

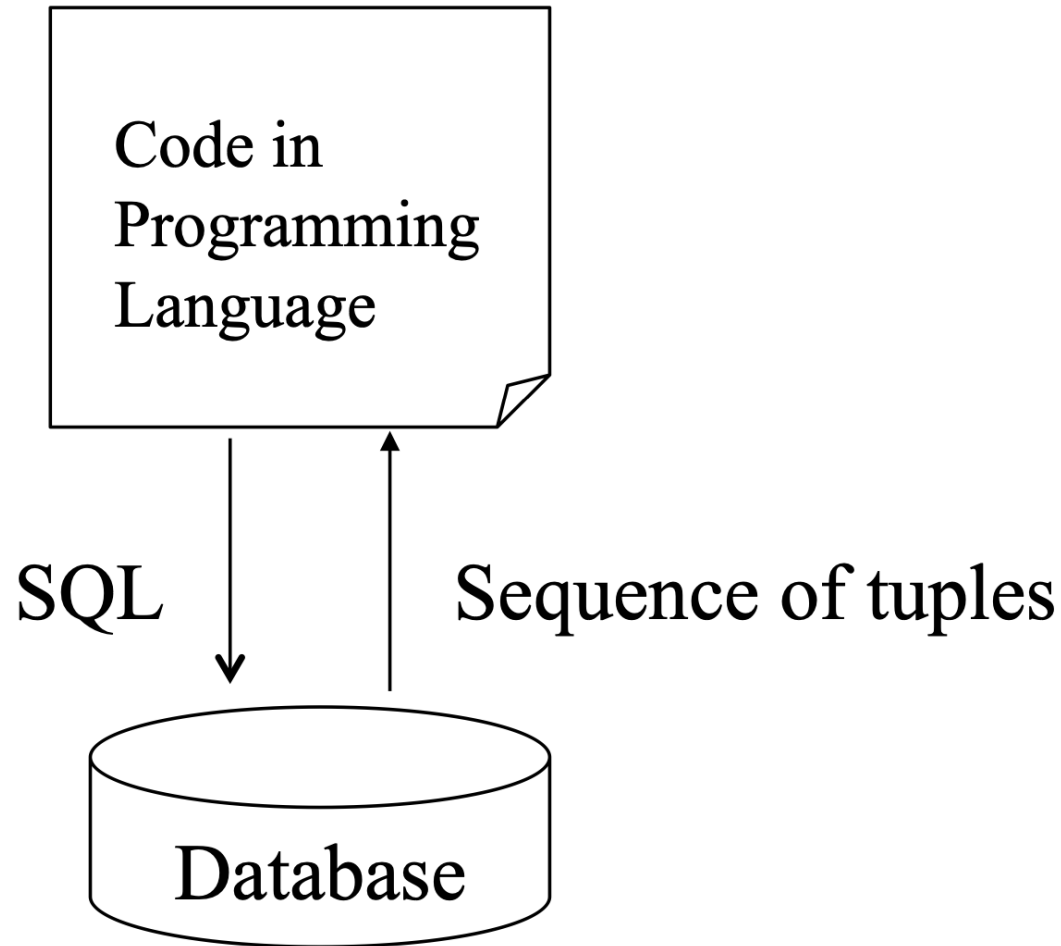
# Lecture 12

## Database Programming

# SQL in Real Programs

- We have only 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
  - Programs in a conventional language like C are written to access a database by “calls” to SQL statements

# Database Programming



# 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
- Two main integration approaches:
  - Statement-level interface (SLI)
    - Embed SQL in the host language (embedded SQL, SQLJ)
    - Dynamic SQL
  - Call-level interface (CLI)
    - Create special API to call SQL commands (JDBC, ODBC)

# Embedded SQL

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



Does the schema of the database must be known at the time the program is written for embedded SQL?

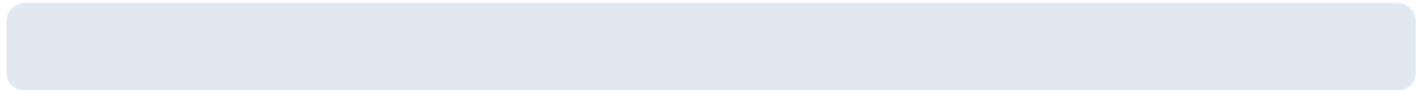
Yes

No



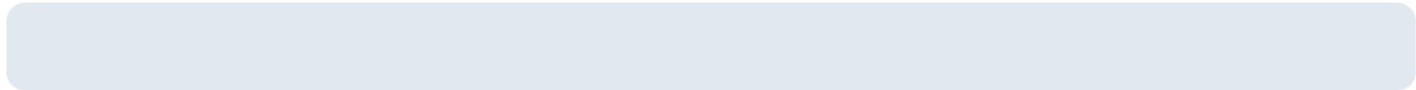
Does the schema of the database must be known at the time the program is written for embedded SQL?

Yes



0%

No

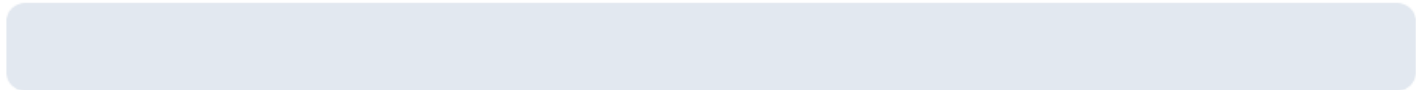


0%



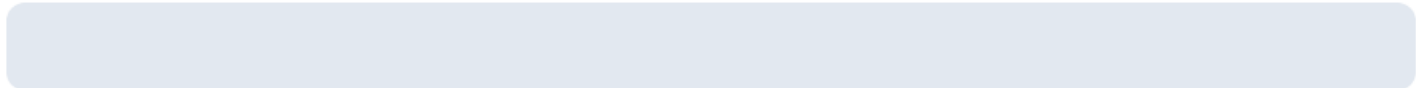
Does the schema of the database must be known at the time the program is written for embedded SQL?

Yes



0%

No



0%



# Language Constructs

- Connecting to a database  
**EXEC SQL CONNECT TO**
- Declaring variables  
**EXEC SQL BEGIN DECLARE SECTION;**  
**EXEC SQL END DECLARE SECTION;**
- Embedding Statements  
**EXEC SQL** SQL Statements;

# Connections

- The connection initiates an SQL session on the server
  - **CONNECT TO** {DEFAULT | *db-name-string*} [**AS** *connection-name-string*] [**USER** *user-id-string*]
  - *db-name-string*: the name of the data source
  - *connection-name-string*: the name you give to the connection
  - *user-id-string*: the name of a user account
  - **DISCONNECT** {DEFAULT | *db-name-string*}
- A program can execute additional CONNECT statements to different servers

# Variable Declaration

- Can use host-language variables in SQL statements
  - Must be declared between  
**EXEC SQL BEGIN DECLARE SECTION;**  
.  
.  
.  
**EXEC SQL END DECLARE SECTION;**
  - Must be prefixed by a colon (:) when used later

# Variable Declaration in C

- Variables in C program are defined as:  
**EXEC SQL BEGIN DECLARE SECTION;**  
char c\_sname[20];  
int c\_sid;  
long c\_rating;  
float c\_age;  
**EXEC SQL END DECLARE SECTION ;**

# “Error” Variables

Two special variables for reporting errors:

- `SQLCODE` (older)
  - A negative value to indicate a particular error condition
  - The appropriate C type is *long*
- `SQLSTATE` (SQL-92 standard)
  - Predefined codes for common errors
  - Appropriate C type is `char[6]` (a character string of five letters along with a null character at the end to terminate the string)
- One of these two variables must be declared (we assume `SQLSTATE`)

# SQL Statements

- All SQL statements embedded within a host program must be clearly marked
- In C, SQL statements must be prefixed by EXEC SQL. For example:

**EXEC SQL**

**INSERT INTO Sailors**

**VALUES(:c\_sname,:c\_sid,:c\_rating,:c\_age);**

- Java embedding (SQLJ) uses **#SQL { .... };**

# SELECT - Retrieving Single Row

```
EXEC SQL SELECT S.sname, S.age  
          INTO :c_sname, :c_age  
          FROM Sailors S  
          WHERE S.sid = :c_sid;
```

# SELECT - Retrieving Multiple Rows

- What if we want to embed the following query?

```
SELECT S.sname, S.age  
FROM Sailors  
WHERE S.rating > :c_minrating
```

- Potentially, multiple rows will be retrieved
- How do we store a set of rows?
  - No equivalent data type in host languages like C



# Impedance Mismatch

- SQL statement: a set of tuples
- Host language: a variable
- SQL supports a mechanism called *cursor* to handle this

# Cursors

- **DECLARE** a cursor on a relation or query statement (which generates a relation)
- **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 a tuple pointed to by a cursor
- **CLOSE** a cursor

# Declaring a Cursor

- Cursor that points to names and ages of sailors whose ratings are greater than “minrating”, in alphabetical order

```
EXEC SQL DECLARE sinfo CURSOR FOR  
SELECT S.sname, S.age  
FROM Sailors S  
WHERE S.rating > :c_minrating  
ORDER BY S.sname;
```

# Opening, Fetching a Cursor

- To open the cursor (executed at run-time):
  - **OPEN sinfo;**
  - The cursor is initially positioned prior to the first row
- To read the current row that the cursor is pointing to:
  - **FETCH sinfo INTO :c\_sname, :c\_age**
- When FETCH is executed, the cursor is positioned to point at the next row
  - Can put the FETCH statement in a loop to retrieve multiple rows, one row at a time

# Closing a Cursor

- When we're done with the cursor, we can close it:
  - **CLOSE** `sinfo`;
- We can re-open the cursor again. However, the rows retrieved might be different (depending on the value(s) of the associated variable(s) when the cursor is opened)
  - E.g., If `:c_minrating` is now set to a different value, then the rows retrieved will be different



Do changes made to the values of the parameters after the cursor is opened have any effect on the tuples that are retrieved through it?

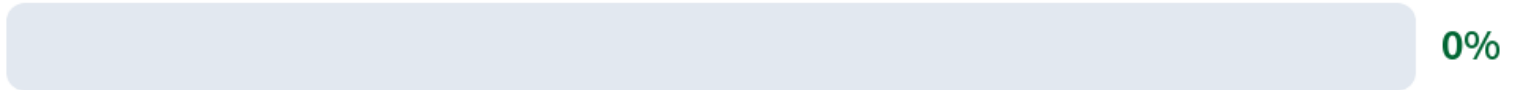
Yes

No

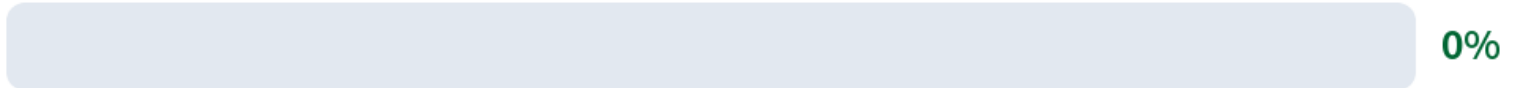


Do changes made to the values of the parameters after the cursor is opened have any effect on the tuples that are retrieved through it?

Yes



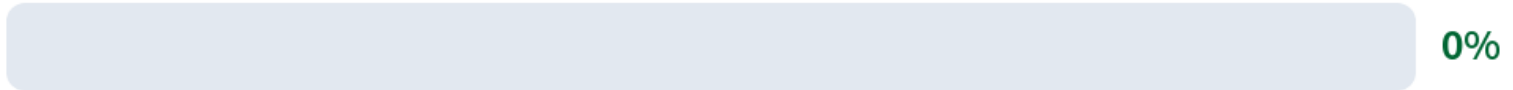
No



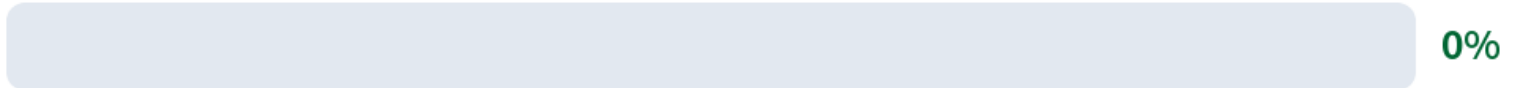


Do changes made to the values of the parameters after the cursor is opened have any effect on the tuples that are retrieved through it?

Yes



No





# Embedding SQL in C: An Example

```
EXEC SQL BEGIN DECLARE SECTION;
    char c_sname[20]; long c_minrating; float c_age;
    char SQLSTATE[6];
EXEC SQL END DECLARE SECTION;
c_minrating = random();
EXEC SQL DECLARE sinfo CURSOR FOR           // declare cursor
    SELECT S.sname, S.age
    FROM Sailors S
    WHERE S.rating > :c_minrating
    ORDER BY S.sname;
EXEC SQL OPEN sinfo;                       // open cursor
do {
    EXEC SQL FETCH sinfo INTO :c_sname, :c_age;
    printf("%s is %d years old\n", c_sname, c_age);
} while (strcmp(SQLSTATE, "00000")!=0);
EXEC SQL CLOSE sinfo;                     // close cursor
```

# Update/Delete Commands

- Modify the rating value of the row currently pointed to by cursor sinfo

```
UPDATE Sailors S  
SET S.rating = S.rating + 1  
WHERE CURRENT of sinfo;
```

- Delete the row currently pointed to by cursor sinfo

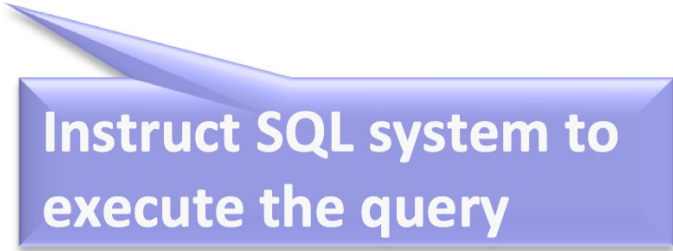
```
DELETE Sailors S  
FROM CURRENT of sinfo;
```

# Dynamic SQL

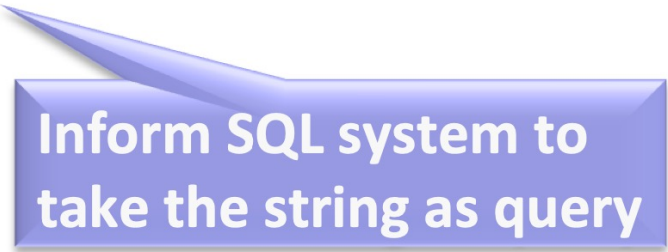
- SQL query strings are not always known at compile time
  - Such application must accept commands from the user; and based on what the user needs, generate appropriate SQL statements
  - The SQL statements are constructed **on-the-fly**
- Dynamic SQL allows programs to construct and submit SQL queries at run time

# Dynamic SQL - Example

```
char c_sqlstring[ ] = "DELETE FROM Sailor WHERE rating > 5";  
EXEC SQL PREPARE readytogo FROM :c_sqlstring;  
EXEC SQL EXECUTE readytogo;
```



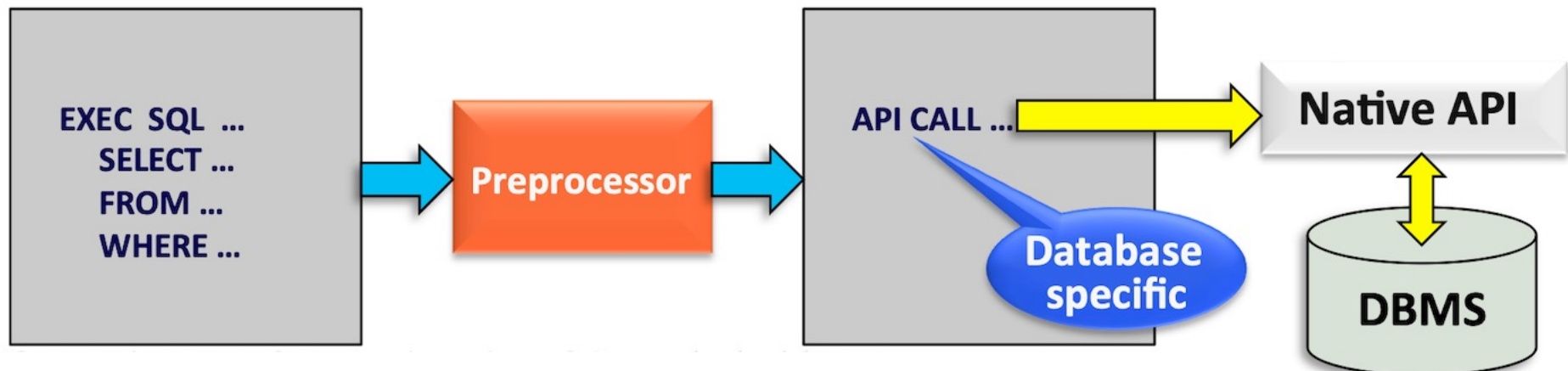
Instruct SQL system to  
execute the query



Inform SQL system to  
take the string as query

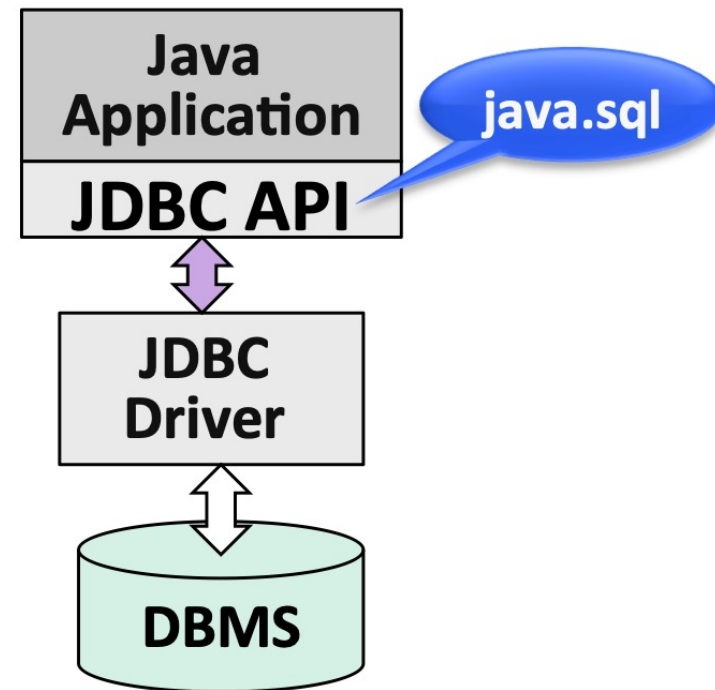
# Limitation of Embedded SQL

- DBMS-specific preprocessor transform the embedded SQL statements into function calls in the host language
- This translation varies across DBMSs (API calls vary among different DBMSs)
- Even if the source code can be compiled to work with different DBMS's, the final executable works only with one specific DBMS



# Database API

- ODBC and JDBC are API (application-program interface) for a program to interact with a database server
- Application makes calls to
  - Connect with the database server
  - Send SQL commands to the database server
  - Fetch tuples of result one-by-one into program variables
- ODBC (Open Database Connectivity) works with C, C++, C#, and Visual Basic
- JDBC (Java Database Connectivity) works with Java



# JDBC

- JDBC is a collection of Java classes and interface that enables database access
- JDBC contains methods for
  - connecting to a remote data source
  - executing SQL statements
  - receiving SQL results
  - transaction management
  - exception handling
- The classes and interfaces are part of the `java.sql` package



Using which of the following, neither the DBMS nor the schema need be known at compile time?

JDBC

Embedded SQL

Dynamic SQL





Using which of the following, neither the DBMS nor the schema need be known at compile time?

JDBC

0%

Embedded SQL

0%

Dynamic SQL

0%



Using which of the following, neither the DBMS nor the schema need be known at compile time?

JDBC

0%

Embedded SQL

0%

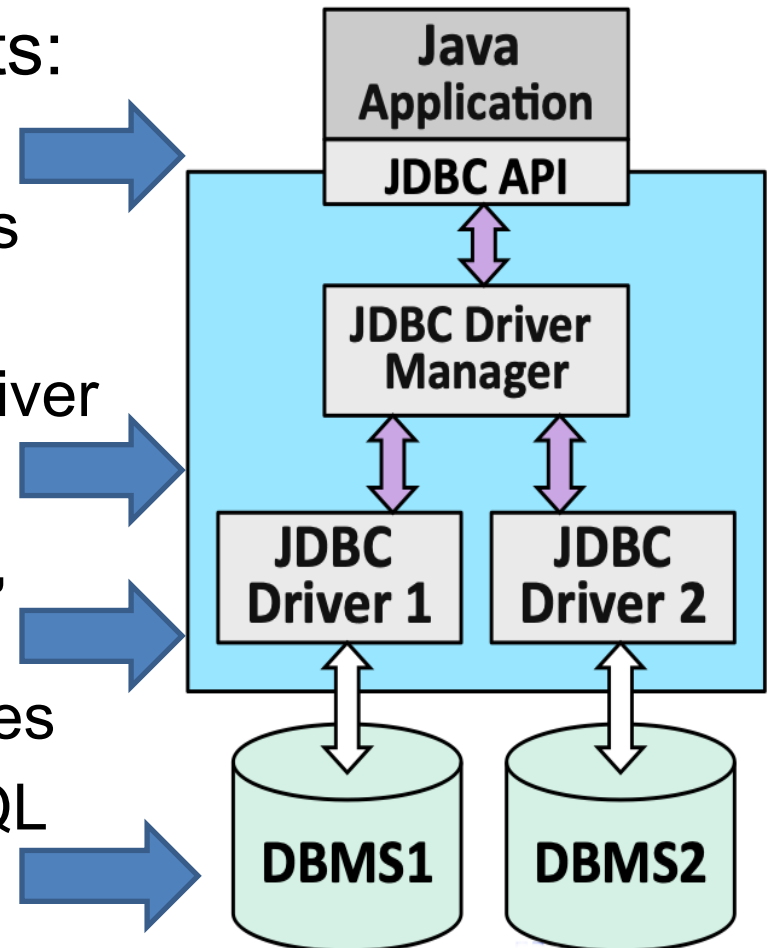
Dynamic SQL

0%

# JDBC: Architecture

Four architectural components:

- Application: initiates and terminates connections, submits SQL statements
- Driver manager: loads JDBC driver and passes function calls
- Driver: connects to data source, transmits requests and returns/ translates results and error codes
- Database server: processes SQL statements



# JDBC Advantages

- Portability across database server
  - The JDBC drivers take care of the server dependencies
- Portability across hardware architecture
  - Result of Java language

# JDBC Main Steps

1. Load the JDBC driver
  - `Class.forName()`
2. Connect to the database
  - `DriverManager.getConnection()`
    - Establish the connection using user Id and password
    - Create a Connection object and assign it to a variable (e.g., con1)
3. Execute SQL statements
  - `con1.createStatement()`
    - Create a Statement object and assign it to a variable (e.g., stat1)
  - `stat1.executeQuery()`
    - Prepare and execute an SQL statement
    - Create a ResultSet object for returning data
4. Close the connection
  - `stat1.close()`
  - `con1.close()`

# The Connection Object

- All interactions with the database are performed via a **Connection** object
- A Connection object can be created only by using the getConnection method on the DriverManager class
- It is used to send SQL statements to the database server
- Two classes for sending SQL statements:
  - Statement: for SQL statements without parameters
  - PreparedStatement: precompiled

# The Connection Object (Cont.)

- Methods from the Connection class:
  - createStatement: returns a Statement object
  - prepareStatement: an SQL statement can be precompiled and stored in a PreparedStatement object
  - close: immediately release a Connection object's database and JDBC resources
  - getMetaData: get database information



If the same SQL statement is to be executed many times, which is more efficient to use?

CreateStatement

PrepareStatement





If the same SQL statement is to be executed many times, which is more efficient to use?

CreateStatement

0%

PrepareStatement

0%



If the same SQL statement is to be executed many times, which is more efficient to use?

CreateStatement

0%

PrepareStatement

0%

# JDBC Detailed Steps

1. Importing packages
2. Registering JDBC drivers
3. Opening a Connection to a database
4. Creating a Statement object
5. Executing a query and returning a ResultSet object
6. Processing the result set
7. Closing the ResultSet and Statement Objects
8. Closing the Connection

# 1: Importing Packages

```
import java.sql.*;    //import JDBC packages  
import java.math.*;  
import java.io.*;  
import oracle.jdbc.driver.*;
```

## 2. Registering JDBC Drivers

```
class MyExample {  
    public static void main (String args[]) throws  
        SQLException {  
        // Load Oracle driver  
        Class.forName("oracle.jdbc.driver.  
            OracleDriver");  
        ...  
    }  
}
```

### 3. Opening Connection to a Database

```
//Prompt user for username and password
```

```
String user;
```

```
String password;
```

```
user = readEntry("username: ");
```

```
password = readEntry("password: ");
```

```
//Connect to the database
```

```
Connection conn = DriverManager.getConnection
```

```
    ("jdbc:oracle:thin:@artemis.vsnnet.gmu.edu:1521/vse18c.v  
    snet.gmu.edu", user, password);
```

```
//Format: Connection <connection_object> =
```

```
DriverManager.getConnection("jdbc:oracle:drivertype:
```

```
@<hostname>:<port>/<servicename>","<username>","<password>");
```

## 4. Creating a Statement Object

- Suppose Books has attributes isbn, title, author, quantity, price, year. (Initial quantity is always zero.) ?'s are placeholders

```
String sql = "INSERT INTO Books VALUES(?,?,?,0,?,?)";  
PreparedStatement pstmt=conn.prepareStatement(sql);
```

- Set values for placeholders

```
pstmt.clearParameters();  
pstmt.setString(1, isbn);  
pstmt.setString(2, title);  
pstmt.setString(3, author);  
pstmt.setFloat(4, price);  
pstmt.setInt(5, year);
```

## 5. Executing a Query, Returning a Result Set & 6. Processing the Result Set

- `executeUpdate()` is used if the SQL statement does not return any records (e.g. UPDATE, INSERT, ALTER, and DELETE)
- Returns an integer indicating the number of rows the SQL statement modified  
`int numRows = pstmt.executeUpdate();`



# Step 5&6 (Cont.)

- `executeQuery()` is used if the SQL statement returns data, such as in a SELECT query

```
String sqlQuery = "SELECT title, price FROM Books WHERE  
author=?";
```

```
PreparedStatement pstmt2 = conn.prepareStatement (sqlQuery);  
pstmt2.setString(1, author);
```

```
ResultSet rset = pstmt2.executeQuery();
```

- Print query results

```
while (rset.next ())  
    System.out.println(rset.getString (1)+ " " +  
        rset.getFloat(2));
```

// the (1) in getString refers to the title value,

// and the (2) refers to the price value

## 7. Closing the ResultSet and Statement Objects & 8. Closing the Connection

- Close the result set, statement, and the connection

```
rset.close();
```

```
pstmt.close();
```

```
pstmt2.close();
```

```
conn.close();
```

# Result Set

- `executeUpdate()` only returns the number of affected records
- `executeQuery()` returns data, encapsulated in a `ResultSet` object
  - `ResultSet` has a cursor that allows us to read one row at a time
  - Initially, the cursor is positioned before the first row
  - Use `next()` to move to the next row
  - `next()` returns false if there are no more rows

# ResultSet Navigation Methods

## POSITIONING THE CURSOR

<code>next()</code>	Move to next row
<code>previous()</code>	Moves back one row
<code>absolute(int num)</code>	Moves to the row with the specified number
<code>relative(int num)</code>	Moves forward or backward (if negative)
<code>first()</code>	Moves to the first row
<code>last()</code>	Moves to the last row

# ResultSet getXXX() Methods

## RETRIEVE VALUES FROM COLUMNS

getString(string columnName):	Retrieves the value of designated column in current row
getString(int columnIndex)	Retrieves the value of designated column in current row
getFloat (string columnName)	Retrieves the value of designated column in current row

⋮

# Mapping Data Types

- There are data types specified to SQL that need to be mapped to Java data types if the user expects Java to be able to handle them
- Conversion falls into three categories:
  - SQL type to Java direct equivalents
    - SQL INTEGER direct equivalent of Java int data type
  - SQL type can be converted to a Java equivalent
    - SQL CHAR, VARCHAR, and LONGVARCHAR can all be converted to the Java String data type
  - SQL data type is unique and requires a special Java data class object to be created specifically for their SQL equivalent
    - SQL DATE converted to the Java Date object that is defined in java.util.Date especially for this purpose

# SQLJ

- Embedded SQL for Java
- SQLJ is similar to existing extensions for SQL that are provided for C, FORTRAN, and other programming languages
- IBM, Oracle, and several other companies have proposed SQLJ as a standard and as a simpler and easier-to-use alternative to JDBC

# SQLJ

```
#sql { ... };
```

- SQLJ can span multiple lines
- Java host expressions in SQL statement



# SQLJ Example

```
String title; Float price; String author="Lee";  
// declare iterator class  
#SQL iterator BooksIter(String title, Float price);  
BooksIter books1;  
  
// initialize the iterator object books; sets the  
// author, execute query and open the cursor  
#SQL books1 =  
    {SELECT title, price INTO :title, :price  
     FROM Books WHERE author=:author};  
  
// retrieve results  
while(books1.next())  
    System.out.println(books1.title()+", "+books1.price());  
books1.close();
```

# JDBC Equivalent

```
String sqlQuery = "SELECT title, price FROM Books  
WHERE author=?";
```

```
PreparedStatement pstmt2 = conn.prepareStatement(sqlQuery);  
pstmt2.setString(1, author);
```

```
ResultSet rset = pstmt2.executeQuery();
```

```
// Print query results. The (1) in getString refers to the  
// title value, and the (2) refers to the price value  
while (rset.next ())
```

```
    System.out.println (rset.getString (1)+ " " +  
    rset.getFloat(2));
```

# SQLJ Summary

- SQLJ is analogous to embedded SQL but was designed specifically to be embedded in Java programs
- An SQLJ program can connect to multiple DBMSs using different JDBC drivers
- SQLJ statements and JDBC calls can be included in the same Java program

# SQLJ vs. JDBC

- Performance
  - SQLJ: faster
  - JDBC: slower
- Debugging
  - SQLJ: easier
  - JDBC: harder
- Use JDBC when your application requires dynamic capabilities

# Useful JDBC Tutorials

- <https://docs.oracle.com/javase/tutorial/jdbc/basics/index.html>
- <http://infolab.stanford.edu/~ullman/fcdb/oracle/or-jdbc.html>