Database System

北京交通大学软件学院

王方石 教授

E-mail: fshwang@bjtu.edu.cn

Lecture 9

3.6 Programmatic SQL

3.6.1 PSM (*persistent stored modules*).

- Stored Procedure
- Constraints and Trigger

3.6.2 Embedded SQL

- **Static Embedded SQL**
- **Dynamic Embedded SQL**

3.6.3 API (*Application programming interface*).

- CLI
- ODBC
- JDBC

3.6.2 Embedded SQL

- **SQL can be *embedded* in high-level procedural language.**
- **In many cases, SQL language is identical although SELECT statement differs in embedded SQL.**
- **SQL supports Ada, C, COBOL, FORTRAN, and Pascal.**
- ***Embedded SQL statements***
  - **Static Embedded SQL**
  - **Dynamic Embedded SQL**

1. Static Embedded SQL

- **The entire SQL statement is known when the program is written.**
- **Embedded SQL starts with identifier, usually EXEC SQL.**
- **Ends with terminator dependent on host language:**
  - **Ada, 'C': terminator is semicolon (;)**
  - **COBOL: terminator is END-EXEC**
- **Embedded SQL can appear anywhere that an executable host language statement can appear.**

Example in Oracle- CREATE TABLE

```
EXEC SQL CREATE TABLE Viewing
(   propertyNo VARCHAR (25) NOT NULL,
    clientNo VARCHAR (25) NOT NULL,
    viewDate DATE NOT NULL,
    comment VARCHAR(40)
);
if (sqlca.sqlcode >= 0)
   printf("Creation successful\n");
```

SQL Communications Area (SQLCA)

- **SQLCA is used to report runtime errors to the application program.**
- **The most important part of SQLCA is SQLCODE variable:**

  =0 - statement executed successfully;

  < 0 - an error occurred;

  > 0 - statement executed successfully, but an exception occurred, such as no more rows returned by SELECT.

WHENEVER Statement

- **Every embedded SQL statement can potentially generate an error.**
- **WHENEVER is directive to precompiler to generate code to handle errors after every SQL statement:**

```
EXEC SQL WHENEVER
        <condition>      <action>
```

WHENEVER Statement

- ***condition*** **can be:**

  SQLERROR – tell the precompiler to generate code to handle errors (SQLCODE < 0).

  SQLWARNING - tell the precompiler to generate code to handle warnings (SQLCODE > 0).

  NOT FOUND - tell the precompiler to generate code to handle specific warning that a retrieval operation has found no more records (SQLCODE > 0).

WHENEVER Statement

- ***action*** **can be:**

  CONTINUE - ignore condition and proceed to next statement.

  DO - transfer control to an error handling function and then go back to the failed SQL statement.

  DO BREAK - place an actual "break" statement in the program. If used within a loop, then exit from loop.

  DO CONTINUE - place an actual "continue" statement in the program. If used within a loop, continue with the next iteration of the loop.

  GOTO *label* or GO TO *label* - transfer control to specified *label*.

  STOP - rollback all uncommitted work and terminate the program.

WHENEVER Statement

```
EXEC SQL WHENEVER SQLERROR GOTO error1;
EXEC SQL INSERT INTO Viewing VALUES
('CR76', 'PA14', '12-May-2001', 'Not enough
space');
```

- would be converted to:

```
EXEC SQL INSERT INTO Viewing VALUES
('CR76', 'PA14', '12-May-2001', 'Not enough
space');
if (sqlca.sqlcode < 0) goto error1;
```

Host Language Variables

- **It is the program variable declared in host language.**
- **It is used in embedded SQL to transfer data from database into program and vice versa.**
- **It can be used anywhere that a constant can appear.**
- **It cannot be used to represent database objects, such as table names or column names.**
- **To use host variable, prefix it by a colon (:).**

Host Language Variables

```
EXEC SQL UPDATE Staff
        SET salary = salary + :increment
        WHERE staffNo = 'SL21';
```

- Need to declare host language variables to SQL, as well as to host language:

```
EXEC SQL BEGIN DECLARE SECTION;
        float increment;
EXEC SQL END DECLARE SECTION;
```

Indicator Variables

- **It indicates presence of null:**

  =0 means that the associated host variable contains a valid value.

  <0 means that the associated host variable should be assumed to contain a null; the actual content of the host variable is irrelevant.

  >0 means that the associated host variable contains a valid value, which may have been rounded or truncated. E.g. round（4.2）=4, round（4.5）=5,

- **It is used immediately following the associated host variable with a colon (:) separating two variables.**

Indicator Variables - Example

```
EXEC SQL BEGIN DECLARE SECTION;
        char        address[51];
        short       addressInd;
EXEC SQL END DECLARE SECTION;
   addressInd = -1;
EXEC SQL UPDATE PrivateOwner
SET address = :address :addressInd (null)
WHERE ownerNo = 'CO21';
```

It is a two-byte integer variable, so we declare addressInd as type *short* within the BEGIN DECLARE SECTION

Singleton SELECT - Retrieves Single Row

```
EXEC SQL SELECT fName, lName, address
INTO :firstName, :lastName, :address :addressInd
FROM PrivateOwner
WHERE ownerNo = 'CO21';
```

- There must be 1:1 correspondence between expressions in SELECT list and host variables in INTO clause.
- If successful, SQLCODE is set to 0;

  if there are no rows that satisfies WHERE, SQLCODE is set to NOT FOUND.

Cursors

- **If query can return arbitrary number of rows, need to use *cursors*.**
- **Cursor allows host language to access rows of query one at a time.**
- **Cursor acts as a pointer to a row of query result. Cursor can be advanced by one to access next row.**
- **A cursor must be declared and opened before it can be used and it must be closed to deactivate it after it is no longer required.**

Cursors - DECLARE CURSOR

- The **DECLARE CURSOR** statement defines the specific SELECT to be performed and associates a cursor name with the query.
- The format of the statement is:

  EXEC SQL DECLARE <cursorName>

  CURSOR FOR <select statement> ;

- For example:

```
EXEC SQL DECLARE propertyCursor CURSOR
FOR   SELECT propertyNo, street, city
          FROM PropertyForRent
          WHERE staffNo = 'SL41';
```

Cursors - OPEN

- **OPEN statement opens the specified cursor and positions it before the first row of query result:**

EXEC SQL OPEN propertyCursor;

PNo	street	city
SG13	….	London
SG45	…	Paris
SG37	..	New York

Cursors - FETCH and CLOSE

- **Once the cursor has been opened, the rows of the query result can be retrieved one at a time using FETCH:**
- **FETCH retrieves the next row of query result table: EXEC SQL FETCH cursorName INTO :hostVariable,…**
- **FETCH is usually placed in a loop. When there are no more rows to be returned, SQLCODE is set to NOT FOUND.**
- **EXEC SQL CLOSE cursorName;**

Example: Fetching

while (**SQLCODE !=NOT FOUND**)

{

EXEC SQL FETCH propertyCursor

INTO :propertyNo, :street, :city

}

PNo	street	city
SG13	….	London
SG45	…	Paris
SG37	..	New York

2. Dynamic Embedded SQL

- **There are many situations where the pattern of DB access is not fixed and is known only at runtime.**
- ***Dynamic SQL*** **allows all or part of the SQL statement to be specified at runtime. It provides increased flexibility and help produce more general-purpose software.**

- **The basic difference between the two types of embedded SQL is that *static* embedded SQL does not allow host variables to be used in place of database object (table or column) names.**

We can not write the following static SQL:

```
EXEC SQL BEGIN DECLARE SECTION;
  char TableName[2];
EXEC SQL END DECLARE SECTION;
EXEC SQL INSERT INTO :TableName  // unallowable
        VALUES('S12','John',18,'m');
// But  Dynamic SQL allows this.
```

- **The basic idea of *Dynamic SQL* is to place the complete SQL statement in a host variable, which is passed to DBMS to be executed.**

Two types of Dynamic SQL statement

(1) EXECUTE IMMEDIATE （立即执行方式）

(2) PREPARE and EXECUTE (准备执行方式)

(1) EXECUTE IMMEDIATE

- **If SQL statements do not involve SELECT statement, use EXECUTE IMMEDIATE statement which has the format:**

EXEC SQL EXECUTE IMMEDIATE
[hostVariable | stringLiteral]

- **This command allows the SQL statements stored in *hostVariable*, or in the literial, *stringLiteral*, to be executed.**
- **It cannot pass the parameters to SQL.**

E.g.: increase wage for employee'SL21'

```
EXEC SQL BEGIN DECLARE SECTION;
  char  buffer[100];
EXEC SQL END DECLARE SECTION;
 sprintf(buffer, "UPDATE Staff
     SET salary = salary + %f
     WHERE staffNo = 'SL21' ", increment);
EXEC SQL EXECUTE IMMEDIATE :buffer;
```

No need to have EXEC SQL and ;

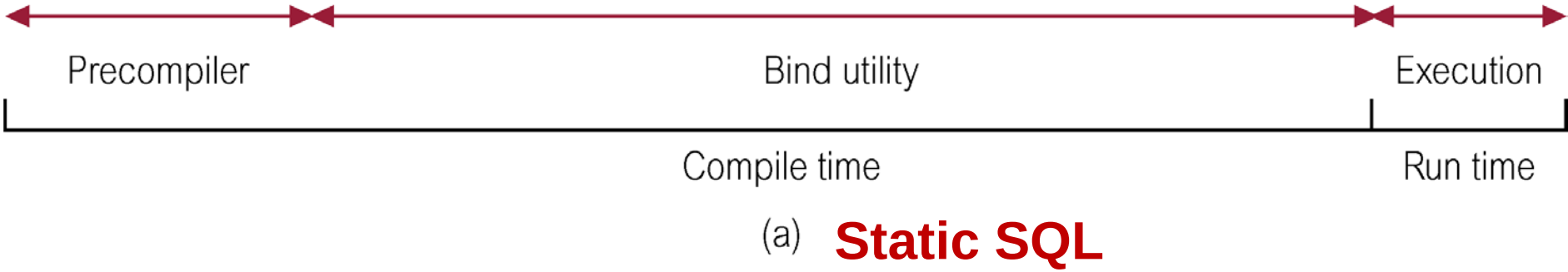
Without :, No need to be declared

SQL statement Itself, not the parameter passed to SQL

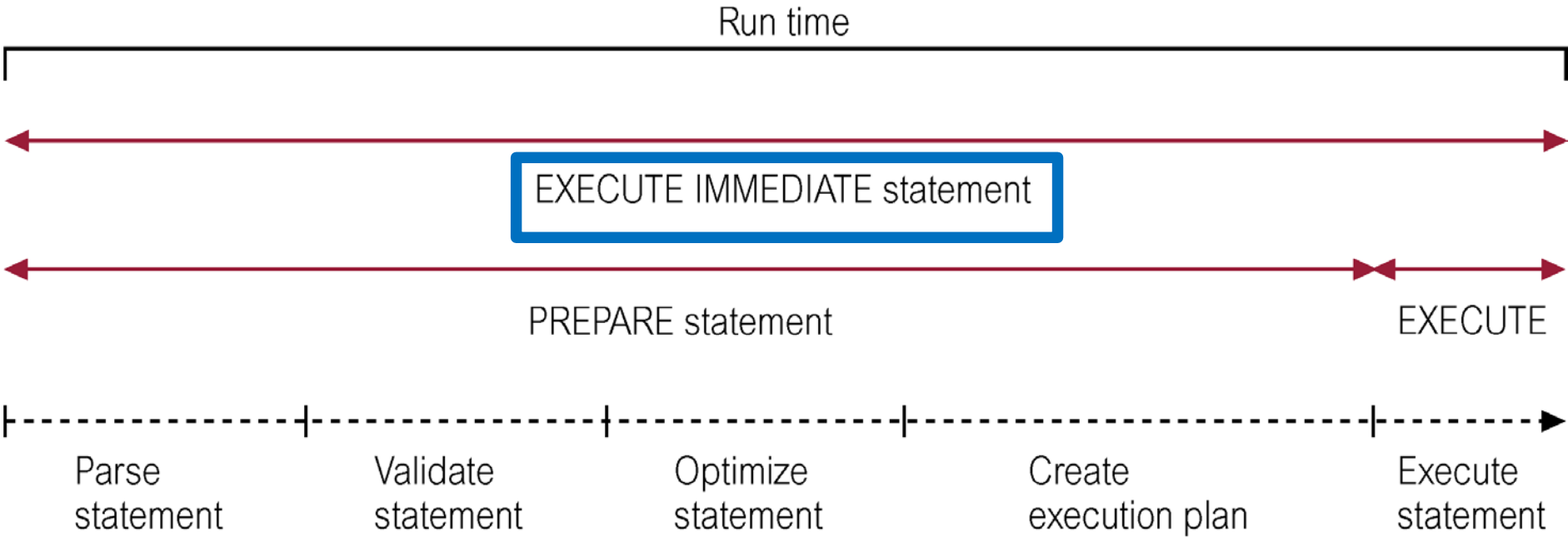
(2) PREPARE and EXECUTE

- **Every time an EXECUTE IMMEDIATE statement is processed, the DBMS must parse, validate, and optimize the statement, build an execution plan for the statement, and finally execute this plan.**
- **OK if SQL statement is only executed once in program; otherwise inefficient.**
- **Dynamic SQL provides alternative: PREPARE and EXECUTE.**
- **PREPARE tells DBMS to ready dynamically built statement for later execution.**

Static versus Dynamic SQL



(a) **Static SQL**



(b) **Dynamic SQL**

PREPARE and EXECUTE

- **The prepared statement is assigned a specified statement name.**
- **When the statement is subsequently executed, the program need only specify this name:**

```
EXEC SQL PREPARE statementName
        FROM [hostVariable | stringLiteral]
EXEC SQL EXECUTE statementName
      [ USING hostVariable [indicatorVariable] [, ...] |
      USING DESCRIPTOR descriptorName ]
```

Example for PREPARE and EXECUTE

E.g.： delete an employee with a specified empNo many times.

execute: **delete from emp where empno=:pempno**

scanf("%s",**dstring**); **// input the above *delete* by keyboard**

EXEC SQL PREPARE s1 FROM :dstring;

scanf("%d", & staffno);

WHILE (staffno!=0)

{ EXEC SQL **EXECUTE s1 USING :staffno**;

scanf("%d", & staffno);

}

Also can : USING :VAR1, :VAR2,…

3.6.3 *Application programming interface* (API)

- **API is an alternative technique to provide the programmer with a standard set of functions that can be invoked from the software.**
- **The DBMS vendor provides an API.**
- **An API can provide the same functionality as embedded statements and removes the need for any precompilation. e.g.**
  - **SQL/CLI （call-level interface 调用层接口）**
  - **ODBC**
  - **JDBC**

1. SQL/CLI

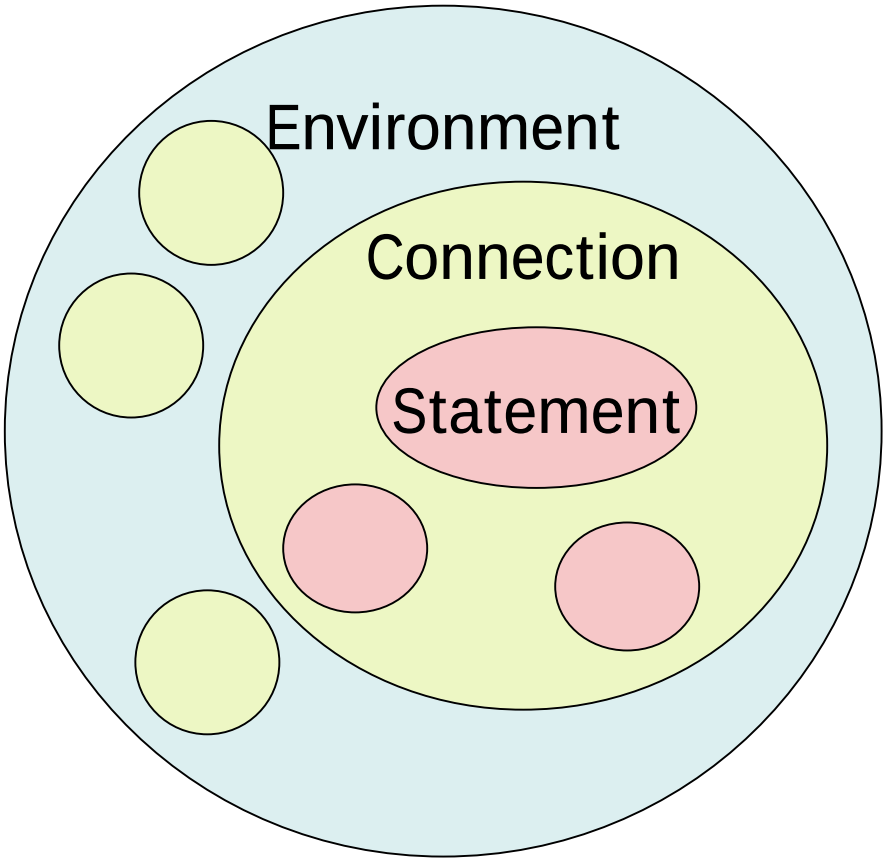
- Instead of using a preprocessor (as in embedded SQL), we can use **a library of functions**.
  - The library for C is called SQL/CLI = "*Call-Level Interface*." **No need for any precompilation.**

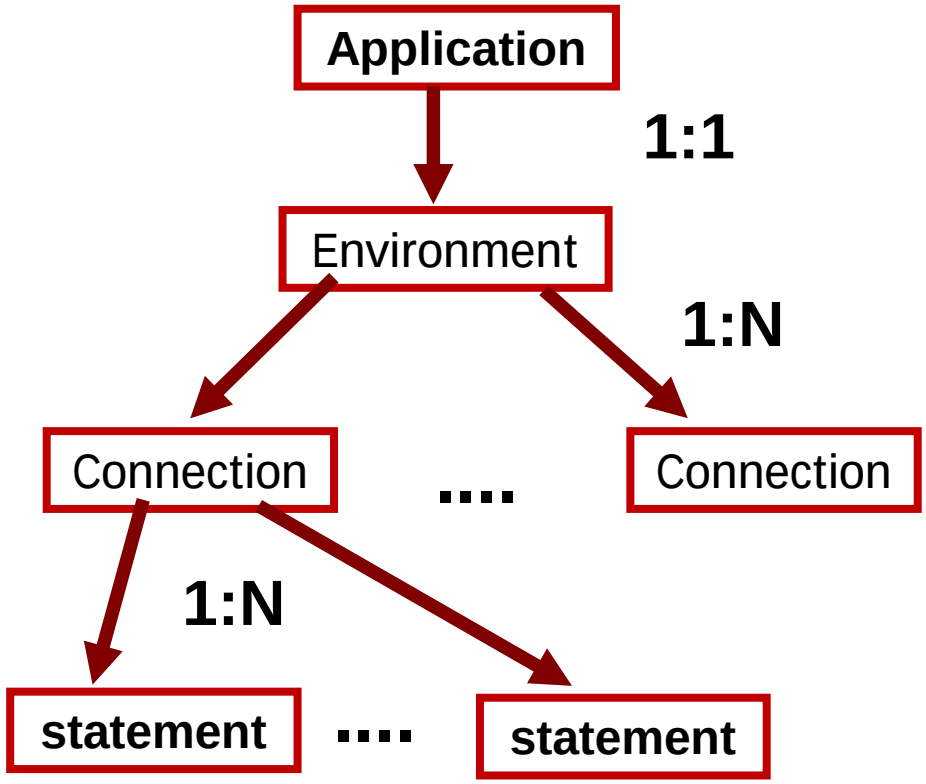
    CLI was defined in SQL-92 and SQL:1999.
  - Embedded SQL's preprocessor will translate the EXEC SQL ... statements into CLI or similar calls, anyway. **Need for precompilation.**

Environments, Connections, Statements

- The database is, in many DB-access languages, an *environment*.
- Database servers maintain some number of *connections*, so application servers can ask queries or perform modifications.
- The application server issues *statements* : queries and modifications, usually.

Diagram to Remember





Data Structures

In C, header file sqlcli.h will be included. The program is then able to create and deal with four types of records. **C** connects to the database by structs of the following :

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. In CLI, description records will generally invisible.

Each of these records is represented in the application program by **a handle**, which is a **pointer** to the record.

Handles

- Function **SQLAllocHandle (hType,hIn,hOut)** is used to create these structs, which are called environment, connection, and statement *handles*.
  - *hType* is the type of the handle , e.g., SQL_HANDLE_STMT.
  - *hIn* is the handle of the higher-level element in which the newly allocated element lives. (statement < connection < environment).
  - *hOut* is the address of the handle that is 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** (***sh***, ***st***, ***sl***) causes

  sh—a statement handle

  st —a pointer to a SQL statement

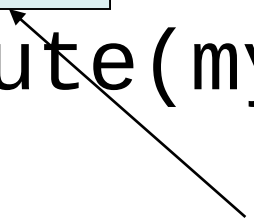
  sl —the length of the character string pointed to by ***st***.

  st is to be interpreted as a SQL statement and optimized; the executable statement is placed in statement handle ***sh***.

- **SQLExecute**(***sh***) causes the SQL statement represented by statement handle ***sh*** to be executed.

Example: Prepare and Execute

```
SQLPrepare(myStat,
"SELECT sname, age FROM S
  WHERE sno = 's1'",
  SQL_NTS);
SQLExecute(myStat);
```



This defined constant says the second argument is a “null-terminated string”; i.e., figure out the length by counting characters until encountering the endmarker ’\0’.

Direct Execution

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

  SQLExecuteDirect (***sh***, ***st***, ***sl***);

  – As before, ***sh*** is a statement handle and ***sl*** is the length of string ***st***.

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** (***sh***) gets the next tuple from the result of the statement with handle ***sh***.

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(sh,colNo,colType,pVar,varSize,varInfo)

- This function has 6 arguments, of which we shall show only 1, 2, and 4:

  1 = **sh:** handle of the query statement.

  2 = **colNo** is the column number of the component(within the tuple) whose value we obtain.

  4 = **pVar** is a pointer to the variable into which the value is to be placed.

Example: Binding

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

```
SELECT sname, age FROM S
WHERE sno = 's1'
```

- Bind the result to *theSname* and *theAge*:

SQLBindCol(myStat, 1, , & *theSname*, ,);

SQLBindCol(myStat, 2, , & *theAge*, ,);

Example: Fetching

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

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

It represents that SQLSTATE = 02000 = “failed to find a tuple.” It is used to get out of the loop in which we repeatedly fetch new tuples from the result.

Lecture 10-1

2. Open Database Connectivity (ODBC)

- **Rather than embedding raw SQL within program, DBMS vendor instead provides an API.**
- **API consists of set of library functions for many common types of database accesses.**
- **One problem with this approach has been lack of interoperability.**
- **To standardize this approach, Microsoft produced ODBC standard.**
- **ODBC provides common interface for accessing hetero'geneous SQL databases, based on SQL.**

Open Database Connectivity (ODBC)

- **Interface (built on ‘C’) provides high degree of interoperability: single application can access different SQL DBMSs through common code.**
- **Enables developer to build and distribute *client-server* application without targeting specific DBMS.**
- **Database drivers are then added to link application to user’s choice of DBMS.**
- **ODBC has emerged as a *de facto* industry standard.**

ODBC's Flexibility

- **Applications not tied to specified vendor API.**
- **SQL statements can be explicitly included in source code or be constructed dynamically.**
- **An application can ignore underlying data communications protocols.**
- **Data can be sent and received in format that is convenient to application.**
- **ODBC is designed in conjunction with**

  ISO CLI standard.
- **There are ODBC drivers available today for many of most popular DBMSs.**

ODBC Interface

The ODBC Interface defines the following items:

- **Library of functions that allow application to connect to DBMS, execute SQL statements, and retrieve results.**
- **A standard way to connect and log on to a DBMS.**
- **A standard representation of data types.**
- **A standard set of error codes.**
- **SQL syntax based on ISO Call-Level Interface (CLI) specifications.**

ODBC Architecture

- **ODBC architecture has four components:**
  - **Application,**
  - **Driver Manager,**
  - **Driver and Database Agent,**
  - **Data Source.**

ODBC Architecture

Application - performs processing and calls ODBC functions to submit SQL statements to DBMS and to retrieve results from DBMS.

Driver Manager - loads drivers on behalf of application. Driver Manager, provided by Microsoft, is Dynamic-Link Library (DLL).

用于连接各种数据库系统的驱动程序，其主要作用是用来加载ODBC驱动程序，检查ODBC调用参数的合法性和记录ODBC函数的调用，并为不同驱动程序的ODBC函数提供单一的入口，调用正确的驱动程序，提供驱动程序信息等。

ODBC Architecture

Driver - process ODBC function calls, submit SQL requests to specific data source, and return the results to application.

•If necessary, driver modifies application's request so that it conforms to syntax supported by associated DBMS.

Data Source - consists of data which user wants to access and its associated DBMS, and its host operating system, and network platform, if any.

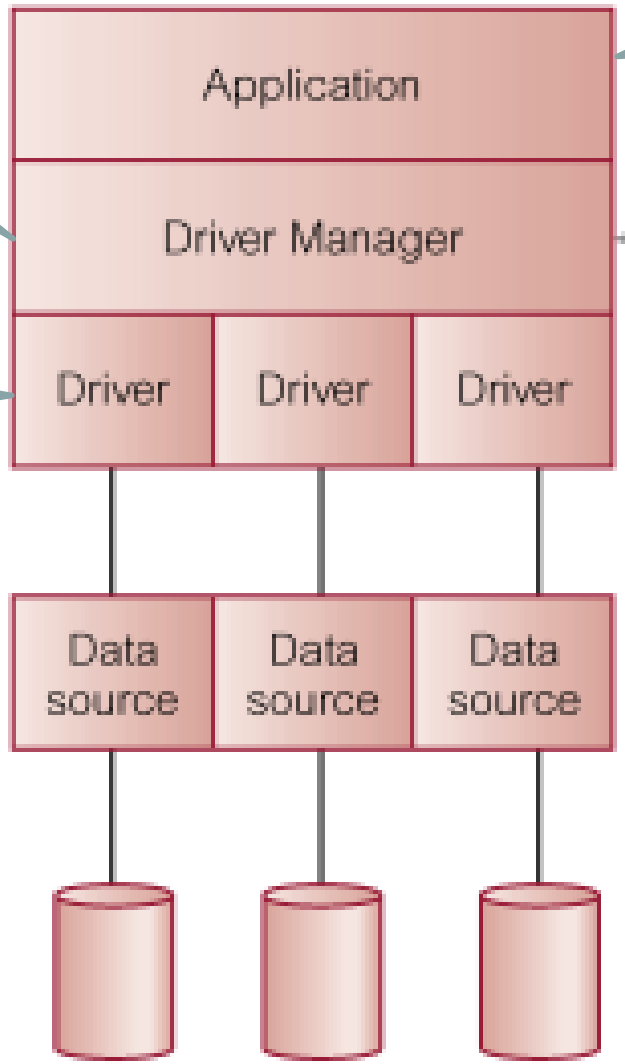
ODBC 体系结构

- 应用程序 – 执行处理，调用ODBC函数将 SQL 语句提交给DBMS，并从DBMS获得查询结果.
- 驱动程序管理器 – 代表应用程序加载和卸载驱动程序。它也可以处理ODBC函数调用，或者将调用递交给驱动程序。微软提供的驱动程序管理器是一个动态链接库(Dynamic-Link Library -DLL)。
- 驱动程序和数据库代理 – 处理ODBC函数调用，将SQL请求提交给特定的数据源，将结果返回给应用程序。若需要，驱动程序修改应用程序的请求，以便使请求符合相关联DBMS支持的语法。
- 数据源 – 由用户希望访问的数据、相关联的DBMS、宿主操作系统以及网络平台（若有的话）共同组成。

ODBC Architecture--- Using metaphor

overseer of a decorating group

House owner



Different driver-- Different professionals: carpenter, electrician, plumber, bricklayer

Different jobs— different DBMS: Oracle, SQL Server, DB2,Sybase,MySQL

ODBC Code

```
•int ODBCexample()
{
    RETCODE error;
    HENV    env;     /* environment */
    HDBC    conn;  /* database connection */
    SQLAllocEnv(&env);
    SQLAllocConnect(env, &conn);
    SQLConnect(conn, "aura.bell-labs.com", SQL_NTS, "avi",
    SQL_NTS, "avipasswd", SQL_NTS);
    { …. Do actual work … }

    SQLDisconnect(conn);
    SQLFreeConnect(conn);
    SQLFreeEnv(env);
}
```

DB Server name

It represents that its preceding parameter is a string ending with NULL.

ODBC Code (Cont.)

- Main body of program

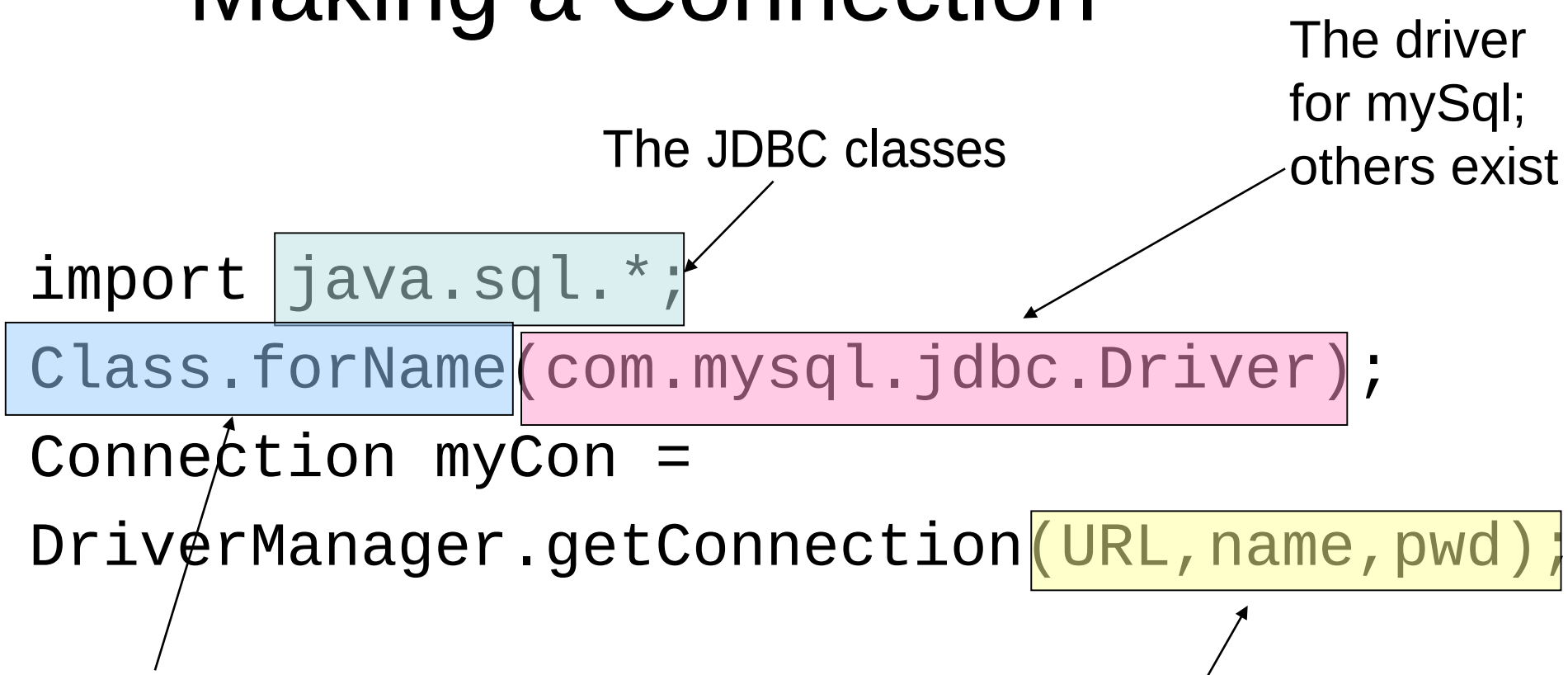
```
char branchname[80];
float  balance;
int  lenOut1, lenOut2;
HSTMT   stmt;
SQLAllocStmt(conn, &stmt);
char * sqlquery = "select branch_name, sum (balance)
                   from account group by branch_name";
error = SQLExecDirect(stmt, sqlquery, SQL_NTS);
if (error == SQL_SUCCESS)
{
  SQLBindCol(stmt, 1, SQL_C_CHAR, branchname , 80, &lenOut1);
  SQLBindCol(stmt, 2, SQL_C_FLOAT, &balance,  0 , &lenOut2);
  while (SQLFetch(stmt) >= SQL_SUCCESS)
        printf (" %s  %g\n", branchname, balance);
}
SQLFreeStmt(stmt, SQL_DROP);
```

3. *Java Database Connectivity* (JDBC)

- JDBC is a library similar to SQL/CLI, but with **Java as the host language**.

- Like CLI, but with a few differences for us to cover.

Making a Connection



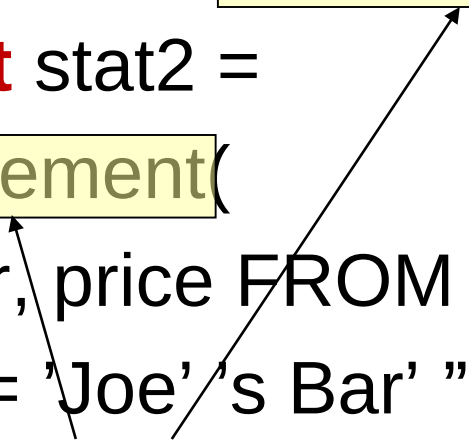
Statements

- JDBC provides two classes:
  1. ***Statement*** = an object that can accept a string that is a SQL statement and can execute such a string.
  2. ***PreparedStatement*** = an object that has an associated SQL statement ready to execute.

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, which it calls “updates.”
- ***Statement*** and ***PreparedStatement*** each have two methods
  - **executeQuery** which return a ***ResultSet*** object
  - **executeUpdate** with no returned result set.
  - For Statements: with one argument. **executeQuery** (Q) or **executeUpdate** (U).
  - For PreparedStatements: no argument. **executeQuery** ( ) or **executeUpdate** ( )

Example: Update

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

```
stat1.executeUpdate(
  "INSERT  INTO Sells " +
   "VALUES('Brass Rail','Bud',4.00)"
);
```

String of SQL statement as argument **U, no returned result set**

Example: Query

- stat2 is a ***PreparedStatement*** holding the query

  ”SELECT sname, age FROM S

  WHERE sno = ‘s1’ ”.

- **executeQuery** returns an object of class ***ResultSet.***

- The query:

  ResultSet menu = stat2.executeQuery();

without argument

Accessing the ResultSet

- An object of type ***ResultSet*** is something like a cursor.
- Method **next()** advances the "cursor" to the next tuple.
  - At the first time when **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 (like a pointer) 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.
  - The value must have type ***X***.
- X canbe Int，Float, String，then：

  getInt (*i*)，getFloat (*i*)，getString (*i*)

Example: Accessing Components

- **Menu** = ResultSet for query

  “SELECT lname, salary FROM Staff

  WHERE staffNo = ’s1’ ”.

- Access *lname* and *salary* from each tuple by:

```
while ( menu.next() ) {
  theLname = Menu.getString(1);
  theSalary = Menu.getFloat(2);
  /*something with theLname and
        theSalary*/
}
```

Like a cursor

JDBC Code

```
public static void JDBCexample (String dbid, String userid,
String passwd)
 {
    try {
      Class.forName ("oracle.jdbc.driver.OracleDriver");
      Connection conn =
         DriverManager.getConnection(   "jdbc:oracle:thin:@aura.bell-
                                    labs.com:2000:bankdb", userid, passwd);
      Statement stmt = conn.createStatement();
         … Do Actual Work ….
      stmt.close();
      conn.close();
   }
   catch (SQLException sqle)
    {   System.out.println("SQLException : " + sqle);  }

}
```

API Protocol supported by both DB and driver

host computer name where DB Server is located and port number

JDBC Code (Cont.)

- Update to database account(ANo, branchName, Balance)

```
try {
      stmt.executeUpdate(  "insert into account values
                                ('A-9732', 'Perryridge', 1200)");
    } catch (SQLException sqle)
      {  System.out.println("Could not insert tuple. " + sqle);}
```

- Execute query and fetch and print results

```
ResultSet rset = stmt.executeQuery(
                          "select branchName, avg(balance)
                           from account
                           group by branchName");
while (rset.next()) //若非空
{ System.out.println(
        rset.getString("branchName") + "   " + rset.getFloat(2));
}
```

JDBC Code Details

- Getting result fields:
  - **rs.getString (“branchName”)** and **rs.getString(1)** equivalent if branchName is the first argument of select result.
- Dealing with Null values

  if (**rs.wasNull()**) Systems.out.println(“Got null value”);

Prepared Statement

- **Prepared statement allows queries to be compiled once and executed multiple times with different arguments**

```
PreparedStatement pStmt = conn.prepareStatement(
    “insert into account values(?,?,?)”);
pStmt.setString(1, "A-9732");
pStmt.setString(2, "Perryridge");
pStmt.setInt(3, 1200);
pStmt.executeUpdate();

pStmt.setString(1, "A-6933");
pStmt.executeUpdate();
```

(A-9732, Perryridge,1200)
(A-6933, Perryridge,1200)