Select all records from file CUSTOMER where key field STATE has value 'PA'. Return all fields ('*')
    - `WHERE` clause defines record filtering (the "predicate")

- `SELECT` is part of the `DECLARE CURSOR` statement in embedded static SQL

- `SELECT` can also be issued directly using the CLI "SQLExecDirect" statement

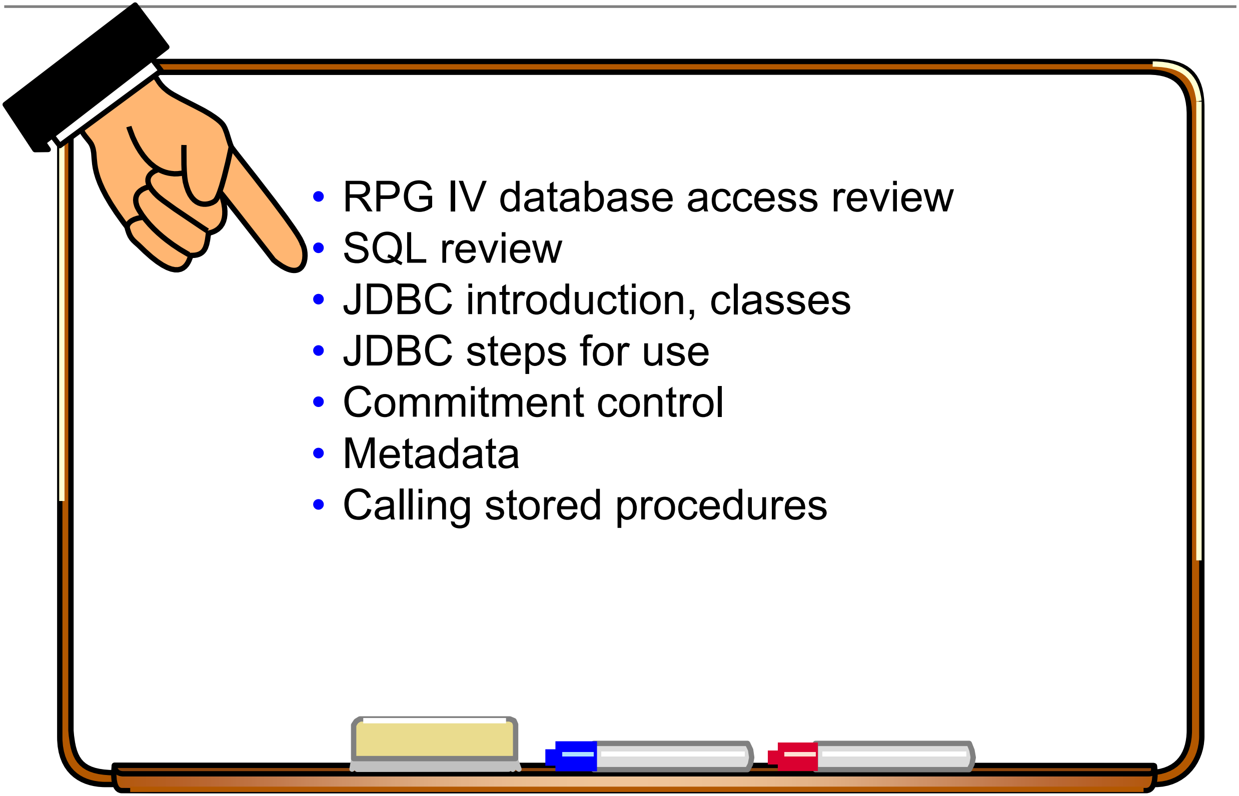
- Returns a "result set" of matching records
  - Traverse it one record a time using `FETCH` in RPG

# SQL UPDATE, DELETE

- Two methods of usage for UPDATE, DELETE:
  - Stand-alone, multi-record action using WHERE clause:
    - `UPDATE CUSTOMER SET RATE=2 WHERE STATE='PA'`
    - `DELETE CUSTOMER WHERE STATE='PA'`
  - As part of processing a result set. Acts on a single record (the current record in the result set)
    - `DELETE FROM CUSTOMER WHERE CURRENT OF cursor-name`
      - Known as "positioned update or delete"

- INSERT statement adds new records:
    - `INSERT INTO CUSTOMER (CUSTOMER, STATE, RATE) VALUES('Bobs Bait', 'MA', 1)`
  - Can also insert multiple records in single statement

# Agenda



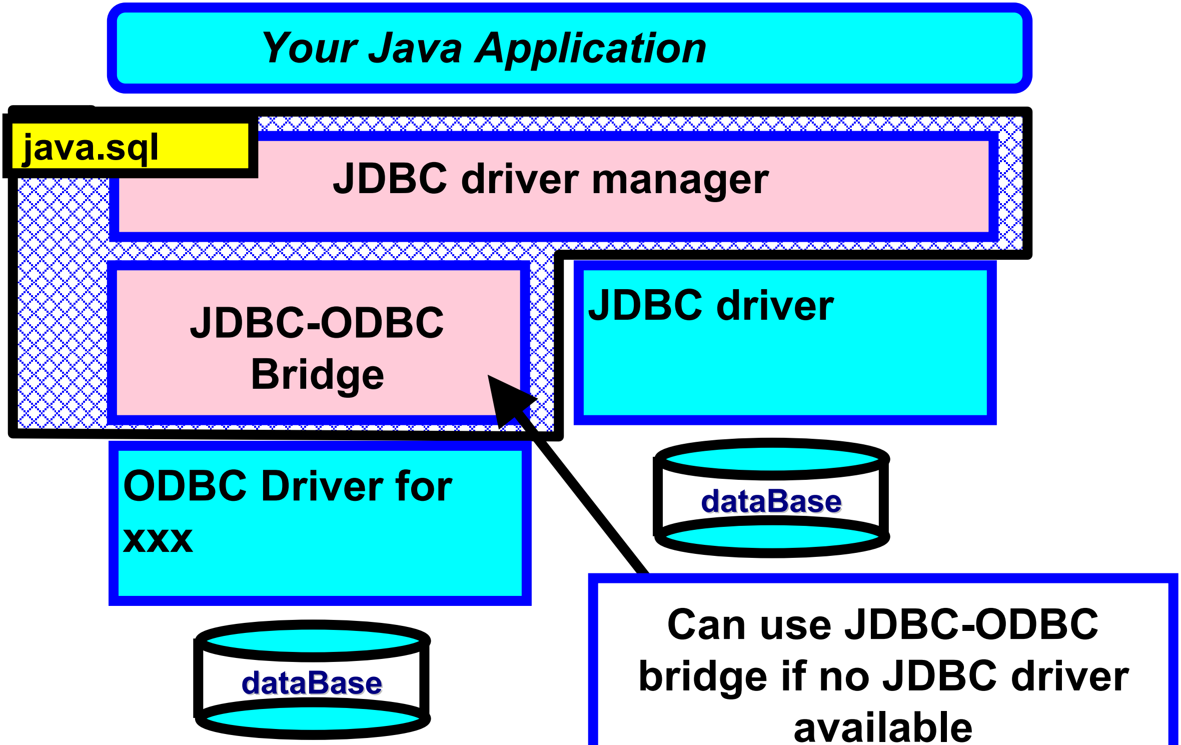
- RPG IV database access review
- SQL review
- JDBC introduction, classes
- JDBC steps for use
- Commitment control
- Metadata
- Calling stored procedures

# Java database access

- Java has an industry standard way of accessing databases: JDBC
  - – Patterned after ODBC
  - – Like a Java authored form of CLI APIs
  - – Framework of interfaces each DB vendor "implements"
    - All the interfaces are in JDK-supplied package `java.sql`

- Like ODBC, JDBC requires:
  - – Database Driver Manager
    - Built-in part of the Java language
  - – Database Drivers
    - Supplied by database vendors like IBM's DB2, Oracle, Sybase, Informix, and so forth. All major vendors now supply JDBC drivers.

# JDBC architecture



Can use JDBC-ODBC bridge if no JDBC driver available

# Java.sql package (1 of 2)

- All JDBC drivers implement the interfaces in the java.sql package as concrete classes
  - However, you simply code to the vendor-neutral interfaces

- JDBC is SQL-based
  - You execute dynamic SQL statements by passing them to methods in the JDBC interface

- You use JDBC methods to:
  - Load the JDBC driver for your DB
  - Connect to a database
  - Execute SQL statements against the database. These are passed directly to the target database
  - Close the database connection

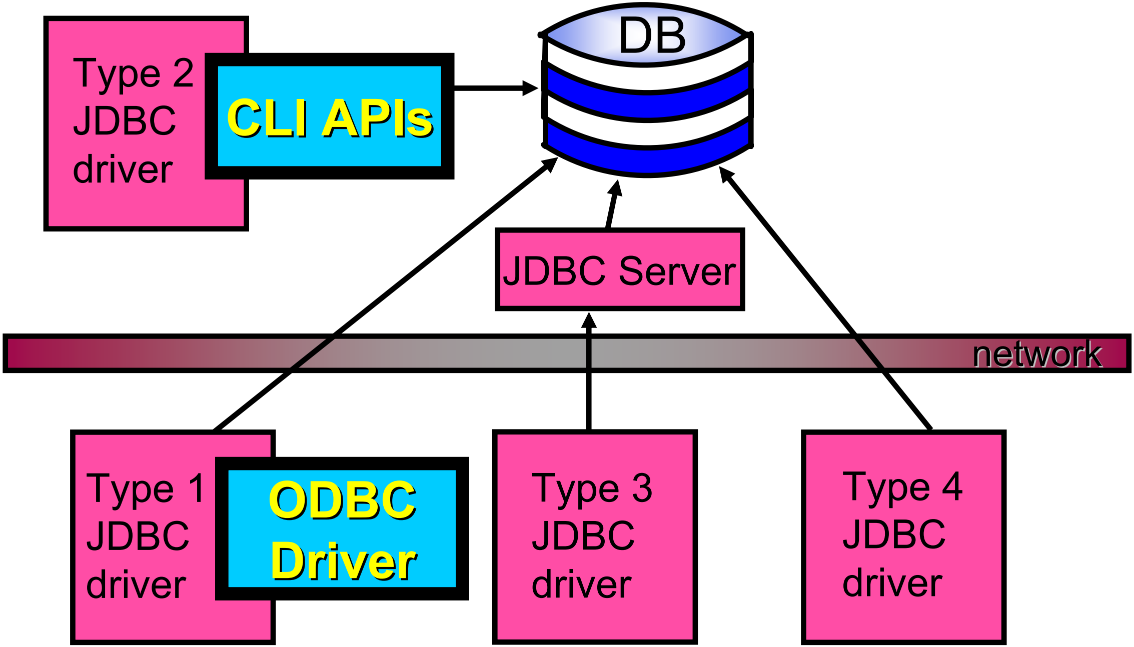
# Java.sql package (2 of 2)

| Interface | Description |
|---|---|
| `DriverManager` | Supplies connect method for accessing database |
| `Connection` | A specific session with a specific database |
| `Statement` | For executing explicit SQL statements |
| `Prepared Statement` | For executing "prepared" SQL statements (better performance if used multiple times). Extends Statement |
| `ResultSet` | Returned from SQL query or stored procedure call |
| `Callable Statement` | Used to call stored procedures. Extends PreparedStatement |
| `ResultSet MetaData` | Used to dynamically retrieve record format information |
| `Database MetaData` | Used to dynamically retrieve database meta information and catalog information |

# JDBC driver types (1 of 2)

- JDBC drivers come in four flavors:
  - Type 1
    - Supplied with the JDK, and can be used to connect to any existing ODBC driver.
  - Type 2
    - Supplied by the DB vendor, and under the covers calls existing CLI APIs for that particular database. Must be run on the same system as the database.
  - Type 3
    - Pure Java implementation, for use in remote clients. Requires code running on server (supplied by DB vendor) to listen for requests sent by the client driver.
  - Type 4
    - Pure Java implementation, for use in remote clients. No listener code required on server. Client communicates directly with the database engine on the server.

# JDBC driver types (2 of 2)



# JDBC for DB2/400

- Entire IBM DB2 family supports JDBC
  - JDBC for DB2/400 comes in two flavors:
    - iSeries Toolbox for Java JDBC driver
      - www.ibm.com/iseries/toolbox
    - iSeries Developer Kit for Java JDBC driver
      - www.ibm.com/iseries/java
  - Both are free and come with OS/400

**Type 4**

**Type 2**

- iSeries Toolbox for Java
  - Runs on OS/400 or on any client with a JVM
  - Is written entirely in Java. Uses TCP/IP for comm
  - Includes much iSeries function beyond JDBC

- iSeries Developer Kit for Java JDBC driver
  - Runs only on OS/400 and optimized for it

# Using JDBC

| Step | Example |
|---|---|
| 1. Import java.sql | `import java.sql.*;` |
| 2. Load JDBC driver | `DriverManager.registerDriver(new`<br>`com.ibm.as400.access.AS400JDBCDriver());` |
| 3. Connect to DB | `Connection conn =`<br>`DriverManager.getConnection(`<br>`"jdbc:as400://mySystem");` |
| 4. Prepare SQL statements | `Statement stmt = conn.createStatement();` *or*<br>`PreparedStatement pstmt =`<br>`conn.prepareStatement("...?...");` |
| 5. Run SQL statements | `stmt.execute(String sql);`<br>`stmt.executeQuery(String sql);`<br>`stmt.executeUpdate(String sql);` |
| 6. Retrieve results | `ResultSet rs = stmt.executeQuery(sql);`<br>`while (rs.next()) col1 = rs.getString();` |
| 7. Monitor for errors | `try {... // eg, stmt.execute(sql);`<br>`} catch (SQLException exc) { ... }` |
| 8. Close statements and connection | `rs.close(); stmt.close(); pstmt.close(); conn.close();` |

# 2. Loading driver

- Load JDBC driver into memory
  - Use static method registerDriver in DriverManager
    - Pass an instance of XXXDriver class
      - Each driver will supply a different class
        - > Toolbox driver: com.ibm.as400.access.AS400JDBCDriver
        - > Toolkit driver: com.ibm.db2.jdbc.app.DB2Driver
    - Need to monitor for SQLException

```
import java.sql.*; // import JDBC package
```

```
try
{
   DriverManager.registerDriver(
      new com.ibm.as400.access.AS400JDBCDriver());
   //or new com.ibm.db2.jdbc.app.DB2Driver());
} catch(SQLException exc)
{
   System.out.println("DB2/400 JDBC driver not found!");
   System.exit(1);
}
```

# 3. Connecting to DB

- Connect to the database
  - Use static getConnection method in DriverManager
    - Supply a string (URL format) identifying the "database" to connect
  - Each JDBC driver recognizes unique URL syntax
    - Toolbox driver: "jdbc:as400://system-name"
    - Toolkit driver: "jdbc:db2://system-name"

```
Connection conn; // instance variable
String     sys;  // set to system name
...
try
{
   String url = "jdbc:as400://" + sys;
   conn = DriverManager.getConnection(url);
} catch (SQLException exc)
{
   System.out.println("connect failed with: '" +
                      exc.getMessage() + "'");
}
```

# Connection properties (1 of 3)

- You can also specify default library and properties in JDBC/400 connection URL:
    - `"jdbc:as400://system-name</default-library<;list-of-properties>>"`
- Properties:
    - `"property=value"` syntax (semicolon delimited)
        - See toolbox or ADK documentation for list
- Some interesting properties:
    - user=xxxx: user name for signon
    - password=xxxx: user password for signon
    - naming=sql or =system: use SQL (lib.file) or AS/400 (lib/file) syntax for qualifying files. Default is "=sql"
    - access=all or =read call or =read only (read/write, read/stored-proc, read-only)

# Connection properties (2 of 3)

- Be careful:
  - SQL versus system naming affects how unqualified file names are found:
    - SQL ==> search library with same name as userid
    - SYSTEM ==> search library list
  - Default library name property, if specified, is used to find unqualified file names

# Connection properties (3 of 3)

- User ID and Password: three ways to get it
  - Specify as properties on connection URL
  - Specify as second and third parms on getConnection:

```
getConnection(
 "jdbc:as400://mySystem","myUserId","myPassword");
```

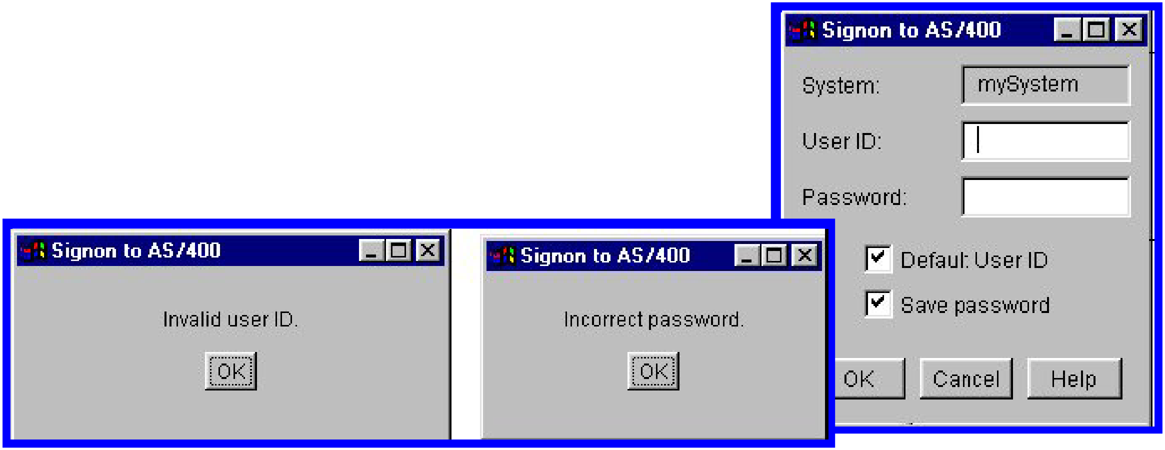
  - Do not specify it! User will be prompted for it
    - UI, expired passwords and invalid input handled for you by JDBC/400!

- System name
  - For Toolbox JDBC driver:
    - Either DNS name or TCP/IP address of AS/400
  - For Toolkit JDBC driver:
    - Use host name as specified in WRKRDBDIRE (*LOCAL entry)

Be sure STRHOSTSRV(*ALL) is running!

# System login

- For Java running on client, Toolbox classes (including JDBC) automatically handle all aspects of login:
  - Prompt for system name, user ID, password
    - If any of them were not specified programmatically
  - Inform user of expired password



Signon to AS/400

Invalid user ID.

Signon to AS/400

Incorrect password.

Signon to AS/400

System: mySystem

User ID:

Password:

☑ Default User ID

☑ Save password

# 4. Preparing statements

- To run an SQL statement, you need to create an object

- Two ways to run statements:
  - Dynamically: use Statement objects
  - Prepared: use PreparedStatement objects

- Creating Statement objects:
  - `Statement stmt = conn.createStatement();`

- Creating PreparedStatement objects:
  - `PreparedStatement = conn.prepareStatement(str);`

- When to use which?
  - Need to run statement only once? Statement
  - Need to run statement more than once? Prepared

# Prepared statements

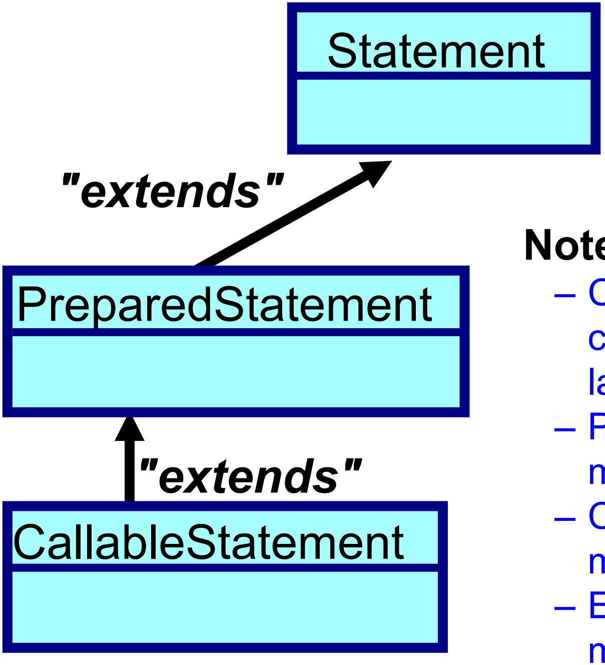
- For repeated-use statements, PreparedStatement offers better performance.
- Must specify statement string at create time:

```
PreparedStatement pstmt = conn.prepareStatement(
     "SELECT * FROM CUSTOMER WHERE CUSTNO=?");
```

- Can use "markers" (question marks) as substitution variables for data values.
- Database called to verify statement string.
- Must monitor for SQLException in case statement not valid.

```
try {
     pstmt = conn.prepareStatement(...);
} catch (Exception exc) { ... }
```

# Statement hierarchy



**Notes:**

- CallableStatement is used for calling stored procedures (covered later)
- PreparedStatement inherits all methods in Statement
- CallableStatement inherits all methods in PreparedStatement
- Each adds to and overrides some methods of parent

# 5. Running statements

- Having created a Statement or PreparedStatement object, it is time to run it
  - Use one of two methods on object:
    - executeQuery: for an SQL SELECT statement
    - executeUpdate: for any other SQL statement, such as INSERT, DELETE, UPDATE, or DDL statement

- ExecuteQuery returns a ResultSet object:

```
ResultSet rs = stmt.executeQuery(" SELECT * FROM
CUSTOMER WHERE STATE='PA' ");
```

- ExecuteUpdate returns number of affected records:

```
int count = stmt.executeUpdate(" DELETE FROM
CUSTOMER WHERE STATE='PA' ");
```

# Running prepared statements

- Prepared statements do not take string parameter to executeQuery and executeUpdate methods
  - You already specified this at object create time!

```
ResultSet rs = pstmt.executeQuery();
int count = pstmt.executeUpdate();
```

- Did you specify markers (?) at create time?

```
PreparedStatement pstmt = conn.
  prepareStatement(" SELECT * FROM
          CUSTOMER WHERE STATE = ?");
```

  - You must substitute these before running it!
- Marker substitution:
  - Use PreparedStatement setXXX() methods
  - Method depends on type of value (for example, setString)
  - First parm is RRN of marker, second is value:

```
setString(1, "PA");
```

# Marker set methods

| Method | SQL Type | DDS Type |
|---|---|---|
| setBigDecimal | NUMERIC | decimal |
| setBoolean | SMALLINT | binary(4,0) |
| setByte | SMALLINT | binary(4,0) |
| setBytes | VARBINARY | char CCSID(65535) |
| setDate | DATE | date |
| setDouble | DOUBLE | float FLTPCN(*DOUBLE) |
| setFloat | FLOAT | float, precision depends on value |
| setInt | INTEGER | binary(9,0) |
| setLong | INTEGER | binary(9,0) |
| setNull | NULL | ALWNULL keyword |
| setShort | SMALLINT | binary(4,0) |
| setString | CHAR, VARCHAR | char, VARLEN char |
| setTime | TIME | time |
| setTimestamp | TIMESTAMP | timestamp |

# 6. Retrieving result sets

- executeQuery returns a ResultSet object that is the result of the SELECT statement:

```
ResultSet rs = pstmt.executeQuery(); // ... or ...
ResultSet rs = stmt.executeQuery(
    "SELECT * FROM CUSTOMER WHERE STATE = 'PA'");
```

- The ResultSet object represents the list of all rows (records) that meet the SELECT criteria
  – Note they are not all actually in memory, but retrieved one at a time or a block at a time
- Iterate through list using "next()", until it returns false (end of list):

```
while ( rs.next() )
{
  ... process current row ...
}
```

# Processing result sets

- Want just one row (record)? Same process:

```
ResultSet rs = stmt.executeQuery(
  "SELECT * FROM CUSTOMER WHERE CUSKEY = 123456");
if ( !rs.next() ) // not even one record returned?
  // error: record not found
else
  // process the first and only record
```

- How do you "process" each row?
  - You need to 'extract' the value for each column (field) into your own Java variable
  - Use appropriate getXXX method of ResultSet

```
String stateName = rs.getString("STATE");
```

- One getXXX method for each Java type
  - All methods take as the first parameter either the 1-based relative column number or the column name

# GetXXX methods

| Method | SQL Type | Description |
|---|---|---|
| getBigDecimal | DECIMAL, NUMERIC | Returns a java.math.BigDecimal object |
| getBoolean | NUMERIC | Returns false for zero or null values |
| getByte | SMALLINT | Returns a byte variable |
| getBytes | BINARY, VARBINARY | Returns a byte array: byte[] |
| getBinaryStream | BINARY, VARBINARY | Returns java.io.InputStream object |
| getAsciiStream | CHAR, BINARY, VARxxx | Returns java.io.InputStream object |
| getUnicode Stream | CHAR, BINARY, VARxxx | Returns java.io.InputStream object |
| getDate | CHAR, VARCHAR, DATE, TS | Returns a java.sql.Date object |
| getDouble | FLOAT, DOUBLE | Returns a double variable |
| getFloat | REAL | Returns a float variable |
| getInt | INTEGER | Returns an int variable |
| getLong | INTEGER | Returns a long variable |
| getShort | SMALLINT | Returns a short variable |
| getString | any | Returns java.lang.String object |
| getTime | CHAR, VARCHAR, TIME, TS | Returns java.sql.Time object |
| getTimeStamp | CHAR, VARCHAR, DATE, TS | Returns java.sql.Timestamp object |

**Note: TS == TIMESTAMP**

# Other ResultSet methods

- findColumn(String name): returns relative column number, given a column name:

```
int columnIndex = rs.findColumn("STATE");
String stateValue = rs.getString(columnIndex);
```

- wasNull(): returns true if previous getXXX column value was "null":

```
String stateValue = rs.getString(columnIndex);
if ( rs.wasNull() )
  System.out.println("State not set");
else
  System.out.println("State = " + stateValue);
```

- getCursorName(): returns implicitly assigned name of this result set's cursor:

```
String sql = "DELETE FROM CUSTOMER WHERE CURRENT OF ";
sql += rs.getCursorName();
Statement stmt = conn.createStatement();
stmt.executeUpdate(sql);
```

# Other statement methods

- setCursorName(String name): explicitly sets result set cursor name versus using generated name:

```
Statement stmt = conn.createStatement();
stmt.setCursorName("ResultSet1");
ResultSet rs = stmt.executeQuery(
     "SELECT FROM CUSTOMER WHERE STATE='PA'");
...
```

- cancel(): cancels long running SQL query. Use it when executeQuery is run in a thread:

```
if (userPressedStop)
  stmt.cancel();
```

- setMaxTimeout(int seconds): sets upper time limit for executing the statement:

```
stmt.setMaxTimeout(60 * 5); // max time = 5 minutes
```

# 7. Handling errors

- Place all  JDBC code inside try-catch blocks for SQLException errors. The following throw it:
  - DriverManager's registerDriver, getConnection methods
  - All Connection methods
  - All Statement methods
  - All PreparedStatement methods
  - All CallableStatement methods
  - All ResultSet methods
  - All ResultSetMetaData methods
  - All DatabaseMetaData methods

# SQL warnings

- As well, statement execution may result in SQL warnings:
  - These are exceptions of class SQLWarning
  - Use Statement methods clearWarnings() and getWarnings() to clear and retrieve any warnings after executeXXXX
  - Use SQLWarning method getNextWarning() to iterate through multiple warnings

```
stmt.clearWarnings(); // stmt == Statement object
stmt.execute(sqlString); // execute sqlString SQL statement
SQLWarning warning = stmt.getWarnings();
while (warning != null)
{
  warning = warning.getNextWarning();
  System.out.println("Warning: " + warning.getMessage());
}
```

# 8. Closing

- To free up database resources, use close() method when done with each of:
  - ResultSet objects
  - Statement, PreparedStatement, CallableStatement objects
  - Connection objects

- If you do not call close, the garbage collector will when the object is swept up.

- But better to do it yourself to ensure efficient use of resources

# DB query example

```
try
{
  if (pstmt == null) // only create once
    pstmt = conn.prepareStatement(
               "SELECT * FROM QGPL.QAUOOPT");
  ResultSet rs = pstmt.executeQuery();
  System.out.println("query results:");
  while (rs.next())
         System.out.println(
           rs.getString(1) + " " +
           rs.getString(2).trim());
  rs.close();
  pstmt.close();
} // end try
catch (SQLException exc)
{
    System.out.println(
      "query all failed with: '" +
      exc.getMessage() + "'");
    return false;
}
System.out.println("query done");
return true;
```

```
query results:
 C CALL &O/&N
 CC CHGCURLIB
 CURLIB(&L)
 CD STRDFU
 OPTION(2)
   ...
 TD STRSDA
 OPTION(3)
 TSTFILE(&L/&N)
 WS WRKSBMJOB
query done
```

# JDBC commitment control

- Default is no commitment control

- Use Connection object's method setTransactionIsolation(...) to enable it:
  - Parameter is a constant from java.sql.Connection interface
  - Choose the type of row locking relative to other transactions
  - Can also specify it as a "transaction isolation" property on the getConnection statement URL

# Transaction isolation

| Property | Method Parameter | CRTSQLxx COMMIT parm |
|---|---|---|
| "none" | TRANSACTION_NONE | *NONE or *NC |
| "read committed" | TRANSACTION_READ_COMMITTED | *CS |
| "read uncommitted" | TRANSACTION_READ_UNCOMMITTED | *CHG or *UR |
| "repeatable read" | TRANSACTION_REPEATABLE_READ | *RS |
| "serializable" | TRANSACTION_SERIALIZABLE | *RR |

- What about commit and rollback?
  - Use conn.setAutoCommit(false);
  - Explicitly do conn.commit(); to commit
  - Explicitly do conn.rollback(); to undo

# ResultSetMetaData

- What about level checks?
  - Sorry, not supported in JDBC
  - They are AS/400 unique

- Do I have to hardcode datatypes and field names?
  - No, you can query them dynamically
  - Use ResultSetMetaData object, returned by ResultSet method getMetaData():

```
ResultSet rs = stmt.executeQuery("SELECT ... ");
ResultSetMetaData rsmd = rs.getMetaData();
```

  - ResultSetMetaData has methods for querying record format information

# ResultSetMetaData methods

| Method | Description |
|---|---|
| `getColumnCount` | Number of columns |
| `getColumnDisplaySize` | Display size needed to show this column's value. For example, for decimal columns it is the total length plus 2 for the decimal point and a sign. |
| `getColumnName` | Column name (DDS field name) |
| `getColumnType` | Column data type. One of the java.sql.Types integer constants |
| `getColumnTypeName` | Same as above, but as readable string |
| `getPrecision` | Total length (including decimals) |
| `getScale` | Decimal positions |

# ResultSetMetaData example

- Query record format information about file QCUSDATA in library QPDA:

```
Statement stmt = conn.createStatement();
ResultSet rs = stmt.executeQuery("SELECT * FROM QPDA.QCUSDATA");
ResultSetMetaData rsmd = rs.getMetaData();
int nbrColumns = rsmd.getColumnCount();
int colIdx;
System.out.println("Column Information");
System.out.println("==================");
System.out.println("Name       " + "Label      " + "TypeName   "
                    + "Digits " + "Decs " + "DispSize ");
System.out.println("-----------" + "-----------" + "-----------"
                    + "-------" + "-----" + "---------");
for (colIdx=1; colIdx<nbrColumns; colIdx++) // loop through columns
{
   System.out.print(padString(rsmd.getColumnName(colIdx),11));
   System.out.print(padString(rsmd.getColumnLabel(colIdx),11));
   System.out.print(padString(
                    rsmd.getColumnTypeName(colIdx),11));
   System.out.print(padString(rsmd.getPrecision(colIdx),8));
   System.out.print(padString(rsmd.getScale(colIdx),5));
   System.out.println(padString(
                    rsmd.getColumnDisplaySize(colIdx),9));
}
```

# Example output

- Output of previous example:

```
Column Information
==================
Name        Label       TypeName    Digits Decs DispSize
-------------------------------------------------------
CUST        CUST        CHAR        5       0    5
NAME        NAME        CHAR        20      0    20
ADDRESS     ADDRESS     CHAR        20      0    20
CITY        CITY        CHAR        20      0    20
STATE       STATE       CHAR        2       0    2
ZIP         ZIP         DECIMAL     5       0    7
SEARCH      SEARCH      CHAR        6       0    6
CUTYPE      CUTYPE      CHAR        1       0    1
ARBAL       ARBAL       DECIMAL     8       2    10
ORDBAL      ORDBAL      DECIMAL     8       2    10
LSTAMT      LSTAMT      DECIMAL     8       2    10
LSTDAT      LSTDAT      DECIMAL     6       0    8
CRDLMT      CRDLMT      DECIMAL     8       2    10
SLSYR       SLSYR       DECIMAL     10      2    12
```

# DataBaseMetaData

- Did you notice "name" == "label" in previous example?
  - ResultSetMetaData doesn't return TEXT keyword
- How do you get it?
  - Use DatabaseMetaObject, returned by Connection method getMetaData():

    ```
    DatabaseMetaData dbmd = conn.getMetaData();
    ```
  - Use getColumns() method to get ResultSet of info
  - Use column 12 to get TEXT value

    ```
    DatabaseMetaData dbmd = conn.getMetaData();
    ResultSet rs =
     dbmd.getColumns(null,"QPDA","QCUSDATA",null);
    String label;
    while (rs.next())
       label = rs.getString(12);
    ```

No way to get COLHDG!

# DatabaseMetaData object

- DatabaseMetaData also has much information about the database itself.
  - Database support information
  - Database terminology information
  - Database name and version information
  - Catalog information
  - Column (field) information
  - Stored procedure information
  - Cross reference information
  - Exported and imported information
  - Index information

# Stored procedures

- Stored procedures are *PGM objects on iSeries that you can call with an SQL statement
  - You can pass updateable parameters
  - They can return result sets

- Standard way to call programs from client

- Efficient use of network:
  - Data filtering done at server, not client

# CallableStatement

- In JDBC, use CallableStatement to call a stored procedure
  - CallableStatement extends PreparedStatement, which extends Statement

- Parameters passed are defined as one of:
  - input-only => read by called program
  - output-only => set by called program
  - input-output => read by, set by, called program

- Called program can use embedded SQL to define a cursor (result set) that is returned by the CallableStatement executeQuery method.
  - In fact, it can return multiple result sets! Use execute method in this case.

# Calling stored procedures

- Steps for calling a stored procedure:
  - Use Connection's prepareCall method, with a CALL statement and markers for parameters:

```
CallableStatement proc1 =
    conn.prepareCall("CALL MYLIB/MYPROG(?,?,?)");
```

  - Set input, input-output parameter values and types using setXXX methods:

```
proc1.setInt(1, 1); // marker position, parm value
proc1.setString(2, "a string"); //mrkr posn, parm val
```

  - Set out, in-out parameter types using registerOutParameter method:

```
proc1.registerOutParameter(2, java.sql.Types.CHAR);
```

  - Call it! Use executeQuery method. Or execute method if it returns multiple result sets

```
ResultSet rs = proc1.executeQuery();
```

  - Retrieve value of output, input-output parameters using getXXX methods

```
String parm2Value = proc1.getString(2);
```

  - Process the result set, if any, using rs.next and rs.getXXX methods

# New in JDBC 2.0

- JDBC was significantly enhanced as of JDK 1.2
  - Known as "jdbc 2.0"
  - DB2/400 JDBC drivers support these enhancements
- Enhancements include:
  - New data types
    - BLOB, CLOB, ARRAY, STRUCT, REF, DISTINCT, JAVA_OBJECT
    - New getXXX and setXXX methods in Statement and ResultSet classes
  - ResultSet enhancements
    - Methods for moving forward and backward incrementally or absolutely
    - Methods for positioned insert, update, and delete
    - Methods for setting and getting fetch direction and fetch size
    - New updateXXX methods for updating field values of current row
    - Miscellaneous enhancements
  - Batch updates
    - Accumulate multiple SQL statements and execute them as a single operation
    - Supports both immediate and prepared style

# New in JDBC 3.0

- JDBC was again enhanced as of JDK 1.4
  - Known as "jdbc 3.0"
  - DB2/400 JDBC 1.4 drivers support these enhancements
- Enhancements include:
  - Universal data access
    - Access any kind of data (not just relational) via JDBC interface
  - Other enhancements
    - Savepoints in transactions
    - Keeps result sets open after commit
    - Multiple result sets open at same time
    - Re-use of prepared statements
    - Retrieve keys that are automatically generated
    - Two new datatypes: BOOLEAN and DATALINK
    - Establishes relationship between JDBC Service Provider interface and the Java Connector Architecture (JCA)

# SQLJ

- Another option for Java on the AS/400 itself is SQLJ
  - This is embedded SQL inside Java
  - Industry standard, initiated by Oracle

- SQLJ involves:
  - Syntax for embedding SQL statements
    - For example: #sql { DELETE FROM CUSTOMER WHERE STATE=:state };
  - .sqlj file extension versus .java
  - different syntax for compiling
    - java sqlj.tools.Sqlj MyClass.sqlj

- Why SQLJ?
  - Potentially better performance because of static versus dynamic
    - However, requires at least V4R5 for this performance gain

# Designing data access

- Suggestion when writing database access code
  - Isolate the database access inside a class so that other programmers just use the class, and do not write their own direct database access
    - For example, supply a Customer class, Employee class, Order class, ...
  - Each field in database becomes an instance variable in class
    - Supply methods for:
      - Setting the key field values
      - Reading the values from the database
      - Getting the non-key field values
      - Setting the non-key field values
      - Updating the database from current values in the instance variables
      - Deleting this record from the database

# Power Systems Announcements for IBM i 6.1

- JDBC was again enhanced as of JDK 5.0
  - JDBC 4.0
  - DB2/400 JDBC 4.0 drivers support these enhancements
- 64 KB DB Page size
  - i5/OS V5R4 PTFs
  - IBM I 6.1
    - Supply methods for:
      - Setting the key field values
      - Reading the values from the database
      - Getting the non-key field values
      - Setting the non-key field values
      - Updating the database from current values in the instance variables
      - Deleting this record from the database

# Final notes

- WebSphere Studio Application Developer's Data Access wizards make it easy to access any JDBC data.
  - Configure database and create SQL statements via wizards

- References:
  - *Accessing the AS/400 with Java*. IBM Redbook. sg24-2152-00
  - *Database Design and Programming for DB2/400*. Paul Conte, Duke Press. ISBN 1882419065
  - *JDBC Database Access with Java - A Tutorial and Annotated Reference*. Hamilton, Cattell and Fisher. JavaSoft Press, Addison-Wesley. 0-201-30995-5

# Topics covered

- RPG IV database access review
- SQL Review
- JDBC introduction, classes
- JDBC steps for use
- Commitment control
- Metadata
- Calling stored procedures

# Unit summary

Having completed this unit, you should be able to:

- Describe basic SQL syntax
- Describe the structure of JDBC API, including the interfaces it provides and the process for accessing iSeries and IBM i data
- Leverage commitment control and stored procedures within JDBC