SQL Programming

- SQL in Application Programs
- JDBC: Java Database Connectivity
- CLI: Call-Level Interface
- Embedded SQL

SQL in Applications Programs

- We have seen how SQL is used at the generic query interface --- an environment where we sit at a terminal and ask queries of a database.
- Reality is almost always different: conventional programs interacting with SQL.
- Want to consider:
  - How do we enable a database to interact with an “ordinary” program written in a language such as C or Java?
  - How do we deal with the differences in data types supported by SQL and conventional languages?
    - In particular, relations, which are the result of queries, are not directly supported by conventional languages.

Three-Tier Architecture

- A common environment for using a database has three tiers of processors:
  1. *Web servers* --- Connect users to the database, usually over the Internet, or possibly a local connection.
  2. *Application servers* --- Execute the “business logic” – whatever it is the system is supposed to do.
  3. *Database servers* --- Run the DBMS and execute queries and modifications at the request of the application servers.

Example: Amazon

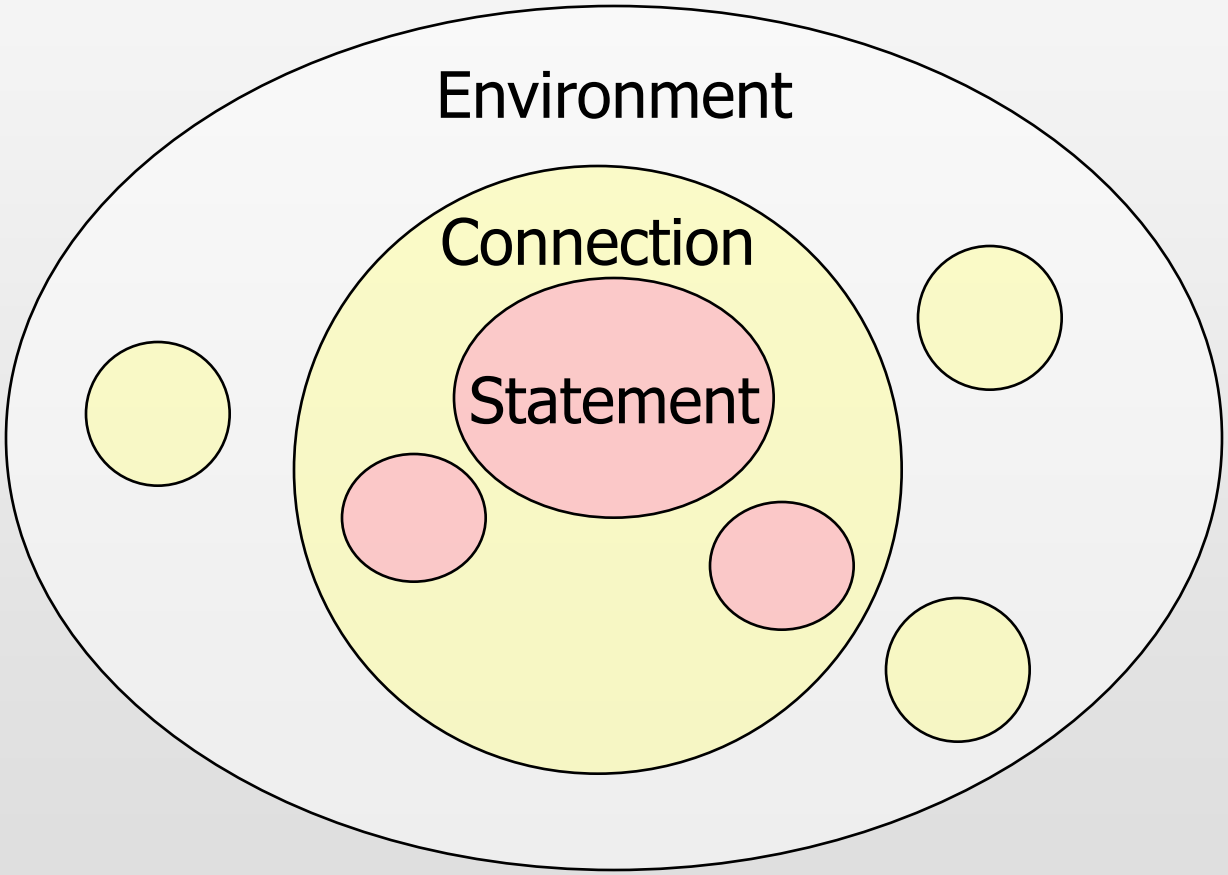
- Database holds the information about products, customers, etc.
- Business logic includes things like “what do I do after someone clicks ‘checkout’?”
  - Answer: Show the “How will you pay for this?” screen.

For this section, we will deal with the interaction between the application and the DBMS

Environments, Connections, Queries

- A SQP environment is the framework under which data exists and SQL operations are executed.
- Think of a SQL environment as a DBMS running at some installation.
  - So tables, triggers, views, etc are defined within a SQL environment
- Database servers maintain some number of *connections*, so app servers can ask queries or perform modifications.
- The app server issues *statements*: queries and modifications, usually.

Diagramatically



The SQL/Host Language Interface: Options

1. Code in a specialized language is stored in the database itself (e.g., PSM, PL/SQL).
   - Not covered (see text for info)
2. Connection tools are used to allow a conventional language to access a database (e.g. JDBC, CLI, PHP/DB).
3. SQL statements are embedded in a *host language* (e.g., C).

The Impedance Mismatch Problem

Basic problem: impedance mismatch – the data model of SQL differs significantly from the models of other languages.

- SQL uses the relational data model
- C, Java, etc., use a data model with ints, reals, pointers, records, etc.
- Consequently, passing data between SQL and a host language is not straightforward.

Host/SQL Interfaces Via Libraries

- The first approach to connecting databases to conventional languages is to use library calls.
  - Java + JDBC
  - C + CLI

SQL Programming: JDBC

- *Java Database Connectivity* (JDBC) is a library similar to SQL/CLI, but with Java as the host language.
- JDBC supports
  - Establishing a connection
  - Creating JDBC statements
  - Executing SQL statements
  - Getting a ResultSet
  - Closing connection

Making a Connection

Three initial steps:

1. Include

   import java.sql.*;

   to make the JAVA classes available.

2. Load a (vendor specific) “driver” for the database system being used.

   Class.forName(“com.microsoft.sqlserver.jdbc.SQLServerDriver”);

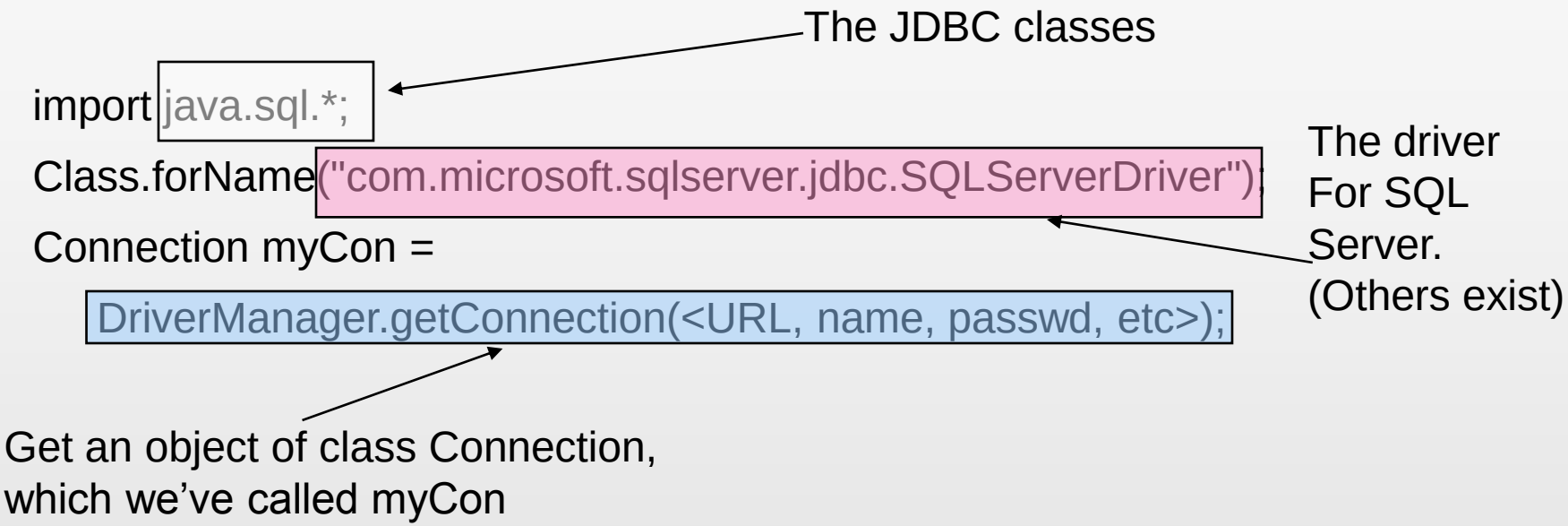
   dynamically loads a driver class for SQL Server db.

3. Establish a connection to the database.

   Connection con = DriverManager.getConnection(“jdbc:mysql://
   localhost/Food? User=UserName&password=Password”);

   establishes connection to database (Food) by obtaining a *Connection* object.

Making a Connection



Statements

- JDBC provides two classes:

1. *Statement* = an object that can accept a string that is a SQL statement and can execute the string.
2. *PreparedStatement* = an object that has an associated SQL statement ready to execute.

- Created by methods createStatement() (or prepareStatement(Q) for prepared statements).

Creating Statements

- The Connection class has methods to create Statements and PreparedStatements.

```
Statement stat1 = myCon.createStatement();

PreparedStatement stat2 =
    myCon.createStatement(
        ”SELECT beer, price FROM Sells ” +
        ”WHERE bar = ’Joe’ ’s Bar’ ”
    );
```

createStatement with no argument returns a Statement; with one argument it returns a PreparedStatement.

Executing SQL Statements

- JDBC distinguishes queries from modifications, all of which it calls “updates.”
- Statement and PreparedStatement each have methods executeQuery and executeUpdate.
  - For Statements: one argument, consisting of the query or modification to be executed.
  - For PreparedStatements: no argument, since a prepared statement already has an associated object.

The 4 “Execute” Methods

- executeQuery(Q) takes a statement Q, which must be a query, that is applied to a Statement object. Returns a ResultSet object, the multiset of tuples produced by Q.
- executeQuery() is applied to a PreparedStatement object.
- executeUpdate(U). Takes a database modification U, and when applied to a Statement object, executes U. No ResultSet is returned
- executeUpdate() is applied to a PreparedStatement object.

Example: Update

- stat1 is a Statement.
- We can use it to insert a tuple as:

```
stat1.executeUpdate(
    ”INSERT INTO Sells ” +
    ”VALUES(’Brass Rail’,’Export’,3.00)”
);
```

Example: Query

- stat2 is a PreparedStatement holding the query ”SELECT beer, price FROM Sells WHERE bar = ’Joe’’s Bar’ ”.
- executeQuery returns an object of class ResultSet (next slide)
- The query:

```
ResultSet menu = stat2.executeQuery();
```

Accessing the ResultSet

- An object of type *ResultSet* is something like a *cursor* (which we'll see later).
- Aside: A cursor is essentially a tuple-variable that ranges over all tuples in the result of some query.
  - Using a cursor lets one successively iterate through tuples satisfying a query.
- Method next() advances the "cursor" to the next tuple.
  - The first time next() is applied, it gets the first tuple.
  - If there are no more tuples, next() returns the value false.

Accessing Components of Tuples

- When a ResultSet is referring to a tuple, we can get the components of that tuple by applying certain methods to the ResultSet.
- Method get*X* (*i*), where *X* is some type, and *i* is the component number, returns the value of that component.
  - Examples: getString(i), getInt(i), getFloat(i), etc.
  - The value must have type *X*.

Example: Accessing Components

- menu = ResultSet for query “SELECT beer, price
  FROM Sells
  WHERE bar = ’Joe’’s Bar’ ”.
- Access beer and price from each tuple by:

```
while ( menu.next() ) {
    theBeer = menu.getString(1);
    thePrice = menu.getFloat(2);
        /* do something with theBeer and thePrice */
}
```

Accessing Components of Tuples (Method 2)

- Method get*X* (ColumnName), where *X* is some type, and *ColumnName* is the component number, returns the value of that component.
  - The value must have type *X*.

Example: Accessing Components

- menu = ResultSet for query “SELECT beer, price
  FROM Sells
  WHERE bar = ’Joe’’s Bar’ ”.
- Access beer and price from each tuple by:

```
while ( menu.next() ) {
    theBeer = Menu.getString(“beer”);
    thePrice = Menu.getFloat(“price”);
        /* do something with theBeer and thePrice */
}
```

SQL Programming: SQL/CLI

- SQL/CLI is a library which provides access to DBMS for C programs.
  - The library for C is called SQL/CLI = "*Call-Level Interface*."
  - The concepts here are similar to JDBC.

Data Structures

- C connects to the database by records (structs) of the following types:
  1. *Environments* : represent the DBMS installation.
  2. *Connections* : logins to the database.
  3. *Statements* : SQL statements to be passed to a connection.
  4. *Descriptions* : records about tuples from a query, or parameters of a statement.
     - Will ignore here.
- Each of these records is represented by a handle, or pointer to the record.
- The header file sqlcli.h provides types for the handles of environments, etc.

Handles

- Function SQLAllocHandle(T,I,O) is used to create these structs, which are called environment, connection, and statement handles.
  - *T* = type, e.g., SQL_HANDLE_STMT. Also SQL_HANDLE_ENV, SQL_HANDLE_DBC
  - *I* = input handle
    = struct at next higher level (statement < connection < environment).
    SQL_NULL_HANDLE for an environment
    E.g. if you want a statement handle, then I is the handle of the "host" connection
  - *O* = (address of) output handle (created by SQLAllocHandle)

Example: SQLAllocHandle

SQLAllocHandle(SQL_HANDLE_STMT, myCon, &myStat);

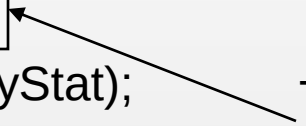
- myCon is a previously created connection handle.
- myStat is the name of the statement handle that will be created.

Preparing and Executing

- SQLPrepare(H, S, L) causes the string *S*, of length *L*, to be interpreted as a SQL statement and optimized; the executable statement is placed in statement handle *H*.
- SQLExecute(H) causes the SQL statement represented by statement handle *H* to be executed.

Example: Prepare and Execute

```
SQLPrepare(myStat, ”SELECT beer, price FROM Sells
    WHERE bar = ’Joe’’s Bar’”,
    SQL_NTS);
SQLExecute(myStat);
```



This constant says the second argument is a “null-terminated string”; i.e., figure out the length by counting characters.

Direct Execution

- If we are going to execute a statement *S* only once, we can combine PREPARE and EXECUTE with:

  SQLExecuteDirect(H,S,L);

  - As before, *H* is a statement handle and *L* is the length of string *S*.

Fetching Tuples

- When the SQL statement executed is a query, we need to fetch the tuples of the result.
  - A cursor is implied by the fact we executed a query; the cursor need not be declared.
- SQLFetch(H) gets the next tuple from the result of the statement with handle *H*.

Accessing Query Results

- When we fetch a tuple, we need to put the components somewhere.
- Each component is bound to a variable by the function SQLBindCol.
  - This function has 6 arguments, of which we shall show only 1, 2, and 4:

    1 = handle of the query statement.

    2 = column number.

    4 = address of the variable.

Example: Binding

- Suppose we have just done SQLExecute(myStat), where myStat is the handle for query

  SELECT beer, price

  FROM Sells

  WHERE bar = ’Joe’’s Bar’

- Bind the result to theBeer and thePrice:

  SQLBindCol(myStat, 1, , &theBeer, ,);

  SQLBindCol(myStat, 2, , &thePrice, ,);

Example: Fetching

- Now, we can fetch all the tuples of the answer by:

```
while ( SQLFetch(myStat) != SQL_NO_DATA)
    {
        /* do something with theBeer and
           thePrice */
    }
```

CLI macro representing SQLSTATE = 02000 = “failed to find a tuple.”

SQL Programming: Embedded SQL

- Key idea: A preprocessor turns SQL statements into procedure calls that fit with the surrounding host-language code.
- All embedded SQL statements begin with EXEC SQL, so the preprocessor can find them easily.

Shared Variables

- To connect SQL and the host-language program, the two parts must share some variables.
- Declarations of shared variables are bracketed by:

```
EXEC SQL BEGIN DECLARE SECTION;
        <host-language declarations>
EXEC SQL END DECLARE SECTION;
```

Use of Shared Variables

- In SQL, the shared variables must be preceded by a colon.
  - They may be used as constants provided by the host-language program.
  - They may get values from SQL statements and pass those values to the host-language program.
- In the host language, shared variables behave like any other variable.
  - Not preceded by a colon here.

Example: Looking Up Prices

- We'll use C with embedded SQL to sketch the important parts of a function that given a beer and a bar, looks up the price of that beer at that bar.
  - Note that a query here returns a single value (tuple)
- Assumes database has our usual Sells(bar, beer, price) relation.

Example: C Plus SQL

Single row SELECT Statements:

```
EXEC SQL BEGIN DECLARE SECTION;
    char theBar[21], theBeer[21];
    float thePrice;
EXEC SQL END DECLARE SECTION;
    /* obtain values for theBar and theBeer */
EXEC SQL SELECT price INTO :thePrice
         FROM Sells
         WHERE bar = :theBar AND beer = :theBeer;
    /* do something with thePrice */
```

21-char arrays needed for 20 chars + endmarker

SELECT-INTO: used for a single row

Embedded Queries

- Embedded SQL (so far) has a limitation regarding queries:
  - SELECT-INTO for a query guaranteed to produce a single tuple.
  - Otherwise, you have to use a cursor.
- Recall: A cursor is essentially a tuple-variable that ranges over all tuples in the result of some query.
- Using a cursor lets one iterate through tuples satisfying a query.

Cursor Statements

- Declare a cursor *c* with:

  EXEC SQL DECLARE *c* CURSOR FOR <query>;

- Open and close cursor c with:

  EXEC SQL OPEN CURSOR c;

  EXEC SQL CLOSE CURSOR c;

  - The OPEN statement causes the query to be evaluated.
  - The CLOSE statement causes the database to delete the temporary relation that holds the result of the query.

- Fetch from *c* by:

  EXEC SQL FETCH c INTO <variable(s)>;

  - Repeated calls to FETCH get successive tuples in the query result.
  - Macro NOT FOUND is true if and only if the FETCH fails to find a tuple.

Example Cursor

- From within a host language, want to find the names and cities of customers with more than the variable amount dollars in some account.

- Specify the query in SQL and declare a *cursor* for it

```
EXEC SQL
  DECLARE c CURSOR FOR
  SELECT depositor.customer_name, customer_city
  FROM depositor, customer, account
  WHERE depositor.customer_name = customer.customer_name
      AND depositor account_number = account.account_number
      AND account.balance > :amount ;
```

Updates Through Cursors

- Can update tuples fetched by cursor by declaring that the cursor is for update

```
DECLARE c CURSOR FOR
   SELECT *
   FROM account
   WHERE branch_name = 'Perryridge'
FOR UPDATE
```

- To update tuple at the current location of cursor *c*

```
UPDATE account
SET balance = balance + 100
WHERE CURRENT OF c
```

Example: Print Joe's Menu

- Write C + SQL to print Joe's menu – the list of beer-price pairs that we find in Sells(bar, beer, price) with bar = Joe's Bar.
- A cursor will visit each Sells tuple that has bar = Joe's Bar.

Example: Declarations

```
EXEC SQL BEGIN DECLARE SECTION;
    char theBeer[21]; float thePrice;
EXEC SQL END DECLARE SECTION;
EXEC SQL DECLARE c CURSOR FOR
    SELECT beer, price FROM Sells
    WHERE bar = 'Joe''s Bar';
```

The cursor declaration goes outside the declare-section

Example: Executable Part

```
EXEC SQL OPEN CURSOR c;
while(1) {
    EXEC SQL FETCH c
                    INTO :theBeer, :thePrice;
    if (NOT FOUND) break;
    /* format and print theBeer and thePrice */
}
EXEC SQL CLOSE CURSOR c;
```

Need for Dynamic SQL

- Most applications use specific queries and modification statements to interact with the database.
  - The DBMS compiles EXEC SQL … statements into specific procedure calls and produces an ordinary host-language program that uses a library.
- Dynamic SQL allows programs to construct and submit SQL queries at run time.

Dynamic SQL

- Preparing a query:

  EXEC SQL PREPARE <query-name>

  FROM <text of the query>;

- Executing a query:

  EXEC SQL EXECUTE <query-name>;

- “Prepare” = optimize query.
- Prepare once, execute many times.

Example: A Generic Interface

```
EXEC SQL BEGIN DECLARE SECTION;
    char query[MAX_LENGTH];
EXEC SQL END DECLARE SECTION;

while(1) {
    /* issue SQL> prompt */
    /* read user's query into array query */
    EXEC SQL PREPARE q FROM :query;
    EXEC SQL EXECUTE q;
}
```

q is an SQL variable representing the optimized form of whatever statement is typed into :query

Execute-Immediate

- If we are only going to execute the query once, we can combine the PREPARE and EXECUTE steps into one.
- Use:

```
EXEC SQL EXECUTE IMMEDIATE <text>;
```

Example: Generic Interface Again

```
EXEC SQL BEGIN DECLARE SECTION;
    char query[MAX_LENGTH];
EXEC SQL END DECLARE SECTION;

while(1) {
    /* issue SQL> prompt */
    /* read user's query into array query */
    EXEC SQL EXECUTE IMMEDIATE :query;
}
```

Another Example

Example of the use of dynamic SQL from within a C program.

```
CHAR   *sqlprog = "UPDATE account
                   SET balance = balance * 1.05
                   WHERE account_number = ?"
EXEC SQL PREPARE dynprog FROM :sqlprog;
CHAR account[10] = "A-101";

EXEC SQL EXECUTE dynprog USING :account;
```

- The dynamic SQL program contains a ?, which is a place holder for a value that is provided when the SQL program is executed.