Using SQL in an Application

Outline

- Embedded SQL
- Dynamic SQL
- JDBC

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
- **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 cont'd

- SQL constructs in an application take two forms:
  - Standard SQL statements (***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
 int year;
    char director [11];
    char  SQLSTATE [6];
EXEC SQL END DECLARE ;
        ……
EXEC  SQL  SELECT  M.year
     INTO  :year
     FROM  Movies M
     WHERE  M.director = :director;
```

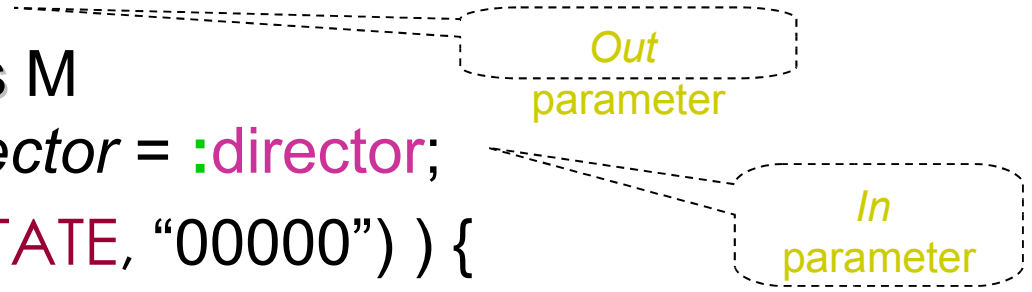
Variables shared by host and SQL

":" 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 M.year
    INTO :year
    FROM Movies M
    WHERE M.director = :director;
if ( !strcmp (SQLSTATE, "00000") ) {
     printf ( "statement failed" )
};
```



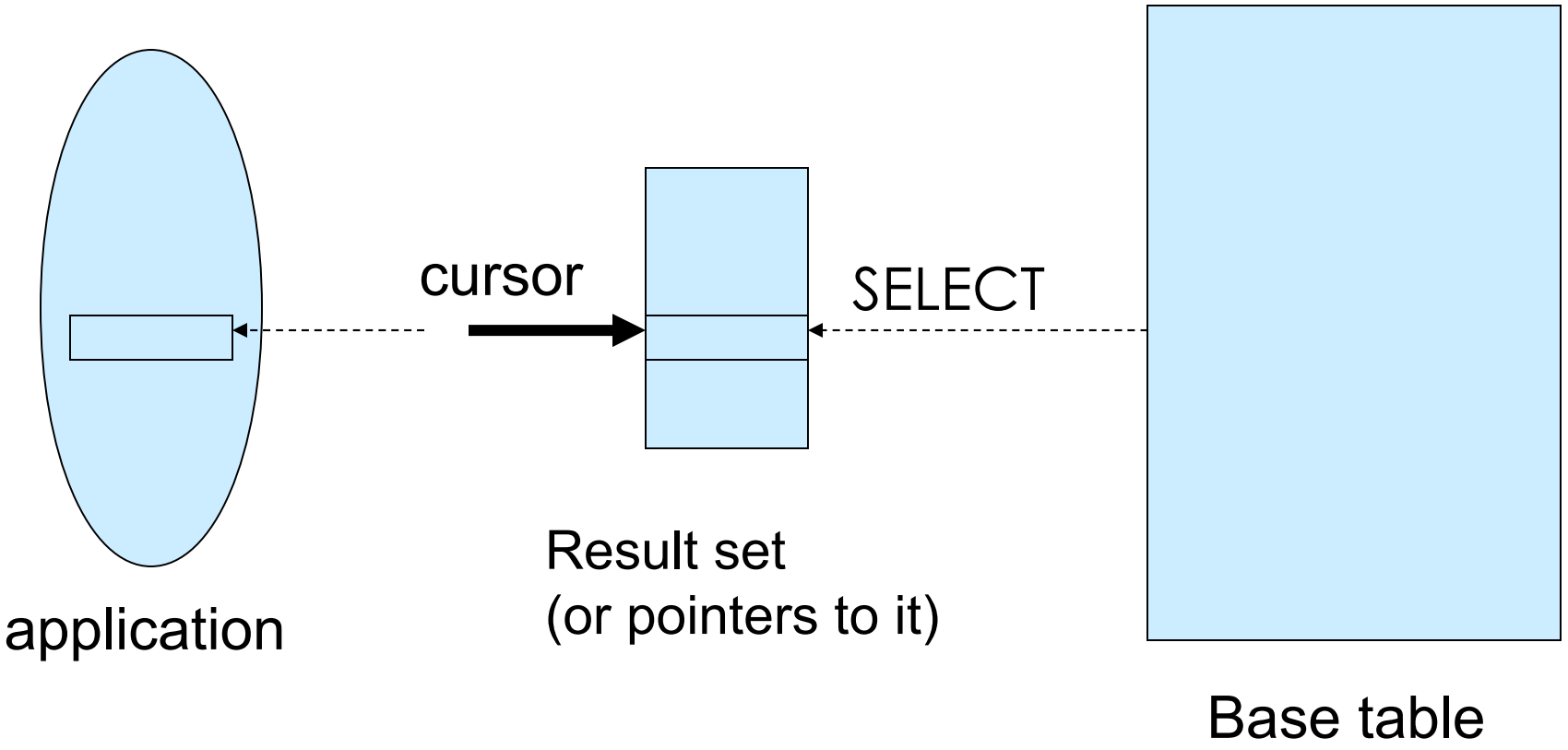
Buffer Mismatch Problem

- **Problem**: SQL deals with tables (of arbitrary size); host language program deals with fixed size buffers
  - How does the application 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 cont'd



Cursors cont'd

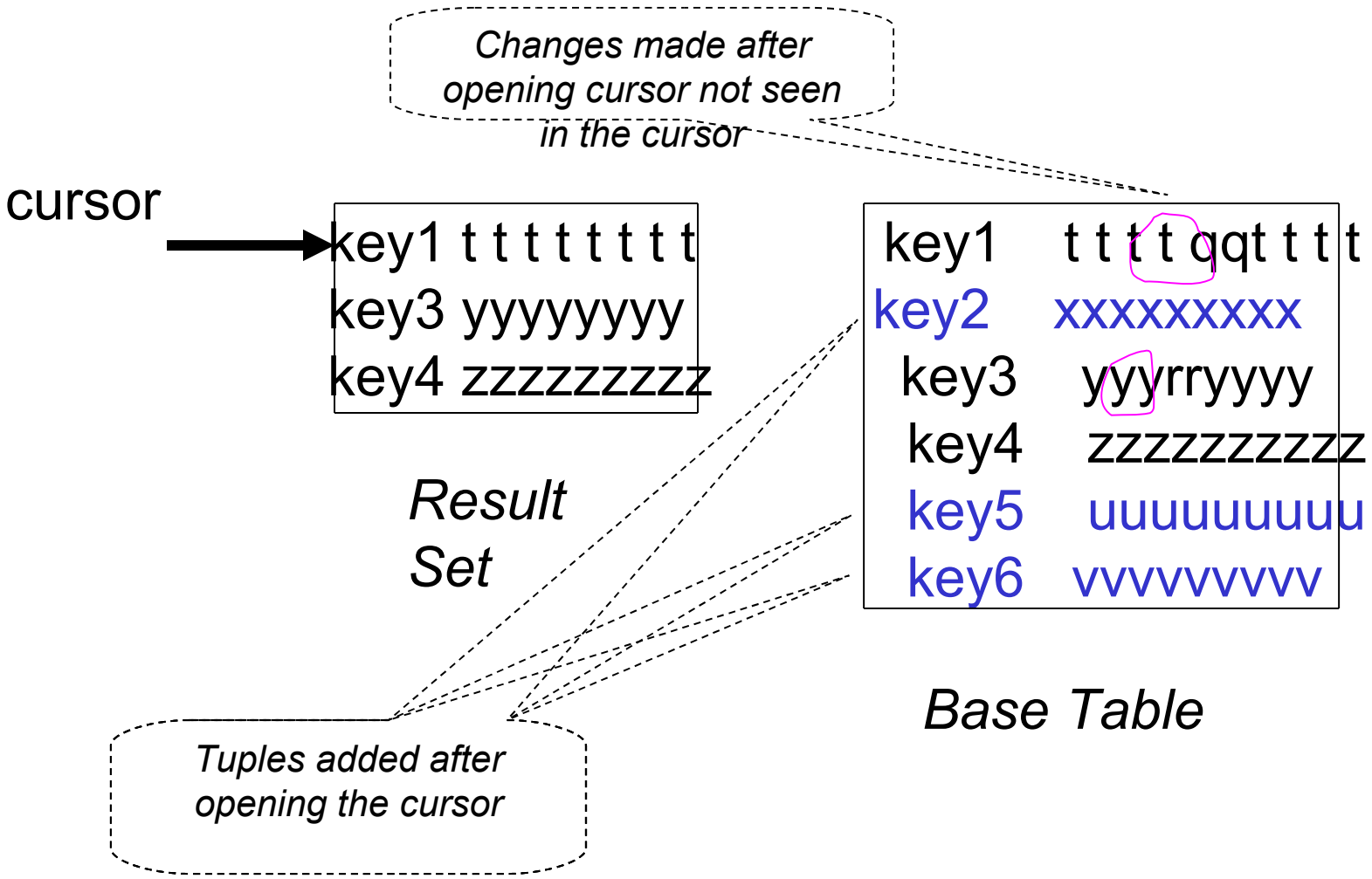
```
EXEC SQL DECLARE  GetTitle CURSOR FOR
    SELECT  M.mID, M.title          --cursor is not a schema element
     FROM Movies M
     WHERE  M.director = :director  AND M.year < 1980;
    ……
EXEC SQL OPEN GetTitle;
if ( !strcmp ( SQLSTATE, "00000")) {... No error... };
    ……
EXEC SQL FETCH GetTitle INTO :mID, :title;
while  ( SQLSTATE = "00000")  {
    … process the returned row...
    EXEC SQL FETCH  GetTitle  INTO :mID, :title;
}
if ( !strcmp ( SQLSTATE, "02000")) {…No tuples found... };
    ……
EXEC SQL CLOSE  GetTitle;
```

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

Insensitive 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*

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 GetTitle INTO :mID, :title;
```

Also: FIRST,
LAST,
ABSOLUTE n,
RELATIVE n

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 cont'd

- 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 cont'd

```
strcpy (tmp, "SELECT  M.year FROM Movies M  \
                WHERE M.director = ?" ) ;
EXEC SQL PREPARE st FROM :tmp;
EXEC SQL EXECUTE st INTO :year USING :director;
```

placeholder

- st is an SQL variable; names the SQL statement
- tmp, year, director are host language variables (note colon notation)
- director is an *in* parameter; supplies value for placeholder (?)
- year is an *out* parameter; receives value from M.*year*
- **PREPARE** names SQL statement st and sends it to DBMS for preparation
- **EXECUTE** causes the statement named st to be executed

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

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

- Different RDBMS systems have surprisingly little in common other than their use of SQL; each has its own unique API.
- JDBC (Java Database Connectivity) provides a *standard, generic* SQL database access interface.
- The JDBC API defines classes to represent major DB functionality, such as database connections, SQL statements, result sets, and database metadata.
- JDBC allows a Java program to issue SQL statements and process the results.

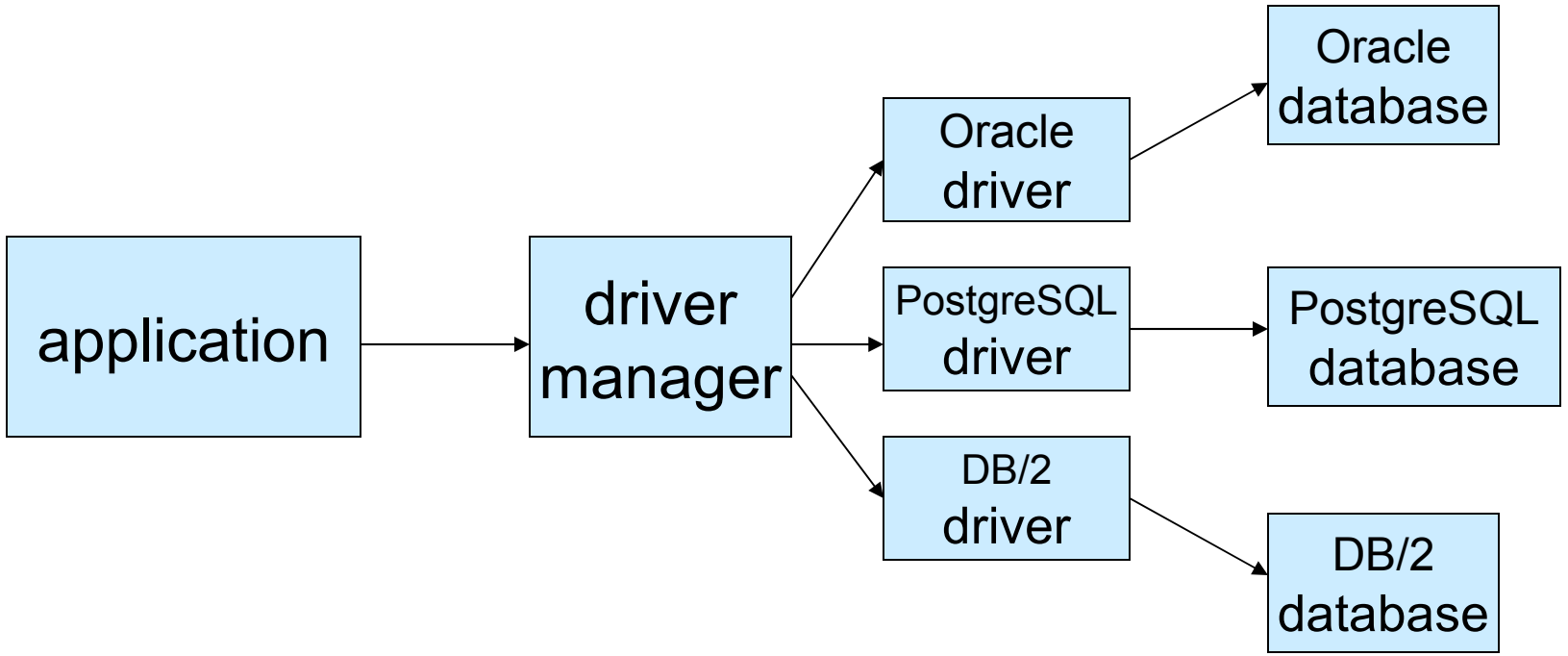
JDBC Goals

- DB independence: provide Java programmers with a uniform, simple interface to a wide range of relational databases. Can replace underlying database with minimal code impact.
- Platform independence.
- Provide a common base on which higher level tools and interfaces can be built.
- Note JDBC does not attempt to standardize SQL syntax across vendor DB products, which often implement their own proprietary SQL extensions.

JDBC API

- 4 main interfaces:
  - **java.sql.DriverManager** – handles loading of drivers and provides support for creating new database connections
  - **java.sql.Connection** –represents a connection to a particular database
  - **java.sql.Statement** – acts as a container for executing an SQL statement on a given connection. Passes SQL strings to the DB for execution and result set return
  - **java.sql.ResultSet** – controls access to the row results of a given statement

JDBC Run-Time Architecture



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*

Executing a Query cont'd

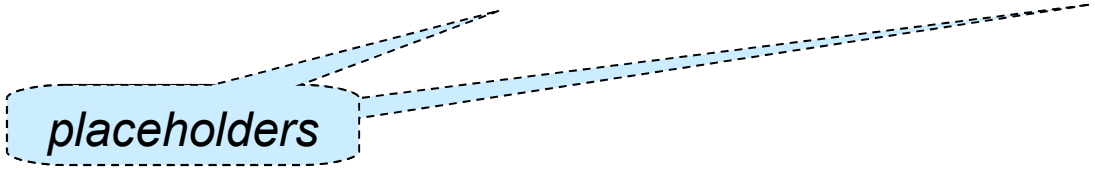
```
String query = "SELECT  M.title  FROM  Movies M" +
               "WHERE  M.director = 'Polanski' " +
               "AND  M.year < 1980";
```

```
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*

Preparing and Executing a Query

String query = "SELECT M.*title* FROM Movies M" +
"WHERE M.director = ? AND M.*year* < ?";



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  director, year;
……
ps.setString(1, director);      // set value of first in parameter
ps.setString(2, year);          // 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.getString ("title");            // fetch output int-value
      …process output value…
}
```

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
  - 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