

# Database-Connection Libraries

CALL-LEVEL INTERFACE

---

JAVA DATABASE CONNECTIVITY

PHP PEAR/DB

# An Aside: SQL Injection

---

SQL queries are often constructed by programs.

These queries may take constants from user input.

Careless code can allow rather unexpected queries to be constructed and executed.

# Example: SQL Injection

---

Relation **Accounts**(name, passwd, acct).

**Web interface**: get name and password from user, store in strings *n* and *p*, issue query, display account number.

```
SELECT acct FROM Accounts
WHERE name = :n AND passwd = :p
```

# User Types

---

Name:

gates'

--

Comment  
in Oracle

Password:

monday139

Your account number is 1234-567

# The Query Executed

---

```
SELECT acct FROM Accounts
```

```
WHERE name = 'gates' --'
```

```
passwd = 'monday139?'
```

AND

All treated as a comment

# Host/SQL Interfaces Via Libraries

---

The other approach to connecting databases to conventional languages is to use library calls.

1. C + CLI
2. Java + JDBC
3. PHP + PEAR/DB

# Three-Tier Architecture

---

A common environment for using a database has three tiers of processors:

1. *Web servers* --- talk to the user.
2. *Application servers* --- execute the business logic.
3. *Database servers* --- get what the app servers need from the database.

# Example: Amazon

---

Database holds the information about products, customers, etc.

Business logic includes things like “what do I do after someone clicks ‘checkout’?”

- **Answer:** Show the “how will you pay for this?” screen.



# Environments, Connections, Queries

---

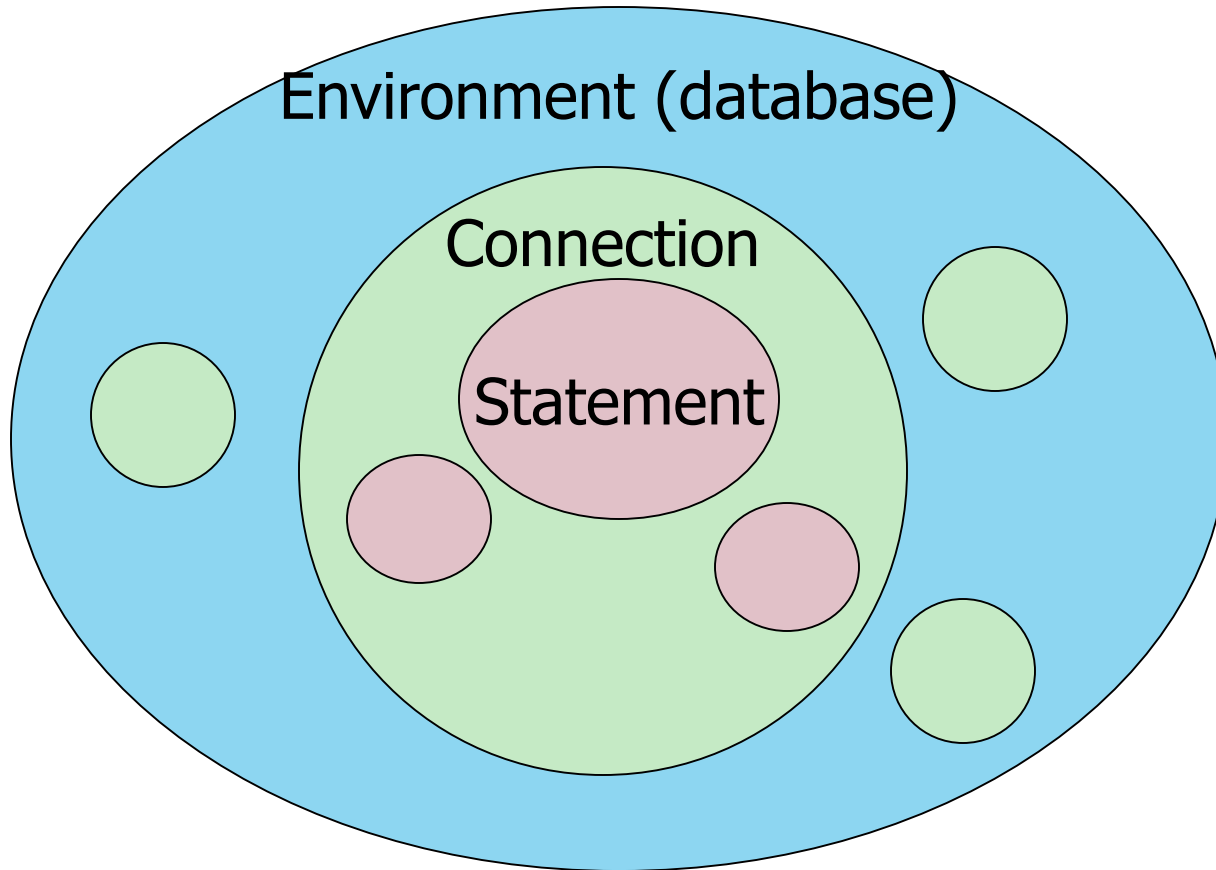
The database is, in many DB-access languages, an *environment*.

Database servers maintain some number of *connections*, so app servers can ask queries or perform modifications.

The app server issues *statements* : queries and modifications, usually.

# Diagram to Remember

---



# SQL/CLI

---

We can use a library of functions.

- The library for C is called SQL/CLI = *Call-Level Interface*.

# Data Structures

---

C connects to the database by structures of the following types:

1. *Environments* : represent the DBMS installation.
2. *Connections* : logins to the database.
3. *Statements* : SQL statements to be passed to a connection.
4. *Descriptions* : records about tuples from a query, or parameters of a statement.

# Handles

---

Function `SQLAllocHandle(T,I,O)` is used to create these structures, which are called environment, connection, and statement *handles*.

- $T$  = type, e.g., `SQL_HANDLE_STMT`.
- $I$  = input handle = struct at next higher level (statement < connection < environment).
- $O$  = (address of) output handle.

# Example: SQLAllocHandle

---

```
SQLAllocHandle (SQL_HANDLE_STMT,  
myCon, &myStat);
```

myCon is a previously created connection handle.

myStat is the name of the statement handle that will be created.

# Preparing and Executing

---

**SQLPrepare(*H*, *S*, *L*)** causes the string *S*, of length *L*, to be interpreted as a SQL statement; the executable statement is placed in statement handle *H*.

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

# Example: Prepare and Execute

---

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

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



# Direct Execution

---

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

`SQLExecuteDirect(H,S,L);`

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

# Fetching Tuples

---

When the SQL statement executed is a query, we need to fetch the tuples of the result.

- A cursor is implied by the fact we executed a query; the cursor need not be declared.

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

# Accessing Query Results

---

When we fetch a tuple, we need to put the attribute values somewhere.

Each component is bound to a variable by the function `SQLBindCol`.

- This function has 6 arguments, of which we shall show only 1, 2, and 4:
  - 1 = handle of the query statement.
  - 2 = column number.
  - 4 = address of the variable.

# Example: Binding

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

```
SELECT beer, price FROM Sells  
WHERE bar = 'Joe''s Bar'
```

Bind the result to `theBeer` and `thePrice`:

```
SQLBindCol(myStat, 1, , &theBeer, , );  
SQLBindCol(myStat, 2, , &thePrice, , );
```

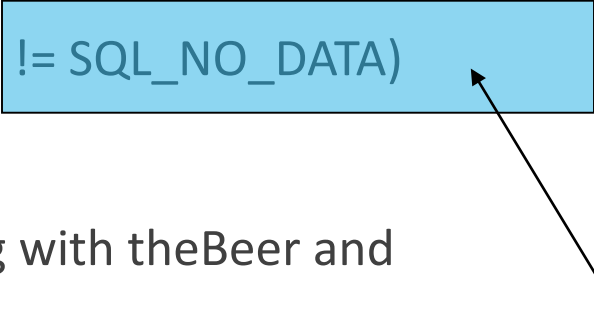
# Example: Fetching

---

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

```
SQLBindCol(myStat, 1, , &theBeer, , );  
SQLBindCol(myStat, 2, , &thePrice, , );
```

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



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

# JDBC

---

*Java Database Connectivity* (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

---

```
import java.sql.*;
Class.forName("com.mysql.jdbc.Driver");
Connection myCon =
    DriverManager.getConnection(...);
```

The JDBC classes

Loaded by  
forName

URL of the database  
your name, and password  
go here.

The driver  
for mySql;  
others exist

# 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 methods `executeQuery` and `executeUpdate`.

- For Statements: one argument: the query or modification to be executed.
- For PreparedStatements: no argument.

# Example: Update

---

stat1 is a Statement.

We can use it to insert a tuple as:

```
stat1.executeUpdate(  
    "INSERT INTO Sells "  
    "VALUES ('Brass Rail', 'Bud', 3.00) "  
);
```

# Example: Query

---

stat2 is a PreparedStatement holding the query "SELECT beer, price FROM Sells WHERE bar = 'Joe''s Bar'".

`executeQuery` returns an object of class `ResultSet` – we'll examine it later.

The query:

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

# Accessing the ResultSet

---

An object of type *ResultSet* is something like a cursor.

Method `next()` advances the “cursor” to the next tuple.

- The first time `next()` is applied, it gets the first tuple.
- If there are no more tuples, `next()` returns the value `false`.

# Accessing Components of Tuples

---

When a ResultSet is referring to a tuple, we can get the components of that tuple by applying certain methods to the ResultSet.

Method `getX(i)`, where  $X$  is some type, and  $i$  is the component number, returns the value of that component.

- The value must have type  $X$ .

# Example: Accessing Components

menu = ResultSet for query "SELECT beer, price FROM Sells WHERE bar = 'Joe' 's Bar'".

Access beer and price from each tuple by:

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

# Example: Passing Parameters

---

```
1) PreparedStatement studioStat =
    myCon.prepareStatement("INSERT INTO
        Studio(name, address) VALUES (?, ?)");

/* get values for variables studioName and
studioAddr from the user */

...

2) studioStat.setString(1, studioName) ;
3) studioStat.setString(2, studioAddr) ;
5) studioStat.executeUpdate();
```



# Example: Handling Exceptions

---

```
try{  
    ...  
}catch (SQLException ex) {  
    System.err.println("SQLException: " +  
        ex.getMessage()) ;  
    ...  
}
```

# PHP

---

A language to be used for actions within HTML text.

Indicated by `<? PHP code ?>`.

DB library exists within *PEAR* (PHP Extension and Application Repository).

- Include with `include (DB.php)`.

# Variables in PHP

---

Must begin with \$.

OK not to declare a type for a variable.

But you give a variable a value that belongs to a “class,” in which case, methods of that class are available to it.

# String Values

---

PHP solves a very important problem for languages that commonly construct strings as values:

- How do I tell whether a substring needs to be interpreted as a variable and replaced by its value?

PHP solution: Double quotes means replace; single quotes means don't.

# Example: Replace or Not?

---

```
$v = "one hundred dollars";
```

```
$sue = 'You owe me $v.';
```

```
$joe = "You owe me $v.";
```

Value of **\$sue** is 'You owe me \$v', while the value of **\$joe** is 'You owe me one hundred dollars'.

# PHP Arrays

---

Two kinds: *numeric* and *associative*.

Numeric arrays are ordinary, indexed 0,1,...

- **Example:** `$a = array("Paul", "George", "John", "Ringo");`
  - Then `$a[0]` is "Paul", `$a[1]` is "George", and so on.

# Associative Arrays

---

Elements of an associative array  $\$a$  are pairs  $x \Rightarrow y$ , where  $x$  is a key string and  $y$  is any value.

If  $x \Rightarrow y$  is an element of  $\$a$ , then  $\$a[x]$  is  $y$ .

# Example: Associative Arrays

---

An environment can be expressed as an associative array, e.g.:

```
$myEnv = array(  
    "phptype" => "oracle",  
    "hostspec" => "www.uoit.ca",  
    "database" => "cs145db",  
    "username" => "szlichta",  
    "password" => "notMyPW");
```



# Making a Connection

---

With the DB library imported and the array \$myEnv available:

```
$myCon = DB::connect($myEnv) ;
```

Function connect  
in the DB library

Class is Connection  
because it is returned  
by DB::connect().

# Executing SQL Statements

---

Method `query` applies to a Connection object.

It takes a string argument and returns a result.

- Could be an error code or the relation returned by a query.

# Example: Executing a Query

Find all the bars that sell a beer given by the variable \$beer.

```
$beer = 'Bud';  
$result = $myCon->query(  
    "SELECT bar FROM Sells"  
    „WHERE beer = $beer ;”);
```

Concatenation  
in PHP



Remember this  
variable is replaced  
by its value.

# Cursors in PHP

---

The result of a query *is* the tuples returned.

Method `fetchRow` applies to the result and returns the next tuple, or `FALSE` if there is none.

# Example: Cursors

---

```
while ($bar =  
    $result->fetchRow()) {  
    // do something with $bar  
}
```

# Actions

---

Review slides!

Go through code examples „List of Examples” and documentation :  
<http://jdbc.postgresql.org/documentation/93/>

Read chapter from the book about SQL libraries (study all the examples).