# CSCD 327: Relational Database Systems

## Database application development

Instructor: Dr. Dan Li

# Overview

- SQL in application code
- Embedded SQL
- Cursors
- ODBC
- JDBC
- Stored procedures/Functions/Triggers

# SQL as an Independent Query Language

- SQL is a independent query language; as such, it has limitations.
- via programming languages :
  - Complex computational processing of the data.
  - Specialized user interfaces.
  - Access to more than one database at a time.

# SQL in Application Code

- SQL commands can be called from within a host language (e.g., C++ or Java) program.
  - SQL statements can refer to host variables (including special variables used to return status).
  - Must include a statement to *connect* to the right database.
- Limitations
  - SQL relations are (multi-) sets of records, with no *a priori* bound on the number of records. No such data structure exist traditionally in procedural programming languages such as C++. (Though now: STL)
    - SQL supports a mechanism called a *cursor* to handle this.
    - A cursor is an additional SQL construct that allows applications to obtain a handle on a collection and iterate over the records one at a time.

# Desirable features of such systems:

- Ease of use.
- Conformance to standards for existing programming languages, database query languages, and development environments.
- Interoperability: the ability to use a common interface to diverse database systems on different operating systems

# Vendor specific solutions

- Oracle PL/SQL: is Oracle's procedural extension to SQL. It supplements SQL with several high-level programming language features.
- Advantages:
  - Many Oracle-specific features, not common to other systems, are supported.
  - Performance may be optimized to Oracle-based systems.
- Disadvantages:
  - Ties the applications to a specific DBMS.
  - The application programmer must depend upon the vendor for the application development environment.
  - It may not be available for all platforms.

# Vendor Independent solutions based on SQL

There are two basic strategies which may be considered:

- Embed SQL in the host language
  - Embedded SQL
    - Cursors
  - Dynamic SQL
- SQL APIs
  - JDBC
  - ODBC
- Stored Procedures and Triggers
  - Having DBMS take on more responsibility

# Embedded SQL

- Approach: Embed SQL in the host language.
  - A preprocessor converts the SQL statements into special API calls.
  - Then a regular compiler is used to compile the code.

- Language constructs:
  - Connecting to a database:
    EXEC SQL CONNECT
  - Declaring variables:
    EXEC SQL BEGIN (END) DECLARE SECTION
  - Statements:
    EXEC SQL Statement;

# Embedded SQL: Variables

```
EXEC SQL BEGIN DECLARE SECTION
char c_sname[20];
long c_sid;
short c_rating;
float c_age;
EXEC SQL END DECLARE SECTION
```

- Two special “error” variables:
  - SQLCODE (long, is negative if an error has occurred)
  - SQLSTATE (char[6], predefined codes for common errors)

# Cursors

- Can declare a cursor on a relation or query statement (which generates a relation).
- Can *open* a cursor, and repeatedly *fetch* a tuple then *move* the cursor, until all tuples have been retrieved.
  - Can use a special clause, called ORDER BY, in queries that are accessed through a cursor, to control the order in which tuples are returned.
  - Fields in ORDER BY clause must also appear in SELECT clause.
  - The ORDER BY clause, which orders answer tuples, is *only* allowed in the context of a cursor.
- Can also modify/delete tuple pointed to by a cursor.

# Cursor that gets names of sailors who've reserved a red boat, in alphabetical order

```
EXEC SQL DECLARE sinfo CURSOR FOR
    SELECT  S.sname
    FROM  Sailors S, Boats B, Reserves R
    WHERE  S.sid=R.sid AND R.bid=B.bid AND B.color='red'
    ORDER BY  S.sname
```

- Note that it is illegal to replace *S.sname* by, say, *S.sid* in the ORDER BY clause! (Why?)

# Embedding SQL in C: An Example

```
char SQLSTATE[6];
EXEC SQL BEGIN DECLARE SECTION
char c_sname[20]; short c_minrating; float c_age;
EXEC SQL END DECLARE SECTION
c_minrating = random();
EXEC SQL DECLARE sinfo CURSOR FOR
    SELECT S.sname, S.age        FROM Sailors S
    WHERE S.rating > :c_minrating
    ORDER BY S.sname;
do {
    EXEC SQL FETCH sinfo INTO :c_sname, :c_age;
    printf("%s is %d years old\n", c_sname, c_age);
} while (SQLSTATE != '02000');
EXEC SQL CLOSE sinfo;
```

# Dynamic SQL

- SQL query strings are not always known at compile time (e.g., spreadsheet, graphical DBMS frontend): Allow construction of SQL statements on-the-fly
- Example:

```
char c_sqlstring[]=
    {“DELETE FROM Sailors WHERE rating>5”};
EXEC SQL PREPARE readytogo FROM :c_sqlstring;
EXEC SQL EXECUTE readytogo;
```

- Disadvantages
  - It is a real pain to debug preprocessed programs.
  - The use of a program-development environment is compromised substantially.
  - The preprocessor must be vendor and platform specific.

# SQL APIs

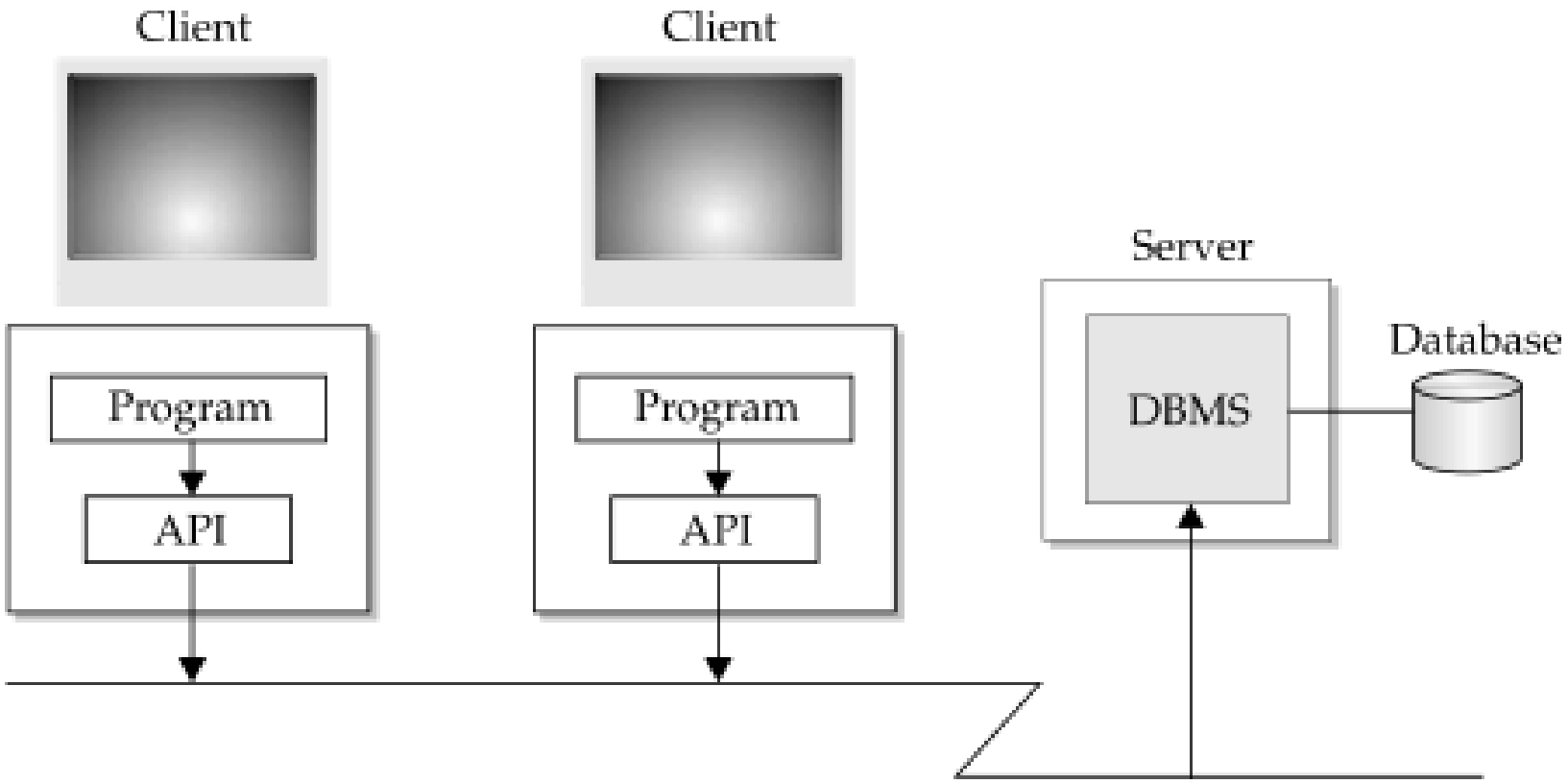
- Instead of trying to blend SQL with another programming language, many database products provide a library of function calls as an application programming interface (API) for the DBMS.
- To pass SQL statements to the DBMS, an application program calls functions in the API, and it calls other functions to retrieve query results and status information from the DBMS.

# Pros & Cons

- Advantages over embedded SQL:
  - Clean separation of SQL from the host programming language.
  - Debugging is much more straightforward, since no preprocessor is involved.

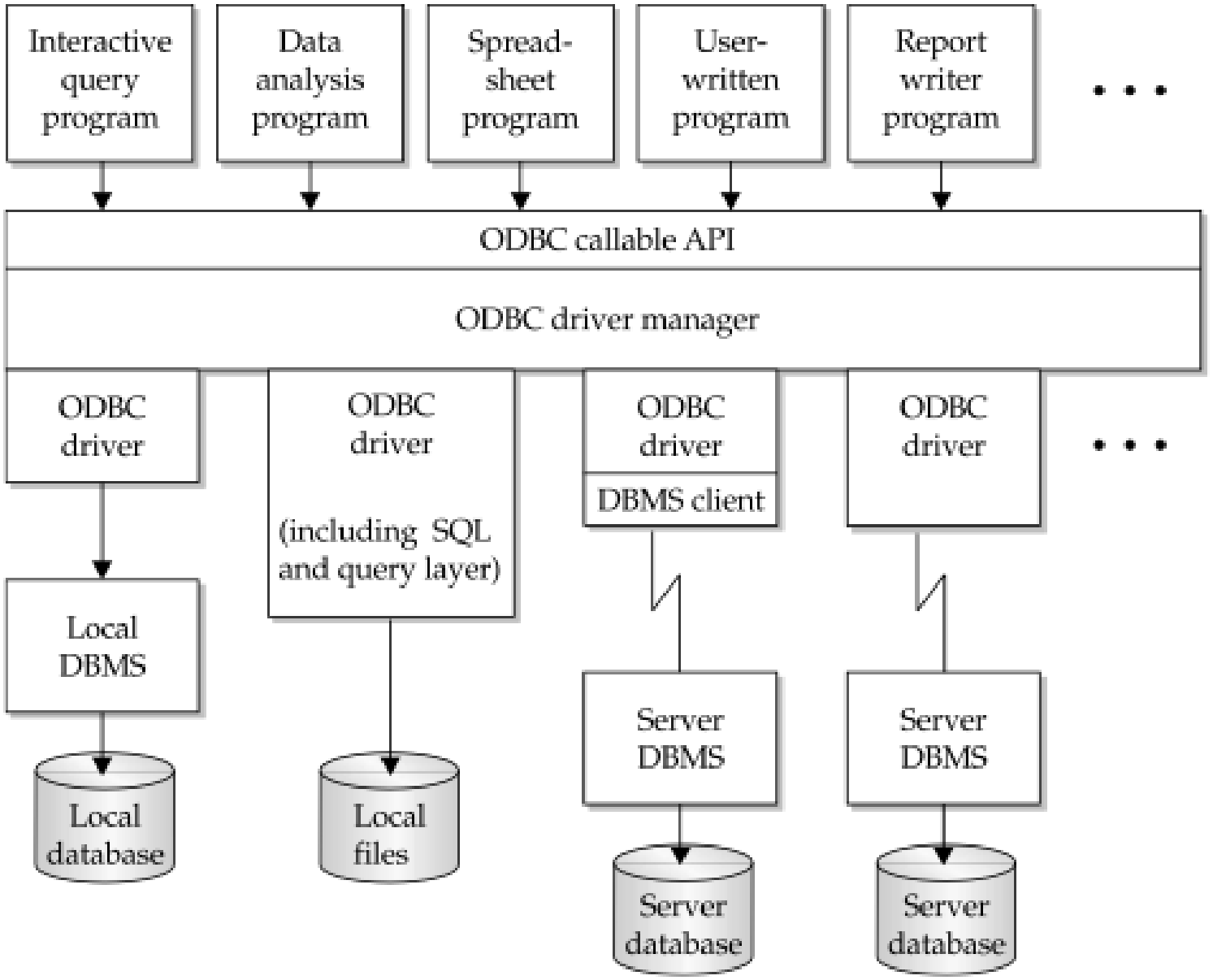
- Disadvantages:
  - The module libraries are specific to the programming language and environment. Thus, portability is compromised greatly.

# An SQL API in a client/server Architecture



# **O**pen **D**ata**B**ase **C**onnectivity

- Shorten to ODBC, a standard database access method
- The goal:  make it possible to access any data from any application, regardless of which (DBMS).
- ODBC manages this by inserting a middle layer, called a database *driver ,* between an application and the DBMS.
- The purpose of this layer is to translate the application's data queries into commands that the DBMS understands.
- For this to work, both the application and the DBMS must be ODBC-compliant -- that is, the application must be capable of issuing ODBC commands and the DBMS must be capable of responding to them.


Interactive query program
Data analysis program
Spread-sheet program
User-written program
Report writer program
ODBC callable API
ODBC driver manager
ODBC driver
ODBC driver
(including SQL and query layer)
ODBC driver
DBMS client
ODBC driver
Local DBMS
Local database
Local files
Server DBMS
Server database
Server DBMS
Server database

# Configuring a Data Source (Access) under Windows

- Open the ODBC menu in the control panel.
- Click on the User DSN tab.
  - click on Add.
- From the menu in the new window,
  - select Microsoft Access Driver (sailors.mdb),
  - click on Finish.
- From the menu in the new window,
  - type in a data source name (mysailors), and optionally, a description.
  - Then click on either Select or Create, depending upon whether you want to link to an existing database, or create a new blank one.
- In the new window, give the path to the database.
- "OK" away the pile of subwindows; the new database should appear under the top-level ODBC User DSN tab.

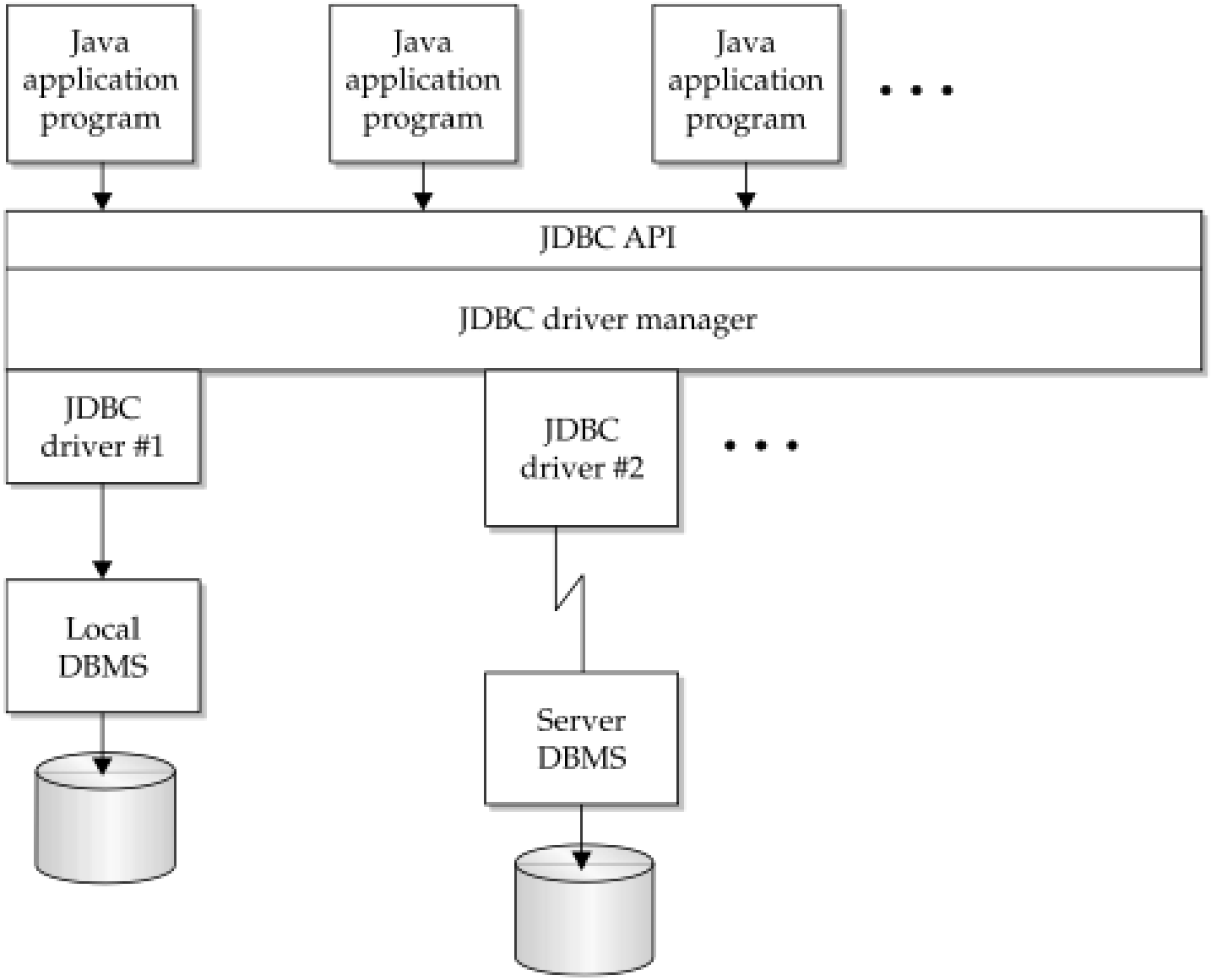
```
// program connects to an ODBC data source called “mysailors“ then executes SQL
    statement “SELECT *  FROM Sailors';"
#include <windows.h>
#include <sqlext.h>
#include <stdio.h>

int main(void)
{
    HENV   hEnv = NULL;                                  // Env Handle from SQLAllocEnv()
    HDBC   hDBC = NULL;                                  // Connection handle
    HSTMT          hStmt = NULL;                                  // Statement handle
    UCHAR          szDSN[SQL_MAX_DSN_LENGTH] = “mysailors";// Data Source Name
    buffer
    UCHAR*         szUID = NULL;                                  // User ID buffer
    UCHAR*         szPasswd = NULL;                              // Password buffer
    UCHAR          szname[255];                                  //  buffer
    SDWORD         cbname;                                        // bytes recieved
    UCHAR          szSqlStr[] = "Select * From Sailors”;         // SQL string
    RETCODE        retcode;                                       // Return code

// Allocate memory for ODBC Environment handle
SQLAllocEnv (&hEnv);

// Allocate memory for the connection handle
SQLAllocConnect (hEnv, &hDBC);
```

```
// Connect to the data source “mysailors" using userid and password.
retcode = SQLConnect (hDBC, szDSN, SQL_NTS, szUID, SQL_NTS, szPasswd, SQL_NTS);

if (retcode == SQL_SUCCESS || retcode == SQL_SUCCESS_WITH_INFO)
    {
     // Allocate memory for the statement handle
     retcode = SQLAllocStmt (hDBC, &hStmt);

     // Prepare the SQL statement by assigning it to the statement handle
     retcode = SQLPrepare (hStmt, szSqlStr, sizeof (szSqlStr));

     // Execute the SQL statement handle
     retcode = SQLExecute (hStmt);

     // Project only column 2 which is the name
     SQLBindCol (hStmt, 2, SQL_C_CHAR, szname, sizeof(szname), &cbModel);

     // Get row of data from the result set defined above in the statement
     retcode = SQLFetch (hStmt);
```

```
while (retcode == SQL_SUCCESS || retcode == SQL_SUCCESS_WITH_INFO)
      {       printf ("\t%s\n", szname);            // Print row (sname)
              retcode = SQLFetch (hStmt);       // Fetch next row from result set
      }

      // Free the allocated statement handle
      SQLFreeStmt (hStmt, SQL_DROP);

      // Disconnect from datasource
      SQLDisconnect (hDBC);
     }

     // Free the allocated connection handle
     SQLFreeConnect (hDBC);

     // Free the allocated ODBC environment handle
     SQLFreeEnv (hEnv);
     return 0;
}
```

# JDBC: Architecture

- Four architectural components:
  - Application (initiates and terminates connections, submits SQL statements)
  - Driver manager (load JDBC driver)
  - Driver (connects to data source, transmits requests and returns/translates results and error codes)
  - Data source (processes SQL statements)


Java application program
Java application program
Java application program
JDBC API
JDBC driver manager
JDBC driver #1
JDBC driver #2
Local DBMS
Server DBMS

# JDBC API

- Java is an object-oriented language, so it's probably no surprise that JDBC organizes its API functions around a collection of database-related objects and the methods that they provide:
- **Driver Manager object** The entry-point to JDBC
- **Connection objects** Represent individual active connections to target databases
- **Statement objects** Represent SQL statements to be executed
- **ResultSet objects** Represent the results of a SQL query
- **MetaData objects** Represent metadata about databases, query results, and statements
- **Exception objects** Represent errors in SQL statement execution

# DriverManager Object Methods

| Method | Description |
|---|---|
| `getConnection()` | Creates and returns a database connection object, given a URL for the datasource, and optionally a user name and password, and connection properties |
| `registerDriver()` | Registers a driver with JDBC driver manager |
| `setLoginTimeout()` | Sets timeout for connection login |
| `getLoginTimeout()` | Obtains login timeout value |
| `setLogWriter()` | Enables tracing of JDBC calls |

# Connection Object Methods

| Method | Description |
|---|---|
| `close()` | Closes the connection to the datasource |
| `createStatement()` | Creates a Statement object for the connection |
| `prepareStatement()` | Prepares a parameterized SQL statement into a `PreparedStatement` for execution |
| `prepareCall()` | Prepares a parameterized call to a stored procedure or function into a `CallableStatement` for execution |
| `commit()` | Commits the current transaction on the connection |
| `rollback()` | Rolls back the current transaction on the connection |
| `setAutoCommit()` | Sets/resets autocommit mode on the connection |
| `getWarnings()` | Retrieves SQL warning(s) associated with a connection |
| `getMetaData` | Returns a `DatabaseMetaData` object with info about database |

# Statement Object Methods

| Method | Description |
|---|---|
| *Basic statement execution* | |
| `executeUpdate()` | Executes a nonquery SQL statement and returns the number of rows affected |
| `executeQuery()` | Executes a single SQL query and returns a result set |
| `execute()` | General-purpose execution of one or more SQL statements |
| *Statement batch execution* | |
| `addBatch()` | Stores previously supplied parameter values as part of a batch of values for execution |
| `executeBatch()` | Executes a sequence of SQL statements; returns an array of integers indicating the number of rows impacted by each one |
| *Query results limitation* | |
| `setMaxRows()` | Limits number of rows retrieved by a query |
| `getMaxRows()` | Retrieves current maximum row limit setting |
| `setMaxFieldSize()` | Limits maximum size of any retrieved column |
| `getMaxFieldSize()` | Retrieves current maximum field size limit |
| `setQueryTimeout()` | Limits maximum time of query execution |
| `getQueryTimeout()` | Retrieves current maximum query time limit |
| *Error handling* | |
| `getWarnings()` | Retrieves SQL warning(s) associated with statement execution |

# ResultSet Object Methods

| Method | Description |
|---|---|
| *Cursor motion* | |
| `next()` | Moves cursor to next row of query results |
| `close()` | Ends query processing; closes the cursor |
| *Basic column-value retrieval* | |
| `getInt()` | Retrieves integer value from specified column |
| `getShort()` | Retrieves short integer value from specified column |
| `getLong()` | Retrieves long integer value from specified column |
| `getFloat()` | Retrieves floating point numeric value from specified column |
| `getDouble()` | Retrieves double-precision floating point value from specified column |
| `getString()` | Retrieves character string value from specified column |
| `getBoolean()` | Retrieves true/false value from specified column |
| `getDate()` | Retrieves date value from specified column |
| `getTime()` | Retrieves time value from specified column |
| `getTimestamp()` | Retrieves timestamp value from specified column |
| `getByte()` | Retrieves byte value from specified column |
| `getBytes()` | Retrieves fixed-length or variable-length `BINARY` data from specified column |
| `getObject()` | Retrieves any type of data from specified column |
| *Large object retrieval* | |
| `getAsciiStream()` | Gets input stream object for processing a character large object (`CLOB`) column |
| `GetBinaryStream()` | Gets input stream object for processing a binary large object (`BLOB`) column |
| *Other functions* | |
| `getMetaData()` | Returns a `ResultSetMetaData` object with metadata for query |
| `getWarnings()` | Retrieves SQL warnings associated with the `ResultSet` |

# DatabaseMetaData Methods

| Function | Description |
|---|---|
| `getTables()` | Returns result set of table information of tables in database |
| `getColumns()` | Returns result set of column names and type info, given table name |
| `getPrimaryKeys()` | Returns result set of primary key info, given table name |
| `getProcedures()` | Returns result set of stored procedure info |
| `getProcedureColumns()` | Returns result set of info about parameters for a specific stored procedure |

# SQLException Methods

| Method | Description |
|---|---|
| `getMessage()` | Retrieves error message describing the exception |
| `getSQLState()` | Retrieves `SQLSTATE` value (5-char string, as described in Chapter 17) |
| `getErrorCode()` | Retrieves driver-specific or DBMS-specific error code |
| `getNextException()` | Moves to next SQL exception in a series |

# JDBC Classes and Interfaces

Steps to submit a database query:

- Load the JDBC driver
- Connect to the data source
- Execute SQL statements

# JDBC Driver Management

- All drivers are managed by the DriverManager class
- Loading a JDBC driver:
  - In the Java code:
    Class.forName("com.mysql.jdbc.Driver").newInstance();

# Connections in JDBC

We interact with a data source through sessions. Each connection identifies a logical session.

- JDBC URL:
  jdbc:<subprotocol>:<otherParameters>

Example:

```
String url= "jdbc:mysql://localhost/danl_4”;
Connection con;
try{
    con = DriverManager.getConnection(url, username, password);
} catch (SQLException e) {
                e.printStackTrace();}
```

# Executing SQL Statements

- Three different ways of executing SQL statements:
  - Statement (both static and dynamic SQL statements)
  - PreparedStatement (semi-static SQL statements)
  - CallableStatment (stored procedures)

# Executing SQL Statements (Contd.)

```
query = "select course_id, sec_id from section”+
        “where year = 2010 and semester  = \‘Fall\’;";

resultSet = statement.executeQuery(query);
```

# ResultSets

```
while (resultSet.next()) {
        // It is possible to get the columns via name
        // also possible to get the columns via the column number
        // which starts at 1
        // e.g. resultSet.getSTring(2);
        String cid = resultSet.getString(”course_id");
        String sid= resultSet.getString(”section_id");
        System.out.println(cid+" "+sid+"\n");
                }
```

# ResultSets (Contd.)

A ResultSet is a very powerful cursor:

- previous(): moves one row back
- absolute(int num): moves to the row with the specified number
- relative (int num): moves forward or backward
- first() and last()

# Matching Java and SQL Data Types

| SQL Type | Java class | ResultSet get method |
|---|---|---|
| BIT | Boolean | getBoolean() |
| CHAR | String | getString() |
| VARCHAR | String | getString() |
| DOUBLE | Double | getDouble() |
| FLOAT | Double | getDouble() |
| INTEGER | Integer | getInt() |
| REAL | Double | getFloat() |
| DATE | java.sql.Date | getDate() |
| TIME | java.sql.Time | getTime() |
| TIMESTAMP | java.sql.TimeStamp | getTimestamp() |

# Examining Database Metadata

DatabaseMetaData object gives information about the database system and the catalog.

```
DatabaseMetaData md = con.getMetaData();
// print information about the driver:
System.out.println(
   “Name:” + md.getDriverName() +
   “version: ” + md.getDriverVersion());
```

# Stored Procedures

- What is a stored procedure:
  - A block of program executed through a single defined SQL routine.
  - Executed in the process space of the server
- Advantages:
  - Improve network performance
  - Can encapsulate application logic
  - Reuse of application logic by different users
  - Avoid tuple-at-a-time return of records through cursors

# Capabilities of Stored Procedure

- **Conditional execution** An IF...THEN...ELSE structure allows a SQL procedure to test a condition and to carry out different operations depending on the result.
- **Looping** A WHILE or FOR loop or similar structure allows a sequence of SQL operations to be performed repeatedly, until some terminating condition is met.
- **Block structure** A sequence of SQL statements can be grouped into a single block and used in other flow-of-control constructs as if the statement block were a single statement.
- **Named variables** A SQL procedure may store a value that it has calculated, retrieved from the database, or derived in some other way into a program variable, and later retrieve the stored value for use in subsequent calculations.
- **Named procedures** A sequence of SQL statements may be grouped together, given a name, and assigned formal input and output parameters, like a subroutine or function in a conventional programming language. Once defined in this way, the procedure may be called by name, passing it appropriate values for its input parameters.

```
/* Add a customer procedure */
create procedure add_cust (
   c_name   in varchar2,          /* input customer name */
   c_num    in integer,           /* input customer number */
   cred_lim in number,            /* input credit limit */
   tgt_sls  in number,            /* input target sales */
   c_rep    in integer,           /* input salesrep emp # */
   c_offc   in varchar2)          /* input office city */
as
begin
   /* Insert new row of CUSTOMERS table */
   insert into customers (cust_num, company, cust_rep, credit_limit)
          values (c_num, c_name, c_rep, cred_lim);

   /* Update row of SALESREPS table */
   update salesreps
      set quota = quota + tgt_sls
    where empl_num = c_rep;

   /* Update row of OFFICES table */
   update offices
      set target = target + tgt_sls
    where city = c_offc;

   /* Commit transaction and we are done */
   commit;
end;
```

# Calling Stored Procedures

- Once defined by the CREATE PROCEDURE statement, a stored procedure can be used.
- An application program may request execution of the stored procedure, using the appropriate SQL statement.
- Another stored procedure may call it to perform a specific function.
- The stored procedure may also be invoked through an interactive SQL interface.
- The various SQL dialects differ in the specific syntax used to call a stored procedure.
  - Here is a call to the ADD_CUST procedure in the PL/SQL dialect:

```
EXECUTE ADD_CUST('XYZ Corporation', 2137, 30000.00, 50000.00, 103,
                 'Chicago');
```

# MySQL Stored Procedure Examples

```
DELIMITER //
CREATE PROCEDURE GetAllProducts()
BEGIN
SELECT * FROM products;
END //
DELIMITER ;

CALL GetAllProducts();
```

# MySQL Stored Procedure Examples

```
DELIMITER //
CREATE PROCEDURE GetOfficeByCountry(IN countryName VARCHAR(255))
BEGIN
SELECT city, phone
FROM offices
WHERE country = countryName;
END //
DELIMITER ;
```

```
CALL GetOfficeByCountry('USA')
```

# MySQL Stored Procedure Examples

```
DELIMITER $$
CREATE PROCEDURE CountOrderByStatus(
IN orderStatus VARCHAR(25),
OUT total INT)
BEGIN
SELECT count(orderNumber)
INTO total
FROM orders
WHERE status = orderStatus;
END$$
DELIMITER ;

CALL CountOrderByStatus('Shipped',@total);
SELECT @total AS total_shipped;
```

# MySQL Stored Procedure Examples

```
DELIMITER $$
CREATE PROCEDURE `Capitalize`(INOUT str VARCHAR(1024))
BEGIN
DECLARE i INT DEFAULT 1;
DECLARE myc, pc CHAR(1);
DECLARE outstr VARCHAR(1000) DEFAULT str;
WHILE i <= CHAR_LENGTH(str) DO
SET myc = SUBSTRING(str, i, 1);
SET pc = CASE WHEN i = 1 THEN ' '
ELSE SUBSTRING(str, i - 1, 1)
END;
IF pc IN (' ', '&', '"', '_', '?', ';', ':', '!', ',', '-', '/', '(', '.') THEN
SET outstr = INSERT(outstr, i, 1, UPPER(myc));
END IF;
SET i = i + 1;
END WHILE;
SET str = outstr;
END$$
DELIMITER ;
```

```
SET @str = 'mysql stored procedure tutorial';
CALL Capitalize(@str);
SELECT @str;
```

| @str |
| --- |
| Mysql Stored Procedure Tutorial |

# Using CallableStatements to Execute Stored Procedures in Java

```
CREATE PROCEDURE demoSp(IN inputParam VARCHAR(255), \
                                        INOUT inOutParam INT)
BEGIN
    DECLARE z INT;
    SET z = inOutParam + 1;
    SET inOutParam = z;

    SELECT inputParam;

    SELECT CONCAT('zyxw', inputParam);
END
```

# Using CallableStatements to Execute Stored Procedures in Java

```
import java.sql.CallableStatement;

...

    //
    // Prepare a call to the stored procedure 'demoSp'
    // with two parameters
    //
    // Notice the use of JDBC-escape syntax ({call ...})
    //

    CallableStatement cStmt = conn.prepareCall("{call demoSp(?, ?)}");



    cStmt.setString(1, "abcdefg");
```

# Using CallableStatements to Execute Stored Procedures in Java

```
import java.sql.Types;
```

Register output parameters in two ways:

```
//
// Registers the second parameter as output, and
// uses the type 'INTEGER' for values returned from
// getObject()
//

cStmt.registerOutParameter(2, Types.INTEGER);

//
// Registers the named parameter 'inOutParam', and
// uses the type 'INTEGER' for values returned from
// getObject()
//

cStmt.registerOutParameter("inOutParam", Types.INTEGER);
```

# Using CallableStatements to Execute Stored Procedures in Java

Now it's ready to execute the stored procedure.

```
boolean hadResults = cStmt.execute();
```

Get the output in two ways:

```
int outputValue = cStmt.getInt(2); // index-based

outputValue = cStmt.getInt("inOutParam"); // name-based
```

# Functions

- In addition to stored procedures, most SPL dialects support a stored *function* capability.
- The distinction is that a function returns a single thing (such as a data value, an object, or an XML document) each time it is invoked, while a stored procedure can return many things or nothing at all.
  - Support for returned values varies by SPL dialect.
- Functions are commonly used as column expressions in SELECT statements, and thus are invoked once per row in the result set, allowing the function to perform calculations, data conversion, and other processes to produce the returned value for the column.

# Function Example

```
/* Return total order amount for a customer */
create function get_tot_ords(c_num in number)
                return number
as

/* Declare one local variable to hold the total */
tot_ord number(16,2);

begin
   /* Simple single-row query to get total */
   select sum(amount) into tot_ord
     from orders
    where cust = c_num;

   /* return the retrieved value as fcn value */
   return tot_ord;
end;
```

# Call a Function

```
SELECT COMPANY, NAME
  FROM CUSTOMERS, SALESREPS
 WHERE CUST_REP = EMPL_NUM
   AND GET_TOT_ORDS(CUST_NUM) > 10000.00;
```

# Triggers

- A trigger is a special set of stored procedural code whose activation is caused by modifications to the database contents.
- Unlike stored procedures, a trigger is not activated by a CALL or EXECUTE statement. Instead, the trigger is associated with a database table.
- Some DBMS brands allow definition of specific updates that cause a trigger to fire.
- Also, some DBMS brands, notably Oracle, allow triggers to be based on system events such as users connecting to the database or execution of a database shutdown command.

# Trigger Example

```
Create or replace trigger upd_tgt
    /* Insert trigger for SALESREPS */
    before insert on salesreps
    for each row
    begin
        if :new.quota is not null
         then
           update offices
               set target = target + new.quota;
         end if;
    end;
```

# Summary

- Embedded SQL allows execution of parameterized static queries within a host language
- Dynamic SQL allows execution of completely ad-hoc queries within a host language
- Cursor mechanism allows retrieval of one record at a time and bridges impedance mismatch between host language and SQL
- APIs such as JDBC introduce a layer of abstraction between application and DBMS
- Stored procedures execute application logic directly at the server