Non-interactive SQL

EECS3421 - Introduction to Database Management Systems

Using a Database

- Interactive SQL: Statements typed in from terminal; DBMS outputs to screen. Interactive SQL is inadequate in many situations:
  - It may be necessary to process the data before output
  - Amount of data returned not known in advance
- Non-interactive SQL: Statements included in an application program written in a host language — such as C, Java, PHP, Python, …

Non-interactive SQL

- Traditional applications often need to “embed” SQL statements inside the instructions of a program written in a procedural programming language (C, JAVA, etc.)
- There is a severe problem (impedance mismatch) between the computational model of a programming language (PL) and that of a DBMS:
  - The variables of a PL take as values single records, those of SQL whole tables
  - PL computations are generally on a main memory data structure, SQL ones on bulk data

The best of both worlds

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

Elements of Non-interactive SQL

- Non-interactive SQL may use a pre-compiler to manage SQL statements
- Program variables may be used as parameters in the SQL statements (variable interchange)
- Results may be
  - a single row (easy to handle)
  - sets of rows (tricky to handle)
- Execution status
  - predefined variable sqlstate (="00000" if executed successfully).

SQL Statement Preparation

- Before any 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 size, etc.
  - Result is a query execution plan

Non-interactive SQL Approaches

- In the DBMS
  - Persistent Stored Modules (PSM):

    Code in a specialized language is stored in the database itself (e.g., PSM, PL/SQL, PL/pgSQL)
- Out of the DBMS
  - Statement-level Interface (SLI):

    SQL statements are embedded in a host language (e.g., C)
  - Call-level Interface (CLI):

    Connection tools are used to allow a conventional language to access a database (e.g., CLI, JDBC, PHP/DB)

PERSISTENT STORED PROCEDURES

Persistent Stored Procedures

- Allow to store procedures as database schema
- A mixture of conventional statements (if, while, etc.) and SQL
- Allow do things we cannot do in SQL alone
- Most DBMSs offer SQL extensions that support persistent stored procedures:
  - PostgreSQL: PL/pgSQL
  - Oracle: PL/SQL
  - …

Basic PSM Form

```
CREATE PROCEDURE <name> (
        <parameter list>
  )
  <optional local declarations>
  <body>;
```

Function alternative:

```
CREATE FUNCTION <name> (
        <parameter list>
  )
  RETURNS <type>
```

Parameters in PSM

- Unlike the usual name-type pairs in languages like C, PSM uses mode-name-type triples, where the mode can be:
  - IN = procedure uses value, does not change value
  - OUT = procedure changes value, does not use value
  - INOUT = both

Example

- Write a procedure that takes two arguments *b* and *p*, and adds a tuple to Sells(bar, beer, price) that has bar = ‘Joe’’s Bar’, beer = *b*, and price = *p*
  - Used by Joe to add to his menu more easily.

```
CREATE PROCEDURE JoeMenu (
          IN      b       CHAR(20),
          IN      p       REAL
)
INSERT INTO Sells
VALUES (’Joe’’s Bar’, b, p);
```

Parameters are both read-only, not changed

The body is a single insertion

Invoking Procedures

- Use SQL/PSM statement **CALL**, with the name of the desired procedure and arguments.

  CALL JoeMenu('Moosedrool', 5.00);

Advantages of Stored Procedures

- 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

Statement-level Interface (SLI)

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

- Pre-compiler scans program and translates SQL statements into calls to host language library procedures that communicate with DBMS
- Host language compiler then compiles program

Static vs Dynamic Embedding

- SQL constructs in an application take two forms:
  - Standard SQL statements (static 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
- Pre-compiler translates statements and directives into arguments of calls to library procedures

Example of Static SQL

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

- Variables shared by host and SQL (num_enrolled, crs_code)
  - “:” used to set off host variables
  - Names of (host language) variables are contained in SQL statement and available to pre-compiler
- Routines for fetching and storing argument values can be generated
- Complete statement (with parameter values) sent to DBMS when statement is executed

Example of 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;
```

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

Call-level Interface (CLI)

Call Level Interface

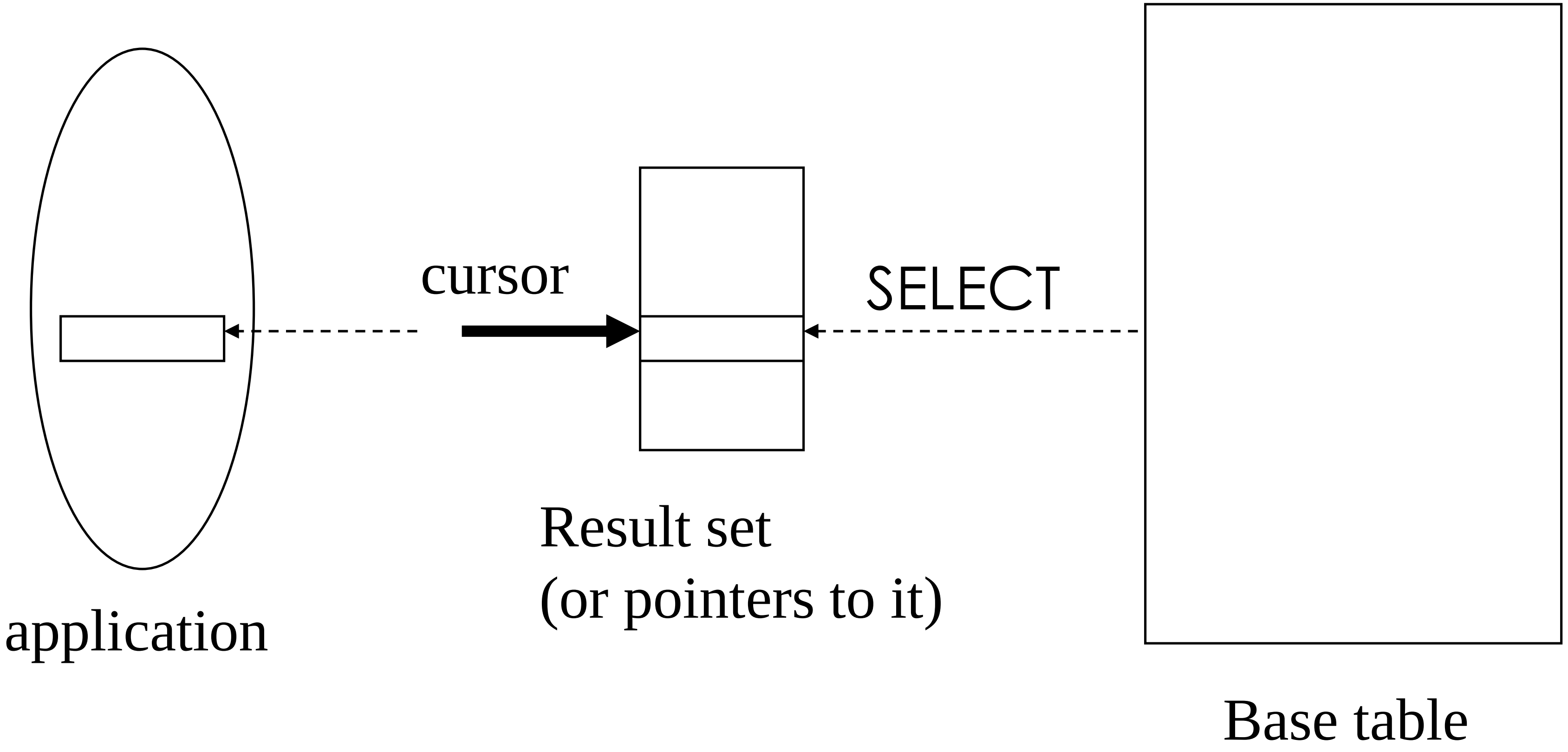
- Application program written entirely in host language (no precompiler) using library calls
  - Java + JDBC
  - PHP + PEAR/DB
- SQL statements are values of string variables constructed at run time using host language
  - similar to dynamic SQL
- Application uses string variables as arguments of library routines that communicate with DBMS

  e.g. executeQuery(“SQL query statement”)

Cursors

- Fundamental problem with database technology: *impedance mismatch*
  - traditional programming languages process records one-at-a-time (tuple-oriented)
  - SQL processes tuple sets (set-oriented).
- *Cursors* solve this problem: A cursor returns tuples from a result set, to be processed one-by-one

How Cursors Work?



Operations on Cursors

- Result Set: rows returned by a **SELECT** statement
- To execute the query associated with a cursor:

  open CursorName

- To extract one tuple from the query result:

  fetch [Position from] CursorName **into** FetchList

- To free the cursor, discarding the query result:

  close CursorName

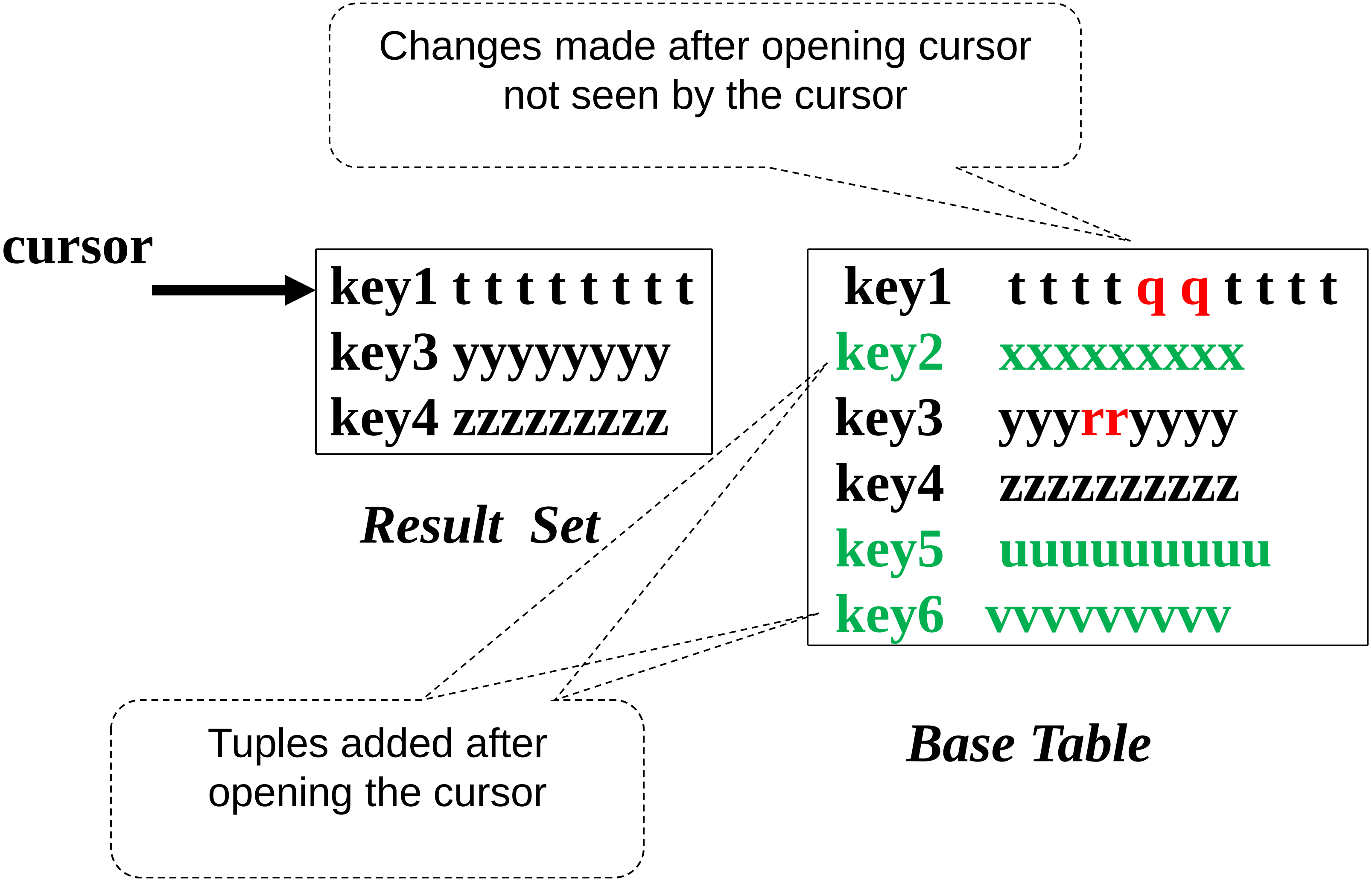
- To access the current tuple (when a cursor reads a relation, in order to update it):

  current of CursorName (in a where clause)

Cursor Types

- Insensitive cursors: Result set computed and stored in 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
- Sensitive cursors: Specification not part of SQL standard
  - Changes made to base table subsequent to OPEN (by any transaction) can affect result set
  - Cursor is updatable

Insensitive Cursor



Changes made after opening cursor not seen by the cursor

cursor

key1	t t t t t t t t
key3	yyyyyyyy
key4	zzzzzzzzz

Result Set

key1	t t t t q q t t t t
key2	xxxxxxxxxx
key3	yyyrryyyy
key4	zzzzzzzzzz
key5	uuuuuuuuu
key6	vvvvvvvvvv

Tuples added after opening the cursor

Base Table

Cursor Scrolling

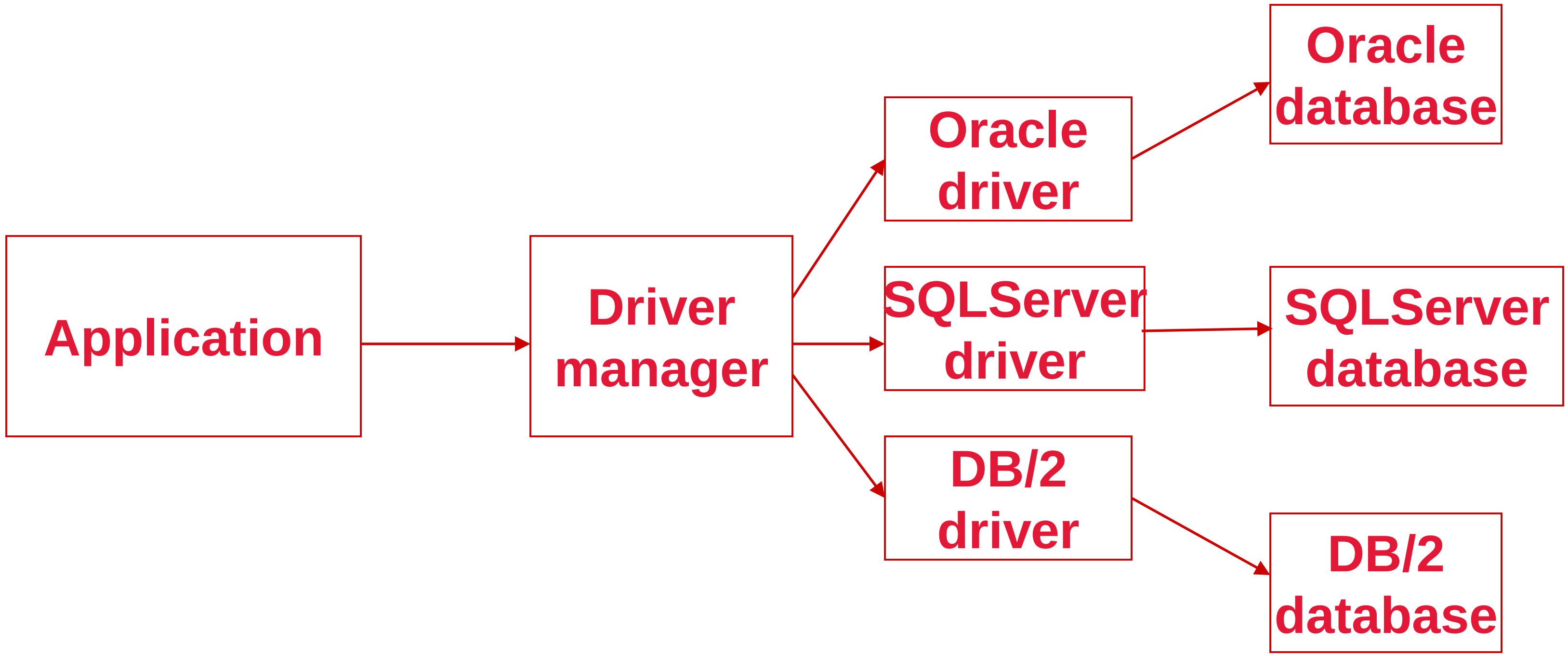
- If SCROLL option is not specified in cursor declaration, FETCH always moves cursor forward one position
- If SCROLL option is included in cursor declaration, cursor can be moved in arbitrary ways around result set (e.g., FIRST, LAST, ABSOLUTE n, RELATIVE n)

Java: JDBC

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 Standard

JDBC Run-Time Architecture



Application

Driver manager

Oracle driver

SQLServer driver

DB/2 driver

Oracle database

SQLServer database

DB/2 database

Making a Connection

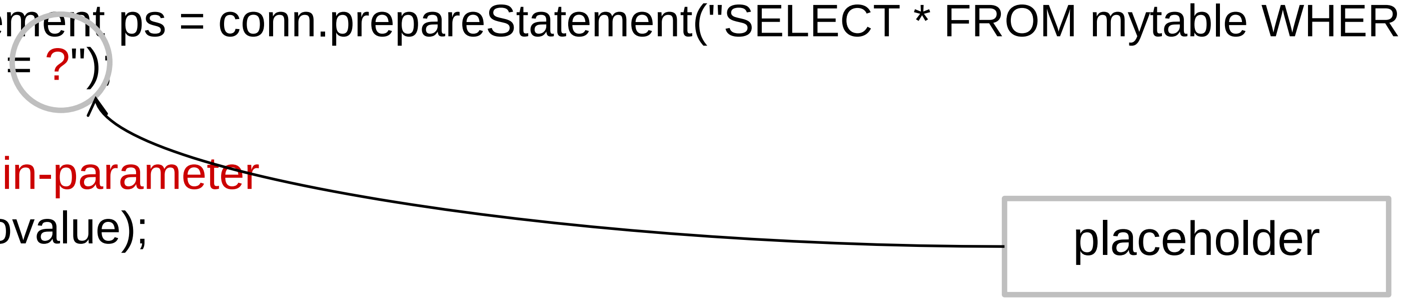
```
// Importing JDBC
import java.sql.*

//load the driver for PostgreSQL
Class.forName("org.postgresql.Driver");

//connect to the db
Connection conn =
            DriverManager.getConnection(url, user, passwd);

//disconnect
conn.close();
```

Processing a Simple Query in JDBC

```
// Create a Statement
Statement st = conn.createStatement();

//Execute Statement and obtain ResultSet
ResultSet rs = st.executeQuery("SELECT * FROM mytable WHERE
    columnfoo = 500");

// Process the Results
while (rs.next()) {
    System.out.println(rs.getString(1));
}

// Close ResultSet and Statement
rs.close(); st.close();
```

Same, but using PreparedStatement

```
int foovalue = 500;

// Prepare Statement
PreparedStatement ps = conn.prepareStatement("SELECT * FROM mytable WHERE
    columnfoo = ?");

// Set value of in-parameter
ps.setInt(1, foovalue);

// Execute Statement and obtain ResultSet
ResultSet rs = ps.executeQuery();

// Process the Results
while (rs.next()) {System.out.println(rs.getString(1));}

// Close ResultSet and PreparedStatement
rs.close();ps.close();
```



placeholder

Advantages of PreparedStatements

- Performance:

  The overhead of compiling and optimizing the statement is incurred only once, although the statement is executed multiple times

- Security:

  Resilient against SQL injection (see next)

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 result set

Result Set

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

- Concurrency mode of ResultSet (read-only/updatable cursor):
  - CONCUR_READ_ONLY
  - CONCUR_UPDATABLE
- Type of ResultSet (cursor operations allowed):
  - TYPE_FORWARD_ONLY
  - TYPE_SCROLL_INSENSITIVE
  - TYPE_SCROLL_SENSITIVE

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.

Transactions in JDBC

- Default for a connection is autocommit
  - each SQL statement is a transaction
- Group several statements into a Transaction:
  - Set autocommit to false: conn.setAutoCommit (false);
  - Several SQL statements: …UPDATE, UPDATE, INSERT, etc.
  - Commit statements: conn.commit();
  - Set autocommit back to true: conn.setAutoCommit(true);

PHP: PEAR DB

PHP

- A language to be used for actions within HTML
  - Indicated by <? PHP code ?>
- Basic programming elements:
  - Variables: must begin with $
  - Two kinds of Arrays: numeric and associative
- DB library exists within PEAR (PHP Extension and Application Repository)
  - include with include(DB.php)

Making a Connection

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

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

Function connect
in the DB library

$conn is a Connection
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

Ex. Query: “Find all the bars that sell a beer given by the variable $beer.”

```
$beer = ’Bud’;
$result = $conn->query(”SELECT bar FROM Sells WHERE beer = $beer ;”);
```

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

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