Host Language Database Programming: JDBC

COM 3563: Database Implementation

Avraham Leff

Yeshiva University

avraham.leff@yu.edu

COM3563: Fall 2020

Today's Lecture: Overview

1. Database Programming: Review & Some Context

2. Using JDBC APIs

3. When & Whether To Use JDBC?

Database Programming: Review & Some Context

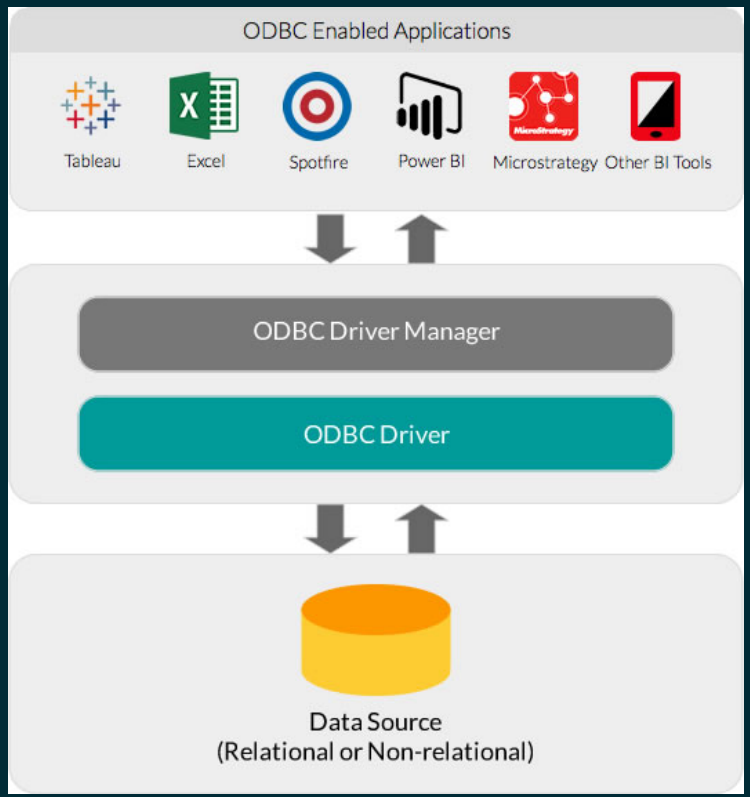
Introduction

- Last lecture we discussed the motivation for, and general approaches to, bridging SQL & general programming language code
- We discussed the static embedding approach (in Java, this is SQLJ)
- We discussed the "library" approach in which a host-language is provided with language-specific APIs for accessing a relational database
- The most prevalent standard: ODBC (early 1990s)
- Key API features:
  - Open a connection with a database
  - Send queries and updates to the database "over" the connection
  - Receive result-sets (and results in general) from the database "through" the connection

Commonalities

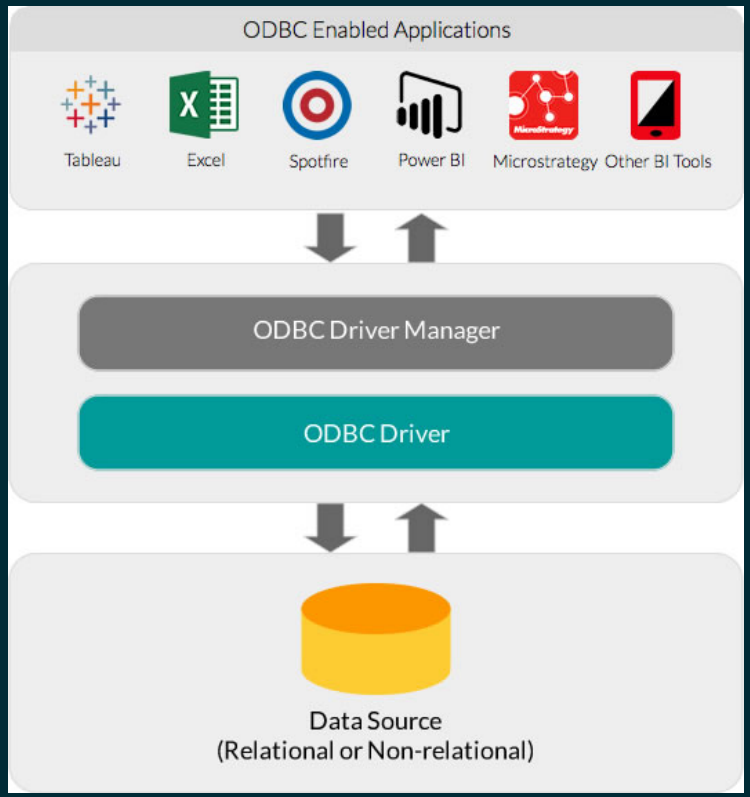
- I stressed that the ODBC approach doesn't differ fundamentally from the embedded approach …
- Meaning: the code emitted by the pre-processor in the static embedded approach also opens a database connection and uses it for bilateral communication with the database
- In both approaches, bindings must be established between host-language variables and state that will be passed to and from the program and the database
- The difference is whether these bindings are
  - Static: depend on the state of the database at the time that the code was written (at the time that the bindings were generated in a pre-processor phase) or …
  - Dynamic: able to access the current state of the database

Architectural Advantage of ODBC (I)



- The ODBC approach has another (software architecture) advantage that's related to the dynamic, "at runtime", advantage that we've been discussing
- By using the well-known CS principle of "indirection", the ODBC design allows applications to be independent of database systems and OS ☺
- Implication: an application that uses ODBC can be ported to different clients and databases almost painlessly

Architectural Advantage of ODBC (II)

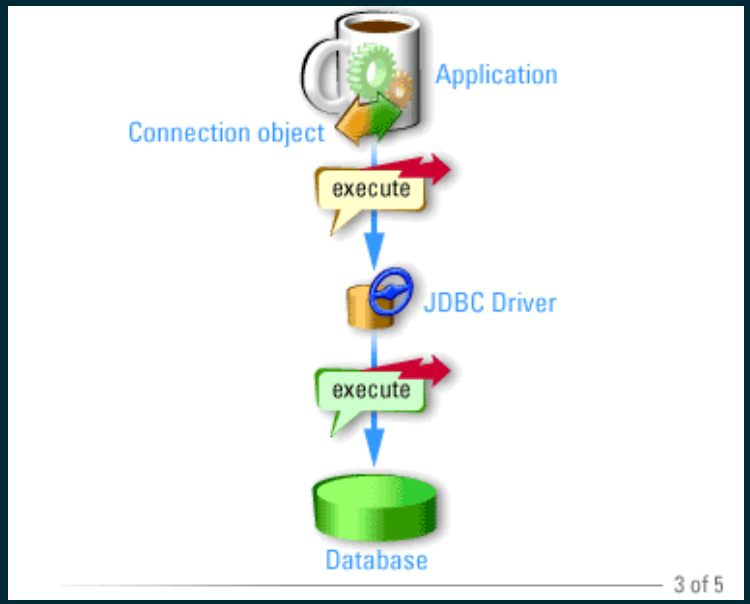


- The architectural "trick" is the use of an ODBC driver layer (between application & DBMS)
- When the application invokes the "library functions" that we've been discussing, the driver translates the invocation into something that the specific DBMS understands
- Analogy: printer "drivers"
- ODBC drivers exist for non-DBMS data sources!
  - Examples: spread-sheets, CSV files, address books

Today: JDBC

- JDBC ("ODBC for Java") is the approach of choice when using the "library" approach for host-language database programming in Java
- Although conceptually identical to ODBC, JDBC could be a textbook example of the benefits of object-oriented programming ☺
- Additional benefit: if you understand JDBC, will be easy for you to do database programming with other languages
- This is why I'm not simply pointing you to the JDBC tutorial
- JDBC material is often taught as a way to introduce SQL commands and relational database concepts
  - We've already covered that ☺
- Instead: we'll dive into the nitty-gritty details of how a Java client can
  - Issue queries to a database and get data back
  - Execute DML to modify the database state

Event-Sequence For Using JDBC



1. Client application connects to the database
   - Typically requires authentication and a system-specific database “url”
2. Client application issues an SQL command (“CRUD”)
   - Can be arbitrarily complicated (“nested query + aggregation operators”), so long as syntax is supported by database
3. JDBC driver
   - 3.1 Translates SQL command into database’s proprietary network format
   - 3.2 Issues command to database
   - 3.3 Translates result (including response & error codes) into JDBC API format, and returns result to client

Using JDBC APIs

Take-away Lessons

- We're going to take a look at a small, self-contained, JDBC program
  - Use that code as a "walk-through" of the basic JDBC APIs and execution flows
- As we do this, keep in mind: the capabilities demonstrated by this program are simply impossible for the "embedded SQL" approach!
- Why? because they are (almost) completely dynamic ...
  - Granted, we do "hard-code" the name of the table
  - But: the program dynamically interrogates the "meta-data" to determine the number of, and types of the result-set's columns
  - And: the program dynamically interrogates the values in the result-set and displays them to the user
- For the sharp-eyed amongst you: the code does issue a query that includes the attribute names to be selected
  - But we didn't need to do that: could have simply issued a `select *` query

The *AuthorsAndBooks* Assignment

- The next few slides are background for the *AuthorsAndBooks* JDBC assignment
  - I've borrowed them from the "Deitel & Deitel" textbook
  - You may want to read Chapter 24 which does a good job discussing JDBC ☺
- Our database is a very small, but well-designed, set of three tables
- The sample code we're discussing only queries one of these tables
- The assignment requires you to use JDBC "to glue these tables back together" in "object-space"

Books Database: I

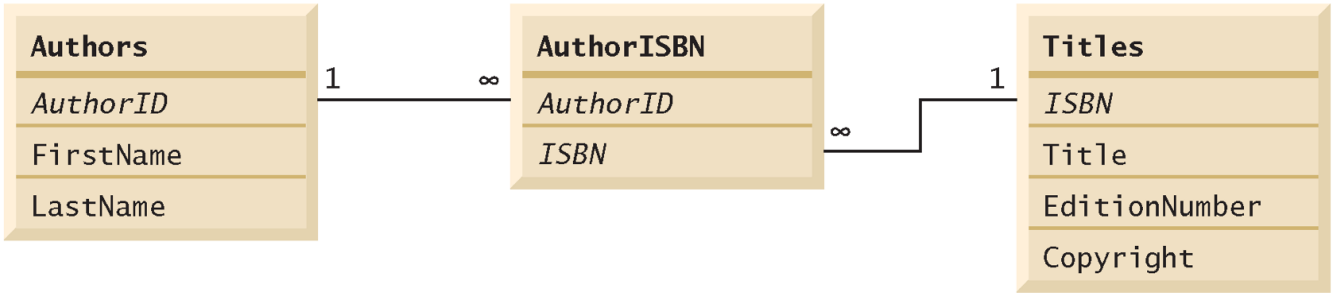


Fig. 24.9 | Table relationships in the books database.

- Database consists of three tables: *Authors*, *AuthorISBN*, and *Titles*
- One-to-many relationship between "Authors" and "AuthorISBN"
  - A single author (the AuthorID attribute) may write an arbitrary number of books
  - Hence the foreign key specification on "AuthorISBN" for this attribute

Books Database: II

Fig. 24.9 | Table relationships in the books database.

- Database consists of three tables: *Authors*, *AuthorISBN*, and *Titles*
- One-to-many relationship between "Titles" and "AuthorISBN"
  - One book (the ISBN attribute) may have been written by many authors
  - Hence the foreign key specification on "AuthorISBN" for this attribute

Books Database: III

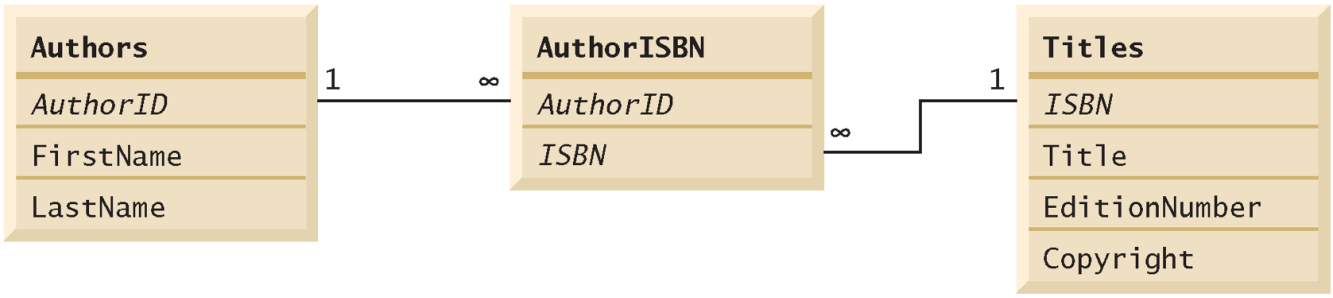


Fig. 24.9 | Table relationships in the books database.

- Database consists of three tables: *Authors*, *AuthorISBN*, and *Titles*
- Database design: there is a many-to-many relationship between "Authors" and "Titles"
  - This relationship is mediated by the "AuthorISBN" table
  - Any author may write many books
  - Any book may be written by many authors

Book Database: Attributes

- Almost all of the relation attributes are straightforward
- One exception: authorID
  - The domain is not the issue: that's INT
  - What's interesting is how those values are generated

Author's ID number in the database. In the books database, this integer column is defined as **autoincremented**—for each row inserted in this table, the AuthorID value is increased by 1 automatically to ensure that each row has a unique AuthorID. This column represents the table's primary key. Autoincremented columns are so-called identity columns. The SQL script we provide for this database uses the SQL **IDENTITY** keyword to mark the AuthorID column as an identity column. For more information on using the IDENTITY keyword and creating databases, see the Java DB Developer's Guide at http://docs.oracle.com/javadb/10.10.1.2/devguide/derbydev.pdf.

Database Driver Discovery

```
1  public static void main(String args[]) {
2    final String DATABASE_URL = "jdbc:derby:books";
3    final String SELECT_QUERY =
4      "SELECT authorID, firstName, lastName FROM authors";
```

- How does the program know which database driver to use?
  - DB2 versus Derby, for example
- The secret is the database URL
  - **jdbc** is the communication protocol
  - **derby** is the communication sub-protocol
  - The third component of the URL (**books**) specifies the database name
- At runtime, the JVM will search your classpath for a JDBC driver that can process the database URL that you've specified
  - JDK 8 used to ship with Derby, but no longer does
  - That's one reason that we'll be using POSTGRESQL
  - Maven does the classpath lookup automatically if you've specified the right dependency
  - Whatever works ...

Database URLs

RDBMS	Database URL format
MySQL	`jdbc:mysql://`*hostname*`:`*portNumber*`/`*databaseName*
ORACLE	`jdbc:oracle:thin:@`*hostname*`:`*portNumber*`:`*databaseName*
DB2	`jdbc:db2:`*hostname*`:`*portNumber*`/`*databaseName*
PostgreSQL	`jdbc:postgresql://`*hostname*`:`*portNumber*`/`*databaseName*
Java DB/Apache Derby	`jdbc:derby:`*dataBaseName* (embedded; used in this chapter) `jdbc:derby://`*hostname*`:`*portNumber*`/`*databaseName* (network)
Microsoft SQL Server	`jdbc:sqlserver://`*hostname*`:`*portNumber*`;databaseName=`*dataBaseName*
Sybase	`jdbc:sybase:Tds:`*hostname*`:`*portNumber*`/`*databaseName*

Connecting to the Database: I

```
1    try (
2         Connection connection = DriverManager.getConnection
3         (DATABASE_URL, "dbimpl_user", "dbimpl_password");
4         Statement statement = connection.createStatement();
```

- Note: try-with-resource syntax!
- A very big blessing ☺
- Ensures that all instances of classes that implement the *AutoCloseable* interface will be automagically closed for us
  - When the try block terminates
  - Or: when the exception block terminates
  - Otherwise: your programs can suddenly run out of JDBC resources

Connecting to the Database: II

```
1    try (
2         Connection connection = DriverManager.getConnection
3         (DATABASE_URL, "dbimpl_user", "dbimpl_password");
4         Statement statement = connection.createStatement();
```

- Unfortunately, your JDBC code will often compile fine
- Then fail at runtime

> Note: this is the "flip side" of the advantages of the dynamic nature of the "library" approach for database programming ☹

- Examples:
  - Did the JVM find the database driver that you specified (on your classpath)?
  - Are the *username* and *password* parameters valid?
- You'll only find out, at runtime, via the *SQLException* ☹

Executing a Query

```
1       Connection connection = DriverManager.getConnection(
2       DATABASE_URL, "dbimpl_user", "dbimpl_password");
3       Statement statement = connection.createStatement();
4       ResultSet resultSet = statement.executeQuery(
            SELECT_QUERY)) {
```

- If *getConnection* succeeds, you'll have a "handle" (Connection) through which you can interact with the database
- You can use the Connection to generate a Statement
  - Statements are the mechanism for asking the database to execute an SQL statements on your behalf
- If your SQL statement is a query, use the *executeQuery* API
- If your query succeeds, the relation is returned as a *ResultSet*
  - Your query will not succeed if there was anything wrong with the SQL syntax in `SELECT_QUERY`!
  - Remember: JDBC can't do this compile-time checking for you

Processing a ResultSet: MetaData

```
1  ResultSetMetaData metaData = resultSet.getMetaData();
2  int numberOfColumns = metaData.getColumnCount();
3  for (int i = 1; i <= numberOfColumns; i++) {
4    System.out.printf("%-8s\t", metaData.getColumnName(i));
5  }
```

- SQL Relations are fundamentally self-describing (via their schema)
- JDBC ResultSetMetaData is the construct that gives you this information
- Read the Javadoc: it's very rich ...
- Here we use the meta-data to determine the number of columns ("attributes") in the result-set and the names of those columns
- We could also get the attribute types from this meta-data

ResultSet: Masking the Cursor

```
1       while (resultSet.next()) {
2         for (int i = 1; i <= numberOfColumns; i++) {
3           System.out.printf("%-8s\t", resultSet.getObject(i));
4         }
5       }
```

> "A ResultSet object maintains a cursor pointing to its current row of data. Initially the cursor is positioned before the first row. The next method moves the cursor to the next row, and because it returns false when there are no more rows in the ResultSet object, it can be used in a while loop to iterate through the result set."

- You must first invoke `resultSet.next()` to advance the cursor to point to the first tuple
- We're cheating with `getObject`: should really use `ResultSetMetaData.getColumnType` followed by a *switch* statement on the returned *Types* value

DisplayAuthors: Output

```
avraham@leff-3:AuthorsAndBooks$ mvn -q exec:java
Authors Table of Books Database:

AUTHORID FIRSTNAME LASTNAME
1        Paul      Deitel
2        Harvey    Deitel
3        Abbey     Deitel
4        Dan       Quirk
5        Michael   Morgano
```

Putting It All Together

```
import java.sql.Connection;
import java.sql.Statement;
import java.sql.DriverManager;
import java.sql.ResultSet;
import java.sql.ResultSetMetaData;
import java.sql.SQLException;
public class DisplayAuthors {
  public static void main(String args[]) {
    final String DATABASE_URL = "jdbc:derby:books";
    final String SELECT_QUERY =
      "SELECT authorID, firstName, lastName FROM authors";
    try (
      Connection connection = DriverManager.getConnection(
        DATABASE_URL, "dbimpl_user", "dbimpl_password");
      Statement statement = connection.createStatement();
      ResultSet resultSet = statement.executeQuery(SELECT_QUERY)) {
      ResultSetMetaData metaData = resultSet.getMetaData();
      int numberOfColumns = metaData.getColumnCount();
      System.out.printf("Authors Table of Books Database:%n%n");
      for (int i = 1; i <= numberOfColumns; i++) {
        System.out.printf("%-8s\t", metaData.getColumnName(i));
      }
      System.out.println();
      while (resultSet.next()) {
        for (int i = 1; i <= numberOfColumns; i++) {
          System.out.printf("%-8s\t", resultSet.getObject(i));
        }
        System.out.println();
      }
    }
    catch (SQLException sqlException) {
      sqlException.printStackTrace();
    }
  }
}
```

Important JDBC Tips: I

- ResultSet columns are numbered 1..n
- Two ways to access a given attribute
  - By number: e.g., `int getInt(int columnIndex)`
  - By name: e.g., `int getInt(String columnLabel)`
- Q: which do you think is the better approach?
- A_1: if you specified the columns in the SELECT, "by number" is more efficient
- A_2: if the system supplied the columns via SELECT *, "by name" is far more robust
- A_3: I would use "by name" in all cases ☺

Important JDBC Tips: II

- A *ResultSet* instance is bound to the *Statement* that created it
  - Accessing the ResultSet after the Statement is closed triggers an *SQLException*
- A given *Statement* instance can only "keep open" a single *ResultSet* instance
- When a Statement returns a new ResultSet instance, the previous ResultSet bound to that Statement is implicitly closed
- If you must process multiple ResultSets in parallel, you'll have to use multiple Statement instances

createStatement: More Detail

Javadoc for the no-arg constructor states:

> Result sets created using the returned Statement object will by default be type TYPE_FORWARD_ONLY and have a concurrency level of CONCUR_READ_ONLY. The holdability of the created result sets can be determined by calling getHoldability().

What are they talking about?

- Note: A two-argument constructor exists that allows you to
  1. Override the default TYPE_FORWARD_ONLY specification
  2. Override the default CONCUR_READ_ONLY specification
- Note: A three-argument constructor exists that allows you to override the default values for the above parameters and the DBMS supplied, third "ResultSet holdability" value
  - You specify either ResultSet.HOLD_CURSORS_OVER_COMMIT or ResultSet.CLOSE_CURSORS_AT_COMMIT

Default ResultSet Behavior: Scroll Types

- By default, a ResultSet has a cursor that can only move forward
- That is: iteration proceeds from the first to the last tuple in the result-set
- You can override this behavior to specify
  - TYPE_SCROLL_SENSITIVE: can scroll in either direction and changes made to the underlying data will be visible to the client
  - TYPE_SCROLL_INSENSITIVE: can scroll in either direction and changes made to the underlying data will not be visible to the client
- As you can see, both of these override specifications enable bi-directional scrolling (and moving to an absolute position)
- But what does behavior have to do with a ResultSet being “sensitive” ☺?

Default ResultSet Behavior: Sensitive Scroll Types

- Remember: other clients may be using your database at the same time that you're accessing the database
- The first two scroll types produce ResultSets that are insensitive to changes made to the underlying data source while it is open
  - So: concurrent changes made to rows in your *ResultSet* will not be reflected to you as you process the result-set
- A "SENSITIVE" scroll type provides a "live view" of the data
  - Concurrent changes made to the data can therefore affect the values you see in your ResultSet

Default ResultSet Behavior: ResultSet Concurrency

- Scroll type "sensitivity" determines whether you'll see the database changes made by others
- The ResultSet "concurrency" specifications determine whether you'll be able to update the database using the ResultSet APIs
- Two concurrency levels
  - CONCUR_READ_ONLY: ResultSet cannot be used to update the database state
  - CONCUR_UPDATABLE: ResultSet object can be used to update the database state

```
1  rs.updateRow(); // updates the row in the data source
2  rs.moveToInsertRow(); // moves cursor to the insert row
3  // updates the first column of the insert row
4  // to be AINSWORTH
5  rs.updateString(1, "AINSWORTH");
6  rs.updateInt(2,35); // updates second column to be 35
7  rs.updateBoolean(3, true); // updates third column to true
8  rs.insertRow();
9  rs.moveToCurrentRow()
```

“Insert Row” *Versus* “Current Row” (I)

- In the previous slide, the code inserts a row without doing an SQL UPDATE!
- Q: how can that possibly work?
- A: it’s all about the cursor ☺
- Using the JDBC v2 API, you can both build a new row and insert it into the result-set and the “server-side” table in one step
- You use the “insert row buffer” that’s associated with every *ResultSet*
  - You “compose” the new row in that buffer
- Then:
  1. Move the cursor to the “insert row” by invoking `moveToInsertRow`
  2. Set values for all columns in that row by invoking `updateXXX`
  3. Invoke `insertRow`

“Insert Row” *Versus* “Current Row” (I)

- When you invoke `moveToInsertRow`, the *ResultSet* records the position of the cursor (the “current row”)
- After performing an `insertRow`, you can invoke `moveToCurrentRow` to get back to what was previously the “current row”
- The API includes corresponding CRUD methods such as `updateRow` and `deleteRow`
  - These are applied to the “current row”
  - Both apply to the *ResultSet* contents and the underlying database
- Many other “fine-grained” cursor control methods such as `first`, `beforeFirst`, `last`, `afterLast`, and `absolute`

Default ResultSet Behavior: Holdability

- This feature doesn't seem to be used much
- It refers to the issue of whether committing a transaction (via `Connection.commit`) should close (invalidate) the ResultSet (or cursor)
- Two values possible
  - HOLD_CURSORS_OVER_COMMIT: ResultSets are not closed on commit
    - We say that they are holdable
    - Motivation: you keep a result set available across transaction boundaries for use by multiple JDBC calls
  - CLOSE_CURSORS_AT_COMMIT: ResultSets are closed on transaction commit
    - Motivation: may improve (back-end) performance since the database can immediately reuse the state formerly associated with your ResultSet

DBMS Must Support Your Override

- Some JDBC drivers do not support scrollable ResultSets
- You're specifying an overide, but the DBMS can't meet your demands
  - You'll get an *SQLFeatureNotSupportedException* when you issue a `createStatement`
  - If you try to scroll backwards on a ResultSet from such a database, you'll get an *SQLFeatureNotSupportedException* then ☹
- Same semantics for "updateable" ResultSets
- Same semantics for "holdable" ResultSets

PreparedStatements

- Previous discussion all revolve around the *Statement* API
- Advantage: create a String that is exactly the SQL that you want the DBMS to execute
- Disadvantages
  - Performance
  - Vulnerable to SQL injection attacks
- These disadvantages are so large that aside from "toy examples" (or "initial exploration"), you should never, ever use *Statement*
  - Instead: use (the sub-classed) PreparedStatement

Performance Cost

The server must (conceptually) perform the following four processing steps for every incoming SQL statement client request

1. Parse the incoming SQL statement
2. Compile the statement
3. Construct an execution plan (more later in the semester) and possibly invoke the optimizer to refine the data access path
4. Execute the optimized statement, construct a response, and return data

- There is no way around this: you must pay that processing time at the server
- But not always ☺
- A DBMS will cache the optimized, executable, statement
  - But only helpful if the same statement is executed multiple times!

Implications of Statement Performance Cost: I

- Key difference between *Statement* and *PreparedStatement* is that the latter can be parameterized
- Compare this code fragment ...

```
1  for(int i = 0; i < 1000; ++i) {
2    PreparedStatement ps =
3      conn.prepareStatement("select a,b from t where c = " + i
           );
4    ResultSet rs = ps.executeQuery();
5  }
```

- The DBMS cannot use its cache for the previous code fragment!
  - Each statement is a different SQL statement and needs a different access plan

Implications of Statement Performance Cost: II

Compare previous code to this code fragment

```
1  PreparedStatement ps =
2    conn.prepareStatement("select a,b from t where c = ?");
3  for(int i = 0; i < 1000; ++i) {
4    ps.setInt(1, i);
5    ResultSet rs = ps.executeQuery();
6  }
```

- Even though they are functionally different statements
- From the DBMS perspective, they are identical
  - Because of the parameters from the PreparedStatement cache maintained by the DBMS
- Performance win! because DBMS can omit the intermediate processing steps

PreparedStatement: ParameterBinding

```
1  String stmt = "INSERT INTO Person " +
2    "(name, email, birthdate, photo) VALUES (?, ?, ?, ?)");
3  ps = connection.prepareStatement(stmt);
4  ps.setString(1, person.getName());
5  ps.setString(2, person.getEmail());
6  ps.setTimestamp
7    (3, new Timestamp(person.getBirthdate().getTime()));
8  ps.setBinaryStream(4, person.getPhoto());
9  ps.executeUpdate();
```

- You no longer in-line the values via string concatenation
- Instead, bind the parameter value using the appropriate "setter" from the suite of `setXXX` methods
- Or: use `setObject` for all parameters

Parameter Binding: Other Advantages

- Code is cleaner
  - Query code is explicitly separated from query parameters
- Use `getParameterMetaData()` to "retrieve the number, types and properties of this PreparedStatement object's parameters"
- Use `getMetaData()` to "retrieves a ResultSetMetaData object that contains information about the columns of the ResultSet object that will be returned when this PreparedStatement object is executed"
- Use `setArray`, `setAsciiStream` etc to set non-primitive values
  - Not everything has a usable `toString` method ☺

PreparedStatement: Block SQL Injection Attacks

- Typically, SQL statement incorporates "state" derived from end-user
  - Example: "lookup up username"
  - Must get a "username" from an end-user
- *Statement* approach requires that these parameters be concatenated in the SQL
- This is fine for normal data from non-malicious users
- But data is sometimes not normal
  - Includes apostrophes
- And users are sometimes malicious ☹
- SQL injection: when naive concatenation creates an SQL statement that harms the DBMS

SQL Injection: Trivial Example

- Consider an application that interacts with users with an HTML form
  - With username and password
- Not hard to guess (correctly) that the form submission will do a POST that takes the values of the form fields and builds the following SQL query

```
String query = "SELECT * FROM users  WHERE userid ='" +
    userid
   + "'" + " AND password='" + password + "'";
```

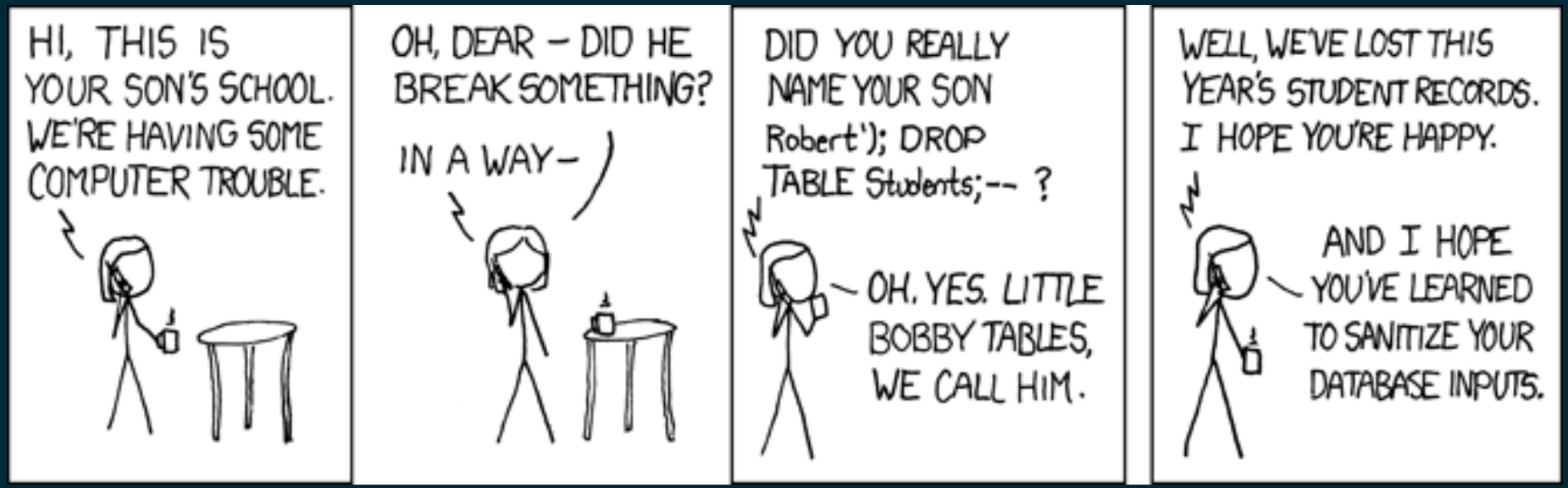
- So far, no problem ...
- But: malicious user supplies **avraham' /*** as the userid
- Result: you've just stolen all of that user's personal data ☹

```
    select * from users
    where username = 'avraham' /*'and password = '...'";
```

Sanitizing

- The process of dealing with “malformed” input (whether malicious or not) is called sanitization
- For example: “escape” any apostrophes that may be in a username input
- You – the JDBC client – can do the sanitization
  - But you’ll probably do a bad job ☺
- It’s really a job that should be done by the JDBC Driver implementation
- For example: “standard” SQL requires that String literals be demarcated with apostrophes
  - But MySQL allows String literals to be demarcated by double-quotes
- Key point: *PreparedStatement* binding allows the JDBC Driver to do the work correctly

SQL Injection Attacks in a Nutshell



Source: https://xkcd.com/327/

Understanding the Exploit: What Happened (I)

The student, nicknamed Little Bobby Tables ☺, is officially named

> **Robert'); DROP TABLE students;–**

```
database.execute
  ("INSERT INTO students (name) VALUES ('" + name + "');");
```

```
-- normal usage
INSERT INTO students (name) VALUES ('Elaine');

-- exploit
INSERT INTO students (name)
VALUES ('Robert'); DROP TABLE students;--');
```

Understanding the Exploit: What Happened (II)

- In SQL
  - Commands are terminated by semicolons, String data are quoted
  - The quotation mark in Bobby's name is erroneously parsed as a closing quote inside that statement, rather than being processed as part of the name
  - The start of a comment is represented by --
- The dashes in Bobby's name ensures that when the DBMS appends the rest of its string to the input
  - (Which would normally cause an error and disable the exploit ...)
  - The erroneous part is now ignored, parse error does not occur
- Result: the harmless insert ⇒ deleted the entire table

Stored Procedures

- Today's discussion: JDBC implementation of the "dynamic SQL" concept
- Next lecture: the server-side alternative to host language database programming: stored procedures & functions
- Q: how do these constructs fit into the JDBC API?
- A: JDBC has a *CallableStatement* sub-class that can invoke these server-side stored procedures
- The API is more complicated than the *Statement* and *PreparedStatement* APIs
  - Have to use `registerOutParameter` so that the stored procedure can return values
  - And retrieved via `getDataType` after the statement executes
- Even though the server-side syntax for creating a stored procedure is highly DBMS-specific ...
  - By using a *CallableStatement*, we get a uniform API for specifying the input & output parameters and for invoking the stored procedure itself

Transaction Processing

- You may be wondering how JDBC provides a transaction API
- Scenario:
  - You create a number of books, create a number of books, then the program crashes in the middle of inserting tuples into the "AuthorsISBN" table
    - May even be your fault: you used wrong syntax ☹
- *"Help, we need transactions"* (e.g., to commit every 500 tuples)
- Q: where are our transactions?
  - Note: remember to ask this question when we discuss server-side programming ☺
- A: *Connection* has `commit` and `rollback` methods
- Also has a `setAutoCommit` method which specifies whether each *Statement* executes in its own implicit transaction

java.sql Versus javax.sql

- We've been using classes from the java.sql package
- Historically, after this package was released, new database function ("JDBC extensions") was released in the javax.sql package
  - With the latter only being available as part of the Java EE (in contrast to Java SE) distribution
- Forget about all that ☺
  - Both packages are now part of Java SE, equally available
  - javax has more sophisticated function than java
  - Worthwhile to see what's available
- Will give one example

RowSet Interface

- The javax.sql.RowSet interface combines
  - Database configuration
  - Statement preparation
  - Iteration over the result set
- Into one interface

```
try (JdbcRowSet rowSet =
      RowSetProvider.newFactory().createJdbcRowSet()) {

   rowSet.setUrl(DATABASE_URL);
   rowSet.setUsername(USERNAME);
   rowSet.setPassword(PASSWORD);
   rowSet.setCommand("SELECT * FROM Authors"); // set query
   rowSet.execute(); // execute query

   // process query results
   ResultSetMetaData metaData = rowSet.getMetaData();
   int numberOfColumns = metaData.getColumnCount();
   // use while-loop over rowSet to iterate over tuples
   // just as before
}
```

RowSets: Some Commentary

I never like the RowSet or updatable ResultSet. Those seem to violate the Spirit of Codd (peace be upon him). It reinforces the "wrong" view that the order of tuples in the DB is important, and breaks the separation of `executeQuery()` *and* `executeUpdate()`

Source: a former colleague

When & Whether To Use JDBC?

Segue

- I hope that this lecture has convinced you that JDBC is a very powerful tool for interacting with relational databases
- We're going to discuss now when you should go for a JDBC solution …
- Or: should you even use JDBC at all
- Q: JDBC is so great, why not use it?
- A: because you shouldn't extrapolate from "toy" scenarios to general "database programming"
- Alert: highly opinionated comments to follow ☺
- JDBC is a great tool for querying databases (once you've properly sanitized inputs)
- JDBC is an OK tool for CUD statements when they involve only a single table!
  - Not "great", because (1) you've exposed database structure and (2) application programmers don't really know SQL ☺

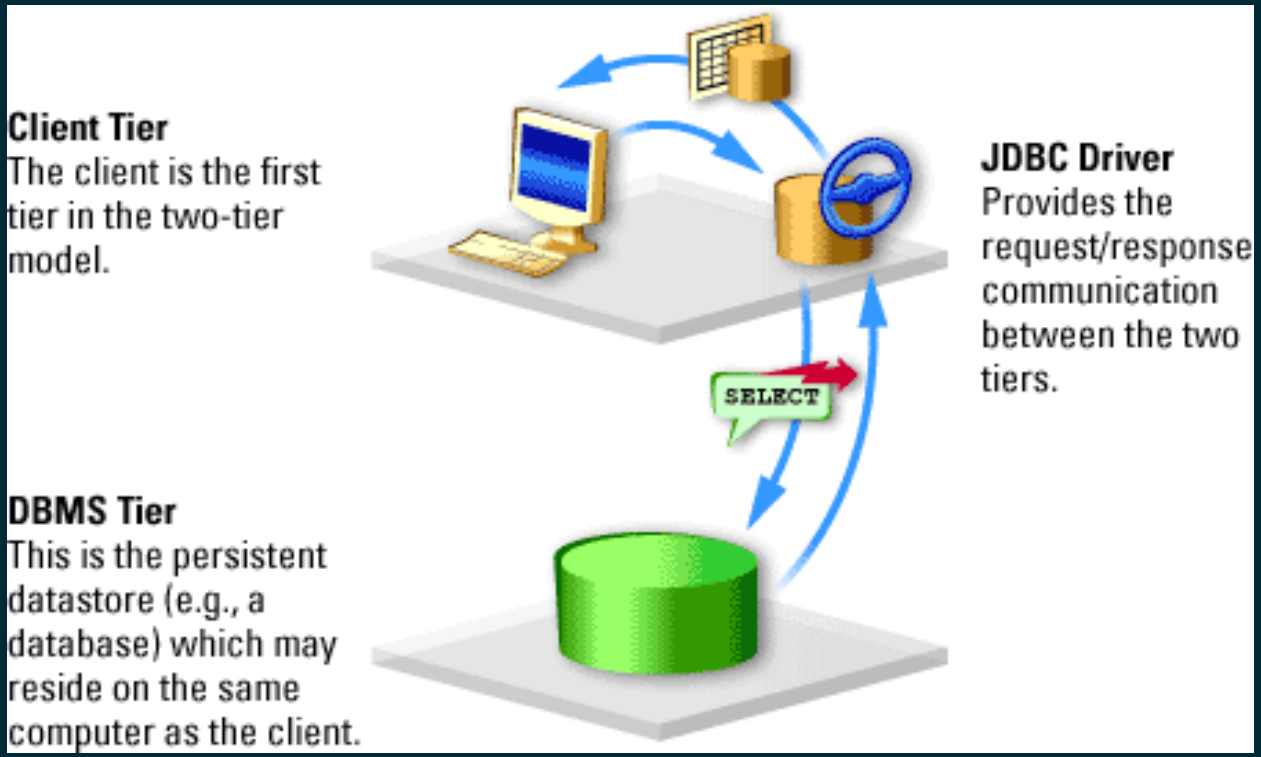
JDBC & Multiple Tables

- Easiest way to think about this is in terms of our discussion of database views
- Views are a great idea, so long as they're applied to a single table
- But: we showed that it can be impossible for view code to "reverse engineer" the join(s) that created the view so as to apply CUD operations ☹
- Similar problem with JDBC: the programmer thinks she understands how to properly propagate CUD operation logic to the database
- She may even be correct at the time that she wrote the code!
  - What happens as the database schema evolve?
  - Or: integrity constraints are modified and added?
- Massive violation of the DRY principle!

Security, Encapsulation, & Web-Architectures

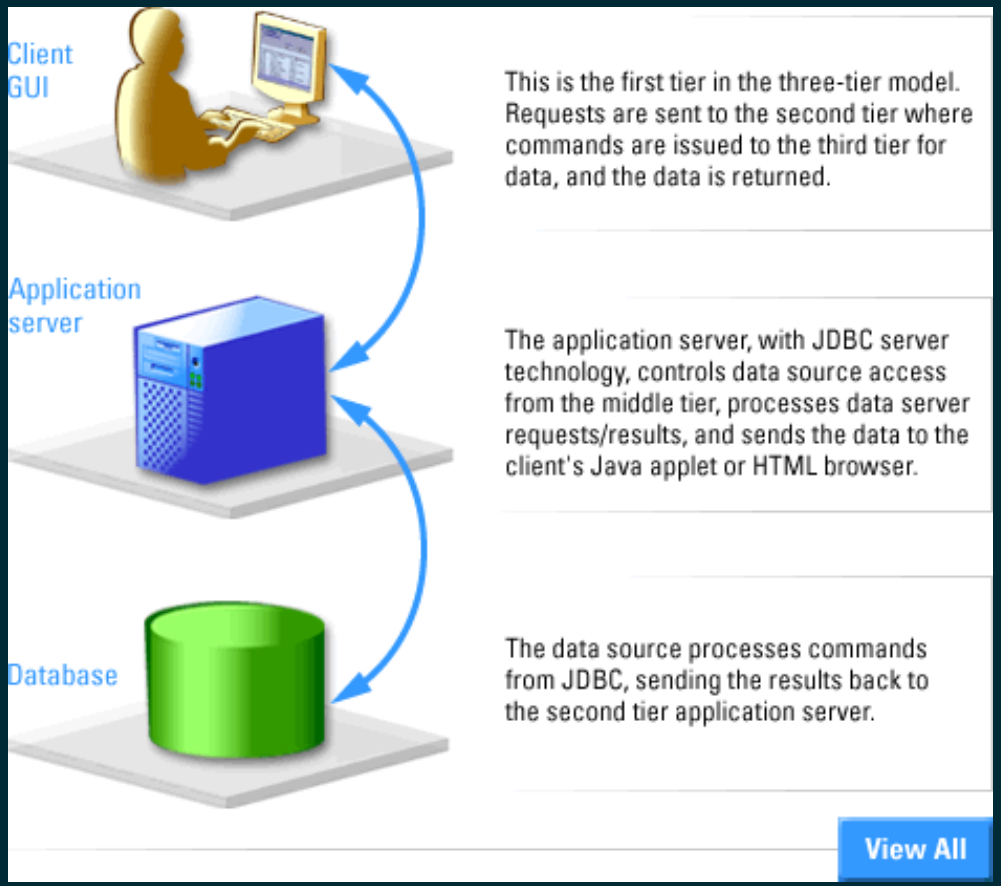
- Another version of this JDBC problem is that (by definition), an enterprise has to expose its business logic outside the database
- Also: you're implicitly granting a larger set of users with the permissions to modify the database
  - How do we verify that a `drop table students` statement isn't buried somewhere in the code?
- OTOH, we absolutely must have code that mediates between end-user (think "web-forms") input and the database ...
- One way to satisfy these contradictory requirements is to deploy a three-tier architecture
- Quick review on next slides ...
  - Note: usually, people tout "performance" (scalability) as the key benefit of multi-tiered architectures
  - Here, I'm arguing for the advantage of security & encapsulation

Application Architectures: Two-Tier



- In this course, we'll be using a two-tier architecture
  1. Your Java client program (using JDBC APIs) interacts with the database tier
  2. Database tier stores the data used by the first tier
- Doesn't matter if the tiers reside on the same computer (e.g., your laptop)
  - Still a two-tier architecture

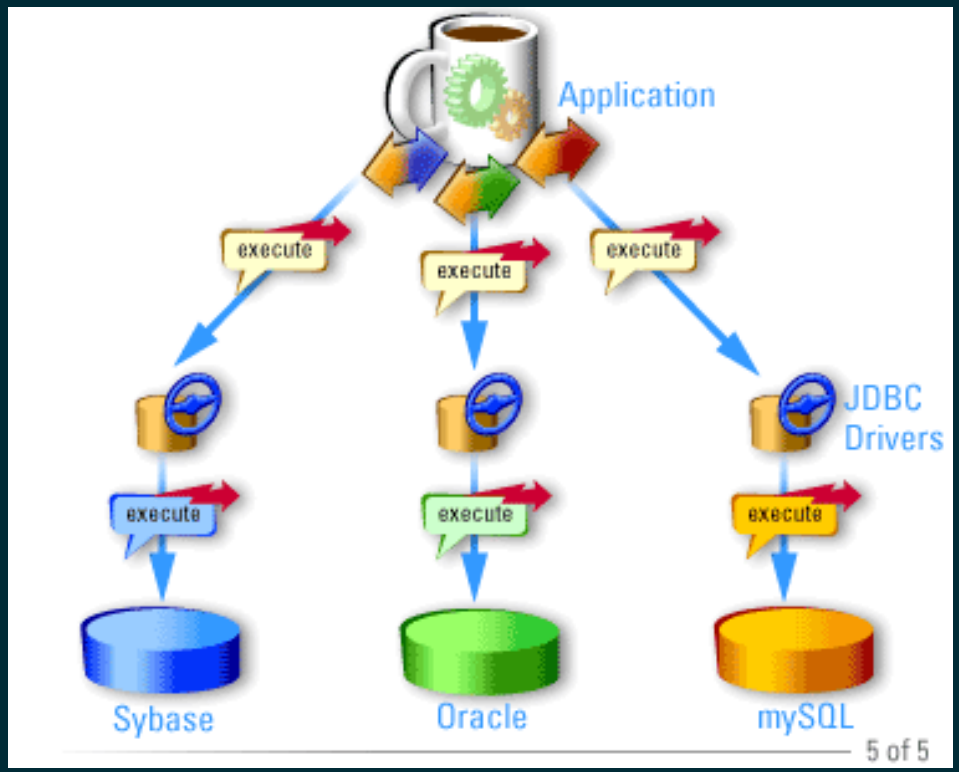
Application Architectures: Three-Tier



- As enterprises grow, they often move to a three-tier architecture
  1. Your Java client program interacts with an application server using application-specific APIs
     - Not JDBC
  2. Application server uses JDBC to talk to the database tier
  3. Database tier stores the data used by the first tier

Reminder: JDBC "Standard" Is No Panacea

- By using JDBC APIs you're shielded from (almost all) the system-specific client issues
  - Connections, cursor manipulation, error codes
- Warning: does not shield you from the database-specific SQL issues!
  - All those "vendor proprietary" disclaimers from previous lectures ☹



Today's Lecture: Wrapping it Up

Database Programming: Review & Some Context

Using JDBC APIs

When & Whether To Use JDBC?

Readings

- Textbook discusses JDBC in Chapter 5.1, somewhat limited IMNSHO