

# Using SQL in an Application

CSE 305 – Principles of Database Systems

Paul Fodor

Stony Brook University

<http://www.cs.stonybrook.edu/~cse305>

# Interactive vs. Non-Interactive SQL

- *Interactive SQL*: SQL statements input from terminal; DBMS outputs to screen
  - Inadequate for most uses
    - It may be necessary to process the data before output
    - Amount of data returned not known in advance
    - SQL has very limited expressive power (not Turing-complete)
- *Non-interactive SQL*: SQL statements are included in an application program written in a host language, like C, Java, COBOL

# Application Program

- *Host language*: A conventional language (*e.g.*, C, Java) that supplies control structures, computational capabilities, interaction with physical devices
- *SQL*: supplies ability to interact with database.
- *Using the facilities of both*: the application program can act as an intermediary between the user at a terminal and the DBMS

# Preparation

- Before an SQL statement is executed, it must be *prepared* by the DBMS:
  - What indices can be used?
  - In what order should tables be accessed?
  - What constraints should be checked?
- Decisions are based on schema, table sizes, etc.
- Result is a *query execution plan*
- Preparation is a complex activity, usually done at run time, justified by the complexity of query processing

# Introducing SQL Into the Application

- SQL statements can be incorporated into an application program in two different ways:
  - *Statement Level Interface* (SLI): Application program is a mixture of host language statements and SQL statements and directives
  - *Call Level Interface* (CLI): Application program is written entirely in host language
    - SQL statements are values of string variables that are passed as arguments to host language (library) procedures

# Statement Level Interface

- SQL statements and directives in the application have a *special syntax* that sets them off from host language constructs
  - e.g., EXEC SQL *SQL\_statement*
- *Precompiler* scans program and translates SQL statements into calls to host language library procedures that communicate with DBMS
- *Host language compiler* then compiles program

# Statement Level Interface

- SQL constructs in an application take two forms:
  - Standard SQL statements (*static* or *embedded* SQL): Useful when SQL portion of program is known at compile time
  - Directives (*dynamic* SQL): Useful when SQL portion of program not known at compile time. Application constructs SQL statements *at run time* as values of host language variables that are manipulated by directives
- Precompiler translates statements and directives into arguments of calls to library procedures.

# Call Level Interface

- Application program written entirely in host language (no precompiler)
  - Examples: JDBC, ODBC
- SQL statements are values of string variables constructed *at run time* using host language
  - Similar to dynamic SQL
- Application uses string variables as arguments of library routines that communicate with DBMS
  - e.g. `executeQuery("SQL query statement")`



# Static SQL

```
EXEC SQL BEGIN DECLARE SECTION;  
  unsigned long num_enrolled;  
  char crs_code;  
  char SQLSTATE [6];  
EXEC SQL END DECLARE SECTION;
```

*Variables  
shared by host  
and SQL*

```
.....  
EXEC SQL SELECT C.NumEnrolled  
  INTO :num_enrolled  
  FROM Course C  
  WHERE C.CrsCode = :crs_code;
```

*“:” used to set off  
host variables*

- Declaration section for host/SQL communication
- Colon convention for value (WHERE) and result (INTO) parameters

# Status

```
EXEC SQL SELECT C.NumEnrolled  
    INTO :num_enrolled  
    FROM Course C  
    WHERE C.CrsCode = :crs_code;  
if ( !strcmp (SQLSTATE, "00000") ) {  
    printf ( "statement failed" )  
};
```

*Out parameter*

*In parameter*

# Connections

- To connect to an SQL database, use a connect statement

```
CONNECT TO database_name AS connection_name  
USING user_id
```

# Transactions

- No explicit statement is needed to begin a transaction
  - A transaction is initiated when the first SQL statement that accesses the database is executed
- The mode of transaction execution can be set with  
`SET TRANSACTION READ ONLY`  
`ISOLATION LEVEL SERIALIZABLE`
- Transactions are terminated with `COMMIT` or `ROLLBACK` statements

# Example: Course Deregistration

```
EXEC SQL CONNECT TO :dbserver;
if ( ! strcmp (SQLSTATE, "00000") ) exit (1);

.....
EXEC SQL DELETE FROM Transcript T
  WHERE T.StudId = :studid AND T.Semester = 'S2000'
        AND T.CrsCode = :crscode;
if (! strcmp (SQLSTATE, "00000") ) EXEC SQL ROLLBACK;
else {
  EXEC SQL UPDATE Course C
    SET C.Numenrolled = C.Numenrolled - 1
    WHERE C.CrsCode = :crscode;
  if (! strcmp (SQLSTATE, "00000") ) EXEC SQL ROLLBACK;
  else EXEC SQL COMMIT;
}
```

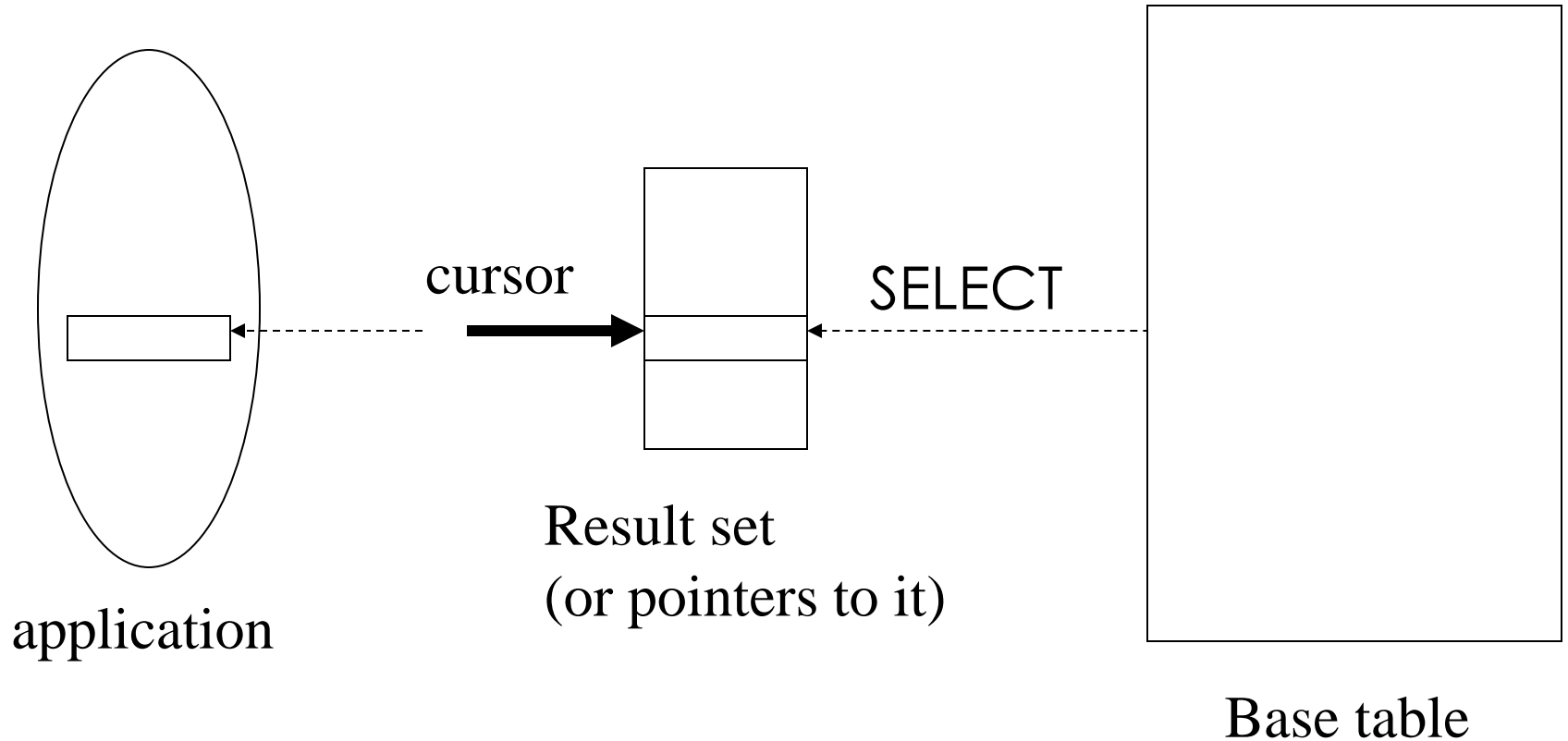
# Buffer Mismatch Problem

- **Problem:** SQL deals with tables (of arbitrary size); host language program deals with fixed size buffers
  - How is the application to allocate storage for the result of a `SELECT` statement?
- **Solution:** Fetch a single row at a time
  - Space for a single row (number and type of *out* parameters) can be determined from schema and allocated in application

# Cursors

- *Result set* — set of rows produced by a `SELECT` statement
- *Cursor* — pointer to a row in the result set.
- Cursor operations:
  - *Declaration*
  - *Open* — execute `SELECT` to determine result set and initialize pointer
  - *Fetch* — advance pointer and retrieve next row
  - *Close* — deallocate cursor

# Cursors





# Cursors

```
EXEC SQL DECLARE GetEnroll INSENSITIVE CURSOR FOR
  SELECT T.StudId, T.Grade      --cursor is not a schema element
  FROM Transcript T
  WHERE T.CrsCode = :crscode AND T.Semester = 'S2000';
```

.....

```
EXEC SQL OPEN GetEnroll;
if ( !strcmp ( SQLSTATE, "00000" ) ) { ... fail exit... };
```

.....

```
EXEC SQL FETCH GetEnroll INTO :studid, :grade;
while ( SQLSTATE = "00000" ) {
  ... process the returned row...
  EXEC SQL FETCH GetEnroll INTO :studid, :grade;
}
```

```
if ( !strcmp ( SQLSTATE, "02000" ) ) { ... fail exit... };
```

.....

```
EXEC SQL CLOSE GetEnroll;
```

*Reference resolved at  
compile time,  
Value substituted at  
OPEN time*

# Cursor types

- *Insensitive cursor*: Result set (effectively) computed and stored in a separate table at OPEN time
  - Changes made to base table subsequent to OPEN (by any transaction) do not affect result set
  - Cursor is read-only
- *Cursors that are not insensitive*: Specification not part of SQL standard
  - Changes made to base table subsequent to OPEN (by any transaction) can affect result set
  - Cursor is updatable

# Inensitive Cursor

*Changes made after opening cursor not seen in the cursor*

cursor

key1	t t t t t t t t
key3	yyyyyyyyyy
key4	zzzzzzzzzz

*Result Set*

key1	t t t t qqt t t t
key2	xxxxxxxxxx
key3	yyyrryyyyy
key4	zzzzzzzzzz
key5	uuuuuuuuuu
key6	vvvvvvvvvv

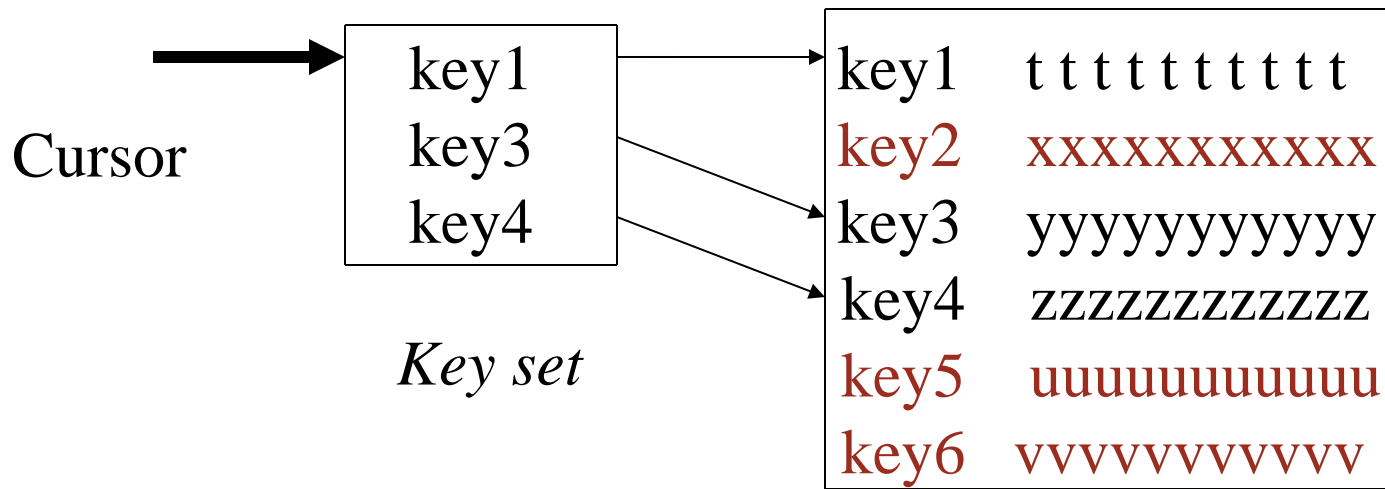
*Base Table*

*Tuples added after opening the cursor*

# Keyset-Driven Cursor

- Example of a cursor that is not insensitive
- Primary key of each row in result set is computed at open time
- **UPDATE** or **DELETE** of a row in base table by a concurrent transaction between **OPEN** and **FETCH** might be seen through cursor
- **INSERT** into base table, however, not seen through cursor
- Cursor is updatable

# Keyset-Driven Cursor



Tuples added after cursor is open are not seen, but updates to key1, key3, key4 are seen in the cursor.

# Cursors

```
DECLARE cursor-name [INSENSITIVE] [SCROLL]
  CURSOR FOR table-expr
  [ ORDER BY column-list ]
  [ FOR {READ ONLY | UPDATE [ OF column-list ] } ]
```

For updatable (not insensitive, not read-only) cursors

```
UPDATE table-name                --base table
  SET assignment
  WHERE CURRENT OF cursor-name
```

```
DELETE FROM table-name          --base table
  WHERE CURRENT OF cursor-name
```

Restriction – *table-expr* must satisfy restrictions of updatable view

# Scrolling

- If SCROLL option not specified in cursor declaration, FETCH always moves cursor forward one position
- If SCROLL option is included in DECLARE CURSOR section, cursor can be moved in arbitrary ways around result set:

*Get previous tuple*

FETCH **PRIOR** FROM GetEnroll INTO :studid, :grade;

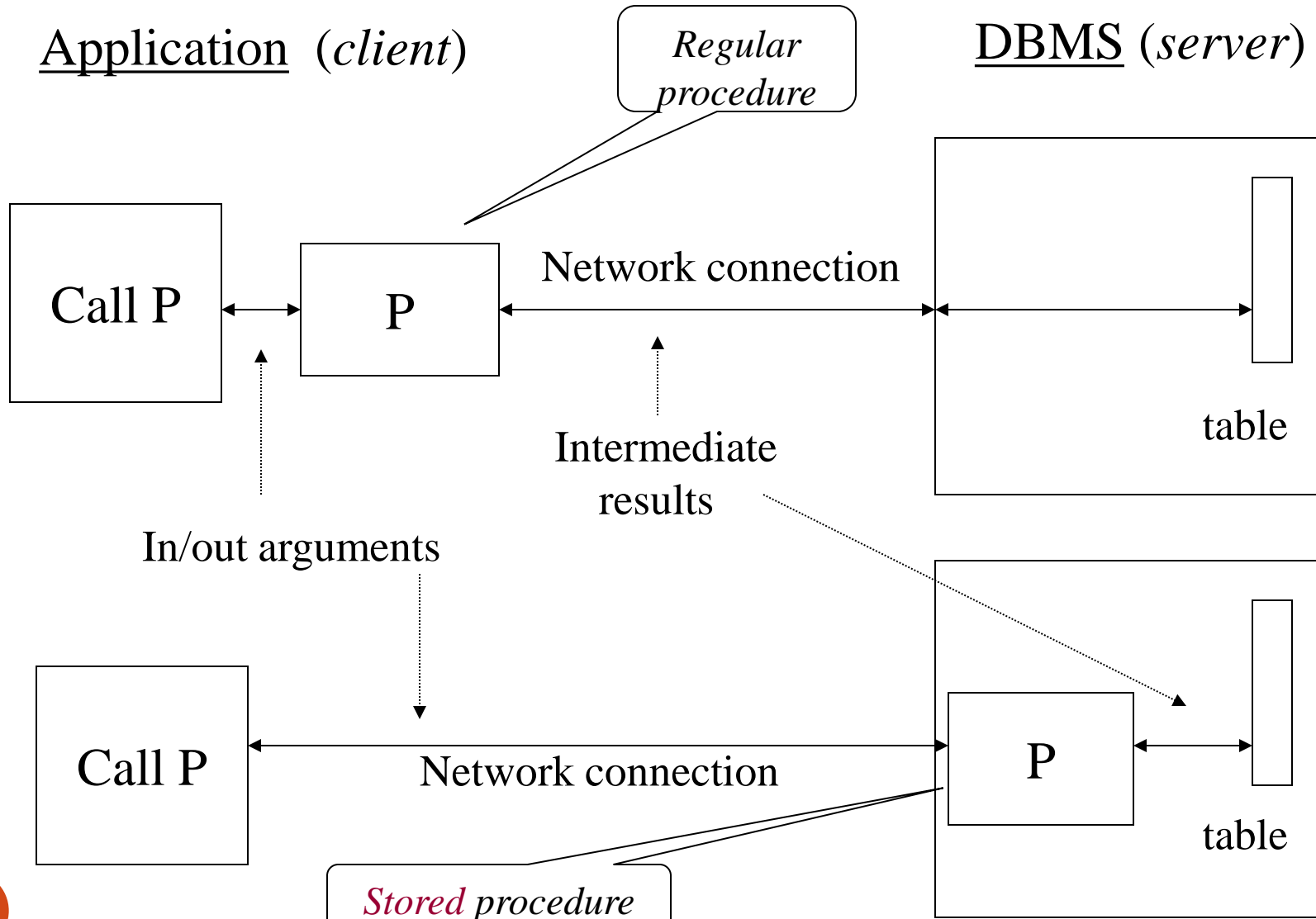
- Also: FIRST, LAST, ABSOLUTE n, RELATIVE n

# Stored Procedures

- Procedure – written in a conventional algorithmic language
  - Included as schema element (stored in DBMS)
  - Invoked by the application
- Advantages:
  - Intermediate data need not be communicated to application (time and cost savings)
  - Procedure's SQL statements prepared in advance
  - Authorization can be done at procedure level
  - Added security since procedure resides in server
  - Applications that call the procedure need not know the details of database schema – all database access is encapsulated within the procedure



# Stored Procedures



# Stored Procedures

*Schema:*

```
CREATE PROCEDURE Register (char :par1, char :par2)
AS BEGIN
    EXEC SQL SELECT ..... ;
    IF ( ..... ) THEN ..... -- SQL embedded in
        ELSE ....           -- Persistent Stored Modules
                                -- (PSM) language
END
```

*Application:*

```
EXEC SQL EXECUTE PROCEDURE Register ( :crscode, :studid);
```

# Integrity Constraint Checking

- Transaction moves database from an initial to a final state, both of which satisfy all integrity constraints but:
  - Constraints might not be true of intermediate states
  - Constraint checks at statement boundaries might be inappropriate
- SQL (optionally) allows checking to be deferred to transaction **COMMIT**

# Deferred Constraint Checking

*Schema:*

```
CREATE ASSERTION NumberEnrolled  
CHECK ( .....)  
DEFERRABLE;
```

*Application:*

```
SET CONSTRAINT NumberEnrolled DEFERRED;
```

Transaction is aborted if constraint is false at commit time

# Dynamic SQL

- **Problem:** Application might not know in advance:
  - The SQL statement to be executed
  - The database schema to which the statement is directed
- **Example:** User inputs database name and SQL statement interactively from terminal
- In general, application constructs (as the value of a host language string variable) the SQL statement at run time
- Preparation (necessarily) done at run time

# Dynamic SQL

- SQL-92 defines syntax for embedding directives into application for constructing, preparing, and executing an SQL statement
  - Referred to as *Dynamic SQL*
  - Statement level interface
- Dynamic and static SQL can be mixed in a single application

# Dynamic SQL

```
strcpy (tmp, "SELECT C.NumEnrolled FROM Course C \
        WHERE C.CrsCode = ?");
EXEC SQL PREPARE st FROM :tmp;
EXEC SQL EXECUTE st INTO :num_enrolled USING :crs_code;
```

placeholder

- `st` is an SQL variable; names the SQL statement
- `tmp`, `crscode`, `num_enrolled` are host language variables (note colon notation)
- `crscode` is an *in* parameter; supplies value for placeholder (?)
- `num_enrolled` is an *out* parameter; receives value from `C.NumEnrolled`

# Dynamic SQL

- **PREPARE** names SQL statement **st** and sends it to DBMS for preparation
- **EXECUTE** causes the statement named **st** to be executed



# Parameters: Static vs Dynamic SQL

- *Static SQL*:
  - Names of (host language) parameters are contained in SQL statement and available to precompiler
  - Address and type information in symbol table
  - Routines for fetching and storing argument values can be generated
  - Complete statement (with parameter values) sent to DBMS when statement is executed

```
EXEC SQL SELECT C.NumEnrolled
      INTO   :num_enrolled
      FROM   Course C
      WHERE  C.CrsCode = :crs_code;
```

# Parameters: Static vs Dynamic SQL

- *Dynamic SQL*: SQL statement constructed at run time when symbol table is no longer present
- Case 1: Parameters are known at compile time

```
strcpy (tmp, "SELECT C.NumEnrolled FROM Course C \  
        WHERE C.CrsCode = ?" );  
EXEC SQL PREPARE st FROM :tmp;
```

- Parameters are named in EXECUTE statement: *in* parameters in USING; *out* parameters in INTO clauses

```
EXEC SQL EXECUTE st INTO :num_enrolled USING :crs_code;
```

- EXECUTE statement is compiled using symbol table
  - *fetch()* and *store()* routines generated

# Parameters – Dynamic SQL

(Case 1: parameters known at compile time)

- Fetch and store routines are executed at client when `EXECUTE` is executed to communicate argument values with DBMS
- `EXECUTE` can be invoked multiple times with different values of *in* parameters
  - Each invocation uses same query execution plan
- Values substituted for placeholders by DBMS (in order) at invocation time and statement is executed

# Parameters in Dynamic SQL

(parameters supplied at runtime)

- Case 2: Parameters *not* known at compile time
- *Example*: Statement input from terminal
  - Application cannot parse statement and might not know schema, so it does not have any parameter information
- EXECUTE statement cannot name parameters in INTO and USING clauses

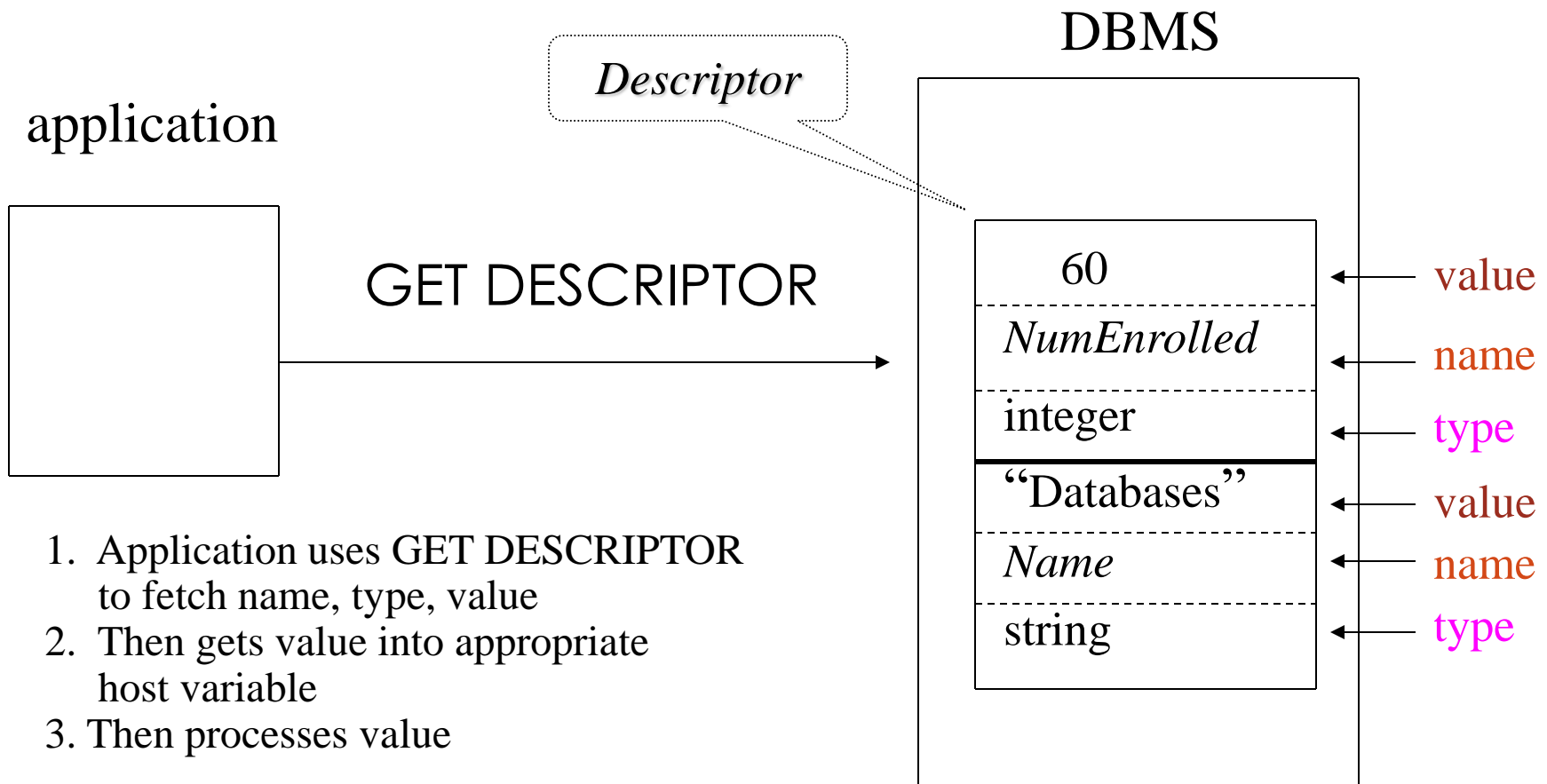
# Parameters in Dynamic SQL (cont'd)

## (Case 2: parameters supplied at runtime)

- DBMS determines number and type of parameters after preparing the statement
- Information stored by DBMS in a *descriptor* – a data structure inside the DBMS, which records the *name*, *type*, and *value* of each parameter
- Dynamic SQL provides directive **GET DESCRIPTOR** to get information about parameters (e.g., number, name, type) from DBMS and to fetch value of *out* parameters
- Dynamic SQL provides directive **SET DESCRIPTOR** to supply value to *in* parameters

# Descriptors

```
temp = "SELECT C.NumEnrolled, C.Name FROM Course C \
WHERE C.CrsCode = 'CS305'"
```



# Dynamic SQL Calls when Descriptors are Used

```
... .. construct SQL statement in temp ... ..  
EXEC SQL PREPARE st FROM :temp;           // prepare statement  
  
EXEC SQL ALLOCATE DESCRIPTOR 'desc'; // create descriptor  
EXEC SQL DESCRIBE OUTPUT st USING  
        SQL DESCRIPTOR 'desc';           // populate desc with info  
                                           // about out parameters  
  
EXEC SQL EXECUTE st INTO                 // execute statement and  
        SQL DESCRIPTOR AREA 'desc'; // store out values in desc  
  
EXEC SQL GET DESCRIPTOR 'desc' ...; // get out values  
  
... .. similar strategy is used for in parameters ... ..
```

# Example: Nothing Known at Compile Time

```
sprintf(my_sql_stmt,  
        "SELECT * FROM %s WHERE COUNT(*) = 1",  
        table); // table – host var; even the table is known only at run time!
```

```
EXEC SQL PREPARE st FROM :my_sql_stmt;  
EXEC SQL ALLOCATE DESCRIPTOR 'st_output';
```

```
EXEC SQL DESCRIBE OUTPUT st USING SQL DESCRIPTOR 'st_output'
```

- The SQL statement to execute is known only at run time
- At this point DBMS knows what the exact statement is (including the table name, the number of *out* parameters, their types)
- The above statement asks to create descriptors in *st\_output* for all the (now known) *out* parameters

```
EXEC SQL EXECUTE st INTO SQL DESCRIPTOR 'st_output';
```



# Example: Getting Meta-Information from a Descriptor

```
// Host var colcount gets the number of out parameters in the SQL statement
```

```
// described by st_output
```

```
EXEC SQL GET DESCRIPTOR 'st_output' :colcount = COUNT;
```

```
// Set host vars coltype, collength, colname with the type, length, and name of the
```

```
// colnumber's out parameter in the SQL statement described by st_output
```

```
EXEC SQL GET DESCRIPTOR 'st_output' VALUE :colnumber;
```

```
:coltype = TYPE, // predefined integer constants, such as SQL_CHAR, SQL_FLOAT,...
```

```
:collength = LENGTH,
```

```
:colname = NAME;
```

# Example: Using Meta-Information to Extract Attribute Value

```
char strdata[1024];
```

```
int intdata;
```

```
... ..
```

```
switch (coltype) {
```

```
case SQL_CHAR:
```

```
    EXEC SQL GET DESCRIPTOR 'st_output' VALUE :colnumber :strdata=DATA;
```

```
    break;
```

```
case SQL_INT:
```

```
    EXEC SQL GET DESCRIPTOR 'st_output' VALUE :colnumber :intdata=DATA;
```

```
    break;
```

```
case SQL_FLOAT:
```

```
    ... ..
```

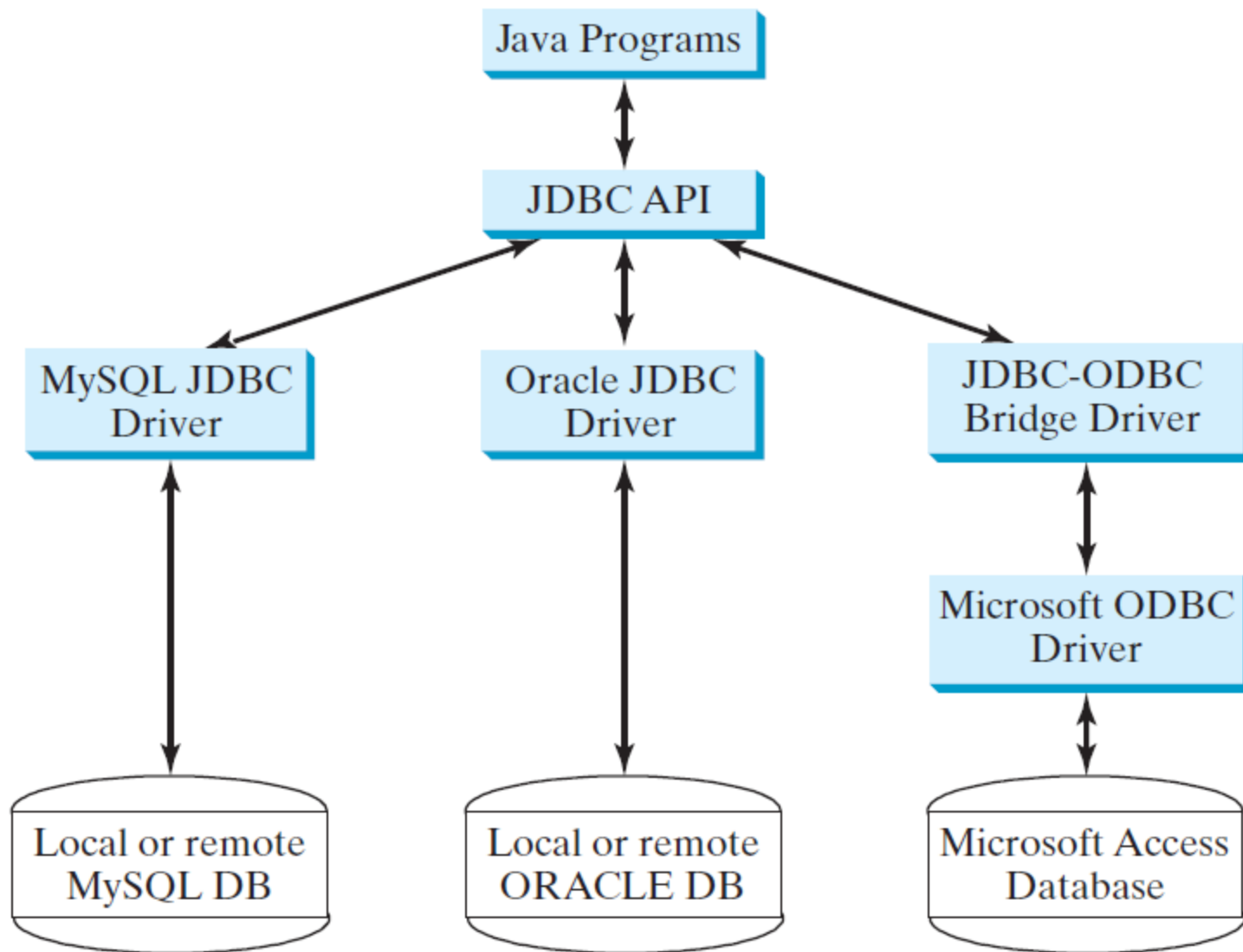
```
}
```

*Put the value of attribute  
colnumber into the  
variable strdata*

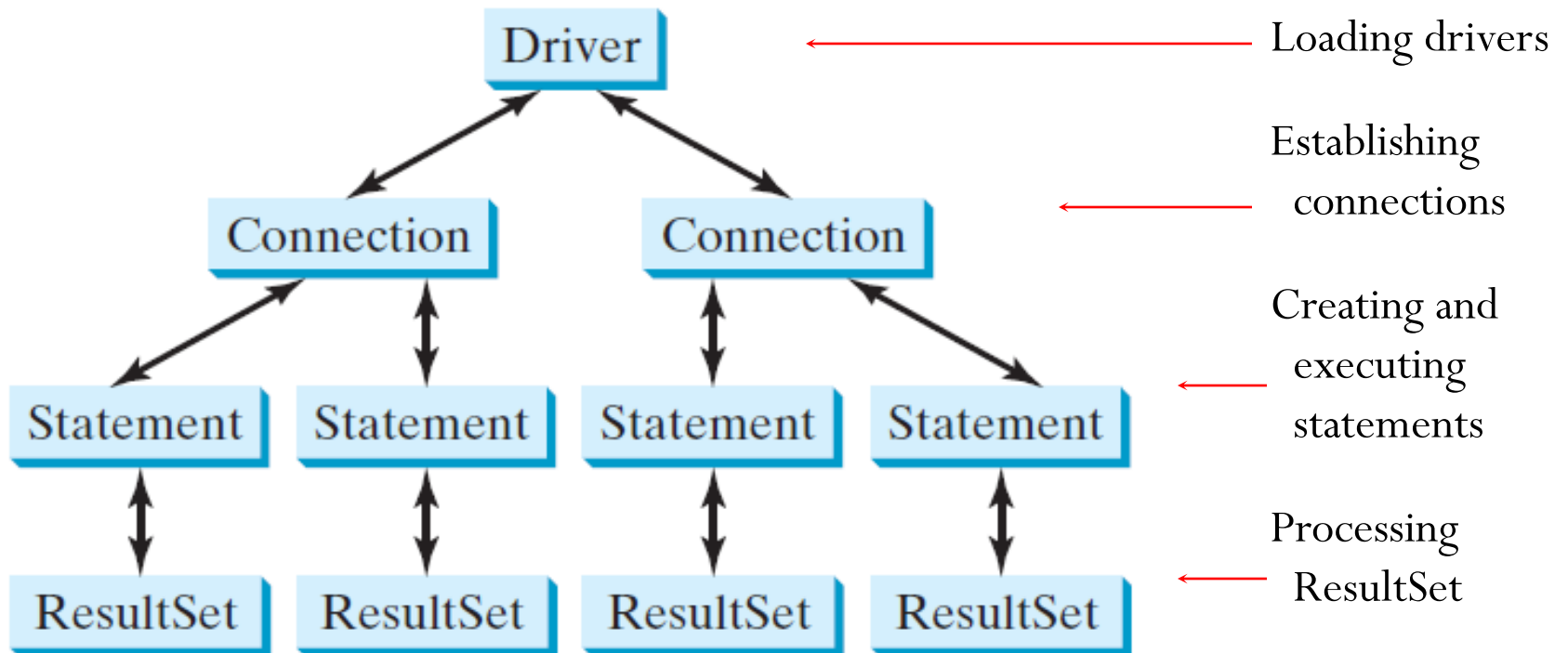
# JDBC

- Call-level interface (CLI) for executing SQL from a Java program
- SQL statement is constructed at run time as the value of a Java variable (as in dynamic SQL)
- JDBC passes SQL statements to the underlying DBMS. Can be interfaced to any DBMS that has a JDBC driver
- Part of SQL:2003

# JDBC



# JDBC



# Developing JDBC Programs

Loading  
drivers

Establishing  
connections

Creating and  
executing  
statements

Processing  
ResultSet

Statement to load a driver:

```
Class.forName("JDBCDriverClass");
```

A driver is a class. For example:

Database	Driver Class	Source
Access	sun.jdbc.odbc.JdbcOdbcDriver	Already in JDK
MySQL	com.mysql.jdbc.Driver	Website
Oracle	oracle.jdbc.driver.OracleDriver	Website

The JDBC-ODBC driver for Access is bundled in JDK.

MySQL driver class is in mysqljdbc.jar

Oracle driver class is in classes12.jar

To use the MySQL and Oracle drivers, you have to add mysqljdbc.jar and classes12.jar in the classpath using the following DOS command on Windows:

```
classpath=%classpath%;c:\mysqljdbc.jar;c:\classes.jar
```

# Developing JDBC Programs

Loading drivers

Establishing connections

Creating and executing statements

Processing ResultSet

```
Connection connection = DriverManager.getConnection(databaseURL);
```

Database	URL Pattern
----------	-------------

Access	<code>jdbc:odbc:dataSource</code>
--------	-----------------------------------

MySQL	<code>jdbc:mysql://hostname/dbname</code>
-------	---

Oracle	<code>jdbc:oracle:thin:@hostname:port#:oracleDBSID</code>
--------	---

Examples:

For Access:

```
Connection connection = DriverManager.getConnection  
("jdbc:odbc:ExampleMDBDataSource");
```

For MySQL:

```
Connection connection = DriverManager.getConnection  
("jdbc:mysql://localhost/test");
```

For Oracle:

```
Connection connection = DriverManager.getConnection  
("jdbc:oracle:thin:@localhost:1521:orcl", "paul", "tiger");
```

# Developing JDBC Programs

Loading drivers

Establishing  
connections

Creating and  
executing  
statements

Processing  
ResultSet

**Creating statement:**

```
Statement statement = connection.createStatement();
```

**Executing statement (for update, delete, insert):**

```
statement.executeUpdate  
("create table Temp (col1 char(5), col2 char(5))");
```

**Executing statement (for select):**

```
// Select the columns from the Student table  
ResultSet resultSet = statement.executeQuery  
("select firstName, mi, lastName from Student where lastName "  
+ " = 'Smith'");
```



# Developing JDBC Programs

Loading  
drivers

Establishing  
connections

Creating and  
executing  
statements

Processing  
ResultSet

**Executing statement (for select):**

```
// Select the columns from the Student table
```

```
ResultSet resultSet = stmt.executeQuery
```

```
("select firstName, mi, lastName from Student where lastName " +  
" = 'Smith');"
```

**Processing ResultSet (for select):**

```
// Iterate through the result and print the student names
```

```
while (resultSet.next())
```

```
System.out.println(resultSet.getString(1) + " " + resultSet.getString(2)  
+ ". " + resultSet.getString(3));
```

# Complete JDBC Example

```
import java.sql.*;
public class SimpleJdbc {
    public static void main(String[] args)
        throws SQLException, ClassNotFoundException {
        // Load the JDBC driver
        Class.forName("com.mysql.jdbc.Driver");
        System.out.println("Driver loaded");

        // Establish a connection
        Connection connection = DriverManager.getConnection
            ("jdbc:mysql://localhost/test");
        System.out.println("Database connected");

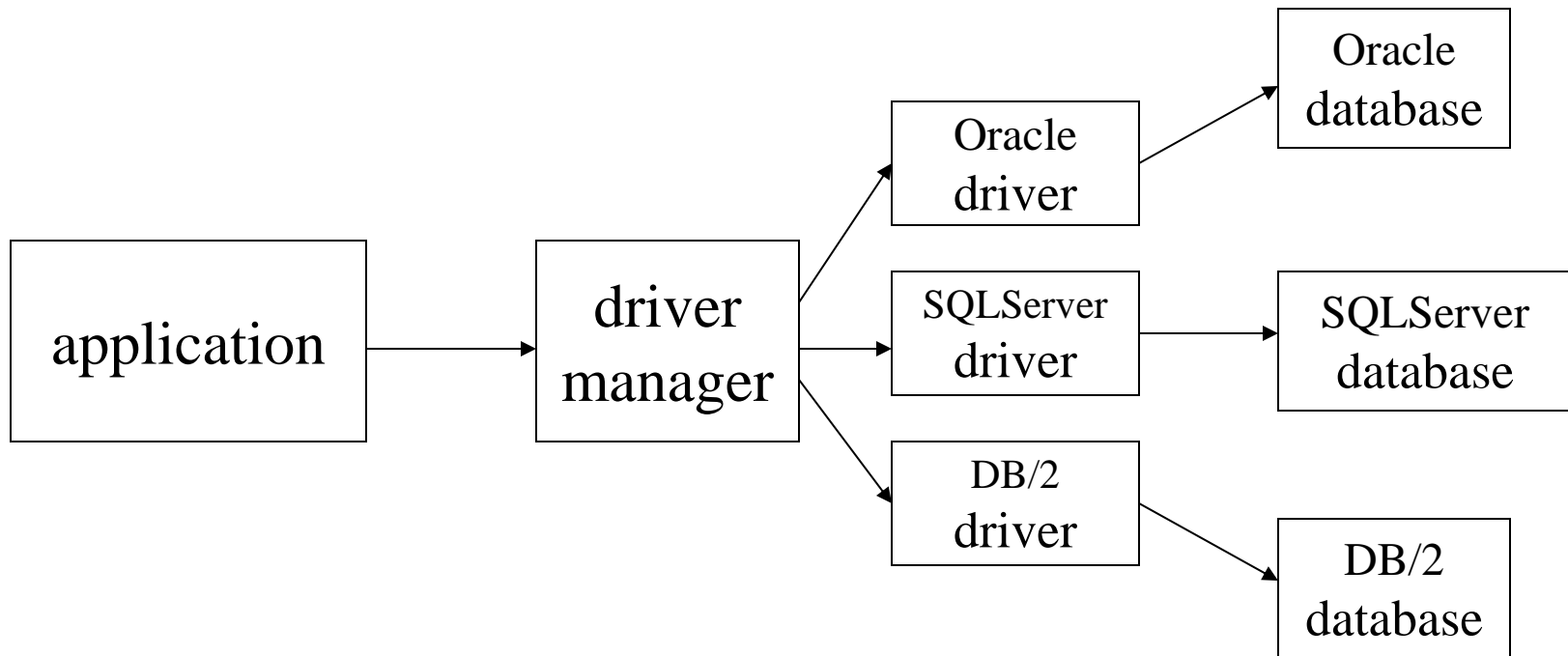
        // Create a statement
        Statement statement = connection.createStatement();

        // Execute a statement
        ResultSet resultSet = statement.executeQuery
            ("select firstName, mi, lastName from Student where lastName "
            + " = 'Smith'");

        // Iterate through the result and print the student names
        while (resultSet.next())
            System.out.println(resultSet.getString(1) + "\t" +
                resultSet.getString(2) + "\t" + resultSet.getString(3));

        // Close the connection
        connection.close();
    }
}
```

# JDBC Run-Time Architecture



# Summary: Executing a Query

```
import java.sql.*;    -- import all classes in package java.sql
```

```
Class.forName(driver name);    // static method of class Class  
                                // loads specified driver
```

```
Connection con = DriverManager.getConnection(Url, Id, Passwd);
```

- *Static method of class DriverManager; attempts to connect to DBMS*
- *If successful, creates a connection object, con, for managing the connection*

```
Statement stat = con.createStatement ();
```

- *Creates a statement object stat*
- *Statements have executeQuery() method*

# Summary: Executing a Query

```
String query = "SELECT T.StudId FROM Transcript T" +  
              "WHERE T.CrsCode = 'cse305' " +  
              "AND T.Semester = 'S2000' ";
```

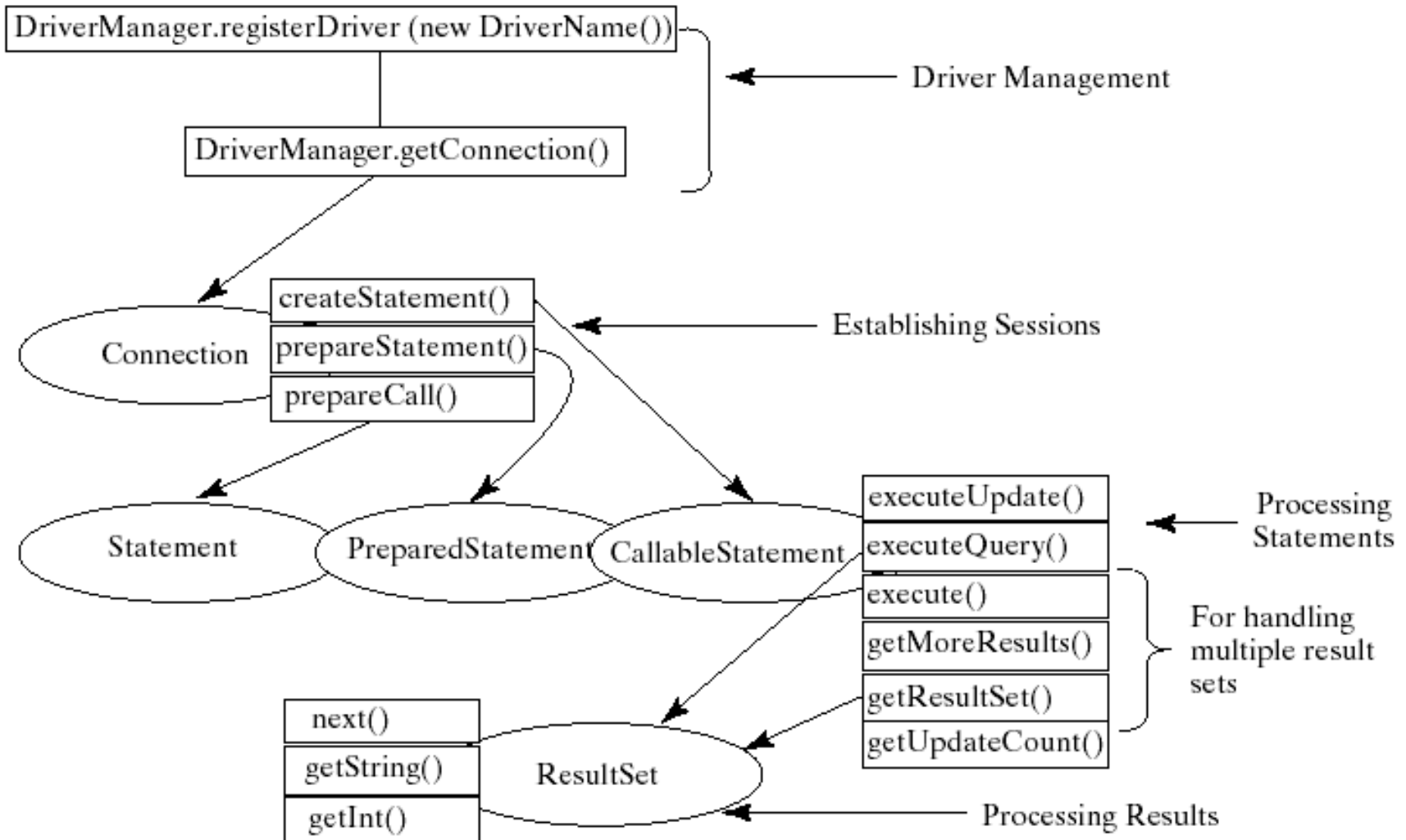
```
ResultSet res = stat.executeQuery(query);
```

- *Creates a result set object, res.*
- *Prepares and executes the query.*
- *Stores the result set produced by execution in res (analogous to opening a cursor).*
- *The query string can be constructed at run time (as above).*
- *The input parameters are plugged into the query when the string is formed (as above)*

# Processing Statements

- Once a connection to a particular database is established, it can be used to send SQL statements from your program to the database.
- JDBC provides the Statement, **PreparedStatement**, and **CallableStatement** interfaces to facilitate sending statements to a database for execution and receiving execution results from the database.

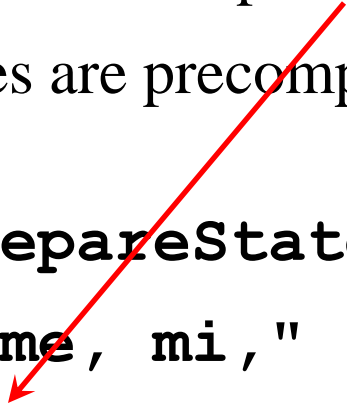
# Processing Statements Diagram



# Preparing and Executing a Query

- The PreparedStatement interface is designed to execute dynamic SQL statements and SQL-stored procedures with IN parameters.
- These SQL statements and stored procedures are precompiled for efficient use when repeatedly executed.

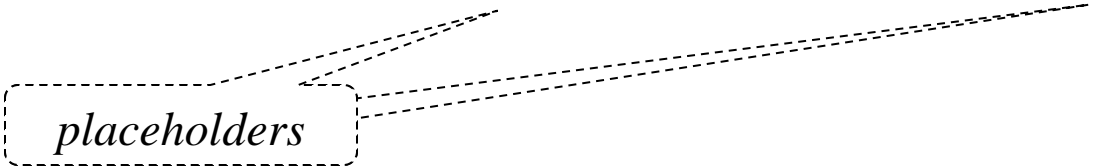
```
Statement pstmt = connection.prepareStatement  
("insert into Student(firstName, mi,"  
+ " lastName) values (?, ?, ?)");
```





# Preparing and Executing a Query

String query = “SELECT T.StudId FROM Transcript T” +  
“WHERE T.CrsCode = ? AND T.Semester = ?”;



*placeholders*

**PreparedStatement** ps = con.**prepareStatement** ( query );

- *Prepares the statement*
- *Creates a prepared statement object, ps, containing the prepared statement*
- *Placeholders (?) mark positions of in parameters; special API is provided to plug the actual values in positions indicated by the ?'s*

## Preparing and Executing a Query (cont'd)

```
String crs_code, semester;
```

```
.....
```

```
ps.setString(1, crs_code); // set value of first in parameter
```

```
ps.setString(2, semester); // set value of second in parameter
```

```
ResultSet res = ps.executeQuery ( );
```

- *Creates a result set object, res*
- *Executes the query*
- *Stores the result set produced by execution in res*

```
while ( res.next ( ) ) { // advance the cursor  
    j = res.getInt ( "StudId" ); // fetch output int-value  
    ...process output value...  
}
```

# Processing Statements

- The `execute`, `executeQuery`, and `executeUpdate` Methods
  - The methods for executing SQL statements are `execute`, `executeQuery`, and `executeUpdate`, each of which accepts a string containing a SQL statement as an argument.
  - This string is passed to the database for execution.
  - The `execute` method should be used if the execution produces multiple result sets, multiple update counts, or a combination of result sets and update counts.
  - The `executeQuery` method should be used if the execution produces a single result set, such as the SQL `select` statement.
  - The `executeUpdate` method should be used if the statement results in a single update count or no update count, such as a SQL `INSERT`, `DELETE`, `UPDATE`, or `DDL` statement.

# Result Sets and Cursors

- Three types of result sets in JDBC:
  - *Forward-only*: not scrollable
  - *Scroll-insensitive*: scrollable; changes made to underlying tables after the creation of the result set are not visible through that result set
  - *Scroll-sensitive*: scrollable; updates and deletes made to tuples in the underlying tables after the creation of the result set are visible through the set

# Result Set

```
Statement stat = con.createStatement (
    ResultSet.TYPE_SCROLL_SENSITIVE,
    ResultSet.CONCUR_UPDATABLE );
```

- Any result set type can be declared *read-only* or *updatable*
  - **CONCUR\_UPDATABLE** (assuming SQL query satisfies the conditions for updatable views)
- *Updatable*: Current row of an updatable result set can be changed or deleted, or a new row can be inserted. Any such change causes changes to the underlying database table

```
res.updateString ("Name", "John" ); // change the attribute "Name" of
                                     // current row in the row buffer.
```

```
res.updateRow ( ); // install changes to the current row buffer
                  // in the underlying database table
```

# Handling Exceptions

```
try {  
    ...Java/JDBC code...  
} catch ( SQLException ex ) {  
    ...exception handling code...  
}
```

- try/catch is the basic structure within which an SQL statement should be embedded
- If an exception is thrown, an exception object, *ex*, is created and the catch clause is executed
- The exception object has methods to print an error message, return `SQLSTATE`, etc.

# Retrieving Database Metadata

- Database metadata is the information that describes database itself.
- JDBC provides the DatabaseMetaData interface for obtaining database wide information and the ResultSetMetaData interface for obtaining the information on the specific ResultSet.
- The DatabaseMetaData interface provides more than 100 methods for getting database metadata concerning the database as a whole.
- These methods can be divided into three groups: for retrieving general information, for finding database capabilities, and for getting object descriptions.

# Retrieving Database Metadata

- General Information:
  - The general information includes the URL, username, product name, product version, driver name, driver version, available functions, available data types and so on.
- Database Capabilities:
  - whether the database supports the GROUP BY operator, the ALTER TABLE command with add column option, supports entry-level or full ANSI92 SQL grammar.
- Obtaining Object Descriptions:
  - the examples of the database objects are tables, views, and procedures.



```
DatabaseMetaData dbMetaData = connection.getMetaData();
System.out.println("database URL: " + dbMetaData.getURL());
System.out.println("database username: " +
    dbMetaData.getUserName());
System.out.println("database product name: " +
    dbMetaData.getDatabaseProductName());
System.out.println("database product version: " +
    dbMetaData.getDatabaseProductVersion());
System.out.println("JDBC driver name: " +
    dbMetaData.getDriverName());
System.out.println("JDBC driver version: " +
    dbMetaData.getDriverVersion());
System.out.println("JDBC driver major version: " +
    new Integer(dbMetaData.getDriverMajorVersion()));
System.out.println("JDBC driver minor version: " +
    new Integer(dbMetaData.getDriverMinorVersion()));
System.out.println("Max number of connections: " +
    new Integer(dbMetaData.getMaxConnections()));
System.out.println("MaxTableNameLentgh: " +
    new Integer(dbMetaData.getMaxTableNameLength()));
System.out.println("MaxColumnsInTable: " +
    new Integer(dbMetaData.getMaxColumnsInTable()));
connection.close();
```

# Batch Updates

- To improve performance, JDBC 2 introduced the batch update for processing non-select SQL commands.
  - A batch update consists of a sequence of non-select SQL commands.
  - These commands are collected in a batch and submitted to the database all together.

```
Statement statement = conn.createStatement();
```

```
// Add SQL commands to the batch
```

```
statement.addBatch("create table T (C1 integer, C2 varchar(15))");
```

```
statement.addBatch("insert into T values (100, 'Smith')");
```

```
statement.addBatch("insert into T values (200, 'Jones')");
```

```
// Execute the batch
```

```
int count[] = statement.executeBatch();
```

The executeBatch() method returns an array of counts, each of which counts the number of the rows affected by the SQL command. The first count returns 0 because it is a DDL command. The rest of the commands return 1 because only one row is affected.

# Scrollable and Updateable Result Set

- The result sets used in the preceding examples are read sequentially.
  - A result set 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 forward to the next row. - this is known as *sequential forward reading*. It is the only way of processing the rows in a result set that is supported by JDBC 1.
- With JDBC 2, you can scroll the rows both forward and backward and move the cursor to a desired location using the `first`, `last`, `next`, `previous`, `absolute`, or `relative` method.
- Additionally, you can insert, delete, or update a row in the result set and have the changes automatically reflected in the database.

# Creating Scrollable Statements

To obtain a scrollable or updateable result set, you must first create a statement with an appropriate type and concurrency mode. For a static statement, use

```
Statement statement = connection.createStatement  
(int resultSetType, int resultSetConcurrency);
```

TYPE\_FORWARD\_ONLY  
TYPE\_SCROLL\_INSENSITIVE  
TYPE\_SCROLL\_SENSITIVE

For a prepared statement, use

```
PreparedStatement statement = connection.prepareStatement  
(String sql, int resultSetType, int resultSetConcurrency);
```

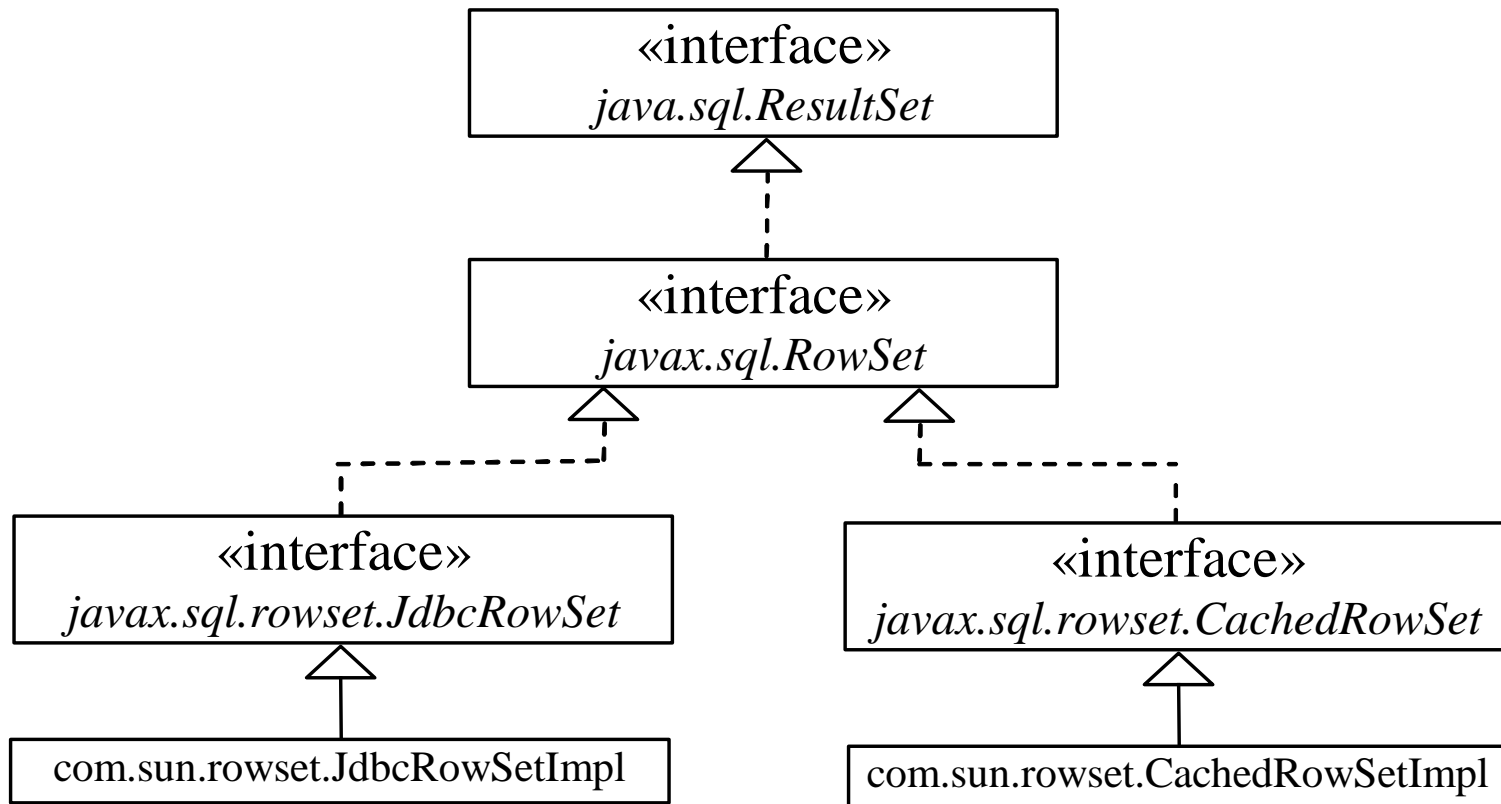
CONCUR\_READ\_ONLY  
CONCUR\_UPDATABLE

The resulting set is scrollable

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

# RowSet: JdbcRowSet and CachedRowSet

JDBC 2 introduced a new RowSet interface that can be used to simplify database programming. The RowSet interface extends `java.sql.ResultSet` with additional capabilities that allow a RowSet instance to be configured to connect to a JDBC url, username, password, set a SQL command, execute the command, and retrieve the execution result.



# SQL BLOB and CLOB Types

**BLOB** Database can store not only numbers and strings, but also images. SQL3 introduced a new data type BLOB (*Binary Large Object*) for storing binary data, which can be used to store images.

**CLOB** Another new SQL3 type is CLOB (*Character Large Object*) for storing a large text in the character format. JDBC 2 introduced the interfaces [java.sql.Blob](#) and [java.sql.Clob](#) to support mapping for these new SQL types. JDBC 2 also added new methods, such as [getBlob](#), [setBinaryStream](#), [getClob](#), [setBlob](#), and [setClob](#), in the interfaces [ResultSet](#) and [PreparedStatement](#) to access SQL BLOB, and CLOB values.

To store an image into a cell in a table, the corresponding column for the cell must be of the BLOB type. For example, the following SQL statement creates a table whose type for the flag column is BLOB.

```
create table Country(name varchar(30), flag blob,  
description varchar(255));
```

# Storing and Retrieving Images in JDBC

To insert a record with images to a table, define a prepared statement like this one:

```
PreparedStatement pstmt = connection.prepareStatement(  
    "insert into Country values(?, ?, ?)");
```

Images are usually stored in files. You may first get an instance of `InputStream` for an image file and then use the `setBinaryStream` method to associate the input stream with a cell in the table, as follows:

```
// Store image to the table cell  
File file = new File(imageFileNames[i]);  
InputStream inputImage = new FileInputStream(file);  
pstmt.setBinaryStream(2, inputImage, (int)(file.length()));
```

Store  
image

To retrieve an image from a table, use the `getBlob` method, as shown below:

```
// Store image to the table cell  
Blob blob = rs.getBlob(1);  
ImageIcon imageIcon = new ImageIcon(  
    blob.getBytes(1, (int)blob.length()));
```

Retrieve  
image

# Transactions in JDBC

- Default for a connection is
  - Transaction boundaries
    - *Autocommit mode*: each SQL statement is a transaction.
    - To group several statements into a transaction use `con.setAutoCommit(false)`
  - Isolation
    - default isolation level of the underlying DBMS
    - To change isolation level use `con.setTransactionIsolationLevel(TRANSACTION_SERIALIZABLE)`
- With autocommit off:
  - transaction is committed using `con.commit()`.
  - next transaction is automatically initiated (chaining)
- Transactions on each connection committed separately



# SQLJ

- A statement-level interface to Java
  - A dialect of embedded SQL designed specifically for Java
  - Translated by precompiler into Java
  - SQL constructs translated into calls to an SQLJ runtime package, which accesses database through calls to a JDBC driver
- Part of SQL:2003

# SQLJ

- Has some of efficiencies of embedded SQL
  - Compile-time syntax and type checking
  - Use of host language variables
  - More elegant than embedded SQL
- Has some of the advantages of JDBC
  - Can access multiple DBMSs using drivers
  - SQLJ statements and JDBC calls can be included in the same program

# SQLJ Example

```
#SQL {  
    SELECT C.Enrollment  
    INTO :numEnrolled  
    FROM Class C  
    WHERE C.CrsCode = :crsCode  
           AND C.Semester = :semester  
};
```

# Example of SQLJ Iterator

- Similar to JDBC's ResultSet; provides a cursor mechanism

```
#SQL iterator GetEnrolledIter (int studentId, String studGrade);  
GetEnrolledIter iter1;
```

```
#SQL iter1 = {  
    SELECT T.StudentId as "studentId",  
           T.Grade as "studGrade"  
    FROM Transcript T  
    WHERE T.CrsCode = :crsCode  
           AND T.Semester = :semester  
};
```

*Method names by  
which to access the  
attributes StudentId  
and Grade*

# Iterator Example (cont'd)

```
int id;
```

```
String grade;
```

```
while ( iter1.next() ) {
```

```
    id = iter1.studentId();
```

```
    grade = iter1.studGrade();
```

```
    ... process the values in id and grade ...
```

```
};
```

```
iter1.close();
```

# ODBC

- Call level interface that is database independent
- Related to SQL/CLI, part of SQL:1999
- Software architecture similar to JDBC with driver manager and drivers
- Not object oriented
- Low-level: application must specifically allocate and deallocate storage

## Sequence of Procedure Calls Needed for ODBC

```
SQLAllocEnv(&henv);           // get environment handle
SQLAllocConnect(henv, &hdbc); // get connection handle
SQLConnect(hdbc, db_name, userId, password); //
    connect
SQLAllocStmt(hdbc, &hstmt);   // get statement handle
SQLPrepare(hstmt, SQL statement); // prepare SQL statement
SQLExecute(hstmt);
SQLFreeStmt(hstmt);           // free up statement space
SQLDisconnect(hdbc);
SQLFreeEnv(henv);             // free up environment space
```

# ODBC Features

- Cursors

- *Statement handle* (for example `hstmt`) is used as name of cursor

- Status Processing

- Each ODBC procedure is actually a function that returns status

```
RETCODE retcode1;
```

```
Retcode1 = SQLConnect ( ... )
```

- Transactions

- Can be committed or aborted with

```
SQLTransact (henv, hdbc, SQL_COMMIT)
```